Master thesis
Computing Science

Radboud University

Title Master Thesis

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`

February 7, 2022

Abstract

A few dimensionality reduction method comparisons

Contents

Chapter 1

Introduction

- describe the problem / research question
- motivate why this problem must be solved
- demonstrate that a (new) solution is needed
- explain the intuition behind your solution
- motivate why / how your solution solves the problem (this is technical)
- explain how it compares with related work

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 [15, 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 [15, 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_θ)*, 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*:* $S \times A \times$

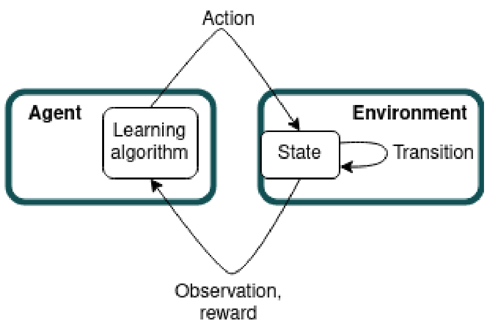


Figure 2.1: The cycle of interaction between the environment and an agent in reinforcement learning.

$S \to \mathbb{R}$ and initial states distribution S_0. 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) [15, p. 45-62].

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

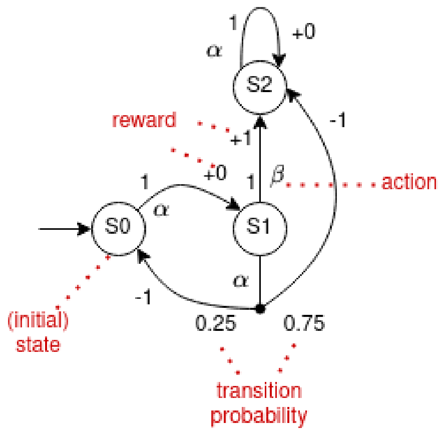


Figure 2.2: Example of a Markov decision process (MDP).

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:

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

where t is the timestep and $R_t \in R \subset \mathbb{R}$ [15, 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 [15, 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 π 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 π. 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 γ. 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 [15, 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 π 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 γ, and weigh this sum with the probability of this transition happening [15, 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* [15, 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[5, 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 [5, 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 2.3.

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 kernels (or: filters): 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.

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.

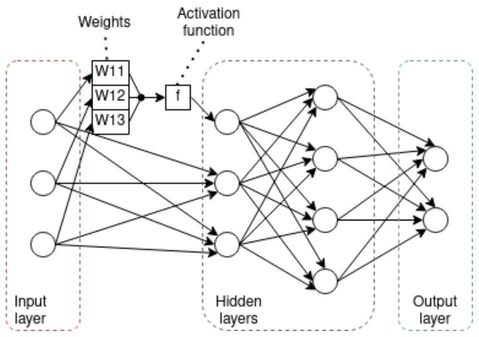


Figure 2.3: Example of a fully connected feedforward artificial neural network. For one node, example weights and activation function are shown.

For more details on artificial neural networks, we refer to [5]. 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 [10]. 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 *ϵ-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 ϵ value, a random action is taken, otherwise it takes chooses an action greedily based on the Q-table. In this strategy, ϵ 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, α the learning rate and γ 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 [14].

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). DDQN is every similar to DQN. The difference is in how the policy network is trained [19]. 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 [15, 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.

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 ϵ-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.

Algorithm 1 DDQN training step [15, 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 $loss$
```

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 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) [18]. This is especially true for CNNs, which are commonly used in RL when dealing image data [9]. Lastly, higher dimensional data may include more noise, which can prevent an agent from finding an optimal policy [20]. 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 *deepmdp*. 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) [11].

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 [2].

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^TX.

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. λ 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^TX. 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*[8]. 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 2.4.

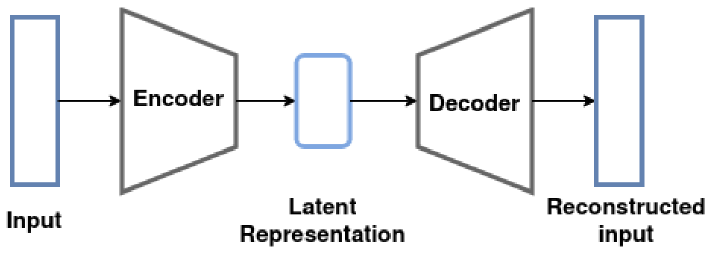


Figure 2.4: The architecture of an autoencoder.

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

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

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

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

Equation (2.7) defines the encoder and equation (2.8) the decoder, both satisfying (2.9) where Δ 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[8].

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

Info over deepmdp

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 environment 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 Method

In this section we will explain our method: how we researched the effect of state-space dimensionality reduction on an RL agent. Before going into the details of the different experiments that we ran in section 3.1.2, we will first look at the environment in which we ran the experiments in section 3.1.1.

3.1.1 Environment: Starcraft II

For our experiments we used 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*[21].

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 3.1.

An **observation** received by the agent in this minigame is given by a

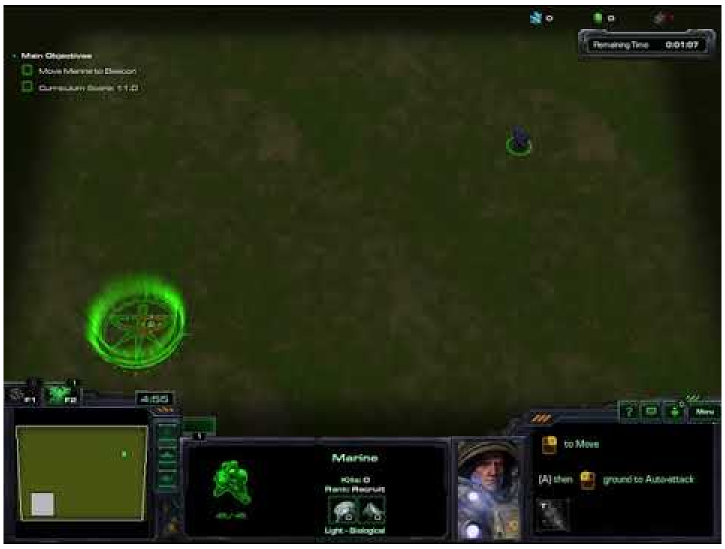


Figure 3.1: Screenshot of the minigame *MoveToBeacon* in *StarCraft II*.

32×32 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. 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 32×32 with a cardinality of $27 \cdot 27 \cdot 1003 = 731.187$. An example of such a state observation can be seen in figure 3.2.

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

Is this a correct usage of state-space?

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 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

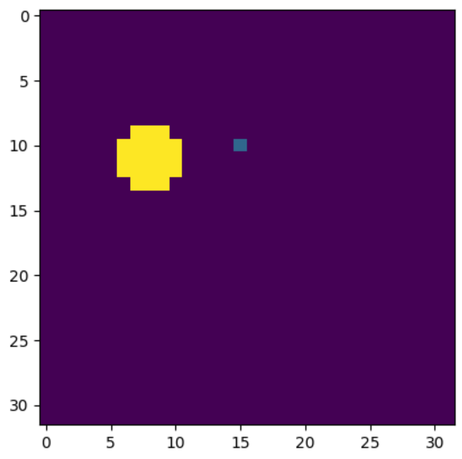


Figure 3.2: A state observation received by the RL agent, for the StarCraft II minigame MoveToBeacon. The yellow cells represent one beacon; the blue cell represents the army unit controlled by the player; all other cells are empty.

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 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 this section we will discuss the agents that we used. We will give a general overview here, referring to appendix A section A.1 for details on the neural network architectures and hyperparameter settings.

The first agent mentioned is the baseline agent, which does not use a dimensionality reduction method, therefore using the full 32×32 dimensions of the observation. All other agents reduce the dimensionality to 16×16. This means that the number of features in an observation are reduced from 1024 to 256.

Again, is this correct?

Furthermore, to allow for a fair comparison of their performance, all agents must share as many architectural design choices and hyperparameter settings as possible. This is done by extending the baseline agent in all other agents. However, one important change must be made. The baseline agent's neural network receives a 32×32 input, whereas the other agents receive a 16×16 input. In all cases, the output dimensions must be 32×32. 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 32×32 possible actions. To deal with this difference in input dimensions, the first layer of the networks of the non-baseline agents are modified to increase the dimensionality.

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. First, the agent receives an observation from the environment. This observation is passed to its neural network approximating the Q-function. 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. An action corresponds to choosing coordinates for the army unit to walk to. Then, the chosen action is performed and we repeat this cycle until the end of the episode, whilst often training the neural network on stored transitions.

The neural network consists of three convolutional layers: the first layer being a transposed convolutional layer [3], 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 16×16, but the output needs to be 32×32 to cover the action space. However, for our baseline agent, the input dimensions, 32×32, 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 32×32 (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. 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 from 32×32 to 16×16. This latent representation is then used by the agent as if it is the actual observations. This means that it is passed to the network to give an action valuation, as well as being stored in transitions used to train the network.

The output of the network representing the Q-function must remain 32×32, 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 16×16 to 32×32.

The PCA component is trained separately before being used by the agent. This is done by training the PCA on 240.000 previously stored observations, which corresponds to observations from 1000 episodes. It is important that these observations give a good representation of the environment to get a well trained PCA component. The first 256 principal components in our PCA (representing a 16×16 dimensional observation), contain roughly 96% of the information of the original data. Two PCA versions have been examined: one using a scalar to standardize the observation data as explained in section 2.2.1, and one without such a scalar.

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. Just like the PCA component, the autoencoder is pre-trained on the same 240.000 observations. After this it 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 16×16. 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 32×32. The other two layers are regular convolutional layers.

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. Specifcally, 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 has the exact same design as the pre-trained autoencoder agent. 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 by the agent.

In this case, the agent itself still only uses the encoder part of the autoencoder. However, we now also store 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
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 design as the encoders of the autoencoders: one convolutional layer with stride 2 to reduce dimensions and a second convolutional layer with stride 1. 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. The agent's network therefore consists of an encoder part and a policy part. This means that when the agent receives an observation from the environment and passes it to its network, it first goes through the encoder whose output is passed to the policy part. In effect this means that the encoder and policy network are now trained on the same loss, using a single optimizer.

Additionally, the DeepMDP makes use of an auxiliary objective to calculate the loss: the transition loss. This 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, 16×16 represent the next (predicted) latent observation. It has only one layer, a convolutional layer, with 32×32 output channels to represent the action dimensions. 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 gradient penalty is calculated on all three parts of the network separately. This represents a Lipschitz-constrained Wasserstein Generative Adversarial Network (lol wat). Its penalty is used in calculating the loss while training the network.

3.2 Results

In this section we will show and discuss the results from the experiments mentioned in section 3.1.2. We start by discussing the general results of the agents in section 3.2.1, and then discussing and interpreting these results in section 3.2.2.

3.2.1 Research results

The results from running the different agents in the Starcraft II minigame MoveToBeacon, can be seen in figure 3.3. 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 3.3a, 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 3.3b. Though this shows the results for the agent using a PCA with a scalar, similar results were achieved without using a scalar. Furthermore, 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 3.3c, show that this agent preforms a little better. 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.

We also 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 3.4. After roughly $6000 \cdot 25 = 150.000$ observation it converges to its final loss.

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.

3.2.2 Discussion

The first notable results is the PCA agent giving a policy that remains at a policy at the level of a random policy during its 800 episode training, there-with 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 3.5 shows an example of the PCA transformation on an observation. figure 3.5a shows the original observation that the agent would get from the environment. Figure 3.5b 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 32×32 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 3.1.1). This combination might lead to the PCA not being able to generalize well enough.

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. To get an understanding of why this agent achieves such good results, we will now examine the autoencoder in detail. Firstly, a correlation matrix is given in figure 3.6. 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 original data has dimensions of 32×32, whereas the reduced data has dimensions of 16×16, resulting in 1024 and 246 features respectively. 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.

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.

We also show the feature map visualisation for the enoder of the autoencoder in figure 3.7. This shows the output of each channel in each convolutional layer of the encoder of the autoencoder. The original 32×32 input for the encoder is given in figure 3.7a. 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 feature map for the first convolutional layer is given in figure 3.7b. It shows the output of its 32 channels, each having a dimensionality of 16×16. What can be seen here is that each channel seems to pay attention to a different part of the observation, sometimes putting more emphasis on the empty cells, sometimes more on a certain side of the beacon and/or the army unit. Lastly the feature map of the second layer is given in figure 3.7c. Since this has only one channel, this also corresponds to the output of the encoder.

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. This allows for the possibility of training an autoencoder once, after which multiple 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.

Lastly, the DeepMDP performs horribly and something seems off with its implementation, since it goes to a decent policy scoring around 15 per episode, then quickly dropping back to the level of a random policy. I'm not sure what is going wrong here. I did print out the loss during a few different moments in training. Its loss calculation consists of several different losses:

firstly the usual DDQN loss calculated like in all other agents. Secondly, the loss of the auxiliary objective: the transition loss. Lastly we have the gradient penalty for each part of the network. All these losses are added together, with the sum of the gradient penalties being multiplied by 0.1 (a hyperparameter).

After 133 episodes, when the agent gets decent scores of around 15 points per episode, the losses look as follows:

- DDQN Loss: 0.008
- Transition Loss: 400
- Encoder gradient penalty: 20
- Policy gradient penalty: 20
- Auxiliary objective gradient penalty: 3.5 million
- Total loss: 38.000

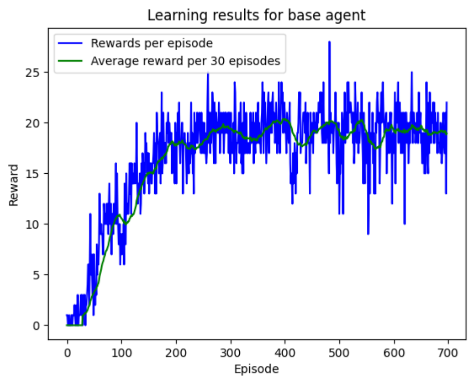
A few episodes later, it goes down to a policy scoring about 0 per episode:

- DDQN Loss: 0.002
- Transition Loss: 1000
- Encoder gradient penalty: 90
- Policy gradient penalty: 90
- Auxiliary objective gradient penalty: 1.5 million
- Total loss: 16.000

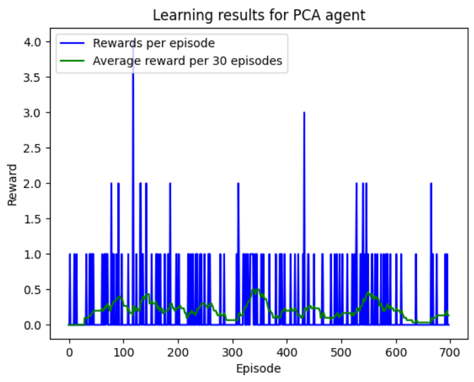
After 248 episodes, it is still at the level of a random policy:

- DDQN Loss: 0.0008
- Transition Loss: 22.000
- Encoder gradient penalty: 3.5
- Policy gradient penalty: 3.5
- Auxiliary objective gradient penalty: 2000
- Total loss: 22.000

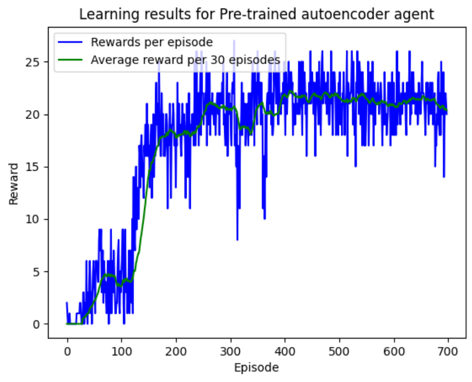
We can see here that there is probably something off with the loss calculation, since they are extremely disproportional. OR PERHAPS the problem lies in that there is no good prediction of transition (i.e. next state) possible due to the continuous random replacement of the beacon when following a good policy and therefore the only way to lower the transition loss (does this also entail the aux penalty?) would be to follow a policy that does not reach the beacon.



(a) Results for the baseline agent.

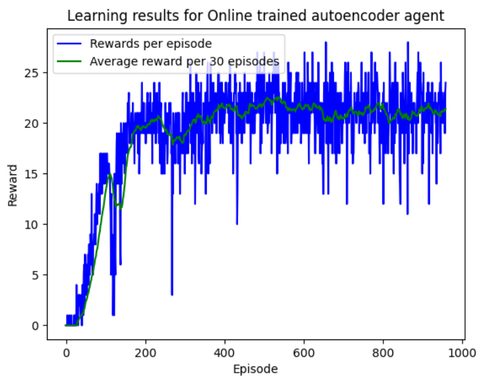


(b) Results for the PCA agent using a scalar.

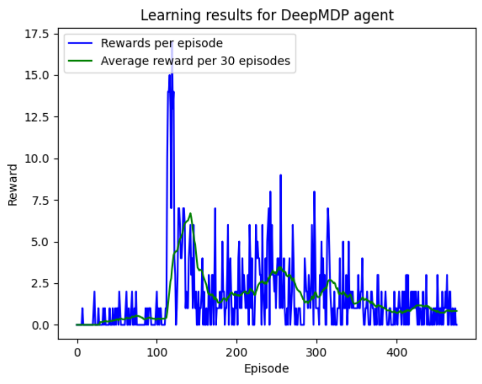


(c) Results for the pre-trained autoencoder agent.

Figure 3.3: Results of the different RL agents.



(d) Results for the online trained autoencoder agent.



(e) Results for the DeepMDP agent.

Figure 3.3: Results of the different RL agents(cont.).

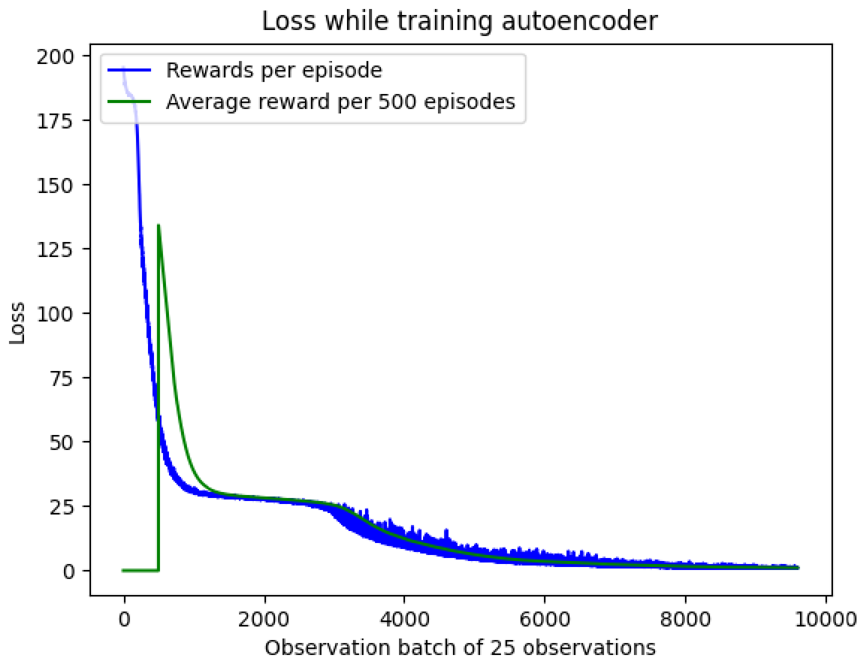
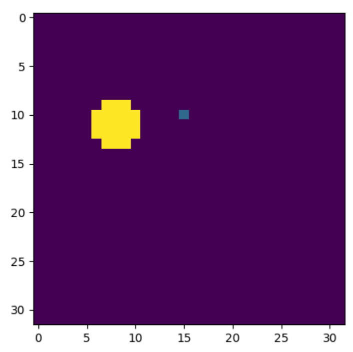
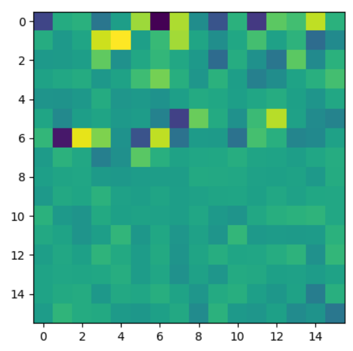


Figure 3.4: Losses per 25 observations for training the autoencoder on 240.000 state observations.



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



(b) The latent representation given by the PCA transformation

Figure 3.5: Latent representation and reconstruction of a state observation using PCA.

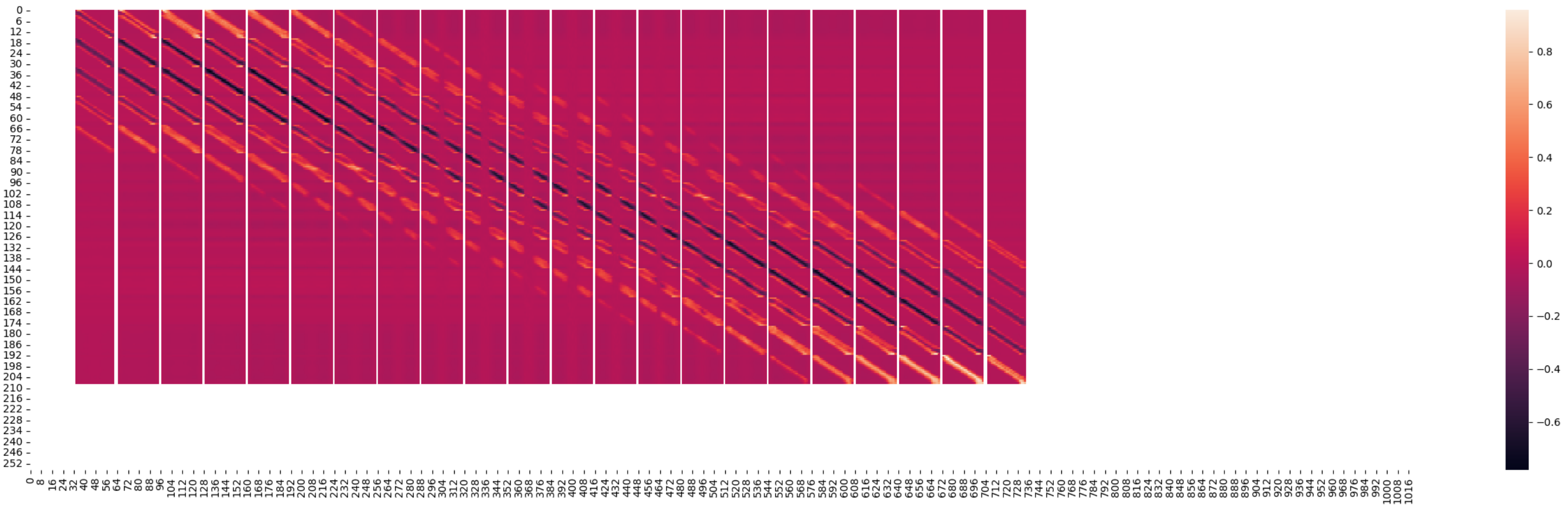
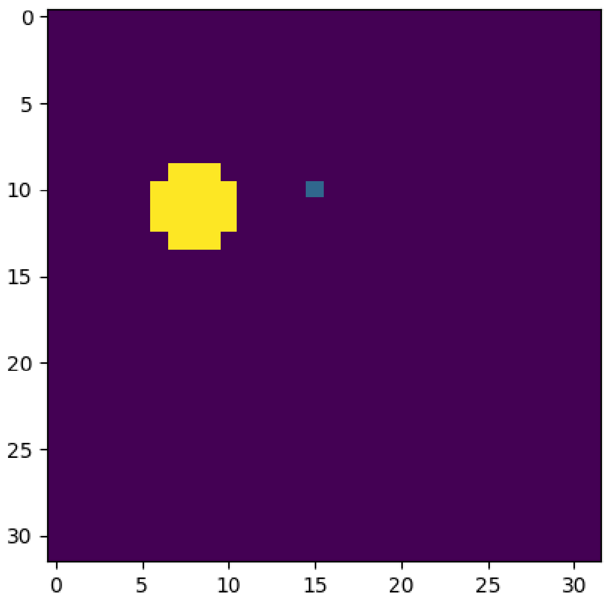


Figure 3.6: Correlation matrix for the autoencoder used in the pre-trained autoencoder agent, based on 60.000 state observations. The x-axis contains the features of the original observations, and the y-axis contains the features of the reduced state observations.


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

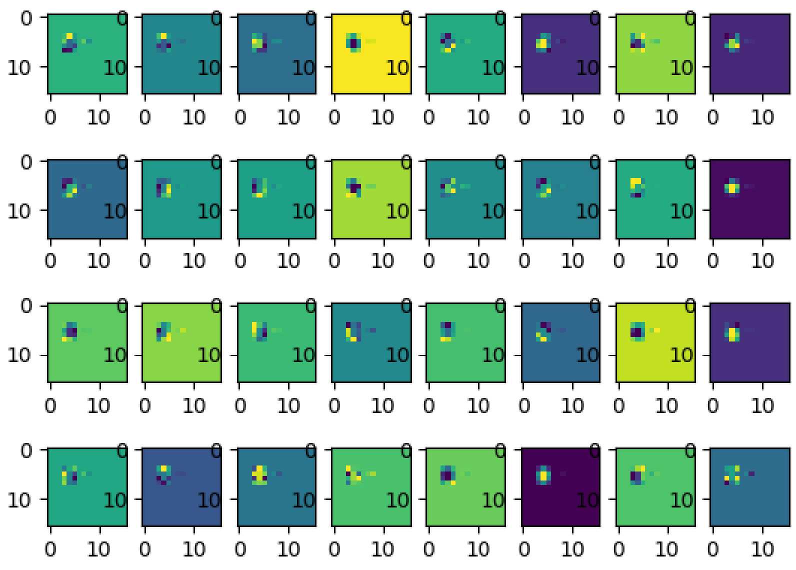
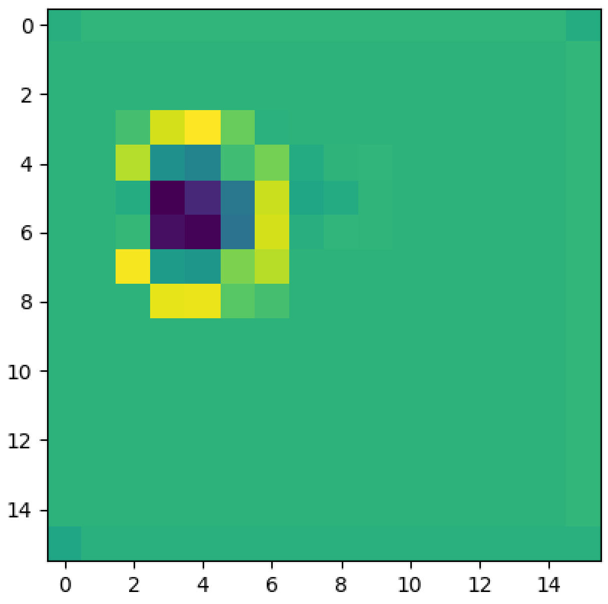

(b) The feauture map of the first convoluational layer, showing the output of its 32 channels.

Figure 3.7: A feature map visualisation for the autoencoder used by the pre-trained autoencoder agent.



(c) The feature map of the second convolutional layer, showing the output of its single channel. Since it has only one channel, this also corresponds to the output of the autoencoder.

Figure 3.7: A feature map visualisation for the autoencoder used by the pre-trained autoencoder agent (cont.).

Chapter 4

Related Work

State-space dimensionality reduction has been a topic of interest in RL for several years. Curran et al. [2] 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 [13] 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. [20] 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 (i.e. vanilla) agent is able to train to a good policy.

In 2019, Prakash et al. [18] 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. [4] 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 (i.e. vanilla) agent.

Chapter 5

Conclusions

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.

Bibliography

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

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

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

[4] 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.

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

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

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

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

[9] 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.

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

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

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

[13] 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.

[14] 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.

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

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

[17] 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.

[18] 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.

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

[20] 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.

[21] 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.

Appendix A

Appendix

A.1 RL agent architectures

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 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 section 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 [3] 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 32×32 to 16×16. This 16×16 observation is the input for their policy and target network. The output of these networks however need to be 32×32, capturing the action space. Although this upscaling is not needed for the baseline agent since it uses the full 32×32 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 $ReLU$ [16] 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* [12]. For loss calculation we refer to section 2.1.3, more specifically algorithm 1.

Lastly, an ϵ-greedy strategy is used for balancing exploration and exploitation. This means that a random number in $[0, 1]$ is chosen; whenever this is is equal to or lower than the current epsilon value, a random action is chosen, and a greedy action otherwise. The epsilon decay can be seen in figure

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, 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 32×32 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 16×16 input to 32×32 output. In both autoencoder agents, the autoencoder has the same architecture and

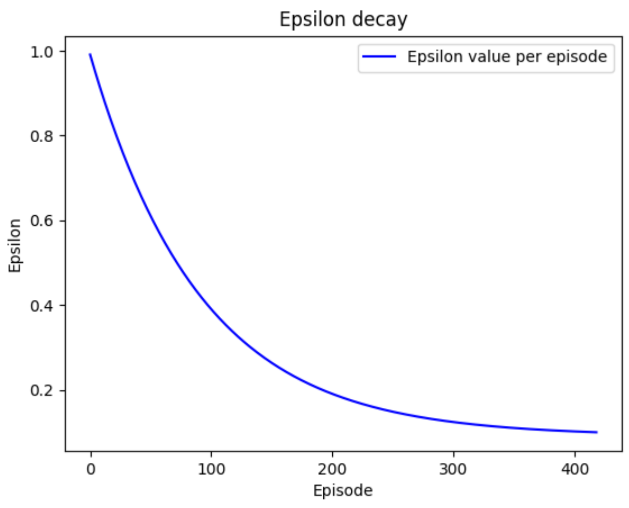


Figure A.1: The decay of the epsilon value per episode.

hyperparameters.

Autoencoder neural network architecture and hyperparameters The encoder of the autoencoder consists of two convolutional layers. The first layer has a stride of 1 to introduce dimensionality reduction from 32×32 to 16×16. The second layer has a stride of 1 to keep this 16×16 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 [7].

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 16×16 input back to 32×32. 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 16×16 to 32×32.

DeepMDP encoder and transition loss architectures and hyperparameter settings

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 32×32 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 [17]. 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 [6].