Reinforcement Learning to Train Ms. Pac-Man Using Higher-order Action-relative Inputs

Luuk Bom, Ruud Henken and Marco Wiering (*IEEE Member*)
Institute of Artificial Intelligence and Cognitive Engineering
Faculty of Mathematics and Natural Sciences
University of Groningen, The Netherlands

***Abstract*—Reinforcement learning algorithms enable an agent to optimize its behavior from interacting with a specific environment. Although some very successful applications of reinforcement learning algorithms have been developed, it is still an open research question how to scale up to large dynamic environments. In this paper we will study the use of reinforcement learning on the popular arcade video game Ms. Pac-Man. In order to let Ms. Pac-Man quickly learn, we designed particular smart feature extraction algorithms that produce higher-order inputs from the game-state. These inputs are then given to a neural network that is trained using Q-learning. We constructed higher-order features which are relative to the action of Ms. Pac-Man. These relative inputs are then given to a single neural network which sequentially propagates the action-relative inputs to obtain the different Q-values of different actions. The experimental results show that this approach allows the use of only 7 input units in the neural network, while still quickly obtaining very good playing behavior. Furthermore, the experiments show that our approach enables Ms. Pac-Man to successfully transfer its learned policy to a different maze on which it was not trained before.**

I. Introduction

Reinforcement learning (RL) algorithms [1], [2] are attractive for learning to control an agent in different environments. Although some very successful applications of RL exist, such as for playing backgammon [3] and for dispatching elevators [4], it still remains an issue how to deal effectively with large state spaces in order to obtain very good results with little training time. This paper describes a novel approach which is based on using low-complexity solutions [5], [6] in order to train Ms. Pac-Man effectively at playing the game. The low-complexity solution is obtained by using smart input feature extraction algorithms that transform the high-dimensional game-state to only a hand full of features that characterize the important elements of the environmental state.

Ms. Pac-Man was released in 1982 as an arcade video game, and has since become one of the most popular video games of all time. The simplicity of the game rules in combination with the complex strategies that are required to obtain a proper score, have made Ms. Pac-Man an interesting research topic in Artificial Intelligence [7]. The game of Ms. Pac-Man meets all the criteria of a reinforcement learning task [8]. The environment is difficult to predict, because the ghost behaviour is stochastic. The reward function can be well defined by relating it to in-game events, such as collecting a pill. Furthermore, there is a small action space, which consists of the four directions in which Ms. Pac-Man can move: left, right, up and down. However, because agents are always in the process of moving and there are many possible places for pills in a maze, there is a huge state space for which a large amount of values are required to describe a single game state. This prohibits the agent from calculating optimal solutions, which means they must be approximated in some way. It makes the game an interesting example of the reinforcement learning problem. Previous research on Pac-Man and Ms. Pac-Man often imposed a form of simplification on the game. For example, by limiting the positions of agents to discrete squares in the maze [9], [10]. To decrease the amount of values to describe a state, the size of the maze has been reduced as well [8].

The agent we constructed consists of a multi-layer perceptron [11] trained using reinforcement learning. This method of machine learning has yielded promising results with regards to artificial agents for games [12]. Recently, it has been used for training an agent in playing a first-person shooter [13], as well as for the real-time strategy game Starcraft [14]. The performances of reinforcement learning and an evolutionary approach have been compared regarding the board-game Go, in which game strategies rely heavily on in-game positions as well. Reinforcement learning was found to improve the performance of the neural network faster than evolutionary learning [15], however this may be specific to Go, and the research question of using reinforcement learning or evolutionary algorithms to optimize agents is still an open problem. In this paper we combine neural networks with Q-learning [16], [17] .

We pose that a higher-order representation of input values relative to Ms. Pac-Man's current position would better suit the game environment than a direct representation of all details. Where an absolute representation of pill positions would require a binary input value for every possible position, very useful input information about pills could also be represented using four continuous input values that express the distance to the closest pill for each direction around Ms. Pac-Man. The use of few inputs to characterize the game-state in Ms. Pac-Man has also been used in [18] and [8]. However, in [18] elementary inputs are used and they used evolutionary neural networks instead of value-function based RL. In [8] rule-based policies were learned, where the rules were human designed and the values were learned with RL, also using few features to describe the state.

There are numerous benefits associated with this approach. The amount of inputs required to describe the state of the game is very low, allowing faster training. The influence of an input value on the desirability of actions can be easily established, making training more effective. The resulting neural network is trained independently of maze dimensions and structure, which allows the agent to exhibit its learned policy in any maze. Finally our approach makes any form of restricting the maze dimensions or positions of agents obsolete.

A common approach to reinforcement learning with function approximation uses multiple action neural networks [19], which all use the entire state representation as input. Each network is trained and slowly learns to integrate this information into an output value representing the desirability of a specific action. The networks must slowly converge to expressing desirability on the same scale, or the agent will be unable to select the best action. In this paper, we argue for the use of a single action neural network that can receive inputs associated with or related to a single direction (action). At each time step the input for each direction is propagated separately, resulting in again multiple action values. This structure imposes that inputs will be weighted the same way for every direction and that the desirability of actions will be expressed on one scale.

Contributions. There are a number of major contributions to the field of RL in this paper. First of all, we show that it is possible to use few higher-order inputs in order to capture the most important elements of the game of Ms. Pac-Man. Second, we show that the use of single neural networks with action-relative inputs allows for training Ms. Pac-Man with only 7 input neurons, while the learned behavior is still very good. Furthermore, we show that these higher-order relative inputs also allow for effective policy transfer to a different maze, on which Ms. Pac-Man was not trained before.

This article will attempt to answer three research questions: (1) Is a neural network trained using Q-learning able to produce good playing behavior in the game of Ms. Pac-Man? (2) How can we construct higher-order inputs to describe the game states? (3) Does incorporating a single action neural network offer benefits over the use of multiple action neural networks?

Outline. Section II describes the framework we constructed to simulate the game and train the agent. Section III discusses the theory behind the used reinforcement learning algorithms. Section IV outlines the various input algorithms that were constructed to represent game states. Section V describes the experiments that were conducted and their results. Finally, we present our conclusions in Section VI.

II. Framework

We developed a framework that implements a simulation of the game (see figure 1), holding a representation of all agents and objects, and their properties and states. A number of input algorithms were designed that transform the in-game situation into numeric feature input values. At every time step, neural networks generate a decision on which action to take by propagating the input through the neurons. When a move is carried out, a reward representing the desirability of the action is fed back by the simulation. Reinforcement learning is then employed to alter the weights between nodes of the neural network, which corresponds with a change in game strategy.

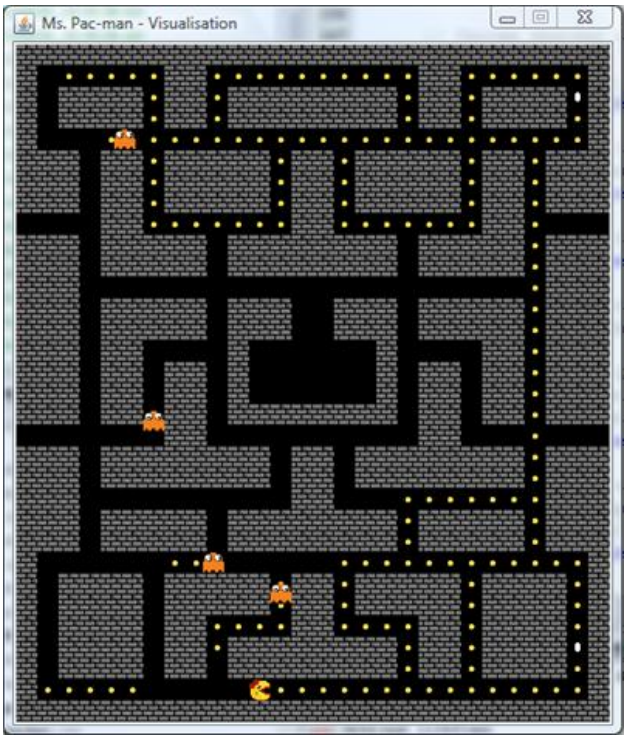

Figure 1. Screenshot of the game simulation in the first maze. The maze structure matches that of the first maze in the original game.

In the game, Ms. Pac-Man must navigate through a number of mazes filled with pills. As a primary objective, all of these pills must be collected to successfully finish the level. Unfortunately, a swarm of four ghosts has set out to make this task as hard as possible. When Ms. Pac-Man collides with one of these ghosts, she will lose a life. To stay in the game long enough to collect all pills, avoiding these ghosts is another primary objective. Four powerpills are scattered throughout the maze. When Ms. Pac-Man collects one of these, the ghosts will become afraid for a small period of time. In this state, the ghosts are not dangerous to Ms. Pac-Man. In fact, a collision between a scared ghost and Ms. Pac-Man will result in a specific amount of bonus points. Levels contain warp tunnels, which allow Ms. Pac-Man and the ghosts to quickly travel from one side of the maze to the other.

Some small differences exist between our simulation and the original game. The most notable is the moving speed of agents. For example, in the original game the moving speed of Ms. Pac-Man seems dependent on a number of factors, such as whether she is eating a pill and whether she is moving through one of the warp tunnels. In our simulation, Ms. Pac-Man's speed is fixed, independent of these factors. The original game consists of four different mazes. Players have three lifes to complete as much of these levels as they can. Our simulation features duplicates of three of these mazes and Ms. Pac-Man

has only one life. If the level is completed successfully the game ends in a win, as opposed to the original game where a new level would commence. If Ms. Pac-Man collides with a ghost the game ends in a loss, as opposed to the original game where the player would be allowed to continue as long as she has one or more lifes left. The ghost behavior has been modeled after the original game as well. Unfortunately, little documentation exists on the specifics of the original ghost behavior. This prevented us from implementing an exact copy. Finally, in our simulation we did not model the bonus fruits that are sometimes spawned in the original game.

Despite these differences, our simulation still closely resembles the original game. Given this fact, it is safe to assume that an agent trained using our simulation, would be able to perform at a comparable level when playing the original game.

III. Reinforcement Learning

Reinforcement learning is used to train a neural network on the task of playing the game of Ms. Pac-Man. This method differs from standard supervised learning in that correct input/output pairs are never presented, nor sub-optimal actions explicitly corrected [2]. Reinforcement learning systems consist of five main elements: a *model* of the environment, an *agent*, a *policy*, a *reward function*, and a *value function* [1]. The simulation we created acts as a *model*, representing the environment. It is used to simulate the behavior of objects and agents, such as the movement of the ghosts. The neural network forms the decision-making *agent*, interacting with the environment (or model). The agent employs a *policy* which defines how states are mapped to actions. The policy is considered to be the core of a reinforcement learning system, as it largely defines the behavior of the agent. A *reward function* maps each state into a numeric value representing the desirability of making a transition to that state. The goal of a reinforcement learning agent is to maximize the total reward it receives in the long run. Reward functions define what is good for the agent in the short run. On the other hand, a *value function* defines the expected return – expected cumulative future discounted reward – for each state. For example, a state might yield a low (immediate) reward but still have a high value since it offers consecutive states with high rewards. It is the goal of a reinforcement learning agent to seek states with the highest value, not the ones with the highest reward.

In the following, a state is referred to as s_t and an action as a_t, at a certain time t. The reward emitted at time t after action a_t is represented by the value r_t. An important assumption made in reinforcement learning systems, is that all relevant information for decision making is available in the present state. In other words, the traveled path or history is irrelevant in deciding the next action. This assumption is called the Markov property [20]. A system is said to have the Markov property if and only if specifying the probability of the next state and reward based on the complete history (Eq. 1):

$$Pr\{s_{t+1} = s', r_t = r | s_t, a_t, r_{t-1}, \ldots, r_0, s_0, a_0\} \quad (1)$$

Table I
List of in-game events and the corresponding rewards, as used in our experiments.

Event	Reward	Description
Win	+50	Ms. Pac-Man has eaten all pills and powerpills
Lose	-350	Ms. Pac-Man and a non-scared ghost have collided
Ghost	+20	Ms. Pac-Man and a scared ghost have collided
Pill	+12	Ms. Pac-Man ate a pill
Powerpill	+3	Ms. Pac-Man ate a powerpill
Step	-5	Ms. Pac-Man performed a move
Reverse	-6	Ms. Pac-Man reversed on her path

is equal to the probability of the next state and reward based only on the present information (Eq. 2):

$$Pr\{s_{t+1} = s', r_t = r \mid s_t, a_t, r_{t-1}\} \quad (2)$$

This assumption holds reasonably well in the game of Ms. Pac-Man. The ghosts are the only other agents in the game, and thereby the only source of uncertainty regarding future states. In hallways their behavior can be predicted based on their current moving direction – which can be part of the present state description. At intersections the behavior of a ghost is completely unpredictable.

The reward function in our project is fixed. Rewards are determined based on the original Ms. Pac-Man game. That is, static rewards are offered when a game event occurs, such as collecting a pill or colliding with a ghost. The reward function used in our research is listed in Table I. In the case of an action triggering multiple rewards, these rewards are added and then treated as if they were a single reward. The reward for reversing on a direction was added to discourage the agent from hanging around an empty corridor in the maze.

Exploration is required to make sure the network will not converge to a local optimum. We define the exploration rate as the chance that the agent would perform a random action, rather than executing the policy it has learned thus far. If by chance exploration is repeatedly triggered the agent will move in a consistent direction using the same random action. This means that the policy could always change directions in a corridor, but that the random exploration action would focus on one specific direction as long as Ms. Pac-Man stays in the same corridor.

A. Learning rule

The Q-value function is learned over time by the agent and is stored in a single action neural network or multiple action neural networks, as will be explained later. For this project, Q-learning [16], [17] was used as the reinforcement learning algorithm. It specifies the way in which immediate rewards should be used to learn the optimal value of a state. The learning rules of SARSA [21] and QV-learning [22] were also implemented in the framework, although they were not used in the final experiments. The general Q-learning rule is:

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

The new quality or Q-value of a state-action pair is updated using the immediate reward plus the value of the best next state-action pair. The constants α and γ refer to the learning rate and discount factor, respectively. The discount factor decides how distant rewards should be valued, when adjusting the Q-values. The learning rate influences how strongly the Q-values are altered after each action.

The actual altering of weights in the neural network(s) is done by an adaptation of the backpropagation algorithm [11]. The standard backpropagation algorithm requires a target output for a specific input. In reinforcement learning, the target output is calculated based on the reward, discount factor and the Q-value of the best next state-action pair. If the last action ended the game, Eq. 3 is used to compute the target output value:

$$Q^{\text{target}}(s_t, a_t) \leftarrow r_t \tag{3}$$

Otherwise Eq. 4 is used to compute the target output:

$$Q^{\text{target}}(s_t, a_t) \leftarrow r_t + \gamma \max_a Q(s_{t+1}, a) \tag{4}$$

Then the Q-value of the state-action pair is updated by using this target value to train the neural network.

B. Single action neural network

When using a single action network, at every time step each action or direction is considered in turn using the same neural network. Only the inputs relevant for that direction are offered to the neural network and the output activation is stored in memory. After a network run for each action has been completed, the output values are compared and the action associated with the highest activation is performed, unless an exploration step occurs. The single action neural network has the following structure:

Input layer, contains 7 input neurons.
Hidden layer, contains 50 hidden neurons.
Output layer, contains a single output neuron.

C. Multiple action neural networks

When using multiple action networks, at every time step each action or direction is associated with its own neural network. They will all be offered the entire state representation, instead of just the input related to the direction in question. The output values are compared and the action associated with the highest activation is performed, unless an exploration step occurs. The four action neural networks have the following structure:

Input layer, contains $2 + 4 \times 5 = 22$ input neurons.
Hidden layer, contains 50 hidden neurons.
Output layer, contains a single output neuron.

The number of hidden neurons was selected after some preliminary experiments, but we have also used different numbers of hidden neurons. When the number of hidden neurons was smaller than 11, the results were much worse. With 20 hidden neurons, the results were slightly worse and with more than 50 hidden neurons, the results became slightly better at the cost of more computation.

IV. State representation

The neural networks have no direct perception of the game or its objectives and states. They must instead rely on the value-free numeric data offered by input algorithms, and the reward function. Because of this, the nature of these must be carefully considered. The next few subsections outline the various smart input algorithms that were implemented. The first two algorithms produce a single feature value that is independent of the direction (left, right, up, down). They describe global characteristics of the current state. The remaining 5 feature extraction algorithms produce four input values: one associated with each direction. Multiple action networks will receive the entire state representation – a concatenation of the values for every direction. A single action network is only concerned with one direction at a time and will only receive the input values associated with that specific direction or a global characteristic. Because of the use of higher-order relative inputs, the agent requires a total of $2 + 4 * 5 = 22$ inputs to decide on an action in every game-state, independent of the maze, which also allows for policy transfer [23] as we show later when using single action networks.

A. Level progress

As previously stated, there are two primary goals that Ms. Pac-Man should be concerned with. All of the pills need to be collected to complete the level. At the same time, ghosts must be avoided so Ms. Pac-Man will stay in the game long enough to collect all of the pills. The artificial agent needs to find the right balance between these two tasks. Game situations that offer a certain degree of danger need to be avoided. However, engaging into a dangerous situation can be worthwhile if it allows Ms. Pac-Man to collect all remaining pills in the maze.

To allow the neural network to incorporate these objectives into its decisions, we constructed an input algorithm that represents the current progress in completing the level. Given:

$a =$ Total amount of pills
$b =$ Amount of pills remaining

The first input *PillsEatenInput* is computed as:
PillsEatenInput $= (a - b)/a$

B. Powerpill

When Ms. Pac-Man eats a powerpill, a sudden shift in behavior needs to occur. Instead of avoiding the ghosts, the agent can now actively pursue them. Whether it is worthwhile to pursue a scared ghost is influenced by how long the powerpill will remain active. This makes up the second input algorithm, which simply returns the percentage of the powerpill duration that is left. If there is no powerpill recently

consumed, this value is set to 0. Given:

a = Total duration of a powerpill
b = Time since powerpill was consumed

The second input *PowerPillInput* is computed as:
PowerPillInput $= (a - b)/a$

C. Pills

To allow the neural network to lead Ms. Pac-Man towards pills, it needs some sense of where these pills are. In every state there are 4 possible moving directions, namely: left, right, up and down. The input algorithm that offers information about the position of pills makes use of this fact, by finding the shortest path to the closest pill for each direction. The algorithm will use breadth-first search (BFS) to find these paths [24]. If this does not yield immediate results, it will switch over to the A*-algorithm to find the shortest path to the pills [25]. The result, in the form of four values, is then normalized to always be below 1. Given:

a = Maximum path length[1]
$b(c)$ = Shortest distance to a pill for a certain direction, c

The third input *PillInput(c)* for direction c is computed as:

PillInput$(c) = (a - b(c))/a$

PillInput is the first algorithm that relates its output to the various directions. This means that it adds one input to the single action network or four inputs to the multiple action networks.

D. Ghosts

Naturally, a game situation becomes more threatening as non-scared ghosts approach Ms. Pac-Man. This should be reflected by an increase in some input, signalling that it is dangerous to move in certain directions. Figure 2 shows an example of a game situation where Ms. Pac-Man is fleeing from two ghosts. From Ms. Pac-Man's perspective, does the presence of the ghost to the far left influence the degree of danger she is in? At this point in time there is no way for that ghost to collide with Ms. Pac-Man, as the other ghost will collide and end the game first. Figure 3 illustrates a game situation where a ghost has just reached an intersection directly next to Ms. Pac-Man. From that moment on, moving in the corresponding direction would be futile, as the ghost has closed of the hallway. We mark this point as the most dangerous possible.

[1]For example, in the first maze a path between two positions can never be longer than 42 steps, when the shortest route is taken. This is because the warp tunnels offer a quick way of traveling from one outer edge of the maze to another.

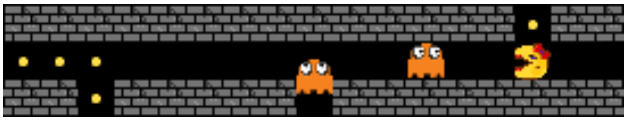

Figure 2. Example of a game situation where two ghosts are approaching Ms. Pac-Man from one side. As the closest ghost will reach Ms. Pac-Man first, only the distance to that ghost matters for the amount of danger that should be associated with the direction 'left'.

Figure 3. Example of a game situation where a ghost has just closed off any route belonging to the direction 'right'. If the ghost would be more to the right, the amount of danger would have been less. However, if the ghost would be even more to the left, the same danger holds for the direction 'right'.

The next input algorithm offers information on the danger associated with each action. First, it finds the shortest path between each non-scared ghost and each intersection that is directly next to Ms. Pac-Man. Then for each action, it selects the shortest distance between the corresponding intersection and the nearest non-scared ghost. Based on these distances, the resulting danger values are calculated. These are based on the expected time before Ms. Pac-Man will collide with a ghost when travelling towards that ghost, assuming the worst case scenario where the ghost will keep approaching Ms. Pac-Man. These values are then normalized to always be below 1 and above 0. Given:

a = Maximum path length[1]
v = Ghost speed constant[2] $= 0.8$

$b(c)$ = Distance between the nearest non-scared ghost and the nearest intersection for a certain direction, c
$d(c)$ = Distance to the nearest intersection in a certain direction, c

The fourth input *GhostInput(c)* is computed as:
GhostInput$(c) = (a + d(c) \times v - b(c))/a$

E. Scared ghosts

When Ms. Pac-Man collects a powerpill, all of the ghosts become afraid for a short period of time. If a scared ghost collides with Ms. Pac-Man, it will shortly disappear and then respawn in the center of the maze, as a non-scared ghost. This means that the game can contain both ghosts that need to be avoided and ghosts that can be pursued, at the same time. As a result we need separate input values for scared and non-scared ghosts. This will allow the neural network to differentiate between the two.

The input algorithm that considers the scared ghosts is very straight-forward. For each action, it returns the shortest

[2]The ghost speed is defined relatively to Ms. Pac-Man's speed.

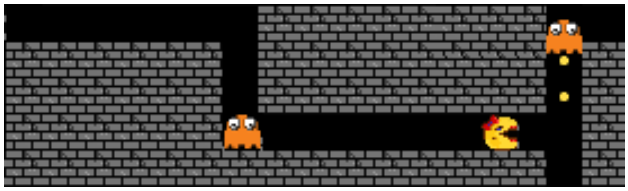

Figure 4. Example of a game situation where the only way for Ms. Pac-Man to escape involves approaching a ghost.

distance between Ms. Pac-Man and the nearest scared ghost. These values are then normalized to always be below 1. Given:

a = Maximum path length[1]
$b(c)$ = Shortest distance to a scared ghost for a certain direction, c

The fifth input *GhostAfraidInput(c)* is computed as:
$\text{GhostAfraidInput}(c) = (a - b(c))/a$

F. Entrapment

Figure 4 illustrates a situation where Ms. Pac-Man needs to approach a ghost in order to escape it. She should be able to avert the present danger by reaching the next intersection before any of the ghosts will. It shows how at times, one should accept a temporary increase in danger knowing it will lead to a heavy decrease in danger later on.

In addition to the general sense of danger that the GhostInput provides, the neural network needs information on which paths are safe for Ms. Pac-Man to take. We define these as *safe routes*: a path that can safely lead Ms. Pac-Man three intersections away from her current position. This means that Ms. Pac-Man should be able to reach every one of those possible intersections sooner than any ghost. In case there are no safe paths 3 intersections away in any direction, the number of safe paths is computed for 2 intersections away, etc.

First, the algorithm establishes a list of all intersections. For each intersection, it compares the time that Ms. Pac-Man needs to reach the intersection with the times that each of the ghosts need to reach the intersection. A list is created, holding all intersections that Ms. Pac-Man can reach sooner than any ghost. BFS is then used to find all safe routes. The algorithm returns for each action, the percentage of safe routes that do not start with the corresponding direction – the values will increase with danger. Given:

a = Total amount of safe routes
$b(c)$ = Amount of safe routes in a certain direction, c

The sixth input *EntrapmentInput(c)* is computed as:
$\text{EntrapmentInput}(c) = (a - b(c))/a$

G. Action

Figure 5 shows Ms. Pac-Man being approached by two ghosts in a symmetrical maze. What is the best direction to move in, when they all offer the same promise of danger and treasure? In this case one might argue the direction doesn't matter as long as the agent sticks with it. If subtle differences in input values were to cause Ms. Pac-Man to reverse on her path, she is bound to be closed in by both ghosts. The seventh and final input algorithm returns for each direction a binary value, which signals if Ms. Pac-Man is currently moving in that direction.

Figure 5. Example of a game situation where all possible actions are equally desirable.

V. Experiments and Results

To be able to compare the use of a single action network with using multiple action networks, we performed several experiments. In the first experiment six trials were conducted in which the agent was trained on 2 mazes for which we compare a single action network to multiple action networks. In the second experiment we transfered the learned policies to a different maze to see how well the learned policies can be transfered to a novel maze, unseen during training.

A. Experiment 1: Training on 2 mazes

In the first experiment, the rate of exploration was slowly brought down from 1.0 to 0.0 during the first 4500 games. The learning rate was set to 0.0005 during training (and 0.0 during testing). The discount factor was set to 0.95. Although the input algorithms and simulation support continuous data regarding agent positions, the network still has to run in discrete time steps. Therefore, we decided to let Ms. Pac-Man move half a square in every step. As in the original game, the speed of ghosts is defined relative to Ms. Pac-Man's speed, thus ghosts travel 0.4 squares every step.

As previously mentioned, three mazes from the original game were implemented in the simulation. In the first experiment the first two mazes were used. Afterwards performance was tested without learning on the mazes used during training. Pilot experiments showed the agent's performance stabilized after 7.000 games. We therefore decided to let the training phase last 10.000 games during the experiments, followed by a test phase of 5.000 games.

The performance is defined by the average percentage of level completion – the percentage of pills that were collected in a maze before colliding with a non-scared ghost. If all pills were collected (100% level completetion), we call it a successful game or a "win". The level completetion performance during training is plotted in figures 6 and 7. They show a steady increase over time for both the single and the multiple action neural networks. It appears the networks converge to a certain policy, as the performance stabilizes around 88% after approximately 8.000 games.

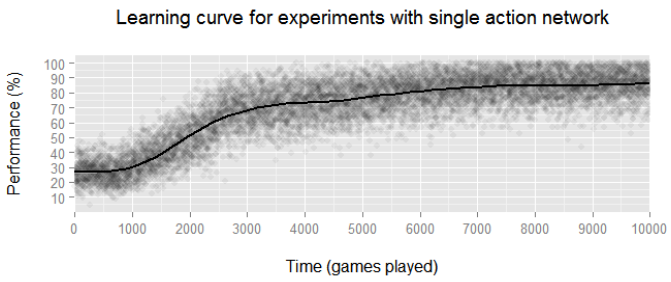

Figure 6. This graph shows how the performance (percentage of level completion) develops while training on the first two mazes during the first 10.000 games, with the use of a single network. Results were averaged over six independent trials.

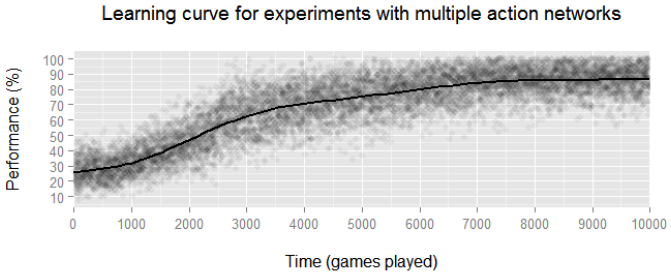

Figure 7. This graph shows how the performance develops while training on the first two mazes during the first 10.000 games, with the use of four action networks. Results were averaged over six independent trials.

Table II lists the results during the test trials. The table shows the average percentages of level completions as well as the winning percentages. It also shows the standard errors, based on 5000 games for the single runs, and based on the 6 runs for the final averages. We can easily observe that the single action network and multiple actions networks appear to perform equally well when tested on the first two mazes.

Table II
AVERAGE PERCENTAGE OF LEVEL COMPLETION (AND STANDARD ERROR) ALONG WITH PERCENTAGE OF SUCCESSFUL GAMES (OUT OF 5.000 TOTAL GAMES) DURING TESTING ON THE FIRST TWO MAZES. THE BOTTOM ROW CONTAINS RESULTS AVERAGED OVER THE VARIOUS TRIALS.

	Single network		**Action networks**	
Trial	Level completion	Wins	Level completion	Wins
1	90.1% (SE 0.2)	46.3%	93.8% (SE 0.2)	65.5%
2	89.2% (SE 0.2)	55.2%	88.7% (SE 0.2)	53.5%
3	86.9% (SE 0.3)	53.5%	86.7% (SE 0.2)	46.3%
4	85.7% (SE 0.3)	50.1%	84.8% (SE 0.2)	40.2%
5	85.6% (SE 0.3)	49.9%	81.7% (SE 0.3)	52.2%
6	82.1% (SE 0.4)	47.8%	77.5% (SE 0.4)	43.8%
Avg.	86.6% (SE 1.2)	50.5%	85.6% (SE 2.3)	50.3%

B. Experiment 2: Policy transfer to a different maze

Table III lists the results during the test trials on a different maze. In this case, the single action network outperforms multiple action networks when tested on the unknown maze. It shows that in 5 of the 6 runs the performance of the single action networks is better than the performance of the multiple action networks.

Table III
AVERAGE PERCENTAGE OF LEVEL COMPLETION (AND STANDARD ERROR) ALONG WITH PERCENTAGE OF SUCCESSFUL GAMES (OUT OF 5.000 TOTAL GAMES) DURING TESTING ON THE THIRD MAZE. THE BOTTOM ROW CONTAINS RESULTS AVERAGED OVER THE VARIOUS TRIALS.

	Single network		**Action networks**	
Trial	Level completion	Wins	Level completion	Wins
1	90.5% (SE 0.2)	46.1%	84.9% (SE 0.3)	50.0%
2	89.1% (SE 0.2)	49.0%	80.8% (SE 0.3)	34.7%
3	88.8% (SE 0.2)	47.6%	79.8% (SE 0.4)	42.3%
4	86.3% (SE 0.3)	54.6%	70.5% (SE 0.4)	31.3%
5	83.0% (SE 0.3)	47.2%	68.4% (SE 0.5)	29.1%
6	82.3% (SE 0.3)	47.3%	61.8% (SE 0.5)	28.0%
Avg.	86.7% (SE 1.4)	48.6%	74.4% (SE 3.6)	35.9%

Unpaired t-tests were performed to compare the data collected during the test phases. For testing on the first two mazes, the difference in performance between single and multiple action networks was not significant (p=0.695). For testing on the third maze, a significant difference in performance was found between single and multiple action networks (p=0.017).

VI. DISCUSSION

It is important to note that only 7 values were required to describe each direction. Among these inputs is the duration of the powerpill and the distance to scared ghosts, which only have a value above zero for small parts of the game. This means that most of the time, Ms. Pac-Man navigates using only 5 inputs. We performed additional experiments in which we left each time one of the inputs out. These experiments showed that the performance without the *Action* input was the worst. The performance without the *PillsEaten*, *PowerPill* and *GhostAfraid* inputs, were quite similar to when they were used. Finally, the results without the *Pill*, *Ghost*, and *Entrapment* inputs dropped with around 8-10% compared to using them.

The higher-order relative input implicitly incorporated the structure of the maze into its values through the path finding algorithms. In comparison, an absolute representation would need to include the position of each pill, non-scared ghost, and scared ghost. This would amount to thousands of binary input values for a regular sized maze.

The level of performance that was reached with the small amount of input shows that a neural network that was trained using reinforcement learning is able to produce highly competent playing behavior. This is confirmed by the data gathered during the test phases of the experiments. Single action networks showed to extend to unknown mazes very well, which is an important characteristic of competent playing behavior.

Table III shows how a single action network outperforms multiple action networks in an unknown maze. The use of a single action network imposes certain restrictions on the decision-making process. It was mentioned earlier that inputs will be weighted the same way for every direction and that the desirability of actions will be expressed on one scale. It also ensures that when considering a certain direction, only input relevant to that direction will be used. The data suggests that without these restrictions the policy of the networks will not generalize as well to unseen mazes. We conclude that

incorporating a single network offers benefits over the use of action networks.

When we visualize some of the levels that Ms. Pac-Man played, it is clear that an even higher performance could be achievable if the input algorithms were to be extended. Ms. Pac-Man seems to have learned how to avoid ghosts, but does not know what to do if both ghosts and pills are at the other end of the maze. It is a consequence of how the *PillInput* was set up: when there is a large distance between Ms. Pac-Man and the nearest pills, the associated input values will be very low. If the distance to the pills approaches the maximum path size, the influence of the presence of pills on the output values of the neural networks will approach zero. The *Entrapment* and *Ghost* inputs signal a maximum amount of danger when Ms. Pac-Man is fully closed in by ghosts. When all directions are equally unsafe the next move will be decided based on the other input values, such as the distance to the nearest pills. It would be more prudent for Ms. Pac-Man to keep equal distance between her and the ghosts surrounding her. The framework currently lacks an input algorithm that considers distances to all ghosts in the maze.

The results of this paper lead the way to a new approach to reinforcement learning. Reinforcement learning systems are often trained and tested on the same environment, but the evidence in this paper shows that not all networks are capable of forming a generalized policy. Using higher-order relative inputs and a single action network, a reinforcement learning system can be constructed that is able to perform well using very little input and is highly versatile. The learned policy generalizes to other game environments and is independent of maze dimensions and characteristics. Future research could apply the ideas presented in this paper on different reinforcement learning problems, specifically those with a small action space and a large state space – such as first-person shooters and racing games.

REFERENCES

[1] R. Sutton and A. Barto, *Reinforcement Learning: An Introduction*. Mit Press, 1998.

[2] M. Wiering and M. van Otterlo, *Reinforcement Learning: State of the Art*. Springer Verlag, 2012.

[3] G. Tesauro, "Temporal difference learning and TD-Gammon," *Communications of the ACM*, vol. 38, pp. 58–68, 1995.

[4] R. Crites and A. Barto, "Improving elevator performance using reinforcement learning," in *Advances in Neural Information Processing Systems 8*, D. Touretzky, M. Mozer, and M. Hasselmo, Eds., Cambridge MA, 1996, pp. 1017–1023.

[5] J. H. Schmidhuber, "Discovering solutions with low Kolmogorov complexity and high generalization capability," in *Machine Learning: Proceedings of the Twelfth International Conference*, A. Prieditis and S. Russell, Eds. Morgan Kaufmann Publishers, San Francisco, CA, 1995, pp. 488–496.

[6] J. Schmidhuber, "The speed prior: A new simplicity measure yielding near-optimal computable predictions," in *Proceedings of the 15th Annual Conference on Computational Learning Theory (COLT 2002), Lecture Notes in Artificial Intelligence*. Springer, 2002, pp. 216–228.

[7] M. Gallagher and M. Ledwich, "Evolving pac-man players: Can we learn from raw input?" in *IEEE Symposium on Computational Intelligence and Games (CIG'07)*, 2007, pp. 282–287.

[8] I. Szita and A. Lõrincz, "Learning to play using low-complexity rule-based policies: illustrations through ms. pac-man," *J. Artif. Int. Res.*, vol. 30, no. 1, pp. 659–684, Dec. 2007.

[9] J. De Bonet and C. Stauffer, "Learning to play pacman using incremental reinforcement learning," in *IEEE Symposium on Computational Intelligence and Games (CIG'08)*, 2006.

[10] D. Shepherd, "An agent that learns to play pacman," Bachelor's Thesis, Department of Information Technology and Electrical Engineering, University of Queensland, 2003.

[11] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," in *Parallel Distributed Processing*. MIT Press, 1986, vol. 1, pp. 318–362.

[12] I. Ghory, "Reinforcement learning in board games," Department of Computer Science, University of Bristol, Tech. Rep. CSTR-04-004, May 2004.

[13] M. McPartland and M. Gallagher, "Reinforcement learning in first person shooter games," *Computational Intelligence and AI in Games, IEEE*, no. 1, pp. 43 – 56, March 2011.

[14] A. Shantia, E. Begue, and M. Wiering, "Connectionist reinforcement learning for intelligent unit micro management in starcraft," in *IEEE/INNS International Joint Conference on Neural Networks*, 2011.

[15] T. P. Runarsson and S. M. Lucas, "Coevolution versus self-play temporal difference learning for acquiring position evaluation in small-board go," *Trans. Evol. Comp*, vol. 9, no. 6, pp. 628–640, Dec. 2005.

[16] C. J. C. H. Watkins, "Learning from delayed rewards," Ph.D. dissertation, King's College, Cambridge, England, 1989.

[17] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. 8, pp. 279–292, 1992.

[18] S. M. Lucas, "Evolving a neural network location evaluator to play ms. pac-man," in *Proceedings of the 2005 IEEE Symposium on Computational Intelligence and Games (CIG05)*, 2005.

[19] L.-J. Lin, "Reinforcement learning for robots using neural networks," Ph.D. dissertation, Carnegie Mellon University, Pittsburgh, January 1993.

[20] A. Markov, *Theory of algorithms*, ser. Works of the Mathematical Institute im. V.A. Steklov. Israel Program for Scientific Translations, 1971.

[21] R. S. Sutton, "Generalization in reinforcement learning: Successful examples using sparse coarse coding," in *Advances in Neural Information Processing Systems 8*, D. S. Touretzky, M. C. Mozer, and M. E. Hasselmo, Eds. MIT Press, Cambridge MA, 1996, pp. 1038–1045.

[22] M. Wiering, "QV(lambda)-learning: A new on-policy reinforcement learning algorithm," in *Proceedings of the 7th European Workshop on Reinforcement Learning*, D. Leone, Ed., 2005, pp. 29–30.

[23] M. Taylor, *Transfer in Reinforcement Learning Domains*. Springer, 2009.

[24] D. E. Knuth, *The art of computer programming, volume 1 (3rd ed.): fundamental algorithms*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1997.

[25] P. Hart, N. Nilsson, and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, Jul. 1968.