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

December 25, 2021

**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

## 2.1 Reinforcement learning

### 2.1.1 General overview

RL,

### 2.1.2 Neural networks

neural networks (incl non-linnear function approximation)

### 2.1.3 (D)DQN

si

## 2.2 State-space dimensionality reduction

Definition, general info Methods:

### 2.2.1 Principal Component Analysis

algemene info pca

### 2.2.2 Autoencoder

Another way of projecting data onto a lower dimensional space, is using an *autoencoder*[2]. 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 2.1.

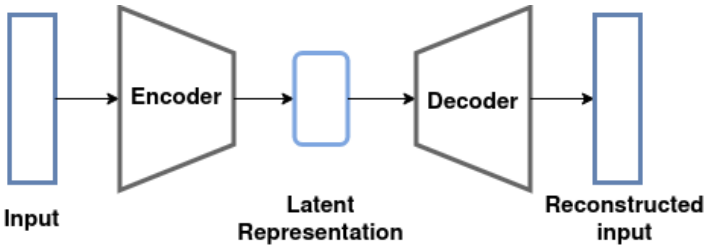



Figure 2.1: The architecture of an autoencoder.

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

$$\phi : \mathbb{R}^n \to \mathbb{R}^m \tag{2.1}$$

$$\psi : \mathbb{R}^m \to \mathbb{R}^n \tag{2.2}$$

$$\phi, \psi = \arg\min_{\phi,\psi} \Delta(x, \psi \circ \phi(x)) \tag{2.3}$$

Equation (2.1) defines the encoder and equation (2.2) the decoder, both satisfying (2.3) 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. 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[2].

### 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*[3].

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*.

$32 \times 32$ 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. 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 3.2.

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.

Is this a correct usage of state-space?

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. 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.

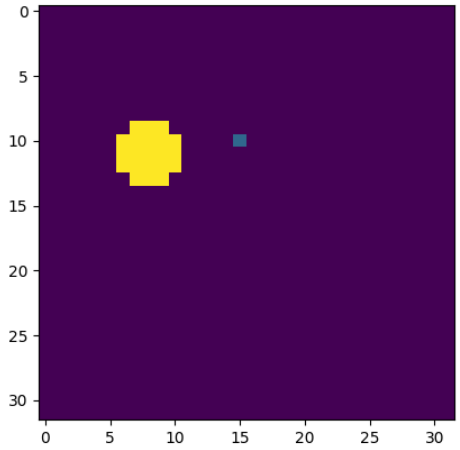



Figure 3.2: A state observation received by the RL agent, for the StarCraft II minigame MoveToBeacon. The yellow cells represent one beacon; the blue cell represents the army unit controlled by the player; all other cells are empty.

### 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 vanilla agent, which does not use a dimensionality reduction method, therefore using the full $32 \times 32$ dimensions of the observation. 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?

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 vanilla agent in all other agents. However, one important change must be made. The vanilla agent's neural 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 networks

of the non-vanilla agents are modified to increase the dimensionality.

**Vanilla agent**
The vanilla 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 Conv2DTranspose, the other two being Conv2D layers. The use of the Conv2DTranpose layer allows for the possibility of changing the dimensionality of the given input. For our vanilla 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. As mentioned, this is done by extending the vanilla agent: after receiving an observation from the environment, the observation is processed by a PCA component lowering the observation dimensionality from $32 \times 32$ to $16 \times 16$. 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 $32 \times 32$, since we have $32 \cdot 32$ possible actions: one action per coordinate. Therefore, the first layer in the policy network, the aforementioned Conv2DTranspose layer, has a stride of 2. This changes the dimensions from $16 \times 16$ to $32 \times 32$.

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 $16 \times 16$ dimensional observation), contain roughly 96% of the information of the original data.

**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 Conv2D 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 Cond2DTranspose layer with a stride of 2, to bring the dimensions back to $32 \times 32$. The other two layers are Conv2D 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**
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.

I guess "online" is not strictly correct

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 Conv2D layer with stride 2 to reduce dimensions and a second Conv2D layer. 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 ex-

tends 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, $16 \times 16$ represent the next (predicted) latent observation. It has only one layer, a Conv2D layer, with $32 \times 32$ 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 wel 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 $0 - 3$.

Firstly, the results of training the PCA agent on 800 episodes are not shown in these results. This is because, no matter what neural network architecture used in the agent (having used multiple different ANN's and CNN's), it always remained at a policy at the level of a random policy.

As can be seen in subfigure 3.3a, the vanilla 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 scoring around 10. After this it still needs another 300 episodes to get to a more consistent policy.

The results for the agent using a pre-trained autoencoder, in figure 3.3b, 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.

Todo: create figure of this.

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 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 does it converge 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. Uncertainty remains whether the agent could find a good policy after enough episodes. In any case it performs 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.

Perhaps try this?

Citing

Another interesting result is the performance of the pre-trained autoencoder agent. Not only does it match the vanilla 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 vanilla 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 3.5. 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 $32 \times 32$, whereas the reduced data has dimensions of $16 \times 16$, 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 3.6. This shows the output of each channel in each convolutional layer of the encoder of the autoencoder. The original $32 \times 32$ input for the encoder is given in figure 3.6a. 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 3.6b. It shows the output of its 32 channels, each having a dimensionality of $16 \times 16$. 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 3.6c. Since this has only one channel, this also corresponds to the output of the encoder.

This autoencoder agent not only outperforms the vanilla 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 vanilla agent (even scoring slightly better ong average) in a similar number of episodes; the main difference being that the vanilla 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 performance 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 normal DQN 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:

- DQN 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

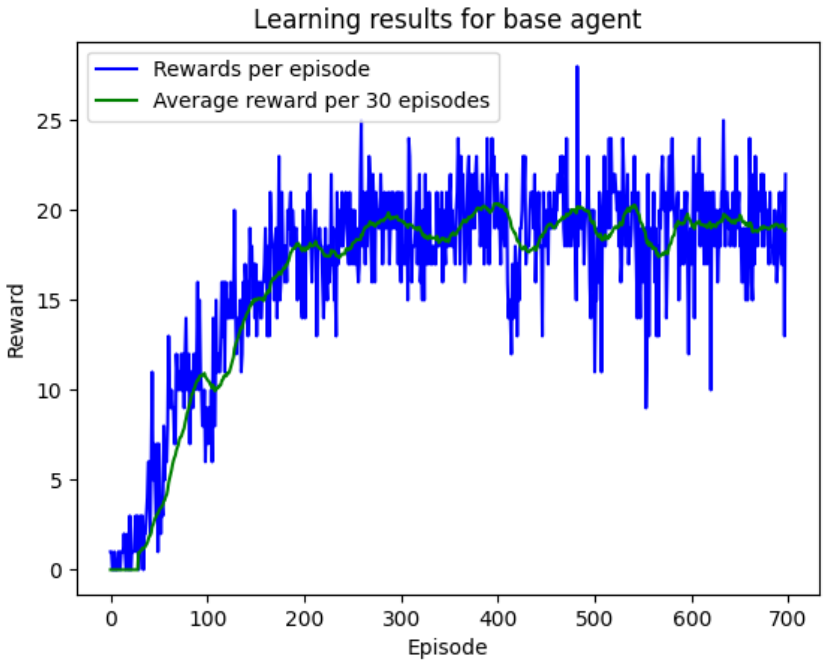
A few episodes later, it goes down to a policy scoring about 0 per episode:

- DQN 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

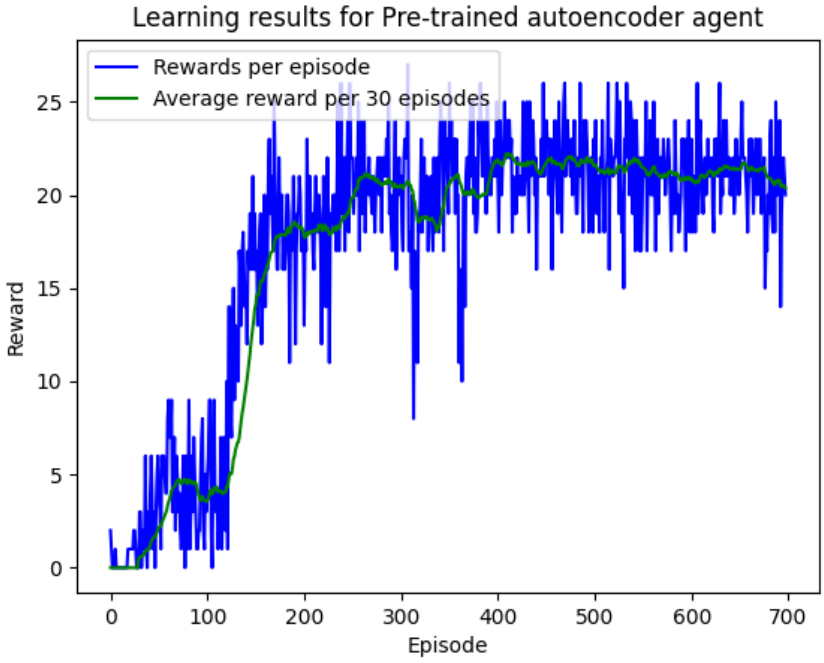
After 248 episodes, it is still at the level of a random policy:

- DQN 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.

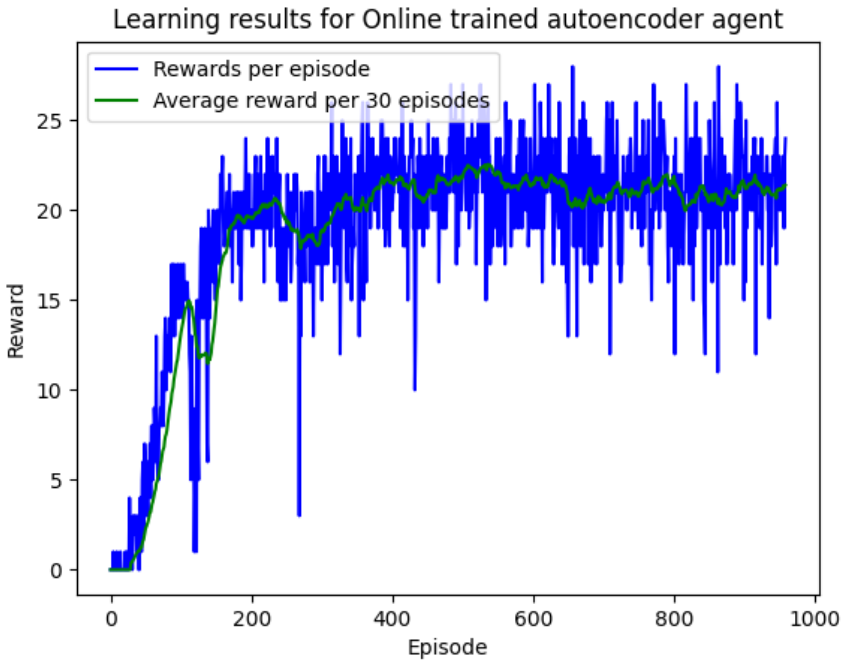



(a) Results for the vanilla agent.

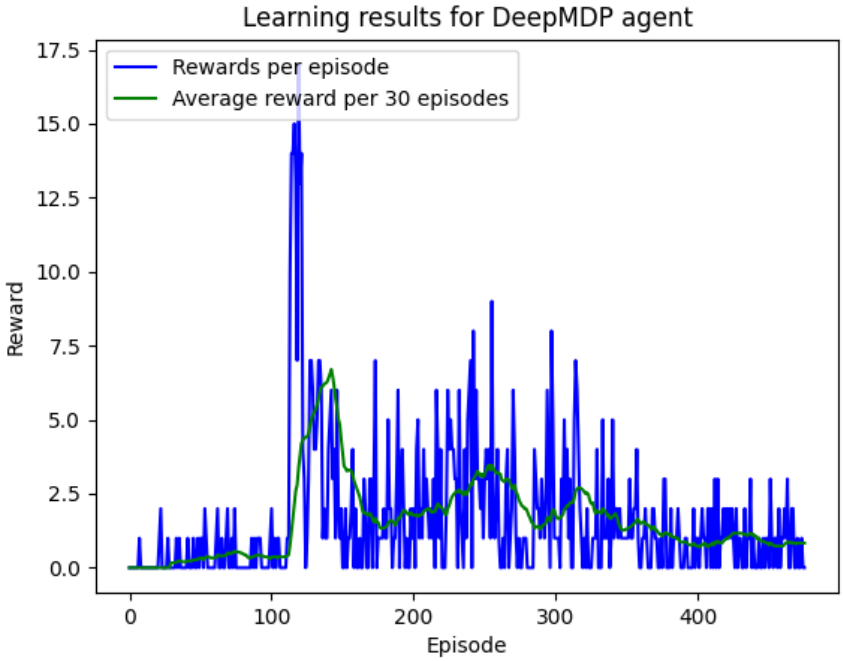



(b) Results for the pre-trained autoencoder agent.

Figure 3.3: Results of the different RL agents.




(c) Results for the online trained autoencoder agent.




(d) Results for the DeepMDP agent.

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

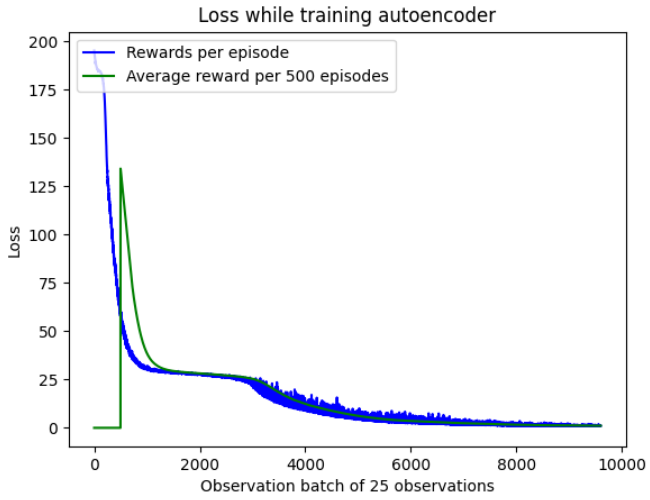



Figure 3.4: Losses per 25 observations for training the autoencoder on 240.000 state observations.

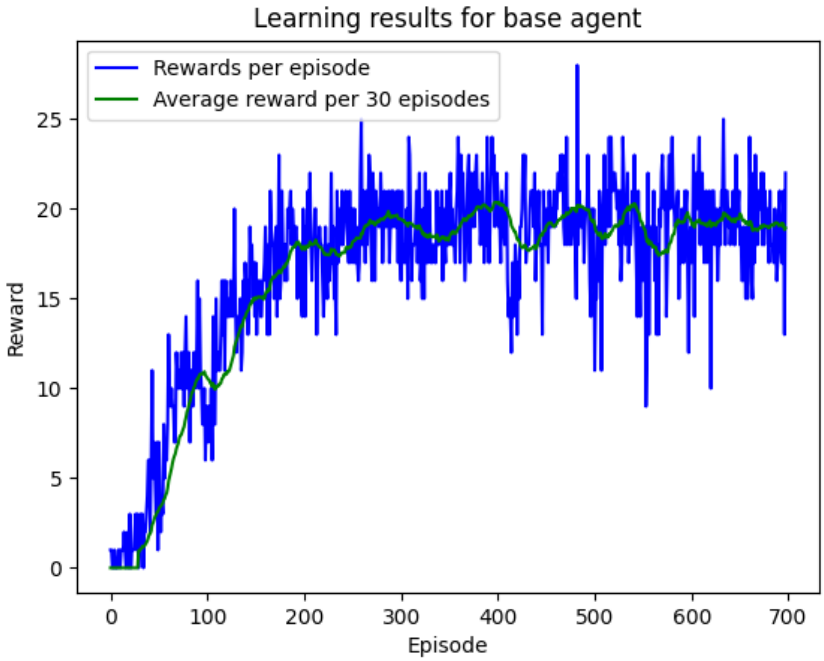
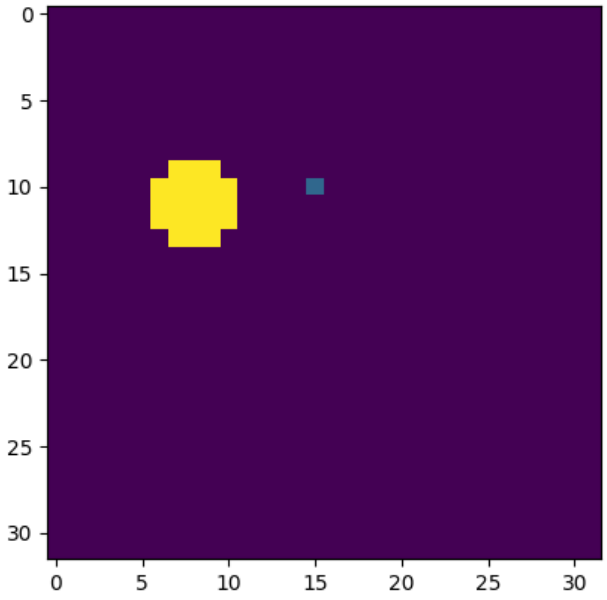
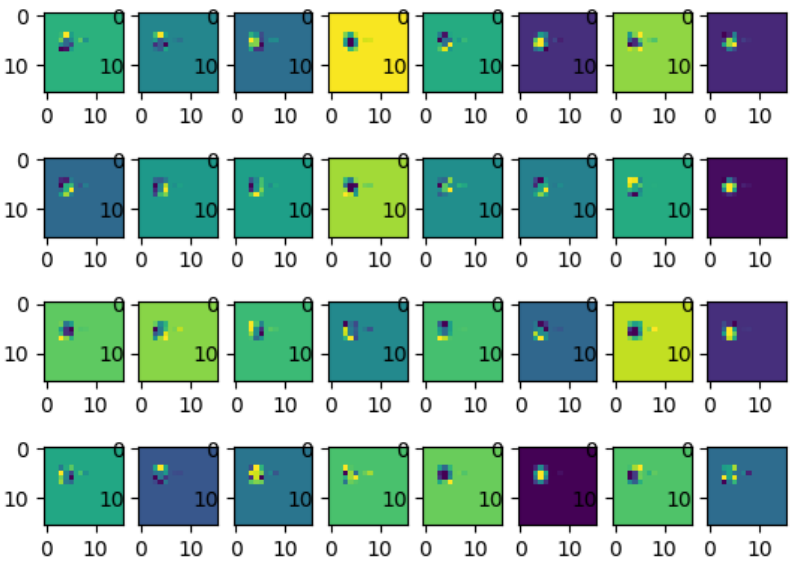



Figure 3.5: Correlation matrix for the autoencoder used in the pre-trained autoencoder agent, based on 60.000 state observations. The x-axis contains the features of the original observations, and the y-axis contains the features of the reduced state observations.

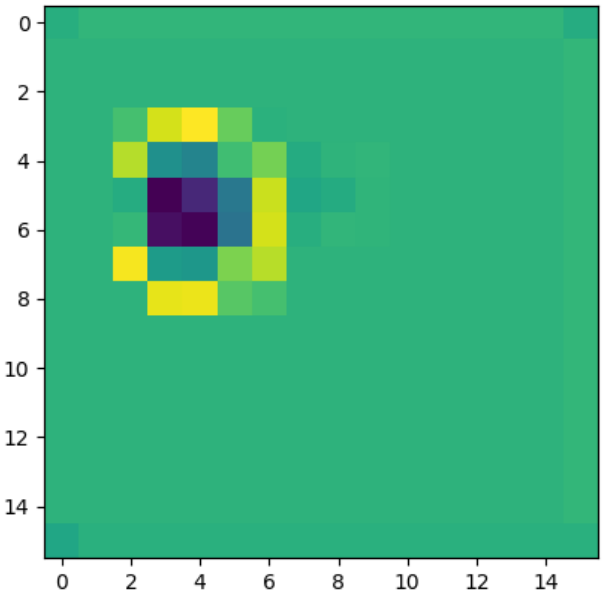



(a) The original observation, i.e. the input for the autoencoder.



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

Figure 3.6: 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 3.6: A feature map visualisation for the autoencoder used by the pre-trained autoencoder agent (cont.).

# Chapter 4

# Related Work

In this chapter you demonstrate that you are sufficiently aware of the state-of-art knowledge of the problem domain that you have investigated as well as demonstrating that you have found a *new* solution / approach / method.

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

[3] 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 vanilla agent, which uses an architecture and hyperparameters that are shared by all agents. For all other agents, each extending this vanilla agent, we will only give the additional architecture and parameters.

### A.1.1 Vanilla agent

Jaaa JAAAAA