Journal of Artificial Intelligence Research 30 (2007) 659-684 Submitted 06/07; published 12/07

Learning to Play Using Low-Complexity Rule-Based Policies: Illustrations through Ms. Pac-Man

István Szita SZITYU@EOTVOS.ELTE.HU
András Lőrincz ANDRAS.LORINCZ@ELTE.HU
Dept. of Information Systems
Eötvös University, Hungary, H-1117

Abstract

In this article we propose a method that can deal with certain combinatorial reinforcement learning tasks. We demonstrate the approach in the popular Ms. Pac-Man game. We define a set of high-level observation and action modules, from which rule-based policies are constructed automatically. In these policies, actions are temporally extended, and may work concurrently. The policy of the agent is encoded by a compact decision list. The components of the list are selected from a large pool of rules, which can be either hand-crafted or generated automatically. A suitable selection of rules is learnt by the cross-entropy method, a recent global optimization algorithm that fits our framework smoothly. Cross-entropy-optimized policies perform better than our hand-crafted policy, and reach the score of average human players. We argue that learning is successful mainly because (i) policies may apply concurrent actions and thus the policy space is sufficiently rich, (ii) the search is biased towards low-complexity policies and therefore, solutions with a compact description can be found quickly if they exist.

1. Introduction

During the last two decades, reinforcement learning (RL) has reached a mature state, and has been laid on solid foundations. We have a large variety of algorithms, including value-function-based, direct policy search and hybrid methods. For reviews on these subjects, see, e.g., the books of Bertsekas and Tsitsiklis (1996) and Sutton and Barto (1998). The basic properties of many such algorithms are relatively well understood, e.g. conditions for convergence, complexity, the effect of various parameters, although it is needless to say that there are still lots of important open questions. There are also plenty of test problems (like various maze-navigation tasks, "pole-balancing", "car on the hill" etc.) on which the capabilities of RL algorithms have been demonstrated, and the number of large-scale RL applications is also growing steadily. However, current RL algorithms are far from being out-of-the-box methods, so there is still need for more demonstrations showing that RL can be efficient in complex tasks.

We think that games (including the diverse set of classical board games, card games, modern computer games, etc.) are ideal test environments for reinforcement learning. Games are intended to be interesting and challenging for human intelligence and therefore, they are ideal means to explore what artificial intelligence is still missing. Furthermore, most games fit well into the RL paradigm: they are goal-oriented sequential decision problems, where each decision can have long-term effects. In many cases, hidden information, random events, unknown environment, known or unknown players account for (part of) the difficulty of playing the game. Such circumstances are in the focus of reinforcement learning. Games are also attractive for testing new methods: the decision space is huge in most cases, so finding a good strategy is a challenging task.

There is another great advantage of using games as test problems: the rules of the games are fixed, so the danger of 'tailoring the task to the algorithm' – i.e., to tweak the rules and/or the environment so that they meet the capabilities of the proposed RL algorithm – is reduced, compared, e.g., to various maze navigation tasks.

RL has been tried in many classical games, including checkers (Samuel, 1959), backgammon (Tesauro, 1994), and chess (Baxter, Tridgell, & Weaver, 2001). On the other hand, modern computer games got into the spotlight only recently, and there are not very many successful attempts to learn them with AI tools. Notable exceptions are, for example, role-playing game *Baldur's Gate* (Spronck, Sprinkhuizen-Kuyper, & Postma, 2003), real-time strategy game *Wargus* (Ponsen & Spronck, 2004), and possibly, *Tetris* (Szita & Lőrincz, 2006). These games pose new challenges to RL, for example, many observations have to be considered in parallel, and both the observation space and the action space can be huge.

In this spirit, we decided to investigate the arcade game Ms. Pac-Man. The game is interesting on its own as it is largely unsolved, but also imposes several important questions in RL, which we will overview in Section 8. We will provide hand-coded high-level actions and observations, and the task of RL is to learn how to combine them into a good policy. We will apply rule-based policies, because they are easy to interpret and enable one to include human domain-knowledge easily. For learning, we will apply the cross-entropy method, a recently developed general optimization algorithm. We will show that the hybrid approach is more successful than either *tabula rasa* learning or a hand-coded strategy alone.

In the next section we introduce the Ms. Pac-Man game briefly and discuss how it can be formalized as a reinforcement learning task. In sections 3 and 4, we shall shortly describe the cross-entropy optimization method and rule-based policies, respectively. In section 5, details of the learning experiments are provided, and in section 6 we present our results. Section 7 provides a review of related literature, and finally, in section 8 we summarize and discuss our approach with an emphasis on the implications for other RL problems.

2. Pac-Man and Reinforcement Learning

The video-game Pac-Man was first released in 1979, and reached immense success. It is considered to be one of the most popular video games to date (Wikipedia, 2006).

The player maneuvers Pac-Man in a maze (see Fig. 1), while Pac-Man *eats* the dots in the maze. In this particular maze there are 174 dots,[1] each one is worth 10 points. A level is finished when all the dots are eaten. To make things more difficult, there are also four *ghosts* in the maze who try to catch Pac-Man, and if they succeed, Pac-Man loses a life. Initially, he has three lives, and gets an extra life after reaching 10,000 points.

There are four power-up items in the corners of the maze, called *power dots* (worth 40 points). After Pac-Man eats a power dot, the ghosts turn blue for a short period (15 seconds), they slow down and try to escape from Pac-Man. During this time, Pac-Man is

1. The maze of the original Pac-Man game is slightly different. This description applies to the open-source Pac-Man implementation of Courtillat (2001). The two versions are about equivalent in terms of complexity and entertainment value.

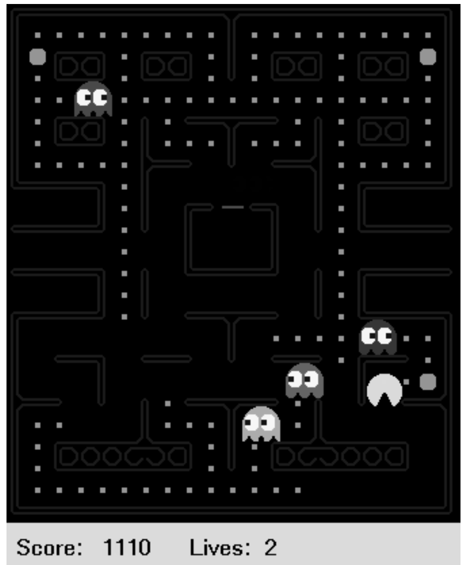

Figure 1: A snapshot of the Pac-Man game

able to eat them, which is worth 200, 400, 800 and 1600 points, consecutively. The point values are reset to 200 each time another power dot is eaten, so the player would want to eat all four ghosts per power dot. If a ghost is eaten, his remains hurry back to the center of the maze where the ghost is reborn. At certain intervals, a fruit appears near the center of the maze and remains there for a while. Eating this fruit is worth 100 points.

Our investigations are restricted to learning an optimal policy for the first level, so the maximum achievable score is $174 \cdot 10 + 4 \cdot 40 + 4 \cdot (200 + 400 + 800 + 1600) = 13900$ plus 100 points for each time a fruit is eaten.

In the original version of Pac-Man, ghosts move on a complex but deterministic route, so it is possible to learn a deterministic action sequence that does not require any observations. Many such *patterns* were found by enthusiastic players. In most of Pac-Man's sequels, most notably in *Ms. Pac-Man*, randomness was added to the movement of the ghosts. This way, there is no single optimal action sequence, observations are necessary for optimal decision making. In other respects, game play was mostly unchanged.

In our implementation, ghosts moved randomly in 20% of the time and straight towards Pac-Man in the remaining 80%, but ghosts may not turn back (following Koza, 1992, Chapter 12). To emphasize the presence of randomness, we shall refer to our implementation as a Ms. Pac-Man-clone.

2.1 Ms. Pac-Man as an RL Task

Ms. Pac-Man meets all the criteria of a reinforcement learning task. The agent has to make a sequence of decisions that depend on its observations. The environment is stochastic (because the paths of ghosts are unpredictable). There is also a well-defined reward function (the score for eating things), and actions influence the rewards to be collected in the future.

The full description of the state would include (1) whether the dots have been eaten (one bit for each dot and one for each power dot), (2) the position and direction of Ms. Pac-Man, (3) the position and direction of the four ghosts, (4) whether the ghosts are blue (one bit for each ghost), and if so, for how long they remain blue (in the range of 1 to 15 seconds) (5) whether the fruit is present, and the time left until it appears/disappears (6) the number of lives left. The size of the resulting state space is astronomical, so some kind of function approximation or feature-extraction is necessary for RL.

The action space is much smaller, as there are only four basic actions: go north/-south/east/west. However, a typical game consists of multiple hundreds of steps, so the number of possible combinations is still enormous. This indicates the need for temporally extended actions.

We have a moderate amount of domain knowledge on Ms. Pac-Man: for one, it is quite easy to define high-level observations and action modules that are potentially useful. On the other hand, constructing a well-performing policy seems much more difficult. Therefore, we provide mid-level domain knowledge to the algorithm: we use domain knowledge to preprocess the state information and to define action modules. On the other hand, it will be the role of the policy search reinforcement learning to combine the observations and modules into rule-based policies and find their proper combination.

3. The Cross-Entropy Method

Our goal is to optimize rule-based policies by performing policy search in the space of all legal rule-based policies. For this search we apply the *cross-entropy method* (CEM), a recently published global optimization algorithm (Rubinstein, 1999). It aims to find the (approximate) solution for global optimization tasks in the following form

$$\mathbf{x}^* := \arg\max_{\mathbf{x}} f(\mathbf{x}).$$

where f is a general objective function (e.g., we do not need to assume continuity or differentiability). Below we summarize the mechanism of this method briefly (see also section 7.2 for an overview of applications).

3.1 An Intuitive Description

While most optimization algorithms maintain a single candidate solution $\mathbf{x}(t)$ in each time step, CEM maintains a *distribution* over possible solutions. From this distribution, solution candidates are drawn at random. This is essentially random guessing, but with a nice trick it is turned into a highly effective optimization method.

3.1.1 The Power of Random Guessing

Random guessing is an overly simple 'optimization' method: we draw many samples from a fixed distribution g, then select the best sample as an estimation of the optimum. In the limit case of infinitely many samples, random guessing finds the global optimum. We have two notes here: (i) as it has been shown by Wolpert and Macready (1997), for the most general problems, uniform random guessing is not worse than any other method, (ii) nonetheless, for practical problems, uniform random guessing can be extremely inefficient.

Thus, random guessing is safe to start with, but as one proceeds with the collection of experience, it should be limited as much as possible.

The efficiency of random guessing depends greatly on the distribution g from which the samples are drawn. For example, if g is sharply peaked around $\mathbf{x}^*$, then very few samples may be sufficient to get a good estimate. The case is the opposite, if the distribution is sharply peaked around $\mathbf{x} \neq \mathbf{x}^*$: a tremendous number of examples may be needed to get a good estimate of the global optimum. Naturally, finding a good distribution is at least as hard as finding $\mathbf{x}^*$.

3.1.2 Improving the Efficiency of Random Guessing

After drawing moderately many samples from distribution g, we may not be able to give an acceptable approximation of $\mathbf{x}^*$, but we may still obtain a *better sampling distribution*. The basic idea of CEM is that it selects the best few samples, and modifies g so that it becomes more peaked around them. Consider an example, where $\mathbf{x}$ is a 0-1 vector and g is a Bernoulli distribution for each coordinate. Suppose that we have drawn 1000 samples and selected the 10 best. If we see that in the majority of the selected samples, the i^{th} coordinate is 1, then CEM shifts the Bernoulli distribution of the corresponding component towards 1. Afterwards, the next set of samples is drawn already from the modified distribution.

The idea seems plausible: if for the majority of the best-scoring samples the i^{th} coordinate was 1, and there is a structure in the fitness landscape, then we may hope that the i^{th} coordinate of $\mathbf{x}^*$ is also 1. In what follows, we describe the update rule of CEM in a more formal way and sketch its derivation.

3.2 Formal Description of the Cross-Entropy Method

We will pick g from a family of parameterized distributions, denoted by $\mathcal{G}$, and describe an algorithm that iteratively improves the parameters of this distribution g.

Let N be the number of samples to be drawn, and let the samples $\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)}$ be drawn independently from distribution g. For each $\gamma \in \mathbb{R}$, the set of high-valued samples,

$$\hat{L}_\gamma := \{\mathbf{x}^{(i)} \mid f(\mathbf{x}^{(i)}) \geq \gamma, 1 \leq i \leq N\},$$

provides an approximation to the level set

$$L_\gamma := \{\mathbf{x} \mid f(\mathbf{x}) \geq \gamma\}.$$

Let U_γ be the uniform distribution over the level set L_γ. For large values of γ, this distribution will be peaked around $\mathbf{x}^*$, so it would be suitable for random sampling. This raises two potential problems: (i) for large γ values $\hat{L}_\gamma$ will contain very few points (possibly none), making accurate approximation impossible, and (ii) the level set L_γ is usually not a member of the parameterized distribution family.

The first problem is easy to avoid by choosing lower values for γ. However, we have to make a compromise, because setting γ too low would inhibit large improvement steps. This compromise is achieved as follows: CEM chooses a ratio $\rho \in [0, 1]$ and adjusts $\hat{L}_\gamma$ to be the set of the best $\rho \cdot N$ samples. This corresponds to setting $\gamma := f(\mathbf{x}^{(\rho \cdot N)})$, provided that the samples are arranged in decreasing order of their values. The best $\rho \cdot N$ samples are called the *elite samples*. In practice, ρ is typically chosen from the range $[0.02, 0.1]$.

The other problem is solved by changing the goal of the approximation: CEM chooses the distribution g *from the distribution family* $\mathcal{G}$ that approximates best the empirical distribution over $\hat{L}_\gamma$. The best g is found by minimizing the distance of $\mathcal{G}$ and the uniform distribution over the elite samples. The measure of distance is the *cross-entropy distance* (often called Kullback-Leibler divergence). The cross-entropy distance of two distributions g and h is defined as

$$D_{CE}(g||h) = \int g(\mathbf{x}) \log \frac{g(\mathbf{x})}{h(\mathbf{x})} \, \mathrm{d}\mathbf{x}$$

The general form of the cross-entropy method is summarized in Table 1. It is known that under mild regularity conditions, the CE method converges with probability 1 (Margolin, 2004). Furthermore, for a sufficiently large population, the global optimum is found with high probability.

input: $\mathcal{G}$	% parameterized distrib. family
input: $g_0 \in \mathcal{G}$	% initial distribution
input: N	% population size
input: ρ	% selection ratio
input: T	% number of iterations
for t from 0 to $T-1$,	% CEM iteration main loop
for i from 1 to N,	
draw $\mathbf{x}^{(i)}$ from distribution g_t	% draw N samples
compute $f_i := f(\mathbf{x}^{(i)})$	% evaluate them
sort f_i-values in descending order	
$\gamma_{t+1} := f_{\rho \cdot N}$	% level set threshold
$E_{t+1} := \{\mathbf{x}^{(i)} \mid f(\mathbf{x}^{(i)}) \geq \gamma_{t+1}\}$	% get elite samples
$g_{t+1} := \arg\min_{g \in \mathcal{G}} D_{CE}(g\|\|\mathit{Uniform}(E_{t+1}))$	% get nearest distrib. from $\mathcal{G}$
end loop	

Table 1: Pseudo-code of the general cross-entropy method

3.3 The Cross-Entropy Method for Bernoulli Distribution

For many parameterized distribution families, the parameters of the minimum cross-entropy member can be computed easily from simple statistics of the elite samples. We provide the formulae for Bernoulli distributions, as these will be needed for the policy learning procedure detailed in the next section. Derivations as well as a list of other discrete and continuous distributions that have simple update rules can be found in the tutorial of de Boer, Kroese, Mannor, and Rubinstein (2004).

Let the domain of optimization be $D = \{0,1\}^m$, and each component be drawn from independent Bernoulli distributions, i.e., $\mathcal{G} = \mathit{Bernoulli}^m$. Each distribution $g \in \mathcal{G}$ is parameterized with an m-dimensional vector $\mathbf{p} = (p_1, \ldots, p_m)$. When using g for sampling, component j of the sample $\mathbf{x} \in D$ will be

$$x_j = \begin{cases} 1, & \text{with probability } p_j; \\ 0, & \text{with probability } 1 - p_j. \end{cases}$$

After drawing N samples $\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(N)}$ and fixing a threshold value γ, let E denote the set of elite samples, i.e.,

$$E := \{\mathbf{x}^{(i)} \mid f(\mathbf{x}^{(i)}) \geq \gamma\}$$

With this notation, the distribution g' with minimum CE-distance from the uniform distribution over the elite set has the following parameters:

$$\begin{aligned} \mathbf{p}' &:= (p'_1, \dots, p'_m), \quad \text{where} \\ p'_j &:= \frac{\sum_{\mathbf{x}^{(i)} \in E} \chi(x_j^{(i)} = 1)}{\sum_{\mathbf{x}^{(i)} \in E} 1} = \frac{\sum_{\mathbf{x}^{(i)} \in E} \chi(x_j^{(i)} = 1)}{\rho \cdot N} \end{aligned} \tag{1}$$

In other words, the parameters of g' are simply the component wise empirical probabilities of 1's in the elite set. For the derivation of this rule, see the tutorial of de Boer et al. (2004).

Changing the distribution parameters from $\mathbf{p}$ to $\mathbf{p}'$ can be too coarse, so in some cases, applying a step-size parameter α is preferable. The resulting algorithm is summarized in Table 2.

input: $\mathbf{p}_0 = (p_{0,1}, \dots, p_{0,m})$	% initial distribution parameters
input: N	% population size
input: ρ	% selection ratio
input: T	% number of iterations
for t from 0 to $T-1$,	% CEM iteration main loop
for i from 1 to N,	
draw $\mathbf{x}^{(i)}$ from $Bernoulli^m(\mathbf{p}_t)$	% draw N samples
compute $f_i := f(\mathbf{x}^{(i)})$	% evaluate them
sort f_i-values in descending order	
$\gamma_{t+1} := f_{\rho \cdot N}$	% level set threshold
$E_{t+1} := \{\mathbf{x}^{(i)} \mid f(x^{(i)}) \geq \gamma_{t+1}\}$	% get elite samples
$p'_j := \left(\sum_{\mathbf{x}^{(i)} \in E} \chi(x_j^{(i)} = 1)\right) / (\rho \cdot N)$	% get parameters of nearest distrib.
$p_{t+1,j} := \alpha \cdot p'_j + (1 - \alpha) \cdot p_{t,j}$	% update with step-size α
end loop	

Table 2: Pseudo-code of the cross-entropy method for Bernoulli distributions

We will also need to optimize functions over $D = \{1, 2, \dots, K\}^m$ with $K > 2$. In the simplest case, distributions over this domain can be parameterized by $m \cdot K$ parameters: $\mathbf{p} = (p_{1,1}, \dots, p_{1,K}; \dots; p_{m,1}, \dots, p_{m,K})$ with $0 \leq p_{j,k} \leq 1$ and $\sum_{k=1}^{K} p_{j,k} = 1$ for each j (this is a special case of the multinomial distribution).

The update rule of the parameters is essentially the same as Eq. 1 for the Bernoulli case:

$$p'_{j,k} := \frac{\sum_{\mathbf{x}^{(i)} \in E} \chi(x_j^{(i)} = k)}{\sum_{\mathbf{x}^{(i)} \in E} 1} = \frac{\sum_{\mathbf{x}^{(i)} \in E} \chi(x_j^{(i)} = k)}{\rho \cdot N}. \tag{2}$$

Note that constraint $\sum_{k=1}^{K} p'_{j,k} = 1$ is satisfied automatically for each j.

4. Rule-Based Policies

In a basic formulation, a rule is a sentence of the form "`if [Condition] holds, then do [Action]`." A rule-based policy is a set of rules with some mechanism for breaking ties, i.e., to decide which rule is executed, if there are multiple rules with satisfied conditions.

Rule-based policies are human-readable, it is easy to include domain knowledge, and they are able to represent complex behaviors. For these reasons, they are often used in many areas of artificial intelligence (see section 7.3 for a short overview of related literature).

In order to apply rule-based policies to Ms. Pac-Man, we need to specify four things: (1) what are the possible actions (2) what are the possible conditions and how are they constructed from observations, (3) how to make rules from conditions and actions, and (4) how to combine the rules into policies. The answers will be described in the following sections.

4.1 Action Modules

While defining the action modules for Ms. Pac-Man, we listed only modules that are easy to implement but are considered potentially useful (see Table 3). This way, we kept human work at a minimum, but still managed to formalize a part of our domain knowledge about the problem. As a consequence, this list of action modules is by no means optimal: some actions could be more effective with a more appropriate definition, others may be superfluous. For example, there are four different modules for ghost avoidance: `FromGhost` escapes from the nearest ghost, without considering the position of the other ghosts; `ToLowerGhostDensity` tries to take into account the influence of multiple ghosts; `FromGhostCenter` moves out from the geometrical center of ghosts, thus, it is able to avoid being surrounded and trapped, but, on the other hand, it can easily bump into a ghost while doing so; and finally, `ToGhostFreeArea` considers the whole board in search of a safe location, so that the agent can avoid being shepherded by the ghosts. All of these modules may have their own strengths and weaknesses, and possibly a combination of them is needed for success. There can also be actions, which are potentially useful, but were not listed here (for example, moving towards the fruit).

Note also that the modules are not exclusive. For example, while escaping from the ghosts, Ms. Pac-Man may prefer the route where more dots can be eaten, or she may want to head towards a power dot. Without the possibility of concurrent actions, the performance of the Ms. Pac-Man agent may be reduced considerably (which is investigated in experimental section 5.3).

We need a mechanism for conflict resolution, because different action modules may suggest different directions. We do this by assigning *priorities* to the modules. When the agent switches on an action module, she also decides about its priority. This is also a decision, and learning this decision is part of the learning task.[2]

2. Action priorities are learnt indirectly: each rule has a fixed priority, and when an action is switched on by a rule, it also inherits this priority. The same action can be switched on by different rules with different priorities. The mechanism is described in detail in section 4.6.

Table 3: List of action modules used for rule construction.

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 nearest edible (blue) ghost.
`FromGhost`	Go in direction opposite to the nearest ghost.
`ToSafeJunction`	Go towards the maximally safe junction. For all four directions, the "safety" of the nearest junction is estimated in that direction. If Ms. Pac-Man is n steps away from the junction and the nearest ghost is k steps away, then the safety value of this junction is $n - k$. A negative value means that Ms. Pac-Man possibly cannot reach that junction.
`FromGhostCenter`	Go in a direction which maximizes the Euclidean distance from the geometrical center of ghosts.
`KeepDirection`	Go further in the current direction, or choose a random available action (except turning back) if that is impossible.
`ToLowerGhostDensity`	go in the direction where the cumulative ghost density decreases fastest. Each ghost defines a density cloud (with radius = 10 and linear decay), from which the cumulative ghost density is calculated.
`ToGhostFreeArea`	Choose a location on the board where the minimum ghost distance is largest, and head towards it on the shortest path.

Table 4: List of observations used for rule construction. Distances denote the "length of the shortest path", unless noted otherwise. Distance to a particular object type is $+\infty$ if no such object exists at that moment.

Name	Description
`Constant`	Constant 1 value.
`NearestDot`	Distance of nearest dot.
`NearestPowerDot`	Distance of nearest power dot.
`NearestGhost`	Distance of nearest ghost.
`NearestEdGhost`	Distance of nearest edible (blue) ghost.
`MaxJunctionSafety`	For all four directions, the "safety" of the nearest junction in that direction is estimated, as defined in the description of action `ToSafeJunction`. The observation returns the value of the maximally safe junction.
`GhostCenterDist`	Euclidean distance from the geometrical center of ghosts.
`DotCenterDist`	Euclidean distance from the geometrical center of uneaten dots.
`GhostDensity`	Each ghost defines a density cloud (with radius = 10 and linear decay). Returns the value of the cumulative ghost density.
`TotalDistToGhosts`	"travelling salesman distance to ghosts:" the length of the shortest route that starts at Ms. Pac-Man and reaches all four ghosts (not considering their movement).

We implemented this with the following mechanism: a decision of the agent concerns action modules: the agent can either *switch on* or, *switch off* an action module. That is, in principle, the agent is able to use any subset of the action modules, instead of selecting a single one at each time step. Basically, the module with highest priority decides the direction of Ms. Pac-Man. If there are more than one equally ranked directions, then lower-priority modules are checked. If the direction cannot be decided after checking switched-on modules in the order of decreasing priority (for example, no module is switched on, or two directions are ranked equally by all switched-on modules), then a random direction is chosen.

Ms. Pac-Man can make decisions each time she advances a whole grid cell (the above mechanism ensures that she never stands still), according to 25 game ticks or approx. 0.2 seconds of simulated game time.

4.2 Observations, Conditions and Rules

Similarly to actions, we can easily define a list of observations which are potentially useful for decision making. The observations and their descriptions are summarized in Table 4.

Modules could have been improved in many ways, for example, checking whether there is enough time to intercept edible ghosts when calculating `NearestEdGhost` or taking into consideration the movement of ghosts when calculating `NearestGhost`, `NearestEdGhost` or `MaxJunctionSafety`. We kept the implementation of the modules as simple as possible. We designed reasonable modules, but no effort was made to make the module definitions optimal, complete or non-redundant.

Now we have the necessary tools for defining the conditions of a rule. A typical condition is true if its observations are in a given range. We note that the status of each action module is also important for proper decision making. For example, the agent may decide that if a ghost is very close, then she switches off all modules except the escape module. Therefore we allow conditions that check whether an action module is 'on' or 'off'.

For the sake of simplicity, conditions were restricted to have the form `[observation] < [value]`, `[observation] > [value]`, `[action]+`, `[action]-`, or the conjunction of such terms. For example,

```
(NearestDot<5) and (NearestGhost>8) and (FromGhost+)
```

is a valid condition for our rules.

Once we have conditions and actions, rules can be constructed easily. In our implementation, a rule has the form "`if [Condition], then [Action]`." For example,

```
if (NearestDot<5) and (NearestGhost>8) and (FromGhost+)
                                              then FromGhostCenter+
```

is a valid rule.

4.3 Constructing Policies from Rules

Decision lists are standard forms of constructing policies from single rules. This is the approach we pursue here, too. Decision lists are simply lists of rules, together with a mechanism that decides the order in which the rules are checked.

Each rule has a priority assigned. When the agent has to make a decision, she checks her rule list starting with the ones with highest priority. If the conditions of a rule are fulfilled, then the corresponding action is executed, and the decision-making process halts.

Note that in principle, the priority of a rule can be different from the priority of action modules. However, for the sake of simplicity, we make no distinction: if a rule with priority k *switches on* an action module, then the priority of the action module is also taken as k. Intuitively, this makes sense: if an important rule is activated, then its effect should also be important. If a rule with priority k *switches off* a module, then it is executed, regardless of the priority of the module.

It may be worth noting that there are many possible alternatives for ordering rules and actions:

- Each rule could have a fixed priority, as a part of the provided domain knowledge (Spronck, Ponsen, Sprinkhuizen-Kuyper, & Postma, 2006).

- The priority of a rule could be a free parameter that should be learned by the CEM method.

- Instead of absolute priorities, the agent could also learn the relative ordering of rules (Timuri, Spronck, & van den Herik, 2007).
- The order of rules could be determined by some heuristic decision mechanism. For example, the generality of the rule – e.g., rules with few/many conditions and large/small domains – could be taken into account. Such heuristics have been used in linear classifier systems (see e.g. the work of Bull & Kovacs, 2005)

In principle, one would like to find interesting solutions using the computer with minimal bias from 'domain knowledge'. In this regard, the efficiency of our simple priority management method was satisfactory, so we did not experiment with other priority heuristics.

4.4 An Example

Let us consider the example shown in Table 5. This is a rule-based policy for the Ms. Pac-Man agent.

Table 5: **A hand-coded policy for playing Ms. Pac-Man.** Bracketed numbers denote priorities, `[1]` is the highest priority.

```
[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-
```

The first two rules manage ghost avoidance: if a ghost is too close, then the agent should flee, and she should do so until she gets to a safe distance. Ghost avoidance has priority over any other activities. The next two rules regulate that if there is an edible ghost on the board, then the agent should chase it (the value of `NearestEdGhost` is infinity (> 99) if there are no edible ghosts, but it is ≤ 41 on our board, if there are). This activity has also relatively high priority, because eating ghosts is worth lots of points, but it must be done before the blueness of the ghosts disappears, so it must be done quickly. The fifth rule says that the agent should not turn back, if all directions are equally good. This rule prevents unnecessary zigzagging (while no dots are being eaten), and it is surprisingly effective. The remaining rules tweak the management of power dots. Basically, the agent prefers to eat a power dot. However, if there are blue ghosts on the board, then a power dot resets the score counter to 200, so it is a bad move. Furthermore, if ghost density is low around the agent, then most probably it will be hard to collect all of the ghosts, so it is preferable to wait with eating the power dot.

4.5 The Mechanism of Decision Making

The mechanism of decision making is depicted in Fig 2. In short, the (hidden) state-space is the world of the Ms. Pac-Man and the Ghosts. The dynamics of this (hidden) state-space determines the vector of observations, which can be checked by the conditions. If the conditions of a rule are satisfied, the corresponding action module is switched on or off. As a consequence, multiple actions may be in effect at once. For example, the decision depicted in Fig. 2 sets two actions to work together.

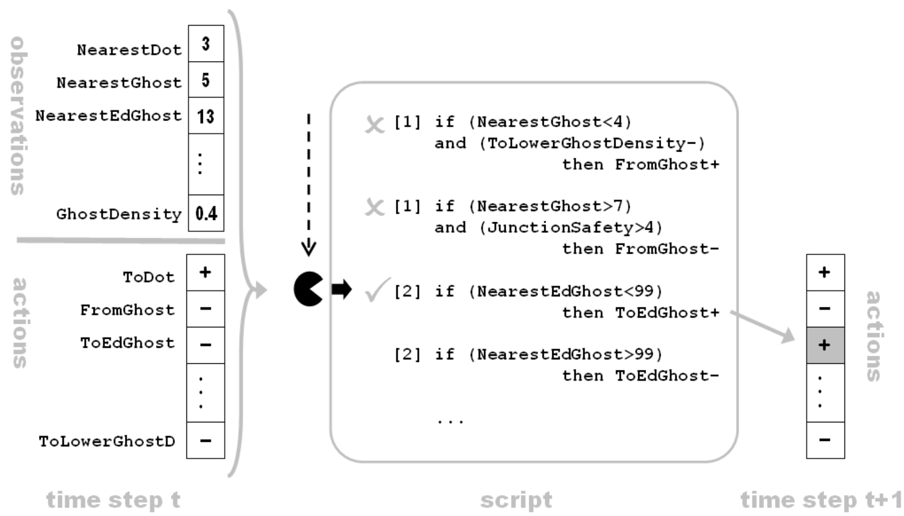

Figure 2: **Decision-making mechanism of the Ms. Pac-Man agent.** At time step t, the agent receives the actual observations and the state of her action modules. She checks the rules of her script in order, and executes the first rule with satisfied conditions.

Initially, each action module is in switched-off state. After a module has been switched on, it remains so until it is either explicitly switched off or another module of the same priority is switched on and replaces it.

4.6 Learning Rule-Based Policies by CEM

We will apply CEM for searching in the space of rule-based policies. Learning is composed of three phases: (1) the generation of random policies drawn according to the current parameter set, (2) evaluation of the policies, which consists of playing a game of Ms. Pac-Man to measure the score, and (3) updating the parameter set using the CEM update rules.

4.6.1 Drawing Random Scripts from a Predefined Rule-Base

Suppose that we have a predefined rule-base containing K rules (for example, the one listed in Appendix A). A policy has m *rule slots*. Each slot can be filled with any of the K rules, or left empty. As a result, policies could contain up to m rules, but possibly much less. Each rule slot has a fixed priority, too, from the set $\{1, 2, 3\}$.[3] The priority of a rule slot does not change during learning. Learning can, however, push an important rule to a high-priority slot from a low-priority one, and vice versa.

For each $1 \le i \le m$, slot i was filled with a rule from the rule-base with probability p_i, and left empty with probability $1 - p_i$. If it was decided that a slot should be filled, then a particular rule j ($1 \le j \le K$) was selected with probability $q_{i,j}$, where $\sum_{j=1}^{K} q_{i,j} = 1$ for each slot $i \in \{1, \dots, m\}$. As a result, policies could contain up to m rules, but possibly much less. Both the p_i values and the $q_{i,j}$ values are learnt simultaneously with the cross-entropy method (Table 2), using the update rules (1) and (2), respectively. This gives a total of $m + m \cdot K$ parameters to optimize (although the effective number of parameters is much less, because the $q_{i,j}$ values of unused slots are irrelevant). Initial probabilities are set to $p_i = 1/2$ and $q_{i,j} = 1/K$.

4.6.2 Drawing Random Rules without a Predefined Rule-Base

We studied situations with lessened domain knowledge; we did not use a predefined rule-base. Script generation was kept the same, but the rule-base of K rules was generated randomly. In this case we generated different rule-bases for each of the m rule slots; the low ratio of meaningful rules was counteracted by increased rule variety.

A random rule is a random pair of a randomly drawn condition set and a randomly drawn action. Random condition sets contained 2 conditions. A random action is constructed as follows: an action module is selected uniformly from the set of modules listed in Table 3, and switched on or off with probability 50%. The construction of a random condition starts with the uniformly random selection of a module from either Table 3 or Table 4. If the selected module is an action, then the condition will be `[action]-` or `[action]+` with equal probability. If the selected module is an observation, then the condition will be `[observation]<[value]` or `[observation]>[value]` with equal probability, where `[value]` is selected uniformly from a five-element set. The values in this set were determined separately for each observation module as follows: we played 100 games using a fixed policy and recorded the histogram of values for each observation. Subsequently, the five-element set was determined so that it would split the histogram into regions of equal area. For example, the value set for `NearestGhost` was $\{12, 8, 6, 5, 4\}$.

The design of the random rule generation procedure contains arbitrary elements (e.g. the number of conditions in a rule, the number of values an observation can be compared to). The intuition behind this procedure was to generate rules that are sufficiently versatile, but the ratio of "meaningless" rules (e.g. rules with unsatisfiable conditions) is not too large. However, no optimization of any form was done at this point.

5. Description of Experiments

According to our assumptions, the effectiveness of the above described architecture is based on three pillars: (1) the human domain knowledge provided by the modules and rules; (2)

3. According to our preliminary experiments, the quality of the learned policy did not improve by increasing the priority set or the number of the slots.

the effectiveness of the optimization algorithm; (3) the possibility of concurrent actions. Below, we describe a set of experiments that were designed to test these assumptions.

5.1 The Full Architecture

In the first experiment, random rules are used. In their construction, we use all the modules defined in sections 4.1 and 4.2. In the second experiment, rules were not generated randomly, but were hand-coded. In this case, the role of learning is only to determine which rules should be used.

5.1.1 Learning with Random Rule Construction

In the first experiment, the rule-base was generated randomly, as described in section 4.6.2. The number of rule slots was fixed to $m = 100$ (priorities were distributed evenly), each one containing $K = 100$ randomly generated rules. The values for K and m were selected by coarse search over parameter space.

The parameters of CEM were as follows: population size $N = 1000$, selection ratio $\rho = 0.05$, step size $\alpha = 0.6$.[4] These values for ρ and α are fairly standard for CEM, and we have not tried varying them. In each step, the probabilities of using a rule slot (that is, the values p_i, but not $q_{i,j}$) were slightly decreased, by using a decay rate of $\beta = 0.98$. With larger decay rate, useful rules were also annulled too often. On the other hand, smaller decay did not affect the performance, but many superfluous rules were left in the policies.

The score of a given policy has huge variance due to the random factors in the game. Therefore, to obtain reliable fitness estimations, the score of each policy was averaged over 3 subsequent games. Learning lasted for 50 episodes, which was sufficient to tune each probability close to either 0 or 1. We performed 10 parallel training runs. This experiment type is denoted as CE-RandomRB.

5.1.2 Learning with Hand-Coded Rules

In the second experiment we constructed a rule-base of $K = 42$ hand-coded rules (shown in Appendix A) that were thought to be potentially useful. These could be placed in one of the $m = 30$ rule slots.[5] Other parameters of the experiment were identical to the previous one. This experiment type is denoted as CE-FixedRB.

5.2 The Effect of the Learning Algorithm

In the following experiment, we compared the performance of CEM to simple stochastic gradient optimization. This single comparison is not sufficient to measure the efficiency of CEM; it serves to provide a point of reference. The comparison is relevant, because these algorithms are similar in complexity and both of them move gradually towards the best samples that were found so far. The difference is that SG maintains a single solution

4. Note that α is the per-episode learning rate. This would correspond to a per-instance learning rate of $\alpha' = \alpha/(\rho \cdot N) = 0.012$ for an on-line learning algorithm.
5. In contrast to the previous experiment, all of the rules are meaningful and potentially useful. Therefore there is no need for a large pool of rules, and a much lower m can be used. We found that the algorithm is fairly insensitive to the choice of m; significant changes in performance can be observed if parameter m is modified by a factor of 3.

candidate at a time, whereas CEM maintains a distribution over solutions. Thus, CEM maintains a memory over solutions and becomes less fragile to occasional wrong parameter changes.

The particular form of stochastic gradient search was the following: the initial policy was drawn at random (consisting of 6 rules). After that, we generated 100 random mutation of the current solution candidate at each step, and evaluated the obtained policies. The best-performing mutation was chosen as the next solution candidate. Mutations were generated using the following procedure: (1) in each rule, each condition was changed to a random new condition with probability 0.05; (2) in each rule, the action was changed to a random new action with probability 0.05. The listed parameter values (number of rules in policy, number of mutated policies, probabilities of mutation) were the results of coarse parameter-space optimization.

The number of episodes was set to 500. This way, we evaluated the same number of different policies (50,000) as in the CEM experiments. Both the random rule-base and the fixed rule-base experiments were repeated using the stochastic gradient method, executing 10 parallel training runs. The resulting policies are denoted as SG-RandomRB and SG-FixedRB, respectively.

5.3 The Effect of Parallel Actions

According to our assumptions, the possibility of parallel actions plays a crucial role in the success of our architecture. To confirm this assumption, we repeated previous experiments with concurrent actions disabled. If the agent switches on an action module, all other action modules are switched off automatically. These experiment types are denoted as CE-RandomRB-1action, CE-FixedRB-1action, SG-RandomRB-1action and SG-FixedRB-1action.

5.4 Baseline Experiments

In order to isolate and assess the contribution of learning, we performed two additional experiments with different amounts of domain knowledge and no learning. Furthermore, we asked human subjects to play the game.

5.4.1 Random Policies

In the first non-learning experiment, we used the rule-base of 42 hand-coded rules (identical to the rule-base of CE-FixedRB). Ten rules were selected at random, and random priorities were assigned to them. We measured the performance of policies constructed this way.

5.4.2 Hand-Coded Policy

In the second non-learning experiment, we hand-coded both the rules and the priorities, that is, we hand-coded the full policy. The policy is shown in Table 5, and has been constructed by some trial-and-error. Naturally, the policy was constructed before knowing the results of the learning experiments.

Table 6: Ms. Pac-Man results. See text for details. Abbreviations: CE: learning with the cross-entropy method, SG: learning with stochastic gradient, randomRB: randomly generated rule-base, fixedRB: fixed, hand-coded rule-base, 1action: only one action module can work at a time.

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)[6]
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)

5.4.3 Human Play

In the final experiment, five human subjects were asked to play the first level of Ms. Pac-Man and we measured their performance. Each of the subjects has played Pac-Man and/or similar games before, but none of them was an experienced player.

6. Experimental Results

Human experiments were performed on the first level of an open-source Pac-Man clone of Courtillat (2001). For the other experiments we applied the Delphi re-implementation of the code.

In all learning experiments, 10 parallel learning runs were executed, each one for 50 episodes. This training period was sufficient to tune all probabilities close to either 0 or 1, so the learned policy could be determined unambiguously in all cases. Each obtained policy was tested by playing 50 consecutive games, giving a total of 500 test games per experiment. In the non-learning experiments the agents played 500 test games, too, using random policies and the hand-coded policy, respectively. Each human subject played 20 games, giving a total of 100 test games. Results are summarized in Table 6. We provide 25% and 75% percentile values instead of the variances, because the distribution of scores is highly non-Gaussian.

6. The fact that the average is smaller than the 25% percentile is caused by a highly skewed distribution of scores. In most games, the agent reached a score in the range 5800 ± 300, except for a few games with extremely low score. These few games did not affect the 25% percentile but lowered the average significantly.

```
[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+
```

Figure 3: Best policy learned by CE-fixedRB. Average score over 50 games: 9480.

```
[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+
```

Figure 4: Best policy learned by CE-randomRB. Average score over 50 games: 7199. Note the presence of always-true (and thus, superfluous) conditions like `ToLowerGhostDensity-`, `FromGhostCenter-`, `ToGhostFreeArea-` or `ToDot-`.

Fig. 3 shows the best individual policy learned by CE-fixedRB, reaching 9480 points on average. Ghost avoidance is given highest priority, but is only turned on when a ghost is very close. Otherwise Ms. Pac-Man concentrates on eating power dots and subsequently eating the blue ghosts. She also takes care not to eat any power dot while there are blue ghosts on the board, because otherwise she would miss the opportunity to eat the 1600-point ghost (and possibly several others, too). With lowest priority setting, the agent looks for ordinary dots, although this rule is in effect only when the previous rules can not decide on a direction (for example, in the endgame when there are no power dots left and all ghosts are in their original form).

Policies learnt by CE-randomRB behave similarly to the ones learnt by CE-fixedRB, although the behavior is somewhat obscured by superfluous conditions and/or rules, as demonstrated clearly on the example policy shown in Fig. 4. Because of the "noise" generated by the random rules, the algorithm often fails to learn the correct priorities of various activities.

The effect of enabling/disabling concurrent actions is also significant. It is instructive to take a look at the best policy learned by CE-fixedRB-1action shown in Fig. 5: the agent has to concentrate on eating ghosts, as it is the major source of reward. However, she cannot use modules that are necessary for ghost avoidance and long-term survival.

The results also show that CEM performs significantly better than stochastic gradient learning. We believe, however, that this difference could be lowered with a thorough search over the parameter space. SG and many other global optimization methods like evolutionary methods or simulated annealing could reach similar performances to CEM. According to de Boer et al. (2004) and the applications cited in section 7.2, an advantage of CEM is

```
[2] if NearestEdGhost>99 then ToPowerDot+
[2] if NearestEdGhost<99 then ToEdGhost+
```

Figure 5: Best policy learned by CE-fixedRB-1action. Average score over 50 games: 6041.

that it maintains a distribution of solutions and can reach robust performance with very little effort, requiring little or no tuning of the parameters: there is a canonical set of parameters ($0.01 \leq \rho \leq 0.1$, $0.5 \leq \alpha \leq 0.8$, population as large as possible) for which the performance of the method is robust. This claim coincides with our experiences in the parameter-optimization process.

Finally, it is interesting to analyze the differences between the tactics of human and computer players. One fundamental tactic of human players is that they try to lure the ghosts close to Ms. Pac-Man such that all ghosts are very close to each other. This way, all of them can be eaten fast when they turn blue. No such behavior evolved in any of our experiments. Besides, there are other tactics that CEM has no chance to discover, because it is lacking the appropriate sensors. For example, a human player can (and does) calculate the time remaining from the blue period, the approximate future position of ghosts, and so on.

7. Related Literature

In this section, we review literature on learning the Pac-Man game, and on various components of our learning architecture: the cross-entropy method, rule-based policies, and concurrent actions.

7.1 Previous Work on (Ms.) Pac-Man

Variants of Pac-Man have been used previously in several studies. A direct comparison of performances is possible only in a few cases, however, because simplified versions of the game are used in most of the other studies.

Koza (1992) uses Ms. Pac-Man as an example application for genetic programming. It uses different score value for the fruit (worth 2000 points instead of the 100 points used here), and the shape of the board (and consequently, the number of dots) is also different, therefore scores cannot be directly compared. However, Koza reports (on p. 355) that "The Pac Man could have scored an additional 9000 points if he had captured all four monsters on each of the four occasions when they turned blue". This score, the only one reported, translates to approximately 5000 points in our scoring system.

Lucas (2005) also uses the full-scale Ms. Pac-Man game as a test problem. He trains a neural network position evaluator with hand-crafted input features. For the purposes of training, he uses an *evolutionary strategy* approach. The obtained controller was able to reach 4781 ± 116 points, averaged over 100 games.

Bonet and Stauffer (1999) restrict observations to a 10×10 window centered at Ms. Pac-Man, and uses a neural network and temporal-difference learning to learn a reactive controller. Through a series of increasingly difficult learning tasks, they were able to teach basic pellet-collecting and ghost-avoidance behaviors in greatly simplified versions of the game: they used simple mazes containing no power dots and only one ghost.

Gallagher and Ryan (2003) defines the behavior of the agent as a parameterized finite state automata. The parameters are learnt by *population-based incremental learning*, an evolutionary method similar to CEM. They run a simplified version of Pac-Man; they had a single ghost and had no power dots, which takes away most of the complexity of the game.

Tiong (2002) codes rule-based policies for Pac-Man by hand, but uses no learning to improve them. His tests, similarly to ours, are based on the Pac-Man implementation of Courtillat (2001), but he limits the number of ghosts to 1. The best-performing rule set reaches 2164 points on average out of the maximal 2700. However, the results are not likely to scale up well with increasing the number of ghosts: the ghost is eaten only 1.4 times on average (out of the possible 4 times per game).[7]

7.2 The Cross-Entropy Method

The cross-entropy method of Rubinstein (1999) is a general algorithm for global optimization tasks, bearing close resemblance to estimation-of-distribution evolutionary methods (see e.g. the paper of Muehlenbein, 1998). The areas of successful application range from combinatorial optimization problems like the *optimal buffer allocation problem* (Allon, Kroese, Raviv, & Rubinstein, 2005), *DNA sequence alignment* (Keith & Kroese, 2002) to *independent process analysis* (Szabó, Póczos, & Lőrincz, 2006).

The cross-entropy method has several successful reinforcement learning applications, too: Dambreville (2006) uses CEM for learning an input-output hierarchical HMM that controls a predator agent in a partially observable grid world; Menache, Mannor, and Shimkin (2005) use radial basis function approximation of the value function in a continuous maze navigation task, and use CEM to adapt the parameters of the basis functions; and finally, Mannor, Rubinstein, and Gat (2003) apply CEM to policy search in a simple grid world maze navigation problem. Recently, the cross-entropy method has also been applied successfully to the game Tetris by Szita and Lőrincz (2006).

7.3 Rule-Based Policies

The representation of policies as rule sequences is a widespread technique for complex problems like computer games. As an example, many of the Pac-Man-related papers listed above use rule-based representation.

Learning classifier systems (Holland, 1986) are genetic-algorithm based methods to evolve suitable rules for a given task. Bull (2004) gives an excellent general overview and pointers to further references. The *Hayek machine* of Baum (1996) is a similar architecture, where agents (corresponding to simple rules) define an economical system: they make bids for executing tasks in the hope that they can obtain rewards. Schaul (2005) applies this architecture for the *Sokoban* game.

Dynamic scripting (Spronck et al., 2006) is another prominent example of using and learning rule-based policies. It uses a hand-coded rule-base and a reinforcement-learning-

7. Results are cited from section 3.6.

like principle to determine the rules that should be included in a policy. Dynamic scripting has successful applications in state-of-the-art computer games like the role-playing game *Neverwinter Nights* (Spronck et al., 2006) and the real-time strategy game *Wargus* (Ponsen & Spronck, 2004).

7.4 Concurrent Actions

In traditional formalizations of RL tasks, the agent can select and execute a single action at a time. The only work known to us that handles concurrent actions explicitly is that of Rohanimanesh and Mahadevan (2001). They formalize RL tasks with concurrent actions in the framework of semi-Markov decision processes and present simple grid world demonstrations.

8. Summary and Closing Remarks

In this article we have proposed a method that learns to play Ms. Pac-Man. We have defined a set of high-level observation and action modules with the following properties: (i) actions are temporally extended, (ii) actions are not exclusive, but may work concurrently. Our method can uncover action combinations together with their priorities. Thus, our agent can pursue multiple goals in parallel.

The decision of the agent concerns whether an action module should be turned on (if it is off) or off (if it is on). Furthermore, decisions depend on the current observations and may depend on the state of action modules. The policy of the agent is represented as a list of if-then rules with priorities. Such policies are easy to interpret and analyze. It is also easy to incorporate additional human knowledge. The cross-entropy method is used for learning policies that play well. Learning is biased towards low-complexity policies, which is a consequence of both the policy representation and the applied cross entropy learning method. For CEM, higher complexity solutions are harder to discover and special means should be used to counteract premature convergence. For solutions of higher complexities, noise injection has been suggested in our previous work (Szita & Lőrincz, 2006). Learned low complexity policies reached better score than a hand-coded policy or the average human players.

The applied architecture has the potentials to handle large, structured observation- and action-spaces, partial observability, temporally extended and concurrent actions. Despite its versatility, policy search can be effective, because it is biased towards low-complexity policies. These properties are attractive from the point of view of large-scale applications.

8.1 The Role of Domain Knowledge

When demonstrating the abilities of an RL algorithm, it is desirable that learning starts from scratch, so that the contribution of learning is clearly measurable. However, the choices of test problems are often misleading: many 'abstract' domains contain considerable amount of domain knowledge implicitly. As an example, consider grid world navigation tasks, an often used class of problems for *tabula rasa* learning.

In a simple version of the grid world navigation task, the state is an integer that uniquely identifies the position of the agent, and the atomic actions are moves to grid cells north/-south/east/west from the actual cell. More importantly, the unique identification of the position means that the moves of the agent do not change the direction of the agent and the task is in laboratory coordinate framework, sometimes called allocentric coordinates, and not in egocentric coordinates. The concepts of north, south, etc. correspond to very high-level abstraction, they have a meaning to humans only, so they must be considered as part of the domain knowledge. The domain knowledge provided by us is similar to the grid world in the sense that we also provide high-level observations in allocentric form, such as 'distance of nearest ghost is *d*' or 'Ms. Pac-Man is at position $(11, 2)$'. Similarly, action 'go north' and action 'go towards the nearest power dot' are essentially of the same level.

The implicit presence of high-level concepts becomes even more apparent as we move from abstract MDPs to the 'real world'. Consider a robotic implementation of the maze task: the full state information, i.e. its own state as well as the state of the environment is not available for the robot. It sees only local features and it may not see all local features at a time. To obtain the exact position, or to move one unit's length in the prescribed direction, the robot has to integrate information from movement sensors, optical/radar sensors etc. Such information fusion, although necessary, is not a topic of reinforcement learning. Thus, in this task, there is a great amount of domain knowledge that needs to be provided before our CE based policy search method could be applied.

In our opinion, the role of human knowledge is that it selects the set of observations and actions that suit the learning algorithm. Such extra knowledge is typically necessary for most applications. Nonetheless, numerous (more-or-less successful) approaches exist for obtaining such domain knowledge automatically. According to one approach, the set of observations is chosen from a rich (and redundant) set of observations by some feature selection method. The cross-entropy method seems promising here, too (see the paper of Szita, 2006, for an application to feature selection from brain fMRI data at the 2006 Pittsburgh Brain Activity Interpretation Competition). According to a different approach, successful combinations of lower level rules can be joined into higher level concepts/rules. Machine learning has powerful tools here, e.g. arithmetic coding for data compression (Witten, Neal, & Cleary, 1987). It is applied in many areas, including the writing tool Dasher developed by Ward and MacKay (2002). Such extensions are to be included into the framework of reinforcement learning.

8.2 Low-Complexity Policies

The space of legal policies is huge (potentially infinite), so it is an interesting question how search can be effective in such huge space. Direct search is formidable. We think that an implicit bias towards low-complexity policies can be useful and this is what we studied here. By low-complexity policy, we mean the following: The policy may consist of very many rules, but in most cases, only a few of them are applied concurrently. Unused rules do not get rewarded nor do they get punished unless they limit a useful rule, so the *effective length* of policies is biased towards short policies. This implicit bias is strengthened by an explicit one in our work: in the absence of explicit reinforcement, the probability of applying a rule decays, so indifferent rules get wiped out quickly. It seems promising to use frequent low complexity rule combinations as building blocks in a continued search for more powerful but still low-complexity policies.

The bias towards short policies reduces the effective search space considerably. Moreover, for many real-life problems, low-complexity solutions exist (for an excellent analysis of possible reasons, see the paper of Schmidhuber, 1997). Therefore, search is concentrated on a relevant part of the policy space, and pays less attention to more complex policies (which are therefore less likely according to Occam's razor arguments.)

Acknowledgments

Please send correspondence to András Lőrincz. The authors would like to thank the anonymous reviewers for their detailed comments and suggestions for improving the presentation of the paper. This material is based upon work supported partially by the European Office of Aerospace Research and Development, Air Force Office of Scientific Research, Air Force Research Laboratory, under Contract No. FA-073029. This research has also been supported by an EC FET grant, the 'New Ties project' under contract 003752. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the European Office of Aerospace Research and Development, Air Force Office of Scientific Research, Air Force Research Laboratory, the EC, or other members of the EC New Ties project.

Appendix A. The Hand-Coded Rule-Base

Below is a list of rules of the hand-coded rule-base used in the experiments.

```
if Constant>0 then ToDot+
if Constant>0 then ToCenterofDots+
if NearestGhost<4 then FromGhost+
if NearestGhost<3 then FromGhost+
if NearestGhost<5 then FromGhost+
if NearestGhost>5 then FromGhost-
if NearestGhost>6 then FromGhost-
if NearestGhost>7 then FromGhost-
if Constant>0 then ToSafeJunction+
if MaxJunctionSafety<3 then ToSafeJunction+
if MaxJunctionSafety<1 then ToSafeJunction+
if MaxJunctionSafety>3 then ToSafeJunction-
if MaxJunctionSafety>3 then FromGhost-
if MaxJunctionSafety>5 then ToSafeJunction-
if MaxJunctionSafety>5 then FromGhost-
if Constant>0 then KeepDirection+
if Constant>0 then ToEdGhost+
if NearestGhost<4 then ToPowerDot+
if NearestEdGhost<99 then ToPowerDot-
if NearestEdGhost<99 and NearestPowerDot<5 then FromPowerDot+
if NearestEdGhost<99 then FromPowerDot+
if NearestEdGhost>99 then FromPowerDot-
if NearestEdGhost>99 then ToPowerDot+
if GhostDensity>1 then ToLowerGhostDensity+
if GhostDensity<0.5 then ToLowerGhostDensity-
```

```
if NearestPowerDot<2 and NearestGhost<5 then ToPowerDot+
if NearestGhost>7 and MaxJunctionSafety>4 then FromGhost-
if GhostDensity<1.5 and NearestPowerDot<5 then FromPowerDot+
if NearestPowerDot>10 then FromPowerDot-
if TotalDistToGhosts>30 then FromPowerDot+
if MaxJunctionSafety<3 then FromGhost+
if MaxJunctionSafety<2 then FromGhost+
if MaxJunctionSafety<1 then FromGhost+
if MaxJunctionSafety<0 then FromGhost+
if Constant>0 then FromGhostCenter+
if NearestGhost<4 then FromGhost+
if NearestGhost>7 and MaxJunctionSafety>4 then FromGhost-
if NearestEdGhost>99 then ToEdGhost-
if NearestEdGhost<99 then ToEdGhost+
if FromPowerDot- then ToPowerDot+
if GhostDensity<1.5 and NearestPowerDot<5 then FromPowerDot+
if NearestPowerDot>10 then FromPowerDot-
```

References

Allon, G., Kroese, D. P., Raviv, T., & Rubinstein, R. Y. (2005). Application of the cross-entropy method to the buffer allocation problem in a simulation-based environment. *Annals of Operations Research*, *134*, 137–151.

Baum, E. B. (1996). Toward a model of mind as a laissez-faire economy of idiots. In *Proceedings of the 13rd International Conference on Machine Learning*, pp. 28–36.

Baxter, J., Tridgell, A., & Weaver, L. (2001). *Machines that learn to play games*, chap. Reinforcement learning and chess, pp. 91–116. Nova Science Publishers, Inc.

Bertsekas, D. P., & Tsitsiklis, J. N. (1996). *Neuro-Dynamic Programming*. Athena Scientific.

Bonet, J. S. D., & Stauffer, C. P. (1999). Learning to play Pac-Man using incremental reinforcement learning.. [Online; accessed 09 October 2006].

Bull, L. (2004). *Applications of Learning Classifier Systems*, chap. Learning Classifier Systems: A Brief Introduction, pp. 3–13. Springer.

Bull, L., & Kovacs, T. (2005). *Foundations of Learning Classifier Systems*, chap. Foundations of Learning Classifier Systems: An Introduction, pp. 3–14. Springer.

Courtillat, P. (2001). NoN-SeNS Pacman 1.6 with C sourcecode.. [Online; accessed 09 October 2006].

Dambreville, F. (2006). Cross-entropic learning of a machine for the decision in a partially observable universe. *Journal of Global Optimization*. To appear.

de Boer, P.-T., Kroese, D. P., Mannor, S., & Rubinstein, R. Y. (2004). A tutorial on the cross-entropy method. *Annals of Operations Research*, *134*, 19–67.

Gallagher, M., & Ryan, A. (2003). Learning to play pac-man: An evolutionary, rule-based approach. In et. al., R. S. (Ed.), *Proc. Congress on Evolutionary Computation*, pp. 2462–2469.

Holland, J. H. (1986). Escaping brittleness: The possibilities of general-purpose learning algorithms applied to parallel rule-based systems. In Mitchell, Michalski, & Carbonell (Eds.), *Machine Learning, an Artificial Intelligence Approach. Volume II*, chap. 20, pp. 593–623. Morgan Kaufmann.

Keith, J., & Kroese, D. P. (2002). Sequence alignment by rare event simulation. In *Proceedings of the 2002 Winter Simulation Conference*, pp. 320–327.

Koza, J. (1992). *Genetic programming: on the programming of computers by means of natural selection*. MIT Press.

Lucas, S. M. (2005). Evolving a neural network location evaluator to play Ms. Pac-Man. In *IEEE Symposium on Computational Intelligence and Games*, pp. 203–210.

Mannor, S., Rubinstein, R. Y., & Gat, Y. (2003). The cross-entropy method for fast policy search. In *20th International Conference on Machine Learning*.

Margolin, L. (2004). On the convergence of the cross-entropy method. *Annals of Operations Research*, *134*, 201–214.

Menache, I., Mannor, S., & Shimkin, N. (2005). Basis function adaptation in temporal difference reinforcement learning. *Annals of Operations Research*, *134*(1), 215–238.

Muehlenbein, H. (1998). The equation for response to selection and its use for prediction. *Evolutionary Computation*, *5*, 303–346.

Ponsen, M., & Spronck, P. (2004). Improving adaptive game AI with evolutionary learning. In *Computer Games: Artificial Intelligence, Design and Education*.

Rohanimanesh, K., & Mahadevan, S. (2001). Decision-theoretic planning with concurrent temporally extended actions. In *Proceedings of the 17th Conference on Uncerainty in Artificial Intelligence*, pp. 472–479.

Rubinstein, R. Y. (1999). The cross-entropy method for combinatorial and continuous optimization. *Methodology and Computing in Applied Probability*, *1*, 127–190.

Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, *6*, 211–229.

Schaul, T. (2005). Evolving a compact concept-based Sokoban solver. Master's thesis, École Polytechnique Fédérale de Lausanne.

Schmidhuber, J. (1997). A computer scientist's view of life, the universe, and everything. In Freksa, C., Jantzen, M., & Valk, R. (Eds.), *Foundations of Computer Science: Potential - Theory - Cognition*, Vol. 1337 of *Lecture Notes in Computer Science*, pp. 201–208. Springer, Berlin.

Spronck, P., Ponsen, M., Sprinkhuizen-Kuyper, I., & Postma, E. (2006). Adaptive game ai with dynamic scripting. *Machine Learning*, *63*(3), 217–248.

Spronck, P., Sprinkhuizen-Kuyper, I., & Postma, E. (2003). Online adaptation of computer game opponent AI. In *Proceedings of the 15th Belgium-Netherlands Conference on Artificial Intelligence*, pp. 291–298.

Sutton, R. S., & Barto, A. G. (1998). *Reinforcement Learning: An Introduction*. MIT Press, Cambridge.

Szabó, Z., Póczos, B., & Lőrincz, A. (2006). Cross-entropy optimization for independent process analysis. In *ICA*, pp. 909–916.

Szita, I. (2006). How to select the 100 voxels that are best for prediction – a simplistic approach. Tech. rep., Eötvös Loránd University, Hungary.

Szita, I., & Lőrincz, A. (2006). Learning Tetris using the noisy cross-entropy method. *Neural Computation*, *18*(12), 2936–2941.

Tesauro, G. (1994). TD-Gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, *6*(2), 215–219.

Timuri, T., Spronck, P., & van den Herik, J. (2007). Automatic rule ordering for dynamic scripting. In *The Third Artificial Intelligence and Interactive Digital Entertainment Conference*, pp. 49–54.

Tiong, A. L. K. (2002). Rule set representation and fitness functions for an artificial pac man playing agent. Bachelor's thesis, Department of Information Technology and Electrical Engineering.

Ward, D. J., & MacKay, D. J. C. (2002). Fast hands-free writing by gaze direction. *Nature*, *418*, 838–540.

Wikipedia (2006). Pac-Man — Wikipedia, the free encyclopedia. Wikipedia. [Online; accessed 20 May 2007].

Witten, I. A., Neal, R. M., & Cleary, J. G. (1987). Arithmetic coding for data compression. *Communications of the ACM*, *30*, 520–540.

Wolpert, D. H., & Macready, W. G. (1997). No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, *1*, 67–82.