Master thesis
Computing Science


Radboud University

# State-space dimensionality reduction in model-free reinforcement learning

*Author:*
Niels van Velzen
s4269454

*First supervisor/assessor:*
Dr., N. H. Jansen
n.jansen@cs.ru.nl

*Second supervisor:*
MSc., D. M. Groß
D.Gross@cs.ru.nl

*Second supervisor:*
Ing., C. Schmidl
christoph.schmidl.1@ru.nl

April 9, 2022

**Abstract**

A few dimensionality reduction method comparisons

# Contents


**1 Introduction** **3**
- 1.1 Problem statement . . . 3
- 1.2 Research questions . . . 3
- 1.3 Method, results and possible benefits . . . 4
- 1.4 Paper structure . . . 5

**2 Preliminaries** **7**
- 2.1 Reinforcement learning . . . 7
  - 2.1.1 General overview . . . 7
  - 2.1.2 Artificial neural networks . . . 10
  - 2.1.3 Double deep-q-network . . . 12
- 2.2 State-space dimensionality reduction . . . 14
  - 2.2.1 Principal Component Analysis . . . 15
  - 2.2.2 Autoencoder . . . 16
  - 2.2.3 DeepMDP . . . 17

**3 Research** **19**
- 3.1 Methodology . . . 19
  - 3.1.1 Experiments . . . 19
  - 3.1.2 Environment: Starcraft II . . . 24
    - 3.1.2.1 Agents setup . . . 25
  - 3.1.3 Environment: OpenAI Pong . . . 28
    - 3.1.3.1 Agents setup . . . 29
  - 3.1.4 Remarks on method . . . 30
- 3.2 Results . . . 31
  - 3.2.1 Research results: Starcraft II . . . 31
    - 3.2.1.1 Discussion . . . 32
      - 3.2.1.1.1 PCA Agent: losing all spatial information . . . 32
      - 3.2.1.1.2 Autoencoder agents analyses: outperforming baseline agent . . . 32
      - 3.2.1.1.3 DeepMDP agent: unable to balance multiple loss calculations . . . 34
  - 3.2.2 Research results: OpenAI Pong . . . 35

3.2.2.1 Discussion . . . . . 35
3.2.2.1.1 PCA agent: losing spatial information . . . . . 35
3.2.2.1.2 Pre-trained autoencoder agent: slightly worse than the baseline agent . . . . . 36
3.2.2.1.3 Online trained autoencoder agent: . . . . . 37
3.2.3 Results discussion . . . . . 37

**4 Related work** **52**

**5 Conclusions and future research** **54**

**A Appendix** **58**
A.1 Starcraft II: RL agent architectures . . . . . 58
A.1.1 Baseline agent . . . . . 58
A.1.2 PCA agent . . . . . 59
A.1.3 Pre-trained and online trained autoencoder agent . . . . . 59
A.1.4 DeepMDP agent . . . . . 61
A.2 OpenAI Pong: RL agent architectures . . . . . 61
A.2.1 Baseline agent . . . . . 61
A.2.2 PCA agent . . . . . 62
A.2.3 Pre-trained and online trained autoencoder agent . . . . . 63

# Chapter 1

# Introduction

## 1.1 Problem statement

With the advance of *deep reinforcement learning* in 2013, *intelligent agents* have increasingly been able to solve more complex problems [17]. In reinforcement learning (RL), an agent is trained to solve a problem through trial and error, using feedback in the form of rewards. This way it tries to find an optimal *policy*: a mapping of a state and its available actions to a valuation for each action. For *deep* RL, *artificial neural networks* are used to train such an agent. The usage of increasingly complex problem settings, comes with more complex environments for an agent to work in; observations representing the current state of the environment are becoming larger.

This means that the *state-space* of the environment, the number of possible states the environment can be in, is increasing. In general, the larger the state-space, the more difficult time intensive the training of the agent will be, and thus the more computing cost is needed. This is due to several reasons. Firstly, with a larger state-space, the agent will need to explore more states before having explored the entire state-space. This leads to having to gather larger datasets of state observations which can be impractical [**?**].

Secondly, for deep RL, the agent uses artificial neural networks. These networks need to be of a size proportional to the state-space to be able to accurately approximate an optimal policy; otherwise it could lead to under-fitting or over-fitting [27]. Generally, the larger the state-space, the larger such a network needs to be. This means the network will contain more trainable parameters, leading to more computation for its training and its usage [22].

Lastly, higher dimensional states may include more irrelevant information and noise. This can lead to longer training times for an agent and even to agents not being able to find good policies [25].

## 1.2 Research questions

A way for dealing with large state-spaces is needed for RL to be scalable. One way to deal with this, is by lowering the state-space of a specific problem, whilst retaining

enough state information for the agent to find an optimal policy. This is known as *state-space dimensionality reduction*. Here, a state observation would be projected to a lower dimensional space; this lower dimensional representation is called the *latent representation*. The main problem with state-space dimensionality reduction, is the loss of possibly essential information, which could lead to RL agents not being able to find an optimal policy.

In this paper, we will examine the effect of using state-space dimensionality reduction on an RL agent, focusing on whether an agent can find optimal policies on a lower dimensional state-space. More specifically, we will look at the effect on a model-free, value based RL agent (as explained in section 2.1). For this, we will use the *double deep-Q-learning* (DDQN) learning algorithm (see section 2.1.3) [24]. We will project the state observation to a lower dimensional space using three different methods, whose effects on the RL agent we will compare: *principal component analysis*, *autoencoders* and *DeepMDPs* (see sections 2.2.1, 2.2.2 and 2.2.3 respectively for information about each method). We will apply these methods on two environments that use high dimensional grid-based observations: Starcraft II MoveToBeacon (see 3.1.2) and OpenAI Atari Pong (see 3.1.3) [26][2].

The main research question we will examine is: *what is the effect of state-space dimensionality reduction on model-free value-based reinforcement learning in an environment using grid-based observations, by using PCA, autoencoders or DeepMDPs*? To answer this, we will answer the research sub-question: *how do the training results of a double deep-Q-learning reinforcement learning agent change when using PCA, autoencoders and DeepMDPs for state-space dimensionality reduction in Starcraft II MoveToBeacon and OpenAI Atari Pong*?

## 1.3 Method, results and possible benefits

We gathered the training results for several RL agents trained in Starcraft II and Pong. The first agent is a baseline agent: a DDQN RL agent using the observations given by the environment. The second agent uses PCA to reduce the dimensionality of the observations. Thus, the observations given by the environment first goes through PCA, thereby projected to a lower dimensional space, before being used by the RL agent. The PCA is trained on previously stored observation traces, before being used by the RL agent. There are also two agents using an autoencoder, that work similar to the PCA agent. The first of the two uses a pre-trained autoencoder, again trained on stored observations. The second autoencoder agent, the online trained autoencoder agent, is trained simultaneously with the RL agent, using the observations received by the agent. Lastly, there is a DeepMDP agent, which is only used in the Starcraft II environment.

In both environments, the PCA agent lost all spatial information, leading to RL agents learning sub-optimal policies. The DeepMDP was also unable to train to an optimal policy in Starcraft II. However, the autoencoder agents were able to learn good policies. In the Starcraft II environment, the pre-trained autoencoder agent learned a better, more stable policy in less episodes than the baseline agent. The online trained

autoencoder agent had training results similar to the baseline agent, yet slightly more inconsistent, and needing more episodes to train well. Similar results were found for the second environment, Pong. However, the pre-trained autoencoder agent learned a slightly worse and less consistent policy than the baseline agent. This resulted from the agent needing more precise spatial information than in the Starcraft II environment. This showed a limit with regards to the usage autoencoder agent.

Several benefits may arise from training agents on lower state-spaces. Though we will mention these benefits now here, the extend of these benefits are not examined in this paper and are left for future research.

Firstly, RL agents training in a lowered state-space, might need less training time and less observation data, and therefore less computational cost for training and using its neural network.

Secondly, in very complex, large scaled environments, a normal RL agent possibly cannot find an optimal policy in feasible amount of time; here, state-space dimensionality reduction could to lead to better a better policy.

Thirdly, depending on the dimensionality reduction method, we could reuse the dimensionality reduction component in different RL agents solving different problems in the same environment. Thus, we would only have to train a dimensionality reduction component once, then being able to train different agents on the same latent representation. This possibility though depends on two things. It would need to be a dimensionality reduction method that is trainable separately from the RL agent. This is the case for only two of the three methods we consider in our research: PCA and autoencoders. Secondly, the latent representation would have to be problem-independent: it would have to be a good representation of the entire state, without losing information essential to any of the RL problems being used for this environment.

## 1.4 Paper structure

We will start by giving preliminary information in section 2. Here we will first give an overview of reinforcement learning principles in section 2.1. This will include an explanation of artificial neural networks, the RL learning algorithm DDQN, which we use for our agents. After this, we will discuss state-space dimensionality reduction in section 2.2, which will include an examination of the three methods used in our research: PCA, autoencoders and DeepMDPs.

After the preliminaries, we will show our methodology in section 3.1. Here we will give a general overview of the different RL agents that we will use for our experiments. Then we will give information about the environments used and the specific agent setups per environment.

After having explained the experiment setups, we will give the results of our experiments in section 3.2. We will show and discuss the results per environment, before discussing the results of both environments as a whole.

Then, we will discuss related work in section 4, before giving our conclusions and suggestions for future research in section 5. Lastly, the appendices in section A will

contain more details on the specific agent architectures per environment, including hyperparameter settings.

# Chapter 2

# Preliminaries

To get a better understanding of our research, we provide some preliminary information in this section. We will start by introducing the concept of *reinforcement learning* in section 2.1. Following this, we will discuss *state-space dimensionality reduction*, including several methods to do this in section 2.2.

## 2.1 Reinforcement learning

The main theoretical framework we are working in, is called *reinforcement learning* (RL). This is a form of *machine learning*, the area of artificial intelligence that is concerned with creating computer programs that can solve problems by learning from data. The two other main forms of machine learning are *supervised learning* and *unsupervised learning*. The distinct feature of RL is that it learns from feedback through trial and error [19, p. 2-5].

We will now examine RL more closely and formally, as well as discuss *neural networks* and a specific RL algorithm used in our research called *DDQN*.

### 2.1.1 General overview

As mentioned, RL is the area of artificial intelligence where problems are solved through trial and error using feedback. An example is a robot learning to bring a given package to a given location. The portion of the code responsible for the decision making, i.e. choosing an action, is called the *agent*. Besides an agent, there is also the *environment*, which entails everything other than the agent. In our package delivery example this could for instance be the hardware of the robot, the package to be delivered, wind conditions, the road, and any obstacle.

The environment is encoded in a set of variables, where all possible values of these variables are called the *state-space*. In our example, some of these variables are the coordinates of the location of the robot, wind speed and direction, and the coordinates for the location where the package needs to be delivered. A single instantiation of the state-space is a *state*.

To be able to make any informed decision, the agent will need some information with regards to the current state. The information about a state received by the agent, i.e. the variables making up the state-space, is called an *observation*. This observation might be partial; the agent might not receive all information about a state. In our example, the agent might not have any information about an obstacle on the road that it hasn't sensed yet.

Using this information, the agent makes a decision about which action to take. The total set of possible actions for all states is the *action-space*. A lot of different algorithms exist with regards to decision making and learning how to make better decision. In section 2.1.3 we will discuss the learning algorithm we will use, called *double deep-Q-network*. Depending on the current state and the action taken by the agent, the environment might go to a new, different state. This change is encoded by a *transition function*; given a state and action pair it returns the next state.

The current mapping in the agent between a given state and a distribution over possible actions is called its *policy*. The better its policy, the better it is able to solve the problem. To be able to improve its policy, the agent needs information about how well it has been performing. This feedback is given in the form of *rewards*: the environment sends positive or negative rewards to the agent, informing it about how well it has performed. In our package delivery example, the robot might get a +1 reward whenever it delivers a package.

This results in an interaction cycle between the agent and the environment, depicted in figure 2.1. It begins with the agent receiving an observation. Then, it chooses an action. This results in the environment transitioning into a new state (possibly the same state as before having taken the action). After this, the environment send a new observation along with a reward to the agent, and the the agent chooses a new action. This interaction stops once the problem has been solved, or when some other constraint has been violated (like a time limit, or the robot getting into an unwanted state like the robot crashing into an object). When the cycle stops, we have finished an *episode*. After this, the environment can be reset to start a new episode. To get to a well performing policy, the agent often needs hundreds or thousands of episodes [19, p. 6-10].

Let us now formally define the reinforcement learning paradigm. RL problems are commonly modeled as *Markov decision processes* (MDPs).

**Definition 1.** *A Markov decision process (MDP) is a tuple* $(S, A, T, R, S_\theta)$*, with state-space* $S$*, the action-space* $A$*, transition probability function* $T$*:* $S \to 2^{A \times Distr(S)}$ *(thus modeling stochasticity), reward function* $R$*:* $S \times A \times S \to \mathbb{R}$ *and initial states distribution* $S_\theta$*.* $T$ *also describes the probability distribution for transitions,* $p(s'|s, a)$*: the probability of a transitioning to state* $s'$ *given state-action pair* $(s, a)$ *[19, p. 45-62].*

A graphical example of an MDP can be seen in figure 2.2.

The reward function models the expected reward for a transition; given a transition consisting of a state, action and next state, it returns the expected reward.

The reward function is defined as function $r$:

$$r(s, a, s') = \mathbb{E}[R_t | S_{t-1} = s, A_{t-1} = a, S_t = s'] \tag{2.1}$$

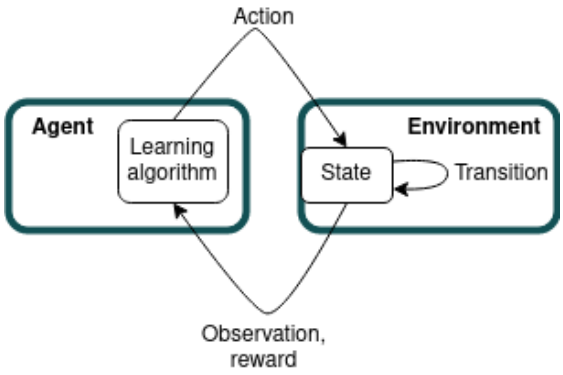



Figure 2.1: The cycle of interaction between the environment and an agent in reinforcement learning.

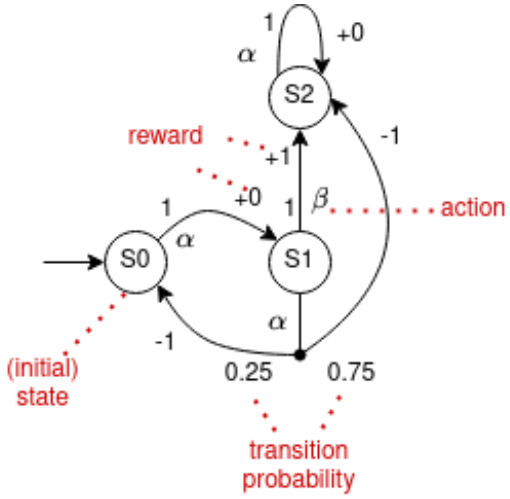



Figure 2.2: Example of a Markov decision process (MDP).

where $t$ is the timestep and $R_t \in R \subset \mathbb{R}$ [19, p. 54].

To get to its best possible policy, an agent tries to maximize its total sum of expected reward. This is also known as the *return* $G$:

$$G_t = R_{t+1} + \gamma G_{t+1} \tag{2.2}$$

with timestep $t$ and a discount factor $\gamma \in [0, 1]$ to set the weight of future rewards [19, p. 67].

Often, an agent does not know the underlying MDP to a problem. Therefore, we must use a way without using the underlying MDP to find the best possible policy (i.e. the policy giving the highest expected return). To evaluate any policy $\pi$ and be able to compare its performance to other policies, we can use the *Bellman equation*. This tells us what the expected return is when starting at any state $s$, following policy $\pi$. This is

known as the *state-value function* (V-function, or $V^{\pi}(s)$).

$$v_{\pi}(s) = \sum_{a} \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_{\pi}(s')], \forall s \in S \tag{2.3}$$

For any state, we look at all possible actions. For each resulting state-action pair, we look at all possible transitions and sum the corresponding reward with the value of the next state weighted by the discount factor $\gamma$. This sum is then weighted by the probability of this transition. The resulting value is summed for all possible transitions of this state-action pair, and we then sum these results for all actions of the given state. This results in the value for this state [19, p. 73].

Having a method of finding the value of a state, is the first step towards an agent being able to figure out the best possible policy. When an agent tries to improve its policy, it can make use of the *action-value function* (Q-function, or $Q^{\pi}(s, a)$. This tells us the expectation of returns following policy $\pi$ after having taken action $a$ in state $s$. It is defined as follows:

$$q_{\pi}(s, a) = \sum_{s',r} p(s', r|s, a)[r + \gamma v_{\pi}(s')], \forall s \in S, \forall a \in A(s) \tag{2.4}$$

For all possible state-action pairs $(s, a)$, we calculate their value by looking at all their possible transitions, taking the corresponding reward, sum this by the state-value of the next state weighted by the discount factor $\gamma$, and weigh this sum with the probability of this transition happening [19, p. 74].

An agent is in search of an optimal policy: a policy that has an expected return greater or equal to all other possible policies. Such a policy has the optimal action-value function, known as the $q*$ function: this is the action-value function with the highest value over all possible policies. Therefore, if the agent is able to figure out the $q*$ function, it knows the optimal action for each state and thus the optimal policy. Approximating this $q*$ is the approach for most current state-of-the-art learning algorithms, including the one used in this research: double deep-Q-network (DDQN).

We will explain how DDQN is able to approximate the $q*$ function in section 2.1.3. However, to understand DDQN, we will first need to introduce *neural networks*. This is because in order to approximate the $q*$ function, an agent will need to keep track of the action-value for all state-action pairs. Storing these values in a table is not scalable: for problems with large (or continuous) state-spaces and large action-spaces, the table would be became way too large to be able to work with. Therefore, we need a way to approximate this table, which is what neural networks are used for. We will now first explore neural networks, before explaining DDQN.

### 2.1.2 Artificial neural networks

The use of artificial neural networks (ANN) in reinforcement learning, is known as *deep RL* [19, p. 5]. ANNs are a way of being able to use RL in problems with large state-space and large action-spaces. This is because ANNs are able to approximate both linear and

non-linear functions in an efficient way[6, p. 165-166]. Therefore, they can be used to approximate the $q*$, instead of using an action-value table to know the $q*$ function. We will now shortly examine ANNs, using [6, p. 164-366].

ANNs consist of artificial neurons: a mathematical function that sums its input which is used to produce an output. These neurons are grouped into different layers, consisting of any number of neurons. Firstly, there is the input layer. In the case of RL, its input is most often the variables making up an observation. Secondly, there is the output layer, creating the output of the network. In between these two layers there usually are other layers. These are known as the hidden layers.

To produce an output, a neuron sums its input. Each input of a neuron has its own weight: a variable settings its importance. The network learns to produce better output by changing these weights. After summing the weighted inputs, the result is passed through an (often nonlinear) activation function. This produces the output of the neuron.

Different types of neural networks exist. One important difference here is the connections between neurons. The simplest form is a *feedforward* neural network. Here, each neuron is only connected to neurons of the next layer, meaning there are no cycles. They are often fully connected: a neuron passes its output to each neuron of the next layer. An example of fully connected feedforward neural network can be seen in figure 2.4.

Another important type of neural network is a *convolutional neural network* (CNN). This is a network that is useful for grid-like data, like image data. It is well suited for applications like image recognition. Such a network makes use of at least one convolutional layer. A convolutional layer consists of multiple filters (or: kernels): a set of trainable parameters. Each kernel convolves across the input to produce a feature map (or: activation map). Since each kernel has different parameters, each feature map will highlight different features of the input. These feature maps are then stacked to produce the output of the convolutional layer. This process is shown in figure 2.3, where a convolutional with a single filter is shown. Adding filters would mean that those filters would convolve in the same way over the same input, creating multiple feature maps that are stacked to produce its output.

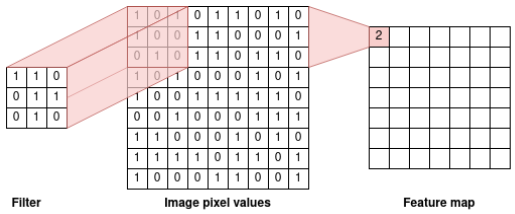



Figure 2.3: A convolutional layer with a single filter. The filter convolves across the input image to produce one feature map. Adding more filters would results in a stack of feature maps (one per filter).

To improve the given output, a network can update its weights. For this it needs

a loss function that computes how incorrect given output was. To calculate how much each weight needs to be changed, backpropagation is used. This computes the gradient for each weight with respect to the loss function. This way, loss can be minimized out therefore the output optimized.

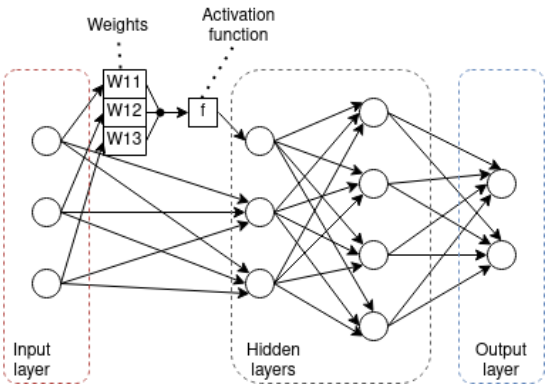



Figure 2.4: Example of a fully connected feedforward artificial neural network. For one node, example weights and activation function are shown.

For more details on artificial neural networks, we refer to [6]. ANNs are a central part of the learning algorithm used in this paper: double deep-Q-network, which we will now examine.

### 2.1.3 Double deep-q-network

Before going into the learning algorithm DDQN, it is useful to start by looking at other algorithms preceding DDQN. Firstly we will look at *Q-learning* and *deep-Q-learning* (DQN).

The RL approach taken in Q-learning is arguably the most used approach in RL [13]. It uses a table, the Q-table, to keep track of the Q-function determining the agent's policy. This Q-table is updated with new information gathered by the agent, to ultimately find an optimal policy by approximating $Q*$.

This is done through a cycle of interactions of the agent with the environment. The agent starts by getting an observation and choosing an action. The environment moves to a new state and the agent receives a new observation and a reward for the current transition. Then the Q-table is updated and the cycle starts over until we reach the end of our episode. Depending on the problem, the agent will need hundreds or thousands of episodes to get the Q-table to approximate the $Q*$ function.

When choosing an action, exploitation needs to be balanced with exploration. The agent needs to explore different actions in the same state to find out what happens and explore as many states as possible. Simultaneously, the agent will only start performing well once it starts exploiting the updated Q-table. To do this, an $\epsilon$*-greedy strategy* is used. Each time an action must be taken, the agent gets a random number between 0 and 1. Whenever this is lower than or equal to the $\epsilon$ value, a random action is taken,

otherwise it takes chooses an action greedily based on the Q-table. In this strategy, $\epsilon$ is usually decayed over time, to slowly put more emphasis on exploitation.

To update the Q-table after a transition, the old Q-value is added to the new information gotten from this transition, known as the *temporal difference*. The temporal difference is first weighted by a hyperparameter called the *learning rate*. This temporal difference is the difference between the current estimate in the Q-table and the *TD-target*, which is the Q-value we are now moving towards. Thus, it represents the error between the current Q-value and the new estimate and is therefore known as the *TD-error*.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \cdot (r_t + \gamma \cdot \max_a Q(s_{t+1}, a) - Q(s_t, a_t)) \tag{2.5}$$

In this equation, $r_t$ is the reward for this transition, $\alpha$ the learning rate and $\gamma$ the discount factor. The TD-target is calculated by adding the reward to the maximum Q-value of the next state weighted by the discount factor. Subtracting the old Q-value from the TD-target gives the TD-error.

As mentioned in section 2.1, the use of a Q-table is not scalable. Therefore, new algorithms were introduced that use neural networks. One such algorithm is DQN. DQN was first introduced in 2015 by V. Mnih, K. Kavukcuoglu and D. Silver, and was the first algorithm to apply RL to high dimensional data while getting human-level scores for Atari-games [18].

Instead of computing Q-values directly using a Q-table, DQN *approximates* the Q-function using neural networks. Such a network takes as input a state observation, and produces as output the Q-value for all possible actions in this state. To act greedily, the agent must simply pick the action with the highest value in the output of the network.

To get the Q-values for a state, we pass the state to its policy network. This is the network representing the policy of the agent. To be able to compute the TD-error, we also pass the next state to a network, to get the max Q-value of the next state. However, we use a second network for this. If we were to use the same network, the target would change every time the policy network is updated, thereby having it chase its own tail. Having a second network, the target network, allows us to freeze targets for a certain number of steps (a hyperparameter), before updating the target network to equal the policy network.

Additionally, we don't train the policy network with every step, but use a hyperparameter to set how often we train the network. Thus, we must store *experiences*: tuples containing information about a transition, including the initial state, next state, reward and whether we ended in a terminal state. When training the network, we train it on batches of these experiences, known as *replay memory*.

Thus, when training the policy network we pass batches of next states to the policy network to get their maximum Q-values. The corresponding rewards are added to calculate the TD-target. Then the initial states are passed to the policy network to get their Q-values. These values are then used to calculate the TD-error which used to calculate the loss using the loss-function. Then the policy network is updated by backpropagating the loss.

Now we can look at the algorithm used in this paper: double-DQN (DDQN). An overview of how a DDQN agent works is given in figure 2.5. DDQN is every similar to DQN. The difference is in how the policy network is trained [24]. In DQN we are taking the maximum of all Q-values. Since these values are estimates that differ from their true value, DQN tends to overestimate the highest Q-value. Doing this often will lead to divergence [19, p. 293-297]. Implicitly we are using the target network for two things: to get the action with the highest value, and what its value is. To solve the problem of overestimating Q-values, DDQN separates these two concerns: we use the policy network to get the action with the highest Q-value and then use the target network to get this action's Q-value. This way, the target network cross-validates the estimates of the policy network.

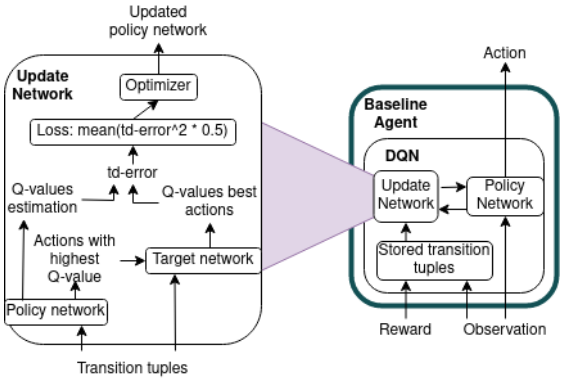



Figure 2.5: An overview of how a DDQN RL agent works.

The pseudocode for DDQN is given in figure 1. This shows its training step for optimizing the policy network. This method is called within the interaction between agent and environment as explained in section 2.1. In short, the agent will receive an observation from the environment and the agent will, in our case, use a $\epsilon$-greedy strategy to choose an action, where the policy network is used. Then, the environment transitions and gives a new observation to the agent. Hyperparameters are used to set the number of such steps in between training the policy network, and to set the number of steps between updating the target network.

## 2.2 State-space dimensionality reduction

In RL, states and observations are determined by variables, as mentioned in section 2.1. A single state, and thus also a single observation, is simply a single instantiation of these variables. All possible values of these variables together form the total state-space. A larger state-space means more computing cost for the agent. This has several reasons. Firstly, the agent needs to explore more states to explore the entire state-space. This in turn means having to gather larger datasets containing observations, which can be impractical [**?**]. Secondly, for an agent to be able to form a decent policy, the policy

**Algorithm 1** DDQN training step [19, p.299].

Sample *experiences* from replay buffer
$states, actions, rewards, next\_states, terminals \leftarrow experiences$
$indices\_a\_q\_sp \leftarrow \arg\max_a policy\_network(next\_states)$
$q\_sp \leftarrow target\_network(next\_states)$
$max\_a\_q\_sp \leftarrow q\_sp[indices\_a\_q\_sp]$
$max\_a\_q\_sp \leftarrow max\_a\_q\_sp \cdot (1 - terminals)$ ▷ where in *terminals* True equals 1, False equals 0.
$target\_q\_sa \leftarrow rewards + \gamma \cdot max\_a\_q\_sp$
$q\_sa \leftarrow policy\_network(states, actions)$
$td\_error \leftarrow q\_sa - target\_q\_sa$
$loss \leftarrow mean(td\_error^2 \cdot 0.5)$
Optimize *policy_network* with backpropagation using *loss*

network (and target network in case of DDQN) needs to be of an appropriate size. When it is too small for a certain state-space, it will not be able to approximate the Q-function well enough. Therefore, in general, the larger the state-space, the larger the policy network has to be. Training and using a larger network takes more computing cost, since we are dealing with more trainable variables (i.e. the weights of a neural network) [22]. This is especially true for CNNs, which are commonly used in RL when dealing image data [12]. Lastly, higher dimensional data may include more noise, which can prevent an agent from finding an optimal policy [25]. Thus, having a smaller state-space results in a less computationally expensive agent; decisions can be made quicker (since a forward pass through the network will be slightly faster) and the agent is able to faster converge to a good policy.

The size of the state-space is problem and environment specific. Generally, more complex problems and environments come with larger state-spaces. A way for dealing with large state-spaces is needed for RL to be scalable. One such way is to make the state-space of a specific problem smaller, while retaining enough information for the agent to get to a good policy. This problem of reducing the dimensionality of the state-space while keeping the necessary information for an agent to train on, is known as *state-space dimensionality reduction*.

Several methods for state-space dimensionality reduction exist. We will discuss and use three of them in this research: *principal component analysis*, *autoencoder*, and *deep-mdp*. We will now discuss these methods in detail.

### 2.2.1 Principal Component Analysis

The first method for state-space dimensionality reduction we will discuss, is *principal component analysis* (PCA) [14].

The general idea of using PCA to reduce dimensionality of the latent space, is to apply PCA on a state observation of the agent immediately after receiving it from the environment. Applying PCA will project the observation data to a lower dimensional

space. This lower dimensional observation will then be the observation the agent will use. Thus, the dimension PCA projects to determines the new state-space dimensionality [3].

The first important mathematical concept behind PCA, is the *covariance matrix*. This computes the covariance between each variable-pair, thereby computing their correlation. To compute the covariance matrix, we first subtract the mean of each column in the original data matrix from every value in that column. Then, depending whether variance between features corresponds to the importance of that feature, we may standardize the data by dividing each value with the standard deviation of its column. This results in matrix $X$. To compute its covariance matrix, we simply transpose $X$ and multiply it with $X$ giving matrix $X^T X$.

The other two important mathematical concepts behind PCA are *eigenvectors* and *eigenvalues*. An eigenvector $v$ of a matrix $X$ is a vector such that multiplying it with $X$ results in a variable number of vector $v$:

$$A \cdot v = \lambda \cdot v \tag{2.6}$$

The number of vectors $v$ that we end up with, i.e. $\lambda$ in equation 2.6, denotes how much we are scaling the eigenvector and is called the *eigenvalue* of that eigenvector. The number of eigenvectors that a matrix has, is at most $min(\#rows, \#columns)$.

For PCA, we calculate the eigenvectors and eigenvalues of the covariance matrix $X^T X$. These eigenvectors, called *principal components*, represent the axis of the original matrix $X$ with the highest variance, i.e. capturing the most information. Together, all principal components capture the entire original data. However, the higher the eigenvalue of a principal component is, the higher its variance is and thus the more information it captures. Therefore, we sort the eigenvectors based on their eigenvalue; thus the first principal component captures the most information of all principal components. We can then take the first $x$ number of principal components to project the original data onto. Taking for instance 10 principal components of the original data that contained 15 features, means we project the original data onto a 10 dimensional space.

To use PCA for state-dimensionality reduction, we simply apply PCA to state observations that the agent receives from the environment. After applying PCA, the observation is projected to a lower dimensional space (depending on the number of principal components we take). This lower dimensional observation is then further used by the agent, thus training the agent on a lower dimensional state-space.

### 2.2.2 Autoencoder

Another way of projecting data onto a lower dimensional space, is using an *autoencoder*[9]. An autoencoder is a neural network consisting of two parts: an encoder network and a decoder network. The encoder projects the given input data onto a lower dimensional space, also called the *latent space*. The output of the encoder, called the *latent representation*, is used as input for the decoder. This decoder tries to reconstruct the original input from the latent representation as closely as possible. The autoencoder architecture is shown in figure 2.6.

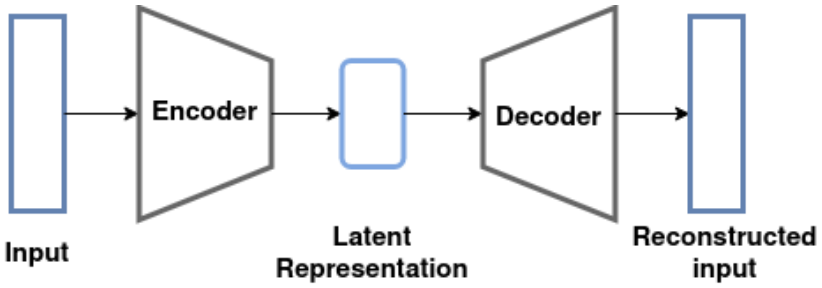



Figure 2.6: The architecture of an autoencoder.

Formally, an autoencoder can be defined by the two functions it learns:

$$\phi : \mathbb{R}^n \to \mathbb{R}^m \tag{2.7}$$

$$\psi : \mathbb{R}^m \to \mathbb{R}^n \tag{2.8}$$

$$\phi, \psi = \arg\min_{\phi,\psi} \Delta(x, \psi \circ \phi(x)) \tag{2.9}$$

Equation (2.7) defines the encoder and equation (2.8) the decoder, both satisfying (2.9) where $\Delta$ is the reconstruction loss function for input x. The closer the output of the autoencoder approximates the original input, the lower the loss.

A fundamental difference with using PCA, is the type of functions that can be approximated for lowering the dimensionality. Since an autoencoder uses neural networks, they can approximate nonlinear functions, as mentioned in section 2.1.2. This is in contrast with PCA, which can only approximate linear functions. Because of this, an autoencoder can learn more powerful generalisations which leads to lower information loss[9].

To use an autoencoder for state-space dimensionality reduction, we again (like with PCA) use as input an observation the agent got from the environment. We can train the autoencoder using both the encoder and the decoder to calculate the loss. When used in RL, only the encoder is used. The encoder will project the observation input to the dimensionality of the latent space.

### 2.2.3 DeepMDP

[5] [7] plaatje deepmdp agent The last method for state-space dimensionality reduction, is a *DeepMDP* [5]. In contrast with the previous methods, the DeepMDP is a partial RL agent: training the dimensionality reduction goes hand in hand with training the RL agent.

This is done through several steps that alter the way the loss for the policy network is calculated. Firstly, there is an encoder neural network that projects the original data to a lower dimensional space, similar to the encoder of the autoencoder. However, in contrast with the autoencoder, this encoder is trained together with the network representing the Q-function using the same optimizer (and thus the same loss). This

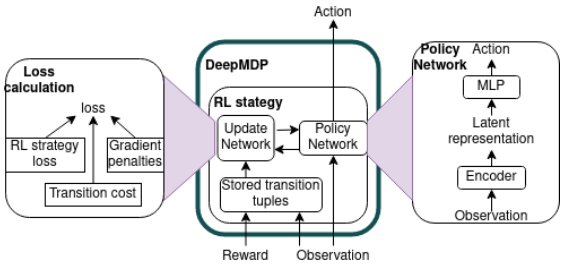



Figure 2.7: An overview of how a DeepMDP agent works.

can be seen in figure 2.7, specifically the *policy network* part. Like a normal value-based RL agent, it trains its policy network using stored transition tuples. Differently from value-based strategies, its policy network now also contains an encoder. An observation first goes through the encoder, lowering the state-space dimensionality, before going into the network representing the Q-function (the *MLP* in figure 2.7. The presence of the encoder will not only project the data to a lower dimensional space, but will also influence the loss calculation for the entire policy network, since the encoder is trained simultaneously with the network representing the Q-function.

The loss calculation is also altered in a second way, which in figure 2.7 can be seen in the *Update policy network* part. The DeepMDP makes use of an auxiliary objective to calculate the loss, using another neural network: the transition cost network. This network calculates the cost for predicted transitions. For a given latent state (given by the encoder), it calculates the cost for all possible transitions to a next latent state. This means that its output has dimensions $actionspace \times latentspace$. The action space here represents the actions that can be taken in the current latent state, while the latent space represents the next (predicted) latent observation.

Again this network is a part of the agent's network. Consequently, the agent's network consists of three parts: an encoder part, a policy part, and a transition loss part, all trained on the same loss using a single optimizer.

Lastly, a Lipschitz-constrained Wasserstein metric is used. The Wasserstein metric corresponds to the minimum cost of transforming one distribution into another; thus it calculates the cost of moving a particle in a point in one distribution to a point in the other [5]. The Lipschitz constraint then sets a bound to the degree to which a (value) function can change its values. This is implemented through calculating a gradient penalty on all three parts of the network separately (i.e. the Q-function, the encoder and the transition cost network), as done in [7].

Consequently, the loss calculation is a weighted sum of three parts: the loss calculation of the RL strategy, the transition cost given by the transition cost network, and the gradient penalties for the Q-function network, the encoder network, and the transition cost network.

# Chapter 3

# Research

The aim of this paper is to examine the effect of reducing the dimensionality of the state-space in reinforcement learning (RL). In this section we will discuss the our research and its results. We will start by detailing our method in section 3.1; here we will explain the environments we used for our experiments, as well as the experiments that we ran. After this, we will show and discuss the results from these experiments in section 3.2. The discussion of the results will include an examination of how the different state-space reduction methods led to their results.

## 3.1 Methodology

In this section we will explain our method: how we researched the effect of state-space dimensionality reduction on an RL agent. First, in section 3.1.1, we will explain the experiments in general: the different agent setups that we compared. Then, we will look at the environments in which we ran the experiments in sections 3.1.2 and 3.1.3, including their specific agent architectures.

### 3.1.1 Experiments

To examine the effect of the different dimensionality reduction methods, we implemented multiple RL agents using different reduction methods and compared their performance in two environments. In this section we will discuss the agents that we used. We will give a general overview here, and give more details about the architectures of the agents in the sections explaining the used environments: sections 3.1.2 and 3.1.3.

The first agent mentioned is the baseline agent, which does not use a dimensionality reduction method, therefore using the full dimensions of the observation received from the environment. All other agents reduce the dimensionality of the observations, thus using less features. The general work flow of these agents is given in figure 3.1.

Furthermore, to allow for a fair comparison of their performance, all agents must share as many architectural design choices and hyperparameter settings as possible.

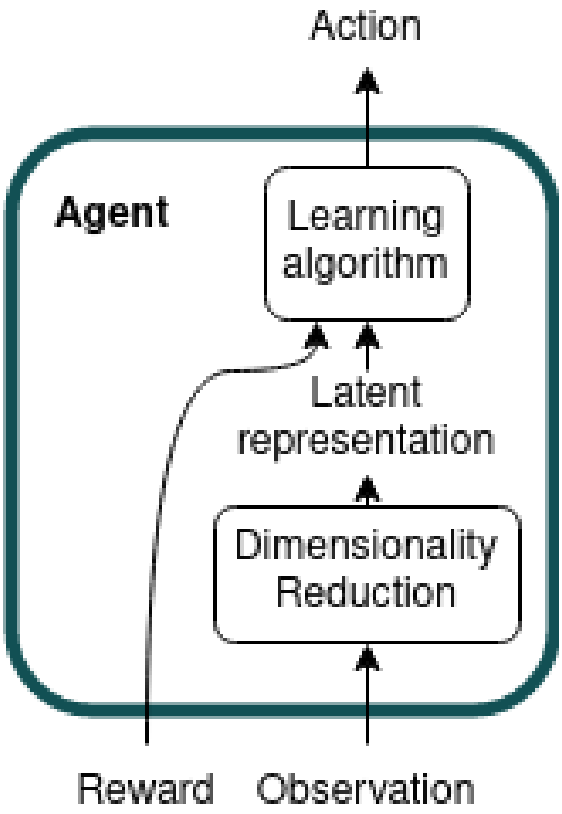



Figure 3.1: Overview of the general architecture of an RL agent using a state-space dimensionality reduction method. In our experiments, the learning algorithm used is DDQN.

This is done by extending the baseline agent in all other agents.

**Baseline agent**
The baseline agent is a standard RL agent that does not use any dimensionality reduction. This is the agent that is extended by all other agents. It uses a DDQN strategy, as explained in section 2.1.3.

An overview of the agent, i.e. a high-level view of DDQN, is given in figure 3.2. First, the agent receives an observation from the environment. This observation is passed to its neural network approximating the Q-function: its policy network. This returns a valuation for each action taken in this state. Then, the agent either chooses the action with the best valuation (i.e. acting greedily) or chooses a random action. Then, the chosen action is performed and we repeat this cycle until the end of the episode, whilst often training the policy network on stored transitions.

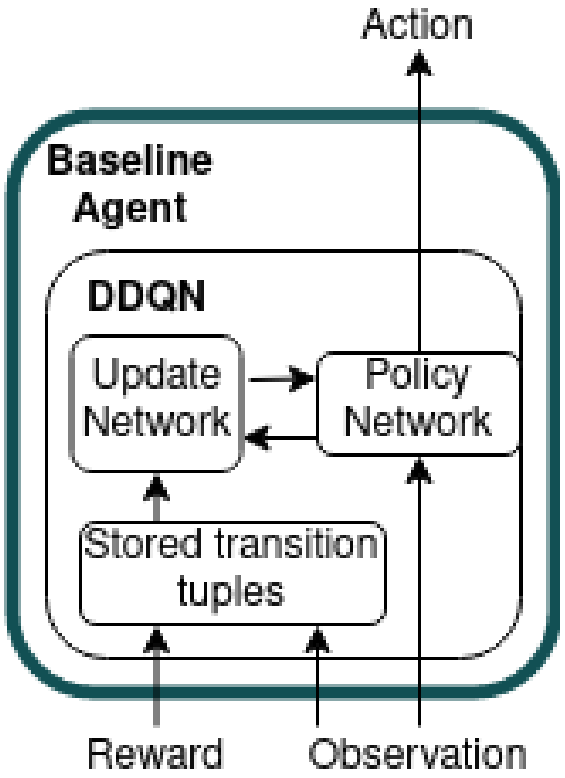



Figure 3.2: Overview of the baseline agent, using DDQN.

**PCA agent**
The PCA agent uses PCA to reduce the dimensionality of the state observations. As mentioned, this is done by extending the baseline agent: after receiving an observation from the environment, the observation is processed by a PCA component lowering the observation dimensionality. This latent representation is then used by the agent as if it is the actual observations. This means that it is passed to the policy network to give an action valuation, as well as being stored in transitions used to train the network. This is shown in figure 3.3.

The PCA component is trained separately before being used by the agent. This is done by training the PCA on previously stored observations. It is important that these observations give a good representation of the entire environment to get a well trained PCA component.

**Pre-trained autoencoder agent**
This agent is very similar to the PCA agent, except instead of using a PCA component, we are using an autoencoder to reduce dimensionality. Its overview is given in figure 3.4. Just like the PCA component, the autoencoder is pre-trained on stored observations. After this it is used by the agent to reduce the dimensionality of the observation.

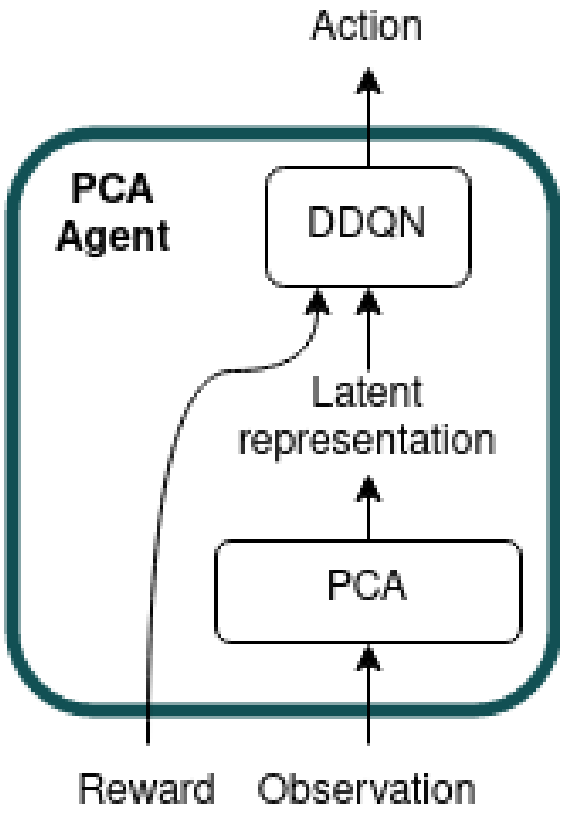



Figure 3.3: Overview of PCA agent.

The autoencoder is trained by passing batches of observations to the encoder, which performs the dimensionality reduction. Its output is then passed to the decoder which tries to reconstruct the original data. The loss is then calculated by how similar the decoder output is, compared to the original data. Specifically, the *mean squared error* is used.

When being used by an agent to reduce the dimensionality of an observation, only the encoder part of the autoencoder is used. When we speak of the output of the autoencoder in the context of being used by an agent, we mean the output of the encoder part. The autoencoder is not being trained further while in use by an agent.

**Online trained autoencoder agent**

I guess "online" is not strictly correct

This agent is very similar to the pre-trained autoencoder agent, so we again refer to figure 3.4. The only difference is the moment of training the autoencoder. In the pre-trained autoencoder agent, the autoencoder is trained before being used by an agent, using previously stored observations. In this online trained autoencoder agent, we are using an autoencoder that has not been pre-trained; it is being trained while being used

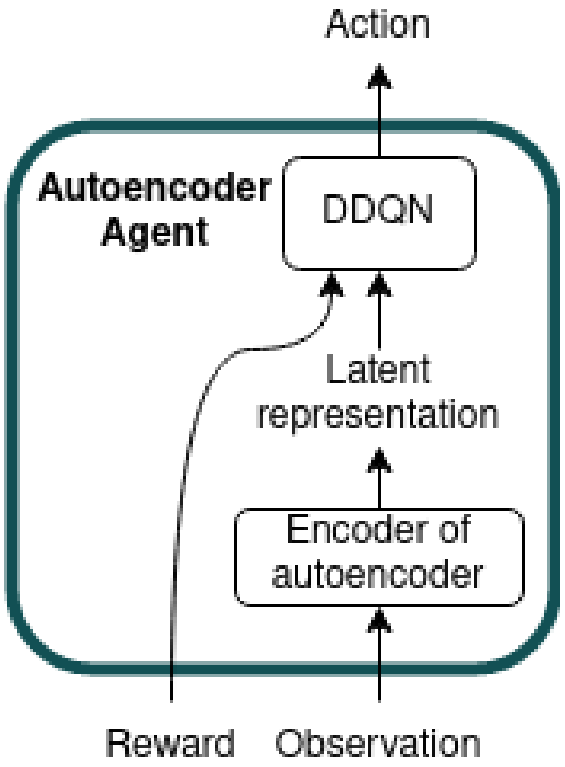



Figure 3.4: Overview of the pre-trained and online trained autoencoder agents.

by the agent.

In this case, the agent itself still only uses the encoder part of the autoencoder. However, we now also store batches of observations and pass these to the training method of the autoencoder. This training method is the same as before: passing the observations to the encoder, whose output is passed to the decoder, whose output is compared to the original observation to calculate the loss and train the network.

**DeepMDP agent**

This agents has only been used in the Starcraft II environment, due to time limitations. Just like the online trained autoencoder agent, the DeepMDP agent is completely trained while being used by the agent. It also uses an encoder, which has the same design as the encoders of the autoencoders. Differently from the autoencoder though, this encoder is actually part of the agent's network; whereas the autoencoder is a separate network, the DeepMDP simply extends the network of the agent, thus training the policy network and encoder simultaneously. This is further explained in section 2.2.3.

In figure 3.5 it can be seen that the observation given by the environment goes directly into the policy network. However, in contrast with the baseline agent, the observation first goes through the encoder, before going into the network representing the Q-function.

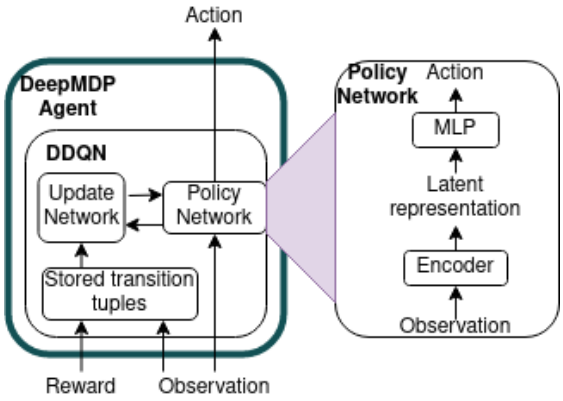



Figure 3.5: Overview of the DeepMDP agent.

Another difference with the baseline agent is not shown in the figure: the DeepMDP makes use of an auxiliary objective to calculate the loss while training the policy network: the transition loss. This is also explained in section 2.2.3.

### 3.1.2 Environment: Starcraft II

The first environment used for our experiments is the *StarCraft II* environment by *Blizzard*[1]. StarCraft II is a real-time strategy game, which has been used in RL research after the introduction of a learning environment created in collaboration with *DeepMind*, called *SC2LE* and a corresponding Python component called *PySC2*[26].

In particular we are using a PySC2 minigame called *MoveToBeacon*. This minigame simplifies the StarCraft II game. Here, the RL agent must select an army unit and move it to a given beacon. To simplify our RL agent, selecting the army unit is implemented as a script, thereby focusing our research on moving the army unit to the beacon. A screenshot of the game is given in figure 3.6.

An **observation** received by the agent in this minigame is given by a $32 \times 32$ grid, representing the entire state of the game, giving a total of 1024 **features**. Each cell in the grid represents a tile in the game. It can have one of three values: a 0 denoting an empty tile, a 1 denoting the army unit controlled by the agent, or a 5 denoting the beacon. The beacon comprises more than one tile, namely a total of 21 tiles; it comprises five adjecent rows, where the first comprises three adjecent columns, followed by three rows of five columns, followed by a row of three columns, creating a octagonal-like shape. Because of this, the beacon has $27 \cdot 27$ places where it could be, with the army unit having 1003 tiles left to be. This gives a total state-space of $32 \times 32$ with a cardinality of $27 \cdot 27 \cdot 1003 = 731.187$. An example of such a state observation can be seen in figure 3.7.

Is this correct? Or are there actually perhaps only 3 features or something?

An **action** taken by the agent is given by an $(x, y)$ coordinate with $x, y \in \{0..31\}$. This denotes the (indices of the) cell in the grid that the army unit will move to. Each action sent to the environment will result in 8 in-game steps, meaning that if the given

Is this a correct usage of state-space?


Figure 3.6: Screenshot of the minigame *MoveToBeacon* in *StarCraft II*.

coordinates are further away than 8 steps, the unit will only walk 8 steps towards the given coordinates before the agent has to choose a new action.

Lastly, an **episode** takes 120 seconds. The goal is to move the army unit to the beacon as often as possible in this time limit, each time adding 1 point to the episode score. At the start of each episode, the beacon and army unit are placed randomly. Whenever the army unit reaches the beacon, only the beacon will be relocated randomly.

Within the 120s time limit, the agent will be able to take 239 steps/actions. An agent following a random policy gets a score of about $0 - 3$ points per episode (again, one point for each time the army unit reaches the beacon), whereas a scripted agent scores about $25 - 30$ points per episode (meaning the agent on average needs 8 or 9 actions before reaching a beacon).

#### 3.1.2.1 Agents setup

We will now explain the architectures of the agents used in the Starcraft II environment. We will only give a general overview, referring to appendix A section A.1 for details on the neural network architectures and hyperparameter settings.

Whereas the baseline agent uses the $32 \times 32$ observations given by the environment, all other agents reduce the dimensionality to $16 \times 16$. This means that the number of features in an observation are reduced from 1024 to 256.

Again, is this correct?

Though the baseline agent's architecture is extended by all other agents (in order to keep the agents as similar as possible), one important change must be made. The policy networks of all agents are convolutional neural networks, and the output dimensions of a

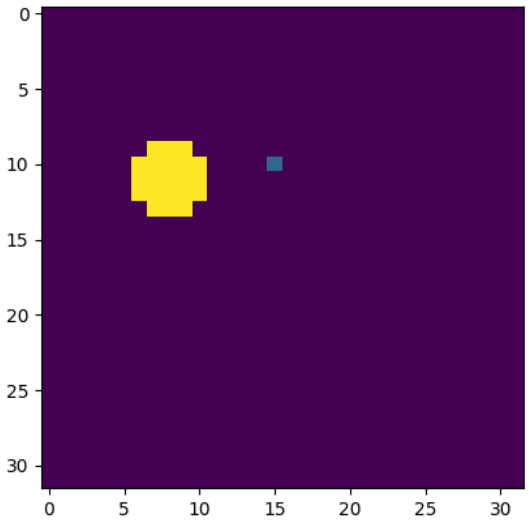



Figure 3.7: A state observation received by the RL agent, for the StarCraft II minigame MoveToBeacon. The yellow cells represent one beacon; the blue cell represents the army unit controlled by the player; all other cells are empty.

convolutional layer depends on the dimensions of its input. The baseline agent's policy network receives a $32 \times 32$ input, whereas the other agents receive a $16 \times 16$ input. In all cases, the output dimensions must be $32 \times 32$. This is because the network approximates the Q-function: a valuation of all actions for a given state. Since an action in our environment is defined by the coordinates the army unit must walk to, there are $32 \times 32$ possible actions. To deal with this difference in input dimensions, the first layer of the policy networks of the non-baseline agents are modified (using different stride and padding sizes) to increase the dimensionality to $32 \times 32$.

**Baseline agent**

The baseline agent is a standard RL agent that does not use any dimensionality reduction and is extended by the other agents.

Its policy network consists of three convolutional layers: the first layer being a transposed convolutional layer [4], the other two being regular convolutional layers. The use of the transposed convolutional layer allows for the possibility of upsampling the dimensionality of the given input. This is needed in agents using dimensionality reduction, where the input for the network is $16 \times 16$, but the output needs to be $32 \times 32$ to cover the action space. However, for our baseline agent, the input dimensions, $32 \times 32$, must remain the same, which is achieved by setting the stride of the first layer to 1. This way, both the dimensions of the input and the output of the network are $32 \times 32$ (where its input represents the current state observation and its output the action valuation).

**PCA agent**
The PCA agent uses PCA to reduce the dimensionality of the state observations from $32 \times 32$ to $16 \times 16$. To do this, the observation input must first be flattened, and its output must be reshaped to $16 \times 16$.

The output of the policy network representing the Q-function must remain $32 \times 32$, since we have $32 \cdot 32$ possible actions: one action per coordinate. Therefore, the first layer in the policy network, the aforementioned transposed convolutional layer, has a stride of 2 and an output padding of 1. This changes the dimensions from $16 \times 16$ to $32 \times 32$.

The PCA component is trained on 240.000 previously stored observations, which corresponds to observations from 1000 episodes. These have been gathered using a scripted agent. The first 256 principal components in our PCA (representing a $16 \times 16$ dimensional observation), contain roughly 96% of the variance of the original data. We are using a scalar to standardize the observation data as explained in section 2.2.1.

**Pre-trained autoencoder agent**
Just like the PCA component, the autoencoder is pre-trained on the same 240.000 observations. After this its encoder is used by the agent to reduce the dimensionality of the observation.

The encoder and decoder of the autoencoder are convolutional neural networks. The encoder uses two convolutional layers: the first has a stride of 2, which reduces the dimension to $16 \times 16$. The decoder, which tries to reconstruct the original data, uses three convolutional layers. The first layer is a transposed convolutional layer with a stride of 2, to bring the dimensions back to $32 \times 32$. The other two layers are regular convolutional layers.

**Online trained autoencoder agent**
This agent has the exact same design as the pre-trained autoencoder agent. As mentioned, the only difference is the moment of training the autoencoder: instead of training on previously stored observations, we are now training the autoencoder simultaneously with the RL agent.

I guess "online" is not strictly correct

**DeepMDP agent**
The encoder of the DeepMDP has the same design as the encoders of the autoencoders: one convolutional layer with stride 2 to reduce dimensions and a second convolutional layer with stride 1.

The auxiliary objective to calculate the transition loss represents the cost of all possible transitions from a given latent representation. This means that its output has dimensions $(32 \times 32) \times 16 \times 16$. The tuple $(32, 32)$ represents the actions that can be taken in the current state, while the other two dimensions, $16 \times 16$ represent the next (predicted) latent observation. It has only one layer, a convolutional layer, with $32 \times 32$ output channels to represent the action dimensions.

### 3.1.3 Environment: OpenAI Pong

The second environment in which we ran our experiments is the Atari game Pong, using the OpenAI Gym environment [2]. In figure 3.8, a screenshot of the game is given. In the game, two players try to score points by getting a ball behind the opponent. Both players control one paddle that can be moved vertically, to bounce to ball off of. The first player to score 21 points wins.

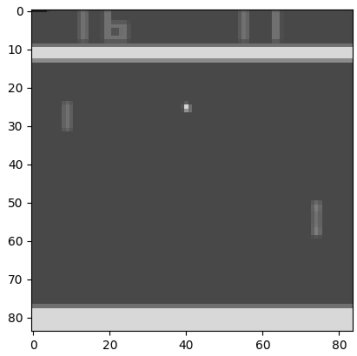



Figure 3.8: A screenshot from the OpenAI Gym Atari game Pong.

*Observations* are created from frames of the game: screenshots containing pixel values. Each frame is processed to a $84 \times 84$ grayscaled image. To capture movement in an observation, each observation consists of four consecutive frames. This is shown in figure3.9. This is done by stacking the four frames, thereby creating 4 colour channels for a CNN input. Since each step in the game gives a single new frame, each observation only has one of its four frames different from the previous observation.

An $84 \times 84$ frame means there are $84 \cdot 84 = 7056$ features per frame (thus for a single observation consisting of four frames, this would total to $84/cdot84/cdot4 = 28224$ features). Both paddles can move up and down to 64 spaces, giving $64 \cdot 64$ possible states when only looking at the paddle placement. Furthermore, the ball can be in any place in the $64 \times 84$ playing field. This gives a total state-space of $84 \times 84$, with a cardinality of $(64 \cdot 64) \cdot (64 \cdot 84) = 22020096$.

The agent, controlling one paddle, has six possible *actions* it can take. Most important are the actions *right* and *left*, moving the paddle up and down respectively. Another relevant action is *noop*, to not move the paddle. Other actions include an action to start the game (*fire*) and actions used for other Atari games, but meaningless in Pong (*rightfire* and *leftfire*).

An *episode* ends when one player reaches 21 points. After each step, the agent either receives a reward of 1 if it scored a point, $-1$ if the opponent scored a point, or 0 otherwise. At the end of the episode, the final score of the agent is calculated as the difference its own and the opponents points (e.g. a final score of 2 means it won the game $21 - 19$). Thus, a perfect score would be 21. A random agent mostly scores between

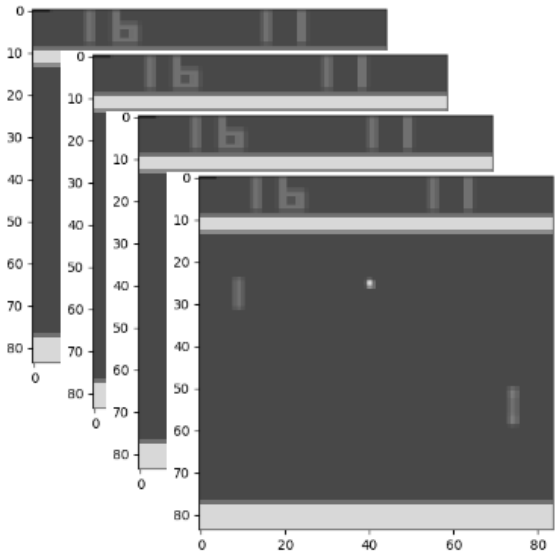


Figure 3.9: A single observation in OpenAI Gym's Pong: four consecutive stacked frames.

$-21$ and $-18$.

#### 3.1.3.1 Agents setup

The same agents are used for experiments in this environment as in the Starcraft II environment, excluding the DeepMDP agent due to time limitations. Again, we only give a general overview of the agents' architectures, referring to appendixA section A.2 for more details.

The baseline agent receives a $4 \times 84 \times 84$ state observation input: four consecutive $84 \times 84$ stacked frames. This totals to 28224 features per observation. All state-space dimensionality reduction methods, project a single $84 \times 84$ frame down to $42 \times 42$, thus reducing the number of features down from 7056 to 1764. Thus, agents using a dimensionality reduction method, receive a $4 \times 42 \times 42$ state observation, totaling to 7056 features per observation.

**Baseline agent**

The baseline agent is a standard DDQN RL agent that does not use any dimensionality reduction and is extended by the other agents.

Its policy network, representing the Q-function, consists of three convolutional layers and two linear layers. The first conv layer has a kernel size of 6 and a stride of 3, the second conv layer a kernel size of 4 and stride of 2, and the final conv layer a kernel size of 3 and stride of 1. These settings affect the output shape of each layer. For this, the 4 colour channels are irrelevant. Looking at a single colour channel, its $84 \times 84$ input is first downsampled to $27 \times 27$, then to $12 \times 12$ and finally to $10 \times 10$. The final conv

layer's output is flattened before going into the first linear layer. The final linear layer has an output of 6, corresponding to the action-space.

**PCA agent**
The PCA agent uses PCA to reduce the dimensionality of a single frame from $84 \times 84$ to $42 \times 42$. To do this, the observation input must first be flattened, and its output must be reshaped to $42 \times 42$. The final observation shape is thus $4 \times 42 \times 42$.

The policy network, representing the Q-function, is identical to the baseline agent. Now that we have a $42 \times 42$ input per colour channel, the convolutional layers in the policy network downsample the input first to $13 \times 13$, then to $5 \times 5$ and finally to $3 \times 3$.

The PCA component is trained on 40.000 previously stored frames, which corresponds to roughly 40 episodes, but were taken randomly across 300 episodes. These have been gathered using the baseline agent. The first 1764 principal components in our PCA (representing a $42 \times 42$ dimensional observation), capture roughly 99% of the variance of the original data. Since we are already using (gray)scaled data, we do not use a scalar for the PCA as mentioned in 2.2.1.

**Pre-trained autoencoder agent**
The autoencoder is pre-trained on 332.000 observations, taken from 300 episodes. After this its encoder is used by the agent to reduce the dimensionality of a single frame. This, like the PCA agent, results in a observation of $4 \times 42 \times 42$.

The encoder and decoder of the autoencoder are convolutional neural networks. The encoder uses three convolutional layers: the first has a stride of 2, which reduces the dimension to $42 \times 42$. The decoder, which tries to reconstruct the original data, uses two convolutional layers. The first layer is a transposed convolutional layer with a stride of 2, to bring the dimensions back to $84 \times 84$. The second layer is a regular convolutional layer.

**Online trained autoencoder agent**
This agent has the exact same design as the pre-trained autoencoder agent. As mentioned, the only difference is the moment of training the autoencoder: instead of training on previously stored observations, we are now training the autoencoder simultaneously with the RL agent.

I guess "online" is not strictly correct

### 3.1.4 Remarks on method

There are two methods we would like to shortly mention that we did not use in our experiments. Firstly, another popular method to reduce dimensionality of data is *t-distributed stochastic neighbor embedding* (t-SNE) [10]. Though widely used to reduce the dimensionality of data, it is unsuitable for our purposes. This has to do with the way t-SNE works. To calculate the data points in the lower dimensional space, it first calculates the Euclidean distance between all data points, which is transformed to a probability distribution. From this it follows that the data projection to a lower dimensional space is inherently dependent on the input data. In other words, we cannot train t-SNE on training data and then apply its transformation on new data. This makes it unusable

for us, since we need to be able to transform a single observation (and therefore need to be able to separate the training process and the transformation process).

A second method we did not use in our experiments is *Max-pooling*. Max-pooling is a type of layer in a CNN that downsamples its input. However, compared to using a convolutional layer with a stride of 2 as used in our experiments, "when pooling is replaced by an additional convolution layer with stride $r = 2$ performance stabilizes and even improves on the base model" [23].

## 3.2 Results

In this section we will show and discuss the results from the experiments mentioned in section 3.1.1. We will start by showing the results from the Starcraft II environment in section 3.2.1, before showing the results of the OpenAI Pong environment in section 3.2.2. These sections will focus on the research sub-questions: what are the effects of using PCA, autoencoders and DeepMDPs for state-space dimensionality reduction on reinforcement learning agents? They will also include a discussion of the results and an examination of how the different state-space reduction methods led to their results. In section 3.2.3 we will give a general discussion of all results to answer the main research question: what is the effect of state-space dimensionality reduction on reinforcement learning?

### 3.2.1 Research results: Starcraft II

The results from running the different agents in the Starcraft II minigame MoveToBeacon, can be seen in figure 3.10. For each agent, the score, i.e. reward, per episode is shown, as well as an average score per 30 episodes. Again, a scripted agents scores around $25 - 30$ per episode, and an agent following a random policy around $0 - 3$.

As can be seen in subfigure 3.10a, the baseline agent converges to a policy scoring around 19 per episode. This policy is reached after roughly 600 episodes. Already after episode 300 does it oscillate around scoring $17 - 20$ while also sometimes having an episode score of around 10. After this it still needs another 300 episodes to get to a more consistent policy.

The results for the PCA agent can be seen in figure 3.10b. No matter what neural network architecture used in the agent (having used multiple different linear network architectures and different CNN architectures), it always remained at a policy performing at the level of a random policy, i.e. scoring around $0 - 3$ per episode.

The results for the agent using a pre-trained autoencoder, in figure 3.10c, show that this agent performs a little better than the baseline agent. Not only does it converge to a slightly better policy scoring around 21 per episode, it also converges quicker; it is already consistent after roughly 400 episode, instead of 600.

Compared to the pre-trained autoencoder agent, the agent using an untrained autoencoder that is being trained while the agent is trained, performs a little worse. It converges after roughly 500 episodes to a policy around $19 - 21$. Not only does it take

longer to converge and converges to a slightly worse policy, it is also less consistent. Even after 400 it still has episodes scoring around 13. This shows it is less consistent than its pre-trained counterpart.

Lastly we have the DeepMDP agent. After 130 episodes, this agent suddenly jumps up from a random policy to scoring around 15. However, after several episodes it starts going back down, alternating between scoring at a random policy and scoring around 7, until finally reaching a random policy again. Furthermore it takes a lot longer to train. For the other agents, each episode took only a few seconds, whereas each episode in the DeepMDP agent takes 2.5 minutes.

#### 3.2.1.1 Discussion

TODO

##### 3.2.1.1.1 PCA Agent: losing all spatial information

The first notable result is the PCA agent giving a policy that remains at a policy at the level of a random policy during its 800 episode training, therewith performing way worse than the other agents. A possible explanation might be that the PCA reduction loses (too much) spatial information. Although PCA can be used for image compression, where spatial information is retained, it is a lot more lossy than for instance an autoencoder.

Citing

Figure 3.11 shows an example of the PCA transformation on an observation. figure 3.11a shows the original observation that the agent would get from the environment. Figure 3.11b shows the latent representation after using PCA for dimensionality reduction. as mentioned before, this uses 256 principal components, capturing roughly 96% of the original data. Here we can see that no spatial information is retained, making it impossible for the agent to train a meaningful policy on.

Furthermore, we also tested a different PCA setup where we used all 1024 principal components. Here, we again fitted the PCA on the same observations but keeping all principal components, giving a new $32 \times 32$ observation (and therefore not doing any dimensionality reduction). Transformation of an observation gave similar results: spatial information was lost. The problem therefore lies in the fitting process. A possible explanation for the bad transformation performance, might be that a single observation is very sparse. Even though the PCA is fitted on 240.000 observations, each single observation is very sparse and at the same time, there are a lot of different possible observations due to the large state-space (see section **??**). This combination might lead to the PCA not being able to generalize well enough.

##### 3.2.1.1.2 Autoencoder agents analyses: outperforming baseline agent

Another interesting result is the performance of the pre-trained autoencoder agent. Not only does it match the baseline agent's performance, it even slightly surpasses it: it finds a slightly better policy (scoring around 21 per episode on average, versus 19), and more quickly converges to a consistent policy (after 400 episodes versus 600). Its better

performance is despite the agent using imperfect information, while the baseline agent uses complete information.

Its performance is highly dependent on the quality of the output of the autoencoder, which is known in the discussion of the experiments in the second environment, Pong 3.2.2.1. To get an understanding of why this agent achieves such good results, we will now examine the autoencoder in detail. We first show the results of training the autoencoder itself. As mentioned, it is trained on 240.000 observations, corresponding to 1000 episodes. Whereas each agent's training (except for the DeepMDP) took roughly 90 minutes, training the autoencoder itself on 1000 episodes worth of observations only took about 2.5 minutes. The loss history for the autoencoder can be seen in figure 3.12. After roughly $6000 \cdot 25 = 150.000$ observation it converges to its final loss.

To analyse the autoencoder, we show the latent representation of the autoencoder (given by the encoder) in figure 3.13. The original $32 \times 32$ input for the encoder is given in figure 3.13a. Like before, the yellow octagon is the beacon, the blue square is the army unit controlled by the agent, and purple parts are the empty tiles. The latent representation is given in figure 3.13b. Here we can see that the autoencoder compresses the observation down to a similar but smaller observation: each latent cell seems to represent a block of cells of the original data.

Such a projection onto a lower dimensional space, could create problems for the agent where high precision is involved, since the latent representation will lose at least some of the spatial information. However, in a setting like this, such lack of precision is not important due to the beacon encompassing several pixels. The agent only needs to reach some part of this beacon, thus there is room for locational variance.

To analyse the autoencoder further, a correlation matrix is given in figure 3.15. Based on 60.000 state observations, it shows the correlation between the features of the original state observations (i.e. the input for the autoencoder) and the reduced state observations (i.e. the output of the autoencoder). The process of creating this matrix is shown in figure 3.14. The original data has dimensions of $32 \times 32$, whereas the reduced data has dimensions of $16 \times 16$, resulting in 1024 and 246 features respectively. The 256 latent features are shown on the y-axis, and the 1024 original features are shown on the x-axis in figure 3.15. Dark/black and light/beige colouring mean a high correlation (negative and positive correlation respectively), whereas a red colouring means no correlation and the white space means there was to too little variance to calculate a correlation. The latter case simply follows from the beacon and army unit not visiting these parts of the map enough times during the used episode observations, which results in too little variance for a correlation to be calculated.

What can be seen from this matrix, is that each cell in the reduced data grid, correlates to a block (a few adjacent rows and columns) of cells of the original data grid. This explanation can be abducted from each feature of the reduced data being highly correlated to a few features of the original data, then being uncorrelated for 32 features, after which highly correlated features are found again. This jump of 32 features corresponds to a jump of one row in the original data grid. Thus, a reduced feature correlates to a block of the original features.

This is better visualised by taking the correlation matrix of a single latent feature. The process of this is also shown in figure 3.14. We choose a latent feature and take the correlation results of that feature from the correlation matrix, simply by taking the corresponding row from the matrix. We now have the correlation data of a single latent feature with all original features. We then reshape this data to a $32 \times 32$ shape for visualisation. In figure 3.16 we have done this for latent feature 68. In subfigure 3.16a we can see where latent feature 68 lies in the $16 \times 16$ latent space: in the 4th column of the 4th row. In subfigure 3.16b the correlation matrix for this latent feature with all original features is shown.

Here we can see clearly the high correlation of the latent feature with a hexagonal shape of original features. This relates to the hexagonal shape of the beacon, the most important part of the observation. Thus, the autoencoder is taking in the values of a roughly 14 by 14 octagonal shape of pixels to get the value for a latent feature, thereby compressing the image to a lower dimension.

This autoencoder agent not only outperforms the baseline agent, it also (less surprisingly) outperforms the online trained autoencoder agent. This latter agent converges to a slightly worse policy and takes more episodes to get there, while remaining not very consistent (sometimes scoring only about 13 points in an episode). This can of course be explained by the fact that this agent not only trains a policy network, but also, separately, the autoencoder. This means that for many episodes, the policy network is trained on a rather imperfect representation; the pre-trained autoencoder needed roughly 150.000 observations before getting close to its final loss. Still, it got to a policy similar to the baseline agent (even scoring slightly better on average) in a similar number of episodes; the main difference being that the baseline agent is much more consistent.

Besides outperforming the online trained autoencoder agent, the pre-trained agent has another advantage. Since the autoencoder can be trained separately from the agent, we can reuse the autoencoder for different agents acting in the same environment but solving a different RL problem. This allows for the possibility of training an autoencoder once, after which different agents can be trained on less features, allowing for less computation cost to train each agent. This is also substantiated by the autoencoder taking little time train.

#### 3.2.1.1.3 DeepMDP agent: unable to balance multiple loss calculations

Lastly, the DeepMDP agent is unable to converge. After roughly 120 episodes, it gets to a decent policy, scoring around 15 per episode. Quickly after though, it drops down to scoring between 0 and 10, before dropping further down after 400 episode to scoring between 0 and 5, only slightly better than a random policy.

The DeepMDP agent seems to have trouble with balancing the multiple losses being calculated; a recurring problem that is also described in the paper introducing DeepMDPs [5]. When after approximately 130 episodes the score falls from roughly 15 down to around 5, the total loss being calculated decreases significantly. However, it is only the gradient penalty for the auxiliary objective, i.e. the gradient penalty for the transition cost network, that has a lower loss value. All other losses, i.e. the remaining gradient

penalties, the transition cost and the DDQN loss, are all increased. Most notably the DDQN loss is increased, resulting in worse Q-function network, thus giving lower scores.

This trouble with balancing the loss is further demonstrated at 240 episodes into training. At this moment, the DDQN loss, as well as the gradient penalties for the Q-function and encoder networks, are all decreasing in value. This results in higher average episode scores. However, after this they start increasing again, leading to a further drop in episode scores around 330 episodes.

### 3.2.2 Research results: OpenAI Pong

The results for the different RL agents in OpenAI Pong can be seen in figure 3.17. Like with Starcraft II, each agent's score is shown per episode, as well as a 30 episode average. A perfect agent would score 21 points per episode, whereas a random agent will score mostly between $-21$ and $-18$.

The results for the baseline agent, in subfigure 3.17a, show that it converges to a policy scoring between 12 and 21 per episode (meaning it wins the game having scored 21 points, while its opponent scores 0 to 9 points), with an average score around 18. It converges to this policy after roughly 220 episode.

The results for the PCA agent can be seen in figure 3.17b. This agent trains significantly worse than the baseline agent. Though it eventually does improve compared to a random policy, it takes a lot of episodes before it does improve and it remains incredibly inconsistent.

The pre-trained autoencoder agent performs a lot better than the PCA agent, yet slightly worse than the baseline agent. This can be seen in figure 3.17c. Though the pre-trained autoencoder agent converges slightly quicker than the baseline agent, taking roughly 180 episodes instead of 200, it converges to a worse policy. Most notably, it is a lot less consistent: its average score oscillates between 11 and 16, whereas the baseline agent's average is around 18. This is mostly the result of consistently having occasional episodes scoring only slightly above 0.

The online trained autoencoder on the other hand, is never able to train and remains at the level of a random policy.

#### 3.2.2.1 Discussion

TODO

##### 3.2.2.1.1 PCA agent: losing spatial information

We will again start by examining the PCA agent. Like in the Starcraft II environment, the PCA agent converges to a very sub-optimal policy. Though after 300 episode better than a random policy, the PCA agent performs very inconsistently, having both episodes of scores of 21 and of $-21$.

Like with the Starcraft II environment, we are working with spatial information. This information is not retained, despite using 1764 principal components to transform a single observation frame, capturing roughly 99% of the variance of the original input.

This is shown in figure 3.18, which shows a single observation frame and its latent representation through PCA. We can see that again (like in Starcraft II) the spatial information is lost in the latent representation , making it difficult for the agent to train. A possible explanation for learning a better than random policy might be the absence of enough stochasticity in this environment, making it possible to train (at least to a certain level) merely by directly mapping a state to an action, instead of learning generalisations. However, the results clearly show that the agent is not able to train well using latent representations that lose spatial information.

Citing, same as in pysc2

#### 3.2.2.1.2 Pre-trained autoencoder agent: slightly worse than the baseline agent

Unlike in the Starcraft II environment, the pre-trained autoencoder agent performs slightly worse than the baseline agent in Pong. Though getting similar scores for most episodes, the autoencoder agent is much less consistent, which is shown by the oscillating average score and the consistent occasional low score episodes. However, the agent still converges to a rather decent policy despite using $42 \times 42$ latent representations. Furthermore, it also converges slightly faster than the baseline agent. This might be because the policy network can train on a smaller state-space.

To analyse and validate the results of the autoencoder, we first show the output of the encoder, i.e. the latent representation, in figure 3.20b. What can be seen here is a compression of the data similar to the autoencoder in Starcraft II. Each pixel in the latent representation correlates to a block of pixels of the original data. Thus, a pixel's value in the latent representation will reflect the values of a block of pixels in the original image, and therewith compress the data to a lower dimensional space.

This can explain why the agent learns a sub-optimal policy: the lower dimensional space creates slight imprecisions with regards to the exact locations of the ball and the paddles. In the Starcraft II environment this did not create problems since the beacon spans a relatively large space. However, in Pong the ball is only four pixels large and its precise location relative to paddle is very important for deciding where to move to paddle to.

To further validate the autoencoder agent's results, we trained two other autoencoders on a lower number of frames; the autoencoder used so far will be named *autoencoder 1* and the two remaining autoencoders will be named *autoencoder 2* and *autoencoder 3*, where autoencoder 3 is trained on the fewest number of frames. These two additional autoencoders are used in two ways. First, we take the trained autoencoder agent (i.e. trained on the original autoencoder: autoencoder 1) and replace autoencoder 1 with autoencoders 2 and 3. This way we can see how much the agent depends on the specific encoding of the autoencoder it was trained on. Secondly, we will train two additional autoencoder agents on the other two pre-trained autoencoders. This way we can compare the effect of using a less accurate autoencoder on the RL agent.

The losses of the autoencoders (computed by the difference between the input image and the latent representation, as explained in section 2.2.2) during training can be seen in figure 3.19. An untrained autoencoder has a loss of roughly 1500. Autoencoder 1

was trained on 340.000 frames and ended in a loss of 0.9. Autoencoder 2 was on 25.000 frames and ended in a loss of 30.7. Finally, autoencoder 3 was trained on 10.000 frames and ended in a loss of $86,2$.

The effect of this difference in loss and number of training frames, is shown in figure 3.20. Here we can see the latent representation for each autoencoder. In other words, they show the output of the encoder, for a given frame. It can be clearly seen that, although each latent representation is similar, the better autoencoder gives a more precise representation of the original image. Thus, it retains more spatial information and should result in a better autoencoder agent.

This is indeed the case as can bee in figure 3.21: the better the autoencoder being used by the RL agent, the better the agent trains. We have already looked at the results for autoencoder 1, in figure 3.21a, and how it compared to the baseline agent. We can see in figure 3.21b that using autoencoder 2, results in a worse RL agent. It takes more episodes to converge compared to the agent using autoencoder 1 (180 and 250 episodes respectively) and converges to a less optimal policy. In particular, it is a lot less consistent, often scoring even below 0 (thus losing the Pong game). The agent using autoencoder 3, shown in figure 3.21c performs even worse than this. It takes about 300 episodes before it starts getting to a policy that is better than a random agent, and after 400 gets to a very inconsistent policy: it both has episodes of 21 and of $-21$. It's results are very similar to that of the PCA agent, which lost all spatial information in its latent representation. Thus, it is clearly seen that the less accurate the autoencoder, the less the RL agent is able to train on its latent representation, thereby showing the importance of a well trained autoencoder when used for state-space dimensionality reduction.

#### 3.2.2.1.3 Online trained autoencoder agent:

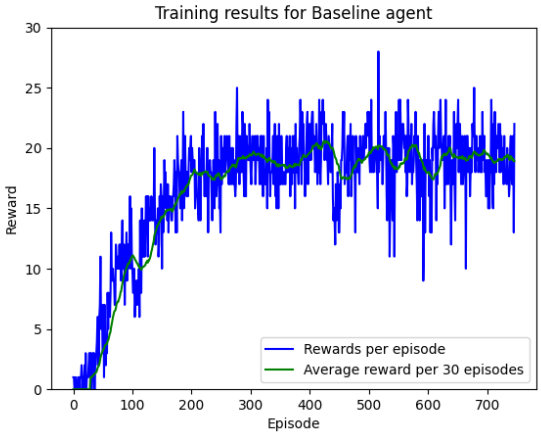
### 3.2.3 Results discussion

As we have seen in the results from our experiments in Starcraft II and OpenAI Pong, the effect of state-space dimensionality reduction on reinforcement learning depends on the technique and environment used.

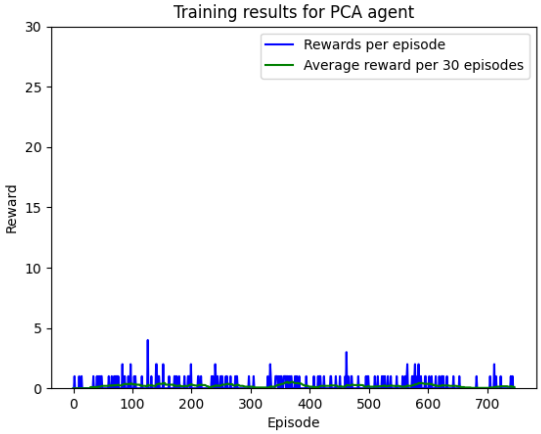
Both PCA and DeepMDP did not work well in both our environments and resulted in RL agents having difficulty training, or not training at all. For PCA, this was the result of losing spatial information in its latent representation. Curran et al. showed that it is indeed possible to get an RL agent to learn on the latent space of PCA [3]. This they did on non-spatial observations. Thus, the effectiveness of using PCA for state-space dimensionality reduction depends on the type of observations being used.

Although the DeepMDP agent was able to get to a decent policy in our Starcraft II environment, it had difficulty with balancing the different loss calculations. This is also mentioned in the paper introducing the DeepMDP architecture [5]. This balancing issue resulted in network updates that lowered one loss calculation, but raised another. Hence, the policy network was not able to train well. Furthermore, the DeepMDP took a lot more time to train per episode than all other agents, due to an additional network being used.

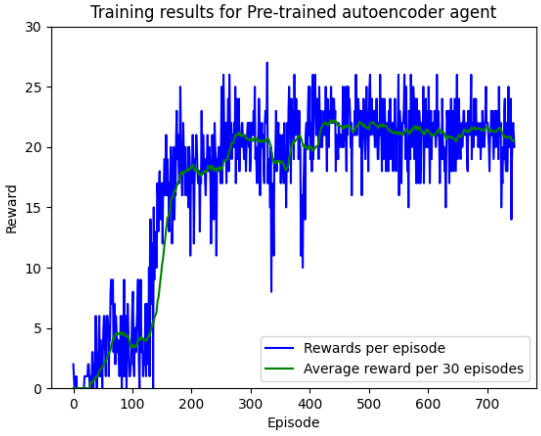
The use of the autoencoder was the most promising method for state-space dimensionality reduction in reinforcement learning. In both the Starcraft II and the Pong environment, the autoencoder was able to project the data to a space that is 4 times lower than the original state-space. However, some spatial information does get lost in its compression. For Starcraft II this was not a problem due to the beacon being fairly large. Here, the pre-trained autoencoder agent was able to train quicker to a better policy than the baseline agent. This is a result of the policy network being trained on smaller input, having to consider fewer states. However, in Pong the location of the ball and the paddle need to be precise in order to get an optimal agent. The imprecise spatial-information therefore resulted in a pre-trained autoencoder agent that performed slightly worse than the baseline agent. This was more clearly seen in the online trained autoencoder where the RL agent was not able to learn at all.




(a) Results for the baseline agent.

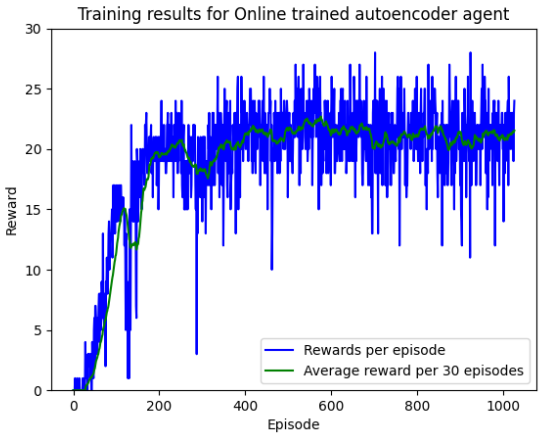



(b) Results for the PCA agent.

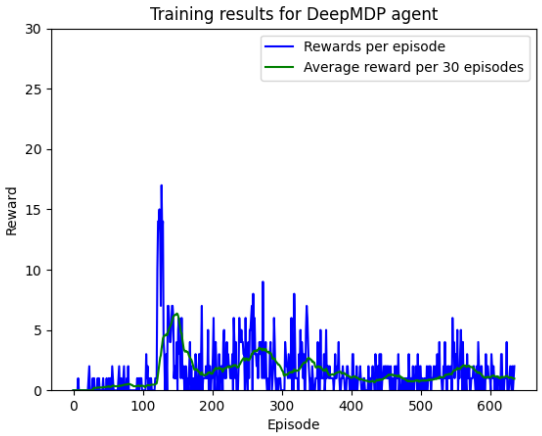



(c) Results for the pre-trained autoencoder agent.

Figure 3.10: Results of the different RL agents in Starcraft II.

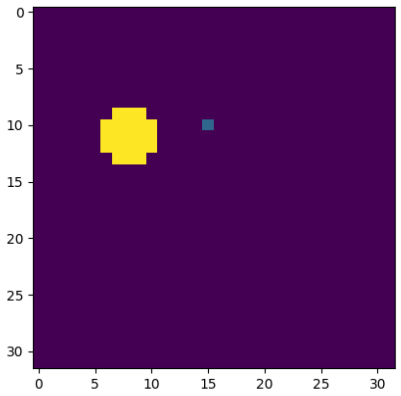



(d) Results for the online trained autoencoder agent.

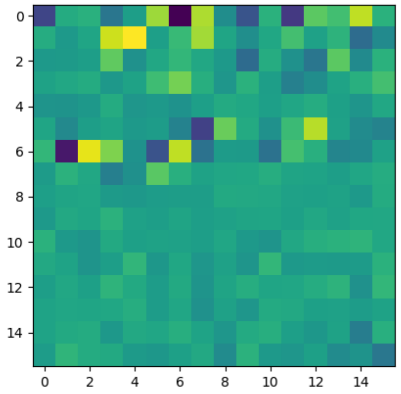



(e) Results for the DeepMDP agent.

Figure 3.10: Results of the different RL agents in Starcraft II(cont.).




(a) The original observation, i.e. the input for the PCA transformation.




(b) The latent representation given by the PCA transformation

Figure 3.11: Latent representation a state observation using PCA.

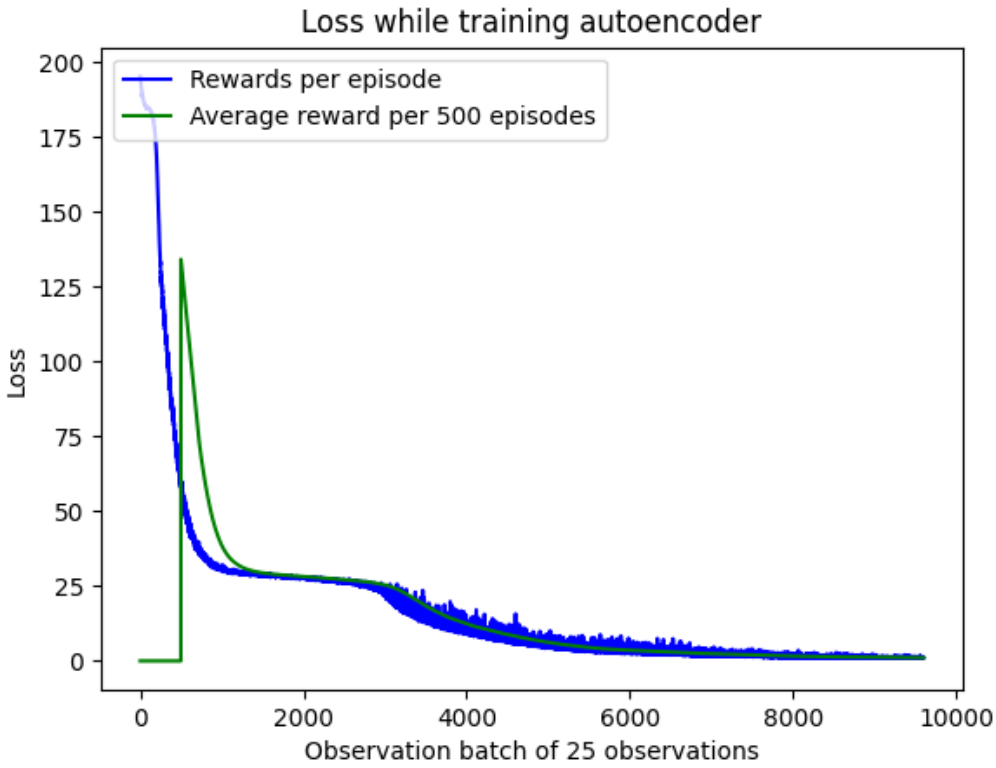
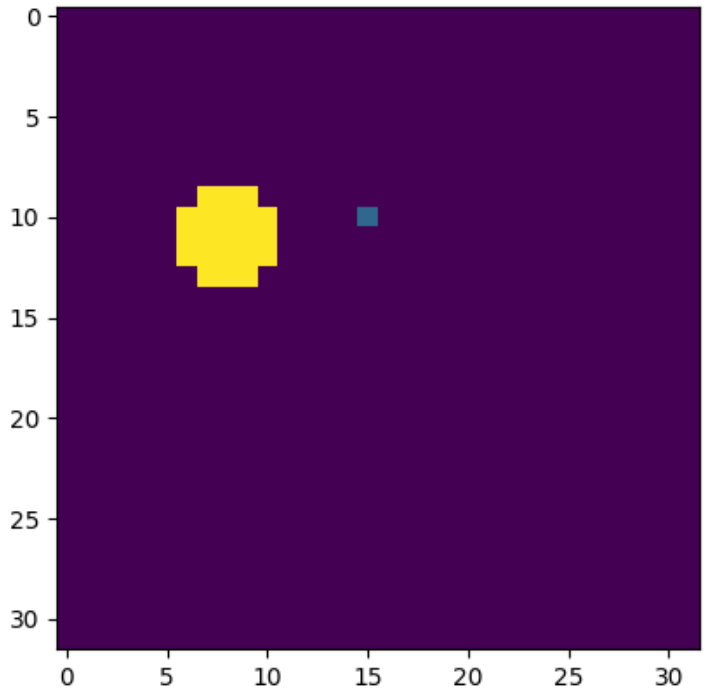
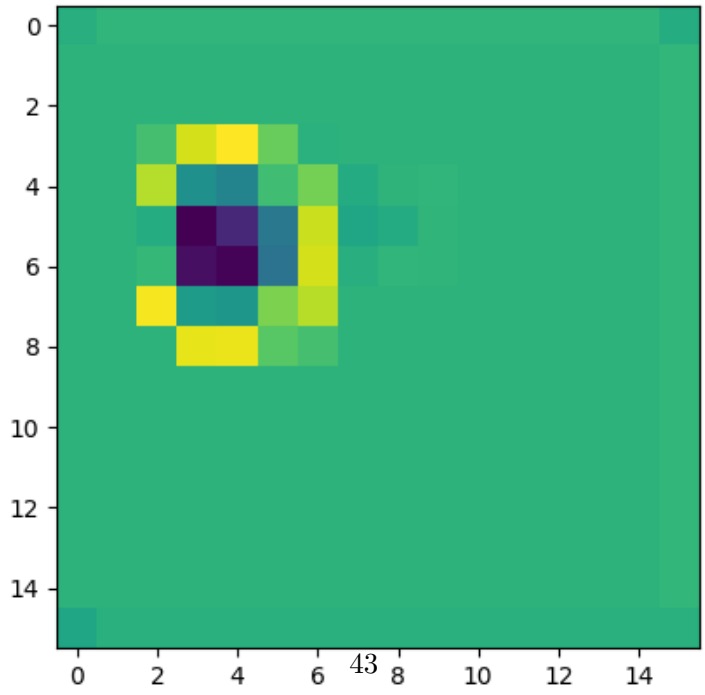



Figure 3.12: Losses per 25 observations for training the autoencoder on 240.000 state observations.




(a) The original observation, i.e. the input for the autoencoder.




(b) The latent representation of the autoencoder, i.e. the output of the encoder.

Figure 3.13: The latent representation of the autoencoder.

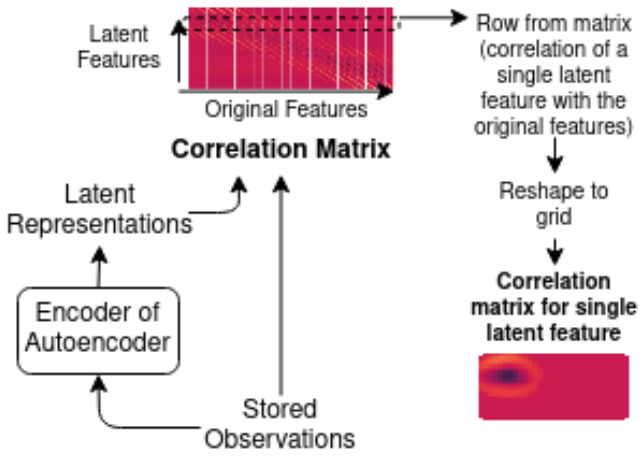



Figure 3.14: Process of creating a correlation matrix for the latent features with the original features, and creating the correlation matrix of a single latent feature with the original features.

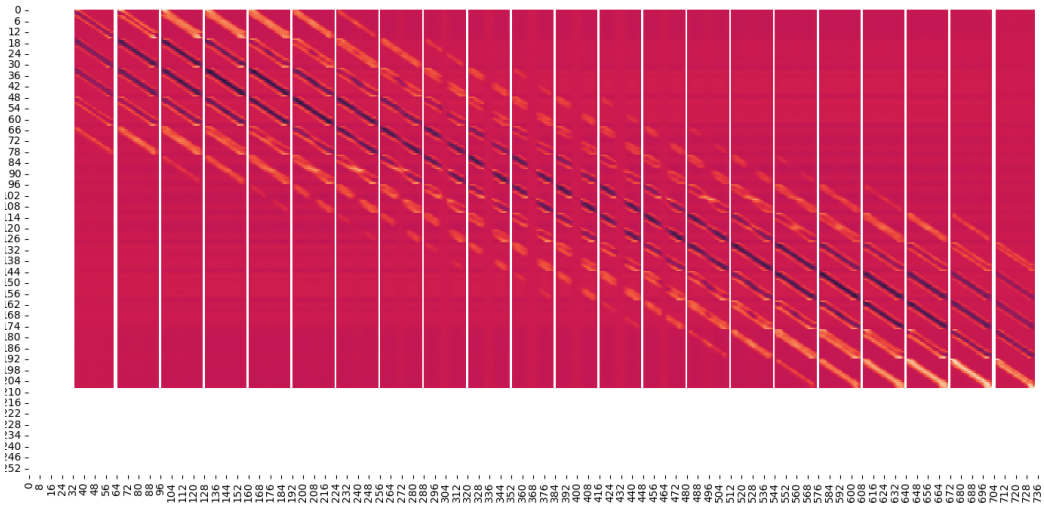
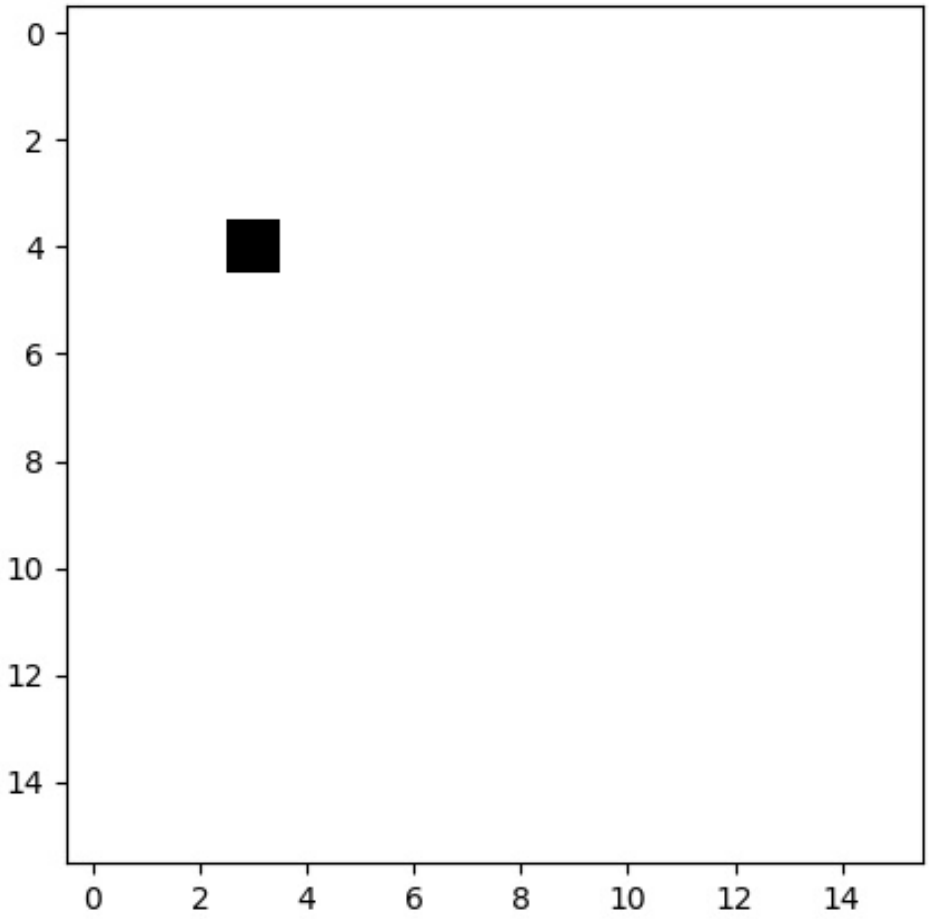
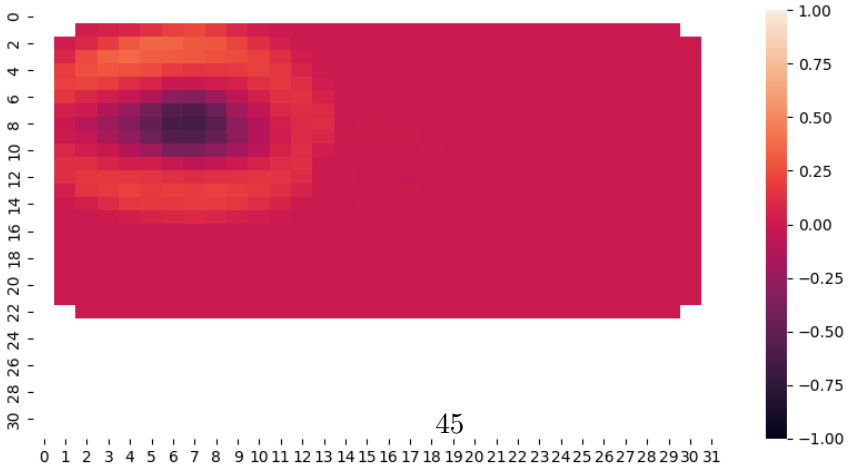


Figure 3.15: Correlation matrix for the autoencoder used in the pre-trained autoencoder agent, based on 60.000 state observations. The x-axis contains the features of the original observations, and the y-axis contains the features of the reduced state observations.



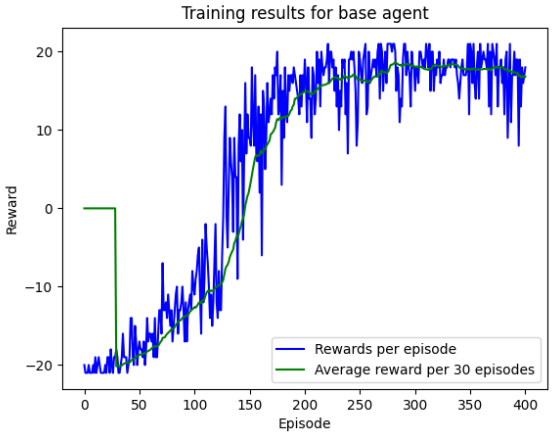

(a) The latent feature that was used.



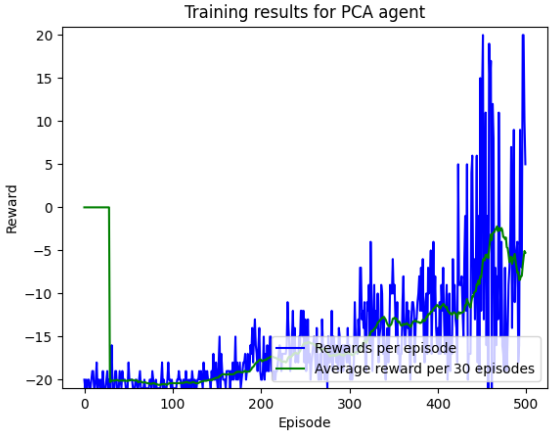



(b) The feauture map of the first convoluational layer, showing the output of its 32 channels.

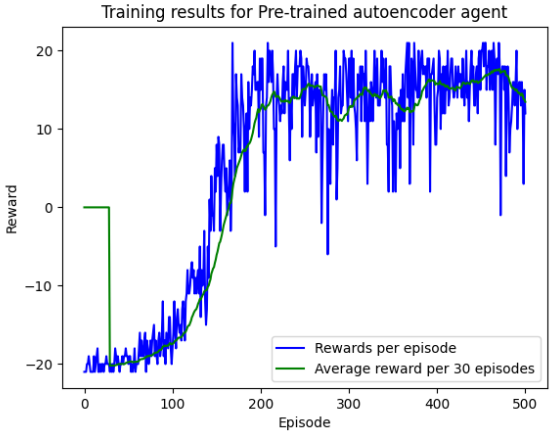
Figure 3.16: The correlation matrix for latent feature with the original features.




(a) Results for the baseline agent.

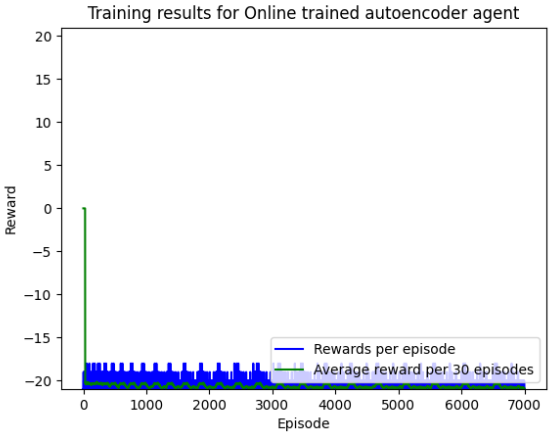



(b) Results for the PCA agent.



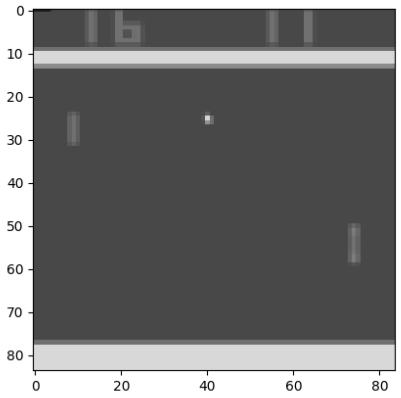

(c) Results for the pre-trained autoencoder agent.

Figure 3.17: Results of the different RL agents in Pong.

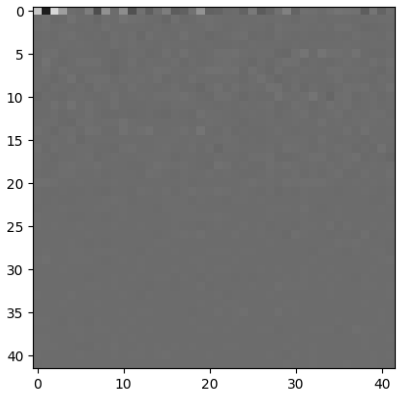



(d) Results for the online trained autoencoder agent.

Figure 3.17: Results of the different RL agents in Pong(cont.).




(a) The original observation, i.e. the input for the PCA transformation.




(b) The latent representation given by the PCA transformation

Figure 3.18: Latent representation of a state observation using PCA.

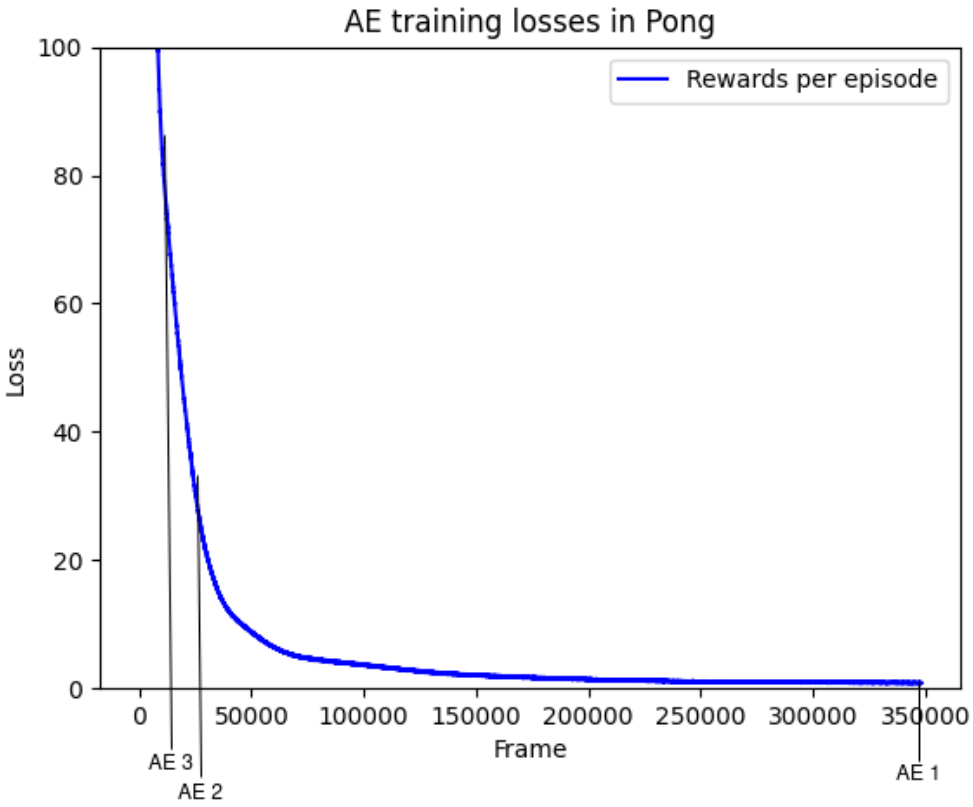
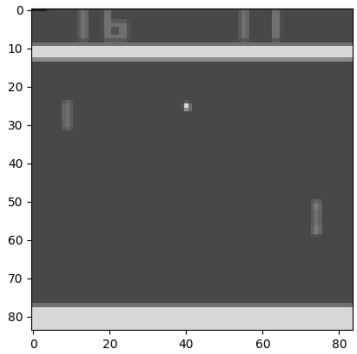
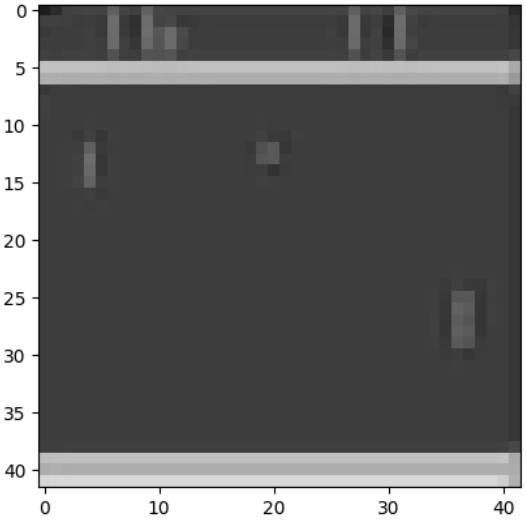


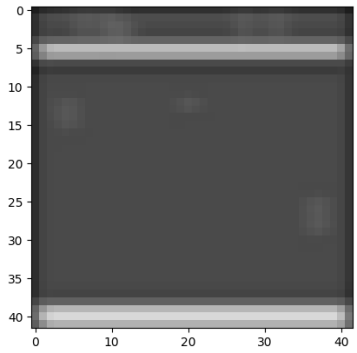

Figure 3.19: Losses frame for training three autoencoders in Pong. Autoencoder 1 trained on 340.000 frames to a loss of 0.9, autoencoder 2 trained on 25.000 frames to a loss of 30.7 and autoencoder 3 trained on 10.000 frames to a loss of 86.2.




(a) The original observation frame, i.e. the input for the autoencoder.

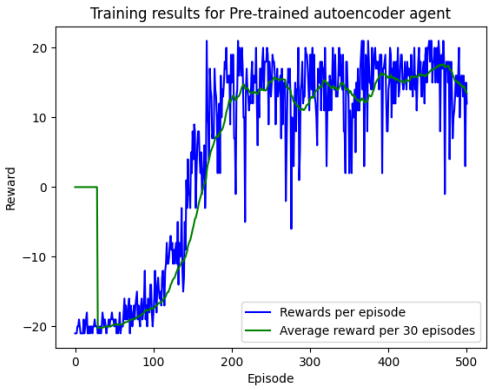



(b) The output of the encoder of autoencoder 1.

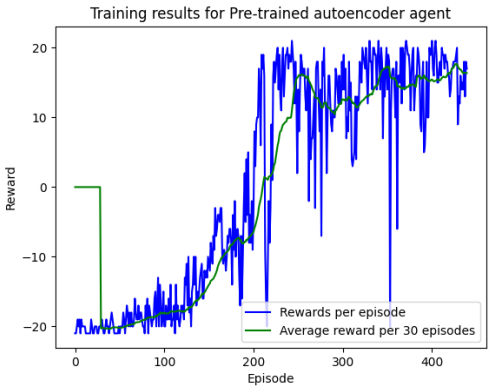



(c) The output of the encoder of autoencoder 2.

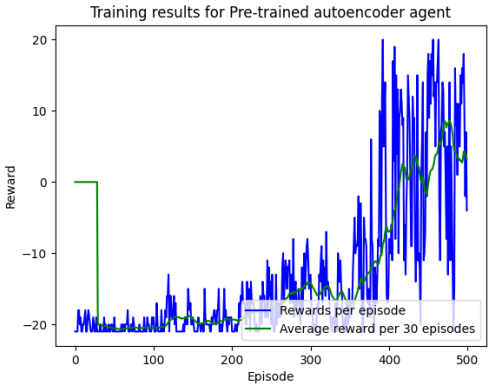





(a) Training results for RL agent using pre-trained autoencoder 1 in Pong.




(b) Training results for RL agent using pre-trained autoencoder 2 in Pong.




(c) Training results for RL agent using pre-trained autoencoder 3 in Pong.

Figure 3.21: Latent representations of three different autoencoders for an observation frame in Pong.

# Chapter 4

# Related work

State-space dimensionality reduction has been a topic of interest in RL for several years. Curran et al. [3] used PCA to reduce the dimensionality of the state-space in a Super Mario environment. They found that with the right number of principal components, an agent using PCA was able to converge to a better policy and need less episodes than an agent using the full observation. Unlike the Starcraft II environment used in our research having image/grid based observations, their states and observations are comprised of non-spatial related variables; to, for instance, denote the presence of one or more enemies within one gridcell distance of Mario's position, they use one variable with $2^8$ possible values (0-255). Hence we are using PCA in a different setting, with an observation format more commonly used in modern RL. Furthermore they use the Q-learning algorithm (mentioned in section 2.1.3) which is not a state-of-the-art learning algorithm anymore.

Research has also been done with regards to using autoencoders for state-space-dimensionality reduction in RL. Lange and Riedmiller in 2010 [16] used visual data as observations and found that the agent was able to find an optimal policy using lower dimensional data from the autoencoder. They did not compare the performance of this agent with any other agent. Furthermore, their system only has 31 possible states, which is very limited; in contrast, our environment has a total of 731.187 possible states.

In 2016, Van Hoof et al. [25] used an autoencoder to project noisy sensor data unto a lower dimensional space. They found that the agent using the autoencoder was far better able to find a good policy than the agent using full observations. This was explained as the agent being too sensitive to noisy data, hence being unable to train even decently. Our research in contrast explores an environment where the baseline agent is able to train to a good policy.

In 2019, Prakash et al. [22] also used an autoencoder on visual data to project the observation data onto a lower dimensional space. They found that an agent using the autoencoder far outperformed the baseline agent. Again though, the baseline agent did not find a decent policy, though the authors claim that with enough episodes it would have.

Another paper in 2019 by Gelada et al. [5] compared using a DeepMDP with using

an autoencoder for state-space dimensionality reduction. They found that in simpler environments a DeepMDP can find better representations on lower dimensionality space than an autoencoder. Simultaneously though, they also find that in more complex environments, like Atari games, DeepMDPs can have trouble finding a good representation and that its loss can be difficult to optimize. They also find that a DeepMDP agent generally outperforms a baseline agent.

# Chapter 5

# Conclusions and future research

In this chapter you present all conclusions that can be drawn from the preceding chapters. It should not introduce new experiments, theories, investigations, etc.: these should have been written down earlier in the thesis. Therefore, conclusions can be brief and to the point.

*future work* - ae met lagere dimensions vergelijken - env zonder rgb maar directe features pakken voor betere pca vergelijking - voordelen dim red bekijken: nu focus op haalbaarheid van dim red, dus zoveel mogelijk gelijek agent architectures. Maar kan kijken naar verglijkingen tussen compleet vershcillende agents als gevolg van dim red (dus bv kijken training tijd en training eps omlaag). - ae met grotere CNN architecture die pas in latere layers downsampled zodat je minder spatial information verliest

# Bibliography


[1] Blizzard. Blizzard/s2client-proto: Starcraft II Client - protocol definitions used to communicate with StarCraft II.

[2] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[3] William Curran, Tim Brys, Matthew Taylor, and William Smart. Using pca to efficiently represent state spaces. 05 2015.

[4] Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning, 2018.

[5] Carles Gelada, Saurabh Kumar, Jacob Buckman, Ofir Nachum, and Marc G. Bellemare. Deepmdp: Learning continuous latent space models for representation learning. *CoRR*, abs/1906.02736, 2019.

[6] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[7] Ishaan Gulrajani, Faruk Ahmed, Martín Arjovsky, Vincent Dumoulin, and Aaron C. Courville. Improved training of wasserstein gans. *CoRR*, abs/1704.00028, 2017.

[8] Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus), 2020.

[9] G. E. Hinton and R. R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006.

[10] Geoffrey Hinton and Sam Roweis. Stochastic neighbor embedding. In *Advances in Neural Information Processing Systems*, volume 15, pages 833–840. MIT Press, 06 2003.

[11] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, page 448–456. JMLR.org, 2015.

[12] Ali Jafari, Ashwinkumar Ganesan, Chetan Sai Kumar Thalisetty, Varun Sivasubramanian, Tim Oates, and Tinoosh Mohsenin. Sensornet: A scalable and low-power deep convolutional neural network for multimodal data classification. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 66(1):274–287, 2019.

[13] Beakcheol Jang, Myeonghwi Kim, Gaspard Harerimana, and Jong Kim. Q-learning algorithms: A comprehensive classification and applications. *IEEE Access*, PP:1–1, 09 2019.

[14] Ian Jolliffe. *Principal component analysis*. Springer Verlag, New York, 2002.

[15] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[16] Sascha Lange and Martin Riedmiller. Deep auto-encoder neural networks in reinforcement learning. In *The 2010 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, 2010.

[17] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. 12 2013.

[18] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, February 2015.

[19] Miguel Morales. Grokking deep reinforcement learning. 2020.

[20] Vinod Nair and Geoffrey Hinton. Rectified linear units improve restricted boltzmann machines vinod nair. volume 27, pages 807–814, 06 2010.

[21] Andrew Y. Ng. Feature selection, l1 vs. l2 regularization, and rotational invariance. In *Proceedings of the Twenty-First International Conference on Machine Learning*, ICML '04, page 78, New York, NY, USA, 2004. Association for Computing Machinery.

[22] Bharat Prakash, Mark Horton, Nicholas R. Waytowich, William David Hairston, Tim Oates, and Tinoosh Mohsenin. On the use of deep autoencoders for efficient embedded reinforcement learning. *CoRR*, abs/1903.10404, 2019.

[23] Jost Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller. Striving for simplicity: The all convolutional net. 12 2014.

[24] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. 2015. cite arxiv:1509.06461Comment: AAAI 2016.

[25] Herke van Hoof, Nutan Chen, Maximilian Karl, Patrick van der Smagt, and Jan Peters. Stable reinforcement learning with autoencoders for tactile and visual data. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3928–3934, 2016.

[26] Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John P. Agapiou, Julian Schrittwieser, John Quan, Stephen Gaffney, Stig Petersen, Karen Simonyan, Tom Schaul, Hado van Hasselt, David Silver, Timothy P. Lillicrap, Kevin Calderone, Paul Keet, Anthony Brunasso, David Lawrence, Anders Ekermo, Jacob Repp, and Rodney Tsing. StarCraft II: A new challenge for reinforcement learning. *CoRR*, abs/1708.04782, 2017.

[27] Chiyuan Zhang, Oriol Vinyals, Remi Munos, and Samy Bengio. A study on overfitting in deep reinforcement learning. 04 2018.

# Appendix A

# Appendix

Here we will lay out the details of the architecture and hyperparameter settings that were used in each agent mentioned in section 3.1. We start with the agents for the Starcraft II environment, then secondly the OpenAI Atari Pong environment.

## A.1 Starcraft II: RL agent architectures

In this section we will give the specific architecture and hyperparameter settings used for RL agents in the Starcraft II environment. We will start by showing the baseline agent, which uses an architecture and hyperparameters that are shared by all agents. For all other agents, each extending this baseline agent, we will only give the additional architecture and parameters. For an overview of how each agent works, we refer to sections 3.1.1 and 3.1.2.

### A.1.1 Baseline agent

**Neural network architecture**
The DDQN policy and target network for the baseline agent consists of a CNN with three convolutional layers. The first layer is a transposed convolutional layer [4] and the other two are regular convolutional layers. The transposed convolutional layer has the possibility to upscale the dimensionality, which is needed in agents using dimensionality reduction. These agents have their observation reduced from $32 \times 32$ to $16 \times 16$. This $16 \times 16$ observation is the input for their policy and target network. The output of these networks however need to be $32 \times 32$, capturing the action space. Although this upscaling is not needed for the baseline agent since it uses the full $32 \times 32$ observation as input, we still use it as the first layer in order to keep all agents' architectures as similar as possible.

Each layer uses a stride of 1, input padding of 1 and no output padding, keeping the dimensionality of the input the same. Furthermore, they also each have a kernel size of 3. The first layer has 32 output channels. The second layer has 32 input channels (following the output channels of its previous layer) and 32 output channels. The third and last

layer has 32 input channels (again following the output channels of its previous layer) and 1 output channel. Both after the second and the first layer, we use the activation function known as *ReLU* [20] whose output for each neuron is defined as the maximum of 0 and the input neuron.

When training the policy network, the optimization algorithm *Adam* is used, which is an extension on *stochastic gradient descent* [15]. For loss calculation we refer to section 2.1.3, more specifically algorithm 1.

**Hyperparameters**

The following hyperparameters are used in the baseline agent:

- Steps in between training the policy network: 4.
- Steps between updating the target network: 250.
- Optimizer learning rate: 0.0001.
- Discount factor: 0.99.
- Batch size for training the policy network: 256.
- Number of stored batches before training: 20.

Lastly, an $\epsilon$-greedy strategy is used for balancing exploration and exploitation. This means that a random number in $[0, 1]$ is chosen; whenever this is equal to or lower than the current epsilon value, a random action is chosen, and a greedy action otherwise. Epsilon decays from 1.0 to 0.1 in 100.000 steps spaced on a logarithmic scale. Each step taken by the agent corresponds to one epsilon decay step. The resulting decay of the epsilon value can be seen in figure A.1.

### A.1.2 PCA agent

The neural network architecture and hyperparameter settings of the PCA agent are almost exactly the same as the baseline agent (see appendix A.1.1). The only difference is that in the policy and target network, the first layer (the transposed convolutional layer) now has a stride of 2 and output padding of 1. This is to upscale dimensionality from the 16 observation (gotten after dimensionality reduction using PCA) to $32 \times 32$ representing the action space.

### A.1.3 Pre-trained and online trained autoencoder agent

Like with the PCA agent, the difference with the baseline agent with regards to the DDQN architecture is merely the first layer of the policy and target network of the agent: it now has a stride of 2 and output padding of 1 to upscale the dimensionality from its $16 \times 16$ input to $32 \times 32$ output. In both autoencoder agents, the autoencoder has the same architecture and hyperparameters.

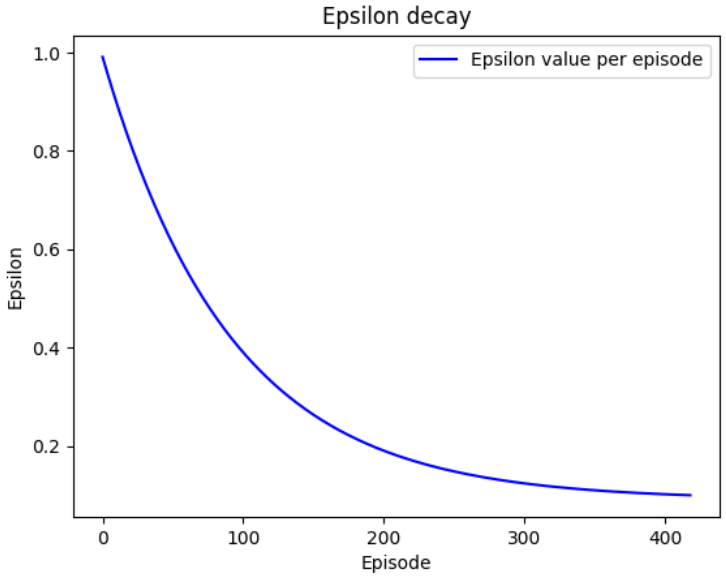



Figure A.1: The decay of the epsilon value per episode.

**Autoencoder neural network architecture and hyperparameters** The encoder of the autoencoder consists of two convolutional layers. The first layer has a stride of 2 to introduce dimensionality reduction from $32 \times 32$ to $16 \times 16$. The second layer has a stride of 1 to keep this $16 \times 16$ dimensionality. Both layers have a kernel size of 3 and an input padding of 1. The first layer has 1 input channel (since the observation input only has one channel) and 32 output channels. Hence, the second layer has 32 input channels and 1 output channel. After both layers, there is a *GELU* activation function, which is a nonlinear variant of ReLU [8].

The decoder consists of three layers. Because the decoder has to reconstruct the original input from the output of the encoder, the first is a transposed convolutional layer to upscale its $16 \times 16$ input back to $32 \times 32$. This is done using a stride of 2 and output padding of 1. It has 1 input channel (corresponding to the 1 output channel of the last encoder layer) and 32 output channels. The other two layers are regular convolutional layers, with stride 1 and no output padding. For the second layer, there are 32 input and output channels, and for the third and last layer there are 32 input channels and 1 output channel (corresponding to the 1 input channel of the first layer of the policy and target network). Each layer has a kernel size of 3 and input padding of 1. After the first and second layer there is a GELU activation function.

For training the autoencoder, the loss is calculated by the mean-scared-error between the original input and the reconstructed input. Furthermore, the Adam optimization algorithm is used, with a learning rate of 0.0001.

### A.1.4 DeepMDP agent

Compared to the baseline agent, the policy and target network of the DeepMDP agent have a prepended encoder. Furthermore it has an additional neural network for the transition loss (i.e. the auxiliary objective). Apart from these changes (including a different loss calculation) the only difference with the baseline agent is again the first layer of the target and policy network that now has a stride of 2 and output padding of 1 to upscale the dimensionality from $16 \times 16$ to $32 \times 32$.

**DeepMDP encoder and transition loss architectures and hyperparameters**
The encoder has the same architecture as the encoder in the autoencoder agents. It has two convolutional layers, with the first layer having a stride of 2 for dimensionality reduction. The first layer has 1 input channel and 32 output channels and vice versa for the second layer. Both have a kernel size of 3 and input padding of 1. After both layers the activation function GELU is used.

The transition loss neural network has only 1 layer, a convolutional layers. Most importantly, it has 1024 output channels, corresponding to the $32 \times 32$ action space. It has a stride of 1 and kernel size 2.

Besides the usual DDQN loss calculation as with the baseline agent, the DeepMDP has an additional loss added to this; this total loss is used to train all neural networks of the DeepMDP. The transition loss is calculated by comparing the prediction of the next observation after dimensionality reduction given by the transition loss network, with the actual next observation after dimensionality reduction (i.e. the output of the encoder). The loss is calculated using the L1 loss function [21]. Added to this loss, is the gradient penalty which is discounted by a hyperparameter set to 0.01. The gradient penalty is a combination of the gradient penalty of the encoder, the policy network and the transition loss network. For each of these three networks, the penalty is calculated using the Wasserstein Generative Adversarial Network penalty [7].

## A.2 OpenAI Pong: RL agent architectures

In this section we will give the specific architecture and hyperparameter settings used for RL agents in the OpenAI Atari Pong environment. We will start by showing the baseline agent, which uses an architecture and hyperparameters that are shared by all agents. For all other agents, each extending this baseline agent, we will only give the additional architecture and parameters. For an overview of how each agent works, we refer to sections 3.1.1 and 3.1.3.

### A.2.1 Baseline agent

**Neural network architecture**
The DDQN policy and target network for the baseline agent consists of three convolutional layers and two linear layers. Its input is $n \times 4 \times 84 \times 84$, for batches of size

$n$. The 4 channels represent the four consecutive frames that are used for a single observation. Each convolutional layers has no padding. The strides for each layer is 6, 4 and 3 respectively, and the kernel/filter sizes are 3, 2 and 1 respectively. The first layer has 4 channels, one for each input frame. It has 32 output channels, thus the second layer has 32 input channels. The second layer has 64 output channels, thus the third layer has 64 input channels. This last convolutional layer also has 64 output channels. For input with batch size $n$, the downsampling through the convolutional layers is from $n \times 4 \times 84 \times 84$, to $n \times 32 \times 27 \times 27$, to $n \times 64 \times 12 \times 12$ to $n \times 64 \times 10$.

The output after the last convolutional layer is flattened to $n \times (64 \cdot 10 \cdot 10) = n \times 6400$, which will be the input for the first linear layer. Thus, the first linear layer has 6400 input features. It has 512 output features. The second linear layer has 512 input features and 6 output features, covering the action-space. Furthermore, after each convolutional layer, there is batch normalisation layer to normalise the layer outputs [11], and a ReLU activation function [20]. After the first linear layer there is only a ReLU activation function. No activation function or batch normalisation is used after the final layer.

When training the policy network, the optimization algorithm *Adam* is used, which is an extension on *stochastic gradient descent* [15]. For loss calculation we refer to section 2.1.3, more specifically algorithm 1.

**Hyperparameters**

The following hyperparameters are used in the baseline agent:

- Steps in between training the policy network: 1.
- Steps between updating the target network: 1000.
- Optimizer learning rate: $5e-5$.
- Discount factor: 0.97.
- Batch size for training the policy network: 32.
- Number of stored batches before training: 313.

Lastly, an $\epsilon$-greedy strategy is used for balancing exploration and exploitation. This means that a random number in $[0, 1]$ is chosen; whenever this is equal to or lower than the current epsilon value, a random action is chosen, and a greedy action otherwise. Epsilon decays from 1.0 to 0.02 in 400.000 steps spaced on a logarithmic scale. Each step taken by the agent corresponds to one epsilon decay step. One episode takes roughly between 800 and 3000 steps (depending on the length of the episode). The resulting decay of the epsilon value can be seen in figure A.2.

### A.2.2 PCA agent

The neural network architecture and hyperparameter settings of the PCA agent are exactly the same as the baseline agent (see appendix A.2.1). In the PCA agent though, the input for the policy network is now $n \times 4 \times 42 \times 42$, having reduced the dimensionality

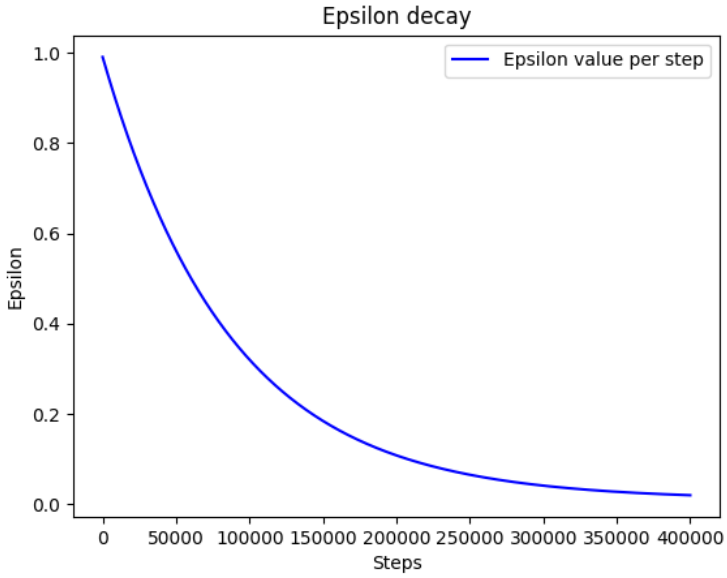



Figure A.2: The decay of the epsilon value per episode in Pong.

of each frame from $84\times84$ to $42\times42$. In the policy network, because of the stride and filter size settings in the convolutional layer, this input is downsampled from $n\times4\times42\times42$, to $n\times32\times13\times13$ after the first layer, to $n\times64\times5\times5$ after the second layer, to $n\times64\times3\times3$ after the third convolutional layer. Thus, the first linear layer has $64\cdot3\cdot3=576$ input features.

### A.2.3 Pre-trained and online trained autoencoder agent

Like with the PCA agent, the policy network and hyperparameter settings are the same as the baseline agent A.2.1. Again, the input for the policy network is now $n\times4\times42\times42$, which is downsampled through the convolutional layers to $n\times64\times3\times3$.

**Autoencoder neural network architecture and hyperparameters** The *encoder* of the autoencoder consists of three convolutional layers. The first layer has a stride of 2 to introduce dimensionality reduction from $84\times84$ to $42\times42$. The second and third layer have a stride of 1 to keep this $42\times42$ dimensionality. All three layers have a kernel size of 3 and an input padding of 1. The first layer has 1 input channel (since the frame input only has one colour channel) and 32 output channels. Hence, the second layer has 32 input channels, and it has 32 output channel. The third convolutional layer has 32 input channels and 1 output channel. After each convolutional layer, there is a *GELU* activation function, which is a nonlinear variant of ReLU [8].

The *decoder* consists of two layers. Because the decoder has to reconstruct the original input from the output of the encoder, the first is a transposed convolutional

layer to upscale its $42 \times 42$ input back to $84 \times 84$. This is done using a stride of 2 and output padding of 1. It has 1 input channel (corresponding to the 1 output channel of the last encoder layer) and 32 output channels. The other layer is a regular convolutional layers, with stride 1 and no output padding. For this second convolutional layer, there are 32 input channels and 1 output channel (corresponding to the 1 colour channel of a frame). Each layer has a kernel size of 3 and input padding of 1. After the first layer there is a GELU activation function.

For training the autoencoder, the loss is calculated by the mean-scared-error between the original input and the reconstructed input. Furthermore, the Adam optimization algorithm is used, with a learning rate of 0.0001.