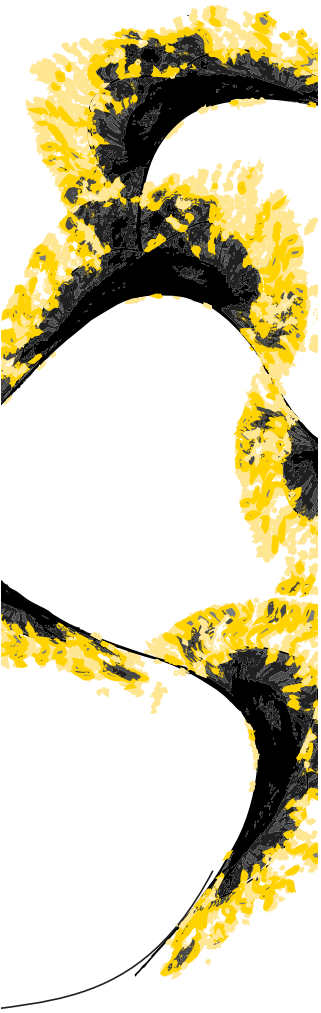


RAM ROBOTICS AND MECHATRONICS

STATE REPRESENTATION LEARNING USING A GRAPH NEURAL NETWORK IN A 2D GRID WORLD

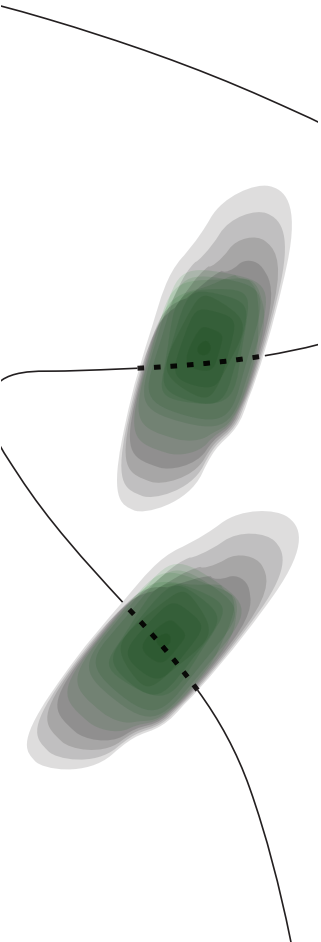
Y.C. (Yannick) van Wettum

MSC ASSIGNMENT

Committee:
prof. dr. ir. G.J.M. Krijnen
N. Botteghi, MSc
dr. M. Poel

May, 2021

027RaM2021
Robotics and Mechatronics
EEMathCS
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands



UNIVERSITY OF TWENTE. | TECHMED CENTRE UNIVERSITY OF TWENTE. | DIGITAL SOCIETY INSTITUTE

Summary

Reinforcement learning algorithms have shown great success in solving complicated robotics tasks. These tasks often involve multiple sensors which generate a high dimensional sensory input, i.e. observation. Learning the optimal policy from the high dimensional observation directly often requires processing large amounts of data. State representation learning aims to map the high dimensional observation to a lower dimensional state space, in order to reduce training time and the required amount of data.

The ability to reason in terms of objects, relations and actions is an important aspect of human cognition (Spelke and Kinzler, 2007). This ability serves as a core motivation behind recent works that aim to incorporate relational reasoning in machine learning models (Battaglia et al., 2016; Kipf et al., 2018; Xu et al., 2019). A graph neural network has shown to be a powerful general framework for reasoning about objects and relations. In this work, the goal is to effectively use a graph neural network in state representation learning. The focus is on a navigation task in a 2D deterministic grid world environment. This environment has clear objects which can be encoded as a graph. Moreover, it is clear what the learned state representation should look like in terms of objects (a 2d grid structure for each moving object).

In order to learn a state representation, an observation and the next observation are encoded into a state representation by using an encoder network. A neural network (i.e. transition model) is trained to predict the transition between the state representations. By simultaneously training an encoder network and a transition model in latent space, a state representation is learned. The transition model is implemented using a graph neural network in order to learn a state representation. The comparison is made with a conventional neural network.

First, the problem is formulated as a supervised learning problem, where the states are manually encoded. This way, the encoding step is excluded and the focus is on learning the transition model. It was shown that the learned transition model and reward model can be used to plan the optimal policy for maze configurations seen during training. However, training a transition and reward model on randomized maze configurations showed to be problematic. The graph neural network was not able to learn the reward function. Moreover, both transition models did not show to be precise for unseen maze configurations. Nevertheless, the graph network showed to handle transition predictions for unseen maze configurations slightly better compared to the conventional neural network. Both learned environment models were not sufficiently accurate to be used for planning in unseen maze configurations.

Secondly, the transition and reward model are trained in an unsupervised setting. A contrastive loss function is added with negative sampling in order learn expressive state representations (Kipf et al., 2020b; van der Pol et al., 2020). Furthermore, a dense reward function is used to make the reward function more informative. The graph neural network was able to decouple the objects in the scene, while the conventional neural network was not able to do this. The state representation of the conventional neural network still adhered to the grid like structure of the problem. For both the graph neural network and conventional neural network, the learned state representation and environment model were used to successfully plan the optimal policy in latent space.

The graph neural network allows to learn expressive representations for each object in the scene, which is not possible using a conventional neural network. For simple environments, there will be no benefit in using a graph neural network as opposed to a neural network to learn a state representation from raw observations.

Acknowledgements

First and foremost, I would like to thank Nicolò Botteghi for giving me the opportunity to do this MSc thesis under his supervision. He was the helping hand I needed during my thesis. Nicolò has been very supportive during the entire process and was always willing to make time. Whenever I stumbled upon a blockade Nicolò was available to provide me with new clear insights and motivation. His patience, understanding, enthusiasm and passion helped me get further. I am thankful for his guidance and could not have wished for anything more in a daily supervisor.

Furthermore, I would like to thank the other members of the committee, prof.dr.ir. G.J.M Krijnen and dr. M. Poel for their time to read my work and for their feedback.

Next, I would like to thank my parents, Mark van Wettum (I wish you would still be around) and Joan Petrus. I am very grateful for my rich upbringing, the support for pursuing a study which I adore and for always supporting me.

Moreover, I like to express my gratitude to the people who proof-read my work. Thanks to my sister, Martine van Wettum and my brother, Floris van Wettum for proof-reading my thesis and providing feedback.

Many thanks to my friends for providing support, distraction and simply for being in my life. In particular, Rob Kers and Bart Ettema for being a cycling/running buddy during this pandemic.

Finally, I would like to thank my girlfriend, Anouk Molenaar, for her endless moral support. Moreover, for moving in during this pandemic which gave me a lot of support. I am thankful for her fantastic cooking skills and the many forest/city walks we had during the lockdown. She was always willing to help me when I stumbled upon difficulties by providing a listening ear. I would not be where I am today as a person without having her in my life.

Contents

List of Abbreviations

ADAM	Adaptive Moment Estimation
AI	artificial intelligence
C-SWM	Contrastively-trained Structured World Model
CNN	convolutional neural network
DGL	Deep Graph Library
DMDP	Deterministic Markov Decision Process
DQN	Deep Q-Learning
GN	Graph Network
GNN	graph neural network
MSE	mean squared error
MDP	Markov Decision Process
ML	machine learning
MLP	multilayer perceptron
MPNN	Message Passing Neural Network
NN	neural network
PCA	Principal Component Analysis
POMDP	Partially Observable Markov Decision Process
PyG	PyTorch Geometric
ReLU	Rectified Linear Unit
RNN	recurrent neural network
tanh	hyperbolic tangent
TF	TensorFlow

Summary of Notation

This following provides a reference for the notation used throughout the thesis.

Notation	Explanation
t	(discrete) time step
s_t	state at time t
s_{t+1}	subsequent state from s_t, sometimes s' is used instead of s_{t+1} to keep the notation clean
a_t	action at time t
r_t	reward as a result of s_t and a_t
π	policy
γ	discount-rate parameter
λ	decay-rate parameter
$\mathcal{S}$	set of states
$\mathcal{A}$	set of actions
$\mathcal{O}$	set of observations
Π	set of policies
$\mathcal{L}$	denotes a scalar-valued objective function
$(.)^*$	denotes the optimal variant of a certain variable or function
$\hat{()}$	denotes the approximation of a certain variable or function
$\mathcal{G}$	a graph
$\mathcal{E}$	set of edges
$\mathcal{V}$	set of nodes (vertices)
$\mathcal{N}(i)$	set of neighboring nodes of node i
f_{edge}, f_{node}, f_{global}	respectively: edge update function, node update function and global update function
θ	parameters of a neural network

Table 1: Notation used throughout the thesis.

1 Introduction

1.1 General problem statement

Over the past decade, the field of artificial intelligence (AI) has exploded. Deep learning methods such as convolutional neural networks (CNNs), recurrent neural networks (RNNs) and autoencoders have made great strides with advancements in numerous fields, such as computer vision (Krizhevsky et al., 2012; Silver et al., 2016) and natural language processing (Dai et al., 2019). Moreover, deep learning algorithms have been successful in the field of reinforcement learning. Recently, a reinforcement learning agent (AlphaGo) was shown to be able to defeat human professional players in the full-sized game of Go (Silver et al., 2016). This was extended to AlphaZero which was able to achieve super human level performance in games of chess, shogi and Go by self-play (Silver et al., 2017).

Recent advances in deep reinforcement learning have been demonstrated mostly in the domain of computer games. Nevertheless, reinforcement learning algorithms have shown promising developments in the field of robotics and already tackled many complex problems (Kober et al., 2013), such as, control of tensegrity robots (Geng et al., 2016) and training a human-like robot hand to manipulate physical objects with unprecedented dexterity (OpenAI et al., 2018). However, reinforcement learning algorithms still face important challenges such as low sample efficiency, generalization to unseen environment situations and choosing appropriate state representations. A large variety of problems in robotics can be naturally formulated as a reinforcement learning problem, which makes it an important topic of research.

In robotics, multiple sensors are typically used. These sensors produce a very high dimensional observation. The relevant information of the observation can often be encoded into a much lower dimensional state. The *observation* refers to the raw (high dimensional) sensory data, and the *state* refers to a compact description of this observation that includes all relevant information needed to determine a robot's action. In robot navigation, often generic sensors (e.g. cameras and Light Detection and Ranging (LIDAR) sensors) are used to detect obstacles. The generic sensors produce a high dimensional observation. The high dimensional observation data slows down learning, increases the training time and increases hardware requirements. Therefore, it is desired to reduce the high dimensional observation to a lower dimensional state, e.g. in the context of robot navigation the most important information is the position of the obstacles. The dimension of the state is typically significantly smaller than the dimensionality of the observations space. The automatic process of finding a lower dimensional representation for the raw observations is referred to as 'state representation learning', and is an important topic in the field of reinforcement learning. The objective of state representation learning is to reduce the observations into states, that contain the most representative features which are sufficient for efficient policy learning. Once, such a low dimensional state representation for an environment is learned, policies can be learned quickly compared to directly learning a policy from raw observations.

In state representation learning, the underlying state is learned from observations in an unsupervised manner, because the true state is not available in the reinforcement learning problem. However, some supervision is possible by making use of prior knowledge of the environment. Prior knowledge about the environment (robotic priors) can be used to aid the process of learning a state representation (Jonschkowski and Brock, 2015). This is achieved by encoding some prior knowledge about the environment into the loss functions that are used for training a neural network. In this work, the focus is not on the prior knowledge that is incorporated in the loss function, but rather the prior knowledge included in the underlying structure of the state. Mostly, states are represented as *vectors*, while this might not always be the most obvious

choice. For example in problems with a clear relational structure e.g. n-body problems, the state can also be represented as a *graph*. A graph is able to represent data in a clear structure of entities and relations.

A recent approach in an attempt to generalize deep neural networks to non-Euclidean domains such as graphs, is *geometric deep learning* (Bronstein et al., 2016) and in specific *graph neural networks (GNNs)*. By enforcing a graph structure on the state, a relational inductive bias is incorporated into the state representation. Incorporation of an inductive bias in the model architecture, allows the model to generalize better to novel situations and makes the learning process more sample efficient (Battaglia et al., 2018). Accordingly, this thesis contributes by exploring the application of GNNs in the field of state representation learning in order to improve a reinforcement learning agent's generalizability to novel situations.

1.2 Problem formulation

We, as humans, understand the world around us in terms of objects and relations (Spelke and Kinzler, 2007). Reinforcement learning algorithms still lack the general ability for relational and causal reasoning, conceptual abstraction and many other cognitive abilities. Furthermore, generalizing beyond experiences remains a challenge for current AI algorithms. It is argued that relational reasoning and combinatorial generalization is needed to achieve human-like abilities (Battaglia et al., 2018). As humans, we make a representation of the world in our minds in order to interact with the world. This representation enables us to navigate the world, achieve goals we set and adapt to unforeseen situations. The human ability to reason in terms of objects and relations serves as a core motivation for incorporating relational inductive biases in deep learning models (Battaglia et al., 2016, 2018; Hamrick et al., 2018; Kipf et al., 2020b).

The mental representation of the world we humans have, gives us the ability to predict the future, i.e. plan ahead in certain situations. This is useful in nearly all activities we engage in, for example, while driving a car, playing sports or playing board games. The ability of planning is also an important aspect of reinforcement learning algorithms, such as autonomous navigation. Planning algorithms typically require a model of the environment, which can be used by the agent to predict how the environment will respond to its actions. When a sufficiently accurate model of the environment is available, planning algorithms (e.g. value iteration, Monte-Carlo tree search) can be applied to compute the optimal value function or optimal policy. In this work, the focus is on learning a representation of the environment that is suitable for finding an optimal policy. The decision-making problem is therefore split into a model learning step and a planning step, similar to the model-based reinforcement learning approach.

Before the planning step can be done, a model of the environment must be learned. In the reinforcement learning framework, the environment can be formulated as a Markov Decision Process (MDP). The MDP describes a fully observable environment in the reinforcement learning framework (Sutton and Barto, 2018). The model of the environment that is needed to do efficient planning approximates the MDP in a lower state space. Ideally, the learned environment model is in perfect correspondence with the original MDP, this is a so-called MDP *homomorphism*. When a policy is learned using the homomorphic model of the MDP, the policy can be lifted to the original MDP. Hence, by first learning a model of the environment in a latent space, planning can be done efficiently under the assumption that the learned environment model is sufficiently accurate. With an accurate model of the environment, the agent can learn to generalize to novel environment scenarios, opposed to model-free approaches that aim to directly determine a policy from experiences.

Planning plays an important role in robot navigation, therefore the focus is on state representation learning in a robot navigation context. In navigation problems, there is an agent that needs to traverse to a goal while interacting in an environment with obstacles. By modeling the obstacles and agent as a graph, where entities are nodes and their relations are edges, an

object-oriented representation of the environmental state is constructed. By representing the state as a graph, a relational inductive bias is incorporated and expressive state representations can be learned directly from raw observations (Kipf et al., 2020b). Furthermore, the relational state representation facilitates more accurate counterfactual reasoning about effects of actions and therefore allows for generalization to novel environment situations (Battaglia et al., 2018).

The main goal of this project is to improve upon the state of the art in representation learning using a GNN. The aim is to learn useful state representations from raw pixel observations without direct supervision. The state representation will be learned by predicting the next state in latent space. By encoding the observation and next observation into latent space and putting the loss in latent space, both a model of the environment and a state representation are learned. Additionally, there is no need to learn a decoder because the loss is applied in latent space. The learned state representation and the learned model of the environment are used to learn a policy. The effectiveness of learning a state representation using a GNN will be compared to representation learning using a neural network (NN) architecture. Particularly, the focus is on generalization performance to novel environment situations. In the remainder of this thesis, the abbreviation NN will refer to the explicit feedforward neural network architecture. If 'neural network' is written out, this refers to neural networks in general. A navigation task in its simplest form will be used to analyze different state representation learning approaches. More specifically, the navigation task entails navigation in a deterministic 2D grid world, in which fully observability is assumed.

1.3 Research questions

Based on the main goal described above, the following main research question is formulated:

"To what extent can a graph neural network be used effectively in state representation learning in a fully observable deterministic 2D grid world, to increase an agent's generalization performance to novel environment scenarios?"

To elaborate, the focus is on learning expressive state representations and a model of the environment that facilitates generalization to novel environment situations. By using a model of the environment that generalizes to novel environment scenarios, an agent can be trained for unseen environment situations without the need for additional simulation. By incorporating an inductive bias in the model architecture by means of a GNN, it is expected that the environment models can generalize better to novel environment situations and expressive state representation will be learned.

In order to find an answer to the main research question, it is split into two separate subquestions, see research question 1 (RQ1) and research question 2 (RQ2).

RQ1 *"How can a graph neural network be used to learn a transition and reward model in a fully observable deterministic 2D grid world?"*

RQ2 *"To what extent can a graph neural network be used effectively to learn a transition and reward model in a latent space for a fully observable deterministic 2D grid world?"*

The main focus of RQ1 is the effectiveness of a GNN as a choice to model the environment (transition model and reward model) compared to a NN architecture. The two approaches will be compared with respect to generalization performance of the environment model to unseen scenarios. The environment state is directly accessible such that no state representation has to be learned. Furthermore, for both learned environment models the value function will be calculated, which will be used to determine a policy. The calculated value functions will be compared with respect to the true value function.

In real world scenarios, the input data is often high dimensional, e.g. images or laser data. A compression of the observation is needed to do efficient learning. Therefore, RQ2 extends to a more realistic setting, where the raw observations will be automatically mapped to a latent state representation (either as a graph or as a vector). Simultaneously, a transition model and a reward model will be learned in latent space. Hence, in RQ2 the focus is on learning a useful state representation from raw observations in a self-supervised setting. This will be achieved by training a model of the environment in latent space. The learned environment model and the learned state representation will be used to train an agent in latent space. This will be done for both a GNN implementation and NN implementation of the environment model.

The main contributions of this work are:

- The application of a GNN to learn an approximate MDP homomorphism;
- A comparison of a MDP homomorphism learned using a NN, against a MDP homomorphism learned using a GNN, for a fully observable deterministic 2D grid world.
- The application of an approximate MDP homomorphism (learned using a GNN and NN) to learn a policy in latent space.

1.4 Report outline

The report is organized as follows:

Chapter 2 - Background : In Section 2.1, the background theory regarding reinforcement learning is presented. Followed by Section 2.2, where the different approaches in state representation learning are discussed. Then, the graph data structure is introduced in Section 2.3. The basics on NNs, CNNs and GNN are respectively presented in Section 2.4, Section 2.5 and Section 2.6.

Chapter 3 - Related work : In this chapter, related work regarding unsupervised state representation learning is discussed. In Section 3.1, related work according to the three strategies in state representation learning is discussed briefly. Followed by Section 3.2, where the focus is on learning transition models (using GNNs). In Section 3.3 a recent success story of a GNN in reinforcement learning is presented.

Chapter 4 - Method : In this chapter the methodology used to answer the research questions is presented. In Section 4.1, it is described how a model of the environment is trained with access to the environment state. Followed by Section 4.2, where it is presented how the generalization performance is evaluated. In Section 4.3, the problem is extended to learn a state representation from raw observations. This is done by training an environment model in latent space. Then in Section 4.4, it is described how the environment model is for planning in latent space.

Chapter 5 - Experimental design : In Section 5.1, the used software is presented and the environments used in the experiments are discussed. In Section 5.2, The performed experiments are elaborated.

Chapter 6 - Results and discussion : The results of the experiments are presented in this chapter. In Section 6.1 the results of learning an environment model in a fixed environment are presented. This is extended in Section 6.2, where the environment is randomized. The value functions for unseen maze configurations are presented in section 6.3. In Section 6.4, the learned state representation are presented. In Section 6.5, the results for learning a policy in latent space and applying it in the original environment are shown.

Chapter 7 - Conclusion and recommendation : Here the work is concluded. In Section 7.1, the research questions are answered. In Section 7.2, some future research directions are highlighted.

Appendices : Appendix A - Activation functions, contains supplementary information about the different kind of activation functions. Appendix B - GNN framework - GraphNets, contains a brief summary of the Graph Networks framework. Appendix C - Setup development environment, is a guide to setup the development environment using TensorFlow and CUDA in Pycharm. Appendix D - Graph learning libraries, presents an overview of the different graph learning libraries that are available at the time of writing. Appendix E - Reproduction results C-SWM paper, the C-SWM architecture (Kipf et al., 2020b) is made in TensorFlow and the result are compared to the original paper for one of the environments. Appendix F - Additional figures, some additional results are added.

2 Background

This chapter introduces the background information related to state representation learning, reinforcement learning and different types of neural networks. This starts with Section 2.1, which provides an overview of the reinforcement learning framework. Secondly, in Section 2.2, the state representation learning approaches are presented. In Section 2.3, the data structure of a graph is discussed. Then, in Section 2.4, the basics on NNs are covered. This is followed by a discussion on CNNs in Section 2.5. Lastly, the GNNs are introduced in Section 2.6.

2.1 Reinforcement learning

Reinforcement learning is one of the three paradigms in machine learning (ML). The other paradigms are *supervised learning* and *unsupervised learning*. Reinforcement learning refers to the learning process where an *agent* learns how to act in an environment from its observations. It is inspired on how animals and humans learn. As humans, we explore the world around us by means of our observations through our sensors (e.g. eyes, ears, tongue, skin, nose). Based on our observations we perform actions according to what we want to achieve (our goal). The consequence of our actions is then again observed. In this loop of observing and acting we learn how to behave in a world to satisfy our goal.

In this section, the main concepts of reinforcement learning are introduced which will be used throughout the rest of this thesis.

2.1.1 Markov Decision Processes

A Markov Decision Process (MDP) is a mathematical framework for modeling sequential decision-making under uncertainty. The goal in a MDP is for an agent to reach a certain desired state. The MDP formalizes a set of *states* of the environment and a set of *actions* for the agent to take. Furthermore, the MDP formalizes a *reward function* that gives a scalar reward for taking a certain action in a certain state and ending up in a next state. Likewise, a *transition function* is defined that gives the probability of changing from one specific state to the next state, given a certain action. Thus, a MDP is formally defined by the four-tuple $\mathcal{M} = (\mathcal{S}, \mathcal{A}, R, T)$, where:

- $\mathcal{S}$ is the set of possible *states*;
- $\mathcal{A}$ is the set of *actions*;
- $R : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow \mathbb{R}$ is the *reward function* which is a three-argument function that specifies the expected scalar valued *reward r* for taking action a in state s and ending up next state s_{t+1}, $R(s, a, s') = \mathbb{E}[r_{t+1} \mid s_t, a_t, s_{t+1}]$;
- $T : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0, 1]$ is the *transition function* which is a three-argument function that specifies the probability for the transition to a certain next state s_{t+1} given an action a and state s, $T(s, a, s') = P[s_{t+1} \mid s_t, a_t]$.

The complete dynamics of the environment are characterized by the transition and reward function. A MDP satisfies the *Markov Property* if the transition function depends only on the current state s and the performed action a in this state. The current state captures all relevant information from the history to predict the future, i.e. the future is independent of the past given the present. In reinforcement learning, the problem is often considered to satisfy the Markov Property. The Markov property is formulated in Equation 2.1 (Sutton and Barto, 2018).

$$P(s_{t+1} \mid s_t, a_t, r_t, s_{t-1}, a_{t-1}, r_{t-1} \cdots, s_0, a_0, r_0) = P(s_{t+1} \mid s_t, a_t) \tag{2.1}$$

In some cases, the entire environmental state cannot be observed. The agent gets an *observation o* of the environment by means of sensors. In case of a navigation robot that faces a blind wall, the robot does not know which part it is looking at without knowing the route it took. When the agent perceives the entire environmental state this is called *fully observable*, $o_t = s_t$. However, in any practical problem the state of the environment is not equal to the observation of the agent. When the observation is not sufficient to uniquely identify the underlying environment state, the environment is called *partially observable*.

In the formulation for the MDP presented above, the system was considered to be fully observable. In the partially observable setting, the problem transforms to a Partially Observable Markov Decision Process (POMDP) where the observation of the agent does not equal the state of the environment $o_t \neq s_t$. The POMDP is defined by the tuple $\mathcal{M} = (\mathcal{S}, \mathcal{A}, R, T, \Omega, O)$ (Silver, 2015a), where the additional Ω and O are:

- Ω is a set of *observations*;
- $O : \Omega \times \mathcal{A} \times \mathcal{S} \rightarrow [0,1]$ is the *observation function* which is a three-argument function that gives the probability for the observation o as a result of a certain next state s_{t+1} and an action a, $O(o, a, s') = P[o_{t+1} \mid s_{t+1}, a_t]$

The MDP framework is abstract and flexible and can be applied to many different problems in a wide range of areas. Above, the MDP is presented for the stochastic case but there are many variations possible, such as the POMDP which was discussed briefly. Another variation is the case where the MDPs is deterministic, which means that an action uniquely determines the next state, i.e. the probability distribution associated with each actions gives probability 1 to one of the states such that $T(s_t, a_t) = s_{t+1}$ and $R(s_t, a_t) = r_t$. Hence, the name Deterministic Markov Decision Process (DMDP).

2.1.2 Reinforcement learning framework

In reinforcement learning there is no external supervisor that tells the agent which actions to take. The agent learns by doing, by interacting with an environment it learns which of its actions are good. Which actions are good or bad is defined by the reward function. An agent is considered to be any high-level controller that is used for example to control a robot or to operate in a game. The agent analyzes which action to take by maximizing the rewards it receives. Therefore, the reinforcement learning problem boils down to maximizing a numerical reward signal. Hence, the name 'reinforcement learning', a stimulus - a reinforcer - is received after the agent performs actions.

A MDP formally describes an environment for reinforcement learning. The reinforcement learning problem is a discrete time stochastic control process. The agent interacts with the environment in discrete time steps. The agent start in a certain initial environment state $s_0 \in \mathcal{S}$ and it gets its initial observation $o_0 \in \Omega$. At each time step t, the agent observes the environment state $s_t \in \mathcal{S}$ through its *observations* $o_t \in \Omega$ by means of sensors (e.g. Light Detection and Ranging (LIDAR) sensor, camera, microphone). Every time step the agent takes an action $a_t \in \mathcal{A}$ according to a certain *policy* π. After taking the action, the agent receives a reward $r_t \in \mathbb{R}$ from the environment defined by the reward function $R(s_t, a_t, s_{t+1})$. After performing the action, the environment advances to a next state $s_{t+1} \in \mathcal{S}$ which is then again observed by the agent. The same cycle is repeated for the next time step. The described reinforcement learning framework is shown in Figure 2.1. The advancement to the next time step is at the boundary of the agent's input and the environment's output, this is indicated by the dashed line in Figure 2.1

In short, reinforcement learning is the learning process where an agent learns to map situations to actions in order to maximize its total reward. For the reinforcement learning problem it does

not matter what the states, goal, actions or reward signal are. It is a general framework for solving a wide range of problems with a similar setup. The context is given by the application at hand.

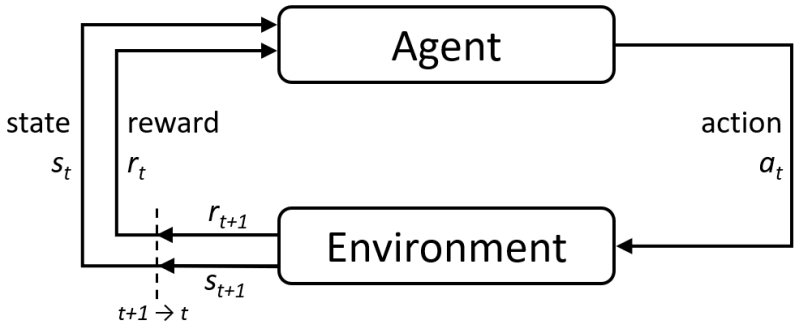


Figure 2.1: The interaction between agent and environment in the reinforcement learning framework (Sutton and Barto, 2018). s_t, r_t and a_t are respectively the state, reward and action at time t. s_{t+1} and r_{t+1} are the state and reward at time $t+1$ as a result of action a_t. The dashed line at the output of the environment indicates the advancement to the next time step.

2.1.3 Terminology within reinforcement learning

Many aspects of reinforcement learning are common across all methods. Therefore, the used terminology is stated here and briefly discussed.

Agent and environment

The learner and decision maker is called the *agent*. An agent follows a certain *policy* π to select actions such that it interacts with an *environment*. The environment selects the next observation and reward. The separation between agent and environment is made by a general rule: anything that cannot be changed arbitrarily by the agent is considered to be part of the environment. The environment can be *continuous* or *episodic*. In the episodic case, the environment resets when a certain terminal state is reached. The agent-environment interaction breaks down naturally in subsequences which are called *episodes*. An example of episodic task is playing an Atari game or navigation in a maze. Here the episode ends respectively when the agent dies/the level is finished or when the end of the maze is reached. In the continuous case, the task does not stop when a goal is reached. The problem goes on forever, e.g. an ongoing process-control task such as maximizing the power output of a wind turbine.

State, observation and action

State and action were extensively discussed in Section 2.1.1. In short, *state* refers to the situation the environment is in which is perceived by the agent through its *observation*. The observation is the information used to determine what happens next. *Action* can be any decision that the agent makes.

Goal, reward and return

The *goal* in reinforcement learning is a high-level objective of the agent, e.g. win a chess game, optimize power output of a wind turbine or reach the end of the maze with the shortest path. Goals are very diverse and depend on the application. The goal is communicated to the agent by means of a special signal, the *reward signal*. The idea that the reward signal formalizes the goal is one of the most distinctive features of reinforcement learning (Sutton and Barto, 2018).

The reward is always a simple scalar value, i.e. $r \in \mathbb{R}$. The reward signal r indicates the amount of the desirability (value) of an observed transition. As mentioned in section 2.1.1, the reward function is a function of state and action that returns the expected reward: $R(s_t, a_t, s_{t+1}) =$

$\mathbb{E}[r_t|s_t, a_t, s_{t+1}]$. In order to fulfill the goal in an optimal way, the agent maximizes the cumulative reward it receives in the long run. The cumulative reward is called *return* G_t, see Equation 2.2. In the case of a continuous task, the concept of *discounting* is introduced to limit the growth of the reward indefinitely. Furthermore, the *discount* factor $\gamma \in [0, 1]$ can be used to put emphasis on immediate rewards or future rewards. If γ is close to 0 this leads to *myopic* evaluation and when γ is close to 1 this leads to *far-sighted* evaluation. In the case of a continuous task, the return is defined as the total discounted reward from time step t, see Equation 2.3.

$$G_t = r_{t+1} + r_{t+2} + r_{t+3} + \cdots + r_T = \sum_{k=0}^{T} r_{t+1+k} \tag{2.2}$$

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k r_{t+1+k} \tag{2.3}$$

The reward signal can either be *dense* or *sparse*. In the dense reward case the agent receives non-zero rewards at every time step. In the sparse reward setting, the agent receives zero rewards most of the time and a significant reward when it reaches the termination state. The dense reward signal allows for faster learning because the agent learns directly what good and bad actions are. While in the sparse reward setting, often more iterations are needed because the agent first needs to get the large reward more or less by accident. Only after the termination state is reached once, the agent can start learning. In simple problems, constructing the dense reward function can be straight forward but can become very complex in challenging tasks or environments.

Policy

A *policy* π is the agent's behaviour. The policy is a mapping from state space to action space, it gives the action the agent should take in a specific state. The reward signal is the tool for communicating the goal to the agent. 'How' this goal is achieved is what the agent tries to learn by maximizing the return and is referred to as the policy. The *optimal policy* π^* is the policy that maximize the expected return when acting according to that policy J^π. The optimal policy is formally defined as in Equation 2.4.

$$\pi^* = \arg\max_\pi \left(J^\pi\right) = \arg\max_\pi \left(\mathbb{E}_\pi\left[G_t\right]\right) \tag{2.4}$$

The policy can come from a lookup table in the case of tabular Q-learning (Watkins, 1989). The policy can also be more involved using a function approximater such a neural network (NN) which is the case in Deep Q-Learning (DQN) (Mnih et al., 2015). The different methods to learn a policy are discussed in the next Section 2.1.4.

The policy can be deterministic, where the policy is a function that maps a state to an action, $a = \pi(s)$. The policy can also be stochastic, in that case the the policy is a probability distribution over the possible actions conditioned on the state. Hence, $\pi(s)$ is a probability distribution over $a \in \mathcal{A}$ and $\pi(a|s)$ is the probability of selecting action a in state s, i.e. $\pi(a|s) = P[a|s]$.

Value functions

The expected *value* of the return following a certain policy is given by the *state-value function*, often referred to as *value function*, $V^\pi(s) : \mathcal{S} \to \mathbb{R}$. The value function is used to evaluate the goodness/badness of states in the long run (Sutton and Barto, 2018). The value of a state is not simply the reward that the agent receives in that state. Formally, the state-value function $V^\pi(s)$ of a MDP is defined as the expected return starting from state s, and then following a particular policy π, see Equation 2.6. Note that the value of the terminal state (if there is one) is always zero (Sutton and Barto, 2018).

$$V^{\pi}(s) = \mathbb{E}_{\pi}\left[G_t \mid s_t = s\right] \tag{2.5}$$

$$= \mathbb{E}_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s\right] \tag{2.6}$$

Important to note is that the objective of the agent is to maximize the total reward not necessarily the immediate reward. A state can look bad if solely the immediate rewards are considered but might have a high value. Therefore, the value function is an important tool in determining the optimal policy in the reinforcement learning problem. The estimation of the value function is an essential part of many reinforcement learning algorithms. The agent tries to find the policy which results in the largest sum of rewards. Therefore, the agent must seek for the state with the highest value. Hence, the value function can be used to achieve the goal in an optimal way.

In addition to the value function there is the *state-action value function*, also referred to as *Q-value function* or *action-value function*, $Q^{\pi}(s,a) : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$. The Q-value function gives the expected return starting from state s, taking action a, and following policy π thereafter. The output of the Q-value function is called the *Q-value*. Opposed to the value function, the first action is explicitly defined in the Q-value function and is not necessarily according to the policy that is being followed. The definition is formulated in Equation 2.8.

$$Q^{\pi}(s,a) = \mathbb{E}_{\pi}\left[G_t \mid s_t = s, a_t = a\right] \tag{2.7}$$

$$= \mathbb{E}_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s, a_t = a\right] \tag{2.8}$$

When the agent acts according to an optimal policy the *optimal value function* and *optimal Q-value function* are defined as in Equation 2.9.

$$V^*(s) = \max_{\pi \in \Pi} V^{\pi}(s) \qquad , \qquad Q^*(s,a) = \max_{\pi \in \Pi} Q^{\pi}(s,a) \tag{2.9}$$

An important property of both value functions is that they can be written using a recursive relationship. This allows for *dynamic programming* algorithms to estimate the value of a policy given the transition function and reward function. This recursive formulation is called *Bellman equation* and can be formulated for both the value function and the Q-value function, see Equation 2.12 and Equation 2.15 (Sutton and Barto, 2018).

$$V^{\pi}(s) = \mathbb{E}_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s\right] \tag{2.10}$$

$$= \mathbb{E}_{\pi}\left[r_{t+1} + \gamma V^{\pi}(s_{t+1}) \mid s_t\right] \tag{2.11}$$

$$= \sum_{a \in \mathcal{A}} \pi(a|s) \sum_{s' \in \mathcal{S}} T(s,a,s')\left[R(s,a,s') + \gamma V^{\pi}\left(s'\right)\right] \tag{2.12}$$

$$Q^{\pi}(s,a) = \mathbb{E}_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s, a_t = a\right] \tag{2.13}$$

$$= \mathbb{E}_{\pi}\left[r_{t+1} + \gamma V^{\pi}(s_{t+1}) \mid s_t, a_t\right] \tag{2.14}$$

$$= \sum_{s' \in \mathcal{S}} T(s,a,s')\left[R(s,a,s') + \gamma V^{\pi}\left(s'\right)\right] \tag{2.15}$$

The value function and Q-value function are related to one another by the relation shown in Equation 2.16.

$$V^*(s) = \max_a Q^*(s,a) \tag{2.16}$$

The Bellman equation plays an important role in finding the optimal policy in many reinforcement learning algorithms, as will be presented in the next section.

2.1.4 Reinforcement learning methods

If the transition function $T(s_t, a_t, s_{t+1})$ and reward function $R(s_t, a_t, s_{t+1})$ are known, the optimal policy can be found by *planning*, using dynamic programming methods that exploit the Bellman equation shown in Equation 2.12. However, often the dynamics of the environment is not known upfront. The agent needs to estimate the value of taking a certain action in a state without the knowledge about the transition function and reward function. When this is the case, reinforcement learning algorithms are suitable to find the optimal policy. In reinforcement learning, an agent learns to map states to actions from the experience gained during interaction with the environment and receiving feedback by means of a reward signal.

Reinforcement learning algorithms can be categorized in two main categories, *model-free* algorithms and *model-based* algorithms (Sutton and Barto, 2018). In model-based reinforcement learning, the agent samples experiences from the environment to estimate the transition function $T(s_t, a_t, s_{t+1})$ and reward function $R(s_t, a_t, s_{t+1})$. Then, planning algorithms are used to find the optimal policy. In model-free reinforcement learning, there is no model made of the underlying dynamics, the Q-function or the policy are directly estimated from experiences.

Dynamic programming

The term 'dynamic programming' refers to the algorithms that can be used to compute optimal policies using a perfect model of the environment. A disadvantage of dynamic programming is that a perfect model is assumed, which is never known upfront in reinforcement learning. Furthermore, dynamic programming algorithms are computational expensive. Dynamic programming methods form the foundation of many reinforcement learning algorithms.

Policy iteration

Policy iteration consists of two simultaneous processes that interact in order to learn an optimal policy. Considering finite MDPs, there are a finite number of policies and therefore policy iteration converges to an optimal policy and optimal value function in a finite number of iterations. The algorithm for policy iteration is given in Algorithm 1.

Policy evaluation makes the value function consistent with the current policy and the *policy improvement* makes the policy greedy with respect to the current value function. The general idea of the interacting policy evaluation and policy improvement is referred to as *General Policy Iteration*, see Figure 2.2. Many reinforcement learning algorithms can be described using this principle, i.e. they all have policies and value functions, with the policy being improved with respect to the value function and vice versa. When both the policy evaluation and the policy improvement stabilize, an optimal policy and optimal value function are found according to the Bellman optimality equation, see Equation 2.17.

$$V^*(s) = \max_a \sum_{s' \in \mathcal{S}} T\left(s,a,s'\right)\left[R(s,a,s') + \gamma V^*\left(s'\right)\right] \tag{2.17}$$

Algorithm 1: Policy iteration (Sutton and Barto, 2018)

Algorithm parameter: θ, a small threshold $\theta > 0$ determining the accuracy of estimation
Initialize: $V(s) \in \mathbb{R}$ and $\pi(s) \in \Pi$ arbitrarily for all $s \in \mathcal{S}$
1. Policy Evaluation
while $\Delta > \theta$ **do**
  $\Delta \leftarrow 0$
  forall $s \in S$ **do**
    $v_{old} \leftarrow V(s)$
    $V(s) \leftarrow \sum_{s' \in \mathcal{S}} T(s, \pi(s), s') \left[R(s, \pi(s), s') + \gamma V^{\pi}(s') \right]$
    $\Delta \leftarrow \max(\Delta, |v_{old} - V(s)|)$
  end
end
2. Policy Improvement
policy stable $\leftarrow$ True
forall $s \in S$ **do**
  old action $\leftarrow \pi(s)$
  $\pi(s) \leftarrow \arg\max_a \sum_{s' \in \mathcal{S}} T(s, a, s') \left[R(s, a, s') + \gamma V^{\pi}(s') \right]$
  if *old action* $\neq \pi(s)$ **then** *policy stable* $\leftarrow$ False
end
if *policy stable* **then** stop and return $V(s) \approx V^*$ and $\pi \approx \pi^*$ **else** go to **1. Policy Evaluation**

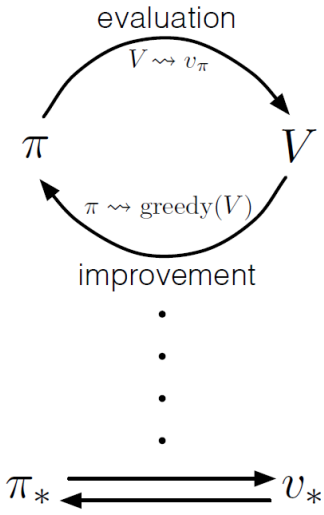


Figure 2.2: General Policy Iteration visualized (Sutton and Barto, 2018).

Value iteration

Value iteration is used in planning and is similar to policy iteration except that it skips the policy evaluation step. The policy evaluation step requires multiple sweeps through the state set. Value iteration uses the Bellman optimality equation in an update rule. The value iteration algorithm is shown in Algorithm 2.

Algorithm 2: Value iteration (Sutton and Barto, 2018)

Algorithm parameter: θ, a small threshold $\theta > 0$ determining the accuracy of estimation
Initialize: $V(s)$, for all $s \in \mathcal{S}$, arbitrarily except that $V(terminal) = 0$
while $\Delta > \theta$ **do**
  $\Delta \leftarrow 0$
  forall $s \in S$ **do**
    $v_{old} \leftarrow V(s)$
    $V(s) \leftarrow \max_a \sum_{s' \in \mathcal{S}} T(s,a,s') \left[R(s,a,s') + \gamma V\left(s'\right) \right]$
    $\Delta \leftarrow \max(\Delta, |v_{old} - V(s)|)$
  end
end
Output: Output a deterministic policy, $\pi \approx \pi^*$, such that
$\pi(s) = \arg\max_a \sum_{s' \in \mathcal{S}} T(s,a,s') \left[R(s,a,s') + \gamma V\left(s'\right) \right]$

Q-learning

Q-learning (Watkins and Dayan, 1992) is a model-free reinforcement learning algorithm. In Q-learning, the Q-value is estimated for a certain state s and action a. Q-learning is an *off-policy* algorithm, which means that the learned policy is not necessarily the same as the policy used for the action selection. Consequently, an optimal policy can be learned while taking random actions. The optimal policy π^* is obtained by acting greedy with respect to the optimal Q-value, see Equation 2.18.

$$\pi^*(s) = \underset{a \in A}{\arg\max} \left[Q^*(s,a) \right] \tag{2.18}$$

In tabular Q-learning, the agent learns to fill in a lookup table with an entry for each state-action pair. For each entry the Q-learning is updated using Equation 2.19. The update makes use of the Bellman equation. Intuition behind this update rule can be gained when comparing it to Equation 2.14.

$$Q_{t+1}(s_t,a_t) \leftarrow \underbrace{Q_t(s_t,a_t)}_{\text{current value}} + \alpha \overbrace{\underbrace{\left(r_t + \gamma \cdot \underbrace{\max_a Q_t(s_{t+1},a)}_{\text{estimate of optimal future value}}\right)}_{\text{temporal difference target}} - \underbrace{Q_t(s_t,a_t)}_{\text{current value}}}^{\text{temporal difference}} \tag{2.19}$$

Here α is the learning rate and γ is the discount factor. The difference between the current estimate of Q-value $Q_t(s_t,a_t)$ and the true future value is called *temporal difference (TD)* or *TD error*. However, the true future value is not known, the agent uses the current reward and the maximizing Q-value of the next state s_t as an estimate of the true value, which is called the *temporal difference target*. Note that the subscript t at the Q-values indicates the iteration steps of Q-learning. The complete algorithm is shown in algorithm 3. Tabular Q-learning converges,

given sufficient samples for each state and a discrete actions space (Watkins and Dayan, 1992).

Algorithm 3: Tabular Q-learning (Sutton and Barto, 2018)

Algorithm parameter: learning rate $\alpha \in (0,1]$, small $\epsilon > 0$
Initialize: $Q(s,a)$, for all $s \in \mathcal{S}$, $a \in \mathcal{A}$ arbitrarily except that $Q(terminal,\cdot) = 0$
for *each episode* **do**
  Initialize s
  for *each step t of episode* **do**
    choose $a = \pi(s)$ using policy derived from Q (e.g. ϵ-greedy)
    take action a
    observe reward r and new state s'
    $Q_{t+1}(s,a) \leftarrow Q_t(s,a) + \alpha\left[r + \gamma \max_a Q_t\left(s',a\right) - Q_t(s,a)\right]$
    $s \leftarrow s'$
  end
end

In addition to Q-learning there are many other methods that use the same principle of temporal difference, e.g. *SARSA* (Rummery and Niranjan, 1994). Opposed to Q-learning, SARSA is an *on-policy* algorithm. On-policy means that the agent updates the Q-value of the current state and action using the estimation of the Q-value for the next state and action of the current policy. On-policy algorithms can only train their policy using samples collected by their current policy. This makes off-policy algorithms more sample efficient because samples from the past can be used to train its policy. SARSA does not use a greedy policy for interacting in the environment, the maximization over all actions is not used. Therefore, the update rule is similar to Equation 2.19 except that there is no maximization over a, i.e. $\max_a Q_t(s_{t+1},a)$ is changed for $Q_t(s_{t+1},a_{t+1})$.

Q-learning works well in small domains, i.e. small state-action spaces. However, many real-world problems have large or continuous state-action spaces. When the state-action space is large, the tabular Q-learning method is not practical because the size of the lookup table will increase rapidly or is simply not possible in continuous problems. In order to deal with large state-action spaces, function approximators can be used to approximate the Q-function instead of keeping track of a lookup table. A commonly used function approximator is a (deep) NN. The approach of deep NNs as function approximator has shown to be successful, a popular algorithm using a deep NN as function approximator to estimate the Q-value is DQN (Mnih et al., 2015).

Supervised ML algorithms are used to approximate the Q-function by means of a NN with parameters θ. In this way the Q-value function is a function parametrized by the learned parameters $Q(s,a;\theta)$. The parameters θ need to be found such that $Q(s,a;\theta)$ converges to the optimal Q-value. The parameters can be updated using *gradient descent methods*, minimizing the mean squared error (MSE) between the current estimate of $Q(s,a)$ and the target, as shown in Equation 2.20.

$$L_i(\theta_i) = \mathbb{E}_{,}\left[\left(r + \gamma \max_{a'} Q\left(s',a';\theta_i^-\right) - Q(s,a;\theta_i)\right)^2\right] \tag{2.20}$$

Where $r + \gamma \max_{a'} Q\left(s',a';\theta_i^-\right)$ is the target value for iteration i. The parameters of the network θ_i change every iteration. When the targets depend on the changing network parameters this can cause instability. To make the training more stable, the target value depends on the θ_i^- which is an older version of the network. The parameters are updated $\theta_i^- = \theta_i$ every C number

of steps and is kept fixed between updates. The parameters are updated using gradient descent, see Equation 2.21 for the gradient.

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{,}\left[\left(r + \gamma \max_{a'} Q\left(s', a'; \theta_i^-\right) - Q(s, a; \theta_i)\right) \nabla_{\theta_i} Q(s, a; \theta_i)\right] \tag{2.21}$$

While the agent interacts with its environment it collects experiences. Each experience is defined by a tuple (s_t, a_t, r_t, s_{t+1}) and are stored in a so called *replay buffer*. The replay buffer allows for *experience replay* and was also introduced by (Mnih et al., 2015). From the replay buffer samples are taken in batches and the network is trained using supervised learning methods. The data efficiency is high because all samples can be used in each update. Furthermore, this approach prevents the high correlation between consecutive samples and therefore reduces the variance of the updates. Moreover, by using experience replay with batches, the individual experiences are averaged, this reduces variance in the parameters and makes the learning smooth (avoiding oscillations).

In practice, the Q-network will predict a Q-value for all actions given a state, i.e. the output of the Q-network is a value for each action. For example, take the scenario where an agent has four discrete actions (left, right, up, down). In this scenario, the Q-network will be given a state and the Q-network will output a value corresponding to each action in the given state. This way one does not have to compute the output for every action in each state but can simply take the maximum of the output of the Q-network to get the best action. The DQN algorithm with experience replay from (Mnih et al., 2015) is shown in Algorithm 4.

Algorithm 4: Deep Q-learning with experience replay (Mnih et al., 2015)

Initialize replay buffer D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
for *episode = 1, M* **do**
  Initialize sequences $s_1 = x_1$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  for *t=1, T* **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_1), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $\left(\phi_t, a_t, r_t, \phi_{t+1}\right)$ in D
    Sample random minibatch of transitions $\left(\phi_j, a_j, r_j, \phi_{j+t}\right)$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j + 1} \\ r_j + \gamma \max_{a'} \hat{Q}\left(\phi_{j+1}, a'; \theta^-\right) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q\left(\phi_j, a_j; \theta\right)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  end
end

2.1.5 Model-based reinforcement learning

In *model-based* reinforcement learning an environment model is learned. Anything that can be used to predict how the environment will respond to actions in terms of reward and state transition is considered an environment model (Sutton and Barto, 2018). Therefore, environment models consist of two parts. First, a transition model which encodes the effect of actions on the change in state $T(s, a, s')$. Second, a reward model which encodes the reward signal expected in a certain state or corresponding to a state-action pair $R(s, a, s')$.

The environment model can be learned from the observed data samples (s_t, a_t, r_t, s_{t+1}) which an agent gathered during deployment in an environment. The learned environment model is then used for planning in order to find an optimal policy. Hence, the first step is to learn a model for the environment dynamics. Learning the environment model is a supervised problem where the state-action pair is mapped to a next state and reward. Here, the focus is on one-step models. In general, there are three types of dynamic functions (Moerland et al., 2021):

- *forward model*: $(s_t, a_t) \rightarrow s_{t+1}$, the next state is predicted given the current state and chosen action;
- *reverse model*: $s_{t+1} \rightarrow (s_t, a_t)$, the state and action are predicted which resulted in a particular state;
- *inverse model*: $(s_t, s_{t+1}) \rightarrow a_t$, this predicts which action is needed to get form one state to another state.

As in model-free reinforcement learning, there are tabular methods and function approximation methods. The tabular case is applicable to discrete MDPs. In general, practical problems are continuous and have large state-action spaces which make the tabular method infeasible. Therefore, the function approximation methods are the preferred choice in practice and often NNs are used. In model-based reinforcement learning the forward model is the most common model to do planning by predicting the future. After obtaining a model for the transition and reward, the future states and corresponding rewards can be predicted. The model of the environment is used to plan the best actions by looking ahead. The model-based framework is summarized in Figure 2.3.

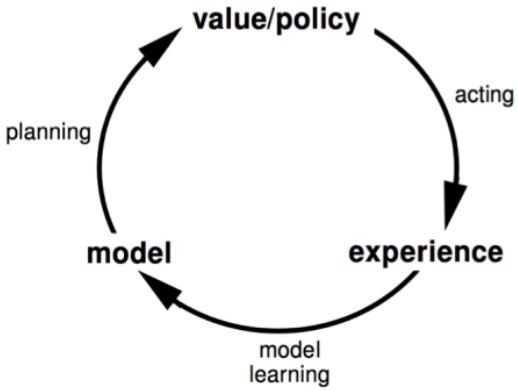


Figure 2.3: The model-based learning framework (Silver, 2015b).

2.2 State representation learning

Reinforcement learning methods have shown to be able to learn impressive policies from raw pixel input, (Mnih et al., 2015). The raw input data is often high dimensional and includes a lot of irrelevant information, e.g. static background. Therefore, high dimensional input data increases computational load during training and more samples are needed to learn to ignore the unimportant features. Sample inefficiency is an important problem in reinforcement learning which is formulated as the *curse of dimensionality*: 'as the number of features grows, the amount of data that is needed to generalize accurately grows exponentially'. Therefore, it is desired to keep the input data dimensionality small with expressive features. This way the model can be simple (fewer parameters needed) and learn fast (no irrelevant information in the input).

Consider the Pong game, an Atari game that simulates table tennis. Players control an in-game paddle in order to hit the ball back and forth. Points are earned when the opponent fails to return the ball. An agent can be trained to play as one of the players in the Pong game. The observation would be a frame of the game, or maybe several consecutive frames in order to know moving directions of the paddles and ball. The frame contains a lot of unimportant information, e.g. the background or the net in the middle. The only information needed to play the Pong game is the positions of the paddles and the position of the ball. This would mean 6 variables as opposed to the 160 x 192 pixels of an Atari game. The input data is often larger by a lot than the number of relevant variables needed to design a controller.

Representation learning is a core research topic within control. Traditionally, the observation space is constructed manually. This is often feasible but for more complex tasks this is non-trivial. Moreover, in simulations we have access to the true states, this is of course not the case in the real world. In the real world this task is more complex, from high dimensional sensor data the relevant information is extracted. Hence, the main idea is to map observations from the high dimensional observation space to a lower dimensional state space. A good state representation is crucial for down stream control tasks.

Reinforcement learning methods have shown great success in learning from raw input data. The NNs are well suited for learning internal representations that are lower dimensional and encapsulate relevant information. By using more layers in the NN, different levels of abstractions are learned. The input data is often mapped to actions in the output layer, (Mnih et al., 2015). Therefore, the learned representations are not interpretable as it is formed within the NN. Additionally, a very specific controller is trained which has to be re-trained when the input changes slightly or the task at hand changes. For example, when an agent is trained to play the Pong game it needs to be re-trained when the input frames are rotated 90 degrees. However, there are techniques to account for this but this introduces additional complexity to the NNs.

In order to make learned representations more interpretable and reusable, a lower dimensional state representation is often first learned. State representation learning aims at learning a state abstraction from observations in an unsupervised or self-supervised manner. The learned state abstraction is then used in down stream reinforcement learning algorithms to speed up the learning process. The aim is to learn a state representation that allows to learn a policy that is optimal when applied in the original state space. In other words, it is desired to have a mapping from the original MDP to a more compact MDP that preserves the underlying dynamics (transition/reward model) of the original MDP. This is referred to as a *MDP homomorphism*. This allows for solving the task in the compact MDP and projecting the solution back into the original problem.

There are three main approaches that can be distinguished for learning a state representation. The first approach is focused on dimensionality reduction by means of compression. For example, Principal Component Analysis (PCA) (Fodor, 2002) is a well known dimensionality reduction technique. In ML the autoencoder framework is widely used. The autoencoder aims at extracting features and generating a low dimensional representation as a self-supervised problem. The input data is compressed by means of hidden layers (encoder). The compressed data contains the lower dimensional representation. The compressed representation is then reconstructed (decoder) into the original dimension. By defining an error between the input data and the reconstructed data, a neural network is trained to learn a lower dimensional representation. A disadvantage of this approach is that task relevant information might be ignored and task irrelevant information might be encoded. For example, a bullet in an Atari game might look visually unimportant but is of importance for solving the task.

Another approach for learning state representations is by means of robotic priors (Jonschkowski and Brock, 2015). This approach is particularly applicable in the context of robotics, hence the name. This idea is directed towards formulating loss functions that incorporate prior

knowledge of the environment to learn a state representation. The reward signal is used to learn a state representation of the environment. The reward signal is directly related to the task that is being solved. Therefore, this method facilitates learning task relevant information and ignoring information that is not relevant.

Thirdly, there is the approach of learning a transition model (Lesort et al., 2018). The transition model predicts the next state s_{t+1} as a function of the previous state s_t and action a_t. In the context of state representation learning, the aim is to learn a mapping from the high dimensional observation o_t to s_t. This is achieved by using the transition model in two steps. First, the observation o_t is encoded into s_t and then the transition from s_t to is $\hat{s}_{t+1}$ is predicted. The true next state s_{t+1} is available and therefore an error can be computed between the true next state s_{t+1} and the predicted next state $\hat{s}_{t+1}$, see Figure 2.4. This error is then used to train the transition model and encoding model (which is the mapping from o_t to s_t).

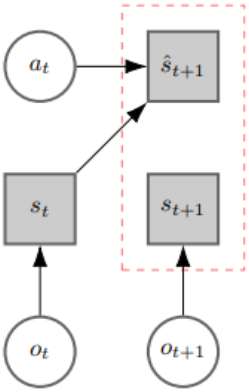


Figure 2.4: Learning a state representation by predicting the next state s_{t+1} from s_t and a_t. The error between $\hat{s}_{t+1}$ and s_{t+1} is used to train a transition model and encoding model simultaneously (Lesort et al., 2018).

2.3 Graph

Before the graph neural network is discussed it is necessary to elaborate on what a graph is. In this section, a formal description of a graph is given along with the relevant terminology related to graphs.

2.3.1 Graph definition

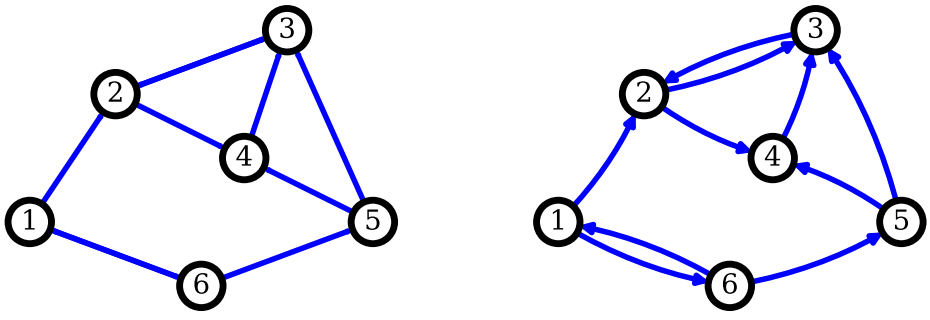
Graphs are a universal data structure for describing complex systems. A *graph* is a data structure format which can be found in a wide range of areas because many practical problems can be represented using graphs. Generally, a graph is a collection of objects (nodes) along with a set of relations (edges) between pairs of these objects. An example in social sciences, a graph can be used to model a social network, where nodes are individuals and edges represent whether two individuals are friends (Zachary, 1977). In chemistry, molecules can be modelled as a graph, where nodes are atoms and edges are bonds between these atoms (Gilmer et al., 2017).

A graph is formally defined as $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where $\mathcal{V}$ is the set of *nodes*, and $\mathcal{E}$ is the set of *edges* that connect the nodes. Both these sets are usually taken to be finite. $\mathcal{V}$ is often assumed to be non-empty, while $\mathcal{E}$ is allowed to be the empty set. Edges are represented as tuple (u, v) of nodes $u, v \in \mathcal{V}$, e.g. an edge going from node $u \in \mathcal{V}$ to node $v \in \mathcal{V}$ is denoted as $(u, v) \in \mathcal{E}$. The *order* of the graph is the number of nodes, $|\mathcal{V}|$. The *size* of the graph is the number of edges in the graph, $|\mathcal{E}|$.

In literature the terms 'nodes' and 'vertices' are used interchangeably. Likewise, for the terms 'edges' and 'links'. Throughout the rest of the report, the term nodes and edges will be used.

2.3.2 Directed and undirected

The formal definition of a graph mentioned above is very broad and there are many extensions to this formal definition which result in different flavours of graphs. A common distinction in types occurs on the type of edges. There are two types of edges, *directed* (uni-directional) edges and *undirected* (bi-directional) edges, see Figure 2.5. Typically in a graph, all edges are either directed or undirected. A graph with solely directed edges is called a *directed graph* or *digraph* and a graph with solely undirected edges is called an *undirected graph* or just *graph*, see Figure 2.5. Note that the two edges between nodes 1 and 6 in Figure 2.5b are actually an undirected edge represented as 2 directed edges.



(a) Undirected graph. **(b)** Directed graph.

Figure 2.5: Two types of graphs with different edges in blue. Undirected graph (left) with undirected edges, direct graph (right) with directed edges. Note that these graphs are **not** equivalent. However, the same node structure is used to emphasize the difference in edge types.

The usage of directed/undirected graphs is dependent on the application. An example of an undirected graph is a social network, where nodes represent people and edges represent whether people are friends. An example of a directed graph is the world wide web, where nodes are web pages and an edge represent a link to another web page. The social network is an undirected graph because a friendship is a mutual relationship, whereas web pages do not necessarily link to each other and are often one way.

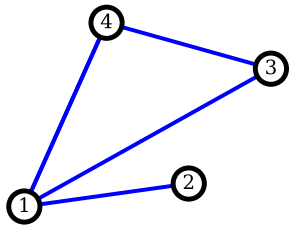
2.3.3 Weighted and unweighted

In graphs where edges should be treated with different importance, *weights* can be used. Weights are real valued scalars. In a road network where the nodes in a graph are places and the edges are roads, the edges can have weights that represent the length of the road. In unweighted graphs the edges are considered with the same importance, i.e. all edges have a weight of 1.

2.3.4 Adjacency matrix

The *adjacency matrix* $\mathbf{A} \in \mathbb{R}^{|\mathcal{V}|\times|\mathcal{V}|}$ is a matrix representation of a finite graph. The elements of the adjacency matrix indicate whether pairs of nodes are 'adjacent' in the graph. The nodes are ordered such that each node indexes to a particular row and column in the adjacency matrix. The adjacency matrix is always square. The presence of an edge between node u and node v is then represented using a 1, if $(u,v) \in \mathcal{E}$ then $\mathbf{A}[u,v] = 1$ and if $(u,v) \notin \mathcal{E}$ then $\mathbf{A}[u,v] = 0$. In short, the adjacency matrix is a (0,1)-matrix where 1 indicates a connection between nodes and 0 indicates that there is not a connection between the nodes. In Figure 2.6 an example of the adjacency matrix for both a directed and undirected graph is shown.

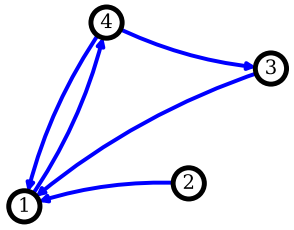
In case of the weighted edges, each cell in the adjacency matrix $\mathbf{A}[u,v]$ represents the edge weight of going from node u to node v. In case of an undirected graph, the adjacency matrix is symmetric and has zeros on its diagonal. In the directed case (edge direction matters), the adjacency matrix is not necessarily symmetric. The adjacency matrix is used a lot in graph analysis because it allows for use of linear algebra. Note that it is often assumed that the edge from a node to itself (*self-loop*) is excluded, $\mathbf{A}[u,u] = 0$. It depends on the application at hand whether you want to include self-loops, e.g. in a social network you will probably exclude self-loops (e.g. being friends with yourself).



(a) Undirected graph.

$$\begin{array}{c} \\ node\ 1 \\ node\ 2 \\ node\ 3 \\ node\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} node\ 1 & node\ 2 & node\ 3 & node\ 4 \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

(b) Adjacency matrix corresponding to the undirected graph (left)



(c) Directed graph.

$$\begin{array}{c} \\ node\ 1 \\ node\ 2 \\ node\ 3 \\ node\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} node\ 1 & node\ 2 & node\ 3 & node\ 4 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

(d) Adjacency matrix corresponding to the directed graph (left)

Figure 2.6: Adjacency matrix example for an undirected graph and a directed graph.

The adjacency matrix is used as a data structure for representing graphs in computer programs. Graphs tend to get very large and therefore adjacency matrices grow rapidly in size, $O(|\mathcal{V}|^2)$. Moreover, in real-world problems the graph is often sparse and therefore space inefficient. Alternatively, the graph can be represented using an *adjacency list* which is more space efficient for representing sparse graphs. The adjacency list is a map from nodes to a list of neighboring nodes. Another way to represent a graph is by means of an *edge list*, it is an unordered list of edges where edges are represented as pairs of nodes.

2.3.5 Degree matrix

Another concept often used in graph theory is *node degree*. Node degree k_i is the number of edges adjacent to node i. *Average degree* is the average amount of edges adjacent to a node in a graph, i.e. for an undirected graph the average degree is $\overline{k} = \frac{1}{|\mathcal{V}|}\sum_{i=1}^{|\mathcal{V}|} k_i = \frac{2|\mathcal{E}|}{|\mathcal{V}|}$. In directed graphs a distinction between in-coming and out-going degree can be defined, it is referred to as *in-degree* and *out-degree*, the sum is equal to the (total) degree. The average degree for a directed graph is $\overline{k} = \frac{|\mathcal{E}|}{|\mathcal{V}|}$. The *degree matrix* $\mathbf{D}$ is a diagonal matrix which contains information about the degree of each node, i.e. the number of edges attached to that node.

$$\begin{bmatrix} 3 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 2 \end{bmatrix}$$

(a) Degree matrix corresponding to undirected graph in Figure 2.6b

$$\underbrace{\begin{bmatrix} 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{in\text{-}degree\ matrix}, \underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 2 \end{bmatrix}}_{out\text{-}degree\ matrix}$$

(b) In-degree matrix and out-degree matrix corresponding to the directed graph shown in Figure 2.6d

Figure 2.7: Degree matrix examples

2.3.6 Laplacian matrix

Adjacency matrices can be used to represent a graph without the loss of information. However, there are other matrix representations of graphs that have useful mathematical properties. An example is the *Laplacian matrix*, which is a representation of the graph and can be used to find useful properties of the graph, e.g. clusters. Furthermore, the Laplacian can be used to construct low dimensional embeddings.

Given a *simple graph* (an unweighted, undirected graph containing no graph loops or multiple edges) the unnormalized Laplacian matrix is defined as $\mathbf{L} = \mathbf{D} - \mathbf{A}$, where $\mathbf{D}$ is the degree matrix and $\mathbf{A}$ is the adjacency matrix.

2.3.7 Other types of graphs

There are many structures possible with graphs. For example, a graph that can have multiple undirected edges is called a *multigraph* as opposed to the *simple graphs*. Furthermore, a graph where each node is connected to every node in the graph and therefore have the maximum amount of edge is called a *complete graph*. Moreover, there can also be graphs considered that have different *types* of edges. A discussion on all the different types of graphs (acyclic, bipartile etc.) is irrelevant for the purpose of graphs here. For a more detailed discussion on all the different types of graphs, the reader is referred to Easley and Kleinberg (2010) and Barabási and Pósfai (2016).

2.3.8 Features

In many cases the graph is equipped with *attributes* or *features*. The graph can have features on different levels, i.e. a graph can have edge-level features, node-level features or global-level features. Considering the example of modeling the interaction between rubber balls connected by springs in an arbitrary gravitational field. A global feature might be the gravitational field. The nodes might represent the balls where the node features can be position, velocity, mass, color. The edges can represent the presence of springs between different balls, in which case the edge features will be springs constants.

The features in a graph can be of any representation format. In deep learning applications, the features are most often real-valued vectors or tensors. However, the data structure can also be sequences, sets or graphs. The problem at hand will determine what representation should be used as features. For a more in depth discussion on graphs applied in ML the reader is referred to Hamilton (2020).

2.4 Neural networks

Artificial neural networks, or simply called neural networks (NNs), are a set of algorithms inspired by the working principle of biological neurons. A NN is composed of artificial neurons which receive an input and process this to recognize patterns. Real-world input data is transformed into numerical data that is referred to as the input *feature vector*. Each element of the feature vector describes a real-world property. A NN provides a mapping from the input feature vector to an output feature vector which can be related to the physical world (Bishop, 2006). The input can be all kinds of data, e.g. images, sound, text.

The architecture of a NN is based on a collection of interconnected artificial neurons, similar to the neurons in an animal brain. The (artificial) neurons are simple processing elements which receive a numerical input signal, process this signal and output a numerical signal that serves as the input of the next neuron. A neuron transforms the input to output through a non-linear function. In the context of NNs, neurons are referred to as *units*. The connections between neurons indicate the flow of the signals. These connections typically have weights that strengthen/weaken the signals between neurons. Bringing this all together, the output of a neuron is the result of a non-linear function on the input feature vector multiplied by the weights plus some bias. The non-linear function is referred to as the *activation function*. The weights and biases are the parameters θ of the model. How the output of a single neuron is calculated is shown in Equation 2.22 and visualized in Figure 2.8. Here y is the output signal, $f(.)$ is a non-linear activation function, x is the input signal, $w \in \theta$ is the weight, $b \in \theta$ is a bias that serves as an offset but is sometimes omitted in certain architectures.

$$y = f\left(\sum_{i=1}^{M} w_i x_i + b\right) \tag{2.22}$$

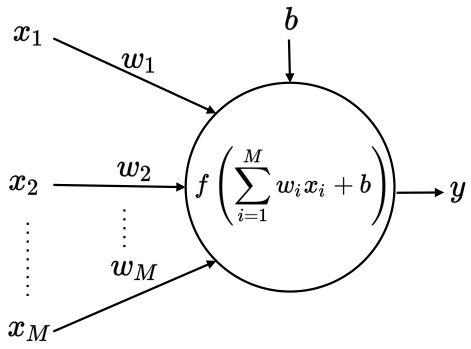


Figure 2.8: Single neuron

Neurons are put in layers to construct a NN. A NN typically has multiple layers and is also referred to as multilayer perceptron (MLP). A NN consists of an input layer, one or more hidden layers and an output layer. Each layer has multiple neurons and each neuron is interconnected via weighted connections to neurons in the subsequent layer. When neurons are connected to all subsequent neurons, this is called *fully connected*. The general architecture for a NN is shown in Figure 2.9. NNs with many layers are referred to as *deep* NNs and are studied in the field of *deep learning*. Note that the input layer neurons, hidden layers and output layer are colored green, blue and red respectively. This convention will be used throughout the rest of the thesis.

Finding the mapping from input vector to output is done by calculating the values for all nodes in the network. The value for the j^{th} neuron in the N^{th} layer that is connected to D previous neurons can be calculated as is shown in Equation 2.23. This notation is adopted from Bishop

(2006). Note that h is used for *hidden* neurons, $w^N_{j,i}$ are the weights associated with the i^{th} neuron in the previous layer and $w^N_{j,0}$ represents the bias. This equation is the same for all neurons in the network. The *forward pass* is from left to right in which the output of the N$^{\text{th}}$ layer is the input of the N+1$^{\text{th}}$ layer. This way the layers are function compositions of the previous layers and are chained as is shown in Equation 2.24.

$$h^{(N)}_j = f\left(\sum_{i=1}^{D} w^{(N)}_{j,i} x_i + w^{(N)}_{j,0}\right) \tag{2.23}$$

$$h^{(N+1)}_j = f\left(\sum_{j=1}^{L} w^{(N+1)}_{k,j} \cdot f\left(\sum_{i=1}^{D} w^{(N)}_{j,i} x_i + w^{(N)}_{j,0}\right) + w^{(N+1)}_{k,0}\right) \tag{2.24}$$

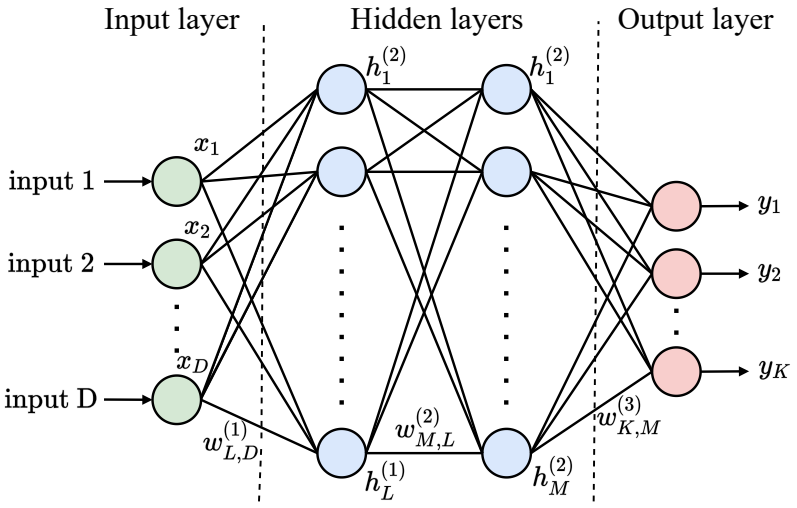


Figure 2.9: Neurons are represented as circles, and connections with weights are represented using links. Note that the biases are omitted in the figure for clarity. The bias parameters would result in additional input neuron x_0 and hidden neurons $h_{1,0}$, $h_{2,0}$ that output constant value 1 and are connected to the neurons in the subsequent layers via the corresponding weights.

Equation 2.23 can be written using matrix-vector notation to compute the output of all neurons in a layer. In Equation 2.25 the calculation for all neurons in the N$^{\text{th}}$ layer is shown, with the written out example for the first hidden layer in Figure 2.9. Note that the bias x_0 is included in the $\mathbf{x}$ vector, this will always have value 1 after which it is multiplied by the corresponding weight, $w^{(N)}_{x,0}$. The non-linear activation function is applied element wise to get the output of each neuron in the layer. Note that the dimensions of the vectors and matrices for the written out example of the first hidden layer are: $\mathbf{h}^{(N)} \in \mathbb{R}^{L\times 1}$, $\mathbf{W}^{(N)} \in \mathbb{R}^{L\times(D+1)}$ and $\mathbf{x} \in \mathbb{R}^{(D+1)\times 1}$.

$$\begin{gathered}\mathbf{h}^{(N)} = f\left(\mathbf{W}^{(N)}\mathbf{x}\right) \\ \begin{bmatrix} h^{(N)}_1 \\ \vdots \\ h^{(N)}_L \end{bmatrix} = f\left(\begin{bmatrix} w^{(N)}_{1,0} & \cdots & w^{(N)}_{1,D} \\ \vdots & \ddots & \vdots \\ w^{(N)}_{L,0} & \cdots & w^{(N)}_{L,D} \end{bmatrix}\begin{bmatrix} x_0 \\ \vdots \\ x_D \end{bmatrix}\right)\end{gathered} \tag{2.25}$$

Typically, for the hidden layers non-linear activation functions are used. These functions are differentiable functions. Commonly used activation functions (e.g. *sigmoid* function, *hyperbolic tangent (tanh)* function, *Rectified Linear Unit (ReLU)*) and the corresponding derivatives are shown in Appendix A. The strength of neural networks is that the mapping from input

to output is non-linear. Therefore, the functions that neural networks can approximate can be very complex. According to the *universal approximation theorem* (Csáji, 2001) neural networks can approximate any continuous function.

The parameters θ of the network are initialized at random. In order to find a meaningful mapping from input to output the network needs to be "trained". First, a *loss function* $\mathcal{L}$ is defined which quantifies the error between the output of the neural network and the target values for a certain input. Given a set of input vectors and corresponding targets (output vectors) the network needs to minimize the loss function. This loss function is dependent on the parameters. Hence, the parameters are tuned to minimize the loss function and fit to the target values. The tuning of the parameters is done iteratively and consists mainly of two parts.

First, the gradient of the error with respect to the parameters is calculated, $\nabla\mathcal{L}\left(x_i;\theta^{(t)}\right)$. This is achieved using some form of *back propagation*. During back propagation, the error is passed through the output layer back to the individual nodes in the network. Hence, the name back propagation. A NN is in essence a bunch of nested differentiable functions. Therefore, the chain rule can be used to compute the gradient of the error with respect to the parameters. The second step is to update the parameters in the network accordingly using the calculated gradient. A well known method to train the weights of a NN is *stochastic gradient descent*. In stochastic gradient descent the parameters are updated as is shown in Equation 2.26 (Bishop, 2006). Here α is the learning rate. The loss function is minimized by updating the parameters of the network in the direction of the negative gradient.

$$\theta^{(t+1)} = \theta^{(t)} - \alpha\nabla\mathcal{L}\left(x_i;\theta^{(t)}\right) \tag{2.26}$$

A widely used loss function for training NNs in regression problems is the mean squared error (MSE). The MSE is the sum of squared differences between the predicted values and the target values, see Equation 2.27. In Equation 2.27, n indicates the amount of samples that are considered.

$$\mathcal{L}_{\text{MSE}} = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - \hat{y}_i\right)^2 \tag{2.27}$$

Deep NNs are difficult to train using standard gradient descent. Therefore, new optimization methods have been proposed that converge more reliably. An often used optimizer is Adaptive Moment Estimation (ADAM) (Kingma and Ba, 2014).

NNs come in many different flavours, e.g. CNN, RNN. The type of NN that is needed is mostly determined by the nature of the input data. CNNs are particularly useful when 2D data is considered, e.g. images. While, RNN are useful on data with a clear causal structure, e.g. audio. For an extensive discussion on different NN architectures the reader is referred to Goodfellow et al. (2016).

2.5 Convolutional neural networks

Prior to the discussion on the application of NNs on graph structured, it is relevant to look at the CNN model because graph neural networks are based on the same principle.

The CNN architecture is suited for input data that is 2D, e.g. color images consisting of three 2D arrays containing pixel intensities in three different color channels (LeCun et al., 2015). A famous application of a CNN is on the MNIST dataset (LeCun and Cortes, 2010), here low-resolution images of handwritten digits are classified using a CNN. The CNN can be trained similar to a 'normal' NN, i.e. using backpropagation (LeCun et al., 1990).

Image classification using a traditional NN is impractical. Consider an image of size 100×100 pixels, this will be represented in an array with size $100 \times 100 \times 3$. The number of input nodes is already 30000 and increases rapidly with the size of the images. If 32 hidden nodes are used for the first layer in the network, the size of the input weights matrix would be almost 1 million. Imagine what happens with multiple layers and larger images. Besides the fact that NNs do not scale well, squeezing the images into vectors ignores the 2D spatial inductive bias. In an image, neighbouring pixels are often correlated. Hence, the pattern recognition should be location invariant which is realised using CNNs.

The CNN can have different types of layers. A CNN typically has an input layer and an output layer, as well as hidden layers. The hidden layers are typically *convolutional layers*, *activation layers*, *pooling layers* and *fully connected layers*.

2.5.1 Convolutional layer

The first layer of a CNN is always a convolutional layer and is the core principle of this type of NN. Hence, the name 'convolutional' NNs. Let us consider each pixel location in a 5×5 image to be an input neuron. In the normal NN architecture each input neuron connects to a hidden neuron in the next layer. However, in the CNN architecture learnable *filters* (or *kernels*) are used. The filter is a 2D array of weights and works on a same sized part of the image. The part in the image on which the filter works is called the *local receptive field*. The values in the filter are multiplied element wise by the values in the local receptive field and then summed into a single value, see Figure 2.10. This is repeated for every location in the image, by sliding the receptive field across the image, i.e. the filter is convolved with the input image. The operation is shown in Equation 2.28, i, j indicate the position on the input layer or hidden layer, r, c refer to the row and column in the filter.

$$h_{i,j} = (\mathbf{W} * \mathbf{X})_{(i,j)} = \sum_{r=0}^{2} \sum_{c=0}^{2} w_{(r,c)} x_{(i+r,j+c)} \tag{2.28}$$

Note that Equation 2.28 is actually the cross-correlation. Many libraries implement cross-correlation but call it convolution, because it is the same except that the kernel is flipped (Goodfellow et al., 2016). In the NN case it does not matter whether a flipped kernel is learned as long it is done consistently.

The local receptive field is a small window on the input neurons (input pixels) that connects to a hidden neuron. The hidden neuron learns to analyze a particular local receptive field. The amount of pixels by which the local receptive field is shifted, is indicated by the *stride*. Note that the amount of neurons in the hidden layer is smaller because the filter can be moved 3 neurons across when a stride of 1 is used. The size of the hidden layer is dependent on the stride and whether the input image is padded with zeros.

The filter is the set of weights for the hidden neurons and are shared among neurons, a bias can also be used here. The map from input layer to hidden layer is called *feature map*. The more filters are used, the greater the depth of the feature map. By sharing weights the number

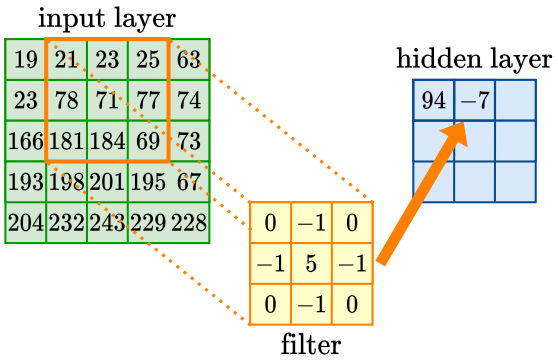


Figure 2.10: Graphical representation how the convolution is performed on a 5×5 pixel image (green frame), using a 3×3 size filter (orange frame) resulting in a hidden layer (blue frame). The corresponding calculation for this specific position of the filter is: $21 \times 0 + 23 \times -1 + 25 \times 0 + 78 \times -1 + 71 \times 5 + 77 \times -1 + 181 \times 0 + 184 \times -1 + 69 \times 0 = -7$

of parameters is greatly reduced. Consider an image of 100×100 and a filter of size 7×7 is used, for each feature map using a filter of size 7×7 the number of parameters is 50 (7×7 for the shared weights +1 for the shared bias). When we use 20 feature maps, there are $20 \times 50 = 1000$ parameters to learn. This allows for faster training and will aid the building of deeper networks. It is not a totally fair comparison because the models are fundamentally different where the CNN exploits spatially local correlation by enforcing a local receptive field between neurons of adjacent layers.

There is much more to say about CNNs but it is out of the scope of this thesis to give an extensive survey on CNNs. For more information on CNNs, the reader is referred to Goodfellow et al. (2016). The CNN architecture operates on grid-like structured data, i.e. Euclidean data. An emerging field in AI is *geometric deep learning* which focuses on generalizing the structured deep-learning toolbox for a non-Euclidean domain such as graphs. GNN is a NN architectures that operates on graphs in a similar way CNN operates on structured 2D grid-like data. The next section is dedicated to GNNs.

2.6 Graph neural networks

A GNN is a NN model suitable for operating on graph-structured data. Although the first ideas on GNNs have been around for more than a decade (Gori et al., 2005; Scarselli et al., 2009), the GNNs gained a lot of interest recently. The field of GNNs, or more generally geometric deep learning, is one of the most active areas of deep learning at the time of writing.

GNNs are the most general class of deep learning architectures, i.e. most deep learning architectures can be understood as a special case of a GNN with additional geometric structure (Bronstein et al., 2021). For example, a CNN works on grid-structured data such as images. A CNN applied on image data can be seen as a special case of a GNN, where a 2D grid-structure is enforced on the graph and the nodes correspond to the pixels. Neighboring pixels are related, this is realised by means of an edge between related nodes. A similar notion can be made for RNNs, where the nodes are in a directed graph along the temporal sequence.

A GNN can be seen as a generalization of a CNN to non-Euclidean data structures (Bronstein et al., 2016). There are some additional challenges with this generalization. As opposed to CNNs, GNNs have an arbitrary size and complex topological structure. Furthermore, there is not a fixed node ordering or reference point. The shape of the graph is not defined, while in an image the pixels are fixed on a grid-structure. Moreover, the convolution operation in a CNN is not permutation-invariant. For a graph, the order of the nodes is undefined. Therefore, in a GNN the operation should be permutation-invariant because the graph is a set of nodes and different order of the nodes should not change the output of the GNN. At last, graphs are often dynamic and have multimodal features.

A naive approach of implementing a GNN would be to concatenate the adjacency matrix with the features in the graph and feed it through a deep NN. However, this is in conflict with the generalization properties that were mentioned in the previous paragraph. The naive approach does not scale $O(N)$, is not applicable to graphs of different sizes and not invariant to the node ordering.

Since the release of the first papers on GNNs, many different variations on the GNN have been presented. Nevertheless, the majority of the presented architectures is derived from one of the three flavours: GCN (Graph Convolutional Network) (Kipf and Welling, 2016), GAT (Graph Attention Network) (Veličković et al., 2018) and Message Passing Neural Network (MPNN) (Gilmer et al., 2017). However, it must be noted that this categorization is very general and not all encompassing (Bronstein et al., 2021). In Gilmer et al. (2017) the MPNN was introduced which is a general framework for supervised learning on graphs. The MPNN is a GNN framework that unifies the commonalities between existing NN models for graph structured data. The GCN and GAT can be seen as special cases of the MPNN (Bronstein et al., 2021). Therefore, the discussion on the GNN model in the next section is focused on the MPNN framework which unifies various GNN architectures by analogy to message passing in graphical models. A framework presented by Battaglia et al. (2018) called Graph Network (GN) generalizes and extends the MPNN framework.

2.6.1 Neural Message Passing

The defining feature of a GNN is that it uses *neural message passing* in which vector messages are passed between nodes and updated using a NN. The notion of message passing is not new, it is an important part of inference on graphical models (Section 8.4 in Bishop (2006)). To explain the working principle of a GNN the focus is on the neural message passing framework (Gilmer et al., 2017).

As an example, take the simple graph shown in Figure 2.11 as input graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. Nodes are identified by a unique index $i \in \mathcal{V}$, ranging from 1 to $|\mathcal{V}|$. Edges are represented by an ordered

pair of nodes $(i, j) \in \mathcal{V} \times \mathcal{V}$. In case of undirected graphs where i and j are connected, both (i, j) and (j, i) are in the set of edges $\mathcal{E}$.

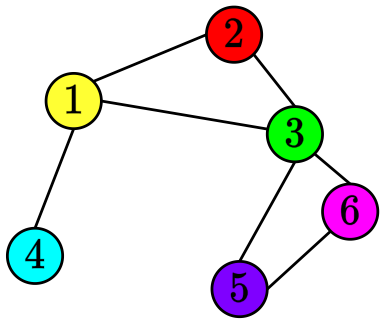


Figure 2.11: Simple graph taken as input for the GNN. The number indicates the unique index of the node. The figure is derived from the example shown in Hamilton et al. (2017)

The GNN is structured according to a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. A GNN takes as input a graph, where nodes are associated with feature vectors $\mathbf{x}_i$ and edges can also have feature vectors $\mathbf{x}_{(i,j)}$. *Hidden representations* or *embeddings* will be denoted with $\mathbf{h}_i$ for hidden node features and $\mathbf{h}_{(i,j)}$ for hidden edge features. The initial hidden node representation is set to the node feature vector, i.e. $\mathbf{h}_i = \mathbf{x}_i$. For simplicity reasons the graph is considered to be undirected and only have node features (edge features will be neglected for now). For the graph (with its nodes and node features) node embeddings are generated, $\mathbf{h}_i, \forall i \in \mathcal{V}$.

For each message passing iteration in a GNN, the node embedding $\mathbf{h}_i^{(k)}$ corresponding to node $i \in \mathcal{V}$ and iteration k will be *updated* according to the information *aggregated* from node i's neighbourhood $\mathcal{N}(i)$. Note that k indicates the iteration of message passing, also referred to as the 'layer' of the GNN. The amount of layers indicates which nodes can be reached, i.e. the nodes k hops away can be reached. The calculation for a node embedding can be expressed as is shown in Equation 2.29, where UPDATE() and AGGREGATE() are arbitrary differentiable functions (Hamilton, 2020). The aggregation of the neighbours is referred to as *messages* (Gilmer et al., 2017) and therefore this operation is shortened to the notation $\mathbf{m}_{\mathcal{N}(i)}^{(k)}$

$$\mathbf{h}_i^{(k+1)} = \text{UPDATE}\left(\mathbf{h}_i^{(k)}, \text{AGGREGATE}\left(\left\{\mathbf{h}_j^{(k)}, \forall j \in \mathcal{N}(i)\right\}\right)\right) \tag{2.29}$$

$$= \text{UPDATE}\left(\mathbf{h}_i^{(k)}, \mathbf{m}_{\mathcal{N}(i)}^{(k)}\right) \tag{2.30}$$

At each iteration k of the GNN, the AGGREGATE() function takes as input the set of node embeddings of the nodes in i's neighborhood $\mathcal{N}(i)$ and generates and the message for node i. The message is then combined with the previous embedding for node i, $\mathbf{h}_i^{(k)}$ to obtain the updated embedding $\mathbf{h}_i^{(k+1)}$. The calculation for the node embedding of node 1 with a one layer GNN can be visualized by 'unfolding' Equation 2.29 for that specific node. This is shown in Figure 2.12.

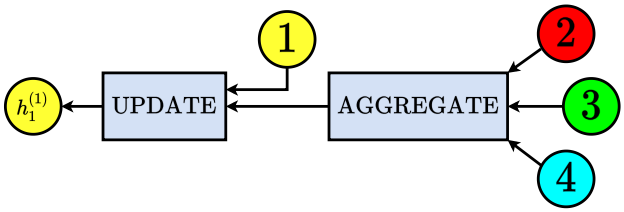


Figure 2.12: Visualization of how node 1 aggregates from its neighbors and updates to obtain a node embedding, $\mathbf{h}_1^{(1)}$. The GNN aggregates the information from node 1's local graph neighbors (node 2, 3, 4). The information from the neighbors is based on information aggregated from their respective neighborhoods in case of a multi-layered GNN. Note that the computation graph of the GNN has one layer in this case. The numbers in the nodes refer to the initial node features i.e. $i = \mathbf{h}_i^{(0)} = \mathbf{x}_i, \forall i \in \mathcal{V}$, while $\mathbf{h}_i^{(k)}$ represents the node i embedding for k-hop neighbors away.

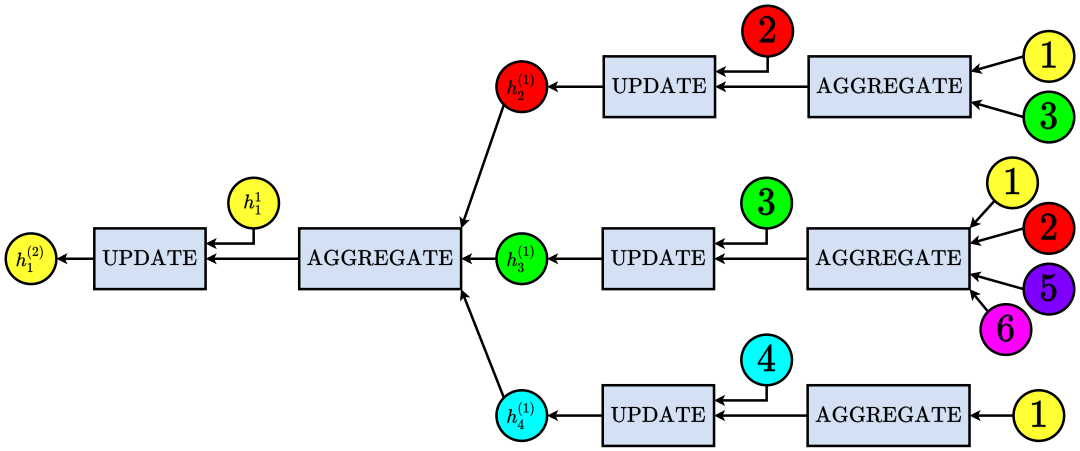


Figure 2.13: Computation graph of the GNN with two 'layers' for the update of node 1.

The structure of the graph determines the message passing updates. During each update the hidden embedding corresponding to each node $i \in \mathcal{V}$, is updated according to information aggregated from $i's$ graph neighborhood $\mathcal{N}(i)$. The final node embedding after running K amount of iterations is often denoted as $\mathbf{z}_i = \mathbf{h}_i^{(K)}, \forall i \in \mathcal{V}$. In Figure 2.13 a two layer message passing network is shown.

2.6.2 General formulation graph neural network

The GNN architectures differ in the way the neighboring node information is aggregated and how the node features are updated. In the MPNN framework (Gilmer et al., 2017) the AGGREGATE() operator is a summation preceded by a learned differentiable function (e.g. NN). The UPDATE() function is also a learned differentiable function. In Equation 2.31 and Equation 2.32 the aggregation function and update function are denoted respectively according to the MPNN framework (Gilmer et al., 2017).

$$\mathbf{m}_{\mathcal{N}(i)}^{(k)} = \sum_{j \in \mathcal{N}(i)} \text{MLP}\left(\mathbf{h}_i^{(k)}, \mathbf{h}_j^{(k)}\right) \tag{2.31}$$

$$\mathbf{h}_i^{(k+1)} = \text{MLP}\left(\mathbf{h}_i^{(k)}, \mathbf{m}_{\mathcal{N}(i)}^{(k)}\right) \tag{2.32}$$

The AGGREGATE() operator and UPDATE() operator have been a key focus for researchers to introduce novel architectures. It is important that the aggregation operation is permutation invariant because the ordering of nodes is not defined. Hence, maximum and mean operators are valid aggregation operators.

The most simple neighborhood aggregation operation is a summation of the neighboring nodes. This can lead to numerical instabilities and is sensitive to the node degrees. A solution to this problem, is to normalize the aggregation operator by taking an average instead of performing a summation, see Equation 2.35.

$$\mathbf{m}_{\mathcal{N}(i)}^{(k)} = \text{AGGREGATE}\left(\left\{\mathbf{h}_j^{(k)}, \forall\, j \in \mathcal{N}(i)\right\}\right) \tag{2.33}$$

$$= \sum_{j \in \mathcal{N}(i)} \frac{\mathbf{h}_i^{(k)}}{|\mathcal{N}(i)|} \tag{2.34}$$

$$\tag{2.35}$$

This is for example applied in the GCN (Kipf and Welling, 2016), where the aggregation operation is normalized by the degree of the node at consideration and the degree of the neighboring node, see Equation 2.36.

$$\mathbf{m}_{\mathcal{N}(i)}^{(k)} = \sum_{j \in \mathcal{N}(i)} \frac{\mathbf{h}_j^{(k)}}{\sqrt{|\mathcal{N}(i)||\mathcal{N}(j)|}} \tag{2.36}$$

In addition to weighting the nodes by an average that depends on the degree of the nodes, another approach is to use *attention* as is done in the GAT architecture (Veličković et al., 2018). The attention weights define the weighted sum of the neighbors, see Equation 2.37, where $\alpha_{i,j}$ denotes the attention on neighbor $j \in \mathcal{N}(i)$ when aggregating information at node i. The discussion on the concept of attention is out of the scope of this text. For more information, the reader is referred to (Vaswani et al., 2017; Veličković et al., 2018).

$$\mathbf{m}_{\mathcal{N}(i)}^{(k)} = \sum_{j \in \mathcal{N}(i)} \alpha_{i,j} \mathbf{h}_j^{(k)} \tag{2.37}$$

So far the focus was on the message passing part of the GNN with focus on the node features, i.e. information of neighboring nodes was aggregated in order to update node features. As was indicated in Section 2.3.8, a graph can have features on three different levels, i.e. edge features, node features and global features. The layout of the graph (presence of edge/node/global features) is dependent on the problem and the output of the GNN is also problem specific. For example, when a decision is made about the interactions among entities (Hamrick et al., 2018), the output is on the edge features. Moreover, when a GNN is used to reason about physical systems (e.g. n-body systems (Battaglia et al., 2016)), the output is conditioned on the nodes. Furthermore, a GNN can predict a global feature vector for a given graph, e.g. encoding a molecule as a graph and predicting properties of a molecule (Gilmer et al., 2017). In that case the GNN is conditioned on the global features as output.

The GNN message passing approach can be generalized to leverage edge, node and global information of the graph, this approach is presented by Battaglia et al. (2018) and called Graph Network (GN). The GN framework is flexible and can be used to formulate different GNN architectures (e.g. MPNN, GCN and GAT). Moreover, the GN framework allows to leverage edge, node and global level features at each stage of message passing. The features on the three different levels can be updated according to the following equations:

Edge update:

$$\mathbf{h}_{(i,j)}^{(k+1)} = \text{UPDATE}_{\text{ edge}}\left(\mathbf{h}_{(i,j)}^{(k)}, \mathbf{h}_i^{(k)}, \mathbf{h}_j^{(k)}, \mathbf{h}_{\mathcal{G}}^{(k)}\right) \tag{2.38}$$

Node update:

$$\mathbf{m}_{\mathcal{N}(i)}^{(k+1)} = \text{AGGREGATE}_{\text{ edge}\rightarrow\text{node}}\left(\left\{\mathbf{h}_{(i,j)}^{(k+1)} \forall\, j \in \mathcal{N}(i)\right\}\right) \tag{2.39}$$

$$\mathbf{h}_i^{(k+1)} = \text{UPDATE}_{\text{ node}}\left(\mathbf{h}_i^{(k)}, \mathbf{m}_{\mathcal{N}(i)}^{(k+1)}, \mathbf{h}_{\mathcal{G}}^{(k)}\right) \tag{2.40}$$

Global update:

$$\mathbf{m}_{\mathcal{V}}^{(k+1)} = \text{AGGREGATE}_{\text{ node}\rightarrow\text{global}}\left(\left\{\mathbf{h}_i^{(k+1)}, \forall\, i \in \mathcal{V}\right\}\right) \tag{2.41}$$

$$\mathbf{m}_{\mathcal{E}}^{(k+1)} = \text{AGGREGATE}_{\text{ edge}\rightarrow\text{global}}\left(\left\{\mathbf{h}_{(i,j)}^{(k+1)} \forall (i,j) \in \mathcal{E}\right\}\right) \tag{2.42}$$

$$\mathbf{h}_{\mathcal{G}}^{(k+1)} = \text{UPDATE}_{\text{ graph}}\left(\mathbf{h}_{\mathcal{G}}^{(k)}, \mathbf{m}_{\mathcal{V}}^{(k+1)}, \mathbf{m}_{\mathcal{E}}^{(k+1)}\right) \tag{2.43}$$

Where $\text{AGGREGATE}_{\text{ edge}\rightarrow\text{node}}$ indicates the aggregation takes information from the edges which is used in the node update. Similar for $\text{AGGREGATE}_{\text{ edge}\rightarrow\text{global}}$ and $\text{AGGREGATE}_{\text{ node}\rightarrow\text{global}}$, where respectively edge embeddings and node embeddings are aggregated in order to use this information in the global update. Multiple message passing updates can be performed consecutively by using the updated graph as input in the next iteration, i.e. setting $\mathbf{h}_{(i,j)}^{(k)} \leftarrow \mathbf{h}_{(i,j)}^{(k+1)}$, $\mathbf{h}_i^{(k)} \leftarrow \mathbf{h}_i^{(k+1)}$ and $\mathbf{h}_{\mathcal{G}}^{(k)} \leftarrow \mathbf{h}_{\mathcal{G}}^{(k+1)}$.

First, the edge embeddings $\mathbf{h}_{(i,j)}$ are updated based on the the edge feature itself, incident node features and the global feature, see Equation 2.38. Next, the node embeddings $\mathbf{h}_{(i)}$ are updated using the corresponding node feature, by aggregating the edge embeddings for all their incident edges and the global level feature, see Equation 2.39 and 2.40. The global embedding $\mathbf{h}_{\mathcal{G}}$ is updated using the global level feature and the aggregation of both updated node embeddings and update edge embeddings, see Equation 2.41, 2.42 and 2.43.

All individual aggregation operations can be any permutation invariant operation and the update functions are learned differentiable functions. In the rest of this work, relatively small graphs are considered. Hence, there is no danger of instability and therefore a summation is used as aggregation operation. Furthermore, the update functions are typically small MLPs ($\pm$3 layers). In the rest of this work, the different update functions will be denoted as f_{edge}, f_{node}, f_{global}. Using this notation for the update functions, the summation as aggregation and a prime superscript $'$ to indicate the next GNN layer $(k+1)$, Equations 2.38-2.43 can be denoted as follows:

Edge update:

$$\mathbf{h}'_{(i,j)} = f_{\text{edge}}\left(\mathbf{h}_{(i,j)}, \mathbf{h}_i, \mathbf{h}_j, \mathbf{h}_{\mathcal{G}}\right) \tag{2.44}$$

Node update:

$$\mathbf{h}'_i = f_{\text{node}}\left(\mathbf{h}_i, \sum_{j \in \mathcal{N}_i} \mathbf{h}'_{(i,j)}, \mathbf{h}_{\mathcal{G}}\right) \tag{2.45}$$

Global update:

$$\mathbf{h}'_{\mathcal{G}} = f_{\text{global}}\left(\mathbf{h}_{\mathcal{G}}, \sum_{i \in \mathcal{V}} \mathbf{h}'_i, \sum_{(i,j) \in \mathcal{E}} \mathbf{h}'_{(i,j)}\right) \tag{2.46}$$

The GN framework allows to generate hidden embeddings for each edge in the graph $\mathbf{h}_{(i,j)}$, as well as an embedding for the entire graph $\mathbf{h}_{\mathcal{G}}$. For example, considering a n-body system, the hidden edge embedding $\mathbf{h}_{(i,j)}$ represents the interaction between two bodies (i.e. nodes). Another example, the global embedding $\mathbf{h}_{\mathcal{G}}$ can be the encoding of a property of a molecule (i.e. graph). Equations 2.38-2.43 show all possible update functions and aggregation functions with all possible arguments. Dependent on the GNN architecture, arguments are left out or update/aggregation functions are skipped, e.g. if the GNN is focused on node updates, the global update is left out.

As a companion to the unifying GN framework (Battaglia et al., 2018), an open-source software library for building GNNs was released, called 'Graph Nets'. The Graph Network (GN) library is an open-source Python library, based on TensorFlow and Sonnet (DeepMind, 2018). The GN library unifies GNN designs. The main component for computation in the framework is the *GN-block*, which operates as is shown in Equation 2.38-2.43. The GN-block allows for adjustments within the block structure in order to operate like a MPNN or GCN for example. The GN-block updates the graph features on three different levels, i.e. it updates the edge features, node features and global features. GN-blocks can be easily concatenated due to their graph-to-graph feed forward process. The input of the GN-block is a graph and the output is also a graph. This allows for deep architectures with different kinds of layers. For more detail on the Graph Nets framework the reader is referred to Appendix B or Battaglia et al. (2018).

For more information on GNN models and a more theoretical background of the GNN model the reader is referred to Hamilton (2020) and Bronstein et al. (2021).

2.6.3 Contrastive loss

A widely used method for learning multi-relational node embedding is *contrastive learning* (Hamilton, 2020). This approach is extensively used in the field of natural language processing (Mikolov et al., 2013). Contrastive learning is used to learn general features of a dataset without the use of labels but by teaching a model which input samples are similar or dissimilar. This is why it gained a lot of interest in the field of state representation learning. The contrastive loss is used in different areas of ML and therefore has different names such as *margin loss*, *hinge loss*, *margin loss*, *ranking loss*.

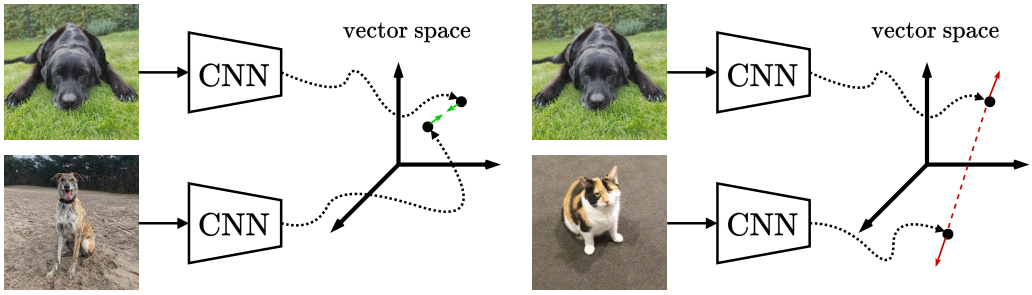
The objective of contrastive losses is to predict the similarity or dissimilarity between inputs by means of some distance metric. The general idea is to construct pairs of input samples that are related (positive pairs) and pairs of unrelated input samples (negative pairs). A loss function is formulated that scores the positive pairs different than the negative pairs. The objective is to learn representations with a small distance d in the embedding space for positive pairs and a

distance greater than some margin γ for negative pairs. The loss function is shown in Equation 2.47.

$$\mathcal{L} = d\left(z, z_{\text{positive}}\right) + \max\left(0, \gamma - d\left(z, z_{\text{negative}}\right)\right) \tag{2.47}$$

Here z, z_{positive} and z_{negative} are respectively the embedding of the input sample, positive sample and negative sample. The distance d is a specified distance measure, e.g. Euclidean distance. The loss will be low if positive pairs are encoded to similar representations and negative pairs are encoded to different representations.

For the positive pairs, the loss will be 0 when representations for the input sample and positive sample coincide in the embeddings space, i.e. the two representations have no distance between them. Therefore, the loss will increase with the distance between them. For the negative pairs, the loss will be 0 when the distance between the input sample and negative sample is greater than some margin γ. When the distance is smaller than the margin, the loss is larger than 0. The network parameters will be updated to produce more distant embeddings. The purpose of the margin is to ensure that when negative pairs are distant enough, no effort is put into enlarging the distance between negative pairs. Therefore, additional training can focus on more difficult pairs.



(a) Positive pair. **(b)** Negative pair.

Figure 2.14: Example of contrastive loss setup to train a network to classify cats and dogs. Note that the CNNs share parameters.

2.7 Summary

In this chapter, the relevant background information is presented needed to understand the rest of the report. First, the MDP was introduced in Section 2.1.1, this mathematical framework is used to formally describe the environment in reinforcement learning. Next, in Section 2.1, the reinforcement learning framework was formulated together with the jargon and main concept used in the field. Furthermore, relevant reinforcement learning methods were discussed, e.g. value iteration and Q-learning. Moreover, the model-based reinforcement learning framework is presented.

In the subsequent Section 2.2, state representation learning was discussed. The motivation to learn expressive representations was given here. By learning to predict the next state in latent space, a state representation can be learned. Two other methods to learn a state representation were mentioned, i.e. robotics priors and dimensionality reduction by means of autoencoders.

Next, in Section 2.3 the graph data structure was introduced. Followed by Section 2.4 and Section 2.5, which introduced the concept of NNs and CNNs respectively. Finally, in Section 2.6, a new type of NN, the GNN, was presented that works on a graph data structure. This new approach allows for modeling the input data in a different manner, i.e. as a graph opposed to a vector representation. The emerging new field of GNNs served as the research direction of this thesis in the field of state representation learning.

3 Related work

In this chapter, the related work regarding unsupervised/self-supervised state representation learning methods is discussed. More specifically, the focus is on the related work regarding the learning of transition models, and in particular using a GNN.

The remainder of this chapter is organized as follows. In Section 3.1 the three different strategies in state representation learning are briefly discussed. Next, in Section 3.2, the focus is on the related work on learning transition models. In Section 3.3, a recent success where a GNN and reinforcement learning are combined is discussed. At last this chapter is briefly summarized in Section 3.4.

3.1 State representation learning

State representation learning is an important topic in the field of reinforcement learning. State representation learning aims at learning a low dimensional encoding for the state from the of raw observations. The quality of the low dimensional state representations helps to overcome the curse of dimensionality, are often better interpretable, improve performance and speed in reinforcement learning algorithms (Lesort et al., 2018). Early work on representation learning in reinforcement learning used methods like state aggregation (Singh et al., 1995), principal component analysis (Curran et al., 2015) or by using linear function approximation with complex basis functions (Mahadevan and Maggioni, 2007). Recently, deep neural networks have gained a lot of interest in representation learning. The mapping from observations to state is commonly learned using neural networks (Lesort et al., 2018), mostly by means of autoencoders. In the work by Lesort et al. (2018), three main strategies are considered in state representation learning: reconstructing the observation, using prior knowledge to constrain the state space and learning a latent model to predict the state transition.

The first approach of reconstructing the observation is often realized using autoencoders. An autoencoder is composed by an encoder and a decoder. The encoder maps the observation to a latent state space of a lower dimensionality. Whereas, the decoder reconstructs the observation from the latent state representation by mapping the latent state back to observations space. The autoencoder is trained by minimizing the loss between the observation and the reconstructed observation. After training, the encoder maps the observation to a latent state representation which is used in down stream reinforcement learning algorithms. Often Variational Autoencoders (VAEs) are used, such as in the work by Ha and Schmidhuber (2018), where a VAE is used to learn a latent state representation from 2D images. However, often a combination of the three strategies in state representation learning are used to find useful state representations. For example in Watter et al. (2015), a VAE is used in combination with additional constraints on the learned dynamics model (locally linear for state transitions). Autoencoder based methods learn to discover representations by performing reconstruction. This approach might fail to include visually small but relevant features (Kaiser et al., 2019). Furthermore, a disadvantage of reconstructing the observation, is that a decoder needs to be trained. This is requires additional computation and is often not required for the control task. Hence, in this work the aim is to learn a representation without observation reconstruction.

The second strategy for learning a state representation, is by using prior knowledge to constrain the state space. The prior knowledge about the environment is incorporated in the form of loss functions that are used to train a neural network which maps the observations to latent space (Jonschkowski and Brock, 2015; Lesort et al., 2017). Additionally, the approach of guiding the state representation by means of a loss functions is also applied in unsupervised learning. This is done using contrastive learning, which is used to learn general features of a dataset without

the use of labels. The model is forced to group similar states together and dissimilar states apart. This is for example used in learning representation for images (Chen et al., 2020) or representations in natural language processing (Mikolov et al., 2013). Moreover, it is used graph representation learning, for learning multi-relational node embeddings (Hamilton, 2020).

The third strategy, and main focus in this work, is learning a state representation by predicting the state transition in latent space. The main idea, is to force a neural network to encode the necessary information to predict the next state. In the works where autoencoders are used for observation reconstruction, a transition model is used to predict the next observations (van Hoof et al., 2016; Watter et al., 2015). Next section is focused on structured transition models.

3.2 Structured transition model

Recent works on modeling structured environments have made great progress in improving predictive accuracy by taking the structure of the problem into account (Battaglia et al., 2016; Watters et al., 2017; Kipf et al., 2018; Wang et al., 2018). Prior knowledge on the environment structure is incorporated in the state representation by using a GNN. In (Battaglia et al., 2016) the Interaction Network is proposed for explicit reasoning about objects and relations in complex systems. Its ability to learn and generalize dynamics is tested in different physical domains (e.g. n-body problems, rigid-body collision and non rigid dynamics). Moreover, in (Kipf et al., 2018) a model is presented that infers the interactions while also learning the dynamics in an interacting system. Neural networks that operate on graph structured data have been developed and explored for more than a decade (Gori et al., 2005; Scarselli et al., 2009). However, the popularity of these so called GNNs have only recently gained a lot of traction and attention in the ML field (evanzd, 2021).

Incorporation of an inductive bias in the model architecture makes the model generalize better to novel situations and makes the learning process more sample efficient (Battaglia et al., 2018). The output of a CNN is shift invariant to displacement in the input due to the use of kernels which introduces a locality inductive bias. The same shift invariance can be learned with the use of a MLP. However, the MLP can only learn this from the presented data. Hence, to learn shift invariance using a MLP a lot more data is needed opposed to using a CNN. By using a GNN, the relational structure of the problem is incorporated in the model architecture and does not have to be learned from the presented data. This allows the model to generalize better to novel scenarios and improves the sample efficiency in the learning process.

Previous work has shown that incorporating a relational inductive bias by means of a GNN is effective in modelling objects and relations (Battaglia et al., 2016) and it greatly facilitates the ability to generalize to novel scenarios (Hamrick et al., 2018). Several recent works have shown to be able to learn structured models from raw image observations (van Steenkiste et al., 2018; Xu et al., 2019; Watters et al., 2019), but these rely on pixel-based loss functions. In the work presented by Kipf et al. (2020b), it was shown that GNNs can be used to learn interpretable representations from raw image observations in an end-to-end self-supervised setting using a contrastive loss. The presented model called Contrastively-trained Structured World Model (C-SWM), can be used to predict next states in latent space accurately. The C-SWM learns a transition model and discovers the objects with state representations that can be related to objects in the scene. The learned transition model is able to accurately model effects of actions in novel environment scenarios.

The objective of state representation learning is to learn a lower dimensional state space that is sufficient to learn a policy that can be transferred to the original state space. In (van der Pol et al., 2020) a contrastive loss is introduced that enforces action equivariance on the learned representations. By learning equivariant maps, structured latent representations are learned which are then used to do value iteration. They show that for deterministic MDPs, a MDP

homomorphism can be learned that allows for learning an optimal policy in the abstract state space.

3.3 Reinforcement learning and GNN

In (Hamrick et al., 2018) it is shown that the incorporation of an inductive bias by means of graphs is necessary to solve relational structured problems effectively. The problem considered is the task of gluing together randomly placed blocks in order to have a stable tower when gravity is applied. Towers are created randomly by placing blocks on top of each other in a 2D plane. Each block is stacked on top of a previous block except for the first block. The agent or human participant is asked to glue certain blocks together in order to have a stable tower when gravity is applied. After the agent or participant is done applying the glue, gravity is applied and a reward is obtained corresponding to the stability of the tower with glued blocks. Although physical knowledge is important here, it is shown that by translating the problem into a relational reasoning problem the problem is solved effectively. The tower of blocks is encoded as a graph and a GNN is trained as part of a reinforcement learning agent that performs the same gluing task as the humans faced. Three different agents are compared, a MLP agent, a fully connected GNN agent and a sparse connected GNN agent (where there are edges according to the blocks that are in contact). The MLP performed the worse, then the human participants, then the fully connected GNN agent, followed by the sparsely connected GNN and the agent with access to a perfect simulator performed the best. However, the performance using the relational structure as opposed to the MLP agent suggests that relational knowledge is key in solving complex physical reasoning tasks and can be effective without an explicit model (i.e. fully connected graph). Furthermore, the graph structured agents generalized to problems with more blocks than seen during training. This is not possible for the MLP agent since the architecture has to change, and therefore training is needed to apply the agent effectively to novel situations. This work shows that deep reinforcement learning can be improved by incorporating an inductive bias such that it performs better and generalizes better.

3.4 Summary

In this chapter, the related work is discussed regarding state representation learning. The main focus was on learning structured environment models. These structured models show that incorporating the relational structure in the model greatly improves the predictive accuracy and generalization to novel scenarios.

The use of a contrastive loss has shown useful in different works, to aid the representation learning in a unsupervised or self-supervised setting. The C-SWM model seems to be a promising model to learn expressive state representations that can be used for planning in latent space.

The C-SWM architecture will be used to learn state representations and a model of the environment. The learned environment model is then used to learn a policy in latent space. Whether the learned environment model is sufficiently accurate with the original MDP, will determine whether the learned policy can be applied in the original MDP. Having a latent model of the environment allows for planning in latent space. This is beneficial in the context of robot navigation because it allows to plan the consequence of the actions in advance.

4 Method

In this chapter, the methodology used to answer the research questions is presented, see Section 1.3 for the research questions. The methodology is divided into two parts, according to the sub-questions. In the first part, the focus is on learning the underlying dynamics of the MDP, i.e. the transition function and reward function. The state of the environment is accessible to the agent in order to investigate the applicability of a GNN as a transition and reward model in comparison to a NN. The second part will extend this by learning the underlying dynamics of the MDP in a more challenging setup, where raw observations are used. The raw observations are mapped to a latent space in which the dynamics of the MDP are learned by predicting the next state. Hence, a transition model is trained to learn a state representation. The application at hand is a robot navigation task in a deterministic 2D grid world. In the presented methodology, the Markov property is assumed, i.e. the next observation and reward are only dependent on the current observation and the action the agent takes.

The outline of this chapter is as follows. First, in Section 4.1, the focus is on learning the underlying dynamics of the MDP. Secondly, in Section 4.2, it is explained how the learned environment models are used to generalize to novel environment scenarios for which a policy is learned. The approach presented in Section 4.1 and Section 4.2, is used to answer RQ1. Thereafter, the approach is extended to a setting where a state representation is learned from raw observations, while simultaneously learning an environment model in the learned latent space. In Section 4.3, the pipeline is presented that is used to learn a state representation from raw observations and train a model of the environment in latent space. Followed by Section 4.4, where it is discussed how the learned environment model is used to plan in latent space in order to learn a policy. The method discussed in Section 4.3 and Section 4.4, is used to answer RQ2. Finally, this chapter is summarized in Section 4.5.

4.1 Learn a model of the environment

In this work, the aim is to apply representation learning using a GNN effectively in a reinforcement learning framework. The goal is to learn useful state representations from raw pixel observations. Many state representation learning methods use an autoencoder architecture to learn a state representation, where the loss is put in pixel space. The autoencoder approach has the disadvantage of training a decoder. Therefore, the focus here is to directly learn a state representation in latent space such that there is no decoder needed. The state representation will be learned by training a transition model in latent space. By mapping the observation to a latent space and predicting the next state in latent space, the error can be computed with next observation that is mapped to the same latent space. This concept is shown in Figure 2.4.

By mapping the observation and next observation to a latent space and applying a loss function in this latent space, both a state representation and a transition model are learned. Before the full problem of mapping the raw observation to latent space and learning an environment model in this latent space is considered, the focus is on a simplified setup. The mapping of the observation to latent space is skipped. An environment model is trained with direct access to the environment state. This is the focus of RQ1, where the goal is to compare an environment model (transition and reward function) learned using GNN against an environment model learned using a NN. The simplified setup allows for analyzing the problem of modeling the transition and reward model. The point of focus is to investigate whether it is advantageous to use a GNN for modeling the transition model and reward model versus a conventional NN.

A NN will be trained to approximate the transition function $\Delta\hat{s}_t = T(s_t, a_t; \theta_{\text{NN}})$ and also the reward function $\hat{r}_{t+1} = R(s_t, a_t; \theta_{\text{NN}})$. The models take a vector representation of the state s_t and

a one-hot encoded action a_t as input. The conventional NN does not infer any prior knowledge on the problem. On the other hand, a GNN incorporates a relational inductive bias. In many problems there are underlying dynamics which are not obvious to discover. However, by assuming this relational structure in the model architecture, by means of a GNN, it can enhance the generalization performance of a transition model and, therefore, be advantageous for planning in unseen environment scenarios. Hence, also a transition model $T(s,a;\theta_{\text{GNN}})$ and reward model $R(s,a;\theta_{\text{GNN}})$ will be trained using a GNN. The models take as input a graph representation of the state and the one-hot encoded action.

The goal is to see to what extent a GNN is able to model the dynamics of the MDP and how it compares to a conventional NN architecture. The aim is to learn an environment model that can generalize to novel environment situations. This allows for planning in new environment scenarios. The policy obtained using planning with a learned environment model can only be as good as the learned environment model. Therefore, it is important to learn an accurate environment model.

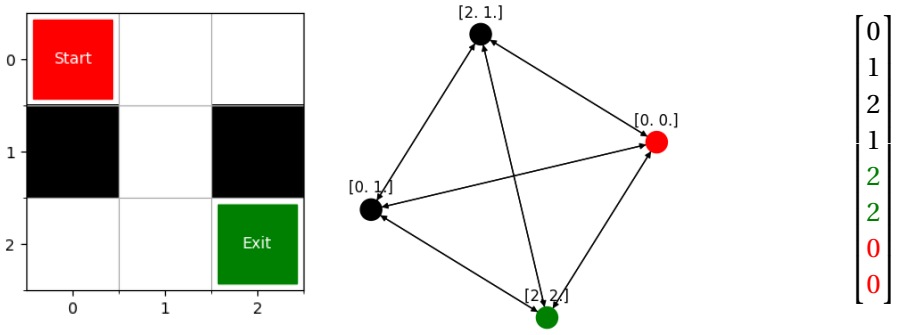
Learning the dynamics of the environment is a self-supervised learning problem, i.e. the target samples are generated by interaction of the agent with the environment. The transition model $T(s,a;\theta)$ is trained using a MSE loss between starting state s_t plus the prediction of the model $\Delta\hat{s}_t$ and the true next state s_{t+1}. The reward model $R(s,a;\theta)$ is also trained using a MSE loss, the error is computed between the predicted reward $\hat{r}_{t+1}$ and the true reward r_{t+1}. The used loss functions for the transition model and reward model are shown in respectively Equation 4.1 and Equation 4.2, N is the number of samples in the batch.

$$\mathcal{L}_{\text{transition}} = \frac{1}{N}\sum_{i=1}^{N}\left((s_t + T(s,a;\theta)) - s_{t+1}\right)^2 \tag{4.1}$$

$$\mathcal{L}_{\text{reward}} = \frac{1}{N}\sum_{i=1}^{N}\left((R(s,a;\theta)) - r_{t+1}\right)^2 \tag{4.2}$$

In theory, perfect models allow for planning optimal policies. By learning a transition model and reward model in essence a model is learned of the MDP. When the loss functions of the transition model and reward model converge to zero, the learned MDP model is homomorphic with the original MDP of the environment. However, in practice, a zero loss is almost never the case. Therefore, the learned model will be an approximation of the original MDP. This does not have to be a problem, as long as the learned model is sufficiently accurate. Sufficiently accurate means that the learned model can still be used to do planning in order to find a policy that is optimal. By learning a policy using the learned environment model, it is verified whether the learned model is sufficiently accurate.

The problem at hand is the simplest form of a navigation problem: 'navigation in a 2D grid world'. A 3x3 maze is shown in Figure 4.1a. Note that the term 'grid world' and 'maze' are used interchangeably. In Figure 4.1a, the red square is the location of the agent, the green square is the goal and the black squares are obstacles. The maze lay-out is manually encoded in a graph representation and a vector representation, see Figure 4.1b and Figure 4.1c. These encoded representations can be used as a low dimensional input state for the transition and reward model implemented using a GNN and NN. The representation consists of the x,y-coordinates of each object, i.e obstacle positions, target position, agent position. For the graph representation, a fully connected undirected graph is used where there is a node per object and the node features are the object positions, see Figure 4.1c. This encoding is depicted because it is similar to the representation that is learned in Kipf et al. (2020b) in an unsupervised setting. The vector representation is a one-dimensional vector with all object positions concatenated, see Figure 4.1c.



(a) Maze observation **(b)** Graph state representation **(c)** Vector state representation

Figure 4.1: The different state representations, where black represents obstacles, green the target and red the position of the agent. a) is the observation of the maze, b) is the graph state representation, c) is the vector state representation.

For the architecture of the GNN transition model $\Delta\hat{s}_t = T(s_t, a_t; \theta_{\text{GNN}})$, the architecture presented by Kipf et al. (2020b) is used. The transition model takes as input the state represented as a fully connected graph, with the position of the object as node feature $\mathbf{s}_t$ and one hot encoded action $\mathbf{a}_t$ per object. The intermediate interaction $\mathbf{h}_t^{(i,j)}$ between nodes i and j is calculated using an edge update function, see Equation 4.3. The node features, actions and intermediate representation of the interaction between node i and node j are concatenated and passed to a node update function to approximate the state transition, see equation 4.4. Equation 4.3 and Equation 4.4, describe the transition model implemented using a GNN. For the update functions f_{edge} and f_{node}, MLPs are used, of which the architecture is shown in Figure 4.2. Note that a lower number of hidden units is used compared to the number (512 units) presented in the work by Kipf et al. (2020b). The problem setting here is simplified and additional units did not show a significant improvement in accuracy and generalization performance of the model. Hidden layers with more than 128 units resulted in a longer training time, therefore 128 hidden units are used. Furthermore, a fully connected graph is used as a state representation, because it facilitates the learning of relevant interactions between all the nodes in the graph. When a certain interaction between nodes is not important for the problem, it will simply not use that interaction for inference. Similar to the case where irrelevant features in a feature vector will not contribute to the inference in a conventional NN.

Edge update:

$$\mathbf{h}_t^{(i,j)} = f_{\text{edge}}\left(\left[\mathbf{s}_t^i, \mathbf{s}_t^j\right]\right) \tag{4.3}$$

Node update:

$$\Delta\mathbf{s}_t^i = f_{\text{node}}\left(\left[\mathbf{s}_t^i, \mathbf{a}_t^i, \sum_{j \in \mathcal{N}_i} \mathbf{h}_t^{(i,j)}\right]\right) \tag{4.4}$$

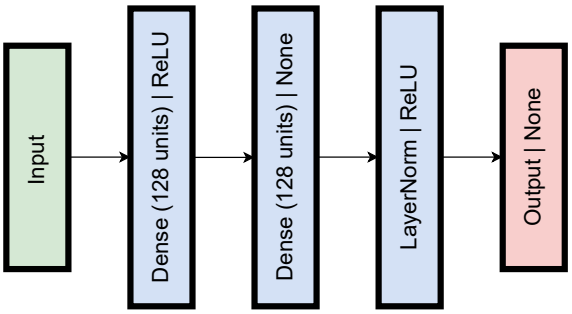


Figure 4.2: The MLP architecture used for f_{edge}, f_{node} and f_{global} in the transition and reward model implemented using a GNN. It contains 2 hidden layers with 128 units and a normalization layer after which the units are activated using a ReLU activation. At last, there is the output layer which is a linear layer. The input dimension for f_{edge} is 2 times the node feature dimension. The output dimension is chosen to be equal to the number of hidden units, i.e. 128. For f_{node} the input dimension is the node feature dimension plus action dimension (4) plus the output dimension of f_{edge}. The output dimension of f_{node} is equal to the dimension of the node feature. The input dimension of f_{global} is the node feature dimension plus the output dimension of f_{edge}. The output dimension of f_{global} is a scalar value corresponding to the reward. The input dimension of f_{global} is the node feature dimension plus output dimension of f_{edge}. The output layer is initialized with zeros. The other layers are initialized using a truncated normal distribution centered around 0 with a standard deviation of $\sqrt{2/\text{number of hidden input units}}$. A bias is used in all layers.

For the reward model the same edge update function and node update function are used, see Equation 4.3 and Equation 4.4. For the reward model, an additional global update function is added in order to map the state and action to a scalar, i.e. the reward r_{t+1}. The additional global update function for the reward model is shown in Equation 4.5. The same MLPs is used for f_{global} as was used for the edge update function f_{edge} (Equation 4.3) and node update function f_{node} (Equation 4.4). The MLP architecture used for f_{global} is shown in Figure 4.2.

Global update:

$$\mathbf{r}_{t+1} = f_{\text{global}}\left(\left[\sum_{i\in\mathcal{V}} \mathbf{h}_t^i, \sum_{(i,j)\in\mathcal{E}} \mathbf{h}_t^{(i,j)}\right]\right) \tag{4.5}$$

In order to make a fair comparison, the NN architecture is of a similar complexity, i.e. the same amount of hidden layers is used as in the GNN architecture. The architecture of the transition model implemented using a NN is shown in Figure 4.3 and the same architecture is used for the reward model except that a scalar value is predicted instead of a state vector. The initialization is the same as for the MLPs used in the GNN architecture.

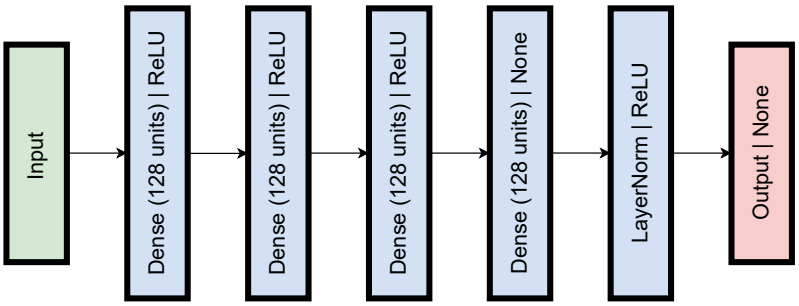


Figure 4.3: Transition model $T(s_t, a_t; \theta_{\text{NN}})$ and reward model $R(s_t, a_t; \theta_{\text{NN}})$ implemented using a NN. It contains 4 hidden layers with 128 units and a normalization layer after which the units are activated using a ReLU activation. At last, there is the output layer which is a linear layer. The input dimension is the summation of the state dimension plus the one-hot encoded action vector. The output dimension for the transition model is equal to the state dimension. The output dimension of the reward model is a scalar, i.e. the dimension is 1. The output layer is initialized with zeros. The other layers are initialized using a truncated normal distribution centered around 0 with a standard deviation of $\sqrt{2/\text{number of hidden input units}}$. A bias is used in all layers.

The pipeline of learning an environment model consists of two steps. First, an agent interacts with the environment following a random policy. While interacting with the environment the agent collects experience samples, $(s_t, a_t, r_{t+1}, s_{t+1}, \text{status})$. Secondly, these experience samples are used to learn both a transition model $T(s_t, a_t; \theta)$ and reward model $R(s_t, a_t; \theta)$. For both models, a NN and GNN implementation is trained. The process of learning the dynamics of the environment is summarized in Figure 4.4.

In Section 5.2.1, the experiment is described to learn an environment model in a 2D deterministic grid world. The learned environment model is then used in the next step, which is to learn a policy.

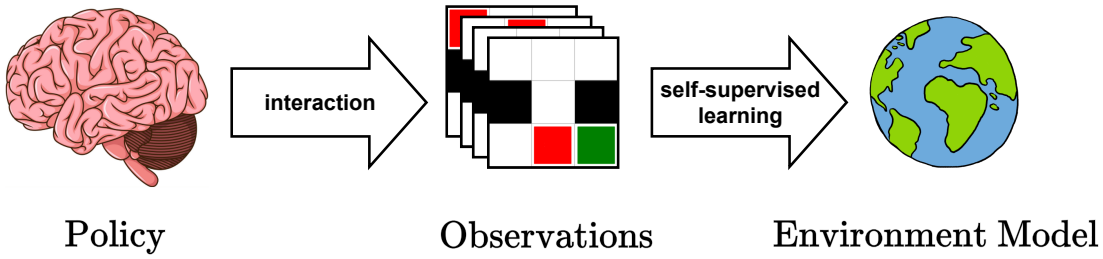


Figure 4.4: Pipeline for learning the dynamics of the environment. An agent interacts with the environment by following a random policy. The resulting observations, rewards and actions are stored in experience samples. The experience samples are used to train a transition and reward model in an offline setting.

4.2 Generalization of the learned environment model

The learned environment model ($\hat{T}(s, a; \theta)$ and $\hat{R}(s, a; \theta)$) is an approximate MDP homomorphism with the original MDP. When the learned environment model is sufficiently accurate, the environment model can be used to learn the optimal policy. The learned environment model allows the agent to plan ahead from any state in any novel environment situation. In order to verify whether the learned environment model is accurate and generalizes to novel situations, a separate set of experience samples is used to test the performance. This separate set of experience samples is referred to as 'test set'. The test set is used during training to verify the performance of the environment model on a different set of experience samples. After each training update, the environment model that is being trained is used to predict the next states

$\hat{s}_{t+1}$ and rewards $\hat{r}_{t+1}$ of the test set which are compared to the true next states s_{t+1} and rewards r_{t+1} respectively.

After learning a model of the environment, planning can be done. Note that this is the model-based reinforcement learning approach, shown in Figure 2.3. In this framework a policy is obtained by acting (generating experience), after which a model is learned that is used to do planning. Here, the loop of 'acting → model learning → planning', is cycled only once. In order to validate whether the learned environment model is sufficiently accurate and generalizes to new environment situations, a DQN agent is trained. For both the GNN model and NN model of the environment, a DQN agent is trained by solely using the learned model. This iterative process is shown in Figure 4.5. Note the similarity with the framework that is shown in Figure 2.1, where the environment is now simulated using the learned environment model (i.e. the transition model $\hat{T}(s,a;\theta)$ and reward model $\hat{R}(s,a;\theta)$). Although the used environment in the experiments is discrete, the environment models are not assumed to be discrete. Therefore, DQN agents are trained to obtain the value function. The learned value functions are then compared to the true value function which can either be obtained by dynamic programming or training a DQN agent that has access to the environment.

In the experiment presented in Section 5.2.2, the learned environment models are used to learn a value function. The obtained value functions are compared with respect to the true value function. The experiment in Section 5.2.1 is aimed at learning an accurate environment model that generalizes to new environment situation, for both a GNN and NN implementation of the environment model. The learned environment models are used in the experiment presented in 5.2.2 to see whether they are sufficient to learn the optimal policy. The experiments presented in Section 5.2.1 and Section 5.2.2 are used to answer RQ1.

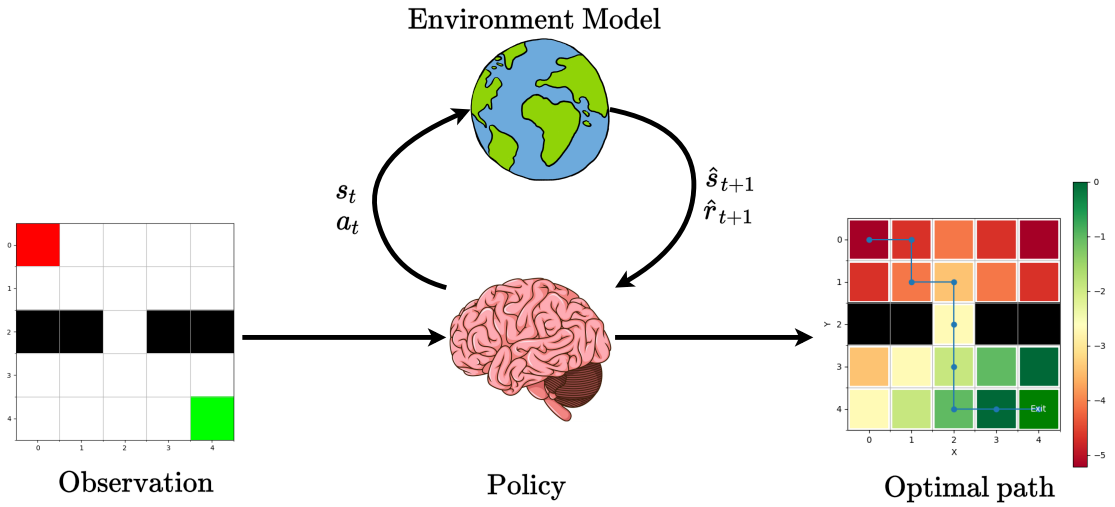


Figure 4.5: The learned environment model can be used by the agent to do planning. The agent is presented with a new environment state, i.e. an observation. Starting in this state, the agent can plan the optimal trajectory to the goal state. If the learned environment model is accurate enough, the learned policy can be lifted to the original MDP.

4.3 Learn a model of environment in latent space

As was mentioned in Chapter 3, many approaches in state representation learning focus on learning a representation by reconstruction of the state into the original state space. This has the disadvantage that it is needed to train a decoder. Furthermore, decision relevant but visual small features are often ignored by putting the loss in pixel space. Therefore, it would be beneficial to skip the decoding step and learn a meaningful state representation that can be used to learn the environment dynamics without the need of going back to the original state space.

This can be achieved, by mapping the observation to a latent space and predicting the next state in this latent space using a transition model. An error is computed between the predicted next state and the next observation which is mapped to the same latent space. The error is used to train the transition model. This way, both a transition model and a state representation are learned. This concept is shown in Figure 2.4. The latent state representation can be used to train a reward model. The predicted reward can be directly compared to the true reward because it is only a scalar value.

By training a transition model in latent space, constraints can be imposed on the model of the environment. The additive form of the transition model $\hat{z}_{t+1} = z_t + \Delta\hat{z}_t = z_t + T(z_t, a_t; \theta)$, imposes a constraint for modeling the effects of actions in the environment as translations in the latent space. To prevent that all states are mapped to the same point a contrastive loss can be used. By using a contrastive loss the distance between a latent next state and a set of random other states is maximized. A similar approach has been used in the work of van der Pol et al. (2020) and Kipf et al. (2020b).

A transition and reward model will be learned using both a GNN and NN, but now in an unsupervised setting and training directly on the raw observations. The pipeline to learn the environment model (transition and reward model) is shown in Figure 4.6. The transition loss function is shown in Equation 4.6. The γ is set to 1. Where z_t^k indicates the k-th object representation. Note that the loss will be averaged over the batch size, this is not indicated in the loss function below to keep the expression simple. The subscript 'pos' and 'neg' refer to the positive sample or negative sample. The reward model is trained using a MSE loss, see Equation 4.7. The same architecture for the GNN environment model and NN environment model is used as was presented in Section 4.1. However, the latent state is learned. The number of nodes in the graph is a hyperparameter and set to the number of objects in the scene. The output dimension of the object encoder is set to 2 for each object. Hence, 2 objects in the environment will give a graph with 2 nodes with feature vectors of size 2. The vector state representation is a concatenation of the output of the object encoder, i.e. 2 objects in the scene will result a

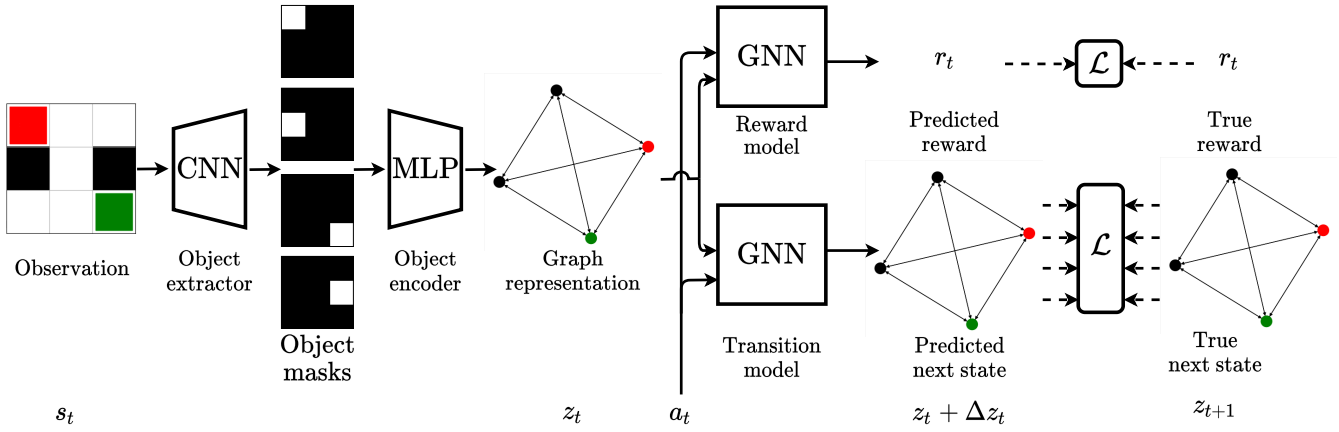
$$\begin{aligned} \mathcal{L}_{\text{transition}} &= \mathcal{L}_{\text{pos}} + \max\left(0, \gamma - \mathcal{L}_{\text{neg}}\right) \\ \mathcal{L}_{\text{pos}} &= \frac{1}{K}\sum_{i=1}^{K}\left(\left(z_t^k + T^k(z_t, a_t; \theta)\right) - z_{t+1}^k\right)^2 \\ \mathcal{L}_{\text{neg}} &= \frac{1}{K}\sum_{i=1}^{K}\left(\tilde{z}_t^k - z_{t+1}^k\right)^2 \end{aligned} \tag{4.6}$$

$$\mathcal{L}_{\text{reward}} = \frac{1}{N}\sum_{i=1}^{N}\left(\left(R(z_t, a; \theta)\right) - r_{t+1}\right)^2 \tag{4.7}$$

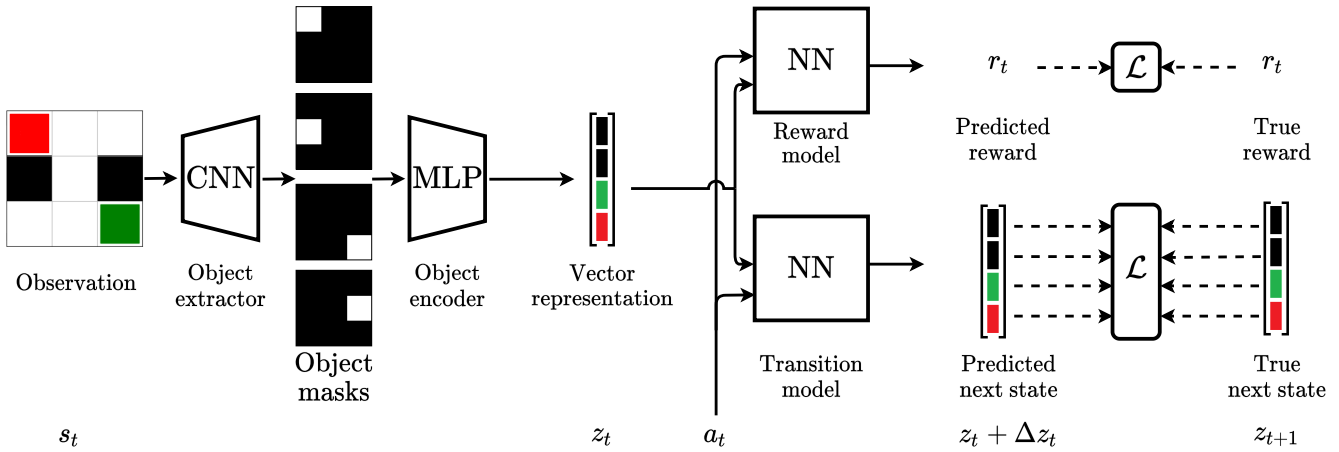
The raw observation will be passed through a CNN and encoded using a MLP into either a graph representation or vector representation, similar to the representation shown in Figure 4.1. This representation is then used to train a transition model which predicts the state transition in latent space Δz_t. A reward model is trained simultaneously which predicts the next reward using the latent state representation. The predicted transition plus the encoded state $z_t + \Delta\hat{z}_t = \hat{z}_{t+1}$ are then contrasted with negative samples of encoded next states z_{t+1}. The error is used to train the CNN object extractor, MLP object encoder and transition model all at the same time. The additive form of the transition model provides an inductive bias for modeling the effects of actions in the environment as translations in the latent state space. This way the GNN and NN model are guided to learn meaningful state representations in the latent space that are interpretable.

In Section 5.2.3, the experiment is described that trains the environment models in latent space. 2D plots of the latent state space can be made to give inside into the learned state repre-

sentation. The underlying problem is discrete and lies on a grid. Ideally, a state representation is learned per object that adheres to the grid structure of the environment. The learned environment models are used in the next step, which is to plan the optimal policy in latent space.



(a)



(b)

Figure 4.6: The pipeline of training a reward and transition model in latent space. The transition model and reward model are implemented using a) GNN and b) NN.

4.4 Planning in latent space

After a transition model and reward model are learned in latent space, planning can be done in latent space using these models. The same approach can be applied as was presented in Section 4.2. The learned reward model and transition model are used to plan the optimal policy in latent space. The learned policy is applied in the original state space.

In Section 5.2.4, the experiment is described that is performed to plan the optimal policy in latent space. The learned environment models, trained in the experiment presented in 5.2.3, are used to plan in latent space. The experiments presented in Section 5.2.3 and Section 5.2.4 are used to answer RQ2.

4.5 Summary

In this chapter, the methodology used to answer the research questions is presented. The research questions are stated in Section 1.3. The methodology is split into two steps in order to answer the two sub-questions.

First, a simplified setup is considered where the focus is on learning an environment model from manually encoded representations. The observations are manually encoded in a graph representation for the GNN model and vector representation for the NN model. This is done to focus on the training of an environment model without the need for the additional complexity of an encoding step. In Section, 4.1 the method is presented to learn an environment model. Here, the procedure for gathering the experience samples and training of the environment model is described. A GNN model and NN model are trained to approximate the transition function and reward function of the environment. Furthermore, The used architectures for the reward models and transition models are presented. In Section 4.2, it is explained how the learned environment models are used to generalize to novel environment scenarios. By training a DQN agent using the learned environment model, a value function is obtained. For both environment models (GNN and NN implementation) a DQN agent is trained of which the value function is compared to the true value function.

Secondly, in Section 4.3 the simplified setup is extended to a setup where a state representation is learned from raw observations. By training a transition model in latent space, a state representation is learned from raw observations. A CNN object extractor and MLP object encoder are used to map the raw observations to a latent space. The state representation is then used to learn a transition and reward model in latent space. A transition model and reward model are learned for both a GNN and NN implementation. The loss is put in latent space, which results in both training an encoder (that maps the raw observations to a latent space) and an environment model (that is used to plan in latent space). The complete pipeline is visualized in Figure 4.6. The learned environment models are used to do planning in latent space. The optimal policy learned in latent space is then lifted to the original space, this is described in Section 4.4.

5 Experiments

In this chapter, the performed experiments are elaborated. First, in Section 5.1, the used software and the used environments are discussed. Next in Section 5.2, the performed experiments are explained which are used to assert the effectiveness of using a GNN environment model as opposed to a NN environment model in a model-based reinforcement learning setting. In Section 5.2.1, the training of the environment models is elaborated. Followed by Section 5.2.2, where the details are presented of the conducted experiments to validate the generalization of the learned models to new maze scenarios. Next, in Section 5.2.3 the same experiments are explained for the setup where a state representation is learned from raw observations.

5.1 Experimental setup

5.1.1 Software

All the code is written in Python 3.7. Python is one of the most popular programming languages today (PYPL, 2021). Python is widely used in the ML field, the online community is large and there a lot of ML libraries. To construct NN architectures the TensorFlow library (Abadi et al., 2016) is used together with CUDA and CuDNN. To be specific, TensorFlow 1.15 is used. TensorFlow is a popular framework for constructing NN architectures. Besides TensorFlow, a popular tensor based library used for constructing NN architectures is PyTorch (Paszke et al., 2019). The used IDE is PyCharm. In Appendix C it is explained how to set up TensorFlow with CUDA and CuDNN in PyCharm. TensorFlow 1.15 is used for constructing the NN architectures. At the time of writing, the only available library to create GNN architectures in TensorFlow, is the GraphNets library. Therefore, the GNN architectures are created using the GraphNets library by DeepMind (Battaglia et al., 2018). In Appendix D other GNN Python libraries are surveyed. The GraphNets library offers flexibility, i.e. almost any existing GNN can be implemented using the 6 core functions. More details on the GraphNets library are described in Appendix B.

5.1.2 Environments

In this work, two different environments are used. Besides the layout differences, the main difference is the observation space of the agent. In the first experiments, the focus is on learning the environment model. Therefore, a low dimensional state representation is manually encoded, the agent observes all the necessary information to solve the task. For the experiments where the agent observes a manually encoded state representation of the environment, the 2D maze environment from de Lange (2017) is used. This will be referred to as 'environment 1'. In the subsequent experiments, the observation of the agent is the full image of the maze. The environment that is used for the experiments where the observation is the raw image of the maze, is the 2D maze environment called Minimalistic Gridworld Environment (MiniGrid) (Chevalier-Boisvert et al., 2018). The MiniGrid environment will be referred to as 'environment 2'. Both environments are a 2D deterministic grid world.

Environment 1

The environment that is used here is a 2D maze environment. In Figure 5.1 an empty maze of size 5×5 is visualized. The maze has a starting point (red cell), empty cells (white cells), walls (black cells) and a single exit (green cell). The agent can be placed at any of the empty cells and the agent's goal is to reach the exit (goal) as fast as possible. In order to reach the goal the agent must decide which action to take (left, right, up, down), i.e. it can move in one of the four cardinal directions each time step and does this without rotating. Hence, the action space contains the four cardinal directions. The environment is fully observable because the state representation contains the position of all the objects (walls, target, agent) in the maze.

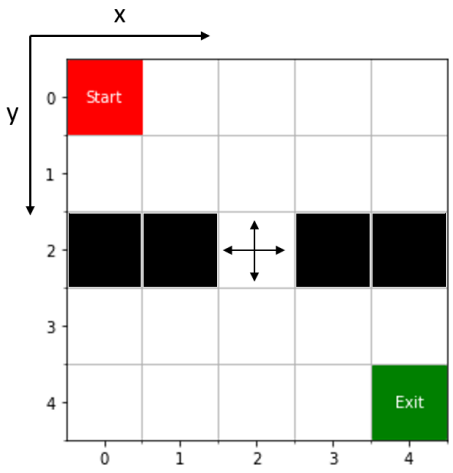


Figure 5.1: Visualisation of environment 1, best viewed in color. The maze is square sized (in this case 5×5). The start position is at $(0,0)$ indicated by the red square. The target position is at $(4,4)$ indicated by the green square. The black squares indicate obstacles and the agent cannot traverse to these positions. In the middle cell the possible actions for each empty cell are indicated, i.e. in every cell, except the obstacles the agent can take one of the four moves in the cardinal directions.

Environment 2

For the experiments where the observation is an image, environment 2 is used. In Figure 5.2 an example of the environment is shown. There are different configurations possible in environment 2, e.g. multiple rooms, keys, lava. This allows for extension to more advanced settings. This environment is depicted because it comes with a wrapper to generate observations of the maze as images. The image of the maze can be small in terms of pixel size without losing any information. Furthermore, the environment can be easily adjusted because the environment complies with the OpenAI gym environment structure (OpenAI, 2000), whereas environment 1 does not. Moreover, it is designed to be particularly simple, fast and lightweight. Few dependencies are needed to make it work and no external sprites or images are needed.

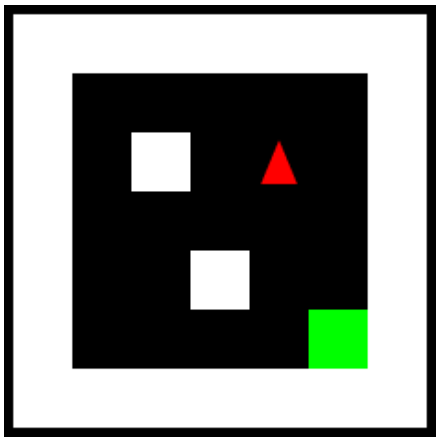


Figure 5.2: Visualisation of environment 2. Walls are colored white and the agent is the red triangle. The goal state is the green square and the free space is colored black. Note that an outer black border is added to make the outer walls (white space) visible. The edges of the environment consist always of a row of walls.

5.2 Experiments

5.2.1 Experiment - learn a model of the environment

For this experiment, environment 1 is used. The goal of this experiment is to train a transition model and reward model using both a GNN architecture and a NN architecture. A simplified setup is considered, where the states are manually encoded such that no encoder needs to be trained.

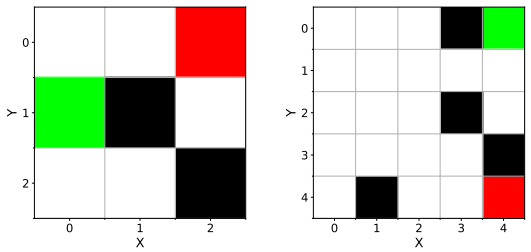
First, the environment model is trained for a specific maze configuration to verify the models can actually learn to predict state transitions and rewards. The maze configurations shown in Figure 5.5 are used for training and testing the environment models. The number of training epochs is set to 100.

Secondly, the obstacle positions, starting position and target position are randomized every episode during the generation of experience samples. This is done in order to have data that is uniformly distributed across the state space. This way the trained environment models will not overfit on one specific environment and it can be verified to what extent the learned models generalize to unseen environment configurations. The process of gathering experience samples is shown in Algorithm 5.

Algorithm 5: Gather experience samples by random interaction

Initialize number of episodes (dependent on the maze size)
Initialize max number of iterations per episode (dependent on the maze size)
for *every episode* **do**
    initialize environment with random objects positions, start position and target position
    while *not in terminal state* ***OR*** *number of steps* < *max steps* **do**
        perform random action in environment
        store in replay buffer: s, a, r, s', done, sample number
    end
end

In this experiment two different sized grid worlds are considered. Different environment models are trained for the following maze sizes: 3×3, 5×5. The amount of obstacles that is used, is such that a 2 room maze can be created during evaluation of the learned models. In Figure 5.3 a training sample for each maze with obstacles is shown. The obstacle positions, target position and start position are randomly initialized. The used hyperparameters for generating the replay buffer are shown in Table 5.1. The maximum number of steps is set to 2 times the amount of cells in a maze.



(a) Maze 3×3 **(b)** Maze 5×5

Figure 5.3: Training samples of mazes used for generating experience samples. Agent position (red), target position (green) and obstacle positions (black) are randomly generated.

Parameter	3 × 3 maze	5 × 5 maze
number of episodes train set	60	100
max number of steps per episode	18	50

Table 5.1: Hyperparameters for generating the experience samples by random interaction.

The (sparse) reward function used in environment 1 is shown in Equation 5.1. The agent receives a penalty for every step it takes. Consequently, by maximizing the cumulative reward it will find the fastest route to the target.

$$R(s,a) = \begin{cases} 0, & \text{reaching terminal state} \\ -1, & \text{otherwise} \end{cases} \tag{5.1}$$

During the collection of experience samples, a *train* set and a *test* set are created. After all the experience samples are collected, the environment model is trained. The transition model and reward model for both GNN and NN are learned offline in a supervised setting using the samples from the train set. The experience samples from the test set are used to validate the accuracy of the models on a separate set of experience samples. This is done to see how well the model performs on different experience samples, that are not included in the training data. The test set contains half of the number of episodes as in the train set. Furthermore, it is used to see whether we are overfitting on the training data. A MSE error between the predicted state and true state is used as a measure of accuracy. Moreover, for the accuracy of the predicted reward a MSE error is used as well. Hence, the measure of accuracy is the same as the used loss function that is used to train the models, Equation 4.1 and 4.2.

The hyperparameters used to train the environment models, are shown in Table 5.2.

Parameter	3 × 3 maze	5 × 5 maze
number of epochs	300	300
batch size	16	32
Hidden units	128	128
learning rate transition model	0.0005	0.0005
learning rate reward model	0.0005	0.0005

Table 5.2: Hyperparameters for training the reward model and transition model. For both the GNN and NN model.

5.2.2 Experiment - generalization of the learned environment model

In the previous experiment the models are evaluated on a test set. In this experiment, a closer look is taken at the generalization performance of the environment models to an unseen environment configuration. The goal is to see whether the learned environment models are sufficiently accurate, such that they can be used for planning.

The generalization capabilities of the transition and reward model implemented using a GNN, are compared to the generalization capabilities of the transition and reward model using a NN. This is achieved by learning the value function for environment configurations that are not included in the data used for training the environment models. The maze configurations shown in Figure 5.5 are used to assess whether the learned environment models are general enough to learn a good approximation of the value function.

The value function is obtained by sampling from the environment model and applying Q-learning by means of a DQN. The resulting value functions are compared to the true value function. The true value function is obtained by training a DQN with access to the environ-

ment. The obtained value functions from iterating on the learned environment models, are compared to the true value function with respect to the values. Furthermore, a greedy DQN agent is applied to see whether an optimal policy is obtained when the environment models are used to train the DQN.

The used DQN architecture is shown in Figure 5.4, a similar architecture is used in (Ouyang, 2019). The input state is low dimensional, therefore only three full connected layers are used, which should be sufficient for the simple problem at hand. The DQN architecture is the same for both the case of the training of a DQN using a GNN environment model, and for training a DQN using a NN environment model. The input layer is a 1D state vector. The GNN graph state representation is converted into a vector before passing it to the DQN network, i.e. the feature of the nodes are concatenated into a 1D vector. Note that the nodes in a graph do not have a strict order. However, in the graph format used in the Graph Nets library, the graph is defined by means of different numpy arrays. Hence, the node features can be taken in a consistent order to squeeze them into a vector representation. The first two hidden layers contain 128 neurons in each layer which are activated using a ReLU activation function. The Last layer is a linear layer, which corresponds to the Q-value for each possible action. The maximum value at the output of the DQN corresponds to the value.

The parameters used to train the DQN are shown in Table 5.3.

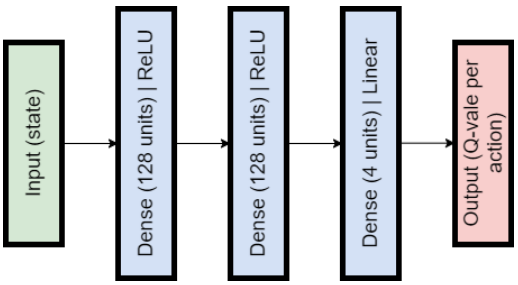
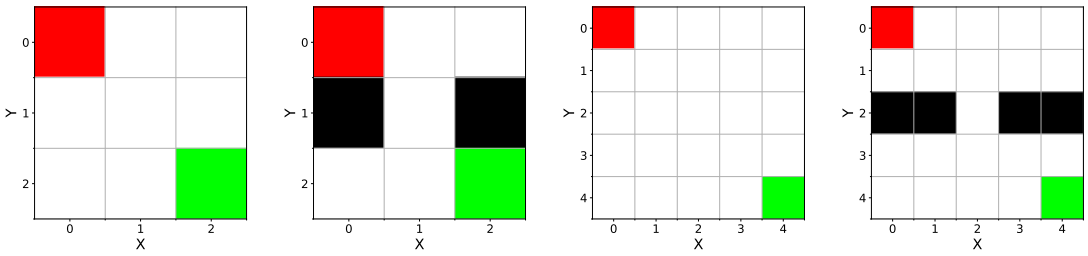


Figure 5.4: DQN architecture.

Parameter	3×3 maze	5×5 maze
Exploration factor (ϵ)	$0.8 \rightarrow 0.05$	$0.8 \rightarrow 0.05$
Minimal ϵ after % episodes	60%	60%
Discount factor (γ)	0.90	0.90
Learning rate	0.001	0.001
Batch size	128	128
Number of episodes	100	100
Maximum steps per epoch	100	100

Table 5.3: Hyperparameters for the trained DQN agent.



(a) Empty maze 3 × 3 **(b)** 2 room 3 × 3 **(c)** Empty maze 5 × 5 **(d)** 2 room maze 5 × 5

Figure 5.5: Mazes used for evaluation of the learned transition and reward models. These configurations are excluded from the train set that is used to train the environment models. A value function is used for these maze configurations by training a DQN on the learned environment models.

5.2.3 Experiment - learn state representation

This experiment builds upon the previous experiments. In this experiment, environment models will be learned from raw observation in order to learn a state representation. A latent state representation is learned by predicting the next state in latent space. Hence, a state embedding and transition model is learned simultaneously. Furthermore, a reward model is trained that takes as input the raw observation and the applied action. The transition model and reward model together form the environment model that takes as input a raw observation and action (one-hot encoded). The learned state representation obtained using a GNN environment model is compared to the learned state representation using a NN environment model.

For this experiment, environment 2 is used. This is the 'empty' environment from the MiniGrid environment (Chevalier-Boisvert et al., 2018). The default MiniGrid environment is slightly adjusted. MiniGrid environments are by default partially-observable, environment 2 is configured to be fully-observable. Furthermore, by default the agent turns left and right and takes a step forward to move to another tile. This requires the agent to know its orientation when navigating the environment. The actions space is adjusted, such that the actions are the four cardinal directions without turning. Moreover, the observation is adapted such that the outer edge walls are not included. These walls provide no information and make the observation space larger. The walls are simply cut-off before storing the observations in a replay buffer.

In the previous experiments, the state representation is either a graph representation or vector representation and directly used to learn the transition model and reward model, i.e. the state is the observation $s_t = o_t$. In this experiment, the state does not equal the observation $s_t \neq o_t$. The state representation is learned from raw image observations and encoded as either a graph representation or vector representation. The pipeline to learn the reward model and transition model is shown in the previous chapter, see Figure 4.6.

A slightly different reward function is used in environment 2. This reward function is dense and depends on the (normalized) distance of the agent to the target. The reward function is shown in Equation 5.2. A dense reward function facilitates learning a policy faster. Furthermore, a dense reward function makes it easier to learn the reward model. Moreover, when reward are sparse, the representation may collapse (van der Pol et al., 2020).

$$R(s,a) = \begin{cases} 1, & \text{reaching terminal state} \\ -\frac{|x_{\text{agent}} - x_{\text{target}}| + |y_{\text{agent}} - y_{\text{target}}|}{\text{width}_{\text{maze}} + \text{height}_{\text{maze}}}, & \text{otherwise} \end{cases} \tag{5.2}$$

The same architectures for the transition model and reward model are used as in the previous experiments, except that the number of hidden units is set to 512. A larger number of hidden units is used because the dimensionality of the input data is much larger. Furthermore, a CNN (object extractor) followed by a MLP (object encoder) are added to encode raw observations

into latent representations. The used CNN architecture for the object extractor is shown in Figure 5.6 and the architecture for the MLP object encoder is shown in Figure 5.7. This architecture is based on the presented object encoder and extractor in Kipf et al. (2020b), where the architecture was used for a similar environment.

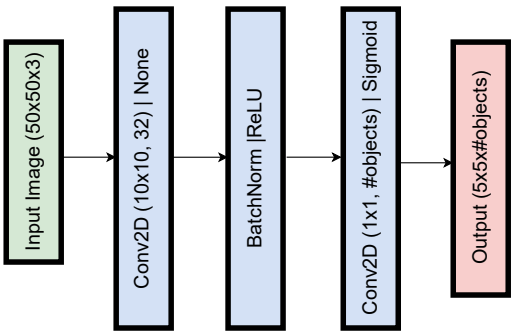


Figure 5.6: CNN architecture, object extractor. Dimensions based on input image of [50x50x3].

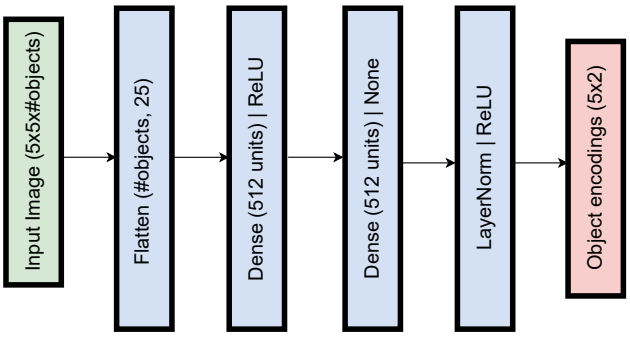


Figure 5.7: MLP architecture, object encoder. Dimensions based on input image of [50x50x3].

The object encodings resulting from the object encoder shown in Figure 5.7 are either used as a vector representation where the encodings are flattened, or graph representation where the encodings are the features of the nodes in a fully connected graph. For each object, a latent state dimension of 2 is used, corresponding to the x and y position of the objects. Ideally, the underlying grid structure of the environment appears in the learned state representation.

A contrastive loss function (see Equation 4.6) is used to learn the state representation and transition model. A MSE loss function (see Equation 4.7) is used to learn the reward model. The sum of the losses (see Equation 5.3) is minimized to train the environment model.

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{reward}} + \mathcal{L}_{\text{transition}} \tag{5.3}$$

First, an empty maze is considered with two objects, shown in Figure 5.8. The agent is the red triangle and the green square is the target. This is a 5x5 grid, tiles have a pixel size of 10. Hence, the total dimension of the observation is 50x50 pixels. Note that the outer white walls are cutt-off.

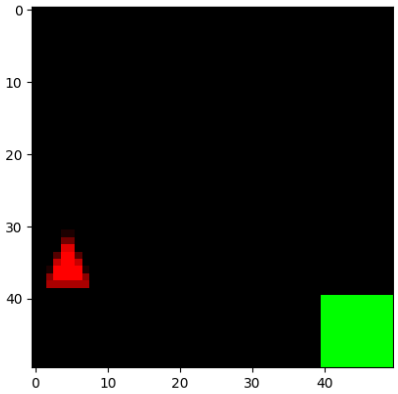


Figure 5.8: Example of an image observation of environment 2.

Experience samples are generated in the same way as in the first experiment, see Section 5.2.1, i.e. an agent interacting with the environment following a random policy. The agent walks randomly in the maze in order to generate experience samples for training. The replay buffer consists of 100 episodes. An episode either terminates after 100 steps or when the terminal state is reached. After the experiences buffer is created, learning the transition model and state representation is done offline. A learning rate of 0.0005 is used for both the transition and reward model for both NN and GNN implementation.

In order to check the correctness of the implemented GNN architecture in TensorFlow using the GraphNets library, the performance is compared to the model from (Kipf et al., 2020b) for one of the environments used in their work. This is can be found in Appendix E.

Parameter	5×5 empty maze
Number of epochs	100
Maximum steps per episode	50
Batch size	128
Hidden units	512
Learning rate transition model	0.0005
Learning rate reward model	0.0005

Table 5.4: Hyperparameters for training the reward model and transition model in latent state space. The same parameters are used in (Kipf et al., 2020b). A smaller batch size is used to relax memory management.

5.2.4 Experiment - planning in latent space

In this experiment, the goal is to train a DQN agent in latent space. The learned policy is applied in the original space.

Using the learned transition model and reward model from the previous experiment, see Section 5.2.3, a DQN agent is trained in latent space. Instead of iterating using an environment simulator, the learned models of the environment are used. An initial observation of the environment is fed through the object extractor and encoded (trained in the previous experiment) to obtain a latent state representation. The latent state representation and action are supplied to the transition model and reward model. Consequently, the predicted next latent state and predicted reward are obtained. The predicted next latent state and a new action serve as an input for environment model to predict the corresponding next latent state and reward, this is

repeated until termination (goal reached or maximum number of steps). The agent uses the environment model to plan the consequence of its actions and learns a policy in latent space. After training the DQN agent in latent space using the learned environment model, a greedy policy with respect to the predicted Q-values is applied in the environment.

The used parameters to train a DQN agent in environment 2 are shown in Table 5.5. Note that the parameters are similar to the DQN parameters in the previous experiment. The Discount factor was set to 0.99, this was crucial for the algorithm to converge.

Parameter	5×5 maze
Episodes	200
Maximum number of steps	100
Exploration factor (ϵ)	$0.6 \rightarrow 0.05$
Minimal ϵ after % of total episodes	60%
Discount factor (γ)	0.99
Learning rate	0.001
Batch size	128

Table 5.5: Hyperparameters for the trained DQN agent in environment 2.

5.3 Summary

In this chapter, the design of the experiments is elaborated. In Section 5.1, the used setup was discussed. In Section 5.1.1 the used software was presented. All the code is written in Python 3.7. To construct NN architectures the TensorFlow library is used and consequently the Graph-Nets library is used to construct GNN architectures. In Section 5.1.2 the used environments are described.

In Section 5.2 the different experiments are covered. Starting with learning the environment dynamics (in a simplified setting) in Section 5.2.1. A replay buffer is constructed from the experience samples of an agent following a random policy. With the obtained replay buffer, the environment dynamics (transition model and reward model) are learned in a supervised setting where the true labels are the next state and reward. In Section 5.2.2 the experiment concerning the generalization performance of the learned environment model is described. An unseen maze configuration is used, for which a policy is learned by means of a (greedy) DQN agent which is trained using a model of the environment. In section 5.2.3 the experiment is elaborated to learn a state representation from the image observations. An observation encoder and environment model is trained simultaneously in an unsupervised setting using a contrastive loss. In section 5.2.4, the model for the environment dynamics and the observation encoder are used to train a DQN agent in latent space.

This concludes the chapter on the design of the experiments. In the next chapter the results are presented.

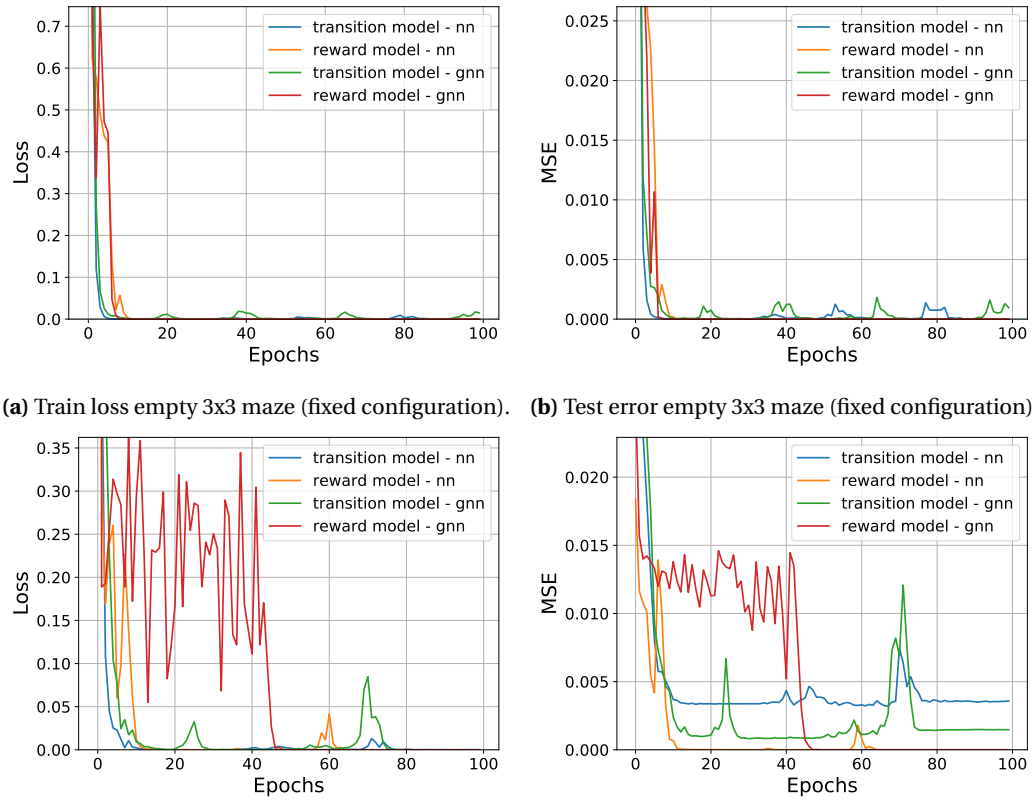
6 Results and Discussion

In this chapter, the results of the experiments discussed in Chapter 5 are presented. First, in Section 6.1 the results for training the transition model and reward model are presented for a fixed maze configuration. Followed, by section 6.2 where environment dynamics are learned for random maze configurations. The value functions for unseen maze configurations are presented in section 6.3. The value functions are obtained by training a DQN using solely the learned environment model. In section 6.4 the learned state representations for environment 2 are presented. The learned state representations are used to learn a policy in latent space. In Section 6.5, the results for learning a policy in latent space and applying it in the original environment are shown.

6.1 Learn environment model - fixed maze

First the transition model and reward model are trained and evaluated on the same maze environment. Hence, the maze is kept constant, i.e. the object positions, target position and starting position of the agent are fixed during training and testing. The only difference in this case are the random actions taken by the agent.

A reward model and transition model are learned for the mazes shown in Figure 5.5. The training loss plot and test error plot for the 3x3 maze are shown in Figure 6.1. Sometimes, the loss error spikes, this might be due to the randomly sampled batches and the used ADAM optimizer in combination with small error values.



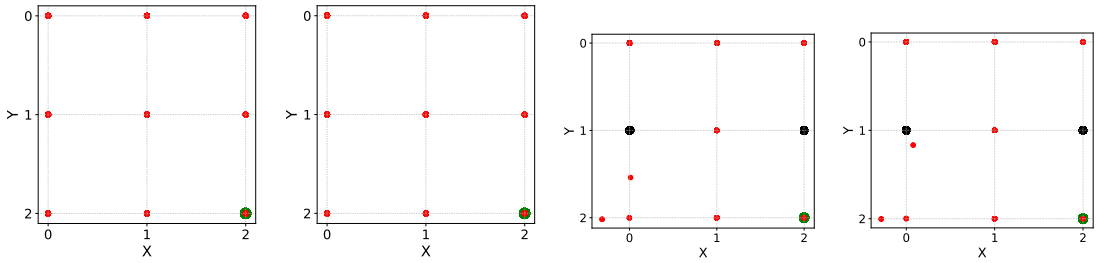
(a) Train loss empty 3x3 maze (fixed configuration). **(b)** Test error empty 3x3 maze (fixed configuration).

(c) Train loss 2 room 3x3 maze (fixed configuration) **(d)** Test error 2 room 3x3 maze (fixed configuration)

Figure 6.1: Train and test curves for both transition and reward model implemented using a GNN and NN for the 3x3 mazes. The maze is kept constant during training and tested on the same maze.

In Figure 6.2, the next states are predicted for all the experience samples in the test set for the 3x3 maze configurations. The dimension in Figure 6.2 correspond to the X and Y dimensions in the maze. A dotted line is used to indicate the discrete positions in the maze. Ideally, the predicted state map to cross sections of these dotted lines. Note that in Figure 6.2c and Figure 6.2d, there are two outliers for the predicted position of the agent for both models. This is due to an action that is invalid (e.g. moving to a position where there is an obstacle). The model is not able to predict the next state correctly, see Figure 6.3. This is reflected in the plot for the test set in Figure 6.6d, where the outliers cause the transition curves not to converge. Note that the learned environment models are not assumed to be discrete. The discrete nature of the grid world needs to be learned by the environment models. It is important that the mapping to the discrete position is correct otherwise the transition model is inaccurate and cannot be used for planning.

Training a GNN takes more time. To give an idea, for the 3x3 maze a training epoch for the NN takes approximately 0.1 second and a training epoch for the GNN takes approximately 1 second. For the 5x5 maze a training epoch for training the NN takes approximately 0.5 seconds and a training epoch for the GNN takes approximately 4 seconds. The code is executed on a laptop with a CUDA enabled GPU: NVIDIA Quadpro P1000. The additional computation time might be due to the Graph Nets library implementation. If the message passing is not implemented as a matrix operation this can significantly slow down the performance.



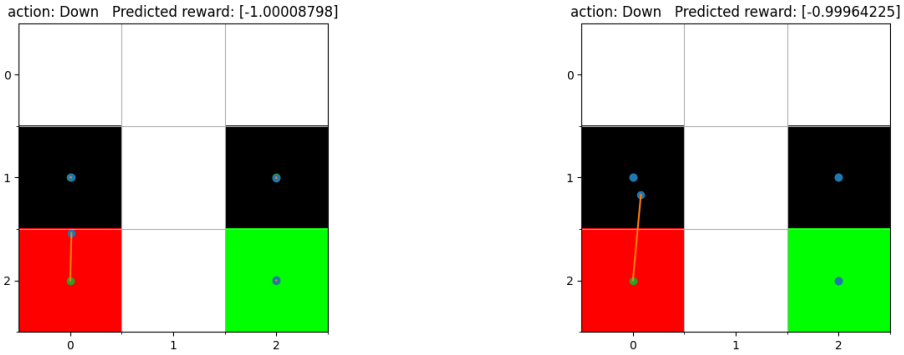
(a) No room 3x3 maze, GNN transition model.

(b) No room 3x3 maze, NN transition model.

(c) 2 room 3x3 maze, GNN transition model.

(d) 2 room 3x3 maze, NN transition model.

Figure 6.2: Next state predictions for all states in the test set for the 3x3 mazes. Dotted line indicated the discrete positions of the maze. Red is used for the position of the agent, green for the target position and black for the obstacle position.



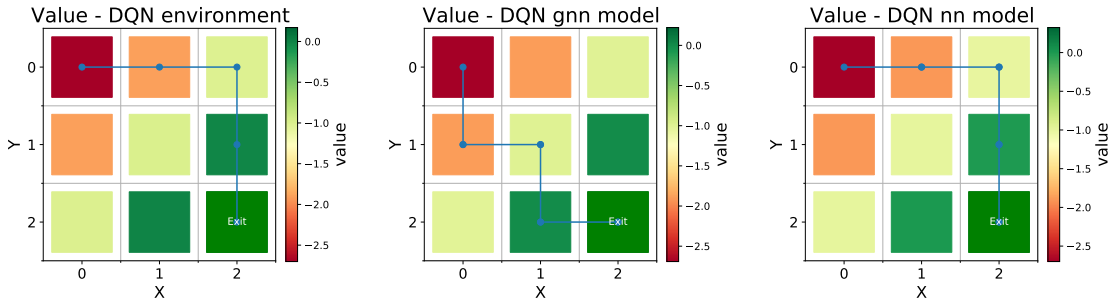
(a) Outlier in the GNN transition model.

(b) Outlier in the NN transition model.

Figure 6.3: Visualisation of the situation of 1 of the outliers for both models. The agent takes an action (down) which results in no change of position due to the edge of the maze. However, the model predicts a position which is the resulting outlier. The blue dots are the predicted positions, the green dots are the true positions and the orange line is used to indicate to which green dot the predictions belong.

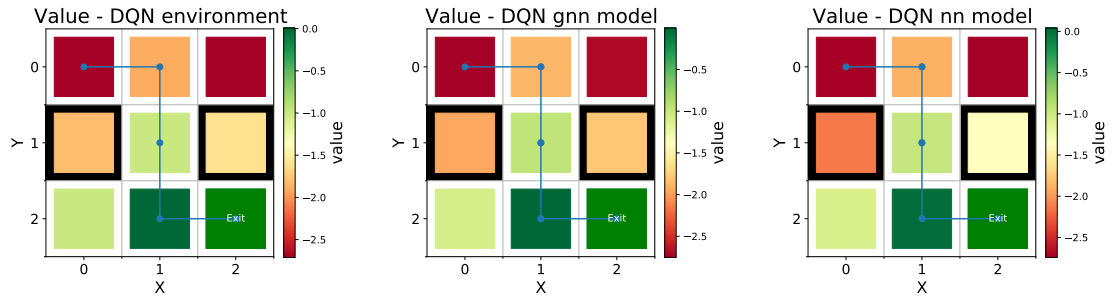
The learned transition model and reward model are used to train a DQN agent to find the optimal path to the target position. In Figure 6.4 and Figure 6.5 the value functions is shown that was obtained using direct interaction with the environment, planning using the learned GNN environment model and planning using the learned NN environment model respectively.

The target states are reached in the minimal amount of steps. However, the path (blue line) towards the goal differs for the DQN agent trained using a GNN environment model. This is caused by the randomness that occurs when training a DQN agent. However, the path is optimal for the problem at hand and in both cases the learned models are sufficiently accurate to perform planning.



(a) DQN agent, access environment **(b)** DQN agent, GNN model **(c)** DQN agent, NN model

Figure 6.4: Value function calculated using DQN agents trained for the empty 3x3 maze: a) with direct access to the environment simulator, b) using the trained GNN model, c) using the trained NN model. The blue line indicates the greedy policy that is obtained when acting greedily with respect to the obtained value function indicated by the colored tiles.



(a) DQN agent, access environment **(b)** DQN agent, GNN model **(c)** DQN agent, NN model

Figure 6.5: Value function calculated using DQN agents trained for the 3x3 maze with 2 rooms: a) with direct access to the environment simulator, b) using the trained GNN model, c) using the trained NN model. The blue line indicates the greedy policy that is obtained when acting greedily with respect to the obtained value function indicated by the colored tiles.

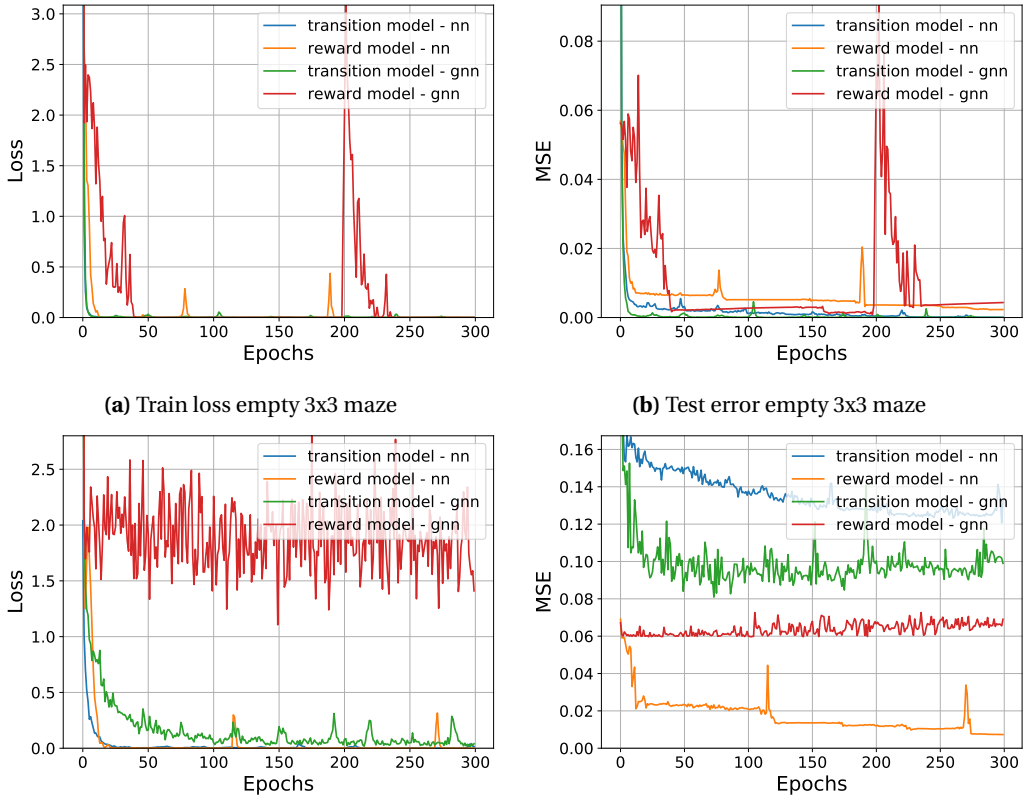
In Appendix F, the same plots are shown for the 5x5 maze configuration. Similar results are observed, besides the fact that the reward model is problematic for the GNN to learn. Nevertheless, it still finds the optimal path.

6.2 Learn environment model - random maze

Next, the mazes are randomized during training of the transition and reward models in order to generalize to novel maze situations. For reproducibility of the results, seed(1) was used for Python, Numpy and TensorFlow.

In Figure 6.6 and Figure 6.8 the train losses are presented for training on the experience samples gathered in randomly generated mazes, some of these random maze samples were shown in 5.3. Every epoch, the test error is calculated on a separate test set of experience samples. The test error plots are shown in the figures on the right in Figure 6.6 and Figure 6.8. Note that the errors tend to get very small for the test error curves. The reason for this is that the average error is computed for the entire test set at once. The train loss is computed for a batch of training samples. Hence, the train loss is the summation of training losses for all batches in an epoch.

In Figure 6.7 the states for each object (e.g. obstacles, agent, target) are predicted for the samples in the test set. The transition model predicts the next states for the objects, target and agent in the maze. The maze environment has discrete positions and therefore the output of the transition model should ideally be close to these discrete positions indicated by the dotted lines.



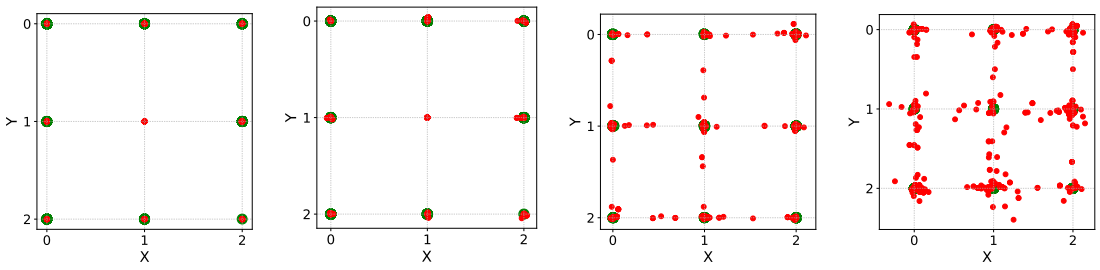
(a) Train loss empty 3x3 maze

(b) Test error empty 3x3 maze

(c) Train loss 2 room 3x3 maze

(d) Test error 2 room 3x3 maze

Figure 6.6: Train and test curves for both transition and reward model implemented using a GNN and NN for the 3x3 mazes.



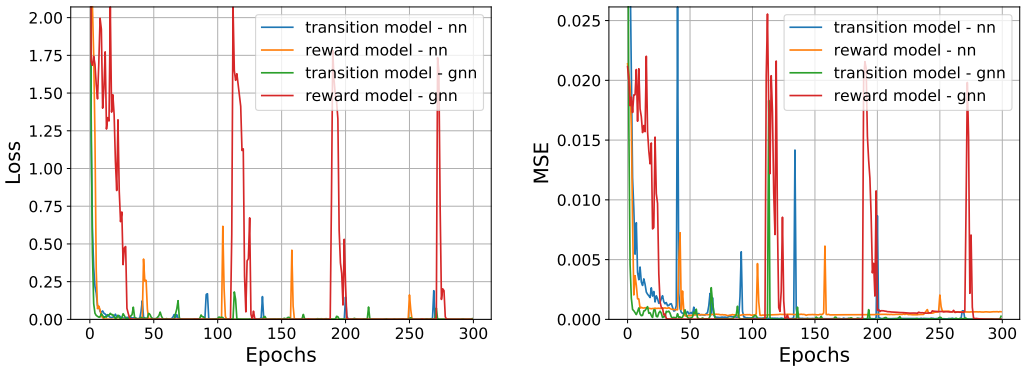
(a) No room 3x3 maze, GNN transition model.

(b) No room 3x3 maze, NN transition model.

(c) 2 rooms 3x3 maze, GNN transition model.

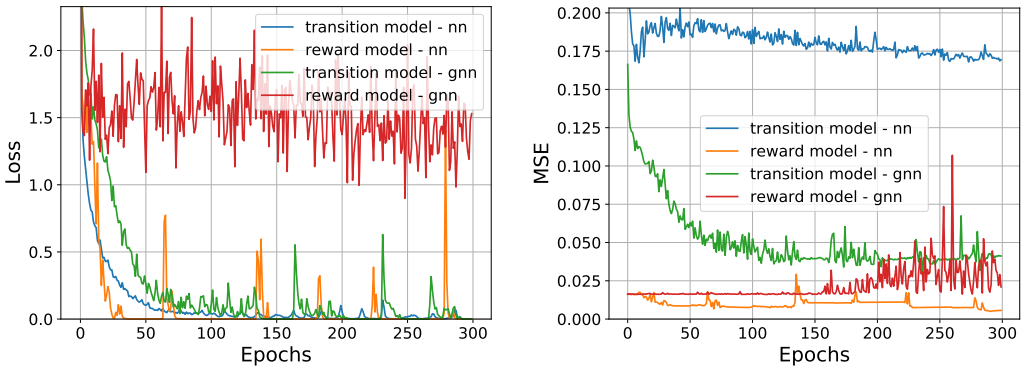
(d) 2 rooms 3x3 maze, NN transition model.

Figure 6.7: Prediction for the states in the test set for the 5x5 mazes with fixed obstacles. Red is used for the position of the agent, green for the target position and black for the obstacle position.



(a) Train loss empty 5x5 maze

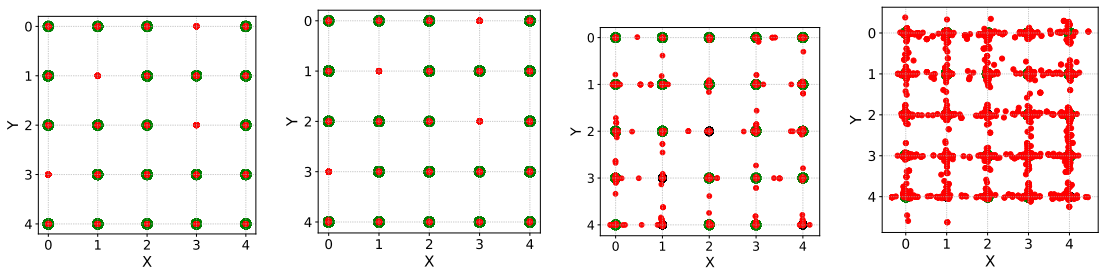
(b) Test error empty 5x5 maze



(c) Train loss 2 room 5x5 maze

(d) Test error 2 room 5x5 maze

Figure 6.8: Train and test curves for 5x5 mazes



(a) No room 5x5 maze, GNN transition model. **(b)** No room 5x5 maze, NN transition model. **(c)** 2 rooms 5x5 maze, GNN transition model. **(d)** 2 rooms 5x5 maze, NN transition model.

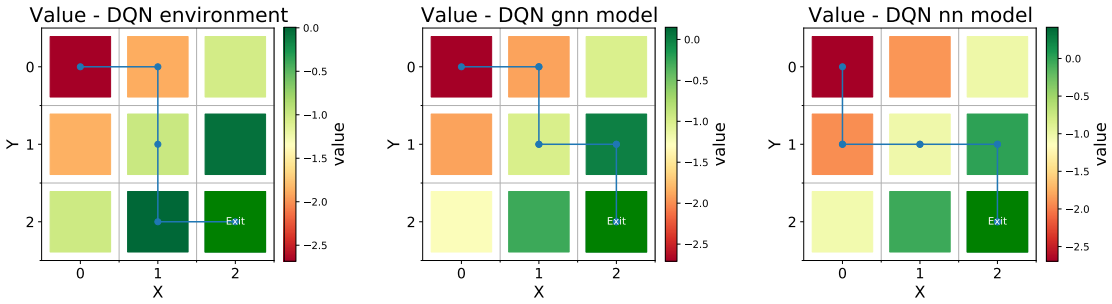
Figure 6.9: Prediction for the states in the test set for the 5x5 mazes with fixed obstacles. Red is used for the position of the agent, green for the target position and black for the obstacle position. Note that the assumed locations are continuous.

From the training and testing curves it can be concluded that the transition model implemented using GNN is best performing. Furthermore, the reward function for the GNN is worst because it learns to map everything to the most seen reward, i.e. -1. In the Figures 6.7 and 6.9 it can be seen that the GNN predicts the state better compared to the NN. However, GNN state predictions are not perfect either. Especially in the case when the maze has obstacles, the same problem occurs as mentioned in Figure 6.3. An incorrect transition is predicted, it cannot handle that state and corresponding action. To get an idea of the error that occurs in the transition model, the true states are plotted connected to the predicted states. This is added as an Appendix, see Figure F.5. It is similar to the plots shown in Figure 6.7 and Figure 6.9 but with an line between the true state and corresponding predicted state.

Hence, it cannot handle the generalization to unseen environments properly, both for the GNN model and NN model. Furthermore, the GNN cannot correctly predict the reward when obstacles are present in the maze. The NN transition model performs badly when obstacles are present in the maze. Both environment models will not be accurate enough to be used for planning, as is shown in the next section.

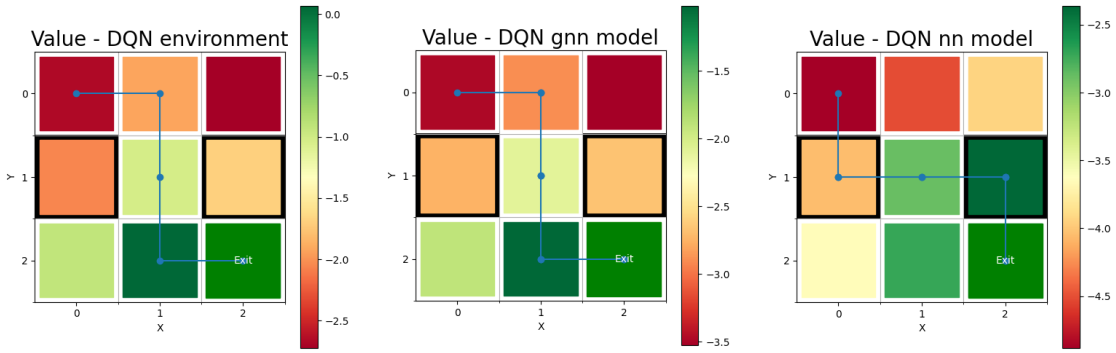
6.3 Generalization to new maze configurations

The learned environment models are used to do planning. DQN agents are trained that use the models to do planning. The obtained value functions are shown in Figure 6.3, 6.11, 6.3 and 6.13.



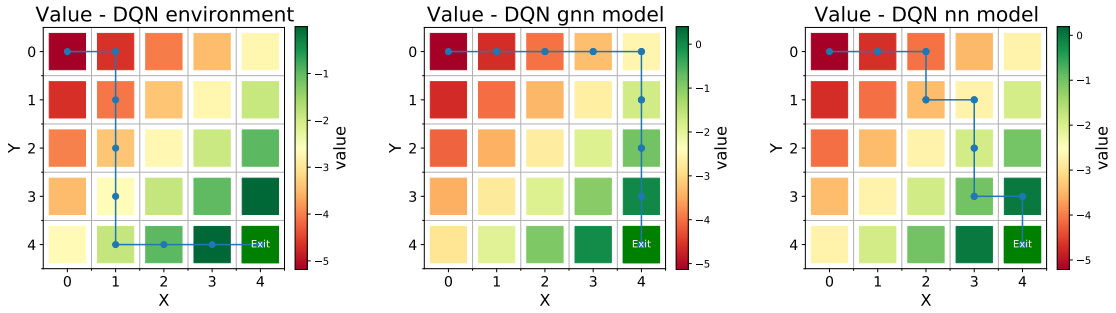
(a) DQN agent, access environment **(b)** DQN agent, GNN model **(c)** DQN agent, NN model

Figure 6.10: Value function calculated using trained DQN agent for the empty 3x3 maze: a) with direct access to the environment simulator, b) using the trained GNN environment model, c) using the trained NN environment model. The blue line indicates the greedy policy.



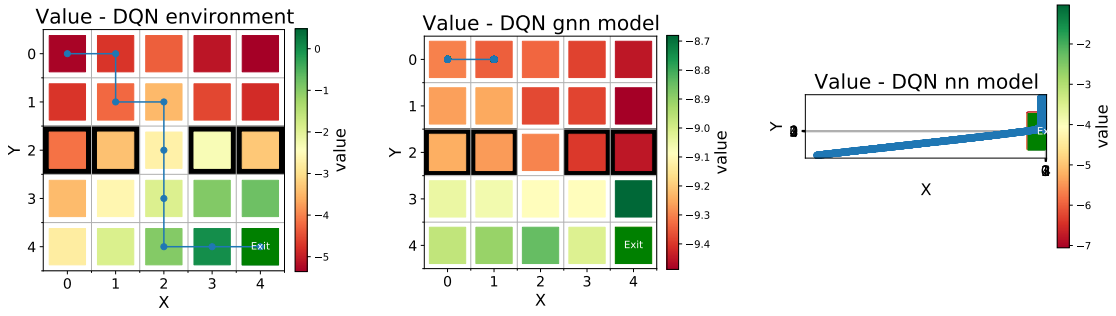
(a) DQN agent, access environment **(b)** DQN agent, GNN model **(c)** DQN agent, NN model

Figure 6.11: Value function calculated using DQN agents trained for the 3x3 maze with 2 rooms: a) with direct access to the environment simulator, b) using the trained GNN model, c) using the trained NN environment model. The blue line indicates the greedy policy.



(a) DQN agent, access environment **(b)** DQN agent, GNN model **(c)** DQN agent, NN model

Figure 6.12: Value function calculated using trained DQN agent for the empty 5x5 maze: a) with direct access to the environment simulator, b) using the trained GNN environment model, c) using the trained NN environment model. The blue line indicates the greedy policy.



(a) DQN agent, access environment **(b)** DQN agent, GNN model **(c)** DQN agent, NN model. The agent drifts out of the maze.

Figure 6.13: Value function calculated using trained DQN agent for the 3x3 maze empty maze: a) with direct access to the environment simulator, b) using the trained GNN environment model, c) using the trained NN environment model. The blue line indicates the greedy policy. Note the somewhat strange figure on the right. Here the environment model is incorrect and the agent thinks it can move outside the maze. It drifts outside the maze.

In the maze configurations without obstacles, the agent learns the optimal policy. This can be seen from Figures and . For the 3x3 maze with obstacles, the learned GNN environment model complies with the state space of the environment, and can be used to learn the optimal policy, see Figure 6.11b. However, the learned values do not correspond with the true values. For the 3x3 maze with obstacles, the learned NN environment model does not adhere to the state space and the agent thinks it can move through obstacles. It learns policy which is impossible, see Figure 6.11c.

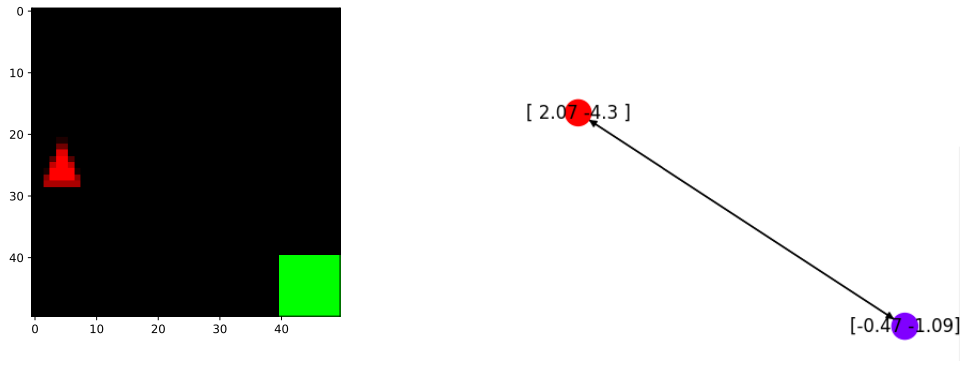
The learned NN environment model does not generalize well to other situations with obstacles. The environment model thinks it can cross obstacles positions, see Figure 6.11c or it drifts out of the maze, see Figure 6.13c. The GNN does the job a bit better as a transition model. However, due to its poor reward model, it does not learn the correct value, see Figure 6.13c. Hence it is not able to learn a good policy.

6.4 State representation learning

The state representation for environment 2 is learned using two different architectures for the transition model. The first approach uses a GNN architecture as transition model. The second approach uses a NN architecture as transition model.

6.4.1 State representation using GNN transition model.

The GNN transition model takes as input the state encoded as a graph. The features of the nodes in the graph are the object encodings, as shown in Figure 4.6. Ideally, the network learns that one node corresponds to the agent and the other node corresponds to the target, as was the case in Kipf et al. (2020b). A random observation sample and its corresponding latent state are shown in Figure 6.14.



(a) Observation of maze.

(b) State representation as a graph.

Figure 6.14: In a) an observation of the maze and b) its corresponding state representation as a graph. The color of the nodes in the graph has no meaning, nodes are colored to distinguish nodes between different state representation plots.

The training losses are shown in Figure 6.15. The best model is stored for later use when learning a policy. Note that the model converges rather quick, after 25 epochs there are no significant improvements. It is easier to learn a reward model for a dense reward function compared to the sparse reward function used in environment 1.

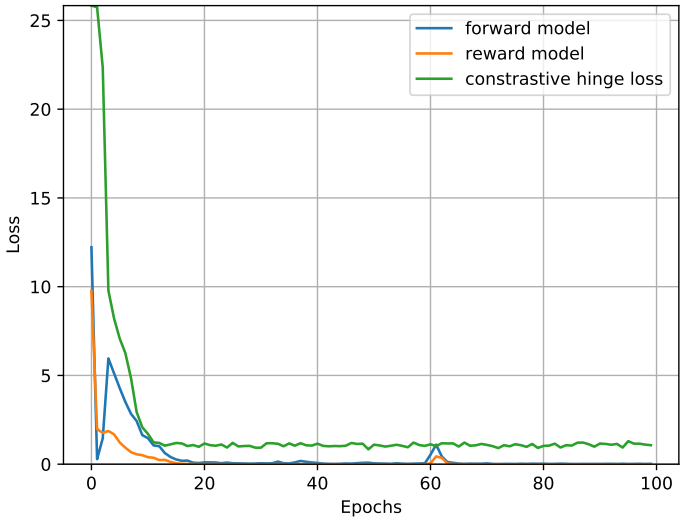
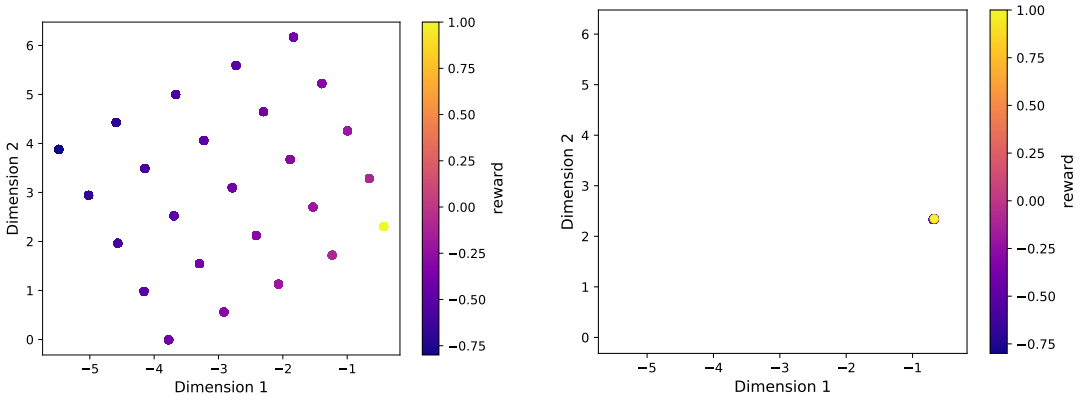


Figure 6.15: Training curve - GNN environment model.

All the observations in the replay buffer are encoded using the trained object extractor and trained object encoder preceding the GNN transition model. The pipeline was shown above in Figure 4.6a. Plotting the node features for all latent (graph) state representation results in the plots shown in Figure 6.16. Note that the network learned the x and y position of the objects apart from a random rotation. The same behaviour was observed in Kipf et al. (2020b). The additive format of the transition function did not put any constraint on the rotation in the latent state space. The agent adheres clearly to the underlying grid structure of the grid world. Note that the state space is continuous, and yet the discrete behaviour is captured. Moreover, the node features of the target object are all mapped to the same point because the target is static for all the observations in the replay buffer, i.e. the target is not moving. This is also the reason that the contrastive hinge loss, shown in Figure 6.15 in green, does not converge to 0 but to 1 (the used margin in the hinge loss).



(a) State representation node 1.

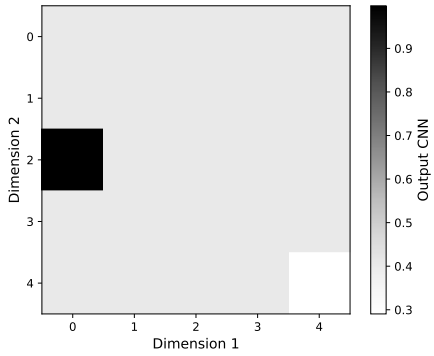
(b) State representation node 2.

Figure 6.16: The node features are plotted for each node in the graph state representation. The color bar indicates the reward obtained for traversing to that specific state. Dimension 1 and Dimension 2 are the dimensions of the feature vector on the nodes.

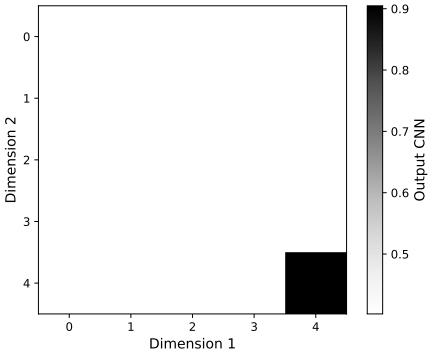
The object extractions, obtained from the object extractor shown in the pipeline in Figure 4.6a, are given in Figure 6.17. The number of object slots (i.e. nodes in the graph) is predefined and

determines the number of object extractions that will be generated by the CNN object extractor. In the case of 2 objects slots, the object extractor generates 2 object extractions. Each extraction is used to make an object encoding. Hence, 2 object encodings are obtained (Figure 6.16).

Ideally, an object extraction completely extracts one object. The learned object extractions in Figure 6.17, clearly extract the individual objects. However, in the object extraction of the agent some artifacts of the target extraction are present. This influence is small and still an expressive state representation is learned, which captures the underlying grid structure of the environment.



(a) CNN output filter 1 (Object extraction 1).

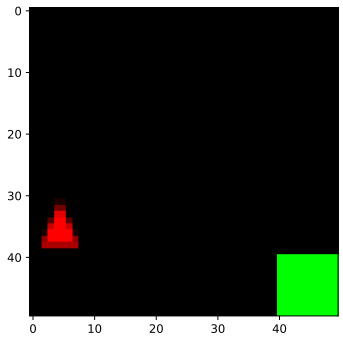


(b) CNN output filter 2 (Object extraction 2).

Figure 6.17: The object extraction per object slot for the observation shown in Figure 6.14. The transition model is implemented using a GNN. Black corresponds to the highest value in the object extraction and white to the lowest value. Everything in between is gray scaled according to the output value of the CNN layer.

6.4.2 State representation using object-oriented NN transition model.

As comparison, the same architecture is used to learn a state representation except the GNN is replaced by a NN. However, the same object-oriented abstraction of the observation is used. Instead of using the object encodings as node features in a graph, the object encodings are concatenated in a vector. This vector is used as a state vector for the NN transition model.



(a) Observation of maze.

```
[[-0.46671706 -1.0921156   2.0698962  -4.3016367 ]]
```

(b) State representation as a vector.

Figure 6.18: a) an observation of the maze and b) its corresponding state representation as a vector. The first two elements in the vector corresponds to the encoding of object extraction 1, and the third and fourth element corresponds to object extraction 2.

The training losses are shown in Figure 6.19. The best model is stored for later use when learning a policy.

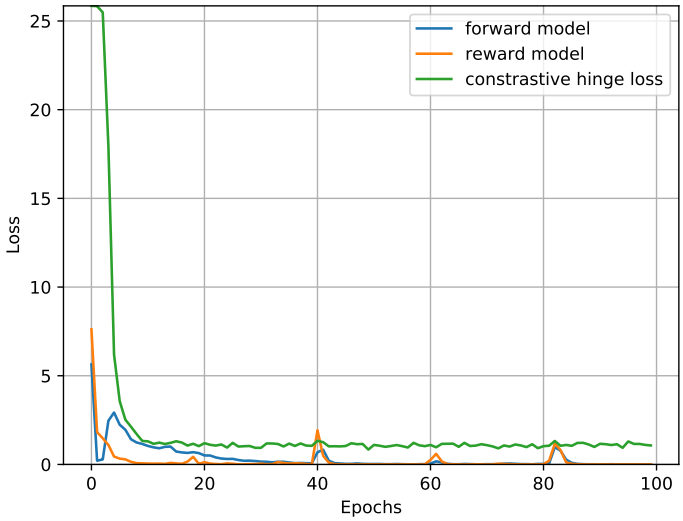
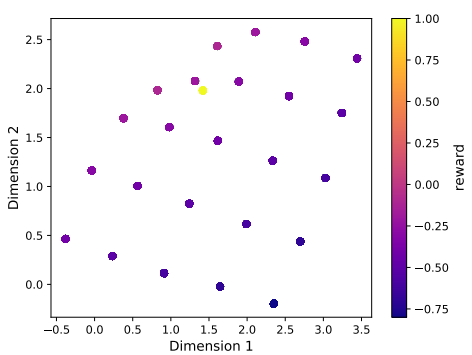
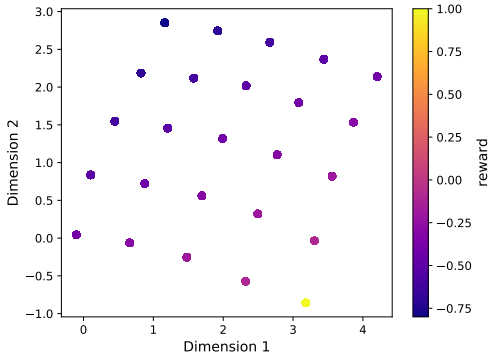


Figure 6.19: Training curve - NN environment model.

The object-oriented state representation for both objects is shown in Figure 6.20. The state vector consists of the concatenated MLP encodings. In this case, 2 objects are present in the environment (agent and target) resulting in a 4 dimensional state vector. The first 2 dimensions of the state vector should correspond to object 1 and the state embedding is plotted for all the samples in the replay buffer in Figure 6.20a. The replay buffer contains 3325 samples and all map to one of the 25 positions the agent can take. Furthermore, the state embedding for object 2 is plotted in Figure 6.20b. In both figures a clear grid structure is observed. However, the target is not moving and hence the object-oriented state representation does not succeed to decouple the states of the different objects. The object extractions for the object-oriented NN environment model, are not able to separate the 2 objects. The object extractions per object are shown in Figure 6.21. Nevertheless, it correctly learns that there are 25 different states in the environment considered.

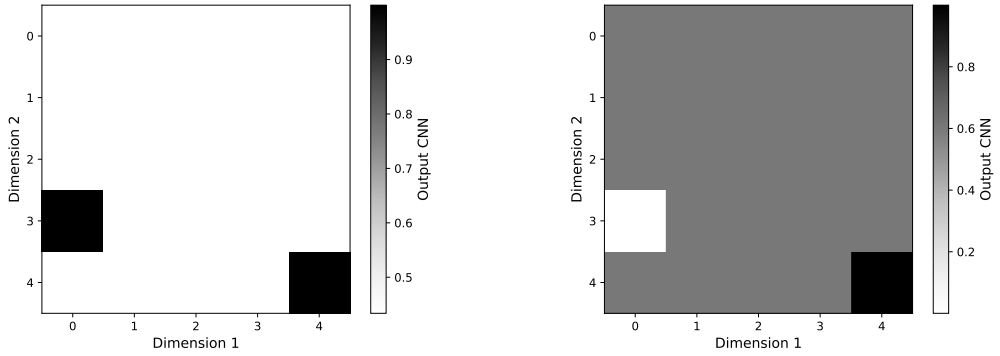


(a) State representation vector (dimension 1 and 2).



(b) State representation vector (dimension 3 and 4).

Figure 6.20: The elements of the state vector plotted per object. a) the first two elements in the vector representation, b) the third and fourth element in the vector representation. The color bar indicates the reward obtained for traversing to that specific state.



(a) Object extraction 1.

(b) Object extraction 2.

Figure 6.21: The object extraction per object slot for the observation shown in Figure 6.18. The transition model is implemented using an object-oriented NN. The color has no specific meaning other than that the relative difference is important. Black corresponds to the highest value in the object extraction and white to the lowest value. Everything in between is gray scaled according to the value. Values range from 0 to 1 due to the used sigmoid activation.

Applying PCA to the total state vector results in the plot shown in Figure 6.22. The two principal components show the underlying grid structure of the environment, apart for a random rotation.

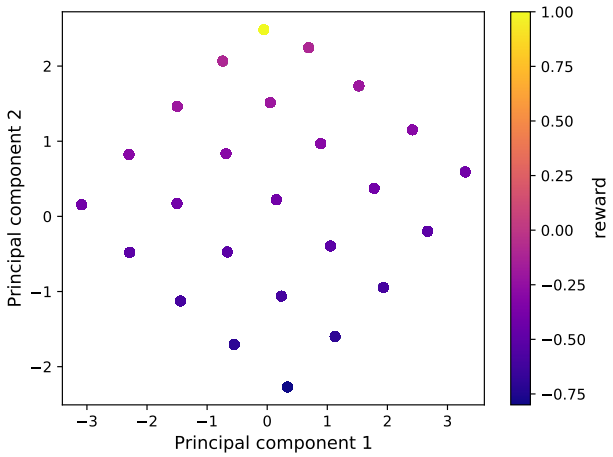


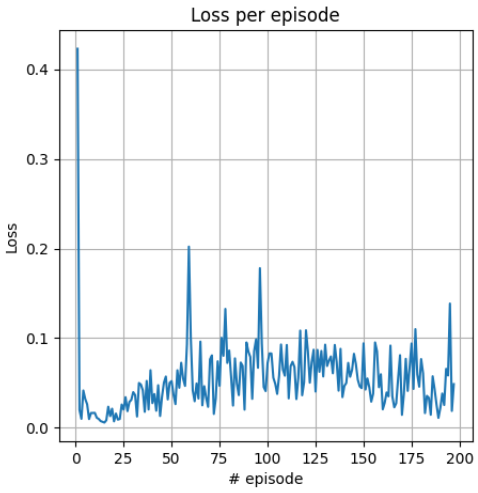
Figure 6.22: PCA plot of the 4 dimensional state vector for all the states in the replay buffer. The color bar indicates the reward.

6.5 Planning in latent space

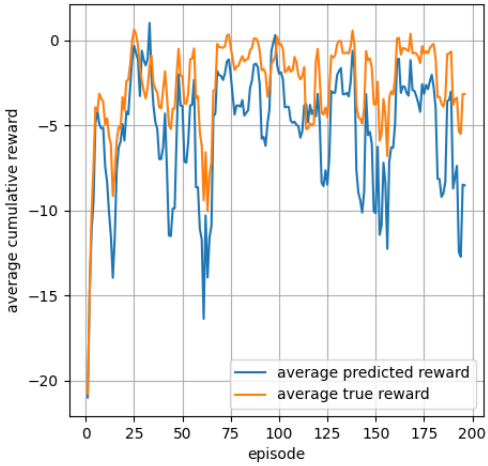
The state representations learned using both models (NN and GNN), shown in the previous section, are clearly separated and preserve the underlying grid structure. The trained object encoders for the different architectures are used to encode the initial observation. The initial state embedding is then used as a starting point for a DQN agent to learn a policy. Instead of interacting with an environment, the DQN agent uses the learned transition model and reward model. The transition and reward model take as input the latent state representation and action. The original action space is used, action equivariance is assumed. Hence, a policy in latent state space is learned which is then used in the original state space.

6.5.1 Training in latent space using a GNN environment model.

The DQN agent that is trained in latent space using the GNN environment model, is able to learn the optimal policy. During training, the predicted cumulative reward and true cumulative reward are logged and shown in Figure 6.23b. Furthermore, the training loss of the DQN network is plotted in Figure 6.23a. The predicted rewards follow the true rewards with some offset. An offset in the reward prediction is not problematic when this offset is constant. If this offset is constant, the relative value difference is preserved. If the offset is not constant, a different value function will be learned and the greedy policy might not be optimal.



(a) DQN training loss.



(b) Average cumulative reward obtained during training. Moving average over 10 episodes.

Figure 6.23: Training a DQN agent using the GNN environment model.

The DQN agent acts greedy with respect to the Q-values. The value function obtained for the DQN trained using the GNN environment model is shown in Figure 6.24.

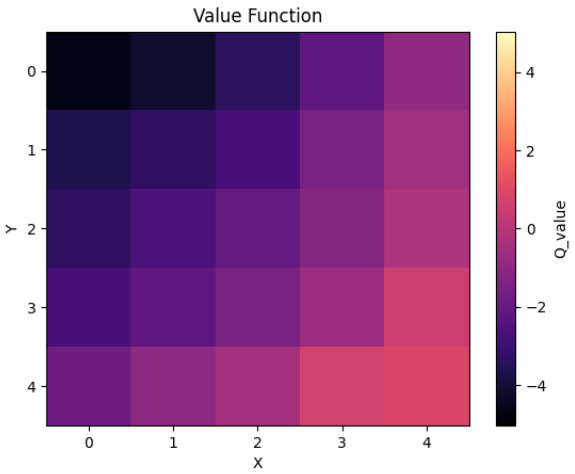
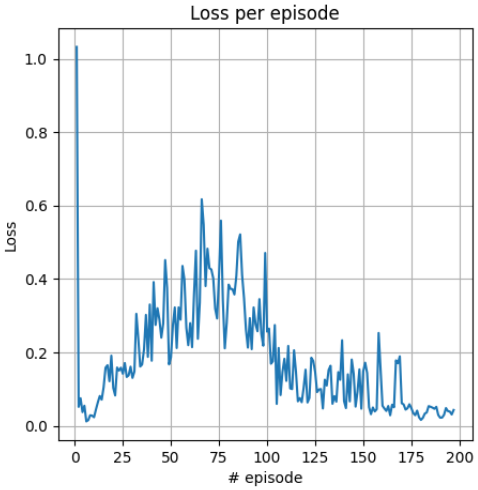


Figure 6.24: The value function obtained by taking the maximum of the Q-values for all possible states. Colors indicate the value for that state.

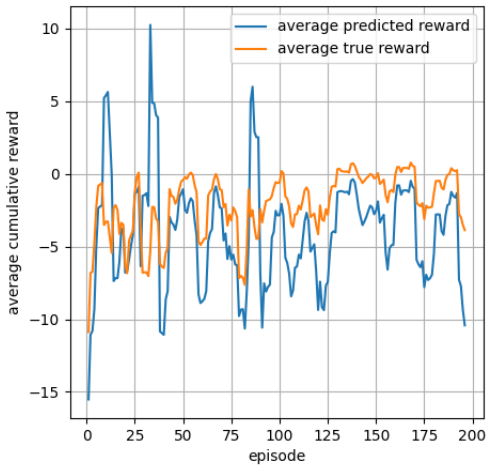
If you are reading this on an electronic device with internet connection, you can click the blue text to see a video of the agent in action.

6.5.2 Training in latent space using an object-oriented NN environment model.

The DQN agent trained in latent space, using the environment model implemented with an object-oriented NN, is able to learn the optimal policy as well. The training loss of the DQN network and the average cumulative reward per episode are shown in Figure 6.25. Note however, that the reward function has some clear spikes above the maximum achievable cumulative reward of 1.



(a) DQN training loss.



(b) Average cumulative reward obtained during training. Moving average over 10 episodes.

Figure 6.25: Training a DQN agent using the object-oriented NN environment model.

The learned reward function has some error but is still sufficient enough to learn a decent value function. By acting greedy with respect to the value, an optimal policy is obtained. The value function obtained from the DQN trained using the object-oriented NN environment model is shown in Figure 6.26.

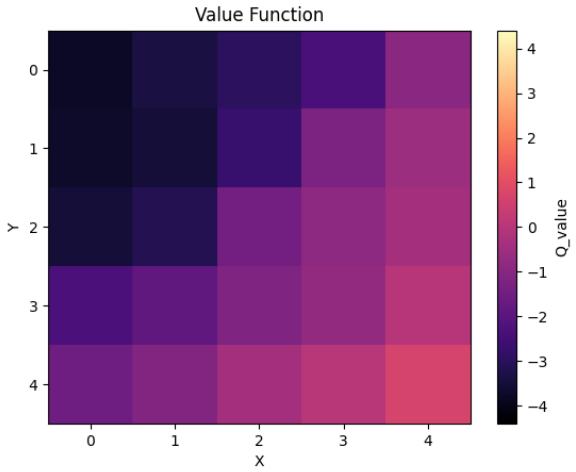


Figure 6.26: The value function obtained by taking the maximum of the Q-values for all possible states. Colors indicate the value for that state.

If you are reading this on an electronic device with internet connection, you can click the blue text to see a video of the DQN agent in action.

7 Conclusion

In this chapter, the work is concluded. In Section 7.1 the research questions are answered. At last, in Section 7.2 recommendations regarding future work are discussed.

7.1 Answering the research questions

The main goal of this project was to improve upon the state of the art in representation learning. The focus was on learning useful state representations from raw pixel observations without supervision. To achieve this, a transition model is trained in latent space, this is one of the strategies to learn a state representation (Lesort et al., 2018). A GNN model and NN model have been compared in their ability to model the transition and reward function. To study the problem, a navigation task in a deterministic 2D grid world is used.

Before learning a state representation, the GNN was tested as a transition and reward model using an object-oriented state representation that was manually encoded. The GNN architecture was compared to a NN architecture to answer the first research question:

RQ1 *"How can a graph neural network be used to learn a transition and reward model in a fully observable deterministic 2D grid world"*

The used encoded representation of the state was inspired on state encodings used in related work (Battaglia et al., 2016; Hamrick et al., 2018; Kipf et al., 2020b). A transition model and reward model implemented using both a GNN and NN was learned in a supervised setting in a static environment. A DQN agent was trained using the learned environment models, in order learn a policy. Both NN and GNN implementation for the environment model was sufficient to learn an optimal policy in static maze environments.

Learning a policy for unseen maze configurations showed to be more challenging. The NN transition model and reward model could not effectively be used to learn a policy for unseen maze configurations. The NN transition model fails to learn the correct state space. The agent thinks it can go through walls (Figure 6.11c), or even walk out of the maze (Figure 6.13c). The GNN transition model does a better job at generalizing to unseen maze configurations. However, when the environment is larger, the GNN environment model is not accurate enough to generalize to unseen maze configurations, and cannot be used to learn an optimal policy. This is a result of the inaccurate GNN reward model.

To answer RQ1, the GNN showed to perform better as a transition model than the NN transition model. However, the GNN reward model is not suited to learn a sparse reward function. The NN environment model was sufficient in the static maze configurations to learn a policy, but failed for the case of unseen maze configurations with obstacles.

The focus of the second research question was to learn a transition model and reward model in latent space, in order to learn a state representation:

RQ2 *"To what extent can a graph neural network be used effectively to learn a transition and reward model in a latent space for a fully observable deterministic 2D grid world?"*

By using a GNN as transition model, the true underlying state for each object was learned. The part of the state representation corresponding to the agent, adhered clearly to the underlying grid structure of the grid world. Moreover, the part of the state representation belonging to the target position was mapped to 1 point in the latent state space, see Figure 6.16. The learned reward model and transition model were sufficient to do planning in latent space using a DQN agent and learn an optimal policy.

Using the NN transition model to learn a state representation, the learned state representation was not decoupled per object. However, the two principal components of the total state vector also showed a clear grid structure. The learned state representation showed to be sufficient to train a DQN agent using solely the learned NN environment model.

The advantage of using a GNN as opposed to NN is that the learned state representation is decoupled per object. However, in the considered environment, the decoupled state representation does not show clear benefits when learning a policy in latent space, compared to the state representation learned using a NN environment model. For both implementations of the environment model, a DQN agent was successfully trained that followed the optimal policy.

The main research question of this work is:

"To what extent can a graph neural network be used effectively in state representation learning in a fully observable deterministic 2D grid world, to increase an agent's generalization performance to novel environment scenarios?"

To conclude, the main benefit of using a GNN in representation learning is that it allows for decoupling the state representation per object. In a problem with multiple objects of which the state information is needed to perform a certain task, state representation learning using a GNN will be beneficial. The GNN allows to learn expressive representations per object, which is not possible using a conventional NN architecture. For simple environments, there will be no benefit in using a GNN as opposed to a NN to learn a state representation.

7.2 Recommendations for future work

7.2.1 More challenging environments

The environments considered in this work are extremely simple. Therefore, it is important to extend this work to more complicated environments. For example, environments with more objects, non deterministic environments, partially observable environments.

7.2.2 Automate the object slot choice

The number of object slots (nodes in the graph) is a hyperparameter. Automatic selection of the object slots would make the GNN more applicable in unsupervised scenarios in dynamic environments.

7.2.3 DQN-GNN agent

An interesting research direction would be to implement a model free DQN agent that uses a GNN as estimator network.

A Activation functions

In this appendix the most often used activation functions are given. This is by no means a complete list but serves as a reference to be able to understand what activation function is used in the majority of scientific work related to NNs.

There are several activation functions that can be used in neural network architectures. For example, in AlexNet (Krizhevsky et al., 2012) the *ReLU* activation function is used in their computer vision architectures because of the speed advantage ReLU provides with respect to saturating nonlinearities such as *tanh* and *sigmoid*. Furthmore, the language representation model BERT (Bidirectional Encoder Representations from Transformers) (Devlin et al., 2018) uses a smooth version of the ReLU called *GELU* (Hendrycks and Gimpel, 2016).

Activation functions are chosen based on their empirical results from previous works and their mathematical properties. An important property is that the activation should be differentiable in order to do backpropagation. Another important property is that the activation function is non-linear. According to the the 'Universal Approximation Theorem' (Csáji, 2001), a neural network consisting of one hidden layer and sufficient but finite units can approximate any continuous function to a reasonable accuracy using non-linear activation function similar like the sigmoid activation functions (Cybenko, 1989). Furthermore, (Hornik, 1991) showed that it is not the choice of activation function but rather the multi-layer architecture that facilitates the NNs to be universal approximators.

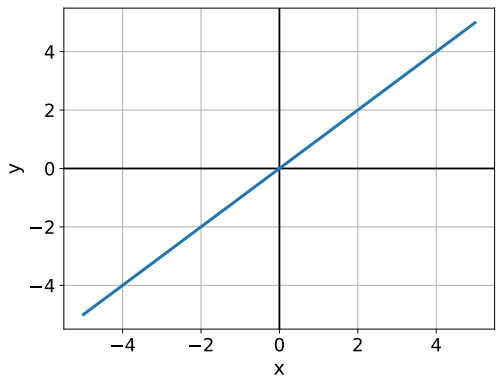
In the following sections most common activation functions are shown using their equations and plots of both the activation functions and the derivatives.

Linear activation

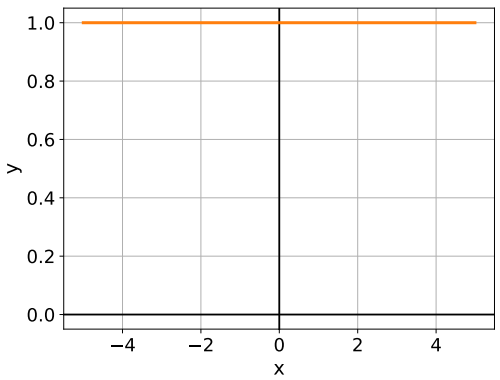
The linear activation function and its derivative is defined as shown here:

$$f(x) = ax \tag{A.1}$$

$$f'(x) = a \tag{A.2}$$



(a) Linear activation function.



(b) First derivative of linear activation function.

Figure A.1: Linear activation function (left) and its first derivative (right). $a = 1$ is used here. Therefore, the linear activation function is $f(x) = x$ and its derivative is $f'(x) = 1$.

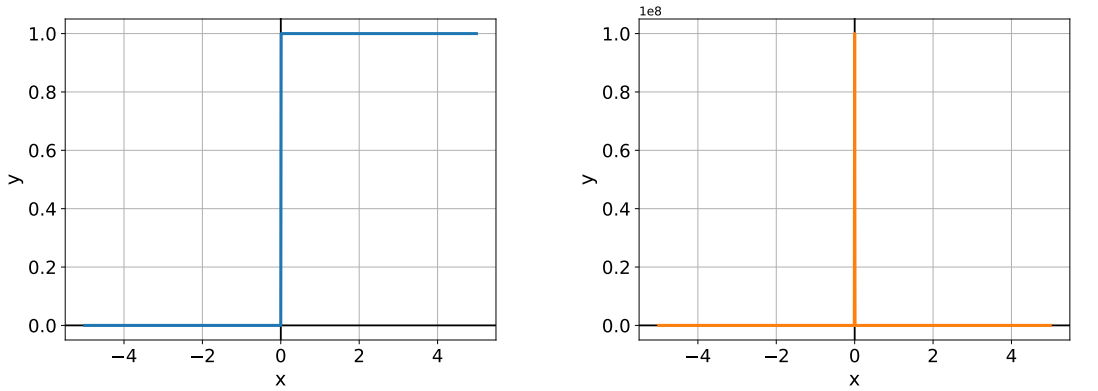
Binary step activation

The binary step activation function and its derivative is defined as shown here:

$$f(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases} \tag{A.3}$$

$$\delta(x) = \begin{cases} 0 & \text{if } x \neq 0 \\ +\infty & \text{if } x = 0 \end{cases} \tag{A.4}$$

Note that this function is not feasible for gradient-based method because of its derivative that is 0 everywhere except for $x = 0$ where it is infinite.



(a) Binary step activation function.

(b) First derivative of the binary step activation function

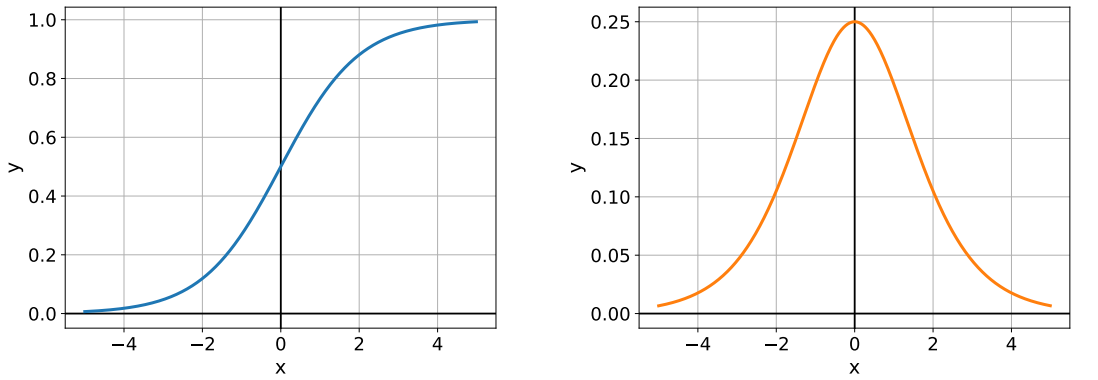
Figure A.2: Binary step activation function (left) and its derivative (right). Note that the impulse function is approximated here and in theory goes up to ∞.

Logistic activation

The logistic activation function and its derivative is defined as shown here:

$$\sigma(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{e^x+1} \tag{A.5}$$

$$\sigma'(x) = \sigma(x)\,(1-\sigma(x)) = \frac{e^x}{(1+e^x)^2} \tag{A.6}$$



(a) Logistic activation function.

(b) First derivative logistic activation function.

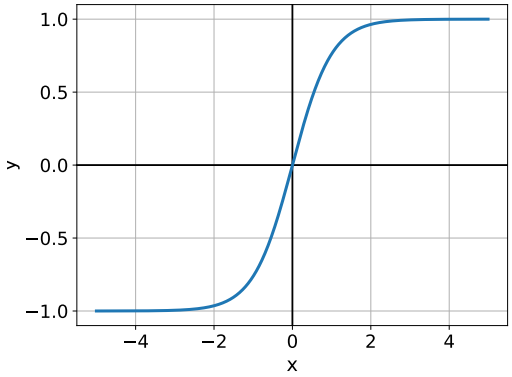
Figure A.3: Logistic activation function (left) sometimes referred to as sigmoid because of its S-shape and its first derivative (right).

Hyperbolic tangent

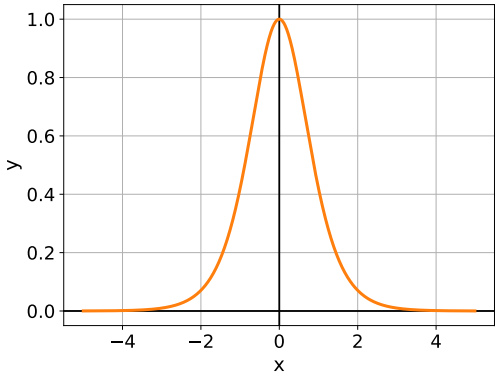
The hyperbolic tangent (tanh) activation function and its derivative is defined as shown here:

$$\tanh(x) = \frac{e^x - e^x}{e^x + e^{-x}} \tag{A.7}$$

$$\tanh'(x) = \left(1 - \tanh^2(x)\right) \tag{A.8}$$



(a) Hyperbolic tangent activation function



(b) First derivative of hyperbolic tangent function

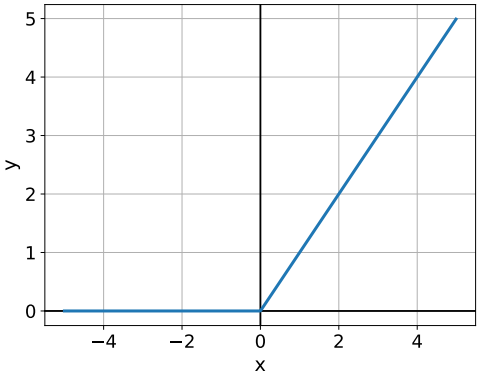
Figure A.4: Hyperbolic tangent (tanh) activation function (left) and its first derivative (right).

Rectified linear unit ReLU

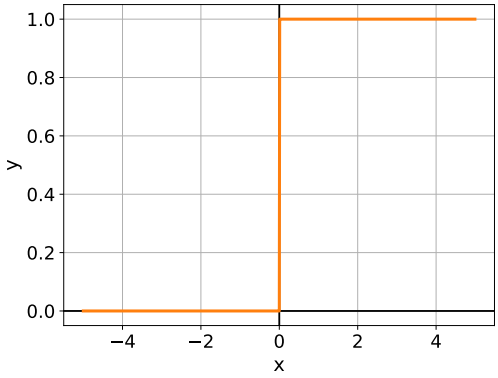
The rectified linear unitReLU activation function and its derivative is defined as shown here:

$$\text{ReLU}(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases} = \max\{0, x\} \tag{A.9}$$

$$\text{ReLU}'(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ 1 & \text{if } x > 0 \end{cases} \tag{A.10}$$



(a) Rectified linear unit (ReLU) activation function



(b) First derivative of the rectified linear unit.

Figure A.5: Rectified linear unit (ReLU) activation function (left) and its first derivative (right).

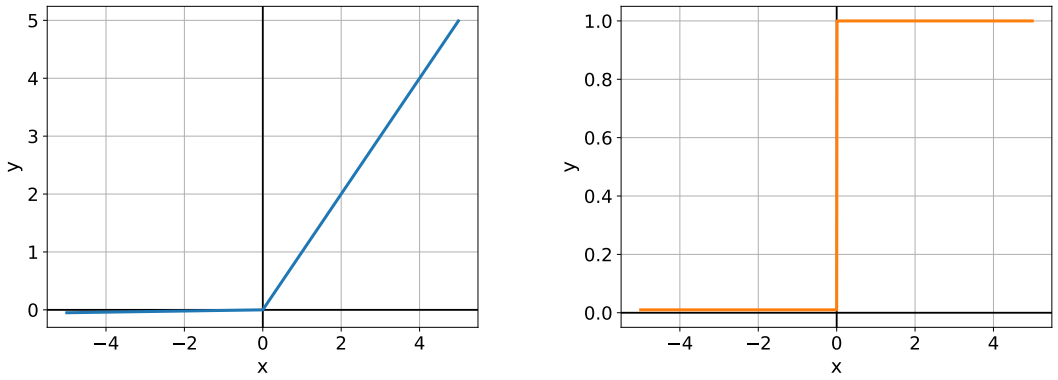
Leaky rectified linear unit (Leaky ReLU)

The leaky rectified linear unit (LReLU) activation function and its derivative is defined as shown here:

$$\text{LReLU}(x) = \begin{cases} 0.01x & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases} \tag{A.11}$$

$$\text{LReLU}'(x) = \begin{cases} 0.01 & \text{if } x \leq 0 \\ 1 & \text{if } x > 0 \end{cases} \tag{A.12}$$

When the 0.01 in the Leaky ReLU equation is parameterized this is called the *Parametric* ReLU. The paremter in the Leaky ReLU case is $\alpha = 0.01$.



(a) LReLU activation function.

(b) First derivative of the LReLU.

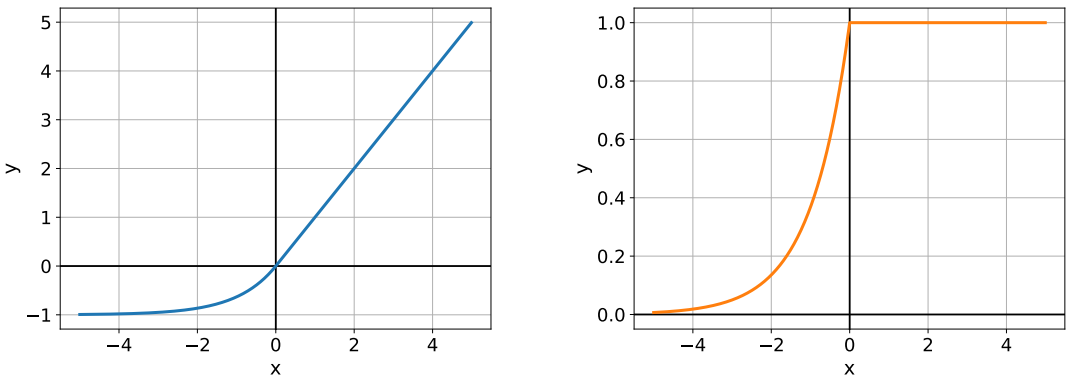
Figure A.6: Leaky rectified linear unit (LReLU) activation function (left) and its first derivative (right).

Exponential linear unit (ELU)

The **ELU!** (**ELU!**) activation function and its derivative is defined as shown here:

$$\text{ELU}(x) = \begin{cases} \alpha\,(e^x - 1) & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases} \tag{A.13}$$

$$\text{ELU}'(x) = \begin{cases} \alpha e^x & \text{if } x \leq 0 \\ 1 & \text{if } x > 0 \end{cases} \tag{A.14}$$



(a) ELU activation function.

(b) First derivative of the ELU activation function

Figure A.7: Exponential linear unit (ELU) activation function (left) and its first derivative (right).

Gaussian error linear unit (GELU)

A fairly new activation function is the GELU (Hendrycks and Gimpel, 2016) which is used in transformers used in NLP applications (Devlin et al., 2018; Dai et al., 2019). The GELU activation function and its derivative is defined as shown here:

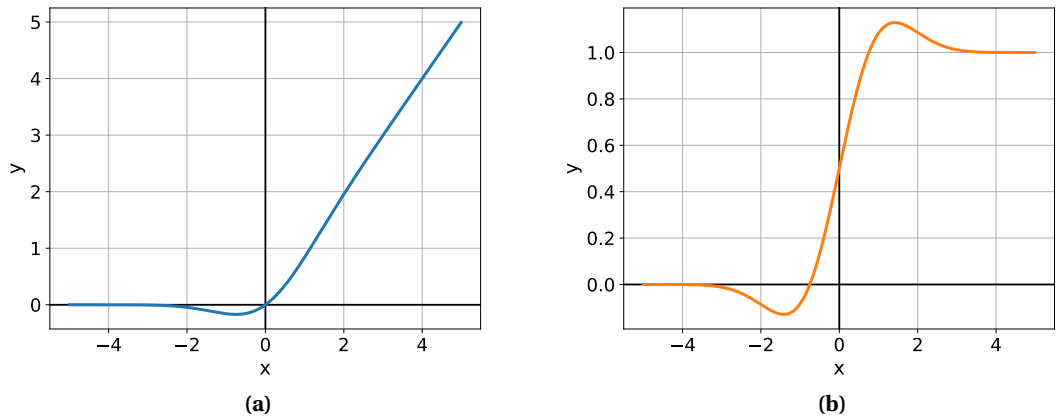
$$\text{GELU}(x) = xP(X \leq x) \tag{A.15}$$
$$= x\Phi(x) \tag{A.16}$$
$$= x \cdot \frac{1}{2}[1 + \text{erf}(x/\sqrt{2})] \tag{A.17}$$
can be approximated with: (A.18)
$$= \frac{x}{2}\left(1 + \tanh\left[\sqrt{2/\pi}\left(x + 0.044715x^3\right)\right]\right) \tag{A.19}$$
$$= x\sigma(1.702x) \tag{A.20}$$

Where $\Phi(x)$ is the standard Gaussian cumulative distribution function and erf() refers to the Gauss error function.

The derivative of the GELU is:

$$\text{GELU}'(x) = \Phi(x) + x\phi(x) \tag{A.21}$$
can be approximated with: (A.22)
$$= 0.5\tanh\left(0.0356774x^3 + 0.797885x\right) + \left(0.0535161x^3 + 0.398942x\right)\text{sech}^2\left(0.0356774x^3 + 0.797885x\right) + 0.5 \tag{A.23}$$

Where $\phi(x)$ is the Gaussian probability density function and $\Phi(x)$ is the standard Gaussian cumulative distribution function. The approximation was obtained by filling in equation A.19 into WolframAlpha.



(a) (b)

Figure A.8

Softmax function

Another frequently used activation function is the *softmax*. This function is not a function of a single fold x from its previous layer and therefore this cannot be graphed as the previous shown activation functions. The softmax is used to compute a probability distribution from a vector

of real numbers, i.e it produces output values ranging from 0 to 1, $0 \leq y \leq 1$, with the sum of probabilities equal to 1.

The softmax is an extension of the logistic activation function. The logistic activation function is used in binary classification problems while the softmax is used in multivariate classification problems. The softmax is mostly used in the output layer to give the highest probability to the target class. The softmax is defined as follows:

$$\sigma_i(x_i) = \frac{e^{(x_i)}}{\sum_j e^{(x_j)}} \tag{A.24}$$

The derivative of the softmax is:

$$\frac{\partial \sigma_i(x_i)}{\partial x_j} = \begin{cases} \sigma_i(x_i)(1-\sigma_i(x_i)) & \text{if } i = j \\ \sigma_j(x_j)\sigma_i(x_i) & \text{if } i \neq j \end{cases} \tag{A.25}$$

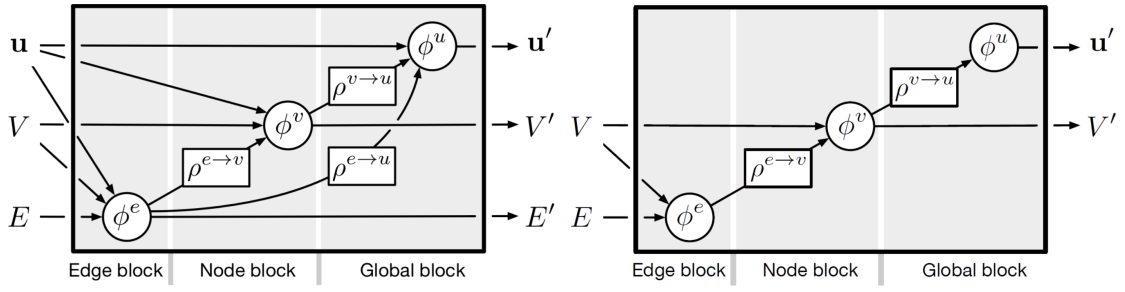
B GNN framework - GraphNets

In (Gilmer et al., 2017) the MPNN framework is presented in order to unify the different GNN approaches. In (Battaglia et al., 2018) the Graph Network (GN) framework (GraphNets) is introduced which is suited for relational reasoning over graph structured data. It is an attempt to generalize and extend previous GNN approaches and facilitate the construction of complex architectures from simple building blocks. In the GN framework the main processing unit is the *GN-block*. In essence, it is a set of composed functions that take a graph as input, applies the functions to the graph, and returns the processed graph as output. The functions inside the GN-block are typically NNs that operate on the features of the graph. The GraphNets library is released publicly (DeepMind, 2018).

The formulation of graphs in (Battaglia et al., 2018) is similar as the formulation presented in Section 2.6. Adhering to their notation, a graph is defined as a tuple $G = (\mathbf{u}, V, E)$, with N_v vertices and N_e edges. Here, the $\mathbf{u}$ represents graph-level attributes, $V = \{\mathbf{v_i}\}_{i=1:N_v}$ is the set of nodes where $\mathbf{v_i}$ represent the attributes of the i$^{\text{th}}$ node and $E = (\mathbf{e_k}, r_k, s_k)_{k=1:N_e}$ is the set of edges where $\mathbf{e}_k$ represents the attributes of the k$^{\text{th}}$ edge. The r_k and s_k are the indices of the two receiver and sender nodes connected by the k$^{\text{th}}$ edge.

The core unit of the GN framework is the full GN block shown in Figure B.1a. The full GN block contains three *update* functions (ϕ^e, ϕ^v, ϕ^u) and three *aggregation* functions $(\rho^{e\to v}, \rho^{e\to u}, \rho^{v\to u})$, shown in Equation B.1. The update functions take in a set of objects (e.g. edge attributes, node attributes and/or global attributes) with a fixed size representation, apply the same function to each of the elements in the set and output an updated representation also with a fixed size. The aggregation functions take a set of objects and create one fixed size representation for the entire set. The implementation of the update functions and aggregation functions is not fixed. This is where architectures can differ dependent on the application. However, the update functions are most often implemented as NNs, e.g. MLP. The aggregation operation is typically a permutation invariant reduction operator (e.g. elementwise summation, mean, maximum) in order to maintain permutation equivariance.

$$
\begin{aligned}
\mathbf{e}'_k &= \phi^e\left(\mathbf{e}_k, \mathbf{v}_{r_k}, \mathbf{v}_{s_k}, \mathbf{u}\right) & \bar{\mathbf{e}}'_i &= \rho^{e\to v}\left(E'_i\right) \\
\mathbf{v}'_i &= \phi^v\left(\bar{\mathbf{e}}'_i, \mathbf{v}_i, \mathbf{u}\right) & \bar{\mathbf{e}}' &= \rho^{e\to u}\left(E'\right) \\
\mathbf{u}' &= \phi^u\left(\bar{\mathbf{e}}', \bar{\mathbf{v}}', \mathbf{u}\right) & \bar{\mathbf{v}}' &= \rho^{v\to u}\left(V'\right)
\end{aligned}
\tag{B.1}
$$



(a) Full GN block (Battaglia et al., 2018).

(b) MPNN block (Battaglia et al., 2018).

Figure B.1: Two examples of GN building blocks. (a) A full GN block predicts node, edge and global attributes given a graph with node, edge and global attributes. (b) A MPNN (Gilmer et al., 2017) predicts node, edge and global attributes based on incoming node and edge attributes.

A GN block takes a graph as input. The computations start at edge level and proceed to node level and then to global level. In the GN block shown in Figure B.1a the *edge block* computes

one output for each edge, $\mathbf{e}'_k$. The set of updated edges that project to node i are then aggregated and used in the *node block* together with the node features $\mathbf{v}_i$ and global features $\mathbf{u}$ to update the node features $\mathbf{v}'_i$. In the *global block*, the edge level and node level features are all aggregated in order to update the global features. The output of the GN block is the graph with updated edge, node and global features, $G' = (\mathbf{u}', V', E')$. The entire process is described in Algorithm B.2. In this work, the constructed GN blocks are referred to as GNNs because the implementation of the update functions is a MLP.

Algorithm 1 Steps of computation in a full GN block.

function GRAPHNETWORK(E, V, $\mathbf{u}$)
  for $k \in \{1 \dots N^e\}$ **do**
    $\mathbf{e}'_k \leftarrow \phi^e(\mathbf{e}_k, \mathbf{v}_{r_k}, \mathbf{v}_{s_k}, \mathbf{u})$ ▷ 1. Compute updated edge attributes
  end for
  for $i \in \{1 \dots N^n\}$ **do**
    let $E'_i = \{(\mathbf{e}'_k, r_k, s_k)\}_{r_k=i,\ k=1:N^e}$
    $\bar{\mathbf{e}}'_i \leftarrow \rho^{e\to v}(E'_i)$ ▷ 2. Aggregate edge attributes per node
    $\mathbf{v}'_i \leftarrow \phi^v(\bar{\mathbf{e}}'_i, \mathbf{v}_i, \mathbf{u})$ ▷ 3. Compute updated node attributes
  end for
  let $V' = \{\mathbf{v}'\}_{i=1:N^v}$
  let $E' = \{(\mathbf{e}'_k, r_k, s_k)\}_{k=1:N^e}$
  $\bar{\mathbf{e}}' \leftarrow \rho^{e\to u}(E')$ ▷ 4. Aggregate edge attributes globally
  $\bar{\mathbf{v}}' \leftarrow \rho^{v\to u}(V')$ ▷ 5. Aggregate node attributes globally
  $\mathbf{u}' \leftarrow \phi^u(\bar{\mathbf{e}}', \bar{\mathbf{v}}', \mathbf{u})$ ▷ 6. Compute updated global attribute
  return $(E', V', \mathbf{u}')$
end function

Figure B.2: Steps of computation in a full GN block as described by (Battaglia et al., 2018).

Note that the GN block, as described, follows a sequence of steps. However, this order is not strictly enforced and it is possible to change the order of update functions. Furthermore, it is not strictly necessary to use all the components in the GN block. For example, when the MPNN is formulated in the GN framework it can be represented as is shown in Figure B.1b.

C Setup development environment

This appendix serves as tutorial to setup the development environment in PyCharm with TensorFlow including GPU support. It is worthwhile noting down how the development environment is setup because it took some time to get everything working correctly. This appendix covers how to install CUDA, cuDNN and TensorFlow on windows 10 within PyCharm.

Step 0 : Hardware check

The first thing to do is to check whether your graphics card is CUDA-capable. The graphics card you have can be checked by going to **Device Manager** -> **Display adapters**, see Figure C.1. If your card is not an NVIDIA card you can stop here because CUDA is not supported on other cards besides NVIDIA cards. If your NVIDIA card is listed here: https://developer.nvidia.com/cuda-gpus the GPU is CUDA-capable.

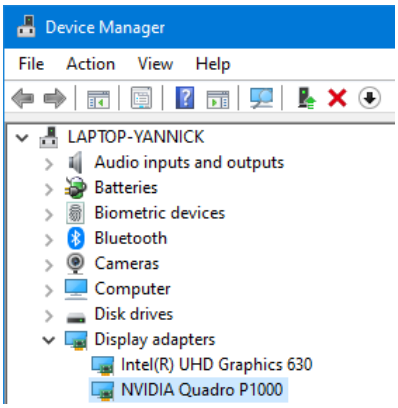


Figure C.1: Check your graphics card in **Device Manager**-> **Display adapters**.

Step 1 : Check the software you need

In this tutorial it will be assumed that Windows 10 is already installed. The additional software that is needed is listed here:

- Microsoft Visual Studio
- NVIDIA CUDA toolkit
- NVIDIA cuDNN
- Python
- TensorFlow (with GPU support)

Determine which TensorFlow (TF) version is desired, see Figure C.2. Then the corresponding Python, CUDA and cuDNN version can be determined using the list here: https://www.tensorflow.org/install/source#gpu. I went with the latest TensorFlow (TF)1 version, i.e. tensorflow_gpu-1.15.2. Therefore the CUDA 10.0 and cuDNN 7.4 were installed together with Python 3.7.7

GPU

Version	Python version	Compiler	Build tools	cuDNN	CUDA
tensorflow-2.4.0	3.6-3.8	GCC 7.3.1	Bazel 3.1.0	8.0	11.0
tensorflow-2.3.0	3.5-3.8	GCC 7.3.1	Bazel 3.1.0	7.6	10.1
tensorflow-2.2.0	3.5-3.8	GCC 7.3.1	Bazel 2.0.0	7.6	10.1
tensorflow-2.1.0	2.7, 3.5-3.7	GCC 7.3.1	Bazel 0.27.1	7.6	10.1
tensorflow-2.0.0	2.7, 3.3-3.7	GCC 7.3.1	Bazel 0.26.1	7.4	10.0
tensorflow_gpu-1.15.0	2.7, 3.3-3.7	GCC 7.3.1	Bazel 0.26.1	7.4	10.0
tensorflow_gpu-1.14.0	2.7, 3.3-3.7	GCC 4.8	Bazel 0.24.1	7.4	10.0

Figure C.2

The software and their versions I used in this thesis, and will be assumed during this tutorial are listed here:

- Windows 10 - version 1909
- Python 3.7.7
- TensorFlow 1.15.2 (GPU support)
- NVIDIA CUDA 10.0
- NVIDIA cuDNN 7.4
- PyCharm 2020.3
- Microsoft Visual Studio 2017

Step 2 : Download and install Microsoft Visual Studio

The most recent version of Microsoft Visual Studio can be downloaded here: https://visualstudio.microsoft.com/vs/express/. Note however that Microsoft Visual Studio sometimes comes pre-installed on you pc. So check whether you already have Visual Studio installed.

Run the downloaded executable and installation should be pretty straight forward. During the installation you will be asked whether *'you want to continue without workloads?'*. I pressed 'continue' because I will not use Visual Studio after the CUDA installation.

After the installation of Visual Studio make sure you do a reboot before continuing with the CUDA installation.

Step 3 : Download and install CUDA

The desired CUDA version can be downloaded from here https://developer.nvidia.com/Cuda-Toolkit-archive, all version are listed here. I downloaded the CUDA 10.0 version (exe local), see Figure C.3. CUDA 10.0 comes with a base installation. Sometimes there can be additional patches as is the case for CUDA version 9.0. In that case download the base installation and all patches. First install the base installation. After succesful installation of the base installer, install all the patches in numerical order.

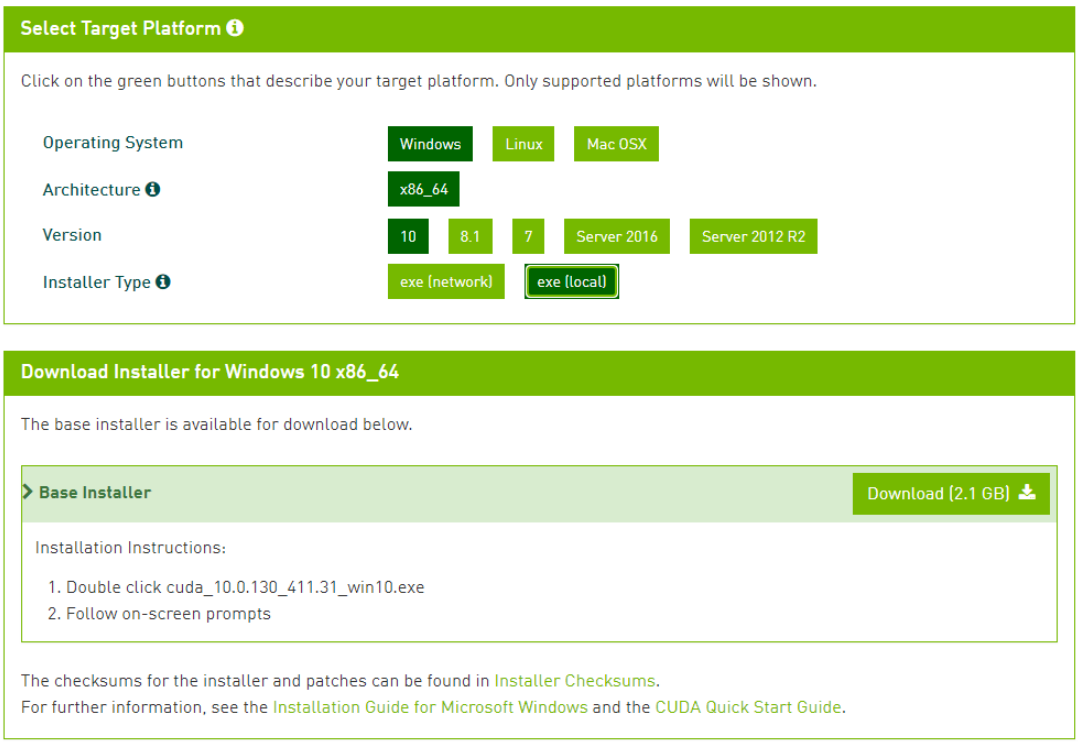


Figure C.3

Run the base installation executable which has just been downloaded. Extract the files to a desired location, I recommend to use the default location, see Figure C.4. After the extraction process has finished the NVIDIA CUDA installer will start. I used the express installation option. During the installation the installer will ask for installation locations, I recommend to use the default location.

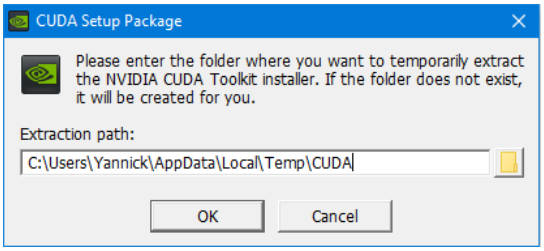


Figure C.4

After the installation of the base file, install the patches (if applicable to your CUDA version). To verify the installation go to:

'C:\ProgramData\NVIDIA Corporation\CUDA Samples\v10.0\5_Simulations\nbody'

This is the default location specified during the installation process. If a different location was specifie during installation go to this path you specified. Open the Visual Studio solution corresponding to your version of Visual Studio you have installed, see Figure C.5.

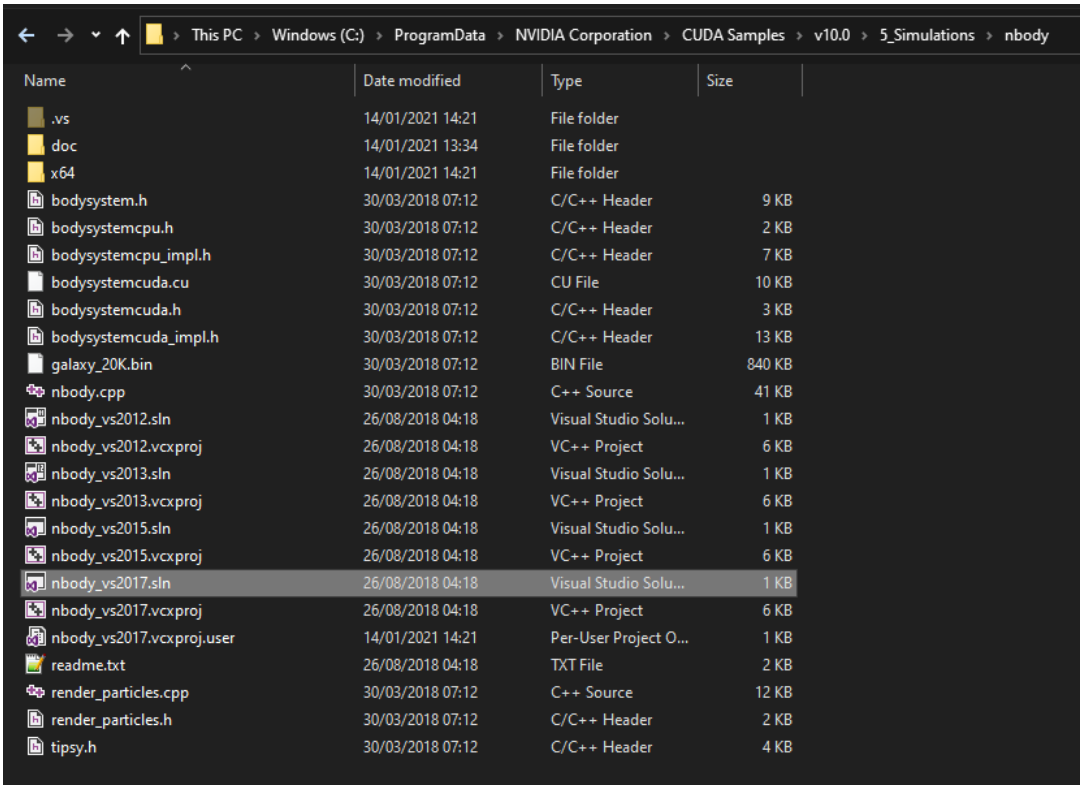


Figure C.5: Visual Studio solution. Open the solution corresponding to the installed Visual Studio version.

Next, build the solution by going to the "Build" menu within Visual Studio and clicking "Build Solution", see Figure C.6. Whether you did everything correct until this point will be indicated by a 'succeeded' build, see Figure C.6.

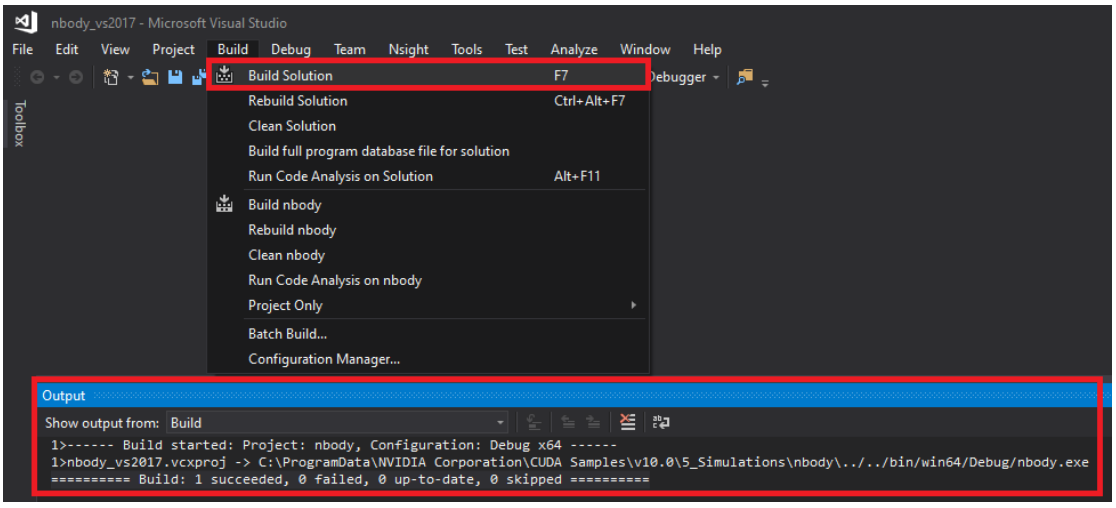


Figure C.6: Build the solution to verify correct installation of CUDA.

After successful build, go to:

'C:\ProgramData\NVIDIA Corporation\CUDA Samples\v10.0\bin\win64\Debug'

and run the executable 'nbody.exe', see Figure C.7

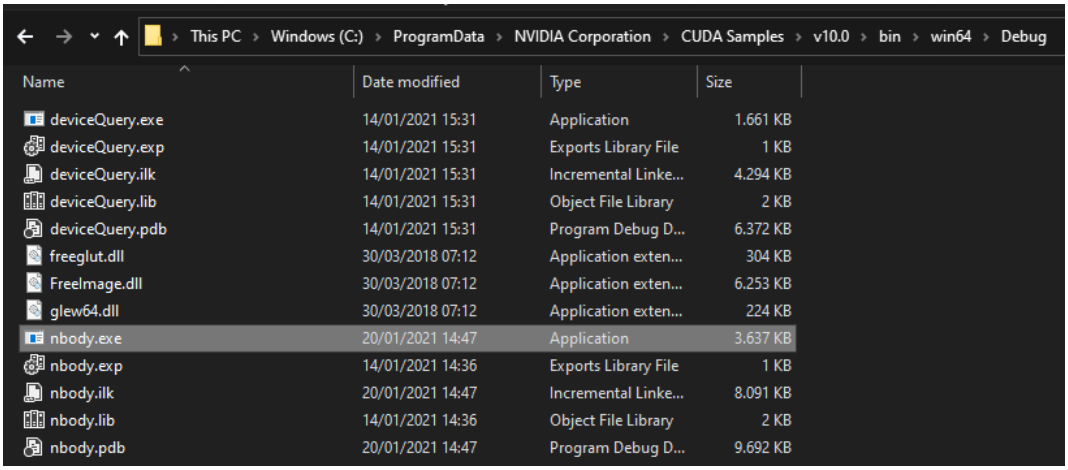


Figure C.7: Run the build solution 'nbody.exe'.

If you were able to build and run the executable CUDA is installed correctly. When this is not the case verify you installed everything correctly. For troubleshooting and more info see section C. more info

Step 4 : Download and install cuDNN

In **step 1** the we determined the corresponding cuDNN version. In this tutorial we installed CUDA 10.0 with the aim of installing TensorFlow 1.15 with GPU support. Therefore, we install cuDNN version 7.4. This can be downloaded from: `https://developer.nvidia.com/rdp/cudnn-download`. Note that an account for the NVIDIA Developer program is required. First, register for a free account. Then login and download the correct version of cuDNN. Older versions can be downloaded from the archive, see Figure C.8

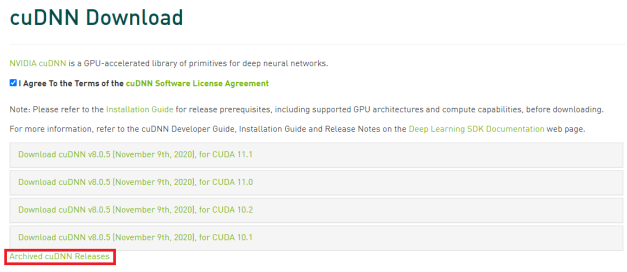


Figure C.8: Download the required cuDNN version from the archive.

After downloading, unzip the files. After unzipping the relevant files are located at:

'<download_path>\cuda\'

Where the 'download_path' is user specific. The folders 'bin', 'include' and 'lib' have to be copied to the installation folder of CUDA: **'C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v10.0'**

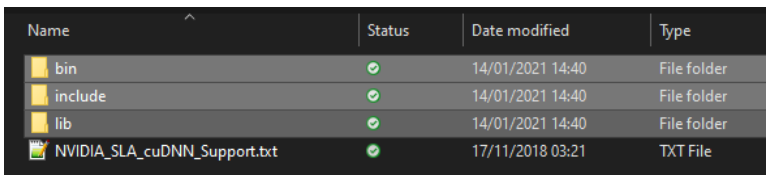


Figure C.9: cuDNN files to copy to the CUDA installation folder.

Check that the paths shown in Figure C.10 are correctly set in the environment variables.

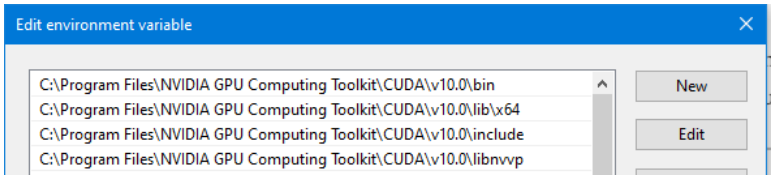


Figure C.10: Paths that need to be set under 'Environment variables -> 'System variables' -> 'Path'.

Now we can run the command 'nvidia-smi' in command prompt to see what the NVIDIA graphics card is doing, see Figure C.11. When the command is not recognized make sure the following path: **C:\Program Files\NVIDIA Corporation\NVSMI** is also included under 'Environment variables -> 'System variables' -> 'Path'.

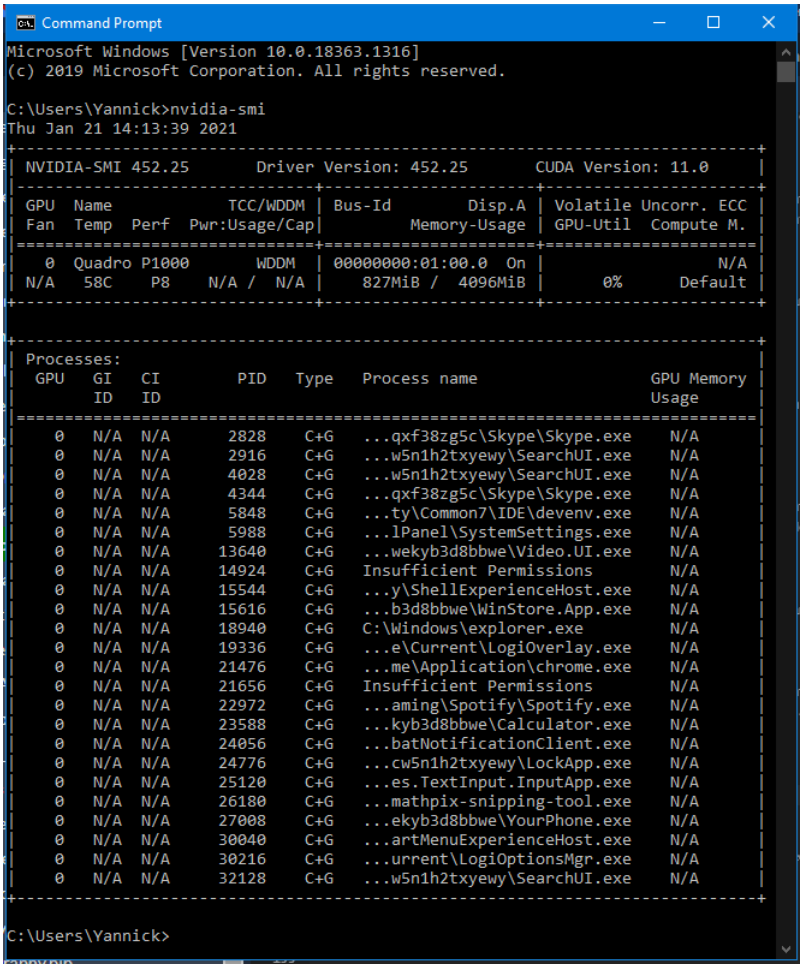


Figure C.11: Start command prompt and use command: 'nvidia-smi' to monitor you NVIDIA GPU device.

Step 5 : Make you virtual environment in PyCharm.

The instructions to install both Python and PyCharm are skipped. This is straight forward and comes down to just running the executables.

What is left to do is setup our venv inside PyCharm. Before doing so, make sure you do a reboot first! After the reboot startup PyCharm and make a new venv for you project, see Figure C.12

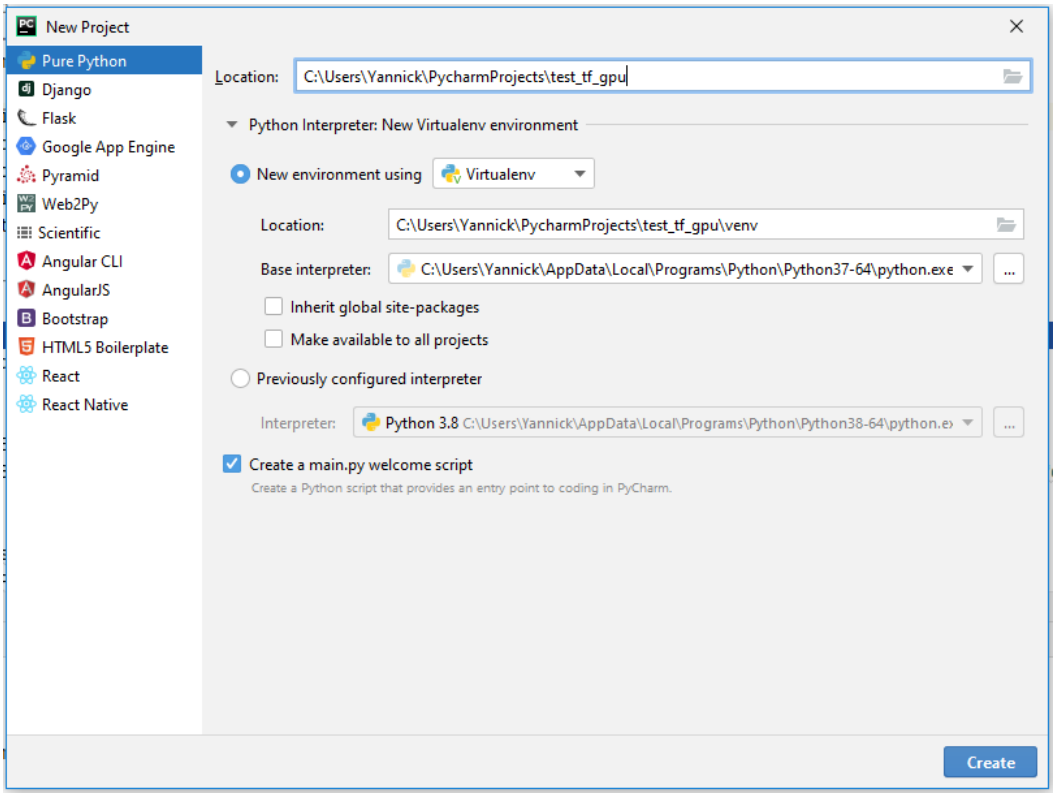


Figure C.12: Make a new venv for you project.

After creating the new vitrual environment, install the desired TensorFlow package version, see Figure C.13, you can specify the desired tensorflow version, see figure C.14

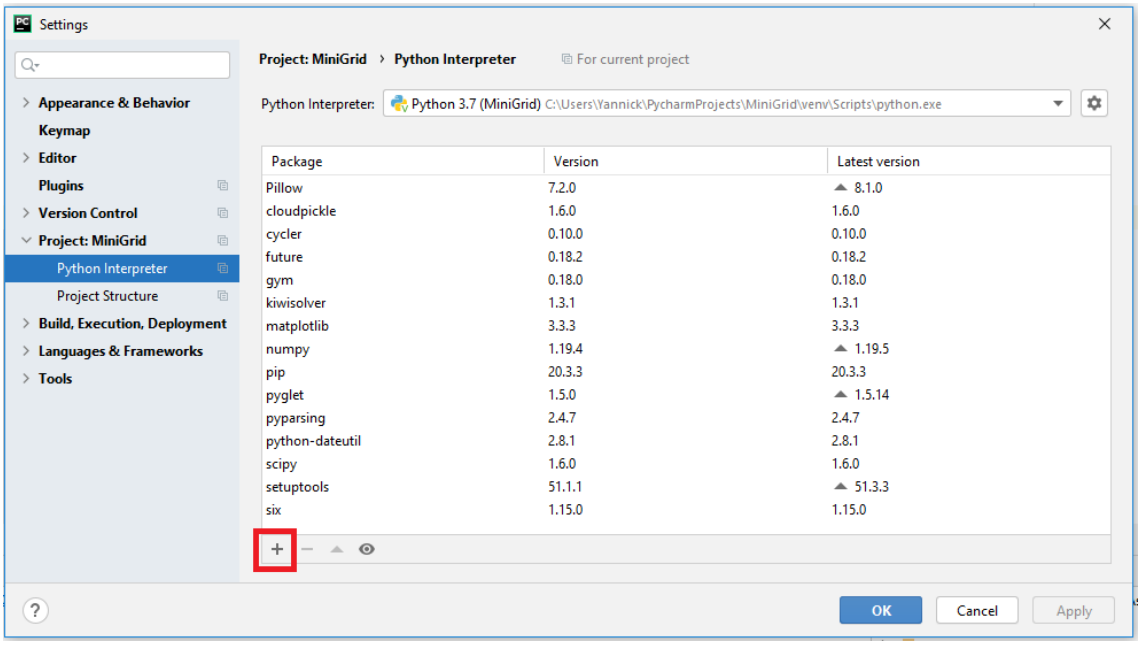


Figure C.13: Search for the tensorflow-gpu package.

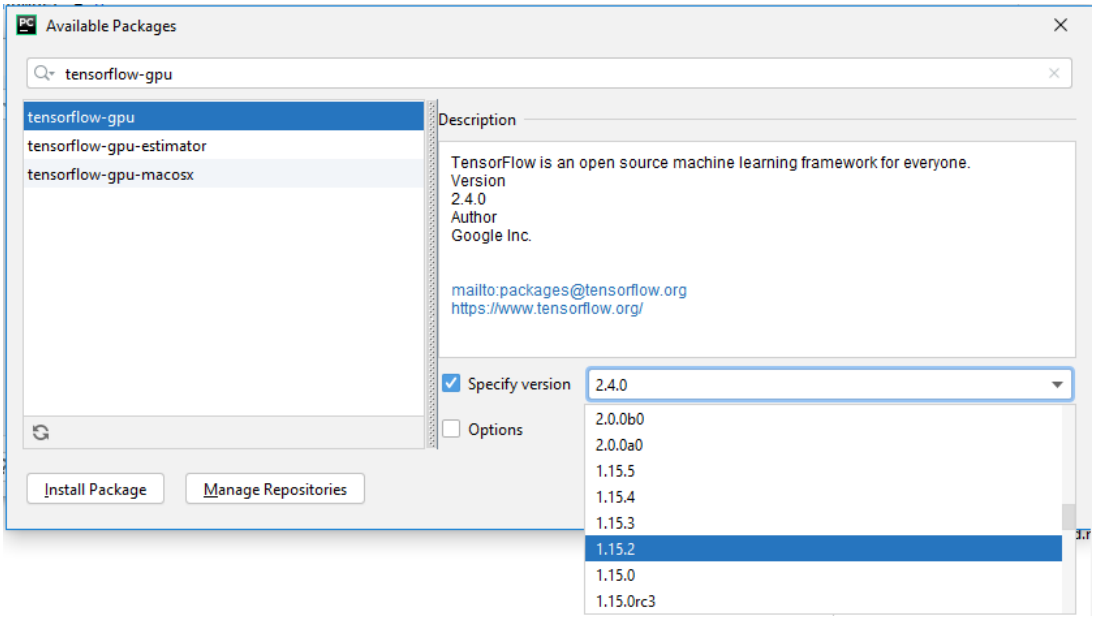


Figure C.14: Specify the desired tensorflow version.

Step 6 : Verify the installation

The last step is to check whether everything is installed correctly. Run the following piece of code, it should print 2 times 'True'.

```
import tensorflow as tf

print(tf.test.is_built_with_cuda())
print(tf.test.is_gpu_available(cuda_only=False,
 ↪  min_cuda_compute_capability=None))
```

If this is the case the installation was correct and you can make use of CUDA from within PyCharm. If this is not the case, see the next section for resources for more info and troubleshooting resources.

Important is that the venv is created with the TF package after CUDA and cuDNN were installed correctly!

More info

For more info or troubleshooting the following resource might be helpful: https://docs.nvidia.com/cuda/cuda-installation-guide-microsoft-windows/index.html

D Graph learning libraries

The Python programming language, released in 1991 (van Rossum, 2009), is one of the most popular programming languages today (PYPL, 2021). The Python programming language is oriented towards developers, it is highly productive, simple and powerful. Furthermore, the syntax is clean and there are many libraries for ML available. Hence, the natural choice for a programming language in AI research is Python. Therefore, next only graph learning libraries are considered that can be used in Python.

The field of deep learning on graphs is fairly new. The tools available are primarily oriented towards well structured data on a 1D or 2D grid. In (Bronstein et al., 2016) it is mentioned that computational frameworks for efficient processing of graph structured data is an important future direction in the graph learning field. There are different libraries emerging to facilitate graph learning. However, there is not one widely used library. Moreover, code released accompanying research papers often implements the GNN architecture without the use of any specific graph learning library, i.e. the GNN architecture is implemented directly into existing tensor libraries using custom graph functions from the authors. As a result to facilitate faster development of GNN architectures, open source deep learning libraries are being developed that work with existing tensor libraries such as PyTorch (Paszke et al., 2019), TensorFlow (Abadi et al., 2016), Apache MXNet (Chen et al., 2015). The common principle behind all libraries mentioned below is that they rely on the message-passing principle (Gilmer et al., 2017) in order to perform operations on graphs.

Pytorch Geometric

PyTorch Geometric (PyG) (Fey and Lenssen, 2019) is a deep learning library for graphs build upon the PyTorch library. It recently became part of the PyTorch Ecosystem Tools. PyG comes with many re-implementations of existing models and is well integrated with benchmark data sets. Therefore, it is particularly useful for reproducing results from previous works. Furthermore, PyG is well documented and the online community is active.

Graph Nets

DeepMind's library for building GNNs is the Graph Nets library (Battaglia et al., 2018) and it is build on TensorFlow and Sonnet. The Graph Nets library was released as a companion to (Battaglia et al., 2018) to demonstrate the practical use of GNNs. It is still being updated and recently received compatibility with TensorFlow 2. The online community is fairly small but there are extensive coding tutorials released (DeepMind, 2018).

Spektral

Another recent Python library for deep learning on graphs is Spektral (Grattarola and Alippi, 2020). This library is based on the Keras API and Tensorflow 2. Spektral has implemented most popular GNN layers and is focused on user-friendliness. Therefore, Spektral is well suited for fast implementation of existing GNN architectures.

Deep Graph Library

Deep Graph Library (DGL) (Wang et al., 2019) is introduced in order to provide an efficient multi-platform API. The API can be used using different backend frameworks (e.g. PyTorch, MXNet and TensorFlow). At the time of writing the DGL for TensorFlow is still in experimental phase.

Jraph

Jraph (DeepMind, 2020) is a library for working with graph neural networks in JAX (Bradbury et al., 2018). Jraph is the Graph Nets library (mentioned above) for JAX. JAX can automatically differentiate Python and NumPy code, this makes common operations in ML trivial. Jraph was released recently at the end of 2020.

Euler and NeuGraph

Other Python libraries for learning on graphs are Euler (Alibaba, 2019) and NeuGraph (Ma et al., 2019). However, the Euler documentation is mainly in Chinese. NeuGraph is a framework that facilitates efficient and scalable parallel neural network computation on graphs. Unfortunately, the code for NeuGraph is not open source at this moment.

E Reproduction results C-SWM paper

The end-to-end architecture presented in (Kipf et al., 2020b) is implemented in TensorFlow using the Graph Nets library (Battaglia et al., 2018). In order to validate that the architecture is implemented correctly, the results are compared against the results produced by the code released by the authors (Kipf et al., 2020a).

The goal of the C-SWM architecture is to learn an object-oriented abstraction of a particular observation or environment state. Additionally, an action-conditioned transition model of the environment is learned that takes the object-oriented abstraction as input. The object-oriented abstraction is realised by encoding the observation as a graph. The transition model is implemented using GNN.

An off-policy setting is considered, the model is trained solely using an experience buffer generated using a random policy. The experience buffer contains tuples of states $s_t \in \mathcal{S}$, actions $a_t \in \mathcal{A}$ and next states $s_{t+1} \in \mathcal{S}$. The goal is to learn a latent representation $z_t \in \mathcal{Z}$ for the environment states $s_t \in \mathcal{S}$ that discards all the irrelevant information that is not needed for predicting the next $z_{t+1} \in \mathcal{Z}$. Simultaneously, an encoder $E : \mathcal{S} \rightarrow \mathcal{Z}$ which maps the environment state to the latent state and a transition model $T : \mathcal{Z} \times \mathcal{A} \rightarrow \mathcal{Z}$ that operates on the latent state.

The goal of the C-SWM architecture is to take into account the compositional nature of the observations. Hence, a relational an object-oriented model of the environment is learned that operates on a factored abstract state space $\mathcal{Z} = \mathcal{Z}_1 \times \mathcal{Z}_2 \times ... \times \mathcal{Z}_K$, where K is a predefined constant which indicates the available object slots. The same assumption is made for the action space, i.e. an object-factorized action space is assumed $\mathcal{A} = \mathcal{A}_1 \times \mathcal{A}_2 \times ... \times \mathcal{A}_K$. The object factorization ensures that each object is represented independently and allows for sharing model parameters for the different objects in the transition model. This introduces a strong inductive bias that facilitates generalization to novel scenes, learning and object discovery. The C-SWM architecture is presented in Figure E.1.

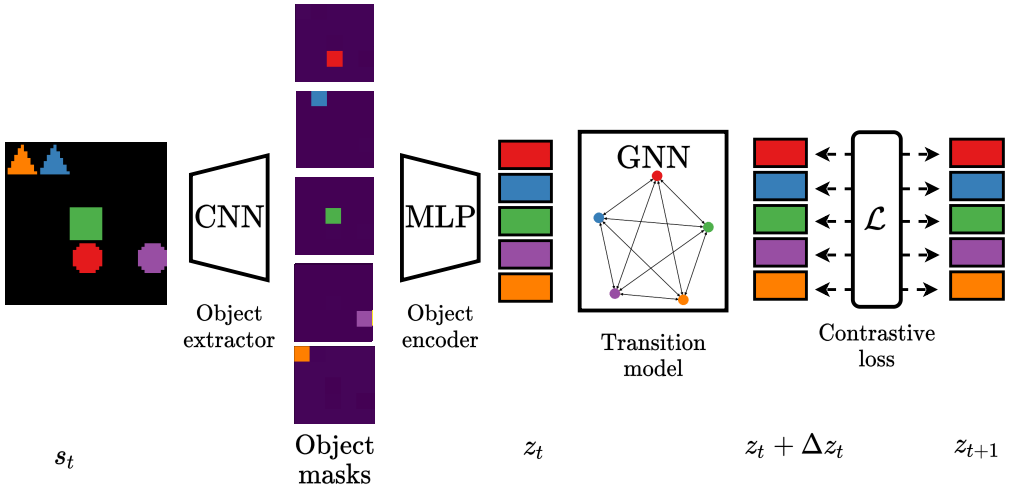


Figure E.1: The C-SWM architecture as presented in (Kipf et al., 2020b). Figure is inspired on Figure 1 in (Kipf et al., 2020b). The C-SWM architecture is composed of a CNN object extractor, a MLP object encoder, a GNN transition model and an object factorized contrastive loss. Colored blocks in the object masks and latent states are used to indicate the correspondence with the objects in the observation image.

The first part of the architecture consists of the object extractor and object encoder. The object extractor is a CNN and the object encoder is a MLP. The CNN takes as input the observation

as an image and has K features maps in the last layer. Each feature map is a mask for a specific object after training the model is finished. For simple problems, single feature maps suffice but it can be extended to multiple feature maps per object slot. The MLP object encoder maps the feature maps to a latent space $z_t^k \in \mathcal{Z}_k$, where $\mathcal{Z}_k = \mathbb{R}^D$ where D is a hyperparameter.

The second part contains the transition model which is implemented as a GNN. This allows for modeling pairwise interactions between objects. The GNN architecture makes the model invariant to the ordering in which the objects are presented to the network. The latent state representations after the object encoder are used as node features in a fully connected graph without self loops, see Figure E.3a. The actions are encoded per object as one-hot encoded vectors. The transition model takes the latent state representation and actions for each object as input and puts this in a graph representation such that the transition model can be implemented as a GNN, see Equation E.1 from (Kipf et al., 2020b).

$$\Delta z_t = T\left(z_t, a_t\right) = \mathrm{GNN}\left(\left\{\left(z_t^k, a_t^k\right)\right\}_{k=1}^{K}\right) \tag{E.1}$$

The GNN as presented in (Kipf et al., 2020b) consists of a node update function f_{node} and edge update function f_{edge} and are implemented as MLPs with shared parameters. The message passing scheme is shown in Equation E.2 from (Kipf et al., 2020b).

$$\begin{aligned} e_t^{(i,j)} &= f_{\text{edge}}\left(\left[z_t^i, z_t^j\right]\right) \\ \Delta z_t^j &= f_{\text{node}}\left(\left[z_t^j, a_t^j, \sum_{i \neq j} e_t^{(i,j)}\right]\right) \end{aligned} \tag{E.2}$$

In the GraphNets framework a similar scheme can be realised by using the one-hot action encoding per object as a global and putting the actions on the nodes after the edge block in the node block update. The GraphNets formulation is shown in Figure E.2. Where the latent state representation is V and the actions per object are taken as global feature $\mathbf{u}$. The latent state transition is V'.

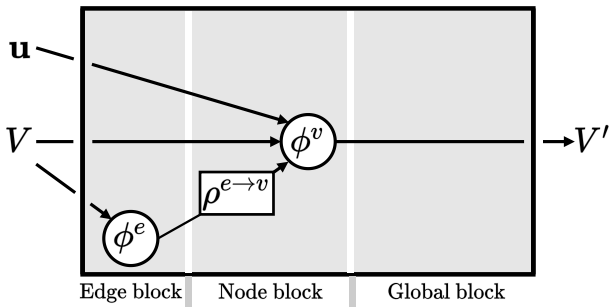


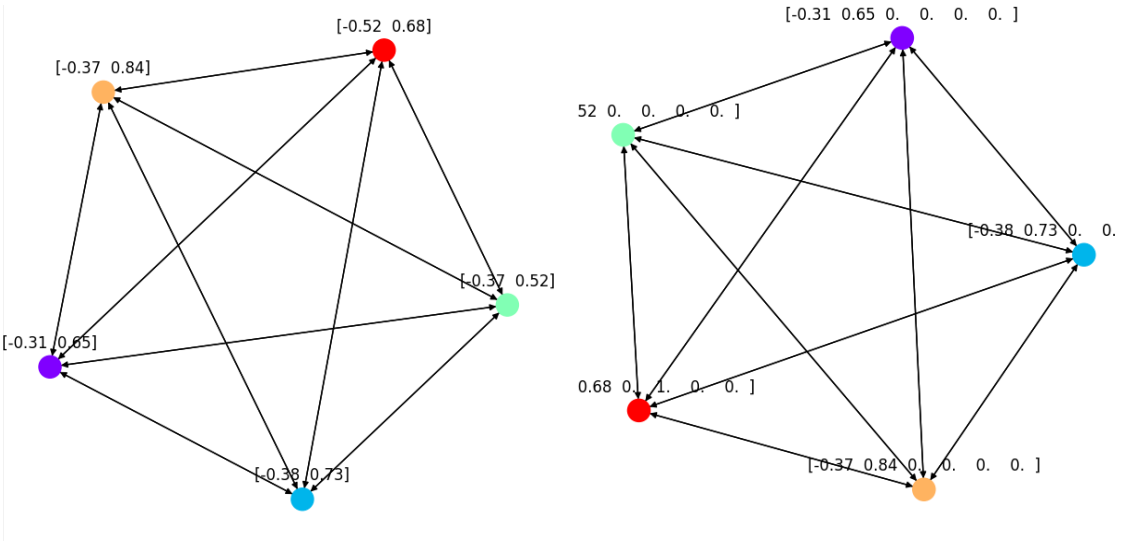
Figure E.2

In equations this is (conform the notation in (Battaglia et al., 2018)):

$$\begin{aligned} \mathbf{e}_k' &= \phi^e\left(\mathbf{v}_{r_k}, \mathbf{v}_{s_k}\right) = f^e\left(\mathbf{v}_{r_k}, \mathbf{v}_{s_k}\right) \\ \bar{\mathbf{e}}_i' &= \rho^{e\to u}\left(E'\right) = \sum_k \mathbf{e}_k' \\ \mathbf{v}_i' &= \phi^v\left(\bar{\mathbf{e}}_i', \mathbf{v}_i, \mathbf{u}\right) = f^v\left(\bar{\mathbf{e}}_i', \mathbf{v}_i, \mathbf{u}\right) \end{aligned} \tag{E.3}$$

It must be noted that encoding the action as a global in 1 vector is differently than is done here. Here the global feature is a 5×4 matrix, such that each node can concatenate a one-hot encode action to the node features.

The abstract states encoded as a graph, is shown in Figure E.3a

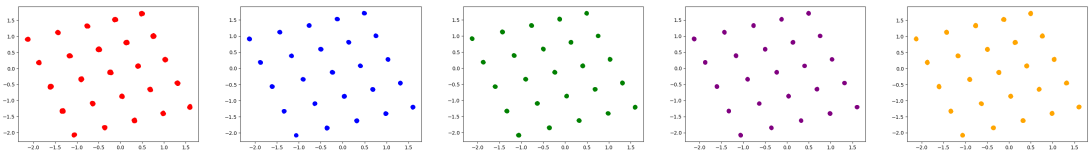


(a) Latent state representation z_t^k

(b) Latent state representation with the actions $\left[z_t^k, a_t^k\right]$

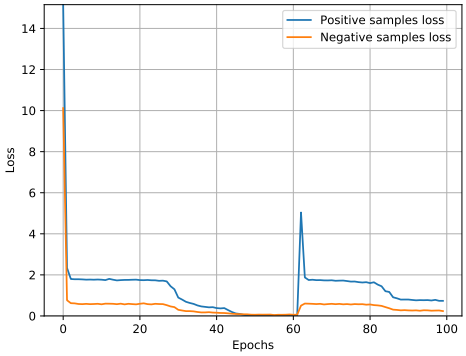
Figure E.3: The graph representations, a) the graph representations with abstract states $z_t^k \in \mathbb{R}^D$ on the nodes, and b) the graph representation with abstract states and one-hot encoded action $\left[z_t^k, a_t^k\right]$ on the nodes. Note that the colors of the nodes have no specific meaning in this figure. Colors are used to distinguish nodes in different plots because the structure of a graph is not defined. The node labels for Figure a) are: 0: '[-0.31 0.65]', 1: '[-0.38 0.73]', 2: '[-0.37 0.52]', 3: '[-0.37 0.84]', 4: '[-0.52 0.68]', the node labels for Figure b) are: 0: '[-0.31 0.65 0. 0. 0. 0.]', 1: '[-0.38 0.73 0. 0. 0. 0.]', 2: '[-0.37 0.52 0. 0. 0. 0.]', 3: '[-0.37 0.84 0. 0. 0. 0.]', 4: '[-0.52 0.68 0. 1. 0. 0.]'. Self loops are excluded.

The same contrastive loss function as presented in (Kipf et al., 2020b) is used. The TF (with GraphNets) implementation is trained for a 100 epochs on a training set with 500 episodes containing 100 steps in the environment. After training is finished, the obtained encoded states for each object slot are put nicely on a grid as is shown in Figure E.4. The model was trained on a smaller training set (500 episodes opposed to 1000 episodes) in order to speed up the learning process. A similar encoding for the objects was obtained while training the model implemented by the authors on the same dataset.

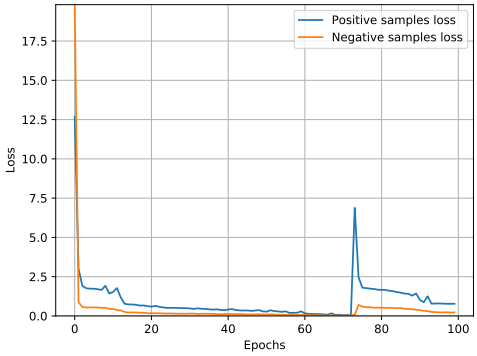


(a) Object slot 1. **(b)** Object slot 2. **(c)** Object slot 3. **(d)** Object slot 4. **(e)** Object slot 5.

Figure E.4: The learned object encodings for all the samples in the training buffer. Note that the plot is continuous and that the discrete nature of the environment is learned without supervision.



(a) Training loss using the source code provided by the authors (Kipf et al., 2020a).



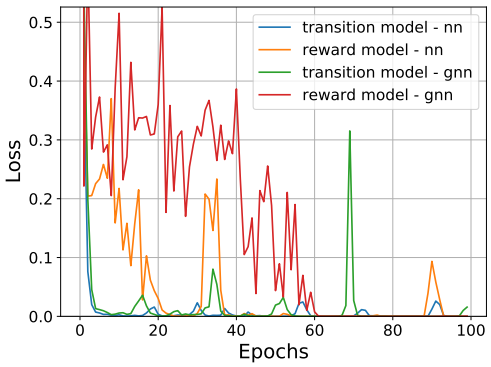
(b) Training loss using the TF with Graph Nets implementation.

Figure E.5: Training losses for a) the implementation released by the authors, b) the implementation in TF using GraphNets. The same training set of 500 episodes with 100 steps per episode in the environment is used. Due to the random sampling the training curves slightly differ but overall they have the same shape.

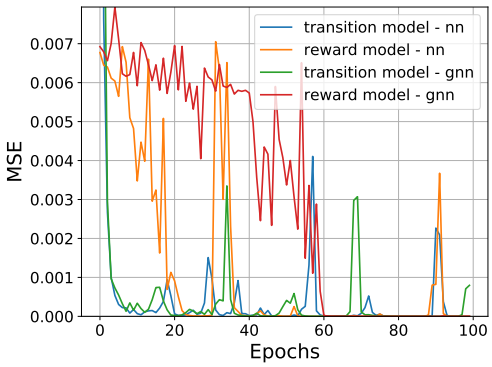
F Additional figures

F.1 Train and test errors - 5x5 maze - fixed configuration

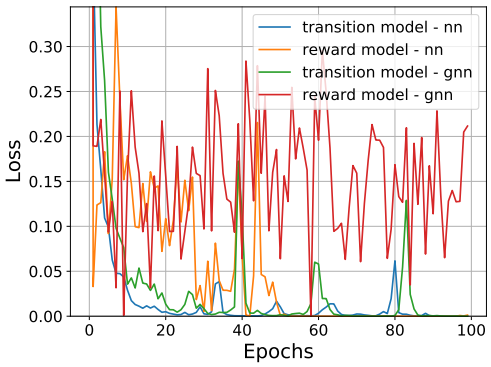
It must be noted that the reward model implemented using a GNN is performing worse as can be concluded from the figures shown in Figure F.1. Furthermore, in Figure 6.13c it can be observed that the maximum value it can have is -1 due to the reward model that maps everything to -1. The reward function is sparse and due to the aggregation function in the GNN and the unstructured ordering of the nodes, it might not distinguish between bumping into an obstacle or a target position. Therefore, the reward model learns to map everything to -1 because this is the reward that is most often seen.



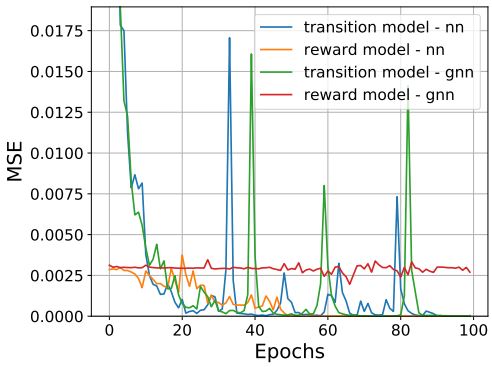
(a) Train loss empty 5x5 maze.



(b) Test error empty 5x5 maze.



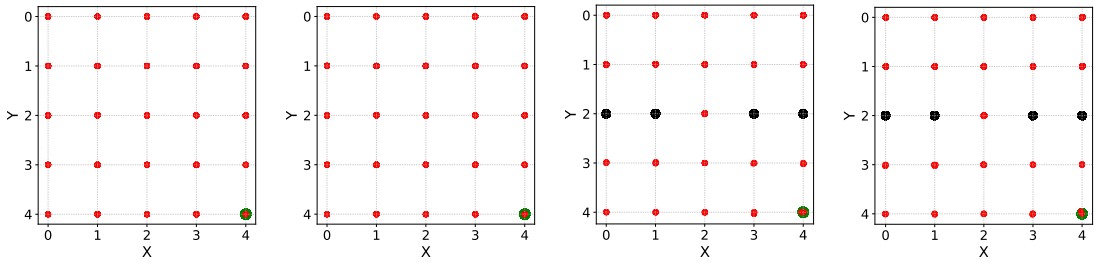
(c) Train loss 2 room 5x5 maze (fixed configuration)



(d) Test error 2 room 5x5 maze (fixed configuration)

Figure F.1: Train and test curves for both transition and reward model implemented using a GNN and NN for the 5x5 mazes. The maze is kept constant during training and tested on the same maze.

F.2 Next state prediction - 5x5 maze - fixed configuration



(a) No room 5x5 maze, GNN transition model.

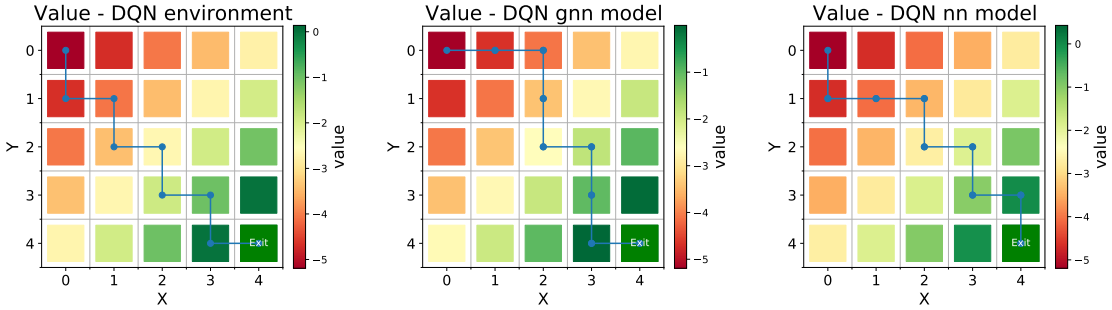
(b) No room 5x5 maze, NN transition model.

(c) 2 room 5x5 maze, GNN transition model.

(d) 2 room 5x5 maze, NN transition model.

Figure F.2: Next state predictions for all states in the test set for the 5x5 mazes. Red is used for the position of the agent, green for the target position and black for the obstacle position.

F.3 Value Functions - 5x5 maze - fixed configuration

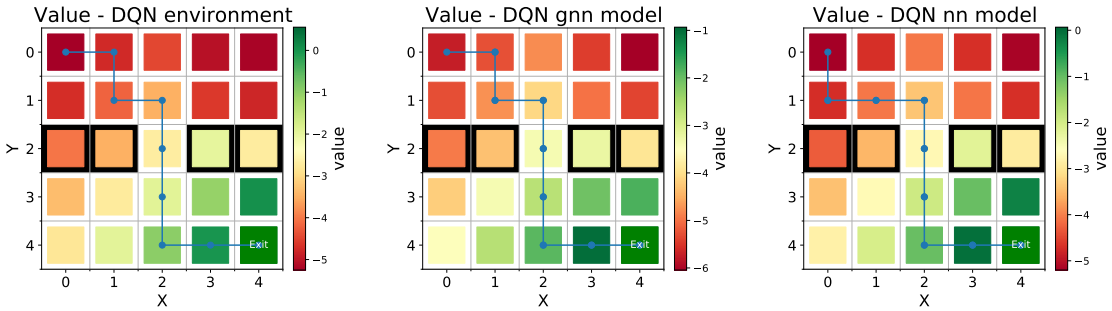


(a) DQN agent, access environment

(b) DQN agent, GNN model

(c) DQN agent, NN model

Figure F.3: Value function calculated using DQN agents trained for the empty 5x5 maze: a) with direct access to the environment simulator, b) using the trained GNN model, c) using the trained NN model. The blue line indicates the greedy policy that is obtained when acting greedily with respect to the obtained value function indicated by the colored tiles.



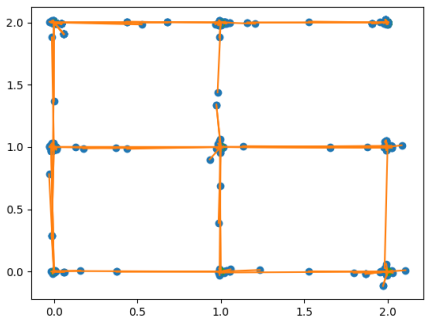
(a) DQN agent, access environment

(b) DQN agent, GNN model

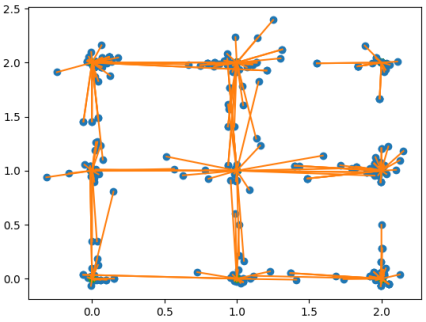
(c) DQN agent, NN model

Figure F.4: Value function calculated using DQN agents trained for the 5x5 maze with 2 rooms: a) with direct access to the environment simulator, b) using the trained GNN model, c) using the trained NN model. The blue line indicates the greedy policy that is obtained when acting greedily with respect to the obtained value function indicated by the colored tiles.

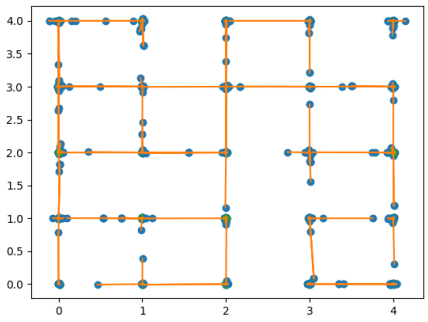
F.4 Predicted state error - random maze



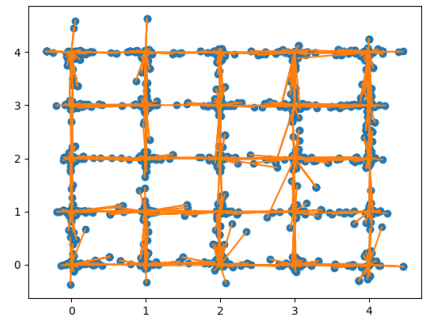
(a) Predicted states, 3x3 maze 2 rooms, GNN model



(b) Predicted states, 3x3 maze 2 rooms, NN model



(c) Predicted states, 5x5 maze 2 rooms, GNN model



(d) Predicted states, 5x5 maze 2 rooms, NN model

Figure F.5: The true state are connect to their corresponding prediction from the learned environments models.

Bibliography

Abadi, M., P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. A. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu and X. Zhang (2016), TensorFlow: A system for large-scale machine learning, *CoRR*, **vol. abs/1605.08695**.
http://arxiv.org/abs/1605.08695

Alibaba (2019), Euler 2.0.
https://github.com/alibaba/euler

Barabási, A.-L. and M. Pósfai (2016), *Network science*, Cambridge University Press, Cambridge, ISBN 9781107076266 1107076269.
http://barabasi.com/networksciencebook/

Battaglia, P. W., J. B. Hamrick, V. Bapst, A. Sanchez-Gonzalez, V. F. Zambaldi, M. Malinowski, A. Tacchetti, D. Raposo, A. Santoro, R. Faulkner, Ç. Gülçehre, H. F. Song, A. J. Ballard, J. Gilmer, G. E. Dahl, A. Vaswani, K. R. Allen, C. Nash, V. Langston, C. Dyer, N. Heess, D. Wierstra, P. Kohli, M. Botvinick, O. Vinyals, Y. Li and R. Pascanu (2018), Relational inductive biases, deep learning, and graph networks, *CoRR*, **vol. abs/1806.01261**.
http://arxiv.org/abs/1806.01261

Battaglia, P. W., R. Pascanu, M. Lai, D. J. Rezende and K. Kavukcuoglu (2016), Interaction Networks for Learning about Objects, Relations and Physics, *CoRR*, **vol. abs/1612.00222**.
http://arxiv.org/abs/1612.00222

Bishop, C. M. (2006), *Pattern Recognition and Machine Learning*, Springer, ISBN 978-0387-31073-2.

Bradbury, J., R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne and Q. Zhang (2018), JAX: composable transformations of Python+NumPy programs.
http://github.com/google/jax

Bronstein, M. M., J. Bruna, T. Cohen and P. Veličković (2021), Geometric Deep Learning: Grids, Groups, Graphs, Geodesics, and Gauges.

Bronstein, M. M., J. Bruna, Y. LeCun, A. Szlam and P. Vandergheynst (2016), Geometric deep learning: going beyond Euclidean data, *CoRR*, **vol. abs/1611.08097**.

Chen, T., S. Kornblith, M. Norouzi and G. Hinton (2020), A Simple Framework for Contrastive Learning of Visual Representations.

Chen, T., M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang and Z. Zhang (2015), MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems, *CoRR*, **vol. abs/1512.01274**.
http://arxiv.org/abs/1512.01274

Chevalier-Boisvert, M., L. Willems and S. Pal (2018), Minimalistic Gridworld Environment for OpenAI Gym, https://github.com/maximecb/gym-minigrid.

Csáji, B. C. (2001), Approximation with Artificial Neural Networks, *Eötvös Loránd University Hungary*.

Curran, W., T. Brys, M. E. Taylor and W. D. Smart (2015), Using PCA to Efficiently Represent State Spaces, *CoRR*, **vol. abs/1505.00322**.
http://arxiv.org/abs/1505.00322

Cybenko, G. (1989), Approximation by superpositions of a sigmoidal function, *Mathematics of Control, Signals, and Systems (MCSS)*, **vol. 2**, pp. 303–314.

Dai, Z., Z. Yang, Y. Yang, J. G. Carbonell, Q. V. Le and R. Salakhutdinov (2019), Transformer-XL: Attentive Language Models Beyond a Fixed-Length Context, *CoRR*, **vol. abs/1901.02860**.
http://arxiv.org/abs/1901.02860

DeepMind (2018), Graph Nets library, https://github.com/deepmind/graph_nets.

DeepMind (2020), Jraph - A library for graph neural networks in jax.
https://github.com/deepmind/jraph

Devlin, J., M. Chang, K. Lee and K. Toutanova (2018), BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding, *CoRR*, **vol. abs/1810.04805**.
http://arxiv.org/abs/1810.04805

Easley, D. and J. Kleinberg (2010), *Networks, Crowds, and Markets: Reasoning About a Highly Connected World*, Cambridge University Press.
http://www.cs.cornell.edu/home/kleinber/networks-book/

evanzd (2021), Crawl and Visualize ICLR 2021 OpenReview Data,
https://github.com/evanzd/ICLR2021-OpenReviewData.

Fey, M. and J. E. Lenssen (2019), Fast Graph Representation Learning with PyTorch Geometric.

Fodor, I. K. (2002), A Survey of Dimension Reduction Techniques, Technical report.

Geng, X., M. Zhang, J. Bruce, K. Caluwaerts, M. Vespignani, V. SunSpiral, P. Abbeel and S. Levine (2016), Deep Reinforcement Learning for Tensegrity Robot Locomotion, *CoRR*, **vol. abs/1609.09049**.

Gilmer, J., S. S. Schoenholz, P. F. Riley, O. Vinyals and G. E. Dahl (2017), Neural Message Passing for Quantum Chemistry.

Goodfellow, I., Y. Bengio and A. Courville (2016), *Deep Learning*, MIT Press,
http://www.deeplearningbook.org.

Gori, M., G. Monfardini and F. Scarselli (2005), A new model for learning in graph domains, in *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, volume 2, pp. 729–734 vol. 2.

Grattarola, D. and C. Alippi (2020), Graph Neural Networks in TensorFlow and Keras with Spektral.

Ha, D. and J. Schmidhuber (2018), World Models, *CoRR*, **vol. abs/1803.10122**.
http://arxiv.org/abs/1803.10122

Hamilton, W. L. (2020), Graph Representation Learning, *Synthesis Lectures on Artificial Intelligence and Machine Learning*, **vol. 14**, pp. 1–159.

Hamilton, W. L., R. Ying and J. Leskovec (2017), Representation Learning on Graphs: Methods and Applications, *CoRR*, **vol. abs/1709.05584**.
http://arxiv.org/abs/1709.05584

Hamrick, J. B., K. R. Allen, V. Bapst, T. Zhu, K. R. McKee, J. B. Tenenbaum and P. W. Battaglia (2018), Relational inductive bias for physical construction in humans and machines, *CoRR*, **vol. abs/1806.01203**.
http://arxiv.org/abs/1806.01203

Hendrycks, D. and K. Gimpel (2016), Bridging Nonlinearities and Stochastic Regularizers with Gaussian Error Linear Units, *CoRR*, **vol. abs/1606.08415**.
http://arxiv.org/abs/1606.08415

van Hoof, H., N. Chen, M. Karl, P. van der Smagt and J. Peters (2016), Stable reinforcement learning with autoencoders for tactile and visual data, in *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 3928–3934, doi:10.1109/IROS.2016.7759578.

Hornik, K. (1991), Approximation capabilities of multilayer feedforward networks, *Neural Networks*, **vol. 4**, pp. 251–257.

Jonschkowski, R. and O. Brock (2015), Learning State Representations with Robotic Priors, *Auton. Robots*, p. 407–428, ISSN 0929-5593.

Kaiser, L., M. Babaeizadeh, P. Milos, B. Osinski, R. H. Campbell, K. Czechowski, D. Erhan, C. Finn, P. Kozakowski, S. Levine, R. Sepassi, G. Tucker and H. Michalewski (2019), Model-Based Reinforcement Learning for Atari, *CoRR*, **vol. abs/1903.00374**.
http://arxiv.org/abs/1903.00374

Kingma, D. P. and J. Ba (2014), Adam: A Method for Stochastic Optimization, published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015.

Kipf, T., E. Fetaya, K.-C. Wang, M. Welling and R. Zemel (2018), Neural Relational Inference for Interacting Systems.

Kipf, T., E. van der Pol and M. Welling (2020a), C-SWM Github,
https://github.com/tkipf/c-swm.

Kipf, T., E. van der Pol and M. Welling (2020b), Contrastive Learning of Structured World Models.

Kipf, T. N. and M. Welling (2016), Semi-Supervised Classification with Graph Convolutional Networks.

Kober, J., J. A. Bagnell and J. Peters (2013), Reinforcement learning in robotics: A survey., **vol. 32**, no.11, pp. 1238–1274.

Krizhevsky, A., I. Sutskever and G. E. Hinton (2012), ImageNet Classification with Deep Convolutional Neural Networks, in *Advances in Neural Information Processing Systems*, volume 25, Eds. F. Pereira, C. J. C. Burges, L. Bottou and K. Q. Weinberger, Curran Associates, Inc., pp. 1097–1105.

de Lange, E. (2017), accessed: 2020-04-20.
https://github.com/erikdelange/Reinforcement-Learning-Maze

LeCun, Y., Y. Bengio and G. Hinton (2015), Deep Learning, *Nature*, **vol. 521**.

LeCun, Y., B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard and L. D. Jackel (1990), Handwritten digit recognition with a back-propagation network, in *Advances in Neural Information Processing Systems (NIPS 1989)*, volume 2, Ed. D. Touretzky, Morgan Kaufman, Denver, CO.

LeCun, Y. and C. Cortes (2010), MNIST handwritten digit database,
http://yann.lecun.com/exdb/mnist/.
http://yann.lecun.com/exdb/mnist/

Lesort, T., N. D. Rodríguez, J. Goudou and D. Filliat (2018), State Representation Learning for Control: An Overview, *CoRR*, **vol. abs/1802.04181**.
http://arxiv.org/abs/1802.04181

Lesort, T., M. Seurin, X. Li, N. D. Rodríguez and D. Filliat (2017), Unsupervised state representation learning with robotic priors: a robustness benchmark, *CoRR*, **vol. abs/1709.05185**.
http://arxiv.org/abs/1709.05185

Ma, L., Z. Yang, Y. Miao, J. Xue, M. Wu, L. Zhou and Y. Dai (2019), NeuGraph: Parallel Deep Neural Network Computation on Large Graphs, in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, USENIX Association, Renton, WA, pp. 443–458, ISBN 978-1-939133-03-8.
https://www.usenix.org/conference/atc19/presentation/ma

Mahadevan, S. and M. Maggioni (2007), Proto-Value Functions: A Laplacian Framework for Learning Representation and Control in Markov Decision Processes, *J. Mach. Learn. Res.*, **vol. 8**, p. 2169–2231, ISSN 1532-4435.

Mikolov, T., K. Chen, G. Corrado and J. Dean (2013), Efficient Estimation of Word Representations in Vector Space, *CoRR*.

Mnih, V., K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg and D. Hassabis (2015), Human-level control through deep reinforcement learning, *Nature*, **vol. 518**, pp. 529–533, ISSN 00280836.

Moerland, T. M., J. Broekens and C. M. Jonker (2021), Model-based Reinforcement Learning: A Survey.

OpenAI (2000), Gym, https://gym.openai.com/.

OpenAI, M. Andrychowicz, B. Baker, M. Chociej, R. Józefowicz, B. McGrew, J. W. Pachocki, J. Pachocki, A. Petron, M. Plappert, G. Powell, A. Ray, J. Schneider, S. Sidor, J. Tobin, P. Welinder, L. Weng and W. Zaremba (2018), Learning Dexterous In-Hand Manipulation, *CoRR*, **vol. abs/1808.00177**.
http://arxiv.org/abs/1808.00177

Ouyang, C. (2019), Robot Navigation in Dynamic Environment Based on Reinforcement Learning.

Paszke, A., S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai and S. Chintala (2019), PyTorch: An Imperative Style, High-Performance Deep Learning Library, *CoRR*, **vol. abs/1912.01703**.
http://arxiv.org/abs/1912.01703

van der Pol, E., T. Kipf, F. A. Oliehoek and M. Welling (2020), Plannable Approximations to MDP Homomorphisms: Equivariance under Actions.

PYPL (2021), Popularity of Programming Language, https://pypl.github.io/PYPL.html.

van Rossum, G. (2009), The History of Python, http://python-history.blogspot.com/2009/01/brief-timeline-of-python.html.

Rummery, G. A. and M. Niranjan (1994), On-Line Q-Learning Using Connectionist Systems, Technical Report TR 166, Cambridge University Engineering Department.

Scarselli, F., M. Gori, A. C. Tsoi, M. Hagenbuchner and G. Monfardini (2009), The Graph Neural Network Model, *IEEE Transactions on Neural Networks*, **vol. 20**, pp. 61–80.

Silver, D. (2015a), Lecture 2 - Markov Decision Processes.

Silver, D. (2015b), Lecture 8 - Integrating Learning and Planning.

Silver, D., A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel and D. Hassabis (2016), Mastering the Game of Go with Deep Neural Networks and Tree Search, *Nature*, **vol. 529**, pp. 484–489.

Silver, D., T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. P. Lillicrap, K. Simonyan and D. Hassabis (2017), Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm, *CoRR*, **vol. abs/1712.01815**.
http://arxiv.org/abs/1712.01815

Singh, S., T. Jaakkola and M. Jordan (1995), Reinforcement Learning with Soft State Aggregation, in *Advances in Neural Information Processing Systems*, volume 7, Eds.

G. Tesauro, D. Touretzky and T. Leen, MIT Press.

Spelke, E. S. and K. D. Kinzler (2007), Core knowledge, *Developmental Science*, **vol. 10**, pp. 89–96.

van Steenkiste, S., M. Chang, K. Greff and J. Schmidhuber (2018), Relational Neural Expectation Maximization: Unsupervised Discovery of Objects and their Interactions, *CoRR*, **vol. abs/1802.10353**.
http://arxiv.org/abs/1802.10353

Sutton, R. S. and A. G. Barto (2018), *Reinforcement Learning: An Introduction*, The MIT Press, second edition.

Vaswani, A., N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser and I. Polosukhin (2017), Attention Is All You Need, *CoRR*, **vol. abs/1706.03762**.
http://arxiv.org/abs/1706.03762

Veličković, P., G. Cucurull, A. Casanova, A. Romero, P. Liò and Y. Bengio (2018), Graph Attention Networks.

Wang, M., L. Yu, D. Zheng, Q. Gan, Y. Gai, Z. Ye, M. Li, J. Zhou, Q. Huang, C. Ma, Z. Huang, Q. Guo, H. Zhang, H. Lin, J. Zhao, J. Li, A. J. Smola and Z. Zhang (2019), Deep Graph Library: Towards Efficient and Scalable Deep Learning on Graphs, *CoRR*, **vol. abs/1909.01315**.
http://arxiv.org/abs/1909.01315

Wang, T., R. Liao, J. Ba and S. Fidler (2018), NerveNet: Learning Structured Policy with Graph Neural Networks, in *International Conference on Learning Representations*.
https://openreview.net/forum?id=S1sqHMZCb

Watkins, C. J. C. H. (1989), *Learning from Delayed Rewards*, Ph.D. thesis, King's College, Cambridge, UK.

Watkins, C. J. C. H. and P. Dayan (1992), Q-learning, *Machine Learning*, **vol. 8**, pp. 279–292.

Watter, M., J. T. Springenberg, J. Boedecker and M. A. Riedmiller (2015), Embed to Control: A Locally Linear Latent Dynamics Model for Control from Raw Images, *CoRR*, **vol. abs/1506.07365**.
http://arxiv.org/abs/1506.07365

Watters, N., L. Matthey, M. Bosnjak, C. P. Burgess and A. Lerchner (2019), COBRA: Data-Efficient Model-Based RL through Unsupervised Object Discovery and Curiosity-Driven Exploration, *CoRR*, **vol. abs/1905.09275**.
http://arxiv.org/abs/1905.09275

Watters, N., D. Zoran, T. Weber, P. Battaglia, R. Pascanu and A. Tacchetti (2017), Visual Interaction Networks: Learning a Physics Simulator from Video, in *Advances in Neural Information Processing Systems*, volume 30, Eds. I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan and R. Garnett, Curran Associates, Inc.

Xu, Z., Z. Liu, C. Sun, K. Murphy, W. T. Freeman, J. B. Tenenbaum and J. Wu (2019), Unsupervised Discovery of Parts, Structure, and Dynamics, *CoRR*, **vol. abs/1903.05136**.
http://arxiv.org/abs/1903.05136

Zachary, W. (1977), An information flow model for conflict and fission in small groups, *Journal of Anthropological Research*, **vol. 33**, pp. 452–473.