Learning to play Pac-Man

Seminar Artificial Intellgience for Games
Prof. Dr. Ullrich Köthe

by
Christian Buschmann
Immatriculation number 3492195
M.Sc. Applied Computer Science

July 2019

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Artificial Intelligence has existed in games for a very long time. Games provide both an excellent testing ground for new technologies in the field of artificial intelligence as well as having a very intuitive connection to humans. Games provide a way for humans to visualize, understand, and design artificial intelligences based on knowledge of the game and an easy - and often fun - way to then test these new technologies. This leads to a much lowered entry barrier into the field, due to which artificial intelligence was able to be designed and tested very early, in the case of for example chess even on paper before computers capable of running these artificial intelligences existed. Over the course of this report I will cover two different papers describing their approaches of playing the popular arcade game "Pac-Man" as well as their findings. The first papercovered is titled "Learning to Play Pac-Man: An Evolutionary, Rule-based Approach" by Gallagher and Ryan [2] and covers a very simplified version of Pac-Man on which an agent based on intuitive and human-readable parameters is trained. The second paper, titled "Learning to play using low-complexity rule-based policies: Illustrations through Ms. Pac-Man" by Szita and Lõrincz [4] attempts to learn the game of Ms. Pac-Man, a variation of Pac-Man, by learning action rules based on observations of the entire surrounding area, greatly increasing the agent's ability to perceive the current game state.

Chapter 2

Pac-Man

In this section, the game of Pac-Man will be introduced, after which the agent trained by Gallagher and Ryan [2] will be explained as well as evaluated.

2.1 The Game

Pac-Man is an arcade game in which the player control a continuously moving character called “Pac-Man” through a maze by changing the direction the character moves. Two sides of the maze contain a “warp exit”, a path that will let the player wrap around the screen to the other side. This maze is filled with a set number of dots and power pills - larger dots. In the standard game, fruits also appear from time to time randomly, which grant a large sum of points when eaten. The goal of the game is to collect all the dots without dying to any of the four ghosts present who attempt to chase the player as he’s collecting the dots. Eating a power pill turns the ghosts blue and makes them run away for a short duration while also grantint the player the ability to “eat” them, resetting them to a jail in the center of the map, which they can leave again after a certain delay. These ghosts move with mostly predetermined behaviours by switching between three modes - Chase, Patrol, and Retreat - of which only the Retreat mode is randomized.

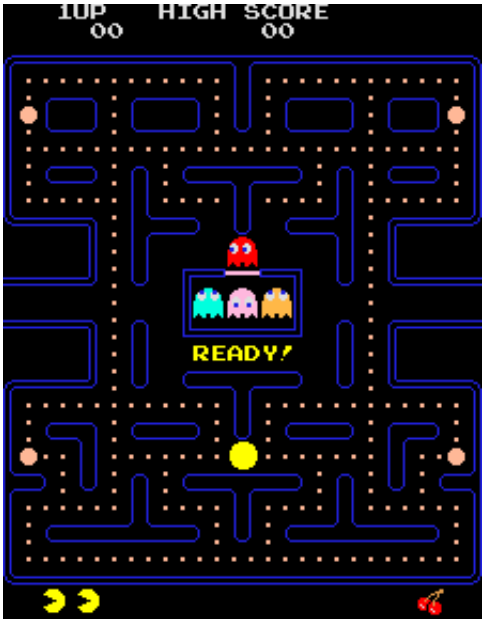
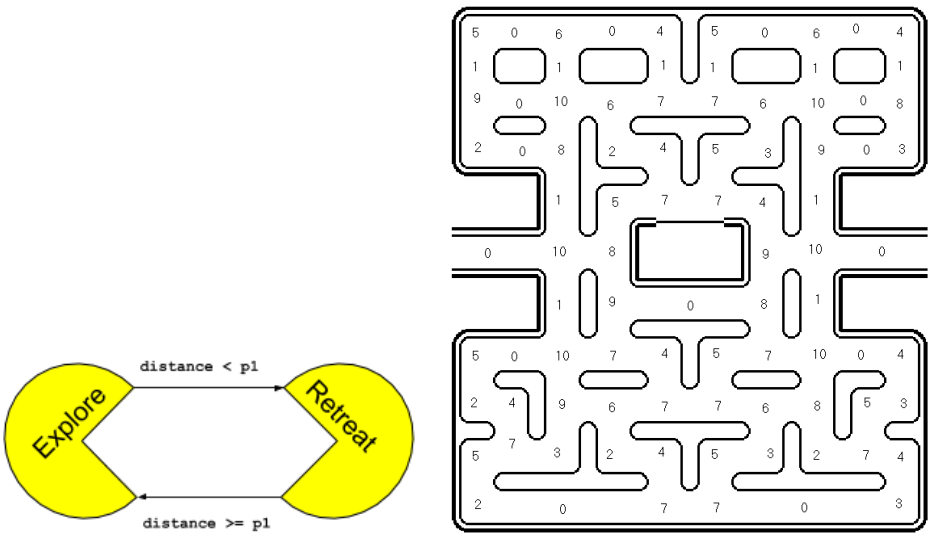


Figure 2.1: A typical game state of Pac-Man. [5]

2.2 Game Playing Agent

In order to play this game, Gallagher and Ryan first define a model of an agent that has the capability of playing the game upon which the actual algorithms can be ran. For this, they first simplified the game to only one ghost being preset as well as there not being any power pills. While this obviously significantly cuts down on usefulness of the agent, they felt that this would allow a good environment to test the viability of such an agent before attempting to extend it to the full game. They modeled the agent as a simple state machine consisting of "Explore" and "Retreat" states, the transition of which is defined by distance to the ghost. Additionally, they modeled all possible moves as their eleven distinct turn types.

By setting the agent up as such, its behaviour can now be modeled by a parameter vector of 85 parameters. The first parameter defined the threshold for the state transitions. The next seventeen parameters defined the likelihoods for each direction at each distinct turn type in the "Explore" state, the remaining 76 parameters define the behaviour in the "Retreat" state. In each turn, the ghost's position is mapped



(a) (b)

Figure 2.2: The state machine and distinct types of turns. [2]

to eight different cases for the eight different quadrants around the player the ghost can be in - for example “back”, “back-left”, “back-right”, and so on. The ghost positions are only included in the retreat state, as in the explore state the ghost is assumed to be far enough away to not have to worry about its position. Furthermore, only in the case of an intersection are all eight directions handled differently, otherwise only the four general directions are used in order to keep the parameter vector from exploding in size. This parameter vector can now be used to define the full behaviour of the agent and can therefore be easily trained.

2.3 Evolutionary Approach

Due to the agent’s behaviour now being entirely defined by this parameter vector, each parameter of which defines a probability of movement (with the exception of the distance to the ghost), its movement is very stochastic. To account for this, training is run 10 times per instance. Gallagher and Ryan do this by applying a

genetic algorithm to this vector, as all that is needed to perform this algorithm is a well-defined fitness function. They define this function as follows:

$$f = \sum_{level} \frac{score_{level}}{maxscore_{level}} + \min\left\{\frac{time_{level}}{maxtime_{level}}, 1\right\}$$

Since the maximum score for a level is defined by the number of dots available to collect, this part of the function can at most reach a value of 1 for collecting all points. In order to contain the amount of score available by sheer dodging of the ghost, the minimum of the second term makes sure that if the agent survives longer than a manually defined "maximum time" for the level, it can still only gain a value of 1. The maximum fitness for a level is therefore 2.

As baseline comparison, three parameter vectors are hard-coded: P_{h1}, which sets every parameter to the same probability, P_{h2}, which makes it slightly less likely to turn around and prevents the agent from moving towards the ghost during the retreat phase, and P_{h3}, which makes it even more unlikely to turn around than P_{h2}. Out of these, only P_{h3} actually ever clears a level fully and can evade the ghost for extended periods of time, while P_{h2} at least gets close to clearing it, although not often. Both P_{h1} and P_{h2} display heavy oscillations in their movement patterns due to them being able to turn around at each turn somewhat likely.

Vector	Mean	Std. Dev	Min.	Max
P_{h1}	0.215	0.080	0.104	0.511
P_{h2}	0.613	0.316	0.187	1.699
P_{h2}	1.372	0.522	0.276	2.062

Table 2.1: Performance of the hand-coded parameter sets [2]

In these tests, limitations of the setup of the agent already become visible. For instance, the agent has no knowledge of the position and count of dots remaining in the maze and can therefore only "accidentally" collect them. Additionally, the estimation of the ghost's position is relatively rough, the agent can become stuck

in corners due to the movement leading out of the corner putting the agent closer to the ghost, even if this movement would be safe. Even still, evolutionary algorithms were used to try and improve performance.
For this, Population-Based Incremental Learning (PBIL [1]) is used. Each generation plays 250 games via a population count of 25, each parameter set of which would run 10 games, as previously mentioned. The End results proved to be slightly better than P_{h1} andP_{h2}, yet still slightly worse than P_{h3}, while converging to similar values.

2.4 Conclusions

In these tests, the limitations of a simple rule-based agent become clear very quickly. Firstly, an issue of parameter bloat exists. For example for straight lines, probabilities for each direction are not needed, as the probability for turning around can be calculated as $1-$ the probability to move forwards. Additionally, even more parameters would be needed to cover every possible case of even this simple agent. The next issue arising is the fact that the agent is, as previously mentioned, lacking "intelligence", due to it having a very simplified view of the board and next to no knowledge of its actual contents. This leads Gallagher and Ryan to believe that extending on such a simple rule set is impractical, it therefore being only useful as a benchmark, not as an actual playing agent.

Chapter 3

Ms. Pac-Man

In this section, I will cover Szita and Lõrincz's paper on playing Ms. Pac-Man, a variation of Pac-Man, using rule-based policies.

3.1 The Game

Ms. Pac-Man is a variation of the regular game of Pac-Man in which two additional warp exists are added, fruits are more random, and ghosts don't follow strict behaviour patterns via an added randomness factor, in addition to a differing base behaviour than in the standard game. This leads to the agent having to actually "learn" how to play more, as it can't just exploit movement patterns of the different ghost types.

3.2 Game Playing Agent

Szita and Lõrincz define their game playing agent by a set of rules it should follow, which include a tie-breaking mechanism in case multiple rules tell the agent to behave differently. These rules should be human-readable to make it easy to include domain knowledge. They therefore define a concept of an Action Module, which should contain conditions based on game observations as well as actions to take. These Action Modules determine the behaviour of the agent. The requirements needed to define this approach are therefore a list of possible actions, possible con-

ditions, a rule on how to create a rule from conditions and actions, as well as a rule on how to combine these rules into policies.

3.3 Action Modules

Each Action Module is ranked by a priority and can be either switched on or off. The Agent can use any subset of these modules at once, while the highest ranked module determines the direction of movement. Equally ranked directions are decided via a tie-breaker by letting the next highest ranked direction decide. If no tie-breaker is possible, a direction is chosen randomly. These decisions are made each full grid cell in order to force the agent to "commit" to its movement instead of turning around or changing direction in the middle of it.

3.3.1 Actions

The actions used in these modules are easy to manually implement, meaning domain knowledge can be induced easily. For example, an action called "ToDot" can be defined to simply tell the agent to move towards the nearest dot. These actions are not exclusive, as there can for example be dots in multiple directions, while only some of these directions may fulfill another action. In order to determine a final direction, these modules are assigned a priority, which has to be learned during training.

Name	Description
ToDot	Go towards the nearest dot.
ToPowerDot	Go towards the nearest power dot.
FromPowerDot	Go in direction opposite to the nearest power dot.
ToEdGhost	Go towards the nearset edible ghost.
ToLowerGhostDensity	Go in the direction where the cumulative ghost density decreases fastest.

Table 3.1: Some example actions [4]

3.3.2 Conditions

The next part necessary to build the action module is a condition. Here, conditions are made up of one or more observations, all of which are manually defined. While these could obviously be improved, they provide a good baseline to build rules out of. These observations can be for example distances to objects, ghosts, pills, or others. They default to the maximum value if unknown. The nature of these means that the agent can easily calculate these in every step in order to maintain performance and accuracy.

Name	Description
Constant	Constant 1 value.
NearestDot	Distance of nearest dot.
NearestGhost	Distance of nearest Ghost
GhostDensity	Cumulative ghost density.

Table 3.2: Some example observations [4]

These observations are then used to build conditions by joining them with logic operators. Each condition can contain multiple observations. An example condition could be $(NearestDot < 5) and (NearestGhost > 8) and (FromGhost+)$, which would evaluate to true if the nearest dot is under 5 tiles away from the agent, the nearest ghost is over 8 tiles away, and the agent is currently avoiding a ghost. Joining these conditions with an action then builds a finished rule.

3.4 Policies

The action modules defined in the previous section can then be combined into a finalized policy determining the entire behaviour of the agent.
In these policies, a rule stays switched on until it is explicitly switched off or replaced by a higher priority rule. In the example hand-coded policy 3.1, the first

```
[1] if NearestGhost<4 then FromGhost+
[1] if NearestGhost>7 and JunctionSafety>4 then FromGhost-
[2] if NearestEdGhost>99 then ToEdGhost-
[2] if NearestEdGhost<99 then ToEdGhost+
[3] if Constant>0 then KeepDirection+
[3] if FromPowerDot- then ToPowerDot+
[3] if GhostDensity<1.5 and NearestPowerDot<5 then FromPowerDot+
[3] if NearestPowerDot>10 then FromPowerDot-
```

Figure 3.1: An example hand-coded policy. [4]

two rules simply state that if the ghost is too close, the agent should flee until it reaches a safe distance, using these rules as highest priority. The next two rules dictate that if there is an edible ghost, chase it (to gain points by eating it). The next rule determines that the ghost should never turn back if all previous directions are of the same value, to avoid oscillations, while the remaining rules determine that a power dot should only be eaten when enough ghosts are around to eat them all at once in order to gain the increasing points for each subsequent ghost eaten within the same power dot timespan.

3.5 Policy Learning

In order to automatically learn such a policy, both a set of rules as well as a set of rule slots (or policy slots) are required. In the paper, each rule slot i is assigned a fixed priority and has a probability p_i to contain a rule. If a slot is picked to contain a rule, it has a probability of $q_{i,j}$ to contain any given rule j. These probabilities are then learned by a learning algorithm.

The required ruleset can be defined in two manners: It can either be manually predefined, or it can be automatically generated. This is dojne by randomly picking two conditions and then selecting their threshold value uniformly from a set for each condition module, based on the histogram of seen values in prior games. After this, an action is randomly picked and given a 50 percent chance to either turn

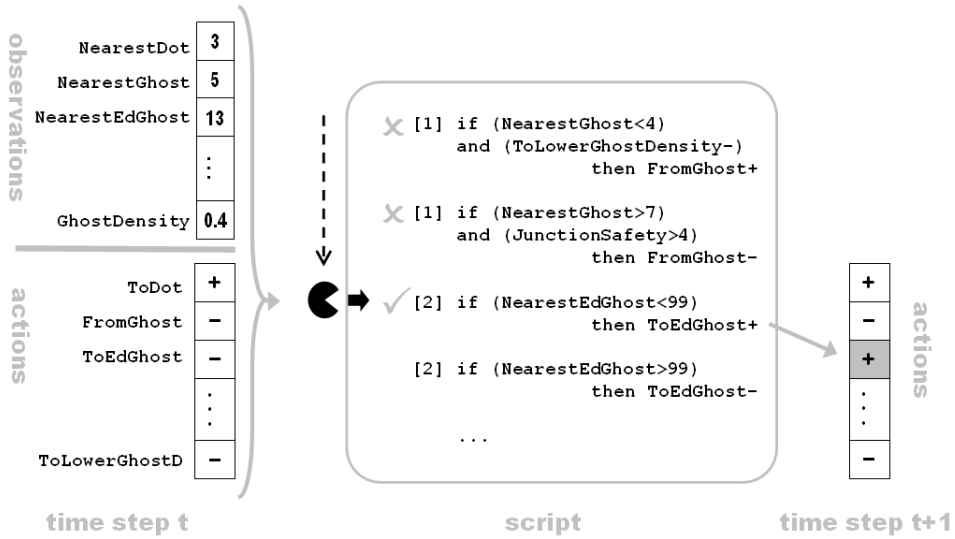


Figure 3.2: The overall setup of the agent. [4]

on or off in this rule.

3.6 Experiments

To create these random rulesets, 100 rules were generated. Additionally, one hand-coded ruleset of 42 rules was defined. Policies generated from the random ruleset were given a maximum of 100 ruleslots with evenly distributed priorities, policies generated from the hand-coded ruleset were given a maximum of 30 rule slots. For a baseline comparison, a policy of 10 random rules was used in addition to one fully hand-coded policy.

Method	Avg. Score	(25%/75% percentiles)
CE-randomRB	6382	(6147/6451)
CE-fixedRB	**8186**	(6682/9369)
SG-randomRB	4135	(3356/5233)
SG-fixedRB	5449	(4843/6090)
CE-randomRB-1Action	5417	(5319/5914)
CE-fixedRB-1Action	5631	(5705/5982)
SG-randomRB-1Action	2267	(1770/2694)
SG-fixedRB-1Action	4415	(3835/5364)
Random policy	676	(140/940)
Hand-coded policy	7547	(6190/9045)
Human play	8064	(5700/10665)

Table 3.3: Results of the experiments [4]. CE denotes Cross-Entropy Method [3], SD denotes Stochastic Gradient. 1Action are the versions of the algorithm that were limited to only activating one action module at a time.

3.7 Conclusions

While overall, the random ruleset policies behaved similarly to the fixed ruleset, yet the random ruleset contains a lot of superfluous rules, as well as superfluous conditions (such as always-true conditions) due to them being picked randomly. As you can see, the ability to perform more than one action at once is essential, as the agent could otherwise for example not walk away from a ghost and also try to collect dots at the same time, severely limiting its capabilities. Another observation that can be made is that, at least in this setup of experiments, Cross-Entropy Method performed a lot better than Stochastic Gradient, although this could be fixed with a more thorough search over the entire parameter space. One thing of note, however, is that no agent ever evolved the tactic of "luring ghosts in" before eating a power pill, a tactic which is very common in human players. Further improvements could be made by adding time-related conditions such as time until a ghost reaches a player, as this could let the agent decide whether it can "cut ahead"

```
[1] if NearestGhost<3 then FromGhost+
[1] if MaxJunctionSafety>3 then FromGhost-
[2] if NearestEdGhost>99 then ToPowerDot+
[2] if NearestEdGhost<99 then ToEdGhost+
[2] if GhostDensity<1.5 and NearestPowerDot<5 then FromPowerDot+
[3] if Constant>0 then ToCenterofDots+
```

(a)

```
[1] if MaxJunctionSafety>2.5 and ToLowerGhostDensity- then FromGhost-
[1] if NearestGhost<6 and MaxJunctionSafety<1 then FromGhost+
[1] if NearestGhost>6 and FromGhostCenter- then ToEdGhost+
[2] if ToEdGhost- and CenterOfDots>20 then ToEdGhost+
[2] if ToEdGhost- and NearestEdGhost<99 then ToEdGhost+
[2] if NearestDot>1 and GhostCenterDist>0 then KeepDirection+
[3] if ToGhostFreeArea- and ToDot- then ToPowerDot+
```

(b)

Figure 3.3: The best policy learned via fixed ruleset (a) and random ruleset (b) [4]

of a ghost in a turn or whether it can reach a ghost to eat it before the power pill runs out, and other similar tactics.

Chapter 4

Conclusion

Overall it can be said that, while rule-based approaches are easy for a human to read, they lack in functionality as observations have to be defined manually in order for rules to be able to be built out of them. Automatic generation of features by for example a neural network looking at the entire playing field would be a lot stronger, while only having the downside of not being easily humanly readable. Not relying on domain knowledge being induced and gathered by these agents could free these agents up to more powerful methods that could more confidently beat human play.

Bibliography

[1] Shumeet Baluja. Population-based incremental learning: A method for integrating genetic search based function optimization and competitive learning. Technical Report CMU-CS-94-163, Carnegie Mellon University, Pittsburgh, PA, January 1994.

[2] Marcus Gallagher and Anthony J Ryan. Learning to play pac-man: an evolutionary, rule-based approach. *The 2003 Congress on Evolutionary Computation, 2003. CEC '03.*, 4:2462–2469 Vol.4, 2003.

[3] Reuven Y. Rubinstein. The cross-entropy method for combinatorial and continuous optimization. *Methodology And Computing In Applied Probability*, 1:127–190, 1999.

[4] István Szita and András Lõrincz. Learning to play using low-complexity rule-based policies: Illustrations through ms. pac-man. *Journal of Artificial Intelligence Research*, 30:659–684, 2007.

[5] Wikipedia, the free encyclopedia. Pac-man, 2019. [Online; accessed July 10, 2019].