Reinforcement Learning for Combinatorial Optimization

With application to the fixed charge transportation problem

Master's thesis in Mathematics-Economics

Peter Emil Tybirk - 201407194

Supervisor: Andreas Klose

3rd June 2019

Institute of Mathematics
Aarhus University

Abstract

In this thesis we propose a novel framework for combinatorial optimization with reinforcement learning. To this end, we first cover basic theory on combinatorial optimization, reinforcement learning, function approximation and approximate methods in reinforcement learning. The proposed framework, which may be seen as a kind of hyper-heuristic, is tested on the fixed charge transportation problem, and we validate experimentally that it is capable of discovering effective heuristics. In addition, it provides insights into the efficiency of other heuristic methods. Our experiments indicate that reinforcement learning for combinatorial optimization is a promising research direction.

Furthermore, we develop a new population based iterated random neighbourhood local search heuristic. We test this heuristic on 120 well known instances against the best heuristic from the literature, and in comparable computation time improve the best known solution on 81 of these instances. The heuristic is conceptually simple and easy to implement. Throughout our experiments we in total improve the best known solution on 112 of the 120 considered instances.

Contents

Chapter 1

Introduction

Combinatorial optimization encompasses all problems where an optimal object in a finite collection of objects is sought. A plethora of problems can be described this way, for example subset selection, assignment and sequence construction problems. There exist efficient exact solution methods for many types of combinatorial optimization problems, but there are also problems that seem inherently difficult. For such difficult problems, the pursuit of an optimal solution can be tremendously time consuming and one must often settle for a suboptimal solution. An example of such a problem is the fixed charge transportation problem. The fixed charge transportation problem is a generalization of the transportation problem which is to find the optimal way of shipping a commodity from a number of origins to a number of destinations under the assumption that the cost of shipping from each origin to each destination is proportional to the size of the shipment. In the fixed charge transportation problem, this assumption is relaxed. The cost structure can additionally include a fixed charge, which is incurred regardless of the shipment size. There exist efficient exact solution methods for the transportation problem, but not for the fixed charge transportation problem. Instead, the best methods to solve the fixed charge transportation problem are heuristic; they do not guarantee that an optimal solution is found. In this thesis, we introduce a new framework for discovering efficient heuristic methods to solve combinatorial optimization problems.

The framework we introduce emerges as a fusion between classical heuristics for combinatorial optimization and reinforcement learning. In the last five years there have been startling breakthroughs in reinforcement learning which can be attributed to the development of new techniques, evolution of hardware and the progress in the field of function approximation, specifically deep learning. Reinforcement learning can achieve impressive results on problems which can be thought of as some decision maker (agent) who

has to make decisions (actions) in some environment. For example, reinforcement learning has lead to breakthroughs in board games such as *go* and chess [42], games which are difficult to humans especially due to their combinatorial nature. Inspired by these developments, some interest has recently been directed towards applying reinforcement learning techniques for combinatorial optimization [3, 10]. So far, the methods have had limited success, but the motivation is clear. It appears natural to formulate combinatorial optimization problems in the reinforcement learning framework. In this thesis, we introduce a novel way of doing so. It may be roughly understood as follows: A decision maker observes the current solution of a combinatorial optimization problem and is presented with a number of actions which can change the solution. Based on the experience of the decision maker, an action is chosen and if it improves the best seen solution on the problem, the decision maker is rewarded. The decision maker repeatedly gets to see a number of different instances of the problem and tries to incrementally improve its knowledge in pursuit of reward.

Crucially, the framework we introduce allows us to simulate the decision making process. In many applications, the downfall of reinforcement learning is that it is necessary to continually interact with some environment. If we are considering a decision process where the environment cannot be not simulated, it may be costly to employ a reinforcement learning agent, since the agent needs a lot of experience before it begins to make good decisions. By contrast, the only cost of simulating a decision-making process is computation time.

The techniques we develop find application to most combinatorial optimization problems. We use the fixed charge transportation problem as an example application throughout the thesis. We show that agents trained in the framework we introduce are successful in finding high quality solutions, and when combined with traditional heuristic approaches, are competitive with the best algorithm in the literature. Finally, the knowledge gained through the experiments with reinforcement learning is used to develop a traditional heuristic, which outperforms the current state-of-the-art heuristic for the fixed charge transportation problem, while being conceptually simpler and easier to implement.

1.1 Overview

Chapter 2 introduces the fixed charge transportation problem formally and covers some well-known heuristics as well as a couple of new heuristics we introduce in this thesis. We end it by briefly discussing Conditional Markov

chain search, which the reinforcement learning methods we introduce can be seen as a generalization of. In Chapter 3, we cover the basic concepts of reinforcement learning. Chapter 4 covers basics of function approximation, also known as supervised learning and also contains a section on graph neural networks. In Chapter 5 we cover approximate methods in reinforcement learning, for example deep Q-networks and actor-critic algorithms. In Chapter 6, we finally introduce our new framework for reinforcement learning for combinatorial optimization. In Chapter 7, we present the experiments we conducted, both in the reinforcement learning setup and with metaheuristics. The Appendix contains supplementary material to Chapter 7.

Chapter 2

Combinatorial Optimization

A combinatorial optimization problem is to maximize or minimize some function over a *finite* set of feasible solutions. Formally, it may be stated as follows. Given a finite set, $N = \{1, \ldots, n\}$, a solution may be described as a subset $S \subseteq N$. The set of feasible solutions is some subset $F \subseteq 2^N$ of the power set of N, and the objective function $C\colon F \to \mathbb{R}$ a real-valued function over F. The optimization problem is then to find $S^* \in F$ that maximizes C or to show that $F = \emptyset$, i.e. to find

$$C(S^*) = \max\{C(S) : S \in F\}.$$

This broad definition encompasses a wide variety of problems, such as subset selection, assignment problems and sequence construction.

There are three main classes of algorithms for solving combinatorial optimization problems: Exact algorithms, approximation algorithms and heuristic algorithms. Exact algorithms will, given enough time, find an optimal solution to the problem. Approximation algorithms guarantee that a solution with objective value within some 'distance' of the optimal is found. Heuristic algorithms give no guarantees. Obviously, exact algorithms are preferable if they are efficient, and there are many types of problems for which such algorithms exists, but also many for which they do not. Combinatorial optimization problems can be partitioned into *complexity classes*, and in this thesis we are concerned with problems belonging to the class of $\mathcal{NP}$-complete or $\mathcal{NP}$-hard problems. Such problems can in theory be solved exactly, for example by formulating them as an integer program, but no polynomial time algorithms which guarantee an optimal solution are known [12]. Hence, finding an optimal solution may require a tremendous computational effort. In this case, heuristic methods may be favorable for computing good solutions in a reasonable amount of time. In this thesis, we explore heuristic methods. Throughout the thesis, we use the fixed charge transportation problem as an

example where our methods find application.

2.1 The fixed charge transportation problem

Let $S = \{1, \ldots, m\}$ be a given set of suppliers, each having supply $s_i > 0$ for $i \in S$ and $C = \{1, \ldots, n\}$ a set of customers each having demand $d_j > 0$ for $j \in C$. Without loss of generality, we assume that $\sum_{j \in C} d_j = \sum_{i \in S} s_i$, otherwise an artificial customer and/or supplier can be added to satisfy the assumption. Furthermore, we assume that the cost of shipping from supplier i to customer j consists of a non-negative cost per unit sent, denoted by c_{ij}, and a fixed charge, denoted by f_{ij}, which is incurred if any amount is shipped from supplier i to customer j. We assume, that routes exist between any customer and supplier. The *fixed charge transportation problem* (FCTP) is to find the minimal cost of shipping, such that the demand of each customer is satisfied. It may be stated as follows:

$$\begin{aligned}
\underset{i \in S,\, j \in C}{\text{minimize}} \quad & \sum_{i \in S} \sum_{j \in C} c_{ij} x_{ij} + f_{ij} y_{ij} && && (2.1)\\
\text{subject to} \quad & s_i = \sum_{j \in C} x_{ij} && \forall i \in S, && (2.2)\\
& d_j = \sum_{i \in S} x_{ij} && \forall j \in C, && (2.3)\\
& x_{ij} \leq \min\{s_i, d_j\} y_{ij} && \forall i \in S, j \in C, && (2.4)\\
& x_{ij} \geq 0 && \forall i \in S, j \in C, && (2.5)\\
& y_{ij} \in \{0, 1\} && \forall i \in S, j \in C. && (2.6)
\end{aligned}$$

The objective is to minimize the total cost of shipping (2.1). Constraints (2.2) ensure that the supply is exhausted, (2.3) that demand is met, (2.4) that the shipment from supplier i to customer j is 0 if the route is not open and otherwise no larger than the supply nor the demand, (2.5) that only positive amounts can be sent, and (2.6) that a route is either open or closed, and that fixed costs are incurred accordingly. We will refer to pairs (i, j) as arcs, and the shipment x_{ij} will be referred to as the amount sent on arc (i, j).

In the operations research literature, the FCTP dates all the way back to 1961 where Balinski first formulated it as an integer program [1]. It is a generalization of the (linear) transportation problem, which is the special case of the FCTP where $f_{ij} = 0$ for all pairs (i, j) - this problem is 'easy' - it is well known to be solvable in polynomial time [18]. The introduction of fixed costs,

however, makes the problem inherently difficult. In fact, it is well known, that the FCTP is $\mathcal{NP}$-hard [28, 12].

A solution to an instance of the FCTP can be represented by a bipartite graph with customers on one side and suppliers on the other and edges where $y_{ij} = 1$ with weights x_{ij}. A *basic* solution to the FCTP corresponds to a spanning tree in the bipartite graph. A search for improving solutions can be carried out by introducing an arc not currently in the basis into the basis, which induces a cycle. By increasing the flow on the new arc while 'pushing' flow in the reverse direction until the flow on some other arc is reduced to zero, a new basic solution will appear. This procedure is called a *basic exchange*, and is a key component in this thesis. One could make basic exchanges until there no longer are any which improves the solution, and in this way find a locally optimal solution. Indeed, Dantzig and Hirsch [24] find, that an optimal solution to an instance of the FCTP is a basic solution, but the fixed charge makes the cost function concave and discontinuous and a local minimum is not guaranteed to be global minimum. Hence, it is in theory possible to find an optimal solution by checking all possible basic solutions, which is possible for example by ranking the extreme points according to linear objective function values [36]. This approach is however computationally infeasible, since the number of such basic solutions grows combinatorially with the number of suppliers and customers. Mixed integer programming methods like *Gomory cuts* and *branch and bound* can also be applied to FCTP but have shown to quickly become very computationally costly as the problem size grows [44, 45]. At the current point in time, the most efficient methods for finding good solutions to the FCTP are heuristic methods [16, 5].

2.2 Metaheuristics for combinatorial optimization

In the field of combinatorial optimization, a heuristic is an algorithm which applies a set of rules for constructing or transforming solutions in order to find a good solution. A metaheuristic is a strategy that 'guides' or composes sub-ordinate heuristics in order to find better solutions than these sub-ordinate heuristic alone would be able to find. For example, a sub-ordinate heuristic could be a *local search* (LS) which traverses some 'neighbourhood' $\mathcal{N}(s)$ of a solution s in search for an improving solution and moves to such a solution if possible and otherwise terminates. A metaheuristic could for example be an *iterated local search* (ILS) where a local search is applied until termination, then some mutation of the solution is made, and then again a local search is applied and so forth, see also Algorithm 1. There are an abun-

dance of metaheuristic approaches, but in this thesis we only describe and experiment with a few examples for the fixed charge transportation problem.

Algorithm 1 Iterated local search (ILS)

Generate an initial solution, s_0 and choose a maximum number of iterations with no improvement, $n_{\max}$.

procedure ILS($s_0, n_{\max}$)
  Initialize $n_{\text{fail}} \leftarrow 0$, $s \leftarrow s_0$, $s_{\text{best}} \leftarrow s_0$ and $s_{\text{cur}} \leftarrow s_0$.
  while $n_{\text{fail}} < n_{\max}$ **do**
    $n_{\text{fail}} \leftarrow n_{\text{fail}} + 1$
    $s \leftarrow LocalSearch(s)$
    if $AcceptanceCriterion(s, s_{\text{cur}}, s_{\text{best}}, history)$ **then**
      $s_{\text{cur}} \leftarrow s$
    end if
    if $cost(s) < cost(s_{\text{best}})$ **then**
      $s_{\text{best}} \leftarrow s$
      $n_{\text{fail}} \leftarrow 0$
    end if
    $s \leftarrow s_{\text{cur}}$
    $s \leftarrow Perturbartion(s)$
  end while
  return s_{best}
end procedure

2.2.1 Heuristics and metaheuristics for the fixed charge transportation problem

As mentioned, the most effective methods for finding good solutions to instances of the FCTP rely on metaheuristics. A simple and remarkably efficient heuristic is iterated local search, which we describe in a quite general form in Algorithm 1. The current state-of-the-art heuristic for the FCTP is based on iterated local search guided by reduced cost information [5] in the perturbation phase.

Local search based procedures

Local search procedures for the FCTP are mostly based on basic exchanges, which we introduced above. The neighbourhood is quite large, since you may pick any non-basic arc and make it basic. A **first-accept local search**

(FALS) traverses the candidate arcs and the first one which after a basic exchange improves the solution is introduced into the basis. A **best-accept local search** (BALS) tries all possible basic exchanges and selects the one leading to the largest improvement in objective value. Both of these procedures terminate if no improving solution is found. FALS requires less computation than BALS, but both methods are quite fast. However, a standalone application of any of them will result in a solution which is locally optimal, but likely far from the global optimum.

One way of escaping local minima is by instead applying some threshold accepting procedure. For example, in a **record-to-record travel**, a search akin to a best-accept local search is made, but if no improving solution is found, then the best one found is still accepted, if it is within some threshold, e.g. 10%, of the hitherto best solution. To ensure termination, one could for example let the threshold become smaller and smaller over time.

In this thesis, we introduce another similar local search based approach. Instead of considering the full neighbourhood for each move, we select a *random subset* of arcs, and then move to the best solution found by introducing one of these arcs into the basis, even if it worsens the solution. In this way, the procedure does not naturally terminate, and we may instead for example specify a maximum number of non-improving moves before termination. We call this procedure a **random neighbourhood local search** (RNLS). It turns out to be a quite successful heuristic for the FCTP. The heuristic has some of the traits of variable neighbourhood search [13], and it could also be augmented such that we only move to solutions that do not deteriorate the solution quality too much, akin to be a threshold accepting procedure like record-to-record travel. We present the basic version in Algorithm 2.

Furthermore, we explore tabu-search-like approaches (see e.g. [17]). We forbid some arcs from the current solution based on some (greedy) evaluation measure. The arcs are forbidden by temporarily increasing their costs so much, that they will never appear in a good solution. Then, with these modified costs, a local search procedure is carried out. Finally, the costs are reset, and a local search can for example be applied again. We refer to such a procedure as a **modified cost local search**. We also say that we **kick arcs out** of the solution. We detail some of the arc evaluation measures that can be used in the next subsection.

Arc evaluation measures

For several heuristic methods for the FCTP, it is useful to be able to estimate the 'value' of an arc. The perfect measure would be a Boolean indicating if the arc is present in an optimal solution. Naturally, such a measure is not

available. Instead, we will mostly rely on greedy measures - measures which consider arcs by themselves and not in the context of the full problem. We may even define the measures in context of the current solution, for example to guide a local search along. Let us outline a few greedy approaches.

The first and simplest greedy measure we consider, is simply the cost per unit sent on the arc if the full capacity is used. We call this evaluation measure *greedy1*.

$$greedy1(i,j) = c_{ij} + \frac{f_{ij}}{\min\{si, d_j\}} \tag{2.7}$$

A disadvantage of this measures, is that some arcs may get a poor evaluation, even if they are the cheapest way of sending to one particular customer or from one particular supplier, if the 'general cost' of sending to/from this customer/supplier is high. A way of mitigating this, is to let the arc evaluations somehow be relative to the 'general cost' of the associated supplier/customer. A simple method is to find the k best arcs from each supplier and customer in terms of the greedy value, and then subtract the average of these from the greedy evaluation of each arc. If this is negative, the arc is surely among the k best, otherwise it will be positive, the smaller the better. We use $k = 5$ in our experiments and call this evaluation *greedy2*. Let $\bar{cs}_i^k$ denote the average of the k smallest *greedy*1 evaluations for a fixed supplier i and variable j, and $\bar{cd}_j^k$ the average of the k smallest *greedy*1 evaluations for a fixed customer j and variable i, then

$$greedy2(i,j,k) = greedy1(i,j) - \max\{\bar{cs}_i^k, \bar{cd}_j^k\}. \tag{2.8}$$

Another thing to take into account when evaluating an arc, is whether it uses the full supply of a supplier or meets the full demand of a customer. In this case, fixed costs can be saved, since no more arcs will then be needed for that particular supplier/customer. To take this into account, we could for example add the minimal cost per unit it would cost for the remaining supply/demand for that supplier/customer. We modify *greedy2* such that this is done, and dub this evaluation measure *greedy3*.

$$greedy3(i,j,k) = \begin{cases} greedy2(i,j,k) + \min_{l \neq j} c_{il} + \frac{f_{il}}{s_i - d_j}, & \text{if } s_i > d_j \\ greedy2(i,j,k) + \min_{l \neq i} c_{lj} + \frac{f_{lj}}{s_i - d_j}, & \text{if } s_i < d_j \\ greedy2(i,j,k), & \text{if } s_i = d_j \end{cases} \tag{2.9}$$

An evaluation measure could also be based on reduced-cost information. For example, Buson et al. [5] derive arc evaluation measures based on reduced cost information obtained by solving a series of linear relaxations of the FCTP based on Chvátal-Gomory cuts [6, 19].

Other mutations

The arc evaluation measures we described could be used to guide a modified cost local search. For example by disallowing arcs from the solution which have a bad evaluation by some measure, perhaps in a stochastic fashion. This gives a way of **kicking arcs out** of a solution. Another approach is to **introduce arcs** into a solution. For example, one could choose k random non-basic arcs to introduce into the basis. Instead of doing this at random, arcs could be introduced with probability proportional to their evaluation, i.e. if we let $g_{ij} \geq 0$ denote an evaluation of arc (i, j), then we will introduce (or kick out) arc (i, j) into the basis with probability

$$p_{ij} = \frac{g_{ij}}{\sum_{i \in S, j \in C} g_{ij}}. \tag{2.10}$$

Since we have several ways of evaluating arcs, this also gives rise to several different methods for making a perturbation to a solution. To save computation, the evaluation measures can often be computed once at the beginning of the heuristic procedure. For kicking out arcs from a basis, we set $g_{ij} = 0$ for all non-basic arcs.

A population based method

Based on the above method for making random mutations, we also propose a population based method. If one has constructed n high quality solutions, it is possible make arc evaluations based on their presence in these solutions. For instance, the value of an arc could be set to the average or best objective value for solutions where this arc is a part of the basis. Then an iterated local search starting at each high quality solution could be carried out, where the random perturbations applied use this evaluation measure in accordance with Equation (2.10).

Arcs which are not present in any high of the high-quality solutions from the population should first have equal probability of being selected, but lower than all arcs present in the high quality solutions. This way, we intensify the search around promising arcs. Then, one could also reverse the probabilities to diversify the search.

In this way, we find n new solutions of at least the same quality as the ones we started with. The procedure can then be repeated, i.e. the arc evaluations can be recomputed and another search can take place starting from each of the n solutions. To limit the computation time, one could decrease n for the next run, for example by starting only from the $n/2$ best solutions, or by filtering out solutions in another way, e.g. by keeping diverse solutions -

here one can find inspiration in the literature on genetic and evolutionary algorithms, see e.g. [13].

The population based procedure has turned out to be quite successful for finding good solutions to the FCTP, and a version of it is described in Algorithm 4.

Finding an initial solution

In order to start any of the heuristic procedures we have described an initial solution is needed. There are several ways of obtaining one. A way in which an initial solution of relatively high quality can be found, is to solve the LP-relaxation of the fixed charge transportation problem and use the flows thereby obtained. It is easy to see, that the LP-relaxation of the FCTP can be formulated as a linear transportation problem with variable costs $\bar{c}_{ij} = c_{ij} + f_{ij}/\min\{s_i, d_j\}$. The linear transportation problem can be solved to optimality in polynomial time [37].

Another approach would be to iteratively select arcs based on some evaluation measure or even at random. For example, a solution could be constructed by iteratively selecting the cheapest arc per unit sent until a feasible solution is obtained. In order for the construction procedure not to be deterministic, arcs with evaluation within some bound, say $\alpha = 30\%$, of the best evaluation could be chosen at random. When the evaluation measure used is cost per unit sent we call this approach $RandomGreedy(\alpha)$.

An iterated random neighbourhood local search based heuristic for the fixed charge transportation problem

With the ingredients described above, we are ready to describe a conceptually simple heuristic (Algorithm 3) which yields state-of-the-art results on a number of instances of the FCTP. We use this algorithm as our baseline later in the report.

The dominant subroutine in the heuristic is the aforementioned random neighbourhood local search (RNLS). We describe this procedure fully in Algorithm 2. Furthermore, we use the following **pool of mutations** in the algorithm:

(1) Kick out k arcs with largest cost per unit in the current solution

(2) Kick out k arcs with largest cost according to the *greedy2* evaluation measure.

(3) Kick out k arcs with largest cost according to the *greedy3* evaluation measure.

(4) Introduce k arcs into the basis at random.

(5) Introduce k arcs into the basis according to the *greedy1* evaluation measure

(6) Introduce k arcs into the basis according to *greedy2* evaluation measure.

(7) Introduce k arcs into the basis according to the *greedy3* evaluation measure.

in any case, k is drawn uniformly between 5 and and 20% of the basic arcs.

Algorithm 2 Random neighbourhood local search (RNLS)

Generate an initial solution, s_0 and choose a maximum number of iterations with no improvement $n_{\max}$ and a neighbourhood size, N.

procedure RNLS($s_0, n_{\max}, N$)
  Initialize $improve \leftarrow true$, $i \leftarrow 0$, $s \leftarrow s_0$ and $s_{\text{best}} \leftarrow s_0$.
  while $i < n_{\max}$ or $improve$ **do**
    $i \leftarrow i + 1$
    $improve \leftarrow false$
    Choose $\mathcal{N}_{\text{rand}}(s) \subset \mathcal{N}(s)$ at random such that $|\mathcal{N}_{\text{rand}}(s)| = N$
    $s_{\text{new}} \leftarrow \arg\min_{s' \in \mathcal{N}_{\text{rand}}(s)} cost(s')$
    if $cost(s_{\text{new}}) < cost(s)$ **then** $improve \leftarrow true$
    if $cost(s_{\text{new}}) < cost(s_{\text{best}})$ **then** $s_{\text{best}} \leftarrow s_{\text{new}}$
    $s \leftarrow s_{\text{new}}$
  end while
  return s_{best}
end procedure

2.3 Comparing heuristics

To compare two heuristic methods H_1 and H_2 for solving a combinatorial optimization problem, we solve a series of test problem instances, $I_1, I_2, \ldots, I_n$. Such instances may be seen as a random sample from an infinite population of possible test instances for the particular problem. Considering this randomness, it makes sense to translate the comparison of heuristics to some statistical hypothesis test. To enable this, we need some performance measure to describe the performance of a heuristic on a test instance. Heuristic procedures may themselves be stochastic, and to accommodate this, it is reasonable to make several runs of the algorithm on each test problem instance.

Algorithm 3 Iterated random neighbourhood local search (IRNLS)

Generate an initial solution, s_0 and choose a maximum number of iterations with no improvement $n_{\max}$, acceptance threshold $\alpha > 1$, number of fails before reset β, parameter regulating number of iterations before changing the current solution, $\kappa < \beta$ and parameters for RNLS, n and N.

procedure IRNLS($s_0, n_{\max}, n, N, \alpha, \beta$)
  Initialize $n_{\text{fail}}, n_{\text{cur}}, n_{\text{reject}} \leftarrow 0$ and $s, s_{\text{cur}}, s_{\text{best}} \leftarrow s_0$.
  while $n_{\text{fail}} < n_{\max}$ **do**
    Increment $n_{\text{fail}}, n_{\text{cur}}$ and n_{reject} by 1.
    $s \leftarrow FALS(s)$
    $s \leftarrow RNLS(s, n, N)$
    $s \leftarrow FALS(s)$
    if $cost(s) < cost(s_{\text{cur}})$ or $n_{\text{cur}} > \kappa$ and $cost(s) < \alpha \cdot cost(s_{\text{cur}})$ **then**
      $s_{\text{cur}} \leftarrow s$
      $n_{\text{reject}} \leftarrow 0$
    end if
    if $cost(s) < cost(s_{\text{best}})$ **then**
      $s_{\text{best}} \leftarrow s$
      $n_{\text{fail}}, n_{\text{cur}} \leftarrow 0$
    end if
    if $n_{\text{cur}} \geq \beta$ **then**
      $s \leftarrow s_{\text{best}}$
      $n_{\text{cur}}, n_{\text{reject}} \leftarrow 0$
    end if
    $s \leftarrow Mutation(s)$ ▷ from pool of mutations
  end while
  return s_{best}
end procedure

Algorithm 4 Population based iterated random neighbourhood local search

Choose initial population size n.
procedure PIRNLS(n)
Generate n solutions by the use of IRNLS. Denote the corresponding set of solutions by $\mathcal{S}$.
Set $worst \leftarrow \min_{s \in S} cost(s)$
for all arcs $i \in S$ and $j \in C$ **do**
Set h_{ij} equal to the average value of solutions in $\mathcal{S}$ where arc (i, j) is present. If (i, j) is not present set it to $worst$.
Set $g_{ij} \leftarrow worst - h_{ij} + 1$
end for
Set $p_{ij} \leftarrow \frac{g_{ij}}{\sum_{i \in S, j \in C} g_{ij}}$ for all $i \in S$ and $j \in C$.
for all $s \in \mathcal{S}$ **do**
$s \leftarrow IRNLS(s)$ with mutations based on p_{ij} ▷ Intensify
$s \leftarrow IRNLS(s)$ with mutations based on e.g. $1 - p_{ij}$ ▷ Diversify
end for
return $\arg\min_{s \in \mathcal{S}} cost(s)$
end procedure

This provides several statistics one could compare. For example, one may find it important that the average performance of the heuristic is good, that the worst performance is not too bad or that the best value produced by the heuristic is very good. In any case, we need to be able to describe the performance of a single run on a single test instance in a way which enables statistical hypothesis testing across instances.

2.3.1 Performance measures

The most natural measure for describing performance of a single run on a single test instance is the best objective value observed. A strength of this measure is that people familiar with the problem instance can easily assess whether the objective value is good, since this is a commonly reported statistic. However, it is not necessarily well-suited for comparing heuristics. An improvement in objective value of 100 on one instance may be much more significant than an improvement of 5000 on another. Intuitively, the performance measure should somehow be relative, such that we can speak of percentage improvements - such a measure can be obtained by comparing to a baseline solution. Still, there are instances for which a 1% improvement is more difficult to obtain than for other instances. For example, if the objective

value is dominated by a large invariant which is present in any solution; this could mean that any feasible solution is relatively close (in percent) to the optimal solution. For example, it is apparent that you can add the same constant to all variable costs in the FCTP and still have the same problem up to a constant, but if the constant is very large then the objective value is mostly dominated by this constant, and thus solutions will lie relatively close to each other. We propose a performance measure for which this is (typically) not a problem.

Let $LB(I)$ denote the best lower bound available on the objective value, let $UB(I)$ be some upper bound, e.g. the objective value of the initial solution and let $Z_H(I)$ be the objective value reached by heuristic H on instance I. Then, we define the performance measure

$$RP(H, I) \doteq \frac{Z_H(I) - LB(I)}{UB(I) - LB(I)}. \tag{2.11}$$

Here, any invariant should mostly disappear both in the nominator and denominator (depending on the quality of the lower bound). This performance measure takes values between 0 and 1, if the performance of the heuristic is not worse than the upper bound. A variant of this measure is to use the best available solution from literature instead of the lower bound. This solution may be produced by the heuristic itself, in which case the measure is equal to 0. Otherwise, it is between 0 and 1.

We further examine this performance measure and its usefulness when comparing heuristics in Chapter 7 of this report.

2.4 Hyperheuristics

The term hyperheuristics was first coined by Cowling and Soubeiga [7], who used it to describe the idea of "heuristics to choose heuristics". In contrast to metaheuristics which search in the space of problem solutions, hyperheuristics search in the space of heuristics. The reinforcement learning approach that we consider in this thesis can be considered a kind of hyperheuristic. A recent hyperheuristic approach which we generalize in this thesis, is Conditional Markov chain search [29].

2.4.1 Conditional Markov chain search

In the Conditional Markov chain search (CMCS) framework, the algorithm design is split in two phases: (i) development of algorithmic *components* (actions) such as local search and mutation operators, and (ii) composition of a

meta-heuristic from the available components. Such a composition we call a *configuration*. Phase (ii) is automated by CMCS, phase (i) is not.

The problem of selecting the best performing configuration is difficult, since a configuration may perform well on some instances and on others not. In the original CMCS paper [29], they propose a brute-force-like algorithm to choose the best configuration from the space of all feasible configurations. They restrict to meaningful (what this exactly means is elaborated in the paper, but it is e.g. disallowed to make two consecutive local searches) and deterministic configurations consisting of a fixed number of components. Furthermore, any CMCS algorithm bases its choice of the next component only on the success or failure of the component coming immediately before; in this way such a configuration can actually be seen as a Markov decision process (see Chapter 3) with a state space consisting of a Boolean variable indicating whether or not the most recent used component was a success, and Boolean variables for which component was used. In the CMCS framework, they then make a brute-force search over all deterministic and meaningful policies with a fixed number of components. In this thesis, we look into ways of generalizing this setup to a larger state space and an arbitrary number of components which renders a brute-force search impossible.

Evaluating a configuration

To find the best configuration, we need to find a way to evaluate a configuration. In the CMCS framework, they initially use a small set of seven test instances and short running times to test the algorithm, and they filter out all configurations which are Pareto dominated by some other configuration on these seven instances. Then, all the remaining configurations are tested more thoroughly and they compare them by scaling the objective values to the [0,1] interval and they then choose the configuration which minimizes the sum of scaled objective values obtained on all instances.

Strengths and weaknesses

The biggest flaw of this method is probably the computation time needed. The number of possible configurations grows very quickly with the number of available components. In this setup, it is infeasible to allow more than three different components in a configuration due to computation time. Also, it is difficult to determine what time budget to allow the algorithm and exactly how to scale the objective values.

Furthermore, the markovian structure does not allow the algorithms to include memory beyond the latest state, which is also a significant weakness.

Many successful heuristic approaches are in part based on a memory, e.g. tabu search approaches [17]. Even iterated local search approaches may have something akin to a memory, e.g. if they reset the solution every time no improvement has been found for, say k, iterations. Thus the framework only works really well on problems, where there are a number of individually strong components, which are not memory based.

A strength of the approach is that it is quite simple to understand and implement and it can result in strong heuristics [29]. It does automate the search in the space of heuristics significantly, naturally at the cost of restricting to only a quite limited subspace of heuristics.

2.4.2 From Conditional Markov chain search to reinforcement learning

In the remainder of the thesis, we explore the use of reinforcement learning methods to solve combinatorial optimization problems. It may be enriching to think of it as a generalization of the CMCS framework, even though the idea emerged independently. As mentioned, we may consider the selection of components as a Markov decision process, where in the CMCS framework the selection can only be based upon the latest choice and whether or not that was a success. To find the best configuration (policy), one then tries every feasible configuration which uses at most k components. In the reinforcement learning setup, we instead allow the selection of a component to depend on other features such as current objective value, some history of the search and some problem specific features, such as the percentage of fixed costs in the FCTP. Obviously, it is then impossible to try all meaningful policies, there are simply too many (the state-space is not even necessarily discrete). It is here, that we will use (approximate) reinforcement learning methods to find a good configuration (policy). If we decompose the components down to the simplest form, e.g. a basic-exchange for the FCTP, this in principle also allows automation of phase (1), i.e. component design. However, that setting is significantly more difficult, and we focus most of our attention on the setting where the available components are taken as given. Still, we do spend some pages discussing the other potentially very promising direction as well.

We start by building up some elementary theory of Markov decision processes and reinforcement learning before taking a detour into the field of function approximation, which is used in approximate reinforcement learning which ultimately is what will be used to solve combinatorial optimization problems in this thesis.

Chapter 3

Introduction to Reinforcement Learning

Reinforcement learning (RL) is a computational approach to learning from interaction. Successfully implemented reinforcement learning algorithms is at the core of artificial intelligence which is why many people find the topic so exciting. For many years it has seemed difficult to accomplish anything meaningful in terms of learning within the standard framework of reinforcement learning. Many algorithms are not well understood theoretically or require an abundance of computational resources. But in the recent years, an advance in reinforcement techniques combined with the advance in the field of *deep learning* has resulted in invigorating successes [41, 42].

Still, the nature of the problems which can currently be solved efficiently with reinforcement learning is quite limited. The methods need a stream of incoming data and the opportunity to act and observe the response from the environment. In most practical applications, this is not possible or has not been pursued. However, problems where such data is available, for example in the field of personalized web service, or in fields where the environment can be simulated by a computer simulations, the success of reinforcement learning methods is great. We will in this thesis, describe how combinatorial optimization problems can fit into the reinforcement learning framework.

In this chapter, we aim to provide a brief but clear introduction to basic techniques and results in reinforcement learning. First, we introduce the framework which (sometimes implicitly) underlies most modern reinforcement learning methods namely Markov decision processes (MDPs). We introduce the most fundamental results for MDPs assuming full knowledge of the environment dynamics. Then, we drop this assumption and describe temporal-difference based methods and related concepts.

This chapter is inspired by the newly revised classical reinforcement learn-

ing textbook by Sutton and Barto [46], which is available for free online; the reader may study this for a more comprehensive introduction. We adopt their terminology and notation, but limit out attention to fewer areas of reinforcement learning.

The application of some of these techniques within the field of combinatorial optimization is the main concern of this report, but the discussion of how this is done is postponed mostly to Chapter 6.

3.1 Reinforcement learning terminology

In reinforcement learning, the typical model is a system which can be thought of as one or multiple *agents* who want achieve some goal by interacting with some *environment*. The agent can take *actions* according to its current *policy* to affect the environment in hope of maximizing a (future) *reward*. The agent may not have full information about the environment, but can *learn* about it through interaction. The learning will typically amount in an updated *value function*, which specifies what the agent currently believes is good in the long run. The following are some of the frequently used terms and synonyms, many originating from control theory.

- **Agent**, decision maker, controller.
- **Environment**, dynamic system.
- **Action**, decision, control, move, component.
- **Reward**, (opposite of) cost.
- **Value function**, (opposite of) cost function.
- **Policy**, configuration.
- **Learning**, solving an incompletely known dynamic programming related problem using simulation.
- **Episode**, trial.
- **Episodic tasks**, finite-horizon tasks.
- **Continuing tasks**, indefinite-horizon tasks, infinite horizon tasks.

3.2 Markov decision processes

A Markov decision process (MDP) is a formalization of sequential decision making. We consider an agent that interacts with an environment. At time step $t = 0, 1, 2, \ldots$, the agent receives some representation of the environment's *state* $S_t = s \in \mathcal{S}$, and on that basis selects an *action*, $A_t = a \in \mathcal{A}(s)$, which affects subsequent states and *rewards* obtained by the agent. In the next time step, the agent receives a reward $R_{t+1} = r \in \mathcal{R} \subset \mathbb{R}$ (in some books, this happens at the same time step), and finds itself in a new state, $S_{t+1} = s'$. This continues indefinitely (the *continuing* case) or until some terminal state is reached (the *episodic* case). To be clear, a MDP gives rise to a sequence like:

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, \ldots$$

We only cover *finite* MDPs in the following, which means that $\mathcal{S}, \mathcal{A}$ and $\mathcal{R}$ all have a finite amount of elements. If the MDP is not finite, there exists similar results, but the notation becomes more complex. In addition, it is often possible to quantize the state, action and reward space. The last characteristic of a MDP is its dynamics, defined by a probability distribution, $\mathcal{P}$ over $\mathcal{S}, \mathcal{A}$ and $\mathcal{R}$, which satisfies

$$\mathcal{P}(S_t, R_t \mid S_{t-1}, R_{t-1}, A_{t-1}, \ldots, S_0, R_0, A_0) = \mathcal{P}(S_t, R_t \mid S_{t-1}, A_{t-1})$$

Based on this, we may define a function $p \colon \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times A \to [0, 1]$, such that

$$p(s', r \mid s, a) \doteq \mathcal{P}(S_t = s', R_t = r \mid S_{t-1} = s, A_{t-1} = a).$$

Notice, that p characterizes the dynamics of the environment of the MDP. To summarize, a Markov decision process is a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P})$ where

- $\mathcal{S}$ is the state space,
- $\mathcal{A}(s)$, $s \in \mathcal{S}$ is the set of available actions in state s,
- $\mathcal{R}$ is the reward space,
- $\mathcal{P}$ is a probability measure satisfying:

$$\mathcal{P}(S_t, R_t \mid S_{t-1}, R_{t-1}, A_{t-1}, \ldots, S_0, R_0, A_0) = \mathcal{P}(S_t, R_t \mid S_{t-1}, A_{t-1}).$$

3.2.1 The reward signal and returns

In reinforcement learning and MDPs, the goal of the agent is to maximize the *expected return*. How to precisely define this, may depend on the application.

In any case, the expected return has to do with the rewards that the agent receives from the environment at each time step. One way of defining the expected return, is

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{T-1} R_T \tag{3.1}$$

where $\gamma \in [0, 1]$ is the *discount factor* and $T \in [t, \infty]$ is the, possibly stochastic, termination time. We will refer to G_t as the expected discounted return. Notice, that this notation encompasses both the episodic ($T < \infty$) and the continuing case ($T = \infty$).

Underlying this definition is the *reward hypothesis*; that all which can be thought of as a goal and purpose of some agent can be thought of as maximizing some expected return. It is clear, that it is quite important how the reward signal is constructed. We must provide the agent with a reward signal that is in correspondence with our long term goals. For example, in a board game such as chess or go, a natural reward signal is 1 for the winning move, -1 for losing and 0 in all other states. Importantly, the reward signal should be a way of communicating *what* you want to achieve and not *how* to achieve it.

In many cases, designing a proper reward signal is by no means straightforward. It may not be simple to transform the goal of the agent into a reward signal. In the combinatorial optimization setup, the natural candidate for the long-term goal is to find the best possible solution to one or more combinatorial optimization problems. One way of constructing the reward signal could be by receiving a numerical reward each time a new best solution is found. We will discuss this and several other approaches for designing a reward signal for combinatorial optimization problems in more detail in Chapter 6. For now, let us just highlight one potentially problematic feature with this particular approach. Namely, the problem of *sparse rewards*. Once a very good solution value has been found, the agent may spend much time without finding better solutions, and thus without rewards and it may end up wandering aimlessly between states for long periods of time; this is known as the *plateau problem*.

The sparse rewards problem can for example be tackled with the *shaping* technique introduced by the psychologist B.F. Skinner [43]. The idea is to reinforce successive approximations of the desired behaviour. In the article, it is described, how they tried to train a pigeon to bowl by swiping a wooden ball with its beak. However, this behaviour was apparently too far from typical bird behaviour, and nothing initially happened. They then tried to reinforce (reward) any response with the slightest resemblance to a swipe, at first, merely the behavior of looking at the ball! Soon enough, the bird was

bowling [43]. This example highlights the problem of sparse rewards, but also a possible way of dealing with the issue.

In general, shaping involves changing the reward signal as learning proceeds. At first, the reward signal should not be sparse for the agent's initial behavior - we may reward 'subgoals'. Gradually, the reward signal could then be modified towards the reward signal of original interest. In combinatorial optimization, this could mean giving rewards not only for the final objective value reached, but also intermediate rewards.

3.2.2 Policies and value functions

In reinforcement learning and in particular when dealing with MDPs, the typical end goal is to discover an optimal *policy*, that defines which actions to take and when. Formally, we may define a policy as a probability measure $\pi \colon \mathcal{S} \times \mathcal{A}$ assigning a probability to each action in each state.

The classical way of searching for optimal policies revolve around *value functions*. A value function is an estimate of how good it is for the agent to be in the current state - this naturally depends on the policy! We define the value function of a state s under a policy π as

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi\left[\sum_{i=0}^{T-t} \gamma^i R_{t+i+1} \mid S_t = s\right], \tag{3.2}$$

for all $s \in \mathcal{S}$. We will call v_π the *state-value function* for policy π. Notice that we assume independence of t (homogeneity), which is given from the Markov property in the continuing case, in the episodic case it means that either the distribution of T is memoryless (the time left to termination at time t has the same distribution as the time to termination at time $t+1$) or information on time is also encoded in the state.

Similarly, we may define the *action-value function* for policy π, q_π, as the expected return starting in state s, taking action a and then following policy π:

$$q_\pi(s,a) \doteq \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{i=0}^{T-t} \gamma^i R_{t+i+1} \mid S_t = s, A_t = a\right], \tag{3.3}$$

for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$.

A quintessential property of the value function, is that it satisfies a recur-

sive relationship known as the *Bellman equation*:

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi[G_t \mid S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s', r \mid s, a) \Big[r + \gamma \mathbb{E}_\pi\big[G_{t+1} \mid S_{t+1} = s'\big] \Big] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s', r \mid s, a) \big[r + \gamma v_\pi(s') \big], \text{ for all } s \in \mathcal{S}, \end{aligned} \tag{3.4}$$

where $\pi(a|s)$ is the probability of selecting action a in state s under policy π. The Bellman equation expresses the value of a state in terms of its successor state and is key in the solution methods for MDPs.

3.2.3 Optimal policies

We first introduced $v_\pi(s)$ as the value of some state under a policy. Another way to think about it is as the value of the policy given we start in s. This leads to the definition of an optimal policy.

Definition. *An optimal policy π is a policy such that for all other policies π', $v_\pi(s) \geq v_{\pi'}(s)$ for all $s \in \mathcal{S}$.*

Consequently, all optimal policies share the same state-value function,

$$v_*(s) \doteq \max_\pi v_\pi(s), \text{ for all } s \in \mathcal{S}.$$

We are now ready to derive the *Bellman optimality equation*, which expresses that the value of a state under an optimal policy is the expected return for the best action in that state:

$$\begin{aligned} v_*(s) &= \max_a q_{\pi_*}(s, a) \\ &= \max_a \mathbb{E}_{\pi_*}[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \\ &= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s] \\ &= \max_a \sum_{s',r} p(s', r \mid s, a) \big[r + \gamma v_*(s') \big], \text{ for all } s \in \mathcal{S}. \end{aligned} \tag{3.5}$$

This relationship is fundamental in MDPs and reinforcement learning. If the dynamics of the environment, p, are known, then this relationship also gives rise to a solution method - it is a system of nonlinear equations comprising a variable, $v_*(s)$, and an equation for each state, from which one in principle

can derive v_*. Given v_*, it is straightforward to find an optimal policy; for each state you assign positive probabilities only to the actions that satisfy the Bellman optimality equation. However, this solution approach may not be useful - it relies on three assumptions:

(i) Accurate knowledge of the dynamics of the environment, p.

(ii) Enough computational resources to solve a nonlinear system of equations of size $|\mathcal{S}|$.

(iii) The Markov property.

In this report, we are mainly concerned with assumption (ii), since the state space may be enormous in combinatorial optimization problems. One approach which is more computationally attractive than solving the system of nonlinear equations is that of *dynamic programming*, which we outline next. However, even this approach is computationally infeasible for large enough problems. Still, it is important, since it is the best solution method for smaller problems with known dynamics, but also since it forms a basis of ideas upon which computationally scalable methods are built. One could say, that most reinforcement learning methods attempt to accomplish the same as dynamic programming, but with less computation and without necessarily assuming a perfect model of the environment.

3.2.4 Dynamic programming

In the *dynamic programming* (DP) setup, we assume that the dynamics, p, of the system are known and, in this thesis, also that we are dealing with finite MDPs. One way of finding approximate solutions when these assumptions are violated is to approximate the dynamics and/or to quantize the state and action space. The solution methods we outline is an interplay between *policy evaluation* (also called *prediction*) and *policy improvement*.

Policy evaluation

Policy evaluation (prediction) is the task of determining the value of v_π for a *given* policy π. Instead of solving a system of nonlinear equations to evaluate a policy we can use the Bellman equation (3.4) as basis for iterative computation of the value functions starting from some initial approximation v_0, and

then following the update rule:

$$\begin{aligned} v_{k+1}(s) &\doteq \mathbb{E}_\pi[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s', r \mid s, a) \big[r + \gamma v_k(s')\big], \text{ for all } s \in \mathcal{S}. \end{aligned} \tag{3.6}$$

It is clear from the Bellman equation (3.4) that v_π is a fixed point for this update rule. The sequence v_k can be shown in general to converge to v_π as long as either $\gamma < 1$ or eventual termination is guaranteed from all states under policy π [46]. In contrast to what we will see in methods not assuming knowledge of the dynamics such as TD learning, the update rule here is based on an expectation over all possible next states rather on a sample next state. Accordingly, we call it an *expected update*.

Policy improvement

Evaluating policies is of limited use if we do not also have effective ways of improving them afterwards! For ease of exposition, we limit ourselves to the case of deterministic policies, i.e. $\pi(a|s) = 1$ for exactly one action a in each state, s. The results do extend to the general case of stochastic policies [46]. The key idea to improve policies, is the *policy improvement theorem*.

Theorem. *Let π and π' be any pair of deterministic policies such that*

$$q_\pi(s, \pi'(s)) \geq v_\pi(s), \text{ for all } s \in \mathcal{S}, \tag{3.7}$$

then policy π' is at least as good as π, i.e.

$$v_{\pi'}(s) \geq v_\pi(s), \text{ for all } s \in \mathcal{S}. \tag{3.8}$$

Proof.

$$\begin{aligned} v_\pi(s) &\leq q_\pi(s, \pi'(s)) \\ &= \mathbb{E}\big[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t = \pi'(s)\big] \\ &= \mathbb{E}_{\pi'}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s] \\ &\leq \mathbb{E}_{\pi'}\big[R_{t+1} + \gamma q_\pi(s, \pi'(s))) \mid S_t = s\big] \\ &= \mathbb{E}_{\pi'}\Big[R_{t+1} + \gamma R_{t+2} + \gamma^2 v_\pi(S_{t+2}) \mid S_t = s\Big] \\ &\leq \mathbb{E}_{\pi'}\Big[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 v_\pi(S_{t+3}) \mid S_t = s\Big] \\ &\leq \mathbb{E}_{\pi'}\Big[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \cdots \mid S_t = s\Big] \\ &= v_{\pi'}(s). \end{aligned}$$

□

Now, a straightforward approach to policy improvement is the *greedy* approach, i.e. selecting

$$\pi'(s) = \arg\max_a q_\pi(s, a). \tag{3.9}$$

By construction, this leads to an improved policy according to the policy improvement theorem. Actually, it leads to a strictly improved policy unless the policy we consider is already optimal, since if $v_\pi(s) = v_{\pi'}(s)$ for all $s \in \mathcal{S}$ then

$$\begin{aligned} v_{\pi'}(s) &= \max_a q_\pi(s, a) \\ &= \max_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma v_\pi(s')\big] \\ &= \max_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma v_{\pi'}(s')\big], \end{aligned}$$

for all $s \in \mathcal{S}$ and thus π' satisfies the Bellman optimality equation (3.5) for $v_{\pi'}$, which means it is optimal. Since there are only a finite amount of possible policies, we now have an obvious way of finding one - simply iterating between policy improvement and policy evaluation - we call this procedure *policy iteration* and it is outlined in the box Algorithm 5. Notice, that we use the notation $V(\cdot)$ for approximations of the true value function.

One drawback of policy iteration is that the iterative computations in policy evaluation (3.6) may be computationally costly. Often, this procedure may be truncated without effect on the corresponding greedy policy for policy improvement. A special case of this, is when the policy evaluation is stopped after just one update, and is known as *value iteration*.

The general idea of letting some kind of policy evaluation and policy improvement processes interact, Sutton and Barto [46] refer to as *generalized policy iteration* (GPI), and they claim, that *'almost all reinforcement learning methods are well described as GPI, with the policy always being improved with respect to the value function and the value function always being driven toward the value function for the policy.'*.

3.3 Temporal difference and Q-learning

In this section, we turn our attention to a more difficult setting; we drop the assumption of full knowledge of the dynamics of the environment. The methods and terminology we describe in this section are at the heart of reinforcement learning and are key elements also when some knowledge of the environment's dynamics is assumed.

Algorithm 5 Policy iteration

Step 0 (initialization). Initialize $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$. Furthermore, initialize $\epsilon > 0$, a prespecified threshold and $\Delta = 0$.

Step 1 (policy evaluation).

while $\Delta > \epsilon$ **do**
  $\Delta \leftarrow 0$
  for each $s \in \mathcal{S}$ **do**
    $v \leftarrow V(s)$
    $V(s) \leftarrow \sum_{s',r} p(s', r \mid s, \pi(s)) \left[r + \gamma V(s') \right]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
  end for
end while

Step 2 (policy improvement).

stable $\leftarrow$ *true*
for each $s \in S$ **do**
  old-action $\leftarrow \pi(s)$
  $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r \mid s, a) \left[r + \gamma V(s) \right]$
  If *old-action* $\neq \pi(s)$, **then** *stable* $\leftarrow$ *false*
end for
If *stable* **then** return $V \approx v_*$ and $\pi \approx \pi_*$. **Else** go to step 1.

We start out by describing temporal difference (TD) methods for prediction (estimating the value function) and then their extensions, Q-learning and Sarsa for control (finding an optimal policy). The methods described in this chapter are only directly applicable when the state space is rather small, but the methods used when the state space is large are basically the same, the difference being that the value function is a parameterized approximation - we describe this setting in Chapter 5.

3.3.1 Temporal difference learning

Temporal difference methods learn directly from experience gained from interacting with the environment. This interaction may in many cases be simulated on a computer, which is indeed the case for combinatorial optimization problems. The methods rely on bootstrapping - updating the estimate of the value of one state based on the estimated values of other states. In general, a

TD update has the form

$$V(S_t) \leftarrow V(S_t) + \alpha\big[\hat{v}_\pi(S_t) - V(S_t)\big] \tag{3.10}$$

where $\hat{v}_\pi(S_t)$ is a sample-based estimate of the state-value function, also called the *target*, and α is a parameter called the *learning rate*, which governs the size of the updates. In the simplest case, the estimate of the state-value function is just a one-step look-ahead

$$V(S_t) \leftarrow V(S_t) + \alpha\big[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\big]. \tag{3.11}$$

Here, we bootstrap the estimate of $V(S_t)$ from our estimate of $V(S_{t+1})$ by using $\hat{v}_\pi(S_t) = R_{t+1} + \gamma V(S_{t+1})$ as the target of our update. We basically use a sample of (an estimate) of the value function as our target. This estimate can be augmented by including several successive states when constructing the update target, for example, the n-step update target is

$$G_{t:t+n} \doteq R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n V_{t+n-1}(S_{t+n}) \tag{3.12}$$

where we have added subscripts on $V(S_{t+n})$ which indicate at which time step these values are actually available. A n-step update would hence look like

$$V_{t+n}(S_t) \leftarrow V(S_t) + \alpha\big[G_{t:t+n} - V_{t+n-1}(S_t)\big]. \tag{3.13}$$

Crucially, we have to wait n timesteps to update our estimate of the value of S_t. Going further, a *Monte Carlo* update uses the full return from the episode G_t as defined in Equation (3.1) as the target. Empirically, an n-step update typically allows for faster convergence [46].

It is far from obvious, that TD learning is theoretically sound. It seems almost too good to be true, that you can learn one guess from the next. Still, there is some theoretical backup for the approach; for any fixed policy the one-step TD update has been proven to converge to the true value function in the mean [48] if the learning rate decays according to classical conditions from stochastic approximation theory, namely

$$\sum_{n=1}^{\infty} \alpha_n(a) = \infty \quad \text{and} \quad \sum_{n=1}^{\infty} \alpha_n^2(a) < \infty \tag{3.14}$$

where $\alpha_n(a)$ is the learning rate for the nth selection of action a. For example, one theoretically might choose $\alpha_n(a) = 1/n$. In practice, there is a tradeoff between fast and guaranteed convergence. These convergence proofs generalize to the n-step case and the linear function approximation case, but they do not generalize to the general function approximation case [46].

3.3.2 Sarsa and Q-learning

In reinforcement learning, the end goal is most often to find an optimal policy rather than to estimate the state value function. In other words, we are interested in the *control problem*. There is not that big of a difference between the prediction and the control problem; the main one is that we instead of the state-value function focus on the *action-value function* as defined in equation (3.3). The most famous method for the control problem must be that of Q-learning [53]. Another prominent method is Sarsa [39]. In their simplest form, they are both extensions of the one-step TD update. The two methods belong to two different paradigms; *on-policy* and *off-policy* learning. The difference is easiest understood by outlining the methods. In Q-learning, the action-value function is updated by

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\big] \tag{3.15}$$

whereas the Sarsa algorithm makes the update

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\big[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\big]. \tag{3.16}$$

The subtle difference is, that the Q-learning update always updates according to a greedy forward-look disregarding which action is actually taken (for example the actual policy may sometimes select a random action), thus it is an off-policy method. On the other hand, the Sarsa algorithm always updates according to the actual action taken. This seemingly simple difference has large implications for convergence proofs; on-policy methods are most well-understood theoretically in the function approximation case [46], but in the recent years the most famous applications of reinforcement learning have used off-policy methods [41, 42, 35]. In this light, we mainly focus on off-policy based approaches. The one-step Q-learning algorithm is outlined in Algorithm 6; to fully understand it we first take an important detour and discuss one of the most difficult challenges in reinforcement learning.

3.4 Exploration versus exploitation

A notoriously difficult part of reinforcement learning is to balance the trade-off between *exploration* and *exploitation*. On one hand, it seems like a good idea to exploit the knowledge the agent already has and choose actions which so far have been producing reward. On the other hand, the agents estimate of the reward and long-term value of a given action may be inaccurate. Hence, the agent needs to *explore* actions which are either underused or even actions which have been directly unpromising so far.

An illuminating example is the following. Consider a very simple problem with only one state and two actions. The agent has no prior information. One action, a_1, gives a reward of 1 with probability 1. The other action, a_2, has probability 0.01 of yielding a reward of 1.000.000, but probability 0.99 of yielding a reward of -1. The agent may at first try both actions, and for the first many tries, a_2 may give negative reward and the agent may discard this action completely. Knowing the dynamics of the environment, this is a huge mistake. But seeing a reward of -1 many times in a row may be quite discouraging. In practice, the examples are typically not this extreme, but the underlying problem exists. The agent *must* explore a variety of actions but still progressively favor those that appear to be best.

A similar tradeoff also exists in the field of metaheuristics. Consider again the fixed charge transportation problem; here the actions could consist of basic exchanges or composite mutation operators. The agent may have to choose seemingly unattractive actions to come closer to a neighbourhood in which good solutions exist. In the metaheuristics literature, the terms used are *intensification* (akin to exploitation) and *diversification* (akin to exploration).

The simplest and perhaps most common way of enforcing exploration is the ϵ-greedy strategy. At each time point, the action chosen is random with probability ϵ and otherwise greedy with respect to the current action-value function. Often, one may start with a large value of ϵ and have it decay over time. Without any prior knowledge, there is no exploration strategy which is guaranteed to perform better, but an abundance of heuristic strategies exist [33]. For example, one may choose each action with probability being a function of its estimated value. Another heuristic is optimistic initialization - here the action-value function is initialized optimistically high and thus a greedy strategy will at some point explore the hitherto unexplored actions. We return to this topic in Chapter 5.

3.5 Introduction to approximate methods

All the methods outlined above for solving MDPs, both when the dynamics of the environment are known (dynamic programming) and unknown (TD-learning and variants), suffer from the *curse of dimensionality*. When the state space becomes large, the methods fail - the methods are simply too computationally demanding. One reason is that the algorithms make no generalization across states. They only update the value function for states already 'visited', and preferably visited many times. For combinatorial optimization problems, for example, this approach is totally infeasible - if we were able

Algorithm 6 Q-learning

Initialize $Q(s, a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$ and $\alpha \in (0, 1]$
for each episode **do**
  Observe the initial state, S_0 and set $t = 0$.
  while S_t is not terminal **do**
    Choose A_t from $\mathcal{A}(S_t)$ using a policy derived from Q (e.g. ϵ-greedy)
    Execute A_t and observe R_{t+1} and S_{t+1}
    $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\big]$
    $t \leftarrow t + 1$
  end while
end for

to visit all states we could solve the problem by brute force. Luckily, it is in many applications possible to make some kind of generalization of the value function across states. In many applications it is natural to treat the state space as a feature space with a potentially large amount of both real-valued and categorical features and then use some parametric function approximator which takes as input these features and outputs an estimate of the value of the state.

It is impossible to find a function which approximates the value function for all states well. Instead, we try to find a function which approximates the values well for states which are reached with a reasonably high probability by simply putting more effort into learning to make good decisions in frequently encountered states.

In the next chapter, we present some of the basic function approximation theory and in Chapter 5 we weave these methods together with the reinforcement learning methods described in this chapter. But first, a word of warning.

3.6 Difficulties in reinforcement learning

The ideas behind and the prospects of reinforcement learning are great - it is truly fascinating what has already been achieved with these techniques. Still, one should not get too excited yet. The methods are very sensitive to various hyperparameters and often get stuck in local minima or maxima. Furthermore, the applications are so far limited to problems where you can generate *a lot* of interaction with some environment - the methods do not seem to be very sample efficient. Also, the Markov assumption is in many cases unreasonable.

Theoretically, we struggle to give many guarantees on the prospects of learning. The results regarding convergence to an optimum require that we visit all states an infinite amount of times, which means that we in practice cannot guarantee optimality. A fundamental difficulty, is that what the agent learns as the value of a state is based on bootstrapping; it is always conditional on the current policy and the estimates of values of other states. When these changes, much of what has been learned may turn out to be wrong. This is one of the reasons as to why we require an enormous amounts of samples to learn from.

Two other fundamental difficulties in reinforcement learning are the balance of exploration vs. exploitation and long term credit assignment. With regards to long term credit assignment, the problem is that the agent is basically playing the lottery at every time step and then try to figure out what it did or did not do to hit the jackpot. The reward given at some time point is typically a result of many actions over many time steps possible mixed with some randomness but we still want to represent the proper value of all states/actions in our value function approximation - this is the problem of credit assignment. A critical construction in this regards is the reward signal. We want the agent to maximize long-term rewards, but it may be tempted to go for the immediate rewards.

Some of these issues may be alleviated by having an accurate model of the environment available. To be able to actually make a forward look and deterministically calculate what will happen reduces the margin for error greatly. In some cases, such a model can in part be obtained. For example, if the actions in the fixed charge transportation problem are basic exchanges, then we can calculate exactly what the next state of the environment is for any action, and perhaps utilize lookaheads to make good actions. Here, the problem is not knowledge of the dynamics of the environment, but the sheer amount of computation needed due to the enormous state space. As mentioned, we resort to parametric value function approximation in this case.

Despite all these potential difficulties, reinforcement learning methods have been successful for many tasks, and we also find successful reinforcement learning methods for solving the fixed charge transportation problem. These methods rely on function approximation techniques, which we will now cover the basics of.

Chapter 4

Function approximation

In this chapter, we introduce concepts from the vast field of *function approximation*, also called *supervised learning* or *imitation learning*. Much of the recent success in reinforcement learning can, at least in part, be attributed to the great advance in the field of function approximation in the last ten years. As touched upon in the previous chapter, function approximation is imperative in reinforcement learning when the state and action space is large. The idea is to utilize that many states may have similar features, and hence their values are also closely related. It is in many cases reasonable to assume that there exists a function, f, parameterized by some vector $w \in \mathbb{R}^d$ such that $f(s, w) \approx v_\pi(s)$ for many $s \in \mathcal{S}$. In this chapter, we discuss the methods for approximating such a function and some of the challenges it brings. The topic is vast and expanding quickly, the reader is referred to [22, 20] for a more comprehensive introduction.

4.1 The function approximation problem

The function approximation problem, is to find an approximation of some unknown *target function*, $f\colon \mathcal{X} \to \mathcal{Y}$, given examples of its behaviour. We assume, that we are in possession of N training examples, $(\boldsymbol{x}_1, y_1) \dots (\boldsymbol{x}_N, y_N)$, drawn (ideally independently) from some implicit probability distribution.

To be able to use a concrete function approximation algorithm, we need first to decide on a *hypothesis set* - which class of functions do we think may resemble the target function. For example, the hypothesis set could consist of linear functions, (boosted) decision trees or neural networks with a certain architecture. Given such a hypothesis set, $\mathcal{H}$, one can then *fit* the training examples by using some *learning algorithm*, to produce some final hypothesis $h \approx f$.

4.1.1 Hypothesis set

The first thing one has to decide before embarking on the journey of function approximation, is which hypothesis set to use. For instance, one may use all linear functions as a hypothesis set, and then the resulting learning algorithm could be linear regression. This hypothesis set has the advantage of being easy to fit to data and the resulting function is easy to interpret. However, the resulting function may be a quite poor fit to data, since linear models are not the most flexible. Much flexibility can still be obtained by engineering the features of the data. By contrast, a hypothesis set such as neural networks with a certain architecture (see Section 4.2) may provide tremendous flexibility. However, they may be much more difficult to fit to data, and are also susceptible to *overfitting*, which means that the performance on the sampled training examples does not necessarily generalize out of sample. Neural networks can however, learn from 'raw' data without feature engineering - e.g. they can learn to recognize faces from pixels of a photo without the need to specify features defining traits of faces. We say, that neural networks can do automatic feature extraction [20].

4.1.2 Objective function

Once a hypothesis set has been selected, one should decide on an *objective function*, also called the *error measure*. How do we decide, if some function h is a good approximation of f? If we are dealing with a regression problem (in contrast to classification) a typical objective is to minimize the squared error between the hypothesis and the target. Ultimately, we want to perform well on unseen data, but this is not available for training. Assuming that the out-of-sample data is drawn from the same distribution as the in-sample/training data, it is a reasonable approach to minimize the in-sample error:

$$E_{\text{in}}(h) = \frac{1}{N}\sum_{n=1}^{N} L(h(\boldsymbol{x}_n), f(\boldsymbol{x}_n)) = \frac{1}{N}\sum_{n=1}^{N} (h(\boldsymbol{x}_n) - y_n)^2. \tag{4.1}$$

We minimize this quantity over all $h \in \mathcal{H}$ by applying some *learning algorithm*. The algorithm will depend on the specific choice of error measure, L, and hypothesis set $\mathcal{H}$.

4.1.3 Learning algorithms

When both a hypothesis set and an objective function has been selected, it is time to apply a learning algorithm. As mentioned, the algorithm depends on the nature of the hypothesis set. For linear models, there are often analytic

expressions [22]. For other methods, such as support vector machines or decision trees, there are specialized algorithms [22]. A quite general method is that of *gradient descent*. For neural networks, one typically uses some variant of (stochastic) gradient descent to fit the data [20, 30]. In principle, it can work as long a gradient of the error function is well-defined which is the case for many hypothesis sets. For reinforcement learning purposes, the incremental nature of the gradient descent learning algorithm is incredibly well-suited. Pseudocode for the vanilla version of *stochastic gradient descent* (SGD) is outlined in Algorithm 7. SGD is in general not guaranteed to even arrive at a

Algorithm 7 Stochastic Gradient Descent (SGD)

Gather training data $\mathcal{D} = \{(\boldsymbol{x}_1, y_1) \dots (\boldsymbol{x}_N, y_N)\}$ and choose initial weights, w and learning rate, α.

procedure SGD($\mathcal{D}, w, \alpha$)
  while stopping criterion is not met **do**
    Shuffle training data
    for $i = 1, \dots, N$ **do**
      $w \leftarrow w - \alpha \nabla_w E_{\text{in}}(x_i, y_i, w)$
    end for
  end while
  return w
end procedure

local minimum in a reasonable amount of time, but still it often finds a very low value of the cost function quickly enough to be useful [20]. A lot of extensions/modifications of SGD have been proposed in the litterature, prominent ones include ADAM [30] and RMSprop [14].

4.1.4 Overfitting

Oftentimes, it is the case that the in-sample error is significantly lower than the out-of-sample error. This can happen even if the test data is drawn from the same underlying distribution as the training data. Let us illustrate that by an example. Assume, that we have drawn 11 data points, $(x_1, y_1) \dots (x_{11}, y_{11})$ where $y_i \sim a_0 + a_1 x_i + a_2 x_i^2 + N(0, \epsilon^2)$, such that the y's are a second degree polynomial of the x's, but with some added noise. Then, if we allow a hypothesis set of all polynomials of degree less than or equal to 10, we can find a perfect fit! There will be some 10th degree polynomial which fits the data perfectly. However, it will most likely generalize horribly out of sample - a second degree polynomial would be much more appropriate. We say, that

the learning algorithm has overfit the data. Intuitively, it has fitted more than can be warranted by data. It has fitted the noise term. We talk of overfitting when the training error is small, but the test error is large in comparison.

Overfitting is a common problem in the field of function approximation, and there are various ways of fighting it. One approach is to stick with a simpler hypothesis set; linear functions are for example much less prone to overfitting the data than deep neural networks - they have a much lower variance, but will likely have a higher bias and may not even fit the training data well. Another approach is to choose a hypothesis set and somehow constrain it to reduce the variance - we call this regularization. To further develop our understanding of the subject, we briefly discuss the bias-variance tradeoff.

The bias-variance tradeoff

The bias variance decomposition that we investigate is based on the squared error measure. Let f be the target function and $h_{\mathcal{D}}$ the approximation of f obtained over the training data $\mathcal{D}$. Then the out of sample error is

$$E_{\text{out}}(h_{\mathcal{D}}) \doteq \mathbb{E}_{\boldsymbol{x}}\big[(h_{\mathcal{D}}(\boldsymbol{x}) - f(\boldsymbol{x}))^2\big]$$

where we the expectation is with respect to future datapoints $\boldsymbol{x}$ drawn from the same distribution as $\mathcal{D}$ was. A reasonable goal is to minimize the expected out of sample error over all realizations of the training data $\mathcal{D}$. This expectation may be decomposed into a bias and a variance term in the following way:

$$\begin{aligned}
\mathbb{E}_{\mathcal{D}}\big[E_{\text{out}}(h_{\mathcal{D}})\big] &= \mathbb{E}_{\mathcal{D}}\Big[\mathbb{E}_{\boldsymbol{x}}\big[(h_{\mathcal{D}}(\boldsymbol{x}) - f(\boldsymbol{x}))^2\big]\Big] \\
&= \mathbb{E}_{\boldsymbol{x}}\Big[\mathbb{E}_{\mathcal{D}}\big[(h_{\mathcal{D}}(\boldsymbol{x}) - f(\boldsymbol{x}))^2\big]\Big] \\
&= \mathbb{E}_{\boldsymbol{x}}\Big[\mathbb{E}_{\mathcal{D}}\big[(h_{\mathcal{D}}(\boldsymbol{x}))^2\big] - 2\mathbb{E}_{\mathcal{D}}\big[h_{\mathcal{D}}(\boldsymbol{x})\big]f(\boldsymbol{x}) + f(\boldsymbol{x})^2\Big]
\end{aligned}$$

We let $\bar{h}(\boldsymbol{x}) \doteq \mathbb{E}_{\mathcal{D}}\big[h_{\mathcal{D}}(\boldsymbol{x})\big]$ to simplify notation. $\bar{h}$ can be interpreted as the 'average' hypothesis produced over all datasets. We continue the calculations,

$$\begin{aligned}
\mathbb{E}_{\mathcal{D}}\big[E_{\text{out}}(h_{\mathcal{D}})\big] &= \mathbb{E}_{\boldsymbol{x}}\Big[\mathbb{E}_{\mathcal{D}}\big[(h_{\mathcal{D}}(\boldsymbol{x}))^2\big] - \bar{h}(\boldsymbol{x})^2 + \bar{h}(\boldsymbol{x})^2 - 2\bar{h}(\boldsymbol{x})f(\boldsymbol{x}) + f(\boldsymbol{x})^2\Big] \\
&= \mathbb{E}_{\boldsymbol{x}}\Big[\mathbb{E}_{\mathcal{D}}\big[(h_{\mathcal{D}}(\boldsymbol{x}) - \bar{h}(\boldsymbol{x}))^2\big] + (\bar{h}(\boldsymbol{x}) - f(x))^2\Big].
\end{aligned}$$

Now, the first term inside the expectation can be interpreted as the variance of hypothesis functions over different data sets, while the last term can be interpreted as the expected bias associated with the hypothesis set. So to minimize the out of sample error, it is not enough to just reduce the bias - one also has to consider the variance. That is, one must choose a hypothesis set which 'on average' produces a good hypothesis, but also such that the variance within the hypothesis set is not too large. The above examples with polynomials illustrate this well. The hypothesis set of 10th degree polynomials has a huge variance with respect to the realization of data compared to the hypothesis set of 2nd degree polynomials.

4.2 Neural networks and deep learning

This section will be a brief introduction to neural networks and deep learning, ending up with a description of the newly developed graph neural networks. Neural networks are a very wide class of function approximators which are often well suited to fit a non-linear function given many training examples. In contrast to most other methods for approximating non-linear functions, neural networks often do not need the features of the input to be handcrafted, but can deal with 'raw' data, such as the pixels in a photograph. The neural networks 'discover' the features of the data themselves, which have led to startling breakthroughs in areas such as object and speech recognition [20].

Neural networks can have an abundance of different architectures. Convolutional neural networks are for example well suited for dealing with pictures and recurrent neural networks for sequential data; for an in depth exposition of such architectures the reader is referred to the detailed book written by Goodfellow et al. [20]. Here, we focus on the quintessential *feedforward neural networks*, also known as *multilayer perceptrons* (MLPs) which are the simplest and most well-known type of neural networks, and graph neural networks which is a recently developed class of neural networks to deal with graph-structured data [2].

4.2.1 Multilayer perceptrons

A multilayer perceptron or feedforward neural network is basically a composition of linear functions and nonlinear *activation functions*. For example, a MLP with four *layers* looks like (we count the input as a layer)

$$NN(\boldsymbol{x}) = f_3(f_2(f_1(\boldsymbol{x}))) = g_3(\boldsymbol{W}_3 g_2(\boldsymbol{W}_2 g_1(\boldsymbol{W}_1 \boldsymbol{x} + \boldsymbol{b}_1) + \boldsymbol{b}_2) + \boldsymbol{b}_3) \qquad (4.2)$$

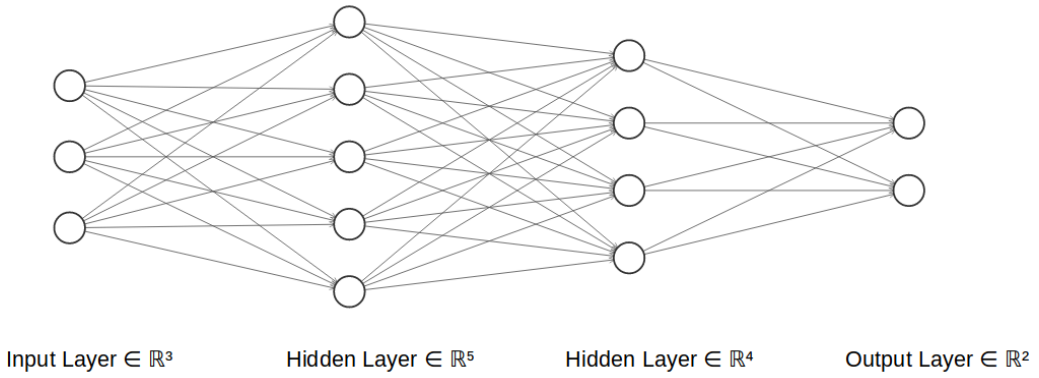

Figure 4.1: A multilayer perceptron with two hidden layers

where $\boldsymbol{W}_1, \boldsymbol{W}_2$ and $\boldsymbol{W}_3$ are weight matrices with fitting dimensions, $\boldsymbol{b}_1, \boldsymbol{b}_2$ and $\boldsymbol{b}_3$ are bias vectors and g_1, g_2 and g_3 are some non-linear functions called activation functions, which are applied elementwise to the input vectors. Popular activation functions include the sigmoid function $\sigma(x) = 1/(1+e^{-x})$ and the rectified linear unit, $\text{RELU}(x) = \max(0, x)$ and many others, each with particular strengths and weaknesses [20].

The reason that these functions are called networks, is that they can be represented as graphs - in the MLP case a directed acyclic graph. For example, we see a four layer neural network represented as a graph in Figure 4.1. The input here has dimension 3 and the output dimension 2. On the edges, the input is multiplied by the corresponding weight from the weight matrix and in the nodes, incoming edges are added together with a bias term and the activation functions are applied. Evidently, this architecture is very flexible. You can have any number of hidden layers with any number of nodes in them and different activation functions. Still, the training of such a network - i.e. fitting the function to data with respect to some loss function - can be done with (stochastic) gradient descent, provided the activation functions are reasonably smooth,. The gradient may simply be derived by the chain rule - and computed efficiently with an algorithm known as *backpropagation*. There are libraries for calculating the gradient *automatically* for general neural network architectures supporting a wide array of activation functions. Libraries such as TensorFlow or PyTorch even support running the calculations on GPUs or TPUs which vastly speeds up the computation.

As a sidenote, the reason they are called *neural* networks is, that the structure is somewhat analogous to the computations of biological neurons in the brain - the idea dates all the way back to a paper by Hopfield from 1982 [26]. Our view of neural networks is, that they are merely a class of functions

which can flexibly approximate many other functions.

4.2.2 Graph neural networks

In the last few years, the study of neural networks for processing graph-structured input has become a top priority in AI research [2, 31, 49, 21]. The reason is, that many real world problems can naturally be represented by a graph. A graph can for example express a relation between entities - the entities being the nodes and relations being the edges. Graphs also naturally describe most combinatorial optimization problems.

Most of the deep-learning approaches explored before graph neural networks (GNNs) follow an 'end-to-end' design philosophy with minimal a priori assumptions on the data [20]. In this way the feature engineering is performed implicitly by the neural network, which has lead to great success, but also requires an abundance of computing resources and data. With graph neural networks, it is attempted to strike a balance between using hand-engineering methods developed in the time when data and computation were at a premium and more modern end-to-end learning methods. This is done by explicitly utilizing some of the inherent information in a given problem by modelling it as a graph.

A key property that may be difficult to model with classical neural networks, but is inherent in graph neural networks is invariance with respect to node and edge permutations - the nodes are invariant to reordering as long as the edges are reordered accordingly [2].

The framework of graph networks (GNs) is quite flexible - for example, they do not have to be neural. We outline formulation of GNs that was introduced by Battaglia et al. [2]. Within the GN framework, a graph is defined as a 3-tuple $G = (\boldsymbol{u}, V, E)$. $\boldsymbol{u}$ is the global attributes of the graph, which may represent for example size, time or objective value for this graph - anything that is not related to a single edge or node. $V = \{\boldsymbol{v}_i\}_{i=1:|V|}$ is the set of node attributes, where $\boldsymbol{v}_i$ is the attribute vector of node i where attributes could for example be supply or demand. Finally, $E = \{\boldsymbol{e}_k, r_k, s_k\}_{k=1:|E|}$ is the set of edges, where each $\boldsymbol{e}_k$ is the attributes of an edge, which could denote variable costs, fixed costs and if it is active or not, r_k is the index of the receiver node and s_k is the index of the sender node. A *graph network block* is a graph-to-graph module which takes as input a graph and outputs a modified graph (where it is possible that the edge and node sets are empty, such that we for example are just left with a number).

The graph network block

A GN block consists of three functions to update edge, node and global attributes and three functions to aggregate edge, node and global attributes respectively.

$$\begin{aligned} \boldsymbol{e}'_k &= \phi^e(\boldsymbol{e}_k, \boldsymbol{v}_{r_k}, \boldsymbol{v}_{s_k}, \boldsymbol{u}) & \bar{\boldsymbol{e}}'_i &= \rho^{e\to v}(E'_i) \\ \boldsymbol{v}'_i &= \phi^v(\bar{\boldsymbol{e}}'_i, \boldsymbol{v}_i, \boldsymbol{u}) & \bar{\mathbf{e}}' &= \rho^{e\to u}(E') \\ \mathbf{u}' &= \phi^u(\bar{\boldsymbol{e}}', \bar{\boldsymbol{v}}', \boldsymbol{u}) & \bar{\mathbf{v}}' &= \rho^{v\to u}(V') \end{aligned} \tag{4.3}$$

where $E'_i = \{(\boldsymbol{e}'_k, r_i, s_k)\}_{k=1:|E|}$, $V' = \{\boldsymbol{v}'_i\}_{i=1:|V|}$ and $E' = \cup_i E'_i$.

The function ϕ^e computes per-edge updates, the function ϕ^v computes per-node updates and ϕ^u computes a global update. The functional form of the ϕ's could be any form you may desire, a popular choice is some kind of MLP [2]. Notice also, that it is not required that the dimension of the attribute vectors are kept unchanged. The ϕ functions each take a set as an input and reduce it to a single element (e.g. a vector) which should represent the aggregated information. Importantly, the ρ functions must be invariant to permutations of their inputs and take a variable number of arguments, such that graphs of different sizes can be taken as input. Examples of ρ functions are mean, maximum and elementwise summation. The procedure typically followed in a GN block is outlined in Algorithm 8.

Algorithm 8 Steps of computation in a full graph network block

function GRAPHNETWORK($E, V, \boldsymbol{u}$)
  for $k = 1, \ldots, |E|$ **do**
    $\boldsymbol{e}'_k \leftarrow \phi^e(\boldsymbol{e}_k, \boldsymbol{v}_{r_k}, \boldsymbol{v}_{s_k}, \boldsymbol{u})$ ▷ Compute updated edge attributes
  end for
  for $i = 1, \ldots, |V|$ **do**
    let $E'_i = \{(\boldsymbol{e}'_k, r_i, s_k)\}_{k=1:|E|}$
    $\bar{\boldsymbol{e}}'_i \leftarrow \rho^{e\to v}(E'_i)$ ▷ Aggregate edge attributes per node
    $\boldsymbol{v}'_i \leftarrow \phi^v(\bar{\boldsymbol{e}}'_i, \boldsymbol{v}_i, \boldsymbol{u})$ ▷ Compute updated node attributes
  end for
  let $V' = \{\boldsymbol{v}'_i\}_{i=1:|V|}$
  let $E' = \cup_i E'_i$
  $\bar{\mathbf{e}}' \leftarrow \rho^{e\to u}(E')$ ▷ Aggregate edge attributes globally
  $\bar{\mathbf{v}}' \leftarrow \rho^{v\to u}(V')$ ▷ Aggregate node attributes globally
  $\mathbf{u}' \leftarrow \phi^u(\bar{\boldsymbol{e}}', \bar{\boldsymbol{v}}', \boldsymbol{u})$ ▷ Compute updated global attributes
  return $(E', V', \boldsymbol{u}')$
end function

It is straightforward to compose GN blocks, such that the output from one is the input to another. Notice also, that the output can be tailored to the demands of the task. You may use edges, nodes, global attributes or any combination of the three as output depending on the task. You may use different types of ϕ functions for nodes, edges and global attributes or no functions at all. Special cases of GNs include message-passing neural networks and non-local neural networks [15, 52] which themselves are umbrellas for many types of neural networks.

Chapter 5

Approximate Methods in Reinforcement Learning

Now that the basics of function approximation are in place we are finally ready to outline some of the methods in approximate reinforcement learning. This chapter is inspired by Chapter 9-13 in the book by Sutton and Barto [46]. We attempted to condense those chapters while also including some of the most recent advances in reinforcement learning. The reader may review those chapters for a more detailed outline as the book is available for free online.

In this chapter, we describe methods that are able to generalize information from a limited subset of visited states to the whole state space by approximating the state/action value function. We focus on cases, where it is possible to represent the state-action space as a feature space, where each state-action pair is associated with a vector $\boldsymbol{x}(s,a) = (x_1(s,a), \dots x_d(s,a))$. Then, we will denote the approximate action-value function parameterized with weight vector $\boldsymbol{w} \in \mathbb{R}^d$ by $\hat{q}(\boldsymbol{x}(s,a), \boldsymbol{w}) = \hat{q}(s,a,\boldsymbol{w})$. We also sometimes use the feature vectors without reference to actions, $\boldsymbol{x}(s) = (x_1(s), \dots x_d(s))$ and finally we use the notation $\hat{v}(s,\boldsymbol{w})$ for the state-value function approximation.

In contrast to supervised learning, we do not have a static set of training examples. Instead, we accumulate data incrementally, and furthermore, the way this data is accumulated is not even static since *the policy changes with the value function*. In other words, the target function we are trying to estimate changes over time and hence data accumulated early on may be misleading. We should thus use learning methods, which work incrementally and mainly focus on the most recent data acquired. Luckily, gradient descent methods fit this description quite well.

There are two main ways of extending standard reinforcement learning

techniques to the approximate case. One class of methods is an extension of the temporal difference based methods, based on state/action-value functions, and the other class of methods rely on a search directly in the policy space and they are called policy gradient methods.

5.1 Approximate temporal difference methods

First, we turn our attention towards the most natural extension of standard reinforcement learning methods. Extensions of the concept of TD methods to the approximate case have been at the heart of many of the recent successes in reinforcement learning [42, 35]. These methods rely on approximation of the state/action-value function.

5.1.1 Prediction objective

To approximate a function, supervised learning methods rely on the concept of a loss function to quantify how accurate or inaccurate an approximation is. A commonly used loss function in regression cases is the mean squared error, which in the reinforcement learning case is called the mean squared value error,

$$\mathrm{VE}(\boldsymbol{w}) \doteq \sum_{s \in \mathcal{S}} \mu(s) \big[v_\pi(s) - \hat{v}(s, \boldsymbol{w}) \big]^2 \tag{5.1}$$

where μ is a discrete probability measure inducing a weighting of the states - often this is chosen to be the uniform distribution, but in cases where some states are inherently more important to approximate well than others, a different weighting can be used. In principle, one could use any loss function from the field of function approximation, but to keep the exposition simple we go through the examples with the mean squared value error and μ as the uniform distribution.

5.1.2 Stochastic gradient descent methods

As already briefly discussed, gradient methods are a natural candidate to minimize the loss function given the incremental nature of the problem. In particular, one could use stochastic gradient descent, and do the following update when obtaining a new observation

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \frac{1}{2}\alpha \nabla \big[v_\pi(S_t) - \hat{v}(S_t, \boldsymbol{w}_t) \big]^2 \tag{5.2}$$

$$= \boldsymbol{w}_t + \alpha \big[v_\pi(S_t) - \hat{v}(S_t, \boldsymbol{w}_t) \big] \nabla \hat{v}(S_t, \boldsymbol{w}_t) \tag{5.3}$$

where α is the learning rate. The shrewd reader might immediately point out, that performing this update is impossible, since we do not have access to the value of $v_\pi(S_t)$. The remedy is to replace $v_\pi(S_t)$ by some approximation. If we denote this approximation by U_t, an update looks like

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \alpha\big[U_t - \hat{v}(S_t, \boldsymbol{w}_t)\big]\nabla\hat{v}(S_t, \boldsymbol{w}_t). \tag{5.4}$$

Here, U_t can in principle be any approximation of $v_\pi(S_t)$, for example the n-step estimate from Chapter 3,

$$U_t = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1}R_{t+n} + \gamma^n\hat{v}(S_{t+n}, \boldsymbol{w}_{t+n}). \tag{5.5}$$

In this case, one has to wait n time steps before an update is carried out. If U_t is an unbiased estimate of $v_\pi(S_t)$ for all t, then $\boldsymbol{w}_t$ is guaranteed to converge to a local optimum under the stochastic approximation conditions (3.14) for decreasing α given a fixed policy [46]. However, the n-step estimate is not in general unbiased, since it depends on the current value of $\boldsymbol{w}_t$. In the case of a linear function approximator, convergence proofs toward an optimum can still be given [46].

One is of course not restricted to the squared error loss function nor to vanilla stochastic gradient descent. Any meaningful differentiable loss function, such as Huber loss [27], and any gradient method from the field of function approximation which just needs a differentiable function and target values can be used in place, such as ADAM [30] or RMSprop [14].

5.1.3 λ-returns

Often, it is difficult to know how to set n in the n-step temporal difference based methods. In many applications, it is simply set to one, since it is the most simple, but for some applications, it may result in very volatile estimates to base your updates on one-step bootstrapping estimates. λ-returns provide a nice way to smooth things out and flexibly include many or all time-steps in the updates.

Recall again the n-step update

$$G_{t:t+n} = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1}R_{t+n} + \gamma^n\hat{v}(S_{t+n}, \boldsymbol{w}_{t+n}),$$

for $0 \leq t \leq T - n$ where T is the termination episode. The idea behind the λ-return, is that an update can be done towards not just any n-step return, but also any average of n-step returns, for example, an update could be done towards one half 3-step return and one half 5-step return. The idea is not limited to just averaging n-step updates for different n, but any update targets one could think of. We focus on averaging of n-step returns here.

The λ-return is one particular way of averaging n-step returns - each step is weighted proportionally to λ^{n-1} where $\lambda \in [0,1)$ is a predetermined constant:

$$G_t^\lambda \doteq (1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}G_{t:t+n}. \tag{5.6}$$

Here $1-\lambda$ is a normalization constant, which ensures that the weights sum to one, since $\sum_{n=1}^{\infty}\lambda^{n-1} = 1/(1-\lambda)$ for $\lambda \in [0,1)$. In the episodic case, all n-step returns following the terminal episode, T, are the same, namely the full return, G_t, see also equation (3.1). Hence, we get the following identity

$$\begin{aligned}
G_t^\lambda &= (1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}G_{t:t+n} \\
&= (1-\lambda)\sum_{n=1}^{T-t-1}\lambda^{n-1}G_{t:t+n} + (1-\lambda)\sum_{n=T-t}^{\infty}\lambda^{n-1}G_t \\
&= (1-\lambda)\sum_{n=1}^{T-t-1}\lambda^{n-1}G_{t:t+n} + (1-\lambda)G_t\sum_{n=0}^{\infty}\lambda^{T-t-1}\lambda^{n} \\
&= (1-\lambda)\sum_{n=1}^{T-t-1}\lambda^{n-1}G_{t:t+n} + \lambda^{T-t-1}G_t
\end{aligned} \tag{5.7}$$

which is certainly easier to calculate than an infinite sum! With this expression, it is also possible to define G_t^λ for $\lambda = 1$, it merely equals the full return G_t. In the other end of the spectrum, $\lambda = 0$ is equivalent to the one-step return. In between, we have weighted averages of all n-step returns, where more weight is put towards the immediate returns if λ is closer to 0.

The λ-return can be used directly as the target for our updates, replacing U_t in equation (5.4). However, we have to wait until the end of an episode before an update is made to the value-function which typically results in slower convergence - especially if episodes are long. Since the agent does not update its beliefs at each time step, it may repeat the same (bad) action indefinitely. In a combinatorial optimization problem, it could for example be the case that the agent for a full episode just repeats doing a local search, even though a local minimum is reached the first time. An obvious idea is to instead use n-step truncated λ-returns. The formula for updates is in this case

$$G_{t,n}^\lambda = (1-\lambda)\sum_{m=1}^{n-1}\lambda^{m-1}G_{t:t+m} + \lambda^{n-1}G_{t:t+n}. \tag{5.8}$$

In comparison to the standard n-step return, we now have a geometrically weighted average of n step returns rather than relying on one single n-step return for the update.

5.2 Q-learning with function approximation

Above, we have only considered the policy evaluation problem. Again, it is not difficult to extend the ideas to the control setting. As in the case without function approximation, we have both the off-policy (Q-learning) and on-policy (Sarsa) approach. For one-step Sarsa, the update is

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \alpha\big[R_{t+1} + \gamma\hat{q}(S_{t+1}, A_{t+1}, \boldsymbol{w}_t) - \hat{q}(S_t, A_t, \boldsymbol{w}_t)\big]\nabla\hat{q}(S_t, A_t, \boldsymbol{w}_t),$$

whereas for one-step Q-learning, it is

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \alpha\big[R_{t+1} + \gamma\max_a \hat{q}(S_{t+1}, a, \boldsymbol{w}_t) - \hat{q}(S_t, A_t, \boldsymbol{w}_t)\big]\nabla\hat{q}(S_t, A_t, \boldsymbol{w}_t).$$

The extensions to the n-step and λ-returns case are straightforward.

Since most successful applications of approximate temporal difference methods are based on the Q-learning approach, we focus on this in the following. The vanilla Q-learning algorithm with function approximation is outlined in Algorithm 9. When a neural network is used as function approximator, the method is best known under the name Deep Q-network (DQN) - an abbreviation we will use throughout the report.

Algorithm 9 Q-learning with function approximation

Initialize $\hat{q}$ with weights $\boldsymbol{w}$ and learning rate $\alpha \in (0, 1]$
for each episode **do**
  Observe the initial state, S_0, and set $t = 0$.
  while S_t is not terminal **do**
    Choose A_t from $\mathcal{A}(S_t)$ using a policy derived from $\hat{q}$ (e.g. ϵ-greedy)
    Execute A_t and observe R_{t+1} and S_{t+1}
    $\delta_t \leftarrow R_{t+1} + \gamma\max_a \hat{q}(S_{t+1}, a, \boldsymbol{w}) - \hat{q}(S_t, A_t, \boldsymbol{w})$
    $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha\delta_t\nabla\hat{q}(S_t, A_t, \boldsymbol{w})$
    $t \leftarrow t + 1$
  end while
end for

5.2.1 Experience replay

One simple idea, which has proven to be effective for stabilizing and perhaps even speeding up learning is experience replay [32]. This idea was for example central for learning human-level control for Atari games by reinforcement learning [35]. The idea is to store experience tuples $(S_t, A_t, R_{t+1}, S_{t+1})$

in memory, and 'replay' these transitions later. They can be used in parameter updates for Q-learning for example

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \alpha\big[R_{t+1} + \gamma \max_a \hat{q}(S_{t+1}, a, \boldsymbol{w}_t) - \hat{q}(S_t, A_t, \boldsymbol{w}_t)\big]\nabla\hat{q}(S_t, A_t, \boldsymbol{w}_t).$$

Notice, that when the experience is 'replayed' the parameters of $\hat{q}$ will have changed, and thus we get a different update than last time. We may even get a different best action in the following state, which is why an off-policy approach such as Q-learning is preferred for experience replay, since our value function is for the current policy.

5.2.2 Additional extensions

In addition to experience replay, the models which have achieved the best results on various benchmarks typically use several heuristic ideas. We will not go into detail, but refer to the paper by Hessel et al. [23] where they provide a nice overview and the paper by Dabney et al. [8] where an additional idea is proposed. In our experiments, we include some of these heuristics by the use of an existing framework.

5.3 Policy gradient methods

Thus far, we have only discussed state-value and action-value methods, the policies revolve around estimates of state or action-value functions. There is another method which first gained attention in the 90's [55, 47]. Instead of a parameterized state/action-value function, these methods revolve around parameterizing the policy directly, and then doing gradient descent in the policy space. Extensions of this idea have in the recent years lead to state-of-the-art performances on multiple reinforcement learning tasks [34, 40]. These methods utilize both state/action-value function and a parameterized policy.

We denote by $\boldsymbol{\theta} \in \mathbb{R}^m$ the policy's parameter vector, and for the policy we use the notation

$$\pi_{\boldsymbol{\theta}}(a|s) = \mathcal{P}\left(A_t = a \mid S_t = s, \boldsymbol{\theta}_t = \boldsymbol{\theta}\right). \tag{5.9}$$

A common kind of parameterization of the policy, is

$$\pi_{\boldsymbol{\theta}}(a|s) = \frac{e^{h(s,a,\boldsymbol{\theta})}}{\sum_{a'} e^{h(s,a',\boldsymbol{\theta})}} \tag{5.10}$$

where $h(s, a, \boldsymbol{\theta})$ is some numerical preference for an action in a state, for example the action-value function. This is called a soft-max in action preferences;

each action has some probability of being taken, which is higher if the actions estimated value is large. This setup automatically enforces some exploration, since all actions have positive probability of being taken. For state/action-value methods, we would typically choose the action with the highest estimated value, with no regard of other actions that are almost equally good - this means that a small change in parameters can lead to a huge switch in the policy. This problem is removed by the soft-max in action preferences.

To optimize the policy, we need some kind of performance measure. This can be defined in several ways, and is often denoted by $J(\boldsymbol{\theta})$. We seek to maximize this quantity over $\boldsymbol{\theta}$, and we do so by approximate gradient ascent, i.e.

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \widehat{\nabla J(\boldsymbol{\theta}_t)} \tag{5.11}$$

where $\widehat{\nabla J(\boldsymbol{\theta}_t)}$ is a stochastic estimate whose expectation approximates the gradient of the performance measure. For our purposes, we choose to focus on a specific performance measure, namely the value of the policy given any start state. We may, without losing significant generality assume that starting state is fixed, and denote it by s_0. Hence, we define

$$J(\boldsymbol{\theta}) \doteq v_{\pi_{\boldsymbol{\theta}}}(s_0). \tag{5.12}$$

The question is then how to find the gradient of this quantity. This is not straightforward - it depends both on action selections, but also on the distribution of states in which the actions are made; both things are affected by $\boldsymbol{\theta}$. The answer is found in the fundamental *policy gradient theorem* [47]. We give a proof which is an extended version of the proof found in [46].

Theorem. *Let μ denote the on-policy distribution under $\pi_{\boldsymbol{\theta}}$ and T the finite stochastic termination time of an episode, then*

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}\left[T\right] \cdot \mathbb{E}_{\pi_{\boldsymbol{\theta}}}\left[q_{\pi_{\boldsymbol{\theta}}}(S, A) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(A|S)\right] \tag{5.13}$$

Proof.

$$\begin{aligned}
\nabla_{\boldsymbol{\theta}} v_{\pi_{\boldsymbol{\theta}}}(s) &= \nabla_{\boldsymbol{\theta}} \left[\sum_a \pi_{\boldsymbol{\theta}}(a|s) q_{\pi_{\boldsymbol{\theta}}}(s, a) \right] \\
&= \sum_a \left[\nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a|s) q_{\pi_{\boldsymbol{\theta}}}(s, a) + \pi_{\boldsymbol{\theta}}(a|s) \nabla_{\boldsymbol{\theta}} q_{\pi_{\boldsymbol{\theta}}}(s, a) \right] \\
&= \sum_a \left[\nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a|s) q_{\pi_{\boldsymbol{\theta}}}(s, a) + \pi_{\boldsymbol{\theta}}(a|s) \nabla_{\boldsymbol{\theta}} \sum_{s', r} p\left(s', r|s, a\right) \left(r + v_{\pi_{\boldsymbol{\theta}}}\left(s'\right)\right) \right] \\
&= \sum_a \left[\nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a|s) q_{\pi_{\boldsymbol{\theta}}}(s, a) + \pi_{\boldsymbol{\theta}}(a|s) \sum_{s'} p\left(s'|s, a\right) \nabla_{\boldsymbol{\theta}} v_{\pi_{\boldsymbol{\theta}}}\left(s'\right) \right]
\end{aligned}$$

Now, we have established a recursive relationship; $\nabla_{\boldsymbol{\theta}} v_{\pi_{\boldsymbol{\theta}}}(s)$ depends on $\nabla_{\boldsymbol{\theta}} v_{\pi_{\boldsymbol{\theta}}}(s')$, and after introducing some notation we will proceed by unrolling the relationship.

We denote by $\mathcal{P}_{\pi_{\boldsymbol{\theta}}}(s \to x, k)$ the probability of going from state s to state x in k steps under policy $\pi_{\boldsymbol{\theta}}$, and we then have the identity

$$\mathcal{P}_{\pi_{\boldsymbol{\theta}}}(s \to x, k+j) = \sum_{s'} \mathcal{P}_{\pi_{\boldsymbol{\theta}}}(s \to s', k) \mathcal{P}_{\pi_{\boldsymbol{\theta}}}(s' \to x, j).$$

Furthermore, to simplify notation, let $\phi(s) \doteq \sum_a \nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a|s) q_{\pi_{\boldsymbol{\theta}}}(s, a)$. We are now ready to unroll the recursive relationship

$$\begin{aligned}
\nabla_{\boldsymbol{\theta}} v_{\pi_{\boldsymbol{\theta}}}(s) &= \phi(s) + \sum_a \pi_{\boldsymbol{\theta}}(a|s) \sum_{s'} p\left(s'|s, a\right) \nabla_{\boldsymbol{\theta}} v_{\pi_{\boldsymbol{\theta}}}\left(s'\right) \\
&= \phi(s) + \sum_{s'} \mathcal{P}_{\pi_{\boldsymbol{\theta}}}(s \to s', 1) \nabla_{\boldsymbol{\theta}} v_{\pi_{\boldsymbol{\theta}}}\left(s'\right) \\
&= \phi(s) + \sum_{s'} \mathcal{P}_{\pi_{\boldsymbol{\theta}}}(s \to s', 1) \left[\phi(s') + \sum_{s''} \mathcal{P}_{\pi_{\boldsymbol{\theta}}}(s' \to s'', 1) \nabla_{\boldsymbol{\theta}} v_{\pi_{\boldsymbol{\theta}}}\left(s''\right)\right] \\
&= \phi(s) + \sum_{s'} \mathcal{P}_{\pi_{\boldsymbol{\theta}}}(s \to s', 1) \phi(s') + \sum_{s''} \mathcal{P}_{\pi_{\boldsymbol{\theta}}}(s \to s'', 2) \nabla_{\boldsymbol{\theta}} v_{\pi_{\boldsymbol{\theta}}}\left(s''\right) \\
&\vdots \\
&= \sum_{x \in \mathcal{S}} \sum_{k=0}^{\infty} \mathcal{P}_{\pi_{\boldsymbol{\theta}}}(s \to x, k) \phi(x)
\end{aligned}$$

Now, consider some arbitrary start state, $s_0 \in \mathcal{S}$. Since T is assumed to be finite, we may define $\eta(s) \doteq \sum_{k=0}^{\infty} \mathcal{P}_{\pi_{\boldsymbol{\theta}}}(s_0 \to s, k)$, which can be interpreted as the (expected) average time spent in state s when starting in state s_0 following policy $\pi_{\boldsymbol{\theta}}$. We then have

$$\sum_{s \in \mathcal{S}} \eta(s) = \sum_{k=0}^{\infty} \sum_{s \in S} \mathcal{P}_{\pi_{\boldsymbol{\theta}}}(s_0 \to s, k) = \sum_{k=0}^{\infty} \mathcal{P}_{\pi_{\boldsymbol{\theta}}}(\{k \leq T\}) = \mathbb{E}\left[T\right].$$

We may further define $\mu(s) \doteq \frac{\eta(s)}{\sum_s \eta(s)}$, which is the on-policy state distribution

under $\pi_{\boldsymbol{\theta}}$. Now, we get

$$\begin{aligned}
\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) &= \nabla_{\boldsymbol{\theta}} v_{\pi_{\boldsymbol{\theta}}}(s_0) \\
&= \sum_{s \in \mathcal{S}} \sum_{k=0}^{\infty} \mathcal{P}_{\pi_{\boldsymbol{\theta}}}(s_0 \to s, k) \phi(s) \\
&= \sum_{s \in \mathcal{S}} \eta(s) \phi(s) \\
&= \left(\sum_{s \in \mathcal{S}} \eta(s) \right) \sum_{s \in \mathcal{S}} \frac{\eta(s)}{\sum_{s \in \mathcal{S}} \eta(s)} \phi(s) \\
&= \mathbb{E}\left[T\right] \sum_{s \in \mathcal{S}} \mu(s) \sum_{a} \nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a|s) q_{\pi_{\boldsymbol{\theta}}}(s, a) \qquad (5.14) \\
&= \mathbb{E}\left[T\right] \sum_{s \in \mathcal{S}} \mu(s) \sum_{a} \pi_{\boldsymbol{\theta}}(a|s) q_{\pi_{\boldsymbol{\theta}}}(s, a) \frac{\nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a|s)}{\pi_{\boldsymbol{\theta}}(a|s)} \\
&= \mathbb{E}\left[T\right] \cdot \mathbb{E}_{\pi_{\boldsymbol{\theta}}}\left[q_{\pi_{\boldsymbol{\theta}}}(S, A) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(A|S) \right]
\end{aligned}$$

which completes the proof. □

Notice, that $\mathbb{E}\left[T\right]$ is merely a constant of proportionality, which is often omitted when writing the theorem, since when applying gradient descent, it is anyway absorbed by the learning rate. Another very important thing to note, is that the policy gradient theorem does not give a direct way of calculating the gradient, since we cannot evaluate the expected value directly. Instead, all policy gradient algorithms are based upon different ways of sampling this expected value and thus the gradient. We outline a couple of these approaches in the coming subsections.

5.3.1 REINFORCE

The original policy-gradient method is REINFORCE [55], which is based on the fact, that $q_\pi(S_t, A_t) = \mathbb{E}_\pi[G_t \mid S_t, A_t]$, and each update is then based on a sample of the full return G_t, i.e.

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha G_t \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}\left(A_t|S_t\right). \qquad (5.15)$$

Obviously, these updates can only be made at the end of an episode, which is often not ideal. Another problem, is that the updates have a high variance [46], since they are based on a single sample of a trajectory following S_t and A_t. A way to combat some of the variance is to introduce a *baseline*. The goal is to reduce variance, while keeping the estimate of the gradient unbiased.

The idea is based on Equation (5.14) and subtracting zero in a clever way. For any function $b(s)$ which does not depend on a, we have

$$\sum_a b(s)\nabla_{\boldsymbol{\theta}}\pi_{\boldsymbol{\theta}}(a|s) = b(s)\nabla_{\boldsymbol{\theta}}\sum_a \pi_{\boldsymbol{\theta}}(a|s) = b(s)\nabla_{\boldsymbol{\theta}}1 = 0$$

and thus

$$\begin{aligned}\nabla_{\boldsymbol{\theta}}J(\boldsymbol{\theta}) &= \mathbb{E}\left[T\right]\sum_{s\in\mathcal{S}}\mu(s)\sum_a \nabla_{\boldsymbol{\theta}}\pi_{\boldsymbol{\theta}}(a|s)q_{\pi_{\boldsymbol{\theta}}}(s,a)\\ &= \mathbb{E}\left[T\right]\sum_{s\in\mathcal{S}}\mu(s)\sum_a \nabla_{\boldsymbol{\theta}}\pi_{\boldsymbol{\theta}}(a|s)(q_{\pi_{\boldsymbol{\theta}}}(s,a) - b(s)) \qquad (5.16)\end{aligned}$$

Now, following the steps in the proof of the policy gradient theorem, we instead end up with the update

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha\big(G_t - b(S_t)\big)\nabla_{\boldsymbol{\theta}}\log\pi_{\boldsymbol{\theta}}\left(A_t|S_t\right). \qquad (5.17)$$

The introduction of such a baseline $b(S_t)$ may reduce variance (it could also increase it). Intuitively, we want the baseline to reflect, that in some states all actions have high values, and we would need a high baseline to 'differentiate' the actions from each other. A natural baseline is the state-value function, which we already discussed ways of learning. Algorithms, which use both some estimate of a state-value and/or action-value function and a policy-gradient algorithm to select actions and update the policy are called *actor-critic methods*.

5.3.2 Actor-critic methods

The first thing, one may notice is that an estimate of the value function can be used not only as a baseline, but also partly in place of the full return G_t which was used for the REINFORCE algorithm. Recall, that the quantity we need an estimate of is $\mathbb{E}_{\pi_{\boldsymbol{\theta}}}\big[q_{\pi_{\boldsymbol{\theta}}}(S,A)\nabla_{\boldsymbol{\theta}}\log\pi_{\boldsymbol{\theta}}(A|S)\big]$. Instead of basing this estimate on the identity $q_{\pi_{\boldsymbol{\theta}}}(S_t,A_t) = \mathbb{E}_{\pi_{\boldsymbol{\theta}}}[G_t \mid S_t, A_t]$ and sampling the full return, one could base it on one-step temporal difference returns $q_{\pi_{\boldsymbol{\theta}}}(S_t,A_t) = \mathbb{E}_{\pi_{\boldsymbol{\theta}}}[R_{t+1} + \gamma v(S_{t+1})]$, where $v(\cdot)$ is the true value function. Now, one can estimate the value based on sampling a single reward instead of the full trajectory, which decreases variance. However, we do not have access to the true value function, and in place we use our estimate of it, which introduces bias into the gradient estimate. Empirically, the reduction is variance seems to be worth the bias [34, 46]. Instead of the one-step temporal difference return, we can of course also use the n-step temporal difference return or truncated λ-returns.

In general, an actor-critic algorithm consists of two models:

- **Actor**: Updates the policy parameters $\boldsymbol{\theta}$ and chooses actions based on $\pi_{\boldsymbol{\theta}}(a|s)$.
- **Critic**: Updates the value function parameters, w and gives 'feedback' on the actions of the actor by providing the update target, based on $\hat{v}(S, \boldsymbol{w})$ and/or $\hat{q}(S, A, \boldsymbol{w}_t)$.

The two models may share some parameters, for example if they are both based on a neural network architecture, one could share parameters of many or all layers but the last, where the actor would typically have a softmax layer (ensuring that what comes out is a probability distribution), whereas the critic could for example have a linear layer.

There are many ways of constructing actor-critic algorithms. For example, with a value function as baseline and the one-step temporal difference return, the updates would be

$$\begin{aligned}\boldsymbol{\theta}_{t+1} &\doteq \boldsymbol{\theta}_t + \alpha \left[G_{t:t+1} - \hat{v}\left(S_t, \mathbf{w}\right)\right] \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}\left(A_t|S_t\right) \\ &= \boldsymbol{\theta}_t + \alpha \left[R_{t+1} + \gamma \hat{v}\left(S_{t+1}, \mathbf{w}\right) - \hat{v}\left(S_t, \mathbf{w}\right)\right] \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}\left(A_t|S_t\right) \end{aligned} \tag{5.18}$$

and the extensions to the n-step case and truncated λ-returns consists merely in changing $G_{t:t+1}$ to $G_{t:t+n}$ and $G^{\lambda}_{t,n}$ respectively. This approach is often called *advantage actor critic*, since $R_{t+1} + \gamma \hat{v}\left(S_{t+1}, \mathbf{w}\right) - \hat{v}\left(S_t, \mathbf{w}\right) \approx q(S_t, A_t) - v(S_t)$ is an estimate of the *advantage function*, $A(s, a) = q(s, a) - v(s)$. We outline the full advantage actor critic (A2C) algorithm with n-step returns in Algorithm 10. To utilize more CPU cores and perhaps stabilize learning, one could initialize several agents at once and then each parameter update could average over all agents. An asynchronous variant of this idea, was done in the paper which popularized the method [34], which they named A3C (asynchronous advantage actor critic) - empirically it has not been verified that the asynchronicity contributed significantly to the results [51], and thus the simpler A2C method is preferred by most.

5.4 Exploration and exploitation

As we already discussed in Chapter 3, the tradeoff between exploration and exploitation is a key topic in reinforcement learning. Often, it is difficult to ensure sufficient exploration. The introduction of parametric value function approximations does not change this fact, but the set of available methods for trying to ensure exploration is a bit different. Of course, the vanilla ϵ-greedy strategy is still available, i.e. choosing a random action with probability ϵ. But the we may also induce exploration by somehow making changes

Algorithm 10 Advantage actor critic with n-step returns

Initialize differentiable policy function $\pi_{\boldsymbol{\theta}}(a|s)$ and value function $\hat{v}(s, \boldsymbol{w})$.
Initialize learning rates $\alpha_{\boldsymbol{\theta}}, \alpha_{\boldsymbol{w}} \in (0, 1)$ and discount factor $\gamma \in (0, 1]$
for each episode **do**
  Set $T = \infty$, $t = 0$ and observe the initial state, S_0.
  Set $\tau = 1 - n$, which is the time whose state's estimate is being updated.
  while $\tau < T - 1$ **do**
    if $t < T$ **then**
      Sample $A_t \sim \pi_{\boldsymbol{\theta}}(a|s)$
      Execute A_t and observe R_{t+1} and S_{t+1}
      If S_{t+1} is terminal, then $T \leftarrow t + 1$
    end if
    $\tau \leftarrow \tau + 1$
    if $\tau \geq 0$ **then**
      $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} R_i$
      If $\tau + n < T$, then: $G \leftarrow G + \gamma^n \hat{v}(S_{\tau+n}, \boldsymbol{w})$
      $\delta_\tau \leftarrow G - \hat{v}(S_\tau, \boldsymbol{w})$
      $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha_{\boldsymbol{w}} \delta_\tau \nabla \hat{v}(S_\tau, \boldsymbol{w})$
      $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha_{\boldsymbol{\theta}} \delta_\tau \nabla \log \pi_{\boldsymbol{\theta}}(A_\tau|S_\tau, \boldsymbol{\theta})$
    end if
  end while
end for

to the value/action/policy-function which enforce exploration. We outline two such approaches in the following.

5.4.1 Entropy in policy-gradient methods

A widely used exploration heuristic which in reinforcement learning dates back to a 1991 paper by Williams and Peng [56] is to include an entropy term in the loss function. The technique is sometimes called *entropy regularization*. The entropy of a probability distribution $\pi_{\boldsymbol{\theta}}(a|s)$ is

$$H(\pi_{\boldsymbol{\theta}}(s)) = -\sum_{a \in \mathcal{A}(s)} \pi_{\boldsymbol{\theta}}(a|s) \log \pi_{\boldsymbol{\theta}}(a|s). \tag{5.19}$$

When all actions have almost the same probability, the entropy will be large, whereas if one action has most of the probability, the entropy will be close to zero. The idea is to subtract the entropy from the loss function is to encourage policies to spread probabilities on several actions. The paper which introduced A3C [34] found that adding entropy regularization was crucial

on several tasks to discourage premature convergence to a suboptimal, almost deterministic policy. With entropy regularization, the one-step temporal difference return update in actor-critic algorithms (see Equation (5.18)) is instead

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \left[R_{t+1} + \gamma \hat{v}\left(S_{t+1}, \mathbf{w}\right) - \hat{v}\left(S_t, \mathbf{w}\right)\right] \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}\left(A_t | S_t\right) + \beta \nabla_{\boldsymbol{\theta}} H(\pi_{\boldsymbol{\theta}}(S_t)),$$

where β is a hyperparameter controlling the magnitude of the entropy regularization.

5.4.2 Noisy networks

A more recent method which has quickly gained popularity due to its efficiency is the use of NoisyNets [11]. The concept is applicable to cases where a neural net is used to approximate the state/action/policy function. The idea is to add stochasticity directly to the functions by randomly perturbing network weights, and then learn the parameters driving this perturbation. In [11] they write, that

> *the key insight is a single change to the weight vector can induce a consistent, and potentially very complex, state-dependent change in policy over multiple time steps – unlike dithering approaches where decorrelated (and, in the case of ϵ-greedy, state-independent) noise is added to the policy at every step.*

In a NoisyNet, one or more of the linear layers,

$$\boldsymbol{y} = \boldsymbol{b} + \boldsymbol{W}\boldsymbol{x}$$

are replaced with a noisy linear layer

$$\boldsymbol{y} = \boldsymbol{b} + \boldsymbol{b}_{\text{noisy}} \odot \boldsymbol{\epsilon}_{\boldsymbol{b}} + (\boldsymbol{W} + \boldsymbol{W}_{\text{noisy}} \odot \boldsymbol{\epsilon}_{\boldsymbol{W}})\boldsymbol{x} \tag{5.20}$$

where $\odot$ denotes the elementwise product and $\boldsymbol{\epsilon}_{\boldsymbol{b}}$ and $\boldsymbol{\epsilon}_{\boldsymbol{W}}$ are random variables of suitable dimensions, and all other variables are parameters to be learned. In the original paper, they let $\boldsymbol{\epsilon}_{\boldsymbol{b}}$ and $\boldsymbol{\epsilon}_{\boldsymbol{W}}$ be independent and normally distributed with mean sampled from a uniform distribution $\mathcal{U}[-\frac{1}{\sqrt{p}}, \frac{1}{\sqrt{p}}]$ where p is the size of the layer, and standard deviation $\frac{1}{2}\sqrt{p}$.

Chapter 6

Reinforcement Learning for Combinatorial Optimization

At last, we are equipped with all the necessary tools to introduce a framework for using reinforcement learning techniques to solve combinatorial optimization problems. To start out this chapter, we first motivate the idea. Then, we describe similar approaches in the literature and discuss the novelty of our approach. Finally, we go into more detail with our framework before exemplifying the idea for the fixed charge transportation problem.

6.1 Motivation

In many real-world applications, it is typically the case that the same optimization problem is solved repeatedly maintaining the same problem structure, but differing in data. For example, one could consider a package delivery company which routes trucks on a daily basis in a given city. For such recurring problems, it seems obvious to learn to exploit the structure of data, but traditional approaches seldom explicitly do. Instead, traditional heuristics often follow some rule-based structure. These rules are handcrafted and often do not exploit anything but the history of performed mutations. For example, in iterated local search heuristics, one sometimes resets the solution to the hitherto best one found and then applies some kind of perturbation to the solution. Determining the rules for when to reset and what kind of perturbation to use is often done by an expert through trial and error. By contrast, we seek to *learn* from experience which mutations to make based on the structure of the solution and the history of the search. This is made possible by formulating combinatorial optimization problems in the framework of reinforcement leaning. Such a formulation appears quite naturally

by considering the environment as the space of possible solutions, the state as a feature vector representing the current solution, the actions as possible mutations to be made to the solution and the rewards as improvements in objective value. By the use of this framework, it is possible to learn an adaptive heuristic (policy) from incrementally acquired experience of actions taken and resulting states.

6.2 Literature review

In recent years, the interest in the use of techniques from machine learning within combinatorial optimization has risen dramatically. Typically, such techniques rely on learning from demonstrations of good behaviour. In fact, most approaches which have been proposed for incorporating machine learning techniques in combinatorial optimization are grounded in the supervised learning paradigm. Most of the suggested approaches are incorporated in some algorithm to alleviate the computational burden [4]. The algorithm may be exact, approximate or heuristic, and incorporating machine learning techniques does not necessarily change this. Typically, supervised learning techniques are used to improve an *already existing* algorithm. By contrast, we pursue an *end-to-end* approach, which means that we construct or learn an algorithm from scratch by the use of methods from reinforcement learning. The reinforcement learning paradigm requires less a priori knowledge of the problem at hand. We rely only on knowledge of actions which can be performed to perturb an existing solution and a way to represent a solution as a feature vector. Such a representation may be very coarse, e.g. only existing of the objective value. It could also consist of which action was performed most recently and whether or not it was a success, as in the Conditional Markov chain search framework. It may also be more detailed and contain features such as the ratio of fixed cost in the fixed charge transportation problem, which may be relevant to the action selection. Finally, a large strength of reinforcement learning in contrast to supervised learning is that an algorithm can be build 'from the ground up' without the need for supervised targets provided by an expert - the agent provides its own targets and incrementally learns from them. Thus, the reinforcement learning approach could in particular be well suited to tackle combinatorial optimization problems which are not already well studied in literature.

In this short literature review, we describe only end-to-end approaches, for a more comprehensive review including approaches where machine or reinforcement learning techniques are used only as subroutines see [4].

The first deep learning based method for end-to-end learning for combi-

natorial optimization was proposed by Vinyals et al. [50] who introduced a pointer network which outputs a permutation of its inputs. The network was trained to solve the travelling salesman problem (TSP) by the use of supervised learning. Hence, the approach requires the access to already solved problems to train on, which in many applications is an enormous limitation.

The first reinforcement learning based approach for combinatorial optimization was suggested by Bello et al. [3], who also focused on the TSP. They used the pointer network suggested by Vinyals et al. [50] and negative tour length as a reward signal, starting out from producing random permutations and gradually learning the parameters of the network using the policy-gradient algorithm REINFORCE with baseline, which we described in Chapter 5. Their results were quite far from the state-of-the-art TSP methods, but still they achieved better results than the supervised learning approach by Vinyals et al. A drawback, is that their method is only applicable for problems in which a solution can in a reasonable manner be described as a permutation of some inputs. The approach is also very sensitive to the structure of the pointer network, and it is unlikely to generalize well.

The approach which is closest to the one we pursue is the one suggested by Dai et. al [10]. They learn greedy algorithms for constructing solutions to three different optimization problems: the minimum vertex cover problem, the maximum cut problem and the travelling salesman problem. Their approach is constructive; they start from an empty solution and greedily add notes until termination. The greedy evaluation is based on a graph neural network `structure2vec` [9] trained by fitted Q-learning [38] with rewards equal to the (negative) objective value of the problem instances. They argue, that for the constructive case, a combination of n-step and fitted Q-learning is more data-efficient than a policy gradient approach. They compare their algorithm to that of Bello et al. [3], which they clearly outperform. In addition, they outperform many simple standard heuristics for the respective problems (for the TSP: Minimum spanning tree, farthest insertion, cheapest insertion, closest insertion, Christofides, nearest neighbor and 2-opt), but they lack comparison to state-of-the-art heuristics. Their technique is, as mentioned constructive - they start from scratch and terminate when a feasible solution has been reached. Such an approach requires an extremely good greedy evaluation to be good on a large scale, which means that the graph neural network has to have capacity to represent the state-value function.

6.2.1 Summary and outline

A common ground for the discussed algorithms, is that the results they achieve are quite far from those of state-of-the-art solvers and that their usefulness

is limited to certain types of problems. In particular, all the earlier frameworks for reinforcement learning for combinatorial optimization rely either on a constructive or permutation based approach. In the permutation based approach, they require that a solution in a reasonable way can be represented as a permutation of its elements. In the constructive approach they start from an empty solution, and their actions are then to add an 'element' to the solution until some stopping criterion is met (e.g. that a feasible solution has been reached).

By contrast, we suggest a method that is mutation based. We start from a feasible solution and move between feasible solutions by applying some kind of mutation to the current solution. Such an approach should be applicable for a wider range of problems, and can also be used in conjunction with a constructive approach. The mutation based approach requires a different setup for rewards, states and actions, which we describe in the next section. We try to keep the description of the setup general in the next section, and afterwards, we exemplify the setup for the fixed charge transportation problem.

6.3 A mutation based framework

In this section, we introduce our framing of combinatorial optimization problems in the reinforcement learning setup. First, we describe the types of actions we consider then the state representation and finally the reward structure.

6.3.1 Actions

We consider two ways one can specify actions in the mutation based setup, based on granularity. The two ways of specifying actions induce different needs for state granularity as well, but can use the same reward structure.

Fine-grained actions

If computational power was not an issue, we would like the actions to be as fine-grained as possible, i.e. a single action should change the solution only slightly. In the fixed charge transportation problem for example, the actions available in each state (solution) could be all possible basic exchanges.

The idea behind decomposing the actions as much as possible is that the agent can discover a good policy purely from experience. When the actions are decomposed, we do not inject bias for strategies we believe to be good

into the agent. Instead, the agent should be able to discover such strategies itself, if they really are good. For example, the second iteration of the Go-engine developed by Deepmind, AlphaZero [42], did not see any human-played games in training, it *only* got experience from self-play. The resulting engine ended up being far superior to the first iteration of the Go-engine, [41], which started out by seeing ten thousands of top-level human games. However, showing the agent examples of good behaviour typically speeds up learning significantly, so if computation is a limited resource, it may be worthwhile.

When actions only have a small impact on the state, the agent needs to be able to differentiate between very similar states. This induces the need for a fine-grained state-representation and a function which can approximate the values of such states. Importantly, the same function should also work for different instances of a problem; a candidate for such a function could be some kind of graph neural network. Learning an action-value function in the fine-grained case is however extremely computationally demanding, since there may be several thousand possible actions in each state. Instead, one could try to learn only a state-value function. The difference is, that the state-value function only has to output a single value for a given state, whereas an action-value function has to output a value for each possible action in a given state. To select actions when only a state-value function is available, one then has to be able to make a forward look, i.e. to try a subset of or all actions and select the one which leads to the state with the largest value (by the Bellman optimality equation (3.5), greediness w.r.t. the state-value function is optimal, if the function is exact). This approach is not always possible in reinforcement learning, but it is for combinatorial optimization since we will have access to a perfect emulator of the environment, and the actions may even be deterministic. One could of course also make a forward look several states ahead, e.g. by the use of a rollout strategy such as Monte Carlo tree search, which was at the heart of the success of AlphaGo and AlphaZero [41, 42]. However, even with just the greedy single state lookahead strategy, this approach is computationally expensive - all possible basic exchanges will be evaluated in terms of a possibly quite complex value function before deciding which one to choose.

A final consideration is, that it should ideally be possible to move to an optimal solution from any start solution by the use of the actions available to the agent. This is indeed the case, if the actions are basic-exchanges in the fixed charge transportation problem [36].

Composite actions

A more computationally convenient approach is to use actions which have a larger immediate impact. The available actions could be predefined compositions of lower level actions. For example, any kind of local search procedure could be such a composite action. Actually, any heuristic procedure which transforms one solution into another could be such an action.

In the case of composite actions, it is possible and perhaps also preferable to utilize an action-value function. First of all, there are far fewer actions than in the fine-grained case, and thus fewer values to estimate in each state. Secondly, the environment is not necessarily deterministic in the composite action case, and evaluating an action may be computationally costly compared to evaluating an action-value function. It would in most cases be too computationally demanding to evaluate the outcomes of all possible actions before determining the action, instead this decision could rely solely on the action value function. It is of course a tradeoff. By looking forward to the actual consequence of an action (if possible), the agent can probably make better actions, but at the cost of increased computation time. Most modern approaches are developed under the assumption, that it is not possible to look forward to the actual consequence of an action before taking it, and we choose follow this line of work, but it could be an interesting topic for further research to utilize such forward looks.

6.3.2 States

The state should ideally contain enough information for the agent to choose the optimal action at each time point. It should also not contain too much information, since it increases the needed computational effort and also increases risk of overfitting, i.e. finding patterns in data that are a product of noise. All information encoded in the state should be relevant for making decisions about which actions to take. It may be necessary to somehow take into account time dependence as well, for example by encoding history in the state representation, or alternatively by using a function approximator with 'built in memory', e.g. some variant of a recurrent neural network [20, 25]. Importantly, the state representation should also depend on the granularity of actions.

If the actions are fine-grained, a fine-grained state representation is needed. You could imagine yourself in the place of the agent taking actions, and think of what you need to know to make an informed choice. For example, in the fixed charge transportation problem, in order to select a good basic exchange,

it is necessary to know the costs on each arc, which arcs are currently basic etc.

In the case of composite actions, the agents does not necessarily need to know as many details about the current solution. Instead, the state could consist of high level attributes such as current and best objective value, the history of the search and possibly other relevant features of the given problem at hand, e.g. the ratio of fixed costs in the fixed charge transportation problem.

6.3.3 Rewards

The final component one has to specify in the reinforcement learning framework is the reward signal. In many use cases of reinforcement learning, the state and action space are more or less given, but it is difficult to determine a proper reward signal. Fortunately, there are a lot of natural options for combinatorial optimization. Recall, that the reward signal should be a way of rewarding the agent for doing well, but it should not be used to inject prior knowledge into the agent, such prior knowledge is better represented in the states. For example, it might be tempting in the fixed charge transportation problem to reward the agent for finding a basis with less active arcs, thus saving fixed costs. However, the objective value itself should be enough for the agent to discover, that moving to such a basis is often good, if the agent is informed about it in the state space. Also, it is difficult to determine how such an event should even be rewarded. In any case, it is not advisable. The reward signal should, importantly, be a way of communicating *what* you want to achieve and not *how* to achieve it!

Recall, that the objective for the agent is to maximize the expected return, G_t, i.e. the discounted sum of rewards. Thus, for combinatorial optimization problems, it would be natural to somehow reward the agent for improving the objective function. One could imagine many different ways of doing so, and we describe the considerations we had before finally determining the reward signal in the following.

Reward the best value found at the end of an episode

The first idea we got was to give the agent a numerical reward equal to (minus) the best objective value found during an episode at the end of each episode. The idea was inspired by how reinforcement learning has been used to tackle board games, where rewards are typically only given for the winning move. However, this structure would cause the rewards to be quite sparse and the agent can only slowly make meaningful updates to its policy.

In addition, it may be difficult to assign credit to which actions actually led to this final objective value, since the reward signal is so delayed. Both of these issues increase the needed computational effort greatly - the idea of giving intermediate rewards or shaping (see subsection 3.2.1), seems to be important in this context.

Improving the current versus improving the best solution

Following this train of thought, our next idea was to provide a reward every time the solution is improved. This can be done in several ways. For example, one could provide a reward of one for improving the current solution. However, this will likely have adverse effects. The agent will probably learn to improve the solution, then worsen it, and then improve it again with no large net effect on the objective value, but accumulating reward. To avoid this, a reward could instead be given each time the best solution found so far is improved. By best so far, we mean best in the current episode, otherwise we will have the problem of sparse rewards again. If this reward is chosen to be binary, the agent will not be rewarded more for one large improvement in objective value, but may instead learn to make many small improvements to the objective value to accumulate reward.

Objective value rewards

From the above discussion it is clear that the magnitude of the reward should be related to the improvement in objective value. An obvious, but also naive approach is then to give the agent a numerical reward equal to the numerical improvement in objective value. This would work just fine if we were only interested in one single instance of a combinatorial optimization problem, but that is not the case. The scale of the reward may be completely different for different problems of the same difficulty. It seldom makes sense to reward the agent much more for solving one problem than another. In the fixed charge transportation problem, we discussed in subsection 2.3.1, how there may be an invariant in the objective function that can be scaled to be arbitrarily large. What we need, is a relative performance measure that gives similar sized rewards across different problem instances.

Relative rewards

We propose the following relative performance measure

$$RP_t \doteq \frac{best_t - LB}{UB - LB} \tag{6.1}$$

where $best_t$ denotes the best hitherto seen objective value at time t, LB and UB a lower bound and upper bound on the objective value respectively. Before explaining how we turn this into a reward, let us dwell on the performance measure for a bit. Notice, that this number will lie between 0 and 1 as long as the objective value is between the upper bound and the lower bound. However, the objective value may also go/start above the upper bound, and in such cases, this measure will be larger than 1. If the lower bound is a true lower bound on the objective value, it cannot go lower, but if we for example allow the lower bound to be, say, the best value seen on this instance ever, then it may be possible to go lower, and in this case the measure will have a negative value. None of these cases are a problem. To make the measure more well suited for a reward, we look at $1 - RP_t$ and notice, that this is negative for a larger value than the upper bound, between 0 and 1 for values between UB and LB and larger than 1 if the lower bound is improved.

As discussed, the idea is then to provide a reward each time a new best solution is found in an episode. This reward could then be

$$R_t = (1 - RP_t) - (1 - RP_{t-1}) = RP_{t-1} - RP_t \tag{6.2}$$

Notice that this will be zero as long as a new best value is not found and otherwise positive.

We may further extend the idea to take into account, that when the objective is already good, only small improvements are possible and thus small rewards, while relatively large rewards will be given at the beginning of each episode. A remedy is to scale the reward with the relative objective value, i.e.

$$R_t = \frac{RP_{t-1} - RP_t}{|RP_{t-1}|} \tag{6.3}$$

this way, if we are already close to the lower bound, a larger reward will be obtained for a small improvement in objective value. A consequence of using this reward structure is that the agent will be guided to find strategies which prioritize best performance over average performance. It is this reward structure, that we use in our experiments.

6.4 Reinforcement learning for the fixed charge transportation problem

We now specify how actions, states and rewards could look like for the fixed charge transportation problem.

6.4.1 Actions

In the fine grained setup, an obvious choice of possible actions is single basic exchanges. In the composite action setup, it could be any mutation to a solution, for example a first-accept local search or a random mutation. The actions we consider in our experiments are all discussed in subsection 2.2.1.

6.4.2 States

As we argued in the previous sections, we should distinguish between the case of fine-grained actions and the case of composite actions when choosing a state representation.

Fine-grained actions

In the case of fine-grained actions, we need a fine-grained state representation. We should consider, what the agent needs to know to make the best basic exchanges in the long run. We may distinguish between node level, edge level and global features - a distinction which is important if we for example implement a graph neural network as a function approximator. We hypothesize, that we at least need the following features:

- **Node level features:** Supply, demand, number of active arcs, predecessor in basis tree.
- **Edge level features:** Variable costs, fixed costs, basic/non-basic and amount sent on the edge in the current basis.
- **Global features:** Number of suppliers and customers, number of edges, current objective value, best seen objective value and number of edges with positive flow.

Composite actions

In the composite action case, there is no reason to give the agent information about the specifics of any edges, since the actions in general are not directly related to any specific edges. However, it is quite important to include history of the search in the state space as e.g. the second of two local searches in a row would be futile. If the agent does not know, that it just made a local search, it has no way of knowing it should not do so again.

We use the following features in the state space for the fixed charge transportation problem with composite actions:

(1) Current performance (in the relative performance measure).

(2) Best performance (in the relative performance measure).

(3) Number of arcs with positive flow relative to number of basic arcs.

(4) Whether or not the latest action improved the current performance.

(5) Percentage of fixed costs in objective value.

(6) Binary variables for each action, one indicating whether it was the last one chosen, one for the second last and one the third last.

(7) Counts for how many times each action was chosen out of the last 20.

Here, (5) should in particular allow the agent to distinguish between different types of instances, which is important, since some strategies may prevail when fixed costs are relatively large and some when they are small. Notice also, that most variables (4 times the number of actions) are used to keep track of the history of the search, this is not ideal but it is a way of circumventing, that the Markov property is actually missing. This is common practice in reinforcement learning [35]. It could be interesting to examine whether some of this memory could instead be kept in part by the function approximator, e.g. by the use of a recurrent neural network.

6.4.3 Rewards

We use the reward structure from Equation (6.3). For upper bounds, we use solution corresponding to the LP relaxation and for 'lower bounds', we use the best found solution values in literature, and if a new instance is considered, the lower bound is the best result obtained by some heuristic.

Chapter 7

Experiments

In this chapter we describe the experiments we conducted for solving instances of the fixed charge transportation problem. First, we describe our experiments with meta- and hyperheuristic approaches, and then we describe the experiments we conducted in the reinforcement learning setting. Finally, we propose a revised meta-heuristic and discuss and compare the approaches. Ultimately, we find new best solutions on 112 out of 120 well-known instances from the literature.

All experiments are performed on a Dell XPS 9570 laptop with an i7-8750H-processor, NVIDIA GeForce 1050Ti-GPU and 16 GB 2 x 8 GB DDR4-2666 MHz RAM. The operating system was Ubuntu 18.04, the Java version open JDK 11.0.1 and the Python version 3.7. The reinforcement learning experiments utilize the PyTorch library to train neural networks on the GPU to speed up computation. All the code is publicly available on GitHub[1], and is also described in Appendix A.

7.1 Meta- and hyperheuristics for the fixed charge transportation problem

First, we outline the experiments we conducted in the more classical setup without the use of reinforcement learning techniques. The experiments establish a baseline we can compare the reinforcement learning approach to, and additionally, they showcase that the population based iterated local search approach we described in Chapter 2 is competitive with the current state-of-the-art heuristic for the fixed charge transportation problem, while being conceptually simpler and easier to implement. We also describe the experiments

[1] `https://github.com/Tybirk/RL-FCTP`

we did in the Conditional Markov chain search framework, since the reinforcement learning approach may be seen as a generalization of that framework.

7.1.1 Problem instances

We conduct our experiments on a testbed of instances originally provided by Sun et al. [45], which was also used by Glover [16] and Buson et al. [5] for their experiments. All problems have 50 suppliers and 100 customers and a total demand of 50.000. There are links between all customers and suppliers. We consider 8 classes of problem instances, denoted by (A)-(H). For all classes, the unit variable cost range from 3 to 8 on each arc. The problem instances are thus mainly defined by their fixed costs. In each class, there are 15 problem instances, numbered 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E. In total, that means we conduct tests on 120 instances. An overview of the instances is provided in Table 7.1.

Instances	Lower	Upper
(A) N3000 - N300E	50	200
(B) N3100 - N310E	100	400
(C) N3200 - N320E	200	800
(D) N3300 - N330E	400	1600
(E) N3400 - N340E	800	3200
(F) N3500 - N350E	1600	6400
(G) N3600 - N360E	3200	12800
(H) N3700 - N370E	6400	25600

Table 7.1: Fixed cost ranges for the problem instances of the FCTP.

7.1.2 Population based iterated random neighbourhood local search

The first heuristic we test is based on Algorithm 4 (PIRNLS) from Chapter 2. We first generate 100 high quality solutions with Algorithm 3, iterated random neighbourhood local search (IRNLS), with parameters $n_{\max} = 800$, $n = 50, N = 250, \alpha = 1.05, \beta = 30$ and $\kappa = 10$. For each run, the start solution is random with 50% probability and the solution based on the LP relaxation and then kicking out half the arcs with 50% probability. Then, we make a round of PIRNLS, and return the 10 best solutions found. To each of these 10

A	B	C	D	E	F	G	H	Total
9/15	10/15	7/15	5/15	7/15	6/15	5/15	6/15	55/120

Table 7.2: Ratio of upper bounds improved for each problem class by one run of the PIRNLS procedure.

solutions, we again apply IRNLS with the same parameters as above, apart from $n_{\max}$ which is set to 2000. The resulting 10 solutions are then used as an input to a final round of PIRNLS. The best solution obtained this way is again fed into the IRNLS procedure, this time with $n_{\max} = 5000$.

The idea is basically to construct a set of diverse, high quality solutions and then to intensify the search around the most promising ones. With this setup, the runtime for one run of the full procedure was between 35 and 40 minutes on all instances. In one run, the algorithm improved the upper bound on 55 out of 120 instances, and is thus competitive with the results of Buson et al. [5] with comparable runtime (they used 50 minutes per instance with a slightly slower processor and the implementation was in C instead of Java). As seen in Table 7.2, our approach is competitive for all 8 problem classes, but seems to perform a bit better when the fixed costs are low. The full results are available in Table B.1-B.8 in the Appendix.

7.1.3 Analysis of performance

To further compare the performance of PIRNLS to the results obtained by Buson et al. [5], we use the performance measure discussed in subsection 2.3.1, namely

$$RP(H, I) \doteq \frac{Z_H(I) - LB(I)}{UB(I) - LB(I)}, \tag{7.1}$$

where $UB(I)$ is the objective value obtained by using the flows obtained by solving the LP relaxation and $LB(I)$ is the best solution seen in the literature on instance I. Notice, that this measure will be equal to 0 if the performance of the heuristic, $Z_H(I)$ is the best seen in the literature. A visualization of the performance of the heuristics is presented in Figure 7.1. Clearly, the variance in performance seems to increase for the instances with larger fixed costs, which renders a statistical comparison such as a paired t-test difficult. However, by inspecting the plot it is clear that the performance of the two heuristics is indeed very similar. In fact, the average difference in performance is just 0.000013. To make a formal comparison, one could use the Wilcoxon signed rank test [54], but we omit it since it clearly would not find a statisti-

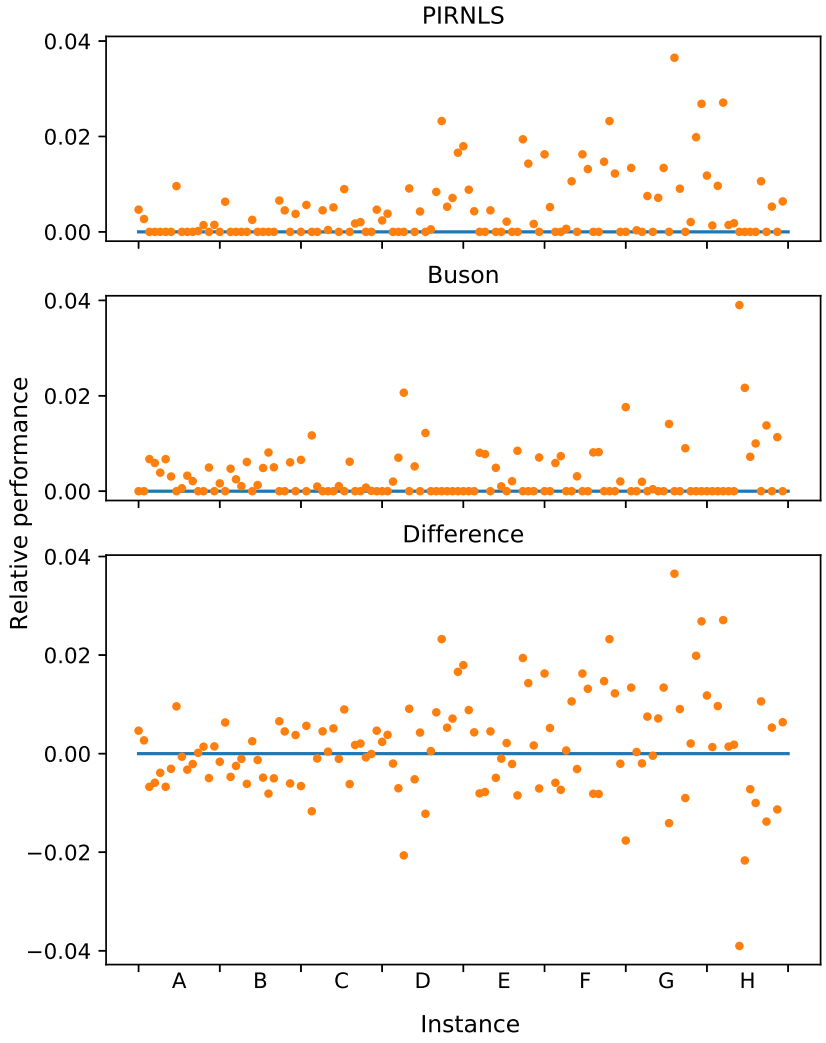

Figure 7.1: Relative performance compared to the best in the literature across problem instances. The instance classes are indicated by A-H.

cal significant difference between the two heuristics, given that the average difference in performance is so close to 0. In subsection 7.3, we revisit the PIRNLS heuristic and describe how we further improved it.

7.1.4 Conditional Markov chain search

The initial experiments in the Conditional Markov chain search (CMCS) setup were discouraging. We quickly realized, that we have to restrict ourselves to configurations of at most three components, otherwise the computational burden is simply too heavy. Even with only three components the computational burden is still large. Ideally, we do not want to 'precompose' components ourselves, e.g. by ensuring that a random perturbation is always followed by an improvement step, but since we have to restrict ourselves to configurations of at most three components, we deemed this necessary. Due to the excessive computation time, we had to limit ourselves to only 11 (precomposed) components. This highlights a weakness of CMCS - the runtime grows extremely quickly in the number of components, and the CMCS framework does not automatize the choice of components. We ended up conducting experiments with the following components:

(1) Kick out k arcs with largest cost per unit in the current solution and do a best accept local search

(2) Kick out k arcs with largest cost per unit in the current solution and do a first accept local search

(3) Kick out k arcs with largest cost according to *greedy1* evaluation measure and do first accept local search.

(4) Kick out k arcs with largest cost according to *greedy2* evaluation measure and do first accept local search.

(5) Kick out k arcs with largest cost according to *greedy3* evaluation measure and do first accept local search.

(6) Introduce k arcs into the basis according to the *greedy1* evaluation measure and do first accept local search.

(7) Introduce k arcs into the basis according to the *greedy2* evaluation measure and do first accept local search.

(8) Introduce k arcs into the basis according to the *greedy3* evaluation measure and do first accept local search.

(9) Random neighbourhood local search with $n_{\max} = 50$, $N = 20$.

(10) Reset to best solution so far and do random neighbourhood local search with $n_{\max} = 50$, $N = 20$.

(11) Reset to best solution so far, kick out k arcs with largest cost per unit in the current solution and do a best accept local search.

in any case, k is drawn uniformly between 5 and and 20% of the basic arcs.

All the components are capable of escaping local minima, which means that the reduction heuristics from the original paper [29] are mostly not applicable. As a result, when we restrict ourselves to heuristics of at most 3 components we have 165 subsets and for each subset 296 configurations to search through. In total 48840 heuristics to evaluate. Obviously, extending to e.g. 4 components is infeasible in this setup. The CMCS framework simply relies too much on brute-force search.

For each subset-configuration pair, we test the corresponding heuristic first on 7 instances limiting the runtime to 0.5 seconds per instance, then we filter out all Pareto dominated configurations, and test the resulting heuristics on 7 new instances with a runtime limited to 2 seconds per instance. Then, we take the 10 configurations with the best average performance as defined by our performance measure and test all these 10 with 1 run on all 120 instances. Then, we select the 2 best and test those on all instances with 10 runs of 10 seconds on each instance. We found, that there were virtually no difference between the two best configurations, but we still chose only one of them for the final experiments.

Clearly, it is not optimal to only allocate such a short search time for each configuration, and hence the first filtering where we only allow 0.5 seconds runtime is largely dominated by randomness. However, with the very large amount of possible configurations, it is simply too computationally demanding to allocate more time. This is of course a significant weakness of the method. To alleviate this, one could consider a kind of local search amongst configurations. The reinforcement learning approach we consider can in a special case be interpreted as a gradient based local search amongst configurations.

Ultimately, the heuristic we discovered through CMCS consisted of the elements (4), (5) and (9). We tested this heuristic with 100 runs on 50 representative instances, and found that the average performance was good, actually about the same as IRNLS, but the best performance across 100 runs on each instance was inferior to the best one of IRNLS on 49 out of 50 instances given the same computation time. The best and average results of the 100 runs on each instance can be found in Table 7.3 and 7.4.

An interesting result, was that heuristics which included a reset to the hitherto best solution were almost entirely filtered out in the first phase of the search. We conjecture, that this is due to the fact that the algorithm only was allocated 0.5 seconds of runtime in the first phase. In addition, the CMCS

framework does not allow such a reset to depend on the history of the search. The component used is always based only on the immediate preceding component. In our experiments in the reinforcement learning setup, we discover that resetting to the best solution once in a while is an ingredient in most good policies, and the framework allows the agent to base such a reset on the history of the search.

On the basis of our experiments, we conclude that the CMCS framework does not seem efficient for finding good heuristics to the fixed charge transportation problem.

7.2 Reinforcement learning for the fixed charge transportation problem

A lot of effort was put into making generally applicable code for reinforcement learning with combinatorial optimization. At first, this effort was not well structured; it was spread between representing the FCTP environment in a way such that it is digestible by a reinforcement learning agent, and actually writing the code for the reinforcement learning agents. We realized, that there was no reason to build a reinforcement learning agent from scratch, as many general frameworks already exist - instead we focused on building the FCTP environment in a way, which fits the interface of a general reinforcement learning agent. Consequently, it is quite easy to use our framework for other combinatorial optimization problems; one only has to implement the environment such that it adheres to the specific interface. How to do this is explained in Appendix A, where we describe the code and how to extend it in more detail.

We start this section out by describing how we failed to successfully implement the approach with fine-grained actions (basic-exchanges). Then we describe our efforts in the composite action setting, including the full training process, which ultimately was very successful.

7.2.1 The basic-exchange approach

Initially, we made the mistake of focusing mainly on the most difficult setting - the one with fine-grained actions. We spent a significant amount of time doing a trial and error search over different graph neural network configurations and other hyperparameters. We never found anything worth spending more time on - the heuristics learned were typically not much different than a local search, but the runtime was large. The results indicate, that we never

quite found a neural network arcitechture capable of representing the state-value function. The sheer amount of computation needed for the training process was also quite limiting. We ultimately decided to abandon this approach and instead focus on the composite action approach, which we found a lot of success with. The more fine-grained approach should have the potential of finding better solutions as we argued in Chapter 6, and it could certainly be an interesting topic for further research. For now, we focus on the composite action approach.

7.2.2 Selection of hyperparameters

The largest issue with experiments in the reinforcement learning setup is probably the amount of hyperparameters. Especially since it is very time consuming to evaluate a configuration of hyperparameters. To evaluate a hyperparameter setting, one has to let the agent train and monitor its development. The agent may act almost completely random for a long time to explore the environment. In our experience, it takes at least 2 hours to get an idea of whether a hyperparameter setting works. Still, evaluating it so early in the training process may not be telling about its final performance. In addition, the same hyperparameter setting may even give quite different result because of the randomness in the environment. Hence, it is both difficult and time consuming to evaluate a single hyperparameter configuration, and as you will see in the following, there are many to choose from! Thus, a structured search with respect to all parameters is too computationally demanding. Instead we chose to fix some parameters early in the process by using a combination of gut feeling and what have worked well on other environments described in the literature. We do a more structured search and analysis with respect to learning rate and algorithm type.

Neural network architecture

An important choice of hyperparameters is the neural network architecture. We settle on multilayer perceptrons. Accordingly, we have to choose the number of layers, the number of nodes in each layer and the activation functions. We default to the ReLU activation function in the hidden layers, the softmax activation function if the output has to be a probability and a linear activation if the output is a real number. We ended up working with a neural network architecture with 3 hidden layers with sizes 256, 256 and 128. Heuristically, we chose the network to have a quite high capacity such that we are sure it can fit the data to a local minimum of the loss function. We indeed observed that convergence towards a local minimum was possible in all

experiments. We did not experiment in depth with many other architecture settings, since this one seemed to work well enough. It could be a subject of future research.

Learning rate

Another important hyperparameter is the learning rate. If it is set too high the network will converge too fast towards a local minimum, which may be far from optimal. For example, we saw in many experiments that the policy degenerated to a single action regardless of the state, which is clearly suboptimal. However, if the learning rate is too small, convergence may be extremely slow and the agent may act close to random for a very long time. For example, we at one point trained a policy-gradient based agent for a whole weekend without much change in the policy, simply because the learning rate was so low, that the agent never moved away from a random policy. We test different settings of learning rate in subsection 7.2.6.

Discount factor

The discount factor tells the agent how much weight to put on future observations. In the literature, it is often set to 0.99. We experimented with values from 0.9 to 0.99 and found, unsurprisingly, that for the case of 1-step returns, the impact is not too big, but for the case of n-step returns it may both impact the speed of learning and the resulting policy. For the experiments we describe, we chose to fix the discount factor to 0.98.

λ and n-step returns

We experimented only with n-step returns, and found the value of n to be important. For example, with a choice of $n = 20$, we observed that learning initially was slow. The reason is, if the policy is random, and each action is valued also according to the following 20, then this evaluation is governed largely by randomness since we only have 17 actions. It is also a problem if n is too small, since it will be more difficult to assign credit to a random permutation or a reset for changing the objective value further than n timesteps away. For instance, if one resets to the hitherto best solution, it is often unlikely that the solution will be improved immediately after, and thus the agent needs to learn, that it may be beneficial in the longer run anyway, and a larger n certainly helps with that. It is the problem of credit assignment - which actions actually made a difference. It is easy to address the immediate impact of an action, but long-term credit assignment requires many observations and either a way to *remember* which actions were taken for a long time,

or a reasonably large n. We present our experiments with n-step returns for $n = 1, 10, 20$. It could be interesting to experiment with λ-returns in the future.

Termination criteria

Another hyperparameter to choose is when to terminate each episode. For example, one could choose to terminate after a fixed number of actions. However, that might stop an agent on a streak of improvement, and we instead choose to terminate after a fixed number of actions without improvement to the objective value, and we choose 500 during training.

Upper and lower bounds on instances

As discussed in Chapter 6, we let the upper and 'lower' bounds be the solution obtained by solving the LP-relaxation and the best known solution in the literature (before this thesis), respectively. In this way, the agent gets a large reward for improving the best known solution in the literature, which they would not get in the same manner, if a true lower bound was chosen. However, a true lower bound would probably be preferable if some relatively tight bounds were available, since the current structure means, that very large rewards are given for instances where the quality of the current best solution in the literature is low.

Weight decay and entropy regularization

We do not use weight decay for regularization in our experiments. In the policy-gradient setup, we use entropy regularization of 1e-6. We did not observe that entropy regularization had a large effect on the policy, even if increased or decreased significantly.

Hyperparameters for the individual actions

There are also an abundance of hyperparameters which govern the individual actions. Or phrased differently, the pool of available actions can also be seen as a hyperparameter. The specific pools of actions we use are presented in the next two subsections. We consider two levels of action aggregation in our experiments.

7.2.3 One component actions

The first level of aggregation is the most decomposed one; we let each mutation be an action by itself but do not explicitly compose mutations, instead we allow the agent to discover these compositions by itself. Compared to the second approach we use, this one allows the agent more flexibility. The components we use here are

(1) First accept local search.

(2) Best accept local search.

(3) Random neighbourhood local search, $n_{\max} = 50, N = 20$.

(4) Random neighbourhood local search, $n_{\max} = (|S| + |C|)/5, N = 20$.

(5) Introduce k arcs into the basis at random.

(6) Introduce k arcs into the basis according to the *greedy1* evaluation measure.

(7) Introduce k arcs into the basis according to the *greedy2* evaluation measure.

(8) Introduce k arcs into the basis according to the *greedy3* evaluation measure.

(9) Reset to best solution seen so far.

(10) Store solution.

(11) Reset to stored solution.

(12) A record to record travel move with threshold 10%.

(13) Kick out k arcs with largest cost according to *greedy1* evaluation measure.

(14) Kick out k arcs with largest cost according to *greedy2* evaluation measure.

(15) Kick out k arcs with largest cost according to *greedy3* evaluation measure.

(16) Kick out k arcs with largest cost per unit in the current solution.

(17) Kick out k arcs at random.

In any case, k is drawn uniformly at random between 5 and and 20% of the basic arcs each time the corresponding action is selected. We realized after completing the experiments, that we should for example have allowed more than one record-to-record move in order for it to be fair with respect to runtime, since the agent does not take this into consideration.

7.2.4 Two component actions

The other approach we try is to ensure that no action is 'bad' by itself. For example, making a completely random mutation of a solution is bad by itself, but may be good if it is followed by an improvement procedure such as a local search. In the two component setup, we explicitly tell the agent, that it must use an improvement procedure after a random permutation, and that it must make a (partly) random permutation after reaching a local minimum. In this setup, even a random policy would perform quite well, but it should be possible to improve.

(1) Random neighbourhood local search with $n_{\max} = 50$ and $n = 20$ followed by first accept local search.

(2) Random neighbourhood local search with $n_{\max} = 50$ and $n = 20$ followed by best accept local search.

(3) Random neighbourhood local search with $n_{\max} = 50$ and $n = 10$ followed by first accept local search.

(4) Introduce k arcs into the basis at random do first accept local search.

(5) Introduce k arcs into the basis at random do best accept local search.

(6) Introduce k arcs into the basis according to the *greedy1* evaluation measure and do first accept local search.

(7) Introduce k arcs into the basis according to the *greedy2* evaluation measure and do first accept local search.

(8) Introduce k arcs into the basis according to the *greedy3* evaluation measure and do first accept local search.

(9) Reset to best solution and do random neighbourhood local search with $n_{\max} = 50$ and $n = 20$ followed by best accept local search.

(10) Reset to best solution, introduce k arcs into the basis according to the *greedy2* evaluation measure and do first accept local search.

(11) Reset to best solution, introduce k arcs into the basis according to the *greedy3* evaluation measure and do first accept local search.

(12) Kick out k random arcs from the current solution and do a first accept local search.

(13) Kick out k arcs with largest cost according to *greedy1* evaluation measure and do first accept local search.

(14) Kick out k arcs with largest cost according to *greedy2* evaluation measure and do first accept local search.

(15) Kick out k arcs with largest cost according to *greedy3* evaluation measure and do first accept local search.

(16) Kick out k arcs with largest cost per unit in the current solution and do a first accept local search.

(17) Kick out k arcs with largest cost per unit in the current solution and do a best accept local search.

In any case, k is drawn uniformly at random between 5 and and 20% of the basic arcs each time the corresponding action is selected.

7.2.5 Training instances

To train the agents, we use the instances that we introduced in subsection 7.1.1. At training time, a random instance from this pool is drawn for each episode. We could also have elected to train on only a subset of instances and then test the generalization ability - however the information that is encoded in the states is so similar from instance to instance (within the same class) that generalization is not an issue, which we did indeed observe during some of the experiments that are not documented here.

7.2.6 Analysis of training

In this section, we analyze data from 30 runs of training, 15 in the one component actions case and 15 in the two component actions case. We use the n-step advantage actor-critic (A2C) and the deep Q-network (DQN) approach introduced in Chapter 5. Of the 15 runs in both the one and two component actions case, 3 are based on 1-step A2C, 3 on 10-step A2C, 3 on 20-step A2C, 3 on 10-step DQN and 3 on 20-step DQN where the 3 runs are with different learning rates, in all cases 1e-6, 5e-6 and 5e-5. We look into the evolution of

the agent over time for each of these parameter settings, both in terms of action selection and reward accumulation. In all runs, we observe that the loss function converges to a value very close to zero, naturally with fluctuations that are large, if the reward in an episode is far greater or smaller than average. This roughly means, that the agent learns the average value of the policy it uses in any case, which suggests that our neural network has a sufficiently detailed architecture.

Analysis of action selection

An interesting aspect of analyzing reinforcement learning agents is to look into which actions they select. The frequency of actions, how the action selections develop over time, and how the agents adapt to instance specifics are all interesting things to analyze. We first look into the case of one component actions.

One component actions Recall, that we have 17 actions in the one component case, all outlined in subsection 7.2.3. In Figure 7.2, we have visualized the relative frequency of actions over time, averaged over the 15 training runs, as training progresses. In the plot, the dots are placed every 10th episode, and the color indicates the relative frequency of the particular action for that particular episode. Initially, all the actions are selected an equal amount since the initialization is random. Interestingly, it seems that action 16, 15, 14, 9, 3, 2 and 1 all get off to a good start, but quickly the agent learns to favor only some of these actions. In particular, action 3 and 16 are popular, which are a random neighbourhood local search and kicking out the arcs with largest cost per unit in the current solution, respectively. Action 9 is also popular, which is resetting to the hitherto best solution. Interestingly, none of the actions which introduce k arcs into the basis according to some evaluation measure are popular (action 5-8). The actions which kick out arcs are all more popular.

The plots for all the individual runs are presented in the Appendix, Figure C.1-C.15. From these plots we see, that when the learning rate is chosen to be 1e-6, the policy is still quite random after 1000 episodes - no actions have yet been ruled out, but the agent has learned that action 3 is good. This is the case for 1-step, 10-step and 20-step A2C as well as 10-step DQN and 20-step DQN. From these plots, we conclude that if the learning rate is chosen to be 1e-6, we should let the agent train longer than 1000 episodes. For a learning rate of 5e-6, the runs have almost converged. A peculiar observation is that that the actor critic agents seem to prefer action 3 while the DQN agents seem to prefer action 16, but this difference may be due to randomness. Also,

the DQN methods seem to have a larger tendency to degenerate to a single action.

Two component actions In the two component actions case, there are no actions which are quite as dominant as in the one component case. The most popular one seems to be number 17 - kick out k arcs with largest cost per unit in the current solution and do a best accept local search. Interestingly, the best accept local search is preferred, but note that the agent is not punished for the additional runtime. Also interesting is, that the other ways of kicking arcs out of the basis do not become very popular in this setup, which is in contrast to the one component case. All the actions where the solution is reset to the best solution (9-11) are quite popular as well as the components based on random neighbourhood local search.

By looking into the individual runs (Figure C.16-C.30 in the Appendix), we again see that a learning rate of 1e-6 is too low for only 1000 training runs, whereas 5e-6 seems sufficient, and 5e-5 is thus perhaps too high, since the agent will not explore the environment to the same degree. It is also clear, that the components are more equal in this setup, resulting in final policies that are governed by randomness to a larger extent than in the one component actions case.

Another thing to analyze is whether the agent adapts to the different types of problem instances. In general, this seems not to be the case. We ran the analysis, and found no significant differences in action frequencies across problem instance types. One reason could be that the ratio of fixed costs drowns in the other information that the neural network is given, and perhaps we should instead indicate this ratio by Boolean variables, which are easier for the network to learn the effect of. For example with a variable that is 1 if the ratio of fixed cost is between 50% and 60% and another variable indicating whether it is between 60% and 70% etc. This could be an interesting prospect for further analysis. Doing so, we may discover if any of the actions we are investigating are significantly better for some problem type than another, but for now these experiments remain inconclusive.

The magnitude of rewards

Recall, that we use the reward structure defined in Equation (6.3). We give the agent larger rewards for finding better solutions, when the solution found is already good. The idea behind this shaping of rewards was that we ultimately care less about the average solution quality and more about the best solution quality. To illustrate the effect and also give the reader a sense of what the magnitude of rewards mean, here are some sample rewards from

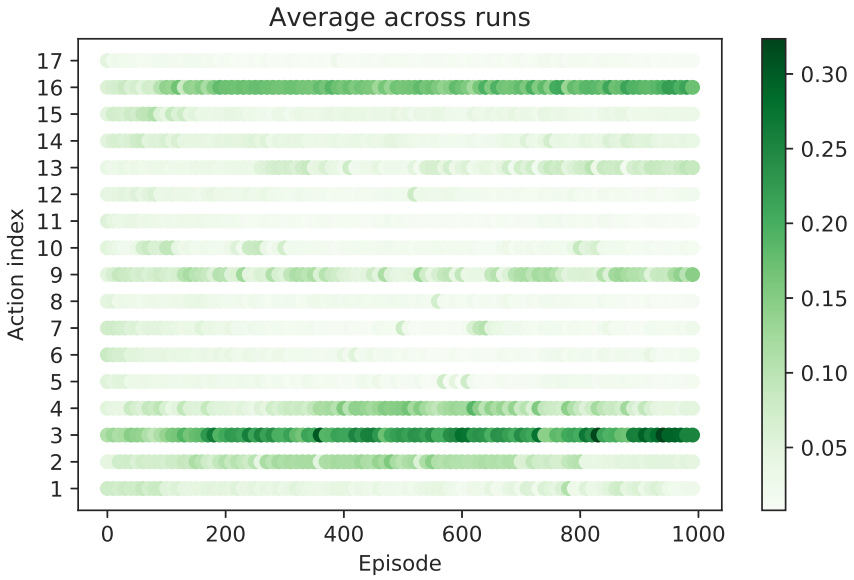

Figure 7.2: Average relative action frequencies across the 15 training runs, one component actions case.

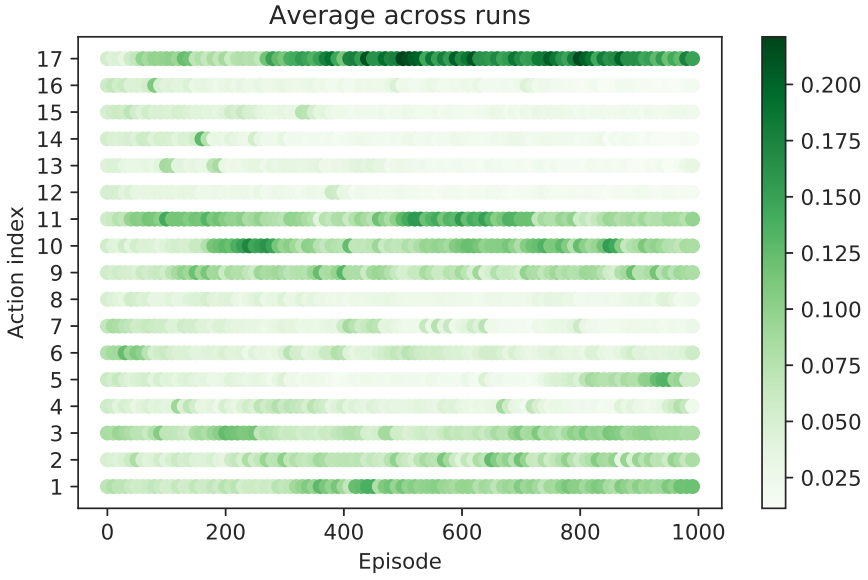

Figure 7.3: Average relative action frequencies across the 15 training runs, two component actions case.

the instance N3004 and corresponding objective values. The 'lower bound' used to compute these rewards was 167074. We for example saw the following objective value, reward pairs during training of the 10-step DQN model with learning rate 5e-5: 167442: 1.08, 167386: 1.26, 167363: 1.31, 167289: 1.58, 167228: 1.89, 167149 : 2.61. Evidently, the reward grows quickly as we come close to a new best solution - typically a reward around 4 means a new best solution has been found. It could be interesting for further research to investigate a reward structure without the scaling factor.

Analysis of training methods

The action analysis becomes much more interesting when coupled with an analysis of accumulated rewards. We now look into the evolution of reward accumulation during training for each hyperparameter setting. Before the analysis proceeds, it is important to realize that each training run is a random sample from the pool of possible runs with the given hyperparameters, which ultimately makes a comparison of parameters somewhat difficult. Ideally, we would try the same hyperparameter setting multiple times, as the randomness in the environment can cause the agent to converge to quite different end policies. Unfortunately, we have not had the time for this, since each training run of 1000 episodes takes approximately 5 hours to complete. Still, we can gain some insights from these 15 runs in both the one and two component actions case.

One component actions We first turn out attention towards one component actions. As described, we let the agent train for 1000 episodes with different hyperparameter settings, specifically we alter the learning rate and the learning algorithm. The full results of training are presented in Figure C.31 - C.35 in the Appendix. The reader may inspect these plots to get a better understanding of the training process for each agent, but we will not go into detail with all the plots here. As an example, we take a look at 10-step A2C. A plot of the training process can be found in Figure 7.4. The blue dotted line represents the average reward of a completely random policy, which was estimated to be 1.12 based on 2000 episodes. We see, that the agent quite quickly learns a policy which is better than random, and keeps on improving in both the 1e-6 and 5e-6 learning rate cases. A reward around 3 for an episode means, that the agent has found a solution in vicinity of the best one in the literature. Both the 1e-6 and 5e-6 agents learn policies that utilize mainly random neighbourhood local search. In the 5e-5 case, the performance unfortunately deteriorates as time progresses, as the agent converged towards a close to deterministic suboptimal policy. This behaviour could be

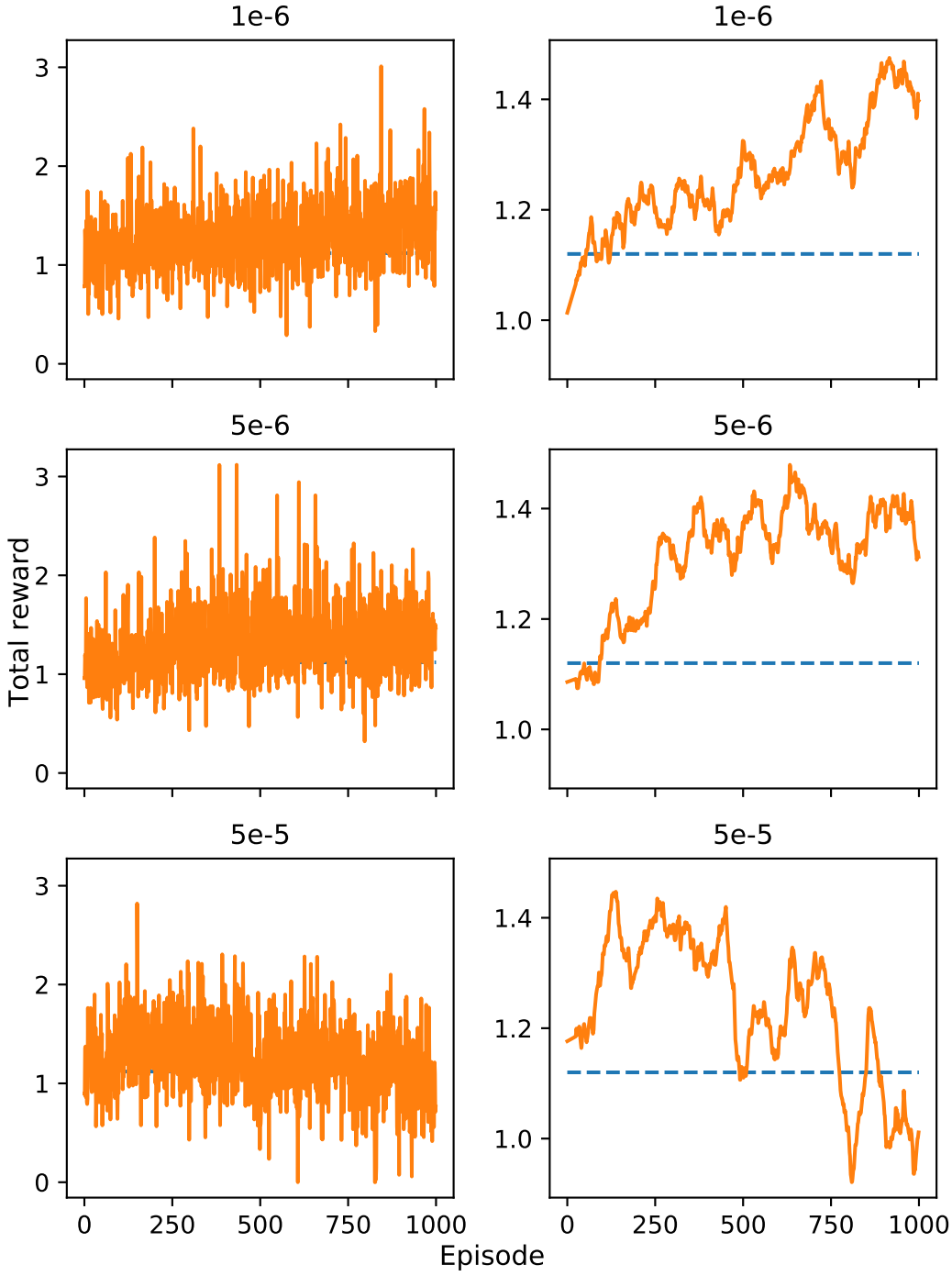

Figure 7.4: 10-step A2C, one component actions. Reward accumulated for each episode of training for different learning rates and corresponding moving average.

due to a too large learning rate, and the rest of our results also suggest, that a learning rate of 5e-5 seems to be too large. In general, it also seems as if a learning rate of 1e-6 is too low for only 1000 episodes of training (but is probably fine with more episodes) as the policy is still quite random towards the end. In general, the analysis suggests that action 3,4 and 16 are the best in the one component case.

Two component actions In the two component case we conducted the same experiments as above. The average reward obtained by a completely random policy was estimated to be 1.48. This is a lot larger than in the one component case, which is due to two factors. First of all, all moves are composed such that they have a chance of improving the solution and secondly, the agents were effectively allowed two times the amount of actions without improvement compared to the one component agent. At test time, we will give the two types of agents approximately the same running time.

Again, we do not present all the plots here, but they can be found in Figure C.36 - C.40 in the Appendix. In the plots, the average obtained from a random policy is again represented by a blue dotted line. As an example, let us examine the training process for the 20-step A2C agents, presented in Figure 7.5. Here, we see that a learning rate of 1e-6 was too low, the policy is not better than random, simply because it is still random! In the 5e-6 case, the agent has learned a policy which is better than random, based mainly on action 3, 6 and 9, which is random neighbourhood local search followed by a first-accept local search, introducing arcs into the basis according to the *greedy1* evaluation measure and resetting to the hitherto best solution followed by random neighbourhood local search. With a learning rate of 5e-5, the agent performs even better as it progressively learns to favor a random neighbourhood local search. This agent also learns, that resetting to the hitherto best solution is good, as it learns to select action 10.

For all the two component action agents, we can conclude that a learning rate of 1e-6 was too low, as the policy remained almost random throughout the 1000 episodes. The 1-step A2C approach was unsuccessful for all three learning rate settings. The 10-step A2C approach produced results significantly better than a random policy, and the same can be said for 20-step A2C where especially the 5e-5 learning rate setup looked promising. The DQN approaches produce policies that do not look much better than random within the 1000 episodes.

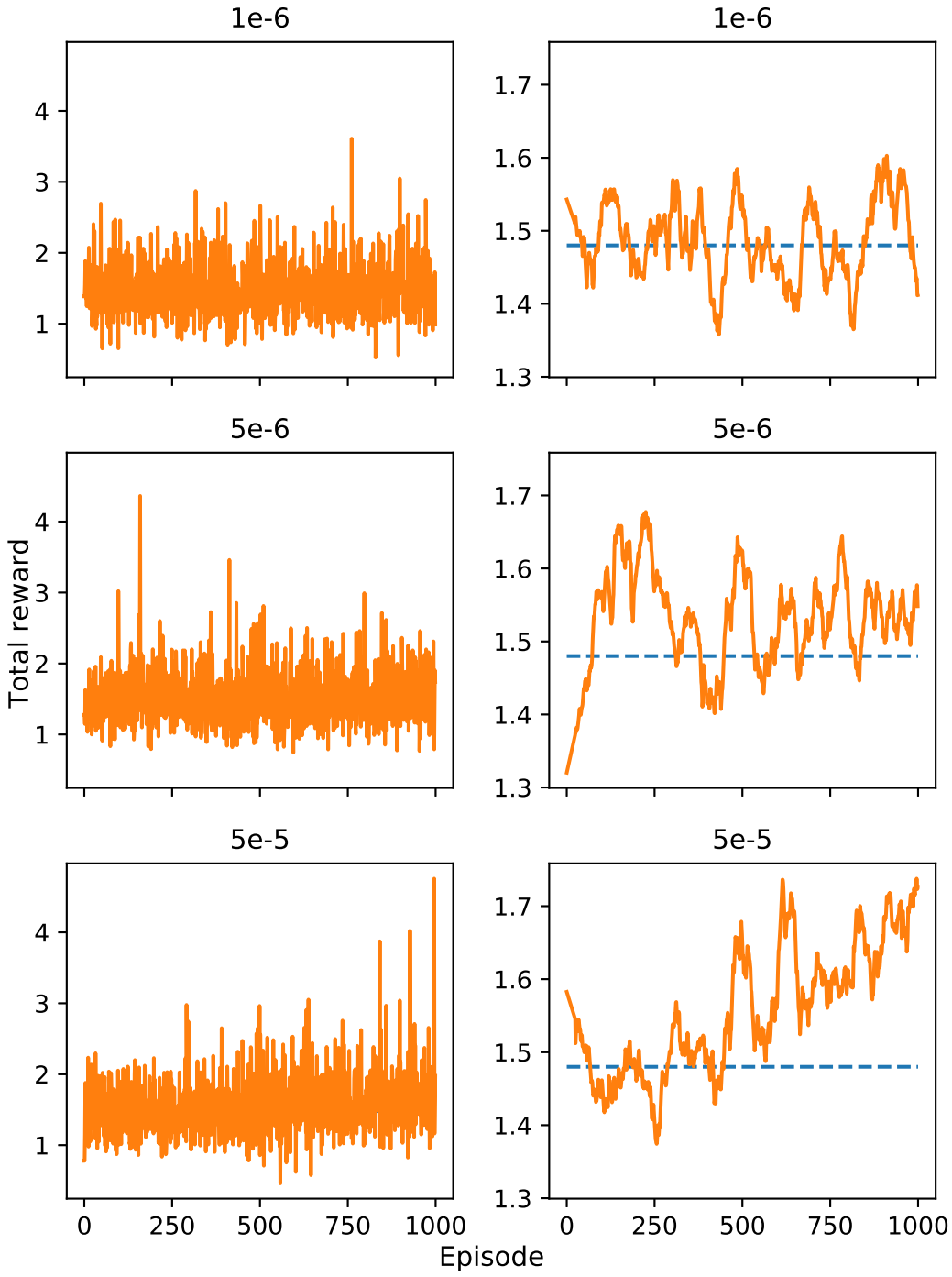

Figure 7.5: 20-step A2C, two component action actions. Reward accumulated for each episode of training for different learning rates and corresponding moving average.

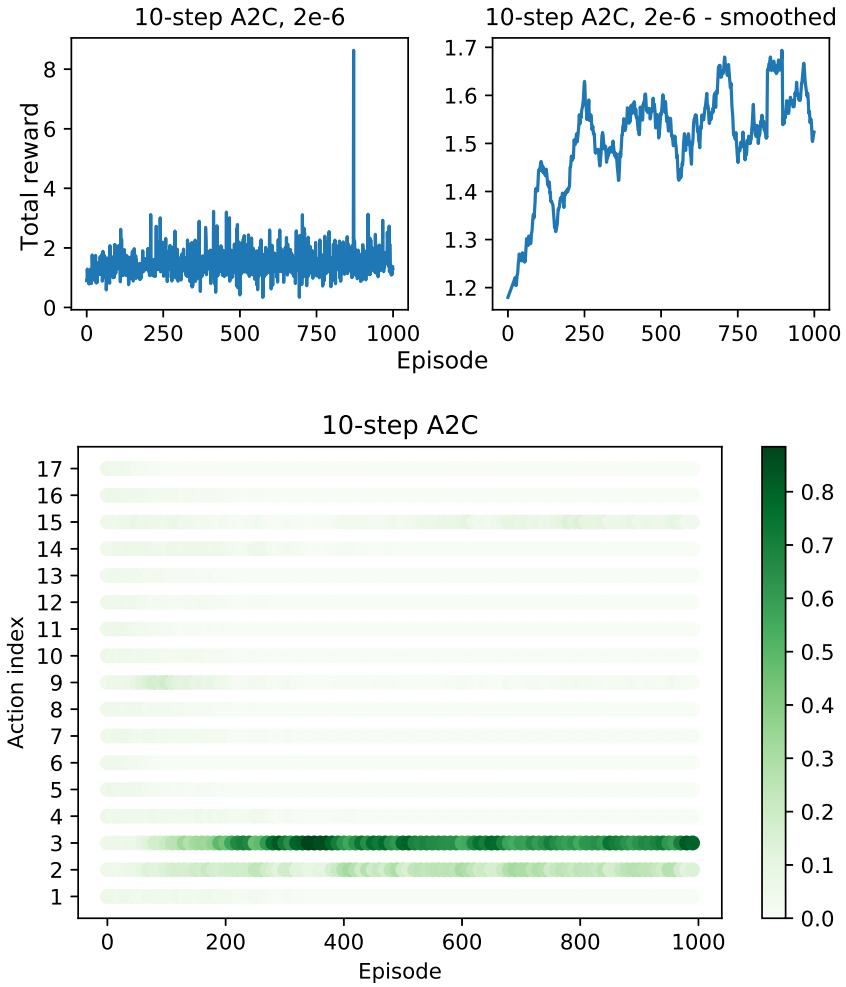

Figure 7.6: 10-step A2C, one component actions, max 800 episodes without improvement. Reward accumulated for each episode of training, corresponding moving average and relative action frequencies across episode.

7.2.7 Testing the best agents

The performance for the best learning rate between the different methods is not wildly different in neither the one nor two component action case. In the one component case, we choose to experiment further only with 10-step A2C. We decide on a learning rate of 2e-6, and we then train the agent allowing a maximum number of actions without improvement to be 800. The results of the training are presented in Figure 7.6.

Evidently, allowing extra episodes increases the performance of the agent. The final policy is based mainly on random neighbourhood local search and

best accept local search. We test this agent with the parameters obtained after the 1000 training episodes. In the testing phase, we use the same instances as for CMCS and IRNLS and to further enable a fair comparison, we allow the agent 1600 non-improving actions, which should correspond to the 800 two component actions, which the two other methods were allowed.

In the two component actions case, we choose to make further experiments in the 20-step A2C setup with a learning rate of 1e-5 and allow 800 actions without improvement for the training phase. The results of the training are presented in Figure 7.7. Here, the agent quickly finds a good policy consisting mainly of random neighbourhood local search followed by first-accept local search, and a reset to the hitherto best solution followed by the introduction of new arcs according to the *greedy3* evaluation measure. We also test this agent with the parameters obtained after the 1000 training episodes and test on the same instances as in the one component case with 800 non-improving actions allowed.

The full results of the methods are presented in Table 7.3 and Table 7.4, which represent the best and average objective value reached over 100 runs, respectively. Clearly, both the one and two component reinforcement learning agents are superior to both the IRNLS and the CMCS heuristic. That is of course encouraging. In addition, it seems that the two component agent performs slightly better than the one component agent, and its runtime is also favorable. Both the reinforcement learning based algorithms have a slightly larger total runtime, since the selection of an action requires the evaluation of the state input into our policy neural network, as compared to only an if statement in the IRNLS and CMCS case. The runtime is still dominated by actually carrying out the actions, and the runtime for this is very similar to both the IRNLS and CMCS heuristic.

The results clearly demonstrate, that it is possible to learn quite good heuristics with the reinforcement learning approach. Inspired by this, we tried to extend our method with the population based approach as we did for IRNLS, this time building the initial population of solutions with the two component actions reinforcement learning agent. The results were good, but to our surprise not better than the what we were able to achieve with PIRNLS, as the reinforcement learning approach only beat PIRNLS on 57 out of 120 instances. Since PIRNLS is a much simpler heuristic it is preferable.

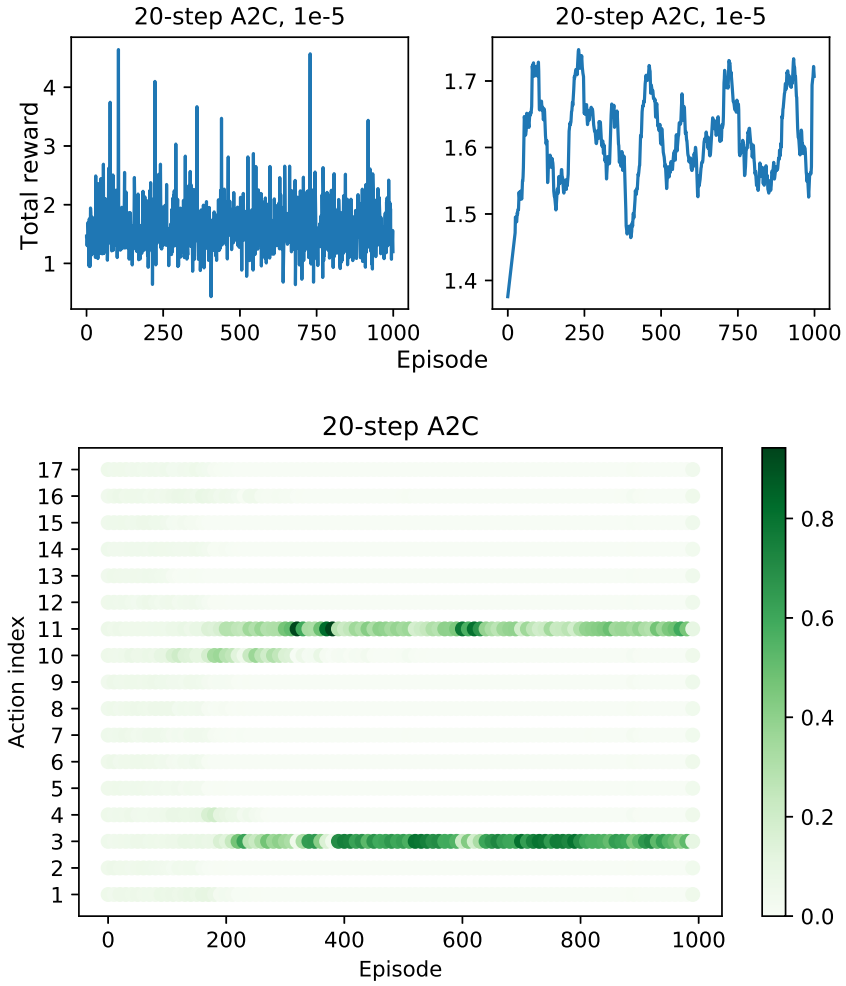

Figure 7.7: 20-step A2C, two component action actions, max 800 episodes without improvement. Reward accumulated for each episode of training, corresponding moving average and relative action frequencies across episodes.

Instance	IRNLS	CMCS	RL-One	RL-Two
N3000	167,859	167,994	**167,760**	167,775
N3001	166,634	166,807	**166,545**	166,580
N3002	167,530	167,783	**167,461**	167,554
N3003	168,282	168,486	168,221	**168,173**
N3004	167,052	167,310	167,099	**167,036**
N3009	**166,812**	167,067	166,910	166,899
N3100	178,692	178,909	178,583	**178,476**
N3101	177,665	178,039	177,565	**177,530**
N3102	178,456	178,712	**178,170**	178,382
N3103	178,785	179,076	178,710	**178,563**
N3104	178,797	179,305	178,860	**178,788**
N3109	177,131	177,541	177,100	**176,961**
N3200	199,188	199,429	**198,643**	198,754
N3201	198,847	199,278	198,463	**198,380**
N3202	199,762	200,232	**199,029**	199,314
N3203	198,503	199,125	198,382	**198,350**
N3204	200,423	200,901	200,337	**200,221**
N3209	197,523	198,131	197,585	**197,186**
N3300	238,373	238,932	237,992	**237,775**
N3301	237,272	238,401	237,293	**237,172**
N3302	239,290	239,517	238,616	**238,208**
N3303	235,381	236,565	235,777	**235,309**
N3304	240,698	241,296	240,254	**240,099**
N3309	237,267	237,687	236,811	**236,452**
N3400	310,583	311,536	309,335	**308,746**
N3401	310,389	310,342	**307,356**	308,742
N3402	311,971	313,654	311,413	**310,136**
N3403	303,821	304,588	**303,328**	303,479
N3404	314,154	314,228	**312,114**	312,615
N3409	310,119	311,725	**308,759**	308,804
N3500	447,509	447,815	446,777	**445,285**
N3501	442,514	445,406	441,780	**441,775**
N3502	446,720	448,828	445,853	**445,701**
N3503	436,599	437,214	432,899	**432,154**
N3504	451,966	454,414	450,893	**448,410**
N3509	446,838	450,274	**446,058**	446,214
N3600	706,045	713,225	700,327	**699,963**
N3601	703,501	707,476	**697,979**	701,100
N3602	699,543	706,633	**695,009**	695,405
N3603	693,158	695,963	**686,896**	687,492
N3604	712,812	723,164	712,521	**710,770**
N3609	706,525	713,309	**699,033**	702,941
N3700	1,221,249	1,230,402	1,211,588	**1,208,888**
N3701	1,211,731	1,229,564	**1,194,869**	1,200,691
N3702	1,198,575	1,212,086	1,185,317	**1,184,914**
N3703	1,195,837	1,211,218	**1,178,003**	1,181,047
N3704	1,221,688	1,237,865	1,212,941	**1,208,104**
N3709	1,204,772	1,225,050	1,191,623	**1,189,740**

Table 7.3: Best results of 100 runs with maximum number iterations without improvement set to 800 (1600 in single component case).

Instance	IRNLS	CMCS	RL-One	RL-Two
N3000	168,185	168,201	168,058	**168,033**
N3001	166,931	166,966	**166,742**	166,823
N3002	167,954	167,940	**167,707**	167,753
N3003	168,637	168,653	**168,387**	168,414
N3004	167,447	167,533	**167,266**	167,311
N3009	167,220	167,248	167,081	**167,054**
N3100	179,420	179,315	179,121	**178,923**
N3101	178,359	178,381	**177,872**	177,999
N3102	179,284	179,142	**178,716**	178,837
N3103	179,559	179,429	179,077	**179,035**
N3104	179,812	179,689	**179,262**	179,337
N3109	177,945	177,870	177,520	**177,495**
N3200	200,915	200,439	199,771	**199,728**
N3201	200,184	200,070	**199,328**	199,370
N3202	201,409	200,944	**200,231**	200,248
N3203	200,030	200,035	199,195	**199,188**
N3204	201,990	201,864	201,163	**201,035**
N3209	199,080	198,802	**198,168**	198,227
N3300	241,121	240,322	239,618	**239,563**
N3301	240,489	240,101	**238,704**	238,882
N3302	241,683	241,183	240,018	**239,632**
N3303	237,922	237,880	236,926	**236,644**
N3304	242,773	242,397	241,609	**241,462**
N3309	240,084	239,454	**238,362**	238,394
N3400	315,219	313,909	312,289	**312,211**
N3401	314,028	312,889	**311,232**	311,627
N3402	315,995	316,270	**314,040**	314,045
N3403	307,704	306,491	**305,623**	305,674
N3404	318,524	316,579	315,664	**315,596**
N3409	314,647	313,773	312,052	**311,717**
N3500	455,357	453,321	451,109	**450,268**
N3501	452,260	450,969	**446,994**	447,135
N3502	453,606	454,661	**449,684**	450,022
N3503	441,992	439,904	**437,601**	437,651
N3504	457,859	457,800	455,484	**454,453**
N3509	455,158	454,407	**450,675**	450,919
N3600	720,351	720,605	712,382	**711,817**
N3601	717,891	717,861	**709,004**	710,812
N3602	714,274	714,447	706,066	**705,847**
N3603	698,872	702,976	695,971	**695,030**
N3604	729,582	728,778	721,618	**719,527**
N3609	718,765	719,250	**709,085**	711,302
N3700	1,244,965	1,245,689	**1,228,992**	1,229,796
N3701	1,235,402	1,242,672	**1,218,006**	1,219,528
N3702	1,221,120	1,229,390	**1,206,596**	1,207,466
N3703	1,216,735	1,221,141	1,199,785	**1,199,233**
N3704	1,249,304	1,253,719	1,233,545	**1,231,707**
N3709	1,232,427	1,236,812	1,211,017	**1,210,962**

Table 7.4: Average results of 100 runs with maximum number iterations without improvement set to 800 (1600 in single component case).

7.3 Population based iterated random neighbourhood local search revisited

The knowledge we gained through the reinforcement learning experiments can be used to modify the pool of mutations we use for population based iterated random neighbourhood local search. By limiting ourselves to random neighbourhood local search, kicking out k arcs with largest cost per unit, kicking out k arcs and introducing k arcs according to the evaluation measure *greedy3* which were some of the preferred actions by the RL agents, we were able to achieve significantly better results from PIRNLS. With parameters $n_{\max} = 800$, $n = 50, N = 250, \alpha = 1.03, \beta = 80$ and $\kappa = 7$ we were in one run with approximately 40 minutes of computation per instance able to beat the heuristic by Buson et al. [5] on 81 out of 120 instances. The performance is illustrated in Figure 7.8 and can be found in more detail in the Appendix, Table B.9-B.16. Interestingly, PIRNLS in particular performs well on the instances with large fixed costs.

We carried out a Wilcoxon signed rank test [54] with the relative performance measure, and the difference is clearly statistically significant with a p-value of 0.00008 in favor of PIRNLS. It may be possible to improve the heuristic further by a structured search in the parameter space, specifically κ and β seem to be important parameters.

Finally, we report the best results we have seen on each instance during all our experiments with both PIRNLS and the reinforcement learning approach. We improve the best solutions in the literature on 112 out of 120 considered instances. The results are visualized in Figure 7.9, and are also presented in Table B.17-B.24 in the Appendix.

7.4 Discussion and outlook

7.4.1 The reinforcement learning framework

Our experiments clearly indicate, that the use of reinforcement learning techniques for combinatorial optimization has potential. There are many ways in which our experiments could be improved, especially the search over hyperparameters. This also highlights an issue with reinforcement learning; a lot of computation is needed. But if one considers the reinforcement learning framework as a hyperheuristic in the same vein as Conditional Markov chain search, the computation needed is not bad by comparison. We are able to search in a much larger space of possible heuristics arguably in shorter computation time than CMCS. The bulk of the effort is in the training phase,

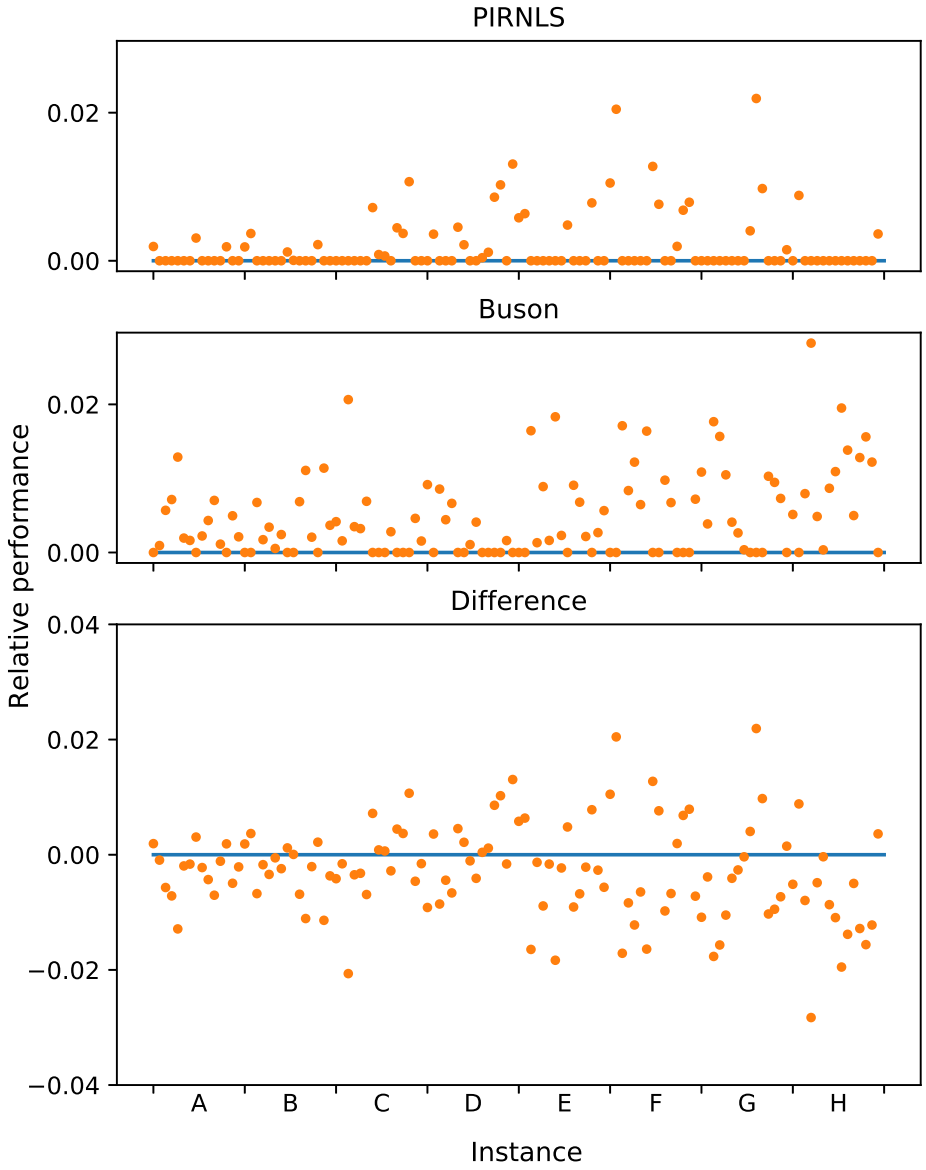

Figure 7.8: Relative performance of one run of PIRNLS compared to the best in the literature across problem instances. The instance classes are indicated by A-H.

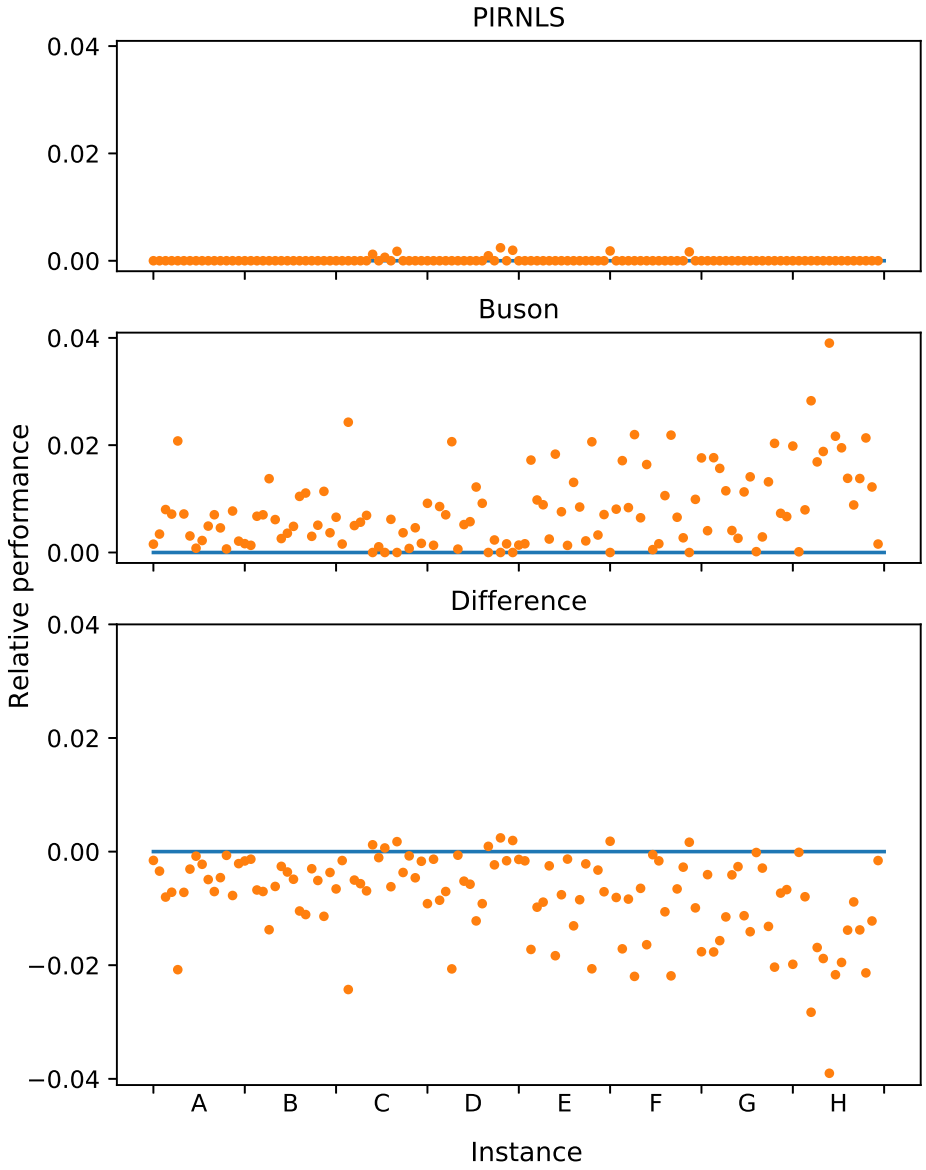

Figure 7.9: Relative performance of our best results compared to the best in the literature across problem instances. The instance classes are indicated by A-H.

since once the training phase is over, the final heuristic is merely a function (in our experiments a neural network), which given a solution as input outputs a probability distribution over actions. Thus, the speed of the final heuristic is comparable to 'standard' metaheuristics, and the ones developed by CMCS.

The techniques we have used are not well equipped for handling time dependent actions, as we have to explicitly encode the history in the states. This means, that the agent has some difficulties discovering longer term strategies - this issue may be alleviated by development in reinforcement learning techniques, and possibly also by using a recurrent neural neural network as a function approximator. Another issue with our approach, is that we do not explicitly take the execution time of an action into consideration. This could perhaps be handled by the use of a semi Markov decision process as the underlying model instead, but it is unclear exactly how. Otherwise, we could encode the time spent in the state, and let each episode terminate after a fixed amount of time instead of after k actions with no improvement to the objective value. In addition, it may be possible to improve the framework by changing the reward structure, e.g. by removing the scaling factor.

Currently, the development in reinforcement learning techniques is happening at an extremely fast pace. This of course increases the potential for success with reinforcement learning methods for combinatorial optimization. Several advances have been made since the introduction of DQN and A2C based agents, and adopting some of the newest methods may also improve the performance. We could also try to utilize rollouts and planning [46] to improve the performance of the reinforcement learning agents. These concepts were for example vital for the chess and *go* engine developed by DeepMind [41, 42]. The use of rollout and planning methods allows the agent to easier discover good long term strategies, but the computational effort is increased.

Another prospect of future research could be the case of fine-grained actions. As discussed, this setup is more challenging and requires additional computational effort, but it has the benefit, that we are not restrained by the perhaps limited potential of the composite actions. With the evolution of graph neural networks, this approach should become more promising.

Finally, we find it important to stress, that the proposed reinforcement learning framework is in no way limited to the fixed charge transportation problem. The method we developed is applicable for all combinatorial optimization problems where mutations can be applied to a feasible solution to move to another one. Given a reasonable set of hyperparameters, our method is able to automatize the search for a good heuristic. The approach with composite actions which we mainly explored of course requires, that a set of composite actions which have potential to reach a good solution if composed properly exists. For some problems there exist multiple natural local

search operators, and for such problems this approach would likely work tremendously well. For any problem where conditional Markov chain search could work, our approach could surely work as well, and probably in many more cases as well.

7.4.2 Population based iterated random neighbourhood local search

It came as quite a surprise, that we were able to achieve results on par with or even better than the best in the literature with this rather simple heuristic. The experiments indicate, that the key ingredient was the introduction of random neighbourhood local search. There may be potential for improving the heuristic further by tuning the parameters, which we did not spend much time on. For example, the neighbourhood size could be changed. In addition, one could introduce restrictions on what kind of solutions are accepted instead of always accepting the best exchange in the neighbourhood. The novel population based method we introduced could also be applied in any domain where iterated local search can be applied.

A hybridization of PIRNLS and the reduced cost iterated local search heuristic of Buson et al. [5] should also be possible, and would likely lead to great results on the fixed charge transportation problem.

Chapter 8

Conclusion

Recent advances in reinforcement learning, propelled by deep learning, has sparked interest in the application of reinforcement learning methods to new domains. For reinforcement learning to be applicable, the domain needs to have certain traits, such as a natural objective and the possibility to acquire data incrementally and inexpensively, e.g. by simulation. Combinatorial optimization problems have exactly these traits, but how to define the nature of actions and rewards is not obvious. In this thesis, we proposed a novel framework for combinatorial optimization with reinforcement learning by focusing on actions which transform one feasible solution into another. For many combinatorial optimization problems, a plethora of such actions have been studied in the literature, for example various local search operators or random perturbations. Our framework is a method for learning incrementally from experience how to compose such actions, and may be seen as a kind of hyperheuristic. We discussed how the proposed framework can cater to different types of action based on granularity. In the fixed charge transportation problem, the available actions could for example be basic-exchanges, and the heuristic we learn would then be a local search guided by a learned value function, which evaluates the quality of a solution. A less challenging setup is when the available actions have a larger immediate impact on the solution. For example, an action could be a precomposed basic-exchange strategy such as a first-accept local search or the introduction of random arcs into the solution. We mainly focused on this composite action case in this thesis, but the more fine-grained approach could be an interesting prospect for further research.

Our experiments showed, that an agent trained in the proposed framework can, in reasonable computation time, learn heuristics which are competitive with the state-of-the-art. As the agent is naturally constrained by the actions it is allowed, the framework has the most potential when a number

of strong components are available. For example if there are several natural local search operators, which is the case for many facility location and vehicle routing problems. By contrast, the space of local search operators is more limited in the fixed charge transportation problem, since they mostly rely on basic-exchanges. However, it is certainly possible to make strong heuristics based only on basic-exchanges, and in this thesis we developed new heuristic, random neighbourhood local search, which was the basis of most strong heuristics discovered by the reinforcement learning agents.

Our experiments also indicated, that it can be difficult for the reinforcement learning agents to learn to utilize the history of the search. This could perhaps be alleviated by development in reinforcement learning techniques, or the use of a function approximator with 'memory' such as a recurrent neural network. To accommodate the lack of memory utilization, we developed a new heuristic which was similar to the ones learned by the reinforcement learning agents, but with explicit use of memory to sometimes reset the solution. The proposed heuristic, iterated random neighbourhood local search, was combined with a population based method for guiding the perturbation phase, and this resulted in a conceptually simple and easy to implement algorithm, which improved the best solution in literature on 81 out of 120 well known instances in one run with similar or even shorter computation time than what hitherto constituted the state-of-the-art heuristic. In total, we improved the best solution in the literature on 112 out of 120 instances during our experiments. The proposed algorithm could easily be hybridized with other heuristics, and a prospect for future research could be to somehow utilize reduced cost information, for example to guide the perturbation phase as done by Buson et al. [5].

In conclusion, we developed a novel framework for combinatorial optimization with reinforcement learning, and our discussion and experiments indicate that it may for many types of combinatorial optimization problems be an effective framework for finding new, strong heuristics based on either existing subordinate heuristics or simple mutation operators.

Bibliography

[1] Michel L. Balinski. Fixed-cost transportation problems. *Naval Research Logistics Quarterly*, 8(1):41–54, 1961.

[2] Peter W. Battaglia, Jessica B. Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinícius Flores Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, Çaglar Gülçehre, Francis Song, Andrew J. Ballard, Justin Gilmer, George E. Dahl, Ashish Vaswani, Kelsey Allen, Charles Nash, Victoria Langston, Chris Dyer, Nicolas Heess, Daan Wierstra, Pushmeet Kohli, Matthew Botvinick, Oriol Vinyals, Yujia Li, and Razvan Pascanu. Relational inductive biases, deep learning, and graph networks. *CoRR*, abs/1806.01261, 2018.

[3] Irwan Bello, Hieu Pham, Quoc V. Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. *CoRR*, abs/1611.09940, 2016.

[4] Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: a methodological tour d'horizon. *CoRR*, abs/1811.06128, 2018.

[5] Erika Buson, Roberto Roberti, and Paolo Toth. A reduced-cost iterated local search heuristic for the fixed-charge transportation problem. *Operations Research*, 62(5):1095–1106, 2014.

[6] V. Chvátal. Edmonds polytopes and a hierarchy of combinatorial problems. *Discrete Mathematics*, 4(4):305 – 337, 1973.

[7] P. Cowling and E. Soubeiga. Neighborhood structures for personnel scheduling: A summit meeting scheduling problem. *proceedings of the 3rd International Conference on the Practice and Theory of Automated Timetabling*, 2000.

[8] Will Dabney, Georg Ostrovski, David Silver, and Rémi Munos. Implicit quantile networks for distributional reinforcement learning. *CoRR*, abs/1806.06923, 2018.

[9] Hanjun Dai, Bo Dai, and Le Song. Discriminative embeddings of latent variable models for structured data. *CoRR*, abs/1603.05629, 2016.

[10] Hanjun Dai, Elias B. Khalil, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. *CoRR*, abs/1704.01665, 2017.

[11] Meire Fortunato, Mohammad Gheshlaghi Azar, Bilal Piot, Jacob Menick, Ian Osband, Alex Graves, Vlad Mnih, Rémi Munos, Demis Hassabis, Olivier Pietquin, Charles Blundell, and Shane Legg. Noisy networks for exploration. *CoRR*, abs/1706.10295, 2017.

[12] M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, New York, 1979.

[13] Michel Gendreau and Jean-Yves Potvin. *Handbook of Metaheuristics*. Springer Publishing Company, Incorporated, 2nd edition, 2010.

[14] N Srivastava Geoffrey Hinton and Kevin Swersky, 2021.

[15] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. Neural message passing for quantum chemistry. *CoRR*, abs/1704.01212, 2017.

[16] Fred Glover. Parametric ghost image processes for fixed-charge problems: A study of transportation networks. *Journal of Heuristics*, 11(4):307–336, Jul 2005.

[17] Fred Glover and Manuel Laguna. *Tabu Search*. Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[18] A. Goldberg and R. Tarjan. Solving minimum-cost flow problems by successive approximation. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC '87, pages 7–18, New York, NY, USA, 1987. ACM.

[19] Ralph E. Gomory. Outline of an algorithm for integer solutions to linear programs. *Bull. Amer. Math. Soc.*, 64(5):275–278, 09 1958.

[20] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[21] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 1024–1034. Curran Associates, Inc., 2017.

[22] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. Springer New York Inc., New York, NY, USA, 2001.

[23] Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Daniel Horgan, Bilal Piot, Mohammad Gheshlaghi Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. *CoRR*, abs/1710.02298, 2017.

[24] Warren M. Hirsch and George B. Dantzig. The fixed charge problem. *Naval Research Logistics Quarterly*, 15(3):413–424, 1968.

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

[26] J J Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences*, 79(8):2554–2558, 1982.

[27] Peter J. Huber. Robust estimation of a location parameter. *Ann. Math. Statist.*, 35(1):73–101, 03 1964.

[28] Tim H. Hultberg and Domingos M. Cardoso. The teacher assignment problem: A special case of the fixed charge transportation problem. *European Journal of Operational Research*, 101(3):463–473, 1997.

[29] Daniel Karapetyan and Boris Goldengorin. Conditional markov chain search for the simple plant location problem improves upper bounds on twelve körkel-ghosh instances. 11 2017.

[30] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014. cite arxiv:1412.6980Comment: Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015.

[31] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *CoRR*, abs/1609.02907, 2016.

[32] Long-Ji Lin. Reinforcement learning for robots using neural networks /. 01 1993.

[33] Robert James Mcfarlane. A survey of exploration strategies in reinforcement learning. 2003.

[34] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *CoRR*, abs/1602.01783, 2016.

[35] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, February 2015.

[36] Katta G. Murty. Solving the fixed charge problem by ranking the extreme points. *Operations Research*, 16(2):268–279, 1968.

[37] James B. Orlin. A polynomial time primal network simplex algorithm for minimum cost flows. *Mathematical Programming*, 78(2):109–129, Aug 1997.

[38] Martin Riedmiller. Neural fitted q iteration – first experiences with a data efficient neural reinforcement learning method. In João Gama, Rui Camacho, Pavel B. Brazdil, Alípio Mário Jorge, and Luís Torgo, editors, *Machine Learning: ECML 2005*, pages 317–328, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[39] G. A. Rummery and M. Niranjan. On-line Q-learning using connectionist systems. Technical Report TR 166, Cambridge University Engineering Department, Cambridge, England, 1994.

[40] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

[41] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman,

Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, January 2016.

[42] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550:354–359, 10 2017.

[43] B.F. Skinner. Reinforcement today. *American Psychologist, 13(3)*, pages 94–99, 1958.

[44] David I. Steinberg. The fixed charge problem. *Naval Research Logistics Quarterly*, 17(2):217–235, 1970.

[45] Minghe Sun, Jay E. Aronson, Patrick G. McKeown, and Dennis Drinka. A tabu search heuristic procedure for the fixed charge transportation problem. *European Journal of Operational Research*, 106(2):441 – 456, 1998.

[46] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

[47] Richard S Sutton, David A. McAllester, Satinder P. Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In S. A. Solla, T. K. Leen, and K. Müller, editors, *Advances in Neural Information Processing Systems 12*, pages 1057–1063. MIT Press, 2000.

[48] John N. Tsitsiklis. Asynchronous stochastic approximation and q-learning. *Mach. Learn.*, 16(3):185–202, September 1994.

[49] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks. *CoRR*, abs/1710.10903, 2017.

[50] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 2692–2700. Curran Associates, Inc., 2015.

[51] Jane X. Wang, Zeb Kurth-Nelson, Dhruva Tirumala, Hubert Soyer, Joel Z. Leibo, Rémi Munos, Charles Blundell, Dharshan Kumaran, and Matthew Botvinick. Learning to reinforcement learn. *CoRR*, abs/1611.05763, 2016.

[52] Xiaolong Wang, Ross B. Girshick, Abhinav Gupta, and Kaiming He. Non-local neural networks. *CoRR*, abs/1711.07971, 2017.

[53] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3):279–292, May 1992.

[54] Frank Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):80–83, 1945.

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

[56] Ronald J. Williams and Jing Peng. Function optimization using connectionist reinforcement learning algorithms. 1991.

Appendix A

About the Code

The code for this thesis is split into two parts. One part is in Java and is responsible for manipulating solutions to instances of the fixed charge transportation problem. The other part is in Python, and is responsible for controlling the reinforcement learning agents, updating their parameters and invoking methods in Java. The code is available on GitHub[1]. On GitHub, there is a small guide to how to get started with the code, if you have any trouble you are welcome to email me on `peteremiltybirk@gmail.com`. Here, we explain a little bit of the structure of the code, we start with the Java part.

A.1 Java code for the fixed charge transportation problem

The code for manipulating solutions to the fixed charge transportation problem is an extension of the framework by Andreas Klose[2]. We have added 3 Java files to the framework: PEheur, RL_composite and CMCS, the documentation can be found on GitHub in the Java folder.

In short, The PEheur file contains implementations of the three greedy measures, a class RandomCollection, which enables us to draw values from a set with probabilities proportional to an evaluation measure, a method for introducing arcs into the basis with probability according to an evaluation measure, a method for modified-cost local search, two methods for RNLS, two methods for two versions of IRNLS and one for PIRNLS. Notice, that the implementation of PIRNLS is slightly different than what is described in Algorithm 4, and to achieve the same results as we do in the thesis, the

[1] `https://github.com/Tybirk/RL-FCTP`

[2] `http://home.math.au.dk/aklose/FCTP/`

maximum number of iterations without improvement should be set twice as large (the runtime is the same).

The RL_composite class is the bridge between the Python code and the Java code, it is the `do_action` method, that the reinforcement learning agents invoke to do actions. If one wishes to change the available actions for an agent, this code needs to be changed.

The CMCS class contains methods for trying a given configuration for a given time on a given instance. The 'higher levels' of the CMCS procedure is written in Python.

A.2 Python code for controlling the reinforcement learning agents

We decided to write the code for the reinforcement learning in Python, since many frameworks exist both for function approximation and reinforcement learning. However, since the code for manipulating solutions to the FCTP is in Java, we need some way to invoke Java methods from Python, and to that end we use Pyjnius[3]. For the implementation of the reinforcement learning agents, we owe most of the code to MediPixel[4], which is a repository based on PyTorch and Gym.

The Python code is split in 3 parts. The first one is an implementation of Conditional Markov chain search, which is quite straightforward. The second part is the implementation of the FCTP environment. If you want to try this framework for a different combinatorial optimization problem, the only thing you need to do is to create e.g. a travelling salesman problem environment, which just needs to adhere to the 'Gym' interface.

A.2.1 The Gym interface

The Gym interface contains 4 methods that must be overwritten in the implementation of a new environment. `__init__`, `step`, `reset` and `render`. `__init__` initializes the parameters needed in the environment, for example the number of possible actions, instance names and lower and upper bounds. `step` applies an action (here we call the method `do_action` in RL_composite), sees the reward and updates the state. `reset` resets the environment, in our case we choose a new instance of the FCTP and generate an initial solution for this instance. `render` is a method detailing what output to show after

[3] `https://pyjnius.readthedocs.io/en/stable/`

[4] `https://github.com/medipixel/rl_algorithms/`

each step. Our implementation of the FCTP environment can be found in the folder `Python/gym-FCTP/gym_FCTP/envs/` on GitHub.

A.2.2 Implementation of the reinforcement learning agents

The fourth part of the code is the implementation of the RL agents. Everything in the folder `Python/rl_algorithms/algorithms` except for the n-step A2C agent and the random agent was taken from MediPixels repository, and the documentation is most easily read there. The hyperparameters for the runs can be adjusted in the folder `Python/rl_algorithms/fctp-v1`. To train an RL agent on the FCTP, you for example call

```
python run_fctp_v1.py --algo n_step_a2c
```

In the file `run_fctp_v1.py`, you also specify certain hyperparameters related to the training. For more information, see the README in our GitHub repository.

Appendix B

Tables

B.1 Population based iterated random neighbourhood local search results

B.1.1 Version 1

Here, we present the results of the PIRNLS runs described in subsection 7.1.2 and also list the hitherto best solutions found in literature, all by Buson et al. [5].

Instance	PIRNLS	Buson
N3000	167,800	**167,737**
N3001	166,516	**166,473**
N3002	**167,457**	167,541
N3003	**168,165**	168,198
N3004	**167,042**	167,074
N3005	**167,328**	167,387
N3006	**165,492**	165,562
N3007	167,052	**166,980**
N3008	**165,379**	165,384
N3009	**166,812**	166,882
N300A	**167,172**	167,192
N300B	168,329	**168,327**
N300C	165,138	**165,123**
N300D	**166,014**	166,062
N300E	169,204	**169,177**

Table B.1: Class A instances

Instance	PIRNLS	Buson
N3100	**178,246**	178,299
N3101	177,322	**177,152**
N3102	**178,159**	178,292
N3103	**178,453**	178,489
N3104	**178,733**	178,753
N3105	**177,521**	177,726
N3106	176,247	**176,161**
N3107	**177,275**	177,299
N3108	**175,711**	175,810
N3109	**176,928**	177,089
N310A	**178,496**	178,558
N310B	179,249	**179,173**
N310C	175,358	**175,298**
N310D	**176,829**	176,941
N310E	179,396	**179,351**

Table B.2: Class B instances

Instance	PIRNLS	Buson
N3200	**198,324**	198,517
N3201	198,095	**197,898**
N3202	**198,956**	199,208
N3203	**198,231**	198,252
N3204	200,207	**200,059**
N3205	197,781	**197,769**
N3206	196,259	**196,071**
N3207	**196,909**	196,937
N3208	195,862	**195,639**
N3209	**196,920**	197,086
N320A	199,446	**199,377**
N320B	200,129	**200,057**
N320C	**194,996**	195,019
N320D	**197,082**	197,085
N320E	199,481	**199,328**

Table B.3: Class C instances

Instance	PIRNLS	Buson
N3300	237,327	**237,220**
N3301	236,713	**236,557**
N3302	**237,648**	237,746
N3303	**234,999**	235,481
N3304	**239,120**	239,606
N3305	236,552	**236,106**
N3306	**233,764**	234,021
N3307	234,025	**233,777**
N3308	**232,805**	233,312
N3309	236,246	**236,227**
N330A	239,861	**239,517**
N330B	239,400	**238,615**
N330C	233,001	**232,750**
N330D	234,771	**234,418**
N330E	236,711	**236,167**

Table B.4: Class D instances

Instance	PIRNLS	Buson
N3400	308,462	**307,279**
N3401	307,181	**306,404**
N3402	310,841	**310,610**
N3403	**302,561**	302,880
N3404	**311,426**	311,985
N3405	307,843	**307,561**
N3406	**304,089**	304,424
N3407	**302,747**	302,817
N3408	305,278	**305,135**
N3409	**308,118**	308,254
N340A	**312,126**	312,607
N340B	311,370	**309,900**
N340C	304,989	**304,055**
N340D	305,025	**304,869**
N340E	**306,883**	307,315

Table B.5: Class E instances

Instance	PIRNLS	Buson
N3500	443,436	**440,880**
N3501	438,514	**438,005**
N3502	**443,400**	444,003
N3503	**430,572**	431,410
N3504	449,335	**449,277**
N3505	440,769	**439,854**
N3506	**436,937**	437,289
N3507	433,459	**432,158**
N3508	439,686	**438,444**
N3509	**442,734**	443,750
N350A	**448,222**	449,041
N350B	446,109	**444,519**
N350C	443,147	**440,203**
N350D	442,460	**440,734**
N350E	**443,483**	443,741

Table B.6: Class F instances

Instance	PIRNLS	Buson
N3600	**695,307**	698,688
N3601	696,870	**694,071**
N3602	691,391	**691,340**
N3603	**684,178**	684,562
N3604	707,553	**706,189**
N3605	**692,851**	692,928
N3606	695,115	**693,612**
N3607	684,142	**681,441**
N3608	**691,154**	693,838
N3609	699,705	**691,868**
N360A	708,475	**707,272**
N360B	**701,188**	703,234
N360C	701,147	**700,759**
N360D	704,723	**700,759**
N360E	698,405	**693,381**

Table B.7: Class G instances

Instance	PIRNLS	Buson
N3700	1,202,480	**1,197,613**
N3701	1,186,935	**1,186,310**
N3702	1,176,000	**1,172,668**
N3703	1,178,344	**1,170,230**
N3704	1,203,732	**1,203,084**
N3705	1,186,963	**1,186,342**
N3706	**1,181,932**	1,196,633
N3707	**1,163,554**	1,169,810
N3708	**1,190,356**	1,192,859
N3709	**1,182,509**	1,186,592
N370A	1,220,614	**1,216,295**
N370B	**1,194,478**	1,199,189
N370C	1,200,215	**1,198,455**
N370D	**1,194,095**	1,198,455
N370E	1,177,967	**1,176,172**

Table B.8: Class H instances

B.1.2 Version 2

Here, we present the results of the PIRNLS runs described in subsection 7.3 and also list the hitherto best solutions found in literature, all by Buson et al. [5].

Instance	PIRNLS	Buson
N3000	167,763	**167,737**
N3001	**166,458**	166,473
N3002	**167,470**	167,541
N3003	**168,158**	168,198
N3004	**166,967**	167,074
N3005	**167,370**	167,387
N3006	**165,525**	165,562
N3007	167,003	**166,980**
N3008	**165,366**	165,384
N3009	**166,789**	166,882
N300A	**167,125**	167,192
N300B	**168,314**	168,327
N300C	165,143	**165,123**
N300D	**166,014**	166,062
N300E	**169,138**	169,177

Table B.9: Class A instances

Instance	PIRNLS	Buson
N3100	178,359	**178,299**
N3101	177,251	**177,152**
N3102	**178,101**	178,292
N3103	**178,464**	178,489
N3104	**178,688**	178,753
N3105	**177,708**	177,726
N3106	**176,078**	176,161
N3107	177,321	**177,299**
N3108	175,811	**175,810**
N3109	**176,953**	177,089
N310A	**178,420**	178,558
N310B	**179,149**	179,173
N310C	175,327	**175,298**
N310D	**176,729**	176,941
N310E	**179,307**	179,351

Table B.10: Class B instances

Instance	PIRNLS	Buson
N3200	**198,395**	198,517
N3201	**197,843**	197,898
N3202	**198,759**	199,208
N3203	**198,177**	198,252
N3204	**199,953**	200,059
N3205	**197,562**	197,769
N3206	196,334	**196,071**
N3207	196,959	**196,937**
N3208	195,655	**195,639**
N3209	**197,011**	197,086
N320A	199,552	**199,377**
N320B	200,187	**200,057**
N320C	195,344	**195,019**
N320D	**196,887**	197,085
N320E	**199,277**	199,328

Table B.11: Class C instances

Instance	PIRNLS	Buson
N3300	**236,806**	237,220
N3301	236,704	**236,557**
N3302	**237,326**	237,746
N3303	**235,178**	235,481
N3304	**239,452**	239,606
N3305	236,328	**236,106**
N3306	234,127	**234,021**
N3307	**233,715**	233,777
N3308	**233,143**	233,312
N3309	236,242	**236,227**
N330A	239,564	**239,517**
N330B	238,905	**238,615**
N330C	233,238	**232,750**
N330D	**234,338**	234,418
N330E	236,595	**236,167**

Table B.12: Class D instances

Instance	PIRNLS	Buson
N3400	307,661	**307,279**
N3401	306,963	**306,404**
N3402	**309,716**	310,610
N3403	**302,828**	302,880
N3404	**311,345**	311,985
N3405	**307,459**	307,561
N3406	**303,155**	304,424
N3407	**302,658**	302,817
N3408	305,456	**305,135**
N3409	**307,661**	308,254
N340A	**312,222**	312,607
N340B	**309,736**	309,900
N340C	304,565	**304,055**
N340D	**304,616**	304,869
N340E	**306,970**	307,315

Table B.13: Class E instances

Instance	PIRNLS	Buson
N3500	442,531	**440,880**
N3501	440,008	**438,005**
N3502	**442,236**	444,003
N3503	**430,455**	431,410
N3504	**448,154**	449,277
N3505	**439,292**	439,854
N3506	**435,415**	437,289
N3507	433,178	**432,158**
N3508	439,164	**438,444**
N3509	**442,528**	443,750
N350A	**448,368**	449,041
N350B	444,729	**444,519**
N350C	441,068	**440,203**
N350D	441,849	**440,734**
N350E	**442,829**	443,741

Table B.14: Class F instances

Instance	PIRNLS	Buson
N3600	**696,619**	698,688
N3601	**693,262**	694,071
N3602	**688,563**	691,340
N3603	**681,479**	684,562
N3604	**704,265**	706,189
N3605	**692,175**	692,928
N3606	**693,054**	693,612
N3607	**681,367**	681,441
N3608	694,596	**693,838**
N3609	696,572	**691,868**
N360A	708,568	**707,272**
N360B	**700,893**	703,234
N360C	**698,956**	700,759
N360D	**699,290**	700,759
N360E	693,659	**693,381**

Table B.15: Class G instances

Instance	PIRNLS	Buson
N3700	**1,195,484**	1,197,613
N3701	1,190,450	**1,186,310**
N3702	**1,169,902**	1,172,668
N3703	**1,161,515**	1,170,230
N3704	**1,200,876**	1,203,084
N3705	**1,186,220**	1,186,342
N3706	**1,193,465**	1,196,633
N3707	**1,166,691**	1,169,810
N3708	**1,186,007**	1,192,859
N3709	**1,180,929**	1,186,592
N370A	**1,214,254**	1,216,295
N370B	**1,194,813**	1,199,189
N370C	**1,193,175**	1,198,455
N370D	**1,193,751**	1,198,455
N370E	1,177,190	**1,176,172**

Table B.16: Class H instances

B.2 Our best results

Here, we present the best results we saw on all instances during our experiments.

Instance	PIRNLS	Buson
N3000	**167,716**	167,737
N3001	**166,418**	166,473
N3002	**167,441**	167,541
N3003	**168,158**	168,198
N3004	**166,900**	167,074
N3005	**167,324**	167,387
N3006	**165,492**	165,562
N3007	**166,974**	166,980
N3008	**165,366**	165,384
N3009	**166,776**	166,882
N300A	**167,125**	167,192
N300B	**168,274**	168,327
N300C	**165,116**	165,123
N300D	**165,987**	166,062
N300E	**169,138**	169,177

Table B.17: Class A instances

Instance	PIRNLS	Buson
N3100	**178,246**	178,299
N3101	**177,116**	177,152
N3102	**178,101**	178,292
N3103	**178,387**	178,489
N3104	**178,489**	178,753
N3105	**177,521**	177,726
N3106	**176,072**	176,161
N3107	**177,232**	177,299
N3108	**175,711**	175,810
N3109	**176,881**	177,089
N310A	**178,420**	178,558
N310B	**179,138**	179,173
N310C	**175,230**	175,298
N310D	**176,729**	176,941
N310E	**179,307**	179,351

Table B.18: Class B instances

Instance	PIRNLS	Buson
N3200	**198,324**	198,517
N3201	**197,843**	197,898
N3202	**198,678**	199,208
N3203	**198,144**	198,252
N3204	**199,873**	200,059
N3205	**197,562**	197,769
N3206	196,115	**196,071**
N3207	**196,909**	196,937
N3208	195,655	**195,639**
N3209	**196,920**	197,086
N320A	199,446	**199,377**
N320B	**199,927**	200,057
N320C	**194,996**	195,019
N320D	**196,887**	197,085
N320E	**199,272**	199,328

Table B.19: Class C instances

Instance	PIRNLS	Buson
N3300	**236,806**	237,220
N3301	**236,502**	236,557
N3302	**237,326**	237,746
N3303	**234,999**	235,481
N3304	**239,120**	239,606
N3305	**236,075**	236,106
N3306	**233,764**	234,021
N3307	**233,442**	233,777
N3308	**232,805**	233,312
N3309	**235,890**	236,227
N330A	239,555	**239,517**
N330B	**238,536**	238,615
N330C	232,865	**232,750**
N330D	**234,338**	234,418
N330E	236,231	**236,167**

Table B.20: Class D instances

Instance	PIRNLS	Buson
N3400	**307,188**	307,279
N3401	**306,261**	306,404
N3402	**309,673**	310,610
N3403	**302,492**	302,880
N3404	**311,345**	311,985
N3405	**307,404**	307,561
N3406	**303,155**	304,424
N3407	**302,290**	302,817
N3408	**305,047**	305,135
N3409	**307,397**	308,254
N340A	**312,126**	312,607
N340B	**309,736**	309,900
N340C	**302,680**	304,055
N340D	**304,562**	304,869
N340E	**306,883**	307,315

Table B.21: Class E instances

Instance	PIRNLS	Buson
N3500	441,167	**440,880**
N3501	**437,208**	438,005
N3502	**442,236**	444,003
N3503	**430,455**	431,410
N3504	**447,235**	449,277
N3505	**439,292**	439,854
N3506	**435,415**	437,289
N3507	**432,115**	432,158
N3508	**438,290**	438,444
N3509	**442,423**	443,750
N350A	**446,823**	449,041
N350B	**443,804**	444,519
N350C	**439,855**	440,203
N350D	440,967	**440,734**
N350E	**442,483**	443,741

Table B.22: Class F instances

Instance	PIRNLS	Buson
N3600	**695,307**	698,688
N3601	**693,220**	694,071
N3602	**688,563**	691,340
N3603	**681,479**	684,562
N3604	**704,078**	706,189
N3605	**692,175**	692,928
N3606	**693,054**	693,612
N3607	**679,141**	681,441
N3608	**691,154**	693,838
N3609	**691,834**	691,868
N360A	**706,885**	707,272
N360B	**700,231**	703,234
N360C	**696,844**	700,759
N360D	**699,290**	700,759
N360E	**692,120**	693,381

Table B.23: Class G instances

Instance	PIRNLS	Buson
N3700	**1,189,266**	1,197,613
N3701	**1,186,250**	1,186,310
N3702	**1,169,902**	1,172,668
N3703	**1,161,515**	1,170,230
N3704	**1,195,324**	1,203,084
N3705	**1,179,777**	1,186,342
N3706	**1,181,932**	1,196,633
N3707	**1,163,554**	1,169,810
N3708	**1,186,007**	1,192,859
N3709	**1,180,929**	1,186,592
N370A	**1,212,656**	1,216,295
N370B	**1,194,478**	1,199,189
N370C	**1,191,194**	1,198,455
N370D	**1,193,751**	1,198,455
N370E	**1,175,728**	1,176,172

Table B.24: Class H instances

Appendix C

Plots

C.1 Action analysis

In this section, we present the plots detailing the actions of the agents as training progresses. We show plots for each of the 15 hyperparameter settings we tested for both one and two component actions. The plots indicate the relative frequency of each action every 10th episode.

C.1.1 One component actions

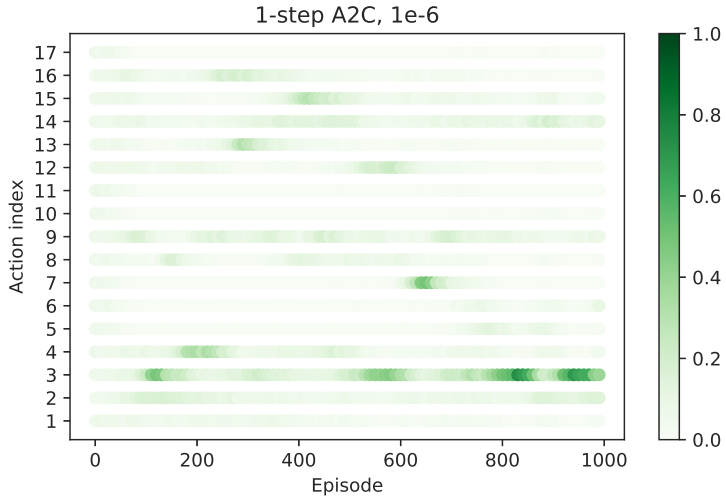

Figure C.1: Relative action frequencies across episodes, one component actions case.

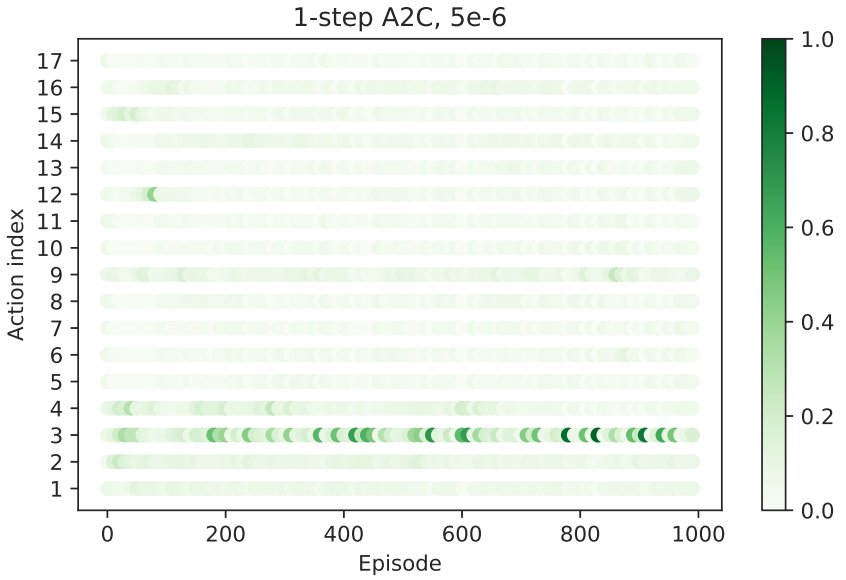

Figure C.2: Relative action frequencies across episodes, one component actions case.

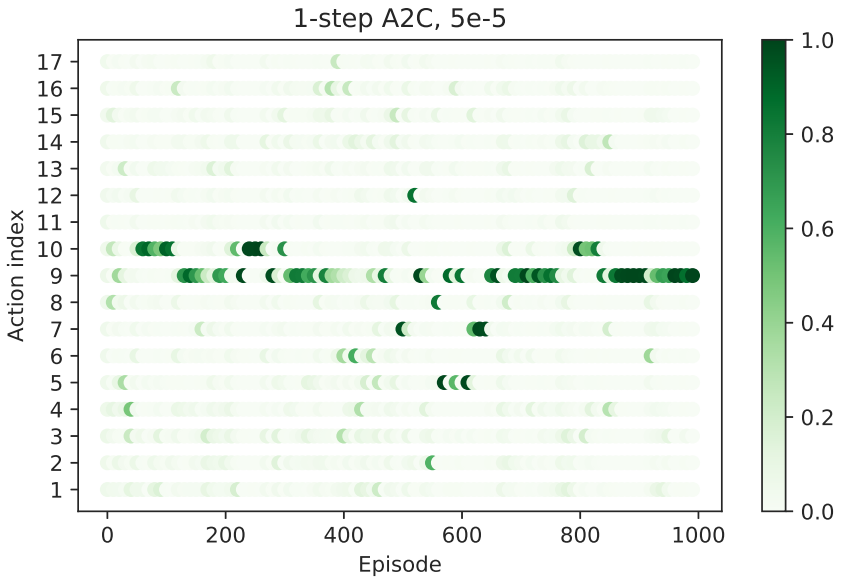

Figure C.3: Relative action frequencies across episodes, one component actions case.

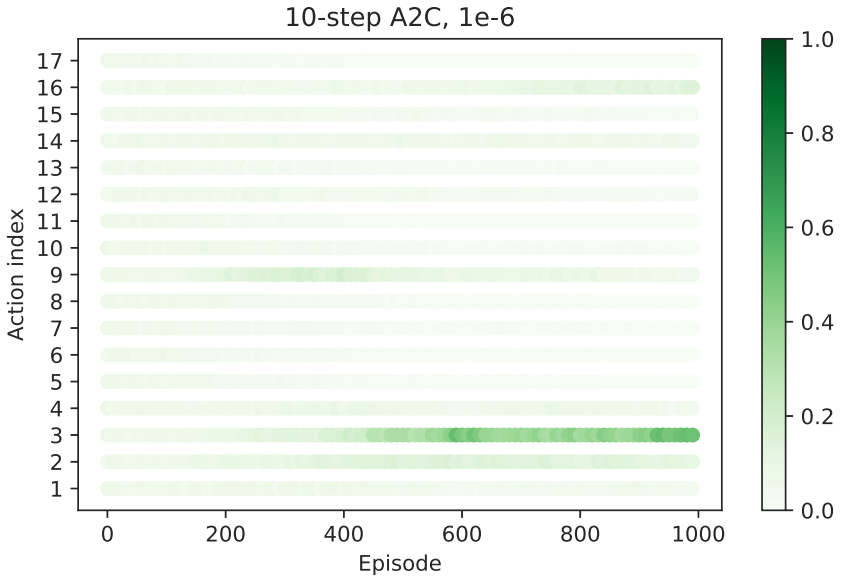

Figure C.4: Relative action frequencies across episodes, one component actions case.

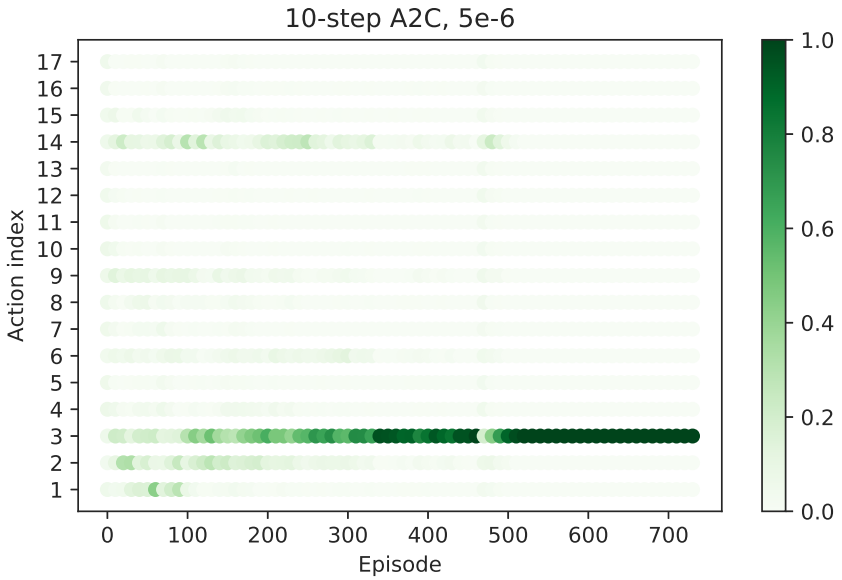

Figure C.5: Relative action frequencies across episodes, one component actions case.

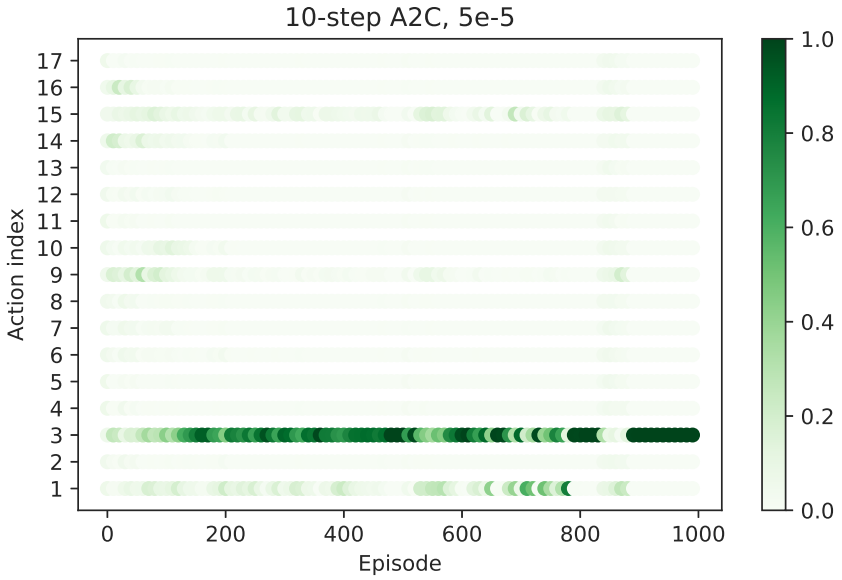

Figure C.6: Relative action frequencies across episodes, one component actions case.

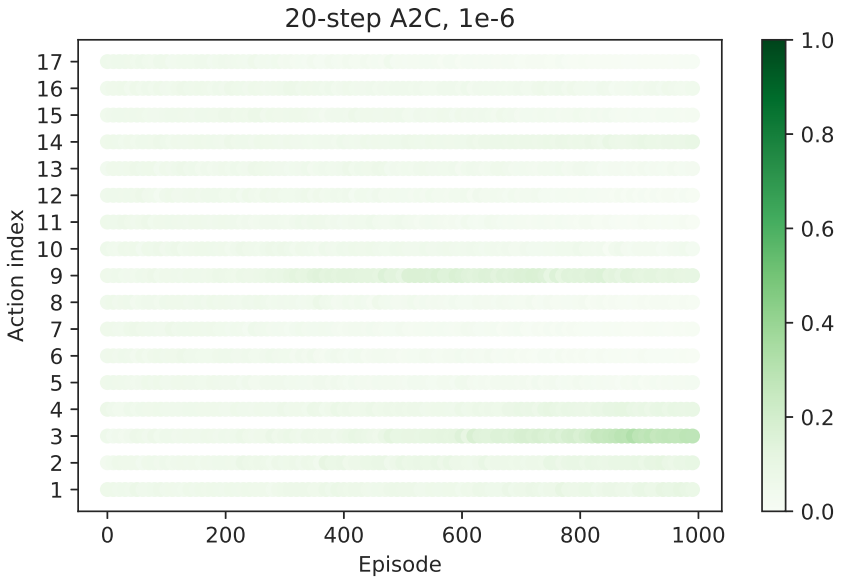

Figure C.7: Relative action frequencies across episodes, one component actions case.

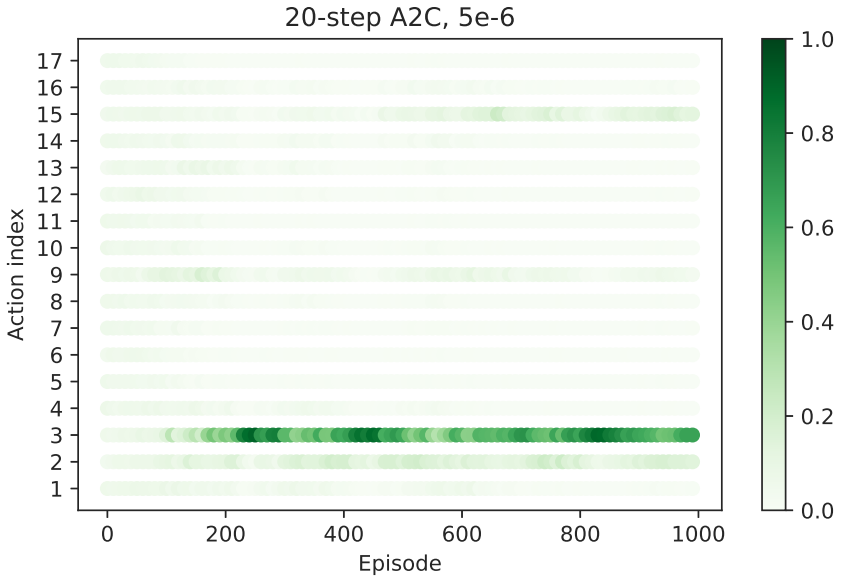

Figure C.8: Relative action frequencies across episodes, one component actions case.

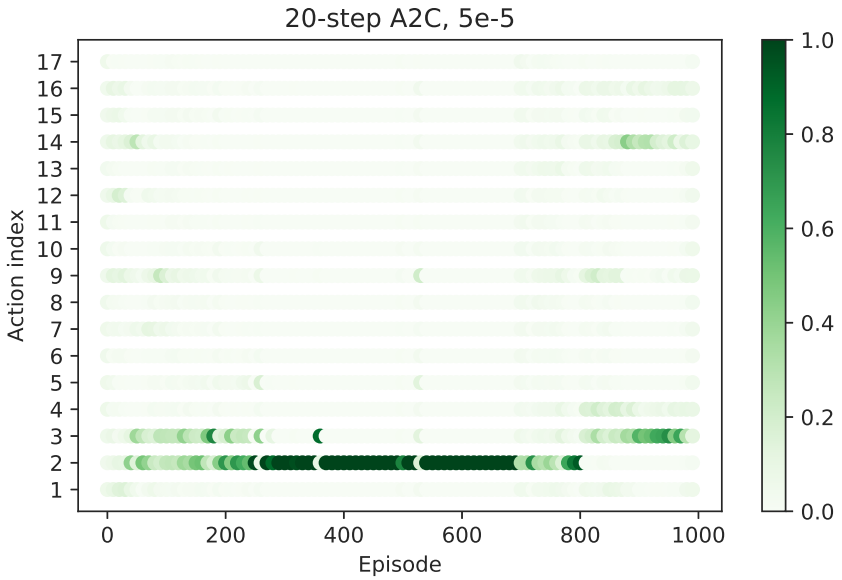

Figure C.9: Relative action frequencies across episodes, one component actions case.

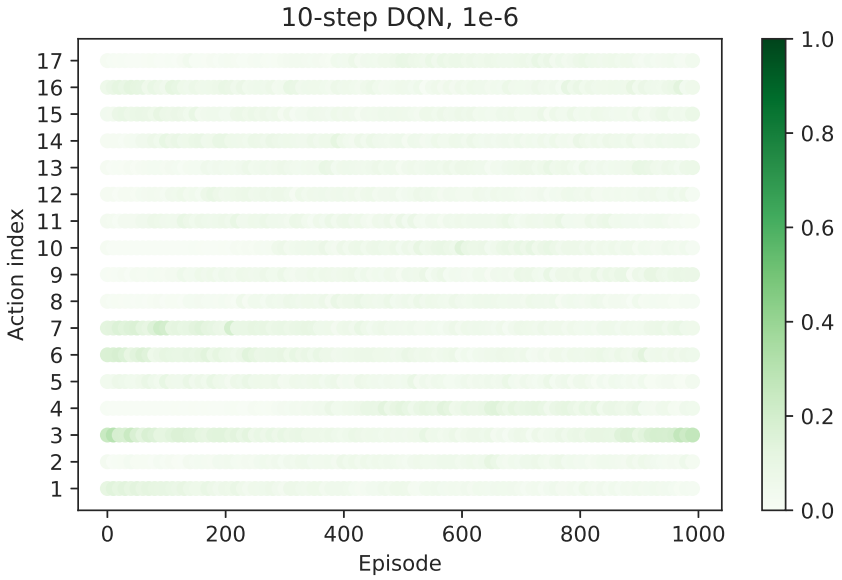

Figure C.10: Relative action frequencies across episodes, one component actions case.

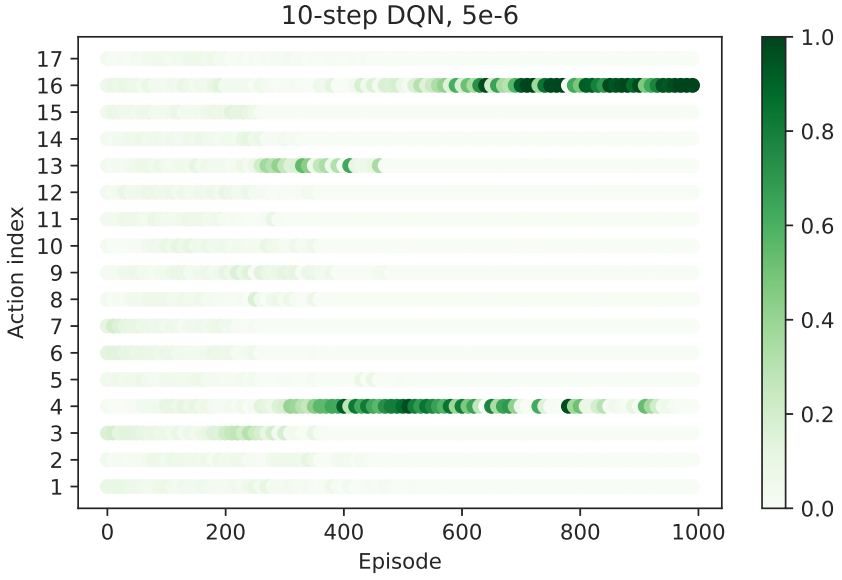

Figure C.11: Relative action frequencies across episodes, one component actions case.

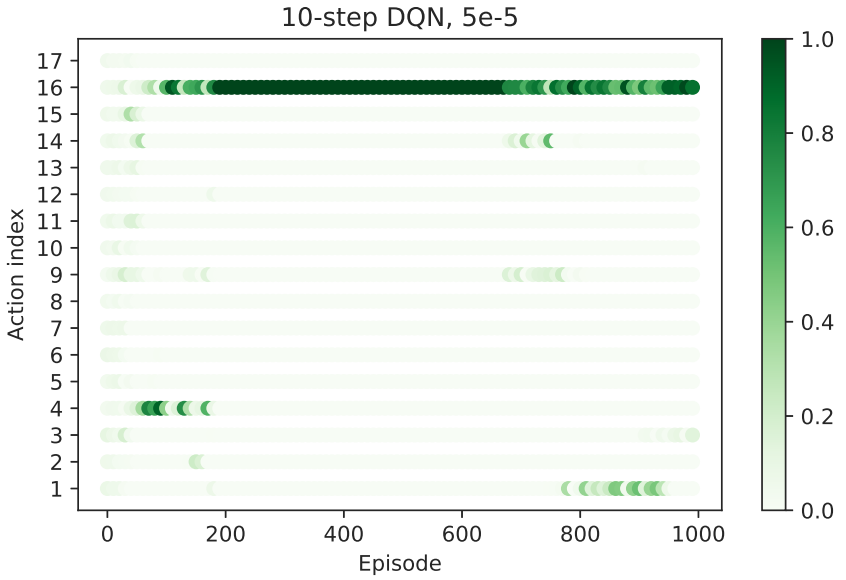

Figure C.12: Relative action frequencies across episodes, one component actions case.

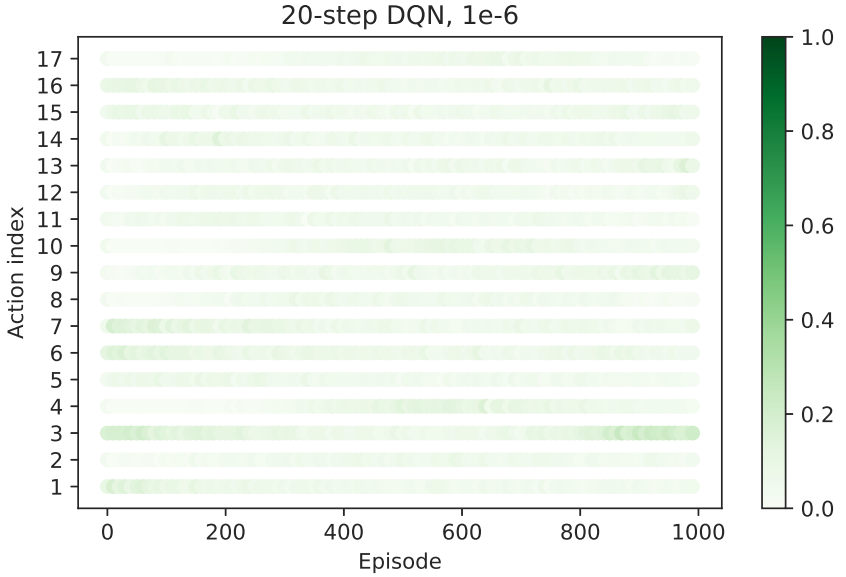

Figure C.13: Relative action frequencies across episodes, one component actions case.

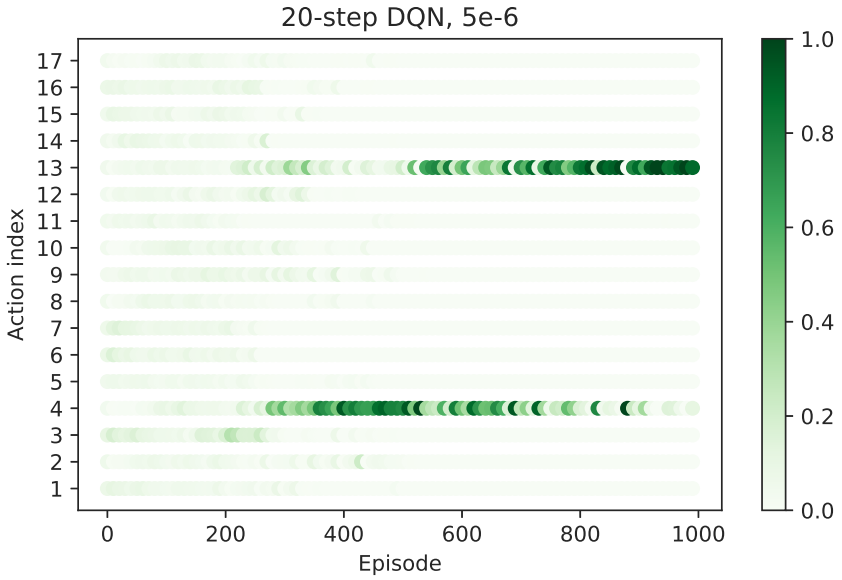

Figure C.14: Relative action frequencies across episodes, one component actions case.

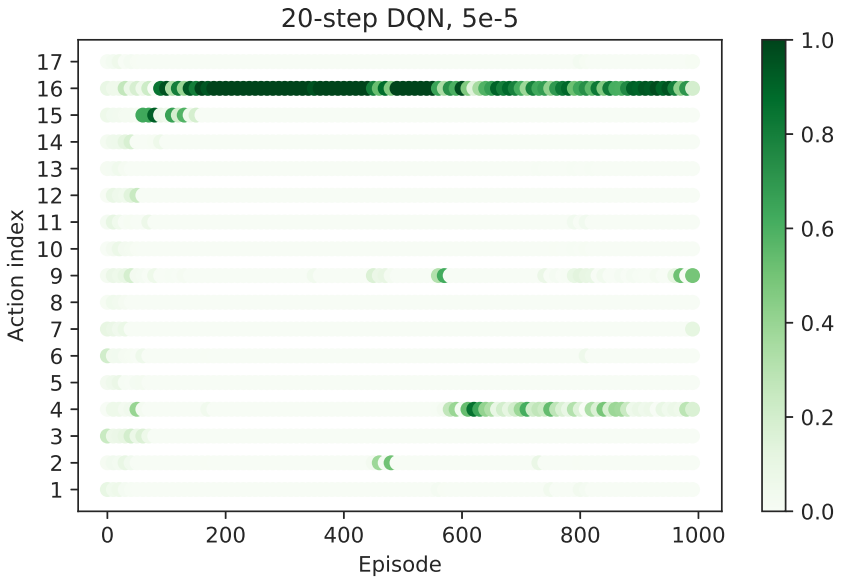

Figure C.15: Relative action frequencies across episodes, one component actions case.

C.1.2 Two component actions

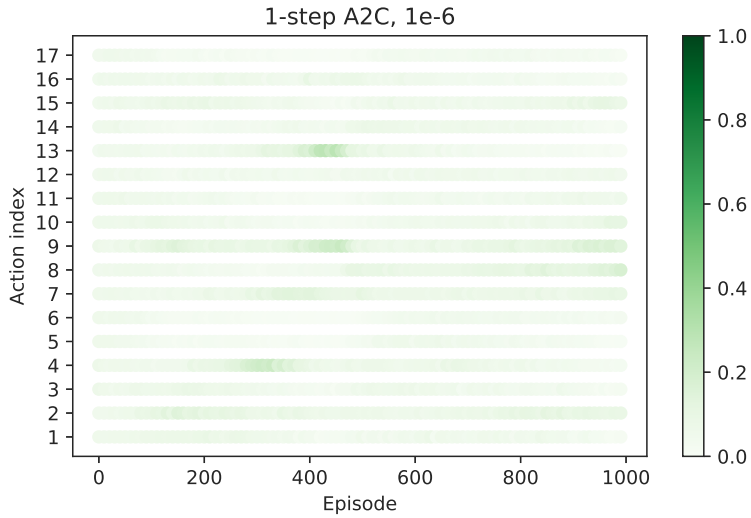

Figure C.16: Relative action frequencies across episodes, two component actions case.

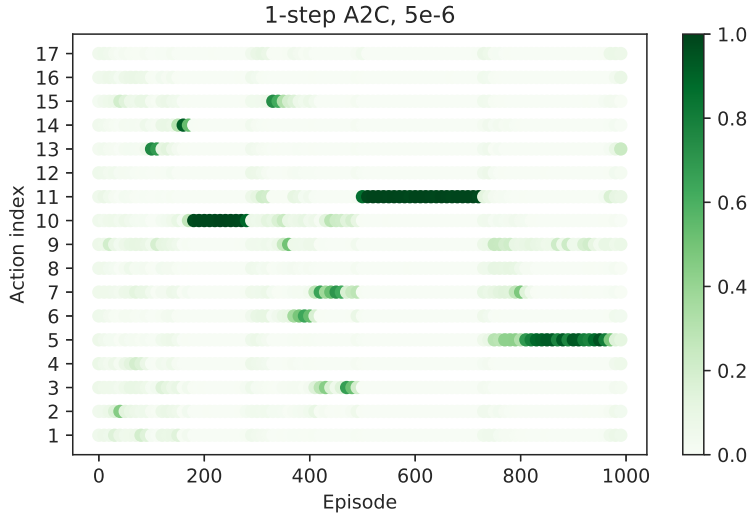

Figure C.17: Relative action frequencies across episodes, two component actions case.

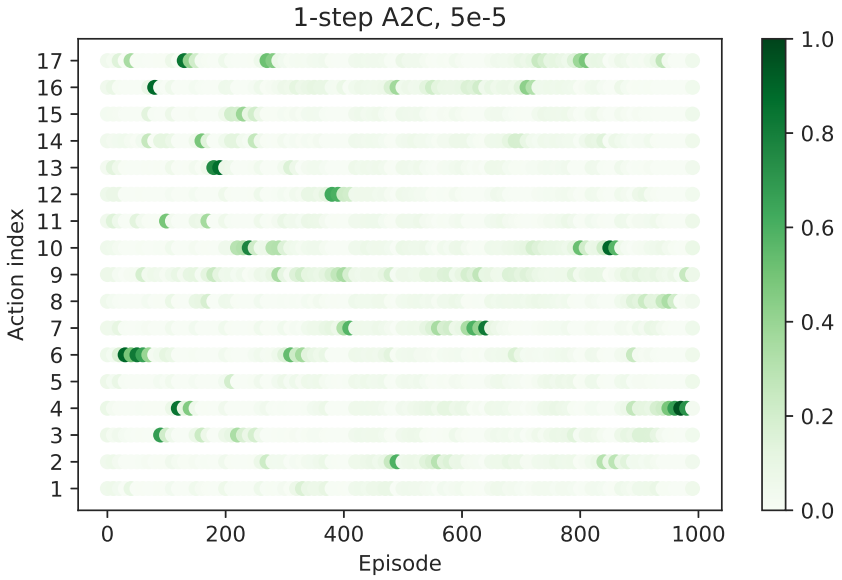

Figure C.18: Relative action frequencies across episodes, two component actions case.

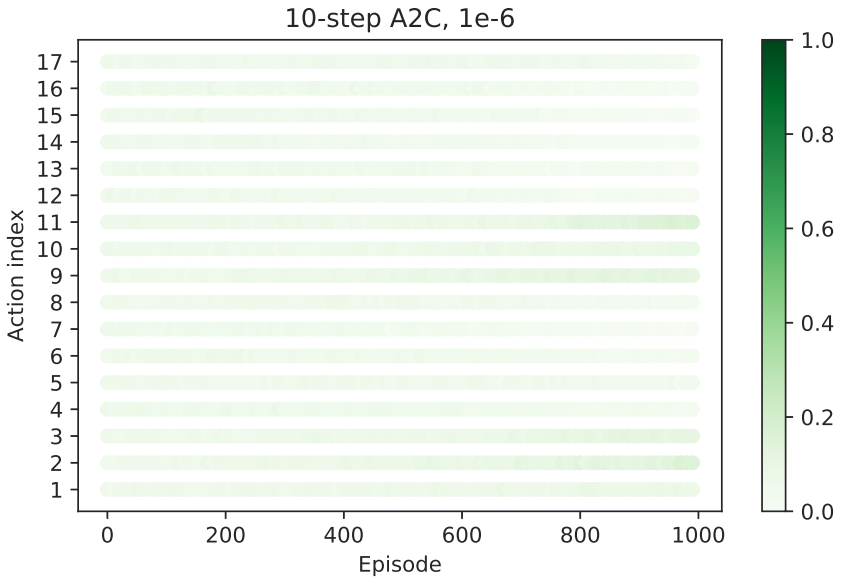

Figure C.19: Relative action frequencies across episodes, two component actions case.

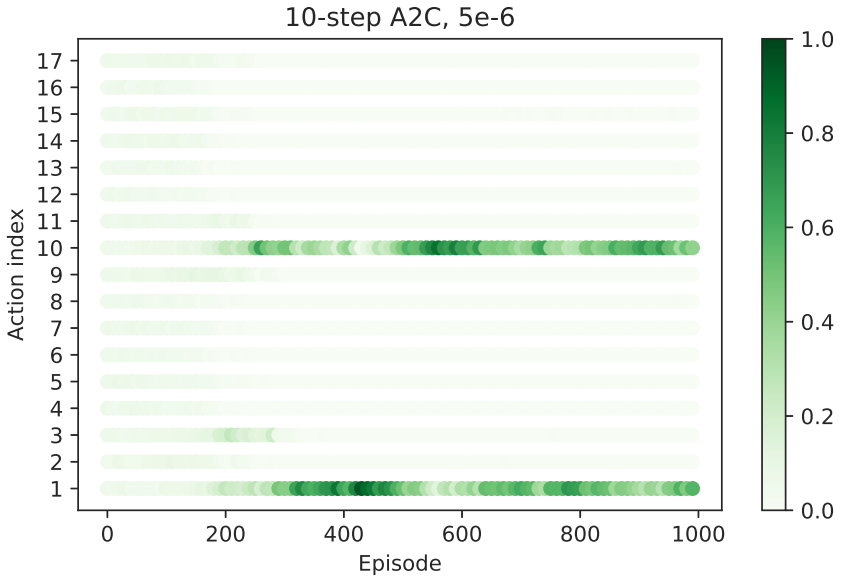

Figure C.20: Relative action frequencies across episodes, two component actions case.

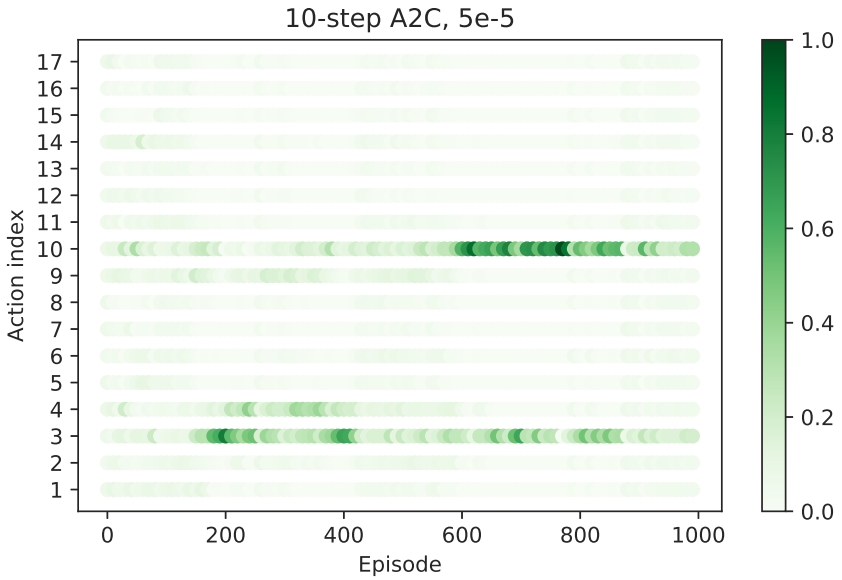

Figure C.21: Relative action frequencies across episodes, two component actions case.

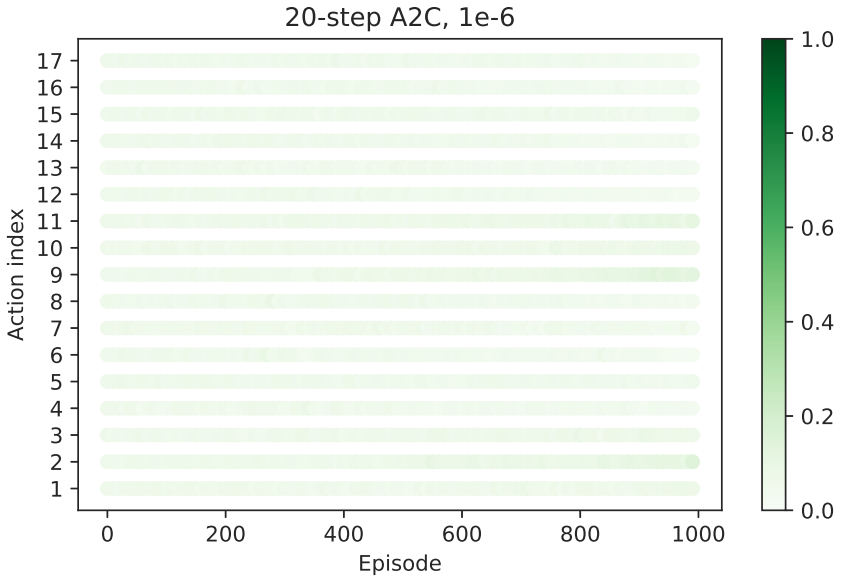

Figure C.22: Relative action frequencies across episodes, two component actions case.

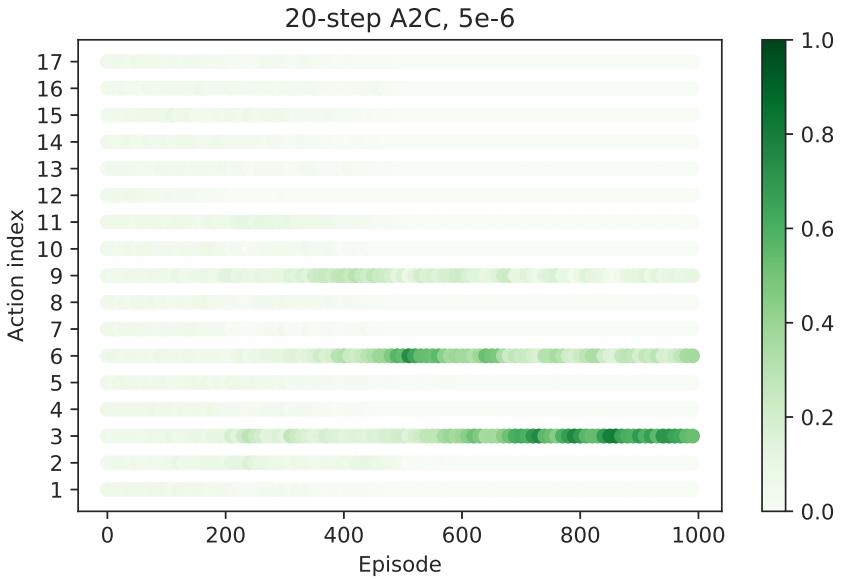

Figure C.23: Relative action frequencies across episodes, two component actions case.

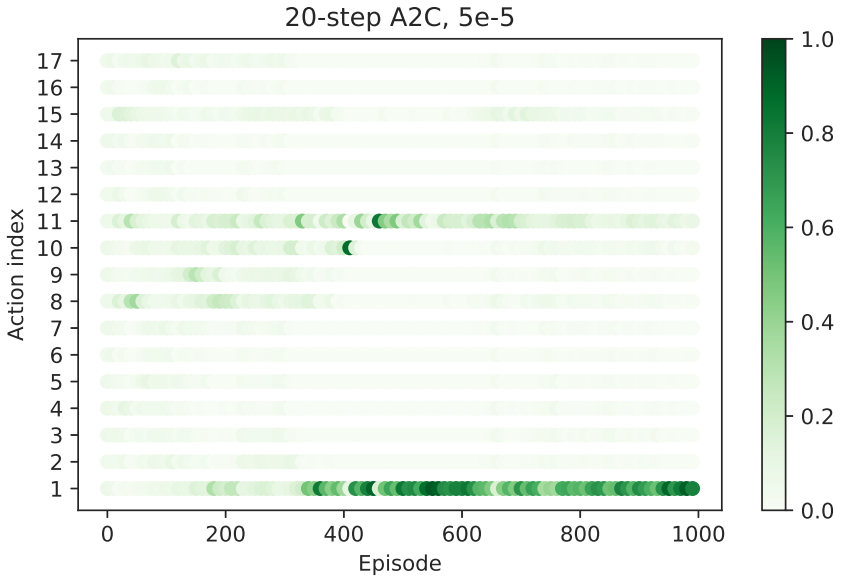

Figure C.24: Relative action frequencies across episodes, two component actions case.

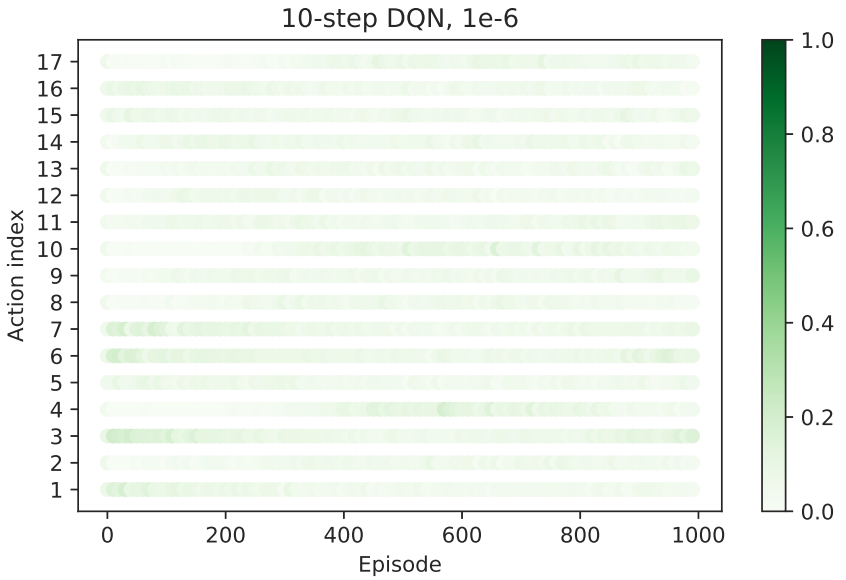

Figure C.25: Relative action frequencies across episodes, two component actions case.

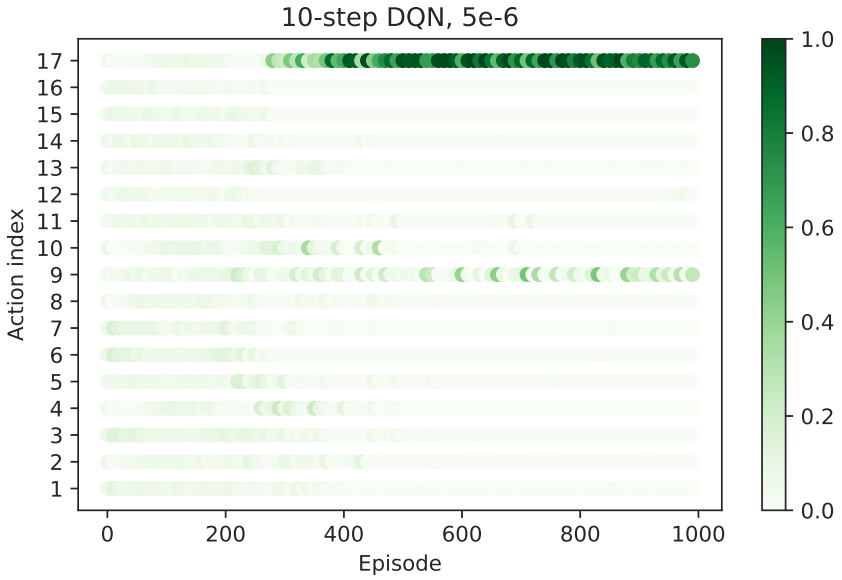

Figure C.26: Relative action frequencies across episodes, two component actions case.

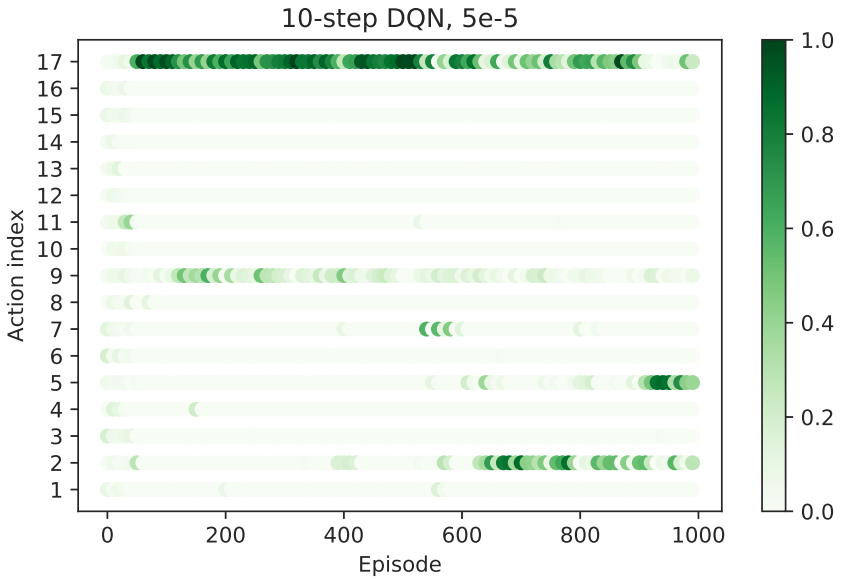

Figure C.27: Relative action frequencies across episodes, two component actions case.

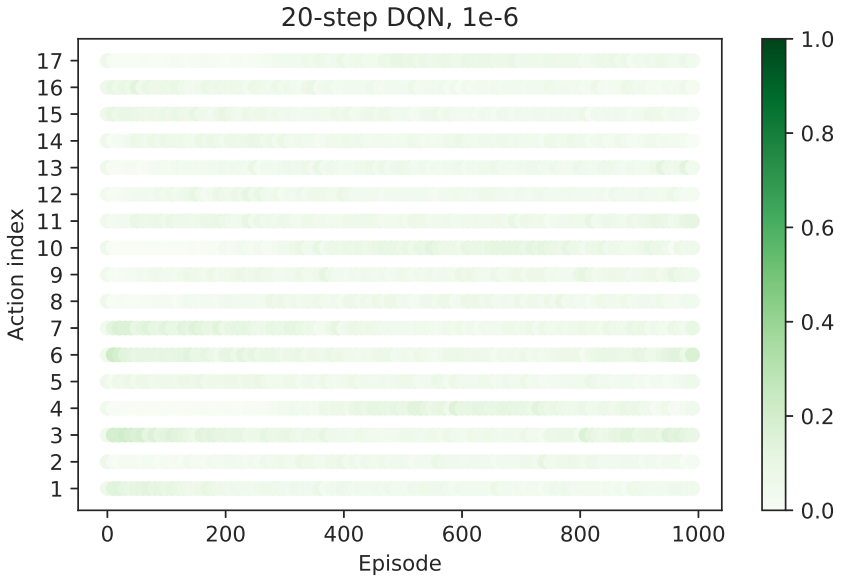

Figure C.28: Relative action frequencies across episodes, two component actions case.

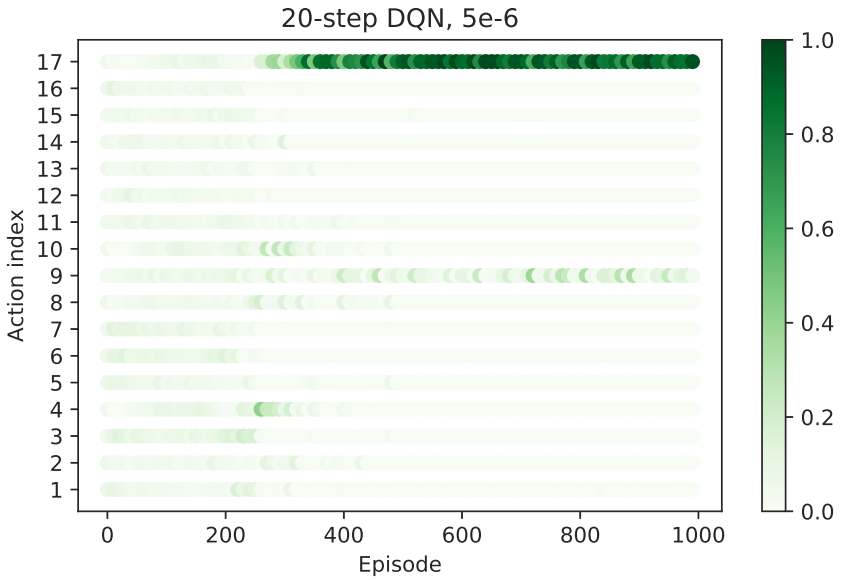

Figure C.29: Relative action frequencies across episodes, two component actions case.

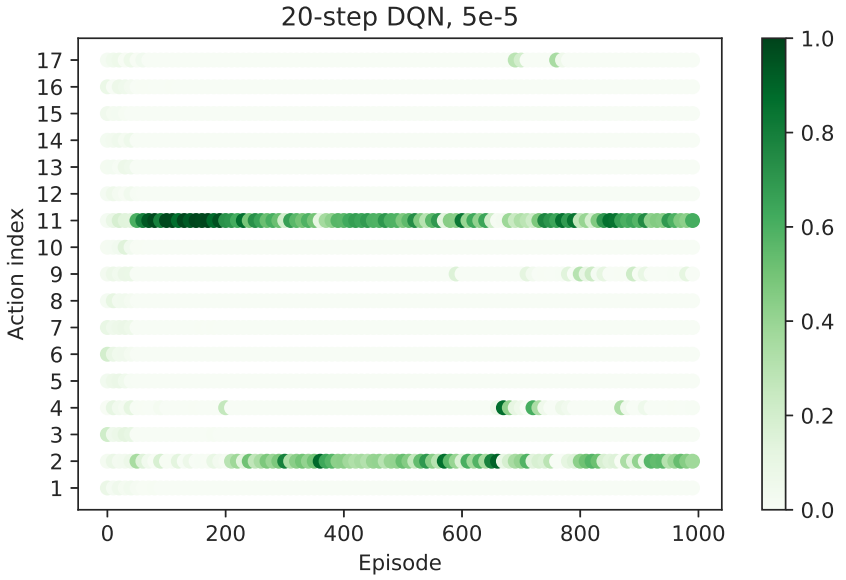

Figure C.30: Relative action frequencies across episodes, two component actions case.

C.2 Analysis of training

Here, we present the plots for accumulated reward as training progresses for each hyperparameter setting we tested in both the one and two component actions case. The blue dotted line represents an estimate of the average reward accumulated by a random policy.

For the 1-step A2C, we see the results in Figure C.31. For a learning rate of 1e-6 there is still a lot of randomness in the policy, but the agent has learned to favor action 3, and it produces quite decent rewards, although it seems as if the evolution comes to a halt quite quickly. The same can be said in the 5e-6 case. In the 5e-5 case, the agent gets stuck in a local maximum where it keeps resetting to the hitherto best solution and does not manage to get out of it within the 1000 episodes.

In Figure C.32 we see the results for 10-step A2C. The plot for the learning rate equal to 1e-6 looks quite nice, the agent seems to be learning continually. The same can be said for 5e-6, although it looks as if it reaches a local maximum about half way through the training phase. For 5e-5, the performance detoriates as time progresses.

In Figure C.33 we see the results for 20-step A2C. Here, it seems as if a learning rate of 1e-6 is a bit too low, whilst 5e-6 provides a nice learning curve. We also see, that for a learning rate of 5e-5, the agent gets stuck in a

local minimum of doing only local search for surprisingly long, but manages to claw its way out.

In Figure C.34 we see the results for 10-step DQN. The start is not as nice as in the A2C case, as the policy is ϵ-greedy, and thus if the random weights sligtly favor one action, then that one is taken a lot in the beginning. A learning rate of 1e-6 seems to be too low, but both 5e-5 and 5e-6 seems alright. For the 20-step DQN case we reach more or less the same conclusions.

C.2.1 One component actions

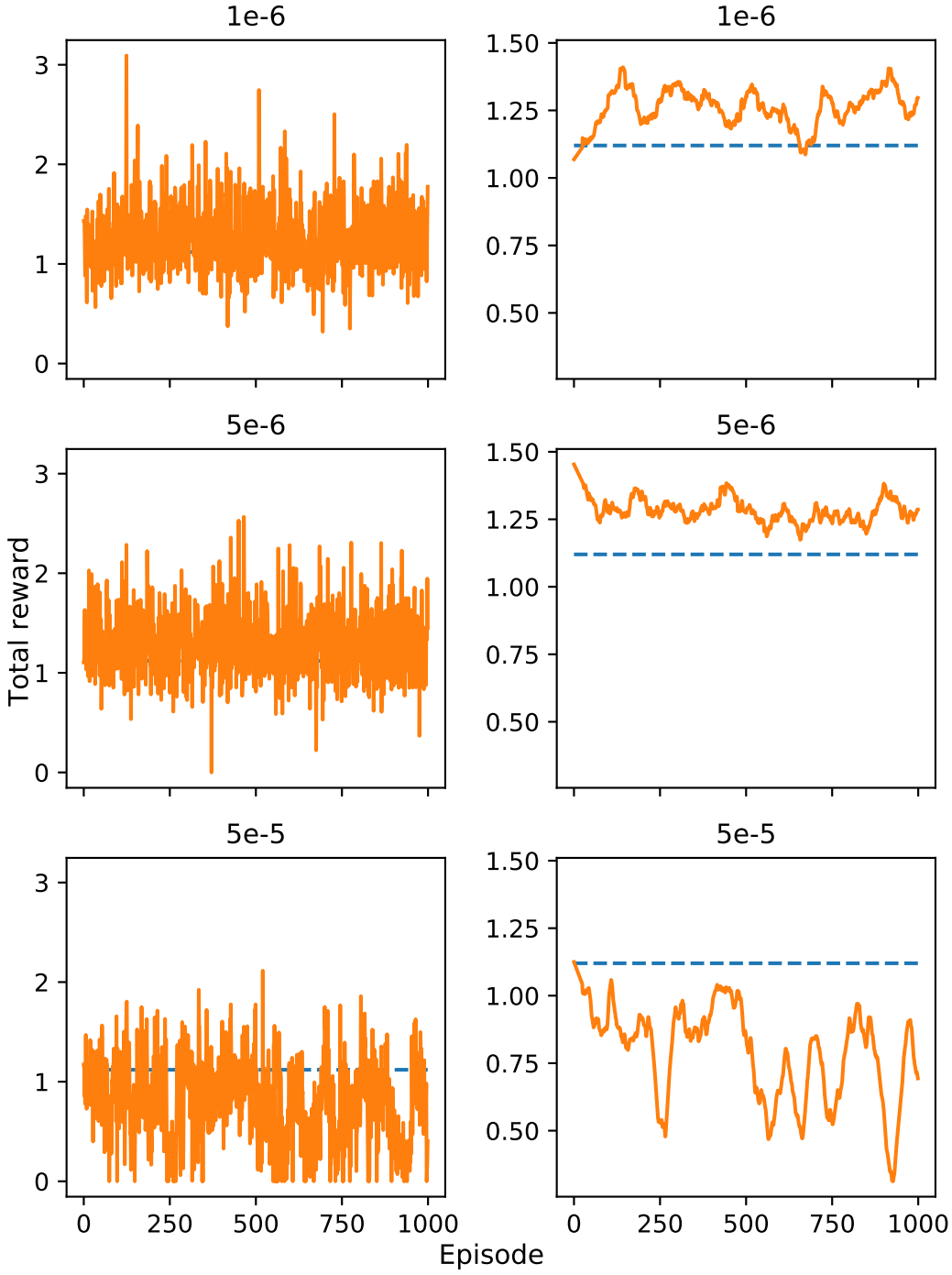

Figure C.31: 1-step A2C, one component actions. Reward accumulated for each episode of training for different learning rates and corresponding moving average.

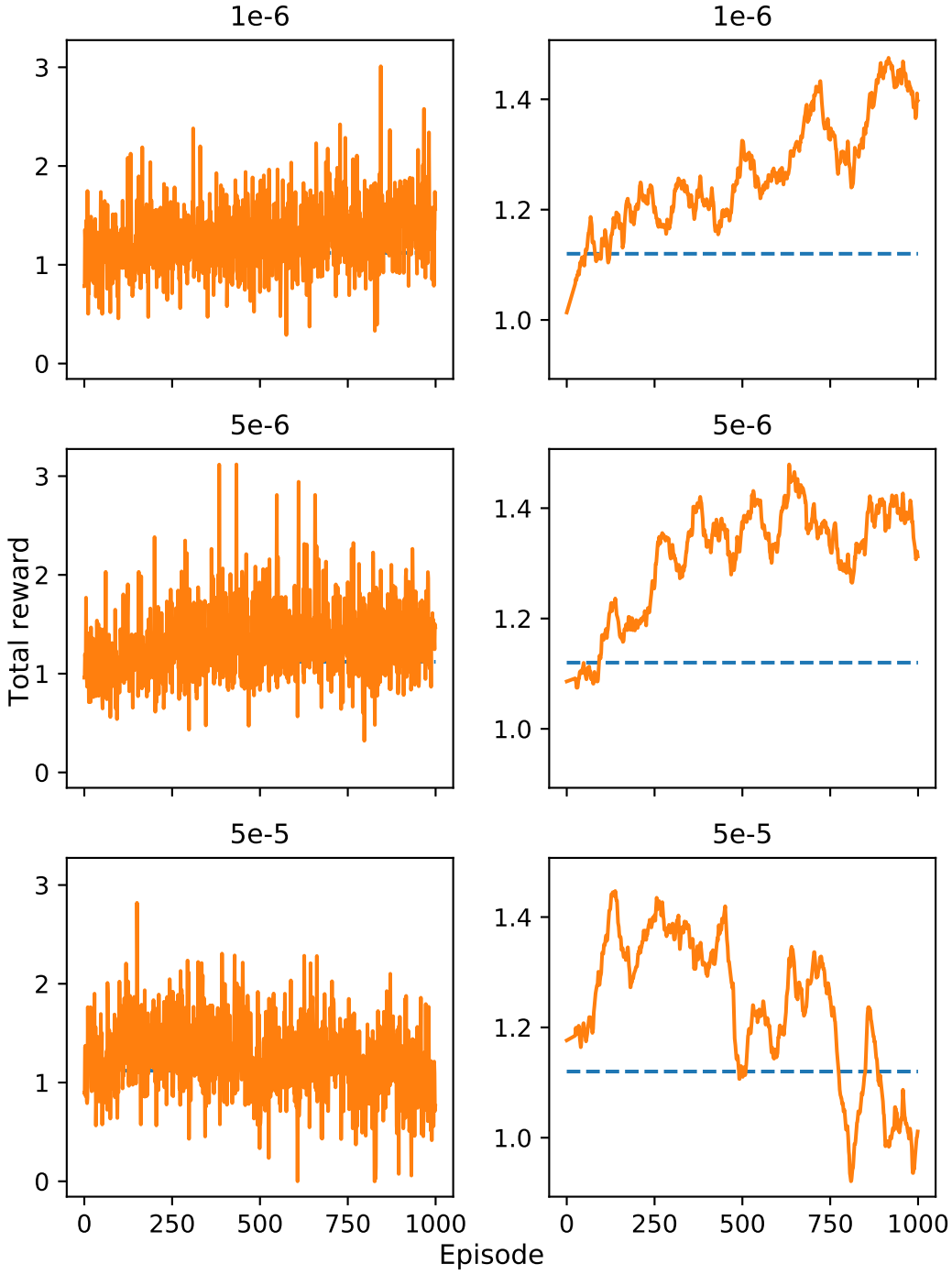

Figure C.32: 10-step A2C, one component actions. Reward accumulated for each episode of training for different learning rates and corresponding moving average.

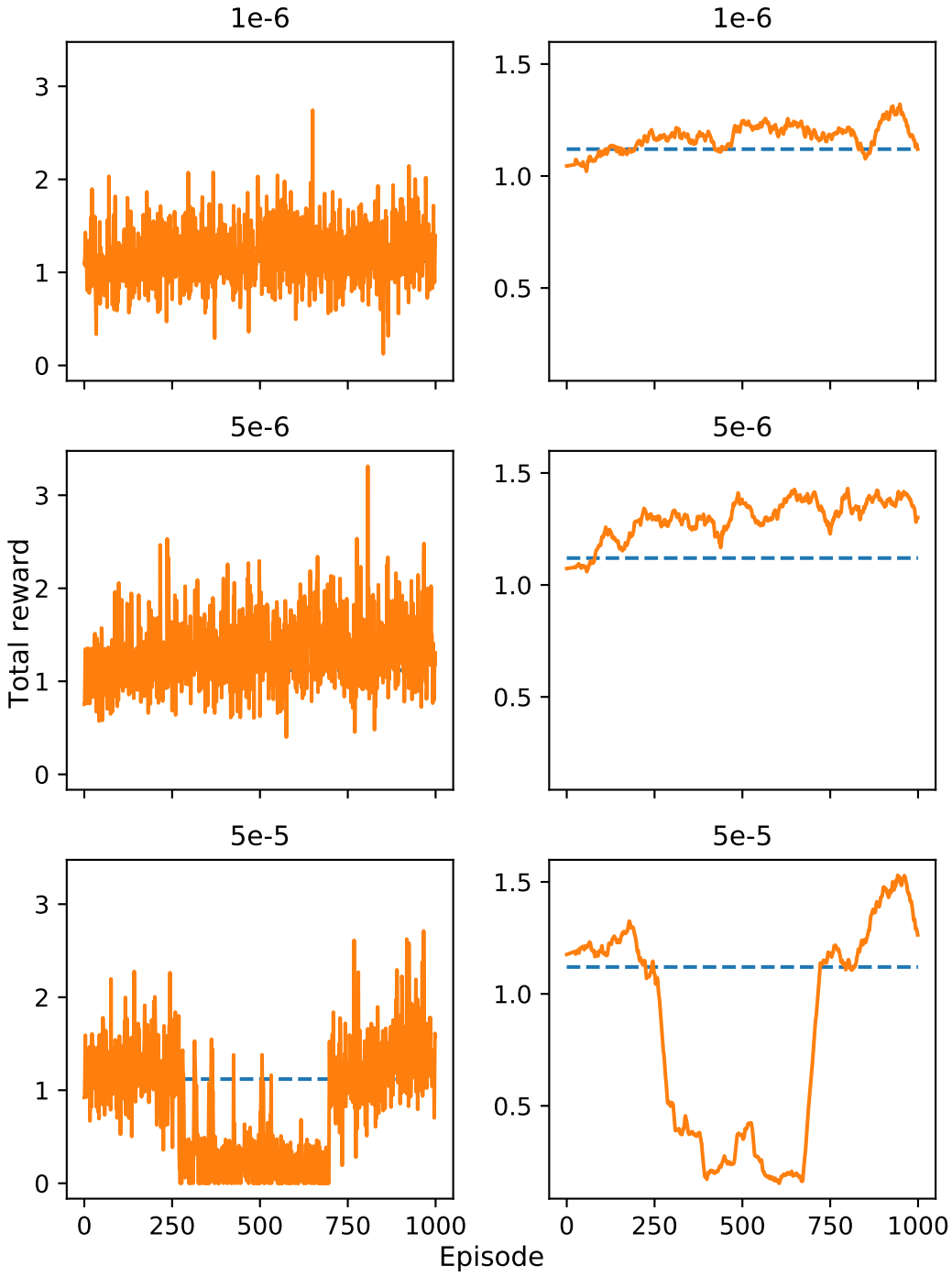

Figure C.33: 20-step A2C, one component actions. Reward accumulated for each episode of training for different learning rates and corresponding moving average.

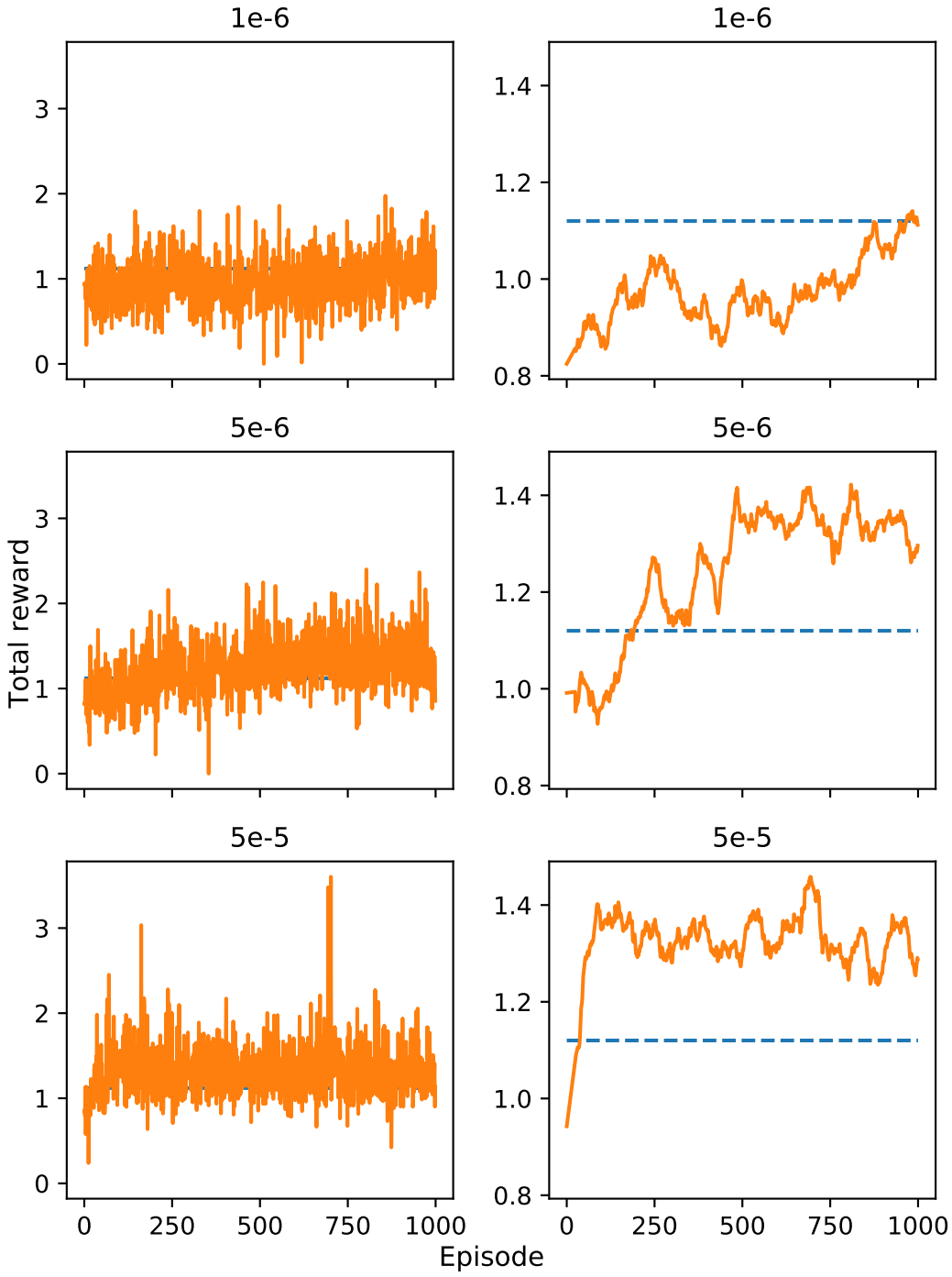

Figure C.34: 10-step DQN, one component actions. Reward accumulated for each episode of training for different learning rates and corresponding moving average.

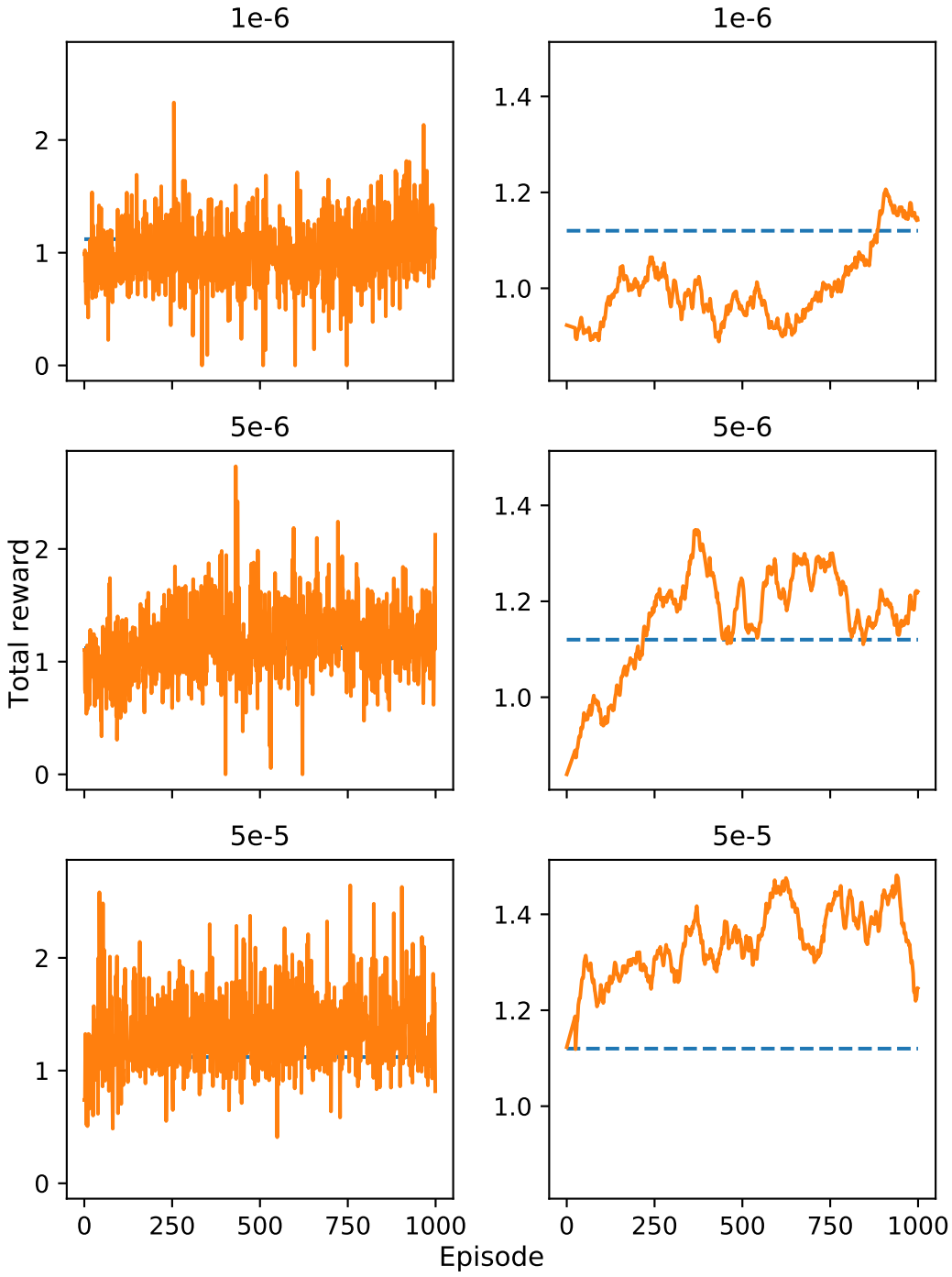

Figure C.35: 20-step DQN, one component actions. Reward accumulated for each episode of training for different learning rates and corresponding moving average.

C.2.2 Two component actions

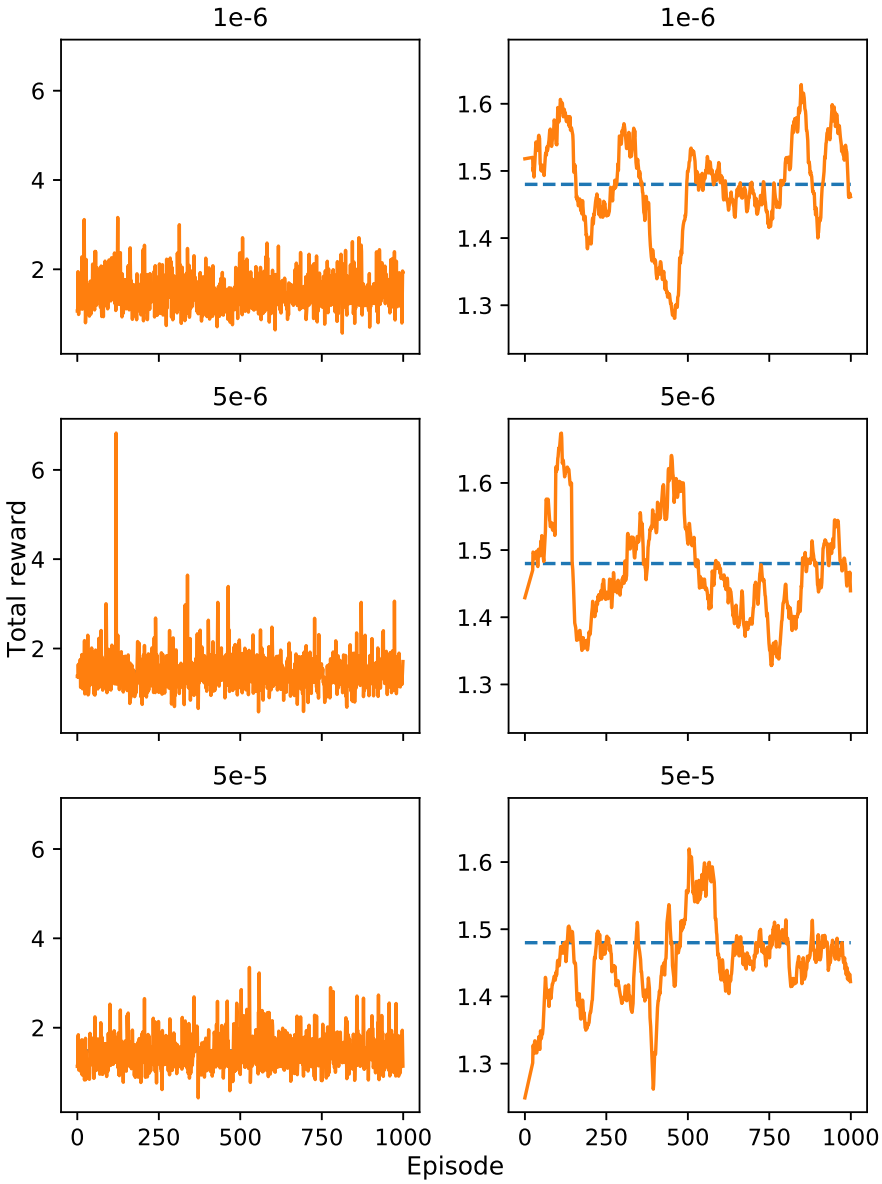

Figure C.36: 1-step A2C, two component action actions. Reward accumulated for each episode of training for different learning rates and corresponding moving average.

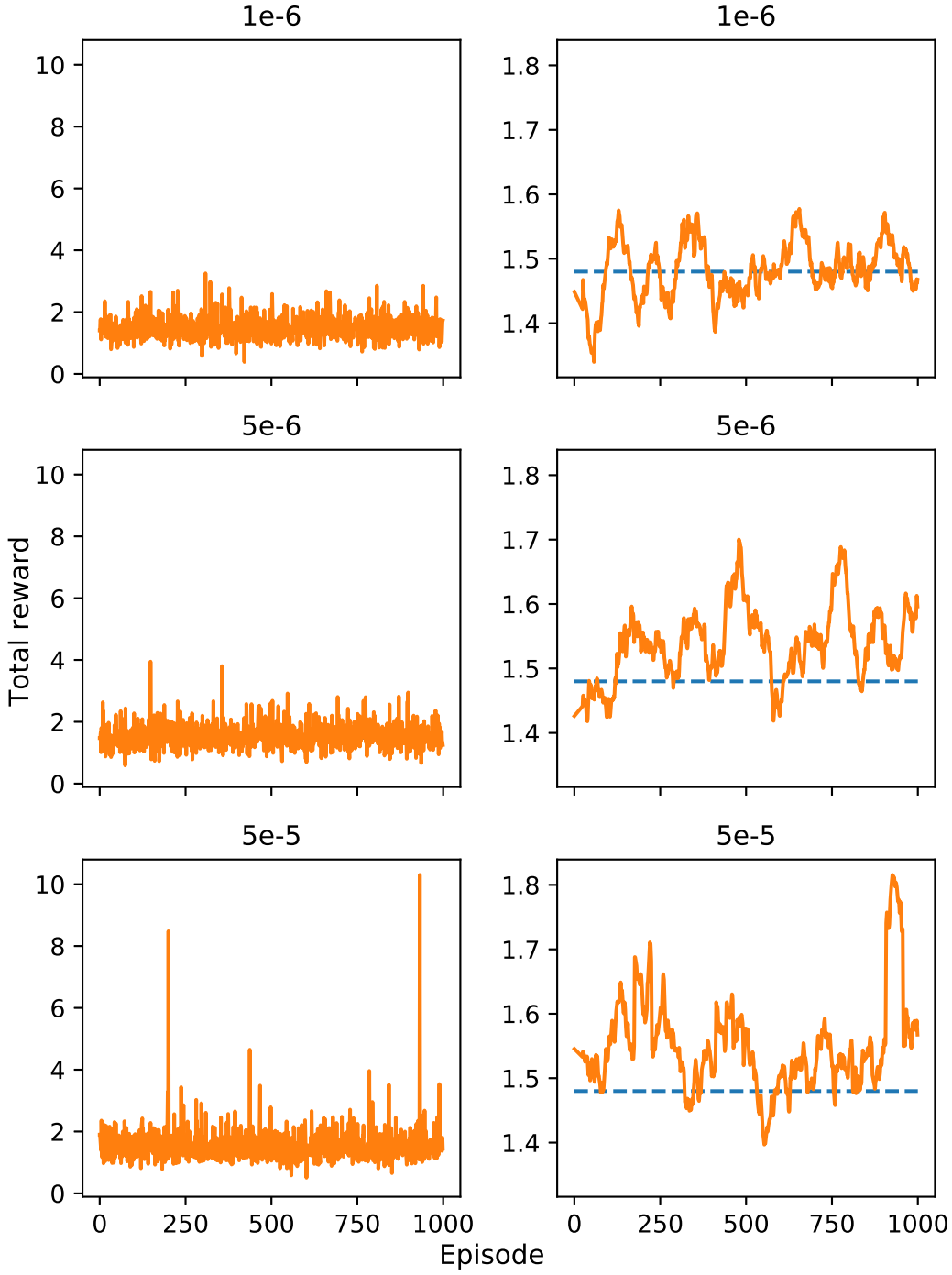

Figure C.37: 10-step A2C, two component action actions. Reward accumulated for each episode of training for different learning rates and corresponding moving average.

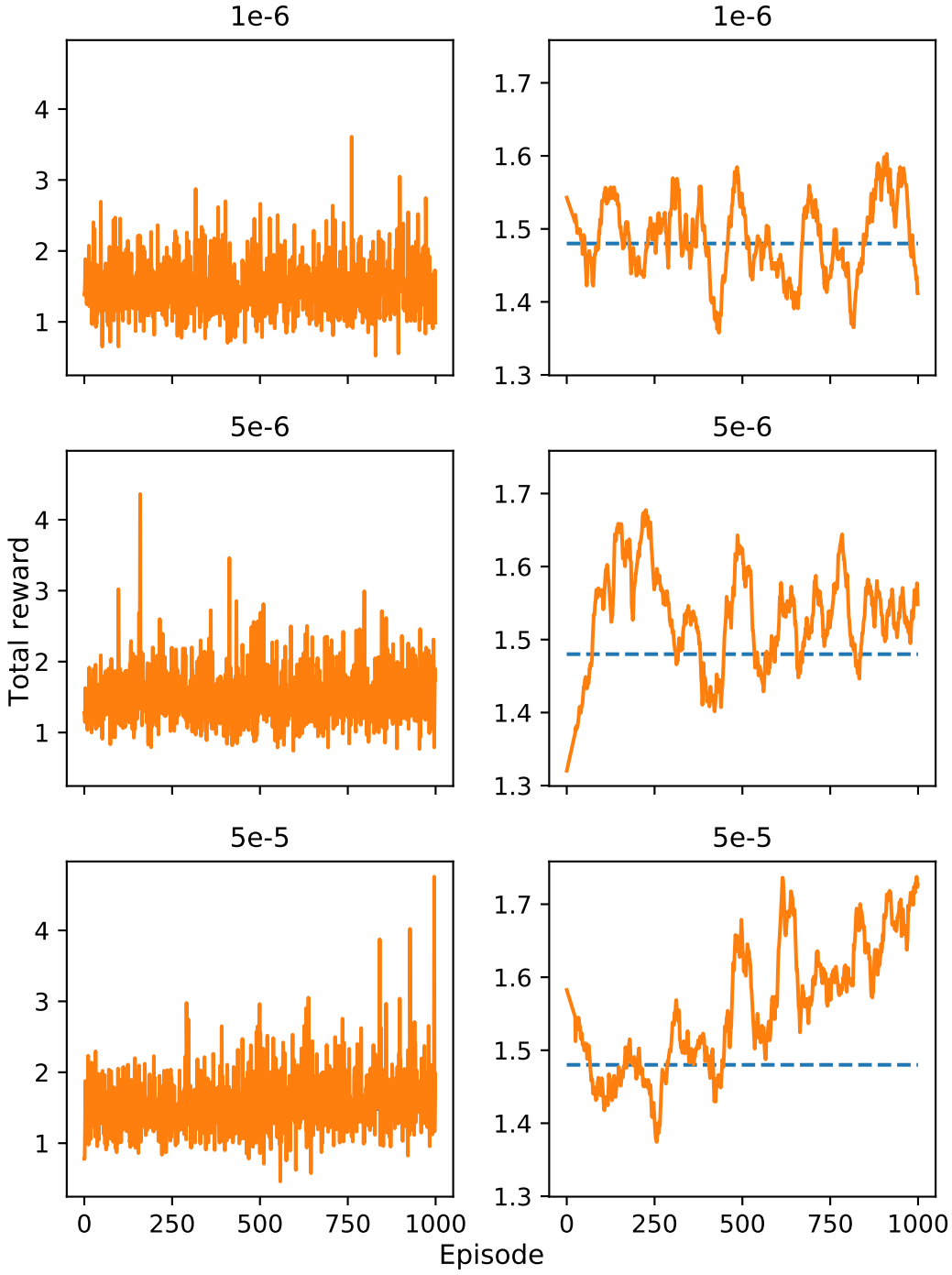

Figure C.38: 20-step A2C, two component action actions. Reward accumulated for each episode of training for different learning rates and corresponding moving average.

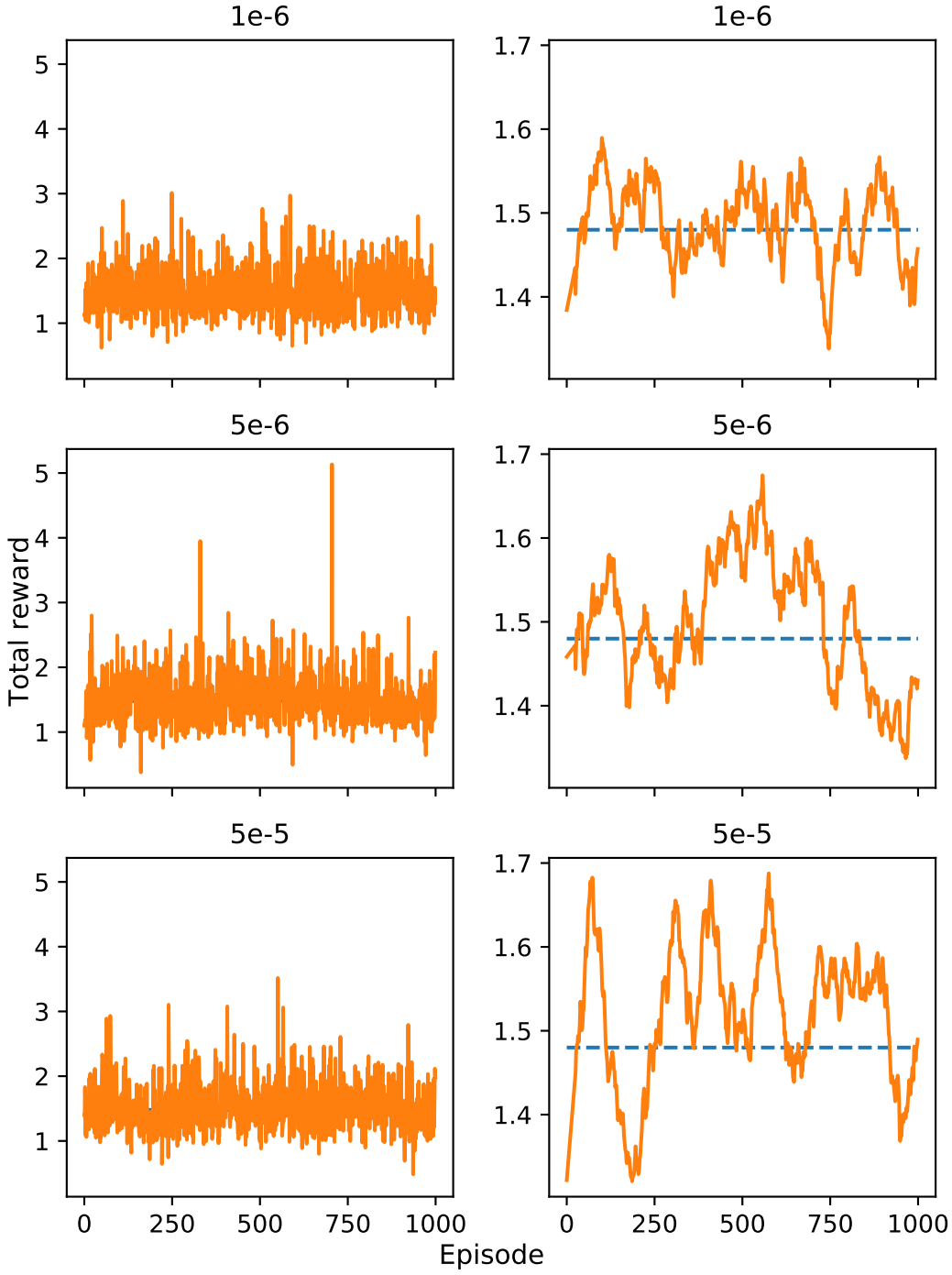

Figure C.39: 10-step DQN, two component action actions. Reward accumulated for each episode of training for different learning rates and corresponding moving average.

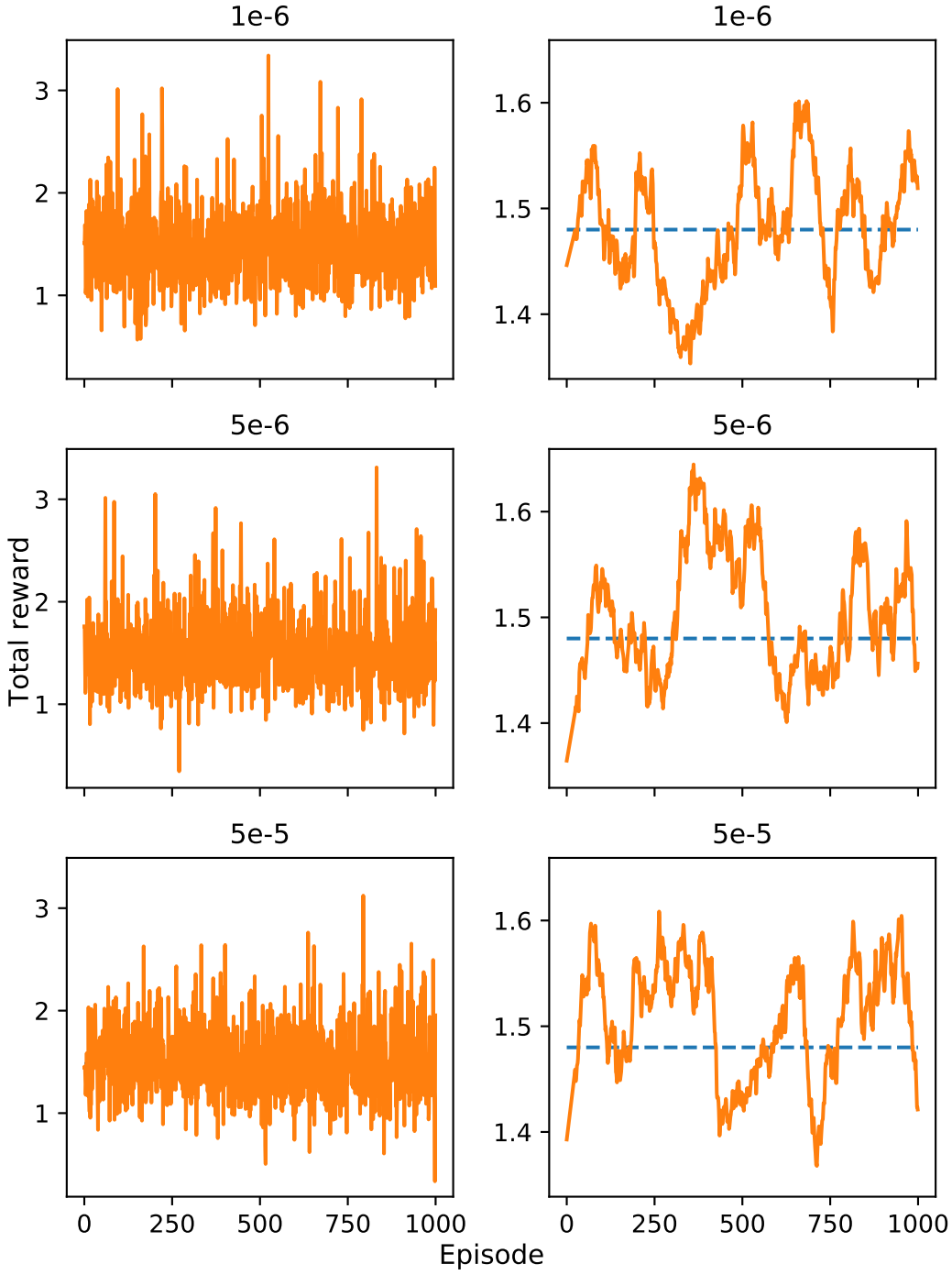

Figure C.40: 20-step DQN, two component actions. Reward accumulated for each episode of training for different learning rates and corresponding moving average.