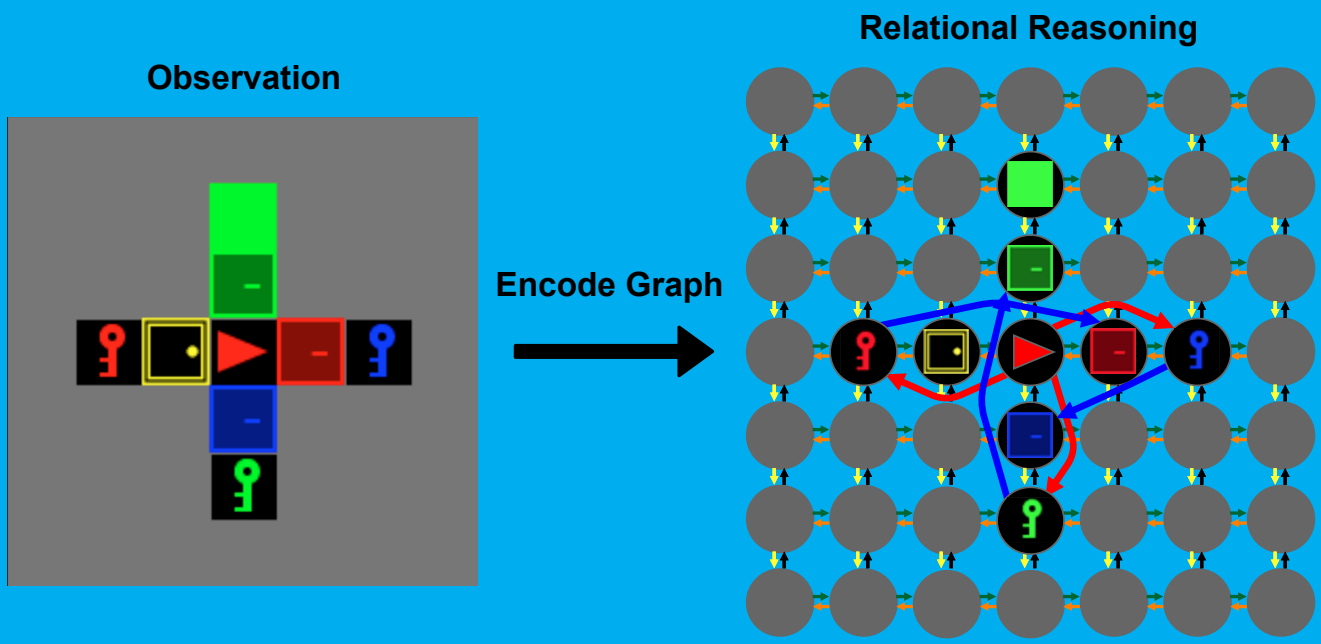
Reinforcement learning with domain-specific relational inductive biases

Using Graph Neural Networks and domain knowledge

Erik Vester



TU Delft

Reinforcement learning with domain-specific relational inductive biases

Using Graph Neural Networks and domain knowledge

by

Erik Vester

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on October 25, 2021 at 13:30.

Student number: 4305388
Thesis committee: Dr. M. T. J. Spaan, TU Delft, supervisor
Dr. J. W. Böhmer, TU Delft, daily supervisor
Dr. L. C. Siebert, TU Delft, external committee member

TU Delft

Preface

My decision to take a 'bridging year' after completing my bachelor's degree in Applied Earth Sciences, to be able to take part in the Computer Science master's program, has been largely based on my interest into the topics of deep learning and reinforcement learning. This thesis, marking the end of my studies at the TU Delft, combines exactly these topics. Just a few years ago I hardly understood the concepts of these two terms and I barely wrote any code, however, I am proud to say that I now feel like I have a solid understanding of these topics. During this thesis project I have learned a lot about the theory behind Deep Reinforcement Learning and Graph Neural Networks - both new topics for me - as well as the details and difficulties of the actual application of these methods in practice.

I would not have been able to write this thesis without the help of my supervisors: Wendelin Böhmer and Matthijs Spaan. I would therefore like to express my gratitude for their support and input throughout this project. Their detailed feedback and insights have helped me a lot and I have experienced their calm and supportive attitude - even when my research did not always go as planned - as very pleasant. Even though Matthijs and Wendelin both have very busy schedules, they always managed to make time for me, even for a last minute or evening call.

Besides my supervisors, I would also like to thank Luciano Cavalcante Siebert for taking the time to evaluate my work and take part in the thesis committee.

Finally, I would like to thank Zhengyao Jiang, who did research on a comparable subject and took the time to talk with me and share his thoughts on the topic.

Besides the theoretical knowledge gained from this project, I have also learned a lot on a personal level. Writing this thesis largely at home was challenging for me, however, this challenge was made easier by the presence of my family, roommates and friends, to whom I would therefore also like to express my gratitude for their support throughout this period.

I hope you enjoy reading!

Erik Vester
Schalkwijk, October 2021

Contents

Acronyms

AI Artificial Intelligence.

CNN Convolutional Neural Network.

DL Deep Learning.

DRL Deep Reinforcement Learning.

GAE Generalized Advantage Estimation.

GAN Graph Attention Network.

GAP Global Attention Pooling.

GCN Graph Convolutional Network.

GNN Graph Neural Network.

MDP Markov Decision Process.

MLP Multilayer Perceptron.

MSE Mean Squared Error.

NN Neural Network.

PPO Proximal Policy Optimization.

R-GCN Relational Graph Convolution Network.

RL Reinforcement Learning.

RRL Relational Reinforcement Learning.

SGA Stochastic Gradient Ascent.

SGD Stochastic Gradient Descent.

1

Abstract

Reinforcement Learning (RL) has been used to successfully train agents for many tasks, but generalizing to a different task - or even unseen examples of the same task - remains difficult. In this thesis, Deep Reinforcement Learning (DRL) is combined with Graph Neural Networks (GNNs) and domain knowledge, with the aim of improving the generalization capabilities of RL-agents.

In classical DRL setups, Convolutional Neural Networks (CNNs) and Multilayer Perceptrons (MLPs) are often applied as neural network architectures for an agent's policy and/or value network. In this thesis, however, GNNs are used to represent the policy and value network of an agent, which allows for the application of relational inductive biases that are more domain-specific than those of MLPs and CNNs. Observations received by the agent from a simple navigation task - which requires some relational reasoning - are encoded as graphs, consisting of entities and relations between them, which are based on domain knowledge. These graphs are then used as structured input for the GNN-based architecture of the agent. This approach is inspired by human relational reasoning, which is argued to be an important factor in human generalization capabilities.

Several GNN-based architectures are proposed and compared, from which two main architectures are distilled: R-GCN$_{\text{domain}}$ and R-GCN$_{\text{GAN}}$. In the R-GCN$_{\text{domain}}$ architecture, the graph encoding of observations is based on domain knowledge, whereas in R-GCN$_{\text{GAN}}$ we aim to combine the relational encoding of a CNN with additional, learned relations, allowing for an end-to-end solution that does not require domain knowledge. Sample efficiency and both in- and out-of-distribution generalization performance of our architectures are tested on a new grid world environment called 'Key-Corridors'. We find that adding domain-specific relational inductive biases with the R-GCN$_{\text{domain}}$ architecture significantly improves sample efficiency and out-of-distribution generalization, when compared to MLPs and CNNs. However, we did not succeed in learning these domain-specific relational inductive biases with R-GCN$_{\text{GAN}}$, which does not manage to significantly outperform a CNN. Overall, the results indicate that applying relational reasoning in RL - through the use of GNNs and domain knowledge - can be an important tool for improving sample efficiency and generalization performance.

2

Introduction

From industry automation, financial and business management, natural language processing, health care decision-making, biological data analysis to traffic signal control - RL has proven increasingly valuable for developing powerful solutions for important and challenging real-life problems in recent years (Arel et al., 2010; Kober et al., 2013; Mahmud et al., 2018; Yu et al., 2019; Zhong et al., 2017). Although trained agents can solve complex tasks, they struggle to generalize their experience to a different task or even unseen examples of the same task (Cobbe et al., 2019; Farebrother et al., 2018). Generalization therefore remains one of the key challenges for (deep) reinforcement learning algorithms. However, Packer et al. (2018), among others, argue that this is actually an indispensable feature for applying advanced intelligent systems in the real world, where they will inevitably encounter unforeseen situations. Battaglia et al. (2018) even advocate that combinatorial generalization should be a top priority for Artificial Intelligence (AI) to reach human-like abilities.

In recognition of the importance of this challenge, much attention and research efforts have been focused on ways to improve generalization in RL. Zhao et al. (2019), for instance, evaluate several techniques for improving generalization and find that reducing architecture size or adding regularizers to reduce overfitting can increase generalization performance. Other work such as by Rajeswaran et al. (2016), Tobin et al. (2017) and K. Wang et al. (2020) has focused on approaches related to sampling from diverse environments during training. The latter aim to effectively increase training data diversity by training the RL-agent on a mix of observations collected from different training environments. Recent work in the field of meta-learning and multi-task reinforcement learning (D'Eramo et al., 2020; Finn et al., 2017; Hessel et al., 2019; Kirsch et al., 2019; Shu et al., 2017) have also shown promising advances. For instance, Kirsch et al. (2019) propose a novel meta RL algorithm, which combines the experiences of many agents to learn an objective function which determines how future individuals will learn. These are just a few examples of various techniques, each of which attempts to improve generalization performance in a different way. However, they fall far short of the human capacity for generalization. Humans solve novel problems - a comparable task to generalization in RL - by composing familiar skills and routines (Anderson, 1982). More specifically, it has been proposed that this human capacity is rooted in a high-level form of relational reasoning (Waltz et al., 1999). As Holyoak and Lu (2021, p. 118) explain, relational reasoning refers to *"the ability to form explicit representations of relations between entities, and the ability to make inferences by integrating multiple relations and comparing relations across domains."*

In this thesis we focus on trying to incorporate this type of relational reasoning into DRL, with the aim to improve the general performance of the RL-agent and its capacity to generalize. This is done by encoding observations of a navigation task - requiring some relational reasoning - as graphs, which are inherently suitable for representing entities (nodes) with relations (edges) between them. Subsequently, GNNs are applied to these graphs, instead of applying other types of neural network architectures such as MLPs and CNNs directly on the observations. In general, the structure of a GNN supports generalization as it performs shared computations across entities and relations and therefore allows for reasoning about new input when this input is built from familiar components (Battaglia et al., 2018). Furthermore, we will compare different types of GNNs, which allow for a variety of possible relational encodings of the observations, each inducing different relational inductive biases and therefore having

different effects on the general performance of the agent and generalization performance specifically.

To encode observations as graphs consisting of entities and relations, we assume domain knowledge to be available about useful domain-specific relations present between the entities in the observations. Even though this is a strong assumption, we argue that in many cases domain knowledge like this is available, for instance in the form of domain experts or databases describing relations between objects or persons. In addition to applying domain knowledge to define relations in the observations, we also explore an approach which tries to learn useful relations in an end-to-end fashion, removing the need for domain knowledge.

In line with the objectives of this thesis, we pose the following research questions:

> **Research question 1**: *Can the sample efficiency of DRL be improved by using GNNs and domain knowledge, compared to traditional architectures such as CNNs and MLPs?*
>
> **Research question 2**: *Can the generalization performance of DRL be improved by using GNNs and domain knowledge, compared to traditional architectures such as CNNs and MLPs?*
>
> **Research question 3**: *Can the fixed relational inductive biases of a CNN be combined with learned domain-specific relational inductive biases and GNNs to improve sample efficiency and generalization performance in an end-to-end fashion, which does not require domain knowledge?*

To answer research question 1 and 2 we explore multiple graph encodings of the observations and apply Graph Convolutional Networks (GCNs) and Relational Graph Convolution Networks (R-GCNs) to these graphs. Much like the previously mentioned works by Rajeswaran et al. (2016), Tobin et al. (2017) and K. Wang et al. (2020), we train our agents on different initialization of our environment, where we, for instance, vary the position of objects and their colors. To test for in-distribution generalization, we train agents on a subset of all of the possible initializations of the environment and test them on the remaining ones, which are unseen during training. To test for out-of-distribution generalization performance, we slightly modify the training task and evaluate whether the agent can generalize - without further training - to this task as well. Finally, for research question 3, we apply a combination of R-GCN and a Graph Attention Network (GAN) with the aim of integrating the relational inductive biases of a CNN - which are known to perform well on grid-shaped data - with learned relational inductive biases. This approach would allow for an end-to-end method which does not require domain knowledge. For research question 3, we again test for sample efficiency and in- and out-of-distribution generalization performance, compared to a baseline architecture using a CNN.

3
Background

In this chapter the necessary background for the following chapters of this thesis will be laid out. For readers with a background in reinforcement learning, machine learning and deep learning many parts of this chapter will be redundant, however, it will introduce notation used in following sections, as well as some of the used algorithms.

3.1. Deep Learning

Machine learning or pattern recognition used to require a lot of time and knowledge to create a feature extractor that mapped raw input (e.g. pixels) to a feature representation which the system could use to classify or detect patterns (Lecun et al., 2015). In representation learning one tries to automatically learn these feature representations and Deep Learning (DL) is a form of representation learning, which introduces representations that are expressed in terms of other, simpler representations (Goodfellow et al., 2016). In DL this is done with a Neural Network (NN) consisting out of multiple layers, from which the adjective ”deep” is derived. In the domain of natural images, the first layer could for instance extract edges, after which a second layer could then extract corners and contours, a third layer could extract complete objects parts, and finally, the last layer - called an ’output layer’ - predicts an actual object that is present in the image. In this case the idea of introducing representations expressed in terms of other, simpler representations is quite intuitive, but in reality - especially on other types of data - this can be much more abstract and happen in a way that is not necessarily easy to understand for humans.

In general, NNs can be seen as function approximators. For an example function f^* mapping an input x - often in the form of a vector or matrix of features - to an output $y = f^*(x)$, a NN could be trained as a mapping $y = f(x;\theta)$, where f is an approximation of f^* and the network learns the parameters θ that give the best approximation. To train such a NN, a cost or loss function $L(\theta)$ has to be defined, which is to be minimized during training. In this training process, input is first fed through the NN in a so called ’forward pass’, which results in an output. This output can then be used to compute the loss, according to $L(\theta)$. A commonly used loss function in a supervised learning setup is the Mean Squared Error (MSE) loss: $L(\theta) = \frac{1}{N}\sum_{i=0}^{N}(y_i - \hat{y}_i)^2$, where $\hat{y}$ represents ground truth labels, y are the output predictions and N represents the amount of samples in $\hat{y}$ and y. Sometimes the actual loss function cannot be optimized in an efficient manner, in which case a surrogate loss can be defined that can be considered as a proxy for the actual loss. This is for instance used in PPO, the RL algorithm used in this thesis, which will be discussed in more detail in section 3.4.4. The forward pass is followed by a backward pass, in which the parameters θ - sometimes also called weights, which is not fully accurate as θ includes both the weights and biases of the network when both are used - of the NN are updated with the goal to minimize the loss. To optimize these parameters, gradient descent methods are usually applied, in which the gradient of the loss function - which is the first-order derivative of the loss function with respect to θ - is computed. This gradient is denoted by $\nabla_\theta L(\theta)$ and during optimization the parameters are updated in the negative direction of this gradient, thus minimizing the loss (Lecun et al., 2015). One gradient descent step would then have the following shape:

$$\theta_{\text{new}} = \theta - \mu\nabla_\theta L(\theta)\ , \tag{3.1}$$

where θ' denotes the updated parameters and μ is a learning rate defining how big the update step should be. To compute the gradient of a multilayer NN with respect to all of its parameters θ, backpropagation (Chauvin & Rumelhart, 2013) is applied, which amounts to a smart application of the chain rule (Lecun et al., 2015). In practice, this is often done with software packages offering automatic differentiation, such as PyTorch (Paszke et al., 2019). This process of first predicting new output values with the current parameters θ in a forward pass and then updating the parameters in the backward pass is applied iteratively until - ideally - a good performance is reached.

The gradient descent optimization methods actually often update parameters θ based on an expected value of the loss, which is estimated based on a subset - also called a mini-batch - of all the available training data. This is called 'mini-batch gradient descent' but is sometimes also simply referred to as Stochastic Gradient Descent (SGD) (Goodfellow et al., 2016).

3.1.1. Multilayer Perceptrons

One of the most common architectures for DL is the MLP, which consists of multiple linear - also called fully connected - layers. Each layer l can be seen as a vector-to-vector function $h^{(l+1)} = \sigma^{(l)}(W^{(l)}h^{(l)} + b^{(l)})$, where $W^{(l)} \in \mathbb{R}^{|h^{(l)}|\times|h^{(l+1)}|}$ represents a weight matrix and $b^{(l)}$ a bias vector, which together form the trainable parameters θ of the NN. Furthermore, $h^{(l)}$ is a vector representing the input features, or the output features of the previous layer, and $\sigma^{(l)}$ is a non-linear activation function.

Another way of looking at MLPs - which is useful for the comparison with GNNs later on - is by thinking of each layer as many units in parallel, where each unit is a neuron (Goodfellow et al., 2016). Each of these neurons receive input from all other neurons in the previous layer - or the actual input in case of the first layer - and computes its own activation. It does this by taking a weighted sum of all of its inputs and adding a bias value, after which the non-linear activation function σ is applied (Lecun et al., 2015).

3.1.2. Convolutional Neural Networks

Another common type of neural networks are CNNs (LeCun et al., 1989). CNNs are well suited for grid-shaped data such as images - which can be seen as a two-dimensional grid of pixels - and are known to have good performance on recognition and classification tasks on images and video (He et al., 2015; Karpathy et al., 2014; Simonyan & Zisserman, 2014).

Instead of only having fully connected layers like a MLP, CNNs usually have three types of layers: convolutional layers, pooling layers and fully connected layers. The principles behind a convolutional layer are the same as those of a MLP, as each layer still consist of trainable neurons with weights, biases and activation functions. However, the input of a convolutional layer usually has a shape of $[X, Y, D]$, where X, Y and D are the width, height and depth of the input, respectively. The depth of the input is also referred to as 'channels' and in the case of images, each pixel in the X by Y grid of pixels often consists of the 3 RGB color channels: red, green and blue. A convolutional layer is comprised out of learned kernels - each having its own weights and bias - that convolve over the input to create a so called feature map. Each feature of the feature map is computed by applying the weights and bias of the kernel and is only based on a region of the input, defined by the size of the kernel. This region is called the receptive field of the convolutional layer. To create the full feature map, the kernels are convolved over the input, which can be thought of as a filter being moved over the input. Whilst the kernel is convolved over the input, the learned weights remain constant, which is also referred to as weight sharing (Gu et al., 2018).

It is important to note that although each kernel moves across the X by Y grid, targeting only a portion of the input grid at each position, it usually does have weights for the entire depth of the input. Therefore, applying a single kernel at one position results in one output feature based on the entire depth of the input and if, for instance, 5 output features are required, 5 kernels need to be learned. Applying a convolutional layer l with kernels of size $n \times m$ to compute features h_{ij} at one position $[i, j]$ of a feature map, can be summarized by the following equation:

$$h_{ij}^{(l+1)} = \sigma^{(l)}\left(\sum_{x=0}^{n-1}\sum_{y=0}^{m-1} W_{xy}^{(l)} h_{(i+x)(j+y)}^{(l)} + b^{(l)}\right). \tag{3.2}$$

Here, $W^{(l)} \in \mathbb{R}^{n \times m \times |h^{(l)}| \times |h^{(l+1)}|}$ is the learned weight matrix, containing all the weights of the $|h^{(l+1)}|$ kernels used to transform the input features $h^{(l)}$. Furthermore, $b^{(l)} \in \mathbb{R}^{|h^{(l+1)}|}$ is a vector containing the learned bias terms of the kernels and $\sigma^{(l)}$ is a non-linear activation function. Equation 3.2 describes the update of one position in the feature map, however, weight matrix $W^{(l)}$ and the bias terms $b^{(l)}$ are shared for all positions in the feature map.

The stride of a kernel defines its 'step size' when moving over the input grid, where a stride of 1 means that the kernel is shifted one pixel at a time. Furthermore, padding can be added around the perimeter of the input of a convolutional layer to avoid or reduce a size difference from the input to the output feature map.

Pooling layers can be applied in between convolutional layers for a reduction of resolution, which helps to achieve shift-invariance. Examples of commonly used pooling layers are max-pooling and average-pooling (Boureau et al., 2010; Gu et al., 2018). After a few convolutional layers - possibly with pooling layers in between - the final feature map is often flattened into a 1D tensor, on which more linear layers can be applied to learn non-linear combinations of all the features in the output feature map.

3.2. Deep learning with graphs

The following sections will focus on deep learning with GNNs, which are NNs working on graph-shaped input. First, graphs and their notation are introduced, after which the general concept of GNNs will be covered, as well as some specific types of GNNs which will be used later in this thesis.

3.2.1. Graphs

To get into the topic of GNNs we first introduce some background on graphs and the corresponding notation used in this thesis. The following section on graphs is based on the works by Battaglia et al. (2018), Bondy and Murty (1976), and Wilson (2005), which together provide an extensive overview of graph theory and its application in GNNs.

Graphs can be used to represent many real-world data structures or situations, especially those that can be expressed in the form of entities with certain relationships or interactions between them, such as the structure of a molecule or a social network. In the case of a social network, nodes can represent individuals and edges can then define certain relationships between these individuals, such as friendships or family ties. Another good example of a graph structure - related to this thesis - is a neural network, where each neuron would be a node and each weight would be represented by an edge.

In this thesis a graph is defined as a 2-tuple $G = (V, E)$, where V represents a - usually finite - set of nodes (also called vertices) $\{v_i\}_{i=1:|V|}$ of cardinality $|V|$. E is a set of edges $\{e_j\}_{j=1:|E|}$ of cardinality $|E|$, and each edge e_j is again a 2-tuple of nodes $e_j = (v_a, v_b)$, indicating a connection or relation between node v_a and node v_b. These edges can be directed or undirected; a directed edge being an edge directed from a source node to a target node, whereas an undirected edge does not have a specific source and target node but is simply an edge between two nodes. When a graph has directed edges it is called a directed graph and in this case the 2-tuple per edge $e_j = (v_s, v_t)$ is ordered, where v_s is the source node and v_t is the target node. See figure 3.1 for an example of how graphs are illustrated in this thesis. Here, edge e_1 is an example of a directed edge from source node v_4 to target node v_5, where the direction of the edge is depicted by the arrowhead. In this thesis, all edges displayed without arrowheads are actually referring to two separate directed edges in both directions. For example, the edge between nodes v_1 and v_3 in figure 3.1, represent both an edge from v_1 to v_3 and an edge from v_3 to v_1.

The degree d_i of a node v_i is equal to the number edges that are attached to this node. For example, in figure 3.1, node v_1 has a degree of 2 and node v_3 has a degree of 4. Two nodes are said to be adjacent if they are connected by an edge and the neighborhood $\mathcal{N}_i$ of node v_i encompasses all nodes that are adjacent to v_i. A k-hop neighborhood of a node v_i defines a set of nodes that are reachable from node v_1, by traversing k edges. For example, in figure 3.1, the 1-hop neighborhood of node v_1 is: $\{v_3, v_4\}$ and the 2-hop neighborhood of node v_1 would already include each node of this small example graph, as all nodes are reachable from v_1 by traversing 2 edges. A graph is called 'fully connected' if there is a directed edge from each node $v_i \in V$ to all other nodes $v_j \in V \setminus \{v_i\}$. Lastly, in this thesis, each node $v_i \in V$ can have a set of properties, described by its node features h_i. In

the previously introduced example of a graph describing a social network where each node represents an individual, the node features could encode properties of these individuals, such as age, sex and occupation.

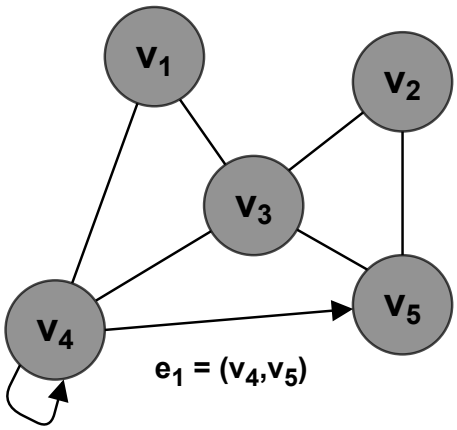


Figure 3.1: A graph, where e_1 is an example of a directed edge from node 4 to node 5, where the direction of the edge is depicted by the arrowhead. In this thesis, all edges displayed without arrowheads are referring to two separate edges in both directions. For example, the edge between nodes v_1 and v_3, represent both an edge from v_1 to v_3 and an edge from v_3 to v_1.

3.2.2. Graph Neural Networks

GNNs is an umbrella term for neural networks designed to operate on graph-shaped input data. The concept of GNNs was originally introduced by Gori et al. (2005) and, in general, each layer in a GNN can be seen as a 'graph-to-graph' function (Battaglia et al., 2018). As there are many different approaches to GNNs, we will not discuss all of them here and instead focus on the type of GNNs applied in this thesis. For the interested reader, Battaglia et al. (2018), Wu et al. (2019), and Zhou et al. (2018) can be used to get a more extensive overview of the research on GNNs. In this section, we will first explain the general concept of GNNs, after which section 3.2.3, 3.2.4 and 3.2.5 go into further detail on the three types of GNNs applied in this thesis.

A GNN layer l takes a graph, in which each node $v_i \in V$ already has some initial node features h_i^l, as input and tries to learn new node features h_i^{l+1} for each node in the graph. The goal is - as with any type of NN - to learn features that are useful for some downstream task, which in the case of GNNs usually is node or graph classification. After learning the new features, the layer returns the graph with the updated node features as output. In the case of node classification, one tries to learn node features that are useful for predicting the class to which a certain node belongs, whereas for graph classification the goal is to predict a class for the entire graph.

An intuitive way to describe the internal workings of a GNN is by using the 'message passing' paradigm. From this perspective and focusing on the type of GNNs implemented in this thesis, one layer of a GNN usually applies the following steps (see figure 3.2 for a visual representation of these steps) in succession to update the node features of a graph:

1. Input graph: Each layer l receives an input graph $G(V, E)$ which has node features h_i^l for each node $v_i \in V$. For the first layer of a GNN network these will be the input features h_i^0 of all the nodes.

2. Message passing: Each node sends a 'message' to all of its neighboring nodes, including itself in case it has a self-loop. This message can be a learned transformation of the node features, however in this thesis it will just contain the current features of the node, in the form of a tensor. In figure 3.2, this step, as well as steps 3 and 4, are only displayed for node v_4 to avoid cluttering, however, in reality this happens for all of the nodes in the graph.

3. Message aggregation: After message passing, each node aggregates all the incoming messages with an aggregation function ρ. This is usually simply summation or averaging of the incoming

messages, however, more complex aggregation functions such as an LSTM network (Hochreiter & Schmidhuber, 1997) can be used as well (Hamilton et al., 2017).

4. Updating node features: The aggregated messages are then used to compute new node features h_i^{l+1} for each node $v_i \in V$, which will now include information from neighboring nodes. This is done by feeding the aggregated messages through an update function ϕ. This update function ϕ is a NN which is shared over all nodes in the graph, i.e., the same NN, with the same parameters, is applied to update each node $v_i \in V$.

5. The final step is to return graph G as output, which now has updated node features h_i^{l+1} for each node $v_i \in V$.

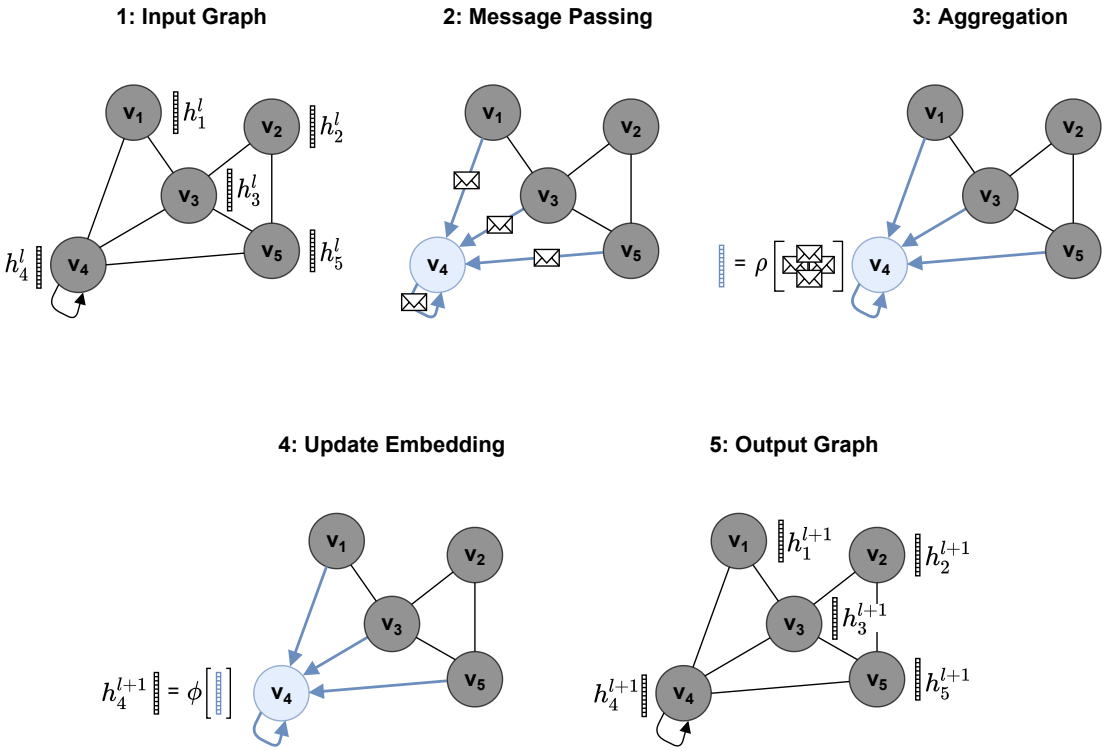


Figure 3.2: A visual breakdown of the 5 steps applied in a GNN layer. Step 1 shows the input graph, where each node $v_i \in V$ already has some node features h_i^l. Step 2-4 are only visualized for node v_4 but in reality happen for all nodes, resulting in an output graph where each node has updated node features h_i^{l+1}, as displayed in step 5. Step 2 applies message passing, which can be seen as sharing information between connected nodes. In step 3, this information is aggregated, and this aggregated information is then used in step 4 to update the features of a node. Finally step 5 returns an output graph with all the updated node features.

Usually a GNN consists of multiple layers of the type described above. When more consecutive layers are applied, information of nodes is spread further throughout the graph; after applying k layers, each node contains information from all nodes located in its k-hop neighborhood. Figure 3.3 shows how this looks like for 2 GNN layers applied to the example graph of figure 3.1. Here it is clear to see that in the second layer, node v_4 will incorporate information from all the other nodes in the graph, which are all in its 2-hop neighborhood.

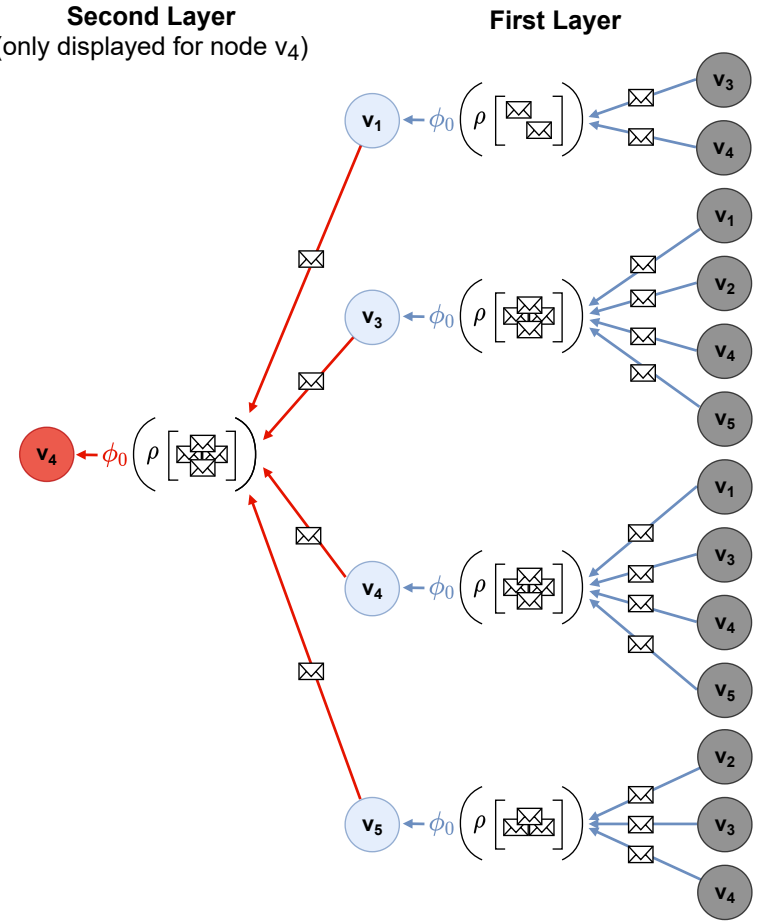


Figure 3.3: A visualization of the update of node features in two consecutive GNN layers applied to the graph of figure 3.1. The colors indicate the distinction between the different layers, where all the grey nodes represent the input graph, all blue elements represent the first layer and all the red elements represent the second layer. The second layer is only displayed for node v_4 to avoid cluttering.

In the case of node classification, the learned node features of the final GNN layer can be used directly to make an output prediction for each node. Usually, this is done by feeding the node features of each individual node into a MLP, which then classifies the nodes. However, for graph classification one needs to somehow summarize the entire graph to be able to use it as input for further MLP layers, which then compute the final output. To do so, a so called 'readout function' is applied. One simple approach would be to just concatenate all node features of the final GNN layer, but this does not allow for working with variable size graphs as input, which is an important feature of GNNs. Therefore, one usually applies a readout function that results in a fixed output size, regardless of the input size. Three commonly used readout functions are summation of the node features, averaging over node features and max-pooling over the features.

3.2.3. Graph Convolutional Networks

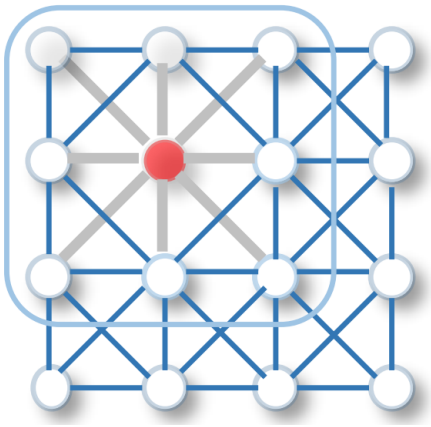
GCNs were introduced by Kipf and Welling (2017) and are based on CNNs. Normal CNNs cannot be applied to most graph shaped data, as they would require each node in the graph to have the same degree, such that the neighborhood of each node would contain an equal amount of nodes. Furthermore, they would require the neighborhood of each node to be an ordered set of nodes. This is because kernels of a CNN layer have a fixed size and learn a unique weight for each position in the kernel. Figure 3.4 visualizes this issue, where figure 3.4a shows how an image can be seen as a special type of grid-shaped graph, where each node represents a pixel (Wu et al., 2019). A 3x3 kernel of a CNN is displayed by the blue rectangle, which, in order to update the features of the red node, computes a weighted average over the node features of all nodes contained in the kernel, including the red node itself. A separate weight is learned and applied for each individual position within the

kernel, i.e., the 9 positions of the nodes that fall within the kernel. In figure 3.4b the concept of graph convolution is displayed. It is clear that the kernel from figure 3.4a would not work on this graph, as nodes have different degrees and the nodes in a neighborhood are unordered, making it impossible to apply the 9 learned weights of the CNN kernel to their corresponding position within the kernel. In order to update the features of the red node in figure 3.4b, a GCN layer takes a summation over the node features of the red node and its neighbors, where each node feature is weighted with the same weight matrix and a normalization constant is applied. Equation 3.3 shows how the features h_i of node v_i are updated in one layer l of a GCN:

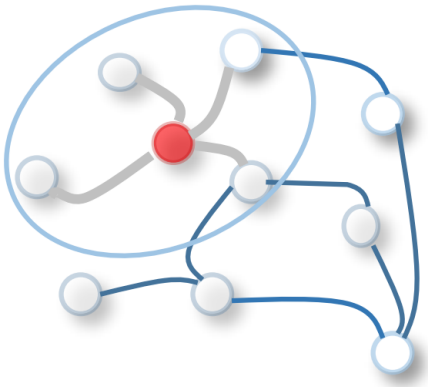
$$h_i^{(l+1)} = \sigma^{(l)}\left(\sum_{j \in \mathcal{N}_i} \frac{1}{c_{ij}} h_j^{(l)} W^{(l)}\right). \tag{3.3}$$

Here, $h_i^{(l+1)}$ are the updated node features, $\mathcal{N}_i$ is the neighborhood of node v_i, $W^{(l)} \in \mathbb{R}^{|\mathbb{h}_j^{(l)}| \times |\mathbb{h}_i^{(l+1)}|}$ are the learned weights and $\sigma^{(l)}$ is a non-linear activation function. Furthermore, c_{ij} is the following normalization constant: $c_{ij} = \sqrt{d_i d_j}$, where d_i is the degree of node v_i. Unlike a CNN layer, this works for any amount neighboring nodes and is independent of node ordering, therefore making GCNs permutation invariant. Equation 3.3 can be efficiently implemented by using sparse-dense matrix multiplications, as described in the orginal paper by Kipf and Welling (2017).

Tying this back to the general concept of GNNs introduced in section 3.2.2, the aggregation function ρ of a CNN is a summation of node features, combined with division by a normalization constant c_{ij}. The update function ϕ is one linear layer, with weights $W^{(l)}$.



(a) The convolutional operation of a CNN. A kernel is displayed by the blue rectangle, which, in order to update the node features of the red node, takes the weighted average over the node features of all nodes within the kernel, with an individual learned weight for each node in the kernel.



(b) The convolutional operation of a GCN. Instead of having a fixed kernel size, the kernel size of a GCN scales with the size of the neighborhood of the node of which the features are updated. For example, the features of the red node are updated by taking a summation over the features of all nodes in its neighborhood, including its own features, which are weighted with the same weight matrix and scaled by a normalization constant.

Figure 3.4: Comparison of convolution in a CNN layer and a GCN layer. This figure is obtained from (Wu et al., 2019).

3.2.4. Relational Graph Convolutional Networks

R-GCNs, introduced by Schlichtkrull et al. (2018), can be seen as an extension of GCNs, allowing for different types of edges, each encoding a different type of relation between nodes. Instead of one weight matrix $W^{(l)}$ per layer, a R-GCN layer has a separate weight matrix $W_r^{(l)}$ for each relation $r \in \mathcal{R}$, where $\mathcal{R}$ defines a set of relations. Furthermore, each node always has a self-loop for which a separate weight matrix $W_0^{(l)}$ is used, which can simply be seen as an additional relation for self-loops. Equation 3.4 shows how the features h_i of node v_i are updated in one layer l of a R-GCN:

$$h_i^{(l+1)} = \sigma^{(l)}\left(\sum_{r\in\mathcal{R}}\sum_{j\in\mathcal{N}_i^r}\frac{1}{c_{i,j}}W_r^{(l)}h_j^{(l)} + W_0^{(l)}h_i^{(l)}\right). \tag{3.4}$$

Here $\mathcal{N}_i^r$ defines the neighborhood of node v_i under relation r, i.e., all nodes that are connected to node v_i with an edge representing relation r. $W_r^{(l)} \in \mathbb{R}^{|\mathbb{h}_j^{(l)}|\times|\mathbb{h}_i^{(l+1)}|}$ represents the learned weight matrix for each relation r and $\sigma^{(l)}$ is again a non-linear activation function.

This addition of relations leads to a much stronger expressive power compared to the traditional GCNs where each edge was sharing the same weights. In the message passing paradigm used in section 3.2.2, one could describe this as follows: instead of treating all incoming messages equally, there are multiple types of messages which are each treated differently in the update of the node features.

3.2.5. Graph Attention Networks

GANs were introduced by Velicković et al. (2017) and apply multi-headed self-attention to aggregate node features from neighboring nodes. The used attention mechanism is a form of additive attention, as described by Bahdanau et al. (2014). Instead of aggregating all the node features of neighboring nodes with equal weight, as in GCNs, GANs attend over the neighboring nodes and learn separate weights for each node, i.e., GANs learn the individual importance of neighboring nodes. Using one single head, the features h_i of node v_i are updated in one layer l of a GAN as follows:

$$h_i^{(l+1)} = \sigma^{(l)}\left(\sum_{j\in\mathcal{N}(i)}\alpha_{ij}^{(l)}W^{(l)}h_j^{(l)}\right). \tag{3.5}$$

Here, $W^{(l)} \in \mathbb{R}^{|h_j^{(l)}|\times|h_i^{(l+1)}|}$ again represent learned weights that are used to transform the node features and $\alpha_{ij}^{(l)}$ are normalized attention coefficients e. These attention coefficients e are computed for all of the nodes in the neighborhood $\mathcal{N}_i$ of node v_i and are normalized by applying the softmax function over these coefficients:

$$\alpha_{ij}^{(l)} = \text{softmax}_j(e_{ij}) = \frac{\exp\left(e_{ij}^{(l)}\right)}{\sum_{u\in\mathcal{N}_i}\exp\left(e_{iu}^{(l)}\right)}, \tag{3.6}$$

where each attention coefficient is computed as follows:

$$e_{ij}^{(l)} = \text{LeakyReLU}\left(\mathcal{A}^{(l)}\left(W^{(l)}h_i^{(l)} \| W^{(l)}h_j^{(l)}\right)\right). \tag{3.7}$$

Here $W^{(l)}$ is the same weight matrix as the one used in equation 3.5 and the || symbol represent concatenation. Furthermore, $\mathcal{A} \in \mathbb{R}^{2|h_i^{(l+1)}|}$ represents the weights of a single fully connected layer, which functions as the attention mechanism. Lastly, LeakyReLU (Maas et al., 2013) is applied as non-linear activation function, which has a hyperparameter α_{LRLU} for its negative slope angle.

Equation 3.5 describes the working of one head of GAN, however, Velicković et al. (2017) found that using multiple heads is beneficial for a more stable learning process, as also discussed by Vaswani et al. (2017). The outputs of the individual heads can either be concatenated or averaged. In the case of averaging and using K attention heads, this would result in the following adjustment to equation 3.5:

$$h_i^{(l+1)} = \sigma^{(l)}\left(\frac{1}{K}\sum_{k=1}^{K}\sum_{j\in\mathcal{N}(i)}\alpha_{ij}^{(l,k)}W^{(l,k)}h_j^{(l)}\right), \tag{3.8}$$

where $\alpha_{ij}^{(l,k)}$ and $W^{(l,k)}$ denote the normalized attention coefficients and weights from the k-th attention head, respectively. Note that this is very similar to equation 3.4, describing the working of a R-GCN. Therefore, one could argue that each attention head can be seen as learning a unique relation.

3.3. Relational Inductive Biases

Mitchell (1980) argues that learning involves generalization from past experiences to deal with new but related problems and that this only seems to be possible under certain biases for choosing one generalization over another. In this thesis we will call these biases 'inductive biases' and define them as any type of bias that causes a learning algorithm to choose one form of generalization over another, independent of the observed data (Hamrick et al., 2018). These inductive biases can be seen as a set of explicit or implicit assumptions of a learning algorithm. Without these assumptions, generalization would not be possible, as there are many different possible ways of generalizing beyond the observed samples during training and the algorithm somehow has to favor one over the others (Hüllermeier et al., 2013). In DL, these inductive biases can, for instance, be encoded in the network architecture, used activation functions, optimization algorithm or the use of dropout (Srivastava et al., 2014). Ideally, these inductive biases lead to better generalization performance, but the induced biases can also be too strong or just plain wrong, leading to worse performance. Battaglia et al. (2018, p. 5) introduce a specific type of inductive biases - termed 'relational inductive biases' - which "*impose constraints on relationships and interactions among entities in a learning process*". In this case an entity can for instance be represented by a single pixel of an input image, but it can also represent an entire object in a more abstract encoding. Each of the weights of a NNs will then represent a 'relation' between two entities.

The general structure of deep neural network, consisting of multiple layers, already creates a form of relational inductive bias; hierarchical processing. For example, in CNNs, more layers typically lead to the inclusion of longer range relations. However, next to hierarchical processing, the individual layers within NNs can have their own relational inductive biases as well (Battaglia et al., 2018). These relational inductive biases of the individual layers is what we will focus on in this thesis and in the following sections we will discuss them for each type of NN layer used in this thesis.

3.3.1. Fully connected layers

A single fully connected layer, has no relational inductive biases. All neurons have all-to-all connections and therefore a weight is learned for each possible relation between the entities in the input. Furthermore, there are no shared weights, i.e., each relation has its own individual weight. Therefore, a linear layer does not impose any constraints on relationships among entities. See figure 3.5a for an illustration of these characteristics. As linear layers do not have relational inductive bias, they provide a good baseline for comparison with convolutional layers and GNN layers. Both of these methods apply a combination of parameter sharing and removing some of the weights that are present in a fully connected layer, to incorporate relational inductive biases. With respect to relational reasoning, parameter sharing can be though of as assuming that certain relations are equal or shared between some of the entities. Removing some of the weights can then be understood as removing some of the all-to-all relations and therefore having a bias towards more sparse relations between entities.

3.3.2. Convolutional layers

As opposed to a fully connected layer, a convolutional layer does have relational inductive biases. First of all, the relations they encode are local, only including entities within the size of the kernel. Therefore the kernels induce a strong relational bias, assuming that entities are only affected by other entities that are in close proximity. Furthermore, convolutional layers apply weight sharing - by using the same kernel weights at each position of the kernel during convolution over the input - which leads to translational equivariance. See figure 3.5b for a visual representation of these relational inductive biases. These biases make a lot of sense when applied to images, which is exactly the domain for which CNNs are often used. For an image, it is intuitively understandable that a patch of pixels that are spatially close to each other should also to be considered together in order to detect, for instance, an object in the image. Moreover, it also makes intuitive sense that detecting certain patterns at a specific location in the image is an equal task as detecting them at any other location in the image.

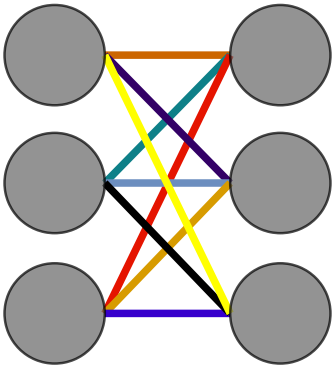
3.3.3. Graph convolutional layer

In GCN layers, the shape of the graph defines the relational inductive bias that is applied during learning. When there is an edge between two nodes, there is a relation between the two entities encoded by the nodes. Therefore, the relational inductive bias can be more dependent on the actual input, whereas in

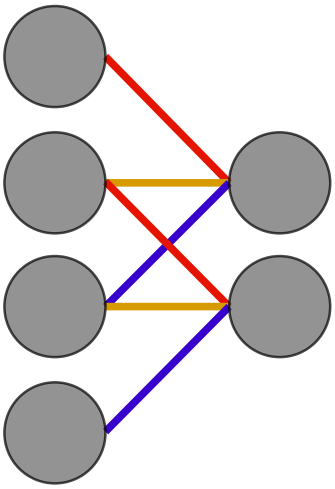
convolutional layers, the biases are always the same for each input and are solely dependent on the used kernels. In a GCN layer, edges can indicate all kinds of relations and are not necessarily present only between entities that are spatially close to each other. Because edges are defined between specific entities, rather than being based purely on position as in a convolutional layer, graph convolutional layers are permutation invariant. All edges in a graph convolutional layer share the same weights, therefore they only encode one type of relation between entities. This is a very strong bias which, as we will also show in the following sections of this thesis, might sometimes be too strong. See figure 3.5c for a visual representation.

3.3.4. Relational graph convolutional layer

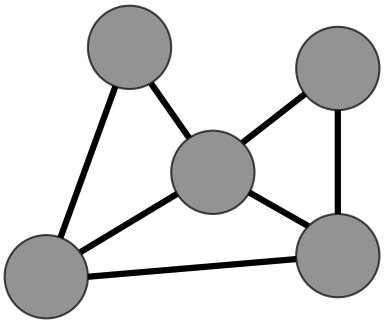
Just as a graph convolutional layer, a relational graph convolutional layer is also permutation invariant. However, relational convolutional layers can encode different types of relations with different types of edges. In this case, weights are only shared between edges that represent the same relation, allowing for much more detailed relational inductive biases than those of a graph convolutional layer, which shares weights across all edges. See figure 3.5d for a visual representation, in which the edges now encode 3 different types of relations, compared to only one type of relation in a GCN. This for instance enables us to replicate the relational inductive biases of both a fully connected layer - by encoding the input as a fully connected graph with a unique relation for each edge - as well as a CNN, which is something we will do in this thesis as well.



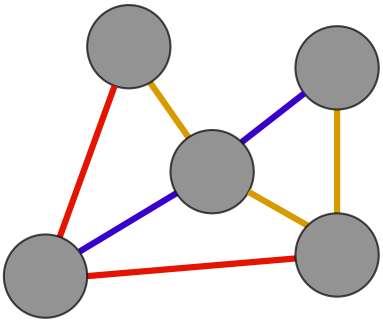
(a) Relational inductive bias of a MLP. MLPs have all-to-all relations, where each edge represents a unique relation, indicated by the different colors.



(b) Relational inductive bias of a CNN with 3×3 kernels. It has sparser relations than a MLP and some weights are shared.



(c) Relational inductive bias of a GCN, which allows for manually choosing the relations. All edges share weights and therefore represent the same type of relation.



(d) Relational inductive bias of a R-GCN. Different type of relations can be defined, displayed by the different edge colors. Edges with the same color share weights and therefore represent the same relation.

Figure 3.5: Comparison of relational inductive biases of a MLP, CNN, GCN and R-GCN. Edges with similar color share weights.

3.4. Reinforcement Learning

RL is one of three main machine learning approaches together with supervised learning and unsupervised learning. In RL one tries to let an agent learn a policy π - in the form of actions - that maximizes a certain reward signal (S.Sutton & G.Barto, 2018). More specifically, the agent interacts with an environment and receives a state s_t at each timestep t, representing the current state of the environment. The agent then takes an action a_t based on the observed state s_t, after which it receives a reward r_{t+1} as feedback, as well as a new state s_{t+1}. This process defines a repeating cycle of observing a state, taking an action and receiving a reward (Arulkumaran et al., 2017). See figure 3.6 for an overview of this process. The rewards the agent receives determine the best sequence of actions and, typically, an agent's goal is not to maximize the immediate reward r_t that it receives after taking an action at timestep t but rather to maximize the expected cumulative reward it will receive in the long-term by using its learned behavior in its environment (François-lavet et al., 2018). This is expected reward, as there might be stochasticity in the transitions between certain states upon taking an action - i.e., twice taking the same action from a certain state, does not always result in transitioning to the same next state - and therefore the cumulative reward is estimated with an expected reward. This is expected reward, since there may be stochasticity in the transitions between certain states upon taking an action, i.e., performing the same action twice from a certain state does not always result in transitioning to the same next state. Therefore, the cumulative reward is estimated with expected reward.

In the following sections, we will first explain and formalize the problem setting in which RL operates, after which some different methods of learning will be discussed, to then take the step to DRL and the algorithm used in this thesis: PPO.

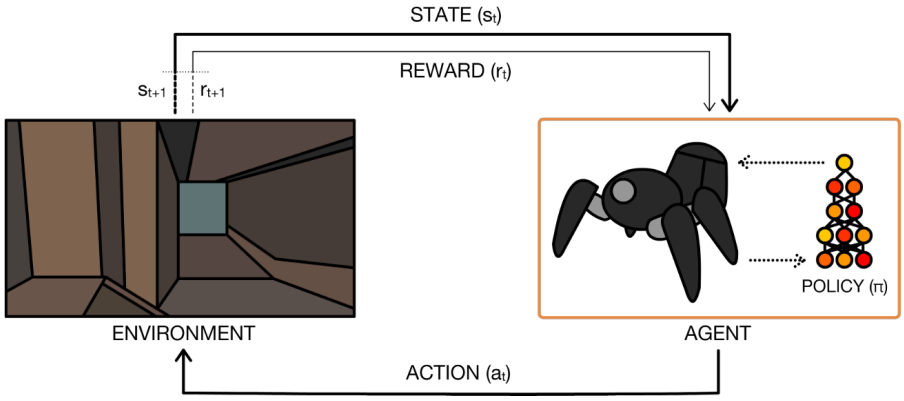


Figure 3.6: The interaction of a RL-agent with its environment. The agent receives a state s_t at each timestep t, based on which it takes an action a_t. After taking an action, the agent receives a reward r_{t+1} as feedback, as well as a new state s_{t+1}. This process defines a repeating cycle of observing a state, taking an action and receiving a reward. This figure is obtained from (Arulkumaran et al., 2017).

3.4.1. The reinforcement learning problem setting

The problems to which RL is applied are sequential decision making problems, in which actions not only influence direct rewards, but potentially also the possible rewards that are obtainable in the future states of the environment. This is because certain actions can determine which future states the agent gets to see. This problem can be formalized as a Markov Decision Process (MDP), which was originally introduced by Bellman (1957). A MDP is a 5-tuple (S, A, T, R, γ), where:

- S: Is a set representing the state space of the environment, i.e., all possible states s of the environment are an element of S (François-lavet et al., 2018).
- A: Is the action space of the problem which represents the set of possible actions that the agent can take.
- $\mathcal{T}(s_{t+1}|s_t, a_t)$: Is a transition function which takes a state s_t and action a_t as input and maps this to a probability distribution over states s_{t+1}.

- $R(s_t, a_t, s_{t+1})$: Is the reward function, indicating the reward r_t which the agent receives when transitioning from state s_t to state s_{t+1} after taking action a_t.
- $\gamma \in [0, 1)$: Is a discount factor, for which lower values indicate a stronger emphasis on immediate rewards (Arulkumaran et al., 2017). The use of this discount factor is further elaborated upon in equation 3.9.

An important underlying assumption of a MDP is that it has the Markov property, which entails that the future of a process does not depend on the past but solely on the current state. Therefore, the agent can base its decision to take action a_t at timestep t solely on state s_t instead of the full history of states.

Observability
When an environment enables the agent to observe the entire state s_t at each timestep t and not just an observation that gives the agent some partial information about the state of the environment, it is called a 'fully observable' environment. In this thesis all environments will be fully observable, however, it is worth noting that RL is also applied to partially observable environments where the agent receives an observation - e.g., some sensory input of a robot - which only partially describes the state that the agent is in.

Episodic tasks
If the task which the agent needs to complete, has a finite amount of timesteps, it is called an episodic task (S.Sutton & G.Barto, 2018). In this thesis all tasks will be episodic and T_e will be used to denote the number of timesteps in an episode. Here T_e will be equal to the amount of actions that the agent needs to take to solve the episode. Therefore, the entire sequence of state and actions in one episode will be as follows: $(s_0, a_0, s_1, a_1, \dots, s_{T_e-1}, a_{T_e-1}, s_{T_e})$, where s_{T_e} is a terminal state. Note that T_e may vary per episode. Contrary to episodic tasks, there are continuous tasks, such as a robot with a long life span operating in the real world, for which $T_e = \infty$.

Deterministic and stochastic policies
An agent's policy π dictates how the agent selects its actions based on the observed state s_t. These policies can be either deterministic - in which case the policy $\pi(s_t)$ is a function mapping a state s_t to an action a_t - or stochastic, for which the policy $\pi(s_t)$ is a function mapping a state s_t to a probability distribution over all possible actions, from which the agent samples its action. In this thesis all policies will be stochastic.

Return
The ultimate goal of a RL-agent is to learn a policy that maximizes the expected cumulative reward, also called return, which can be formalized for an episodic task as follows:

$$G_t = \sum_{k=t+1}^{T_e} \gamma^{k-t-1} r_k, \tag{3.9}$$

where G_t is the return from time step t onward. Equation 3.9 also shows the use of the discount factor γ defined in the MDP. The discount factor controls how much emphasis is placed on rewards in the near future versus rewards that are only obtained further ahead in time. The lower the value of γ, the more emphasis is put on returns in the near feature. Due the use of the discount factor, G_t is also referred to as discounted return.

Value functions
Now the return G_t is defined, we can also introduce a state-value function $V_\pi(s)$ for a given policy π. The state-value function $V_\pi(s)$ is equal to the expected return of following policy π from state s onward, which can be formally defined as:

$$V_\pi(s) = \mathbb{E}_\pi [G_t \mid s_t = s, \pi] \tag{3.10}$$

Based on this state-value function, it is possible to formally define what it means for a policy to be an optimal policy: a policy π^* is an optimal policy if and only if $V_{\pi^*}(s) \geq V_\pi(s)\ \forall s \in S$ and $\forall \pi \in \Pi$, where Π denotes the set of all possible policies (S.Sutton & G.Barto, 2018).

In addition to the state-value function, we also define an action-value function $Q_\pi(s,a)$ for a policy π, which is the expected return of taking action a in state s and from there on out following policy π:

$$Q_\pi(s,a) = \mathbb{E}_\pi\left[G_t \mid s_t = s, a_t = a, \pi\right] \tag{3.11}$$

The optimal action-value function $Q^*(s,a)$ is the action-value function for an optimal policy π^* and can be defined as:

$$Q^*(s,a) = \max_{\pi \in \Pi} Q_\pi(s,a). \tag{3.12}$$

Reversely, an optimal policy π^* can easily be obtained from $Q^*(s,a)$, by taking for each state $s \in S$, the action that yields the maximum action-value:

$$\pi^*(s) = \operatorname*{argmax}_{a \in A} Q^*(s,a)\ \forall s \in S. \tag{3.13}$$

To summarize, the goal of an agent is to find an optimal policy as defined in equation 3.13, which maximizes the expected cumulative return. However, finding such an optimal policy can be difficult. One common difficulty in RL is that, due to the nature of the RL problem, there will always be a trade-off between exploring new policies in the hope of finding a better one and exploiting ones that already yield good performance. This is commonly known as the 'exploration-exploitation trade-off'. Another common complication is the credit-assignment problem, which is the problem of determining - in a successful episode - how to assign credit over the possibly many actions that led to the success (Minsky, 1961). This is partially controlled by γ, but can remain a difficulty, especially in environments with very sparse rewards.

3.4.2. Methods of learning in RL

Before proceeding to deep reinforcement learning, some different methods of learning will be introduced and discussed. First of all, a segmentation can be made into value-based methods, policy-based methods and actor-critic methods:

1. Value-based: These methods aim to learn a value function and pick actions based on this value function. This is usually the action-value function.

2. Policy-based: In policy based methods, one tries to directly learn a policy instead of first learning a value function and using the value function to pick actions. Typically, policy-based methods - especially when used with a function approximator - are more stable but less sample efficient than value-based methods (Nachum et al., 2017).

3. Actor-critic: These methods can be seen as a combination of value-based and policy-based learning. An 'actor' tries to learn the policy and uses feedback from the 'critic' to do so, where the critic is a learned value function.

Model-based vs. model-free

Another division can be made between model-based and model-free methods. All the methods mentioned above can be used in both a model-based and model-free way. Model-based methods attempt to learn a model of the environment - in the form of an estimation of the transition function and reward function - which is then used to define the agent's policy (François-lavet et al., 2018). In this thesis, however, we focus on model-free methods which do not learn a model of the environment.

Off- and on-policy learning

A final differentiation between methods is that of off- and on-policy learning methods. On-policy methods interact with the environment using the same policy as the one being optimized. Off-policy methods, however, take actions in the environment which are not based on the actual policy that is being optimized. They could for instance collect experience in the environment based on random actions. In the remainder of this thesis we will focus on on-policy methods.

3.4.3. Deep Reinforcement Learning

Based on the previously introduced background on RL and the possible methods of learning in RL, it is relatively easy to make the step to DRL. DRL is RL in which deep learning algorithms are applied, which is typically done by using a NN as a function approximator for the policy and/or value function. In addition, a NN can be used in model-based approaches as an approximator for the model of an environment. The use of NNs makes it possible to apply RL to problems with high-dimensional state and action spaces, which was infeasible before the use of NNs (Arulkumaran et al., 2017). Although the combination of RL and NNs had previously been used, for instance to learn to play the game of backgammon (Tesauro, 1995), the DRL approach only truly gained in popularity after advances in the field of DL enabled Mnih et al. (2015) to use CNNs to train an algorithm to play Atari games directly from pixel data.

The DRL method used in this thesis is PPO, which can be seen as an on-policy actor-critic method. Before introducing the details of PPO, we will first introduce the more general concept of policy gradient methods and discuss how these can be translated into an actor-critic method, on which PPO is based.

Policy gradient methods

As the name suggests, policy gradient methods fall under the policy-based methods of RL. In this case, the learned policy is stochastic and is represented by a NN with parameters θ. It is therefore called a parameterized policy π_θ, which is defined as follows: $\pi_\theta(a|s,\theta) = \Pr\{a_t = a|s_t = s, \theta_t = \theta\}$. This parameterized policy dictates the probability of taking action a at timestep t with the current parameters θ of the NN. Furthermore, it is assumed that the policy π_θ is differentiable with respect to its parameters (S.Sutton & G.Barto, 2018).

As always, a loss function is needed to train the NN. However, in this case we actually speak of a performance function, which should be maximized rather than minimized, and therefore Stochastic Gradient Ascent (SGA) is applied instead of SGD. This performance function will be denoted by $J(\theta) = V_{\pi_\theta}(s_0)$, where V_{π_θ} is the true value function for policy π_θ and s_0 is the first state of an episodic task or environment. Therefore, the performance measure is equal to the expected cumulative reward that can be obtained from this start state (S.Sutton & G.Barto, 2018). As with a loss function in DL, the gradient of the performance function $J(\theta)$ with respect to its parameters θ is computed with an automatic differentiation package - in our case PyTorch (Paszke et al., 2019) - and is denoted by $\nabla_\theta J(\theta)$. Based on this gradient, SGA can be applied to optimize the parameterized policy π_θ. Applying the policy gradient theorem - which will not be discussed in detail here, but background information on this topic can be found in (S.Sutton & G.Barto, 2018) - results in the following definition for the gradient of the performance function:

$$\nabla_\theta J(\theta) = \mathbb{E}\left[\nabla_\theta \log \pi_\theta(a|s,\theta) Q_{\pi_\theta}(s,a)\right] , \tag{3.14}$$

where $\log \pi_\theta(a|s,\theta)$ is the log probability of taking action a in state s under the current parameterized policy π_θ. This can be rewritten into the following form as well:

$$\nabla_\theta J(\theta) = \mathbb{E}\left[\nabla_\theta \log \pi_\theta(a|s,\theta) G_t\right] . \tag{3.15}$$

As $\mathbb{E}$ denotes an expectation, one can approximated the gradient by computing an empirical average (Schulman et al., 2017). Computing such an empirical average can be achieved by sampling a batch of trajectories $\tau \in \mathcal{B}$ under the current policy (Duan et al., 2016). Each trajectory is a sequence of states and actions of length T, where length T again represents the amount of actions that the agent takes. A trajectory starts from a start state s_0 and from there on out, actions are sampled from the policy π_θ, leading to the following type of sequence: $\tau = (s_0, a_0, s_1, a_1, \ldots, s_{T-1}, a_{T-1}, s_T)$. To indicate that an approximation of the gradient is used, instead of the actual gradient, the following notation will be used: $\widehat{\nabla_\theta J(\theta)}$.

Directly applying the methods described above to estimate the gradient and then using this estimated gradient to optimize the policy π_θ, may suffer from high variance and therefore result in slow learning. To reduce this problem, a baseline can be subtracted from the policy gradient. An often used baseline is the state-value function $V_\pi(s)$, which can be subtracted from the action-value function $Q_\pi(s,a)$ to form the advantage function:

$$A^\pi(s,a) = Q_\pi(s,a) - V_\pi(s), \tag{3.16}$$

where $A^{\pi}(s,a)$ indicates for state s how good or bad a certain action a is, relative to the expected performance of policy π from state s. With this added baseline, the policy gradient shown in equation 3.14 would now have the following form:

$$\nabla_\theta J(\theta) = \mathbb{E}\left[\nabla_\theta \log \pi_\theta(a|s,\theta) Q_{\pi_\theta}(s,a) - V_{\pi_\theta}(s)\right] = \mathbb{E}\left[\nabla_\theta \log \pi_\theta(a|s,\theta) A^{\pi_\theta}(s,a)\right] \tag{3.17}$$

This formulation of the policy gradient lends itself to an intuitive explanation of what happens in an update step with SGA: if an action leads to a positive advantage, i.e., it is better than the average performance of the policy, the policy gradient will be positive and therefore the probability of taking this action will be increased by making an update with SGA. Vice versa, if the advantage of action a is negative, the gradient will be negative and thus the probability of taking action a will be decreased by an update step with SGA.

The actor-critic method in DRL

The policy gradient defined in equation 3.17, introduces a perfect use case for actor-critic methods, because it not only uses a learned policy π_θ - being the actor - but also requires a state-value function $V_{\pi_\theta}(s)$ to compute the advantage function. This state-value function can also be parameterized and represented by a NN, which will form the critic of the setup. Usually this critic will have different - or at least partially different - parameters than the parameters θ of the actor. Therefore, the critic will be denoted by $V^{\alpha}_{\pi_\theta}$, where α stands for the parameters of the critic. Note that the policy is still dependent on θ, as it is defined by the actor, which is also called 'policy network'. To train the critic - also called 'value network' - an additional loss or objective function is required. A common approach is to use the MSE between the actual returns and the values predicted by the critic as a loss function. To compute this loss, the same batch of trajectories $\tau \in \mathcal{B}$ used to compute the expectations in equations 3.14, 3.15 and 3.17, can be used in the following manner:

$$L^{VF}(\alpha) = \frac{1}{|\mathcal{B}|\,T} \sum_{\tau \in \mathcal{B}} \sum_{0}^{T-1} \left(V^{\alpha}_{\pi_\theta}(s_{\tau,t}) - G_{\tau,t}\right)^2 . \tag{3.18}$$

The loss function $L^{VF}(\alpha)$ - where VF stands for value function - can then be applied with SGD to train the critic. Note that for the critic we apply gradient descent as we want to minimize the loss as opposed to the performance function, for which the goal is maximization.

Advantage estimation

There are multiple methods to now use the critic to compute an estimate of the advantage function. In this thesis Generalized Advantage Estimation (GAE) (Schulman et al., 2016) is used to estimate the advantage for each timestep t of a trajectory of length T as follows:

$$\widehat{A^{\pi_\theta}_t} = \sum_{k=t}^{T} (\gamma\lambda)^{k-t} \delta_k^{V^{\alpha}_{\pi_\theta}} . \tag{3.19}$$

In this equation, $\widehat{A^{\pi_\theta}_t}$ is the estimated advantage at timestep t, γ is the usual discount factor, and $\lambda \in [0,1]$ is a new parameter introduced to control the trade-off between bias and variance. A high value for λ results in high variance, whereas a low value for λ leads to more bias and lower variance. The $\delta_{t+l}^{V^{\alpha}_{\pi_\theta}}$ term in equation 3.19 is defined as follows:

$$\delta_{t+l}^{V^{\alpha}_{\pi_\theta}} = -V^{\alpha}_{\pi_\theta}(s_t) + r_t + \gamma V^{\alpha}_{\pi_\theta}(s_{t+1}) \tag{3.20}$$

Equation 3.20 shows how GAE uses the output of the critic network $V^{\alpha}_{\pi_\theta}(s_t)$, together with the actual reward r_t of each timestep, to estimate the advantage function. In code, equation 3.19 can be easily implemented in a recursive fashion.

Example policy gradient algorithm in an actor-critic setup
Bringing it all together, one could define a simple policy gradient algorithm, using an actor-critic setup. Here the actor is the parameterized policy and the critic is the parameterized state-value function, used to compute an estimate of the advantage. An example is shown below, in the pseudo-code of algorithm 1.

Algorithm 1 Policy Gradient Algorithm, Actor-critic set-up

1: Initialize parameters θ and α of the actor and critic networks
2: **for** iteration = $0, 1, 2, ...$ **do**
3: **for** trajectory $\tau \in \mathcal{B}$ **do**
4: Execute policy π_θ for T timesteps.
5: Keep track of the visited states $s_{\tau,t}$, taken actions $a_{\tau,t}$, received rewards $r_{\tau,t}$, predicted values $V^{\alpha}_{\pi_\theta}(s_{\tau,t})$ and the log probabilities $\log \pi_\theta(a|s_{\tau,t}, \theta)$. [1]
6: Compute and store returns $G_{\tau,t}$ and advantages $\widehat{A^{\pi_\theta}_{\tau,t}}$ for each timestep t in trajectory τ.
7: **end for**
8: Estimate the policy gradient:

$$\widehat{\nabla_\theta J(\theta)} = \frac{1}{|\mathcal{B}|\,T} \sum_{\tau \in \mathcal{B}} \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_{\tau,t}|s_{\tau,t}, \theta) \widehat{A^{\pi_\theta}_{\tau,t}}$$

9: Compute the loss for the critic:

$$L^{VF}(\alpha) = \frac{1}{|\mathcal{B}|\,T} \sum_{\tau \in \mathcal{B}} \sum_{t=0}^{T-1} \left(V^{\alpha}_{\pi_\theta}\left(s_{\tau,t}\right) - G_{\tau,t}\right)^2,$$

10: Update actor parameters θ with SGA
11: Update critic parameters α by computing the gradient $\nabla_\alpha L^{VF}(\alpha)$ and applying SGD
12: **end for**

This pseudo code is inspired by the 'Vanilla Policy Gradient' implementation of Achiam (2018), as well as the work by Schulman (2016b) and Duan et al. (2016).

3.4.4. Proximal Policy Optimization

One of the problems of basic policy gradient methods, such as the one introduced in algorithm 1, is that they have poor data efficiency as each batch of trajectories is only used once. A tempting solution would be to update the parameters of the critic and actor not just once with the batch of collected trajectories, but to perform multiple update steps on the same batch. However, updating the policy parameters θ multiple times with the same batch of trajectories - or only updating them once with a too high learning rate - can cause the updated policy $\pi_{\theta_{\text{new}}}$ to diverge too far from the old policy π_θ. In practice this can lead to instability leading to big drops in performance or even complete 'unlearning' of good policies (Schulman et al., 2017).

PPO - introduced by Schulman et al. (2017) - is a method that does allow for multiple update steps on the same batch of trajectories. It does so by applying a surrogate performance function that is intended to prevent the updated policy $\pi_{\theta_{\text{new}}}$ from deviating too far from the old policy π_θ under which the batch of trajectories was collected. Here 'surrogate' has the same meaning as for a surrogate loss function, as introduced in section 3.1. For a specific timestep t, the surrogate performance function of PPO is defined as follows:

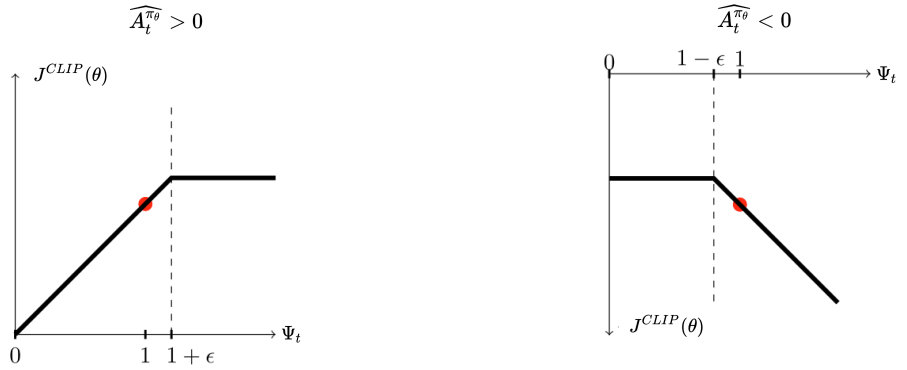
$$J^{CLIP}_t(\theta) = \mathbb{E}\left[\min\left(\Psi_t(\theta)\widehat{A^{\pi_\theta}_t}, \text{clip}\left(\Psi_t(\theta), 1-\epsilon, 1+\epsilon\right)\widehat{A^{\pi_\theta}_t}\right)\right], \tag{3.21}$$

where $\Psi_t(\theta)$ defines a probability ratio between the new and the old policy, as a measure of difference between the two policies:

[1] Here $s_{\tau,t}$, $a_{\tau,t}$ and $r_{\tau,t}$ stand for the state, action and reward at timestep t of trajectory τ, respectively.

$$\Psi_t(\theta) = \frac{\pi_{\theta_{\text{new}}}(a_t|s_t)}{\pi_\theta(a_t|s_t)} \,. \tag{3.22}$$

The second term inside the min operation - $\text{clip}\left(\Psi_t(\theta), 1-\epsilon, 1+\epsilon\right) \widehat{A_t^{\pi_\theta}}$ - clips the probability ratios Ψ_t between $1-\epsilon$ and $1+\epsilon$, where ϵ is a hyper parameter of PPO. This clipping aims to remove the incentive, encoded in the performance function, for updating the policy to the point that Ψ_t falls outside the interval $[1-\epsilon, 1+\epsilon]$. Note that this is not a hard constraint, but just an adjustment to the performance function. To explain this more intuitively, figure 3.7 is used, which displays the surrogate performance function $J^{CLIP}(\theta)$ as a function of Ψ_t, for one single timestep t. Figure 3.7a shows this for cases with a positive advantage $\widehat{A_t^{\pi_\theta}}$ and figure 3.7b shows this for cases with a negative advantage. The red dot on the lines indicates the ratio Ψ_t at the beginning of the first update step, where the old and new policy are still equal and therefore $\Psi_t = 1$.



(a) Positive advantage case.

(b) Negative advantage case.

Figure 3.7: Visualization of the the surrogate performance function of PPO for one single timestep t and two cases: for positive advantage values and for negative advantage values. This figure is obtained from (Schulman et al., 2017), however, some symbols are adjusted for consistency of notation with the rest of this thesis.

In figure 3.7a, the advantage is positive, thus the picked action a_t is favorable over the average performance of the current policy π_θ. Therefore, when the critic network is updated by a step of SGA, the probability of taking action a_t will be increased for the updated policy $\pi_{\theta\text{new}}$. Doing so will increase the value of the ratio $\Psi_t(\theta)$, which also leads to a higher value in the performance function. If this would be repeated for many update steps, or if one update step already leads to a big increase of $\Psi_t(\theta)$, the policy update could become too large. However, in PPO, the ratio $\Psi_t(\theta)$ will be clipped when the threshold of $1+\epsilon$ is reached, as $\text{clip}\left(\Psi_t(\theta), 1-\epsilon, 1+\epsilon\right) \widehat{A_t^{\pi_\theta}}$ then yields a lower value than $\Psi_t(\theta)\widehat{A_t^{\pi_\theta}}$ and is thus chosen by the min operation. As a result, the curve representing the performance function relative to the ratio $\Psi_t(\theta)$ will flatten off, indicating that there is now no incentive to further increase the ratio $\Psi_t(\theta)$ by increasing the probability of action a_t again in the next update steps.

Figure 3.7b shows the case in which the advantage is negative, i.e., the action a_t is unfavorable over the average performance of the current policy π_θ. Because of that, the probability of taking action a_t will now be decreased for the updated policy $\pi_{\theta\text{new}}$, leading to a smaller value for ratio $\Psi_t(\theta)$ and therefore also to a smaller negative value of $J^{CLIP}(\theta)$. Again, doing this for multiple update steps could cause an undesirably large update of the policy, which PPO tries to avoid. This time the ratio will be clipped when the threshold of $1-\epsilon$ is reached, as $\text{clip}\left(\Psi_t(\theta), 1-\epsilon, 1+\epsilon\right) \widehat{A_t^{\pi_\theta}}$ yields a larger negative value than $\Psi_t(\theta)\widehat{A_t^{\pi_\theta}}$ and is therefore selected by the min operator. As a result, the performance function relative to the ratio $\Psi_t(\theta)$ will again flatten out when the updated policy starts to diverge too much from the old policy, removing the incentive to push them even further apart.

Note that, due to the min operation, the ratio $\Psi_t(\theta)$ is never clipped when the change in the probability ratio has a negative effect on the performance function. For example, in figure 3.7b a large ratio $\Psi_t(\theta)$ is not clipped for a negative advantage value. This has an interesting function: if the ratio $\Psi_t(\theta)$ is large and the advantage is negative, it means that a bad action has a larger probability of being chosen under the new policy $\pi_{\theta_{\text{new}}}$ than under the old policy π_θ. This is clearly a bad ting and therefore

it is good to allow for a large update step - proportional to the ratio $\Psi_t(\theta)$ - to reduce the probability of taking this action again.

PPO is often used with an actor-critic setup in which the parameters of the two networks are (partially) shared. The complete performance function of PPO therefore looks as follows:

$$J^{CLIP+VF+S}(\theta) = \mathbb{E}\left[J^{CLIP}(\theta) - c_1 L^{VF}(\theta) + c_2 S_{ent}\left(\pi_\theta\right)\right] . \tag{3.23}$$

Here θ represents the shared parameters between the actor and critic networks, $J^{CLIP}(\theta))$ is the performance function for the actor and $L^{VF}(\theta)$ is the loss function for the critic. Note that the critic loss is subtracted as $J^{CLIP+VF+S}$ is defined as a performance function, which one tries to maximize during training. Furthermore, an entropy bonus $S_{ent}(\pi_\theta)$ is added, which can be used to encourage the agent to explore more. A high entropy indicates a policy that, for a given state s, picks relatively randomly from the possible actions $a \in A$, whereas a policy with low entropy often picks the same action. By adding the entropy of the policy to the performance function, the agent is incentivized to adopt a policy with a higher entropy, thus choosing actions more randomly and therefore exploring more. Both c_1 and c_2 are hyperparameters, controlling how much the critic loss and entropy bonus are factored into the overall performance function.

Algorithm 2 shows pseudocode for an example implementation of PPO. Just as the policy gradient method introduced in algorithm 1 of section 3.4.3, batches of trajectories $\tau \in \mathcal{B}$ of length T are collected and used to update the actor and critic networks. However, now the networks are not just updated once with SGA, but are updated over multiple epochs of mini-batch-SGA. Each of these mini-batches has a size of $M \leq |\mathcal{B}|\, T$ and one epoch covers all experiences collected in the batch of trajectories. Furthermore, these mini-batches are randomly sampled from the entire batch $\mathcal{B}$ of experiences. This is done to give each mini-batch a variety of experiences, which do not solely come from successive steps in the environment, resulting in more stable training.

Algorithm 2 PPO, Actor-Critic set-up

1: Initialize parameters θ of the actor and critic networks
2: **for** iteration = $0, 1, 2, \ldots$ **do**
3: **for** trajectory $\tau \in \mathcal{B}$ **do**
4: Excecute policy π_θ for T timesteps.
5: Keep track of the visited states $s_{\tau,t}$, taken actions $a_{\tau,t}$, received rewards $r_{\tau,t}$, predicted values $V^{\alpha}_{\pi_\theta}(s_{\tau,t})$ and the probabilities $\pi_\theta(a|s_{\tau,t}, \theta)$.
6: Compute and store returns $G_{\tau,t}$ and advantages $\widehat{A^{\pi_\theta}_{\tau,t}}$ for each timestep t in trajectory τ.
7: **end for**
8: Optimize the surrogate performance function $J^{CLIP+VF+S}(\theta)$ with respect to θ, with N_{epochs} epochs of training with minibatch-SGA.
9: **end for**

3.5. Summary and conclusion

After having introduced all the relevant background theory and terminology, we will shortly summarize how this is applied throughout the rest of this thesis. The core idea is that we apply PPO as a RL method to solve a navigation task, however, instead of using standard NN architectures like MLPs and CNNs to form the actor and critic network, we now apply and compare multiple types of GNNs. The idea behind applying these GNNs - especially R-GCNs - is that they allow us to encode stronger relational inductive biases that are more problem specific than those of a MLP or CNN. Ideally, these relational inductive biases lead to a form of relational reasoning, comparable to how humans reason, which improves learning performance and generalization. The exact implementation and the used evaluation methods will be discussed in the following sections.

4

Related work

In this section we will introduce and discuss some recent work that is related to this thesis. This is by no means an exhaustive list of all related work, but it aims to give an overview of some of the most relevant and comparable work.

4.1. Relational reinforcement learning

From a broader perspective, our methods can be seen as a form of Relational Reinforcement Learning (RRL) in a DRL setting. RRL can be described as the combination of reinforcement learning and 'relational learning' (Džeroski et al., 2001a; Getoor & Taskar, 2007), where relational learning is also researched under the name of 'inductive logic programming' (Muggleton & de Raedt, 1994; Železný & Lavrač, 2008). RRL transforms RL to work with states and actions that are represented as relations (Džeroski et al., 2001b). More specifically, Džeroski et al. (2001a) define RRL as RL where states, actions and policies are represented in a relational fashion and background knowledge and declarative biases are used during learning. These declarative biases are biases which define how the policies are described. There is a strong similarity between this definition of RRL and our methods, in which background knowledge is used to define certain relations and the relational inductive biases induced by our network architectures can be loosely compared to the declarative biases of RRL.

4.2. Reinforcement learning with GNNs

The most tightly related research to this thesis can be categorized as reinforcement learning with graph neural networks. In recent years, graph neural networks have gained in popularity and are now also applied to RL. This combination is implemented with both varying RL and GNN methods. Even though this is not mutually exclusive, nor exhaustive, we divide the related work in which this combination of RL and GNN is applied, into 2 categories: first, methods that encode the observations of the agent as graphs, and second, methods that encode the physical shape of the agent itself as graph. There is also research into encoding multi-agent systems as graphs and applying GNNs to them, such as, for instance, the work by J. Jiang et al. (2020). However, as this thesis does not consider multi-agent systems, these are not discussed here.

4.2.1. Observations encoded as graphs

The most comparable work to this thesis - which was actually published during the time span of this thesis project - is the work by Z. Jiang et al. (2021), who introduce a method under the name 'Grid-to-Graph'. In this method, observations are encoded as graphs, to which R-GCNs are applied, as in this thesis. Grid-to-Graph shows improved in- and out-of-distribution generalization on several tasks, including some simple visual navigation tasks. Even though this work is comparable, there are numerous distinctive elements to this thesis: our methods are applied to a different problem set with different domain-specific relations, in-distribution generalization is tested with a different method, a different RL algorithm is employed and there are many implementation differences. Furthermore, we extend this research by trying to learn the domain specific relational inductive biases with R-GCN$_{\text{GAN}}$. In addition,

we test against some other baseline architectures, report results for multiple readout functions and also implement these readout functions for the CNN baseline architecture for a fair comparison with the R-GCNs. Finally, it is evaluated whether the specific relational inductive biases we apply - based on domain knowledge - are better than other, more random, relations.

This thesis is partially inspired by Zambaldi et al. (2018), who present another comparable method under the name 'Relational Deep Reinforcement Learning'. Their method is tested - among other environments - on a visual navigation task comparable to the Key-Corridors environments. The Relational Deep Reinforcement Learning method can be seen as encoding observations as fully connected graphs, on which multi-head dot-product attention (Vaswani et al., 2017) is applied. The multi-head attention approach is somewhat comparable to R-GCN, as each attention head can theoretically learn a different set of edges, with their own weights, which can represent a certain type of relation. Our methods differ in the fact that we apply domain specific relational edges, which are based on domain knowledge, whereas Zambaldi et al. (2018) try to learn all of these edges with attention during training. However, for their methods it is assumed that (x, y) positions of the nodes - or grid cells - are available. This is an assumption that we do not make and therefore their method is not directly applicable to our problem set. The general idea and results of Relational Deep Reinforcement Learning are promising though, with improved sample efficiency and generalization over baseline algorithms.

The work of Lu et al. (2021) shows an example application of RL with GNNs on a large scale, real-world, navigation task. Lu et al. (2021) propose using an abstract map of the environment, modeled as a Markov network (Koller & Friedman, 2009). They reason that, in real world scenarios, there is often a certain regularity in the relative positions of objects, e.g., a chair often being next to a desk. This regularity can be modeled as a probability distribution with a random vector describing the positions of the objects and a joint probability representing the relative position of the objects. This is what Lu et al. (2021) model as the Markov network, to which a GNN is applied during learning. They find that this approach improves the success rate of navigation in novel environments and therefore again shows good generalization capabilities.

Hamrick et al. (2018) explore a problem in which an agent is given the task of stabilizing an unstable tower of blocks by applying glue between some of these blocks. The more blocks remain standing, the higher the reward, but there is also a cost associated with applying glue. The tower of blocks is again modeled as a graph, where both the position and the orientation of the blocks are encoded as nodes and the presence of glue between to blocks is encoded as edges. However, in this setup the RL agent learns an action for each edge, which reflect whether the agent applies glue between two blocks or not, as well as a global action for indicating that the glueing phase is finished. A setting with fully connected graphs and one with more sparse graphs - with edges between blocks that are in contact with each other - were compared against each other. Both topologies gave accurate results, but the sparse topology resulted in more efficient learning. Furthermore, good generalization performance over differently sized towers was recorded, which was much better than baseline architectures.

Another interesting approach is to apply the combination of GNNs and RL on problems that are inherently graph-shaped, as is done by Almasan et al. (2019) on a traffic routing optimization task. This is in contrast to our problem domain, which is grid-shaped and actually has to be formed into a graph with the use of domain knowledge. Their routing tasks form an interesting candidate for GNN-based methods, especially when trying to generalize to unseen graph topologies, which is difficult to do with traditional architectures such as MLPs or CNNs. In this case, a GNN based on Message Passing Neural Networks (Gilmer et al., 2017) is applied and the results again show strong generalization performance.

H. Wang et al., 2020 try to tackle the problem of automatic transistor sizing in circuit design by applying the combination of GCNs and RL. In their work, the actual circuit is modeled as a graph, in which the nodes represent transistors and edges represent wires. This problem is again, like the previous work by Almasan et al. (2019), inherently graph shaped. This method was tested against several baselines, including evolutionary algorithms, human experts and non graph-based reinforcement learning, and consistently outperformed these baselines. In addition, the method was applied in a transfer learning setup, which yielded good results, again indicating good generalization performance.

4.2.2. Modeling the physical shape of an agent as graph

In some some other works, the physical shape of an agent is modeled as a graph, to which GNNs are applied. This is not strictly different from modeling observations as graphs, as the shape of an agent can actually be seen as part of the environment and these type of agents often receive observations

in the form of sensory input for each physical element of the agent. For example, a human shaped robot could receive information about the position and orientation of all of its body parts and joints, which together can represent the entire observation or part of it. Although these works are not entirely different from encoding observations as graphs, they do provide an interestingly different perspective on the application of graph neural networks in reinforcement learning. s T. Wang et al. (2018) introduce NerveNet as an approach to learn structured RL policies with GNNs. They model the physical shape of an agent into a graph and learn actions for the different parts of the agent. See figure 4.1 for a visual representation.

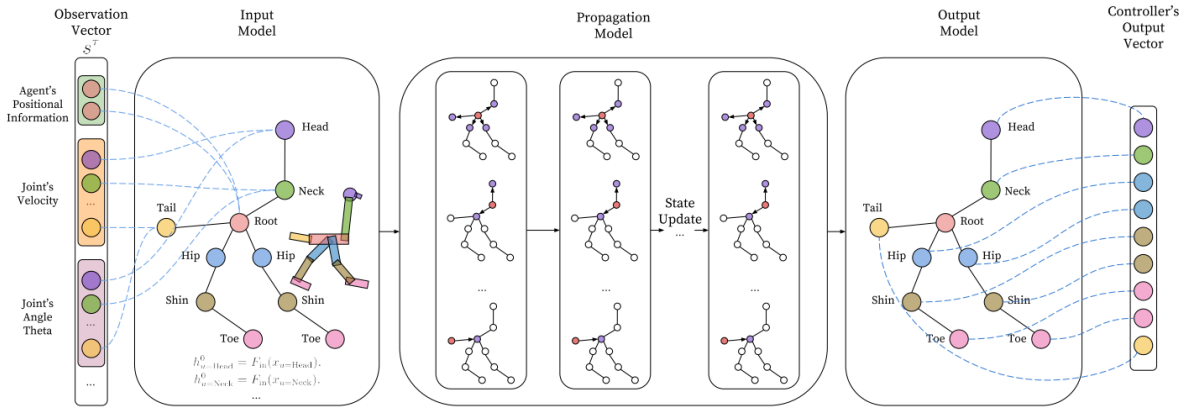


Figure 4.1: A visual overview of the NerveNet approach. This figure shows how the 'Walker-Ostrich' agent is encoded as a graph, using different parts of the observation vector for the features of different nodes. After the graph is constructed, several GNN layers are applied. The output consists of a 'controller' for each body part, which together form the policy of the agent. This figure is obtained from (T. Wang et al., 2018).

T. Wang et al. (2018) show that NerveNet achieves comparable results to state-of-the-art methods in MuJoCo environments (Todorov et al., 2012). They further demonstrate that NerveNet has significantly better transfer learning and generalization performance than other models and even show that NerveNet sometimes works for zero-shot transfer learning. To do so, they test for two types of transfer problems: size transfer and disability transfer. Size transfer refers to training on a smaller agent and then transferring to a larger agent and disability transfer tests scenarios where a policy is trained for one agent and is then used on the same agent but with some body parts disabled.

Huang et al. (2020) apply a comparable method to NerveNet but jointly train a set of several different agents with different morphologies (i.e., body shapes), which consist of similar elements (i.e., body parts). They show that this approach allows for generalizing to new agent morphologies, unseen during training.

Although the physical shapes of agents might form an intuitive graph topology, Kurin et al. (2020) show that these intuitive topologies are not always the best ones to use during training with GNNs. In addition, they introduce 'Amorpheus', a method based on transformers (Vaswani et al., 2017). This method can be seen as applying GNNs on fully connected graphs and using attention to learn which messages - and with what weight - to aggregate. This therefore shows some resemblance to the methods applied by Zambaldi et al. (2018), however, applied to a completely different domain. Whereas Zambaldi et al. (2018) apply this to an object-oriented state space, Kurin et al. (2020) apply this to the different elements of an agent. Their results show that, Amorpheus substantially outperforms the GNN-based approaches which encode the physical shape of the agent into the graph topology.

4.3. Summary and conclusion

The combination of RL and GNNs can be seen as a form of RRL and is used in varying approaches and applied to different types of problems, by various scholars. In general, this often leads to improved generalization performance when compared to methods that do not use GNNs. However, R-GCNs are not often applied and using them to incorporate domain knowledge in navigation tasks is, to our knowledge, only investigated in the recently published study by Z. Jiang et al. (2021). This is the application domain that we focus on in this research.

5

Methodology

This chapter explains the methods adopted for this study. First, the experimental setup will be introduced. Following this, we will discuss the Key-Corridors environments used to test our proposed methods on, after which we introduce the evaluation methods used to test how well our methods perform, both in terms of sample efficiency and generalization. Next, the different architectures used in this thesis will be explained in detail. These are the GNN-based architectures we propose in order to combine relational reasoning with RL, as well as the architectures that are used as baselines. Hereafter, we will briefly go into some implementation details of PPO, discuss the methods used for hyperparameter tuning and finally give an overview of the used hyperparameters.

5.1. Experimental setup

The aim of this section is to gives a general overview of the experimental setup for this research. It presents the methods used to answer the 3 posed research questions, which are repeated here for readability:

> **Research question 1**: *Can the sample efficiency of DRL be improved by using GNNs and domain knowledge, compared to traditional architectures such as CNNs and MLPs?*
>
> **Research question 2**: *Can the generalization performance of DRL be improved by using GNNs and domain knowledge, compared to traditional architectures such as CNNs and MLPs?*
>
> **Research question 3**: *Can the fixed relational inductive biases of a CNN be combined with learned domain-specific relational inductive biases and GNNs to improve sample efficiency and generalization performance in an end-to-end fashion, which does not require domain knowledge?*

All proposed methods in this research focus on applying domain-specific relational inductive biases to the policy and value network of a RL-agent, by encoding observations as graphs and applying GNNs on these observations. For research question 1 and 2, it is assumed that these added relational inductive biases will be provided in the form of domain knowledge. For research question 3 we seek to learn these domain-specific relational inductive biases, which would provide an end-to-end solution that does not require any prior knowledge. All 3 research questions will be explored in a set of simple gridworld environments whilst using PPO in an actor-critic setting as learning algorithm.

Figure 5.1 provides an overview of the general structure of the used architectures for learning. All of these architectures can be split up into two parts: the 'feature extractor' followed by 2 'linear layers'. Here, the feature extractor refers to a shared network - meaning that the trainable parameters are shared between the actor and critic - which extracts hidden features from the observations. Usually, for grid-shaped observations, this would be a CNN. The output of the feature extractor then serves as input for the linear layers which are unshared, meaning that the actor and critic have separate linear layers which therefore also have separate trainable parameters. Multiple GNN-based architectures are explored as feature extractor - which allow us to induce domain-specific relational inductive biases -

and these are compared to baseline architectures; CNN and MLP. To ensure a fair comparison, all compared architectures have approximately the same total amount of trainable parameters.

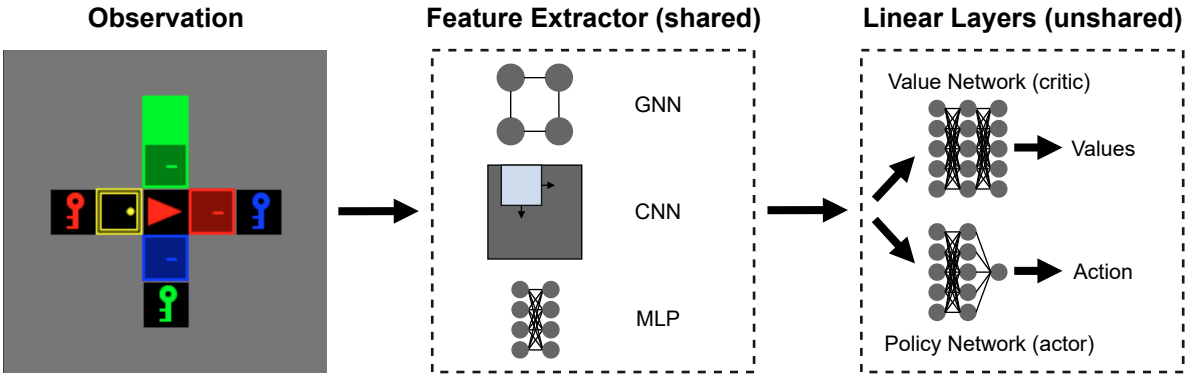


Figure 5.1: Learning setup. All layers are meant as illustrative example and are not displayed with the actual number of neurons used in our methods.

5.2. Environments

The environments that are used to evaluate our approach are navigation tasks which are specifically designed to test for relational reasoning and generalization. They are based upon the MiniGrid implementation by Chevalier-Boisvert et al. (2018), which works with the OpenAI Gym environments introduced by Brockman et al. (2016). The general concept behind the environment is inspired by the Box-World environments developed by Zambaldi et al. (2018). We have chosen to start out with the MiniGrid implementation and adjust this to our needs as it is designed for simple, lightweight and fast grid world environments and includes the possibility to render the grid world which is helpful for visualization. In this section, our implemented environments and some elements of MiniGrid - the ones relevant to this research - are discussed. To discuss these environments, we will make use of the possibility to render the observations as images. However, it is important to note that these rendered visualizations are not equal to the actual observations that the agent receives, which will be discussed in section 5.2.2.

All our environments - from hereon referred to as the Key-Corridors environments - are grid worlds, which consist of an $[X, Y]$ shaped grid of tiles. Each of these tiles can contain zero or one object and all objects have a discrete color and type. For example, the 'red key', where 'red' is the color and 'key' is the type. One of these types is the 'agent' and the presence of this type of object at a particular tile therefore indicates the position of the agent. In the rendered visualizations of the environment, the agent is displayed as a red arrowhead and the direction of the arrowhead displays the orientation of the agent. Furthermore, the agent can move around the gridworld - over empty grid tiles or tiles that contain objects that allow the agent to overlap with them - and its goal is to reach the green goal tile. Next to the agent and the goal, the tiles can also contain walls, doors and keys. The agent cannot overlap with walls and closed doors, but it can pickup and use keys to unlock the doors which then allows the agent to first open and then overlap with (i.e. walk through) the opened door. A key can only be used to open a door that has the same color as the key itself. Figure 5.2 gives an overview of an environment and its present objects.

All of the Key-Corridors environments contain 4 doors, of which one is unlocked at the start of an episode and the three others are locked. See figure 5.3 for two example renderings. Furthermore, the environments all have 5 rooms; one central room and a room behind each of the 4 doors. At the start of an episode the agent is placed at the center tile of the central room and has to reach the room with the tile that contains the goal to receive a reward. To do so, it will have to start off by opening the already unlocked door and pick up the key that is behind this door. With the new key the agent can open the next door, which either opens to a room with another key or to the room with the goal tile.

Two sizes of the Key-Corridors environment are introduced: one small version with a grid of (7x7) tiles and a larger version with a grid of (9x9) tiles. Both these environments - named Key-Corridors-Small and Key-Corridors-Big - are visualized in figure 5.3. The small version requires little exploration and therefore enables relatively fast learning and also simplifies the exploration-exploitation trade-off

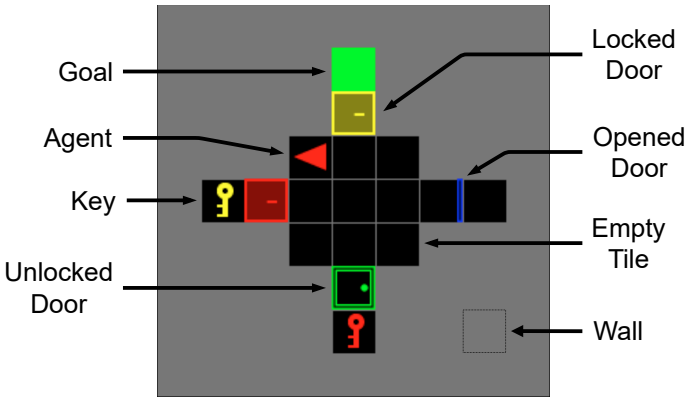
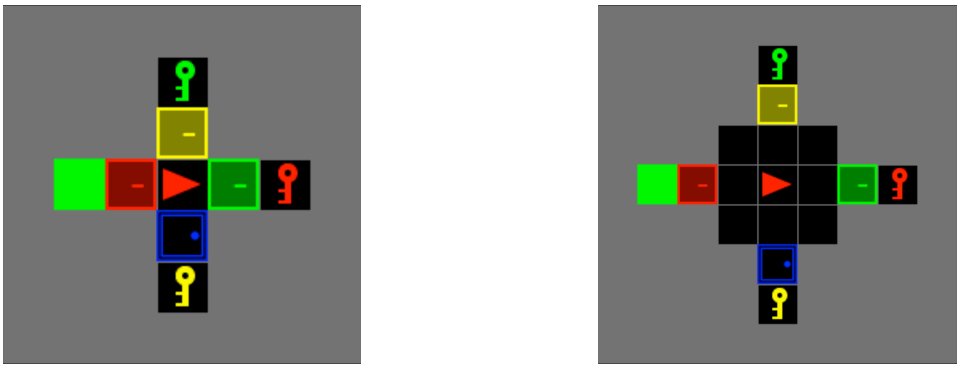


Figure 5.2: Overview of the different elements in a Key-Corridors environment. This image shows the rendered version of the environment, which is not equal to what the agent gets to see in an observation but it is used for visualization. The item that is currently being carried by the agent is encoded at the position of the agent in the actual observations, however this is not displayed during rendering as it would result in an object overlapping with the red arrowhead representing the agent.



(a) The Key-Corridors-Small environment.

(b) The Key-Corridor-Big environment.

Figure 5.3: Overview of the two sizes of the Key-Corridors environments.

in this specific environment. For both sizes of the environment, the amount of keys that are needed to solve the environment is varied as well. The simplest versions only have one key, for which the agent only needs to open the already unlocked door. The most difficult environments have a key in three of the 4 rooms - excluding the room that contains the goal - and therefore require the agent to unlock three doors before being able to reach the goal.

To prevent the agent from infinitely repeating bad actions and getting stuck in an environment, each environment has a step limit which indicates the maximum amount of steps an agent is allowed to take in one episode. When the step limit is reached, the episode is truncated and the environment is reset for a new episode. These step limits are based on the size of the environment and are set to $4 \times X \times Y$.

5.2.1. Random initialization of the environments

If the agent would only be trained on one particular arrangement of keys and doors, it could simply remember the sequence of actions that it has to take to reach the goal. To prevent this from happening, the location and colors of the keys and doors are changed between episodes, as well as the location of the goal tile. The agent, however, is always placed at the same location in the grid and with the same initial orientation. We refer to each of these different initializations of an environment as an 'instance'. The method for random initialization of the keys, doors and goal is described in algorithm 3. Both the keys and doors can have four different colors: Red, Blue, Yellow and Green. Furthermore, the keys can only be placed in the rooms behind the four doors. We will use the following set of positions to indicate the location of the doors and keys: Top, Bottom, Left and Right. Where, for example, the position 'Top' refers both to the position of the door at the top of the environment and the position of the key behind it. 'RandomShuffle' is a function that randomly shuffles all elements in the input list. The 'PlaceDoor' function is used to place a door, which takes the color and position of the door as input and whether

the door is locked or not. In addition, 'PlaceKey' is a function used for placing keys, and takes the color and position of the key to be placed as input. The value 'Keys-Placed' keeps track of the amount of keys that are already placed and the value 'Number-Of-Keys' indicates the amount of keys that should be present in this environment.

Algorithm 3 Random initialization of environments

```
Colors = RandomShuffle([Red, Blue, Yellow, Green])          ▷ Set of possible colors
Positions = RandomShuffle([Top, Bottom, Left, Right])       ▷ Set of possible positions
Keys-Placed = 0
for i in range(Colors) do
    if i == 0 & Keys-Placed ≤ Number-Of-Keys then
        PlaceDoor(Colors[i], Positions[i], Locked = False)          ▷ First door is unlocked
        PlaceKey(Color[i+1], Positions[i])          ▷ Placed key has the color of the next door
    else if Keys-Placed < Number-Of-Keys then
        PlaceDoor(Colors[i], Positions[i], Locked = True)           ▷ All other doors are locked
        PlaceKey(Color[i+1], Positions[i])          ▷ Placed key has the color of the next door
    else if Keys-Placed == Number-Of-Keys then
        PlaceDoor(Colors[i], Positions[i], Locked = True)           ▷ All other doors are locked
        PlaceGoal(Positions[i])   ▷ Places goal behind a door that can be opened with the last key
    else
        PlaceDoor(Colors[i], Positions[i], Locked = True)           ▷ Remaining doors are locked
    end if
    Keys-Placed += 1
end for
```

This method of initialization ensures that the environment is always solvable. The first key, for example, is always placed behind the unlocked door and enables the agent to open the correct next door to be able to solve the environment. The total amount of possible different instances for an environment is the product of all possible permutations of Colors and Positions. Both Colors and Positions have 24 possible permutations, which leads to a total of 576 different possible instances of the environments. Even though slightly unintuitive, there are an equal amount of possible instances with 1, 2 or 3 key(s).

We also introduce environments which have a varying amount of keys between instances, i.e., one episode could be initialized with 1 key whilst another episode in the same environment could be initialized with 2 keys. This is done with the aim of speeding up training to environments with more keys. We reason that solving instances with 1 key is easier than solving those with 2 keys and requires less exploration, but does help the agent to solve the instances with 2 keys as well. Next to that, training on a mix of instances with either 1 or 2 key(s) might also improve generalization to instances with 3 keys, which can be seen as a form of domain randomization. The average number of steps that is required to solve an environment with a certain amount of keys is displayed in table 5.2. This is an average because of the random initialization of keys and doors, which results in different amount of minimally required steps to solve the environment. For the environments that are randomly initialized with both 1 and 2 key(s), there is a total of 2 $\times$ 576 = 1152 different instances, as there is of course a difference between the instances with one key and those with 2 keys. The same holds for the environments with instances containing either 1, 2 or 3 key(s), which have in total 1728 different initializations.

Table 5.1: Average episode length, under an optimal policy, of the environments with a varying amount of keys.

Environment	1 key	2 keys	3 keys	1 and 2 key(s)	1, 2 and 3 key(s)
Key-Corridors-Small (7x7)	$9\frac{5}{12}$	$15\frac{1}{12}$	$20\frac{3}{4}$	$12\frac{1}{4}$	$15\frac{1}{12}$
Key-Corridors-Big (9x9)	11	18	25	$14\frac{1}{2}$	18

5.2.2. Observations

The Key-Corridors environments are fully observable and the agent gets to see the entire grid in each observation. These observations are not in the form of an image, such as the renderings of the environment displayed in figure 5.3, but come in a more abstract matrix encoding. In these observations, each of the $X \times Y$ grid tiles is encoded as a 6-tuple: (**Object**, **Color**, **State**, **Orientation**, **Carried-object**, **Carried-object-color**), of which all elements are one-hot encoded and have the following meaning:

Object	=	The object that is present at the tile.
Color	=	The color of the object that is present at the tile.
State	=	The current state of an object. In this case this is only useful for doors, for which it indicates whether a door is locked, unlocked or open.
Orientation	=	An encoding of the orientation of the agent, i.e., the direction in which the agent is facing.
Carried-object	=	The object that is currently being carried by the agent. This will only be encoded at the current position of the agent and for all other tiles it will indicate that there is no object being carried.
Carried-object-color	=	The color of the object that is currently being carried by the agent. This will again only be encoded at the current position of the agent and at all other positions it will not have any color.

Table 5.2 gives an overview of all the available objects, colors, states and orientations. With all of these objects, colors, states and orientations there are in total 31 features per grid tile. Notice that besides the objects present in figure 5.2, there is also an additional 'Empty Bag' object, which is used to indicate that an agent is currently not carrying anything.

Table 5.2: Overview of all possible objects, colors, states and orientations for the encoding of grid tiles.

Objects	Colors	States	Orientations
Empty Tile	Red	Locked	Up
Door	Blue	Unlocked	Down
Key	Yellow	Open	Right
Wall	Green		Left
Agent	Grey		
Goal			
Empty Bag (indicating that no object is being carried)			

Observation normalization is important in RL. Andrychowicz et al. (2020) suggest normalizing observations by keeping an empirical mean and standard deviation of each feature - based on the observations seen so far - and normalizing by subtracting the mean and dividing by the standard deviation, or 10^6 when the standard deviation becomes too small. Unfortunately this is not possible for our research, because we also test for generalization to different environments which are not seen during training and therefore do not allow us to keep an empirical mean and standard deviation of each observation feature. However, the one-hot encoding of our features acts as some form of observation normalization which at least ensures that all input values are in the same range, in this case just zero or one. The actual shape of the observation tensor depends on the used feature extractor. For the MLP architecture, the grid is flattened into a one dimensional tensor with a resulting tensor shape of $[X \times Y \times 41]$. The CNN can be directly applied to the grid-shaped data and gets the observations in the form of a three dimensional tensor with the following shape: $[X, Y, 41]$. For the GNN-based architectures the observations will be encoded as graphs. The exact form of these graphs is dependent on the type of GNN that is used and the relational biases we want to encode. These graph encodings will discussed in detail in section 5.4. For ease of implementation, these graphs are only constructed during the forward pass of our GNNs and the actual observations are dictionaries which contain the same three dimensional tensor as used for CNN, as well as two additional one dimensional tensors which encode the positions of the keys and doors in the environment.

5.2.3. Actions

Just like the observation space, the action space of our environments is discrete. The agent is allowed to choose actions from the following set: [**Up**, **Down**, **Right**, **Left**, **Open**]. The working of these actions is listed below:

- **Up** = Turn the orientation of the agent to the upwards direction and try to make a step in this direction.
- **Down** = Turn the orientation of the agent to the downwards direction and try to make a step in this direction.
- **Right** = Turn the orientation of the agent to the right and try to make a step in this direction.
- **Left** = Turn the orientation of the agent to the left and and try to make a step in this direction.
- **Open** = Try to open the object that is in front of the agent. When the agent is carrying the correct key and is standing in front of the door, whilst facing it, the door is unlocked and opened upon using this action. Because of this action the orientation of the agent is of importance. Imagine a situation where the agent is in a tile for which two of the adjacent tiles contain a door. In this case the orientation of the agent is needed to know which door it is trying to open.

If the agent tries to move onto a tile which contains an object that does not allow the agent to overlap with it, the agent will stay at its place. However, the agent does change its orientation based on the action that was taken. For example, if the agent is currently directed towards the right and tries to move to the left but there is a wall at this position, the agent will not move tiles but will change its orientation towards the left. As mentioned before, this way of changing the agent's orientation is important for the Open action. Furthermore, notice that there is no separate action to pick up objects. This is because the agent can simply move over an object that can be picked up and by doing so it picks up the object automatically. The agent is forced to drop the object it is currently carrying when it picks up a new object by moving over it. This dropped object is placed at the location of the object that was picked up by the agent. The decision to not add a pick-up and drop action and instead keep the action space very small was based on the limited amount of time and computational power available for this project.

5.2.4. Rewards

The Key-Corridors environments have very sparse rewards: the agent only receives a reward of 1 when it reaches the goal state. For all other steps it receives a reward of 0. If the step limit of the specific environment ($\texttt{steps}_{\texttt{max}}$) is reached, the agent also receives a reward of 0 and the environment is reset. Therefore the total reward that is obtainable by the agent in one episode can be summarized as follows:

$$r = \begin{cases} 1 & \textbf{if } \texttt{steps}_{\texttt{agent}} < \texttt{steps}_{\texttt{max}} \\ 0, & \textbf{if } \texttt{steps}_{\texttt{agent}} \geq \texttt{steps}_{\texttt{max}} \end{cases}, \tag{5.1}$$

where $\texttt{steps}_{\texttt{agent}}$ indicated the amount of steps that the agent took in its current episode. With this method the received rewards are always 0 or 1 and therefore no further reward scaling or normalization is required. One might wonder whether these rewards actually encourage the agent to try to reach the goal state with as few steps as possible, as there is no extra reward for taking less steps. However, due to the discount factor γ the agent does still try to reach the goal state with as few steps as possible, as longer routes encounter states with more discounting and therefore lower expected return.

5.3. Evaluation of agents

5.3.1. Performance metric

To measure the performance of our agents with respect to the posed research questions, a performance metric is required. In the Key-Corridors environments, a good performing agent minimizes the amount of steps that it needs to take to reach the goal tile. Therefore, the average number of steps required to solve an episode is taken as the performance metric.

5.3.2. Testing for sample efficiency

To test for sample efficiency, we examine how many environment steps the agent requires to obtain an optimal, or near-optimal policy. If one agent needs less experience from the environment to obtain a good performing policy than another, it is said to have a better sample efficiency.

5.3.3. Testing for generalization

To test for generalization, agents are evaluated on separate evaluation environments. This allows for evaluation on different environments than the ones being used for training, which enables us to test for generalization to unseen environments. Evaluation of the generalization performance of the policy is done periodically throughout the whole training process. This ensures a fair comparison between architectures, as generalization performance might decline after more training due to over-fitting on the training data, which is a common problem in RL and DL in general (Cobbe et al., 2019; Lawrence & Giles, 2000; Lawrence et al., 1997). As some architectures might reach an optimal policy on the training environments faster than others, it would not be fair to only compare them at the end of the training process. In our setup, evaluation is done after approximately every 50 thousand environment steps during training and the policy is evaluated over 30 episodes in the evaluation environment. The stochastic policy of the agent is also evaluated stochastically, meaning that - during evalutation - actions are sampled from the learned probability distribution over the actions, just as during training. Stochastic policies are often changed to deterministic policies - which take the action with the maximum probability instead of sampling from the probability distribution - during evaluation as this usually leads to better performance. We do, however, evaluate our policies stochastically because this yields better - mainly more stable - results for generalization. This can be explained by the fact that the agent can easily get stuck by taking one bad action with deterministic policies, especially in environmets which have deterministic transitions, which the Key-Corridors environments have. The following can, for example, happen: the agent picks a bad action, walks against a wall, receives the same observation again as it did not move and from there on stays in a loop of taking the same action and receiving the same observation. This would result in a reward of 0, whilst the agent might just have taken one wrong action. By stochastically evaluating the policy, the agent can still take some different actions for the same observation and therefore get out of these loops. We argue that it is fair to compare policies which are stochastically evaluated, because by doing so, we are actually evaluating the real policy that is being optimized by the agent instead of changing from a stochastic to a deterministic policy during evaluation. Furthermore, agents are encouraged to make the stochastic policy as deterministic as possible, as taking random actions usually hurts the rewards, and therefore the agent should learn an almost deterministic policy to obtain good results.

In-distribution generalization

In this thesis, we define in-distribution generalization as generalizing to the same type of environment as the training environment, however with some small, not previously observed, changes in the environment. The evaluation environment, to which we aim to generalize, does however represent the same underlying problem with the same degree of difficulty for the agent.

More specifically, to test for in-distribution generalization, agents are trained on a 'train' set of instances, which is a set of Key-Corridors instances from which a certain percentage of the total amount of possible instances is withheld. The agents are evaluated on a separate 'test' set of instances which only contains the withheld instances. Therefore, the agent encounters unseen arrangements of the keys, doors and goal state during evaluation. However, the amount of keys that the agent needs to use to reach the goal is the same for the train and test set and therefore the task of the agent is equally difficult in both sets. This proper split of train and test instances for testing generalization is suggested by Nichol et al. (2018), who argue that it is actually very special that it is so common in RL to test agents on the exact same environment they are trained on, which is comparable to evaluating on the training set in machine learning. In our experiments, agents are trained on 70% of the possible instances of an environment and are evaluated on the remaining 30% of instances.

Out-of-distribution generalization

Out-of-distribution generalization is defined as generalizing to slightly different problems than the ones seen during training, which is generally a harder task than in-distribution generalization. Testing for out-of-distribution generalization is done in two ways: either by evaluating on instances with a different amount of keys than the instances of the training environment or by evaluation on an instances of an environment with a different size than the environment that the agent is trained on. More specifically, the following 'train' and 'test' splits are tested:

1. **Varying the amount of keys for both the small and big key-corridors environment:**
   In this case, the agent is trained on an environment that has instances with 1 and 2 key(s) and is

evaluated on an environment with 3 keys. The choice to train on environments with both 1 and 2 key(s) is again based on the fact that this leads to faster convergence and in addition also to better generalization. This is probably due to the fact that, in this case, the agent is already used to solving instances with different amounts of keys, which makes the transfer to yet another key a bit easier.

2. **Varying the size of the environment:**
   Here we either train on the Key-Corridors-Small environment and evaluate the agent on Key-Corridors-Big, or we train on Key-Corridors-Big and test for generalization to Key-Corridors-Small.

5.4. Architectures

In this section, the architectures of the different feature extractors will be presented and discussed. These include MLP and CNN as baseline architectures as well as several GNN-based approaches, which form one of the main components of this research. Detailed background on the theory behind these specific architectures can be found in section 3.1, this section focuses on our specific implementation and use of these architectures. All GNNs are implemented with Deep Graph Library (DGL) from M. Wang et al. (2019) and PyTorch. All other architectures are implemented with PyTorch (Paszke et al., 2019).

Only the feature extractor layers differ between the compared methods, while the following linear layers were kept constant between the methods. However, the output of the feature extractors can have different sizes. Therefore the input size of the first linear layer depends on the output size of the final layer of the feature extractor. Details on the number of trainable parameters and how these are distributed over the different layers will be discussed in section 5.7.1 Each of the compared feature extractor architectures always consists of two layers. The design choice to keep the depth of the compared architectures equal is made to allow for a fair comparison between the different methods. Likewise, many of the design choices for the different architectures, especially for the CNN- and GNN-based ones, are made with the goal to make them as comparable as possible. In the following sections, the individual feature extractor architectures will be discussed.

5.4.1. Multilayer Perceptron (MLP)

This is the simplest architecture, which has practically no relational inductive biases since there is an all to all relationship between all grid tiles in the input. This feature extractor consists of two linear layers, where the first linear layer has an input dimension equal to the size of the flattened one-hot observations: $X \times Y \times 41$. The output of the feature extractor is simply the output of the last linear layer, as this can directly be used as input for the following linear layers in the overall architecture, as previously explained in figure 5.1. This architecture will be referred to as '**MLP**' throughout the rest of this report.

5.4.2. Convolutional Neural Network (CNN)

After the MLP based feature extractor, a step is taken towards an architecture with stronger relational inductive biases that are known to work well for grid shaped data; a CNN. The feature extractor again consists out of two layers, which this time are convolutional layers. Both of these layers use kernels with a kernel size of 3×3 and a stride of 1 pixel. Here 'pixel' refers to one element of the input grid, which in our case would be one tile of the observation. The choice for the kernel size of 3×3 pixels is based on the fact that these are the smallest possible kernels that still capture information of a center tile and the tiles that are located above and below this center tile, as well as the ones to its left and right.

Often convolutional layers in a CNN are followed by a pooling layer to reduce the dimensionality of the output. However, in our CNN architecture no pooling layers are added between the two convolutional layers. This is done for two reasons: first, the input is already a relatively small grid and thus pooling is unnecessary or even harmful and second, the intention is to later compare this method with the GNN-based methods which also have no downsampling between layers. A 1-pixel padding was added to also update the features of the outer grid tiles, again leading to better comparability with the GNN-based methods. A final difference between the CNN- and GNN-based methods is the use of a readout function in GNNs to go from a graph-shaped output to a 1D tensor, whereas a CNN would usually just flatten the last feature map into a 1D tensor. To make this comparable between the two

methods, but at the same time also test whether the readout function itself is not the driver between different results, we propose the following two architectures, both using kernels with a kernel size of 3×3, a stride of 1 pixel, 1 pixel padding and no pooling in-between layers:

CNN_{flat} : In this architecture, the feature map outputted by the final layer - which, due to the lack of pooling layers, still has a shape of $[X, Y, F]$, where F is the number of output features per pixel - is flattened into a 1D tensor. This is the most traditional way of going from the output of a CNN to linear layers.

$\text{CNN}_{\text{maxpool}}$: For the second architecture, a similar readout function as used in the GNN-based architectures, is implemented for the output of the CNN. With this architecture, the results of a GNN-based architecture with readout - which is the standard way of using a GNN in graph classification tasks - can be compared to the CNN-based approach with the same readout function. As it turned out that max-pooling was the best performing readout function for our problems, we compare the architectures with max-pooling readout functions.

5.4.3. Graph Convolutional Network (GCN)

The first graph based architecture is based on a GCN (Kipf & Welling, 2017). In order to use a GCN, the observations need to be encoded as a graph. To do so, the grid-shaped observation will have to be transformed into nodes and edges. Fortunately, the observations of our environment can be encoded as nodes and edges quite intuitively. Each grid tile clearly forms a separate entity and will therefore be encoded as a node in the graph, which gets the features of the corresponding grid tile as node features. From now on, 'node' and 'grid tile' are used interchangeably, i.e., the grid tile with the red key, can also be referred to as the node with the red key. All directly adjacent grid tiles - excluding diagonally adjacent tiles - will be connected with a bidirectional edge. Furthermore, each node has an edge to itself, also called a self-loop, to ensure that nodes do not 'forget' their own features. Even though this environment is specifically chosen to be easily encoded into a graph, one could also first apply a CNN to, for instance, raw pixel data to extract entities and then encode these entities as the nodes in a graph. Figure 5.4 illustrates the graph encoding of an example observation.

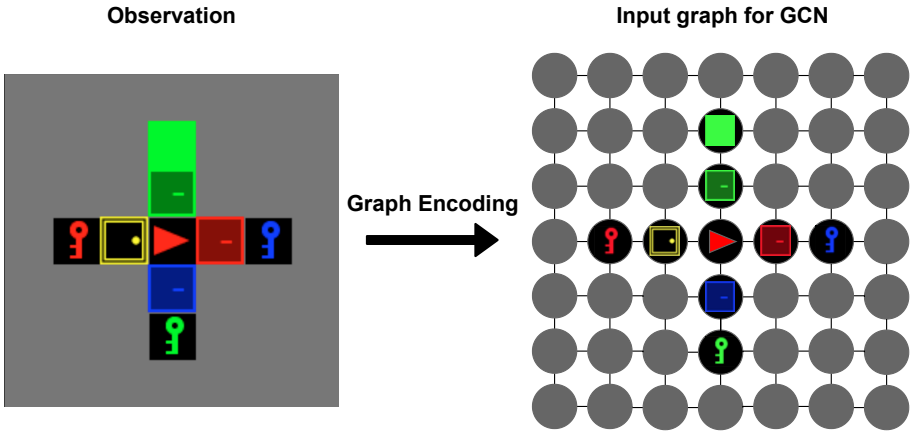


Figure 5.4: Simple graph encoding. Edges between the nodes are bidirectional.

Our implementation of GCN has two small differences from the one introduced in section 3.2.3: the normalization constant c_{ij} is removed and an extra bias term b is added. The new function describing the update of the features h_i of node v_i, in one layer l of our implementation of GCN is shown in equation 5.2.

$$h_i^{(l+1)} = \sigma^{(l)}\left(\sum_{j \in \mathcal{N}_i} h_j^{(l)} W^{(l)} + b^{(l)}\right) \tag{5.2}$$

The bias term is added for stronger expressive power, as suggested by Xu et al. (2019). The normalization constant is dropped as this empirically led to better sample efficiency in our setup. Additionally,

almost all nodes - except the ones on the outer edge of the grid - have the same degree and the maximum difference in node degree is only 2, removing the necessity for the normalization constant. This architecture will be referred to as **GCN** in the following sections.

One problem with the GCN architecture in combination with our grid-shaped environment, is the fact that during the update of the features of a node, the same weights are used for all incoming messages from its neighboring nodes. Because of this, the GCN would compute the exact same features for the central node in both hypothetical observations displayed in figure 5.5, whilst these kind of symmetries are actually often present in the grid-shaped observations of the Key-Corridors environment. Consequently, a method with stronger expressive power is needed, which is discussed in the following section.

Graph construction and graph batching are costly operations in DGL and it is therefore best to avoid using - especially repeating - these operations as much as possible. As each graph actually has the same topology for this architecture, the graphs are constructed once and are reused, whilst only the node features are changed for each observation.

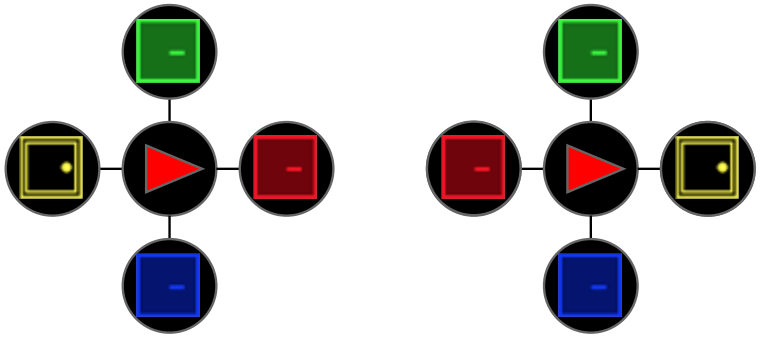


Figure 5.5: Illustrating the shortcoming of a GCN for the grid-shaped Key-Corridors environments. The two displayed graphs are indistinguishable for a GCN.

5.4.4. Relational Graph Convolutional Network imitating a CNN (R-GCN$_{\text{CNN}}$)

Since it was concluded in the previous section that a network with stronger expressive power is required, we propose to use a R-GCN instead. R-GCNs allow for different type of edges between nodes, i.e., they are able to encode different type of relations between entities, where each relation has its own set of weights. This enables us to solve the problem displayed in figure 5.5 by having a different relation for each adjacent node, as shown in figure 5.6. As a result, it is now possible to encode the fact that the red door is located to the right of the agent into the features of the central node with the agent, whereas previously it was only possible to encode the presence of a red door between the adjacent nodes of the central node and not its relative position.

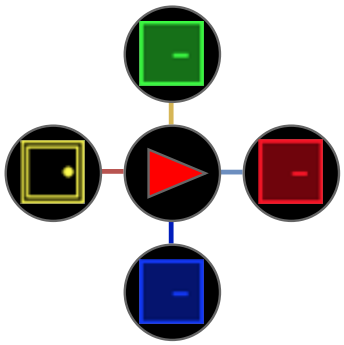


Figure 5.6: R-GCN allows for introducing a different relation for each adjacent node, illustrated with the differently colored edges, which provides stronger expressive power than a GCN.

Equation 5.3 is used to update the features of node v_i in the l'th layer of a R-GCN. Here we again - as for GCN - omitted the normalization constant c_{ij} and added a bias term b, compared to equation 3.4 of the background on R-GCNs.

$$h_i^{(l+1)} = \sigma^{(l)}\left(\sum_{r\in\mathcal{R}_{cnn}}\sum_{j\in\mathcal{N}_i^r} W_r^{(l)} h_j^{(l)} + W_0^{(l)} h_i^{(l)} + b^{(l)}\right) \tag{5.3}$$

As it is known that CNNs perform well on grid-shaped data, we start with imitating the CNN_{flat} architecture with the kernel size of 3×3, as also done by Z. Jiang et al. (2021). This architecture will be referred to as **R-GCN$_{\text{CNN}}$** in following sections. For each node, a unique relation is added to each of its adjacent nodes in the grid, which this time includes its diagonally adjacent nodes as well. There is no need to add a self-loop, as these are always included in the node update of a R-GCN, as can be seen by the $W_0^l h_i^l$ term in equation 5.3. This means that, in total, there are 9 separate weight matrices W for 9 different relations r, of which 8 are included in $\mathcal{R}_{cnn}$ and the 9th is W_0. This is exactly like a 3×3 kernel of a CNN, which would learn 9 different weight matrices as well.

The concept of imitating CNN with a R-GCN is visualized in figure 5.7. On the left side of the figure, a section of the graph is displayed, which shows the similarity with a kernel. Here the middle node - with the light green color - is the node that is updated. The middle section of the image shows how this 'graph kernel' would move over the graph and produce the output that is displayed on the right hand side, where the light green nodes are the nodes that now have been updated. In reality, there is no kernel that moves over the graph, but the edges displayed in the graph kernel are present for each of the nodes that are colored light green in the output. Displaying all these edges would therefore make the figure very cluttered. Note that all nodes also have a self-loop, as is by default already encoded in the update function of a R-GCN. As a result of these self-loops, the nodes on the outer edge of the graph are also updated, but only with their own features. One remaining difference to the CNN architecture is the absence of padding, which is not required in the R-GCN as the size of the graph already remains constant throughout its layers.

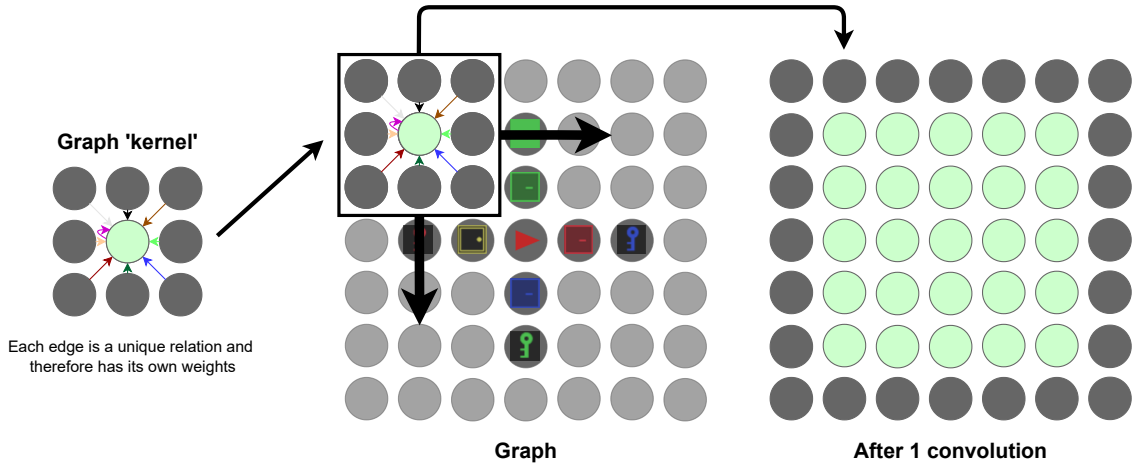


Figure 5.7: Imitating a convolutional layer with a R-GCN.

5.4.5. R-GCN with domain-specific relations (R-GCN$_{\text{domain}}$)

In this architecture domain-specific relational inductive biases will be encoded into the architecture in the form of relations in a R-GCN. The used update function is exactly the same as the one presented in equation 5.3. However, this architecture employs different relational inductive biases than those of a CNN. It is assumed that some form of domain knowledge is available, on which these domain-specific biases are based. In a real-world scenario this could be a domain expert or, for instance, a database which has information on relations between entities. In our case, it is assumed that there is domain knowledge about the fact that an agent can pick up keys and should use these keys to open doors. Furthermore, it is also known that the color of a key indicates which door it can open and that the agent can take 4 different actions to move: up, down, right and left. Based on this knowledge, 6 different relational edges are introduced. The first four are based on the possible actions the agent can take to move around:

left : For each node v_a that has another node v_b to its left, there is an edge from v_b to v_a, allowing node v_a to encode knowledge about the node that can be reached by taking action 'left' at node v_a.

right : For each node v_a that has another node v_b to its right, there is an edge from v_b to v_a, allowing node v_a to encode knowledge about the node that can be reached by taking action 'right' at node v_a.

up : For each node v_a that has another node v_b above its position in the grid, there is an edge from v_b to v_a, allowing node v_a to encode knowledge about the node that can be reached by taking action 'up' at node v_a.

down : For each node v_a that has another node v_b below its position in the grid, there is an edge from v_b to v_a, allowing node v_a to encode knowledge about the node that can be reached by taking action 'down' at node v_a.

The remaining two relations are based on the domain knowledge about keys and doors:

Can pick up : An edge is added from the node at which the agent is currently present to each node that contains a key. These edges can be seen as a 'can pick up' relation between agent and keys. If the agent is carrying a key, there is a self-loop edge of this type present for the node at which the agent is present.

Can open : This relation encodes the knowledge about keys being able to open doors, more specifically, the fact that keys can open doors with the same color as the key. To this end, edges - which represent a 'can open' relation between key and door - are added from each node with a key to the the node containing the door that can be opened with this key. If the agent is carrying a key, there is an edge of this type from the node with the agent to the node containing the door that can be opened by the agent with this key.

Note that all nodes again have an additional self-loop, which is by default encoded in the update function of a R-GCN and can be seen as a seventh relation. As mentioned before, these self-loops enable nodes to not 'forget' their own features. See figure 5.8 for an example graph encoding for this architecture. In this figure the additional self-loops (or seventh relation) are not displayed. In following sections, we will refer to this architecture with **R-GCN$_{\text{domain}}$**.

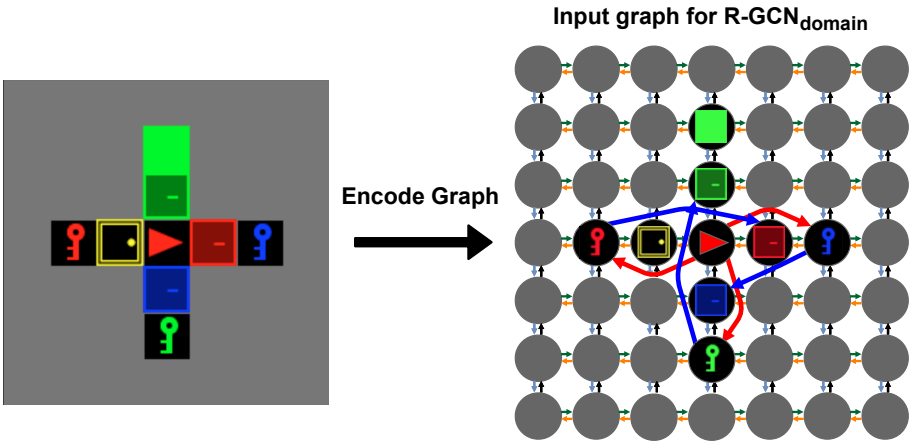


Figure 5.8: R-GCN with domain-specific relations. The four small orange, green, black and light blue edges which are present at all of the nodes, indicate the left, right, up and down relations, respectively. The red edges from the agent to the keys represents the 'can pick up' relations and the blue edges between keys and doors display the 'can open' relations. Note that each node also has an additional self-loop edge, which can be seen as an extra relation, however, these are not displayed in this figure for readability.

With this architecture it is not possible to always use the same graph structure and only update the node features based on the observation. This is due to the extra relations which cause the graph structures to change between observations. To solve this, each graph is stored in a hash-table during the forward pass. An incoming observation tensor is hashed and used as key in the hash-table. If

the observation has not been seen before, the graph is constructed and stored as value with its corresponding key; if the observation has been seen before, the key is used to retrieve the graph from the hash-table.

5.4.6. R-GCN$_{\text{CNN}}$ with additional domain-specific relations (R-GCN$_{\text{CNN+domain}}$)

In the R-GCN$_{\text{domain}}$ architecture, the 'can pick up' and 'can open' relations intuitively seem to be the more important relations based on domain knowledge, when compared to the left, right, up and down relations, which are actually a sub-set of the relations that are also present in the R-GCN$_{\text{CNN}}$ architecture, imitating a CNN with a kernel size of 3×3. In order to separately test the importance of the 'can pick up' and 'can open' relations, instead of in combination with the 'left', 'right', 'up' and 'down' relations, the **R-GCN$_{\text{CNN+domain}}$** architecture is introduced. With this architecture, the 'can pick up' and 'can open' relations are directly combined with the relations imposed by a CNN with a kernel size of 3×3. In this case the R-GCN encodes the 9 relations of the CNN, plus the additional 'can pick up' and 'can open' relations, as described in section 5.4.5. This architecture aims to check whether the diagonal edges that are present between adjacent nodes in the R-GCN$_{\text{CNN}}$ architecture, but are removed in the R-GCN$_{\text{domain}}$ architecture, are not the main cause of any potential performance differences between these two architectures.

5.4.7. R-GCN$_{\text{CNN}}$ with additional random relations (R-GCN$_{\text{CNN+random}}$)

The 'can pick up' and 'can open' relations of R-GCN$_{\text{domain}}$ are based on domain knowledge and therefore seem to be fitting for the problem that needs to be solved in the Key-Corridors environments. However, as also explored by Kurin et al. (2020), intuitively satisfying relations in the graph architecture of a GNN do not always have to be the (only) relations that would improve the performance of the agent the most. It could for instance be the case that other, more random, additional edges between nodes that are not directly adjacent to each other would also improve the performance, simply because they encode some longer distance relations than the ones present in our CNN architecture.

To test for all possible graph structures with a certain amount of nodes and relations is infeasible in this thesis. However, in an attempt to show that - in this case - the relations based on domain knowledge are indeed better than random relations, we introduce the **R-GCN$_{\text{CNN+random}}$** architecture. This architecture again duplicates the relations of a CNN with a kernel size of 3×3 - as is done in the R-GCN$_{\text{CNN}}$ architecture - but now adds two additional relations that intuitively do not seem to help in solving the problem, as opposed to the 'can pick up' and 'can open' relations. The performance of this architecture will be compared tot the performance of the R-GCN$_{\text{CNN+domain}}$ architecture introduced in the previous section. To make this a fair comparison, and therefore allow us to reason about the usefulness of domain-specific relations versus more random relations, the two architectures will have the same amount of additional edges per relation. In practice, we do this by changing the 'can pick up' and 'can open' relations as follows:

Can pick up : This relation will again add an additional edge for each key that is present in the observation, however, these edges now always point from the central room to one of the rooms behind the doors, independent of the actual location of the goal or keys. Furthermore, the edges do not move based on the movement of the keys or agent, whereas before, if a key or the agent moved, the edges were adjusted accordingly.

Can open : For the 'can open' relation, there are still edges between keys and doors, however, now there is always an edge from a key to a door that can not be opened with that key. For example, in an observation with a red key, there is an edge from the red key to the blue door.

5.4.8. Learning additional relations with attention (R-GCN$_{\text{GAN}}$)

In the previously introduced R-GCN$_{\text{domain}}$ architecture, domain-specific relations are used, which are based on domain knowledge. However, there are two strong assumptions that must be met in order to work with this architecture: it requires the availability of domain knowledge to define the relations and additionally presupposes that clearly defined entities, such as "key" and "door", are present in the observation or that these entities can be extracted from the raw input data. This section will introduce a

new architecture that facilitates the learning of additional relations rather than defining them in advance. This not only removes the assumption of domain knowledge being available, but also eliminates the need for clearly defined entities.

For this architecture, we start with the same architecture as R-GCN$_{\text{CNN}}$, as introduced in section 5.4.4. This time, instead of adding predefined relations, we intend to learn extra relations with a GAN. For this purpose, another graph is constructed, for which each grid tile is again encoded as a node, but now the graph is fully connected, i.e., there is an edge between each pair of nodes. A self-loop is added to each node as well. On this fully connected graph, a GAN with 3 different attention heads is applied, which gives 3 separate output tensors; one for each head. Because of the fully connected graph this can be seen as learning pairwise relations between nodes, where, in theory, each head can represent a different type of relation. Learning pairwise relations with a GAN to induce relational inductive biases in DRL, is an approach explored by Zambaldi et al. (2018) as well. However, Zambaldi et al. (2018) assume that the location of a node is encoded in its features. As we do not make this assumption, their approach would fail in our setup because without these positional features, a GAN on a fully connected graph could not distinguish two graphs with the same set of nodes, even though the individual nodes would have different positions within the observation grid. In other words, all positional knowledge is lost due to the fully connected graph and the absence of positional information in the input features of our nodes. To overcome this problem, we propose the **R-GCN$_{\text{GAN}}$** architecture which combines R-GCN and GAN. The overall structure of the architecture is as follows: the first layer is exactly the same as the first layer of the R-GCN$_{\text{CNN}}$ architecture, which encodes some positional information into the node features as the CNN-like relations only encodes local information into each node's features. After this first layer, both the second layer of R-GCN$_{\text{CNN}}$ and the GAN layer are applied in parallel. The 3 outputs of the different heads of the GAN are summed together, after which, they are added to the output of the R-GCN before the activation function of R-GCN is applied. This can be seen as a CNN - implemented with R-GCN - which learns additional relations, next to the local ones of CNN. The following equations give an overview of the update functions that are applied to each node i in the observation graph:

Layer 1:

$$h_i^{(1)} = \sigma^{(0)}\left(\sum_{r\in\mathcal{R}_{cnn}}\sum_{j\in\mathcal{N}_i^r} W_r^{(0)}h_j^{(0)} + W_0^{(0)}h_i^{(0)} + b^{(0)}\right) \tag{5.4}$$

This first layer is exactly the same as the first layer of R-GCN$_{\text{CNN}}$.

Layer 2:

$$\text{For each head } k \in K: \begin{cases} e_{ij}^{(k)} = \text{LeakyReLU}\left(\mathcal{A}^{(k)}\left(W^{(k)}h_i^{(1)} \| W^{(k)}h_j^{(1)}\right)\right) \\ \alpha_{ij}^{(k)} = \frac{\exp\left(e_{ij}^{(k)}\right)}{\sum_{u\in\mathcal{N}(i)}\exp\left(e_{iu}^{(k)}\right)} \end{cases} \tag{5.5}$$

$$h_i^{(2)} = \sigma^{(1)}\left(\sum_{r\in\mathcal{R}_{cnn}}\sum_{j\in\mathcal{N}_i^r} W_r^{(1)}h_j^{(1)} + W_0^{(1)}h_i^{(1)} + b^{(1)} + \sum_{k=1}^{K}\sum_{j\in\mathcal{N}(i)} \alpha_{ij}^{(k)}z_j^{(k)} + b^{(}k)\right) \tag{5.6}$$

Here, equation 5.5 shows the formulas for each attention head k, which were introduced in section 3.2.5 of the background. Equation 5.6 shows how R-GCN$_{\text{CNN}}$ and GAN are combined in the second layer. The LeakyReLU of GAN has a negative slope angle α_{LRLU} of 0.2, as suggested by Velicković et al. (2017). Furthermore, the GAN itself does not apply any activation function σ as this is only applied once the output of the GAN heads is added to the output of the R-GCN. Note that a bias term b is added to the GAN network as well.

5.4.9. Readout functions

Usually, in a graph classification setting - which is most comparable to our setup - a readout function is used after some GNN layers to compute one single set of features for the entire graph. This allows GNNs to work on variable size input, which is one of their key features. We have implemented and tested several readout functions for all of our architectures, including CNN, as this is later used for comparison to the GNN-based architectures. These readout functions are introduced below.

Flattening

For this readout, all node features of each of the nodes in the graph are concatenated into one large 1D tensor. This method of reading out a graph is usually not used for GNNs as it does not allow for variable size input. This readout function allows us to compare the GNN-based architectures to CNN_{flat}.

Sum-pooling

The sum-pooling readout function applies a featurewise summation over the nodes, as per equation 5.7. A potential disadvantage of this readout function is the loss of positional information.

$$r = \sum_{i=1}^{|N|} h_i \tag{5.7}$$

Average-pooling

This readout function is very similar to sum-pooling and only ads a division of the summed features by the total amount of nodes $|N|$. See equation 5.8.

$$r = \frac{1}{|N|} \sum_{i=1}^{|N|} h_i \tag{5.8}$$

Max-pooling

Max-pooling is a frequently used pooling method, both for CNNs and GNNs. Furthermore, it is more often used, with good results, in graph based RL methods such as Zambaldi et al. (2018) and Z. Jiang et al. (2021). For max-pooling, a featurewise maximum over the node features is used as readout, as per equation 5.9.

$$r = \max_{i=1}^{|N|} (h_i) \tag{5.9}$$

Global attention pooling

The final readout function used in this work is Global Attention Pooling (GAP). GAP uses an attention mechanism to determine how much weight should be put on each of the nodes' features. This method of pooling is based on the work by Li et al. (2016) and is displayed in equation 5.10. It works by first applying a linear layer f_{gate} to each node n_i. This layer has as input size the number of features per node and has an output size of 1. The output values of f_{gate} can be seen as attention weights a_i for each node n_i. All of these attention weights are then fed through a softmax function making them sum up to a value of 1, after which each attention weight a_i is multiplied by its corresponding node features h_i. Finally, the node features h_i - which are now scaled by their attention weights - are summed up. GAP

$$r = \sum_{i=1}^{|N|} \text{softmax}\left(f_{\text{gate}}\ (h_i)\right) h_i \tag{5.10}$$

GAP allows for intuitive visualization of the 'focus' of the agent, as we can plot the attention weights a_i for each node n_i, which directly gives us a measure of 'importance' of each node to the agent. This might be an interesting benefit for explainability of the behavior of the agent.

5.5. Implementation of proximal policy optimization

This section will first explain why PPO was chosen as RL algorithm for this research. This is followed by some implementation details of PPO, which differ from the previously introduced background on PPO.

5.5.1. Reasons for the use of PPO

The decision to use PPO as the RL algorithm in this thesis was based on several considerations. First of all, the decision for an on-policy and policy-based method was made, as these are generally known to be more stable than (off-policy) value-based methods (Nachum et al., 2017; S.Sutton & G.Barto, 2018). Furthermore, PPO is know to perform well on multiple benchmark tasks, when compared to alternative RL algorithms and allows us to sample trajectories from multiple parallel environments, which speeds up training time (Schulman et al., 2017). In contrast to trust region policy optimization (TRPO) (Schulman et al., 2015), PPO also enables us to easily apply parameter sharing between the actor and critic (Schulman et al., 2017). Lastly, PPO has good sample efficiency for an on-policy method as it does multiple updates on one batch of trajectories.

5.5.2. Implementation details

Instead of implementing PPO from scratch, an already implemented version from Stable-Baselines 3 (Raffin et al., 2019) is used in this thesis. Stable-Baselines 3 is a set of reliable implementations of reinforcement learning algorithms in PyTorch. There are, however, some minor differences to this implementation compared to the pseudocode for PPO introduced in the background section on PPO (3.4.4). These implementation details will be discussed in this section.

The first difference is the use of multiple vectorized environments. Instead of training the agent on one environment at a time, multiple independent environments are stacked to form a vectorized environment, which allows for training on all of these environments together and thereby makes the training process faster. When using N vectorized environments, the agent computes a vector of N actions at a time, which results in N observations and rewards. Because our environments do not require much computation, the vectorized environments are called in sequence on the same Python process, as this is actually faster than using multiple processes or threads for the different environments. A batch $\mathcal{B}$ of trajectories τ is now obtained by collecting one trajectory τ of length T in each of the N environments. Each of these environments is initialized with a different seed and leading to a variety of different trajectories.

A second implementation detail, different from the pseudocode in the background, is the use of Adam (Kingma & Ba, 2015) as optimizer instead of standard mini-batch-SGA. This is the optimizer used in the PPO implementation by OpenAI (Dhariwal et al., 2017), who originally introduced PPO. In addition, using the Adam optimizer with PPO is also recommended by Andrychowicz et al. (2020). The use of Adam introduces three new hyperparameters: $\mathcal{B}_1$, $\mathcal{B}_2$ and ϵ_{Adam}. Here, $\mathcal{B}_1$ and $\mathcal{B}_2$ are set to 0.9 and 0.999 respectively, which are the default values suggested by Kingma and Ba (2015) and are also suggested by Andrychowicz et al. (2020) for PPO specifically. Furthermore, ϵ_{Adam} is set to $1e-5$ as in the PPO implementation by OpenAI.

Another implementation detail is the use of gradient clipping. This is a commonly used trick in DL, to avoid exploding gradients (Zhang et al., 2020), which is also applied in the PPO implementation of OpenAI. Furthermore, Andrychowicz et al. (2020) report that the gradient of the performance function with respect to all parameters is clipped before it is used by Adam for optimization, to avoid that the global L^2 norm of the gradients exceeds a certain threshold. Here the gradient clipping threshold is an additional hyperparameter.

The final addition to the actual implementation, compared to the previously introduced pseudocode, is the use of advantage normalization. The advantages are normalized for each mini-batch by subtracting the mean advantage from each advantage value and dividing this by the standard deviation of the advantage plus 1e-8 to avoid division by zero. This is again also implemented in OpenAI's implementation of PPO and can be seen as an adaptive learning rate heuristic which limits the variance of the gradient (Tucker et al., 2018).

5.6. Method of hyperparameter tuning

This section will present and discuss how the hyperparameters of PPO were tuned. First, the general approach to tuning the hyperparameters is introduced, after which some of the used tuning metrics are discussed as well as the considered hyperparemter values.

Ideally, one would use some form of automated optimization over the entire search space of possible hyperparameters, as is sometimes applied to deep learning methods and DRL (Bergstra et al., 2011; Feurer & Hutter, 2019; Hertel et al., 2020). Doing this for all compared architectures would allow us to make the claim that each method is tuned to some degree of optimality and that it is therefore fair to compare them with these parameters. Unfortunately this is not possible in our case, due to the following two reasons: there is no single optimization objective for tuning our hyperparameters and the search space over all possible combinations of hyperparameters is too large for the available amount of computational resources. There is no single optimization objective for tuning because we try to balance training time, reaching an optimal policy and stability. Especially stability is hard to define as an optimization objective. The second problem, of having a large search space, makes it hard to - even with automated methods - run enough trials to make any claims about all methods being equally well tuned to allow for fair comparison between the methods. Our implementation of PPO has 12 hyperparameters, excluding network architectures, activation functions, weight initialization and the hyperparameters of Adam. Checking all possible combinations of 3 different values for each of these 12 hyperparameters would already require more than 500.000 runs, which is infeasible for us. Next to being infeasible, this would only be for 3 values for each of the hyperparameters, which is not nearly enough to provide any guarantees on how optimally tuned the algorithm is. Therefore, another approach will have to be taken.

Instead of tuning the hyperparameters for each of the different architectures separately, we tune for CNN and use the same parameters for all other architectures. As CNN is the main baseline against which the GNN-based approaches are compared, we reason that by tuning for CNN, we favor the baseline. Thus, if the GNN-based architectures outperform the baseline, it is most probably not caused by more favorable hyperparameters. This solves the problem of having to separately tune all parameters for each proposed architecture, however, it still does not solve the problem of having a very large search space. To solve this, tuning is done manually and the hyperparameters are initialized with values obtained from literature, if available and relevant to our problem. This enables us to start with a set of initial hyperparameters that should already give reasonable performance and then iteratively tune these parameters further. To interatively tune these parameters, we not only look at the average reward obtained or the average number of timesteps needed to solve an episode, but we also keep track of some extra tuning metrics that are helpful for generating insights into which parameters might have to be adjusted. These tuning metrics will be introduced in the following section.

5.6.1. Additional tuning metrics

The most important and useful tuning metrics for this thesis, were the entropy of the policy, explained variance and the clipping fraction, all of which are easily tracked in Stable Baselines 3. These three metrics will be shortly discussed below, as well as their use for tuning the hyperparameters.

Entropy

This is the same entropy as the entropy term S_{ent} in the performance function of PPO. Given a stochastic policy π, let $\pi(a|s_t)$ define the probability of taking action a from state s_t. The entropy H of this policy at state s_t can now be defined as follows:

$$H\left(\pi\left(\cdot \mid s_t\right)\right) = -\sum_{a \in A} \pi\left(a \mid s_t\right) \log \pi\left(a \mid s_t\right) \ , \tag{5.11}$$

where A is the action space representing all possible actions that the agent can take (Shannon, 1948). For a well-tuned algorithm, the entropy should gradually decrease towards 0 during training. If the entropy does not decrease, the agent continues to take many random actions, i.e., it keeps on exploring instead of exploiting a good policy. If the entropy drops too fast, the agent may not have explored enough of the environment, which could lead to convergence to a sub-optimal policy. The mean entropy of the policy is logged after each update - consisting out of multiple epochs of training - of the policy. This metric helps determining a good value for the entropy coefficient c_2 of PPO.

Explained variance

The explained variance can, in this case, be seen as a measure of how good the critic network - a learned value function - is in predicting the actual returns. The explained variance is calculated as follows:

$$EV = 1 - \frac{\mathrm{Var}[\text{Return} - \text{Predicted Value}]}{\mathrm{Var}[\text{Return}]} = 1 - \frac{\mathrm{Var}[G_t - V_{\pi_\theta}(s_t)]}{\mathrm{Var}[G_t]}, \tag{5.12}$$

where EV stands for the explained variance, G_t is the return at timestep t and $V_{\pi_\theta}(s_t)$ is the predicted value for state s_t at timestep t, under the current parameterized policy π_θ (Gaudet et al., 2020; Schulman, 2016a). An explained variance of 1 indicates that the critic network can flawlessly predict the actual returns. For an explained variance below 0, the predictions by the critic network are worse than just predicting a constant value. The mean EV is logged after each update - consisting out of multiple epochs of training - of the policy. This metric is useful for determining how well the critic network is functioning.

Clipping fraction

This is the fraction of steps at which the probability ratio $\Psi_t(\theta)$ is clipped in the performance function of PPO. At the beginning of training this value will be relatively high, as the policy will still have to be adjusted a lot and therefore the ratio between the old and new policy in an update will be relatively high. Later on, this value should decrease when the policy starts to converge towards an optimal policy. High values for the clipping fraction can indicate a too high learning rate or a too conservative value for the clipping parameter ϵ.

5.6.2. Initial hyperparameter values

In this section the initial hyperparameter values - mostly obtained from literature - are introduced. These values are used as a starting point for tuning the hyperparameter values of PPO with the CNN architecture and are displayed in table 5.3.

Table 5.3: Overview of the used initial values for the hyperparameter of PPO.

Hyperparameter	Starting value	Based on
Minibatch size (M)	64	Reported experience with PPO on MiniGrid environments by Raffin (2020)
Trajectory length (T)	128	Reported experience with PPO on MiniGrid environments by Raffin (2020)
Number of vectorized environments	8	Reported experience with PPO on MiniGrid environments by Raffin (2020)
Batch size $\lvert\mathcal{B}\rvert$	1024	$T \times$ number of vectorized environments = 8 $\times$ 128
Discount factor (γ)	0.99	Andrychowicz et al. (2020)
GAE parameter (λ)	0.9	Andrychowicz et al. (2020) suggest 0.9 and Schulman et al. (2016) recommend a value in the range $[0.9, 0.99]$
Learning rate (μ)	3e-4	Andrychowicz et al. (2020)
Number of epochs (N_{epochs})	10	Andrychowicz et al. (2020) and reported experience with PPO on MiniGrid environments by Raffin (2020)
VF coefficient (c_1)	0.5	Raffin (2020)
Entropy coefficient (c_2)	0.0	The need for this coefficient can be deduced from the policy entropy
Clipping parameter (ϵ)	0.2	Schulman et al. (2017) suggest 0.2 and Andrychowicz et al. (2020) recommend starting with 0.25
Gradient clipping threshold	0.5	Andrychowicz et al. (2020)
Adam parameter $\mathcal{B}_1$	0.9	Andrychowicz et al. (2020) and Kingma and Ba (2015)
Adam parameter $\mathcal{B}_2$	0.999	Andrychowicz et al. (2020) and Kingma and Ba (2015)
Adam parameter ϵ_{adam}	1e-5	Andrychowicz et al. (2020)

5.6.3. Considered values for Key-Corridors-Small

From the initial hyperparameter values, we further tuned the parameters with the CNN architecture on Key-Corridor-Small instances with 1 and 2 key(s). These tuned parameters are then again used as a starting point for tuning the parameters for the Key-Corridors-Big instances. Below, the considered values will be discussed per hyperparameter, as well as some insights gained during the tuning process. Section 5.7 will follow with an overview of the actual parameters used in the experiments.

Number of vectorized environments

The number of vectorized environments was one of the first parameters that was fixed. Training with much more environments than the initially used value of 8 made training much faster in wall clock time. This does lead to a relatively large batchsize, but as there are so many different instances of the Key-Corridors environments, this is actually a good thing. At a certain point, training with even more environments does not further decrease the wall clock training time. This is probably either due to the fact that sample efficiency decreases with too large batch sizes or because the gradient updates actually become the limiting factor instead of the environment simulations. The following values were considered: 8, 128, 200, 400 and 500.

Minibatch size (M)

Larger values for the minibatch size M resulted in slower learning, but did result in a more stable learning process than training with small minibatches. This can probably be explained by the fact that larger minibatches contain more different variations of the Key-Corridors instances, whereas small minibatches might update the policy based on experiences that capture too little of the variation between the instances. The following values were considered: 64, 128, 160, 256, 320, 512, 1280 and 2560.

Trajectory length (T)

The trajectory length T was an important hyperparameter to tune for the Key-Corridors environments. Large values for the trajectory length make learning slow, but too small values do not allow the agent to reach any reward in the the length of trajectory. Especially at the beginning of training this can be a problem, as in this case the agent still needs to explore and might need many steps to reach the goal state and receive a reward. If now the trajectory length is made too short, there might be entire batches that do not contain any experience in which the agent receives a reward. Due to the use of many vectorized environments - each having a different seed, resulting in different initializations - it is not needed to make the trajectories span over several episodes in order to collect experience from multiple instances. The following values for T were considered: 16, 32, 64, 128 and 256.

Discount factor (γ)

Just as the trajectory length, the discount factor γ was an important hyperparameter to tune. Lower values than the initial value of 0.99 resulted in a faster learning, however, too low values put too little emphasis on steps taken further in the past which especially causes problems during the exploration phase, in which the agent might take useful actions many steps before it reaches the actual reward. The following values for γ were considered: 0.8, 0.85, 0.9, 0.95 and 0.99.

Learning rate (μ)

For the learning rate μ, higher values lead to faster training, but too high values result in too large updates, which can make the algorithm unstable. The following learning rates were considered: 2e-4, 2.5e-4, 3e-4 and 4e-4. Furthermore, linearly decreasing rates were considered as well, but these complicated the comparisons between the performance of different architectures.

Number of epochs (K)

The following values were considered for the number of epochs per batch: 2, 3, 4, 5, 6, 8 and 10. More epochs generally results in faster learning, however, to many epochs of training resulted in an unstable

learning process. It was important to consider the number of epochs in combination with the learning rate and the clipping parameter ϵ, as these both also have a big impact on the stability of the algorithm.

Clipping parameter (ϵ)
Picking a value for the clipping parameter ϵ again is a trade-off between training speed and stability. A higher value for the clipping parameter allows for larger updates, which usually results in faster training, however it might cause the algorithm to become unstable. The following values for ϵ were considered: 0.05, 0.1, 0.2, 0.25 and 0.3. A linearly decreasing ϵ was explored as well, but this again - just as with the learning rate - resulted in a more difficult comparison between the different architectures.

GAE parameter (λ)
The GAE parameter λ seemed to be of less importance to the performance of the algorithm. The following values were considered: 0.9, 0.95 and 0.99.

Value function coefficient (c_1)
The initial value of 0.5 for the value function coefficient c_1 already yielded good performance. The following values were considered: 0.1, 0.5 and 1.0.

Entropy coefficient (c_2)
For the entropy coefficient c_2, some other values than the initial value of 0 where tried, however, it could be concluded from the training curves and the logged entropy of the policy that there was no need for a stronger incentive to explore and thus no need for a positive entropy coefficient. Negative values were not considered.

Gradient clipping threshold
For the gradient clipping threshold the following values were considered: 0.1, 0.2, 0.5, 0.8, 2.0 and 5.0. The exact value of this parameter did not seem to have a very significant effect on the performance of the algorithm, as also suggested by Andrychowicz et al. (2020) .

Hyperparameters of Adam
For the hyperparameters of Adam - $\mathcal{B}_1, \mathcal{B}_2$ and ϵ_{adam} - the initial (default) values were used, which were not tuned further.

It is important to emphasize that this was an iterative process instead of a full grid search over all possible combinations of the above mentioned hyperparameter values. Some of the hyperparameters were varied in conjunction with other hyper-parameters, for example, trajectory length, batch size, and the number of vectorized environments are strongly correlated and were therefore often considered together. Some other parameters - such as the VF coefficient - were considered individually by keeping all other parameters constant.

5.6.4. Adjusted values for Key-Corridors-Big

For the Key-Corridors-Big instances, the same hyperparameter values as those used for the Key-Corridors-Small instances were used as starting point. The algorithm is again tuned with the CNN architecture and on Key-Corridors-Big instances with either 1 or 2 key(s).

As the instances of Key-Corridors-Big require more exploration and also more steps to solve the instances under an optimal policy, a larger value for the trajectory length T was required. Moreover, a slightly higher value for the discount factor γ also yielded better performance. This fits with the longer trajectory length and the fact that more exploration is needed, as it then makes sense to give a bit more weight to experiences gained further in the past. Because of the longer trajectory length, the batch size also becomes larger. Lastly, for the Key-Corridors-Big instances, we train for more epochs on the same batch of experiences, resulting in faster training. We reason that this was possible - without the training process becoming unstable - due to the larger batch size.

5.7. Used hyperparameters, activation functions and weight initialization method

This section will give an overview of the used hyperparameters of PPO, the weight initialization of the weights of the neural network layers and the distribution of these weights over the layers of the different architectures. Lastly, we will also mention which activation functions were used.

5.7.1. Hyperparameters of PPO

Table 5.4 gives an overview of all the used hyperparameter values of PPO during the experiments. Hyperparameters that differ between the Key-Corridors-Small and Key-Corridors-Big environment are shown in bold.

Table 5.4: Overview of the used hyperparameter values of PPO for both Key-Corridors-Small and Key-Corridors-Big.

Hyperparameter	Values for Key-Corridors-Small	Values for Key-Corridors-Big
Minibatch size (M)	1280	1280
Trajectory length (T)	32	**128**
Number of vectorized environments	400	400
Batch size $\lvert\mathcal{B}\rvert$	400 $\times$ 32 = 12800	**400 $\times$ 128 = 51200**
Discount factor (γ)	0.9	**0.93**
GAE parameter (λ)	0.95	0.95
Learning rate (μ)	3e-4	3e-4
Number of epochs (N_{epochs})	4	**8**
VF coefficient (c_1)	0.5	0.5
Entropy coefficient (c_2)	0.0	0.0
Clipping parameter (ϵ)	0.2	0.2
Gradient clipping threshold	0.8	0.8
Adam parameter $\mathcal{B}_1$	0.9	0.9
Adam parameter $\mathcal{B}_2$	0.999	0.999
Adam parameter ϵ_{adam}	1e-5	1e-5

5.7.2. Weight initialization and activation functions

The weights in each layer of all the architectures are initialized with orthogonal initialization as described by Saxe et al. (2014). Furthermore, the gain values shown in table 5.5 are applied to each of these layers during initialization. In general, all the feature extractor layers are initialized with a gain of $\sqrt{2}$. This includes the linear layer which is used in GAP. Usually, linear layers are initialized with a gain of 1, but we found empirically that a value of $\sqrt{2}$ for this linear layer gave better results in our specific setup. The last linear layers of the critic (value network) is initialized with a gain value of 1, which is the default for layers that are not followed by an activation function. The last layer of the actor (policy network), is initialized with a much smaller gain value of 0.01, as suggested by Andrychowicz et al. (2020). All the other linear layers in the actor and critic networks - this includes the MLP feature extractor - are initialized with a gain value of $\frac{5}{3}$, which is the default value for layers that are followed by a Tanh activation function. Table 5.5 also gives an overview of the used activation functions for the various layers. Lastly, all biases are initialized at 0.

Table 5.5: Overview of the gain values used for weight initialization and the used activation functions per layer.

Layer	Activation	Gain
CNN and R-GCN	ReLU	$\sqrt{2}$
GAN	-	$\sqrt{2}$
Linear layer of GAP	-	$\sqrt{2}$
Last layer of the policy network	-	0.01
Last layer of the value network	-	1
All other linear layers	Tanh	$\frac{5}{3}$

5.7.3. Overview of architectures and their number of trainable parameters

To provide a fair comparison, all architectures being directly compared to each other have approximately the same number of trainable parameters. For each of the architectures, the total number of trainable parameters and their distribution across the layers is displayed in table 5.6. During tuning with the CNN architecture, different layer sizes were considered for both the convolutional layers in the feature extractor as the following linear layers, however, making these much smaller or larger did not have favorable effects.

It is important to realize that the two linear layers that follow the feature extractor (Linear Layer 1 and 2) are not shared by the actor and critic networks and that they both have a different output layer. The actor network has 5 output values, one for each of the possible actions, and the critic network gives a single predicted value as output. The notation used in table 5.6 to indicate the size of each layer is introduced below.

Linear Layer : To indicate the size of linear layers, the following notation is used: $(h_{\text{in}}, h_{\text{out}})$. Here h_{in} indicates the number of input features and h_{out} represents the number of output features.

CNN layer : The size of a convolutional layer of a CNN is indicated with the following notation: $(h_{\text{in}}, h_{\text{out}}, n, m)$. The variables n and m indicate the width and the height of the kernel, respectively. Here $n \times m$ can be compared to the number of relations in a R-GCN layer.

GCN layer : To indicate the size of a GCN layer, the following notation is used: $(h_{\text{in}}, h_{\text{out}})$.

R-GCN layer : The size of a R-GCN layer is indicated as follows: $(h_{\text{in}}, h_{\text{out}}, |\mathcal{R}|)$. In this case, $|\mathcal{R}|$ represents the number of relations.

GAN layer : A single GAN layer contains both linear layers for the attention mechanism and for the feature transformation. In addition, a GAN layer can consist of multiple attention heads. The size of a GAN layer is therefore indicated with the following notation: $K(h_{\text{in}}, h_{\text{out}}) + K(2h_{\text{in}}, 1)$. Here, K represents the number of attention heads.

Each of these layers also has a trainable bias vector of size $|h_{\text{out}}|$, which is also included in the total number of trainable parameters.

Table 5.6: Number of trainable parameters per architecture, as well as the used readout function and the dimensions of the individual layers. Max is used as abbreviation for max-pooling.

	Feature extractor 1	Feature extractor 2	Readout	Linear layer 1	Linear layer 2	Total parameters
MLP	(1.519,465)	(465,466)	Flatten	(466,128)	(128,128)	1.075.810
CNN_{flat}	(31,77,3,3)	(77,77,3,3)	Flatten	(3.773,128)	(128,128)	1.074.166
GCN_{flat}	(31,82)	(82,82)	Flatten	(4.018,128)	(128,128)	1.071.318
$\text{R-GCN}_{\text{CNN-flat}}$	(31,77,9)	(77,77,9)	Flatten	(3.773,128)	(128,128)	1.074.166
$\text{R-GCN}_{\text{domain-flat}}$	(31,78,7)	(78,78,7)	Flatten	(3.822,128)	(128,128)	1.071.382
CNN_{max}	(31,114,3,3)	(114,114,3,3)	Max	(114,128)	(128,128)	211.462
$\text{R-GCN}_{\text{domain-max}}$	(31,129,7)	(129,129,7)	Max	(129,128)	(128,128)	211.042
$\text{R-GCN}_{\text{CNN+domain-max}}$	(31,103,11)	(103,103,11)	Max	(103,128)	(128,128)	211.676
$\text{R-GCN}_{\text{CNN+random-max}}$	(31,103,11)	(103,103,11)	Max	(103,128)	(128,128)	211.676
$\text{R-GCN}_{\text{GAN-max}}$	(31,101,9)	R-GCN: (101,101,9) GAN: 3(101,101)+ 3(202,1)	Max	(101*128)	(128*128)	210.838

5.8. Used hardware

All of the experiments of this thesis are run on Google Colaboratory, which provides free access to a range of GPUs. The exact type of GPU that is provided varies over time and this is unfortunately not up to the user to decide. For all architectures being directly compared to each other, an effort has been made to ensure as much as possible that, for the same seed, the architectures are tested on the same GPU. Different seeds of the same experiment might however be run on different machines.

6

Experiments and results

In this section we present the experiments carried out to test the different architectures and report and discuss the corresponding results. These experiments are designed to test specific elements of the proposed architectures, such as the readout functions or the added domain-specific relations in R-GCN$_{\text{domain}}$, for sample efficiency and in- and out-of-distribution generalization. All experiments were conducted for 8 different seeds to account for randomness in our methods. Due to the lack of computational resources we chose to not test on all possible environments for each experiment, both with respect to the number of keys and the size of the environment. The details of which environments were considered will be discussed per experiment.

All 'training curves' presented in the results are produced in the same way as the generalization curves. The agent is periodically evaluated with the same frequency as is used to evaluate generalization performance, however, for the training curves it is evaluated on the actual training environment. This resulted in more readable plots, which are more similar to the plots used to display generalization performance and therefore more suitable for comparison.

6.1. Experiment 1: Effects of adding domain-specific relations

In this first experiment MLP, CNN$_{\text{flat}}$, GCN$_{\text{flat}}$, R-GCN$_{\text{CNN-flat}}$ and R-GCN$_{\text{domain-flat}}$ are compared against each other. In this case we apply flattening as readout function for all of the graph-based feature extractors and the CNN, indicated with the '-flat' subscripts. This experiment is conducted to first compare the graph-based methods to CNN in a way that most closely resembles a traditional way of using a CNN. The architectures are compared against each other on sample efficiency, in-distribution-generalization and out-of-distribution generalization. Testing for out-of-distribution generalization to a different size environment is not possible with these architectures - without adding extensive padding - due to the flattening operation. This experiment is only conducted on the Key-Corridors-Small environment with instances containing either 1 or 2 key(s) and for out-of-distribution generalization, we evaluate on instances with 3 keys. In Experiment 3 we do train the best performing architectures on the other environment and with a varying number of keys.

In this experiment we hope to confirm that CNN$_{\text{flat}}$ outperforms MLP, as it has stronger relational inductive biases that are known to work well for grid-shaped observations. We have also included GCN$_{\text{flat}}$ in this experiment to validate whether the problem of indistinguishability, highlighted in figure 5.5 of section 5.4.3, is indeed an issue during training. Furthermore, we also expect to see that R-GCN$_{\text{CNN-flat}}$ performs comparable to CNN$_{\text{flat}}$, indicating that we indeed succeeded in imitating CNN$_{\text{flat}}$ with a graph neural network. Lastly, we want to examine the effects of applying domain-specific relations with R-GCN$_{\text{domain-flat}}$. Here we aim to show that its strong relational inductive biases, that are specifically tailored for the environment, enable this method to outperform the others.

Note that these results will not be directly comparable to those in the following experiments. Due to the flattening operation as readout function in the architectures used in this experiment, a large linear layer is required after the readout, which has a lot of parameters. To make these architectures comparable - in terms of the number of trainable parameters - to those that use other readout functions, the number of parameters in the feature extractors used in this experiment would have to be drasti-

cally reduced, or drastically increased in those architectures that do not use flattening as a readout. Therefore, we decided to make this a separate experiment.

6.1.1. Sample efficiency

Figure 6.1 displays the training curves for all the different architectures trained on the Key-Corridors-Small environment with instances containing 1 or 2 key(s).

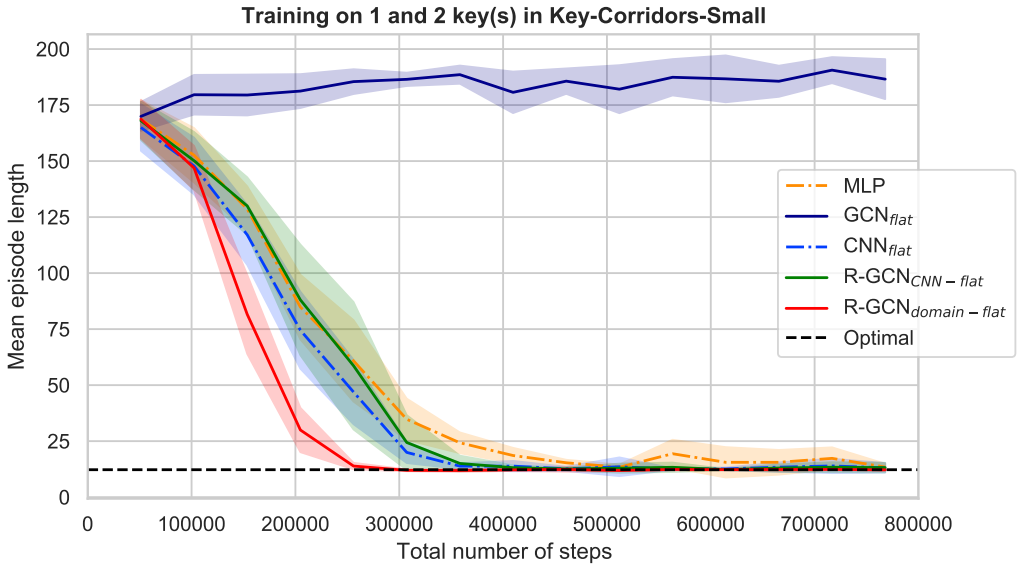


Figure 6.1: Training curves of all architectures with flattening as readout, on Key-Corridors-Small environment with instances containing both 1 and 2 key(s). All lines display the average over 8 different seeds and the shaded areas indicate the standard deviations. The striped black line represents the mean episode length that is obtainable by an optimal policy.

There are several interesting observations to be drawn from figure 6.1. First, it is clearly visible from the results that GCN$_{flat}$ does not perform well. We argue that this is likely due to the problem of non-distinguishable observations, which is described in figure 5.5 of section 5.4.3. The underlying problem of having only one type of edge for which the weights are shared over the entire graph, makes it in general much harder to encode the relative positions of nodes, whilst these relative positions are actually very important for solving the Key-Corridors environments. Furthermore, CNN$_{flat}$ slightly outperforms MLP in terms of sample efficiency, although the difference is not significant. MLP also seems to be a bit less stable around the optimal policy, which might indicate that it has difficulties learning a policy that generalizes over all the different initializations of the Key-Corridors-Small environment. Figure 6.1 also reveals that we indeed seemed to have succeeded in imitating CNN$_{flat}$ with R-GCN$_{CNN\text{-}flat}$. This is an important finding, as it shows that there is no benefit to the R-GCN-based architecture before adding the domain-specific relations. The final, but perhaps most important observation, is the fact that R-GCN$_{domain\text{-}flat}$ outperforms all other architectures. It has the best sample efficiency and also seems to be more stable around the optimal policy than any of the other architectures.

6.1.2. In-distribution generalization

Figure 6.2 shows the result of testing for in-distribution generalization with all the different architectures. On the left side of the figure, the training curve is displayed, where this time we only train of 70% of all the possible instances of the Key-Corridors-Small environment with 1 and 2 key(s). On the right hand side, the results are displayed for evaluation on the remaining 30% of instances, which were withheld during training. There are two important insights that arise from these results. First, all methods perform roughly equally well on the withheld instances as on the other 70% of instances that they were trained on. Second, the training curves of training on 70% of the possible instances are also approximately equal to those of training on all instances, displayed in figure 6.1. The first observation might be due to the fact that with 70% of all initializations, we already train on so many different initializations that good in-distribution generalization is required to reach an optimal policy. The second insight - training on

70% of all instances leads to approximately the same training curves as training on all instances - led to the decision to also use the results of training on 70% of the instances for the evaluation of sample efficiency in some of the following experiments, instead of running separate experiments with 100% of the possible instances for this.

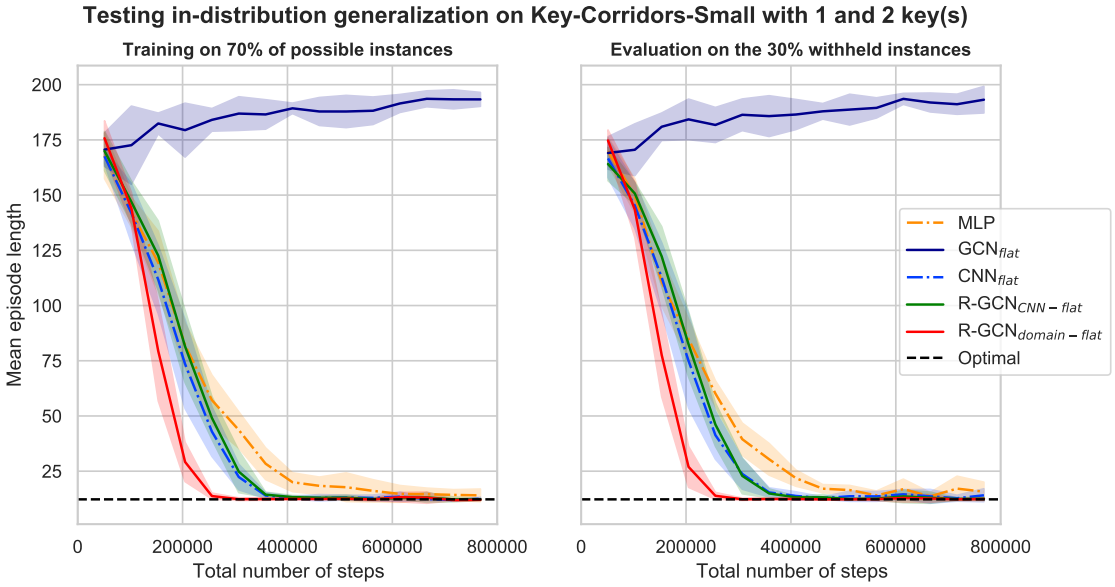


Figure 6.2: Testing in-distribution generalization of all architectures with flattening as readout. The left-hand plot displays the training curves, where the agents are trained on the Key-Corridors-Small environment which is initialized with both 1 and 2 key(s). However, this time 30% of all possible initialization are withheld. The right-hand plot shows the evaluation of the agents on the 30% of instances that were withheld during training. All lines display the average over 8 different seeds and the shaded areas indicate the standard deviations. The striped black line represents the mean episode length that is obtainable by an optimal policy.

6.1.3. Out-of-distribution generalization

The results of testing out-of-distribution generalization performance of the different architectures are displayed in figure 6.3. These results are obtained by periodically evaluating the agents on the Key-Corridors-Small environment with 3 keys, during training on Key-Corridors-Small instances with either 1 or 2 key(s). The corresponding training curves are those shown in figure 6.1. In addition to the evaluation curves of figure 6.3, table 6.1 lists the lowest mean episode length achieved by each of the architectures during the periodic evaluation of their generalization performance.

Table 6.1: Lowest mean episode length achieved by each of the architectures during the periodical evaluation of their generalization performance to Key-Corridors-Small instances with 3 keys. AI agents were trained on Key-Corridors-Small instances with either 1 or 2 key(s). The mean values represent the average over the 8 seeded runs, each of which evaluated the agent for 30 episodes. The best results are shown in bold and the standard deviation is denoted by SD.

Trained on:	Key-Corridors-Small with 1 or 2 key(s)
Evaluated on:	Key-Corridors-Small with 3 keys
MLP	76.30 (SD: 28.0)
GCN$_{flat}$	166.0 (SD: 0)
CNN$_{flat}$	40.65 (SD: 5.59)
R-GCN$_{CNN-flat}$	44.79 (SD: 7.75)
R-GCN$_{domain-flat}$	**20.75** (SD: 0.15)

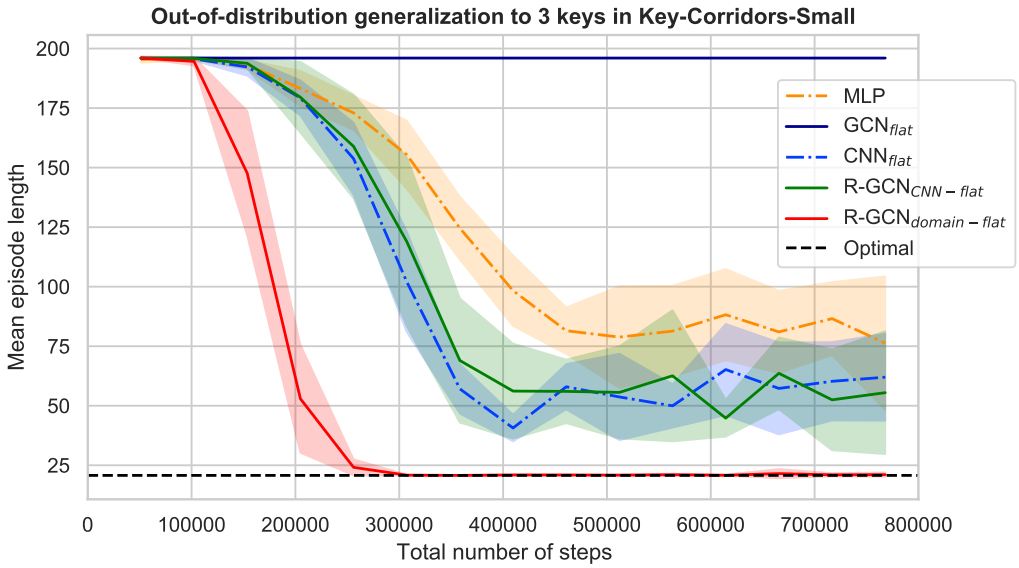


Figure 6.3: Out-of-distribution generalization to Key-Corridors-Small instances with 3 keys. The corresponding training curves are displayed in figure 6.1, for which the agents are trained on Key-Corridors-Small instances with either 1 or 2 key(s). All lines display the average over 8 different seeds and the shaded areas indicate the standard deviations. The striped black line represents the mean episode length that is obtainable by an optimal policy.

The most important observation from figure 6.3, is the fact that R-GCN$_{\text{domain-flat}}$ has really good out-of-distribution generalization performance. Without ever having seen any instance with 3 keys, it still obtains an optimal policy. Next to this, it can also be seen that CNN$_{\text{flat}}$ and R-GCN$_{\text{CNN-flat}}$ again have very similar performances, both showing reasonable out-of-distribution performance but ending up with a policy that, on average, requires significantly more steps to solve the instances than R-GCN$_{\text{domain-flat}}$ does. MLP again seems to perform a bit worse than CNN$_{\text{flat}}$ and R-GCN$_{\text{CNN-flat}}$. Lastly, GCN$_{\text{flat}}$ does not manage to solve any of the instances, which is not surprising as it also did not manage to learn a good policy during training.

6.1.4. Conclusion

From this experiment it can be concluded that adding domain-specific relational inductive biases leads to better sample efficiency during training and a better out-of-distribution generalization performance to the same environment with an extra key. Moreover, the experiment shows that using a R-GCN with the same type of relations as the CNN architecture does not cause these improvements. This suggests that the improved performance of R-GCN$_{\text{domain-flat}}$ over CNN$_{\text{flat}}$ is not just caused by the use of R-GCN, but actually stems from the specific relations used by the R-GCN$_{\text{domain-flat}}$ architecture. For in-distribution generalization, all methods perform equally well. The best performing GNN-based method is R-GCN$_{\text{domain-flat}}$ and CNN$_{\text{flat}}$ is the best performing baseline. Therefore, the performance of these two architectures will be further explored in the following experiments.

6.2. Experiment 2: Comparing different readout functions

Instead of using flattening as readout function, in this experiment the 4 other proposed readout functions - average-pooling, sum-pooling, max-pooling and GAP - are applied and compared. These readout functions are preferred over flattening, as they allow us to use the trained agents on variable size input without having to add extensive padding. The readout functions are tested with the R-GCN$_{\text{domain}}$ architecture, as it was concluded from experiment 1 that this is the best performing GNN-based method. To determine which readout function works best, agents are trained on the Key-Corridors-Small environment with 1 and 2 key(s) and are tested for sample efficiency, in-distribution generalization and out-of-distribution generalization to both instances with an additional keys and the larger environment. In the following figures, 'avg', 'sum' and 'max' are used as abbreviation for average-pooling, sum-pooling and max-pooling, respectively.

6.2.1. Sample efficiency

Figure 6.4 shows the training curves of R-GCN$_{\text{domain}}$ with the 4 different readout functions. It is clear that GAP and max-pooling are the better performing readout functions, with max-pooling being even a bit more sample efficient than GAP. Furthermore, based on the standard deviations, max-pooling also seems to be more consistent over the 8 runs with different seeds. Average- and sum-pooling both perform much worse, although average-pooling eventually does seem to learn a better policy, at least for some of the runs.

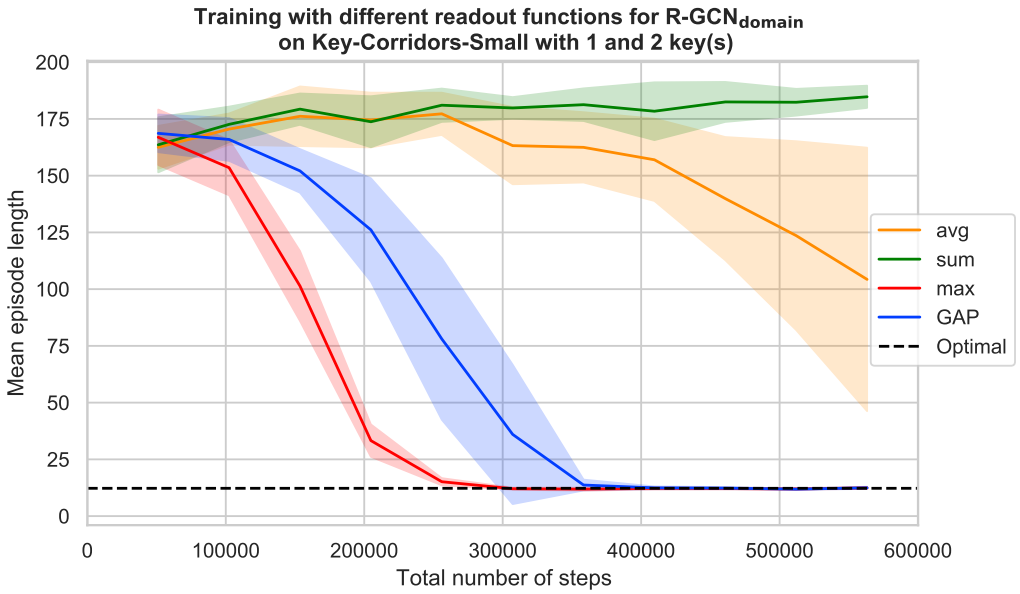


Figure 6.4: Training curves of R-GCN$_{\text{domain}}$ with 4 different readout functions, on the Key-Corridors-Small environment with both 1 and 2 key(s). All lines display the average over 8 different seeds and the shaded areas indicate the standard deviations. The striped black line represents the mean episode length that is obtainable by an optimal policy.

6.2.2. In-distribution generalization

For in-distribution generalization performance, we only consider max-pooling and GAP, because the other two pooling methods - average-pooling and sum-pooling - did not show good performance during training and did not reach an optimal policy on the training environments. Figure 6.5 shows both the results of training on 70% of the possible instances (left-hand plot) and the results of evaluating on the 30% of instances that were withheld during training (right-hand plot). For both readout functions, the in-distribution generalization performance is roughly equal to the actual training performance and the max-pooling readout function again slightly outperforms GAP.

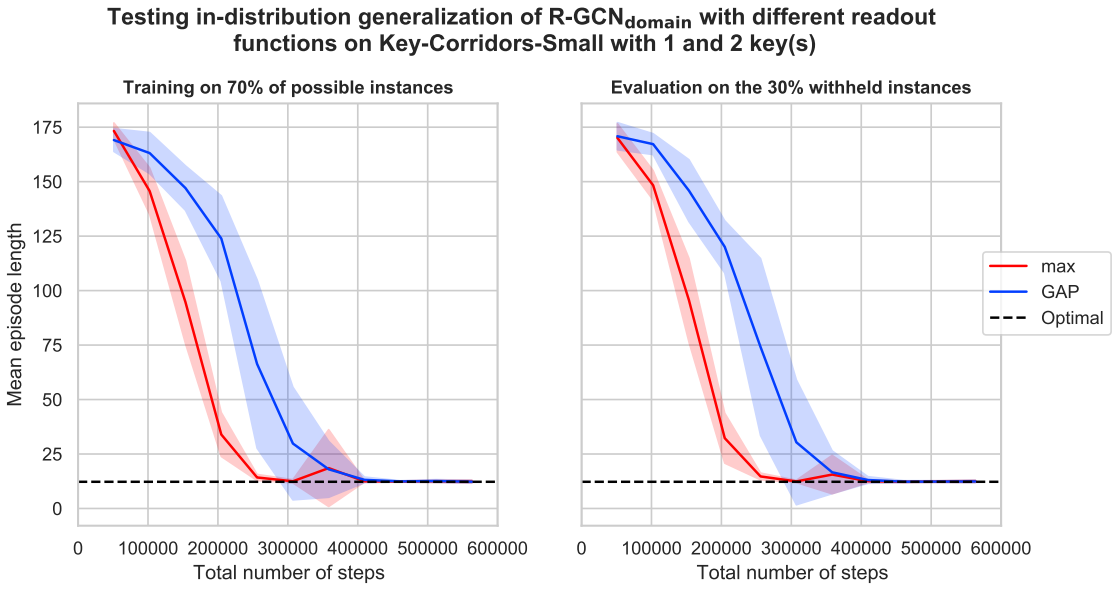


Figure 6.5: Testing in-distribution generalization of R-GCN$_{domain}$ with max-pooling (indicated with 'max') and GAP as readout functions. The plot on the left displays the training curves, where the agents are trained on 70% of all instances of Key-Corridors-Small with both 1 and 2 key(s). The plot on the right shows the evaluation of the agents on the 30% of instances that were withheld during training. All lines display the average over 8 different seeds and the shaded areas indicate the standard deviations. The striped black line represents the mean episode length that is obtainable by an optimal policy.

6.2.3. Out-of-distribution generalization

As for in-distribution generalization, only max-pooling and GAP are considered for out-of-distribution performance, as these readout functions performed well during training. The results for testing out-of-distribution generalization performance of R-GCN$_{domain}$ with these two readout functions are displayed in figure 6.6. These results are obtained by periodically evaluating the agents on the Key-Corridors-Small environment with 3 keys and on the Key-Corridors-Big environment with 1 and 2 key(s), whilst training on Key-Corridors-Small instances with 1 and 2 key(s). The corresponding training curves are plotted in figure 6.4.

From from the left-hand plot in figure 6.6 it can be deduced that both architectures perform equally well for out-of-distribution generalization to instances with 3 keys. Both reach an optimal policy on the out-of-distribution instances with 3 keys and they do so after roughly the same amount of environment steps that were required to learn an optimal policy for the actual training environments. Not surprisingly, the difference in sample efficiency between the two architectures is also reflected in these results as well.

As the readout functions allow for generalization to an environment of a different size than the training environment, out-of-distribution generalization performance to the Key-Corridors-Big environment can now be tested as well. From the right-hand plot of figure 6.6, we see that neither of the readout functions lead to an optimal policy in the larger environment and for both readout functions the out-of-distribution generalization performance starts to drop again after more steps of training. The latter, is most probably caused by overfitting on the training environment, which leads to worse generalization performance to the larger environment. Although both readout function do not achieve optimal performance in the Key-Corridors-Big environment, max-pooling does reach a significantly better performance than GAP.

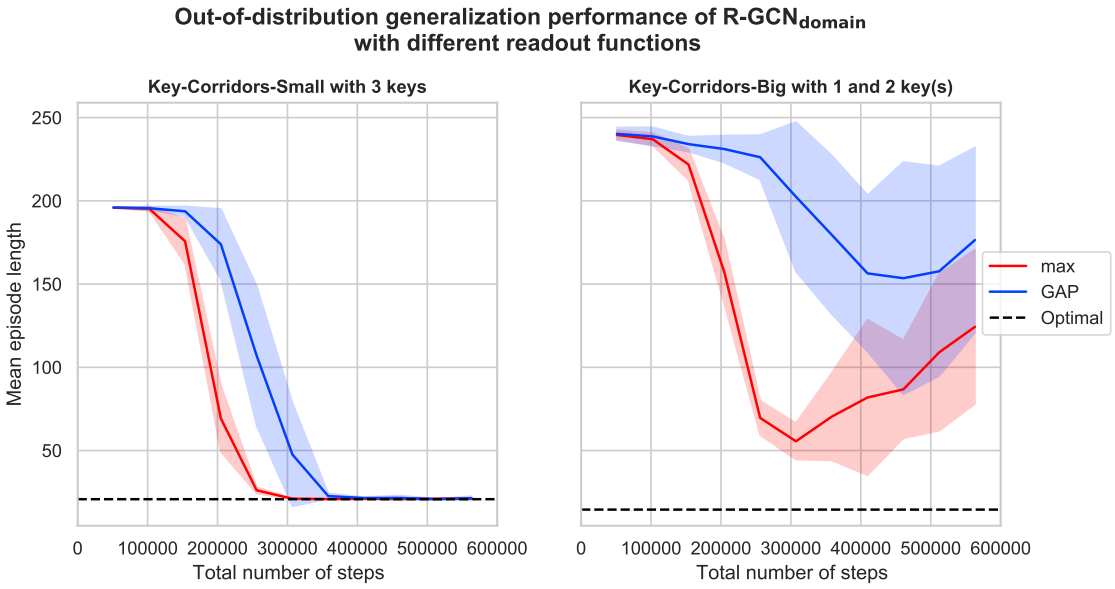


Figure 6.6: Testing out-of-distribution generalization of R-GCN$_{domain}$ with max-pooling (indicated with 'max') and GAP as readout functions. The corresponding training curves are displayed in figure 6.4, for which the agents are trained on Key-Corridors-Small instances with either 1 or 2 key(s). The left-hand plot of this figure displays out-of-distribution generalization performance on Key-Corridors-Small instances with 3 keys, i.e., one key more than in the instances seen during training. The right-hand plot displays out-of-distribution generalization to Key-Corridors-Big. All lines display the average over 8 different seeds and the shaded areas indicate the standard deviations. The striped black line represents the mean episode length that is obtainable by an optimal policy.

6.2.4. Improved explainability through readout functions

Both the GAP and max-pooling readout functions enable us to potentially improve the explainability of the policies learned by our agents. This is because they can be used to reason about which part of the observation was most important in the agent's decision to take a certain action. As this subtopic distracts from and is not directly relevant to this specific experiment and the posed research questions, it is further elaborated in section 9.1 of the Appendix.

6.2.5. Conclusion

Taking all experiments of this section into account, max-pooling is, when combined with the R-GCN$_{domain}$ architecture, the the best performing readout function on our environments. It obtains better sample efficiency than any of the other tested readout functions and also shows better out-of-distribution generalization from Key-Corridors-Small to Key-Corridors-Big.

6.3. Experiment 3: Elaborate testing of the best performing architectures with the best readout function

Based on the results of experiment 1 and 2, the following can be concluded: R-GCN$_{\text{domain}}$ has the best performance of all GNN-based methods, CNN is the better performing baseline architecture and from the 4 readout functions, max-pooling performs best. In this experiment we will further compare the best performing GNN-based architecture (R-GCN$_{\text{domain}}$) against the best performing baseline architecture (CNN), both with max-pooling as readout function for comparability and to allow us to evaluate on different size environments. To indicate the used readout function, we will refer to these architectures with CNN$_{\text{max}}$ and R-GCN$_{\text{domain-max}}$. Both architectures are compared on sample efficiency and in-distribution generalization on both the Key-Corridors-Small and the Key-Corridors-Big environment. This is done for 3 different cases: only training on instances with 1 key, training on instances with either 1 or 2 key(s) and training on instances with 1, 2 or 3 key(s). Out-of-distribution generalization is also tested for both Key-Corridors-Small and Key-Corridors-Big. To do so, agents are trained on instances with 1 or 2 key(s) and generalization performance is tested on instances with 3 keys, as well as generalizing from Key-Corridors-Small to Key-Corridors-Big and vice versa.

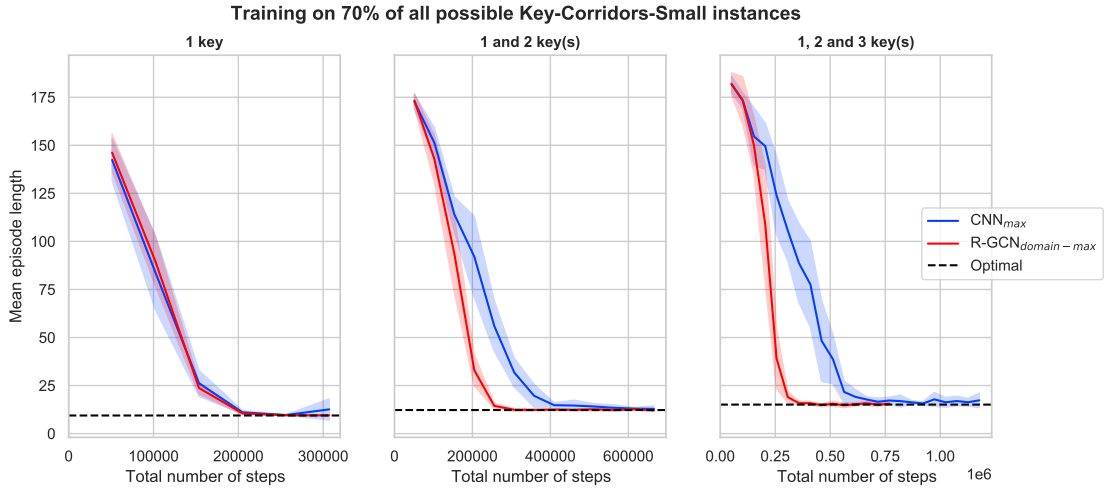
6.3.1. Sample efficiency

Since we found in experiment 1 and 2 that training on 70% of all instances gave approximately the same results for sample efficiency as training on 100% of the instances, we now adopt the same training curves used to test in-distribution generalization to test sample efficiency.

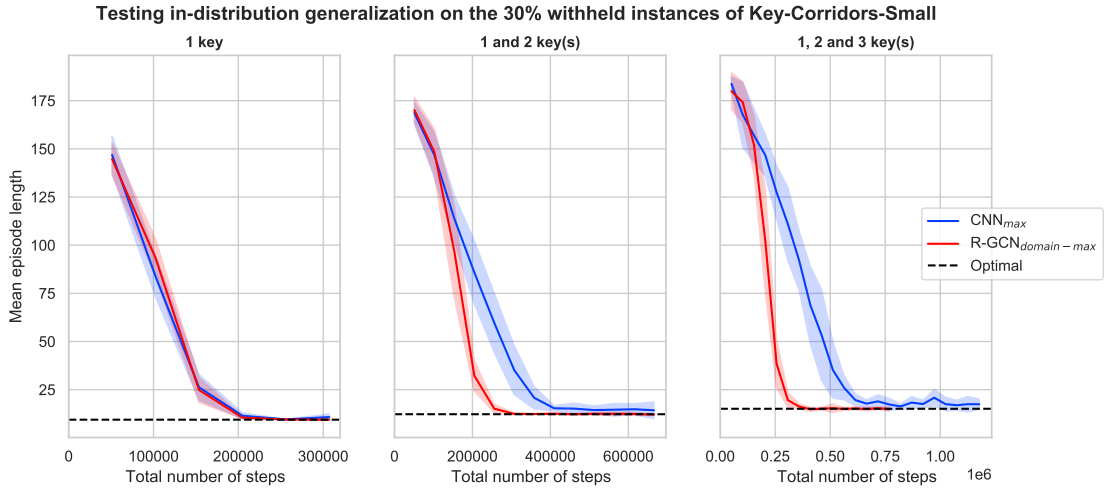
Figures 6.7a and 6.8a show the training curves of training on 70% of all instances for Key-Corridors-Small and Key-Corridors-Big, respectively. For both Key-Corridors-Small and Key-Corridors-Big, training on instances with only 1 key results in approximately equal performance of the CNN$_{\text{max}}$ and R-GCN$_{\text{domain-max}}$ architectures. However, when more keys are added the performance of the two methods starts to diverge. When training on instances with either 1 or 2 key(s) and instances with 1, 2 and 3 key(s), R-GCN$_{\text{domain-max}}$ has a better sample efficiency than CNN$_{\text{max}}$, both on Key-Corridors-Small and Key-Corridors-Big. This indicates that the added domain-specific relational inductive biases of R-GCN$_{\text{domain-max}}$ enable the agent to solve the underlying problem of the Key-Corridors instances with less samples than it needs when using the relational inductive biases of the CNN architecture.

6.3.2. In-distribution generalization

Figures 6.7b and 6.8b show the results of evaluating on the 30% of instances that were withheld during training for Key-Corridors-Small and Key-Corridors-Big, respectively. We again conclude that in-distribution generalization performance is consistently comparable to the actual training performance for both architectures. For the Key-Corridors-Small there seems to be slightly worse performance of CNN$_{\text{max}}$ on the withheld instances with 1 and 2 key(s) as well as on the instances with 1, 2 and 3 key(s). However, based on the standard deviations, we argue that this is too little of a difference to draw any conclusions from.

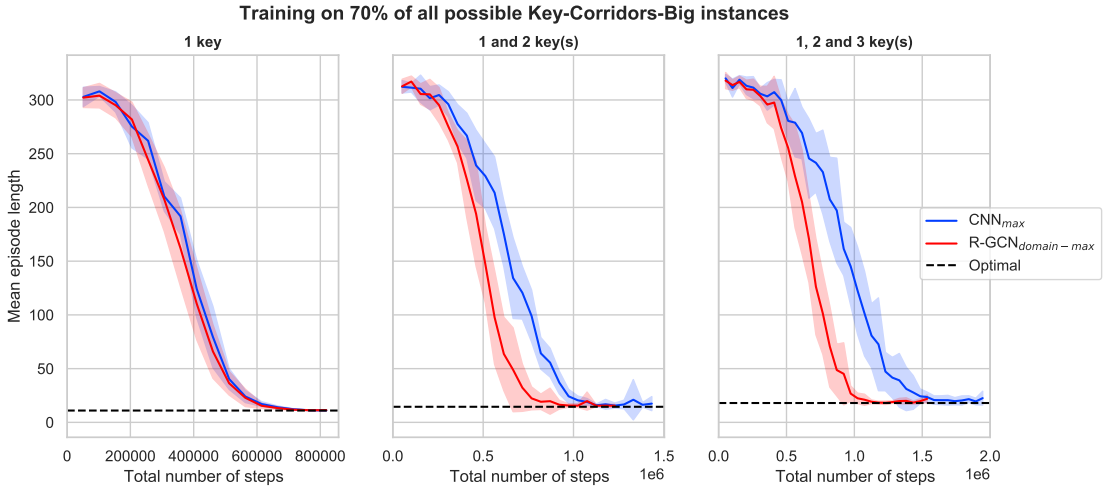


(a)

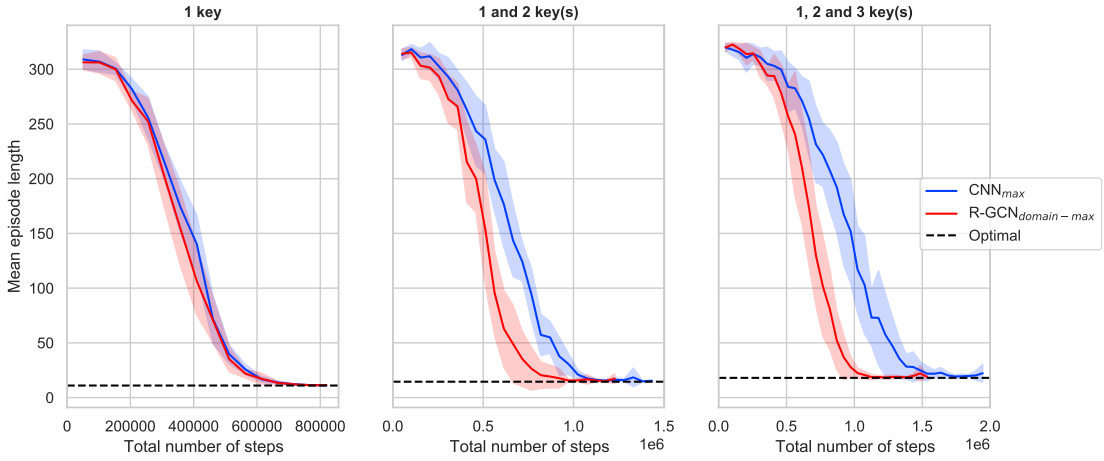


(b)

Figure 6.7: Testing in-distribution generalization of R-GCN$_{\text{domain-max}}$ and CNN$_{\text{max}}$ on Key-Corridors-Small. The upper plots (6.7a) display the training curves, where the agents are trained on 70% of all instances of Key-Corridors-Small with different amount of keys. The lower plots (6.7b) show the evaluation of the agents on the 30% of instances that were withheld during training. All lines display the average over 8 different seeds and the shaded areas indicate the standard deviations. The striped black line represents the mean episode length that is obtainable by an optimal policy.



(a)



(b)

Figure 6.8: Testing in-distribution generalization of R-GCN$_{\text{domain-max}}$ and CNN$_{\text{max}}$ on Key-Corridors-Big. The upper plots (6.8a) display the training curves, where the agents are trained on 70% of all instances of Key-Corridors-Big with different amount of keys. The lower plots (6.8b) show the evaluation of the agents on the 30% of instances that were withheld during training. All lines display the average over 8 different seeds and the shaded areas indicate the standard deviations. The striped black line represents the mean episode length that is obtainable by an optimal policy.

6.3.3. Out-of-distribution generalization

Figures 6.9a and 6.10a show the results of training on 100% of the possible instances with 1 and 2 key(s) of the Key-Corridors-Small and Key-Corridors-Big environment, respectively. During this training process, both architectures are evaluated for out-of-distribution generalization to instances with 3 keys. In addition, when the architectures are trained on Key-Corridors-Small instances, they are also evaluated for generalization to Key-Corridors-Big instances and vice versa for when the architectures are trained on Key-Corridors-Big instances. These results are displayed in figures 6.9b and 6.10b.

The first conclusion that can be drawn from these results is that R-GCN$_{\text{domain-max}}$ has better out-of-distribution generalization performance to 3 keys than CNN$_{\text{max}}$ for both Key-Corridors-Small and Key-Corridors-Big. The difference between the performance of the two architectures is more pronounced on the Key-Corridors-Small environment than on the Key-Corridors-Big environment. On the Key-Corridors-Big environment, R-GCN$_{\text{domain-max}}$ performs worse than it did for the smaller environment and seems to be a bit more unstable around the optimal policy. CNN$_{\text{max}}$ actually performs better than it did for the Key-Corridors-Small environment. This varying performance of the architectures on the two environments can simply be caused by the actual differences between the two environments, however, it can also be induced by the different hyperparameters that were used to train on the two environments. For Key-Corridors-Big, we for instance train for more epochs on the same batch of collected data than we do for the Key-Corridors-Small. This is done since we are also collecting a larger batch of data, due to the larger trajectory length T. However, it might be that training for more epochs made the performance of R-GCN$_{\text{domain-max}}$ more unstable around the optimal policy or caused overfitting on the training environment. On the contrary, CNN$_{\text{max}}$ seems to actually generalize better to 3 keys for the combination of these hyperparameters and this environment. The hyperparameters had to be altered between the two sizes of the environments in order to also achieve good results with KeyCorridorsBig. Unfortunately, this makes it difficult to train with equal hyperparameters to test whether the different environments or the different hyperparameters are the root cause of the performance discrepancy.

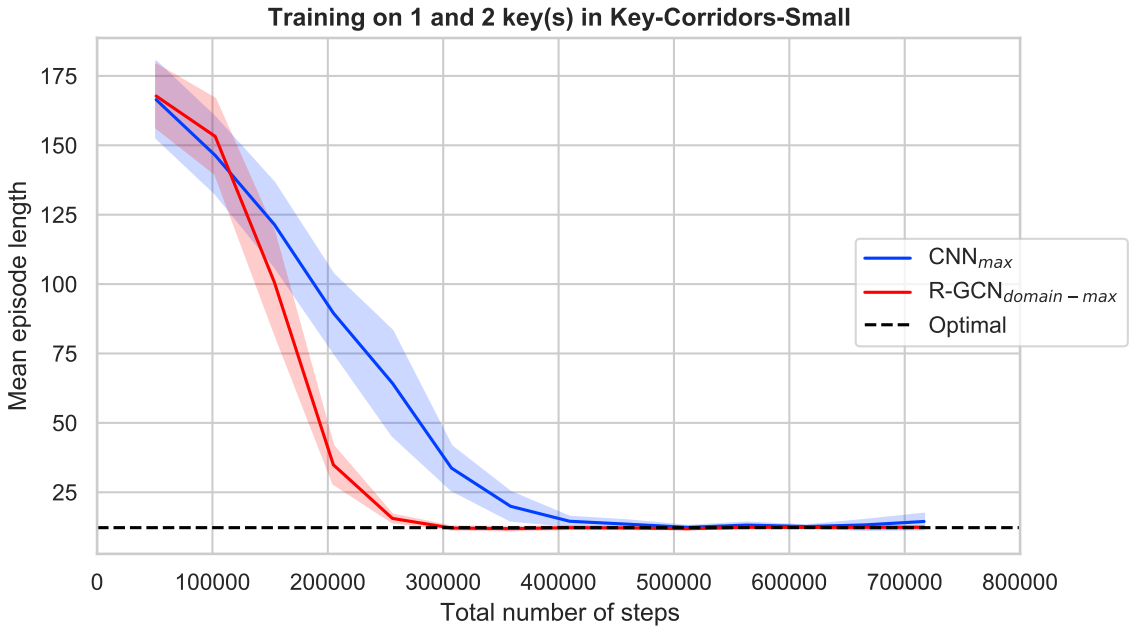
The second conclusion that can be drawn from figures 6.9b and 6.10b is that R-GCN$_{\text{domain-max}}$ also has significantly better out-of-distribution generalization from Key-Corridors-Small to Key-Corridors-Big and vice versa, when compared to CNN$_{\text{max}}$. This might be because the underlying relational reasoning between entities - which R-GCN$_{\text{domain-max}}$ may be better at representing than CNN$_{\text{max}}$ - remains the same between Key-Corridors-Small and Key-Corridors-Big, even though their size is different. Generalizing from the larger to the smaller environment works better for both architectures than generalizing from the Key-Corridors-Small to Key-Corridors-Big. R-GCN$_{\text{domain-max}}$ even almost reaches an optimal policy when generalizing from Key-Corridors-Big to Key-Corridors-Small. As in experiment 2, generalization performance to an environment of a different size begins to worsen again after more training steps. We again reason that this is likely caused by overfitting on the training environment.

A final interesting observation is that training on 100% of the possible instances - instead of only training on 70% as in figure 6.8a - seems to result in a slightly better and more stable policy when training on Key-Corridors-Big.

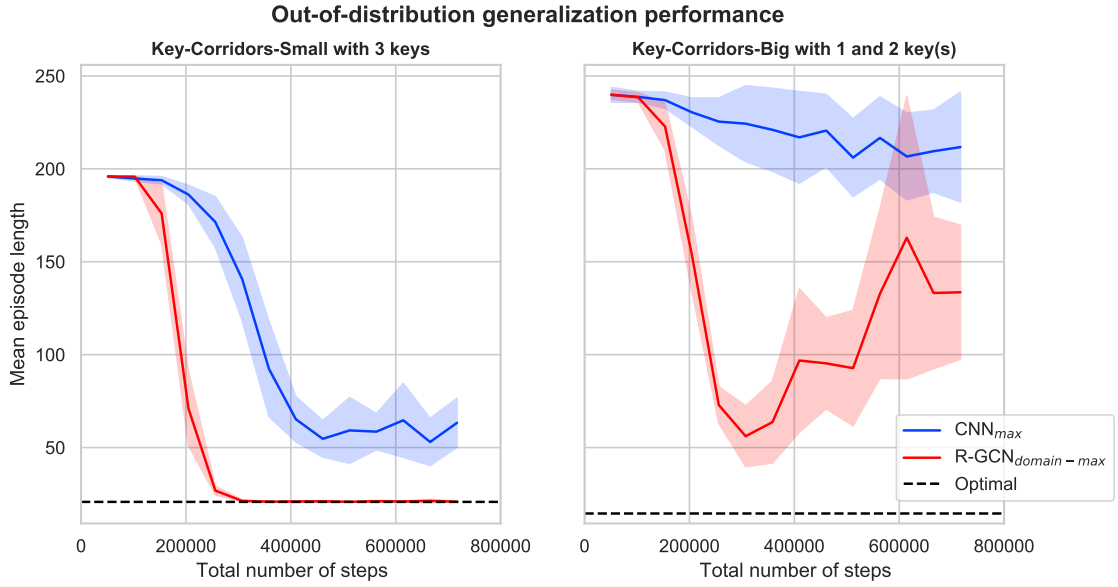
Table 6.2 provides an overview of the lowest mean episode lengths achieved by the CNN$_{\text{max}}$ and R-GCN$_{\text{domain-max}}$ architectures throughout the periodical evaluation of their out-of-distribution generalization performance. It is important to note that the previously discussed plots have shown us that these are snapshots that can be highly dependent on the duration of training and can also deteriorate again with more training.

Table 6.2: Overview of the lowest mean episode lengths achieved by the CNN$_{\text{max}}$ and R-GCN$_{\text{domain-max}}$ architectures during the periodical evaluation of their generalization performance. The reported mean values represent the average over 8 seeded runs, which each evaluated the agent for 30 episodes. The best results are shown in bold and the standard deviation is denoted by SD.

Trained on:	Key-Corridors-Small with 1 or 2 key(s)		Key-Corridors-Big with 1 or 2 key(s)	
Evaluated on:	Key-Corridors-Small with 3 keys	Key-Corridors-Big with 1 or 2 key(s)	Key-Corridors-Big with 3 keys	Key-Corridors-Small with 1 or 2 key(s)
CNN$_{\text{max}}$	53.0 (SD: 12.68)	206.05 (SD: 20.92)	42.93 (SD: 11.85)	75.86 (SD: 9.76)
R-GCN$_{\text{domain-max}}$	**20.78** (SD: 0.17)	**56.10** (SD: 16.37)	**26.78** (SD: 1.61)	**16.33** (SD: 3.0)

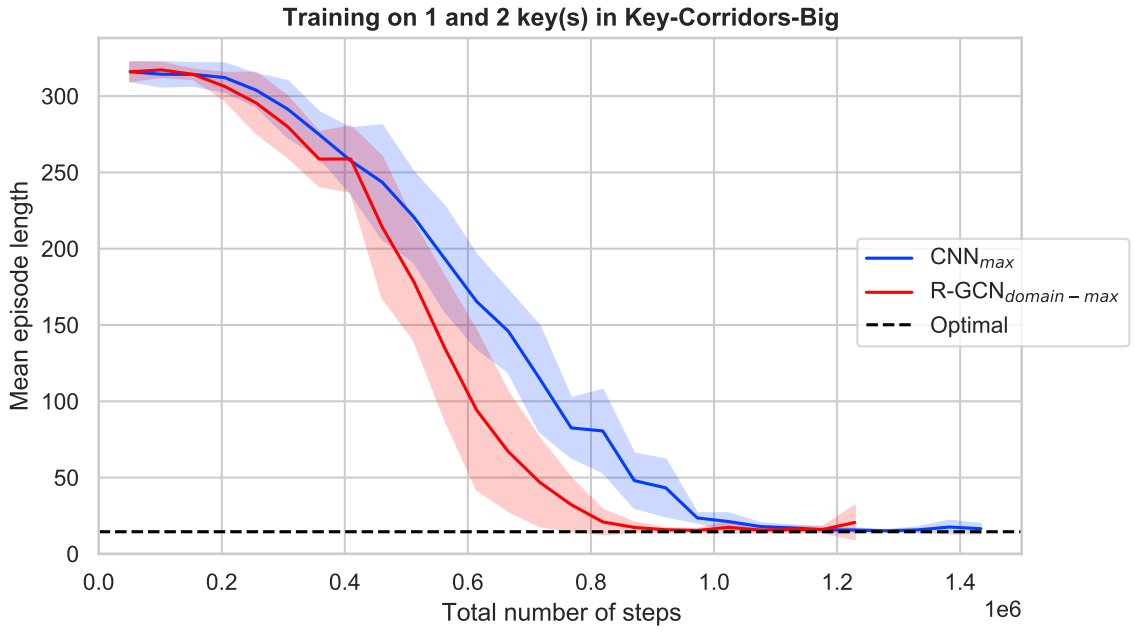


(a)

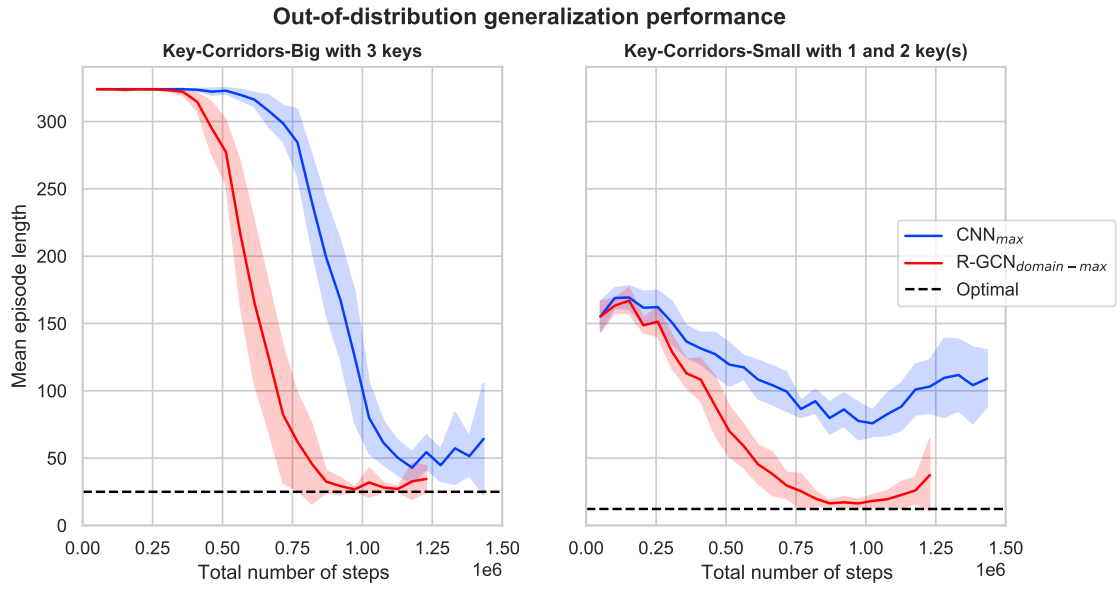


(b)

Figure 6.9: Testing out-of-distribution generalization of R-GCN$_{\text{domain-max}}$ and CNN$_{\text{max}}$. The corresponding training curves are displayed in figure 6.9a, for which the agents are trained on Key-Corridors-Small instances with either 1 or 2 key(s). The left-hand plot of figure 6.9b displays out-of-distribution generalization performance on Key-Corridors-Small instances with 3 keys, i.e., one key more than in the instances seen during training. The right-hand plot displays out-of-distribution generalization to Key-Corridors-Big. All lines display the average over 8 different seeds and the shaded areas indicate the standard deviations. The striped black line represents the mean episode length that is obtainable by an optimal policy.



(a)



(b)

Figure 6.10: Testing out-of-distribution generalization of R-GCN$_{\text{domain-max}}$ and CNN$_{\text{max}}$. The corresponding training curves are displayed in figure 6.10a, for which the agents are trained on Key-Corridors-Big instances with either 1 or 2 key(s). The left-hand plot of figure 6.10b displays out-of-distribution generalization performance on Key-Corridors-Big instances with 3 keys, i.e., one key more than in the instances seen during training. The right-hand plot displays out-of-distribution generalization to Key-Corridors-Small. All lines display the average over 8 different seeds and the shaded areas indicate the standard deviations. The striped black line represents the mean episode length that is obtainable by an optimal policy.

6.3.4. Additional out-of-distribution generalization experiment

An interesting feature of the Key-Corridors environments is that the instances with only 1 or 2 key(s) do not necessarily require the agent to make use of the fact that a key of a certain color can only open a door of the same color. For instances with only 1 key, the agent can hypothetically solve the environment by picking up the first key behind the already unlocked door and then - instead of reasoning about the color of the key and which door it can therefore open - determine behind which door the goal is present and thus open this door. In the case of 2 keys, after picking up the first key, the agent can first determine behind which door there is another key and then open this door to reach the last key, from which the agent can apply the same trick to open the door behind which the goal lies. However, for instances with 3 keys the agent does need to reason about the colors of keys and doors to obtain an optimal policy. In this case, the agent encounters a situation where it is already carrying the first key and the 3 remaining doors, behind which it can find 2 more keys and the goal, are still locked. Now the agent has to decide which door, with a key behind it, it will try to open first. This time the agent can not make this decision solely based on which door has a key behind it, as there are still two doors with keys behind them. Therefore, the agent is forced to reason about the colors of the keys and doors. See figure 6.11 for a visual example of this scenario.

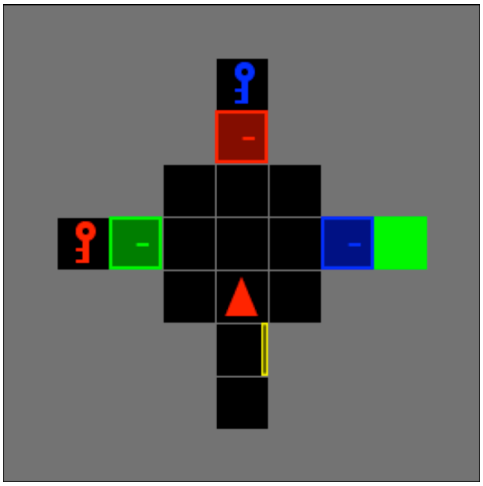
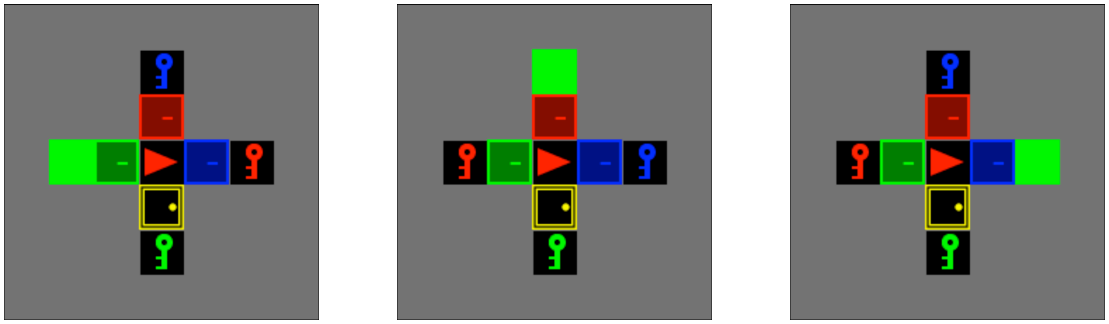


Figure 6.11: Example scenario of a Key-Corridors-Big instance, where the agent is already carrying the green key and now has to decide which door to open next. To make the correct decision the agent will have to reason about the colors of the doors and keys.

This is an important aspect of the Key-Corridors environments, as it could mean that a policy that works well for instances with 1 or 2 key(s) - without learning the color relation between keys and doors - does not scale well to 3 keys. One could argue that the biases added in R-GCN$_{\text{domain-max}}$ favor reasoning about the colors of keys and doors over reasoning about which doors have a key or goal behind them. This might explain, in part, why R-GCN$_{\text{domain-max}}$ generalizes better to instances with 3 keys than CNN$_{\text{max}}$ does. To further explore this, an additional experiment is conducted on a modified version of the Key-Corridors-Small environment. The general setup of this environment is exactly the same as Key-Corridors-Small, however, now there is always a key present behind each door that does not have the goal behind it, regardless of whether the agent needs to use 1, 2 or 3 key(s) to reach the goal. Figures 6.12a, 6.12b and 6.12c each show an example instance of this adjusted environment, requiring 1, 2 and 3 key(s) to reach the goal, respectively. For both the instances that require the use of 2 keys to reach the goal, and those that require 3 keys, the agent will have to reason about the colors of the keys and doors and cannot possibly apply the previously discussed strategy that does not use this information.



(a) Example instance requiring 1 key to reach the goal.

(b) Example instance requiring 2 keys to reach the goal.

(c) Example instance requiring 3 keys to reach the goal.

Figure 6.12: Examples of the different type of instances of the adjusted Key-Corridors-Small environment, which either require the use of 1, 2 or 3 key(s) to reach the goal.

If the agent is now again trained on a mix of instances that require 1 or 2 key(s) to reach the goal, the agent will already encounter scenarios in which it is forced to use the colors of the keys and doors, potentially making it easier to generalize to instances that require 3 keys to reach the goal. The left-hand plot of figure 6.13 shows the results of training on instances requiring 1 or 2 key(s) to reach the goal, and the right-hand plot shows the generalization performance to instances requiring 3 keys.

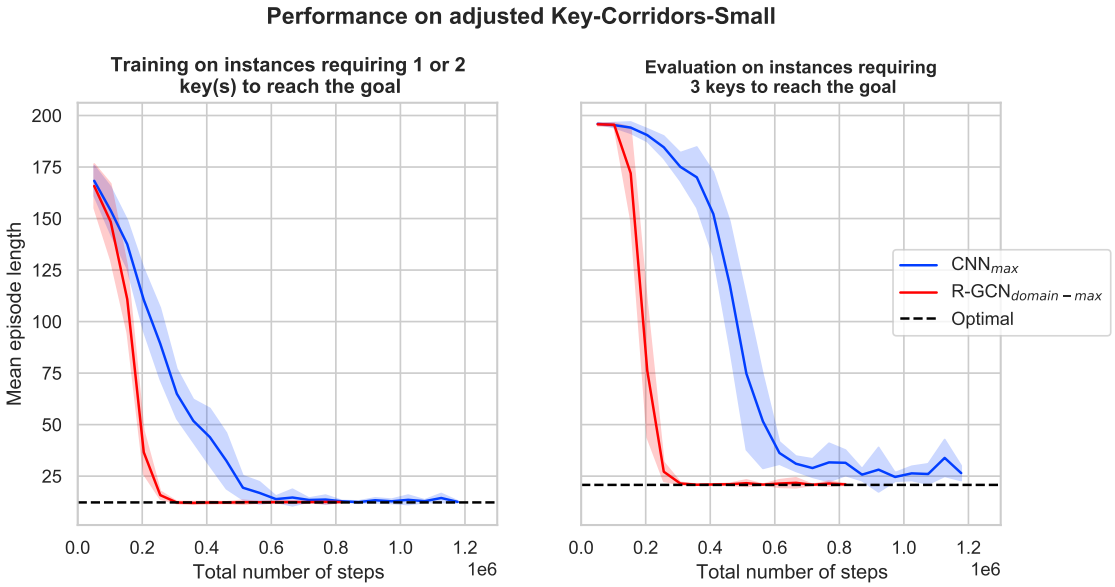


Figure 6.13: Testing out-of-distribution generalization of R-GCN$_{\text{domain-max}}$ and CNN$_{\text{max}}$ on the adjusted Key-Corridors-Small environment. The training curve is displayed in the left plot, for which the agents are trained on the adjusted Key-Corridors-Small instances which either require the agent to use 1 or 2 key(s) to reach a reward. The plot on the right displays the out-of-distribution generalization performance on instances requiring the use of 3 keys to reach a reward. All lines display the average over 8 different seeds and the shaded areas indicate the standard deviations. The striped black line represents the mean episode length that is obtainable by an optimal policy.

Based on the left-hand plot of figure 6.13, it can be concluded that R-GCN$_{\text{domain-max}}$ still has significantly better sample efficiency than CNN$_{\text{max}}$. The right-hand side plot of figure 6.13 shows that the difference in out-of-distribution generalization performance - for generalization to instances which, compared to the training instances, require an extra key to reach the goal - is now much smaller. However, R-GCN$_{\text{domain-max}}$ does still reach a lower mean episode length than CNN$_{\text{max}}$ and also has a much more stable performance during generalization, almost consistently reaching an optimal policy. In table 6.3, we again report the lowest mean episode lengths achieved by the CNN$_{\text{max}}$ and R-GCN$_{\text{domain-max}}$ architectures throughout the periodical evaluation of their out-of-distribution generalization performance.

The fact that the out-of-distribution generalization performance of these two methods is now more similar could indicate that R-GCN$_{\text{domain-max}}$, in the original Key-Corridors environment, indeed made

better use of the relationship between keys and doors during training, than CNN_{max} did. This can actually be seen as a feature of the $R\text{-}GCN_{domain\text{-}max}$ architecture, which is guided by the domain knowledge to learn a policy that better represents the underlying problem of the environment - which is the same for instances with 3 keys - and thus generalizes better. However, it is a satisfying result that $R\text{-}GCN_{domain\text{-}max}$ still has better sample efficiency and out-of-distribution generalization performance than CNN_{max} on the adjusted Key-Corridors-Small environment as it shows that, even in an environment where both architectures are forced to make use of the colors of doors and keys during training, $R\text{-}GCN_{domain\text{-}max}$ still outperforms CNN_{max}.

Table 6.3: Lowest mean episode length achieved by the CNN_{max} and $R\text{-}GCN_{domain\text{-}max}$ architectures during the periodical evaluation of their generalization performance to the adjusted Key-Corridors-Small instances requiring 3 keys to obtain a reward. AI agents were trained on the adjusted Key-Corridors-Small instances which either require 1 or 2 key(s) to reach the goal. The mean values represent the average over 8 seeded runs, each of which evaluated the agent for 30 episodes. The best results are shown in bold and the standard deviation is denoted by SD.

Trained on:	Adjusted Key-Corridors-Small requiring 1 or 2 key(s) to obtain a reward
Evaluated on:	Adjusted Key-Corridors-Small requiring 3 keys to obtain a reward
CNN_{max}	24.60 (SD: 2.11)
$R\text{-}GCN_{domain\text{-}max}$	**20.81** (SD: 0.20)

6.3.5. Conclusion

In general, when comparing $R\text{-}GCN_{domain\text{-}max}$ and CNN_{max}, it can be concluded from this experiment that $R\text{-}GCN_{domain\text{-}max}$ has better sample efficiency for instances with more than 1 key. Both methods have comparable in-distribution generalization performance and $R\text{-}GCN_{domain\text{-}max}$ has a better out-of-distribution generalization performance. All of these results hold for training on Key-Corridors-Small and Key-Corridors-Big, however, $R\text{-}GCN_{domain\text{-}max}$ outperforms CNN_{max} more significantly in out-of-distribution generalization to instances with 3 keys when trained and evaluated on Key-Corridors-Small.

The additional out-of-distribution generalization experiment, conducted in section 6.3.4, verifies that even when both architectures are forced to use the colors of keys and doors during training - they might in reality already do this in the original Key-Corridors environments - $R\text{-}GCN_{domain\text{-}max}$ still outperforms CNN_{max} in sample efficiency and when evaluated for out-of-distribution generalization to an instance that requires the use of an additional key to reach the goal.

6.4. Experiment 4: Are the domain-specific relations the real cause of improvement?

In this experiment the R-GCN$_{\text{CNN+domain}}$ and R-GCN$_{\text{CNN+random}}$ architectures will be used to further validate whether the specific relations of R-GCN$_{\text{domain}}$ are the real cause of improvement over the CNN architecture. First, the R-GCN$_{\text{CNN+domain}}$ architecture will be used to check whether the 'can pick up' and 'can open' relations of R-GCN$_{\text{domain}}$ are the main cause of improvement, instead of the 4 other relations based on the possible movements of the agent: left, right, up and down. To do so, the R-GCN$_{\text{CNN+domain}}$ architecture combines the 9 relations of R-GCN$_{\text{CNN}}$ with the additional 'can pick up' and 'can open' relations, as discussed in section 5.4.6. Secondly, the R-GCN$_{\text{CNN+random}}$ architecture - as introduced in section 5.4.7 - will be applied to check what happens when the relations of the R-GCN$_{\text{CNN}}$ architecture are combined with arbitrary relations, instead of the domain-specific ones of R-GCN$_{\text{domain}}$. With this architecture we aim to show that the 'can pick up' and 'can open' relations, based on domain knowledge, outperform other random relations, thereby demonstrating the importance of domain knowledge.

For both these architectures, max-pooling is again employed as readout function, therefore we will refer to them as R-GCN$_{\text{CNN+domain-max}}$ and R-GCN$_{\text{CNN+random-max}}$ in the following results and discussion. Once again, both the Key-Corridors-Small and Key-Corridors-Big environments are considered for this experiment. For both environments, agents are trained on instances with either 1 or 2 key(s) and evaluated for generalization to instances with 3 keys, as well as generalization to the other environment with 1 or 2 key(s) - the same number of keys as in the training instances. For this experiment we focus on sample efficiency and out-of-distribution generalization, as there was no large difference in the in-distribution generalization performance of the different architectures tested in the previous experiments.

6.4.1. Sample efficiency

Figures 6.14a and 6.15a feature the results for training on 100% of the possible instances with either 1 or 2 key(s) of Key-Corridors-Small and Key-Corridors-Big, respectively. These results will be used to evaluate the sample efficiency of the R-GCN$_{\text{CNN+domain-max}}$ and R-GCN$_{\text{CNN+random-max}}$ architectures. Two important insights can be drawn from these results, which count for both Key-Corridors-Small and Key-Corridors-Big: first, R-GCN$_{\text{CNN+domain-max}}$ has approximately the same sample efficiency as R-GCN$_{\text{domain-max}}$, and second, R-GCN$_{\text{CNN+random-max}}$ has a very similar sample efficiency to that of CNN$_{\text{max}}$. The first insights tells us that the 'can pick up' and 'can open' relations of R-GCN$_{\text{domain-max}}$ are indeed the driving factor for its good performance, as adding these relations to the relations of R-GCN$_{\text{CNN-max}}$ yields almost identical results as those obtained with R-GCN$_{\text{domain-max}}$. From the second insight, it can be concluded that using the 'can pick up' and 'can open' relations based on domain knowledge, leads to a better performance than using the random relations of R-GCN$_{\text{CNN+random-max}}$. This is a satisfactory result, as it indicates that there is indeed a need for domain-knowledge to improve sample efficiency.

6.4.2. Out-of-distribution generalization

During the training process discussed in section 6.4.1, the different architectures are periodically evaluated for out-of-distribution generalization to instances with 3 keys. When trained on Key-Corridors-Small instances, the architectures are also evaluated for generalization to Key-Corridors-Big instances and vice versa for the architectures trained on Key-Corridors-Big instances. Figure 6.14b shows the generalization results for the architectures that were trained on Key-Corridors-Small and figure 6.14b shows the results for architectures trained on Key-Corridors-Big.

For generalization to instances with 3 keys, both on Key-Corridors-Small and Key-Corridors-Big, it can be concluded that R-GCN$_{\text{CNN+domain-max}}$ has comparable performance to that of R-GCN$_{\text{domain-max}}$. It is, however, a bit less stable than R-GCN$_{\text{domain-max}}$ around the optimal policy for Key-Corridors-Small. The R-GCN$_{\text{CNN+random-max}}$ architecture also performs more or less similar to CNN$_{\text{max}}$, just as it did during training. This is inline with expectation, as it shows that the additional relations of R-GCN$_{\text{CNN+random-max}}$, which are not based on domain knowledge but arbitrary, do not help with generalization to instances with 3 keys.

From the right-hand plots of figures 6.14b and 6.15b, it can be concluded that generalizing from Key-Corridors-Small to Key-Corridors-Big and vice versa, works slightly less well for R-GCN$_{\text{CNN+domain-max}}$ than it does for R-GCN$_{\text{domain-max}}$. This could be because the CNN-like relations of R-GCN$_{\text{CNN+domain-max}}$

do not generalize as well as the 'left', 'right', 'up' and 'down' relations of R-GCN$_{\text{domain-max}}$. However, it may also be an effect of the fact that R-GCN$_{\text{CNN+domain-max}}$ encodes 11 relations, compared to the 7 of R-GCN$_{\text{domain-max}}$ and therefore has fewer features (trainable parameters) per relation.

A final insight can be drawn from the right-hand plot of figure 6.14b. Here we see that for R-GCN$_{\text{CNN+random-max}}$, generalization from Key-Corridors-Small to Key-Corridors-Big actually works slightly better than it did for CNN$_{\text{max}}$, however, this is only marginally and both methods do not get anywhere near to the optimal policy. Furthermore, this improvement is not observed for generalization from Key-Corridors-Big to Key-Corridors-Small, as displayed in the right-hand plot of figure 6.15b.

To conclude the results, table 6.4 provides an overview of the best obtained out-of-distribution generalization performances of the compared architectures. It does so by reporting the lowest mean episode length achieved by each of the architectures during the periodical evaluation of their generalization performance.

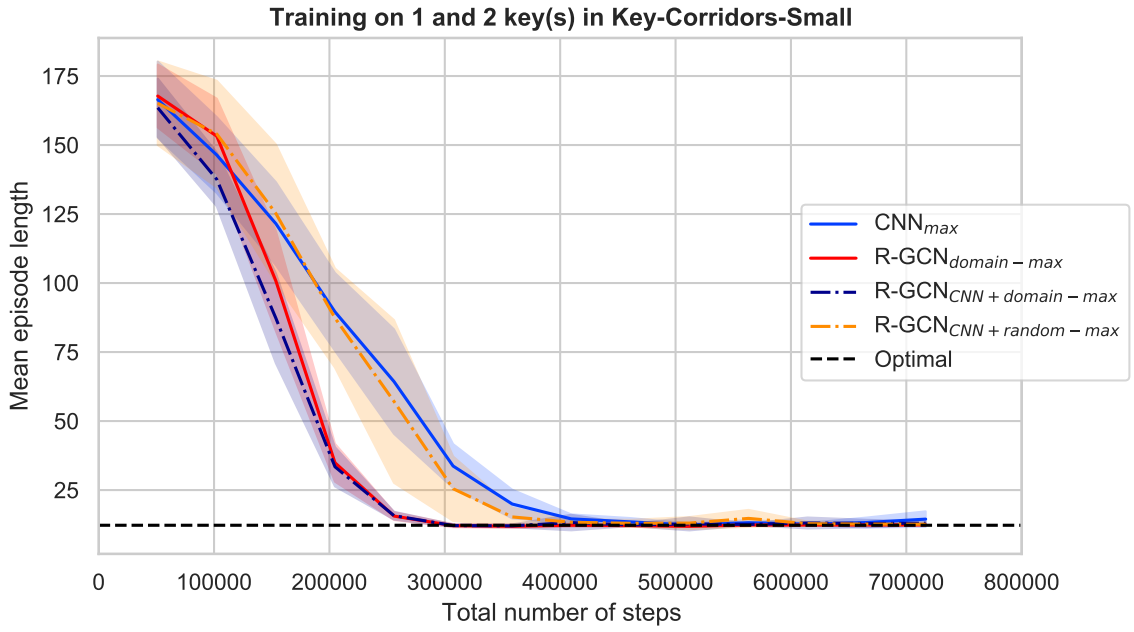
Table 6.4: Overview of the lowest mean episode lengths achieved by each of the architectures during the periodical evaluation of their generalization performance. The reported mean values represent the average over 8 seeded runs, which each evaluated the agent for 30 episodes. The best results are shown in bold and the standard deviation is denoted by SD.

Trained on:	Key-Corridors-Small with 1 or 2 key(s)		Key-Corridors-Big with 1 or 2 key(s)	
Evaluated on:	Key-Corridors-Small with 3 keys	Key-Corridors-Big with 1 or 2 key(s)	Key-Corridors-Big with 3 keys	Key-Corridors-Small with 1 or 2 key(s)
CNN$_{\text{max}}$	53.0 (SD: 12.68)	206.05 (SD: 20.92)	42.93 (SD: 11.85)	75.86 (SD: 9.76)
R-GCN$_{\text{domain-max}}$	**20.78** (SD: 0.17)	**56.10** (SD: 16.37)	**26.78** (SD: 1.61)	**16.33** (SD: 3.0)
R-GCN$_{\text{CNN+domain-max}}$	21.35 (SD: 0.73)	91.60 (SD: 20.41)	27.36 (SD: 1.78)	31.42 (SD: 7.41)
R-GCN$_{\text{CNN+random-max}}$	59.46 (SD: 17.12)	169.45 (SD: 34.95)	51.93 (SD: 16.78)	78.33 (SD: 13.85)

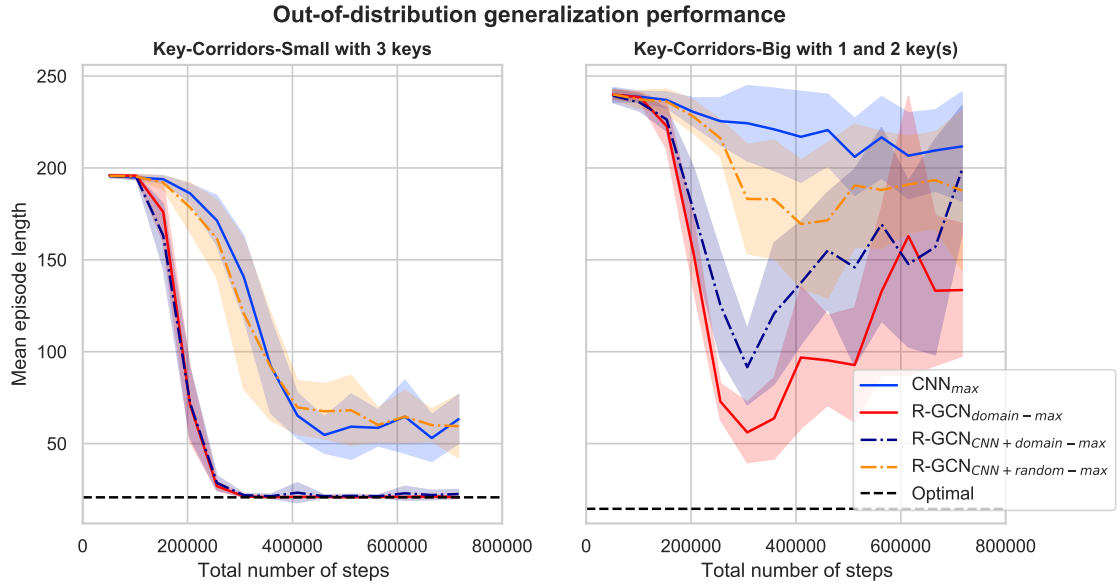
6.4.3. Conclusion

From this experiment, two important conclusions can be drawn. The first finding is that the 'can pick up' and 'can open' relations of R-GCN$_{\text{domain-max}}$ are the main cause for improvement over the CNN$_{\text{max}}$ architecture. Adding these relations to R-GCN$_{\text{CNN-max}}$ - which imitates the CNN$_{\text{max}}$ architecture - yields a performance that is close to that of R-GCN$_{\text{domain-max}}$.

The second conclusion is that we cannot simply add some arbitrary, longer distance relations - which are not based on domain knowledge - to our CNN architecture to increase its performance, as tested with R-GCN$_{\text{CNN+random-max}}$. This architecture has exactly the same number of relations and edges per relation as R-GCN$_{\text{CNN+domain-max}}$ but performs much worse, in fact almost exactly the same as R-GCN$_{\text{CNN-max}}$, which does not have any additional relations.

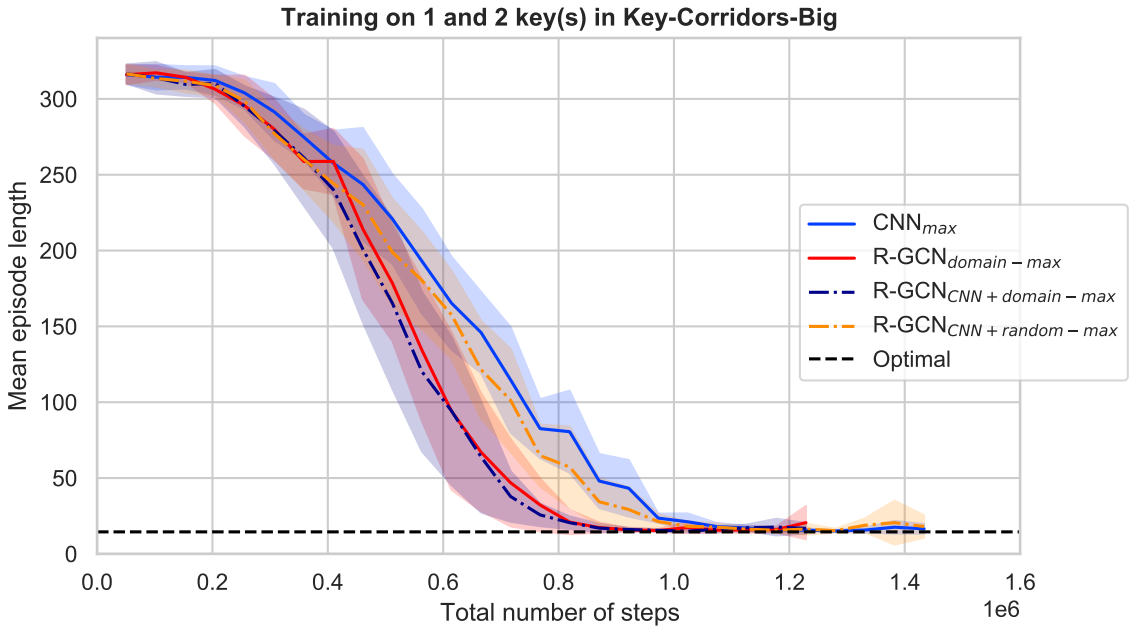


(a)

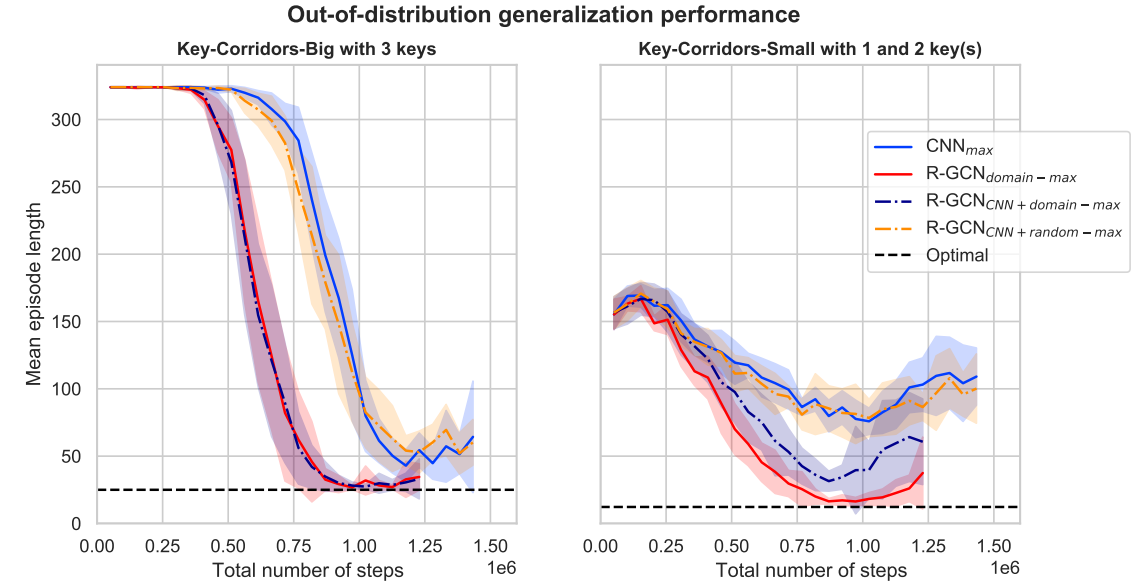


(b)

Figure 6.14: Testing sample efficiency and out-of-distribution generalization of R-GCN$_{\text{CNN+domain-max}}$ and R-GCN$_{\text{CNN+random-max}}$. The results of R-GCN$_{\text{domain-max}}$ and CNN$_{\text{max}}$ are added in the same figure for comparison. Figure 6.14a shows the training curves of training the agents on Key-Corridors-Small instances with either 1 or 2 key(s). The left-hand plot of figure 6.14b displays out-of-distribution generalization performance on Key-Corridors-Small instances with 3 keys, i.e., one key more than in the instances seen during training. The right-hand plot displays out-of-distribution generalization to Key-Corridors-Big. All lines display the average over 8 different seeds and the shaded areas indicate the standard deviations. The striped black line represents the mean episode length that is obtainable by an optimal policy.



(a)



(b)

Figure 6.15: Testing sample efficiency and out-of-distribution generalization of R-GCN$_{\text{CNN+domain-max}}$ and R-GCN$_{\text{CNN+random-max}}$ by training on Key-Corridors-Big instances. The results of R-GCN$_{\text{domain-max}}$ and CNN$_{\text{max}}$ are added in the same figure for comparison. Figure 6.15a shows the training curves of training the agents on Key-Corridors-Big instances with either 1 or 2 key(s). The left-hand plot of figure 6.15b displays out-of-distribution generalization performance on Key-Corridors-Big instances with 3 keys, i.e., one key more than in the instances seen during training. The right-hand plot displays out-of-distribution generalization to Key-Corridors-Small. All lines display the average over 8 different seeds and the shaded areas indicate the standard deviations. The striped black line represents the mean episode length that is obtainable by an optimal policy.

6.5. Experiment 5: Can useful domain-specific relations be learned with R-GCN$_{GAN}$?

In this experiment the proposed R-GCN$_{GAN}$ architecture, introduced in section 5.4.8, is tested against R-GCN$_{domain\text{-}max}$ and CNN$_{max}$. As max-pooling will also be used as readout function for R-GCN$_{GAN}$, we will refer to this architecture as R-GCN$_{GAN\text{-}max}$ in the following sections. The goal is to test whether this architecture is able to learn useful relations with the attention mechanism of the GAN, on top of the relations of a CNN with a 3×3 kernel size, which are in this case encoded by a R-GCN. This would theoretically allow R-GCN$_{GAN\text{-}max}$ to learn similar kind of relations between keys, doors and the agent, as the ones that are encoded in the R-GCN$_{domain\text{-}max}$ architecture. If it succeeds in doing so, we expect to also see improved sample efficiency and out-of-distribution generalization performance compared to CNN$_{max}$, as was the case for R-GCN$_{domain\text{-}max}$ in the previous experiments.

To test whether R-GCN$_{GAN\text{-}max}$ is able to learn useful relations, it is trained on the Key-Corridors-Small environment with 1 and 2 key(s), as well as the they Key-Corridors-Big environment with 1 and 2 key(s). Sample efficiency, in-distribution generalization and out-of-distribution generalization are tested. For out-of-distribution generalization, both generalizing to 3 keys and from Key-Corridors-Small to Key-Corridors-Big, and vice versa, are considered.

6.5.1. Sample Efficiency

In this experiment, the sample efficiency can be evaluated based on both the training curves of training on 70%, and training on 100% of all possible instances of the Key-Corridors-Small and Key-Corridors-Big environments with 1 and 2 key(s). The left-hand side plots of figures 6.16 and 6.17 show the training curves for training on 70% of all possible instances of Key-Corridors-Small and Key-Corridors-Big, respectively. Figures 6.18a and 6.19a show the results of training on 100% of the possible instances for Key-Corridors-Small and Key-Corridors-Big, respectively.

The performance of R-GCN$_{domain\text{-}max}$ and CNN$_{max}$ are added to all of these plots for comparison. On both Key-Corridors-Small and Key-Corridors-Big we have to conclude that R-GCN$_{GAN\text{-}max}$ does not outperform CNN$_{max}$, and performs significantly worse when compared to R-GCN$_{domain\text{-}max}$. This indicates that - at least for sample efficiency - the attention mechanism of R-GCN$_{GAN\text{-}max}$ did not succeed in learning useful relations.

6.5.2. In-distribution generalization

To test for in-distribution generalization performance, the agents trained on 70% of all instances of the Key-Corridors-Small and Key-Corridors-Big environments with 1 and 2 key(s), are evaluated on the remaining 30% of instances. These results are displayed in the right-hand side plots of figures 6.16 and 6.17, for Key-Corridors-Small and Key-Corridors-Big, respectively. Like all other methods in the previous experiments, R-GCN$_{GAN\text{-}max}$ displays perfect in-distribution generalization performance. This is not completely surprising, as this was also the case for CNN$_{max}$ and we do not expect a worse performance by adding extra relations through the attention mechanism of R-GCN$_{GAN\text{-}max}$.

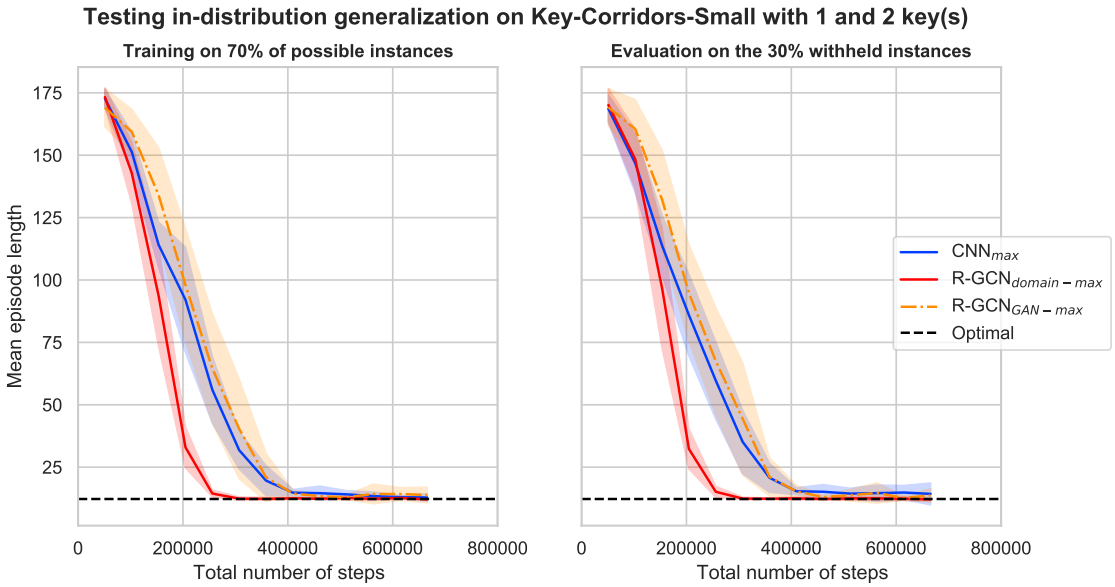


Figure 6.16: Testing in-distribution generalization of R-GCN$_{\text{GAN-max}}$. The left-hand plot displays the training curves, where the agents are trained on 70% of all instances of Key-Corridors-Small with both 1 and 2 key(s). The right-hand plot shows the evaluation of the agents on the 30% of instances that were withheld during training. All lines display the average over 8 different seeds and the shaded areas indicate the standard deviations. The striped black line represents the mean episode length that is obtainable by an optimal policy.

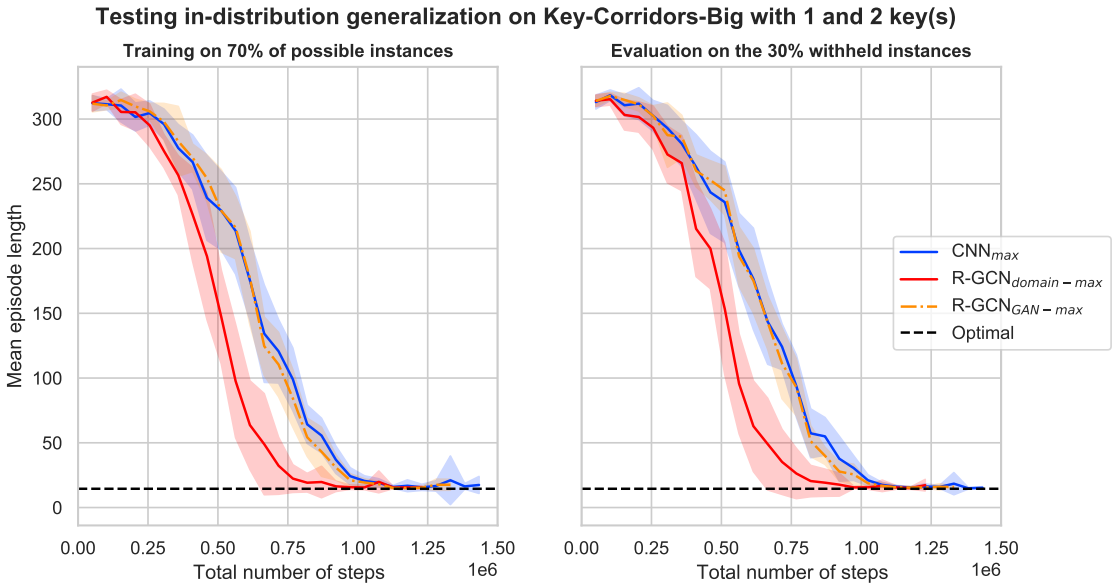


Figure 6.17: Testing in-distribution generalization of R-GCN$_{\text{GAN-max}}$. The left-hand plot displays the training curves, where the agents are trained on 70% of all instances of Key-Corridors-Big with both 1 and 2 key(s). The right-hand plot shows the evaluation of the agents on the 30% of instances that were withheld during training. All lines display the average over 8 different seeds and the shaded areas indicate the standard deviations. The striped black line represents the mean episode length that is obtainable by an optimal policy.

6.5.3. Out-of-distribution generalization

The setup for testing out-of-distribution generalization performance is the same as the one used in experiment 3. Figures 6.18a and 6.19a show the results of training on 100% of the possible instances with 1 and 2 key(s) of the Key-Corridors-Small and Key-Corridors-Big environment, respectively. During training, the agent is periodically evaluated for out-of-distribution generalization to instances with 3 keys. When trained on Key-Corridors-Small instances, the agent is also evaluated for generalization to Key-Corridors-Big instances and vice versa for when the agent is trained on Key-Corridors-Big instances. These results are displayed in figures 6.18b and 6.19b. The results for of R-GCN$_{\text{domain-max}}$ and CNN$_{\text{max}}$ are again added in the same plots for comparison.

From the left plot of figure 6.18b, it can be observed that, on average, R-GCN$_{\text{GAN-max}}$ seems to have slightly better performance than CNN$_{\text{max}}$ on the out-of-distribution instances of Key-Corridors-Small with 3 keys. However, the standard deviations of R-GCN$_{\text{GAN-max}}$ and CNN$_{\text{max}}$ overlap too much to qualify this as a significant difference. Furthermore, when compared to R-GCN$_{\text{domain-max}}$, the performance of R-GCN$_{\text{GAN-max}}$ is much worse. The right-hand side plot of 6.18b, displaying evaluation performance on the out-of-distribution instances of Key-Corridors-Big, show that R-GCN$_{\text{GAN-max}}$ does reach a significantly better performance than CNN$_{\text{max}}$, but still performs much worse than R-GCN$_{\text{domain-max}}$. This does, however, indicate that R-GCN$_{\text{GAN-max}}$ seems to learn some relations with its attention mechanism, which have a positive influence on its generalization capabilities to the larger environment.

In figure 6.19b, which displays the out-of-distribution generalization performance of the agents trained on Key-Corridors-Big instances with either 1 or 2 key(s), the performance of R-GCN$_{\text{GAN-max}}$ is unfortunately even less impressive. For generalization to Key-Corridors-Big instances with 3 keys - displayed in the right-hand side plot - R-GCN$_{\text{GAN-max}}$ performs roughly equal to CNN$_{\text{max}}$. For out-of-distribution generalization to Key-Corridors-Small, R-GCN$_{\text{GAN-max}}$ again seems to outperform CNN$_{\text{max}}$, as was the case with generalization from Key-Corridors-Small to Key-Corridors-Big. However, this time it only performs slightly better on average and there is a lot of overlap between the standard deviations, indicating that for some seeds it might even perform worse.

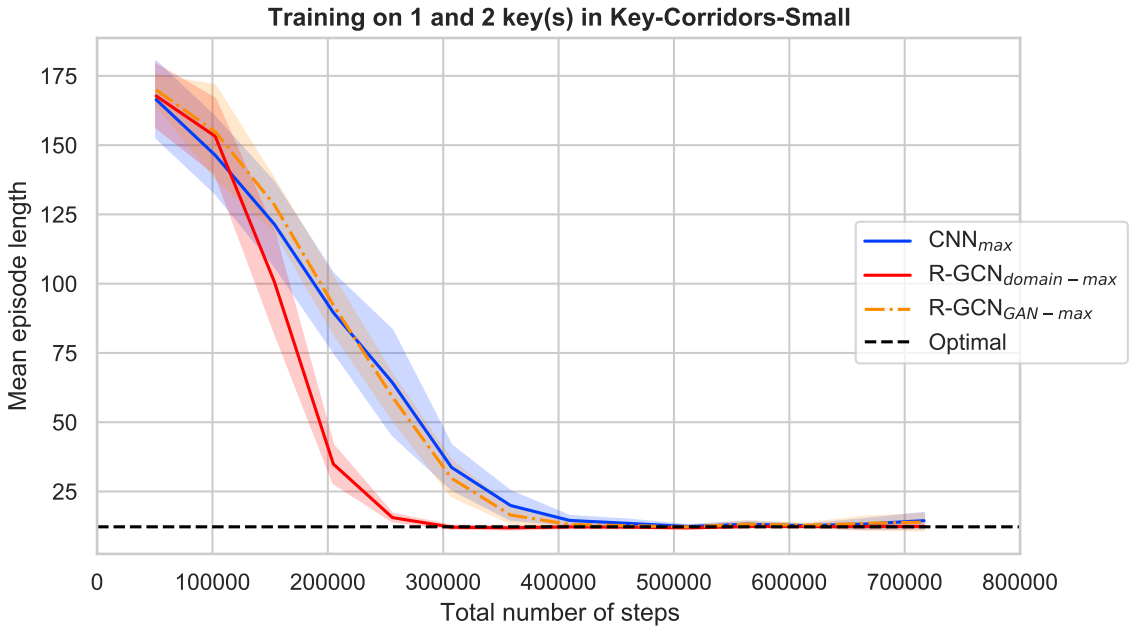
Table 6.5 provides an overview of the lowest mean episode lengths achieved by each of the architectures during the periodical evaluation of their out-of-distribution generalization performance.

Table 6.5: Overview of the lowest mean episode lengths achieved by the R-GCN$_{\text{GAN-max}}$ architecture during the periodical evaluation of its generalization performance. The results for the CNN$_{\text{max}}$ and R-GCN$_{\text{domain-max}}$ architectures are included for comparison. The reported mean values represent the average over 8 seeded runs, which each evaluated the agent for 30 episodes. The best results are shown in bold and the standard deviation is denoted by SD.

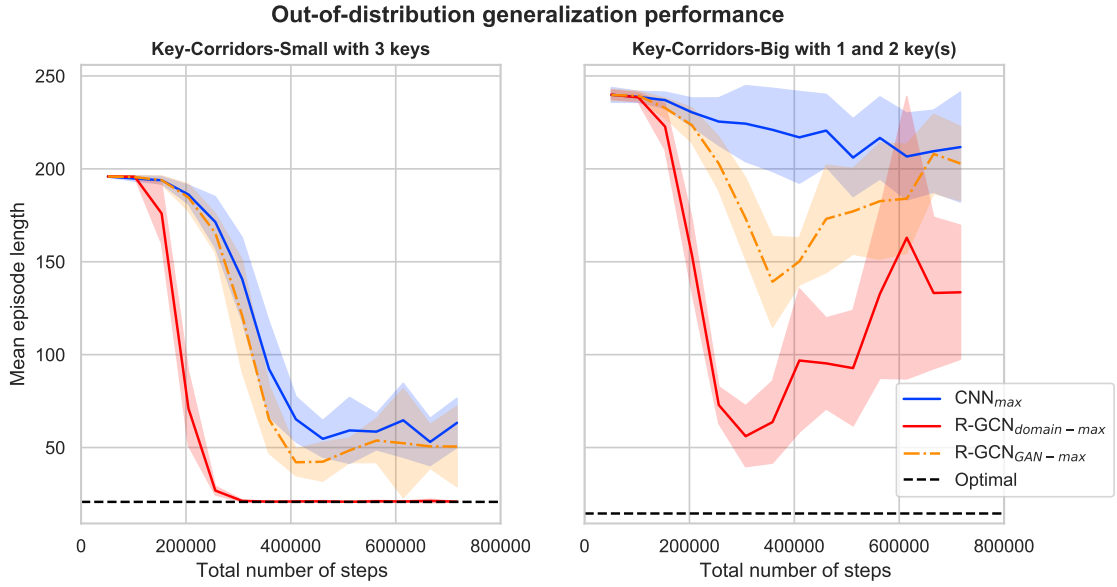
Trained on:	Key-Corridors-Small with 1 or 2 key(s)		Key-Corridors-Big with 1 or 2 key(s)	
Evaluated on:	Key-Corridors-Small with 3 keys	Key-Corridors-Big with 1 or 2 key(s)	Key-Corridors-Big with 3 keys	Key-Corridors-Small with 1 or 2 key(s)
CNN$_{\text{max}}$	53.0 (SD: 12.68)	206.05 (SD: 20.92)	42.93 (SD: 11.85)	75.86 (SD: 9.76)
R-GCN$_{\text{domain-max}}$	**20.78** (SD: 0.17)	**56.10** (SD: 16.37)	**26.78** (SD: 1.61)	**16.33** (SD: 3.0)
R-GCN$_{\text{GAN-max}}$	42.08 (SD: 7.16)	139.23 (SD: 24.32)	42.14 (SD: 8.18)	65.76 (SD: 16.14)

6.5.4. Conclusion

It has to be concluded that the attention mechanism of R-GCN$_{\text{GAN-max}}$ did not succeed in learning useful extra relational inductive biases, when compared to CNN$_{\text{max}}$. In terms of sample efficiency and in-distribution generalization, R-GCN$_{\text{GAN-max}}$ has very similar performance to that of CNN$_{\text{max}}$. For out-of-distribution generalization there are a few cases where R-GCN$_{\text{GAN-max}}$ does seem to perform slightly better than CNN$_{\text{max}}$, however, this is not significant, nor comparable to the performance of the relational inductive biases imposed by R-GCN$_{\text{domain-max}}$.

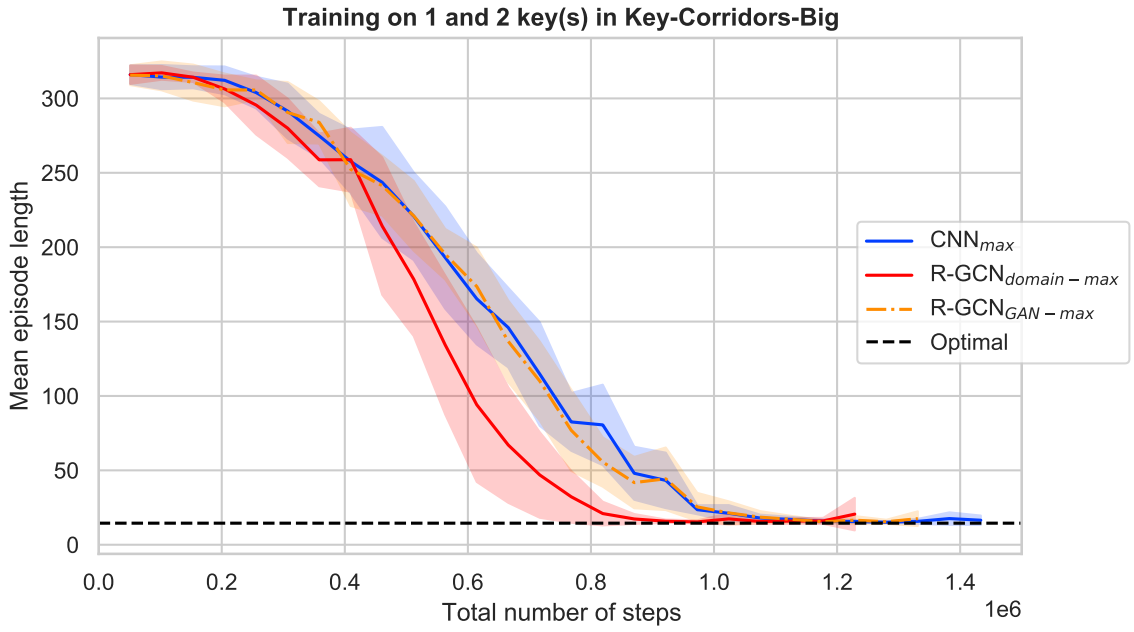


(a)

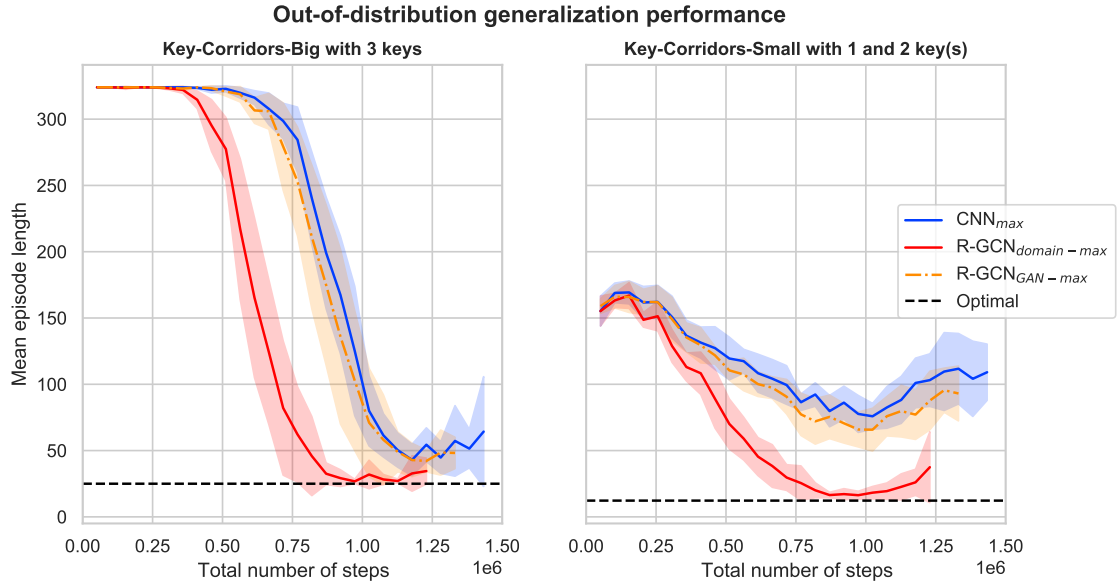


(b)

Figure 6.18: Testing out-of-distribution generalization of R-GCN$_{\text{GAN-max}}$, compared to CNN$_{\text{max}}$ and R-GCN$_{\text{domain-max}}$. The corresponding training curves are displayed in figure 6.18a, for which the agents are trained on Key-Corridors-Small instances with either 1 or 2 key(s). The left-hand plot of figure 6.18b displays out-of-distribution generalization performance on Key-Corridors-Small instances with 3 keys, i.e., one key more than in the instances seen during training. The right-hand plot displays out-of-distribution generalization to Key-Corridors-Big. All lines display the average over 8 different seeds and the shaded areas indicate the standard deviations. The striped black line represents the mean episode length that is obtainable by an optimal policy.



(a)



(b)

Figure 6.19: Testing out-of-distribution generalization of R-GCN$_{GAN\text{-}max}$, compared to CNN$_{max}$ and R-GCN$_{domain\text{-}max}$. The corresponding training curves are displayed in figure 6.19a, for which the agents are trained on Key-Corridors-Big instances with either 1 or 2 key(s). The left-hand plot of figure 6.19b displays out-of-distribution generalization performance on Key-Corridors-Big instances with 3 keys, i.e., one key more than in the instances seen during training. The right-hand plot displays out-of-distribution generalization to Key-Corridors-Small. All lines display the average over 8 different seeds and the shaded areas indicate the standard deviations. The striped black line represents the mean episode length that is obtainable by an optimal policy.

7

Discussion and future work

This chapter provides a more general discussion of the previously reported results. Furthermore, some of the methods and experimental setups applied in this thesis are evaluated, limitations are pointed out and implications for possible future work are discussed.

7.1. Implications of results

The aim of this thesis was to investigate whether the general performance of a RL-agent and its capacity to generalize can be improved by incorporating relational reasoning into DRL through the use of GNNs. To this end, observations of a navigation task were encoded as graphs and GNNs, forming part of the actor and critic network of the agent, were applied to these structured, relational representations of the observations.

More specifically, we first investigated whether the combination of GNNs and domain knowledge - used for the relational graph encodings - can be used to improve sample efficiency and generalization performance, aiming to answer research questions 1 and 2, respectively. The main method used to answer these research questions is the R-GCN$_{\text{domain}}$ architecture. In search of an answer to research question 3, we proposed the R-GCN$_{\text{GAN}}$ architecture to investigate whether the relational inductive biases of a CNN can be combined with learned domain-specific relational inductive biases and GNNs to improve sample efficiency and generalization performance in an end-to-end fashion.

In this section, the main findings obtained and presented in the previous sections are discussed with regards to the research objectives and are placed in a broader perspective. To do so, we will separately look at the results of our two main methods: the R-GCN$_{\text{domain}}$ architecture and the R-GCN$_{\text{GAN}}$ architecture.

7.1.1. Results obtained with R-GCN$_{\text{domain}}$

For experiments 1, 2, 3 and 4 the use of R-GCN$_{\text{domain}}$ as network architecture of the RL-agent improves both sample efficiency and generalization performance, when compared to traditional architectures with MLPs and CNNs. This is in line with the expectations raised by the discussed related work on the combination of GNNs and RL in section 4. In this case, however, it is explored for a novel navigation task requiring relational reasoning and for the specific use of R-GCNs in combination with domain knowledge. These results provide several new insights in the use of R-GCNs and domain knowledge in RL.

First of all, they show that R-GCNs can form a valuable tool for encoding domain knowledge into existing RL algorithms such as PPO, by applying domain-specific relational inductive biases in the network architecture of the agent.

Furthermore, the results indicate that domain-specific relational inductive biases - other than the generic ones of a CNN or MLP - can improve an agent's sample efficiency and generalization performance. This is likely due to the inductive biases that force the architecture to place more emphasis on elements and relationships that we know - from domain knowledge - are important to the underlying problem of the environment. If the underlying problem is the same in the new environments to

which the agent must generalize, the same biases are probably still useful to the agent in these new environments.

Finally, the results of experiment 4 demonstrate that adding the 'can pick up' and 'can open' relations to those of R-GCN$_{\text{CNN}}$ - an imitation of a CNN with a kernel size of 3×3 - results in a comparable performance to that of R-GCN$_{\text{domain}}$. This tells us that, the 'can pick up' and 'can open' relations are the drivers behind the improved performance of R-GCN$_{\text{domain}}$ over the CNN architecture. Perhaps more importantly, experiment 4 also shows that adding random relations - not based on domain knowledge - does not yield this improved performance, suggesting that there is indeed a need to base these relations on domain knowledge.

These results further strengthen our confidence in the use of GNNs to induce relational inductive biases that improve the sample efficiency and generalization of RL-agents. This extents the work of Zambaldi et al. (2018) by applying domain-specific relational inductive biases, based on domain knowledge, with a R-GCN. In addition, our results also confirm the findings reported in the recently published work of Z. Jiang et al. (2021) and provide further insights into the use of different readout functions and which type of relations - including random ones - result in increased performance over baseline architectures.

The improved performance of R-GCN$_{\text{domain}}$ over the baseline architectures is significant, however, it is based on the assumed presence of domain knowledge and the possibility to split-up observations into separate entities. Even though these are both strong assumptions, there are multiple scenarios where this could realistically be applied. First of all, the concept behind the R-GCN$_{\text{domain}}$ architecture can be applied to abstract problem domains, where observations are easily encoded as graphs and domain knowledge is at hand. Examples could be RL problems on power- or traffic-networks, social networks or financial transactions, in which, clearly defined entities and relations are often inherently present in the data used to form the agent's observations. Another option would be to create an end-to-end setup for visual tasks, where some form of object detection is used to first define entities in the visual input. The detected objects can then define the entities in the graph encoding of the observation and these entities could be combined with a database defining relations between objects. Next to the relations from the database, which might be sparsely present, more generic relations - like those of a CNN - could still be added as well, comparable to the R-GCN$_{\text{CNN+domain}}$ architecture of this thesis. Alternatively, a CNN and a R-GCN could be trained in parallel, where the CNN operates on the raw input data and the R-GCN on the extracted objects, combined with relations. This last option has some resemblance to the proposed architecture by Lu et al. (2021), who apply the combination of GNNs and RL on a real-world navigation task.

7.1.2. Results obtained with R-GCN$_{\text{GAN}}$

The R-GCN$_{\text{GAN}}$ architecture was explored as an alternative to R-GCN$_{\text{domain}}$ that does not require domain knowledge. However, the R-GCN$_{\text{GAN}}$ architecture performed significantly worse than R-GCN$_{\text{domain}}$, both in terms of sample efficiency and generalization, and only slightly outperformed CNN in some cases of out-of-distribution generalization. Multiple variations were tested, where we varied the number of attention heads, the negative slope angle α_{LRLU} of the LeakyReLU and also explored the option of concatenating the outputs of the attention heads instead of summing them together. However, none of this improved the performance of R-GCN$_{\text{GAN}}$.

One of the problems with the R-GCN$_{\text{GAN}}$ architecture might be that a GAN, from the perspective of a single node in a fully connected graph, computes attention coefficients for all other nodes and then takes a softmax over these coefficients. The GAN architecture therefore inherently assumes that each relation is present for all of the nodes in the graph. In the R-GCN$_{\text{domain}}$ architecture, however, some of the the relations - 'can pick up' and 'can open' - are actually sparse and are only present between a few nodes. In an attempt to solve this problem, self-loops were added for all nodes, enabling each head of the GAN to let nodes attend to themselves as well. However, this still does not allow for learning sparse relations that are, for instance, only present between a key and a door. The network would have to apply the same weights to self-loop edges, as it applies to the edges that do actually encode a relation between keys and doors.

Another issue could be the fact that the CNN architecture - duplicated in the R-GCN$_{\text{GAN}}$ architecture by the R-GCN - already enables the agent to solve the environments that it is being trained on, as was concluded from experiment 1 and 3. This could result in R-GCN$_{\text{GAN}}$ learning to simply ignore the additional relations learned by the GAN, by making their weights very small.

A final cause for the unsatisfactory results of the R-GCN$_{GAN}$ architecture could be a failure of the first R-GCN layer to encode sufficient positional information into the node features. This would make it difficult for the GAN - which is operating on a fully connected graph - to distinguish between distinct observations where the same nodes occupy different positions in the grid world.

Additions or adjustments to this architecture remain an interesting topic to explore further, as Zambaldi et al. (2018) have shown that GANs alone are capable of learning useful relations for another, comparable RL problem. As mentioned earlier, Zambaldi et al. (2018) do include positional information in the node features, which remains a difference to our setup. Our results might also indicate that the performance of GAN, on these type of tasks, is highly dependent on the specific setup and parameter turning, however, making any statements about this would require further research.

7.2. Limitations

In this sections, some limitations of both the used methods and the experimental setup of this thesis are pointed out and discussed.

7.2.1. Strong assumptions for R-GCN$_{domain}$

One of the biggest limitations of the R-GCN$_{domain}$ architecture - the architecture with the best overall performance - remains its use of domain knowledge and the assumption that observations can be defined as separate entities with relations between them. As this limitation has already been mentioned and explained in previous sections, it will not be discussed in detail here, however, it is important to mention again that these assumptions limit the applicability of this method in its current form.

7.2.2. Setup for testing in-distribution generalization

For all experiments where in-distribution generalization was tested - experiments 1, 2, 3 and 5 - it was hard to detect differences in the performance of the various architectures. All architectures seemed to perform equally well on the withheld instances, as they did on the actual training instances. We therefore have to assume that the setup used for testing in-distribution was not ideal. The initial decision to train and test on many different initialization of the same Key-Corridors environment was made to ensure that our methods were tested on a variety of initializations and were not merely working - or failing - for certain specific initializations used during training or evaluation. However, training on so many different instances works as a form of domain randomization, which is known to improve generalization, as for instance discussed by Tobin et al. (2017). This domain randomization might have caused the good in-distribution generalization performance of all architectures.

7.2.3. Used environments

The Key-Corridors environments used to test the architectures presented in this thesis only provide a limited validation of our methods. The environments serve well as an initial test case, however, more varied environments should be explored to further validate our methods. More specifically, a set of environments designed to test and benchmark generalization would be ideal. Nichol et al. (2018), for instance provide such an environment, with a proper split between 'train' and 'test' environments. However, these environments have to be suitable to use with the strong assumptions of R-GCN$_{domain}$ or a more end-to-end version of R-GCN$_{domain}$ will have to be developed and applied.

7.2.4. Hyperparameter tuning

Due to computational constraints, it was not possible to perform an extensive grid-search or some form of automated tuning for the hyperparameters of PPO and the architecture-specific parameters. However, the choice of hyperparameters can be crucial for a fair comparison between the different architectures, as also argued by Henderson et al. (2018). They also found that many studies in this field do not report the considered hyperparameters, which is why we decided to include these values in this thesis. Although all hyperparameters were chosen to benefit the baseline architectures, it remains difficult to make formal guarantees about the optimality of our hyperparameters for the performance of the baseline architectures.

7.3. Future work

This thesis has been able to provide a 'proof of concept' of applying R-GCNs in combination with domain knowledge to create a form of relational reasoning in DRL and thereby improve sample efficiency and generalization performance. Based on this result, there are many possible extensions to this research that are worth exploring. Some of these will be discussed in this section.

7.3.1. Test on more environments

Our proposed R-GCN$_{\text{domain}}$ architecture produced promising results on the Key-Corridors environments, however, the concepts underlying this architecture should be tested on more environments. This would act both as a broader validation of these methods, and as a test of their applicability to different domains. In particular, environments that require complex relational reasoning would be good candidates.

7.3.2. Making an end-to-end solution

This research can be further extended by developing an end-to-end version of the R-GCN$_{\text{domain}}$ architecture. One way to achieve this, would be through the previously discussed method of extracting entities - for instance with a network trained for object detection - from raw input data and then combining these entities with relations, based on domain knowledge stored in a digital format. This possible approach was already introduced in section 7.1.1, and will therefore not be discussed in further detail here. However, it remains an important topic to address in future work. Another approach to an end-to-end solution is to actually learn relations, instead of using domain knowledge for this, which was explored in this thesis with the R-GCN$_{\text{GAN}}$ architecture. Benefits of these kind of methods are that they do not require domain knowledge and do not necessarily require the observations to be split up into entities between which relations can be defined with domain knowledge. Instead, they would hypothetically also work on more abstract entities, such as pixels or groups of pixels, for which there is no need to first apply a complex method to extract these entities. However, the current results of the R-GCN$_{\text{GAN}}$ architecture were unsatisfactory, but do leave open options for future research. One possible direction of future work would be the application of solutions that allow for more sparse relations in the GAN network, as explored by Ye and Ji (2021). Furthermore, one could investigate methods that ensure that the overall R-GCN$_{\text{GAN}}$ architecture, which also employs the same relations as a CNN with a kernel size of 3×3, does not ignore the additional relations learned by the GAN.

8

Conclusions

This chapter will first reiterate and summarize the main findings of this thesis per research question, followed by a more general conclusion.

Research question 1: *Can the sample efficiency of DRL be improved by using GNNs and domain knowledge, compared to traditional architectures such as CNNs and MLPs?*

Based on experiments 1, 2, 3 and 4 it can be concluded that - for the Key-Corridors environments - it is indeed possible to improve sample efficiency with the combination of domain knowledge and GNNs, compared to applying more traditional architectures such as CNNs and MLPs. The combination of GNNs and domain knowledge allows us to form domain-specific relational inductive biases in the neural network architectures of the actor and critic network of the agent. Using a simple GCN to do so, does not yield good performance, as it can only encode one type of relation between nodes. This does not allow for a rich relational encoding based on the domain knowledge. Moreover, for our specific use on the grid-shaped graphs resulting from the observations of the Key-Corridors environments, using only one relation also makes it impossible for the agent to distinguish between many of the symmetric observations. In contrast, a R-GCN is capable of encoding multiple relations, which does allow for a rich relational encoding of the observations, based on the assumed domain knowledge. Adding these domain-specific relational inductive biases to the agents architecture - as is done in the $\text{R-GCN}_{\text{domain}}$ architecture - significantly increases the sample efficiency of the agent when compared to traditional MLP and CNN architectures, both with and without readout function. Of the 4 considered readout functions (GAP, max-pooling, average-pooling, and sum-pooling), max-pooling gave the best performance. The sample efficiency of agents with the $\text{R-GCN}_{\text{domain}}$ architecture, combined with max-pooling as readout function, was explored with the $\text{R-GCN}_{\text{domain-max}}$ architecture on multiple instances of both the Key-Corridors-Small and Key-Corridors-Big environment. For both the Key-Corridors-Small and Key-Corridors-Big environments, $\text{R-GCN}_{\text{domain-max}}$ has a comparable sample efficiency to CNN when it is solely trained on instances with only 1 key. However, when more difficult instances, that have more than 1 key, are added to the training instances, $\text{R-GCN}_{\text{domain-max}}$ has a significantly better sample efficiency than CNN on both the Key-Corridors-Small and Key-Corridors-Big environment.

Research question 2: *Can the generalization performance of DRL be improved by using GNNs and domain knowledge, compared to traditional architectures such as CNNs and MLPs?*

To answer this research question both in- and out-of-distribution generalization were considered. In order to test in-distribution generalization, agents were trained on a set of instances - each instance being a different initialization of the same environment with the same number of keys - of which 30% of all possible instances were withheld. During training, the agents were periodically evaluated on the set of withheld instances. The results of experiment 1, 2 and 3 show that all of the architectures have a good in-distribution generalization performance, making it hard to determine whether one of the architectures has a better in-distribution generalization performance than the others, as discussed in section 7.2.2 of the discussion.

Adding domain-specific relational inductive biases based on domain knowledge - as is done in the R-GCN$_{domain}$ architecture - significantly increases out-of-distribution generalization performance, when compared to CNN. For out-of-distribution generalization, max-pooling was again the best performing readout function. On both the Key-Corridors-Small and Key-Corridors-Big environments, R-GCN$_{domain\text{-}max}$ outperforms CNN, and it does so for generalizing to instances with an extra key, as well as generalizing from Key-Corridors-Small to Key-Corridors-Big and vice versa.

Research question 3: *Can the fixed relational inductive biases of a CNN be combined with learned domain-specific relational inductive biases and GNNs to improve sample efficiency and generalization performance in an end-to-end fashion, which does not require domain knowledge?*

In an attempt to answer this research question, the R-GCN$_{GAN\text{-}max}$ architecture was introduced, which again applies max-pooing as readout function. However, this architecture did not succeed in improving sample efficiency over CNN. The R-GCN$_{GAN\text{-}max}$ architecture does in some cases have a slightly better out-of-distribution generalization performance than CNN, but this is not significant and R-GCN$_{domain\text{-}max}$ - with its relations based on domain knowledge - still significantly outperforms this method.

Overall, we conclude that adding relational reasoning - in the form of relational inductive biases imposed by a R-GCN and based on domain knowledge - improves the performance of RL-agents, both in terms of sample efficiency and out-of-distribution generalization.

This approach shows several clear advantages over traditional techniques but will have to be further validated on a wider range of environments. Another remaining challenge for future work is to apply these methods in an end-to-end manner which relies less heavily on an abstract encoding of the environment and preferably does not require domain knowledge. However, for problems where domain knowledge is available and observations can easily be encoded as graphs based on this domain knowledge, the concepts of the R-GCN$_{domain}$ architecture are worth applying, especially when sample efficiency and generalization performance are important.

The results of this thesis will be valuable for further advancing the use of GNNs in RL and we hope that it can thereby act as a small step in the direction of human-like relational reasoning in RL, which is known to be a core aspect of our incredible generalization abilities.

Bibliography

Achiam, J. (2018). Spinning Up in Deep Reinforcement Learning. https://github.com/openai/spinningup

Almasan, P., Suárez-Varela, J., Badia-Sampera, A., Rusek, K., Barlet-Ros, P., & Cabellos-Aparicio, A. (2019). Deep Reinforcement Learning meets Graph Neural Networks: exploring a routing optimization use case. http://arxiv.org/abs/1910.07421

Anderson, J. R. (1982). Acquisition of cognitive skill. *Psychological Review*, *89*(4), 369–406. https://doi.org/10.1037/0033-295X.89.4.369

Andrychowicz, M., Raichuk, A., Stańczyk, P., Orsini, M., Girgin, S., Marinier, R., Hussenot, L., Geist, M., Pietquin, O., Michalski, M., Gelly, S., & Bachem, O. (2020). What Matters In On-Policy Reinforcement Learning? A Large-Scale Empirical Study. http://arxiv.org/abs/2006.05990

Arel, I., Liu, C., Urbanik, T., & Kohls, A. (2010). Reinforcement learning-based multi-agent system for network traffic signal control. *Iet Intelligent Transport Systems*, *4*, 128–135.

Arulkumaran, K., Deisenroth, M. P., Brundage, M., & Bharath, A. A. (2017). Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, *34*(6), 26–38. https://doi.org/10.1109/MSP.2017.2743240

Bahdanau, D., Cho, K., & Bengio, Y. (2014). Neural Machine Translation by Jointly Learning to Align and Translate. https://arxiv.org/abs/1409.0473

Battaglia, P. W., Hamrick, J. B., Bapst, V., Sanchez-Gonzalez, A., Zambaldi, V., Malinowski, M., Tacchetti, A., Raposo, D., Santoro, A., Faulkner, R., Gulcehre, C., Song, F., Ballard, A., Gilmer, J., Dahl, G., Vaswani, A., Allen, K., Nash, C., Langston, V., … Pascanu, R. (2018). Relational inductive biases, deep learning, and graph networks. https://arxiv.org/abs/1806.01261

Bellman, R. (1957). A Markovian Decision Process. *Indiana University Mathematics Journal*, *6*(4), 679–684. https://doi.org/10.1512/iumj.1957.6.56038

Bergstra, J., Bardenet, R., Bengio, Y., & Kégl, B. (2011). Algorithms for hyper-parameter optimization. *Advances in Neural Information Processing Systems*, *24*.

Bondy, J. A., & Murty, U. S. R. (1976). *Graph Theory with Applications*. The Macmillan Press Ltd.

Boureau, Y. L., Ponce, J., & Lecun, Y. (2010). A theoretical analysis of feature pooling in visual recognition. *ICML 2010 - Proceedings, 27th International Conference on Machine Learning*, (November), 111–118.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., & Zaremba, W. (2016). OpenAI Gym. http://arxiv.org/abs/1606.01540

Chauvin, Y., & Rumelhart, D. E. (2013). *Backpropagation: theory, architectures, and applications*. Psychology press.

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

Cobbe, K., Klimov, O., Hesse, C., Kim, T., & Schulman, J. (2019). Quantifying generalization in reinforcement learning. *International Conference on Machine Learning (PMLR)*, 1282–1289.

D'Eramo, C., Tateo, D., Bonarini, A., Restelli, M., Milano, P., & Peters, J. (2020). Sharing Knowledge in Multi-Task Deep Reinforcement Learning. *International Conference on Learning Representation (ICLR)*, 1–18. https://openreview.net/forum?id=rkgpv2VFvr

Dhariwal, P., Hesse, C., Klimov, O., Nichol, A., Plappert, M., Radford, A., Schulman, J., Sidor, S., Wu, Y., & Zhokhov, P. (2017). OpenAI Baselines. https://github.com/openai/baselines

Duan, Y., Chen, X., Houthooft, R., Schulman, J., & Abbeel, P. (2016). Benchmarking deep reinforcement learning for continuous control. *33rd International Conference on Machine Learning, ICML 2016*, *3*, 2001–2014.

Džeroski, S., De Raedt, L., & Driessens, K. (2001a). Relational reinforcement learning. *Machine Learning*, *43*(1-2), 7–52. https://doi.org/10.1023/A:1007694015589

Džeroski, S., De Raedt, L., & Driessens, K. (2001b). Relational reinforcement learning: An Overview. *Machine Learning*, *43*(1-2), 7–52. https://doi.org/10.1023/A:1007694015589

Farebrother, J., Machado, M. C., & Bowling, M. (2018). Generalization and Regularization in DQN. https://arxiv.org/abs/1810.00123

Feurer, M., & Hutter, F. (2019). Hyperparameter Optimization. In F. Hutter, L. Kotthoff, & J. Vanschoren (Eds.), *Automated machine learning: Methods, systems, challenges* (pp. 3–33). Springer International Publishing. https://doi.org/10.1007/978-3-030-05318-5

Finn, C., Abbeel, P., & Levine, S. (2017). Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks. In D. Precup & Y. W. Teh (Eds.), *Proceedings of the 34th international conference on machine learning* (pp. 1126–1135). PMLR. https://proceedings.mlr.press/v70/finn17a.html

François-lavet, V., Henderson, P., Islam, R., Bellemare, M. G., François-lavet, V., Pineau, J., & Bellemare, M. G. (2018). *An Introduction to Deep Reinforcement Learning.* (Vol. 2). https://doi.org/10.1561/2200000071

Gaudet, B., Linares, R., & Furfaro, R. (2020). Deep reinforcement learning for six degree-of-freedom planetary landing. *Advances in Space Research*, *65*(7), 1723–1741. https://doi.org/https://doi.org/10.1016/j.asr.2019.12.030

Getoor, L., & Taskar, B. (2007). *Introduction to statistical relational learning (Ser. Adaptive computation and machine learning)*. MIT Press. https://doi.org/10.7551/mitpress/7432.001.0001

Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O., & Dahl, G. E. (2017). Neural message passing for quantum chemistry. *34th International Conference on Machine Learning, ICML 2017*, *3*, 2053–2070.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press. https://doi.org/10.1007/s10710-017-9314-z

Gori, M., Monfardini, G., & Scarselli, F. (2005). A new model for learning in graph domains. *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, *2*, 729–734. https://doi.org/10.1109/IJCNN.2005.1555942

Gu, J., Wang, Z., Kuen, J., Ma, L., Shahroudy, A., Shuai, B., Liu, T., Wang, X., Wang, G., Cai, J., & Chen, T. (2018). Recent advances in convolutional neural networks. *Pattern Recognition*, *77*, 354–377. https://doi.org/10.1016/j.patcog.2017.10.013

Hamilton, W. L., Ying, R., & Leskovec, J. (2017). Inductive representation learning on large graphs. *Proceedings of the 31st International Conference on Neural Information Processing Systems*, 1025–1035.

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

He, K., Zhang, X., Ren, S., & Sun, J. (2015). Deep Residual Learning for Image Recognition. http://arxiv.org/abs/1512.03385

Henderson, P., Islam, R., Bachman, P., Pineau, J., Precup, D., & Meger, D. (2018). Deep reinforcement learning that matters. *32nd AAAI Conference on Artificial Intelligence, AAAI 2018*, 3207–3214.

Hertel, L., Baldi, P., & Gillen, D. L. (2020). Quantity vs. Quality: On Hyperparameter Optimization for Deep Reinforcement Learning. http://arxiv.org/abs/2007.14604

Hessel, M., Soyer, H., Espeholt, L., Czarnecki, W., Schmitt, S., & van Hasselt, H. (2019). Multi-Task Deep Reinforcement Learning with PopArt. *Proceedings of the AAAI Conference on Artificial Intelligence*, *33*(01 SE - AAAI Technical Track: Machine Learning), 3796–3803. https://doi.org/10.1609/aaai.v33i01.33013796

Hochreiter, S., & Schmidhuber, J. (1997). Long Short-Term Memory. *Neural Computation*, *9*(8), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

Holyoak, K. J., & Lu, H. (2021). Emergence of relational reasoning. *Current Opinion in Behavioral Sciences*, *37*, 118. https://doi.org/https://doi.org/10.1016/j.cobeha.2020.11.012

Huang, W., Mordatch, I., & Pathak, D. (2020). One Policy to Control Them All: Shared Modular Policies for Agent-Agnostic Control. *arXiv*. https://arxiv.org/abs/2007.04976

Hüllermeier, E., Fober, T., & Mernberger, M. (2013). Inductive Bias. In W. Dubitzky, O. Wolkenhauer, K.-H. Cho, & H. Yokota (Eds.), *Encyclopedia of systems biology* (pp. 1018–1018). Springer New York. https://doi.org/10.1007/978-1-4419-9863-7

Jiang, J., Dun, C., Huang, T., & Lu, Z. (2020). Graph Convolutional Reinforcement Learning. *arXiv*. https://arxiv.org/abs/1810.09202

Jiang, Z., Minervini, P., Jiang, M., & Rocktaschel, T. (2021). Grid-to-Graph: Flexible Spatial Relational Inductive Biases for Reinforcement Learning. (Aamas). http://arxiv.org/abs/2102.04220

Karpathy, A., Toderici, G., Shetty, S., Leung, T., Sukthankar, R., & Fei-Fei, L. (2014). Large-scale Video Classification with Convolutional Neural Networks. *Proceedings of the IEEE conference on*

Computer Vision and Pattern Recognition, 1725–1732. http://cs.stanford.edu/people/karpathy/deepvideo

Kingma, D. P., & Ba, J. L. (2015). Adam: A method for stochastic optimization. *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*. https://arxiv.org/abs/1412.6980

Kipf, T. N., & Welling, M. (2017). Semi-supervised classification with graph convolutional networks. *5th International Conference on Learning Representations, ICLR 2017 - Conference Track Proceedings*. https://arxiv.org/abs/1609.02907

Kirsch, L., van Steenkiste, S., & Schmidhuber, J. (2019). Improving Generalization in Meta Reinforcement Learning using Learned Objectives. *Published as a conference paper at ICLR 2020*. http://arxiv.org/abs/1910.04098

Kober, J., Bagnell, J. A., & Peters, J. (2013). Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, *32*(11), 1238–1274. https://doi.org/10.1177/0278364913495721

Koller, D., & Friedman, N. (2009). *Probabilistic graphical models: principles and techniques (Ser. Adaptive computation and machine learning)*. MIT Press.

Kurin, V., Igl, M., Rocktäschel, T., Boehmer, W., & Whiteson, S. (2020). My Body is a Cage: the Role of Morphology in Graph-Based Incompatible Control. http://arxiv.org/abs/2010.01856

Lawrence, S., & Giles, C. L. (2000). Overfitting and neural networks: Conjugate gradient and backpropagation. *Proceedings of the International Joint Conference on Neural Networks*, *1*, 114–119. https://doi.org/10.1109/ijcnn.2000.857823

Lawrence, S., Giles, C. L., & Tsoi, A. C. (1997). Lessons in neural network training: overfitting may be harder than expected. *Proceedings of the National Conference on Artificial Intelligence*, 540–545.

Lecun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature*, *521*(7553), 436–444. https://doi.org/10.1038/nature14539

LeCun, Y., Boser, B., Denker, J., Henderson, D., Howard, R., Hubbard, W., & Jackel, L. (1989). Handwritten digit recognition with a back-propagation network. *Advances in neural information processing systems*, *2*.

Li, Y., Zemel, R., Brockschmidt, M., & Tarlow, D. (2016). Gated graph sequence neural networks. *4th International Conference on Learning Representations, ICLR 2016 - Conference Track Proceedings*, (1). https://arxiv.org/abs/1511.05493

Lu, Y., Chen, Y., Zhao, D., & Li, D. (2021). MGRL: Graph neural network based inference in a Markov network with reinforcement learning for visual navigation. *Neurocomputing*, *421*, 140–150. https://doi.org/10.1016/j.neucom.2020.07.091

Maas, A. L., Hannun, A. Y., & Ng, A. Y. (2013). Rectifier nonlinearities improve neural network acoustic models. *ICML Workshop on Deep Learning for Audio, Speech and Language Processing*.

Mahmud, M., Kaiser, M. S., Hussain, A., & Vassanelli, S. (2018). Applications of Deep Learning and Reinforcement Learning to Biological Data. *IEEE Transactions on Neural Networks and Learning Systems*, *29*(6), 2063–2079. https://doi.org/10.1109/TNNLS.2018.2790388

Minsky, M. (1961). Steps Toward Artificial Intelligence. *Proceedings of the IRE*, *49*(1), 8–30. https://doi.org/10.1109/JRPROC.1961.287775

Mitchell, T. M. (1980). The Need for Biases in Learning Generalizations. *Readings in Machine Learning*, (CBM-TR-117), 184–191. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.19.5466

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., & Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, *518*(7540), 529–533. https://doi.org/10.1038/nature14236

Muggleton, S., & de Raedt, L. (1994). Inductive Logic Programming: Theory and methods. *The Journal of Logic Programming*, *19-20*, 629–679. https://doi.org/10.1016/0743-1066(94)90035-3

Nachum, O., Norouzi, M., Xu, K., & Schuurmans, D. (2017). Bridging the gap between value and policy based reinforcement learning. *Advances in Neural Information Processing Systems*, *December*(Nips), 2776–2786. https://arxiv.org/abs/1702.08892

Nichol, A., Pfau, V., Hesse, C., Klimov, O., & Schulman, J. (2018). Gotta Learn Fast: A New Benchmark for Generalization in RL, 1–21. http://arxiv.org/abs/1804.03720

Packer, C., Gao, K., Kos, J., Krähenbühl, P., Koltun, V., & Song, D. (2018). Assessing generalization in deep reinforcement learning. *arXiv*. https://arxiv.org/abs/1810.12282

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

Raffin, A. (2020). RL Baselines3 Zoo. https://github.com/DLR-RM/rl-baselines3-zoo

Raffin, A., Hill, A., Ernestus, M., Gleave, A., Kanervisto, A., & Dormann, N. (2019). Stable Baselines 3. https://github.com/DLR-RM/stable-baselines3

Rajeswaran, A., Ghotra, S., Ravindran, B., & Levine, S. (2016). EPOpt: Learning Robust Neural Network Policies Using Model Ensembles. *Published as a conference paper at ICLR 2017*. https://arxiv.org/abs/1610.01283

Saxe, A. M., McClelland, J. L., & Ganguli, S. (2014). Exact solutions to the nonlinear dynamics of learning in deep linear neural networks. *2nd International Conference on Learning Representations, ICLR 2014 - Conference Track Proceedings*. https://arxiv.org/abs/1312.6120

Schlichtkrull, M., Kipf, T. N., Bloem, P., van den Berg, R., Titov, I., & Welling, M. (2018). Modeling Relational Data with Graph Convolutional Networks. *Lecture Notes in Computer Science*, *10843*, 593–607. https://doi.org/10.1007/978-3-319-93417-4_38

Schulman, J. (2016a). Nuts and Bolts of Deep RL Research. http://joschu.net/docs/nuts-and-bolts.pdf

Schulman, J. (2016b). Optimizing Expectations: From Deep Reinforcement Learning to Stochastic Computation Graphs, Dept. of Computer Science University of California, Berkeley, Ph.D. thesis. *Ph.D thesis*.

Schulman, J., Levine, S., Moritz, P., Jordan, M., & Abbeel, P. (2015). Trust region policy optimization. *32nd International Conference on Machine Learning, ICML 2015*, *3*, 1889–1897.

Schulman, J., Moritz, P., Levine, S., Jordan, M. I., & Abbeel, P. (2016). High-dimensional continuous control using generalized advantage estimation. *4th International Conference on Learning Representations, ICLR 2016 - Conference Track Proceedings*, 1–14.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal policy optimization algorithms. *arXiv*. https://arxiv.org/abs/1707.06347

Shannon, C. E. (1948). A mathematical theory of communication. *The Bell System Technical Journal*, *27*(3), 379–423. https://doi.org/10.1002/j.1538-7305.1948.tb01338.x

Shu, T., Xiong, C., & Socher, R. (2017). Hierarchical and Interpretable Skill Acquisition in Multi-task Reinforcement Learning. *Published as a conference paper at ICLR 2018*. http://arxiv.org/abs/1712.07294

Simonyan, K., & Zisserman, A. (2014). Very Deep Convolutional Networks for Large-Scale Image Recognition. http://arxiv.org/abs/1409.1556

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, *15*, 1929–1958.

S.Sutton, R., & G.Barto, A. (2018). *Reinforcement Learning An Introduction* (Second). MIT Press.

Tesauro, G. (1995). Temporal Difference Learning and TD-Gammon. *Communications of the ACM*, *38*(3), 58–68. https://doi.org/10.1145/203330.203343

Tobin, J., Fong, R., Ray, A., Schneider, J., Zaremba, W., & Abbeel, P. (2017). Domain randomization for transferring deep neural networks from simulation to the real world. *IEEE International Conference on Intelligent Robots and Systems*, *2017-Septe*, 23–30. https://doi.org/10.1109/IROS.2017.8202133

Todorov, E., Erez, T., & Tassa, Y. (2012). MuJoCo: A physics engine for model-based control. *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 5026–5033. https://doi.org/10.1109/IROS.2012.6386109

Tucker, G., Bhupatiraju, S., Gu, S., Turner, R., Ghahramani, Z., & Levine, S. (2018). The Mirage of Action-Dependent Baselines in Reinforcement Learning. In J. Dy & A. Krause (Eds.), *Proceedings of the 35th international conference on machine learning* (pp. 5015–5024). PMLR. https://proceedings.mlr.press/v80/tucker18a.html

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention is all you need. *Advances in Neural Information Processing Systems*, *December*(Nips), 5999–6009.

Velicković, P., Cucurull, G., Casanova, A., Romero, A., Liò, P., & Bengio, Y. (2017). Graph attention networks. *arXiv*. https://arxiv.org/abs/1710.10903

Waltz, J. A., Knowlton, B. J., Holyoak, K. J., Boone, K. B., Mishkin, F. S., de Menezes Santos, M., Thomas, C. R., & Miller, B. L. (1999). A System for Relational Reasoning in Human Prefrontal Cortex. *Psychological Science*, *10*(2), 119–125. https://doi.org/10.1111/1467-9280.00118

Wang, H., Wang, K., Yang, J., Shen, L., Sun, N., Lee, H. S., & Han, S. (2020). GCN-RL circuit designer: Transferable transistor sizing with graph neural networks and reinforcement learning. *57th IEEE Design Automation Conference (DAC)*. https://arxiv.org/abs/2005.00406

Wang, K., Kang, B., Shao, J., & Feng, J. (2020). Improving Generalization in Reinforcement Learning with Mixture Regularization. http://arxiv.org/abs/2010.10814

Wang, M., Zheng, D., Ye, Z., Gan, Q., Li, M., Song, X., Zhou, J., Ma, C., Yu, L., Gai, Y., Xiao, T., He, T., Karypis, G., Li, J., & Zhang, Z. (2019). Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks. http://arxiv.org/abs/1909.01315

Wang, T., Liao, R., Ba, J., & Fidler, S. (2018). Nervenet: Learning structured policy with graph neural networks. *6th International Conference on Learning Representations, ICLR 2018 - Conference Track Proceedings*.

Wilson, R. J. (2005). *Introduction to Graph Theory* (Fourth Edi). Longman.

Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., & Yu, P. S. (2019). A Comprehensive Survey on Graph Neural Networks. https://doi.org/10.1109/TNNLS.2020.2978386

Xu, K., Hu, W., Leskovec, J., & Jegelka, S. (2019). How powerful are graph neural networks? *International Conference on Learning Representations, ICLR 2019 - Conference Track Proceedings*. https://arxiv.org/abs/1810.00826

Ye, Y., & Ji, S. (2021). Sparse Graph Attention Networks. *IEEE Transactions on Knowledge and Data Engineering*, 1. https://doi.org/10.1109/TKDE.2021.3072345

Yu, C., Liu, J., & Nemati, S. (2019). Reinforcement Learning in Healthcare: A Survey. http://arxiv.org/abs/1908.08796

Zambaldi, V., Raposo, D., Santoro, A., Bapst, V., Li, Y., Babuschkin, I., Tuyls, K., Reichert, D., Lillicrap, T., Lockhart, E., Shanahan, M., Langston, V., Pascanu, R., Botvinick, M., Vinyals, O., & Battaglia, P. (2018). Relational deep reinforcement learning. *arXiv*, https://arxiv.org/abs/1806.01830.

Železný, F., & Lavrač, N. (2008). Inductive Logic Programming. *18th International Conference, ILP*.

Zhang, J., He, T., Sra, S., & Jadbabaie, A. (2020). Why gradient clipping accelerates training: A theoretical justification for adaptivity. *ICLR*. http://arxiv.org/abs/1905.11881

Zhao, C., Sigaud, O., Stulp, F., & Hospedales, T. M. (2019). Investigating Generalisation in Continuous Deep Reinforcement Learning. https://arxiv.org/abs/1902.07015

Zhong, V., Xiong, C., & Socher, R. (2017). Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning. https://arxiv.org/abs/1709.00103

Zhou, J., Cui, G., Zhang, Z., Yang, C., Liu, Z., Wang, L., Li, C., & Sun, M. (2018). Graph Neural Networks: A Review of Methods and Applications. http://arxiv.org/abs/1812.08434

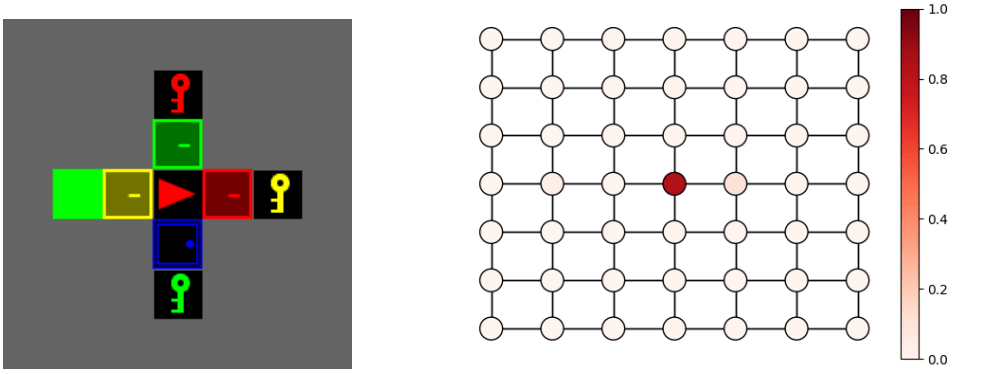
9

Appendix

9.1. Explainability through GAP and max-pooling

Applying GAP enables us to visualize the attention weights which the agent learns. Doing so provides a simple method to improve explainability, as we can then reason which nodes of the observation are weighted most heavily in the agent's decision for a certain action.

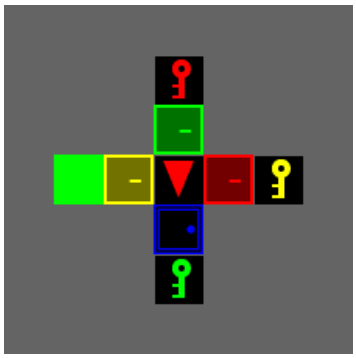
As our best performing graph-based methods use max-pooling as readout function, it would be interesting to have a similar kind of visualization of the 'focus' of the agent as is possible with GAP. We reason that the most important nodes for the agent, are the nodes from which the most features get pooled. To visualize this, we store - for each feature - the index i of the node n_i at which the maximum value was present, after which we count for each node $n_i \in N$ how many features were pooled from that specific node. The number of features pooled from node n_i is denoted with $p_i \in P$ where P has a cardinality of $|N|$. A softmax is then applied over all elements in P, giving a similar kind of measure as the attention scores of GAP. If all features would be pooled from one node, this node would have a value of 1 and if - hypothetically, as this is not exactly possible in our graphs - the exact same amount of features are pooled from each node in N, each node would have a weight of $1/|N|$. Figures 9.1-9.20 show an example of this visualization method for an agent with the R-GCN$_{\text{domain-max}}$ architecture being tested for out-of-distribution generalization to the Key-Corridors-Small environment with 3 keys, whilst it was trained on Key-Corridors-Small with either 1 or 2 keys. One can clearly see in these figures that the agent often 'focuses' on 1 or 2 nodes, which are often the nodes to which the 'can open' relation of the key that the agent is carrying is pointing.



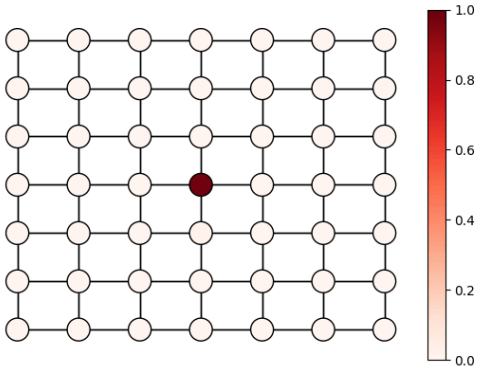
(a) Observation.

(b) The higher the value of a node, displayed by its color, the more features were pooled from this node.

Figure 9.1: The image on the left side (a), shows the rendered version of the observation of the agent. Its corresponding graphical representation is illustrated on the right (b), which displays the 'focus' of the agent. Note that the edges are added to illustrate the grid and do not represent the edges of the R-GCN$_{\text{domain-max}}$ architecture.

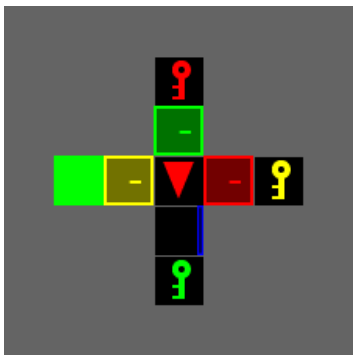


(a) Observation.

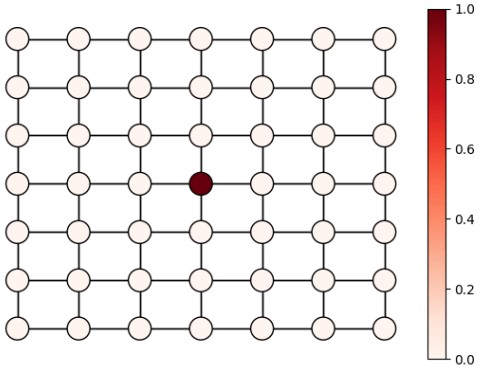


(b) The higher the value of a node, displayed by its color, the more features were pooled from this node.

Figure 9.2: The image on the left side (a), shows the rendered version of the observation of the agent. Its corresponding graphical representation is illustrated on the right (b), which displays the 'focus' of the agent. Note that the edges are added to illustrate the grid and do not represent the edges of the R-GCN$_{\text{domain-max}}$ architecture.

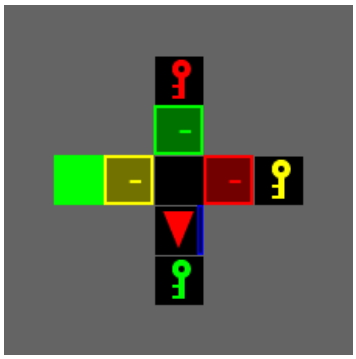


(a) Observation.

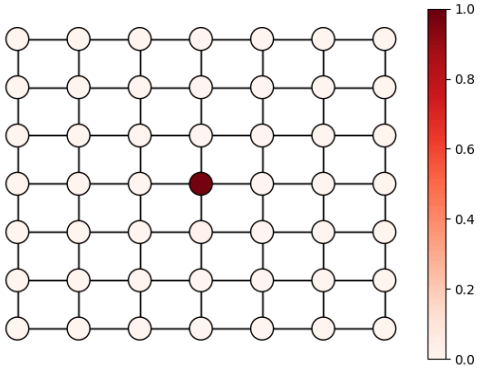


(b) The higher the value of a node, displayed by its color, the more features were pooled from this node.

Figure 9.3: The image on the left side (a), shows the rendered version of the observation of the agent. Its corresponding graphical representation is illustrated on the right (b), which displays the 'focus' of the agent. Note that the edges are added to illustrate the grid and do not represent the edges of the R-GCN$_{\text{domain-max}}$ architecture.

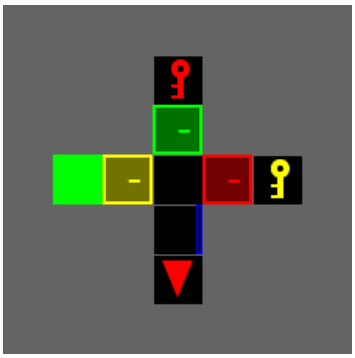


(a) Observation.

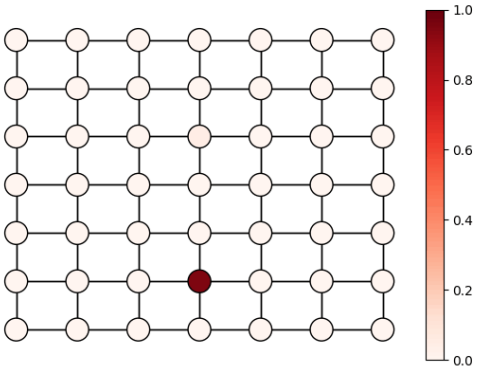


(b) The higher the value of a node, displayed by its color, the more features were pooled from this node.

Figure 9.4: The image on the left side (a), shows the rendered version of the observation of the agent. Its corresponding graphical representation is illustrated on the right (b), which displays the 'focus' of the agent. Note that the edges are added to illustrate the grid and do not represent the edges of the R-GCN$_{\text{domain-max}}$ architecture.

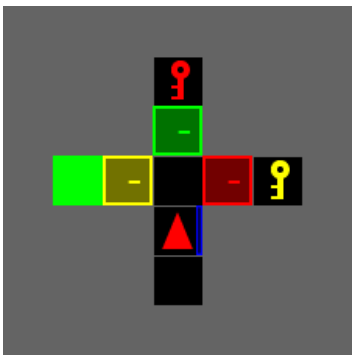


(a) Observation.

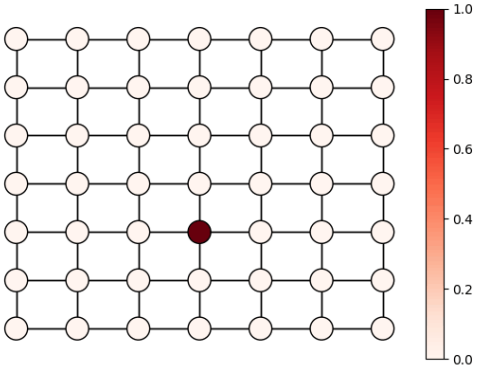


(b) The higher the value of a node, displayed by its color, the more features were pooled from this node.

Figure 9.5: The image on the left side (a), shows the rendered version of the observation of the agent. Its corresponding graphical representation is illustrated on the right (b), which displays the 'focus' of the agent. Note that the edges are added to illustrate the grid and do not represent the edges of the R-GCN$_{\text{domain-max}}$ architecture.

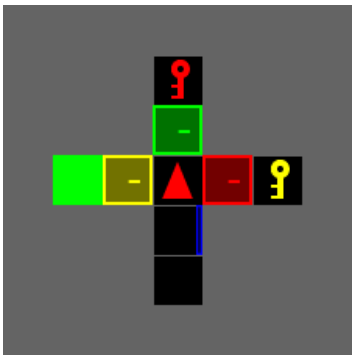


(a) Observation.

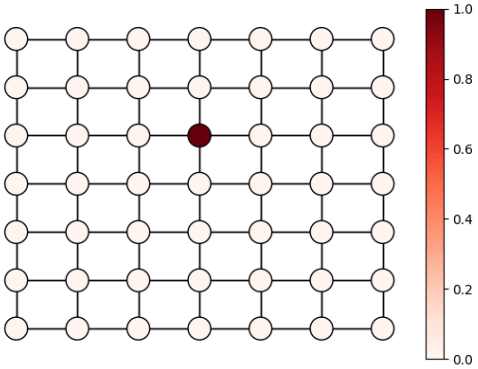


(b) The higher the value of a node, displayed by its color, the more features were pooled from this node.

Figure 9.6: The image on the left side (a), shows the rendered version of the observation of the agent. Its corresponding graphical representation is illustrated on the right (b), which displays the 'focus' of the agent. Note that the edges are added to illustrate the grid and do not represent the edges of the R-GCN$_{\text{domain-max}}$ architecture.

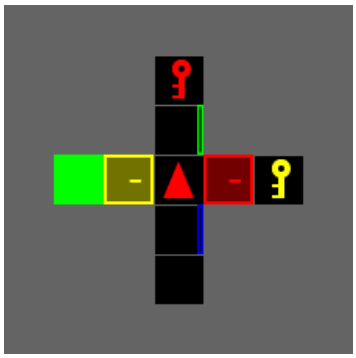


(a) Observation.

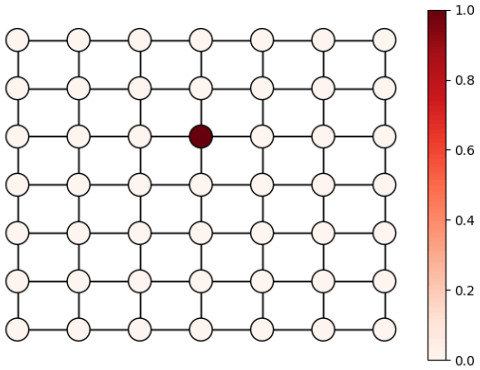


(b) The higher the value of a node, displayed by its color, the more features were pooled from this node.

Figure 9.7: The image on the left side (a), shows the rendered version of the observation of the agent. Its corresponding graphical representation is illustrated on the right (b), which displays the 'focus' of the agent. Note that the edges are added to illustrate the grid and do not represent the edges of the R-GCN$_{\text{domain-max}}$ architecture.

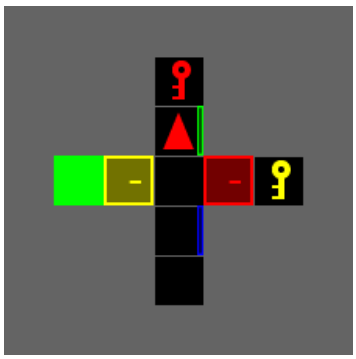


(a) Observation.

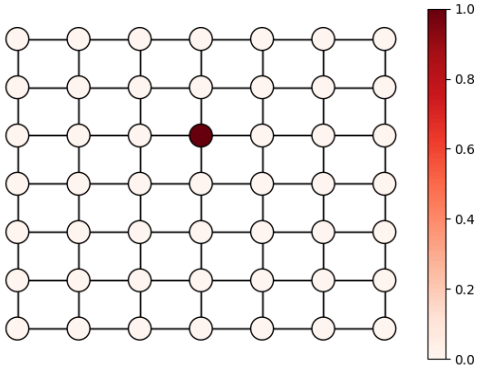


(b) The higher the value of a node, displayed by its color, the more features were pooled from this node.

Figure 9.8: The image on the left side (a), shows the rendered version of the observation of the agent. Its corresponding graphical representation is illustrated on the right (b), which displays the 'focus' of the agent. Note that the edges are added to illustrate the grid and do not represent the edges of the R-GCN$_{\text{domain-max}}$ architecture.

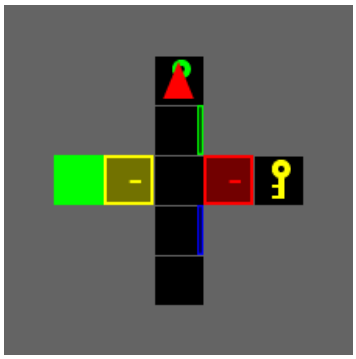


(a) Observation.

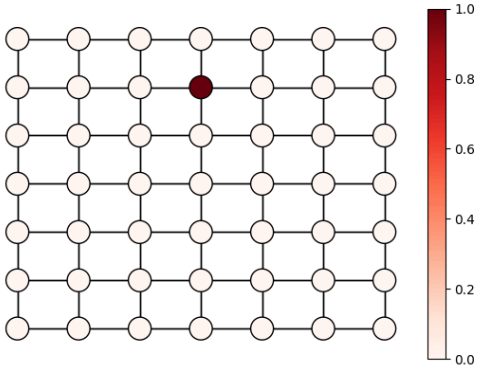


(b) The higher the value of a node, displayed by its color, the more features were pooled from this node.

Figure 9.9: The image on the left side (a), shows the rendered version of the observation of the agent. Its corresponding graphical representation is illustrated on the right (b), which displays the 'focus' of the agent. Note that the edges are added to illustrate the grid and do not represent the edges of the R-GCN$_{\text{domain-max}}$ architecture.

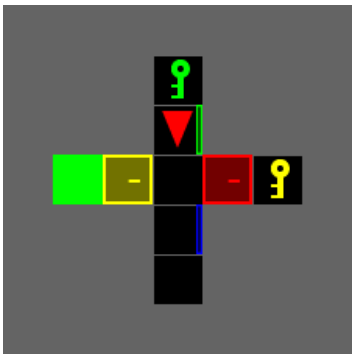


(a) Observation.

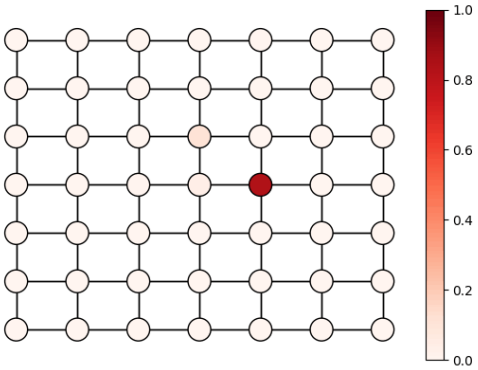


(b) The higher the value of a node, displayed by its color, the more features were pooled from this node.

Figure 9.10: The image on the left side (a), shows the rendered version of the observation of the agent. Its corresponding graphical representation is illustrated on the right (b), which displays the 'focus' of the agent. Note that the edges are added to illustrate the grid and do not represent the edges of the R-GCN$_{\text{domain-max}}$ architecture.

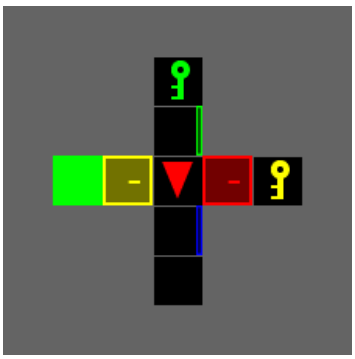


(a) Observation.

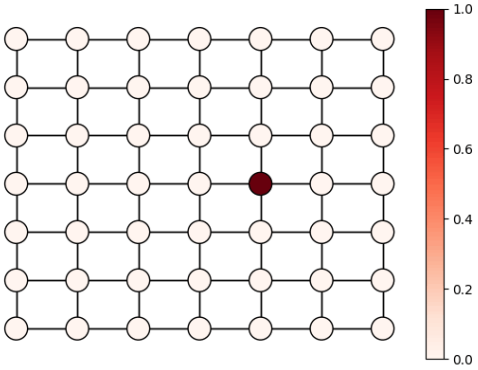


(b) The higher the value of a node, displayed by its color, the more features were pooled from this node.

Figure 9.11: The image on the left side (a), shows the rendered version of the observation of the agent. Its corresponding graphical representation is illustrated on the right (b), which displays the 'focus' of the agent. Note that the edges are added to illustrate the grid and do not represent the edges of the R-GCN$_{\text{domain-max}}$ architecture.

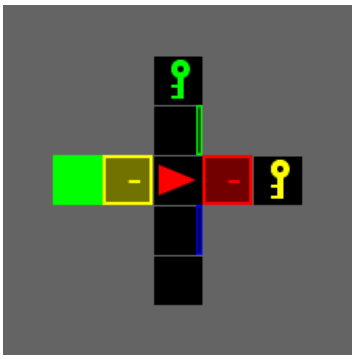


(a) Observation.

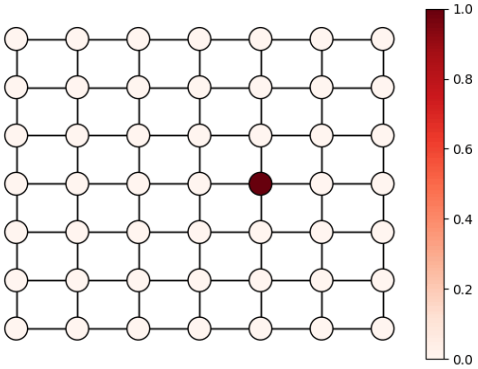


(b) The higher the value of a node, displayed by its color, the more features were pooled from this node.

Figure 9.12: The image on the left side (a), shows the rendered version of the observation of the agent. Its corresponding graphical representation is illustrated on the right (b), which displays the 'focus' of the agent. Note that the edges are added to illustrate the grid and do not represent the edges of the R-GCN$_{\text{domain-max}}$ architecture.

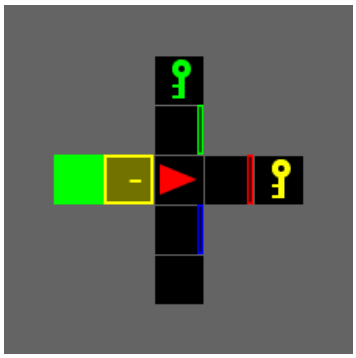


(a) Observation.

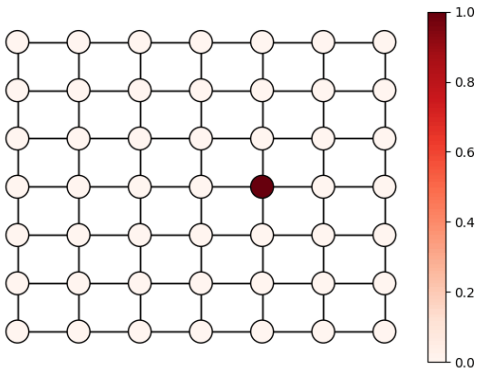


(b) The higher the value of a node, displayed by its color, the more features were pooled from this node.

Figure 9.13: The image on the left side (a), shows the rendered version of the observation of the agent. Its corresponding graphical representation is illustrated on the right (b), which displays the 'focus' of the agent. Note that the edges are added to illustrate the grid and do not represent the edges of the R-GCN$_{\text{domain-max}}$ architecture.

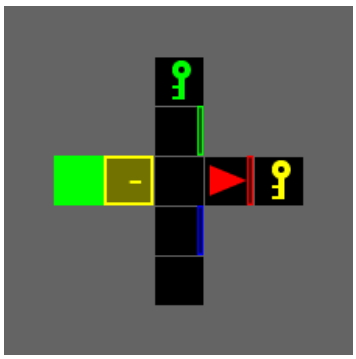


(a) Observation.

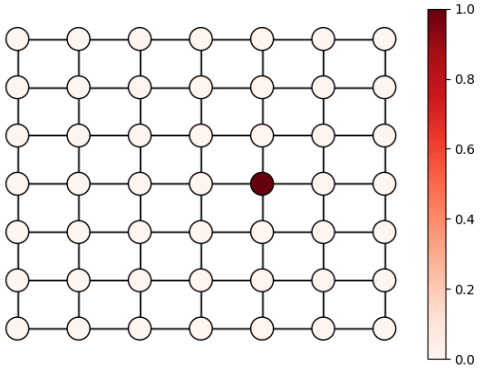


(b) The higher the value of a node, displayed by its color, the more features were pooled from this node.

Figure 9.14: The image on the left side (a), shows the rendered version of the observation of the agent. Its corresponding graphical representation is illustrated on the right (b), which displays the 'focus' of the agent. Note that the edges are added to illustrate the grid and do not represent the edges of the R-GCN$_{\text{domain-max}}$ architecture.

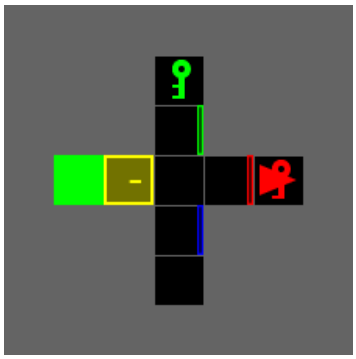


(a) Observation.

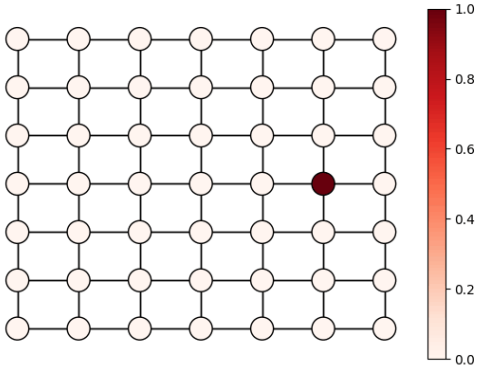


(b) The higher the value of a node, displayed by its color, the more features were pooled from this node.

Figure 9.15: The image on the left side (a), shows the rendered version of the observation of the agent. Its corresponding graphical representation is illustrated on the right (b), which displays the 'focus' of the agent. Note that the edges are added to illustrate the grid and do not represent the edges of the R-GCN$_{\text{domain-max}}$ architecture.

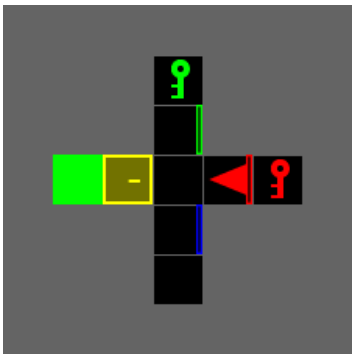


(a) Observation.

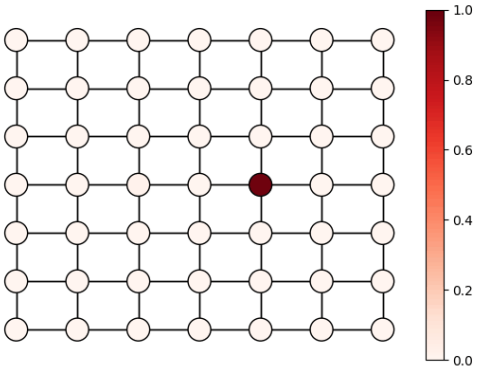


(b) The higher the value of a node, displayed by its color, the more features were pooled from this node.

Figure 9.16: The image on the left side (a), shows the rendered version of the observation of the agent. Its corresponding graphical representation is illustrated on the right (b), which displays the 'focus' of the agent. Note that the edges are added to illustrate the grid and do not represent the edges of the R-GCN$_{\text{domain-max}}$ architecture.

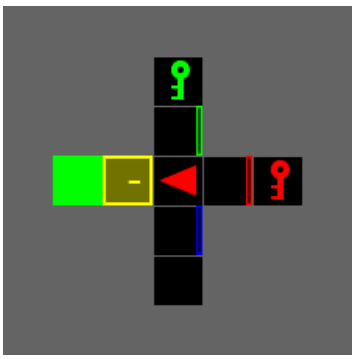


(a) Observation.

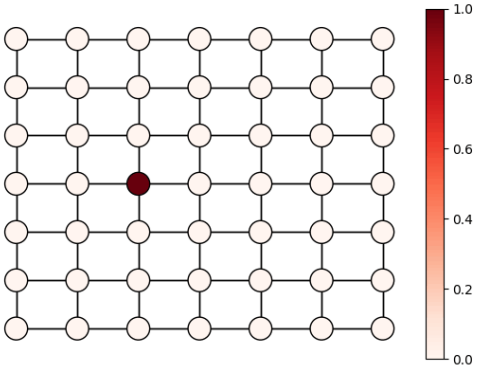


(b) The higher the value of a node, displayed by its color, the more features were pooled from this node.

Figure 9.17: The image on the left side (a), shows the rendered version of the observation of the agent. Its corresponding graphical representation is illustrated on the right (b), which displays the 'focus' of the agent. Note that the edges are added to illustrate the grid and do not represent the edges of the R-GCN$_{\text{domain-max}}$ architecture.

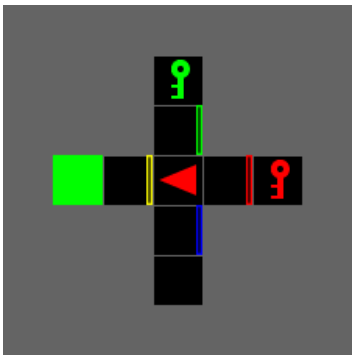


(a) Observation.

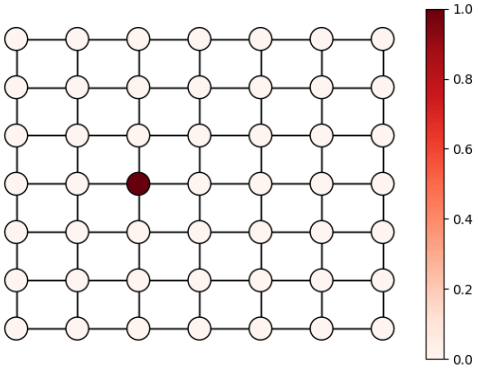


(b) The higher the value of a node, displayed by its color, the more features were pooled from this node.

Figure 9.18: The image on the left side (a), shows the rendered version of the observation of the agent. Its corresponding graphical representation is illustrated on the right (b), which displays the 'focus' of the agent. Note that the edges are added to illustrate the grid and do not represent the edges of the R-GCN$_{\text{domain-max}}$ architecture.

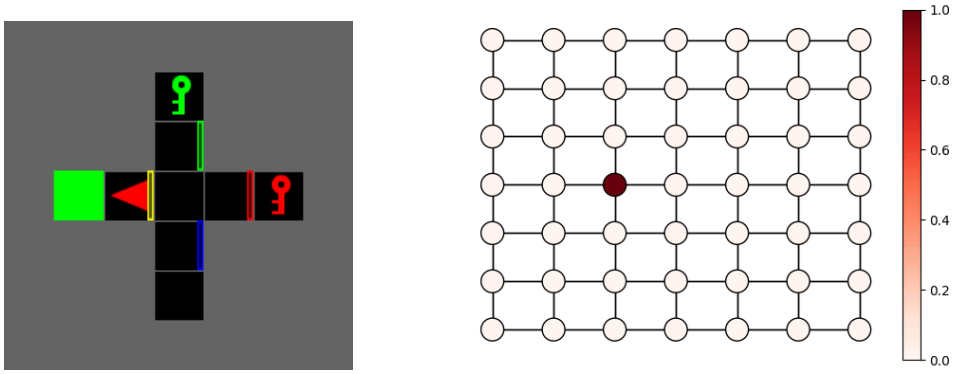


(a) Observation.



(b) The higher the value of a node, displayed by its color, the more features were pooled from this node.

Figure 9.19: The image on the left side (a), shows the rendered version of the observation of the agent. Its corresponding graphical representation is illustrated on the right (b), which displays the 'focus' of the agent. Note that the edges are added to illustrate the grid and do not represent the edges of the R-GCN$_{\text{domain-max}}$ architecture.



(a) Observation.

(b) The higher the value of a node, displayed by its color, the more features were pooled from this node.

Figure 9.20: The image on the left side (a), shows the rendered version of the observation of the agent. Its corresponding graphical representation is illustrated on the right (b), which displays the 'focus' of the agent. Note that the edges are added to illustrate the grid and do not represent the edges of the R-GCN$_{\text{domain-max}}$ architecture.