COMPUTER SCIENCE

Master thesis

Projective Simulation compared to reinforcement learning

By: Øystein Førsund Bjerland

Supervisor: Matthew Geoffrey Parker

June 1, 2015

Acknowledgements

I would like to thank my supervisor Matthew Geoffrey Parker for the assistance through the process of writing this thesis.

Abstract

This thesis explores the model of *projective simulation* (PS), a novel approach for an artificial intelligence (AI) agent. PS was introduced in [8], and further explored in [38, 40] by introducing new features and learning scenarios.

The model of PS learns by interacting with the environment it is situated in, and allows for simulating actions before real action is taken. The action selection is based on a random walk through the *episodic & compositional memory* (ECM), which is a network of clips that represent previous experienced percepts. The network takes percepts as inputs and returns actions. Through the rewards from the environment, the clip network will adjust itself dynamically such that the probability of doing the most favourable action (.i.e most rewarded) is increased in similar subsequent situations. With a feature called generalisation, new internal clips can be created dynamically such that the network will grow to a multilayer network, which improves the classification and grouping of percepts.

In this thesis the PS model will be tested on a large and complex task, learning to play the classic Mario platform game. Throughout the thesis the model will be compared to the typical reinforcement algorithms (RL) algorithms, Q-Learning and SARSA, by means of experimental simulations.

A framework for PS was built for this thesis, and games used in the previous papers that introduced PS were used to validate the correctness of the framework. Games are often used as a benchmark for learning agents, a reason is that the rules of the experiment are already defined and the evaluation can easily be compared to human performance. The games that will be used in this thesis are: The Blocking game, Mountain Car, Pole Balancing and, finally, Mario. The results show that the PS model is competitive to RL for complex tasks, and that the evolving network will improve the performance.

A quantum version of the PS model has recently been proven to realise a quadratic speed-up compared to the classical version, and this was one of the primary reasons for the introduction of the PS model [8]. This quadratic speed-up is very promising as training AI is computationally heavy and requires a large state space. This thesis will, however, consider only the classical version of the PS model.

Contents

List of Figures

List of Tables

Listings

1 Introduction

The previous papers on projective simulation (PS) [8, 38, 40] demonstrated the features and performance of the model by means of simple experiments. The authors showed that the performance of PS was comparable to RL algorithms. This thesis will start by verifying these previous results, by implementing the framework in code [67, 66], and by running the same experiments to confirm the correctness.

The goal of this thesis is to test how the newly developed idea of projective simulation compares to well established RL algorithms. One of the motivations for PS is its potentially significant enhancements using quantum mechanics, which can give a large speed-up and the possibility to be applied on much larger tasks than possible with RL.

The PS model can be used as a reinforcement learning algorithm, as both are independent agents that can be situated in an environment, perceive inputs and return actions, and learn by trial and error through the feedback given. This is also where most of the similarity ends. Whilst RL stores information in matrices, PS starts with an empty network that will evolve as new percepts are seen.

The general goal of artificial intelligence and machine learning is to create a robot or program that can replicate human intelligence. Machine learning is useful on problems that are difficult to formalize in an algorithm (e.g image recognition, robot walking, playing Mario). Reinforcement learning is one area of machine learning, which is inspired by stimulus-response theory from behavioural learning. Experiments on animals show that they can learn an arbitrary policy to obtain rewards (and avoid punishments). Reinforcement learning works in a similar way, with trial and error and corresponding punishments. Calling an artificial agent *intelligent* inevitably leads to much justified debate as to the meaning of the word *intelligent* but, in this thesis, intelligence will be defined as the capability to reason, plan, solve problems and learn from experience [19].

The structure of this thesis is as follows: First some general description of reinforcement learning (RL) is given, then the PS model will be described, and similarities and differences between RL and PS will be discussed. The models will then be tested via different experiments. The first experiment is the Blocking Game, which is a simple toy problem for showing the features of the PS model. After that, some standard RL experiments will be used to check how PS compares to RL. The last game will be a real game, the classic Mario platform. The thesis will end with a discussion of the results, and on the possible future of RL and PS, and the eventual possibility of creating real AI.

The thesis will give a short introduction to RL, but a more detailed description can be found here [56], and more details on PS can be found here [8].

The source code will not be included in the thesis, but the whole framework is available here [67].

1.1 Machine learning

Machine learning (ML) is a field in computer science which is often used when writing an algorithm for a problem is infeasible. Instead the goal is to make the algorithms solve a problem without being explicitly programmed [54]. Typical problems in ML can be image recognition, pattern recognition and making a robot walk (e.g what should the angle be on each foot). These tasks are difficult to program explicitly. Often the solution to the problem is unknown to the creator, e.g what is is the fastest way a robot with 10 legs can walk?

Machine learning tasks can be divided into 3 categories depending on the feedback the agent will receive [49, p. 695]:

Supervised learning The agent is given a set of training samples with the correct answers. The agent will train on the training samples until it reaches a success rate threshold. The goal is to minimise the expected error.

Unsupervised Learning Often called clustering. No feedback is given to the agent. The goal is to recognize patterns in the input and group them by similarities.

Reinforcement Learning The agent interacts with its environment. The environment consists of states and actions. In state s_t, the agent takes action a_t, and ends up in state s_{t+1} , and receives reward r from the environment.

When an agent makes a decision in supervised learning it will be given the correct answer, which it then can adjust itself such that the chance of doing the correct answer is increased. In neural networks, backpropagation is typically used for this. In reinforcement learning, the agent will not be given the correct answer, instead it will be given a scalar reward depending on how good the action was. The task of solving a RL problem is therefore harder, since the best action is still unknown, both to the agent and the creator. RL is used when the correct behaviour is not known in advance, but where the positive and negative effects of an action in the environment can be calculated to give some kind of positive and negative rewards, respectively. The task of RL is to find a good strategy to find the best actions to realise in each state.

2 Reinforcement learning

A defining quote on reinforcement learning in a nutshell: “Imagine playing a new game whose rules you don’t know; after a hundred hundred or so moves, your opponent announces, "You lose".” - Russell and Norvig [49]

A useful framework for solving RL tasks is a Markov Decision Process (MDP), which is a formal model for decision making problems. Most RL tasks model the environment using MDPs.

2.1 Markov decision process and Policies

The model contains:

- set of world states S
- set of actions A
- A real valued reward function $R_a(s, s')$
- A description T of each action’s effects in each state.

At each time step t, the agent observes the state of the environment $s_t \in S$, and chooses an action $a_t \in A$. One time step later, the agent receives a reward r_{t+1} and the environment’s next state $s_{t+1} \in S$.

A criterion for a MDP is to satisfy the Markov Property: "The future is independent of the past given the present"[49, p. 604]. In other words, the response at time $t+1$ only depends on the state s_t and action a_t at time t. An MDP is an extension of a Markov chain, the difference being the addition of actions and rewards [6].

$$Pr(s_{t+1} = s^{'}, r_{t+1} = r \mid s_t, a_t) \tag{1}$$

With just the previous state and the last action, the whole environment is fully described. States and actions executed before $(s_{t-1}, a_{t-1}, s_{t-2}, a_{t-2}, ..., s_0, a_0)$ have no effect.

This also means that the current state of the system is all the information that is needed to decide what action to take. There is no dependency on previous states, actions or rewards. [63]. This is the main advantage of describing a RL problem as a MDP with Markov property.

The rewards are also independent of the history:

$$R = E(r_{t+1} \mid s_t = s,\ a_t = a,\ s_{t+1} = s^{'}) \tag{2}$$

where R is the expected reward after being in state s_t, doing action a and ending in state $s^{'}$ [41].

The goal of the agent is to find an optimal policy $\pi(s)$. A policy π is a mapping from states S to actions A, where $\pi_t(s,a)$ is the probability of doing action $(a_t = a)$ in state $(s_t = s)$. The optimal policy is the one that maximizes the expected rewards over time. The value for taking an action a in a state s using policy π is denoted as $Q^{\pi}(s,a)$.

Any task can be modelled as a MDP by making the state-space detailed enough by encapsulating the relevant history in the states. This property is very useful since it is provable that an optimal solution to an MDP is achievable by using a RL algorithm [47]. The reason is that each individual state contains all information that is needed to decide what to do next. At any time, the same action will always be the optimal action each time it is visited.

2.2 Reinforcement algorithms

There are multiple classes of methods for solving the RL problem: dynamic programming, Monte Carlo and temporal-difference (TD) learning. Temporal difference methods are the most adaptable, requiring less knowledge of the dynamics, and can be used for almost any problem [56].

The aim of the agent is to maximise the reward it receives over a period. Reinforcement learning can be used to find the optimal MDP when probabilities and rewards are unknown (wiki).

Q-Learning and SARSA are some popular and standard algorithms for learning the policy. If the world is a MDP the policy will converge to the optimal solution. Both Q-learning and Sarsa compute $Q(s, a)$ based on the interaction with the environment.

S_t Current state

A_t The action the agent chooses

R_t The reward R the agent gets for choosing this action

S_{t+1} The new state the agent gets to after action A_t

A_{t+1} The action the agent will choose in the new state.

2.2.1 Q-Learning

Q-Learning (Watkins's (1989)) is a simple one-step form:

$$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha * (R + \gamma * maxQ_t(s_{t+1}, a_t) - Q_t(s_t, a_t) \tag{3}$$

γ is the discount factor $0 \leq \gamma \leq 1$. A higher value of $\gamma \approx 1$ means the future rewards become more important, and the agent will strive for longer term rewards, whilst lower values lead to a preference for shorter term rewards.

α is the learning rate, where $0 \leq \alpha \leq 1$, and determines how much new information will override old information. A high value $\gamma \approx 1$ will only consider the most recent information, while a low value will emphasise old information more. The learning rate can be changed over time, and potentially depend on each state-action. $\alpha_t(s, a)$, which determines the learning rate for a state-action pair at time t. It can be useful to start with a high learning rate, and then lower it over time to save the learned information.

2.2.2 SARSA

SARSA (State-Action-Reward-State-Action) is similar to Q-learning, but uses the value of the action actually performed to determine its update, instead of the maximum available action. Using the policy in the learning is called *on-policy*. Q-learning is *off-policy* since it does not follow the policy, but instead take the highest valued action.

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \tag{4}$$

The whole difference is this part: Q-Learning $maxQ_t(s_{t+1}, a_t)$, SARSA $Q(s_{t+1}, a_{t+1})$, where a_{t+1} is chosen using the policy (e.g ϵ-greedy). Q-Learning will take into account the best possible action to take in the next state (s_{t+1}), while SARSA will use the action actually performed to determine its update. Q-Learning will be more optimistic, and value each state by how good it would be if it made all the best moves, whilst SARSA will value it by the moves it expects to take.

2.2.3 Online vs Offline

RL algorithms may be categorized as *online* and *offline* [36] depending on whether the feedback is continuous or delayed. *Online* RL will receive rewards after each time step, and the Q's will be updated after each action. In *offline* learning the rewards are delayed, such that the update of Q must be delayed to. As an example, in a Maze task the agent will only be given a positive reward once the goal is reached.

2.2.4 Eligibility Traces - Q(λ) and SARSA(λ)

Q-learning and SARSA can be combined with eligibility traces, then called Q(λ) and SARSA(λ). [56]. Eligibility Traces are a temporary record of previous events, states and action done later. When an error or reward is perceived, the eligibility states and action are assigned the positive or negative reward. This is a mechanism for temporal credit assignment. For instance, in a game where there are states that are unrecoverable, or always end in a loss, then it is the previous actions fault, and these actions should be blamed for the outcome as well.

Eligibility Traces can be implemented in two ways, *Accumulating Traces* and *Replacing Traces*. Accumulating traces increment the trace for the visited state-action pair by 1. Replacing traces sets the eligibility trace of a visited state to 1. In general, the traces will keep a record of state-actions pairs that are recently used, and to what degree they are eligible for learning updates.

2.2.5 Action selection

When the RL agent is given a state s, the agent will chose an appropriate action. The table $Q(s, a_i)$ for state s stores all the estimated action values for choosing each action. The simplest rule is to select the action with the highest estimated action value $maxQ(s, a_i)$. The problem with this approach is that there is no exploring, i.e. there might be a much better action than the one that is currently estimated as best.

ϵ-Greedy

A common method is to behave greedily most of the time, but explore once in a while by selecting a random action. This method is often called ϵ-Greedy, by using an ϵ parameter as the probability of selecting a random action. The epsilon ($0 \leq \epsilon \leq 1$) value is usually decreased over time, to make it converge to one solution. The epsilon value can be global and shared for all states, or each state or state-action pair can have its own ϵ value. The latter approach will ensure that for each state there is some chance of exploration, even if the state is discovered late in the learning process.

Any action selection method can be used in RL, another typical choice other than ϵ-Greedy is SoftMax 11.

2.2.6 Function Approximation

The state-action pairs $Q(s, a)$ are usually stored as a matrix. It is useful to be able to describe each state as a unique natural number or vector for easy storage. A problem is that many tasks involve multiple floating variables, which could lead to an infinite number of states. Function approximation is therefore often used, since it can map decimals to integers and also decrease the state space. To represent each state as an integer, a function like below can be used:

$$sInd = (int)\lceil \frac{(x - xMin) * nRange}{(xMax - xMin)} \rceil \tag{5}$$

where *sInd* is a normalized value for x,
xMin is the minimum value for x,
xMax is the maximum value for x,
nRang is the new integer range for the value.
If there are multiple variables, a unique state for all can be built by using a polynomial function:

$$uniqueState = (int)(sInd_0 + sInd_1 * nRange + sInd_2 * nRange^2 + ... + sInd_n * nRange^{n-1}) \tag{6}$$

The *uniqueState* will describe the whole state of the world, but depending on the new range chosen, it will lose some information by approximating.

3 Projective Simulation

Projective simulation (PS) is a recently developed artificial intelligence model. The model was proposed in *Projective simulation for artificial intelligence* by Briegel & Cuevas (2012) [8], and further explored in [40, 38, 37].

Projective Simulation allows the agent to simulate itself into future situations, based on previous situations. PS learns by trial and error (with rewards), and can be used as a reinforcement learning algorithm. The most significant difference between PS and other reinforcement learning algorithms, such as Q-learning and SARSA, is that PS consists of a network of clips. This allows for more complex features and, in that way, PS resembles a neural network structure.

The model also allows for creative behaviour, where creativity is here defined as the capability to deal with unprecedented situations, and to relate a given situation to other conceivable situations [8, 9].

The PS model starts without any knowledge of the world in which it is situated. The only information the model needs to know is what basic actions it can perform. New actions can then be learned by composition from the basis actions 3.8.2. The input to the model, called percepts, will dynamically be added to the network when they are perceived.

Perceptual input is given to the episodic compositional memory (ECM), which then creates or excites a corresponding percept-clip. The excited clip will start a "random walk" through the network, and the clip will excite an adjacent clip with probability corresponding to the strength of the edges. The random walk ends when it reaches an action-clip, which corresponds to a real action. The real action will then be executed in the environment.

After the agent performs an action, it receives a reward from the environment. This is how the agent learns, by using the reward to alter the recently used edges between clips in the network. If the action leads to a positive reward, then the edges used in the walk are strengthened. The next time a similar percept is excited, the agent may start a different random walk, and possibly end up performing a different action.

3.1 PS concepts

The key concepts in the PS model are listed below. A more detailed description can be found in [8] and [38] , [37].

The ECM a weighted network of nodes called clips.

Clips represents fragments of episodic experiences. A clip represents a percept or action. Formally a percept-clip is a vector $s = (s_1, s_2, ..., s_K) \in S$, where each s_i is a natural number $s_i \in \{1, 2, ..., |S_i|\}$, $|S| = |S_1| * |S_2| * ... * |S_K|$ is the number of percepts. In the PS model each vector component is called a dimension, and can contain any interval. The same definition is used for action-clips, while the dimension may be different.

Percepts The input to the ECM, which will trigger a random walk from the corresponding percept-clip. Similar to states in RL, which in practice means the input to the agent.

Actions The output, i.e. when a random walk ends in an action-clip, then the corresponding action is executed in the environment.

Edge connects two clips. The edge is directed with dynamic weight $h^t(c_i, c_j)$. Edges are created from every percept clip to every action clip. The edge weights are initialised to $h^0(c_i, c_j) = 1$.

3.2 Episodic Compositional Memory

PS uses a memory system called *episodic compositional memory* (ECM) which provides the platform for simulating future actions before real action is taken. The ECM is a directed weighted network of clips, where the clips represent prior (or some combination 3.8.3) experiences.

The PS scheme distinguishes between real percepts (input) and action (output) using their internal representation. A percept triggers the corresponding percept-clip, and an action-clip or actuator-clip triggers a real action. In most cases, the difference is not important. In most cases a percept always triggers the percept-clip, and the same goes for actions.

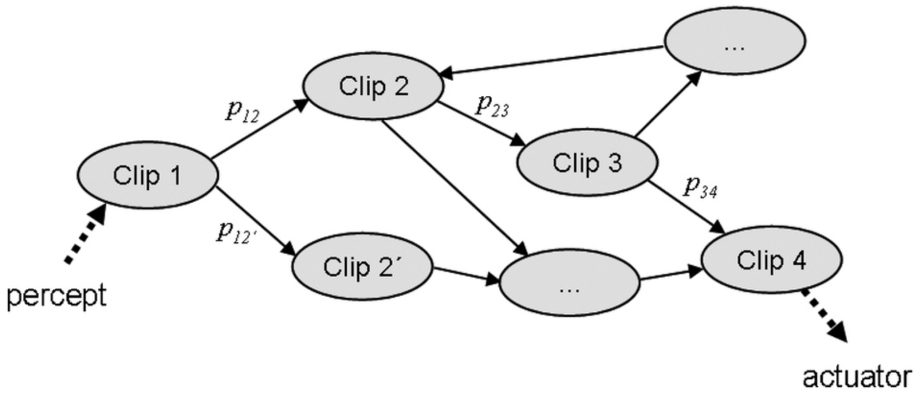


Figure 1: ECM network model [8]

3.3 Learning

Learning in the PS model is achieved through dynamic changes in the network, and can be accomplished in 3 different ways:

1. Adjusting the strength $h_t(c_i, c_j)$ of the edges between clips (Bayesian updating). This leads to a change in the hopping probability.
2. Perceiving new percepts will create a new clip in the network, which may change the probability of other percepts too 3.8.
3. New clips through composition. This is the creation of new fictional clips from a combination of existing clips. A method for achieving this is through generalisation 3.8.3.

3.3.1 Edge weight update

After the model performs an action it will receive a reward $\lambda \geq 0$. If an edge was used (activated) in the random walk leading to that reward, the edges will be updated according to the reward. Activated edges are updated as follows:

$$h^{t+1}(c_i, c_j) = h^t(c_i, c_j) - \gamma(h^t(c_i, c_j) - 1) + \lambda \tag{7}$$

All other edges are damped:

$$h^{t+1}(c_i, c_j) = h^t(c_i, c_j) - \gamma(h^t(c_i, c_j) - 1) \tag{8}$$

The update rule ensures that the weights never go lower than $h = 1$. The damping parameter γ determines how much the agent should forget. The deducted factor for every step is $\gamma(h^t(c_i, c_j) - 1)$.

$\gamma \approx 1$ will make the agent forget the past and value recent experiences more, while $\gamma \approx 0$ ensures the agent never forgets the past. However, the price paid is that the agent now adapts more slowly to changes than a forgetful agent, since it will keep doing the action it got rewarded for in the past. But the agent performs better than a forgetful agent in a static (non-changing) world.

In most cases, γ should be close to zero unless the rules of the environment change often. Experiments show that having some damping can be useful even in a static environment, as it can help to escape a local optimum by allowing for more exploration.

3.4 Parameters

PS in its simplest form only has a few parameters:

Parameter	Range	Field	Default	Explanation
Damping	$0 \leq \gamma \leq 1$	R	0 (or $1/10S$)	Damping the weights of the edges. How much to forget. How much new vs old is weighted.
Reflection	$1 \leq R$	N	1	Short time memory. If the chosen action gave a negative effect previously, try to find an action with a positive effect.
Glow damping	$0 \leq \eta \leq 1$	R	1	For afterglow mechanism. How much credit (reward, strengthening) should edges used previously get from a new reward.
Reward	$0 < \lambda$	R	1	The reward given for a successful action

Table 1: Parameters used in PS

3.5 Hopping probability

The hopping probability is the probability that a clip will excite another clip by passing along the edge connecting the two clips. Edges between clips have a weight $h^{(t)}(c_i, c_j)$., which changes over time. The weights are initialized to $h^0(c_i, c_j) = 1$, which ensure that in the beginning the probability of hopping to another clip is uniform.

Different hopping probability functions can be used for selecting the next clip in a walk through the network. In general the stronger edges are more likely to be chosen, but the highest rewarded edge should not always be chosen because some exploration is needed.

Always selecting the most rewarded edge will usually lead to a non-optimal solution, as the agent will keep traversing that edge without exploring for a better solution. As seen in the RL chapter, the typical choice was ϵ-Greedy (2.2.5) or SoftMax. In PS the typical choice is SoftMax or a simpler probability function (referred to as the standard function from now on).

3.5.1 Standard function

The probability of using edge (c_i, c_j) is the weight of that edge divided by the sum of all the directed edges from clip c_i to adjacent clips (c_i, c_k)

$$p^{(t)}(c_i, c_j) = \frac{h^{(t)}(c_i, c_j)}{\sum_k h^{(t)}(c_i, c_k)} \tag{9}$$

The same function is used for the hopping probability of directed sequences, which is the probability of performing an action after percept s is excited:

$$p^{(t)}(a|s) = \frac{h^{(t)}(s, a)}{\sum_{a \in A} h^{(t)}(s, a)} \tag{10}$$

3.5.2 SoftMax function

The SoftMax function is an alternative to the standard function. It favours the stronger edges more than the standard function by raising the weights to the exponent.

$$p^{(t)}(c_i, c_j) = \frac{e^{h^{(t)}(c_i, c_j)}}{\sum_k e^{h^{(t)}(c_i, c_k)}} \tag{11}$$

SoftMax will select the stronger edges with a higher probability than the standard function, such that in general it will perform better. But it will be slower at adapting to changes. SoftMax will follow the policy more strictly, whilst the standard function explores other solutions more.

3.5.3 ϵGreedy

ϵ-Greedy is the typical choice in RL (see 2.2.5), and it can be used in PS also. It follows the policy more strictly than SoftMax.

```
if random.next() < epsilon
   explore() // random action/clip
else
   chooseMostRewardedClip()
```

The PS model initializes all weights to 1, and they can never go lower than 1. Following the approach of reinforcement algorithms, another approach is to initialize the weights randomly and remove the restriction on the lower bound of 1.

A comparison of PS with different hopping probability functions can be found in 4.2.2.

3.6 Emotion and Reflection

This is a feature where the agent can reflect multiple times if the action it found through a random walk should be executed. The agent has the possibility to simulate multiple times until a promising action is found. The simulation is internal and entirely based on experience (not executed in the environment). The reflection is implemented through tags on edges called emotions, which can be seen as a short time memory. An edge between clips, e.g perception clip to action clip (s, a), can be flagged depending on previous experience doing that action given the same perception. If the edge is flagged with a negative ($e(s,a) =$☹) emotion, the agent can start a new random walk until it finds a positive ($e(s,a) =$☺) action.

Reflection is bounded by a maximum reflection parameter $R >= 1$. $R = 1$ means no reflection, the selected action according to the hopping probability will be executed (even if it is flagged ☹). If reflection is enabled, ($R = n$, $n > 1$) the agent can start n random walks until it finds a positive (☺) edge. If the agent gets stuck in an internal loop looking for a ☺ edge, the last walk will be executed when the reflection time is exceeded. All edges are initialized as positive in the beginning. When a reward is given, the emotion tag for the recently used edge(s) is set to ☹ if the action was bad or ☺ if the action was good.

An agent that is "feeling" positive will quickly think ahead and act, whilst an agent "feeling" negative will spend time simulating the best potential action. Reflection will take control over the action-selection policy in a "negative feeling" agent.

From the standard definition of emotion in PS, the negative edges are not removed or ignored for the next random walk. Another approach is to remove the negative flagged edges (from [46], see 20). Removing negative edges will make the agent learn faster, and also remove the possibility for a long internal loop looking for a positive action, if the best action has a negative flag. Moreover it will lead to a disregard of the hopping probability, by letting emotions matter more. In this thesis negative edges will be kept.

3.7 Afterglow

In most experiments a reward can only be given after a sequence of steps are done. In the Mountain Car problem 4.4, more than 100 correct actions are needed to win the game and the a the agent only receives a positive reward once it has won the game. This means the agent will have to do a sequence of correct actions before the actual reward will be given. A agent will therefore need a mechanism to reward such situations correctly. In RL eligibility traces (2.2.4)) was used in these situations. In PS a similar scheme is used, called *afterglow*. Both try to solve the problem of distributing blame and rewards to the correct places.

The PS model provides two methods for realizing afterglow : edge-glow and clip-glow. With the difference being if the edge or the clip is glowing. The glow value refers to how recently it was used, and how much reward it should receive for a future reward.

3.7.1 Edge glow

Each edge has a parameter g $(0 \leq g \leq 1)$ called edge-glow. When an edge is used, this parameter is excited ($g \leftarrow 1$). Every next steps the value is damped towards zero.

$$g^{t+1} = g^t - \eta * g^t \tag{12}$$

where η $(0 \leq \eta \leq 1)$ is the glow-damping parameter. The edge glow determines how long an edge will be (partial) excited, and therefore how much reward it will receive for future rewards.

By using afterglow, a new update rule can be introduced. The glow now determines how recently it was used, and it can be combined with the update rule 7 to get a equal update rule for all edges in the network:

$$h^{t+1}(c_i, c_j) = h^t(c_i, c_j) - \gamma(h^t(c_i, c_j) - 1) + \lambda * g^t(c_i, c_j) \tag{13}$$

The difference now is that edges that were recently used, $g(c_i, c_j) > 0$ will receive parts of the reward also. Afterglow can be turned off by setting $\eta = 1$, in which case only clips from the previous walk will get rewarded. The lower η, the more reward will be given to previous clips. The most recently excited edges (according to g) will receive a higher portion of reward.

3.7.2 Clip glow

Clip glow is another idea for afterglow introduced in [38]. Instead of making the edges glow, each clip has a glow value. It works is a similar way, for excited clips $g(c_i) = 1$ and for not excited clips $g(c_i) = 0$.

$$h^{t+1}(c_i, c_j) = h^t(c_i, c_j) - \gamma(h^t(c_i, c_j) - 1) + \lambda * g^t(c_i) g^t(c_j) \tag{14}$$

The two models are equal in a simple PS model. But if the association feature is on, the results will be different. See 3.8.1.

3.8 Associative Memory

More advanced features can be added to the model. In this section the model will be allowed to make multiple hops through the network and new fictitious clips may be created.

3.8.1 Advanced Random Walk - Clip to Clip

Before, we have only considered direct sequences, where the activation of a percept clip immediately activates an action clip. However, the PS model allows for walks between clips in the network. Activation of a percept clip may be followed by a sequence of jumps to other, intermediate clips, before it ends up in an action clip. The advantage of this is that the intermediate clips might have a strong links to the correct action, and in a sense the network will automatically categorize the percepts.

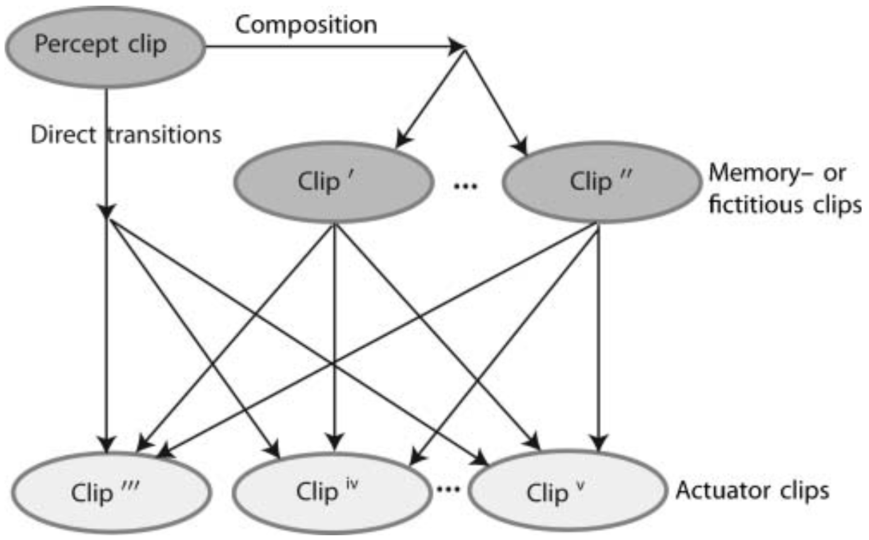


Figure 2: ECM-Composition with deliberation length $D = 1$. Illustration from [8].

1. Every percept s triggers a sequence of clips, where $S = (s, s_1, s_2, ..., s_D, a)$. D is the deliberation length. The case $D = 0$ is the direct sequence $S = (s, a)$. When $D > 0$, the excited percept clip may excite some other intermediate clip.

2. If (s, a) was rewarded recently (☺), the actuator clip a is directly translated into real action a. ⓐ Otherwise, a new (random) sequence $S = (s, s_1, s_2, ..., s_D, a)$ is generated,

starting with the same percept clips s but ending possibly with a different actuator clip a. The maximum number of generated sequences is limited by the reflection time R. The probability for a transition from clip c to clip c_i use the standard function 3.5.1.

3. Once the simulation is over, the action $a^{(n)}$ from the action-clip is executed. If the action $a^{(n)}$ is rewarded, then the weight of all transitions that occurred in the preceding simulation will be enhanced: If it reward is negative, this step is skipped and it goes straight to damping (next step (b)).

   (a) The edges used in the sequence $S = (s, s_1, s_2, ..., s_D, a)$ increase by the amount:

   $$h(s^i, s^{i+1}) += K \qquad i = 1, ..., D-1 \tag{15}$$

   $$h(s, s^1) += h(s^D, a) += 1 \tag{16}$$

   (b) The parameter K define the growth rate of associative connections relative to direct connections. The direct transition $s->a$ will also be increased by unity $h(s, a) += 1$

   (c) The damping works similar as before, except the intermediate clips will damp towards K. All edges will be damped, even non used, at every time step:

   $$h(c, c^{'}) = h(c, c^{'}) - \gamma\big(h(c, c^{'}) - h_0(c, c^{'})\big) \tag{17}$$

   which describes damping towards a stationary value

   $$h_0(c, c^{'}) = \begin{cases} 1, & \text{if } c \in S \text{ and } c^{'} \in A \\ K, & \text{if } c \in S \text{ and } c^{'} \in S \end{cases} \tag{18}$$

4. The standard way of initializing edge weights in PS to to use the unity $h = 1$. By using other weights for K, the clips could prefer hopping to other clips or to go straight to a action-clip.

Difference between using edge and clip glow

The choice of between clip and edge glow will lead to different rewarding of the edges. As an example, a sequence of 3 edges $e(c_1, c_2)$, $e(c_2, c_3)$,$e(c_2, c_4)$ was used. With edge glow only the edges used used and the direct edge $e(c_1, c_4)$ would be rewarded. But with clip glow, all the edges between the exited clips will be rewarded, in this case the edge $e(c_2, c_4)$ since both c_1 and c_4 is glowing.

3.8.2 Action clip composition

The PS model is free to create new actions. Two highly rewarded action clips in the PS model can create a new composite action clip. It is useful to give the agent some kind of limit on the number of actions. If not, the agent could just keep making up actions that do not exist in the real world. These actions would of course not do anything positive, and so the agent would not be encouraged to do them again. But there is no need for an agent that creates new random actions every time it is asked to perform a task. In practice, the agent can either be given a base set of actions (see Mario example (5.4) or given some limit on the possible actions (e.g using natural numbers).

To understand how composite actions might be created, consider the following example. In a 2D game, the agent might have access to the actions $\{\leftarrow, \rightarrow, \uparrow, \downarrow\}$. If it keeps getting rewards for doing both $\{\leftarrow, \uparrow\}$ for the same percept, a new action $\{\nwarrow\}$ could be created.

The requirements for the creation of a new composite action clip from two constituent actions are as follows:

1. Both action clips are sufficiently rewarded for the same percept.
2. The actions are similar, e.g action vectors c_a and c_b only differ in two components.
3. The newly composed action clip does not exist already.

The newly created action clip is connected to all percepts with the initial edge weight $h = 1$, except for the percept that spawned the new clip, which gets an edge weight begin the sum of the two edges from the percept to the actions, $h(c_p, c_{new}) = h(c_p, c_a) + h(c_p, c_b)$. Thus, subsequently, the newly created action-clip has a high chance of being used.

3.8.3 Generalisation

In [39] they introduce a method for evolving the ECM dynamical by creating fictitious clips. It follows a similar idea as done with action composition, where similar clips can create a new clips which resembles them. The idea for generalisation is inspired by the wildcard notation from *learning classifier systems* (LCS). The goal is to an autonomous abstraction of the percepts without any prior knowledge given to the model.

As a previously, a percept is defined as a vector $s = (s_1, s_2, ..., s_K)$ where each $s_i \in |S_i|$. For generalisation, each component will be treated as a category. The idea is that clips with are similar, share multiple components (categories), can be processed the same way.

When a new perception is seen by the ECM, it is compared pairwise to all existing clips. For each pair of clips whose categories carry $1 \leq l \leq K$ different values, a new wildcard clip is created (if it does not already exist). The new clip will have the l different components marked as # wildcard symbol, and will be placed in the lth layer in the network (where the zero-layer is the percepts and the $K+1$ layer is the action-clip layer).

More details example will be found in the experiments later 4.2.4 and 5.5.

3.9 Source code

All the code used in the paper is open source and can be found in [67]. It contains a framework for the PS model, including all the features described above, as well as all the experiments done. The source code for PS with the Mario game is in another repository [66], due to the large library required for the Mario game. Java 8 JDK [45] are required to run the code.

Speed short-cuts for the PS model

The implementation of the afterglow feature can be done in two ways, either by looping over every edge and clip at each action and update or by having a list of the current activated clips/edges and only update them. The last idea is much faster, but requires some additional conditions which goes slightly away from the standard PS model. Afterglow damping decreases the glow according to $g^{t+1} = g^t * \eta$, which never reaches zero. To use the list method, clip/edges with damping lower than $MinGlow = 0.00001$ is removed. The effect will not be of any importance, since the partial reward the edges/clips with $g \leq 0.00001$ would be microscopic.

The list structure also requires damping to be off, since damping requires update to all edges, or another tweak which goes further away from the model can be used. Each edge can have a internal counter of when it was last used, once it is used it will receive the damping from all the steps it did not participate in. This change is more drastic, since it effect the performance. The fear for this approach is that a high rewarded edge will not be damped when it should be. But since it is high rewarded, it will be chosen and damped accordingly.

4 Games

To examine the performance of machine learning, benchmarks are needed. For reinforcement learning, the choice is often toy problems like Mountain car and Pole Balancing.

One reason humans find games fun to play is due to learning [34]. Games involve learning mechanisms and strategies, and this makes it ideal for a computer learning agent. Games can also be seen as a simple representation of the real world. The input to the agent is what it sees, and it chooses actions which will change its world. Most games have a score, which can easily be used as a reward mechanism for RL (such rewards are more difficult to specify in the real world). Games are also visual which makes it easy to evaluate the performance, even for people without any knowledge of machine learning. Using existing computer games as benchmarks is interesting for multiple reasons. It makes it very easy to evaluate the agents, and the non-technical can understand and relate to the performance of such agents. A good example of this is a video of Robin Baumgarten's Mario AI agent , which has been viewed over 1 million times [4].

Multiple games will be used for testing the PS model and its different features, together with a comparison to RL algorithms. The graphs and results shown will usually be an average over multiple agents, so as to smooth out the effect of randomness, due to the many random variables that exist in these models. If 2 agents do the same task, one might get lucky at the first try, while the other might need hundreds of steps to achieve the same performance.

In recent years, more complex and interesting benchmarks have been used, such as the regular games (Angry Birds [26], Poker[12], Mario[29], Platform games[51], Starcraft[44], Car racing[33]) .

This thesis will focus on the Mario game, since it is a game most people know, and many variants of agents have been developed for the game and can, thus, be compared.

4.1 XOR exercise

The XOR exercise is a simple problem to demonstrate that the agent can learn the logical binary operator XOR.

4.1.1 Environment

Perceptions: $P = \{00, 01, 10, 11\}$ ($0 = false, 1 = true$)

Actions: $A = \{0, 1\}$ ($0 = false, 1 = true$)

Reward 1 if correct, 0 else.

4.1.2 Results

In this problem no advanced features are needed. The only thing that is needed is the standard methods for hopping probability and updating the edges. Afterglow, reflection, and associative memory can be turned off.

Damping can be set to $\gamma = 0$ since the environment is static. Reflection which is used for speeding up learning, is not needed for this task. The agent receives feedback instantly from the environment, so no afterglow is needed ($\eta = 1$) either.

In this game, the agent will start to be successful as soon as it has guessed the correct perception-action pair once. And depending on the reward and hopping probability function, it will grow to a 100% success rate. A high reward will make it learn faster, as well as a hopping probability function that follows the policy more strictly (e.g SoftMax and greedy).

```
Random r = new Random();
List<Boolean[]> perceptions = Arrays.asList(new Boolean[][]{
                    {false, false}, {false, true},
                    {true, false}, {false, false}});

List<Boolean> actions = Arrays.asList(false, true);
double damping=0.0, glow=1.0;
int reflection=1;
PS<Boolean[], Boolean> ps = new PS<>(perceptions, actions, damping, reflection, glow);

//Training
for (int i = 0; i < 20; i++) { //Train 20 times, since random chance of seeing a percept
   Boolean[] perception = perceptions.get(r.nextInt(perceptions.size()));
   Boolean action = ps.getAction(perception);
   boolean isCorrect = (perception[0] ^ perception[1]) == action;
   ps.giveReward(isCorrect);
}
//Test and output
perceptions.stream().forEach(p ->
      System.out.printf("\n%s : %s", print(p), print(ps.getAction(p))));
---------------------------------------------------
Output:
00 : 0
01 : 1
10 : 1
11 : 0
```

Listing 1: Code sample of how to train the PS model to learn XOR. The PS model can be found in [67].

4.2 Invasion Game

The Invasion game is an experiment introduced in [8]. The game can easily be made more complex and difficult by extending and reversing the world. Thus it is a suitable task for testing different parameters and features of PS.

The attacker A, will show a flag to indicate which way it will go. The defender D, will make a decision on either going left or right depending on the flag. To begin with the flag has no meaning to the agent. For the agent, the left flag could, for instance, mean that the agent will go right.

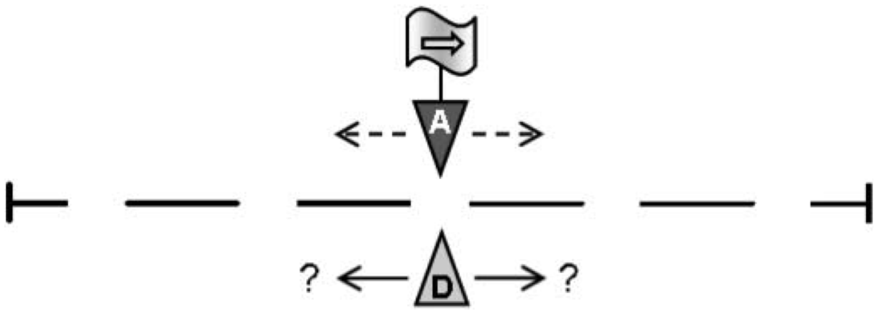


Figure 3: Invasion game. Illustration from [8]

4.2.1 Environment

Perceptions: $P = \{\Leftarrow, \Rightarrow\}$ (Flag left, Flag right)

Actions: $A = \{-, +\}$ (Action left, Action right)

Reward 1 if the agent goes the correct way, 0 otherwise.

The game can be extended by:

- Introducing colours to the flags, $\{red, blue\} * \{\Leftarrow, \Rightarrow\}$. The colours have no effect on the game mechanics (both $blue\text{-}left$ and $red\text{-}left$ hits that the attacker will go to the left). But for the agent this is a completely new percept.
- Introduce more percepts together with new actions.
  $P = \{\overset{\Leftarrow}{\Leftarrow}, \Leftarrow, \Rightarrow, \overset{\Rightarrow}{\Rightarrow}\}$ $A = \{--, -, +, ++\}$
- Reverse the world, such that the left flag $\Leftarrow$ now signals that the agent will go right, and the $\Rightarrow$ that the agent will go left. This will test the agents ability to forget previous knowledge, and its ability to adapt to changes.

4.2.2 Results

As in the XOR game, a simple PS model is all that is needed. The feedback will be given directly after an action is performed, which means no afterglow is necessary.

The damping parameter shows its impact in a changing environment like this (see fig

4). When the world is reversed, the agent will be punished for doing something it got rewarded for previously. The lower the damping (γ close to 0), the faster the agent will learn. But it will also be slower to adapt to changes. In a static environment the damping can usually be set to $\gamma = 0$. The agent will then always remember the past, and use that information for all future decisions.

The regular hopping probability function is used for all graphs for this experiment, unless specifically stated otherwise.

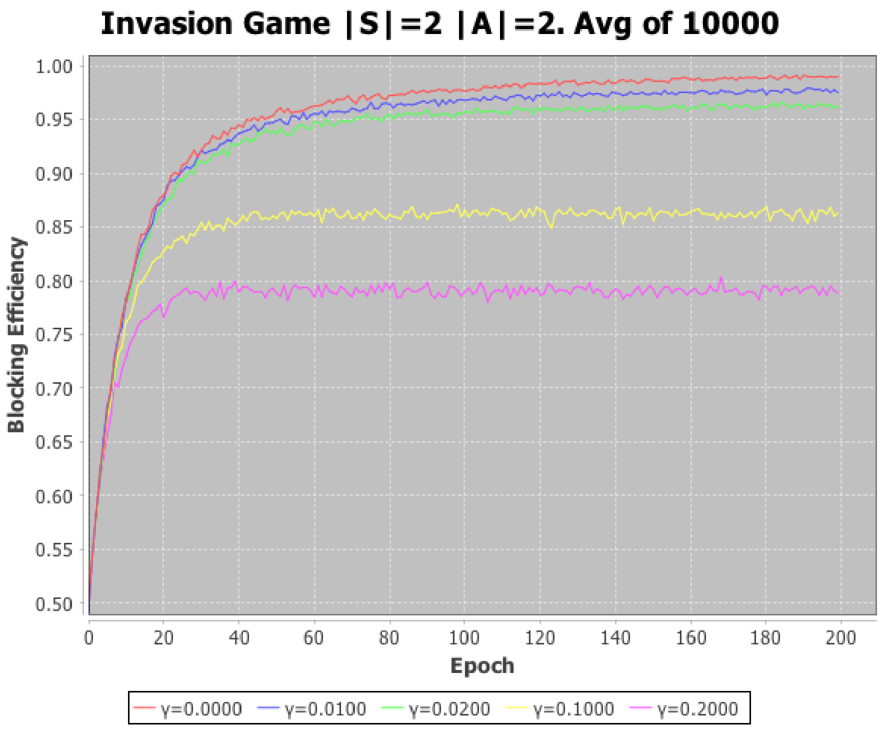


Figure 4: Agents with low damping $\gamma \approx 0$ learn faster and quickly find the optimal solution. Agents with larger γ, will forget more, and be less sure of what to do. We see that an agent with high damping reaches a peak in performance at around time 20, this being where the damping together with the small reward $R = 1$ makes the edge weight stationary.

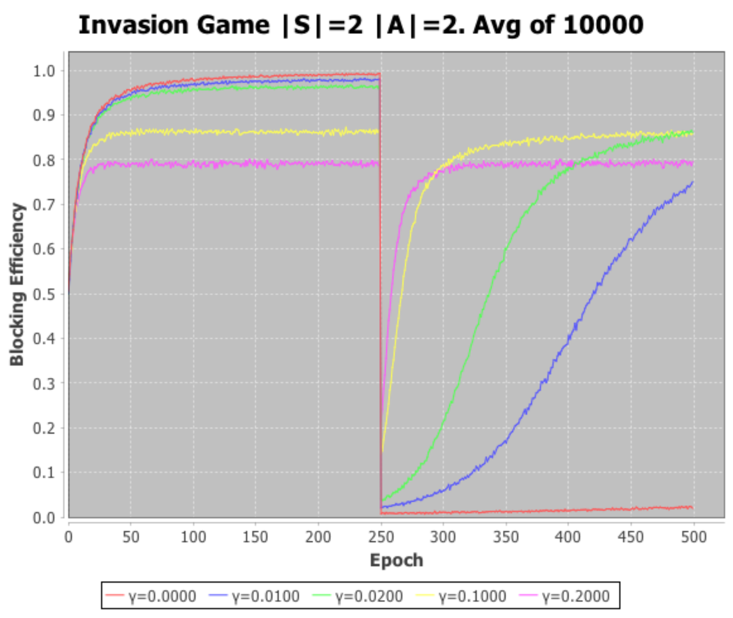


Figure 5: The flags change their meaning at $t = 250$, when everything is reversed. This graph demonstrates the point of having a higher damping value. For some time after $t = 250$, the previous worst agent is now the best. But we see that the other agents (with lower γ) soon catch them again. The lower the damping, the slower the adaptation to reversed actions. The agent with $\gamma = 0$ will need hundreds of epochs of training to recover.

In figure 5 the yellow agent performs decent overall with a peak around 85% success rate. To achieve better performance turning on emotion can help.

Turning on reflection

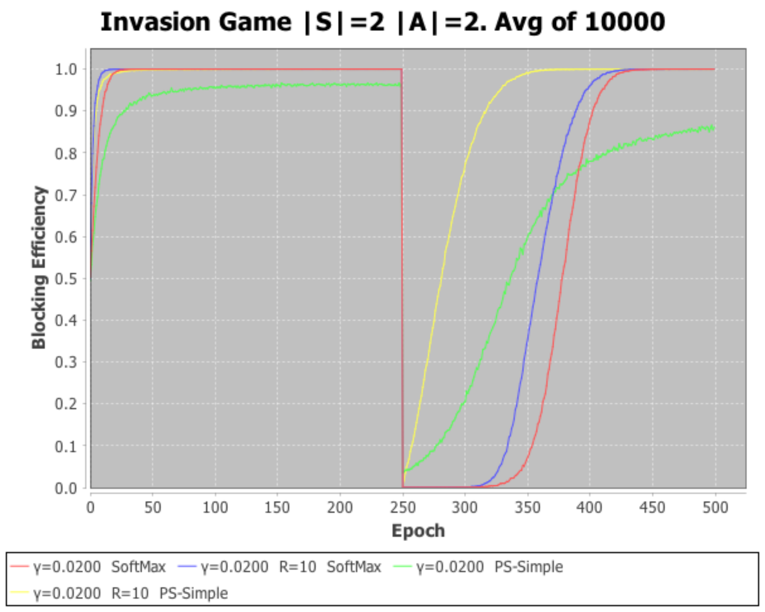


Figure 6: PS with Reflection: The yellow and green agents use the standard hopping probability function, and we see that by letting it reflect, the green agent goes from performing worst to performing best.

Agents with standard hopping probability damping did not perform very well from the previous experiments 5. But if let the agents reflect, then the situation changes drastically.

In figure 6 an agent is allowed to reflect ($R = 10$ in the graph). Agents with the standard hopping probability function now perform better with a peak at 100% accuracy and adapts faster than SoftMax. This is because the reflecting agent uses mostly the short time memory (emotion) for the decision making, whilst ignoring the long term memory (edge weights) in the transition face. The standard function has a higher chance of hopping to less rewarded clips, exploring, and as a result it will have a higher chance of finding a positive edge. For SoftMax (figure 6), the adding of reflection does not change the behaviour as much, due to the high probability of selecting the most rewarded clip.

Note: The reflection function used here does not remove negative rewarded actions. The agent can discard one hop because it was negative, but the next reflection could still give the same hop.

Adding new percepts over time

The PS model allows for new percepts to be added whilst training. When a new percept is seen, the corresponding percept-clip will be added to the ECM. The new clip will be connected to all the action-clips (or all clips, if memory association (3.8) is on) in the network. The network will grow as the new percepts are experienced.

In figure 7 we see how the PS model handles receiving new unseen percepts and a constant increasing percept space. The agent starts with 2 percepts, and over time the percept space is doubled. The percepts given to the agent are random over the whole state space. So in figure 7, at time $t = 1338$, the state space is increased from 128 to 256. The agent needs 600 steps to be able to learn, which is acceptable since the percepts are random chosen from a possible 256. The graph shows that reflecting agents are learning faster.

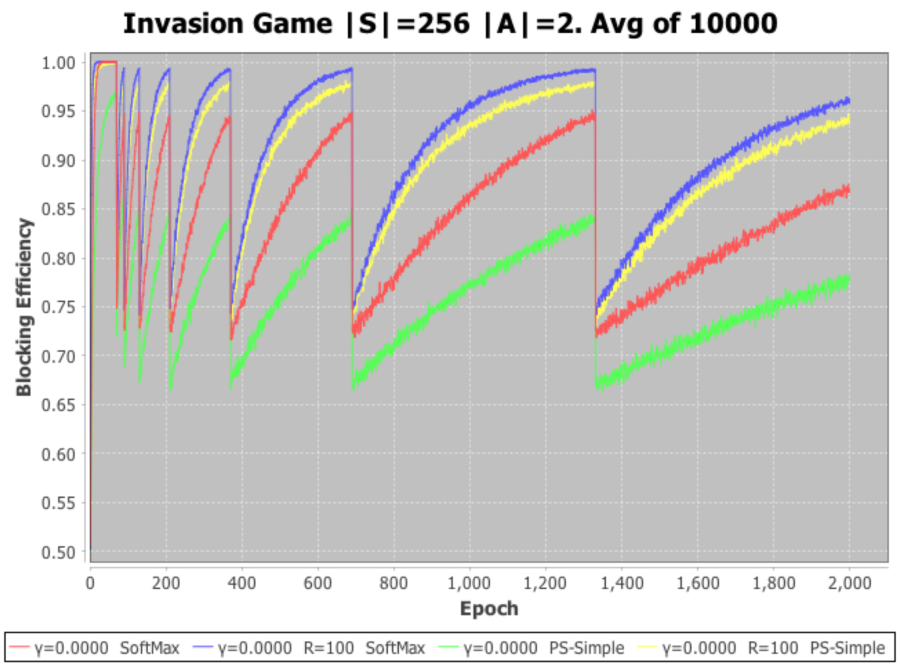


Figure 7: Adding new percepts while training. The number of states is increased $\{colour_1, colour_2, ..., colour_n\} * \{\Leftarrow, \Rightarrow\}$. Every time step $t = states * 10 + 50$, a new colour is added (The reason for having a dynamic time is to make sure the agent has time to see each percept before more percepts are added. The percepts given are randomly chosen from the current state space.)

Multiple actions and percepts

Above it was shown how PS handles new percepts. It is also possible to introduce new actions as well as new percepts. In figure 8 the action- and state-space is increased to 4. From the figure we see that the agent need more time to reach 100% accuracy in a increased state- and action space.

The figure 8 also shows how the agents tackle reversing of this bigger state space. The overall winner agent is the yellow ($\gamma = 0.1, R = 5$). After the reverse at $t = 100$ 50 steps are needed before it is back to 100% success rate, whilst the same agent without reflection (pink) needs ≈ 200 steps to recover.

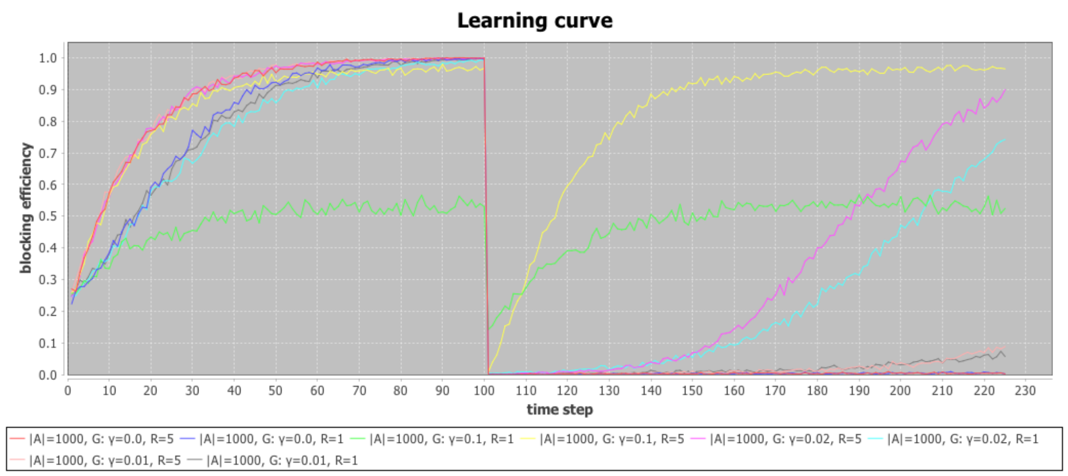


Figure 8: 4 percepts and action. Comparison on using reflection. At time $t = 100$ all 4 percepts were reversed.

In figure 9 we can see the ECM network visualised.

Comparison of hopping functions

4.2.3 Associative memory

In this experiment the associative memory feature will be tested. Before percept clips would only hop directly to action clips, here the clips are free to jump to any clip. When a percept clip is perceived, a direct sequence to an action clip is initiated, if the edge is

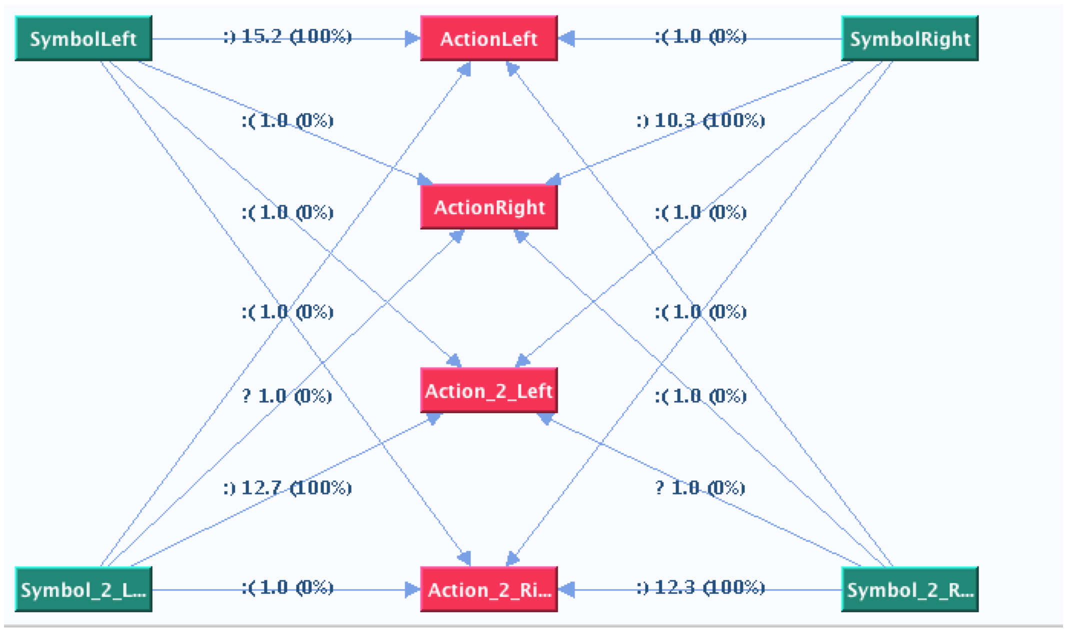


Figure 9: ECM network. Each node is a clip, the green nodes are percepts and the red are actions. The agent($\gamma = 0.1, R = 5$) is shown after 150 time steps. On the edges we see the emotion tag, edge weight and the hopping probability.

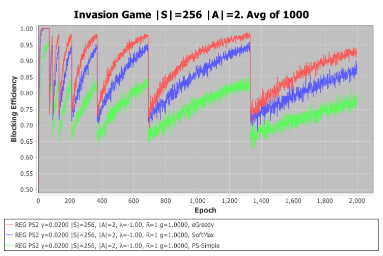
positive the action is executed. If it was negative, the percept clip will hop to another clip, where the same procedure is repeated.

Using different type of afterglow, edge-glow (3.7.1) or clip-glow (3.7.2) will lead to different update in the network.

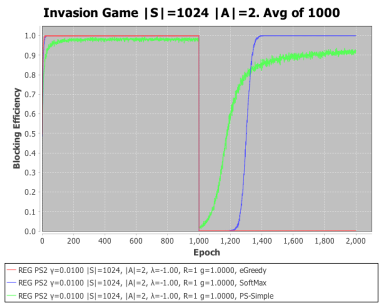
In the figure 11 the different methods are compared to regular directed sequences. The percepts space is of size 6, and 2 actions. In the experiment the maximum deliberation length is set to $D = 2$, which gives sequences of $s = (p, c_1, c_2, a)$, where p is the first percept, c_1 is the percept-clip, c_2 is a possible second clip and a is the action clip.

The agents tested are (all 3 will be tested with reflection off ($R = 1$) and $R = 4$):

- Standard PS
- With associative memory and clip-glow (Standard)
- With associative memory and edge-glow



(a) Same experiment as in figure 7, where a new colour is added at some time. The agents tested have different hopping functions. We see that ϵ-Greedy, which follows the policy more strongly, performs better.



(b) Here the world is reversed, and we see the benefit of functions that exlpore more. Observe that ϵ-Greedy struggles, even with $\epsilon = 0.25$, but ϵ-Greedy could be helped by having negative rewards, as it would then remove old knowledge more quickly.

Figure 10: A comparison of hopping probability functions. The agents have the same parameters except the hopping function. The ϵ-Greedy function does not work that well in a changing environment where damping is the only punishment.

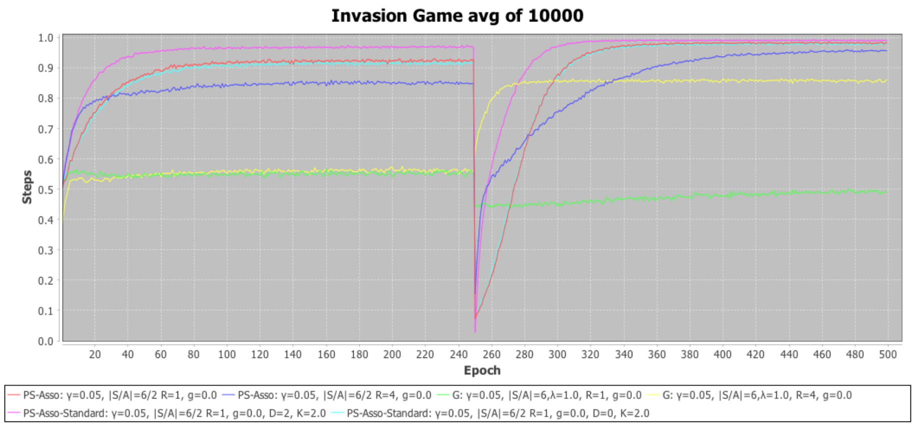


Figure 11: 2 percepts and action. At time $t = 250$ the world is reversed.

From the figure 11 we see that the agents with associative memory performs better than the regular. With the pink, association and clip-glow, is the winner.

The result here might not be accurate with what is done in the the papers, due to its lack of details. One unclear point is the emotion updating in the sequences, where it is wanted to update the whole chain of sequences $s = (c_1, c_2, a)$. Lets say $(c_2 -> a)$ is a strong and positive edge, while $(c_1 -> c_2)$ is new and uniform. c_1 hopped to c_2 with a random chance, c_2 is strongly connected to a with a positive edge. While c_1 choose of c_2 is wrong, an update would adjust the $(c_2 - > a)$ edge too by damping and a negative emotion. The next time c_2 is exited, it would probably skip that edge, due to c_1 triggering the previous sequence.

4.2.4 Generalisation

With associative memory the clips could excite other perception clips. With generalisation, the agent can create new fictional clips, which then can be used later.

Following the example from [39], which is similar to the Invasion Game (new names, just directions and colours), by doing the same experiment. This experiment is related it to traffic lights, green light would mean drive and red light means stop. The percepts are $S = (|colour|, |arrow|)$, with 4 percepts: $\{green, red\} * \{\Leftarrow, \Rightarrow\}$, and 2 actions $\{stop, go\}$

To show the potential, we have a changing environment where the reward scheme change 4 times. A positive reward $r = 1$ will be given in these situations:

a) $0 \leq t \leq 1000$ Drive on green light, stop on red light. Arrows are irrelevant.

b) $1000 < t \leq 2000$ Reversed from above, drive on red light, stop on green light.

c) $2000 < t \leq 3000$ Drive on left arrow, stop on right arrow. Colours are irrelevant.

d) $3000 < t \leq 4000$ Always drive.

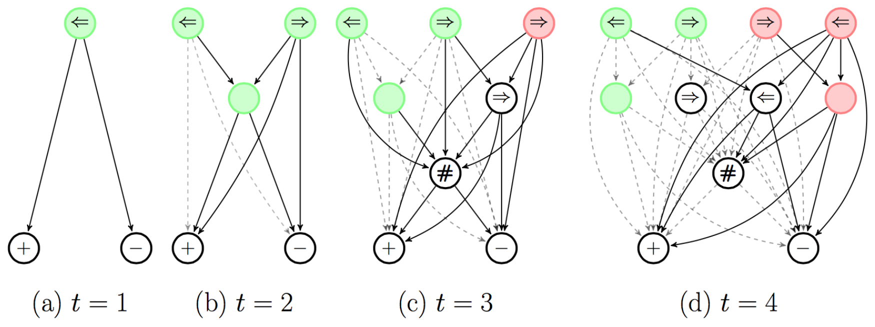


Figure 12: The network visualised, from [39]

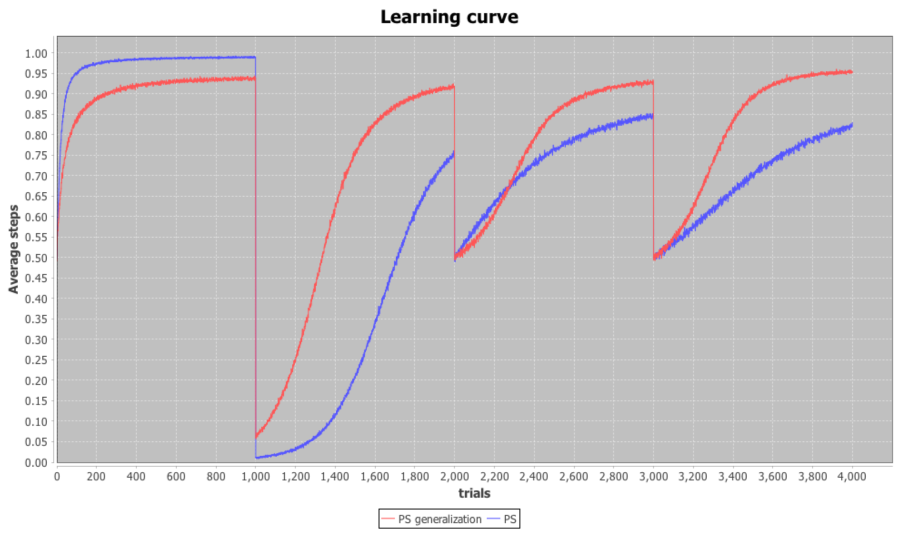


Figure 13: PS generalisation compared to a simple standard PS model. The results show that the generalizing agent are much more adaptable to changes. Both agents use the same parameters $\gamma = 0.005$ and a average of 10^5 agents.

The generalisation will be further explored in the Mario section 5.5.

4.3 Grid World

The grid world game is a maze puzzle where an agent should learn the path to the goal. The end goal is fixed, whilst the start position could change without any difficulty for the agent. The game ends once the goal is reached.

Compared to the previous Invasion game, this is a more difficult problem due to the delayed rewards. An agent will only be given a positive reward once the goal is reached, meaning that until the goal is reached for the first time, the agent will have no idea or preference as to what it should be doing. Once the agent reaches the goal, it will have to remember the steps it did, and later check if a quicker path exists.

4.3.1 Environment

Perceptions: All cells in the game, i.e. width*height percepts.

Actions: $A = \{left, right, up, down, nothing\}$

Reward: 1 if the agent has reached the goal, 0 otherwise.

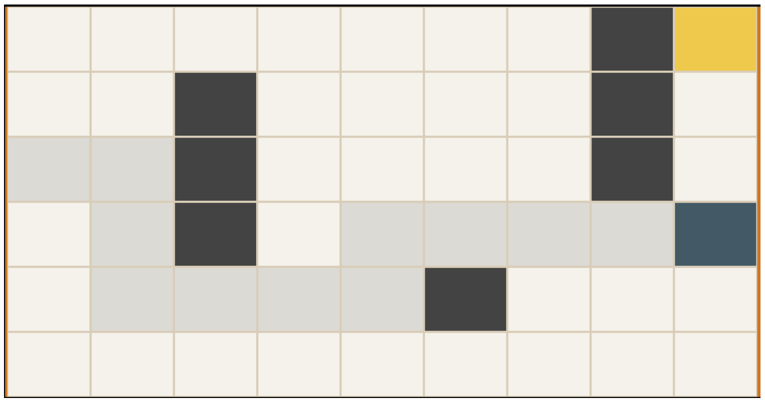


Figure 14: Grid World game. $6 * 9$ matrix. The cell colours represent: yellow is the goal, black are walls, blue is the player, dark-grey is the current path walked.

4.3.2 Results

Afterglow is absolutely necessary in this task. To use afterglow, the damping on afterglow $(0 \leq \eta \leq 1)$ needs to be chosen. This decides how many and to what degree a previously used clip should be excited. η should be sufficiently low so that the agent can have the same number of excited clips as the shortest path. But if it is set so that the agent must try all possible paths until it, by luck, finds the shortest path, then this is really just a brute-forcing to find the solution. The better way is to allow a longer path of excited clips.

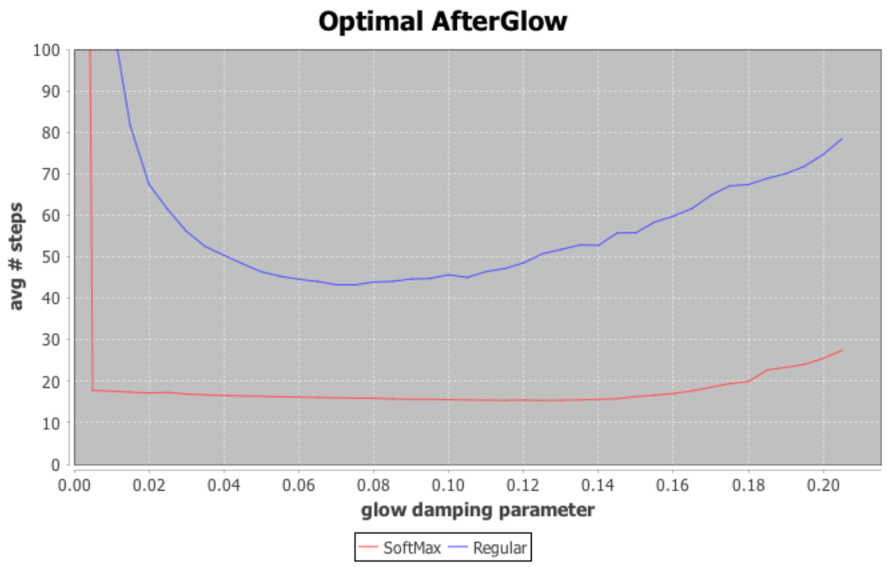


Figure 15: GridWorld $6 * 9$: Showing the performance with different afterglow η parameters and hopping probability functions. Results after training 100 games, and average of 1000 agents. Damping is set to $\gamma = 0.0$. This is the same world used in [40], and the resulting graph is identical which confirms the correctness of the implementation.

The figure 15 shows a clear advantage of using SoftMax over the standard hopping probability function. On the $6 * 9$ grid the optimal glow is $\eta = 0.125$ using Softmax, with 15.36 steps on average. For the regular PS function $\eta = 0.075$ gives the best performance, 43.22 steps.

Using a bigger board: $10 * 10$

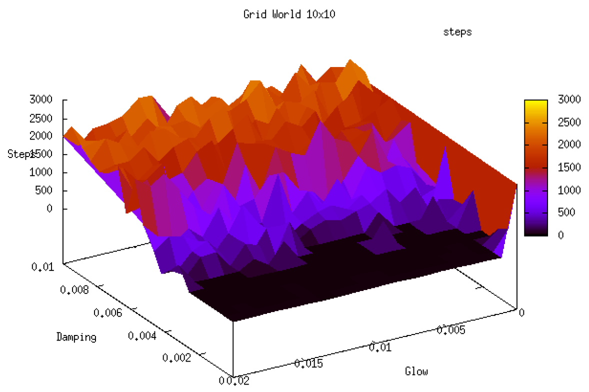
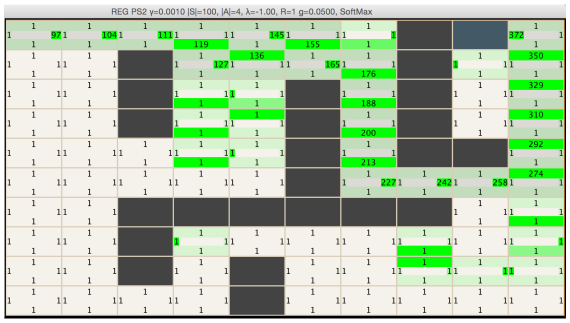
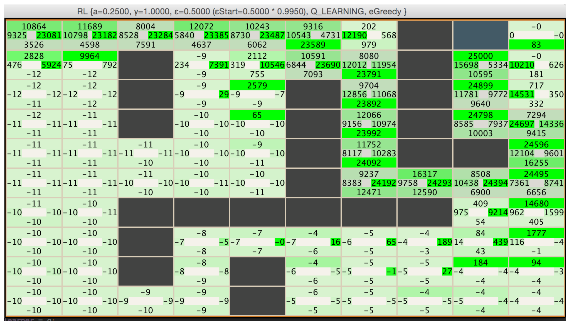


Figure 16: GridWorld $10 * 10$ 3D surface: Shows how both damping and glow affect the performance. The damping should be off or close to zero and glow in the range 0 to 0.02 .


REG PS2 γ=0.0010 |S|=100, |A|=4, λ=-1.00, R=1 g=0.0500, SoftMax

(a) PS edge weights $h(c_i, c_j)$ visualised. The agent has made a detour at $(0, 4)$. Instead of going right, it goes down, right ,up. This only happen on a small proportion of the set, but it is a problem which the current PS model cannot overcome.


RL (a=0.2500, γ=1.0000, ε=0.5000 (εStart=0.5000 * 0.9950), Q_LEARNING, eGreedy)

(b) Q-learning $Q(s, a)$ values visualised. The best path is found.

Figure 17: Agents visualised after 1000 games. Shows the agents desire to go from one state to another. The green colour is strengthened for the strongest edge compared to the other edges, for each state. The PS agent has damping on, which pulls unused and bad percepts down to the stationary weight $h = 1$. Q-Learning receives a punishment of $r = -1$ for all states that are not the goal.

Using a bigger board: $20 * 20$

Another new maze is introduced, with a $20 * 20$.

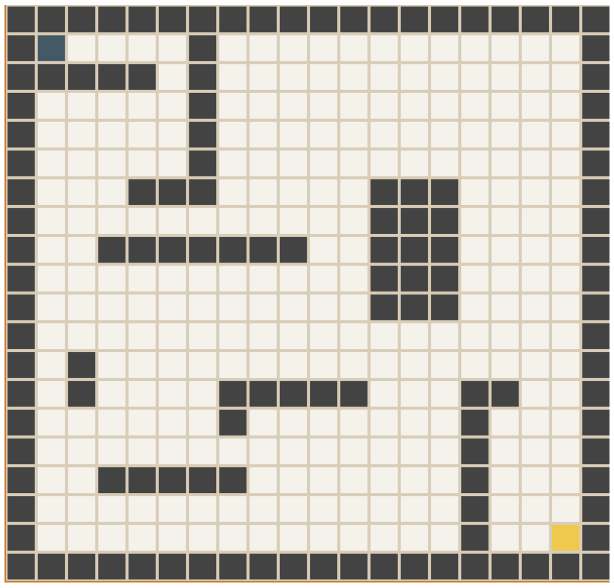


Figure 18: Grid World 20 $*$ 20. Board visualised. The minimum path requires 38 steps.

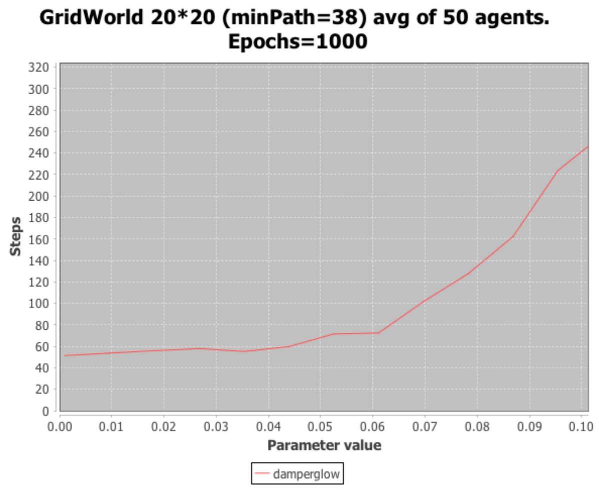


Figure 19: Grid World 20 $*$ 20. Finding best parameters glow value.

From figure 20 we see how the models compare as the state space increases. The PS agents learn a decent path after only $\approx$ 25 games. It takes Q-learning 150 steps to go down to the same performance. But after that, RL keeps looking for better routes, and after 500 steps the optimal route is found.

In general we see that RL is better at planning and finding the optimal path. PS is usually faster at finding a decent solution, which may or may not be the optimal solution, and even with unlimited tries it might not find it. The update equation in PS does not take into account how good the next state is, which can lead to some unnecessary steps. While SARSA and Q-Learning will converge to the optimal path in experiments that are MDP's (like this game).

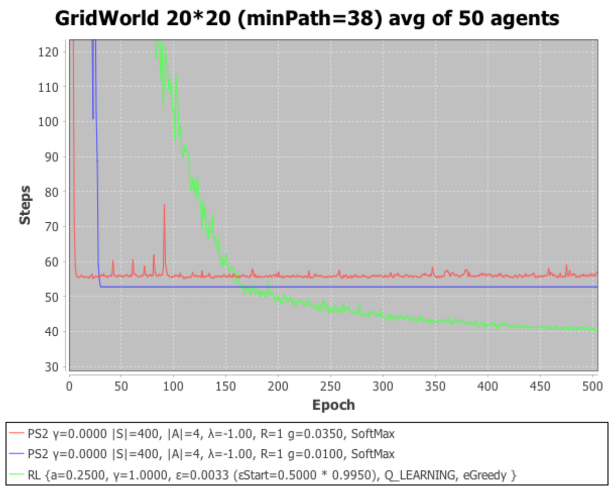


Figure 20: Grid World $20 * 20$. Comparing the performance and learning time of PS and Q-Learning.

4.4 Mountain Car

Mountain Car is a classic RL task where the objective is to drive an underpowered car up the goal the hilltop on the right. The gravity is stronger than the engine, which means the car has to build up momentum by "swinging" back and forth.

4.4.1 Environment

The player has 3 possible actions that will power the engine : $\{left, neutral, right\}$. The game environment contains two variables : *position* and *velocity*. The car starts at the bottom with position $x = 0$ and velocity $v = 0$.

This game is interesting for learning algorithms since the world (perceptions x and y) are real values, which leads to an infinite number of world states. The continuous state can be solved by some type of discretization or function approximation. A discretization

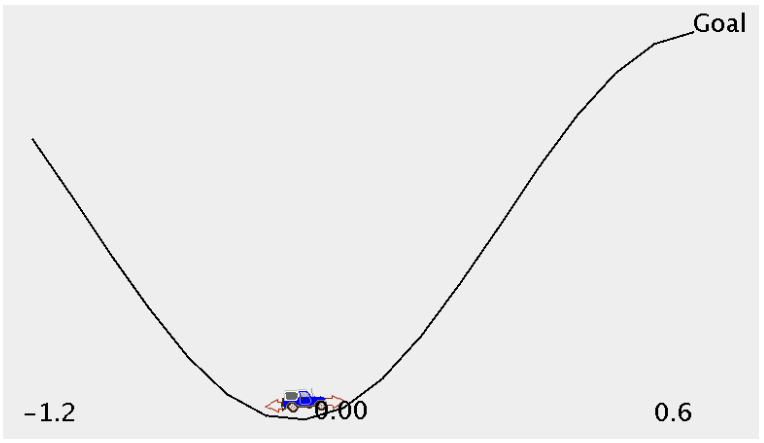


Figure 21: Mountain Car game

method called Tile Coding [56] will be used here. The continuous states will be pushed into discrete state buckets.

Perceptions: position $x \in [-1.2, 0.6]$ is divided into n intervals.
Velocity $v \in [-0.7, 0.7]$ is divided into m intervals.
Total of $n * m$ percepts.
A reasonable choice (23) for n and m are $n = m = 10$, which gives 100 percepts.

Actions: $A = \{left, right, nothing\}$ =$\{-1, 0, 1\}$

Reward 100 if the agent is in goal ($x = 0.6$), 0 otherwise. (RL agents are punished by -1)

For every action the world is updated by:

$$\begin{aligned} v_{t+1} &= v_t + 0.001 * Action - 0.0025cos(3x_t) \\ x_{t+1} &= x_t + v_{t+1} \end{aligned} \tag{19}$$

4.4.2 Results

The results will not be comparable to the paper [40], since that paper uses different variables for the world. The paper [40] states that 36 actions to the left, and 63 actions to the right will get the car to the goal, which is not possible with the original equation 19 with the gravity factor $g = -0.025$ (even with endpoint at $x = 0.5$).

This thesis will use the world defined above and in [56, 42, 13]. The shortest possible path is 106 steps, with one solution being 13 right, 38 left, 55 right. A random agent needs around 40000 steps to finish (an average over 1000 agents).

The standard reward scheme for this experiment is a reward of $r = -1$ for every step that is not the winning move. For PS we can either use the negative scheme and turn off damping or use the regular PS reward scheme (always positive or zero). In figure 24 negative reward is used and in figure 25 a regular PS reward scheme is used. The results are equally good, whilst the damping is more computationally heavy.

ϵ-Greedy is used for RL, with a decreasing ϵ over time as a state is used. This is used to overcome a local best solution. In a delayed reward scheme it can be useful to let the RL algorithms use eligibility traces, a list of recently used $Q(s, a)$ such that rewards can be distributed (similar to PS afterglow). But an important note is that Q-Learning and SARSA do not require it, even for this type of task. Q-learning and SARSA both consider the next $Q(s_{t+1}, a_{t+1})$ when updating $Q(s_t, a_t)$, which means that when the agent randomly wins for the first time then only that Q is rewarded, and the next time it comes to a state leading to that state, it will update itself with some reward since the next state is highly rewarded. Similarly to dynamic programming, the good or bad states will propagate backwards. This is a slow process, and using eligibility traces will speed it up.

To make a fair comparison both PS and RL should be able to have the afterglow and eligibility trace feature respectively.

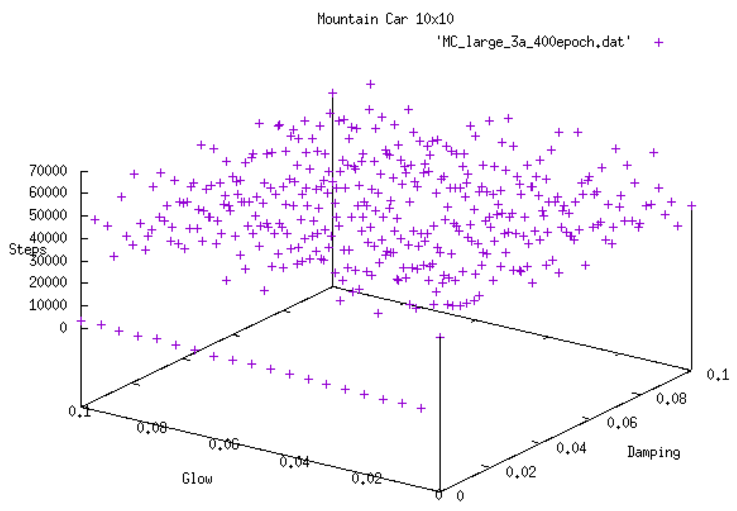


Figure 22: Mountain Car PS parameters: This shows quickly that damping should be off $\gamma = 0$. The best value for glow $\eta = 0.0194$ used 184 steps.

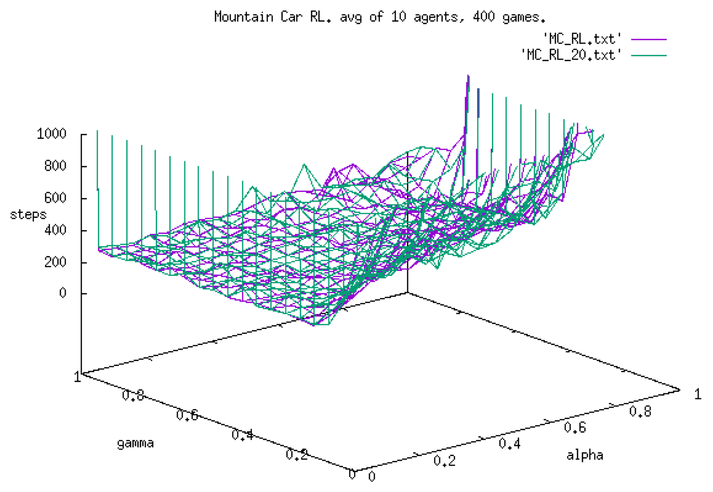


Figure 23: Mountain Car RL parameters: Green lines use a $20 * 20$ state space, while purple use $10 * 10$. Agents displayed: Q-Learning $\epsilon = 0.001 * 0.9$. The best parameters from the graph were $\alpha = 0.9474$, $\gamma = 0.7895$ with 201 steps on $10 * 10$ and 214 steps on $20 * 20$.

Model	Epoch 100	Epoch 400	Parameters
Random	40000	40000	Random(3 actions)
PS	240	184	$\gamma = 0, \eta = 0.0194$, SoftMax
SARSA	256	235	$\alpha = 0.0533, \gamma = 0.0533, \epsilon = 0.0067 * 0.9$, Greedy
Q(λ)	220	220	$\alpha = 0.1, \gamma = 1.0, \eta = 0.0051 * 0.9, \lambda = 0.3$, Greedy

Table 2: Mountain Car performance: Table of the best agents for each model. Average of 1000 agents.

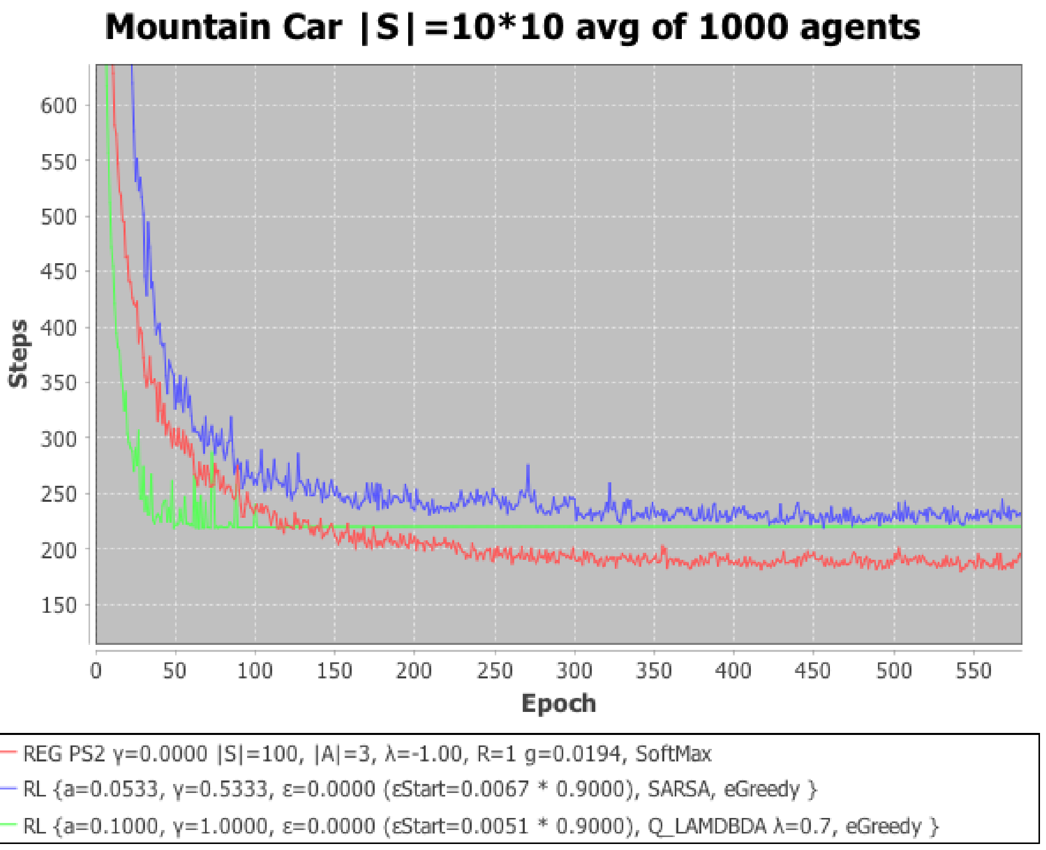


Figure 24: Mountain Car 10*10 states: The red PS agent are the clear winner, by reaching $\approx$ 200 steps. While both Q(λ) and SARSA only reach $\approx$ 225 steps. In this example the PS agent had damping turned off and instead received negative rewards (like RL).

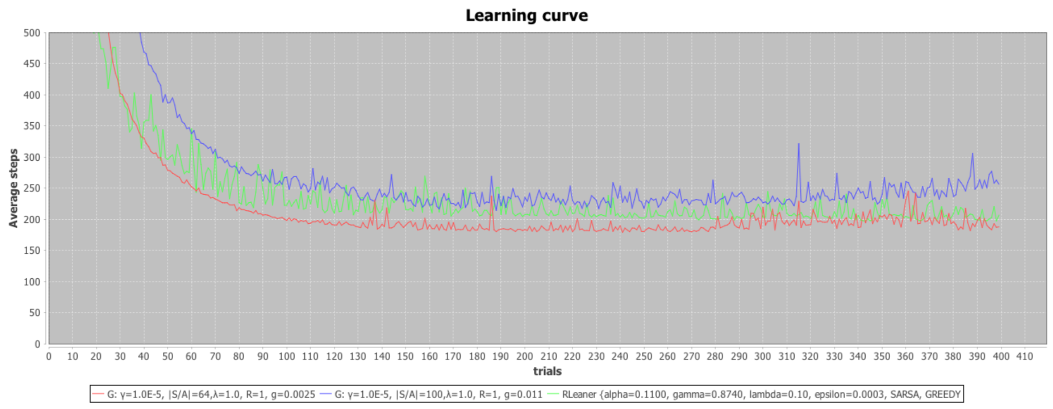


Figure 25: Mountain Car 10*10: PS also performs well with the regular zero $\lambda = 0$ reward. Damping has to be on, which makes the training slightly slower. The red PS agent uses a $8 * 8$ percept space, whilst the blue agent uses a $10 * 10$ space.

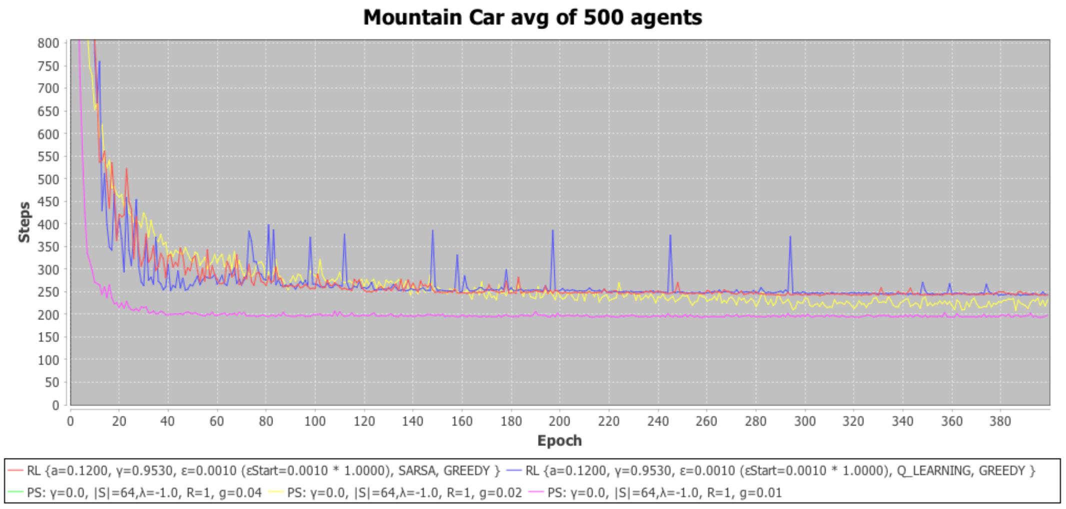


Figure 26: Mountain Car 8*8 : 64 percepts. PS vs SARSA. The pink PS agent quickly learns a good solution, whilst the other PS (Yellow), needs more time.

Both PS and RL algorithms perform well at the game. For the best performance both need a method for rewarding recent state-action pairs. The RL algorithm does not manage to find the optimal solution. One reason might be that the discretisation of the continuous states does not satisfy the markov property 2.1.

4.5 Cart Pole - Pole balancing

Cart Pole (also called Pole balancing, inverted pendulum) is a classic RL problem. The objective is to balance a pole that is attached to the top of the cart, by giving power to the motor in the cart. The cart can move on a horizontal axis by applying left or right thrust from the motor. The game ends when the pole falls or the cart hits the walls at the end of the track.

The problem is difficult due to the complex and non-linear description of the system. Similar to the Mountain Car problem, we have continuous variables, except in this game we have 4 variables. Multiple definitions of the game rules and world variables exist, and this thesis will follow the definitions from [3, 11, 18, 35].

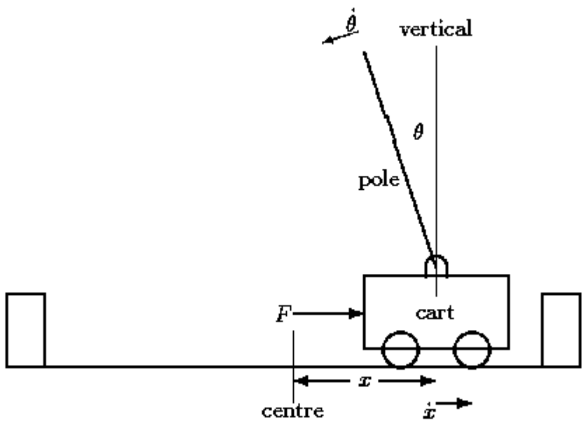


Figure 27: Cart pole illustration from [20]

4.5.1 Environment

The state of the system is defined by four real values: the position of the cart x, the velocity of the cart x' ,the angle of the pole relative to the cart θ, and the angular velocity of the pole θ'. The equations for motion and system parameters are fully described in [11].

Perceptions: Real values : θ', θ', x, x'

Compressed into 162 different states using the BOXES scheme from [3]

Actions: $A = \{left, right\}$

Reward: -100 if the pole has fallen or if the cart hit the walls, 1 otherwise

4.5.2 Results

The game only ends once the agent fails which makes it possible for an agent to keep playing for an infinite time. The agent will want to play the game forever, which is the opposite of Mountain Car where the goal was to end the game as soon as possible. A limit of a maximum 10^5 steps is therefore set, which is accepted as a victory and ends the game. Pressing random actions will on average manage to balance the pole for 22 steps (averaging over 10^6 random agents).

An agent will receive constant positive feedback until the pole falls or the cart hits the wall, in which case a large punishment will be given. A challenge for the agents is that there are multiple situations and states where a failure is irreversible (e.g the poles angle velocity is too large for a recovery to be possible). These irreversible states will get positive rewards from the environment, since they are not failure states. It is therefore important for the agents to be able to look further into the future.

Finding optimal parameters

For PS, the damping parameter can easily be set to $\gamma = 0$ since there is no need to forget learned information. And all agents will be given a large negative reward when the game ends. Afterglow will be required, and by brute-forcing (fig-29), we find a glow of $\eta = 0.579$, which allows for a long sequence of clips to be excited. Looking forward is an important requirement in this game, which is confirmed by the RL algorithm performing best with maximum discount factor $\gamma \approx 1$ (fig-28), which means the future is as important as the current situation.

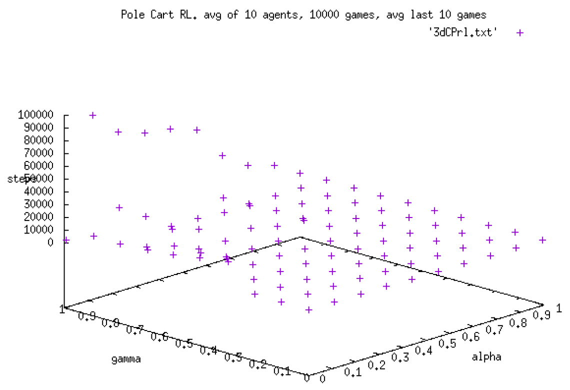


Figure 28: Cart Pole RL: Average of last 10 games, 10^4 games simulated with 10 agents. The agents shown use Q-Learning, $\epsilon = 0.001, *0.9$ and ϵ-Greedy. From the graph we find that the best parameters are $\alpha = 0.1199$ and $\gamma = 0.9990$ with 93292 steps, which is very close to perfect.

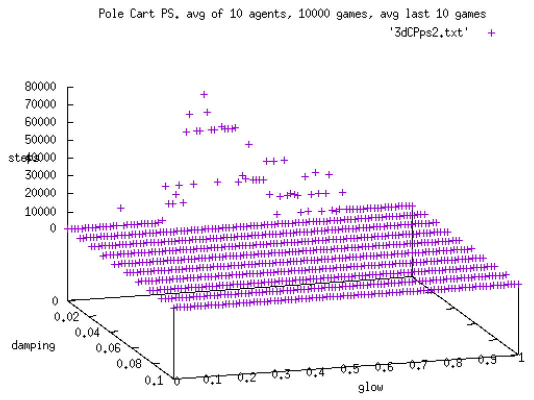


Figure 29: Cart Pole PS: Average of last 10 games, 10^4 games simulated with 10 agents. We see that damping does more damage than good, and the agents forget valuable information, even with low damping. Glow should be around 0.35-0.5, with glow $\eta = 3939$ being the best with 70289 steps.

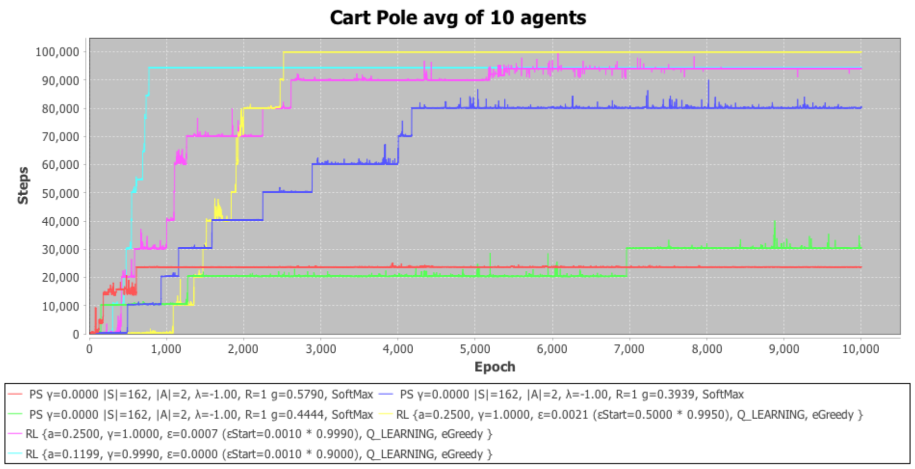
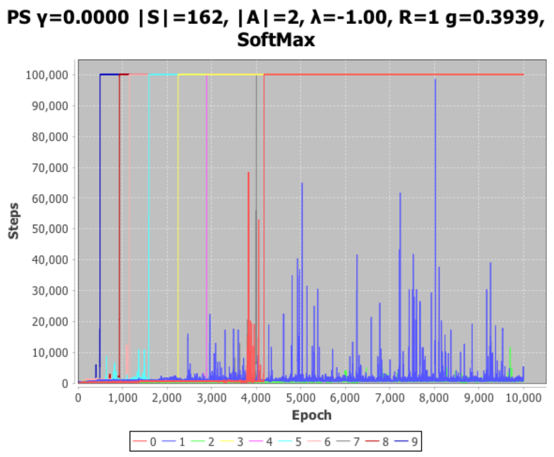


Figure 30: Cart Pole : Average of 10 agents. Trained for 10^4 games.

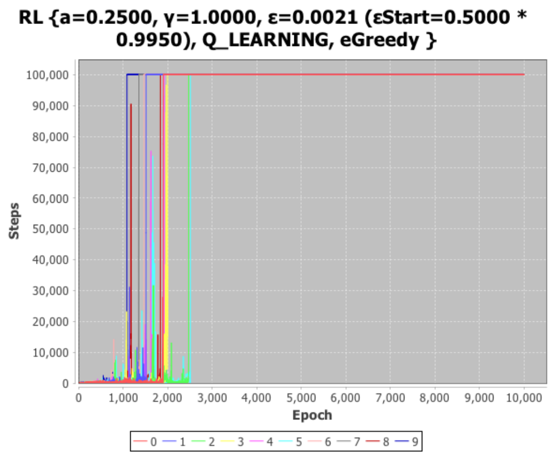
Both PS and RL agents learn to play the game well with well-chosen parameters. The Q-Learning (yellow in figure 30) learns the game perfectly for all agents. The PS sample has problems as some agents fail. The blue PS agent in (fig-30), which is the average of 10 agents, had 8 agents that found the perfect solution, while 2 agents struggled and dragged down the average performance. The individual performance of blue PS agents can be seen in fig-31a. For comparison, the yellow Q-Learning agent from 30 is shown in 31b. In general, the RL agents have a higher chance of overall better performance, and the PS model have problems as some agents do not find an optimal solution.

5 Game: Infinite Mario Bros

Infinite Mario Bros (IMB) is a public domain clone [1] of the classic Nintendo platform game. IMB provides a framework for implementing agents that can play and simulate the game. The game has been used in AI competitions for creating the best game-playing agent.



(a) PS: 8 out of 10 agents learned the game in the time frame, while 2 agents struggled.



(b) RL: All the agents learned a perfect solution after 3000 games.

Figure 31: Cart Pole: Difference between identical individual PS and RL agents. We see that its a fine line between a success and failure, in just a few games an agent can go from a bad performance to a good one. This is due to the luck of finding a good sequence of actions, which the agent can then repeat for the rest of the test.

Figure 32: MarioAI game

5.1 History - relevant research

In 2009 and 2010 Mario AI competitions were held for creating the best game-playing agent [28]. In 2009, 15 submissions entered the competition. The solutions were based on various learning techniques: evolutionary algorithms, neural networks, reinforcement algorithms and deterministic - "hard coded". The top scoring agents used A*, a heuristic path finding algorithm [23, 49].

The next year, the winner was also an A* agent, but it also used an evolutionary algorithm to breed the agent with the best rules (similar to state space for RL) [7]. The difference between REALM and the first A*, is that the first only had 'one task', which was *progress* ('go right without dying'). The REALM trained an evolutionary algorithm (pre hand) to determine which of 4 tasks it should start in different states. The other tasks were *attack*, *evade* and *powerup*. The tasks were executed with A*.

In 2011 and 1012, few submissions were entered. No official competition was therefore held for game-playing agents. There were multiple research papers written [52, 55, 59, 22, 48] on different approaches to the problem, but few were submitted to the contest. The reason might be that the A* solutions play the game perfectly.

The reason the A* agent worked so well, was that it had an internal copy of the game engine. It could therefore simulate with 100% accuracy what would happen in the next frames on a pixel accuracy [27]. The problem was then reduced to a search into future states (which were known to the agent). The agents that were based on learning did not perform as well as the heuristic agents, which is not surprising as, if it is possible to use full search algorithms then they will always outperform learning agents.

5.2 Game mechanics

The objective of the game is to guide Mario safely to the goal which is at the far right. There are 3 modes Mario can be in: *fire*, *large* and *small*. The fire mode allows Mario to shoot fireballs. Each time Mario gets hit or collides with an enemy he goes down one mode. The game ends if Mario gets hit by an enemy in *small* mode, falls in a gap, runs out of time or reaches the goal. To succeed in the game Mario will have to either jump over enemies, kill them by stomping or shoot fireballs at them.

There are an infinite number of different levels in the game (hence the name Infinite Mario), by using a random seed. The levels may contain enemies, helpful items, gaps and road blocks. The amount of hazards that appear in a level can be adjusted with difficulty parameters. The standard length of a level is 260 blocks, and will be used in most of the experiments (unless stated otherwise) to check the distance Mario achieved without dying.

5.2.1 Environment

The API presents a $22 * 22$ byte matrix of the environment at every time step. This matrix represents the scene around Mario, where Mario is situated in the middle (column 11 and row 10 and 11), see [29] for more details. In each cell there can be one of 15 elements : 4 different types of obstacles, 8 enemies and 3 friendly elements (coin, mushroom, flower/power up). That leads to 15^{22*22} states, which is an extremely large number, way too big for storage.

State abstraction is therefore needed to able to represent the world in a feasible size. Enemies will be treated as one element, and blocks will be treated as none or hard-block. This leads to 4 possible elements in each cell. 4^{22*22} is still to large, so the next step will be to reduce the view length without losing too much performance. A simple method is let Mario view a $n * m$ matrix around himself, with 4^{n*m} states. That number increases fast with larger viewing distance, so another trick is to collapse multiple cells into one cell.

Another approach to reduce size is to reduce the state space down to simple mappings from state to action, where the states describe exactly what action Mario should do (e.g "should Mario jump","go right", ...etc). The more preprocessing done, the easier it is for the agent and the better the results. However this preprocessing specialises the agent down to one task, and therefore the agent becomes less general.

The approach used in this thesis is more novel. The states represent a matrix around Mario with some cells merged together to increase the viewing range. Mario will then see a block and enemies that are some block away from him. This is similar to the ideas investigated in [7, 59, 41].

For the graphs a different state representation is used (details can be found in [66]):

1. Dynamic state (DS): 4*4 matrix surrounding Mario. 2^{16} states.
2. Dynamic state 3 (DS3): Larger matrix, and merged multiple blocks into one. 2^{20} states.
3. HardCodedState$N$: are different approaches to choosing what the states should contain.

All state representations also have a boolean flag for $canJump$ and $isStuck$. The stuck variable is useful because the agent might find a loophole where going one step left and then right will give him rewards. But when the stuck length keeps growing the agent will start to receive negative rewards. And if the agent is stuck, it should try do something else. The agent will also need to know if it can jump to know if its on the ground and holding in the jump button will make the agent stand still. The game mechanism does not allow for holding in the jump button for a long time, so Mario will just be standing still if the button is not released and pressed again.

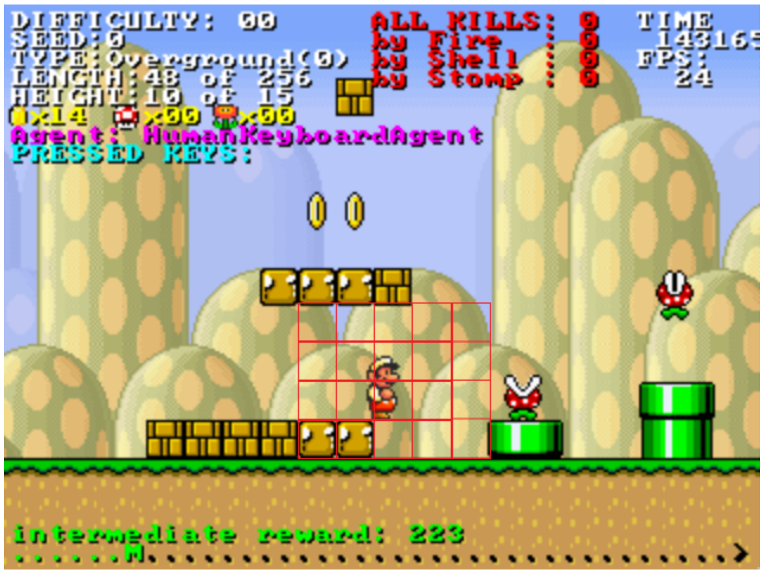


Figure 33: MarioAI game with a $4 * 5$ grid. We see that this grid is small for planning ahead. Some merging of cells will be needed to increase the viewing distance.

5.2.2 Actions

The game has four keys that can be pressed : $\{UP, DOWN, LEFT, RIGHT, JUMP, SPEED\}$. Any combination of the keys can be pressed at the same time, $RIGHT, JUMP, SPEED$

which leads to Mario jumping fast forward. The keys UP and $DOWN$ are mostly ignored in this paper, since the purpose they serve is insignificant. An action is represented as a boolean array of length 4. This gives $2^4 = 16$ possible actions. Some of these actions are useless (e.g holding left and right key at the same time). Ignoring these actions we end up with a total of 12 actions that perform a movement.

ID	Name	LEFT	RIGHT	JUMP	SPEED/FIRE	Vector
0	DO-NOTHING	0	0	0	0	$\{0,0,0,0,0,0\}$
1	LEFT	1	0	0	0	$\{1,0,0,0,0,0\}$
2	RIGHT	0	1	0	0	$\{0,1,0,0,0,0\}$
3	JUMP	0	0	1	0	$\{0,0,0,1,0,0\}$
4	FIRE/SPEED	0	0	0	1	$\{0,0,0,0,1,0\}$
5	LEFT-JUMP	1	0	1	0	$\{1,0,0,1,0,0\}$
6	RIGHT-JUMP	0	1	1	0	$\{0,1,0,1,0,0\}$
7	LEFT-FIRE	1	0	0	1	$\{1,0,0,0,1,0\}$
8	RIGHT-FIRE	0	1	0	1	$\{0,1,0,0,1,0\}$
9	JUMP-FIRE	0	0	1	1	$\{0,0,0,1,1,0\}$
10	LEFT-JUMP-FIRE	1	0	1	1	$\{1,0,0,1,1,0\}$
11	RIGHT-JUMP-FIRE	0	1	1	1	$\{0,1,0,1,1,0\}$

Table 3: All the actions Mario can perform. UP and DOWN are not used for this demonstration, and other duplicate or useless actions are ignored in this table.

Some implementations [59, 7, 59] use a high-level of abstraction for the actions available to the learning agent, such as tasks that will execute a sequence of heuristic key presses, that take the control away from the learning for some steps. These tasks could be : Attack, Upgrade, Pass Through, Avoidance. An avoidance task could take control over Mario, and move him away from the enemy without the agent doing anything.

In this thesis the agent will have to do the key presses directly (as done in [65]). This will increase the difficulty in situations where a sequence of actions are needed. To jump over an enemy or block, the jump key has to be held in for multiple steps, followed by some right key presses, meaning that the agent has to figure out the exact key-presses needed to avoid an enemy itself.

5.2.3 Reward

The goal in the game is to reach the far right without dying. To make it easier for the agent, a reward will be given after each action is performed. A reward scheme similar to that in Cart Pole, i.e. only a reward once the goal is reached, would be near impossible in this game. The agent should therefore be rewarded when it walks to the right, and punished if it goes the wrong way or gets hurt by enemies. The performance of the agent will depend greatly on the reward function. It is possible to make a simple reward function where distance is the only variable, or a specialised multi variable function.

Afterglow might be useful in Mario, since a sequence of actions are needed to clear a gap or jump over a block. But it depends on the reward scheme. In a simple reward scheme Mario will not necessary get a reward for doing the first steps necessary in a sequence. If the reward scheme rewards perfectly, no afterglow would be needed. To jump over a tall block, at least 5 sequential actions are needed : $JUMP * 3$ and $RIGHT * 2$. And, in a simple rewrard scheme, the $JUMP$ actions might not get a reward since Mario is not moving to the right.

One problem discovered whilst training was that the PS agents find loopholes in the reward function. With a naive reward function the agent discovered that going one step forward and then one back was a good enough solution, since the forward step gave a positive reward which was shared with the previous clip.

Multiple reward functions were tested, all leading to different agents (fig-34). A reward function that punishes too hard will make the agent "scared" and it will stop playing. A too low punishment for colliding with enemies will lead to Mario ignoring enemies and charging to death.

The reward function tested was based around these variables:

$$reward = distance_{relative} * a + elevation_{relative} * b - gotHurt * c - isStuck * d,$$

where a, b, c, d are numbers for scaling.

In this game PS will receive negative rewards. From figure 36 we see a (yellow) PS agent that only takes $reward >= 0$ positive rewards, which does not work well. Damping could help, but then the agent requires more internal computation and slower learning.

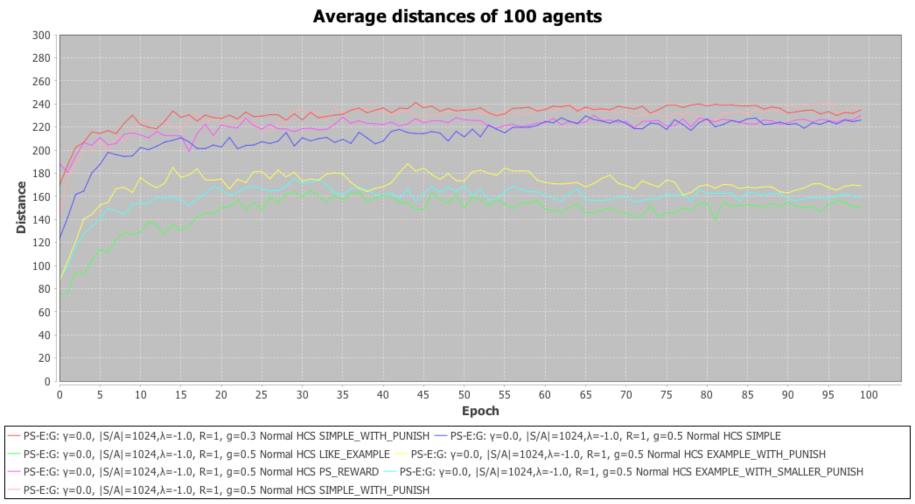


Figure 34: Mario: Difference of reward functions. The yellow and green are the functions with most punishments. The results show that there is too much negative reward. Moreover, the "nicer" functions (red and pink) perform better.

The *PS_REWARD* function is the most simple: a collision, stuck or walking left returns -1 and otherwise 1. From figure 34 we see that this rule makes the agent perform well, being the second best. The other rule follows the formula with different scaling on the parameters. The best function is the *SIMPLE_WITH_PUNISH* which follows the formulae above, with a small punishment. Details of these functions can be found in the source code [66].

In figure 35 the same functions are tested on multiple levels. We see that the same functions as previously perform the best (yellow *PS_REWARD*, cyan *SIMPLE_WITH_PUNISH* and red *SIMPLE*.

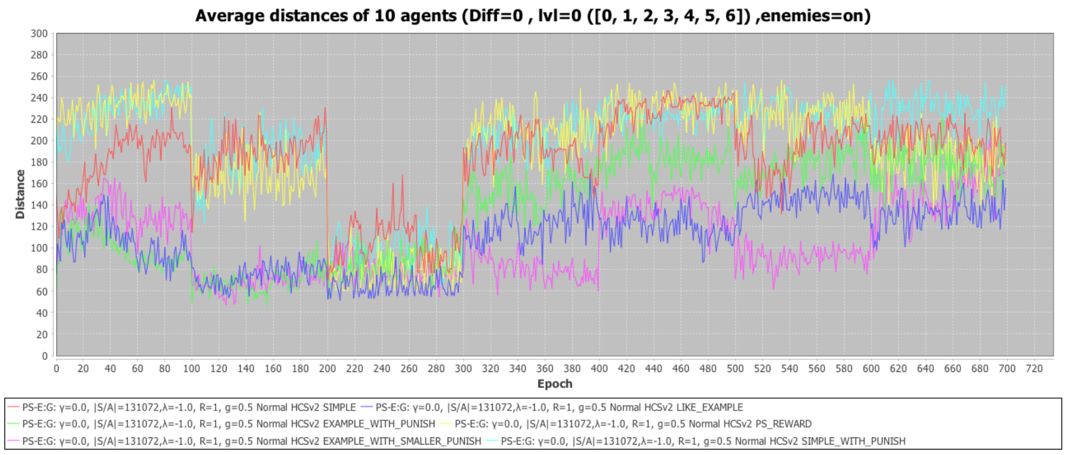


Figure 35: Mario: Agents trained with different reward methods. Trained 100 games on each level. On level 2 (200-300), we see that all agents struggle, due to a dead-end where the agent needs to go back and then jump up to the layer above. The most successful are yellow agents (*PS_REWARD*)

Overall the simple reward scheme works well compared to the more complicated schemes. One could think that large punishments on death would be useful, but it seems to create more uncertainty in the successful agent. Moreover, the goal of this paper is not to create the best Mario agent, as this means doing a lot of preprocessing between PS and the environment. Instead, the goal is to test if PS can be situated in any environment and solve the task.

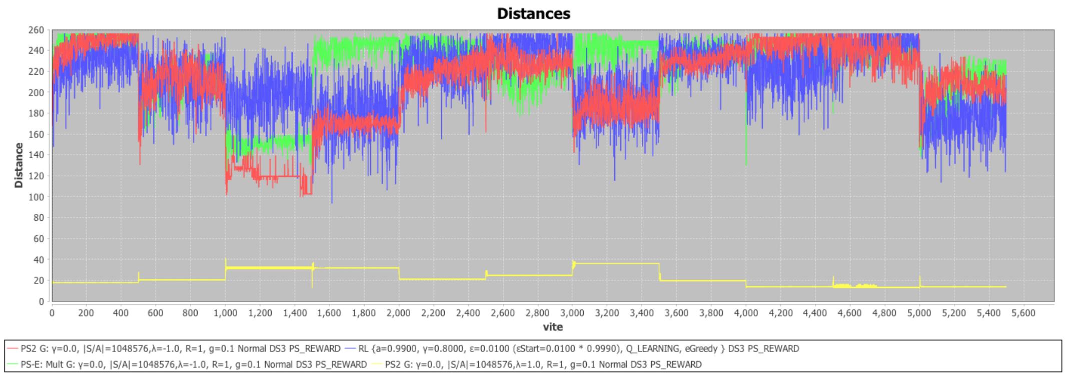


Figure 36: Mario: The yellow agent follows the PS model strictly, by not allowing negative rewards. We see that does not work well in this game. The other ones receive a negative reward $r = -1$ when dying, stuck or going the wrong way.

5.3 Finding optimal parameters

5.3.1 Damping

From figure 37 we see that damping is not really necessary, though it works with a high afterglow $\eta \approx 1$. The rules used in the official competitions put a limit on the allowed internal computation time for the agent. Due to the increase of computation time for damping and that we allow for negative rewards, damping is usually turned off $\gamma = 0$ on this game. Only using damping and a zero-reward would require the agent to die a lot to learn anything, whilst a huge negative reward would quickly make the agent forget that action.

5.3.2 Afterglow

The agents will receive a reward at every step, which partly describes how good that previous action was. In that sense no afterglow would be needed. But since the reward functions used do not reward sequences where the first steps are none-rewarded, some afterglow could be useful. The reward function, *PS_REWARD* only rewards movement to the right, such that the first jumping action required to clear a block would not be rewarded without afterglow. From the figure 37 and 39 the agent manages to play well with different values for the afterglow-damping parameter η.

Reflection will generally make the agent learn and adapt quicker. From figure 4 the results of using reflection are shown. The agent does not benefit by using reflection. In Mario the agent will observe the same states millions of times in long experiments, which means that by then it can be assumed that a good hopping probability is obtained, and the the effect of using short time memory from emotions disappears.

Agent	**Average distance**	**Note**
PS-SoftMax	218	
PS-Standard	220	
PS-Standard	208	with reflection ($R = 20$)

Table 4: Mario PS : Not a big difference with different hopping probability functions, and emotions do not seem to benefit the agent. On level 0-10 with enemies, 50 games per level and average of 30 agents.

4 shows that the standard hopping probability function performs as good or better than SoftMax. That can mean that multiple actions are roughly equally as good in a situation, and therefore do not play a big part. An agent that explores more might have a higher chance of success. Another reason might be that the reward or state abstraction is not optimal, due to the fact that the random hopping probability function performs better.

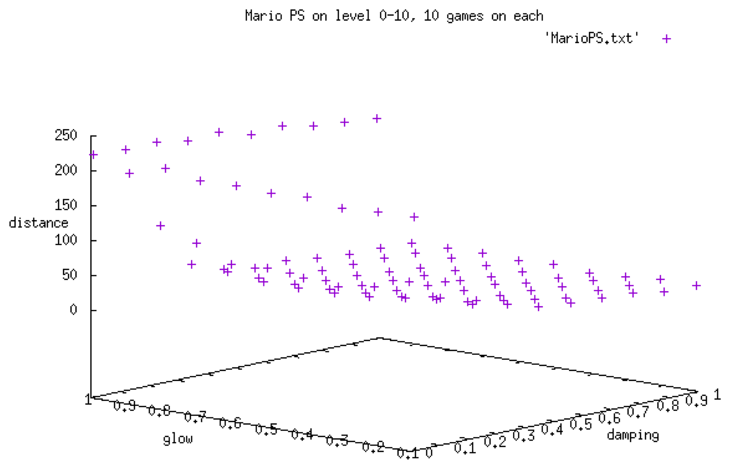


Figure 37: Mario 3D parameters: Without enemies multiple different PS agents will win the game easily. The plot also shows that damping can be turned off.

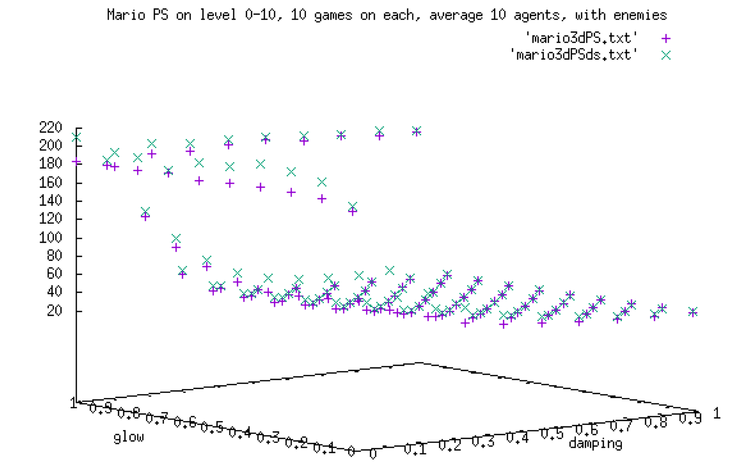


Figure 38: Mario 3D : With enemies. Using different state representations (green is DynamicState (2^16 states) and purple Dynamic State 3 (2^20 states)). The simple state representation seems to be marginally better, with the best agent being $\gamma = 0.0$, $\eta = 0.339$ with 215 steps.

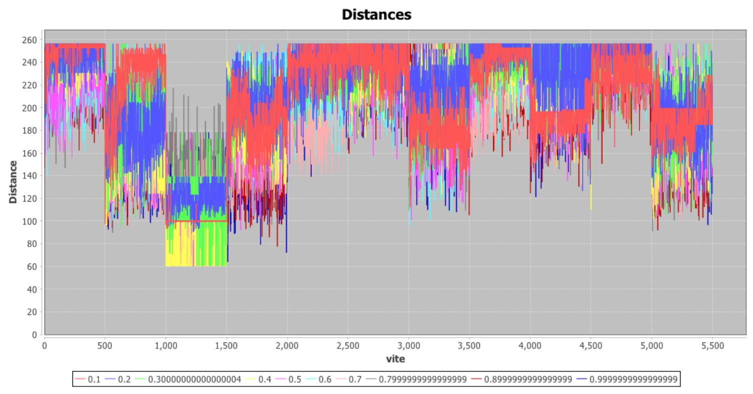


Figure 39: Mario PS afterglow parameter: Shows the average of 10 PS agents with different afterglow. In general there is not a huge difference, but the lower η seems to perform marginally better. With the best being $\eta = 0.3$ with 214 steps, and worst being $\eta = 0.9$ with 195 steps.

Reinforcement parameters

For RL we have 4 choices of algorithms, Q-Learning and SARSA, without eligibility traces. The same discussion about rewarding previous Q's arises for RL as with PS. Since it was of little importance for PS, and that Q-Learning and SARSA can learn sequences without traces, so the simple versions will be enough for this game. Overall the choice between SARSA and Q-Learning plays only a small role since they converge to the same results (as seen in figure 40).

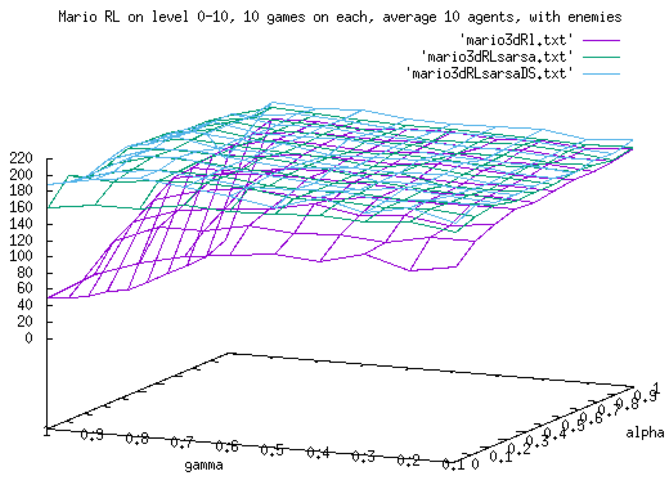


Figure 40: Mario Parameters for RL: Purple is Q-Learning and green is SARSA with the DS3 state representation, blue is SARSA with DS. The best agent with DS3 is $\alpha = 0.66$ and $\gamma = 0.9$ with 200 steps. The same agent with DS state manages 208 steps. This is not a huge difference, but somewhat disappointing that the large state space is not helping. In general all parameters except the extreme $\alpha \approx 0$ work well (obvious since $\alpha = 0$ means learning rate is off, no learning)

5.4 Action Composition

The PS model allows for two action clips to compose new actions (3.8.2), which potentially could benefit the agent. As stated above: Mario actions are a boolean array $\{UP, DOWN, LEFT, RIGHT, JUMP, SPEED\}$.

Here is an example of a possible action composition:

Action name	**Array**	**Operation**
RIGHT	$\{0,0,0,1,0,0\}$	
JUMP	$\{0,0,0,0,1,0\}$	
Can create two new action clips :		
RIGHT-JUMP	$\{0,0,0,1,1,0\}$	XOR
NOTHING	$\{0,0,0,0,0,0\}$	AND

The conditions for this to happen are explained in 3.8.2. The first point in the conditions are that the clips wanting to spawn new clips should be sufficiently rewarded. In practice this rule was used:

1. Both actions must have a hopping probability over a threshold.
   $P(c, a_1), P(c, a_2) >= h_{threshold}\ , \quad 0 <= h_{threshold} <= 0.5$

This threshold reduces the chance of starting the process by only starting when both action are very promising, e.g both have a hopping probability higher than 30%. In practice we see that the agent learns all important actions in a decent time.

Mario is given the base actions $LEFT, RIGHT, JUMP, FIRE$ at the start. After a few games, the actions $JUMP - FIRE, RIGHT - JUMP - FIRE, RIGHT - JUMP$ and $RIGHT - FIRE$ are learned. Other actions like $LEFT - JUMP$ and $LEFT - JUMP - FIRE$ may be learned, but it depends on the agent.

There is a "delay" in the game. If Mario is sprinting to the right, and then presses the left button, Mario will still slide to the right and receive positive rewards, so some positive rewards for LEFT-* actions are therefore possible. Some agents will learn unwanted actions like $LEFT - RIGHT, DO - NOTHING$ and other button smashing sequences (holding all the buttons in at the same time). This does not affect the agent negatively since these actions will be punished since they lead nowhere. In other tasks this might create a storage problem in ECM, but that can easily be countered by removing badly rewarded composed clips.

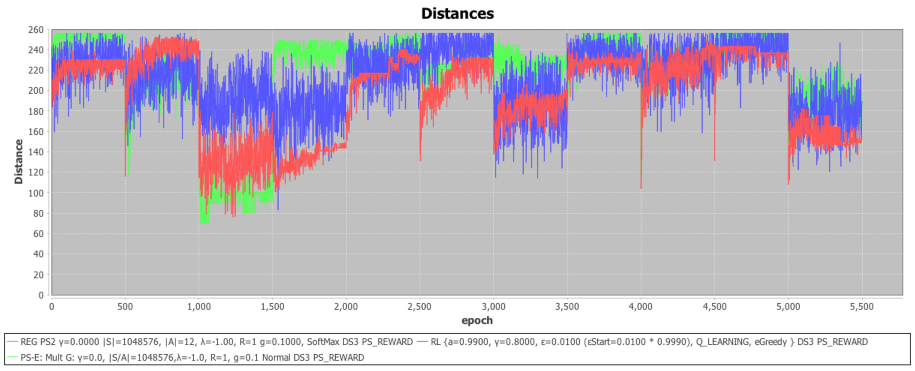


Figure 41: Distance reached for different agents. 500 games with enemies on each level 0 to 10. Average of 10 agents.

In 41 the best performing agents are compared. Here is the total average distance achieved over 5000 games:

Agent	**Average distance**	**Description**
PS	195	Regular PS.
PS-E	220	PS with action composition
RL	214	Q-Learning

41 shows that action composition can benefit the agent. One reason might be that bad (not always) actions like *left-** are never learned. Moreover if *right* and *jump* are rewarded highly, then the creation of *right-jump* starts off with a high reward, and for the next experience of that percept, there is a high chance that the new action is selected, whilst a normal PS agent has more chance of selecting a not so good action.

We have seen that the agent will learn a level better each time it plays. It is also desirable to have a agent that can remember previous levels, even after playing other levels. In figure 42 5 random levels are generated and the agents play 50 games on each. At time $t = 2500$, the same 5 levels are played again. The results shows that the agent ran 6% longer on the second part, meaning the agent remembers and keeps improving the performance.

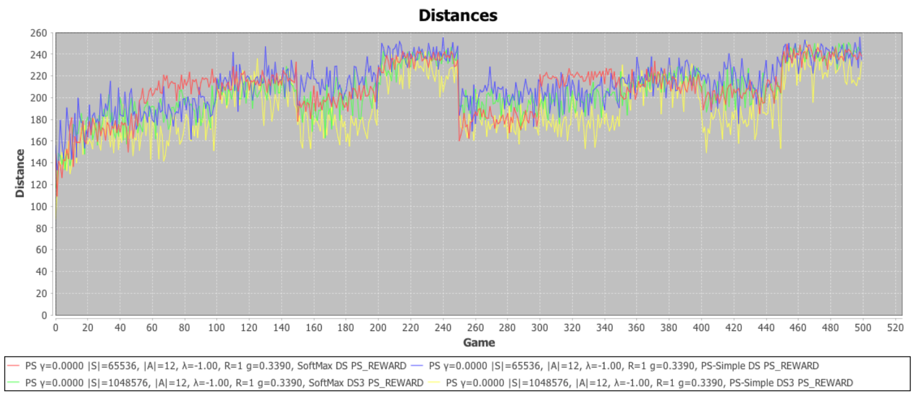


Figure 42: Mario: 5 random levels, played 50 time, then repeat the whole experiment. At time $t = 2500$, the same levels are played once more.

5.5 Generalisation

Following the idea of generalisation from 3.8.3, we apply it to this game. The idea exploits that states often are represented as vectors. A new state abstraction method was used for enabling this function, $s = \{dDistance, canJump, dist_{obstacle}, height_{obstacle}, dist_{enemy}\}$, where the category sizes are $\{3, 2, 4, 4, 4\}$ respectively, being a total of 384 possible percepts.

With 5 categories, the network will develop to a 7 layer network, where the first layer is the percepts and the last is the actions. The PS model starts off with only knowledge of the actions possible, and through perceiving percepts and new clips, both percept and fictional wildcard, will be added.

As an example, after 10 games on one level the number of clips in each layer typically looks like this: 103 percepts, 151, 136, 65, 14,1, 12 actions. The network that started off as a 2 layer network, with only direct percept to action sequences, has now grown to a 7 layer network with sequences containing 7 clips.

Layer 1 will contain clips with 1 wildcard, which means it will ignore one category, as an example the dDistance (distance since last time step) might not be relevant for the agent. In layer 2 the clips will ignore 2 categories, and so on. In layer 4 there is a clip that only cares about the distance to the enemy, and ignores the other categories. This kind of abstraction, letting the network itself find what features are important, might help in Mario. In Mario there are so many variables, and it is difficult to know exactly what parameters are needed in a state representation.

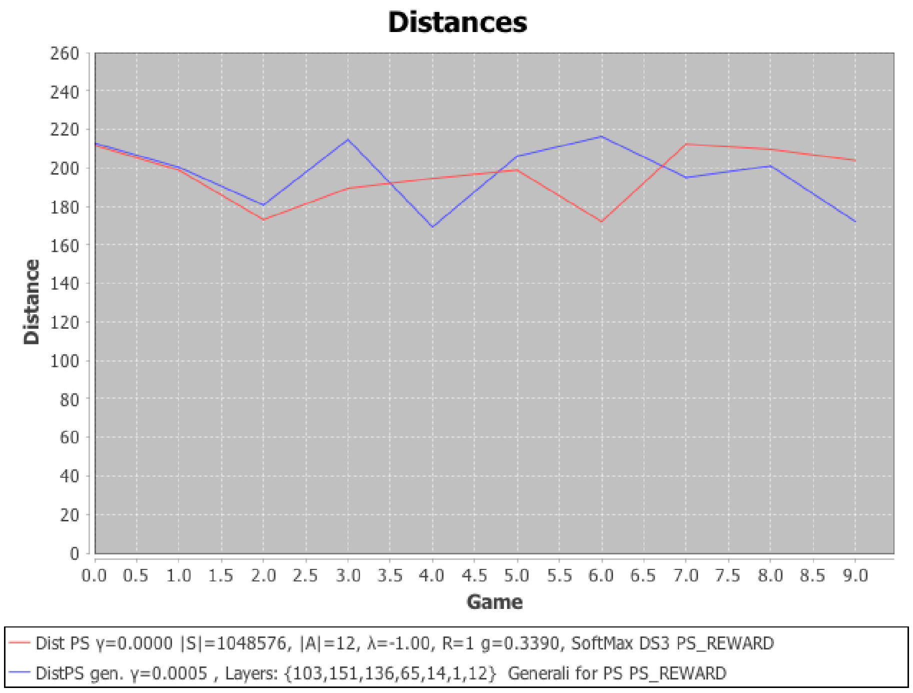


Figure 43: Distance comparisons. The regular PS uses DS, 2^{20}, state representation. A PS with generalisation uses a simpler state model with 384 possible states. We see that the generalising agent performs slightly better.

After 10 games the network has evolved to roughly 500 clips and over 10^5 edges. The PS network has now developed to a massive "black box", visualizing it would be near impossible.

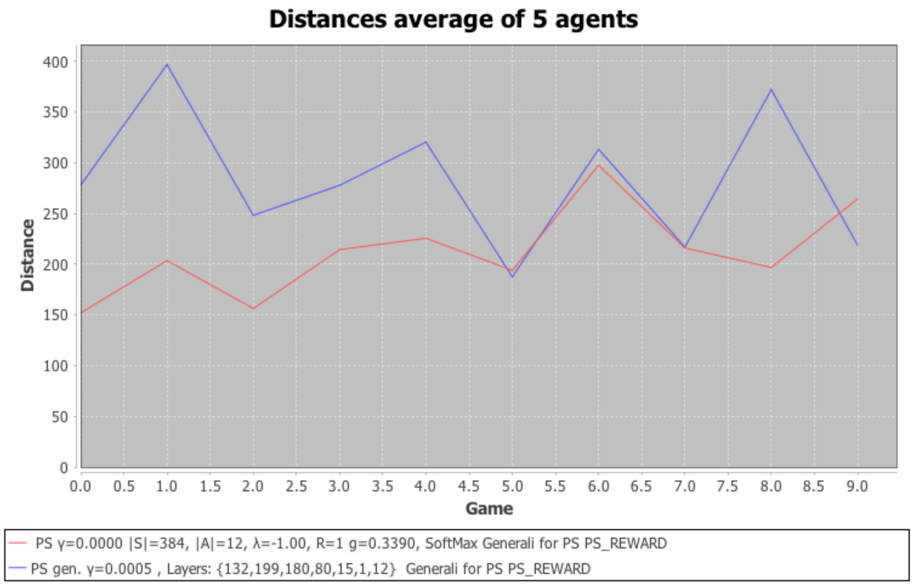


Figure 44: Distance comparison. In this comparison both use the same state abstraction, with a total of 384 possible percepts. The results show the (blue) agent benefits from the generalisation. The length of the level is increased to 400 to see how far they reach.

6 Quantum Speed-up

A critical problem with machine learning is the computational time and memory limit on computers. Adding a new feature to a state space will exponentially increase the state space, which leads to a limited set of possible states.

Quantum information computing is a rapidly developing field. Results have shown that some mathematical problems can receive a significant speed-up from running on a quantum computer. Problems that are difficult or impossible to solve on a classical computer could be solved efficiently on a quantum computer.

Two important quantum algorithms are *Shor's* algorithm [53] and *Grover's* algorithm [21].

Shor's algorithm for factoring integers runs in polynomial time, compared to the best-known classical algorithm which runs in sup-exponential time, therefore the quantum algorithm has an exponential advantage over the classical. The algorithm has been realised in practice by factoring integer 15 using nuclear magnetic resonance (NMR) [60] and, more recently, the number 56153 has been factored, in theory, using only 4 qubits [15], and based on a previous practical factoring of 143 [64].

Grover's algorithm can find an element in a unsorted "database" with n elements in $O(\sqrt{n})$ time, where a classical algorithm requires $O(n)$ steps.

Realizing a quantum artificial agent is a relative new idea. As shown before, an agent needs three things to function : percepts, actions and rewards. The environment for an agent will in general always be classical, in the sense that the percepts received, and the actions undertaken are classical (note however that this does not preclude the possibility of quantum percepts and actions in some more general system). This means that the quantum speed-up can only come from the internal decision making.

An important point is to know that a quantum agent performs at least as well as a classical agent. In [46] it is proven that the output distribution $P^{(t)}(a|s)$ is equal for classical and quantum reflecting agents.

Reflecting PS - RPS

In [46] a Reflecting PS (RPS) agent is shown to obtain a quadratic speed-up for the internal process. This is because the action selection is realised using the π distribution process with reflection, and by using a Grover-like procedure [58].

$$\pi_s(i) = \begin{cases} \frac{\pi_s(i)}{\sum_{j\in f(x)} \pi_s(j)} & \text{if i } \in f(s). \\ 0 & \text{otherwise.} \end{cases} \tag{20}$$

where $f(s)$ is the available actions from state s. When an action gets negative feedback, it can be removed from the set $f(s)$. If every action is removed, the set gets reset with all actions.

In a classical RPS, the excepted number of iterations to find a positive edge are:

$$t^c_{check} \in \text{O}(\frac{1}{\epsilon_s}), \qquad \epsilon_s = \sum_{i\in f(s)} \pi(i)$$

For a quantum RPS the same problem can be solved in [46]:

$$t^{q}_{check} \in_R \in [0, \mathrm{O}(\frac{1}{\sqrt{\epsilon_s}})]$$

The search for finding a positive action can be made quadratically faster by exploiting the quantum ability to search many paths at the same time.

We observe that there is also some research being done on quantum Q-learning [16].

7 Conclusion

In this paper we have studied the model and performance of projective simulation in multiple games. PS is a simple model that can be extended with many features. Without the features such as clip association and generalisation, PS is comparable to Q-learning and SARSA in complexity.

The games Invasion Game, Grid World and Mountain Car were demonstrated in previous papers. We performed the same tasks to both validate the framework built for this thesis as well as confirming their results. Part of the results were done with the same conditions which showed the correctness. We also extended the tasks and compared them to popular RL algorithms.

The Invasion Game was used to demonstrate the features of PS, where the ability to adapt to changes in the environment was tested. A special feature of PS is the ability to grow the network of clips while training, either by experiencing new percepts or creating fictional clips. We saw that associating clips to other clips could benefit the agent, a clip with no preference could connect itself to a clip with stronger preferences to a real action. Generalisation also showed promise, with its ability to dynamical create new layers of clips to generalize and categorize percepts.

In the GridWorld task PS was faster than RL at finding a good path through the maze, but in the end the RL algorithms caught up by finding the optimal path. This is not always the case for PS, since it has less ability to look forward when updating edges.

For Mountain Car the PS agent performed better than any RL algorithm, which was a bit surprising. It might be due to the states not being fully MDP's 1.

The cart Pole game can be seen as an extension to Mountain Car, where an agent has two more discrete variables in the state space. The agent does not know what game it is situated in, but will just act on the percepts it sees and return an action according to the rewards given before.

Infinite Mario Bros (IMB) was used in this thesis to test PS and RL on a game made for humans. The action composition for PS was used in this game, which is somewhat how humans could play the game. The agent started with some basic buttons it knows, then it learns new moves by combining highly rewarded moves. The results showed that this way of learning benefits the agent.

The Mario game demonstrated that learning agents (e.g RL, PS, genetic algorithms) cannot perform as well as deterministic algorithms. This is obvious, if the rules of the task can be hard-coded there is no need for learning. But there are many tasks where the policy and world dynamics is unknown in advance, and then it is not possible to have a game engine embodied.

In all the games we saw that the PS model is competitive with RL algorithms. In experiments that are fully MDP 2.1, SARSA and Q-Learning will find the optimal policy, while PS has no guarantee of that. This means that, in a well defined world, the RL algorithms will perform better. But describing a complex environment as an MDP is not always possible, either by some fault in the design (e.g Mario) or because the rules in the world aresimply unknown and changing.

While doing the experiments, some limitations of RL and PS were shown. The main problem is the exponential increase in state space in high-dimensional tasks, where storing it in matrices is infeasible. In Mountain Car, Cart Pole and Mario there was a need to discretise the state space, as it is not possible to store all the individual states. This has been a problem in RL since the beginning, and there have been multiple approaches for dealing with it [57, 31, 62, 50, 2]. A promising approach is to combine neutral networks with RL, see relevant work section 7.3.1.

The experiments undertaken have shown that the PS model is more or less comparable to RL algorithms. The RL algorithms are proven to find the optimal policy if the environment is a MDP, whilst PS is not guaranteed to find such an optimum. However, whilst PS might not find the best solution, it usually finds a good one. Whilst the PS papers never called PS a reinforcement learner, it is in fact in that class as it acts by reinforcement on the environment.

7.1 Comparison between PS and RL

Emotion

Emotion is shown to be very useful in the Invasion Game. We saw that in situations where the world is changed, emotion acts as a short time memory, helping the agent to overcome the change. If the agent 'feels' bad, after multiple punishments for seemingly good actions, the emotion will take more control, and most actions performed are realised by the emotion, and not by the standard action selection method. As soon as the agent 'feels' positive, the action selection will be undertaken by hopping probability again.

In RL there has also been some research on mapping human-like emotion into RL [25, 24, 10], where each state has a desirability, similar to the PS model. More types of emotions are used: joy, hope and fear and each state has a desirability. This is also allowed in the PS model.

Initialisation

The weights on the edges in PS are all initialised to $h = 1$, while RL typically uses a random initialisation on the $Q(s, a)$ values.

Action Selection

Both RL and PS are free to use any action selection function. As seen the standard choices are ϵ-Greedy, SoftMax and the simple probability function. The difference is how strictly the function follows the policy compared to its ability to explore. The PS model normally uses the simple probability function due its more "free" behaviour, whilst RL aims for an optimal solution.

Glow and Traces

Both models can be extended with functionality for rewarding previous actions, which is useful in experiments where the rewards are delayed. In PS afterglow is used whilst RL uses eligibility traces. Overall the two approaches are very similar.

Learning and planning The PS approach to learning is very novel. The recently used clips will receive parts of the reward, which increase the edge strengths. Q-Learning and SARSA update the Q values with comparisons with how good the next state, that this action moved it to, is. It is possible to say that RL 'thinks' forward in the update rule, whilst PS does not with the current update rule. The RL update will propagate backwards, and converge on environments with delayed reward even without eligibility traces.

Overall the RL update rule is better performance wise compared to PS. A possible improvement 7.2.4 ?

This is often seen in practice in the experiments done, where PS finds a decent solution very fast and keeps doing the same thing without considering that there might be a better path.

7.1.1 Complexity comparison

An interface for a learning agent only requires two methods: *getAction(state)* and *giveReward(reward)*. The complexity, running time, for PS with different features and RL are summarized in the table below (5).

Agent	**getAction**	**giveReward**	**Note**
Q-Learning, SARSA	$O(\|A\|)$	$O(1)$	$\|A\|$ is the number of actions.
Q(λ), SARSA(λ)	$O(\|A\|)$	$O(t)$	t is the length of eligibility traces.
PS	$O(\|A\|))$	$O(1)$	
- with emotion	$O(\|A\| * R)$		R is the maximum number of reflections allowed.
- with glow		$O(t)$	t is the number of excited clips
- with damping		$O(\|E\|)$	$\|E\|$ is the number of edges in the network.
- with damping-v2		$O(1)$	*

Table 5: Computation time comparison. * a "hack" where each edge remembers when it was last used. The updating will not be identical to the real update rule, but it will be similar and worth the massive speed-up it gives.

7.2 Open questions and future work

7.2.1 Quantum speed-up

A quantum agent is promising. Whilst the paper [46] shows a quantum speed-up in the reflection (action selection) on a RPS, that is only one out of the two main components in PS. The learning part has already been mentioned, but to really benefit from a quantum computer, one may query whether the updating of edges should also be faster, as this is the part that is computational heavy?

7.2.2 High dimensionality in PS

As stated often, the biggest limit to RL is the exponential growing state space by adding another dimension to the data. This is often referred to the "curse of dimensionality" [5], and occurs in multiple fields.

For reducing the state space function approximation 5, neural networks , decision trees can be used. There is no best solution to the problem, every technique might lose valuable information. Function approximation is the most simple and novel, but the method has to be specialised for each task. Neural networks and decision trees require more storage and computation time.

For a quantum computer we have potential access to an exponentially large space. This is one of the main motivations for a quantum learning agent. If not some additional structures will have to be placed in front of or embodied in the PS model. Deep neural networks have shown to be especially good at pulling out features from images (see 7.3.1).

7.2.3 General AI

All artificial agents today are specialised on solving one problem. In multiple domains the programs can outperform humans, but if you ask any on these programs to do something they were not trained for, it has no chance of learning it. The ultimate goal of AI is to develop a "general-AI", where an agent that masters one domain could abstract that knowledge into a library structure which can be applied to a new domain. Nobody has been able to do that yet.

One problem with a general AI situated in the real world is the reward scheme. As seen in the experiments above, a reward function needs to be defined. In real life situations it is not that easy (humans look at emotions of other people) .

7.2.4 PS optimal solution

The reason Q-Learning and SARSA will find the optimal solution (and work without eligibility traces) is the update rule, where the update of the current $Q(s, a)$ depends on the next $Q(s_{t+1}, a_{t+1})$. If the next $Q(s_{t+1}, a_{t+1})$ is badly or well rewarded, then this will influence the $Q(s, a)$, meaning that a bad or good valued-state further ahead will propagate backwards.

This mechanism is not implemented in PS, afterglow is therefore always used in experiments with delayed rewards. An idea is to use a similar update in PS:

$$h^{t+1}(c_i, c_j) = h^t(c_i, c_j) + \gamma(h^t(c_{t+1}, c_{t+1}) - h^t(c_i, c_j) - 1) + \lambda \tag{21}$$

7.3 Relevant research

7.3.1 Deep learning combined with RL

Promising research is being done by combining neural networks and reinforcement learning (could also be combined with PS). Deep learning, neural networks with multiple non-linear layers, are showing promise in dealing with high-dimensional tasks. Convolutional Neural Networks (CNN) are specialised deep networks designed for extracting features from images. CNN makes it possible to take a huge input space (e.g a live camera feed) as input and output lower dimensional feature states. These states can in turn be used for reinforcement learning (some frameworks can be found in [30, 14]).

This approach is used for DQN to play multiple Atari games [61], where it performed better than humans in most games. The CNN receives pixel images from the game and reduce it to vector states that can be used for Q-Learning. One limitation is that the reward function had to be given to the agent separate from the game input. This leads to a more generalized reinforcement learning, since less pre computation of the state space is needed. But it is still far from "general" AI, since there the knowledge from one task cannot be transferred to another game. The network had to be restarted for each game, and the training for a game could take multiple days.

A similar approach is done with a physical robot that can learn new tasks [32]. It is done with a similar internal structure to neural network and RL, but the learning is done by showing the robot how to the task beforehand.

7.3.2 The brain

One eventual goal for machine learning is to relate it to the brain. The brain appears to learn in a similar way to reinforcement learning [43], at least in part. Experiments on animals show that they can learn an arbitrary policy to obtain rewards (and avoid punishments).

The brain accepts electrical signals (percepts), without caring where or how it obtained them, and then tries to make sense and act upon them. For example, David Eagleman developed a vest, V.E.S.T (versatile extra-sensory transducer), that converts any perceptions (sound, noise, image) into vibrations on the back of a person [17], the brain will then try to understand what the different signals it recives are. The invention have showed that deaf people can hear through the vest. Sending electrical signals to the brain is not new. We have had cochlear implants for a long time now. V.E.S.T enables any data to be sent and translated into vibrational inputs, and then the brain will try to interpret the signals.

References

[1] Marioai source code, 2012. URL http://www.marioai.org/.

[2] Leemon C. Baird and Harry Klopf. Reinforcement learning with high-dimensional, continuous actions, 1993.

[3] A.G. Barto, R.S. Sutton, and C.W. Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *Systems, Man and Cybernetics, IEEE Transactions on*, SMC-13(5):834–846, Sept 1983. ISSN 0018-9472. doi: 10.1109/TSMC.1983.6313077.

[4] Robin Baumgarten. Infinite mario ai - long level, 2009. URL https://www.youtube.com/watch?v=DlkMs4ZHHr8.

[5] R. Bellman and Rand Corporation. *Dynamic Programming*. Rand Corporation research study. Princeton University Press, 1957. ISBN 9780691079516. URL https://books.google.it/books?id=wdtoPwAACAAJ.

[6] Richard Bellman. A markovian decision process. *Indiana Univ. Math. J.*, 6:679–684, 1957. ISSN 0022-2518.

[7] S. Bojarski and C.B. Congdon. Realm: A rule-based evolutionary computation agent that learns to play mario. In *Computational Intelligence and Games (CIG), 2010 IEEE Symposium on*, pages 83–90, Aug 2010. doi: 10.1109/ITW.2010.5593367.

[8] H. J. Briegel and G. de Las Cuevas. Projective simulation for artificial intelligence. *Scientific Reports*, 2:400, May 2012. doi: 10.1038/srep00400.

[9] Hans J. Briegel. On creative machines and the physical origins of freedom. *Sci. Rep.*, 2, 07 2012. URL http://dx.doi.org/10.1038/srep00522.

[10] Joost Broekens, Walter A. Kosters, and Fons J. Verbeek. Affect, anticipation, and adaptation: Affect-controlled selection of anticipatory simulation in artificial adaptive agents. *Adaptive Behavior - Animals, Animats, Software Agents, Robots, Adaptive Systems*, 15(4):397–422, December 2007. ISSN 1059-7123. doi: 10.1177/1059712307084686. URL http://dx.doi.org/10.1177/1059712307084686.

[11] Jason Brownlee. The pole balancing problem a benchmark control theory problem, 2004. URL `http://en.wikipedia.org/wiki/Inverted_pendulum`.

[12] Star Apple B.V. Ai games - collection of ai games, 2013. URL `http://theaigames.com/`.

[13] L.A. Celiberto, J.P. Matsuura, R. Lopez de Mantaras, and R.A.C. Bianchi. Using transfer learning to speed-up reinforcement learning: A cased-based approach. In *Robotics Symposium and Intelligent Robotic Meeting (LARS), 2010 Latin American*, pages 55–60, Oct 2010. doi: 10.1109/LARS.2010.24.

[14] Francois Chollet. Keras - deep learning library, 2015. URL `http://keras.io/`.

[15] N. S. Dattani and N. Bryans. Quantum factorization of 56153 with only 4 qubits. *ArXiv e-prints*, November 2014.

[16] Daoyi Dong, Chunlin Chen, Hanxiong Li, and T. Tarn. Quantum reinforcement learning. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 38 (5):1207–1220, Oct 2008. ISSN 1083-4419. doi: 10.1109/TSMCB.2008.925743.

[17] David Eagleman. Sensory substitution, 2015. URL `http://www.eagleman.com/research/sensory-substitution`.

[18] S. Geva and J. Sitte. A cartpole experiment benchmark for trainable controllers. *Control Systems, IEEE*, 13(5):40–51, Oct 1993. ISSN 1066-033X. doi: 10.1109/37.236324.

[19] Linda S. Gottfredson. Mainstream science on intelligence, 1994. URL `http://www.udel.edu/educ/gottfredson/reprints/1997mainstream.pdf`.

[20] A. Simon Grant. Modelling cognitive aspects of complex control tasks, 1990. URL `http://www.simongrant.org/pubs/thesis/`.

[21] Lov K. Grover. A Fast quantum mechanical algorithm for database search. 1996.

[22] H. Handa. Dimensionality reduction of scene and enemy information in mario. In *Evolutionary Computation (CEC), 2011 IEEE Congress on*, pages 1515–1520, June 2011. doi: 10.1109/CEC.2011.5949795.

[23] P.E. Hart, N.J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on*, 4(2): 100–107, July 1968. ISSN 0536-1567. doi: 10.1109/TSSC.1968.300136.

[24] Eric Hogewoning, Joost Broekens, Jeroen Eggermont, and Ernst G. Bovenkamp. Strategies for affect-controlled action-selection in soar-rl. In *Proceedings of the 2Nd International Work-conference on Nature Inspired Problem-Solving Methods in Knowledge Engineering: Interplay Between Natural and Artificial Computation, Part II*, IWINAC '07, pages 501–510, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 978-3-540-73054-5. doi: 10.1007/978-3-540-73055-2_52. URL http://dx.doi.org/10.1007/978-3-540-73055-2_52.

[25] E. J. Jacobs. Mapping elements of reinforcement learning to human emotions. 2013. URL http://repository.tudelft.nl/view/ir/uuid:09fe859a-db3a-4157-b341-093820998e08/.

[26] XiaoYu (Gary) Ge Peng Zhang Jochen Renz, Stephen Gould. Ai birds - angry birds ai compeition, 2014. URL https://aibirds.org/.

[27] Sergey Karakovskiy Julian Togelius and Robin Baumgarten. The 2009 mario ai competition, 2010. URL http://julian.togelius.com/Togelius2010The.pdf.

[28] Sergey Karakovskiy Julian Togelius, Noor Shaker and Georgios N. Yannakakis. The mario ai championship 2009–2012, 2013. URL http://julian.togelius.com/Togelius2013The.pdf.

[29] S. Karakovskiy and J. Togelius. The mario ai benchmark and competitions. *Computational Intelligence and AI in Games, IEEE Transactions on*, 4(1):55–67, March 2012. ISSN 1943-068X. doi: 10.1109/TCIAIG.2012.2188528.

[30] Andrej Karpathy. Convnetjs - deep learning in your browser, 2015. URL http://cs.stanford.edu/people/karpathy/convnetjs/.

[31] T. Kobayashi, T. Shibuya, and M. Morita. Q-learning in continuous state-action space with redundant dimensions by using a selective desensitization neural network. In *Soft Computing and Intelligent Systems (SCIS), 2014 Joint 7th International Conference on and Advanced Intelligent Systems (ISIS), 15th International Symposium on*, pages 801–806, Dec 2014. doi: 10.1109/SCIS-ISIS.2014.7044714.

[32] S. Levine, C. Finn, T. Darrell, and P. Abbeel. End-to-End Training of Deep Visuomotor Policies. *ArXiv e-prints*, April 2015.

[33] D. Loiacono, L. Cardamone, and P. L. Lanzi. Simulated Car Racing Championship: Competition Software Manual. *ArXiv e-prints*, April 2013.

[34] Simon M. Lucas. Computational intelligence and ai in games: A new ieee transactions, 2009. URL `http://csee.essex.ac.uk/staff/lucas/tciaig/editorial.pdf`.

[35] Jurgen Schmidhuber Marco Wiering. Performance comparison of sarsa(λ) and watkin's(λ) algorithms. URL `http://www.karanmg.net/Computers/reinforcementLearning/finalProject/KaranComparisonOfSarsaWatkins.pdf`.

[36] Jurgen Schmidhuber Marco Wiering. Fast online q(λ), 1998. URL `http://www.ai.rug.nl/~mwiering/GROUP/ARTICLES/fast_q.pdf`.

[37] J. Mautner, A. Makmal, D. Manzano, M. Tiersch, and H. J. Briegel. Projective simulation for classical learning agents: a comprehensive investigation. *ArXiv e-prints*, May 2013.

[38] Julian Mautner, Adi Makmal, Daniel Manzano, Markus Tiersch, and HansJ. Briegel. Projective simulation for classical learning agents: A comprehensive investigation. *New Generation Computing*, 33(1):69–114, 2015. ISSN 0288-3635. doi: 10.1007/s00354-015-0102-0. URL `http://dx.doi.org/10.1007/s00354-015-0102-0`.

[39] A. A. Melnikov, A. Makmal, V. Dunjko, and H. J. Briegel. Projective simulation with generalization. *ArXiv e-prints*, April 2015.

[40] Alexey A. Melnikov, Adi Makmal, and Hans-J. Briegel. Projective simulation applied to the grid-world and the mountain-car problem. *CoRR*, abs/1405.5459, 2014. URL `http://arxiv.org/abs/1405.5459`.

[41] Shiwali Mohan. Reinforcement learning in infinite mario. 2012. URL `http://www.shiwali.me/content/prelim-paper.pdf`.

[42] Remi Munos and Andrew Moore. Variable resolution discretization for high-accuracy solutions of optimal control problems. In *Proceedings of the 16th*

International Joint Conference on Artificial Intelligence - Volume 2, IJCAI'99, pages 1348–1355, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc. URL `http://dl.acm.org/citation.cfm?id=1624312.1624410`.

[43] Yael Niv. Reinforcement learning in the brain. *Journal of Mathematical Psychology*, 53 (3):139–154, 2009.

[44] S. Ontanon, G. Synnaeve, A. Uriarte, F. Richoux, D. Churchill, and M. Preuss. A survey of real-time strategy game ai research and competition in starcraft. *Computational Intelligence and AI in Games, IEEE Transactions on*, 5(4):293–311, Dec 2013. ISSN 1943-068X. doi: 10.1109/TCIAIG.2013.2286295.

[45] Oracle. Java jdk, 2015. URL `http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html`.

[46] Giuseppe Davide Paparo, Vedran Dunjko, Adi Makmal, Miguel Angel Martin-Delgado, and Hans J. Briegel. Quantum speedup for active learning agents. *Phys. Rev. X*, 4:031002, Jul 2014. doi: 10.1103/PhysRevX.4.031002. URL `http://link.aps.org/doi/10.1103/PhysRevX.4.031002`.

[47] S.M Ross. Introduction to stochastic dynamic programming. 1983.

[48] Stephane Ross and J. Andrew (Drew) Bagnell. Efficient reductions for imitation learning. In *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics (AISTATS)*, May 2010.

[49] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2 edition, 2003. ISBN 0137903952.

[50] A.M. Schaefer, S. Udluft, and H.-G. Zimmermann. A recurrent control neural network for data efficient reinforcement learning. In *Approximate Dynamic Programming and Reinforcement Learning, 2007. ADPRL 2007. IEEE International Symposium on*, pages 151–157, April 2007. doi: 10.1109/ADPRL.2007.368182.

[51] Noor Shaker. Platformer ai competion, 2014. URL `http://www.platformersai.com/`.

[52] S. Shinohara, T. Takano, H. Takase, H. Kawanaka, and S. Tsuruoka. Search algorithm with learning ability for mario ai – combination a* algorithm and q-learning.

In *Software Engineering, Artificial Intelligence, Networking and Parallel Distributed Computing (SNPD), 2012 13th ACIS International Conference on*, pages 341–344, Aug 2012. doi: 10.1109/SNPD.2012.93.

[53] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. Comput.*, 26(5):1484–1509, oct 1997. ISSN 0097-5397. doi: 10.1137/S0097539795293172. URL `http://dx.doi.org/10.1137/S0097539795293172`.

[54] P. Simon. *Too Big to Ignore: The Business Case for Big Data*. Wiley and SAS Business Series. Wiley, 2013. ISBN 9781118642108. URL `http://books.google.gr/books?id=Dn-Gdoh66sgC`.

[55] E.R. Speed. Evolving a mario agent using cuckoo search and softmax heuristics. In *Games Innovations Conference (ICE-GIC), 2010 International IEEE Consumer Electronics Society's*, pages 1–7, Dec 2010. doi: 10.1109/ICEGIC.2010.5716893.

[56] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, 1998. URL `http://webdocs.cs.ualberta.ca/~sutton/book/ebook/the-book.html`.

[57] E. Theodorou, J. Buchli, and S. Schaal. Reinforcement learning of motor skills in high dimensions: A path integral approach. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 2397–2403, May 2010. doi: 10.1109/ROBOT.2010.5509336.

[58] M. Tiersch, E. J. Ganahl, and H. J. Briegel. Adaptive quantum computation in changing environments using projective simulation. *ArXiv e-prints*, jul 2014.

[59] Jyh-Jong Tsay, Chao-Cheng Chen, and Jyh-Jung Hsu. Evolving intelligent mario controller by reinforcement learning. In *Technologies and Applications of Artificial Intelligence (TAAI), 2011 International Conference on*, pages 266–272, Nov 2011. doi: 10.1109/TAAI.2011.54.

[60] Lieven M. K. Vandersypen, Matthias Steffen, Gregory Breyta, Costantino S. Yannoni, Mark H. Sherwood, and Isaac L. Chuang. Experimental realization of Shor's quantum factoring algorithm using nuclear magnetic resonance. *Nature*, 414(6866): 883–887, 12 2001. URL `http://dx.doi.org/10.1038/414883a`.

[61] ... Volodymyr Mnih, Koray Kavukcuoglu. http://www.nature.com/nature/journal/v518/n7540/ 2011. URL http://www.nature.com/nature/journal/v518/n7540/pdf/nature14236.pdf.

[62] Yongjia Wang and J.E. Laird. Efficient value function approximation with unsupervised hierarchical categorization for a reinforcement learning agent. In *Web Intelligence and Intelligent Agent Technology (WI-IAT), 2010 IEEE/WIC/ACM International Conference on*, volume 2, pages 197–204, Aug 2010. doi: 10.1109/WI-IAT.2010.16.

[63] Christopher J.C.H Watkins. Learning from delayed rewards, 1989. URL http://www.cs.rhul.ac.uk/~chrisw/new_thesis.pdf.

[64] Nanyang Xu, Jing Zhu, Dawei Lu, Xianyi Zhou, Xinhua Peng, and Jiangfeng Du. Quantum factorization of 143 on a dipolar-coupling nuclear magnetic resonance system. *Phys. Rev. Lett.*, 108:130501, Mar 2012. doi: 10.1103/PhysRevLett.108.130501. URL http://link.aps.org/doi/10.1103/PhysRevLett.108.130501.

[65] Zhe Yang Yizheng Liao, Kun Yi. Reinforcement learning to play mario. 2012. URL http://cs229.stanford.edu/proj2012/LiaoYiYang-RLtoPlayMario.pdf.

[66] Øystein Førsund Bjerland. Projective simulation source code with infinte mario bros, 2015. URL https://bitbucket.org/mroystein/mariops.

[67] Øystein Førsund Bjerland. Projective simulation source code, 2015. URL https://bitbucket.org/mroystein/projectivesimulation.