Saarland University

Master Thesis

Using Deep Reinforcement Learning to Optimize Assignment Problems

Submitted by:

Joschka Groß

Submitted on:

01.06.2021

Reviewers:

Univ.-Prof. Dr. Verena Wolf

Dr. Andreas Karrenbauer

Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, 01.06.2021

_____

Your name

Abstract

Recently, deep reinforcement learning (DRL) has emerged as an effective method for solving discrete-time sequential decision-making problems. This thesis studies how deep reinforcement learning can be applied to solve combinatorial optimization problems. To this end, we conduct a case study motivated by a real-world problem.

Acting as an educational institution, the Konrad Adenauer foundation (KAS) offers courses to their scholarship students and past members. Based on their preferences, the assignment problem consists in finding a reasonable assignment of students to their requested courses. Drawing inspiration from a catalogue of rules these assignments should obey, we interpret this problem as combinatorial optimization problem and formulate a simplified version thereof as a linear integer program.

In order to apply DRL, we reformulate the problem using Markov decision processes and construct neural network architectures that enable DRL agents to learn a meta-policy which is applicable to arbitrary instances of the assignment problem, as opposed to being forced to learn instance-specific policies only. We compare our proposed DRL methods to a strong solver, namely Gurobi, that uses a branch-and-cut method for solving integer programs, whereby we put our focus on the ability of DRL to approximate optimal solutions and its time-efficiency compared to Gurobi.

Acknowledgments

I would like to thank Dr. Andreas Karrenbauer and Prof. Dr. Verena Wolf for their supervision as well as for reviewing this thesis and making it possible in the first place.

Furthermore, I wish to thank my additional supervisor and colleague Timo P. Gros who provided me with helpful advice and feedback, and greatly supported me throughout the last year. I would also like to thank Thilo Krüger for his valuable input in discussions on my thesis.

My deepest gratitude however belongs to my closest friends and family. To my friends Lukas, Jakob, Pascal and Timm for taking my mind of work when I needed a break. To my mother, Andrea, father, Frank, and siblings Mila and Simon for always supporting and encouraging me in whatever I do. And most importantly, to my partner, Vera, for being such a wonderful human being and supporting me on this journey like no one else could. Without the continued support from all of you I could not have done all of this.

Contents

1 Introduction **1**

2 Related Work **3**

3 Problem Formulation **5**
3.1 Combinatorial Optimization and Integer Programming 5
3.1.1 Terminology and Formulation 5
3.1.2 Algorithms for solving Integer Programs 6
3.2 Assignment Problems formulated as Integer Programs 6
3.2.1 Simple Assignment Problem 6
3.2.2 Valued Assignment Problem 8

4 Background **11**
4.1 Reinforcement Learning 11
4.1.1 Basic Formalisms 11
4.1.2 Deep Reinforcement Learning Algorithms 13
4.2 Deep Learning 19
4.2.1 Feed-forward Networks and Activation Functions 20
4.2.2 Recurrent Graph Neural Networks 22
4.2.3 Neural Attention 23
4.2.4 Gradient Based Optimization 26

5 Applying Deep Reinforcement Learning **29**
5.1 Constructing Markov Decision Processes 29
5.2 Constrained Policies 30
5.3 Single Instance vs. Multi Instance Learning 31
5.4 Single Instance Network Architecture 32
5.5 Formalizing the Multi Instance Objective 32
5.6 Multi Instance Network Architecture 33
5.6.1 Encoder Modules 34
5.6.2 Aggregation Module 37
5.6.3 Policy and State-Action Value Modules 37

6 Experiments **39**
6.1 Instance Training Distributions 40
6.2 Comparison to Single Instance Architectures 42
6.3 Training DRL Agents 43
6.3.1 Tuning Recurrence in Graph Neural Networks 44
6.3.2 Multi Instance Training 45
6.4 Evaluation 48
6.4.1 Evaluation Setup 49

6.4.2 Generalization to unseen instances 50
6.4.3 Inspecting Constraint Violations 53
6.4.4 Solving-Time Comparison 55
6.4.5 Generalization KAS 2020 59

7 Reinforcement Learning and Gradient Ascent: what's the difference anyway? 61

8 Conclusion and Future Work 65

1 Introduction

The kind of assignment problem we address here is commonly encountered in the education sector. Each year or semester, institutions such as universities or political foundations offer a range of different courses to their students or members. In most cases the students have to make participation requests or apply for their preferred courses ahead of time. Given these requests, the educational institutions have to decide on a final assignment of students to courses that does not exceed their capacities, is in accordance with any additional formal constraints and that meets the student preferences as well as possible. An assignment satisfying all constraints or student preferences seldom exists and thus we are faced with the optimization problem of finding an assignment that maximizes student happiness and minimizes the number of constraint violations. We will refer to these problems as assignment problems. For our purposes, we mainly work with artificial problem models, but our work is in parts motivated by a real data set provided by the Konrad Adenauer Stiftung (KAS) which inspired a large part of the artificial problem details. A canonical approach to solving such problems is to interpret them as combinatorial optimization problems and use standard state-of-the-art optimization methods for solving them

Combinatorial optimization [34] is a long standing and well studied field, that already provides general methods for finding optimal, or approximately optimal solutions to arbitrary integer programming problems. If we are able to formulate an integer program that consists of a suitable objective function and constraints for modeling the assignment problem we can apply a solver that uses for instance a branch-and-cut [34] method to solve the problem. These methods are fairly exact in the sense that when given a sufficient amount of time, they will be able to find optimal solutions (within a tolerance).

The purpose of this thesis lies within studying how deep reinforcement learning [29] (DRL) can be applied as an alternative method for solving combinatorial optimization problems by conducting a case study on the aforementioned assignment problems.

DRL algorithms (agents) use deep neural networks to approximate optimal policies for sequential decision-making problems. Given a Markov decision process (MDP) that formalizes the agent's environment, the concrete task is to find a strategy for selecting sequences of actions that maximize a numeric reward signal. The agents can learn to make informed decisions by observing the effects of their actions on the environments states as well the rewards they yield. Our motivation to use DRL stems from the fact that it has recently been the driving factor behind great advancements in challenging applications such as learning to play Chess, Go [28], or Atari video games [14] at a super-human level, just to name a few.

Even though deep reinforcement learning is an inexact method, i.e., it does not come with strong guarantees for finding optimal solutions, the above applications suggest

that it might be able to find sufficiently close to optimal solutions when applied to our assignment problems. A possible advantage of DRL over exact methods that further motivates this approach is that once an approximately optimal policy has been found, using this policy to infer reasonable solutions could serve as a strong, and time-efficient heuristic.

However, in contrast to classical optimization techniques, DRL is not easily applicable to combinatorial optimization out-of-the-box for several reasons. Through the means of our case study, we hope to come one step closer to establishing DRL as a relevant method for solving combinatorial optimization problems and gain insights on what is required to further advance this approach. The contributions of this thesis represent an exemplary procedure on how to reformulate combinatorial optimization problems as MDPs and on how to successfully apply DRL in a way that is adequately efficient

We identify that a naive approach wherein an agent learns to solve each new instance of the assignment problem from scratch does not fulfill our efficiency requirements. As a remedy, we study an approach that enables DRL agents to learn a meta-policy, which is applicable to arbitrary instances of the assignment problem. It will be interesting to investigate, whether such a policy can be learned in a way that allows us to find reasonable assignments for multiple different problem instances.

The key contribution that enables this multi-instance meta-policy approach lies within constructing suitable neural networks architectures. We will construct several such architectures and compare their ability to approximate strong, generalizable policies and also their time-efficiency to Gurobi, a powerful commercial solver, that uses a branch-and-cut method for tackling discrete optimization problems such as the assignment problem.

Let us now go over a brief outline of the exact contents of this thesis. We start with a brief background section on combinatorial optimization that then allows us to properly formulate assignment problems as integer programs in Chapter 3. After a formal introduction of all necessary preliminaries regarding reinforcement learning, deep learning, and deep reinforcement learning in Chapter 4, we will then be able to present our contributions on how to apply DRL to the problem at hand in Chapter 5. Subsequently, as we are then equipped with a single-instance and multi-instance approach for applying DRL, Chapter 6 describes several experiments comparing the two approaches as well as detailed sections on training multi-instance DRL agents and evaluating their resulting policies with a focus on how well they are able to compete with Gurobi. In Chapter 7 we briefly describe an interesting parallel between reinforcement learning and gradient-based optimization that highlights a possible improvement of our approach, which will then be discussed, along with further avenues for future work, in the final, concluding Chapter 8.

2 Related Work

The idea to tackle combinatorial optimization problems with deep reinforcement learning is not new, and the idea to use neural networks in general for combinatorial optimization dates back even further [16].

The first paper we are aware of that successfully used deep reinforcement learning for combinatorial optimization revolves around *pointer networks* that are trained with a reinforcement learning approach [2]. They are introduced as part of a case study on solving the euclidean traveling salesman problem. The cities to be visited are given as coordinates in 2D euclidean space and are encoded into a sequence of hidden states by applying an recurrent long-short-term-memory (LSTM) network [15] to a randomized-order sequence of the cities. Then, a tour is generated by using an attention-based module called „pointing mechanism“. Given a fixed size representation of the current tour and the sequence of city hidden states, it produces a distribution over all these hidden states, where high probabilities mean that the network „points“ to this particular city as a promising candidate for extending the tour. The fixed size tour representation is generated online as the hidden state of another LSTM that takes as input the city tour as it is sampled from the city pointers. Using negative tour length as a sparse reward, the pointer networks are trained with the policy gradient method [29].

A paper published briefly after pointer networks introduces *S2V-DQN* (structure2vec Deep Q Networks) [18] as a general method for solving combinatorial optimization problems that can be formulated over graphs, where solutions are given by subsets of vertices. For this purpose they use structure2vec [6], which can be seen as a special type of recurrent graph neural network [35]. It produces fixed size embeddings for all vertices of a graph. For applying S2V-DQN to the maximum cut and minimum vertex cover problem [17], the tasks are reformulated as MDPs, where actions correspond to selecting a single vertice that is added into the subset state, that is initially empty. The action-value, required by the DQN algorithm, for adding some vertice is calculated using these embeddings by additional neural network layers which are trained jointly with the structure2vec layers.

3 Problem Formulation

In this chapter we first give a brief introduction to combinatorial optimization and integer programs and then formally define two realistic problems for assigning students to courses as integer programs.

3.1 Combinatorial Optimization and Integer Programming

3.1.1 Terminology and Formulation

In one way or another, all optimization problems consist in finding the „best“ solution from a set of „feasible“ candidate solutions, where the „goodness“ of a candidate solution is measured by an objective function mapping them to numeric values, and feasibility is determined by constraints that partition the set of candidate solutions into a feasible and infeasible region.

Discrete optimization problems are characterized by the fact that (at least parts of) the solutions are restricted to be objects from some fixed, discrete domain. By Combinatorial optimization [34] (CO) we denote a special branch of discrete optimization. CO problems usually revolve around some fixed set of objects P that is used to describe a specific structure, e.g., the set of vertices of a graph. The domain from which solutions have to be selected is then given by 2^P, e.g., the task may involve selecting a minimal subset of all vertices of a graph s.t. some conditions are satisfied.

Linear Integer Programs (IPs) [34] provide a standardized way of formulating discrete optimization problems with linear objective functions. They are written as

$$\begin{aligned} \max_{x} \quad & c^T x \\ \text{s.t.} \quad & Ax \leq b \\ & x \geq 0 \end{aligned} \tag{3.1}$$

where $x \in \mathbb{Z}^n$ are the solutions being sought, $c \in \mathbb{R}^n$ defines the objective, and $A \in \mathbb{R}^{m \times n}$ together with $c \in \mathbb{R}^m$ define m constraints. When defining an integer program, instead of giving c, A, and b explicitly, it is often convenient to state them in a more human-readable form, for which it is obvious that it can be easily brought into the above canonical form.

Many combinatorial optimization problems, including the ones we will consider in the next section, can be formulated as binary integer problems, i.e., the constraints require $0 \leq x \leq 1$ where the values 0 and 1 determine the subset of P selected as corresponding solution.

When discussing combinatorial optimization problems, it is sensible to make a distinction between the problem itself and all possible problem *instances* it applies to. For example,

we could formulate an integer program for finding a minimal subset $P \subseteq V$ of vertices s.t. this set covers all edges of the graph, i.e., we require $(u, v) \in E \implies \{u, v\} \cap P \neq \emptyset$. A good formulation of this problem should be agnostic to which graph (V, E) exactly is considered, and allow us to construct an IP for any graph we want. In this case the possible instances would be all possible graphs.

3.1.2 Algorithms for solving Integer Programs

In our experiments, when comparing reinforcement learning to standard methods, we make use of the strong commercial solver Gurobi [12]. We treat Gurobi mainly as a black box and are only interested in whether it is able to find optimal solutions (within a tolerance) and how long the solving process takes. We just note at this point, that according to the official documentation [11], Gurobi uses a branch-and-cut algorithm [34] for solving integer programs. The formalities of branch-and-cut and the particular implementation Gurobi uses are out of the scope of this thesis. Nevertheless, in Chapter 6 on experiments, we give some details on how we set up Gurobi to solve problems (see Section 6.4.1).

3.2 Assignment Problems formulated as Integer Programs

3.2.1 Simple Assignment Problem

The *simple assignment problem* focuses on assignments of students to courses that are constrained in such a way that

- every student should be assigned only to courses of his choice,
- no student that made at least two choices should be left unassigned,
- students should be assigned to at most three courses, and
- every course has a fixed capacity that may not be exceeded.

Definition 1. An *instance of the simple assignment problem* is given by a 4-tuple $I = \langle s, c, C, R \rangle$ with students $s = \{s_1, \ldots, s_n\}$ and courses $c = \{c_1, \ldots, c_m\}$. Student requests are given by a relation $R \subseteq s \times c$ between students and courses where $(s_i, c_j) \in R$ iff some student s_i requests to be assigned to some course c_j. Additionally, courses have finite capacities $C : \{1, \ldots, m\} \mapsto \mathbb{N}$.

Note that these instances essentially represent bipartite graphs over students and courses, with requests as edges, where courses are additionally labeled with some capacity. We will make use of this fact in Section 5.6.1 when designing neural network architectures.

Assuming we are given an arbitrary instance, *assignments* specify which requests are granted and which are declined. They can also be represented as relations $X \subseteq s \times c$ between students and courses, where $(s_i, c_j) \in X$ means that the request of student s_i to be be assigned to course c_j is granted by X. Furthermore we require that $X \subseteq R$, because assignments should only include tuples (s_i, c_j), where s_i really requested course

c_j. The objective is to find an assignment that grants as many requests as possible while still adhering to the above constraints.

Before formulating this notion as an integer program, we first define some notation to improve readability. For this purpose let us for now think of the requests and assignment relations as total functions $s \times c \mapsto \{0, 1\}$, where $r(s_i, c_j) = 1$ iff $(s_i, c_j) \in R$ and $x(s_i, c_j) = 1$ iff $(s_i, cj) \in X$. For the sake of brevity we write $x_{i,j}$ instead of $x(s_i, c_j)$ and $r_{i,j}$ instead of $r(s_i, c_j)$. From the perspective of integer programming, the $x_{i,j}$ can easily be viewed as binary variables. Let us now define several useful quantities.

- $r_{i,:} = \sum_{j=1}^{m} r_{i,j}$, the number of courses requested by s_i.
- $x_{i,:} = \sum_{j=1}^{m} x_{i,j}$, the number of courses assigned to s_i.
- $r_{:,j} = \sum_{i=1}^{n} r_{i,j}$, the number of students requesting c_j.
- $x_{:,j} = \sum_{i=1}^{n} x_{i,j}$, the number of students assigned to c_j.

Given an instance $\langle s, c, C, R\rangle$, we want to find an assignment X that is both a feasible and optimal solution w.r.t.

$$\text{maximize} \quad J(X) = \sum_{(s_i,c_j)\in R} x_{i,j} \tag{3.2}$$

$$\text{subject to} \quad x_{:,j} \leq C(j), \quad j = 1, ..., m \tag{3.3}$$

$$x_{i,:} \geq 1, \quad \forall i \in \{1, ..., n\} : r_{i,:} > 1 \tag{3.4}$$

$$x_{i,:} \leq 3, \quad i = 1, ..., n \tag{3.5}$$

Equation 3.3 ensures that courses are not overfilled beyond their capacity. Equation 3.4 captures the constraint that students who applied for at least two courses should be assigned to at least one of their requested courses. Equation 3.5 guarantees that students are assigned to at most 3 courses.

As feasible assignments, i.e., assignments that satisfy all constraints, might not exist for some instances, we also consider a version of the problem where the constraints defined by 3.4 may be relaxed to soft constraints such that constraint violations are allowed, but incur a penalty to the objective function. In the relaxed formulation, assignments that were previously infeasible are now subject to the penalized objective

$$\text{maximize } J^{-}(X) = \sum_{(s_i,c_j)\in R} x_{i,j} - \sum_{i=1}^{n} 10 \cdot \mathbb{I}\left[r_{i,:} > 1 \wedge x_{i,:} = 0\right], \tag{3.6}$$

where $\mathbb{I}[P]$ is an indicator function that evaluates to 1 if its argument P is true and 0 otherwise. That means we subtract 10 from the objective each time a constraint as defined in 3.4 is violated. The other constraints are left untouched, so assignments violating them are still considered infeasible. The penalty of 10 might seem somehwat arbitrary, but we did not have any external information on how much constraint violations should be weighted and we obviously had to settle on something, so we decided on a penalty that heavily outweighs the value of accepting just one more request

in the objective function, because this should ensure that if feasible assignments that do not violate any constraints exists, then they are also still optimizers for this penalized objective.

3.2.2 Valued Assignment Problem

The previous assignment problem is named simple for a reason. As we will see in Chapter 6, Gurobi performs exceptionally well on its instances. It solves most instances optimally within fractions of seconds, so there is not much room left for possible improvements that might be achieved with DRL methods. This is the reason why we also consider an extended version of the assignment problem that adds complexity in the form of requests with unequal priorities. For these *valued assignment problems*, we assume that students can express their preferences by assigning different values to their requests, instead of making equal priority requests. A request with a higher value signals that this request should be prioritized over other requests. We formalize request values as a total function mapping requests to their respective value.

Definition 2. An *instance of the valued assignment problem* is given by a 5-tuple $I = \langle s, c, C, R, V \rangle$, where s, c, R and C are defined as in Definition 1 and $V : R \mapsto [0, 1]$ assigns a numeric value to each request.

These instances can be interpreted as weighted bipartite graphs where the weights are given by request values.

Assuming an arbitrary assignment X, let us now again define some useful notation.

- $v_{i,j} = V(s_i, c_j)$. Abbreviated notation for the value of the request (s_i, c_j).
- $v_{i,:} = \sum_{j=1}^{m} v_{i,j}$. The total value of requests made by student s_i.
- $v_{:,j} = \sum_{i=1}^{n} v_{i,j}$. The total value of requests made for course s_j.
- $w_{i,:} = \sum_{j=1}^{m} v_{i,j} x_{i,j}$. The total value of requests granted by X for student s_i.
- $\bar{w}_{:,j} = \frac{1}{C(j)} \sum_{i=1}^{n} v_{i,j} x_{i,j}$. The mean value of requests granted by X concerning course c_j.

For the sake of simplicity we assume that every student has a value budget of 1 he can distribute across all of his requests, i.e., we only consider instances where $v_{i,:} = 1$. This is also sensible, because instances where the requests of one student are valued higher than those of another student would entail that the students are not treated equally, making any assignments that optimize for these values unfair.

For this kind of assignment problem we want to maximize the global value of the accepted requests while also adhering to some new constraints. These new constraints are based on the following considerations.

In order to ensure that student preferences are met to some minimal extent, the total value of requests that are granted to any student should be greater than some minimal total student request value v^s_{min}. A similar argument can be made for courses.

Consider the value $w_{:,j}$. From the perspective of the institution that offers the course c_j, it might also be desirable if the value $w_{:,j}$ (normalized by the courses capacity $C(j)$) is large. This is the case, because requests (s_i, c_j) with relatively large values can be interpreted as a consequence of student s_i being highly motivated to participate in course c_j. So assignments where $w_{:,j}$ is larger than some minimum average request value v^c_{min} could help contribute to the average quality of student contributions to the course c_j (assuming that higher motivation leads to more qualitative contributions).

A corresponding integer program that formalizes these constraints and that defines the optimality criterion is given by

$$\begin{aligned}
&\text{maximize} \quad J(X) = \sum_{(s_i,c_j)\in R} v_{i,j}x_{i,j} && && (3.7)\\
&\text{subject to} && (3.3), \quad (3.4), \quad (3.5) && \\
& && w_{i,:} \geq v^s_{min}, \qquad i = 1, ..., n && (3.8)\\
& && \bar{w}_{:,j} \geq v^c_{min}, \qquad j = 1, ..., m && (3.9)
\end{aligned}$$

Depending on the concrete values for v^s_{min} and v^c_{min}, it is also often the case that assignments which satisfy all constraints 3.8 and 3.9 do not exist. So, we will again have to resort to a relaxation of the problem, where we impose a penalized objective function

$$\begin{aligned}
\text{maximize } J^-(X) = &\sum_{(s_i,c_j)\in R} v_{i,j}x_{i,j} \\
&- \sum_{i=1}^{n} 10 \cdot \mathbb{I}\left[r_{i,:} > 1 \wedge x_{i,:} = 0\right] + 10 \cdot \mathbb{I}\left[w_{i,:} \geq v^s_{min}\right] \\
&- \sum_{j=1}^{m} 10 \cdot \mathbb{I}\left[\bar{w}_{:,j} \geq v^c_{min}\right].
\end{aligned} \tag{3.10}$$

For our experiments a systematic study over regarding different values of v^s_{min} and v^c_{min} was out of scope due to long DRL training times. We thus fix $v^c_{min} = 0.5$ and consider two different cases $v^s_{min} \in \{0.5, 0.7\}$ where we expect that it should be more difficult to learn, or find good assignments in the case $v^s_{min} = 0.7$, as the constraints defined by 3.8 becomes more difficult to satisfy.

At this point we would like to summarize our treatment of the different constraints we introduced. Course capacities 3.3 and the maximum number of assigned courses per student 3.5 remain hard constraints no matter what. *If* a feasible solution does not exist, *all other constraints are relaxed*, whereby each constraint is assigned the same penalty of 10, that applies if the constraint is violated by assignments for the relaxed problem, independent of the violation magnitude.

Before closing this chapter we introduce an additional concept that will be useful later on, when defining an MDP in Chapter 5. Given any instance I of either the simple or valued assignment problem and an assignment X, we denote the set of all requests that

are still *open* by

$$\mathrm{Open}_I(X) = \{(s_i, c_j) \in R \mid x_{i,j} = 0 \wedge x_{:,j} < C(j) \wedge x_{i,:} < 3\}\,. \tag{3.11}$$

These are all requests that can still be granted without violating any of the hard constraints.

4 Background

In this chapter we introduce Reinforcement Learning, a method for solving sequential decision-making problems and respective deep reinforcement learning algorithms. In order to be effective in pratice, these algorithms also require carefully designed deep neural networks as function approximators. The deep learning methods we plan to use for this task are also introduced within this chapter.

4.1 Reinforcement Learning

Put simply, reinforcement learning (RL) is the process of solving sequential decision-making problems through trial and error. The learning entity, commonly referred to as RL *agent*, interacts with its environment in a feedback loop, where the agent's actions change the environments state and the agent observes these states to make an informed decision between possible actions. The agent's goal is to choose actions that maximize a numerical reward signal generated in parallel to the state changes. Throughout the learning process the agent is reinforced to make decisions that lead to a large accumulated reward. In this section we focus on formally defining essential reinforcement learning concepts and use them to describe the deep reinforcement learning algorithms we intend to apply to the assignment problems defined in the previous section.

4.1.1 Basic Formalisms

Integer programs, which include our models of the assignment problem, do not feature any form of stochasticity and are inherently discrete. Thus we can restrict our work to a simple formalism of Markov decision processes with deterministic state transitions and deterministic rewards.

Definition 3. A *deterministic Markov Decision Process* (MDP) is a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{T}, s_0, \mathcal{S}^* \rangle$, where $\mathcal{S}$ is a finite set of states, $s_0 \in \mathcal{S}$ is the initial state and $\mathcal{S}^* \subseteq \mathcal{S}$ is a set of terminal states. $\mathcal{A}$ is a finite set of actions, $\mathcal{R} : \mathcal{S} \times \mathcal{A} \mapsto \mathbb{R}$ is a reward function and $\mathcal{T} : \mathcal{S} \times \mathcal{A} \mapsto \mathcal{S}$ is a transition function.

The sequentiality of an MDP is captured by its possible trajectories which are sequences of states, actions and rewards

$$\tau = \langle s_0, a_0, r_1, s_1, \ldots, s_{T-1}, a_{T-1}, r_T, s_T \rangle ,$$

where s_0 is the initial state, $s_T \in \mathcal{S}^*$ is a terminal state and *transitions* (also called *environment steps*) are captured by the relations $s_{t+1} = \mathcal{T}(s_t, a_t)$ and $r_{t+1} = \mathcal{R}(s_t, a_t)$. Consequently, for fixed transition and reward functions, a trajectory is fully specified by its actions and the initial state.

The primary objective of RL agents lies within finding sequences of actions s.t. the accumulated reward along the resulting trajectories is maximized. For the process of iteratively choosing suitable actions, they use their stochastic or deterministic *policy*. Due to this possible policy stochasticity, it is useful to consider trajectories of infinite length that are made up of random variables representing states actions and rewards rather than concrete realizations thereof. Random actions arise as a consequence of stochastic policies and since states and rewards depend on actions they thus also become random variables. We use capital letters S_t, A_t and R_t to denote random states, actions and rewards.

Definition 4. Given an MDP as in Definition 3, a *policy* is a function $\pi : \mathcal{S} \times \mathcal{A} \mapsto [0, 1]$ mapping state action pairs to probabilities s.t. $\sum_{a \in \mathcal{A}} \pi(s, a) = 1$ for all $s \in \mathcal{S}$. If $\max_{a \in \mathcal{A}} \{\pi(s, a)\} = 1$ also holds, the policy is called *deterministic* and otherwise *stochastic*.

We will abuse notation and interpret policies as conditional probabilities $\pi(a \mid s)$ for choosing action a under the condition of currently observing state s. In case of deterministic policies we will sometimes write $\pi(s) = a$ instead of $\pi(a \mid s) = 1$.

At first glance it might be confusing as to why we are considering stochastic policies, but this is simply due to the fact that they come with some desirable theoretical properties. They simplify notation, can be represented by compact end-to-end differentiable functions, and are a natural source of exploration. Due to the latter property, we always make use of stochastic policies when training RL agents in some way. The goal of training an RL agent is to find optimal policies.

Our formal maximization criterion for policies is given by the expected return when starting in the initial state. The expected return of a policy conditioned on some starting state is also known as the *state value*.

Definition 5. The *return* from step t onward is given by the discounted sum of rewards $G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$.

Note that in the above Definition the return is also a random variable. If we consider the concrete return of some trajectory, we write g_t instead. The discount factor $\gamma \in [0, 1]$ is a hyperparameter used to balance short-term and long-term rewards. Values $\gamma < 1$ also guarantee that the return is always finite, but note that they are not strictly necessary in our case, because we only work with episodic MDPs, i.e., trajectories are guaranteed to be of finite length. Their exact length however is still a random quantity, so from a notational point of view it is convenient to model trajectories as infinite sequences of random rewards.

Definition 6. The *state-value* function for following policy π in some state s is defined as $v_\pi(s) = \mathbb{E}_\pi [G_t \mid S_t = s]$

Definition 7. A policy denoted by π^* is *optimal* iff $\pi^* \in \arg\max_\pi v_\pi(s_0)$.

Another quantity that is especially useful when searching for optimal policies are *state-action-values*. Given some policy π and a state-action pair (s, a), the action value of a in state s is the expected return of carrying out a immediately under the assumption

that all subsequent actions follow π. Similarly to the state-value function we define the state-action-value function as

$$q_\pi(s, a) = \mathbb{E}_\pi \left[G_t \mid S_t = s, A_t = a \right]. \tag{4.1}$$

The state-value and state-action-value function can be related by

$$q_\pi(s, a) = \mathcal{R}(s, a) + \gamma v_\pi(s'), \tag{4.2}$$

where $s' = \mathcal{T}(s, a)$ is the successor of s under action a.

4.1.2 Deep Reinforcement Learning Algorithms

Trial and error in the context of RL means that an agent deploys his policy in the environment and uses the trajectory data obtained in this way to find an estimate for an optimal policy. This estimate can be direct or it can be inferred indirectly from estimates of the optimal state-action-values. Roughly speaking, a reinforcement learning algorithm defines

- which policy is used for exploration, i.e., for collecting trajectories that help the agent to learn,
- which functions need to be estimated for the purpose of learning an optimal policy, and
- how the collected trajectory data is used to optimize the corresponding function estimators.

Deep reinforcement learning algorithms all share the same property that they use deep neural networks as function approximators.

The classic, non-deep approach for representing estimates of learned value functions (or policies) is to explicitly store all values in a tabular form, which is theoretically possible as state and action space are usually of finite size, or can be discretized. This approach can work for toy problems, but is often practically infeasible, because the number of possible states can quickly become very large, even for seemingly simple problems.

For such MDPs that have a large state space and also complex transition and reward structures, RL algorithms thus require the use of sufficiently accurate functional approximators to represent value functions or policies in order to work properly.

In recent years, deep neural networks have shown to be an excellent choice for this purpose [23] [28], giving rise to the field of DRL. When compared to the prohibitive memory requirements of tabular representations, they offer the advantage of being significantly more compact. Furthermore they are also better at generalizing when compared to more simple approximators such as parametrized linear functions. As has been shown by many other works [2] [20] [35], the generality of neural networks also allows us to design them in such a way that they can deal with situations where the size of the input, for RL $s \in \mathcal{S}$ or $a \in \mathcal{A}$, is variable. The fact that different assignment

problems may lead to this exact kind of situation, is another reason that makes them a good choice for this thesis.

We will now present the two particular algorithms used in this thesis which are the DQN algorithm [23] and a simple Actor-Critic algorithm similar in spirit to A2C [22]. Both algorithms use deep neural networks as functional approximators. For now, we treat neural networks as abstract, parametrized functions $f(x;\theta)$ differentiable w.r.t. to their parameters θ.

Deep Q-learning

Despite the fact that stochasticity is a desirable property during training, we are searching for optimal policies, some of which are guaranteed to be deterministic. One way to characterize optimal policies is as policies that act greedily upon the optimal state-action-value function $q^* = q_{\pi^*}$. Given some function Q that can be either a true state-action value function, or any estimator thereof, the policy π_Q that acts greedily w.r.t. Q is a deterministic policy given by

$$\pi_Q(a \mid s) = \begin{cases} 1 & a = \arg\max_{a'} Q(s,a') \\ 0 & \text{else} \end{cases}. \tag{4.3}$$

The optimal policy is obviously the same as the policy that acts greedily w.r.t. q^*. The central idea behind Q-learning [32] is to derive approximately optimal policies from learned estimates of q^*.

The core result underlying the actual Q-learning algorithm is the *bellman optimality equation* [29] [32] . It states that the optimal Q-function can be rewritten as

$$q^*(s,a) = \mathbb{E}_{\pi^*}\left[R_t + \gamma \max_{a'} q^*(S_{t+1}, a') \mid S_t = s, A_t = a\right].$$

We can use the right side of this equation to bootstrap target Q-values which then allow us to calculate *temporal difference (TD) errors*. The information provided by TD-errors can finally be used to update the estimator Q. The TD-error for some experience tuple (s_t, a_t, r_t, s_{t+1}), that captures exactly one environment step, is given by

$$\delta_t = Q(s_t, a_t) - (r_t + \gamma \max_{a'} Q(s_{t+1}, a')).$$

The considered experience tuples can be generated by following the greedy policy w.r.t. to the current estimate Q, but in practice it is often better to deploy a stochastic ϵ-greedy policy instead. These policies are parametrized by a random exploration coefficient $\epsilon \in (0,1)$ and choose random actions with probability ϵ and act greedily with probability $1-\epsilon$ [29]. Formally they can be defined as

$$\pi_{\epsilon,Q}(s,a) = \begin{cases} \frac{\epsilon}{|\mathcal{A}|} + (1-\epsilon) & a = \arg\max_{a'} Q(s,a') \\ \frac{\epsilon}{|\mathcal{A}|} & \text{else} \end{cases}.$$

This kind of policy is beneficial, because choosing probabilities at random is a form of (undirected) exploration. Large and small values of ϵ allow to balance between exploration and exploitation. Since the policy is uninformed when learning starts and gradually becomes more informed as training goes on, it is a good choice to start off with a large ϵ that decreases over time, since this enables the agent to better exploit the knowledge gained through early exploration.

RL agents that explore greedily according only to their initially uninformed estimates are more likely to converge to suboptimal policies [29]. This happens because without exploration, it becomes less likely that they observe trajectories that can be used to infer optimal behavior.

A naive implementation of deep Q-learning, i.e., a version where Q is given by a neural network $Q(\cdot;\theta)$ with parameters θ, would collect the experiences (s_t, a_t, r_t, s_{t+1}) and learn from them on-policy. That means the TD-errors are calculated immediately when encountering experiences, or at the end of the respective episode, and then discarded. These TD-Errors would then be used to estimate

$$\mathcal{L}(\theta) = \mathbb{E}_{(s,a,r,s')\sim\pi_{Q_\theta}}\left[(Q_\theta(s,a) - y)^2\right] \tag{4.4}$$

$$\text{where} \qquad y = r + \max_{a'} Q_\theta(s', a'). \tag{4.5}$$

and the network parameters could be optimized by stochastic gradient descent (see Section 4.2.4) on $\nabla_\theta \mathcal{L}(\theta)$. This approach using the loss as formulated above is however subject to two key problems.

First of all, learning on-policy means that the experiences used for learning are highly temporally correlated, making the estimates of 4.4 and thus also the corresponding gradients subject to a large amount of variance. This typically causes the learning (parameter optimization) process to become unstable.

Secondly, the TD-target (4.5) is bootstrapped from the current estimator Q_θ, so updating Q_θ also moves the target. This interdependency between target and estimator can lead to severe oscillations of θ and additional learning instability.

The central contributions of the DQN algorithm, *experience replay* and *fixed Q-targets*, specifically tackle these two issues and enable a more stable learning process. DQN trains an estimator Q_θ with parameters θ by minibatch stochastic gradient descent on the DQN loss

$$\mathcal{L}(\theta) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{U}(B)}\left[(Q(s,a;\theta) - y)^2\right] \tag{4.6}$$

$$\text{where} \qquad y = r + \max_{a'} Q(s', a'; \theta^-). \tag{4.7}$$

Experience replay denotes the fact that the experience tuples are no longer used directly, but instead stored in a FIFO buffer B of fixed size from which random experience batches are sampled during learning. This breaks the temporal correlation among experiences and thus decreases the gradient variance. Additionally, the method can be more sample efficient. If the update frequency, i.e., the number of steps between SGD

updates, is smaller than the batch size, we effectively use some experiences multiple times before discarding them.

Fixing the Q-target means that we store an additional set of parameters θ^- which are not immediately affected by gradient descent updates and use the corresponding estimator Q_{θ^-} for calculating the targets y. This decouples the target from the update process of θ and thus also makes training more stable. In the original DQN algorithm, θ^- is a periodic copy of θ, i.e., every $N = 10000$ environment steps, θ^- is set to equal θ, but between these target parameter updates, θ^- does not change. We use exponentially moving targets instead, i.e., instead of $N \in \mathbb{N}$ we use a hyperparameter $\tau \in (0, 1)$ and update the target parameters after each SGD update as $\theta^- = (1 - \tau)\theta^- + \tau\theta$. This way the target is not completely decoupled, but for values $\tau << 1$ the influence is still small enough to prevent signifcant additional instability. Pseudo-code for our implementation of DQN is shown in Algorithm 1.

Algorithm 1 DQN Algorithm

Input: Markov Decision Process $\langle \mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{T}, s_0, \mathcal{S}^* \rangle$
Input: Number of training episodes N
Input: Initial and final epsilon ϵ_0 and ϵ_{end}
Input: Epsilon decay $\beta \in (0, 1)$
Input: Replay Buffer B (initially empty)
Input: Target parameter temperature $\tau \in (0, 1)$
Input: Number of env. steps between parameter updates K
Initialize online network Q_θ with parameters θ
Initialite target network Q_θ^- with $\theta^- \leftarrow \theta$
$\epsilon \leftarrow \epsilon_0$
for k $= 1, \ldots, N$ episodes **do**
    $t \leftarrow 0$
    $s \leftarrow s_0$
    repeat
        $a \sim \pi_{\epsilon, Q_\theta}(\cdot \mid s)$
        $r, s' \leftarrow \mathcal{R}(s, a), \mathcal{T}(s, a)$
        Store (s, a, r, s') in B
        $t \leftarrow t + 1$
        if t mod K $= 0$ **then**
            Uniformly sample a batch of $(s, a, r, s') \in B$
            Estimate $\mathcal{L}(\theta)$ (4.6) using that batch
            Update θ by SGD using $\nabla_\theta \mathcal{L}(\theta)$
            $\theta^- \leftarrow (1 - \tau)\theta^- + \tau\theta$
        $s \leftarrow s'$
    until $s \in \mathcal{S}^*$
    $\epsilon \leftarrow \max(\beta \cdot \epsilon,\ \epsilon_{end})$
return θ

Policy Gradient Methods

The only strong requirement for applying the policy gradient approach is a parametrized policy $\pi(\cdot;\theta)$ differentiable w.r.t. its parameters θ. To shorten notation we will simply write π_θ. Neural networks which output values that can be interpreted as action probabilities are a natural fit for this kind of algorithm. Given such a policy network one then tries to find an estimate of the policy gradient

$$\nabla_\theta J(\theta) \tag{4.8}$$

where $J(\theta)$ is a numeric measure for the performance of the policy π_θ. These estimates can then be used to update θ through any form of gradient ascent (see Section 4.2.4).

In our case, we are given episodic MDPs with initial state s_0 and are interested in policies that are optimal w.r.t. Definition 7, so $J(\theta) = v_{\pi_\theta}(s_0)$.

The policy gradient theorem [30] is fundamental to the method we are going to use for estimating (4.8). In the episodic case it states that

$$\nabla_\theta v_{\pi_\theta}(s_0) \propto \mathbb{E}_{\pi_\theta}\left[\sum_a q_{\pi_\theta}(S_t, a)\nabla_\theta \pi_\theta(a \mid S_t)\right] \tag{4.9}$$

where the constant of proportionality depends on episode length [29]. The intuitive interpretation to this form of the policy gradient is that it in some state s, it encourages updates to θ that increase the probability of those actions a with relatively high state-action value $q_{\pi_\theta}(s, a)$.

Note that if we update θ by gradient ascent using an unbiased estimate of 4.9, then π_θ is guaranteed to converge to a locally optimal solution [29]. One way for obtaining such estimates is to replace the expectation over π_θ by one or multiple trajectories

$$\tau = \langle s_0, a_0, r_1, s_1, \ldots, s_{T-1}, a_{T-1}, r_T, s_T \rangle,$$

that are generated on-line by following π_θ. The state-action values can also be estimated directly from the same trajectory data. Without going into detail yet we denote these estimates as $Q_{\pi_\theta}(s, a, \tau)$ for now. Another trick that is usually employed to improve numerical stability is to rewrite the sum over actions within the policy gradient as an expectation using the identity $\frac{d\log(f(x))}{dx} = \frac{1}{f(x)}\frac{df(x)}{dx}$,i.e.,

$$\sum_a q_{\pi_\theta}(S_t, a)\nabla_\theta \pi_\theta(a \mid S_t) \tag{4.10}$$

$$= \sum_a q_{\pi_\theta}(S_t, a)\frac{\pi_\theta(a \mid S_t)}{\pi_\theta(a \mid S_t)}\nabla_\theta \pi_\theta(a \mid S_t) \tag{4.11}$$

$$= \mathbb{E}_{a\sim\pi_\theta}\left[q_{\pi_\theta}(S_t, a)\frac{1}{\pi_\theta(a \mid S_t)}\nabla_\theta \pi_\theta(a \mid S_t)\right] \tag{4.12}$$

$$= \mathbb{E}_{a\sim\pi_\theta}\left[q_{\pi_\theta}(S_t, a)\nabla_\theta \log \pi_\theta(a \mid S_t)\right]. \tag{4.13}$$

Putting all of this together, we finally obtain estimates of the form

$$\nabla_\theta v_{\pi_\theta}(s_0) \approx \frac{1}{T} \sum_{t=0}^{T-1} Q_{\pi_\theta}(s_t, a_t, \tau) \nabla_\theta \log \pi_\theta(a_t \mid s_t). \tag{4.14}$$

Since we did not yet specify the way in which the estimates $Q_{\pi_\theta}(s_t, a_t, \tau)$ are calculated, we obtain the following template algorithm.

Algorithm 2 Template Policy Gradient Algorithm

Input: Learning rate α
Initialize policy parameters θ
for i=1, ..., N episodes **do**
  Sample a trajectory $\tau \sim \pi_\theta$ of length T
  Calculate $Q_{\pi_\theta}(s_t, a_t, \tau)$ for all t
  $\Delta_\theta \leftarrow \frac{1}{T} \sum_{t=0}^{T-1} Q_{\pi_\theta}(s_t, a_t, \tau) \nabla_\theta \log \pi_\theta(a_t \mid s_t)$
  $\theta' \leftarrow \theta + \alpha \Delta_\theta$
  Optionally: Update estimators for Q_{π_θ}
  $\theta \leftarrow \theta'$
return θ

The classic instantiation of this, the REINFORCE algorithm [33], fills out $Q_{\pi_\theta}(s_t, a_t, \tau)$ with parameter-free Monte-Carlo estimates

$$Q_{\pi_\theta}(s_t, a_t, \tau) = g_t, \tag{4.15}$$

using the observed return of τ from time step t onward. This is an unbiased estimate, but at the same time a highly variable one, because at every step of τ, we sample a random action from our stochastic policy π_θ. This usually causes REINFORCE agents to be very diffcult to train, as gradients will also be subject to a prohibitively large amount of variance.

A remedy to this problem is achieved by subtracting a *baseline* $b(s_t)$ from the Q estimates that does not depend on actions in any way, s.t. the expectation remains unchanged [29]. This can give rise to an estimate with drastically reduced variance. The literature suggest that a good choice for b are learned state-value functions $V_{\pi_\theta}(\cdot; \psi)$ [22], also represented by a deep neural network with parameters ψ. The result of the corresponding difference

$$A\pi_\theta(s_t, a_t, \tau; \psi) = g_t - V_{\pi_\theta}(s_t; \psi) \tag{4.16}$$

is then an estimate of the action-advantage value $a_\pi(s_t, a_t) = q_\pi(s_t, a_t) - v_\pi(s_t)$. This kind of approach combines policy gradient methods with value-based methods and is usually referred to as an advantage *Actor-Critic*.

This name derives from the interpretation that instead of using a feedback signal based directly on trajectory data only, they use the learned value function to estimate action advantage values that are used for criticizing the actors policy π_θ.

For learning the state-values, the Actor-Critics typically use the same trajectory data τ that is also used to fit the policy. The objective for the value function critic is to minimize the loss

$$\mathcal{L}_v(\psi) = \mathbb{E}_{\pi_\theta}\left[(G_t - V_{\pi_\theta}(S_t;\psi))^2\right],$$

which can be done by SGD using the observed returns g_t and states s_t of the sampled trajectories τ.

Modern Actor-Critic architectures are often constructed s.t. some of the first layers among the networks for $V_{\pi_\theta}(\cdot;\psi)$ and $\pi(\cdot;\theta)$ share parameters which are then optimized jointly [22].

For this purpose, we can reformulate the term in 4.13 as a loss

$$\mathcal{L}(\theta) = -\,\mathbb{E}_{a\sim\pi_\theta}\Big[q_{\pi_\theta}(S_t,a)\log\pi_\theta(a \mid S_t)\Big], \tag{4.17}$$

by changing its sign and dropping the gradient operator.

The final policy loss, $\mathcal{L}_\pi(\theta)$, the Actor is trained to minimize also includes an exploration bonus based on the policies entropy $H(\pi_\theta) = \mathbb{E}_{\pi_\theta}\left[-log(\pi(A_t \mid S_t))\right]$. This leads to the formulation of the policy loss as

$$\mathcal{L}_\pi(\theta) = \mathcal{L}(\theta) - \beta H(\pi_\theta), \tag{4.18}$$

where β is a hyperparameter that controls the influence of the exploration bonus. Because we aim to minimize this loss, subtracting $\beta H(\pi_\theta)$ from the other loss can be understood as an exploration bonus, as policies with larger entropy are more stochastic than others and thus also more likely to explore their environment.

When estimating the gradient of $\mathcal{L}_\pi(\theta)$, we use the estimate in 4.14 for the gradient of $\mathcal{L}_\pi(\theta)$, and $\nabla_\theta\beta H(\pi_\theta)$ for the remaining term.

Finally, given $\nabla_\theta\mathcal{L}_\theta$ and $\nabla_\psi\mathcal{L}_v$, we apply the SGD updates jointly, i.e., the intersection of ψ and θ, that arises as a consequence of sharing initial layers, is affected by the linear combination of both above gradients over their intersecting elements.

The reasoning behind sharing initial layers for the Actor and Critic network is that they will be used for feature learning more effectively. If the Actor and Critic networks were completely disjoint, both would be forced to learn features on their own.

The final Actor-Critic approach we will be using in our experiments when training policy based DRL agents is shown in Algorithm 3.

4.2 Deep Learning

The art of deep learning usually involves choosing suitable neural network modules and stacking them together in such a way that introduces some of the prior knowledge we have about the problems at hand into resulting architectures. In the following subsections

Algorithm 3 Actor Critic Algorithm

Input: Learning rate α
Input: Entropy influence β
Initialize policy and value parameters θ ψ.
for i=1, ..., N episodes **do**
  Sample a trajectory $\tau \sim \pi_\theta$ of length T
  Compute state value estimates $V(s_t; \psi)$ for all $t = 0, \ldots, T-1$
  Use observed policy entropy averaged over all steps in τ as estimate $\hat{H}(\pi_\theta)$
  $\Delta_\theta \leftarrow -\left(\frac{1}{T}\sum_{t=0}^{T-1}(g_t - V(s_t;\psi))\nabla_\theta \log \pi_\theta(a_t \mid s_t)\right) - \nabla_\theta \beta \hat{H}(\pi_\theta)$
  $\Delta_\psi \leftarrow \frac{1}{T}\sum_{t=0}^{T-1}\nabla_\psi (g_t - V(s_t;\psi))^2$
  Update θ and ψ using Δ_θ and Δ_ψ by SGD with learning rate α
return θ

we will introduce all deep learning methods we use for designing and training neural networks as function approximators for the DRL version of the considered assignment problems.

We first go over the functional form of several different types of neural networks modules, as well as the ideas behind them, and then briefly explain how gradients are used to carry out parameter updates by gradient based optimization techniques.

4.2.1 Feed-forward Networks and Activation Functions

Functional Form of a FFN

A fully-connected feed-forward network (FFN) of depth n with d_{in} input units and d_{out} output units can be written as a function FFN : $\mathbb{R}^{d_{in}} \mapsto \mathbb{R}^{d_{out}}$ that arises as the composition

$$\text{FFN}(x) = f_n(f_{n-1}(\ldots f_1(x)\ldots)), \tag{4.19}$$

$$\text{where} \quad f_i(z) = h_i(W_i^T z + b_i), \tag{4.20}$$

of n transformations (also called layers) $f_1, ..., f_n$, which are mostly non-linear, depending on their activation functions $h_1, ..., h_n$. The FFN parameters, the entirety of the W_i and b_i, are called weight matrices and biases where the following holds

$$W_1 \in \mathbb{R}^{d_{in} \times d_1} \tag{4.21}$$

$$W_i \in \mathbb{R}^{d_{i-1} \times d_i}, \qquad i = 2, ..., n-1 \tag{4.22}$$

$$b_i \in \mathbb{R}^{d_i} \qquad i = 1, ..., n-1 \tag{4.23}$$

$$W_n \in \mathbb{R}^{d_{n-1} \times d_{out}}, b_n \in \mathbb{R}^{d_{out}}. \tag{4.24}$$

The layer width hyperparameters $d_1, ..., d_{n-1}$ can be chosen freely, but for the sake of simplicity we restrict our work to cases where $d_1 = ... = d_{n-1} = d$ and call d the *hidden size* of the network. Additionally, we treat $n = 1$ as a special case where $d_1 = d_{out}$ and

$d_{n-1} = d_{in}$. We call f_n the output layer and $f_1, \ldots, f_{n-1}$ hidden layers. Also note that often the activation function for the output layer f_n is chosen differently than for the hidden layers. The biases b_i are optional, but if not stated otherwise, we will include them.

Feed-forward networks can be applied on their own or serve as a modular building block for more complex network types. If they are used on their own, then their input and output dimension d_{in} and d_{out} are usually determined by the problem requirements, e.g., state representation size and number of actions in reinforcement learning. If used in combination with other networks they are often hyperparameters, or depend on the exact context.

Activation Functions

Activation functions do not alter the shape of their input. They only transform it in ways that help networks to learn, or in the case of the output layer, they transform their input s.t. the network output range matches any possible requirements. Most activation functions are of the form $h : \mathbb{R} \mapsto \mathbb{R}$ and their application to multi-element objects such as vectors, matrices or tensors is carried out element-wise.

For hidden layers, we almost exclusively use the ReLU activation function [24] which is defined as $\text{ReLU}(x) = max(0, x)$. Using non-linear activation functions such as ReLU is important, because otherwise, when $h(x)$ is linear in x, FFNs could only be used for approximating linear functions.

Another activation function that is important to our approach is the softmax function. It maps inputs $x \in \mathbb{R}^n$ to outputs that can be interpreted as probabilities and is defined as

$$\text{softmax}(x)_i = \frac{e^{x_i}}{\sum_{i=1}^{n} e^{x_i}} \quad i = 1, \ldots, n. \tag{4.25}$$

We mainly use it as the activation function for the output layer of policy networks, s.t. their outputs can be interpreted as probabilities. Due to the exponentiation, this activation function also allows the resulting stochastic policy to better approach deterministic ones [29].

Finally, we also make use of layer normalization units [1], which can also be viewed as a special form of activation, that can be added on top of the other activation functions. For an input $x \in \mathbb{R}^d$, layer normalization is defined as

$$\text{LayerNorm}(x)_i = \frac{(x_i - \bar{x})}{\hat{\sigma} + \epsilon} \cdot \gamma + \beta \quad i = 1, \ldots, n, \tag{4.26}$$

where $\cdot$ is elementwise multiplication, $\bar{x} = \frac{1}{d}\sum_{i=1}^{d} x_i$ is the sample mean over all features of x and $\hat{\sigma} = \sqrt{\frac{1}{d}\sum_{i=1}^{d}(x_i - \bar{x})^2}$ the respective (biased) sample standard deviation. For each FFN layer f_i, this module may be applied just before the activation h_i, with the idea in mind that this makes the linear transformation $W_i^T z + b_i$ invariant to re-scaling and re-centering of the weight matrix W_i [1]. Introducing these invariances eases the

optimization process which empirically often leads to significantly reduced training times and better final results [1]. $\gamma \in \mathbb{R}^d$ and $\beta \in \mathbb{R}^d$ are learnable parameters that allow the layer to re-scale and shift the output after normalization.

4.2.2 Recurrent Graph Neural Networks

Applying deep feed-forward networks on their own can work very well for unstructured data, e.g., if the input consists of p structurally unrelated features that are represented by real numbers or one-hot encodings. A typical example for data that does not follow such an underlying structure, where plain FFNs are known to fail, are e.g. grayscale images.

In theory, one could flatten the images and interpret the pixel gray values as independent features of the input. Assuming images of width w and height h, a FFN with $d_{in} = wh$ input neurons could then be used to process the images. Practically, this approach completely fails to capture the underlying spatial relations between pixels and introduces unnecessarily many redundant parameters, even for moderately sized images. The long standing state-of-the-art approach to image data are convolutional neural networks where a set of kernels is moved across the image. These kernels are small, usually square-shaped parameter matrices which are repeatedly applied to patches of the image via pointwise aggregation. This enables the model to learn meaningful aggregations of spatially close pixels, ultimately leading to the automatic extraction of features such as edges and other structures present in images.

In our case, the information that defines an assignment problem also follows a clear structure with a lot of redundancy that is however more complex than the spatial structure of an image. As we already mentioned in Chapter 3, bipartite graphs provide a very natural framework for describing this information in a structured manner.

The recent and ongoing boom within the deep learning community has given rise to a variety of neural network architectures designed specifically for processing graph-like data [35]. These networks usually implement forms of message passing among nodes, i.e., nodes use some kind of neural network to transmit information across edges to their neighbor nodes. Carrying out mutliple steps of message passing means that information from node is able to travel beyond the node's immediate neighbours.

Among these graph neural networks one can distinguish between recurrent graph neural networks and graph convolutional neural networks [35]. Closely following related work about learning to solve graph combinatorial optimization problems [18] (also see Chapter 2), we will focus on recurrent graph neural networks. These use the same set of parameters for each step of message passing, while graph convolutional neural networks use a different set of parameters during each message passing step.

Consider graphs $G = (V, E)$ with vertices $v \in V$ and per-vertice features $x_v \in \mathbb{R}^{d_v}$, as well as edges $\{u, v\} \in E$ with per-edge features (weights) $x_{u,v} \in \mathbb{R}^{d_e}$. We use graph neural networks for the purpose of generating latent vertice embeddings $\eta_v \in \mathbb{R}^d$, i.e., discriminative feature vectors of fixed size d. They should capture the given vertice

features within the structural context of the graph and also incorporate the weights into this calculation. Assuming some common initial embedding $\eta_v^0 \in \mathbb{R}^d$, they use the edge information to update these embeddings via learned transformations and aggregations over vertice neighbourhoods. Iterating this update process multiple times constitutes a form of message passing between nodes. To formalize this update process we define embedding rules as networks $F : \mathbb{R}^d \mapsto \mathbb{R}^d$ with

$$F(\eta_v^i) = f\left(\eta_v^i + f_v(x_v) + \mu\left(\left\{\eta_u^i \mid u \in \mathcal{N}(v)\right\}\right) + f_e\left(\sum_{u \in \mathcal{N}(v)} x_{u,v}\right)\right), \tag{4.27}$$

where $f : \mathbb{R}^d \mapsto \mathbb{R}^d$, $f_x : \mathbb{R}^{d_v} \mapsto \mathbb{R}^d$ and $f_e : \mathbb{R}^{d_e} \mapsto \mathbb{R}^d$ are feed-forward networks and μ is an aggregation function that maps the set of neighbour embeddings back into the original embedding space and thus implements the message passing from a vertices neighbourhood into the vertice itself. Note that μ is typically also a parameterized neural network, for example

$$\mu\left(\left\{\eta_u^i \mid u \in \mathcal{N}(v)\right\}\right) = f_\mu\left(\sum_{u \in \mathcal{N}(v)} \eta_u^i\right), \tag{4.28}$$

where f_μ is a FFN of depth 1 with linear activation and no bias [18]. The updated embeddings are then simply obtained by $\eta_v^{i+1} = F(\eta_v^i)$. The final embedding of a node v is given by η_v^U, where $U \in \mathbb{N}^+$, the number of times this update is carried out, is a hyperparameter.

In classical graph combinatorial problems such as finding a minimal vertex cover, one embedding rule for the whole graph is sufficient [18], as there is no further distinction to be made among vertices and how they relate. We will however be confronted with a bipartite graph, where vertices as well as edges can have different interpretations, i.e., student embeddings need to be treated differently than course embeddings. Thus, we will require two instead of one update rules to build a suitable graph neural network. This problem is treated in Section 5.6.1.

4.2.3 Neural Attention

Another kind of structured data that is treated by the deep learning community in ways we can draw inspiration from are natural language sentences. They can be viewed as sequences of words, where the words can also be represented by learned embeddings.

Historically, encoder-decoder [4] architectures that make use of two separate long-short-term-memory networks [15] or gated recurrent units [5] were the favored approach for processing such sequences of word embeddings, but recently attention-based approaches, more specifically transformer architectures [31] have gained a lot of popularity.

In the context of natural language processing, transformers also apply special positional encodings to their input s.t. the word order is preserved. This is due to the fact that neural attention operations are permutationally invariant w.r.t. to their input.

Mainly because of this property, they have been shown to also be a good choice for processing set-structured data [21], i.e., sets of embeddings that describe one unique kind of object. As such, they are also an interesting choice for our setting, since the requests of assignment problems can be formalized as sets of student-course tuples.

The specific form of attention we will be using in all our attention modules is *scaled dot attention* [31].

Definition 8. Given n_q queries, and n_k key-value pairs, $Q \in \mathbb{R}^{n_q \times d}, K \in \mathbb{R}^{n_k \times d}, V \in \mathbb{R}^{n_k \times d}$, scaled dot attention is defined as

$$\text{Att}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right) V \in \mathbb{R}^{n_q \times d}.$$

The division by $\sqrt{d}$ is element-wise and the softmax is applied row-wise, instead of over the whole matrix $QK^T \in \mathbb{R}^{n_q \times n_k}$.

Note that this is a parameter-free function, so it is better to think of it as a special activation function rather than a neural network module. The output of the attention operation is again a matrix in $\mathbb{R}^{n_q \times d}$. Each row of this matrix corresponds to exactly one query and is a convex combination of all values. The weight of the ith value (ith row in V) in this combination depends solely on the similarity of the particular query with the value's key (ith row in K), whereby similarity is being measured by their dot product. These value weights as contained in QK^T are sometimes also called attentions scores. Intuitively $(QK^T)_{i,j}$ gives the amount of attention paid to the key-value pair j when considering query i.

The actual attention-based neural network module we mainly use are multi-headed attention blocks (MAB) [21]. They essentially parameterize this operation and distribute attention across multiple attention heads. This can be compared to the fact that convolutional neural networks usually use not just one kernel for feature learning, but a whole array of kernels.

Definition 9. Given queries, keys and values, $Q \in \mathbb{R}^{n_q \times d}, K \in \mathbb{R}^{n_k \times d}, V \in \mathbb{R}^{n_k \times d}$, the output of the *multi-headed attention block* with h heads is defined as

$$\begin{aligned} \text{MAB}(Q, K) &= \text{LayerNorm}(H + \text{FFN}(H)) \\ \text{where} \quad H &= \text{LayerNorm}(Q + \text{MHA}(Q, K, V)), \end{aligned}$$

and MHA is a multi-headed attention module with

$$\begin{aligned} \text{MHA}(Q, K, V) &= \mathit{concat}(A_1, \ldots, A_h) W^A \\ \text{where} \quad A_j &= \text{Att}(QW_j^Q, KW_j^K, VW_j^V). \end{aligned}$$

LayerNorm is defined in Section 4.2.1 and $\text{FFN} : \mathbb{R}^d \mapsto \mathbb{R}^d$ is a 2-layer feed-forward network with d input and d output units and ReLU activations. The MAB parameters are given by the parameters of the FFN and $W^Q_{1\ldots h} \in \mathbb{R}^{d \times d_p}, W^K_{1\ldots h} \in \mathbb{R}^{d \times d_p}, W^V_{1\ldots h} \in \mathbb{R}^{d \times d_p}, W^A \in \mathbb{R}^{d \times d}$, where $d_p = d/h$ and we require that d is evenly divisible by h s.t.

the concatenation of the A_j is in $\mathbb{R}^{n_q \times d}$.

Each „head“ of this module uses its respective parameter matrices W^Q, W^K, and W^V to project the queries, keys and values into some low dimensional space, on which we then perform the actual attention operation. Intuitively, using multiple heads means that the module is enabled to more effectively pay attention to different kinds of relations of the queries and keys. W^A is then used to gather the results of each head into a final attention result by linearly transforming their concatenation. The sums inside the layer normalization units implement residual skip connections [13] which allow general information about the query to easily flow throughout the whole process of applying the module. Applying them transforms the operation into what can be considered an refinement of the queries with the attention results.

Given a matrix $Z \in \mathbb{R}^{n \times d}$ that stands for a set of n objects, each represented by a d-feature vector, we can use a MAB to embed the elements of this set into n refined embeddings (also of size d) by calculating $\mathrm{MAB}(Z, Z, Z)$.

These embeddings are able to capture interactions among the elements of the set and are de-facto invariant to row-permutations of Z. Rearranging the rows in Z means that the rows in the MAB output will still be the same, just rearranged in the same way as in the input.

A downside of MABs is that these modules are computationally quite expensive. For $\mathrm{MAB}(Z, Z, Z)$ we have $n_k = n_q = n$, so due to the matrix-matrix product present in Definition 8, the computational complexity of the MAB is $\mathcal{O}(n^2 d)$.

To tackle this issue, we can use an *induced set attention block* (ISAB) [21] instead.

Definition 10. Given a set $Z \in \mathbb{R}^{n \times p}$ of n vectors, *induced set attention blocks* come with *learnable parameters* $I \in \mathbb{R}^{k \times d}$ called *inducing points*, where k, the number of inducing points, is a hyperparameter. They use two separate MABs to perform the operation

$$\begin{aligned} \mathrm{ISAB}(Z) &= \mathrm{MAB}_2(Z, Y, Y) \in \mathbb{R}^{n \times d} \\ \text{where} \quad Y &= \mathrm{MAB}_1(I, Z, Z) \in \mathbb{R}^{k \times d} \end{aligned}$$

In order to achieve a lower run-time, an ISAB first essentially compresses the set into a representation $Y \in \mathbb{R}^{k \times d}$ where $k << n$ and then uses these as new keys and values for the final set attention operation. Empirically, ISABs tend to perform slightly worse than pure MABs [21], but their computational complexity compared to MABs is reduced to $\mathcal{O}(nkd)$. In our experiments we will fix $k = 8$.

In a similar way how ISABs use attention to compress a set, we can also use PMA modules to aggregate all elements of a set into a fixed size representation.

Definition 11. *Pooling by multi-headed attention* (PMA) modules are defined as

$$\mathrm{PMA}(Z) = \mathrm{MAB}(I, F(Z), F(Z)) \in \mathbb{R}^{1 \times d}$$

where $I \in \mathbb{R}^{1\times d}$ is a learnable parameter and $F : \mathbb{R}^d \mapsto \mathbb{R}^d$ is a 2-layer feed-forward network with ReLU activations.

4.2.4 Gradient Based Optimization

The most common technique for finding good neural network parameters is gradient based optimization. These methods make use of the fact that most types of neural networks are functions differentiable w.r.t. to their parameters θ. If we can formulate a differentiable *loss function* $\mathcal{L}(\theta)$ that becomes *smaller* as our idea of good performance increases, then we can optimize θ by iterative updates of the form

$$\theta \leftarrow \theta - \alpha \nabla_\theta \mathcal{L}(\theta), \tag{4.29}$$

until some termination criterion is reached. $\alpha \in (0, 1]$ is the step-size or learning-rate. If a loss function is not given, but instead an objective that should be maximized, we can turn the objective into a loss by simply changing its sign, which is equivalent to performing gradient ascent, i.e., adding gradients instead of subtracting them.

The intuition behind gradient-based approaches is that if θ determines a location $\mathcal{L}(\theta)$ on the surface of $\mathcal{L}$, then subtracting $\nabla_\theta \mathcal{L}(\theta)$ from θ can be thought of as moving downhill along the surface's direction of locally steepest descent, which is why these methods are often called gradient descent methods.

Note that usually we have to resort to estimates of $\mathcal{L}(\theta)$ that rely on random samples, because $\mathcal{L}$ often contains an outer expectation over some distribution (see for example the policy gradient estimator in 4.14). In that case the gradients will also be stochastic estimates, and thus the corresponding approach when relying on estimates is called *stochastic gradient descent* (SGD), or minibatch SGD if we use a sample size greater than one.

In this thesis we use the Adam optimizer [19] instead of plain SGD. Adam extends SGD by adding gradient momentum [26] and adaptive learning rates (second-order momentum) [7].

Using gradient momentum means that we keep track of an exponentially moving average over all gradient estimates and use that to update our parameters instead of the estimates directly. This accelerates the optimization process, as it helps to factor out noise present in gradient estimates and gradient accumulation leads to generally larger updates that still point in a reasonable direction[9].

Adaptive learning rates for each parameter are calculated by re-scaling the scalar learning rate with an exponentially moving estimate of the non-central second-order moment of the gradient w.r.t. to that parameter.

So, the learning rate actually used in the update is no longer a scalar, but a tensor that has the same size as the parameters, and their multiplication is carried out elementwise. This leads to smoothed updates, where smooth means that the update magnitudes of different parameters are adapted to be on a more similar level. Updates to parameters

with historically large gradients are scaled down, while updates to parameters with historically small gradients may be scaled up.

These exponentially moving estimates for the first and second-order moment are both biased. Adam can be distinguished from other momentum-based optimizers, by the fact that it also calculates two separate correction terms that are used to remove some of this bias.

5 Applying Deep Reinforcement Learning

In this chapter we finally present how we intend to use DRL algorithms for finding optimal solutions to instances of both simple and valued assignment problems.

We first reformulate the problems as MDPs and also discuss how we are going to treat the hard constraints of the assignment problem. The question how constraints are treated deserves special consideration, as MDPs do not come with the inherent ability to easily model constraints. Our solution consists in constraining policies in such a way that makes it impossible to end up in states that belong to the infeasible region.

Subsequently, we will discuss how we intend to find reasonable policies for MDPs in an appropriate amount of time. A naive approach would be to train a separate DRL agent for each distinct MDP. Since training DRL agents is however often a very time-consuming process, we also describe an alternative approach that allows us to essentially learn a meta-policy that can be applied to every MDP we construct, effectively eliminating the need to carry out an entire training process for each individual instance.

5.1 Constructing Markov Decision Processes

In order to be able to apply DRL to assignment problems, we need to reformulate them as sequential decision-making problems in the form of MDPs. Recall, that assignment problem instances represent bipartite graphs over students and courses and require feasible subsets of edges (requests) $X \subseteq R$ as solutions. Ideally, we would like to select $X = R$ (grant all requests), however, this solution is feasible only in pathological cases. Thus, we need to decide which edges to include and which to exclude. If we stipulate that these decisions are made sequentially on a per-edge basis, we can proceed in one of two ways. We can either start with $X = \emptyset$ and iteratively add single edges, until $\text{Open}_I(X)$ is empty (forward procedure), or, we can start with $X = R$ and iteratively remove edges, until we arrive at an assignment that is feasible (backward procedure).

In this thesis we wanted to focus one of these scenarios only and chose to work with MDPs that implement the forward procedure, because in real-world settings, the total course capacity is often significantly smaller than the total number of requests, so feasible assignments can grant only a small fraction of all requests. The data provided to us by the KAS confirms this assumption.

The reasoning behind this decision is that the number of action decisions and thus also the number of neural network evaluations to be made during the forward procedure will be significantly smaller than during the backward procedure. For DRL agents this means that the corresponding episodes will be significantly shorter, which ultimately allows for smaller training times.

A desirable property for assignment problem MDPs should be that optimal policies

lead to terminal states with a direct correspondence to optimal assignments. For an instance $I = \langle s, c, C, R \rangle$ of the simple assignment problem, a corresponding MDP $M_I = \langle \mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{T}, s_0, \mathcal{S}^* \rangle$ with this property is constructed such that

- $\mathcal{S} = 2^R$. States are given by all possible assignments.
- $s_0 = \emptyset$. Initially, assignments are empty.
- $\mathcal{A} = R$. Actions are given by the individual requests.
- $\mathcal{T}(X, r) = X \cup \{r\}$. Selecting a request as action adds it into the assignment $X \in \mathcal{S}$.
- $\mathcal{S}^* = \{X \in \mathcal{S} \mid \text{Open}_I(X) = \emptyset\}$. In other words, an assignment is terminal, if it is impossible to add further requests without harming constraints. That is, if all courses are full, or if open requests for non-full courses are only coming from students who already had 3 of their requests granted.
- $\mathcal{R}(X, r) = J^-(\mathcal{T}(X, r)) - J^-(X)$ where J^- is the penalized objective function of the simple assignment problem (3.6). This means, changing an assignment is rewarded with the resulting change in objective value. For example, if $\mathcal{T}(X, r)$ no longer violates some constraint that was previously violated by X, then this change generates a reward equal to the magnitude of the respective penalty. This reward structure encourages the satisfaction of as many constraints as possible before a terminal state is reached.

The construction for the MDP corresponding to a valued instance is analogous. The only thing that changes is that J is replaced with the penalized objective function of the valued assignment problem. Final assignments are generated from policies as the terminal states of their trajectories.

In the sections that follow we will use the terms assignment and state, and requests and actions synonymously.

5.2 Constrained Policies

Recall, that the penalized objective functions we used to construct a reward structure in the previous section are obtained by relaxing *only some of the constraints* (see Section 3.2). The relaxation did not include all of the constraints for a good reason. As a consequence, we need to ensure that policies cannot choose actions that lead to solutions which are infeasible w.r.t. to these constraints.

In our case this can be implemented in a straight forward way. During exploration and during inference with final policies, we alter the agents policy s.t. it never picks actions that are not in $\text{Open}_I(X)$ where X is the current state. This is achieved by either zeroing the respective probabilities, or, in the case of Q-learning, by setting the respective state-action values to minus infinity.

If assignments exists that are feasible w.r.t. to all constraints, then optimal policies

should also yield these assignments as terminal states, because they are maximizers of the penalized objective function (adding requests which lift constraint penalties heavily outweighs adding requests that do not help to „unviolate“ any additional constraints).

5.3 Single Instance vs. Multi Instance Learning

Standard RL as we described it in Section 4.1 does not come with a concept like that of a central problem description in the form of an objective function and constraints that can both be applied to a whole class of possible problem instances, as it is the case for combinatorial optimization problems. In RL, we usually just differentiate between separate environments and their respective MDPs.

Naively applying this paradigm, one could think of modeling each new instance of an assignment problem as an MDP and train an RL agent to find an optimal policy just for that particular instance. This *single-instance approach* is compellingly simple and can be applied directly without much effort, but it has several disadvantages.

A more sophisticated view embraces the distinction between problem description and instances and tries to integrate this notion into the RL framework. The core idea of our approach is that is better to train a single RL agent on multiple instances at once, instead of forcing the RL agent to start thinking from scratch for each new instance. More specifically, we train the agent on instances sampled from a distribution of our choice. We hypothesize that the agent will then be forced to learn a meta-policy that applies to any instance with similar properties.

The first obvious advantage of this approach is that it can be much less resource-heavy in the long run. Training an agent to solve just a single instance would entail solving that instance many times over and over again. Training an agent how to solve different instances from a distribution also requires solving many different instances over and over again, but once the agent is trained and has developed a generally applicable policy, solving a new instance is done much faster as learning to solve this instance from scratch.

Furthermore, learning to solve single instances without prior knowledge might even become practically infeasible due to time or even memory constraints, if they grow too large, whereas our proposed method might be able to generalize to larger instances. Solving an instance after training has finished actually requires significantly less memory than learning to solve it, because we no longer need to worry about tracking a computation graph and partial derivatives for the purpose of backpropagation.

We are expecting that a disadvantage of this approach should be that the quality of solutions decreases, as the instances become more dissimilar to the instances observed in the training distribution. When thinking about real world applications, this is however only a real disadvantage, if one has limited or no access at all to prior information about the instances the agent has to solve after training. If such prior information is available, the training distribution can be fit such that it accurately represents these specific types of instances.

5.4 Single Instance Network Architecture

When training an RL agent to solve a single instance only, we do not necessarily require a special network architecture. A rather simple approach which we also used in the early stages of this thesis makes use of a deep neural network $F : \mathbb{R}^{r+m} \mapsto \mathbb{R}^r$, where $r = |R|$ is the number of requests and m is the number of courses. Its input is given by the concatenation of a one-hot-encoding $x \in \{0, 1\}^r$ of the current assignment X and a vector $c_{rem} \in \mathbb{R}^m$ which contains the currently remaining capacity of each course divided by the course's total capacity. These features are not necessarily required, but we found that they speed up learning, so we included them.

When the task is to estimate Q values, we use a single, plain 4-layer FFN with hidden size $d = 64$ and linear activation in the output layer.

When estimating policies we need an additional critic network. In that case, the first few layers among actor and critic are shared, i.e. we use the output of a 2-layer FFN $G : \mathbb{R}^{r+m} \mapsto \mathbb{R}^d$ as the input for both the actor and critic network $F_\pi : \mathbb{R}^d \mapsto \mathbb{R}^r$ and $F_v : \mathbb{R}^d \mapsto \mathbb{R}$, which are again 2-layer FFNs, where F_π uses a softmax output activation and F_v a linear output activation. The concrete computations are

$$\begin{aligned} z &= G(\text{concat}(x, c_{rem})) \\ \pi(\cdot \mid X) &= F_\pi(z) \\ V_\pi(X) &= F_v(z) \end{aligned}$$

Note that such networks of the form $F : \mathbb{R}^{r+m} \mapsto \mathbb{R}^r$ cannot take on the role of a general solver for multi-instance training due to the fact that r and m are subject to change if we consider multiple different instance.

Nevertheless, we empirically compare these networks to the multi-instance networks derived in the following two sections in the single instance setting in our experiments in Section 6.1.

5.5 Formalizing the Multi Instance Objective

In single-instance learning, the objective was to find approximate solutions to

$$\arg\max_\pi v_\pi(s_0) \tag{5.1}$$

where s_0 and the value function are w.r.t. a concrete MDP $\langle \mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{T}, s_0, \mathcal{S}^* \rangle$. Optimizing a policy over a distribution of instances means that we now need to consider a distribution over MDPs $\mathcal{P}_{\mathfrak{M}}$. The objective changes to

$$\arg\max_\pi \mathbb{E}_{\langle \mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{T}, s_0, \mathcal{S}^* \rangle \sim P_{\mathfrak{M}}} \left[v_\pi(s_0) \right], \tag{5.2}$$

as an outer expectation over this distribution is added.

This formulation immediately reveals a central challenge posed by multi-instance training. In Section 4.1.1 we defined policies and Q-functions as functions of the form $\mathcal{S} \times \mathcal{A} \mapsto *$, but for differing MDPs, $\mathcal{S}$ and $\mathcal{A}$ will also be differing mathematical objects. So the term as stated above is technically not valid. We will solve this issue by replacing π with the parameters of a neural network that is expressive enough to take on any functional form $\mathcal{S} \times \mathcal{A} \mapsto *$ required by $\mathcal{P}_{\mathfrak{M}}$. Formally we can view such networks as cascaded functions $\mathfrak{M} \mapsto (\mathcal{S} \times \mathcal{A} \mapsto *)$, where $*$ is either $[0, 1]$ for policies or $\mathbb{R}$ for value functions and $\mathfrak{M}$ is the set of all MDPs supported by $\mathcal{P}_{\mathfrak{M}}$. Intuitively this implements exactly the mapping that is also peformed by the process of single-instance training. Given an MDP as input, this function has to return some valid policy as output.

How exactly we construct networks with this property is treated in the next Section (5.6). If these networks are used within an Actor-Critic context, policies can be extracted directly. If we choose to use DQNs estimating Q instead, we can use the greedy policies π_Q introduced in Section 4.1.2.

Assuming we are given such a network, we also need to modify the training process of the corresponding DRL agents to account for the new outer expectation over $\mathcal{P}_{\mathfrak{M}}$. As we have seen in Section 4.1.2, the training process of DRL agents usually takes places over a span of multiple episodes. Each episode starts out in the initial state of some fixed MDP and lasts until the trajectory obtained via the DRL agent's exploration policy ends in some terminal state. For the next episode, the MDP is again reset to its initial state. In order to estimate the expectation over the instance distribution, we also sample an entirely new MDP from $\mathcal{P}_{\mathfrak{M}}$ at the start of each episode.

5.6 Multi Instance Network Architecture

If the MDPs from some distribution $\mathcal{P}_{\mathfrak{M}}$ were totally unrelated, designing a network that is compatible with all of them would be significantly more difficult. In our case however, all possible MDPs are constructed using the same procedure, as all of them describe very similar problems that share an underlying structure. Our high-level solution takes advantages of this and is sketched in Figure 5.1.

It mainly uses the fact that any instance can be divided into its individual requests. So, if we are able to bring each request into a representation that

- carries sufficient information regarding the problem,
- is of fixed size, and
- can be processed with neural networks,

then we just have to find ways of dealing with different amounts of such representations. We will refer to them as request embeddings, or action embeddings, given the fact that requests and RL actions have a one to one correspondence.

The multi-instance network architecture can be separated into three distinct modules. A structure-aware encoder Φ, an aggregation module μ_X for generating state embeddings,

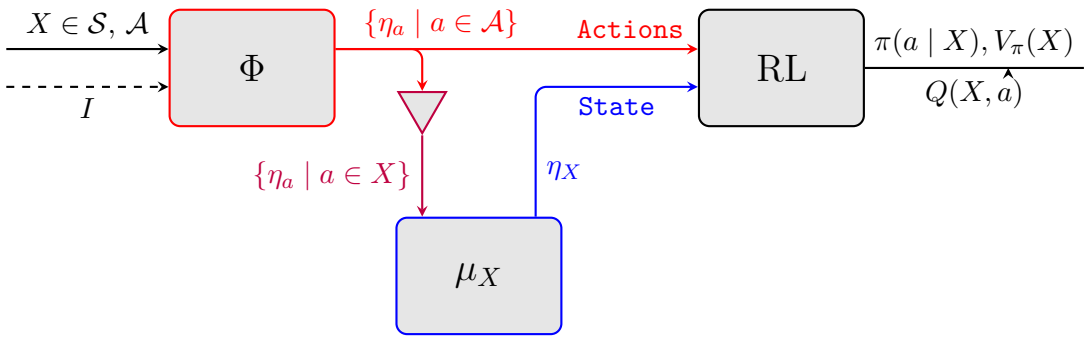

Figure 5.1: Multi Instance Architecture Overview

The first module, the encoder Φ takes care of producing action embeddings based on state X, actions $\mathcal{A}$ and static instance information I. An aggregator μ_X then produces a state embedding. Action and state embeddings are finally fed into either a policy module, or a state-action value module.

and a final policy (or alternatively Q-function) module.

5.6.1 Encoder Modules

Generally, the encoder module Φ receives the current state (an assignment X) and the MDP actions $\mathcal{A}$ (the set of requests) as input, which are already pre-encoded in a suitable way. We also include static information about course capacities and request values in the input, to account for the fact that these vary across different MDPs. For now, we summarize this information as a tuple I that is contains just the course capacities $I = \langle C \rangle$ or also request values $I = \langle C, V \rangle$, depending on whether we are working with simple or valued instances. For concrete instantiations of encoders, this information will be part of the state and action representations for X and $\mathcal{A}$.

The encoder output are $m = |\mathcal{A}|$ distinct embeddings $\eta_a \in \mathbb{R}^d$, one for each action (request), where d, the hidden dimensionality, is a hyperparameter. We also call this the agents *hidden size*, which we now fix to a global value of 64, i.e., all neural network modules we use in our architecture that come with a similar parameter for their internal dimensionality use the that exact same $d = 64$. Due to the large number of modules and algorithms, systematically searching for a good value of d is out of the scope of this thesis. For preliminary experiments, we used $d = 128$, but later it turned out, that $d = 64$ worked just as good, while allowing for shorter training times, so we decided to stick to this smaller value.

The encoder should operate in a way s.t. the request embeddings may encode the corresponding requests in the context of the problems structure. That means we ideally want Φ to not compute the same function for each $a \in \mathcal{A}$ but instead to make use of the fact that we can relate requests with one another, e.g., by using the fact that some requests come from the same student, or are posed for the same course.

Encoding with Recurrent Graph Neural Networks

Graph neural networks allow us to exploit the bipartite nature of assignments problems. We now construct a recurrent graph neural network (GNN) based encoder for valued assignment problem instances $I = \langle s, c, C, R, V \rangle$.

For this purpose, we reformulate instances as graphs $\langle V = s \cup c, E = R \rangle$, where course vertices are labeled with their capacities C, and edge labels (weights) are given by request values V. In order to be able to generate encodings that depend on some current assignment X, we consider X as a second set of edge labels, i.e., an edge (s_i, c_j) is additionally labeled with $x_{i,j} \in \{0, 1\}$ depending on whether X grants the respective request, or not.

Following the principles outlined in Section 4.2.2, we first assume initially zeroed student and course embeddings $\left\{\eta^0_{s_i} \mid s_i \in s\right\}$ and $\left\{\eta^0_{c_j} \mid c_j \in s\right\}$ all in $\mathbb{R}^d$. Our GNN encoder is then described by two separate embedding rules $F_s : \mathbb{R}^d \mapsto \mathbb{R}^d$ and $F_c : \mathbb{R}^d \mapsto \mathbb{R}^d$ (see 4.27), each parametrized by its own set of parameters.

We fix the FFNs used in the Definition of the embedding rule, f, f_v and f_e, to all be of depth 1 with ReLU activations. For the node aggregation functions μ_s and μ_c, we use MABs (see Section 4.2.3) in the following way

$$\mu_s\left(\left\{\eta_{c_{j_1}}, \ldots, \eta_{c_{j_k}} \mid c_{j_*} \in \mathcal{N}(s_i)\right\}\right) = \mathrm{MAB}_s([\eta_{s_i}], K_c, V_c)$$
$$\mu_c\left(\left\{\eta_{s_{i_1}}, \ldots, \eta_{s_{i_l}} \mid s_{i_*} \in \mathcal{N}(c_j)\right\}\right) = \mathrm{MAB}_c([\eta_{c_j}], K_s, V_s)$$
$$\text{where} \quad K_c = V_c = \begin{bmatrix} \eta_{c_{j_1}} \\ \ldots \\ \eta_{c_{j_k}} \end{bmatrix} \in \mathbb{R}^{k \times d}$$
$$K_s = V_s = \begin{bmatrix} \eta_{s_{i_1}} \\ \ldots \\ \eta_{s_{i_l}} \end{bmatrix} \in \mathbb{R}^{l \times d}$$

Put into words, μ_s aggregates all embeddings of courses adjacent to a student by using them as keys and values for an attention operation, where the student's embedding is the query. This breaks down into calculating a weighted sum of these course embeddings where the weights depend on the students embedding. This operation is favored over a regular sum as aggregation, because MABs are universal approximators for functions in the space of permutation-invariant functions [20], and can thus learn to approximate more suitable aggregations if they exist, or just fall back to a regular sum if that happens to be sufficient.

The aggregation μ_c can be described analogously, but students and courses exchange roles.

After U simultaneous applications of the embedding rules F_s and F_c to the initial student and course embeddings, the request embeddings are obtained by jointly processing student and course embeddings with a 2-layer FFN $G : \mathbb{R}^{2d} \mapsto \mathbb{R}^d$ with ReLU activations

only. That is, the embedding of the request (s_i, c_j) is calculated as

$$\eta_{(s_i,c_j)} = G(\text{concat}(\eta^U_{s_i}, \eta^U_{c_j})) \tag{5.3}$$

We refer to this kind of encoder module as GNN encoder.

Regarding a good choice for U, we expect that the encoder might be very sensitive to this parameter which is why in Section 6.3.1 we do a short trial to determine a good value.

An alternative embedding rule is given by the less complicated S2V embedding rule we also discussed in Section 4.2.2. The encoder that uses two of these rules instead of the MAB embedding rules will be referred to as S2V.

The respective encoder modules for simple assignment problem instances work in nearly the same way. The only thing that changes is that edges are only labeled with the $x_{i,j}$ and no longer with request values.

Encoding with Request Tokens

The approaches we define here involve a parameter-free preprocessing step for the purpose of capturing problem structure. Namely, we pre-encode $n = |R|$ requests into n requests *tokens*, i.e., vectors containing p manually chosen features concerning the student and course of each request.

Assuming an arbitrary assignment X for some simple assignment problem instance, the request token for (s_i, c_j) includes $p = 3$ features and is given by

$$[x_{i,:}, r_{i,:}, C(j) - x_{:,j}].$$

For the formal definitions of these features see Section 3.2.1. The first feature conveys how many requests have already been accepted for student s_i, the second feature how many requests the student made and the third feature captures the remaining capacity of course c_j.

In the valued case, we extend the above tokens with

$$[v_{i,j}, w_{i,:}, w_{:,j}],$$

i.e., the request value and the value total already assigned to s_i and c_j (for formal definitions see Section 3.2.2).

In a way, these features also exploit the graph-like structure of the assignment problem. When considering R or X as edges and $s \cup c$ as vertices, they can all be interpreted as features of these graphs:

- $x_{i,:}$ is the size of the neighborhood of s_i in the graph $\langle s \cup c, X \rangle$.
- $r_{i,:}$ is the size of the neighborhood of s_i in the graph $\langle s \cup c, R \rangle$.

- $x_{:,j}$ is the size of the neighborhood of c_j in the graph $\langle s \cup c, X \rangle$, $C(j)$ is simply the label of c_j.
- $v_{i,j}$ is the weight of edge (s_i, c_j) in the graph $\langle s \cup c, R \rangle$.
- $w_{i,:}$ and $w_{:,j}$ are the sums of edge weights over the neighborhood of s_i and c_j in the graph $\langle s \cup c, R \rangle$ and $\langle s \cup c, X \rangle$.

As ISAB encoder we define the module that further processes a set of given request tokens $Z \in \mathbb{R}^{n \times p}$ in the following way. First, we apply a 2-layer FFN $\mathbb{R}^p \mapsto \mathbb{R}^d$ with ReLU activations to each row of Z, yielding preliminary embeddings $\tilde{\eta} \in \mathbb{R}^{n \times d}$. These embeddings are then refined using an induced set attention block (see Section 4.2.3), i.e., the η_a are the rows of $\text{ISAB}(\tilde{\eta}) \in \mathbb{R}^{n \times d}$.

As FFN encoder we define the module that instead applies only a 3-layer FFN $\mathbb{R}^p \mapsto \mathbb{R}^d$ with ReLU activations to each row of Z and uses those as final request embeddings without further processing them with an ISAB. As the request tokens already capture some of the problem structure, an ISAB might not be needed strictly necessary for learning embeddings that allow to learn reasonable policies.

5.6.2 Aggregation Module

The aggregation module μ_X represents an important mathematical relation between the MDP state and actions that arises as a consequence of our deterministic transition function. It is easy to see, that in our case, the state X is the union over all actions that have been executed so far. So μ_X should be a network that takes as input all request embeddings η_a s.t. $a \in X$ and aggregate them into a state embedding $\eta_X \in \mathbb{R}^d$ in such a way that is permutationally invariant w.r.t. to its input, as the order in which requests are accepted does not matter for our notion of states as sets of requests.

If not stated otherwise, we use a PMA module (see Section 4.2.3) for aggregating request embeddings into a state embedding.

5.6.3 Policy and State-Action Value Modules

The final module must represent either a policy, or a state-action function. At their core both use a function that performs a mapping $\mathbb{R}^d \times \mathbb{R}^d \mapsto \mathbb{R}$ that is applied to each state-action embedding pair (η_X, η_a) individually.

When the final module's purpose is to estimate state-action values $Q(X, a)$, we use a 3-layer FFN $Q : \mathbb{R}^{2d} \mapsto \mathbb{R}^1$ where state-action pairs (X, a) are represented by the concatenation of η_X and η_a and the FFN output is computed individually for each action a as $Q(\text{concat}(\eta_X, \eta_a))$.

In the policy case, we use two different approaches. For graph-based encoders and the ISAB encoder, we adopt the pointing mechanism that is used in pointer networks [2]. Given a set of actions embeddings $\eta_{\mathcal{A}} = \{\eta_a \mid a \in \mathcal{A}\} = \{\eta_1, \dots \eta_a, \dots, \eta_r\}$ and a state

embedding η_X of $X \in \mathcal{S}$, its output $\pi(a \mid X)$, the „pointers“, can be described as

$$\begin{aligned} \pi(a \mid X) &= \text{softmax}(u_1, ..., u_r)_a \\ \text{where} \quad u_a &= F(\eta_{a,X}) \\ \eta_{a,X} &= \tanh(f_a(\eta_a) + f_X(\eta_X)) \end{aligned} \tag{5.4}$$

and $f_a : \mathbb{R}^d \mapsto \mathbb{R}^d$, $f_X : \mathbb{R}^d \mapsto \mathbb{R}^d$ and $F : \mathbb{R}^d \mapsto 1$ are 1-layer FFNs with no bias and linear activation.

When creating the FFN request token encoder we intended it to be a kind of baseline DRL architecture, so we also decided to keep the policy module as simple as possible. It is thus more similar to the Q module as above, a 3-layer FFN $\pi : \mathbb{R}^{2d} \mapsto \mathbb{R}$ with ReLU activations for the hidden layers. The final policy probabilities are then computed in a similar way as in the pointing mechanism, i.e., we apply the FFN to each state-action pair and then apply the softmax activation to the vector of all respective outputs.

In Section 4.1.2 we discussed that it is often better to use an Actor-Critic approach instead of a vanilla policy gradient approach. The policy module resembles the actor, but we are still in need of a critic module, that learns state-value estimates. Our proposed solution is to simply reuse the state embedding η_X as input for a 3-layer FFN $V : \mathbb{R}^d \mapsto \mathbb{R}$ that is consequently trained to estimate V_{π_θ}.

6 Experiments

This chapter is segmented into four parts. First, we focus on simple assignment problems only and show results for two experiments that support some of the claims we made in Section 5.3. To this end we briefly compare single-instance learning and single-instance architectures with our concept of multi-instance learning and multi-instance architectures.

The next two parts focus on the training of all different architectures and algorithms we introduced in Chapters 5 and 4 on simple assignment problems and valued assignment problems, as well as how these trained agents compare to Gurobi as general solvers when evaluated on instances of larger size. To assess the performance of DRL agents compared to Gurobi, we focus on mainly two metrics. First, we examine how close to optimal the assignments are that are found by trained DRL agents, especially for instances that are larger than those from the training distribution, and secondly, we want to study how much time it takes them to find these solutions.

When solving any instance, Gurobi is able to determine if a feasible solution exists and more importantly, Gurobi can also provide us with knowledge about how close its found solution is to an optimal one. These are both features our DRL approaches can not provide. Due to approximation biases that arise as a consequence of using neural networks as function approximators, and the bias introduced by training on only a finite number of instances from a fixed training distribution that is not representative for all possible problem instances, DRL methods are bound to be inexact. The only motivation for applying such inexact methods in practice is that they might take less time than exact methods. Since neural networks are usually relatively fast, we anticipated that they should be able to compete with exact combinatorial optimization solvers in that regard. However, as this chapter will show, this is often not the case.

Apart from comparing the DRL approaches to Gurobi only, we also want to compare different DRL algorithms and neural network architectures among each other, in order to study whether we can observe any significant differences regarding their suitability for the task at hand.

All experiments which measured run-time in some way were conducted using an Intel(R) Core(TM) i9-9900K CPU @ 3.60GHz (16 cores) with 32 GB of RAM and no restriction on how many CPU cores can be used. For implementing and training neural networks we used PyTorch [25]. All DRL experiments make use of the Adam optimizer [19] for updating parameters.

6.1 Instance Training Distributions

As a basis for our experiments we mainly used artificial problem instances that are generated randomly using a set of distributions $\mathcal{P}_r$, $\mathcal{P}_c$, $\mathcal{P}_p$, $\mathcal{P}_s$, $\mathcal{P}_v$ over several instance properties, namely the total number of requests, the capacity of courses, the popularity of courses, the number of requests per student and the request values. For this purpose, we first randomly sample an integer $r \sim \mathcal{P}_r$ that determines the size of the set of requests R. In the next step, we generate all courses together with capacities and popularities using $\mathcal{P}_c$ and $\mathcal{P}_p$, such that the amount of requests that need to be generated in order to match the course popularities is exactly r. Finally, we iteratively generate students with randomly sampled requests for those courses and optionally random values, until the limit of r requests is met. The number of requests students make is drawn from $\mathcal{P}_s$ and the value of requests from $\mathcal{P}_v$. For a more detailed description see Algorithm 4.

Note that we only described the instance generation for valued assignment problem instances, as the procedure for the simple assignment problem works in the exact same way, except that we do not need to generate request values.

During training of RL agents we focus on the distribution over instances that arises from Algorithm 4 with the following input.

- Number of total requests drawn from $\mathcal{P}_r = \mathcal{U}\{40, 80\}$,
- course capacities drawn from $\mathcal{P}_c = \mathcal{U}\{10, 20\}$,
- number of requests per student drawn from $\mathcal{P}_s = \mathcal{U}\{1, 4\}$,
- course popularities drawn from $\mathcal{P}_p = \mathcal{U}(1.5, 2.5)$, and
- request values drawn from $\mathcal{P}_v = \mathcal{U}\{1, 4\}$,

where $\mathcal{U}(\cdot)$ are continuous and $\mathcal{U}\{\cdot\}$ discrete uniform distributions. We refer to this as our *training distribution*.

When evaluating whether agents are able to generalize their policies to instances of larger size, we use the same distribution to generate evaluation instances, but instead of a random number of requests $r \sim \mathcal{P}_r$, we will fix r to concrete values, while the remainder of Algorithm 4 is left unmodified.

Algorithm 4 Instance Generation

Input: Distributions $\mathcal{P}_r$, $\mathcal{P}_c$, $\mathcal{P}_p$, $\mathcal{P}_s$, $\mathcal{P}_v$
$n \leftarrow 0$ ▷ student counter
$m \leftarrow 0$ ▷ course counter
$k \leftarrow 0$ ▷ request counter
$R \leftarrow \emptyset$ ▷ Set of requests
$C \leftarrow$ empty mapping of courses to capacities
$V \leftarrow$ empty mapping of requests to values
Sample target num. of requests $r \sim \mathcal{P}_r$
while $k < r$ **do** ▷ generate courses
    $m \leftarrow m + 1$
    Sample course capacity $C(m) \sim \mathcal{P}_c$
    Sample a course popularity $p_m \sim \mathcal{P}_p$
    Calculate number of requests for course as $r_{:,m} \leftarrow \lfloor p_m C(m) \rfloor$
    $k \leftarrow k + r_{:,m}$
if $k > r$ **then**
    Clamp $C(c_m)$ and $r_{;.m}$ s.t. $k == r$
Create a collection $\boldsymbol{B}$ containing the courses c_l exactly $r_{:,l}$ times $(l = 1, \ldots, m)$
while $\boldsymbol{B}$ is not empty **do** ▷ generate students and requests
    $n \leftarrow n + 1$
    Sample a number of choices $j \sim \mathcal{P}_s$ for student s_n
    Sample mutually distinct courses $c_{i_1}, \ldots, c_{i_j} \sim \mathcal{U}(\boldsymbol{B})$ without replacement
    Sample request values $V(s_n, c_{i_1}), \ldots, V(s_n, c_{i_j}) \sim \mathcal{P}_v$
    Normalize $V(s_n, c_{i_1}), \ldots, V(s_n, c_{i_j})$ s.t. they sum to 1
    Add requests $R \leftarrow R \cup \left\{(s_n, c_{i_1}), \ldots (s_n, c_{i_j})\right\}$
return Instance $\langle \{s_1, \ldots, s_n\}, \{c_1, \ldots, c_m\}, R, C, V \rangle$

This algorithm can be used to randomly sample artificial problem instances. We only show the algorithm for valued instances. The algorithm for simple instances is nearly identical, it just does not require $\mathcal{P}_v$ and the related operations.

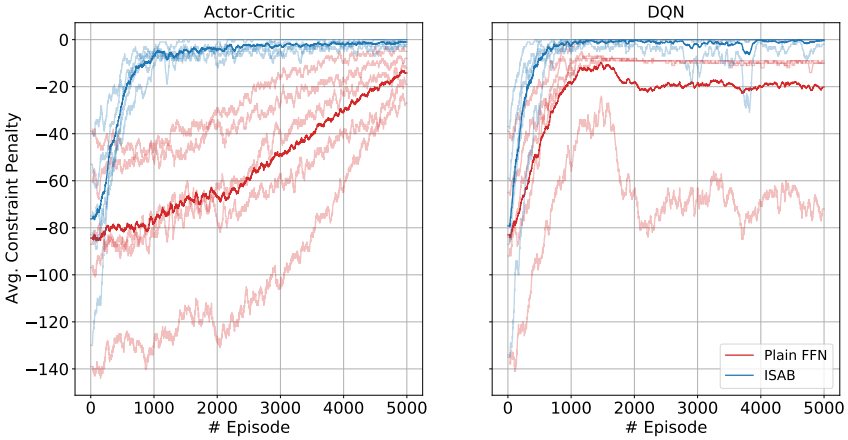

Figure 6.1: Comparison of plain FFN architecture (red) and ISAB encoder architecture (blue).

6.2 Comparison to Single Instance Architectures

We have already discussed how plain feed-forward networks can not be used to train on multiple problem instances of different size in Section 5.4. Now we intend to show that in any case, so also in the case of single instance training, where they can be applied, they are also the inferior choice of neural network architecture for DQN agents as well as Actor-Critics. To this end, we train four different RL agents on the same five simple assignment instances from our training distribution. We compare two Actor-Critic agents and two DQN agents, where for both algorithms, one agent uses plain feed-forward networks with states as input as described in Section 5.4 while the other ones uses the ISAB encoder network described in Section 5.6. We train the agents for 5000 episodes each. The decay for the ϵ-greedy DQN policy is chosen s.t. $\epsilon_{end} = 0.01$ is reached after exactly 3000 episodes. All other algorithmic hyper parameters are kept fixed and are can be viewed in table 6.2.

Figure 6.1 shows the resulting training curves.

Both the DQN and Actor-Critic agent learn much faster and more consistently when supplied with the ISAB encoder network that exploits the problem structure. This is most likely due to the large action space of the assignment problems. The DRL method which uses plain FFNs suffers from the curse of dimensionality, as it is forced to estimate the policy logits (policy before softmax was applied), or state-action values for all actions at once. As the output size of deep neural networks in general increases, it becomes increasingly more difficult to train them. So, if we apply DRL to problems with large discrete action spaces, training may be slow, or even fail to find satisfactory solutions. This is a known issue and a remedy lies within action embeddings [8]. In this thesis they arose naturally as a byproduct of our need for an architecture that is able to process differently sized instances.

Another advantage we already discussed in Section 5.3 is that training an agent on small instances and then using it to solve larger instances afterwards is likely to be more time-efficient compared to the naive approach of retraining a new agent from scratch for each instance. In order to support this claim with empirical results, we first trained an Actor-Critic agent with the FFN encoder architecture from Section 5.6.1 for 30 minutes on our regular training distribution. Then, we sampled four instances of different, fixed sizes (number of requests) 250, 500 and 1000, from our otherwise unchanged training distribution, and used the previously trained agent to solve them, measuring the quality of solutions as well as solving run-time. Next, we also apply the single-instance approach. We retrain the exact same agent, as well as an Actor-Critic agent using plain FFNs, on each of the three tasks individually, until they find a solution of equal or better quality, or until a timeout of 30 minutes is reached. Table 6.1 shows the results of this experiment.

Instance Size	MI		SI-FFN		SI-PLAIN	
	Value	Time [s]	Value	Time [s]	Value	Time [s]
250	0.0	0.1	0.0	340.2	0.0	632.5
500	0.0	0.1	0.0	1314.2	0.0	1667.9
1000	0.0	0.2	-486.7	1800.0	-553.3	1800.0

Table 6.1: Objective value of the best assignment found during inference, or training from scratch (column Value) and elapsed time in seconds until this best assignment was first found (column Time). Results are averaged over five different random seeds. MI is the pre-trained Actor-Critic that solves the instance after it has been trained on our fixed training distribution. SI-FFN and SI-PLAIN are Actor-Critic agents learning to solve each instance from scratch with a multi-instance FFN encoder, or plain FFN architecture as backbone with a timeout of 30 Minutes.

The Actor-Critic FFN encoder agent generalizes perfectly to these three instances and it takes just fractions of a second to solve one instance using this trained agent. Training agents with the same configuration to find optimal solutions from scratch on the exact same instances takes significantly more time and in the cases with 1000 requests, after the 30 minute timeout is reached, a reasonable solution has not yet been found. These results supports our claim that learning a generalizable strategy is the favored approach over single-instance learning.

6.3 Training DRL Agents

Before being able to compare all of our proposed DRL agents to Gurobi, we first need to train them and make sure that they learn reasonably good policies. In order to judge how well the agents were able to learn a policy during training, we study their respective training curves. The training curves of DRL agents plot their performance as they explore the environment and learn to improve their policies. In our case, we plot the current training episode on the x-axis and the penalized objective value of the assignment found in that episode on the y-axis. We smooth the curves by averaging the

assignment objective values using a window of size 300.

We also used Gurobi to find the optimal objective values for all training instances and plot them along with the training curves s.t. we can judge, how well the DRL agents are able to approximate optimal policies for the training distribution. The same smoothing window that was applied to the training curves is also applied to the optimal objective values for better comparability. For simple assignment problems we do not plot the objective value directly but instead just the penalty incurred by any unsatisfied constraints, as the objective value, i.e., number of accepted requests will always be equal to the total course capacity.

We trained eight different DRL agents in total, each on five different seeds. The different configurations arise as the cross product of two algorithms (Actor-Critic and DQN) and four different multi-instance architectures that arise from the four different encoder architectures we introduced in Section 5.6.1. As a reminder, these are the two encoders that process parameter-free request tokens with either a FFN or an FFN in combination with an ISAB module (we call the former FFN encoder, the latter ISAB encoder), and the two graph neural networks, one of which uses the S2V node aggregation, while the other one uses MABs for node aggregation, where we call the former S2V encoder and the latter GNN encoder. For all Actor-Critic agents, the neural networks also differ slightly in the policy module they use, depending on the encoder (see Section 5.6.3).

Nearly all hyperparameters used for all Actor-Critic as well as DQN agents are fixed across all experiments and are shown in Table 6.2. Depending on the number of episodes trained, we just alter the ϵ-greedy parameters of DQN agents. When training for 5000 episodes, the epsilon decay is chosen s.t. ϵ_{end} is reached after 3000 episodes, when training for 10000 episodes s.t. ϵ_{end} is reached after 6000 episodes.

(a) DQN

Hyperparameter	Value
ϵ_0, ϵ_{end}	1.0, 0.01
Replay Buffer Size	10000
Update every	4
Batch Size	16
Learning Rate	0.0005
Target Update τ	0.001
Discount Factor γ	0.995

(b) Actor-Critic

Hyperparameter	Value
Learning Rate	0.0005
Entropy Penalty β	0.001
Discount Factor γ	0.995

Table 6.2: DRL training algorithm hyperparameters.

6.3.1 Tuning Recurrence in Graph Neural Networks

We have not yet fixed the hyperparameter U of the graph-based neural networks that determines the number of recurrent steps. Here we conduct a simple trial where we try different values for U on the simple assignment problem and choose the best setting for the rest of our experiments. To this end, we trained an Actor-Critic with the GNN

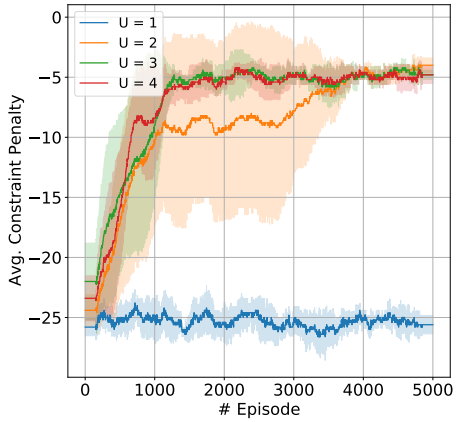

Figure 6.2: Training curves for a GNN Actor-Critic with different amounts of recurrent steps.

encoder for 5000 episodes with $U \in \{1, 2, 3, 4\}$, which yielded the results shown in Figure 6.2. The curves give the mean performance over five different random seeds on our training distribution and the shaded areas illustrates the 95% confidence intervals. As the plot shows, $U = 1$ seems to be a sub-optimal choice, while for $U > 1$, the final results are all very similar. In the end we decided to use $U = 3$, as $U = 2$ seems to be not as robust as $U \in \{3, 4\}$ and increasing U from three to four does not seem to yield any further improvement.

6.3.2 Multi Instance Training

As already discussed in Chapter 3, we focus on three kinds of assignment problems. Simple assignment problems as well valued assignments problems, once with $(v^s_{min}, v^c_{min}) = (0.5, 0.5)$ (S5C5) and once with $(v^s_{min}, v^c_{min}) = (0.7, 0.5)$ (S7C5). We train all eight DRL agents on all instances for all three problems, using the training distribution from Section 6.1.

The two kinds of valued assignment problems differ by the fact that with $v^s_{min} = 0.7$, it becomes significantly more difficult to satisfy the respective constraints which demand that the requests assigned to each student have at least that much accumulated value.

For simple assignment problems, training is set to last for 5000 episodes, for S5C5 as well as S7C5 problems, we use 10000 episodes.

Figures 6.3, 6.4 and 6.5 show the resulting training curves for the simple, S5C5 and S7C5 problems. The left-hand plots show the training curves for our Actor-Critic agents and the right-hand plots show the training curves for our DQN agents. Which network was used is color coded. The dashed red curve in the DQN plots shows the values for the exploration coefficient of the ϵ-greedy policy over the course of training. The grey curve labeled OPT shows the objective value of optimal solutions found by Gurobi. The curves are all averaged over five different training runs, each one using a different

random seed, and the shaded areas depict the per-episode 95% confidence interval after applying the smoothing windows. They are based on the student's t-distribution, since we are estimating both mean smoothed performance and performance standard deviation.

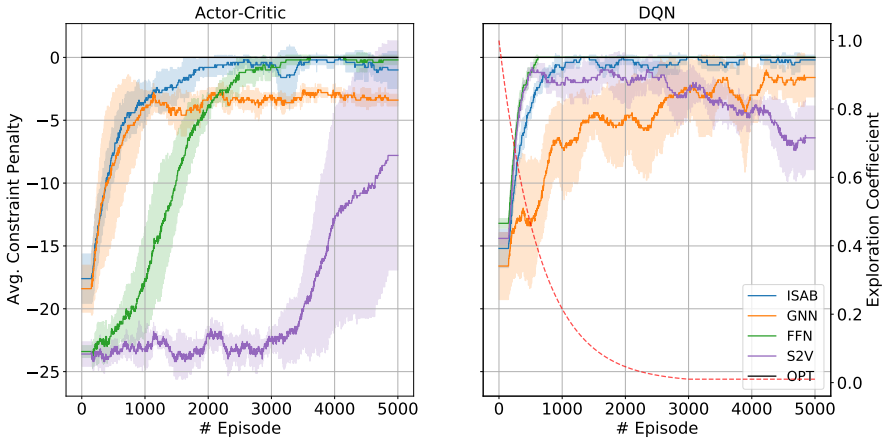

Figure 6.3: Training curves of all configurations on simple assignment problem instances.

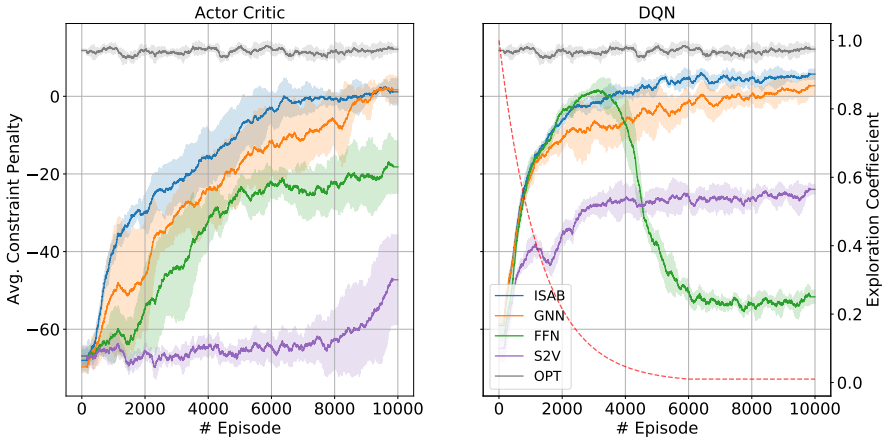

Figure 6.4: Training curves of all configurations on S5C5 valued assignment problem instances.

Simple Assignment Problem

Looking at the training curves for the simple assignment problem, it seems like most non-graph based DRL agents are able to approximate optimal policies for the training distribution fairly well. When using an FFN encoder as backbone, both the DQN and Actor-Critic agent are able to approximate optimal policies almost perfectly across all seeds. Agents equipped with the ISAB encoder perform slightly worse, but are also able

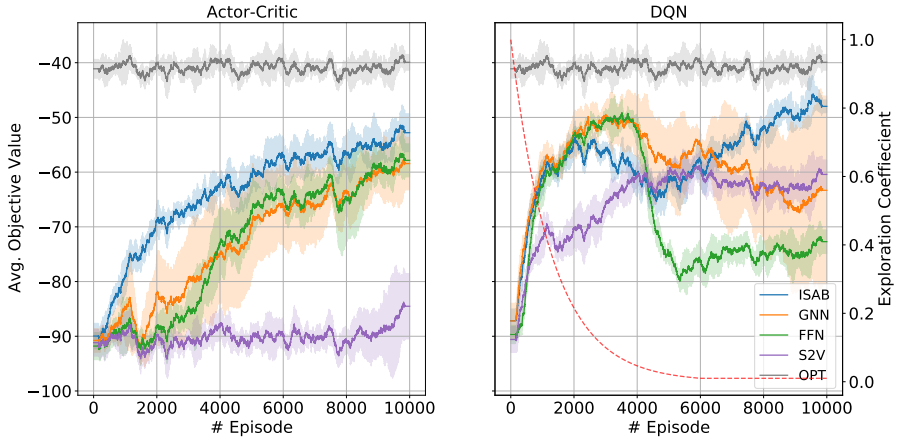

Figure 6.5: Training curves of all configurations on S7C5 valued assignment problem instances.

to find near-optimal policies consistently.

Both graph neural network encoders however exhibit worse performances in both DRL contexts, as well as highly variable results in case of the S2V encoder. While the GNN encoder which uses MABs for neighborhood aggregation still performs reasonably well, results when using the more simple S2V encoder are significantly worse.

The observation about these results that surprised us most is that the FFN request token encoder performs best among all architectures. This means that the simple assignments problems apparently do not require any overly complicated architecture that goes beyond what our hand-crafted request tokens are able to provide.

S5C5 Assignment Problem

For valued assignment problems, the performance hierarchy changes. ISAB and GNN encoders are still able to find close to optimal policies, in both the Actor-Critic as well as DQN context, whereby the DQN agents learn slightly faster and achieve slightly better performance in the end. The results we obtain with the GNN encoder are less robust than those obtained with ISAB in both cases.

The agents using the FFN encoder however perform worse than before. If the FFN encoder is used in the DQN context, the agent even seems to be unable to maintain a good strategy, starting at about 4000 episodes, as the ϵ-greedy exploration coefficient has nearly decayed to zero, the estimates of the state-action values give rise to a sub-optimal policy. Used in combination with Actor-Critics, the FFN encoder is still able to improve his policy compared to the start, but not by as much as his counterparts using ISAB and GNN.

The agents using an S2V encoder exhibit poor performance. In the Actor-Critic context it takes very long for them to start learning at all, and in the DQN context, even though

the assignments they find become better in the beginning, it appears that the estimates of the optimal Q function converge to a highly sub-optimal approximation.

S7C5 Assignment Problem

The results on this kind of problem are similar to the previous scenario, but the instabilities of the DQN algorithm become more severe. Three out of the four DQN agents are subject to a decrease in performance after an initial phase of policy improvement. While the agent with an ISAB encoder is able to recover, the GNN and FFN agent are unable to generate any improvement afterwards.

The Actor-Critic agents learn more slowly, but also appear to be more robust in doing so. The ISAB agent also outperforms all other Actor-Critic agents in training performance, while the GNN and FFN encoder yield very similar results in mean performance, but the GNN agents performance is more variable.

The S2V encoder seems to again not mesh well with the Actor-Critic approach. This agent is not able to improve his policy for a long time, until eventually it starts to learn after 10000 episodes have almost ended. When deployed in the DQN context, it learns more slowly, but also more consistently and ends with a performance that is on a similar level as that of the other graph-based encoder.

Overall these results suggest that the ISAB-encoder based agents are the best choice for learning policies that perform well on our training distribution. None of the agents are however able to approximate policies for valued assignment problems that are near-optimal. Judging the distance between the training curves and the optimality curve one can argue that the learned policies are reasonable, but also clearly not optimal. In the next section, where we evaluate the agents, we also investigate one possible cause for this.

6.4 Evaluation

In this section we want to put our focus on two things. We are primarily interested in how well our DRL approaches compete with Gurobi on instances of increasing size. This should also give us an idea on the ability of our multi-instance RL agents to generalize to problems that are larger than those we trained on. To assess, whether DRL approaches are competitive, we want to see how well they are able to generalize their policies to larger instances, and how much time it takes them to solve these instances. Since Gurobi finds optimal solutions to all evaluation instances within a ten minute timeout, we can not hope to improve upon Gurobi in that regard, but we want to investigate, whether we may be able to find solutions of similar quality in less time.

Moreover, since we implemented two different DRL algorithms and several different multi-instance architectures, it will be also interesting to see, how they compare to one another in terms of solution quality, ability to generalize, and solving run-time.

6.4.1 Evaluation Setup

Data

For the purpose of generating evaluation instances, we stick to our training distribution, but for each distribution, we sample ten instances of increasing size, where instance size is measured by the number of total requests. The smallest instance starts out at a size of 500 requests, which is incremented to a maximum of 5000 in steps of size 500. We repeat this process for five different random seeds, so for each of the three problem settings we trained on, we obtain one evaluation set consisting of 50 instances.

Gurobi

Because a majority of our code is written in Python, we make use of the Python API provided by Gurobi to implement our assignment problems as a Gurobi model and control the optimization process.

For modelling the assignments problems as integer programs we introduce binary variables for each request and implement the exact constraints as shown in sections 3.2.1 and 3.2.2. Fixed values such as course capacities and request values are introduced as constants. When solving an instance, we first call Gurobi's optimization routine with all constraints fixed as hard constraints. If this process returns that the instance is infeasible, we apply a feasibility relaxation to the model [10] that implements the penalized objective functions introduced in Section 3.2, that is, we relax all constraints that were introduced as soft constraints s.t., independent of their magnitude, violations in the relaxed model generate a fixed penalty of minus ten. After that we again call the optimization procedure on the relaxed model.

We do not impose any time limit on Gurobi and we enable its pre-solving routine, as with this routine enabled, opimal solutions were found in less time. The optimization procedure is set to terminate once an optimal solution within a tolerance of 1×10^{-4} is found.

Heuristic Baselines

When evaluating the quality of solutions obtained by our trained RL agents we also compare them to greedy baseline heuristics.

For simple assignment problems it is very easy to implement a heuristic that essentially performs hill-climbing. The idea is to first iteratively accept requests $(s_i, c_j) \in \text{Open}_I(X)$ where the student s has made more than 1 request and has not been assigned to any courses yet. That means we first try to greedily grant requests that help satisfy as many soft constraints as possible before granting any other requests and do so in an arbitrary order (in practice this means we do random tie-breaking). We denote this baseline by HC-Simple.

A slightly more informed version of this heuristic does not accept these requests in an arbitrary order, but instead breaks ties by course popularity, where requests for courses

with low popularity are preferred. That means we will first grant those requests for courses which are not highly requested. This leaves more capacity for later on when we have to grant requests for students who compete for highly popular courses. We denote this heuristic by HC-Simple+.

For valued assignment problems we also implement a greedy heuristic that prefers to grant requests that immediately satisfy some soft constraint, if possible, and breaks ties by request values. We denote this baseline as HC-Valued.

Evaluating trained DRL agents

Trained RL agents can be evaluated by greedily generating assignments from their final policies. For DQN agents, we generate assignments from π_{Q_θ}. For Actor-Critic agents, we calculate π_θ only and for performance reasons disable the critics output stream and do not apply the final softmax activation. This is possible, because the result of any greedification, i.e., of choosing a maximizer from some set of elements, is unaffected by the softmax operation. The Actor-Critic evaluation policies thus select the action with maximum „probability before softmax“. In both cases we use the parameters θ as they were obtained in the end of training.

We refrain from evaluating the cross product of all trained agents (we trained each agent on five seeds) with all evaluation instances (five seeds with ten evaluation instances each), and all three problem cases (simple, and valued S5C5 and S7C5), because it would be too costly. We instead select the agent with median final performance, as measured by the objective value of assignments generated over its last 500 training episodes and perform evaluation only with the saved agent's final network parameters.

While we used PyTorch's automatic differentiation module for the purpose of backpropagation during training, we disable this functionality during evaluation, as we no longer need to compute gradients. This also helps to reduce run-time of DRL agents.

6.4.2 Generalization to unseen instances

We now put our focus on the quality of assignments that our trained RL agents find on larger-than-training instances when compared to assignments found by Gurobi and the heuristic baselines. Due to the already poor training performance, we exclude the agents that used an S2V encoder from evaluation. So we only consider the performance of Actor-Critic agents as well as DQN agents that use an ISAB, FFN, or GNN encoder.

We plot the same quantities on the y-axis as we did on out training plots. On the x-axis we plot the size of the ten evaluation instances. The different neural network architectures used are again color-coded, and the shaded areas again represent the 95% confidence interval over the five different seeds used for generating the total of 50 evaluation instances. The error-bars represent the objective value of the best and worst assignment that was found.

Simple Assignment Problem

In Figure 6.6 we can observe that the agents which were able to learn good policies in the first place, are also best at generalizing these to similar instances of larger size. The GNN-DQN agent seems to be unable to generalize its policy at all. The performance of the GNN-AC agent also decreases as problem size increases, but not as much as for the GNN-DQN agent.

The best agent, the FFN Actor-Critic, is able to generalize its policy perfectly to the presented instances. When paired with DQN it performs slightly worse. Compared to Gurobi and the Hill-Climbing heuristics (see Figure 6.7), FFN-AC is also the only agent that is able to perform on the same level as HC-Simple+ and nearly the same level as Gurobi.

The ISAB agents are partially able to generalize their policies, they are second best among all DRL agents, but their performance also decreases as problem size increases.

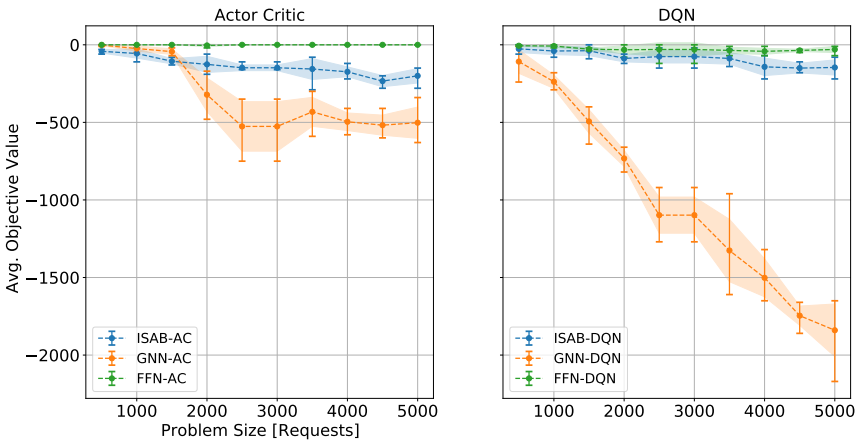

Figure 6.6: Evaluation for simple assignment problems. DRL agents only.

Valued S5C5 Assignment Problem

In this scenario, the ISAB agents both perform best within their agent-type (DQN or Actor-Critic) (see Figure 6.8). Overall, Actor-Critics perform significantly better than their DQN counterparts.

Considering Figure 6.9, The ISAB Actor-Critic and FFN Actor-Critic are the only agents that are able to significantly outperform the HC-Valued heuristic. Only the ISAB Actor-Critic alone comes close to the performance of Gurobi.

Interestingly, except for problems with more than 4000 requests, the solutions found by the GNN Actor-Critic agent are eerily close to those of the HC-Valued heuristic, leading to the assumption that it might learn a similar strategy as the heuristic.

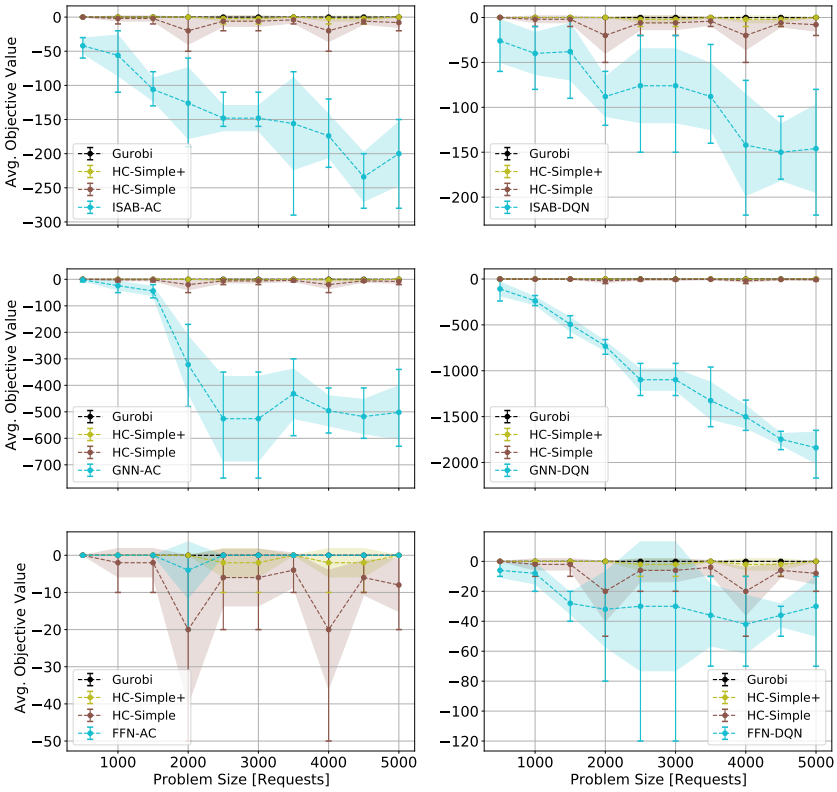

Figure 6.7: Evaluation for simple assignment problems. Per-agent comparison to Gurobi and heuristic.

The fact that the FFN-encoder-based DQN agent is unable to perform well should not be surprising, as it did not perform well during training either (recall Figure 6.4). Similarly to the simple assignment problem, the GNN-DQN based agent is also unable to generalize his policy to instances of larger size. The FFN Actor-Critic performs just slightly worse than its GNN counterpart and the ISAB-DQN performs not quite as good as the HC-Valued heuristic.

Valued S7C5 Assignment Problem

When increasing the minimum accumulated request value for students, the choice of network seems to make less of a difference for the performance of Actor-Critics (see Figure 6.10). They all perform on a very similar level regarding their ability to generalize. For DQN-based agents, ISABs as encoder outperform the GNN and FFN encoders.

Figure 6.8: Evaluation for S5C5 valued assignment problems. DRL agents only.

In direct comparison with Gurobi and the HC-Valued heuristics (Figure 6.11), 4 out of 6 DRL agents outperform the HC-Valued heuristic, all of which also come fairly close to the optimal solution quality of Gurobi.

The two agents that perform significantly worse than the other four, GNN-DQN and FFN-DQN, are also those which did not perform very well during training (see Figure 6.5).

6.4.3 Inspecting Constraint Violations

In order to investigate what caused our agents to be unable to learn near-optimal policies in the valued assignment problem, we separated the objective value of assignments into a number of violations per type of constraint.

Figure 6.12 shows the absolute difference in constraint violations for the ISAB-AC agent and Gurobi averaged over all different evaluation instances. That is, for each kind of constraint, we go over the assignments both methods found and count how many of them are violated. We then subtract the number of violations Gurobi made from the number of violations our agent made.

The label S-REQ refers to the constraint that students should be assigned to at least 1 course, if they applied for more than one, S-VAL and C-VAL refer to the constraints that demand the accumulated value per student and (average) value per course to be larger than some fixed constant. The left plot shows results for the S5C5 setting, the right plot shows results for the S7C5 setting. What can be inferred from these plots is that the approximation errors of the DRL assignments in both settings might appear similar if we just inspect the penalized objective value, but they arise as a consequence of two completely different things going wrong.

While the DRL agent is able to learn to optimally satisfy the constraints labeled S-

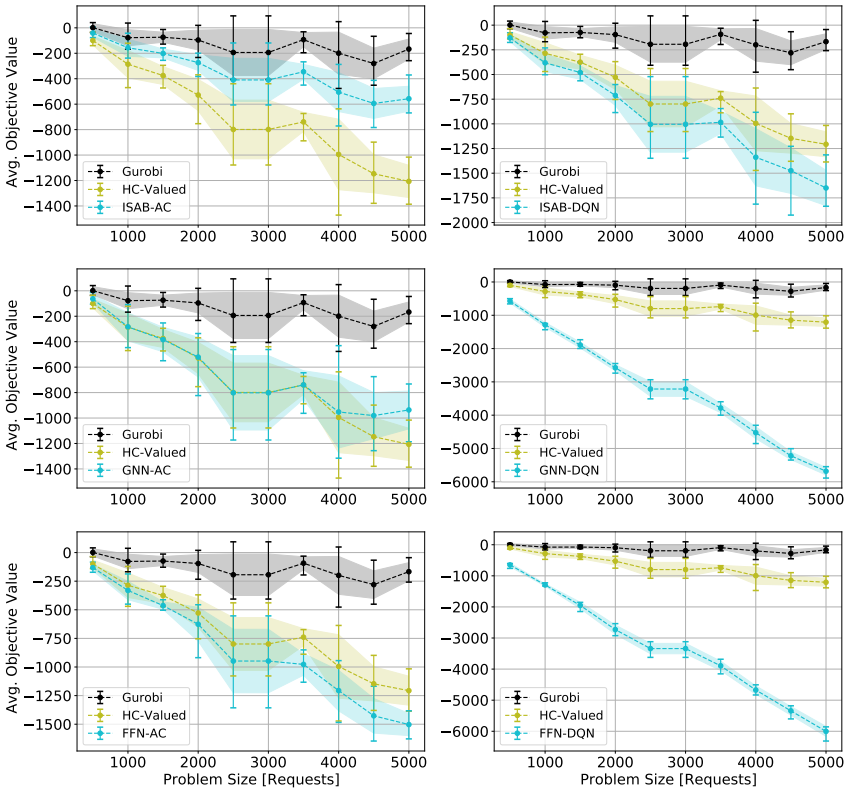

Figure 6.9: Evaluation for S5C5 valued assignment problems. Per-agent comparison to Gurobi and heuristic.

REQ in the S5C5 scenario, he does not come close to achieving the same thing in the S7C5 scenario. Our interpretation on why this happens is that the reward signal in this scenario is dominated by penalties for violating the S-VAL constraints, as a consequence of the increased minimum accumulated student value v^s_{min}. Since the penalized objective function is nothing but a linear combination of the objective and all constraint penalties, penalties for constraints that are less likely to be violated are overwhelmed by the dominating ones that are violated much for frequently by solutions found during exploration. As the resulting policy gradient estimates and value function gradient estimates are averaged over whole trajectories, it thus becomes difficult for the ISAB-AC DRL agent to learn to satisfy the constraints that are in the minority.

A similar argument can be made for the C-VAL constraints. Since there are far less courses than students, situations where the agent can learn to satisfy them are sparse compared to his remaining experiences.

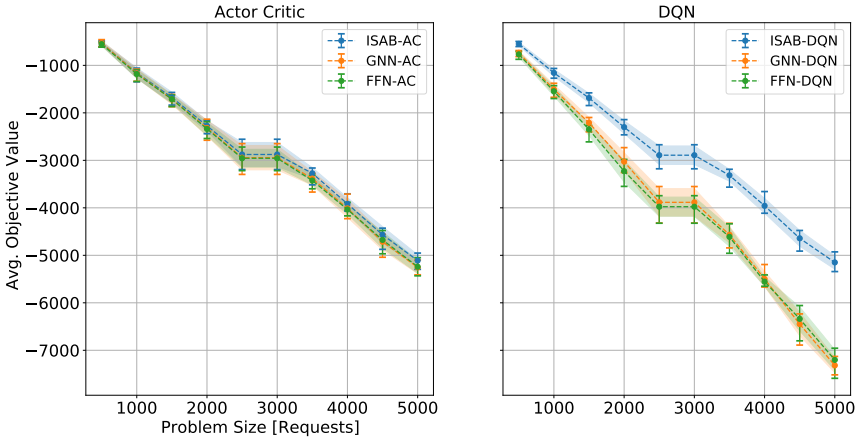

Figure 6.10: Evaluation for S7C5 valued assignment problems. DRL agents only.

One possible remedy to this problem could be to treat the objective and each type of constraint penalty as a distinct objective. Multi-objective reinforcement learning [36] is concerned with adapting RL algorithms to this kind of problem statement. We briefly describe one possible approach in more detail in the Chapter on future work 8.

6.4.4 Solving-Time Comparison

While using Gurobi, the heuristic baselines, and our DRL methods as solvers for the evaluation instances we did not just measure objective value, but also the wall-clock time it took them to solve instances. Figure 6.13 shows the results averaged over all five seeds for the simple assignment problem. Because the solving times partly have very different scales, we plot them using on a logarithmic axis for better visibility. The error-bars show worst (longest) and best (shortest) time across all seeds.

None of the DRL agents are able to solve the instances as quickly as Gurobi, which is faster by more than a factor of ten. This is most likely due to the fact that the simple assignment problem does not pose much of challenge to the methods Gurobi uses.

Since the choice of DRL algorithm (DQN or Actor-Critic) only influences the final module of the neural network used, and since the policy-modules and state-action value module (see Section 5.6.3) are practically identical w.r.t. to their computational complexity, there is no significant solving-time difference at all between DRL algorithms using the same encoder.

What can be observed is that the different encoders lead to different solving-times. Unsurprisingly, the simple FFN encoder that just applies a feed-forward network to the request tokens is the fastest one, whereas the ISAB encoder that also applies induced set attention is slightly slower. The GNN encoders are computationally very expensive. Applying a MAB for each student and each course $U = 3$ times turns out to be quite expensive, which is why the solving-times are so long.

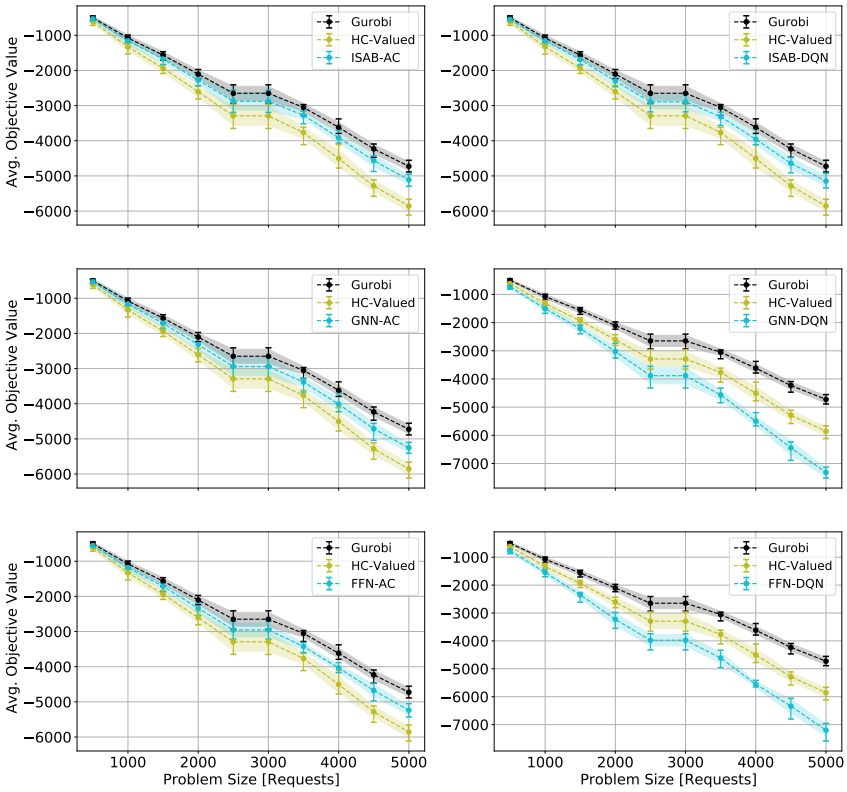

Figure 6.11: Evaluation for S7C5 valued assignment problems. Per-agent comparison to Gurobi and heuristic.

We experimented with two different implementations of the GNN encoder. One of them essentially used a for-loop and applied the MAB operation individually for each student or course respectively and then gathered the results, while the other one performs a single MAB operation that first assumes a fully connected graph, i.e. with all nodes and vertices as queries and keys, but then properly zeroes the attention scores (i.e. query-key similarities in Definition 8 in Section 4.2.3) of unrelated vertices by using the actual adjacency matrix as a mask. The plot shown here gives the running time for the latter version, as it was the more efficient one.

Whereas the simple assignment problem seems to be an easy to solve problem for Gurobi, the time it takes to find optimal solutions for equally sized instances of valued assignment problems is significantly larger, as can be seen in Figure 6.14. The time it takes Gurobi to find solutions is further increased in the S7C5 setting compared to the S5C5 setting.

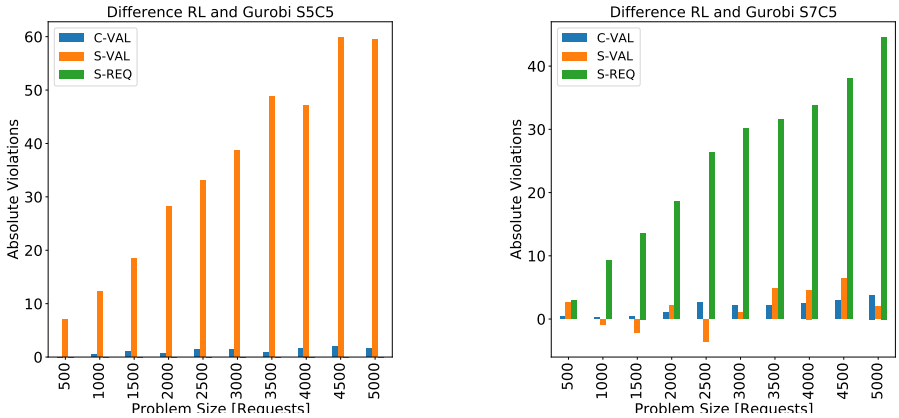

Figure 6.12: Number of constraint violations made by Actor-Critic assignment are subtracted from number of constraint violations made by optimal assignment found by Gurobi.

The time it takes our agents to solve the instances stays roughly time the same, so the gap to Gurobi in terms of solving-time decreases and in the S7C5 setting, two out of three of our methods are indeed faster than Gurobi. These are the ISAB and FFN encoder based agents, whereby FFN encoders are even faster by a factor of 10 on average. Interestingly, the solving-time of Gurobi appears to be subject to a large amount of variance. We believe this is due to the fact that the instances might be of different difficulties, making it easier or harder for Gurobi to reach the optimality gap of 1×10^{-4}.

A systematic problem that might also prevent our approach from being time-efficient is that under some mild assumptions it is currently bound to be subject of at least quadratic time-complexity in the problem size. Assuming a problem size n, then in the case of our MDPs where we add requests one at a time, we have that the length of RL trajectories increases as n increases. If we use for example the total capacity of all courses as n, and assume that all courses have a popularity that is greater than 1, the length of trajectories would be in $\mathcal{O}(n)$. For instances generated using Algorithm 4, this also means trajectory lengths are in $\mathcal{O}(n)$ if n denotes the number of requests. For real world instances, it also a reasonable to assume that the number of requests and total course capacity is relatable, e.g., universities with large enrollment numbers are bound to offer courses with large capacities. So, it is reasonable to assume that the number of environment steps required for solving an instance is in $\mathcal{O}(n)$, where n is the number of requests.

Since some of our neural networks, e.g. policy or state-action value modules, independently process all the n request embeddings, it is also safe to assume that the complexity of choosing one action at the current environment step is in $\mathcal{O}(n)$. So, because we have to perform one such decision per environment step this ultimately means that the time complexity of our approach is at least in $\mathcal{O}(n^2)$. Note that this does not yet factor in the time it actually takes to carry out the action on the environment. However,

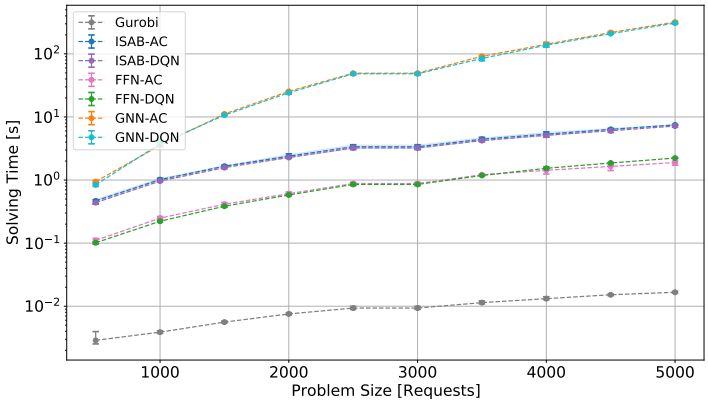

Figure 6.13: Solving times for simple assignment problems

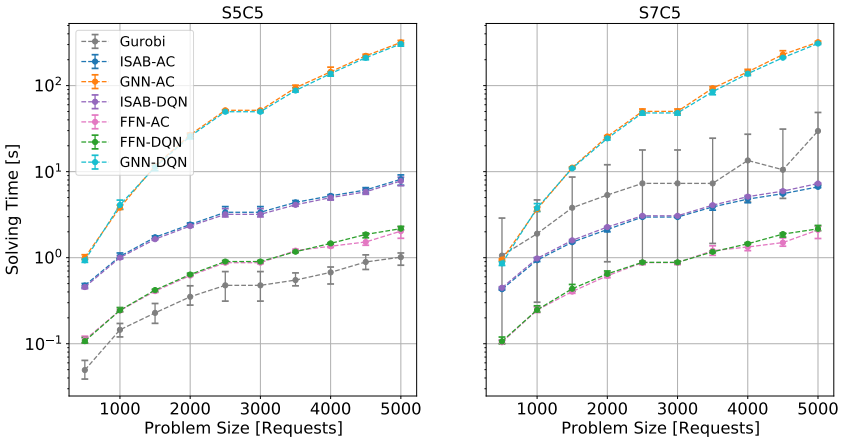

Figure 6.14: Solving times for valued assignment problems

measuring environment time and neural network evaluation time independently shows that the latter dominates the further (for each environment step (selecting an action and executing it within the environment/MDP), about 80% of time spent can be attributed to the neural network evaluation for our fastest method in FFN encoders).

While we do not believe that we can significantly reduce the time complexity of the neural networks we use, we believe it is possible to design a different kind of MDP where the number of required environment steps does not necessarily depend on the problem size. We give an outline on how this could be achieved in Chapter 7. The core idea is to allow actions to make multiple changes to the assignment at once, instead of restricting steps to be minimal (accepting one request at a time).

6.4.5 Generalization KAS 2020

This thesis was originally motivated by the relevancy of the assignment problem for real-world applications, in this case, the task of the KAS to each year decide on how to assign their scholarship students and past members to offered seminars and courses. For this purpose they developed a catalogue of rules, some of which we drew inspiration from in order to define the simple assignment problem in Section 3.2.1. We now use real data about the student requests for courses offered in 2020 to create a corresponding instance of the simple assignment problem and conclude our experiments by evaluating some of our trained agents on this instance. The instance consists of just over 20000 requests, roughly 3000 students, and 281 courses. The total capacity of all courses is 6024.

Note that to our knowledge, the current strategy of the KAS is to search for such assignments manually without any computer-aided, algorithmic assistance. The following experiment investigates our methods and Gurobi as alternative automatic solvers for finding (approximately) optimal assignments to this special instance. The results are shown in table 6.3.

Method	Constraint Violations [%]	Time [s]
ISAB-AC	3.35	64.35
ISAB-DQN	0.50	69.35
FFN-AC	6.41	9.42
FFN-DQN	0.29	10.83
Gurobi	0.00	0.11

Table 6.3: Evaluation of DRL methods and Gurobi on KAS 2020

To put the results into perspective, we do not report the penalized objective value, but instead the number of soft constraint violations normalized by the total number of students that applied for more than one course, so the percentage of students for which the assignments violate a constraint. Surprisingly, as this is not entirely consistent with our results in Section 6.4.2, the DQN agents yield significantly better results than the Actor-Critic agents. The FFN-AC agent, which was the best performing agent when

evaluating on large instance generated from the training distribution, is now performing worse than all other agents, violating more than 6% of all constraints, whereas the best agent, the FFN-DQN agent, violates just 0.29% of constraints. Thanks to Gurobi, we know that a solution optimal w.r.t. to the penalized objective, i.e., a solution feasible w.r.t. constraints, exists. For finding this solution, Gurobi takes slightly more than a tenth of a second, while our agents take between 9 and 70 seconds for finding their respective solutions.

Considering that we did not use any information about the KAS 2020 instance when fitting our training distribution, these are good results, as they allow for the interpretation, that our DRL policies are able to generalize even to instances with different properties. Or it could as well just mean, that this instance is in reality not particularly difficult to solve, apart from being very large.

7 Reinforcement Learning and Gradient Ascent: what's the difference anyway?

Let us start this chapter with a thought experiment. For this, first assume we are presented with an optimization task with parameters θ where the objective is of the form

$$\max_{\theta} J(\theta) \tag{7.1}$$

and J is differentiable w.r.t. θ. If deriving an analytical solution is out of scope, then gradient ascent (see Section 4.2.4) presents an attractive alternative. We can optimize θ by iteratively applying the update scheme

$$\theta^{k+1} \leftarrow \theta^k + \alpha \nabla_{\theta^k} J(\theta^k) \tag{7.2}$$

to some reasonable initial guess θ^0.

For a suitably chosen learning rate α and initial θ^0, the sequence $\theta^0, \theta^1, \dots$ converges to a local maximum [9]. Assuming that J is differentiable essentially means that we assume the existence of an oracle that for any θ gives us the direction of locally steepest ascent in $J(\theta)$.

If the optimization task is an integer program, then gradient ascent in this form is not applicable, mainly for two reasons.

1. Such a gradient oracle does not exist. Integer programming objectives are not differentiable, because the parameters are restricted to be integers.

2. Some parameters in θ are most likely subject to constraints. Even if we knew the direction of steepest ascent, it is not guaranteed that following these directions yields feasible solutions.

We will now discuss how the way we use reinforcement learning can be interpreted as an approach that solves these two issues and by doing so effectively approximates what can be considered a feasible variation of gradient ascent. Strictly speaking, this variation can be even more powerful than normal gradient ascent, in the sense that following the locally steepest ascent is replaced by a method that „thinks more than one step ahead“ and is thus (in theory) able to find global maxima.

Consider an arbitrary integer program with n variables $x = [x_1, \dots, x_n]^T \in \mathbb{N}^n$

$$\begin{aligned} \max_{x} \quad & c^T x \\ \text{s.t.} \quad & Ax \leq b \\ & x \geq 0 \end{aligned} \tag{7.3}$$

In order to stick to our previous notation, we will fix $\theta = x$ and $J(\theta) = c^T x$

Partially generalizing the way we proceeded in Section 5.1, we now construct an MDP $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{T}, s_0, \mathcal{S}^* \rangle$ with

- $\mathcal{S} = \mathbb{N}^n$ (we denote states by $\theta \in \mathcal{S}$),
- $\mathcal{A} \subseteq \mathbb{N}^n$,
- $\mathcal{T}(s, a) = s + a$,
- $\mathcal{R}(s, a) = J(s') - J(s)$ where $s' = \mathcal{T}(s, a)$,
- $s_0 = 0$. [1]
- $\mathcal{S}^*$ are all extreme points of the convex hull of $\{x \in \mathbb{N}^n \mid Ax \leq b\}$. [2]

The actions in this case are a fixed set of directions for moving the parameters $\theta \in \mathcal{S}$. If $\mathcal{A}$ is the standard basis of $\mathbb{N}^n$, i.e., the set of all vectors where exactly one entry is 1, then for each state $\theta \in \mathcal{S}$, a trajectory reaching this state trivially exists.

For a discounting factor of $\gamma = 0$ (only immediate rewards are considered in the return), learning the optimal Q values or optimal policy directly both have interesting interpretations.

The reward for an action a is equal to the increase of the objective when applying that action to the current state. The optimal Q function is thus

$$q^*(s, a) = \mathcal{R}(s, a) + \gamma max_{a'} q^*(\mathcal{T}(s, a), a') \tag{7.4}$$
$$= J(\mathcal{T}(s, a)) - J(s) + 0 \tag{7.5}$$
$$= J(s + a) - J(s) \tag{7.6}$$

So Q-learning with $\gamma = 0$ can be viewed as a prediction task where the goal is to learn to predict the amount of ascent associated with each direction $a \in \mathcal{A}$, i.e., finite difference surrogates for the directional derivative of $J(s)$ along the direction a.

Estimating the optimal policy directly using the policy gradient approach breaks down to a classification task where among a set of possible directions $a \in \mathcal{A}$ we try to classify which of them ascends the objective the most without the need to know how much exactly.

In our case, in Section 5.1, s_0 was part of the feasible region and we additionally constrained the policies (see Section 5.2) s.t. we disable actions that result in infeasible states. By these means, we fixed the second issue mentioned in the beginning of this section.

For $\gamma > 0$, reinforcement learning is theoretically more powerful than gradient ascent, because we no longer care about finding a direction with locally steepest ascent, but instead a whole sequence of directions s.t. that the discounted sum of objective ascents

[1] We restrict this example to cases where the zero vector is feasible, i.e. $0 \in \{x \in \mathbb{N}^n \mid Ax \leq b\}$.
[2] We assume the convex hull is bounded in this case.

is maximal. Optimal policies with sufficiently large γ in this case should be able to find global maxima.

Let us now continue by making an interesting modification to the above MDP. Instead of postulating that $\mathcal{A}$ is the standard basis of $\mathbb{N}^n$ we allow all possible vectors in $\mathbb{N}^n$ to be chosen as *pseudo-continuous* actions. If, for a moment we assume that $J(\theta)$ is differentiable, then gradient ascent would be equivalent to the deterministic policy

$$\pi(s) = \nabla_s J(s), \tag{7.7}$$

and disregarding any possible constraints for a moment, an obvious optimal policy would in any case be the policy with

$$\pi(s_0) = \arg\max_s J(s) \tag{7.8}$$

$$\pi(s) = 0 \qquad \text{for all } s \neq s_0 \tag{7.9}$$

, i.e., the policy that immediately travels to an optimal solution and then stops. Even though this policy most likely is not really something we are going to be looking for in practice (recall the feasibility issues), it illustrates a nice property of policies in the case of pseudo-continuous actions, which is that the length of their trajectories no longer necessarily depends on the size of the problem. In fact, the trajectories for the above policy are always of length one.

This is an interesting insight, as it might allow us to solve the systematic time-complexity limitation of our approach that we discussed at the end of Section 6.4.4. In theory it is possible for a DRL agent equipped with a policy over pseudo-continuous actions to solve instances in any number of steps.

An open question in this case would be how we constrain such policies to generate only actions that lead to feasible states. Our suggestion would be that we do not constrain policies directly, but instead modify the transition function $\mathcal{T}$ s.t. it does not implement $s + a$ directly but instead also projects this result back into the set of feasible states. Constructing a suitable transition function is not trivial at all and would require further work.

Nevertheless, successfully implementing this kind of approach could prove to be beneficial in making DRL approaches for combinatorial optimization more time-efficient, as it resolves the systematic run-time limitation we described earlier.

8 Conclusion and Future Work

The main goal of this thesis was to do a case study to which extent DRL algorithms are applicable to assignment problems as an example for combinatorial optimization problems.

To this end, we reformulated assignment problems as MDPs in such a way that preserves our original notion of optimality and feasibility. As DRL is an inexact method that relies heavily on approximation, we are not able to guarantee that our methods always find optimal policies. However, after implementing suitable neural networks that allow us to use an efficient multi-instance approach built on top of either the DQN or an Actor-Critic algorithm, we were able to show in our experiments that at least some of our considered DRL agents are able to find reasonably good policies in all settings. Nevertheless, there is still room for improvement. A clear limitation of our best performing DRL agents is that they rely on hand-crafted features which we summarized as request tokens, making it more difficult to generalize these approaches to different problem settings. Our graph neural network architectures, which should be easier to generalize to other problems, did not perform as well as the aforementioned architectures during training and evaluation. We believe that this might have been due to our choice of restricting them to recurrent graph neural networks, and it might be worth to investigate more recent graph neural network architectures [35] as alternatives.

Furthermore, while this thesis put a lot of attention into designing suitable neural networks for deep reinforcement learning in the context of combinatorial optimization, an orthogonal approach it did not pursue is to research modifications to the deep reinforcement learning training algorithms themselves. For instance, we regard it as promising to conduct in-depth studies on how different values for hyperparameters such as the discount factor affect solution quality and policy generalizability. Such a study could also investigate more sophisticated DRL algorithms. The ones we used, DQN and a simple Monte-Carlo Actor-Critic are strong DRL baseline algorithms, but the literature offers multiple algorithms that are generally regarded as improvements of them, e.g. PPO [27], Rainbow DQN [14], or AlphaZero [28]. As a simple, first step it could also be promising to just evaluate different reinforcement learning targets. Instead of TD-targets for DQN and Monte-Carlo targets for an Actor-Critic (Section 4.1.2), promising alternatives could be n-step Q-learning targets [29] or TD-λ [29] targets for the Actor-Critic approach, as they allow for a better trade-of between high variance and unbiased Monte-Carlo methods and low-variance, but biased temporal difference methods. Besides looking to existing methods that are generally applicable, more problem-specific approaches could study ways to intelligently choose training data. So far, we always trained on a fixed training distribution, but alternatively one could observe the performance of the agent during training and try to make automatic changes to the training distribution based on this information. For instance, one could explore a form of curriculum learning [3], i.e. start with a distribution of small, simple instances,

where we expect rapid policy improvement, and then gradually increase the complexity and size of the instances. More sophisticated approaches could automate such a process by monitoring the agent's loss or a metric for the slope of his training curve.

Another limiting factor arose as a consequence of the properties of the objective function underlying our reward structures. Because we effectively merged different objectives (some of which are way more prominent than others) by the means of a linear combination, the difficulty of learning to act optimally with respect to the less prominent objectives was increased. Another algorithmic extension to our approach, which be believe could be a remedy to this issue, are multi-objective reinforcement learning approaches such as [36]. They would entail that we decompose the reward function into multiple parts. More specifically, we could treat each component of the penalized objective functions, i.e., the actual objective function and all kinds of constraint penalties, as a separate objective and construct as many dedicated reward signals. A multi-objective DQN could then be instructed to learn a state-action value function for each of these reward signals, or a multi-objective policy gradient approach could consist of a single actor that is criticized by multiple critics, one for each objective component.

Instead of applying an inefficient single-instance approach, i.e., learning polices from scratch for each instance, we used our multi-instance approach which proved to be more time-efficient, as it uses neural networks that are capable of learning a meta-policy just once, which is then applicable to arbitrary instances. Nevertheless, evaluating the respective DRL agents, we observed that they were able to be more time-efficient than Gurobi in only one out of three kinds of assignment problem. A brief theoretical consideration showed that our approach has at least quadratic time complexity, mainly due to the fact that actions are restricted to accepting one request at a time. In Chapter 7 we described an alternative approach, which when applied to our case, would break down into accepting more than one request at once, decreasing episode length and thus potentially also run-time. How exactly this method can be applied also presents an interesting avenue for future work.

Bibliography

[1] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[2] Irwan Bello, Hieu Pham, Quoc V Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940*, 2016.

[3] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 41–48, 2009.

[4] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[5] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

[6] Hanjun Dai, Bo Dai, and Le Song. Discriminative embeddings of latent variable models for structured data. In *International conference on machine learning*, pages 2702–2711. PMLR, 2016.

[7] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7), 2011.

[8] Gabriel Dulac-Arnold, Richard Evans, Hado van Hasselt, Peter Sunehag, Timothy Lillicrap, Jonathan Hunt, Timothy Mann, Theophane Weber, Thomas Degris, and Ben Coppin. Deep reinforcement learning in large discrete action spaces. *arXiv preprint arXiv:1512.07679*, 2015.

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

[10] LLC Gurobi Optimization. Gurobi optimizer python api model.feasrelax, 2021. URL: `https://www.gurobi.com/documentation/9.1/refman/py_model_feasrelax.html`.

[11] LLC Gurobi Optimization. Gurobi optimizer python api model.optimize, 2021. URL: `https://www.gurobi.com/documentation/9.1/refman/py_model_optimize.html`.

[12] LLC Gurobi Optimization. Gurobi optimizer reference manual, 2021. URL: `http://www.gurobi.com`.

[13] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[14] Matteo Hessel, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

[15] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[16] John J Hopfield and David W Tank. "neural" computation of decisions in optimization problems. *Biological cybernetics*, 52(3):141–152, 1985.

[17] Richard M Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972.

[18] Elias Khalil, Hanjun Dai, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. In *Advances in Neural Information Processing Systems*, pages 6348–6358, 2017.

[19] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[20] Juho Lee, Yoonho Lee, Jungtaek Kim, Adam Kosiorek, Seungjin Choi, and Yee Whye Teh. Set transformer: A framework for attention-based permutation-invariant neural networks. In *International Conference on Machine Learning*, pages 3744–3753. PMLR, 2019.

[21] Juho Lee, Yoonho Lee, Jungtaek Kim, Adam Kosiorek, Seungjin Choi, and Yee Whye Teh. Set transformer: A framework for attention-based permutation-invariant neural networks. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 3744–3753. PMLR, 09–15 Jun 2019.

[22] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937. PMLR, 2016.

[23] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[24] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Icml*, 2010.

[25] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory

Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*, pages 8026–8037, 2019.

[26] Boris T Polyak. Some methods of speeding up the convergence of iteration methods. *Ussr computational mathematics and mathematical physics*, 4(5):1–17, 1964.

[27] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[28] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[29] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018. URL: `http://incompleteideas.net/book/the-book-2nd.html`.

[30] Richard S Sutton, David A McAllester, Satinder P Singh, Yishay Mansour, et al. Policy gradient methods for reinforcement learning with function approximation. In *NIPs*, volume 99, pages 1057–1063. Citeseer, 1999.

[31] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[32] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[33] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

[34] Laurence A Wolsey. *Integer programming*. John Wiley & Sons, 1998.

[35] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems*, 2020.

[36] Runzhe Yang, Xingyuan Sun, and Karthik Narasimhan. A generalized algorithm for multi-objective reinforcement learning and policy adaptation. *arXiv preprint arXiv:1908.08342*, 2019.