Faculdade de Engenharia da Universidade do Porto

Mestrado de Engenharia da Informação

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

REINFORCEMENT LEARNING FOR PRIMARY CARE APPOINTMENT SCHEDULING

Tiago Salgado de Magalhães Taveira-Gomes

Supervisor

Jaime S Cardoso

Porto 25th September 2017

REINFORCEMENT LEARNING FOR PRIMARY CARE APPOINTMENT SCHEDULING

Tiago Salgado de Magalhães Taveira-Gomes

Mestrado em Engenharia da Informação

Approved by

____

President

____

Referee

____

Porto, 25th September 2017

Resumo

O agendamento de consultas é uma tarefa complexa, que desempenha um papel fundamental na qualidade dos serviços de saúde. O agendamento eficiente evita a insatisfação dos médicos e dos pacientes, sendo assim um determinante relevante da qualidade dos cuidados prestados. Ainda assim, essa tarefa é frequentemente realizada *ad hoc*, sem consideração por aspetos que determinam o tempo de consulta e eventuais tempos de espera. Uma vez que neste contexto o agendamento é realizado para cada doente à vez, o problema não é facilmente resolvido através de métodos de otimização. Acresce ainda que a natureza de evolução constante dos Cuidados de Saúde Primários implica que heurísticas gerais não alcancem bons resultados, tornando o problema de agendamento automático custoso, e de um modo geral inviável.

O presente trabalho pretende desenvolver uma plataforma para a criação de ambientes de agendamento de consultas totalmente configuráveis, e estudar a performance do algoritmo de aprendizagem por reforço *Advantage Actor-Critic* sobre um conjunto de ambientes de agendamento de dificuldade crescente, comparando a sua performance com heurísticas determinísticas simples.

Para esse efeito foi desenvolvida uma plataforma baseada em micro serviços, que permite a criação e teste de quaisquer ambientes de agendamento, sobre os quais podem ser desenvolvidos e testados agentes. A plataforma permite o acompanhamento em tempo real das ações dos agentes e da medição da sua performance, bem como a colheita de métricas de performance para posterior análise. Permite ainda a interação de agentes humanos. A plataforma foi concebida para correr localmente ou na nuvem.

Foram desenvolvidos sete ambientes que implementam os requisitos básicos do agendamento nos Cuidados de Saúde Primários e foi estudada a performance do algoritmo referido. Foi demonstrado que este algoritmo é capaz de aprender as regras que governam os ambientes de agendamento e cuja manipulação foi necessária para atingir os resultados ótimos ou *quási* ótimos observados, tornando-o assim um candidato promissor para utilização em ambiente de produção.

Abstract

Patient scheduling is a complex task that plays a crucial role in the quality of care. Effective scheduling avoids patient and physician dissatisfaction, and as such, is an important determinant of care. However, the task of patient scheduling is traditionally done *ad hoc*, with disregard of potential aspects that may determine appointment duration and overall waiting time. Since scheduling in this setting is frequently done one patient at a time, the problem cannot be readily solved by optimization methods. Furthermore, the constant changing nature of clinical settings implies that *one-fits-all* heuristic scheduling approaches would generally be poor, rendering the task of automated patient scheduling resource intensive, costly to maintain, and thus, generally unfeasible.

The present work aims to develop a framework to create fully configurable patient scheduling environments, study the performance of the Advantage Actor-Critic reinforcement learning algorithm to schedule appointments in a set of increasingly challenging environments, and benchmark it against a set of simple deterministic heuristics.

A micro-service application was developed to create and test any scheduling environments over which to develop and test agents. The platform enabled online monitoring of the agent performance and data collection for posterior analysis. It also enabled humans to work as agents. The platform was developed in such a way that it can be run in local machines or cloud infrastructure.

We devised and developed a set of seven specific appointment scheduling environments that implement the basic requirements of scheduling in Primary Care, and studied the performance of RL agents implemented using the A3C algorithm.

Finally, we have shown that RL agents are able to learn the underlying rules that govern such environments, that needed to be considered to arrive at optimal or near optimal scheduling solutions, and thus are promising algorithms for production environments.

Foreword

Teaching and learning are critical subjects, that undermine the success of civilizations. Innovative teaching and learning methods have led individuals and institutions to build the world we live in today by overcoming ever greater challenges.

In the digital era, where the massive volumes of information largely surpass the analysis capability of the human intellect, teaching and learning have gone beyond methods aimed at people, and now also seek to create intelligent learning machines, able to extract patterns, and reason from large amounts of data.

Having conducted research on strategies and factors influencing effective teaching and learning in the healthcare setting, it seemed particularly interesting to transition to the creation of machines that learn, in hopes that many redundant, cumbersome and error-prone tasks that are frequently performed by healthcare professionals can be mastered by such systems, freeing individuals to invest their time to deepen skills on tasks that humans should preferably perform, such as providing quality healthcare to patients.

Acknowledgements

To Professor Jaime Cardoso,

for the invaluable teachings, support, understanding and inspiration.

To my Parents,

To my Grand Parents,

To my Brother,

To my Dear Love Isabel,

And finally,

to Mother Isabel and Father José Joaquim,

for their support throughout this great endeavor.

Contents

Introduction 1
Overview 1
Motivation 1
Objective 2
Contributions 2
Structure of the Dissertation 2
Patient appointment scheduling 4
Scheduling 4
Scheduling in Healthcare 4
Patient scheduling 5
Appointment structure in Primary Care 7
Scheduling opportunities in Primary Care 9
Scheduling requirements for Primary Care 10
Reinforcement Learning 13
Machine learning overview 13
Deep learning 15
Reinforcement Learning Overview 22
Reinforcement Learning Methods 27
Concluding remarks 36
Scheduling Framework 39
Environment modelling 39
Agent configuration 44
Simulation 46
Software architecture considerations 48
Final remarks 54
Scheduling environments and agent performance 55

Dummy environment A 57
Dummy environment B 59
Dummy environment C 61
Dummy environment D 64
Dummy environment E 67
Dummy environment F 71
Real daily environment 75
Discussion and Future Work 81
References 86

List of Acronyms

A3C – Asynchronous Advantage Actor-Critic

ANN - Artificial Neural Networks

AWS – Amazon Web Services

CreLU – Concatenated ReLU

DNN – Deep Neural Network

DQN – Deep Q Network

ELU – Exponential Linear Unit

FFN – Feed Forward Network

HQL – Hive Query Language

HTTP – Hypertext Transfer Protocol

JDBC – Java Database Connection

LSTM – Long Shor-Term Memory

MLP - Multi-Layer Perceptron

MDP – Markov Decision Process

PCA – Principle Component Analysis

POMDP – Partial Observable Markov Decision Process

ReLU – Rectified Linear Unit

RL – Reinforcement Learning

RNN – Recurrent Neural Network

S3 – Simple Storage Service

SQL – Structured Query Language

TanH – Hyperbolic tangent

TCP – Transmission Control Protocol

TD – Temporal Difference

WS – Web Socket

List of Algorithms

Algorithm 1 – RMSProp algorithm.. 20
Algorithm 2 - Temporal Difference learning using tabular method 30
Algorithm 3 - Temporal Difference learning using function approximation.................... 31
Algorithm 4 – REINFORCE algorithm 32
Algorithm 5 – Actor-Critic algorithm 33
Algorithm 6 - Asynchronous Advantage Actor Critic (A3C) algorithm........................... 36
Algorithm 7 - Scheduling environment reward computation 43
Algorithm 8 - Near optimal solution heuristic 47

List of Equations

Equation 1 – Feed forward neural network. 16
Equation 2 – Rectified linear unit activation function. 17
Equation 3 – Softmax activation function. 17
Equation 4 – Recurrent neural network. 17
Equation 5 – Sigmoid activation function. 18
Equation 6 – Implementation of long short term RNN cell 18
Equation 7 – Cross entropy cost function. 19
Equation 8 – Regularized cost function 20
Equation 9 – L^2 parameter regularization. 21
Equation 10 – L^1 parameter regularization. 21
Equation 11 – Dropout regularization. 22
Equation 12- Dynamic programming value function. 25
Equation 13 - Value iteration equation. 26
Equation 14 - Q function. 26
Equation 15- Bellman optimality equation 26
Equation 16 - Policy iteration equation. 26
Equation 17- Advantage function estimation. 34
Equation 18 - Regularized policy gradient cost function using advantage estimation 35

List of Tables

Table 1 - Dummy environment feature summary.. 56
Table 2 - RL Agent A1, A2 and A3 configuration for dummy environment A and B 57
Table 3 - RL Agent A1, A4 and A5 configuration for dummy environment C 62
Table 4 - RL Agent A5, A6 and A7 configuration for dummy environment D 65
Table 5 - RL Agent A6, A8 and A9 configuration for dummy environment E................ 68
Table 6 - RL Agent A9, A10, A11 and A12 configuration for dummy environment F.... 72
Table 7 - RL Agent A13 and A14 configuration for Real daily environment 77

List of Figures

Figure 1 - Simplified scheduling model of a Primary Care clinic 7
Figure 2 – Example schedule diagram of a typical primary care doctor schedule 9
Figure 3 – Comparison between classical machine learning and deep learning 14
Figure 4 – Machine learning development cycle 15
Figure 5 - Example of shallow and deep learning networks 15
Figure 6 – Structure of a Markov Decision Process and Partially Observable MDP 24
Figure 7 – Overview of RL algorithms and implementation contexts 28
Figure 8 – Intersection between RL method approaches 29
Figure 9 - High level view of the asynchronous RL method 34
Figure 10 – Asynchronous Advantage Actor Critic (A3C) topology 37
Figure 11 – Slot environment configuration 40
Figure 12 - Task environment definition 41
Figure 13 - Reward configuration and additional parameters 44
Figure 14 - A3C neural network architecture and role of structure parameters 45
Figure 15 - Dashboard for simulation setup 46
Figure 16 - Dashboard with running simulation 48
Figure 17 - High level system architecture 49
Figure 18 - High level simulation classes and relationships... 50
Figure 19 - Structure of the interaction between agent and environment..................... 51
Figure 20 - Internal representation of environment data 52
Figure 21 - Data collection pipeline 54
Figure 22 - Dummy environment goals 55
Figure 23 - Dummy environment configuration parameters 56
Figure 24 - Dummy environment A configuration 57
Figure 25 - Reward results on dummy environment A.. 58
Figure 26 - RL Agent losses on dummy environment A.. 59
Figure 27 - Optimal solution for environment A 59
Figure 28 - Dummy environment B configuration 60
Figure 29 - Reward results on dummy environment B 60
Figure 30 - RL Agent losses on dummy environment B 61
Figure 31 - Optimal solution for environment B 61
Figure 32 - Dummy environment C configuration 62

Figure 33 - Reward results on dummy environment C 63
Figure 34 - RL Agent losses on dummy environment C 63
Figure 35 - Solution evolution on dummy environment C for agent A5 64
Figure 36 - Dummy environment D configuration 65
Figure 37 - Reward results on dummy environment D 66
Figure 38 - RL Agent losses on dummy environment D 66
Figure 39 - Optimal solutions on dummy environment D 67
Figure 40 - Dummy environment E configuration 68
Figure 41 - Reward results on dummy environment E 69
Figure 42 - RL Agent losses on dummy environment E 70
Figure 43 - Examples of optimal on dummy environment E 70
Figure 44 - Dummy environment F configuration 72
Figure 45 - Reward results on dummy environment F 73
Figure 46 - RL Agent losses on dummy environment F 74
Figure 47 - Optimal solutions for dummy environment F 74
Figure 48 – Real daily environment slot structure 76
Figure 49 – Real daily environment task attribute structure 77
Figure 50 - Reward results on real daily environment 78
Figure 51 - RL Agent losses on real daily environment 78
Figure 52 - Optimal solutions for real daily environment 79

Chapter 1

Introduction

Overview

Patient scheduling is a complex task that plays a crucial role in the quality of care. Patient scheduling takes many forms, from allocating resources to patients in need of exams and allocation of surgery rooms to on-demand appointment scheduling with Family Doctors working at Primary Care clinics. Effective appointment scheduling avoids patient and physician dissatisfaction, and as such, is an important determinant of care. However, the task of patient scheduling is traditionally done *ad hoc*, with disregard of potential aspects that may determine appointment duration and overall waiting time. Since appointment scheduling in this setting is frequently done sequentially, one patient at a time, the problem cannot be readily solved by optimization methods. Furthermore, the constant changing nature of clinical settings implies that a one-fits-all heuristic scheduling approach would generally perform poorly, rendering the task of automated patient scheduling resource intensive, costly to maintain and thus, generally unfeasible.

Motivation

The patient scheduling issue has a large impact in the effectiveness of provided care, on one hand through the potentially increasing medical error when professionals are under the pressure of increasing patient waiting times, and on the other, due to the decreasing compliance of unsatisfied patients. Thus, developing intelligent algorithms that can learn near optimal ways of scheduling patients, without the need of human intervention, whether in development, maintenance or production environments, becomes very relevant.

The recent advances in Machine Learning attributed to Deep Learning, together with the stronger intertwine with Reinforcement Learning, namely the Asynchronous Advantage Actor Critic algorithm, makes assessing the performance of such algorithms on patient scheduling problems worthwhile. Because of their learning ability, they become interesting as they can perform resource intensive training in context-specific problems offline, and then used online in real-time.

The task of scheduling patients requires not only appointment time, but also shifting the doctor focus from the realm of case-based clinical problem solving, to the realm of resource planning problems. Shifting between disparate problem realms, taking decision in short time, and the lack of adequate information, increases the odds of poor scheduling decisions, despite the cost imposed by such cognitive effort. Transferring the scheduling responsibility from the doctor to an intelligent and trained agent would keep the doctor focused on case-based clinical problem solving, lessen the doctor cognitive load, and potentially result in near optimal schedules, less prone to put both doctor and patient under the pressures of unmet time constraints.

Objective

The present work aims to develop a framework to create fully configurable patient scheduling environments, and study the potential of Reinforcement Learning techniques to schedule appointments in a set of increasingly challenging environments, and benchmark it against a set of deterministic heuristics.

Contributions

The present work has provided the following contributions:

- It Implemented a software-as-service web-based application that can be deployed in the cloud to create, develop and test configurable scheduling environments and reinforcement learning agents, and monitor their performance in real-time;
- It studied the application on Asynchronous Advantage Actor Critic algorithm in scheduling environments in Healthcare, namely in the Primary Care, describing implications to the general class of task scheduling problems;
- It extends the number of environments made publicly available by the Open AI initiative, and thus enabling a new set of problems that can be useful to benchmark reinforcement learning agents in general.

Structure of the Dissertation

This Dissertation introduces the problem of patient scheduling in Chapter 2 and discusses different formulations of the problem, recent research in the field and highlights the main advantages and drawbacks of strategies employed to solve such problems. Afterwards, Chapter 3 generally considers the field of machine learning

before focusing on the formal definition of deep neural network structure, error optimization and regularization strategies. It then proceeds to introduce reinforcement learning and describes the evolution of algorithms up to the state-of-art Asynchronous Advantage Actor-Critic algorithm. Chapter 4 formulates the scheduling problem as a reinforcement learning environment and presents its structure and implementation as a set of micro-services. Chapter 5 presents a set of dummy scheduling environments, their underlying rules, and reports the candidate agents quantitative and qualitative performance for each environment, along with a small discussion. Chapter 6 discusses major findings, and presents the conclusions.

Chapter 2

Patient appointment scheduling

Scheduling

Scheduling can be described as the process of assigning jobs to resources for some duration. Scheduling problems are ubiquitous, ranging from computer systems and networks, to production factories and patient appointments. Many of such problems are either solved by manual operation or using heuristics specifically designed to the context.

When such heuristics are designed the common approach is the following [1]:

1. Use business knowledge to model a simplified version of the problem;
2. Come up with a clever heuristic for the problem through experimentation;
3. Test and tweak the heuristic to achieve good performance.

Such heuristics must be reviewed whenever any of the underlying assumptions of the model changes becoming costly and difficult to maintain.

Multiple factors explain why development of scheduling solutions is challenging:

1. The system is complex and often impossible to model accurately [2], [3];
2. The algorithm usually has to decide on noisy input or incomplete information [4];
3. The objective measurement is hard to optimize in a principled manner [5].

Thus, scheduling optimization problems are costly and challenging. In addition, because the state space for this class of problems grows exponentially, it belongs to the NP-Hard problem class [6] that are often computationally intractable [7]. Poor decisions on such problems can lead to adverse outcomes, for example, in healthcare, non-urgent patients that experience prolonged wait before their appointment adhere less to treatments, and are prone to miss future appointments [8], [9].

Scheduling in Healthcare

Healthcare providers are stimulated to reduce operation cost while improving the quality of service. This has led to a shift to preventive medicine in order to avoid disease, lessening the demand for emergency department and hospital stays [10], therefore

reducing the cost of care. This shift was accompanied by an increase in outpatient services, that are increasingly being transferred from the outpatient hospital setting to primary healthcare clinics, that are able to provide healthcare services tailored to regional needs [11].

As this shift takes place, the problem of patient scheduling becomes increasingly relevant. Two major end-points are patient waiting times and waiting-room congestion, whose improvement through well-designed appointment systems is worthwhile [10]. In the outpatient setting, such systems have the potential to increase access to medical resources while reducing cost, as well as staff and patient dissatisfaction derived from unmet schedule constraints. Indeed, surveys indicate that patients dissatisfaction is related to waiting times [12], and that patient adherence is improved by shortening such time [13].

Research regarding scheduling in healthcare has focused on development and analysis of algorithms to specific problems, such as optimal allocation of resources for patient rehabilitation [14], maximization of operation room block times [15], optimal patient scheduling subject to patient no-shows and appointment cancelations [16], allocation of diagnostic resources in hospital settings [17], and outpatient appointment scheduling [18].

Thus, studying outpatient scheduling mechanics and factors that influence it becomes relevant. The development of strategies for automated or semi-automated scheduling through scheduling suggestions could be a relevant step in improving the efficiency of patient appointment scheduling and directly impact patient and doctor satisfaction.

Patient scheduling

The goal of optimal patient scheduling is to find an appointment strategy for which a particular measure of performance is optimized under uncertain conditions [10]. This formulation is applicable in different healthcare settings, from scheduling treatment procedures to patient appointments in the primary care setting, the latter being the focus of this work. Patient appointment scheduling can be classified into three processes [19]:

Single Batch Process

Appointment scheduling decisions are not made until after receiving all appointment requests for a given period. This model is commonly used in surgery starting times [19], and allows scheduling with complete information, so that a perfect or near perfect solution can be found through discrete optimization or heuristic methods.

Unit Process

Appointments are assumed to come one at a time and are scheduled at the time of the request arrival [19]. Through this process, a perfect solution will unlikely be found, but may be approximated if the distribution of appointment request types is learned.

Periodic Process

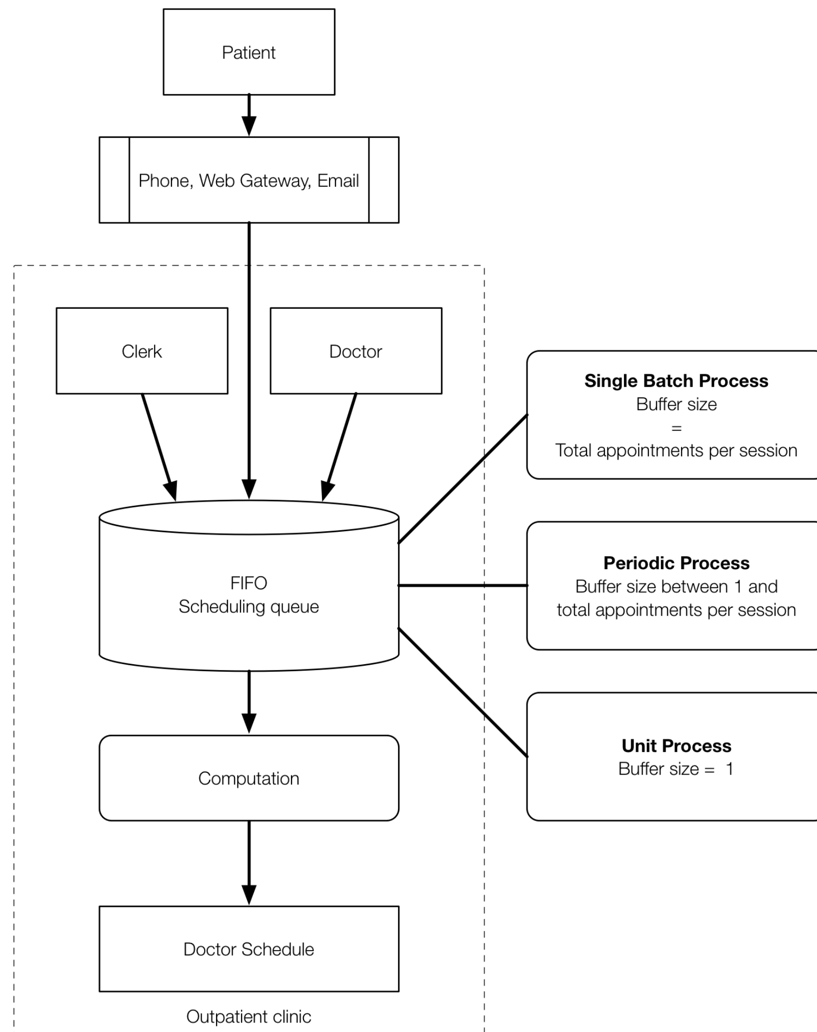
Appointment requests are kept in a buffer of fixed size and are scheduled once the buffer is full. [19]. This allows a better approximation to the optimal solution by considering optimal or near optimal solutions at each period.

An outpatient clinic in the context of Primary Care can be modelled as a *first-in-first-out* queuing mechanism, in which the size of the buffer determines the nature of the underlying process, as shown in Figure 1.

There is a vast body of literature on scheduling formulations considering single batch processes [10]. These have focused on studying the impact of design decisions such as appointment spacing and duration [20], patient arrival at the beginning of the day *versus* at appointment time, as well as how doctor and patient no show rates affect appointment waiting times [21]. However, the unit process which is required for scheduling appointments in the Primary Care setting is not studied as much.

The single batch process is not feasible in the context Primary Care, where appointment requests arrive continuously and the scheduling must be performed either immediately or in very short batches, depending on the channel of appointment request. In addition, during an appointment additional appointments may be scheduled. Thus, the unit and periodic processes are at play in such setting.

Figure 1 - Simplified scheduling model of a Primary Care clinic



FIFO – First in first out queueing policy.

Appointment structure in Primary Care

Primary Care clinics are intended to serve the specific health needs of a given population, and serve as a gateway to additional healthcare resources, available at hospitals and other specialized institutions. Doctors in the Portuguese Primary Care clinics are assigned to several families of the serviced population - up to 1900 patients - and take care of the health needs of such families from birth to death. Clinics usually encompass 5 to 8 doctors, nurses and clerks, among other professionals, and may provide additional services aside from medical appointments, such as x-ray imaging and lab tests. This proximity to the population bestows upon the Primary Care doctors the role of gatekeepers of the healthcare system [19].

Appointments usually take slots of 10, 20 or 30 minutes, with 20 minutes being the standard slot duration. Each doctor schedule is segmented according to appointment type, so that appointments requiring common resources or similar clinical management strategies are grouped together at a time of the day. Appointments can also take longer than standard slot duration to complete, as is the case for new-born and children appointments, among others. Schedules usually span five working days, totalling around 30 hours for appointments, and are usually designed by the doctors working at the clinics. Appointments can usually be scheduled 20 to 30 weeks in advance, but clinics are usually required to ensure a maximal scheduling delay for appointments requested by the patients, which in Portugal is set in 15 calendar days.

In addition, Primary Care clinics offer a variable number of daily slots to accommodate appointment requests for urgent situations, that may require prompt treatment [22]. Scheduling such type of appointments is out of the scope of the current work.

In the Portuguese Primary Care clinics, the following are the main appointment types that are usually scheduled, and their usual duration:

- Child health – 30min
- Adult health – 20min
- Hypertension – 20min
- Diabetes – 20min
- Open access – 10min
- Family planning – 20min
- Oncological screening – 20min
- Low risk pregnancy – 20min

There are other appointment types created to model patient accessibility constraints, which are also out of the scope of this work. A typical primary care doctor schedule has a structure like the example in Figure 2.

Clinics allow patients to schedule appointments on demand, directly at the clinic clerks, through email, telephone or via websites. The doctor or the clerk may also schedule appointments for the patient on their behalf, in case the patient has chronic disease that requires periodic monitoring, or if the patient missed important appointments. Thus, there are several concurrent actors and scheduling processes taking place at a time.

Figure 2 – Example schedule diagram of a typical primary care doctor schedule

	Monday	Tuesday	Wednesday	Thursday	Friday
08:00		Open access	Open access		
08:30					Adult health
09:00	Adult health	Diabetes	Child health		
09:30					
10:00					
10:30					
11:00	Open access		Adult health		
11:30					
12:00					
12:30			Open access		
13:00					
13:30					
14:00	Low risk pregnancy follow-up	Adult health		Open access	
14:30			Adult health	Hypertension	
15:00					
15:30					
16:00	Oncological screening				
16:30					
17:00					
17:30					
18:00					
18:30					
19:00				Adult health	
19:30					

White space indicates space unavailable for scheduling.

Scheduling opportunities in Primary Care

The current scheduling process in the Portuguese Primary Care is ill-defined, presenting several drawbacks that are also opportunities for improvement.

Scheduling is an overlooked *ad-hoc* process

Primary care appointment scheduling is often an overlooked task carried *ad-hoc* with little regard for planning and thus, far from optimal. Furthermore, because the subject requesting the appointment can vary between the doctor, the clerk and the patient, the criteria considered in each situation may differ substantially and conflict in their goal.

Slot schedules and appointment types are not aligned with appointment frequency

Appointment demand can be modelled as a stochastic non-stationary process. The nature of such process implies that the assumptions considered when designing schedules must be periodically revised. Such task imposes additional administrative overhead, not only because of the implied periodic revision of the schedule, but also because when scheduling, health professionals require additional effort to consider the

changes. These circumstances make schedule design far from optimal and rapidly outdated.

Slot times can be further targeted to specific population needs

Because scheduling is usually done manually, the more appointment types are defined, the harder it becomes for a human agent to schedule appointments, because the number of rules to consider increases, overwhelming the agent. Scheduling by human agents undermines further segmentation of appointment types and schedule structure, which could otherwise be tailored to meet population needs and take into consideration more variables that influence slot requirements, such as advancing age, existing comorbidities, among others. Creating automated scheduling systems can leverage such complexity as the increasing number of rules may not impact scheduling performance.

Scheduling requirements for Primary Care

So far, we have seen that there are many ways in which the current method of patient appointment scheduling may be improved by creating scheduling algorithms under the assumption that the process can be modeled either as a unit or periodic process.

Such system must meet three main challenges:

- It must continuously adapt to the underlying changes in the appointment distributions without the need for human intervention;
- It must be able to schedule appointments with incomplete information about the sampling of the appointment request distribution;
- It must execute in real-time so that it can be incorporated into the daily workflow of Primary Care clinics without introducing delay.

In addition, such system must be subject to the following constraints:

- It should prefer slots at specified times according to configurable slot costs;
- It should allocate enough slots to meet the requirement for each appointment type;
- It should group similar appointments together;
- It should avoid leaving empty slots between appointments;
- It should avoid appointment overlap.

Because this is a combinatorial problem that is subject to a variable number of soft constraints, it is NP hard, implying that optimal solutions cannot be computed in real-time except for small dummy settings, since as the problem space grows, the problem becomes computationally intractable.

Next, we will consider existing methods to solve this class of problems.

Methods of discrete optimization and relaxation techniques

Discrete optimization techniques cannot handle this problem easily. Classical, predictive approaches to solve task scheduling problems have covered disjunctive programming, branch-and-bound algorithms, shifting bottleneck heuristic [23], among others [24]. These techniques iteratively explore the state space one solution at a time, looking for optimal solutions. To reduce the number of search states, the constraints can be relaxed through techniques such as backtracking, that assess whether given state subspaces need not be searched by computing context-dependent heuristics that determines whether all states in that subspace are suboptimal. When in presence of soft constraints, even backtracking does not help, firstly because the heuristic is context-dependent and may need revision when underlying assumptions change, a secondly, since the problem formulation poses no predetermined upper or lower bounds, no heuristic can determine if large subspaces can be skipped, and thus, the problem remains intractable.

Implementations considering metaheuristic approaches

Metaheuristics are high-level general heuristics inspired in natural phenomena, developed to find heuristics that may provide sufficiently good solutions to optimization problems, particularly in settings of limited resources or information [25]. Such algorithms sample sets of solutions from spaces that otherwise were too large to be sampled. Because they do not make many assumptions about the underlying structure of the problem to be solved, they can be applied to problems of different nature [26]. Techniques such as simulated annealing [27], tabu search [28] and genetic algorithms [29] have been employed in task scheduling problems. In particular, genetic algorithms have been applied to appointment scheduling problems in the context of healthcare with success [14]. The main drawback of such approaches for the problem at hand is that finding near optimal solutions is time consuming and resource expensive, and thus cannot be performed in real-time.

Machine learning methods

Another class of methods that have been used for the development of scheduling algorithms in clinical practice comes from the machine learning field. Such methods seek to approximate functions from data that can be used to predict outcomes to drive decision [30]. Supervised methods have been employed using annotated scheduling examples to train machine learning systems to output viable scheduling options [18].

A class of machine learning algorithms seems well suited for the scheduling problems, namely reinforcement learning (RL) algorithms. Such algorithms acquire experience to perform a task from trial and error [31] and improve online performance through a balance between exploration and exploitation of the environment with which they interact [31]. Because such agents learn online against an environment, they seem well suited for the scheduling problem, since such environments can be easily created by defining a schedule, appointment attributes, and appointment request probabilities can be estimated from observed appointment frequencies. In addition, the fact that they learn from experience means that they can be trained to perform under a scheduling context before being used online, where they are able to solve problems in real-time.

In fact, such agents have been used to solve scheduling problems such as scheduling of robotic arm movements [32], general task scheduling problems [33], among others, with success. However, to our knowledge, such algorithms have not been applied in the context of appointment scheduling in Primary Care.

Considering the limitations of other algorithm classes, it becomes worthwhile to assess whether RL agents can solve appointment scheduling problems in Primary Care.

Chapter 3

Reinforcement Learning

Reinforcement Learning (RL) consists in having an agent algorithm, capable of interaction with an environment, to learn an optimal policy through trial and error [34].

RL has a strong connection with neural networks, and more recently deep networks, with major breakthroughs such as deep the Q-network [35], AlphaGo [36], and asynchronous architectures [37]. These breakthroughs have enabled RL agents to perform certain tasks at human level, or even beyond human level.

Deep learning potentiates RL because the modelling capability of deep neural networks enables the representation of compound functions that may map observed states to policies effectively. Thus, it is relevant to briefly discuss deep learning in the context of machine learning before delving into RL.

Machine learning overview

Machine learning is broad a field composed of methods and algorithms that are generally employed to extract patterns from data. Machine learning problems are usually classified into the following three categories:

Supervised learning

The algorithm is given labelled data and is trained to predict the labels, such as regression or classification.

Unsupervised learning

The algorithm is given unlabeled data and is intended to find hidden structure in the data, such as clustering.

Reinforcement learning

The algorithm is given unlabeled data in the form of an observation and performs an action that potentially transforms the environment, plus a reward. The algorithm is intended to learn to act given the observed state, through a learned policy and reward. Because reward in RL problems can be delayed, the agent is expected to learn to maximize cumulative reward, which may require taking suboptimal actions in the way [38].

Model training and development

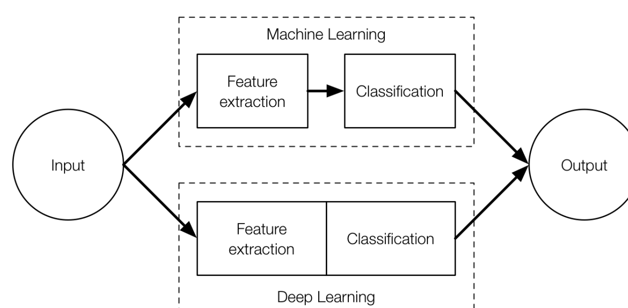
In the case of supervised learning the algorithm is developed using training and validation datasets, and later performance measured against a testing dataset. In reinforcement learning data is sequentially generated from the interaction with the learning environment, and learning is interweaved with execution, which makes development and training of reinforcement learning algorithms distinct from others. Learning is typically achieved through loss function minimization using gradient descent methods.

Model performance

While in supervised learning model performance is assessed through a measure of error rate in the test set, and in unsupervised learning structure is assessed through internal or external fitness measures, in RL the cumulative reward attained by the agent, and the consistency of its actions are ways of measuring model performance.

There are many different machine learning algorithms that can learn models by approximating functions of different nature to observed data. The term *deep learning* intends to contrast with *shallow* learning algorithms such as logistic regression, support vector machines, decision trees, among others in which is a single transformation layer and all potentially interesting features must be prepared and engineered beforehand [38] as depicted in Figure 3 in order for the model to achieve good results.

Figure 3 – Comparison between classical machine learning and deep learning

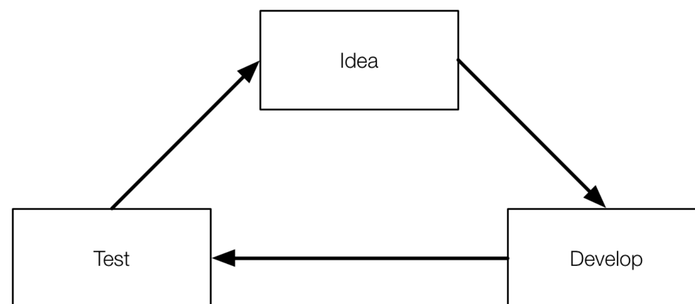


In classical machine learning pipelines features extraction play a relevant role in overall performance. Deep learning neural networks learn composable feature representations that render feature extraction less relevant.
Adapted from [39].

To develop successful machine learning algorithms, it is very important to rapidly iterate over different algorithm developments and test its behaviour, which will likely bring

about ideas on how to tweak the algorithm to achieve better results. Fast iteration along this cycle is very important to achieve good results in a timely fashion.

Figure 4 – Machine learning development cycle

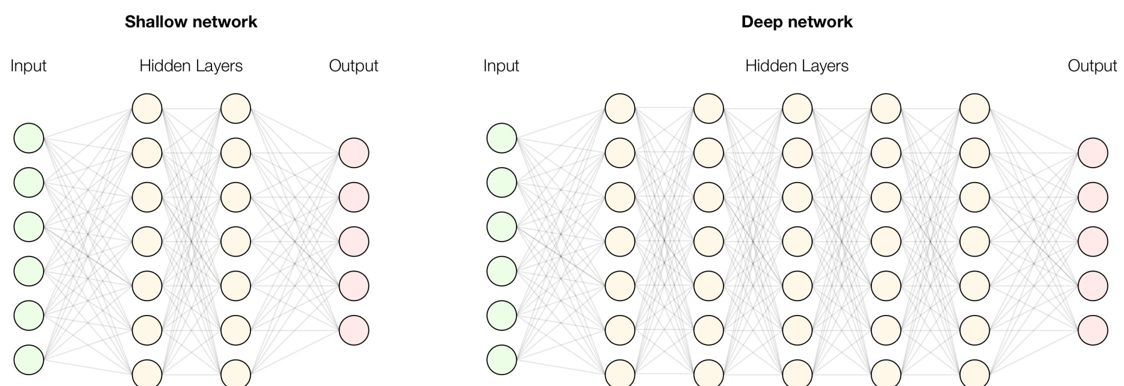


Idea, development and testing iteration for machine learning.

Deep learning

Deep learning, which corresponds to the use of deep neural networks (DNN) with many hidden layers, and can be used for supervised, unsupervised and RL. In deep learning, between input and output layers, there are one or more hidden layers as shown in Figure 5.

Figure 5 - Example of shallow and deep learning networks



Adapted from [39].

These networks learn intermediate representations from granular features passed as input automatically. The composite nature of feature representation is a central idea in deep learning, implying that a feature may be represented by a combination of multiple inputs, and a given input may contribute to computation of multiple features. It becomes the role of the units in the hidden layers to find composite features that map

the input to the desired output. Stacking multiple layers enables powerful feature composition.

Feedforward neural networks

Different Artificial Neural Networks (ANN) architectures can be composed to provide different tasks. Deep Feedforward Neural Networks, often referred as Feed-Forward Neural Networks (FFN) or Multi-Layer Perceptron (MLP), are commonly used deep learning models. A Feedforward Neural Network defines a function of $\boldsymbol{y^0}$, the vector of input values, parametrized by $\boldsymbol{\theta}$, $\boldsymbol{y}^{(L)} = \varphi(\boldsymbol{y^0}, \boldsymbol{\theta})$, or for a lighter notation, $\boldsymbol{y}^{(L)} = \varphi(\boldsymbol{y^0})$. The parameter vector $\boldsymbol{\theta}$, learned by the FFN, seeks to minimize approximation error, through the composition of simpler functions in each unit of each layer. These models are called networks because they compose together many different nodes and layers. They are called feedforward because the data flows forward from the input nodes, through the hidden nodes to the output nodes. There are no cycles, loops in the network or feedback connections. The term *Deep Learning* thus arises because we usually have $\boldsymbol{L}$ functions $\varphi^1, \varphi^2, \dots \varphi^L$ connected in chain structures of neural networks, with an overall length $\boldsymbol{L}$, known as the depth of the model, to form $\varphi(\boldsymbol{y^0}) = \varphi^L(\varphi^{L-1} \dots (\varphi^2\left(\varphi^1(\boldsymbol{y^0})\right) \dots))$. FNNs can be modelled by iterating the following equations:

$$\boldsymbol{h}^{(l)} = \boldsymbol{W}^{(l)}\boldsymbol{y}^{(l-1)} + \boldsymbol{b}^{(l)}$$
$$\boldsymbol{y}^{(l)} = \varphi(\boldsymbol{h}^{(l)})$$

Equation 1 – Feed forward neural network.

where $l \in \{1,\dots,L\}$ denotes the lth layer, $\boldsymbol{h}^{(l)} \in \mathbb{R}^{n_i}$ is a vector of preactivations of layer l, $\boldsymbol{y}^{(l-1)} \in \mathbb{R}^{n_o}$ is the output of the previous layer $(l-1)$ and input to layer l, $\boldsymbol{W}^{(l)} \in \mathbb{R}^{n_i \times n_o}$ is a matrix of learnable weights of layer l, $\boldsymbol{b}^{(l)} \in \mathbb{R}^{n_i}$ is a vector of learnable biases of layer l, $\boldsymbol{y}^{(l)} \in \mathbb{R}^{n_i}$ is the output of layer l, $\boldsymbol{y}^{(0)}$ is the input to the model, $\boldsymbol{y}^{(L)}$ is the output of the final layer L of the model, and φ is a nonlinear activation function applied element-wise.

In modern FFNs, the default recommendation is to use the Rectified Linear Unit (ReLU) [40], [41] defined by the rectifier activation function show below

$$\varphi({h_i}^{(l)}) \ = \ max(0, {h_i}^{(l)})$$

Equation 2 – Rectified linear unit activation function.

where ${h_i}^{(l)}$ represents the ith component of $\boldsymbol{h}^{(l)}$. This function has advantages over other activation functions, such as computational simplicity and faster learning convergence, which is a major requirement for networks with hundreds or thousands of layers [41].

To provide probabilistic interpretations to the model, the output the final layer of the network usually computes a *softmax* nonlinearity instead of other nonlinear activation functions, as given by the equation below,

$$\varphi\left({h_i}^{(l)}\right) = \ softmax\left({h_i}^{(L)}\right) = \frac{e^{{h_i}^{(L)}}}{\sum_{j=1}^{K} e^{{h_j}^{(L)}}}$$

Equation 3 – Softmax activation function.

where K represents the number of output classes of the Lth layer corresponding to the last one. However, this function must be stabilized against underflow and overflow errors.

Recurrent Neural Networks

In contrast with FFNs, Recurrent Neural Networks (RNN) include feedback connections. RNNs process sequential inputs, element by element, and make use of hidden units to store history of past elements [42]. They extend the notion of typical FFN by adding inter-layer and self-connections to units in the recurrent layer [43]. This makes such type of architectures particularly suitable for tasks that involve sequential inputs such as speech. They can be modelled according to the next equation,

$$\boldsymbol{h}_t^{(l)} \ = \ \boldsymbol{W}_y^{(l)} \boldsymbol{y}_t^{(l-1)} + \boldsymbol{W}_s^{(l)} \ \boldsymbol{s}_{t-1}^{(l)} + \boldsymbol{b}^{(l)}$$

Equation 4 – Recurrent neural network.

where t denotes the time step, $\boldsymbol{h}_t^{(l)} \in \mathbb{R}^{n_i}$ is a vector of preactivations of layer l at time step t, $\boldsymbol{y}_t^{(l-1)} \in \mathbb{R}^{n_o}$ is the output of the previous layer $(l-1)$ at time step t and input to layer l at time step t, $\boldsymbol{W}_y^{(l)} \in \mathbb{R}^{n_i \times n_o}$ is a matrix of learnable weights of layer l, $\boldsymbol{s}_{t-1}^{(l)} \in \mathbb{R}^{n_o}$ is the state of layer l at the previous time step $(t-1)$, $\boldsymbol{W}_s^{(l)} \in \mathbb{R}^{n_i \times n_o}$ is a matrix of learnable weights of layer l, and $\boldsymbol{b}^{(l)} \in \mathbb{R}^{n_i}$ is a vector of learnable biases of layer l. For

recurrent architectures, sigmoid functions, such as the logistic function, $\sigma(x)$, presented in the next equation, as well as the hyperbolic tangent (TanH) function, are frequently used as activation unit functions, instead of ReLU, since the latter amplifies the exploding gradient problem due to their unbounded nature [44].

$$\varphi\left({h_i}^{(l)}\right) = \sigma\left({h_i}^{(l)}\right) = \frac{1}{1 + e^{-{h_i}^{(l)}}}$$

Equation 5 – Sigmoid activation function.

A popular variant of the recurrent architectures is the Long Short-Term Memory (LSTM) [45], which uses an explicit memory cell to represent long-term dependencies more effectively. An LSTM network computes a mapping from an input sequence to an output sequence by calculating the network unit activations using the following equations iteratively from $t = 1$ to $t = T$:

$$\boldsymbol{i}_t^{(l)} = \sigma\left(\boldsymbol{W}_{iy}\boldsymbol{y}_t^{(l-1)} + \boldsymbol{W}_{ih}\boldsymbol{h}_{t-1}^{(l)} + \boldsymbol{W}_{ic}\boldsymbol{c}_{t-1}^{(l)} + \boldsymbol{b}_i^{(l)}\right)$$

$$\boldsymbol{f}_t^{(l)} = \sigma\left(\boldsymbol{W}_{fy}\boldsymbol{y}_t^{(l-1)} + \boldsymbol{W}_{fh}\boldsymbol{h}_{t-1}^{(l)} + \boldsymbol{W}_{fc}\boldsymbol{c}_{t-1}^{(l)} + \boldsymbol{b}_f^{(l)}\right)$$

$$\boldsymbol{c}_t^{(l)} = \boldsymbol{f}_t \odot \boldsymbol{c}_{t-1}^{(l)} + \boldsymbol{i}_t \odot tanh\left(\boldsymbol{W}_{cy}\boldsymbol{y}_t^{(l-1)} + \boldsymbol{W}_{ch}\boldsymbol{h}_{t-1}^{(l)} + \boldsymbol{b}_c^{(l)}\right)$$

$$\boldsymbol{o}_t^{(l)} = \sigma\left(\boldsymbol{W}_{oy}\boldsymbol{y}_t^{(l-1)} + \boldsymbol{W}_{oh}\boldsymbol{h}_{t-1}^{(l)} + \boldsymbol{W}_{oc}\boldsymbol{c}_t^{(l)} + \boldsymbol{b}_o^{(l)}\right)$$

$$\boldsymbol{h}_t^{(l)} = \boldsymbol{o}_t^{(l)} \odot tanh\left(\boldsymbol{c}_t^{(l)}\right)$$

Equation 6 – Implementation of long short term RNN cell

where σ is the logistic sigmoid function, $\boldsymbol{i}$, $\boldsymbol{f}$, $\boldsymbol{o}$ and $\boldsymbol{c}$ are the input gate, forget gate, output gate and cell activation vectors respectively, all of which are the same size as the cell output activation vector $\boldsymbol{h}$. The $\boldsymbol{W}$ terms denote weight matrices from input gates to the input, $\mathbf{W}_{\mathrm{fy}}$ is the matrix of weights from the forget gate to the input $\boldsymbol{y}_t^{(l-1)}$, and are all diagonals such that each element in each gate vector only receives input from the same element of the cell vector [46].

Learning

Optimization algorithms used for training deep models differ from traditional optimization algorithms in several ways. Machine learning usually acts indirectly. In most machine learning scenarios, we care about some performance measure P, that is defined with respect to the test set and may also be intractable. We therefore optimize P only indirectly. We reduce a different cost function $C(\varphi(\mathbf{y}^{\mathbf{0}}, \boldsymbol{\theta}, \mathbf{y}^{(\mathrm{L})}))$ in the hope that doing

so will improve P. This contrasts with pure optimization, where minimizing J is a goal in and of itself.

There are many different cost functions that can be selected, but for illustration the cross-entropy cost function [47] is defined as:

$$C(\varphi(\boldsymbol{y}^{\mathbf{0}}, \boldsymbol{\theta}, \boldsymbol{y}^{(L)})) = \sum_{k=1}^{K} y_k log\ (y_k^{(L)})$$

Equation 7 – Cross entropy cost function.

where $y \in \{0, 1\}^k$, is a one-of-K encoded label and $\boldsymbol{y}^{(L)}$ is the output of the model.

The gradients are computed by differentiating the cost function with respect to the model parameters using a mini-batch of *m* examples from the training set, $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m\}$, with corresponding targets $\boldsymbol{y}^{(l)}$ and back propagated to prior layers using the backpropagation algorithm [48]. Training recurrent architectures requires modification to the backpropagation algorithm to compute the gradients with respect to the parameters and states of the model, which is known as the *backpropagation through time* algorithm [49].

These algorithms provide parameters that can be used to control the behavior of the learning algorithm, called hyperparameters. The learning rate is the most important hyperparameter because it has a significant impact on model performance. The values of hyperparameters are not adapted by the learning algorithm itself, though we can design a nested learning procedure where one learning algorithm learns the best hyperparameters for another learning algorithm, or carry a search procedure to find a near-optimal value for the learning rate. The cost is often highly sensitive to some directions in parameter space and insensitive to others. The momentum algorithm can mitigate these issues somewhat, but does so at the expense of introducing another hyperparameter, v that plays the role of velocity. It is the direction and speed at which the parameters move through parameter space. The velocity is set to an exponentially decaying average of the negative gradient.

However, since the directions of sensitivity are somewhat axis-aligned, using separate learning rates for each parameter and automatically adapt these learning rates throughout the course of learning seems worthwhile. Gradient descent or one of its

variants is used to update the parameters of the model using the gradients computed. A per-parameter adaptive variant of gradient descent called *RMSProp* [50] which uses gradient information to adjust the learning rate can be implemented as shown in Algorithm 1.

Algorithm 1 – RMSProp algorithm

while stop criteria not met **do**

sample a mini-batch of m examples from the training set, $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m\}$, and targets $\boldsymbol{y}^{(l)}$

compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_\theta \sum_l C(\varphi(\boldsymbol{x}^{(l)}, \boldsymbol{\theta}, \boldsymbol{y}^{(l)}))$

accumulate squared gradient: $\boldsymbol{r} \leftarrow \eta \boldsymbol{r} + (1 - \alpha) \boldsymbol{g} \odot \boldsymbol{g}$

compute parameter update $\Delta\boldsymbol{\theta} = -\frac{\alpha}{\sqrt{\delta + \boldsymbol{r}}} \odot \boldsymbol{g}$

apply update $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$

end

Implementation of RMSProp algorithm

$\boldsymbol{\theta}$ is the initial parameters vector; η and α are hyperparameters denoting the decay rate and the learning rate respectively;

δ is a small constant usually 10^{-6} to stabilize divisions by small numbers; $\boldsymbol{r}$ is a vector that accumulates squared gradient.

Regularization

Regularization is any modification made to a learning algorithm that is intended to reduce its generalization error but not its training error. Deep neural networks demonstrate excellent results on tasks with complex classification functions and enough training data. However, since DNN models have large numbers of parameters, they easily overfit when the amount of training data is not large enough. Thus, regularization techniques for neural networks are crucially important to make them applicable to a wide range of problems. There are several methods of regularization for deep learning. The most widely used methods being Parameter Norm Penalties and Dropout.

Parameter Norm Penalties

Many regularization approaches are based on limiting the capacity of models by adding a parameter norm penalty $\Omega(\boldsymbol{\theta})$ to the cost function. In this condition, the regularized cost function is given by Equation 8,

$$\tilde{C}\left(\varphi\left(\boldsymbol{y}^{\mathbf{0}}, \boldsymbol{\theta}, \boldsymbol{y}^{(L)}\right)\right) = C\left(\varphi\left(\boldsymbol{y}^{\mathbf{0}}, \boldsymbol{\theta}, \boldsymbol{y}^{(L)}\right)\right) + \alpha\Omega(\boldsymbol{\theta})$$

Equation 8 – Regularized cost function

where $\alpha \epsilon [0, +\infty[$ is a hyperparameter that weights the relative contribution of the norm penalty term, $\Omega(\boldsymbol{\theta})$, relative to the standard cost function $C\left(\varphi\left(\boldsymbol{y}^{\mathbf{0}}, \boldsymbol{\theta}, \boldsymbol{y}^{(L)}\right)\right)$. When $\alpha = 0$ there is no regularization. The parameter norm penalty Ω penalizes only the weights of the affine transformation at each layer and leaves the biases unregularized. The biases typically require less data to fit accurately than the weights. In addition, regularizing the bias parameters can introduce a significant amount of underfitting. A vector $\boldsymbol{w}$ is used to indicate the weights that should be affected by a norm penalty, and the vector $\boldsymbol{\theta}$ denotes the parameters, including both $\boldsymbol{w}$ and the unregularized parameters.

L^2 parameter regularization, commonly known weight decay, is the most common form of weight decay. The regularization strategy drives the weights closer to origin by considering the following [51] Equation 9:

$$\Omega(\boldsymbol{\theta}) = \frac{1}{2}\|\boldsymbol{w}\|_2^2 = \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w}$$

Equation 9 – L^2 parameter regularization.

L^1 parameter regularization on the model parameter $\boldsymbol{\omega}$ is defined show in Equation 10 [51]:

$$\Omega(\boldsymbol{\theta}) = \|\boldsymbol{w}\|_1 = \sum_i |w_i|$$

Equation 10 – L^1 parameter regularization.

L^1 regularization results in frequently sparse solutions since some parameters may have optimal value of 0. This sparsity property induced by L^1 regularization serves as a feature selection mechanism. Feature selection simplifies the learning problem by reducing dimensionality. Since the L^1 penalty causes a subset of the weights to become 0, such features can be discarded.

Dropout

Dropout is another method recently developed and ubiquitously used to train DNNs. This method is of simple implementation and frequently leads to significant performance improvement of DNNs. It consists in randomly removing neurons from the layer with a probability $1 - p$, making neuron output values equal to 0. Unlike other regularization techniques that modify the cost function, dropout modifies the architecture of the model. Using dropout, the feed forward operation is changed into the following [52]:

$$r_j^{(l)} \sim Bernoulli(p)$$
$$y^{(l)} = \boldsymbol{r}^{(l)} \odot \boldsymbol{y}^{(l)}$$
$$\boldsymbol{h}^{(l+1)} = \boldsymbol{W}^{(l+1)}\boldsymbol{y}^{(l)} + \boldsymbol{b}^{(l+1)}$$
$$\boldsymbol{y}^{(l+1)} = \varphi(\boldsymbol{h}^{(l+1)})$$

Equation 11 – Dropout regularization.

where $\boldsymbol{r}^{(l)}$ is a vector of Bernoulli random variables each of which has a probability p of being 1. This vector is sampled for each layer and multiplied element-wise with the outputs of that layer, $\boldsymbol{y}^{(l)}$, to create the pruned outputs $\mathrm{y}^{(l)}$, the inputs of the next layer.

For learning, the derivatives of the cost function are back propagated through the pruned network. At test time, the weights are scaled as $\boldsymbol{W}_{test}^{l} = p\boldsymbol{W}^{l}$ and with such weights considered in place of using dropout.

Reinforcement Learning Overview

The general task of machine learning is to teach a machine to produce a desired output for a given input. As previously described, ML can be divided into three classes that differ in the external feedback to the system during learning, namely Supervised Learning, Unsupervised Learning, and Reinforcement Learning.

RL learns in response to a scalar reward – hence *reinforcement* – that is given in response to the goodness of a sequence of actions. Experience is accumulated from triplets of input states, output actions and corresponding reward. Thus, RL is a class of computational algorithms that specify how an artificial agent can learn to select a sequence of actions that together maximize total reward. RL is frequently considered in between supervised and unsupervised learning methods.

Classically, RL techniques were developed in the realm of dynamic programming (DP) methods, considering concepts such as value and policy iteration [53]. Even though ANNs have been widely used in RL, a crowning achievement of deep learning has been its application to RL [38]. A RL problem can be described by considering the following items.

Environment

Corresponds to any kind of system that models a RL task. It defines the state and actions available to the RL agent, and implements the logic through which actions

transform state. At each step t, it provides a reward $\boldsymbol{r}_t$ to the agent, which serves as evaluation criteria for the action from previous states. The environment state $\boldsymbol{s}_t$ can be discrete or continuous.

Agent

Represents the controller of the system. It can observe either the full or partial state $\boldsymbol{s}_t$ of the environment. It interacts with the environment by performing actions $\boldsymbol{a}_t$ given state $\boldsymbol{s}_t$, and in return retrieving reward $\boldsymbol{r}_t$, which it can use to improve its policy.

Actions

Actions are ways through which the agent affects the environment state. They generally represent discrete actions that transform the system is some way, or continuous control parameters that need to be regulated. According to the problem setting, actions can be discrete or continuous, and one or multiple actions can be taken at each step. This creates a credit-assignment problem, derived from the sequence of actions taken and how each contributed to the total reward, in the case such reward is delayed in time [54].

Policy

The mapping from states of the environment to the action to be taken in the current state is called a policy π. Thus, a policy is formed from a sequence of actions and reflects the learnt behavior of the agent at a given time. Since in most application the goal is to optimize the behavior on a system for a given period, instead of a one-step optimization, one tries to determine an optimal policy regarding an overall objective, given by cumulative reward. As put by Sutton and Barto, *the policy is the core of a reinforcement learning agent in the sense that it alone is sufficient to determine behavior* [55].

Reward

The reward function specifies the overall objective of the RL problem. It depicts the immediate reward the agent receives for performing a certain action at a given system state. Consequently, it defines the desirability of a state and action for the agent. Generally, the simple immediate reward is only of minor interest, because high immediate rewards do not necessarily lead for maximum cumulative reward in the future. Instead, one is usually interested in the discounted value of collected rewards in the long run.

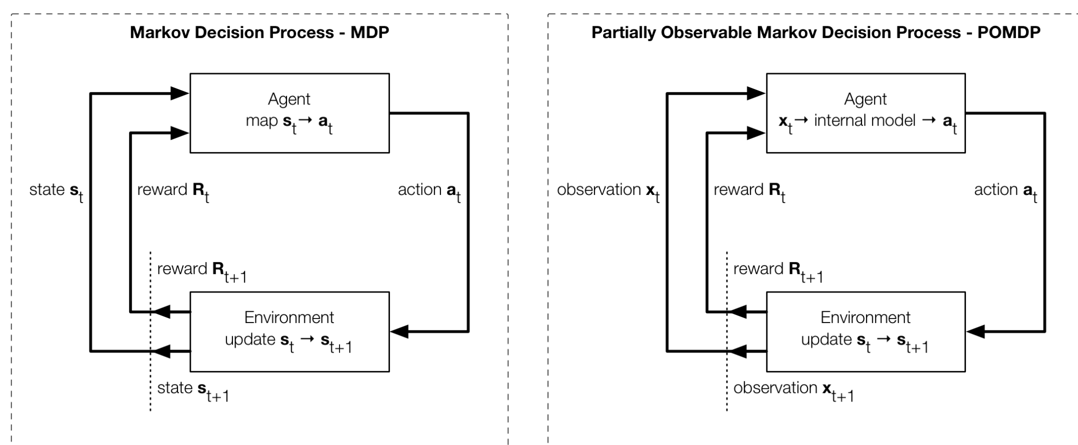
Markov Decision Process

The Markov decision process (MDP) provides the base mathematical formulation for RL problems. The MDP describes the development of a controllable dynamical system whose state is fully observable, or partially observable (POMDP). The MDP is defined by a tuple (S, A, T, R) with the following composition [18, 41, 65]:

- The state space of the environment S;
- The action space A with sets $A(\boldsymbol{s}_t)$ of available or allowed actions in state $\boldsymbol{s}_t \in$ S;
- A deterministic or stochastic state-transition function $T(\boldsymbol{s}_{t+1}|\boldsymbol{s}_t, \boldsymbol{a}_t)$, which defines the probability of arriving at state $\boldsymbol{s}_{t+1}$ from state $\boldsymbol{s}_t$ by applying action $\boldsymbol{a}_t$ with $\boldsymbol{s}_t, \boldsymbol{s}_{t+1} \in$ S and $\boldsymbol{a}_t \in \mathrm{A}(\boldsymbol{s}_t)$;
- A reward function $\boldsymbol{r}_t \leftarrow R(\boldsymbol{s}_t)$ denoting a one-step reward for being in state $\boldsymbol{s}_t$.

The given entities are related in every one-step transitions, such that being in an arbitrary state $\boldsymbol{s}_t$, the agent selects action $\boldsymbol{a}_t$, causing the system to transform into next state $\boldsymbol{s}_{t+1}$, according to the transition function T, and computing a one-step reward $R(\boldsymbol{s}_{t+1})$ [56]. Assuming the process is modelled as a MDP, the next state $\boldsymbol{s}_{t+1}$ depends only on the current state $\boldsymbol{s}_t$ and the applied action $\boldsymbol{a}_t$ [56], [57], and thus it is independent of its history. The Markov property in discrete time can be modelled according to Figure 6 [57]. The actions the agent chooses are defined according to a learnt policy or at random.

Figure 6 – Structure of a Markov Decision Process and Partially Observable MDP



Adapted from [58].

Partially Observable Markov Decision Process

POMDPs differ from MDPs in the fact that the state space S is not fully observable. This is usually the case of most real-world applications, such as patient scheduling in Primary Care. The RL agent receives an observation $\boldsymbol{x}_t$ that proxies the current system state $\boldsymbol{s}_t$, in which $\boldsymbol{x}_t$ is generally not Markovian. A POMDP can be described by a tuple (S, X, A, T, R), where X represents the space observable by the agent, which, while not necessarily a subspace of S, should contain enough information for the agent to learn a model of S and develop good policies, as represented in Figure 6.

Dynamic programming

Dynamic programming refers to a group of algorithms that can solve multi-state decision processes if they are provided a perfect model of the environment, such as an MDP [55]. These are based in *Bellman principle of optimality,* defined as follows [59].

Let $\{\boldsymbol{a}_0^*, \boldsymbol{a}_1^*, \boldsymbol{a}_2^*, \ldots \boldsymbol{a}_n^*,\}$ be an action sequence resulting from an optimal policy π^* for a fully observable problem, and that by using π^* a given state $\boldsymbol{s}_t$ occurs at time t with positive probability. Consider the sub-problem $\sum_{\tau=t}^{\infty} \gamma^{\tau-t} R(\boldsymbol{s}_{\tau+1})$ whereby one is at $\boldsymbol{s}_t$ at time t and wishes to maximize the cumulative reward from time t onwards with a discount factor $\gamma \in [0,1]$. Then, the truncated action sequence $\{\boldsymbol{a}_t^*, \boldsymbol{a}_{t+1}^*, \boldsymbol{a}_{t+2}^*, \ldots \boldsymbol{a}_{t+n}^*,\}$ is optimal for the sub-problem.

This implies that if the solution to the sub-problem was not optimal, then the total reward of the problem could be increased by switching to optimal policy when at state $\boldsymbol{s}_t$. Hence, π^* would not be optimal [60]. Furthermore, it implies an optimal policy can be determined by solving a step-by-step *tail sub-problem,* which is the basis of dynamic programming [60].

Based on this principle, DP operates on a value function $V^{\pi}(\boldsymbol{s}_t)$ that represents the expected cumulative reward [60] for each system state $\boldsymbol{s}_t$ given a policy π as follows:

$$V^{\pi}(\boldsymbol{s}_t) = E\left[\sum_{\tau=t}^{\infty} \gamma^{\tau-t} R(\boldsymbol{s}_{\tau+1})\right]$$

Equation 12- Dynamic programming value function.

The DP algorithm, also known as *value iteration*, seeks to maximize the value function through a backward iteration $(k \in N)$ as show in the following equation [60]:

$$V_{k+1}(\boldsymbol{s_t}) = \max_{\boldsymbol{a}_t \in (\boldsymbol{s}_t)} \left(\sum_{s_{t+1}} T(\boldsymbol{s}_{t+1} | \boldsymbol{a}_t, \boldsymbol{s}_t \left[R(\boldsymbol{s}_{t+1}) + \gamma V_k(\boldsymbol{s}_{t+1}) \right] \right)$$

Equation 13 - Value iteration equation.

The value function converges to V^* leading to the optimal policy π^* [59]. The maximisation of V is done on policy space since the dynamics of the state action space are dependent of the problem to solve. In addition, an intermediate step can be taken to evaluate the goodness of each state-action value pair, through the Q-function, defined by Equation 14:

$$Q^{\pi}(\boldsymbol{s}_t, \boldsymbol{a}_t) = \sum_{\boldsymbol{s}_{t+1} \in S} T(\boldsymbol{s}_{t+1} | \boldsymbol{a}_t, \boldsymbol{s}_t)[R(\boldsymbol{s}_{t+1}) + \gamma Q^{\pi}(\boldsymbol{s}_{t+1}, \pi(\boldsymbol{s}_{t+1}))]$$

Equation 14 - Q function.

Furthermore, $V^{\pi}(\boldsymbol{s}_t) = Q^{\pi}(\boldsymbol{s}_t, \pi(\boldsymbol{s}_t))$. Analogue to V^*, the optimal Q-function can be defined as shown in Equation 15, which is the Bellman optimality equation [59].

$$Q^*(\boldsymbol{s}_t, \boldsymbol{a}_t) \leftarrow \sum_{\boldsymbol{s}_{t+1} \in S} T(\boldsymbol{s}_{t+1} | \boldsymbol{a}_t, \boldsymbol{s}_t)[R(\boldsymbol{s}_{t+1}) + \gamma V^*(\boldsymbol{s}_{t+1})]$$

$$= \sum_{\boldsymbol{s}_{t+1} \in S} T(\boldsymbol{s}_{t+1} | \boldsymbol{a}_t, \boldsymbol{s}_t) \left[R(\boldsymbol{s}_{t+1}) + \gamma \max_{\boldsymbol{a}_{t+1} \in A(\boldsymbol{s}_{t+1})} Q^*(\boldsymbol{s}_{t+1}, \boldsymbol{a}_{t+1}) \right]$$

Equation 15- Bellman optimality equation

The optimal policy π^* is the one that maximizes Q-function $\pi^*(\boldsymbol{s}_t) = \underset{\boldsymbol{a}_t \in A(\boldsymbol{s}_t)}{arg\,max}\, Q^*(\boldsymbol{s}_t, \boldsymbol{a}_t)$.

A variation to value iteration previously defined in Equation 13 is the policy iteration [55], [59], which takes the policy directly into account. The value function is determined by doing a policy evaluation for a given policy π_i $(i, k \in N)$ as shown in Equation 16:

$$V_{k+1}^{\pi_i}(\boldsymbol{s}_t) = \sum_{\boldsymbol{s}_{t+1} \in S} T(\boldsymbol{s}_{t+1} | \boldsymbol{a}_t, \boldsymbol{s}_t)\left[R(\boldsymbol{s}_{t+1}) + \gamma V_k^{\pi_i}(\boldsymbol{s}_{t+1})\right]$$

Equation 16 - Policy iteration equation.

This corresponds to the expected infinite discounted reward, which is obtained by following policy π_i. Usually the equation is iterated until $|V_{k+1}^{\pi_i}(\boldsymbol{s}_t) - V_k^{\pi_i}(\boldsymbol{s}_t)| < \varepsilon$, with $\varepsilon > 0$. In a second step, the policy iteration method determines whether this value could

be improved by changing the immediate action taken. This results in the following policy update:

$$\pi_{i+1}(\boldsymbol{s}_t) = \underset{\boldsymbol{a}_t \in A(\boldsymbol{s}_t)}{arg\,max} \left(\sum_{\boldsymbol{s}_{t+1} \in S} T(\boldsymbol{s}_{t+1} | \boldsymbol{a}_t , \boldsymbol{s}_t) \left[R(\boldsymbol{s}_{t+1}) + \gamma V^{\pi_i}(\boldsymbol{s}_{t+1}) \right] \right)$$

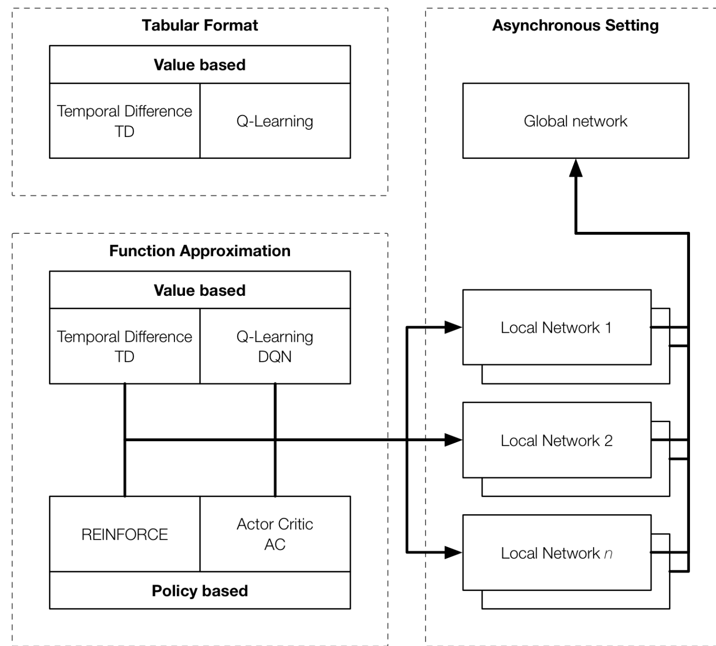
The two steps are iterated until $\pi_i = \pi_{i+1}$ which means that the policy has become stable.

Since the methods presented require a perfect model, they are of limited utility for application in extensive real-world RL problems. Despite this fact, they serve as the cornerstone for more robust methods that take into consideration FFNs [53], and thus, can be regarded as the foundation of RL [55]. By means of DL, it is possible to learn value and policies through function approximation using gradient optimization. The following sections describe classical and recent RL implementations using DNN, as summarized in Figure 7.

Reinforcement Learning Methods

Several different RL methods have been developed over the last years [55]. For illustration purposes, some of those methods will be briefly discussed. There are various ways to classify RL methods, however an important distinction exists between table-based and function approximation methods.

Figure 7 – Overview of RL algorithms and implementation contexts



TD – Temporal difference; **DQN** – Deep Q network; **AC** – Actor Critic.

Table based methods

These methods store the value of each state-action combination within a table. Because the table dimensions impose computational constraints, namely concerning memory, these methods can only be successfully applied to problems of discrete low-dimensional space. Multiple methods were initially implemented in such a way, namely Temporal Difference (TD) Learning and Q-Learning.

Function approximation

These methods replace the state-action value table with a device that learns a function mapping state-action pairs to value. Such methods display higher scalability and portability, thus can be applied to higher dimensional problems composed of continuous state and action spaces. Table based methods can be upgraded into function approximation methods can be seen from Algorithm 2 and Algorithm 3.

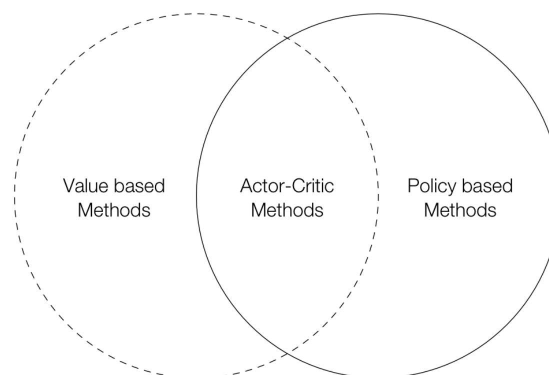
Another important difference between RL methods regards whether the method is model-free of model-based. Model-free methods learn a controller without learning a model, which is to say, without requiring the transition function T. Thus, they learn directly from the available data without building a model. This makes such algorithms generally fast and easy to implement. Model-based methods first learn a model which in turn is used to derive a controller [55]. This requires additional computation but makes

them more data-efficient. TD-learning and Q-learning are model-free whereas DP is model-based.

Moreover, as shown in Figure 8, methods can be classified according to value and policy:

- **Value-based methods** estimate a value function;
- **Policy-based methods** estimate a policy function;
    - **Actor-critic methods** estimate both value - the *critic* - and policy - the *actor*.

Figure 8 – Intersection between RL method approaches



The dashed line for value based methods in contrast with the full line for policy based methods indicates that actor-critic methods are often considered within the scope of policy based methods.

Finally, RL methods can be considered classified based on how they execute learnt policy:

- **On-policy** methods learn the value of the policy by performing it at every step. The agent always follows the policy that it is learning.
- **Off-policy** methods learn the value of the policy without necessarily performing it at every step. Such agents may choose to explore the environment using other strategies instead of the policy, and use experience gathered to update its value.

Value gradient methods

Temporal difference learning

Temporal Difference (TD) learning [61] results from a combination of DP and Monte-Carlo methods [62]. It directly learns from raw experience without a model of the

dynamics and thus is model-free [55]. The transition function T is not explicitly considered in the calculations. TD-Learning served as base for many other methods. The update rule for TD-learning is defined as $V(\boldsymbol{s}_t) \leftarrow V(\boldsymbol{s}_t) + \alpha[R(\boldsymbol{s}_{t+1}) + \gamma V(\boldsymbol{s}_{t+1}) - V(\boldsymbol{s}_t)]$ with a learning rate $\alpha \in [0, 1]$, which can either be fixed or variable [55].

TD-learning can either be table-based or use function approximation [34], [63]. Algorithm 2 presents an implementation of TD learning with immediate reward collection.

Algorithm 2 - Temporal Difference learning using tabular method

initialize V arbitrarily for all states
for *each episode* **do**
    initialize state s
    for *each step of episode, state* s *is not terminal* **do**
        Perform a according to policy $\pi(\cdot\,|s)$
        Receive reward r' and state s'
        $V(s) \leftarrow V(s) + \alpha[r' + \gamma V(s') - V(s)]$
        $s \leftarrow s'$
    end
end

α controls step size such that $\alpha > 0$. Adapted from Sutton and Barto [55].

This algorithm can be extended to use function approximation as shown in Algorithm 3. This way TD-Learning can be implemented using FFN. For that purpose the approximate value function is parameterized as $V(s, \boldsymbol{\theta_v})$, where $\boldsymbol{\theta}_v$ is a vector of weights. The gradient of $V(s, \boldsymbol{\theta_v})$ with respect to $\boldsymbol{\theta}_v$ is given by $\nabla_{\boldsymbol{\theta}_v} V(s, \boldsymbol{\theta_v})$, and the update rule is $\boldsymbol{\theta}_v \leftarrow \boldsymbol{\theta}_v + \alpha[\mathrm{R}(\mathbf{s}_{t+1}) + \gamma V(\boldsymbol{s}_t, \boldsymbol{\theta}_v) - V(\boldsymbol{s}_{t+1}, \boldsymbol{\theta}_v)] \nabla_{\boldsymbol{\theta}_v} V(\boldsymbol{s}_t, \boldsymbol{\theta}_v)$.

Algorithm 3 - Temporal Difference learning using function approximation

Input: the policy π to be evaluated
Input: a differentiable value function $V(s, \boldsymbol{\theta}_v)$, $V(terminal,\ \cdot) = 0$
Output: value function $V(s, \boldsymbol{\theta}_v)$
initialize value parameters $\boldsymbol{\theta}_v$ arbitrarily
for *each episode* **do**
  initialize state s
  for *each step of episode, state s is not terminal* **do**
    Perform a according to policy $\pi(\cdot\,|s)$
    Receive reward r' and state s'
    $\boldsymbol{\theta}_v \leftarrow \boldsymbol{\theta}_v + \alpha[r' + \gamma V(s', \boldsymbol{\theta}_v) - V(s, \boldsymbol{\theta}_v)]\nabla V(s, \boldsymbol{\theta}_v)$
    $s \leftarrow s'$
  end
end

α controls step size such that $\alpha > 0$. Adapted from Sutton and Barto [55].

Unfortunately, the combination of simple online RL algorithms with DNN was revealed to be fundamentally unstable, since the sequence of data encountered by an RL agent is non-stationary and RL updates are strongly correlated [37]. To circumvent the issue, multiple solutions have been proposed to stabilize learning, by storing data into an *experience replay* memory that can be batched [64], [65] or randomly sampled [66], [67] from different time steps. This aggregation tapers non-stationarity and decorrelates updates, but can only be applied to off-policy methods [37]. However, such construct led to significant performance improvements, as was the case of the Q-Learning algorithm implemented using FFN which was named Deep Q-Network (DQN), briefly explained next.

Q-Learning

Q-learning [68] is an off-policy version of the TD-algorithm, considered one of the most important achievements in RL [55]. This algorithm was developed as table-based but has also been updated to use function approximation. In its table-version form, the algorithm stores Q-values in a table updated at every step. The FFN version of Q-Learning, DQN, has shown remarkable results in most standard test environments. The algorithm learns the value of the policy by comparing the followed policy with the value expected from performing a greedy policy, expected to return maximum immediate reward [68], [55]. The algorithm keeps past-time transition triplets $(\boldsymbol{s}_t, \boldsymbol{a}_t, \boldsymbol{s}_{t+1})$ in batches which are provided as training inputs to the FFN. Training is performed at the end of the steps when the buffer holding the triplets is filled with a batch. Triplets are then released and the buffer is emptied. This process of training, called *experience*

replay [64], significantly improved learning quality and data-efficiency of the method compared to its predecessors.

Policy gradient methods

In contrast with Value-based methods, policy-based methods parameterize the policy as $\pi(a|s;\, \boldsymbol{\theta}_p)$ and update parameters $\boldsymbol{\theta}_p$ in order to arrive at optimal policy. In this section, we present two policy based methods, which also consider value function parameterization and thus introduce the actor-critic approach by estimating both value a policy.

REINFORCE algorithm

The REINFORCE [69] is a policy gradient method, that updates $\boldsymbol{\theta}$ in the direction of $\nabla_{\boldsymbol{\theta}_p} \log \pi(\boldsymbol{a}_t|\boldsymbol{s}_t, \boldsymbol{\theta}_p)\boldsymbol{r}_t$. Usually a baseline $b_t(\boldsymbol{s}_t)$, is subtracted from the reward to reduce the variance of gradient estimates making it stable yet unbiased, to yield the gradient direction $\nabla_{\boldsymbol{\theta}} \log \pi(\boldsymbol{a}_t|\boldsymbol{s}_t, \boldsymbol{\theta}_p)(\boldsymbol{r}_t - b_t(\boldsymbol{s}_t))$. Using $V(\boldsymbol{s}_t)$ as the baseline $b_t(\boldsymbol{s}_t)$, we have the advantage function $A(\boldsymbol{a}_t, \boldsymbol{s}_t) = Q(\boldsymbol{a}_t, \boldsymbol{s}_t) - V(\boldsymbol{s}_t)$, since $\boldsymbol{r}_t$ is an estimate of $Q(\boldsymbol{a}_t, \boldsymbol{s}_t)$. Thus, in this algorithm the advantage function is used in place of $Q(\boldsymbol{a}_t, \boldsymbol{s}_t)$. This approach can be viewed as an actor-critic architecture where the policy π is the actor and the baseline b_t is the critic [34], [70]. The following presents the pseudo code for REINFORCE algorithm for the case of immediate reward collection.

Algorithm 4 – REINFORCE algorithm

initialize policy parameters $\boldsymbol{\theta}_p$ and value parameters $\boldsymbol{\theta}_v$ arbitrarily
repeat
    generate an episode $\boldsymbol{s}_0, \boldsymbol{a}_0, \boldsymbol{r}_1, \ldots\ \boldsymbol{s}_{T-1}, \boldsymbol{a}_{T-1}, \boldsymbol{r}_{T-1}$, following $\pi(\cdot \mid \cdot, \boldsymbol{\theta}_p)$
    for *each step* t *of episode* $0, \cdots, T-1$ **do**
        $\boldsymbol{r}_t \leftarrow$ return from step t
        $\delta \leftarrow \boldsymbol{r}_t - V(\boldsymbol{s}_t, \boldsymbol{\theta}_v)$
        $\boldsymbol{\theta}_v \leftarrow \boldsymbol{\theta}_v + \beta\delta\nabla_{\boldsymbol{\theta}_v} V(\boldsymbol{s}_t, \boldsymbol{\theta}_v)$
        $\boldsymbol{\theta}_p \leftarrow \boldsymbol{\theta}_p + \alpha\gamma^t\delta\nabla_{\boldsymbol{\theta}_p} \log \pi(\boldsymbol{a}_t|\boldsymbol{s}_t, \boldsymbol{\theta}_p)$
    end
end

α and β control step size such that $\alpha > 0$, $\beta > 0$. Adapted from Sutton and Barto [55]

Actor-critic algorithm

In actor-critic algorithms, the critic updates action-value function parameters, and the actor updates policy parameters, in the direction suggested by the critic. An actor-critic algorithm learns both a policy and a value function, and the value function is used for

updating a state from subsequent estimates, to reduce variance and accelerate learning [55]. The implementation for the actor-critic algorithm is presented in Algorithm 5.

Algorithm 5 – Actor-Critic algorithm

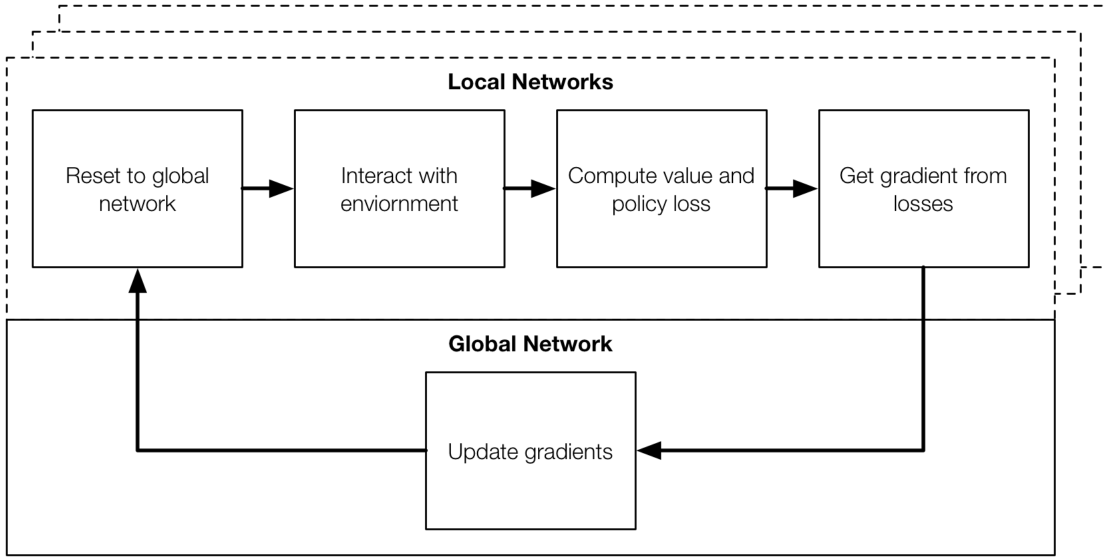
initialize policy parameters $\boldsymbol{\theta}_p$ and value parameters $\boldsymbol{\theta}_v$ arbitrarily
repeat
  initialize the first state of the episode s
  $I \leftarrow 1$
  for *s is not terminal* **do**
    Perform a according to policy $\pi(a|s, \boldsymbol{\theta}_p)$
    Receive reward r' and state s'
    $\delta \leftarrow r + \gamma V(s', \boldsymbol{\theta}_v) - V(s, \boldsymbol{\theta}_v)$, if s' is terminal $V(s', \boldsymbol{\theta}_v) \doteq 0$
    $\boldsymbol{\theta}_v \leftarrow \boldsymbol{\theta}_v + \beta\delta\nabla_{\boldsymbol{\theta}_v} V(s, \boldsymbol{\theta}_v)$
    $\boldsymbol{\theta}_p \leftarrow \boldsymbol{\theta}_p + \alpha I \delta \nabla_{\boldsymbol{\theta}_p} \log \pi(a|s, \boldsymbol{\theta}_p)$
    $I \leftarrow \gamma I$
    $s \leftarrow s'$
  end
end

α and β control step size such that $\alpha > 0$, $\beta > 0$. Adapted from Sutton and Barto [55]

Asynchronous Advantage Actor Critic

Using asynchronous methods, parallel actors interact with their own copy of the environments to stabilize training, and share access to a global network which accumulates gradient updates of the worker instance networks that interact with the environment copies, as show in Figure 9. Different from most deep learning algorithms, asynchronous methods can run on a single multi-core CPU. Particularly, when asynchronous actors are implemented using actor-critic and use the advantage estimation equation to estimate rewards [37], we arrive at the Asynchronous Advantage Actor Critic (A3C) algorithm, which has been shown to outperform DQN and other RL methods over many standardized environments [71], both in speed and policy performance.

Figure 9 - High level view of the asynchronous RL method



Dashed stacked layers represent multiple local network worker instances interacting with the global network.

Indeed, multiple actor learners running in parallel are likely to be exploring different parts of the environment which makes learning and experience more diverse [37]. In addition, different exploration policies can be used in each actor-learner to maximize this diversity. Thus, by running different exploration policies in different threads, the overall changes being made to the parameters by multiple actor-learners applying online updates in parallel are likely to be less correlated in time than a single agent applying online updates.

A3C maintains a policy $\pi(\boldsymbol{a}_t|\boldsymbol{s}_t, \boldsymbol{\theta}_p)$ and an estimate of the value function $V(\boldsymbol{s}_t, \boldsymbol{\theta}_v)$, being updated with a set of state, action and return tuples, after every t_{max} actions or when reaching a terminal state [37]. The gradient update is given by the expression,

$$\nabla_{\theta'_p} \log \pi\left(\boldsymbol{a}_t \middle| \boldsymbol{s}_t, \boldsymbol{\theta}'_p\right) A\left(\boldsymbol{s}_t, \boldsymbol{a}_t;\ \boldsymbol{\theta}_p, \boldsymbol{\theta}_v\right)$$

in which the term $\mathrm{A}\left(\mathbf{s}_\mathrm{t}, \mathbf{a}_\mathrm{t};\ \boldsymbol{\theta}_\mathrm{p}, \boldsymbol{\theta}_\mathrm{v}\right)$ stands for the advantage function shown next in the Equation 17.

$$A(\boldsymbol{a}_t, \boldsymbol{s}_t;\ \boldsymbol{\theta}_p, \boldsymbol{\theta}_v) = \sum_{i=0}^{k-1} \gamma^i \boldsymbol{r}_{t+i} + \gamma^k V(\boldsymbol{s}_{t+k}, \boldsymbol{\theta}_v) - V(\boldsymbol{s}_t, \boldsymbol{\theta}_v)$$

Equation 17- Advantage function estimation.
The term k is up-bounded by t_{max}. Adapted from [37].

While $\boldsymbol{\theta}_p$ and $\boldsymbol{\theta}_v$ function parameters are represented separately, in practice the parameters are shared by most layers of the DNN. Typically, such DNN has one softmax output for the policy $\pi(\boldsymbol{a}_t|\boldsymbol{s}_t, \boldsymbol{\theta}_p)$ and one linear output for the value function $V(\boldsymbol{s}_t, \boldsymbol{\theta}_v)$ with all non-output layers shared [37].

Furthermore, it has been shown that adding the entropy of the policy π to the objective function improves exploration by discouraging premature convergence to suboptimal deterministic policies [72]. The gradient of the policy cost function, including the entropy regularization term with respect to the policy parameters is given by Equation 18:

$$\nabla_{\theta'_p} log\, \pi(\boldsymbol{a}_t|\boldsymbol{s}_t, \boldsymbol{\theta}'_p)\left(\boldsymbol{r}_t - V(\boldsymbol{s}_t, \boldsymbol{\theta}_v)\right) + \beta \nabla_{\theta'_p} H\left(\pi(\boldsymbol{s}_t, \boldsymbol{\theta}'_p)\right)$$

Equation 18 - Regularized policy gradient cost function using advantage estimation.
H stands for the entropy function. β is a regularization term such that $\beta \in [0,1]$. Adapted from [37].

where H is the entropy function. The hyperparameter β controls the regularization term. The A3C algorithm can be implemented as shown in Algorithm 6.

Using this mechanism, *experience replay* can be discarded because this way parallel actors can work on-policy using different policies and thus stabilize learning, which was the aim of using *experience replay* in the DQN training algorithm that worked off-policy [37]. Furthermore, using multiple parallel actor-learners leads to a reduction in training time roughly linear to the number of parallel actor-learners [37], and enables usage of on-policy RL methods over DNN.

The overall architecture for the A3C is represented in Figure 10.

Algorithm 6 - Asynchronous Advantage Actor Critic (A3C) algorithm

Initialize shared global parameter vectors $\boldsymbol{\theta}_v$ and $\boldsymbol{\theta}_p$ arbitrarily
Initialize thread parameter vectors $\boldsymbol{\theta}'_v$ and $\boldsymbol{\theta}'_p$ arbitrarily
Initialize global shared counter $T = 0$
Initialize thread step counter $t \leftarrow 1$
for $T \leq T_{max}$ **do**
    Reset gradients $d\boldsymbol{\theta}_p \leftarrow 0,\ d\boldsymbol{\theta}_v \leftarrow 0$
    Synchronize thread-specific parameters $\boldsymbol{\theta}'_p = \boldsymbol{\theta}_p$ and $\boldsymbol{\theta}'_v = \boldsymbol{\theta}_v$
    $t_{start} = t$
    Get state $\boldsymbol{s}_t$
    for $\boldsymbol{s}_t$ is not terminal **or** $t - t_{start} < t_{max}$ **do**
        Perform $\boldsymbol{a}_t$ according to policy $\pi(\boldsymbol{a}_t|\boldsymbol{s}_t, \boldsymbol{\theta}'_p)$
        Receive reward $\boldsymbol{r}_t$ and state $\boldsymbol{s}_{t+1}$
        $t \leftarrow t + 1$
        $T \leftarrow T + 1$
    end
    $R = \begin{cases} 0 & \text{for terminal } \boldsymbol{s}_t \\ V(\boldsymbol{s}_t, \boldsymbol{\theta}_v) & \text{for non} - \text{terminal } \boldsymbol{s}_t \end{cases}$
    for $i \in \{t-1, \dots, t_{start}\}$ **do**
        $R \leftarrow \boldsymbol{r}_i + \gamma R$
        // Accumulate gradients with respect to $\boldsymbol{\theta}'_p$
        $d\boldsymbol{\theta}_p \leftarrow d\boldsymbol{\theta}_p + \nabla_{\theta'_p} \log \pi(\boldsymbol{a}_i|\boldsymbol{s}_i, \boldsymbol{\theta}'_p)(\boldsymbol{r}_i - V(\boldsymbol{s}_i, \boldsymbol{\theta}'_v)) + \beta \nabla_{\theta'_p} H\left(\pi(\boldsymbol{s}_i, \boldsymbol{\theta}'_p)\right)$
        // Accumulate gradients with respect to $\boldsymbol{\theta}'_v$
        $d\boldsymbol{\theta}_v \leftarrow d\boldsymbol{\theta}_v + \nabla_{\theta'_v}(\boldsymbol{r}_i - V(\boldsymbol{s}_i, \boldsymbol{\theta}'_v))^2$
    end
    Perform asynchronous updates of $\boldsymbol{\theta}_p$ using $d\boldsymbol{\theta}_p$ and $\boldsymbol{\theta}'_v$ using $d\boldsymbol{\theta}'_v$
end

adapted from Mnih et al [37].
H is the entropy function. β is a regularization term such that $\beta \in [0,1]$.

Concluding remarks

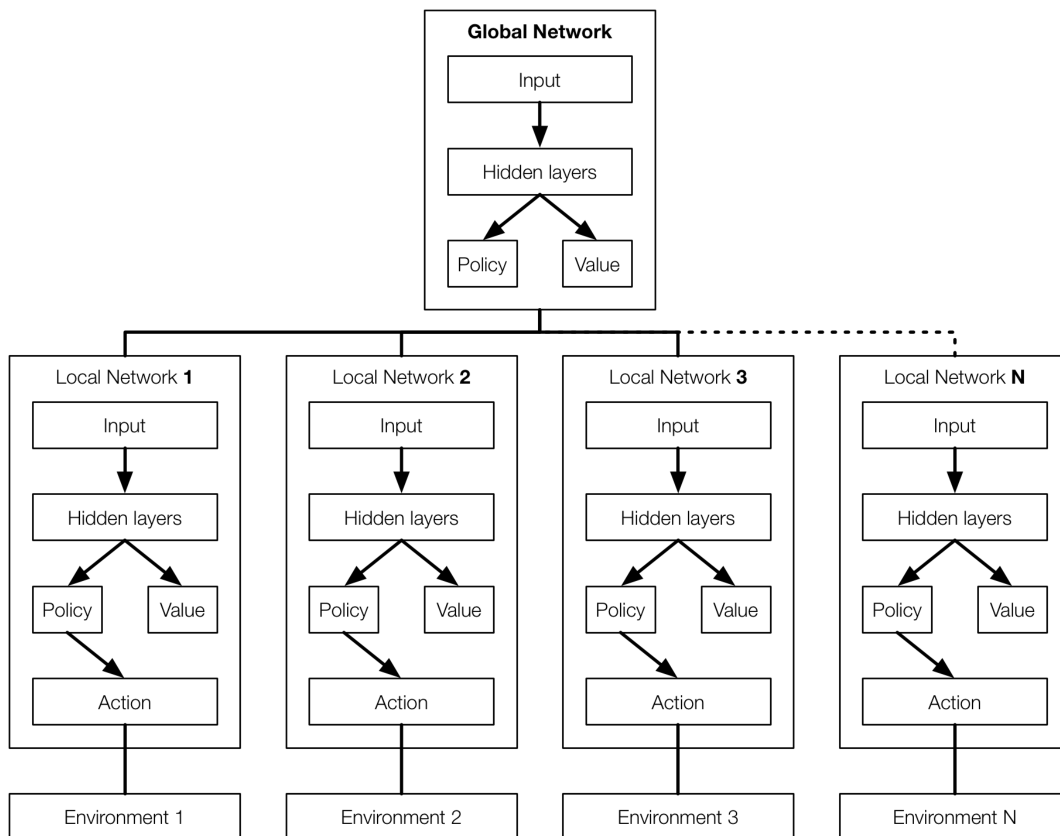
RL methods seem particularly suitable for scheduling problems due to their exploration and exploitation nature, that renders them capable of acquiring experience from interaction with the environment and use it to decide on future actions. The A3C model seems particularly interesting for application in patient appointment scheduling problems in Primary Care:

- It is an on-policy model-free model that was demonstrated to outperform other RL methods namely DQN in standard test environments [37] which the previous state of the art method;
- It overcomes the problem of learning stability by employing multiple actors and global network that accumulates gradients from the actors [37];

- The training speed scales roughly linearly with the number of parallel actors [37];
- It surpassed DQN within half of training time [37];
- It effectively runs on non-specialized hardware [37].

Taking into consideration the ground-breaking results achieved by the A3C framework, we selected it as a candidate RL method to learn to schedule appointments in Primary Care, as it will be presented in Chapter 4 and Chapter 5.

Figure 10 – Asynchronous Advantage Actor Critic (A3C) topology



Each network serves as basis for policy and value gradient computation accumulated by the global network. Updated to the global network are performed asynchronously. The composition of the hidden layers of each network instance has the same arbitrary structure, upon which a softmax layer is used for obtaining policy estimations and a scalar output is used for the value estimation. In case of the local worker networks, the policy is plugged to a final action layer that outputs the desired action to perform on its own environment copy.

The dashed connector in the right indicates connection to an arbitrary number of local networks.

Chapter 4

Scheduling Framework

To study and develop RL agents that can perform scheduling tasks for Primary Care, it is worthwhile to develop a platform that allows the creation of such environments, development and configuration of multiple agents, and assess the effect of environment parameters, RL agent hyperparameters and structure parameters in learning performance. Benchmarking of the algorithm is also paramount to assess its fitness. Such a platform can dramatically decrease iteration time to develop algorithms for different scheduling tasks.

In this Chapter, we present the conceptual and functional framework developed for the creation and simulation of RL agents in patient scheduling environments, illustrated with screenshots from the implemented software system. Finally, software development and architecture design considerations of the solution are discussed.

Environment modelling

A scheduling environment consists of the following components:

- Schedule slots;
- Composable task attributes that can be assigned to tasks.

Schedule configuration

Schedule slots are placeholders for tasks at a given time. The array of slots that compose a schedule is given by the following variables:

- Number of slots per day;
- Number of days on a week;
- Number of working weeks;
- Slot duration.

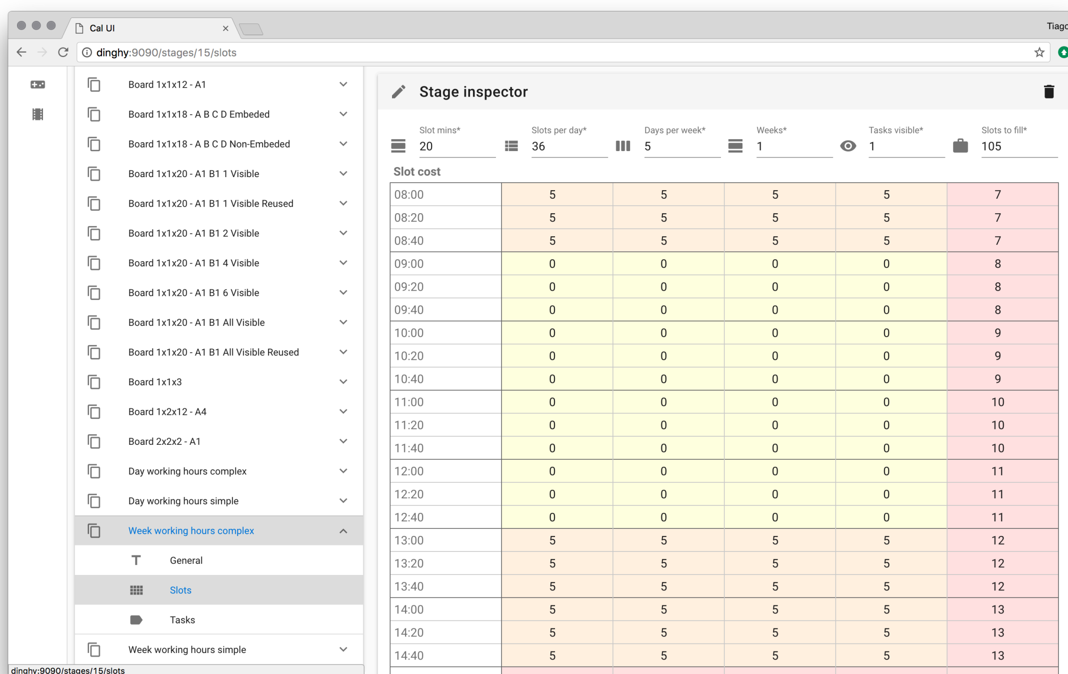
While the first three parameters define the schedule structure, in the context of this work the last one is used simply to build a user-friendly timetable representation of the slots. In addition, each slot has a cost attribute, so that slots with lower cost may be chosen with precedence over the higher cost slots. This conveys useful information

when deciding which slot to schedule. Finally, the schedule requires two additional variables:

- Number of tasks to schedule;
- Number of visible tasks to schedule – Number of scheduling tasks visible to the agent. The type of scheduling process depends on this parameter:
    - When equal to 1, it is a unit process;
    - When between 1 and number of tasks to schedule, it is a periodic process;
    - When equal to the number of tasks to schedule, it is a single batch process.

Figure 11 illustrates the user interface for configuration of the mentioned parameters.

Figure 11 – Slot environment configuration



Left – List of available environments and access to general, slot and task configuration;
Right – Slot configuration inspector. Slot cost is color-coded to aid in configuration by the user.

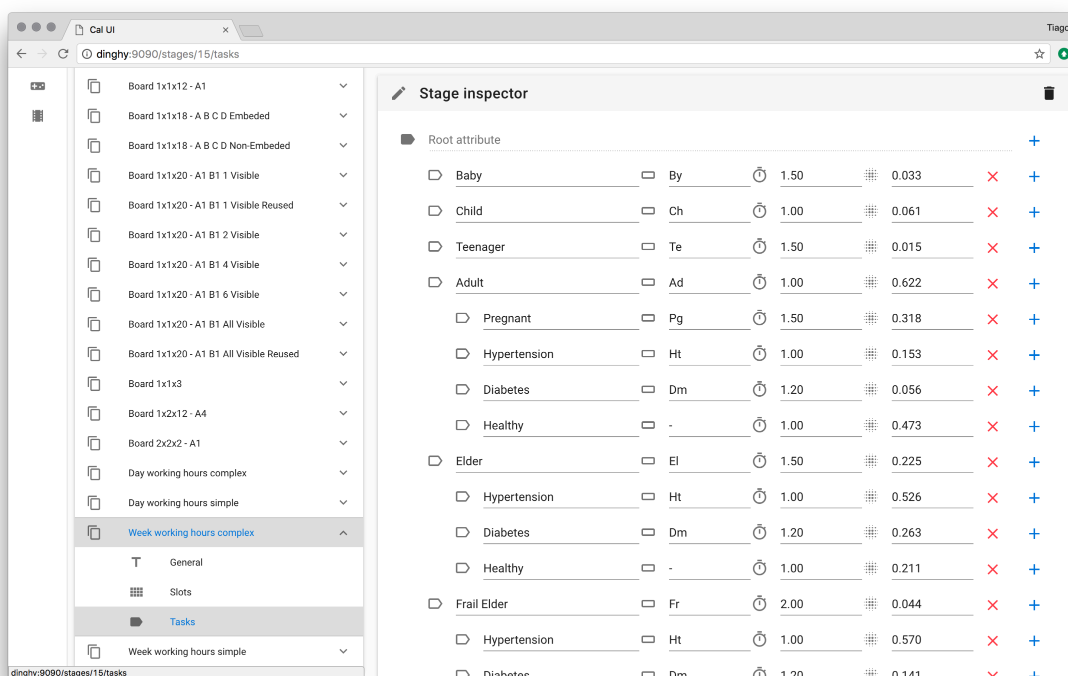
Task attribute configuration

For a given environment, tasks of arbitrary nature should be defined. To achieve this requirement, an attribute tree was created. Each tree node holds the following information:

- **Slot factor** – Controls how many slots a task with the given attribute requires;
- **Sampling probability** – Probability of picking an attribute given the parent attribute. In the context of Primary Care clinics, this probability should be estimated from appointment attribute frequencies on a yearly or monthly basis;
- **Child attributes** – Child attribute nodes under a given node;
- **Name and Abbreviation** – Used as task labels in the task definition editor and in schedule representations, respectively.

All attribute nodes are under a root node with probability 1. The algorithm then randomly picks a child attribute given their conditional probabilities and recursively repeats the process on its child attributes. The collected sequence of attributes defines a task. Figure 12 shows the hierarchical nature of task attributes and its parameters.

Figure 12 - Task environment definition



Attribute nodes hold name, abbreviation, slot factor and conditional probability as well as child nodes.
Left – List of available environments and access to general, slot and task configuration; **Right** – Task configuration inspector.
Attribute Icons: **Tag** - Attribute name; **Rectangle** – Abbreviation; **Clock** – Slot factor; **Pattern** – Conditional probability.

Considering the task attributes presented in Figure 12, the following example tasks can be sampled into the task buffer: Baby, Adult | Pregnant, Elder | Diabetes. The slot factor for a task is given by the product of slot factors of its attributes.

Reward configuration

The environment is expected to reward agent actions. Since reward plays a central role in the training of RL agents, the ability to tweak the parameters that govern reward is crucial for a good environment and RL agent. Several factors were taken in consideration to calculate the reward according to the requirements presented in Chapter 2:

- **Base Reward** – Reward for picking and empty slot;
- **Overlap Penalty** – Penalty for picking a filled slot;
- **Common Neighborhood Factor** – Multiplies the bonus for grouping similar tasks. The bonus corresponds to the fraction of common attributes between the scheduled task and its previous neighbor. For example, if task A|B follows task A|Z, it will receive a bonus of 0.5.
- **Partial Overlap Factor** – Multiplies the penalty received when there is a partial overlap between tasks. As an example, when a task that requires 2 slots is scheduled immediately before another task, it will receive a penalty of 1.

At each training episode, the agent interacts with the environment for a predefined number of steps. An additional factor controls the maximum number of steps that can be taken before the episode finishes:

- **Max Step Factor** – Multiples by the number of slots to fill, to define how many actions the agent can take before the training episode finishes.

This indirectly controls reward since there will be uncollected reward if an agent fails to schedule all tasks in the task buffer. An episode will finish when all tasks are scheduled, or when the maximum number of steps is reached. At the end of an episode, the reward is calculated according to Algorithm 7. In any other step, reward is 0, such that RL agents refrain from strategies of non-terminal states that could otherwise yield high reward.

Algorithm 7 - Scheduling environment reward computation

```
total_reward = 0
if not is_episode_complete: return total_reward
for slot in slot_buffer:
  if is_slot_empty(slot): continue
  reward = base_reward
  reward = reward – get_cost_for_slot(slot)
  overlap_with_previous_task = compute_overlap_with_previous_task_for_slot(slot)
  reward = reward – partial_slot_overlap_factor × overlap_with_previous_tas
  similarity_with_previous_slot = compute_similarity_with_previous_slot(slot)
  reward = reward – common_neighbour_factor × similarity_with_previous_slot
  total_reward = total_reward + reward
total_reward = total_reward – count_stale_slots()
```

Reward is always 0 except in the last step of each episode.
Written in Python pseudo-code.

Additional parameter configuration

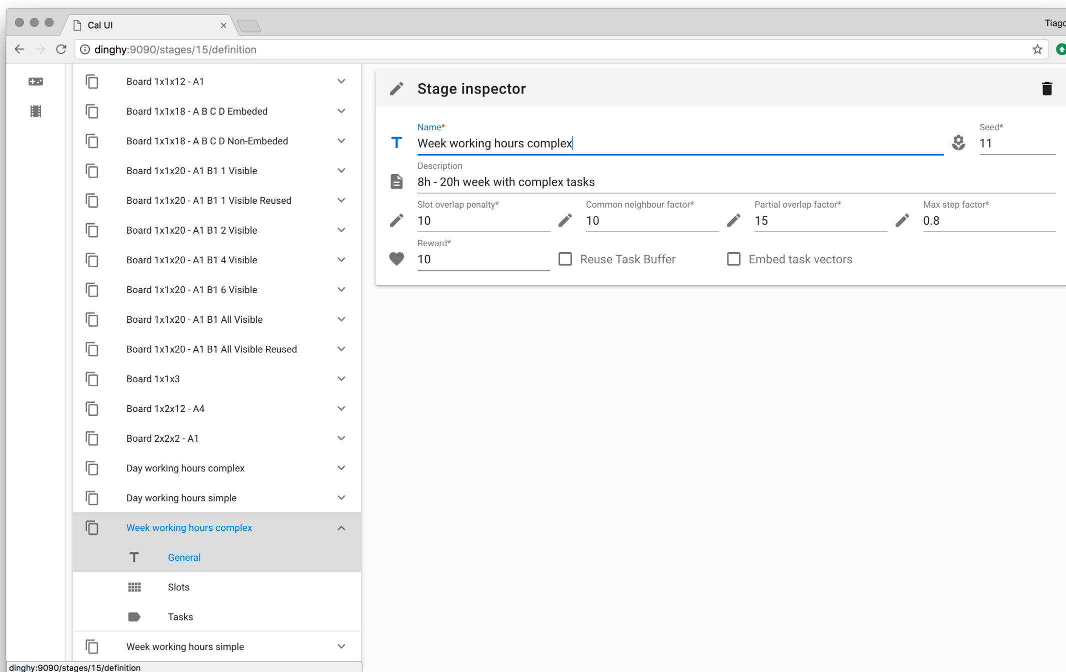
Figure 13 presents two additional parameters that control environment representation:

- Reuse task buffer - Determines whether the same task buffer is considered across training episodes and environment instances, thus becoming fixed during training;
- Embed task vector - Controls whether a task representation should be reduced when it takes more than 3 dimensions to represent. This is useful because task space may be very sparse, since attribute trees can be very different.

Efficiently representing tasks

To illustrate to problem of representing task attributes, let M be an example task attribute space for task attributes A and B. Such attributes can be encoded using a two-dimensional vector as [1,0] and [0,1], respectively. The general rule is to consider a space composed of one binary dimension per each task attribute. However, since task attributes are nested, part of the space represents invalid tasks. Let N be another task attribute space that originates the tasks $A|A1$, $B|B1$, $B|B2$. It will require 5 dimensions to represent attributes A, $A1$, B, $B1$ and $B2$. However, because combinations such as $A|B1$ and $A|B2$ are invalid, some regions of space N are obsolete. By enabling task vector embedding, the task vectors will be transformed using Principle Component Analysis (PCA) into 3 dimensional vectors, that preserve as much variance as possible. This way, tasks $B|B1$ and will still be more alike to $B|B2$ than to $A|A1$ while saving space.

Figure 13 - Reward configuration and additional parameters



Left – List of available environments and access to general, slot and task configuration;
Right – Reward and additional parameter configuration inspector.

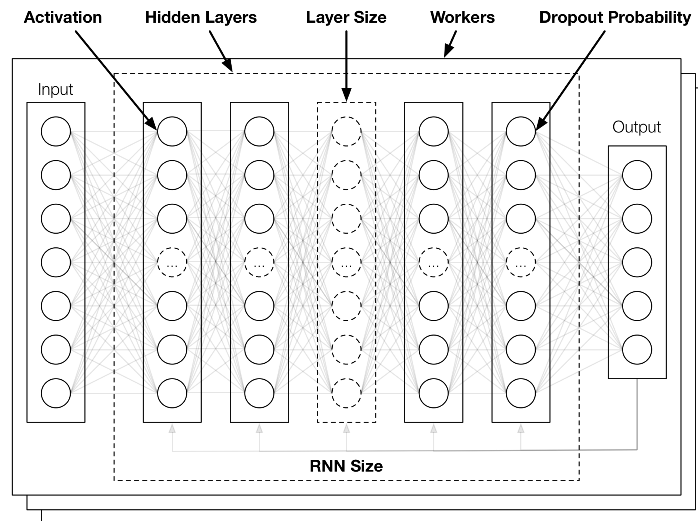
Agent configuration

The system provides the following set of agents:

- **Human** – Passes control to the user interface so that a human agent can interact;
- **Random** – Picks a random action from the action space;
- **Algorithm** – Picks one from the set of actions the yield the highest reward;
- **A3C** – RL agent that implements the A3C algorithm [37] using TensorFlow [73].

In the case of the A3C agent, additional parameters can be defined to build and tune the neural network that serves as basis for the agent. These parameters enable exploration of different network architectures for A3C as shown in Figure 14.

Figure 14 - A3C neural network architecture and role of structure parameters



Input and output vector dimensions are matched to observation and action vector dimensions.
RNN – Recurrent neural network

Configuration of network structure parameters

The available structure parameters are the following:

- **Activation Function** – Pick from implementations available on the TensorFlow library [73], namely ReLU, Exponential Linear Unit (ELU), Concatenated ReLU (CreLU), Sigmoid, TanH, among others;
- **Workers** – Controls the number of local networks;
- **Dropout probability** – Controls the dropout probability of keeping nodes at any layer. A value of 0 means dropout is not applied;
- **Hidden layers** – Controls the number of hidden layers in each network;
- **Layer size** – Controls the number of hidden units on each layer;
- **RNN size** – Controls the number of cells on the RNN component of the network;

Configuration of network hyperparameters

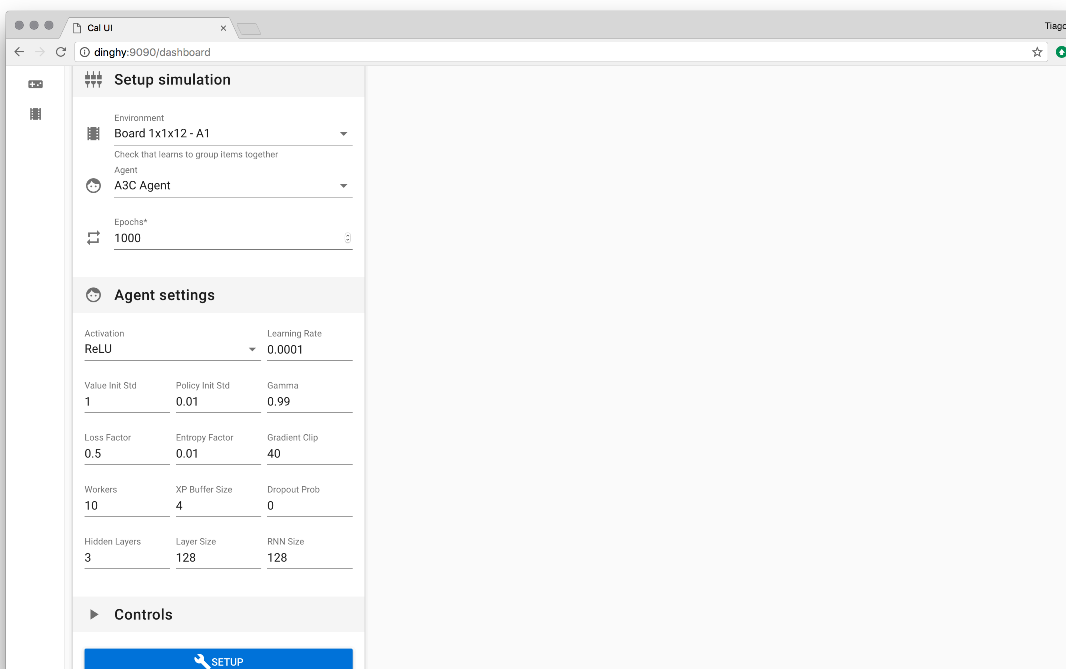
A set of hyperparameters that govern learning are also exposed:

- **Learning rate** – The learning rate considered by optimizer RMSProp [74];
- **Standard deviations** for **value** and **policy** network initializations:
- **Gamma** – Controls the weight of reward discounting;
- **Experience buffer size** – The size of the batch used to train the RL agent;

- **Loss regularization factor** – Controls the weight of the loss regularization term;
- **Entropy regularization factor** – Controls the weight of the entropy regularization term. It affects the degree for preferring exploration over exploitation;
- **Gradient clip** – Clips the gradients to the specified amount to avoid big steps that can lead to gradient overshooting and learning instability.

These parameters allow the creation and tweaking of networks of very diverse nature. Figure 15 presents the interface that allows such parameterization.

Figure 15 - Dashboard for simulation setup



Top left - Environment, agent and episode setup. **Bottom left** – Hyper-parameter and structure settings for the A3C agent.

Simulation

The system allows running simulations for any given environment and agent.

Max reward heuristic

At the beginning of each training episode the system resets the environment and uses the heuristic in Algorithm 8 to find a good approximation of the best reward that can be collected when scheduling all the tasks in the task buffer.

Algorithm 8 - Near optimal solution heuristic

```
sorted_task_buffer = sort(task_buffer)
for task in sorted_task_buffer:
  slot_indexes = pick_slot_indexes_that_yield_highest_reward_for_task(task)
  slot_index = slot_indexes[0]
  schedule_task_at_slot_index(slot_index, task)
```

Sorting the task buffer increases the odds of finding an optimal solution.
Written in Python pseudo-code.

Because the full task buffer is sorted prior to scheduling, the probability that any other better solution is found decreases, and thus considering only an item at a time likely provides a better approximation of the optimal reward than the implementation of the algorithm agent.

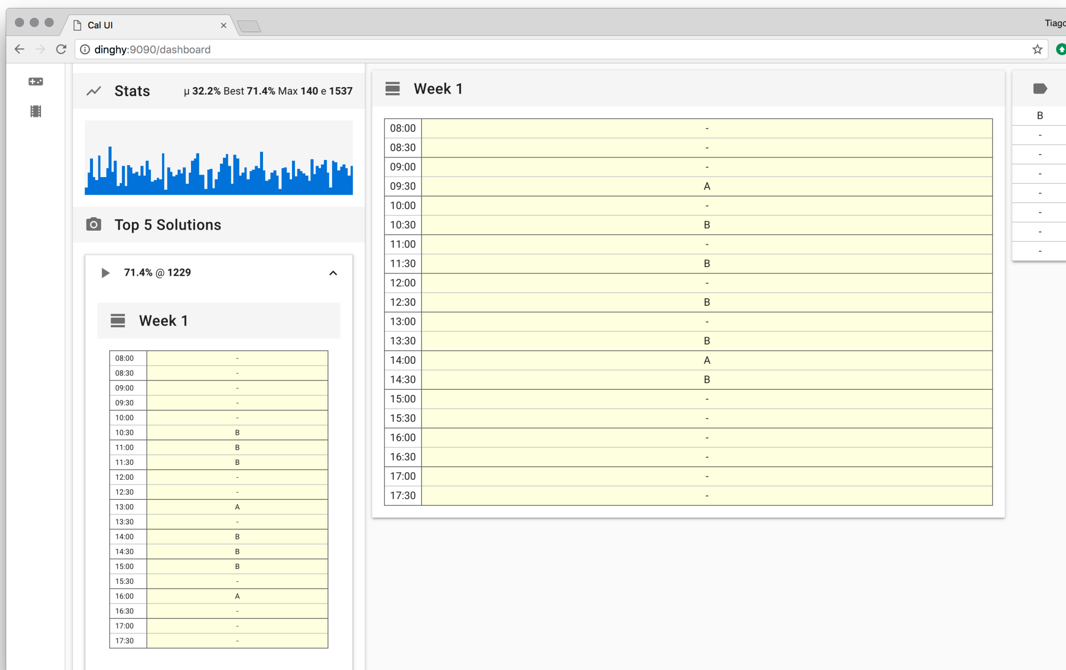
This implementation models the problem as a single batch process presented in Chapter 2. This algorithm is meant to approximate an upper bound to the best possible reward, even though it depends on information that may not be available to the agent. The heuristic is run every iteration since the reward depends on the tasks to schedule.

During runtime, the system provides the following simulation information in real-time:

- **Relative reward for each episode** – Percentage between the reward achieved by the agent and the best computed reward using the previously presented heuristic. Values for the last 100 episodes are plotted in a bar char;
- **Relative reward running average** – Measures the average relative reward for the last 100 episodes;
- **Best relative reward** – Best relative reward since the beginning of training;
- **Max reward** – Maximum reward achieved by the heuristic for the current episode;
- **Episode count** – Current episode number.

Figure 16 presents the interface created to convey the simulation status during runtime. In addition, the system displays the schedule and the tasks to schedule, and updates it in real-time with the actions taken by the agent. At the beginning of each episode the task buffer filled with tasks created from random sampling task attributes from the attribute tree. The visible tasks are displayed in a column on the right side of the screen. Then at each step, the first task on the task buffer is moved from the buffer to the selected slot.

Figure 16 - Dashboard with running simulation



Left - Reward statistics and the top 5 solutions found by the agent; **Center** - Current simulation state, updated in real-time with the agent actions; **Right** - Task buffer with visible tasks to schedule.

The screen updates at each time step so that actions can be tracked. In case of the human agent, the action to be performed is indicated by clicking in the desired slot.

Software architecture considerations

The application was implemented as a software-as-a-service, designed to run both on local machines and distributed cloud systems. The application is composed as a set of 3 micro-services, composed of 4 main elements:

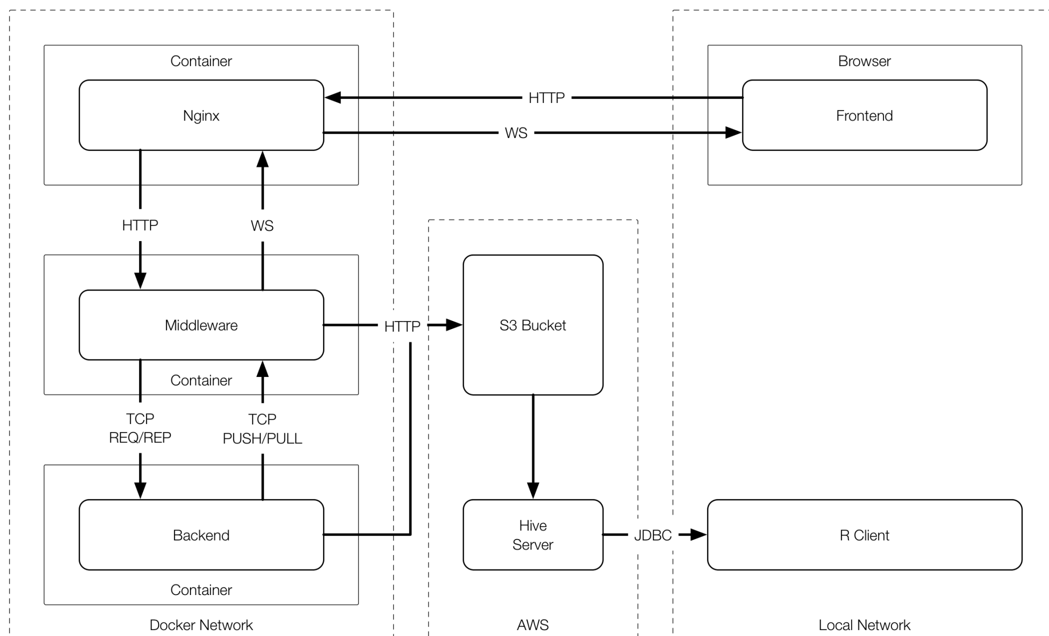
- Backend to run simulations;
- Frontend user interface for environments creation and simulation setup;
- Middleware for routing between multiple backends and frontends;
- Simulation data collection pipeline for *post hoc* analysis.

The components communicate over Hypertext Transfer Protocol (HTTP), Web Socket (WS) and Transmission Control Protocol (TCP), such that actions are dispatched from the frontend to the corresponding backend, and the backend broadcasts simulation data to the frontend for real-time rendering of the simulation.

Each service is deployed using Docker technology [75], which allows applications to be run inside containers – light weight Linux virtual environments that contain all code and dependencies necessary to install, mount and run services in local or cloud platforms.

The overall architecture is depicted in Figure 17. Components are next described in detail.

Figure 17 - High level system architecture



AWS – Amazon Web Services; **HTTP** – Hypertext Transfer Protocol; **JDBC** - Java Database Connectivity; **PUSH/PULL** – Push pull socket pair; **REQ/REP** – Request reply socket pair; **S3** – Simple storage service; **TCP** – Transmission control protocol; **WS** – Web Socket.

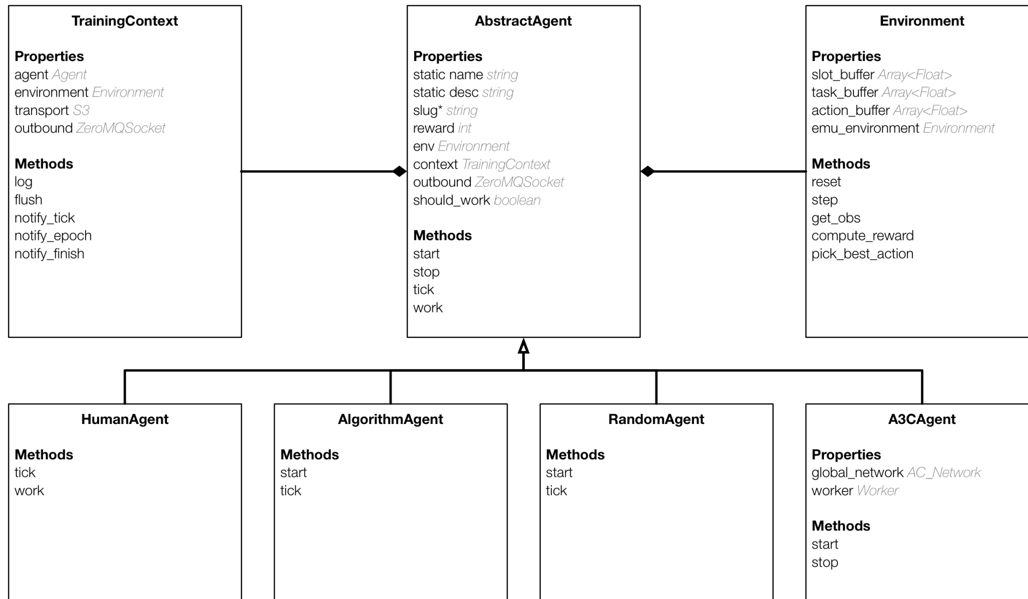
Backend

The backend service was written in the Python language. It exposes services through a TCP interface implemented using the ZeroMQ library [76]. This interface listens for requests to configure and run simulations through one socket, and broadcasts simulation updates over another socket into the middleware service.

The backend uses the environment and agent configurations to instantiate `Environment` and `Agent` classes that are kept in memory during each simulation. Thus, the backend becomes stateful. The agents are implemented as subclasses of an `AbstractAgent`. The available agent classes can be listed through a service. Additional agents can be created and registered for usage in simulations by simply sub-classing

the `AbstractAgent` Class. The main classes and their relationships to run the simulations are presented in Figure 18.

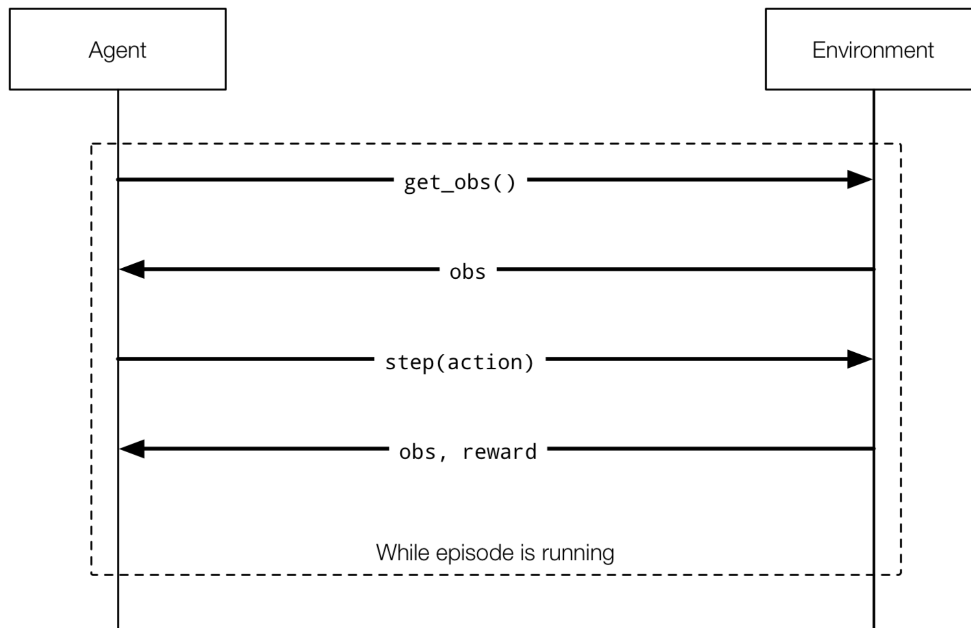
Figure 18 - High level simulation classes and relationships



Grey text indicates property data types.

The simulation loop passes control to the `agent`, that iteratively calls an `environment.step(action)` method on the `environment` and returns a reward. The `agent` then requests the new environment state through `environment.get_obs()`. This state and the reward can then be used by the agent to decide which action to take next. This loop will run until all tasks are scheduled or the maximum number of actions is taken, as previously described. At the end of the episode, `environment.reset()` is called and a new episode is started. The simulation will execute for as many episodes as configured. The simulation loop executes as shown in Figure 19.

Figure 19 - Structure of the interaction between agent and environment



This interaction takes place at every step of the simulation.

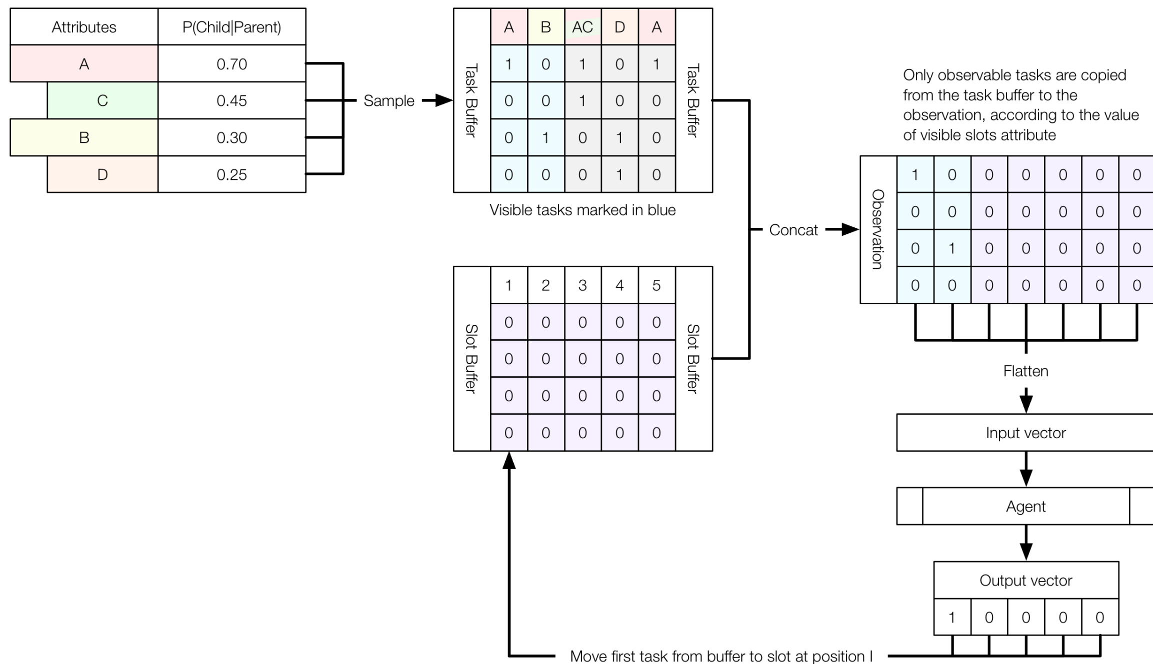
The environment state that is passed as observation is a vector that results from the concatenation of a matrix representing the schedule slots and the visible portion of the task buffer. The resulting matrix is flattened to a vector that is passed to the agent to compute the desired action. That action is then fed into the environment to update the schedule slot buffer and the task buffer. The size of these matrices depends on the schedule factors that affect schedule size, and on the size of the vectors needed to represent the tasks.

The internal representation of these buffers, their combination into and observation and the effect of taking an action in the environment is presented in Figure 20.

RL Agent Implementation

The RL Agent was implemented using the A3C algorithm [37] previously presented using TensorFlow library [73]. The agent networks are built dynamically by considering the structure definition and hyperparameters passed as configuration.

Figure 20 - Internal representation of environment data



Environment internal representation of task buffer, slot buffer and action vectors and the dynamics of their update and transformation into observation fed to agents. The size of the vector used to represent tasks varies according to the number of possible attributes. If the embed task vector option is enabled, the vector size is fixed at 3 and vector values change from binary to floats.

Frontend

The frontend application allows environment creation, setup and visualization of all aspects of the simulations. It is implemented using Type Script [77] and the Vue.js framework [78]. It makes use of a state store that centralizes all possible state transitions in the web interface. This architecture makes it easy to separate business logic from component rendering and allows data to flow from the middleware services to the components that are updated as needed, resulting in a fluid user experience and reactive user interface that can accommodate large simulation data flows with a low memory footprint. This way the simulations can be inspected in real-time, best solutions can be collected and algorithm behavior can be analyzed as it progresses. The frontend service is exposed on a container running Nginx webserver [79]. It responds to requests from the browser requiring frontend static assets, and proxies remaining data requests to the middleware service. It also pipes WS messages to the middleware service.

Middleware

The middleware is responsible for bridging communications between the backend and frontend services. As illustrated in Figure 17, it converts HTTP requests coming from the frontend into TCP requests to the backend, and converts TCP requests from the

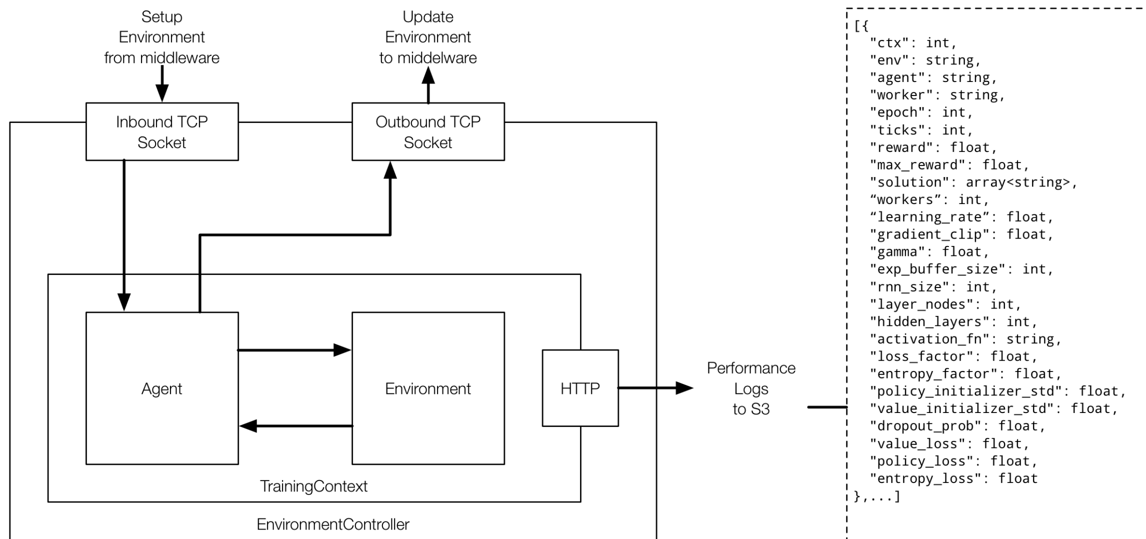
backend into WS messages that are pushed into the frontend for real-time refresh of the simulation visualization. This service is also written in Python and makes use of the Flask framework to handle HTTP and WS traffic [80]. The middleware exposes additional endpoints for environment configuration persistence into a Simple Storage Service (S3) bucket on AWS Web Services. S3 is a distributed filesystem service that allows long term data storage in the AWS cloud. This form of persistence easily allows the simulator to be run at any scale and at any infrastructure and reuse existing configuration.

Data collection

Finally, the system also employs a mechanism for data collection for *post hoc* analysis. This is relevant because the volume of data produced during the simulation cannot be stored by the machine running the simulation, nor by the frontend service because it can easily outgrow available memory resources on local machines. Thus, it becomes relevant to collect data from across all machines running simulations into a centralized store, that can later be used to run analytical queries and data analysis. The data pipeline is implemented as depicted in Figure 21.

Using this architecture pattern, data is continuously sent to the centralized store and freed from memory in each machine running the simulation. Data is stored in `json` format which can easily be queried using Hive technology that exposes a Hive Query Language (HQL), a query language similar to Structured Query Language (SQL), used in relational databases. This provides a simple interface to run queries over structured data files without the need to comply to a predefined database schema [81]. This software is made available as a service through AWS Athena, and can feed on data available on AWS S3. Thus, the backend periodically dumps simulation data to S3, that can later be queried and analyzed on any client that supports the service Java Database Connection (JDBC) drivers. For the purpose of this work, data was analyzed using the R language [82] as depicted in Figure 17.

Figure 21 - Data collection pipeline



HTTP – Hypertext transfer protocol; **S3** – Simple storage service; **TCP** – Transmission control protocol.

Infrastructure specifications

All services ran as single Docker container instances on the Dinghy [83] virtual machine via Virtual Box [84] with shared access to 2Gb RAM and 1 CPU Core of a MacBook Pro (13-inch, 2016) with a 2.9 GHz Intel Core i5 and 8 GB 2133 MHz LPDDR3.

Final remarks

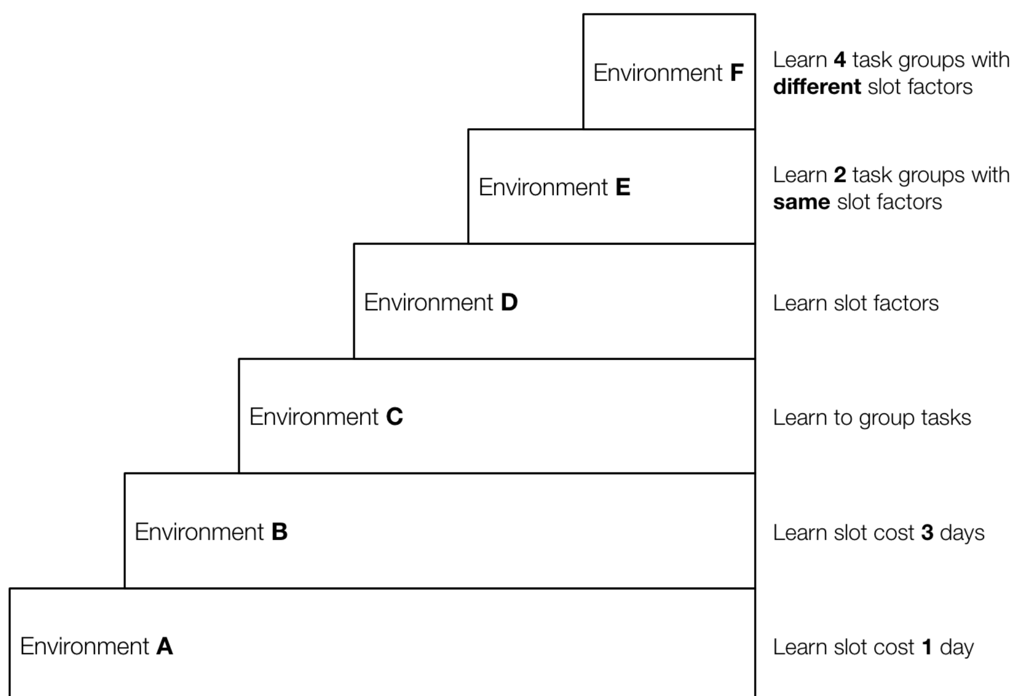
By leveraging this infrastructure, it became possible to easily create, maintain and upgrade a set of environments over which to develop and test RL agents. This platform enables the desired rapid development iteration as depicted Figure 4, thus making research and development faster. Environments and RL agent performance is presented next.

Chapter 5

Scheduling environments and agent performance

A set of basic environments was created to study whether RL agents can learn the rules governing each environment. In this chapter, we present a set the environments and the results of candidate RL agents. Environments were created considering incremental state spaces, in which reward rules are incrementally added. They serve to test whether RL agents can learn the underlying rules and are depicted in Figure 22. In addition, a real single day environment was created and used to test algorithm performance.

Figure 22 - Dummy environment goals



Challenges of incremental difficulty are implemented on each environment.

For each environment, the following agents were run:

- **Random agent** ran for 1000 episodes, to estimate the average reward that can be obtained by chance;

- **Algorithm agent** ran for 1000 episodes, to estimate average reward that can be obtained for a given task attribute distribution by picking the best slot at each step;
- **RL agents** with different network structure ran in each environment until convergence. The following parameters were fixed for all agents:
    - Activation function - ReLU
    - Reward discount gamma - 0.99
    - Policy weight initialization standard deviation - 0.1
    - Value weight initialization standard deviation - 1.0
    - Loss regularization factor - 0.5
    - Entropy regularization factor - 0.1

As environments grew in complexity, the best performing agent from an environment was taken to the next environment, and served as template to create other agents. Summaries of the dummy environments are presented in Table 1. Dummy environments shared a set of fixed parameters shown in Figure 23.

Table 1 - Dummy environment feature summary

Schedule		Tasks		Slot attributes		States		
Code	**Shape**	**Fill**	**Vis**	**Count**	**Factor ≠ 1**	**Possible**	**Optimal**	**Ratio**
A	1x1x3	2	1	1	no	7	1	1.43×10^{-1}
B	1x3x3	2	1	1	no	46	1	2.17×10^{-2}
C	1x1x12	4	1	1	no	794	3	3.78×10^{-3}
D	1x2x12	4	1	1	yes	12 951	40	3.09×10^{-3}
E	1x2x12	4	1	2	no	62 103	12	1.93×10^{-4}
F	1x2x12	8	1	4	yes	176 986 730	4	2.26×10^{-8}

Count – Number of different task attributes; **Factor ≠ 1** – Whether there are attributes with a slot factor other than 1; **Fill** – Number of tasks to fill; **Optimal** – Number of optimal states. There may be more than one number of optimal states, depending on the attribute distributions; **Possible** – State space size; **Shape** – nº weeks × nº days/week × nº slots/day;
Vis – Number of visible tasks; **%** - Ratio between possible and optimal states expressed as percentage of possible states.

Figure 23 - Dummy environment configuration parameters

Slot overlap penalty*	Common neighbour factor*	Partial overlap factor*	Max step factor*
0	2	2	3

Reward*: 1 — ☐ Reuse Task Buffer — ☑ Embed task vectors

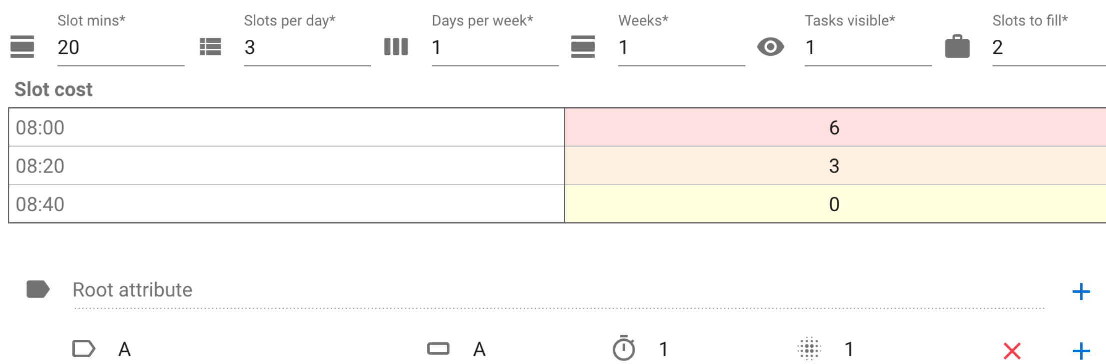
These parameters are shared across all dummy environments.

In the following sections, we will briefly explain the rationale of each environment and present the hyperparameters and structure for the candidate RL agents. Firstly, the results of the simulation were analyzed quantitatively, for which the performance results for the RL agent are shown in 25, 50 and 75 percentile plots for the reward obtained per batch of episodes. In addition, plots are presented for value loss, policy loss and entropy loss of the RL agents. For other agents and for the environment max reward heuristic, the mean reward attained is plotted as labelled horizontal lines. Secondly, a qualitative analysis of the results follows through the presentation of specific solutions. Finally, each section ends with a brief discussion of the overall performance for each environment.

Dummy environment A

This environment is meant to assess whether the RL Agent can learn to prioritize scheduling according to slot cost in a minimal state space. The environment configuration is presented in the Figure 24 and candidate RL agents are presented in Table 2.

Figure 24 - Dummy environment A configuration



Bottom fields: **Tag** - Attribute name; **Rectangle** – Abbreviation; **Clock** – Slot factor; **Pattern** – Conditional probability.

Table 2 - RL Agent A1, A2 and A3 configuration for dummy environment A and B

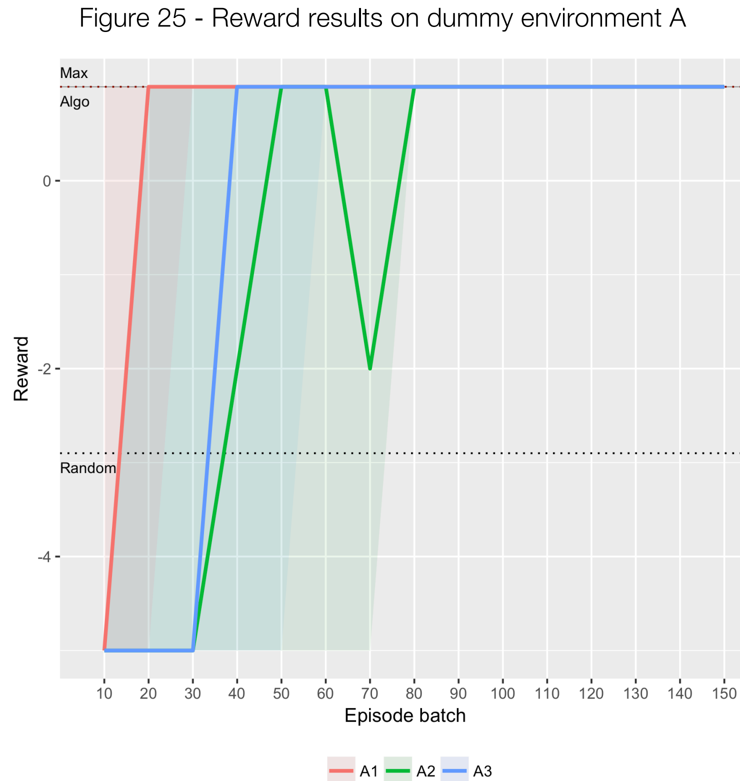
			Structure			
Name	α	**Clip**	**Workers**	**Hidden**	**Nodes**	**RNN**
A1	.0100	40	2	2	8	**0**
A2	.0100	40	2	2	8	**1**
A3	.0100	40	2	2	8	**2**

Varying parameters between the candidate RL agents are displayed using bold typeface.
α – Learning Rate; **Clip** – Gradient clip; **Hidden** – Number of hidden layers;
Nodes – Number of nodes per hidden layer; **RNN** – RNN size; **Workers** – Number of workers;

Quantitative analysis

RL agents learned to perform perfectly, reaching the maximum possible reward of 1. The reward by random action is set close to -1. Figure 25 shows that around episode 80 all agents performed perfectly. The agent A1 learned the rule faster than agent A2 and A3.

Figure 25 - Reward results on dummy environment A



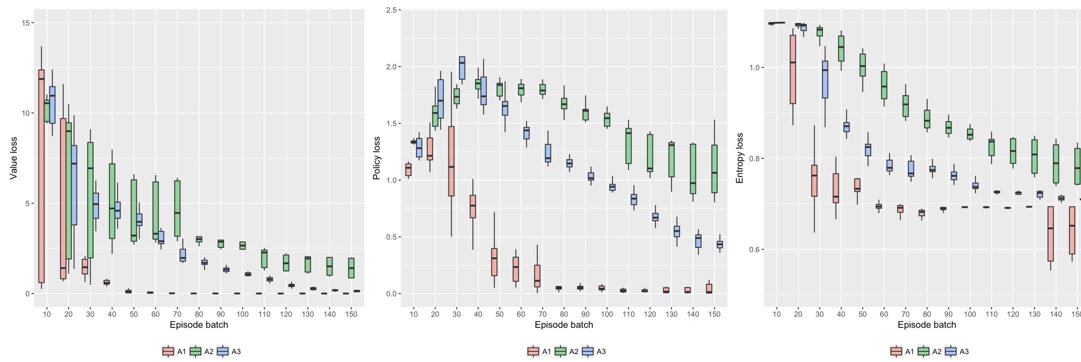
Reward per training epoch for RL agents on environment A.
Lines plot percentile **50** and the shaded areas plot the space between the percentile **25** and **75** for each RL agent.

Figure 26 shows that the losses for value, policy and entropy have steadily fallen across sessions for all agents with agent A1 converging faster. Agent A2 converged poorly, showing high value loss variability until episode 70, time at which A1 had already converged. The charts show that there was still room for further minimizing losses for agents A2 and A3.

Qualitative analysis

All agents arrived at the optimal solution depicted in Figure 27, in which the cheapest slots are preferred over the most expensive one.

Figure 26 - RL Agent losses on dummy environment A



From left to right, value, policy and entropy losses across episodes for RL agents in dummy environment A.

Figure 27 - Optimal solution for environment A

08:00	-
08:20	A
08:40	A

All agents applied optimal policy consistently.

Discussion

The agents successfully learned the intended rule. However, agents A2 and A3 took more time to converge, and did not minimize losses steadily. It seems that adding LSTM cells to the neural network to model a recurrent component is not helpful to this type of task. In fact, the decision at any step depends exclusively in the current observed state, and not in past states. As such the problem is formulated as a MDP. For that reason, considering past states to act becomes a hindrance.

Dummy environment B

This environment is also meant to assess whether the RL Agent can learn to prioritize scheduling according to slot cost on a larger state space. It poses the additional challenge of putting tasks together to bring additional bonus, but that will not make up for higher slot costs. The expected result is that the agent prefers the two cheapest slots presented in Figure 24. Candidate RL agents are presented in Table 2.

The RL Agents developed for environment A were used to solve environment B.

Figure 28 - Dummy environment B configuration

Slot mins*	Slots per day*	Days per week*	Weeks*	Tasks visible*	Slots to fill*
15	3	3	1	1	2

Slot cost

08:00	3	3	3
08:15	0	6	0
08:30	3	3	3

Root attribute

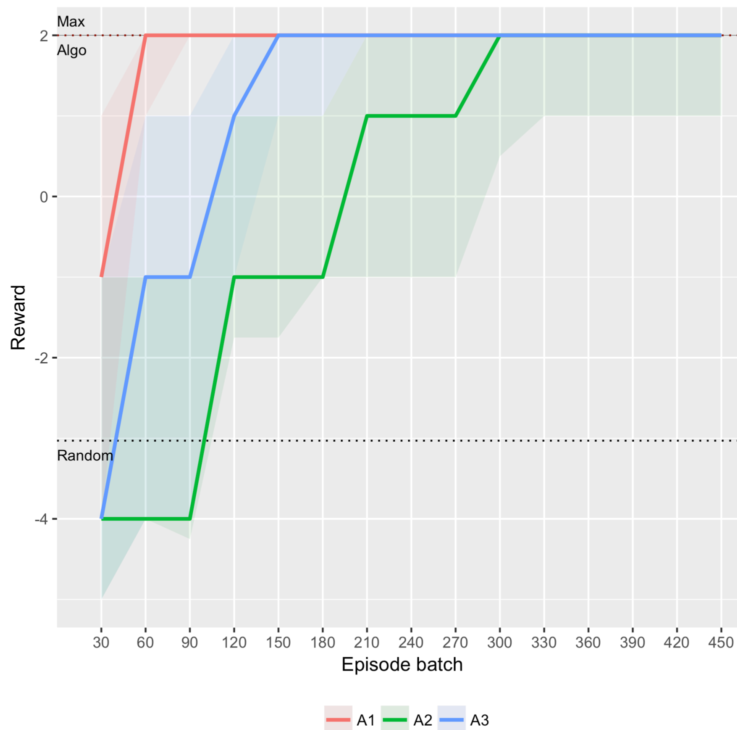
Tag	Rectangle	Clock	Pattern
A	A	1	1

Bottom fields: **Tag** - Attribute name; **Rectangle** – Abbreviation; **Clock** – Slot factor; **Pattern** – Conditional probability.

Quantitative analysis

Both three implementations learned to perform well, with the learning converging around episode 330. Figure 29 shows that the agent A1 converged faster than the remaining agents and kept a consistent behavior ever since, in contrast with the other agents that took suboptimal actions frequently.

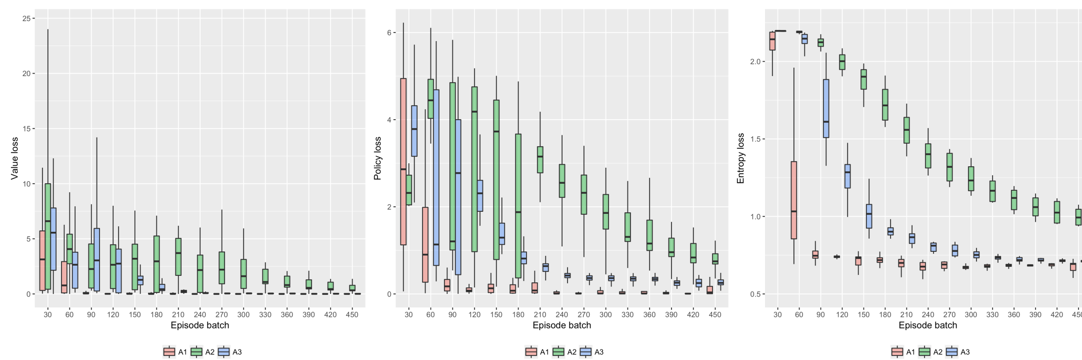
Figure 29 - Reward results on dummy environment B



Reward per training epoch for RL agents on environment B.
Lines plot percentile **50** and the shaded areas plot the space between the percentile **25** and **75** for each RL agent.

Figure 30 shows that losses were also minimized more effectively by agent A1. Value and policy loss approached 0 for agent A1 around episode 90, meaning that the network rapidly became very confident on how to value actions and which action to take. Agent A3 displayed a similar but slower evolution, while agent A2 was unable to minimize losses as much. Entropy flattening around episode 240 for agents A1 and A3, indicates that the agents took the exploitation action known to maximize reward with very high frequency.

Figure 30 - RL Agent losses on dummy environment B



From left to right, value, policy and entropy losses across episodes for RL agents in dummy environment B.

Qualitative analysis

All agents arrived at the optimal solution depicted in Figure 31, in which the two cheaper slots are preferred over the remaining slots.

Figure 31 - Optimal solution for environment B

08:00	-	-	-
08:15	A	-	A
08:30	-	-	-

All agents applied optimal policy consistently.

Discussion

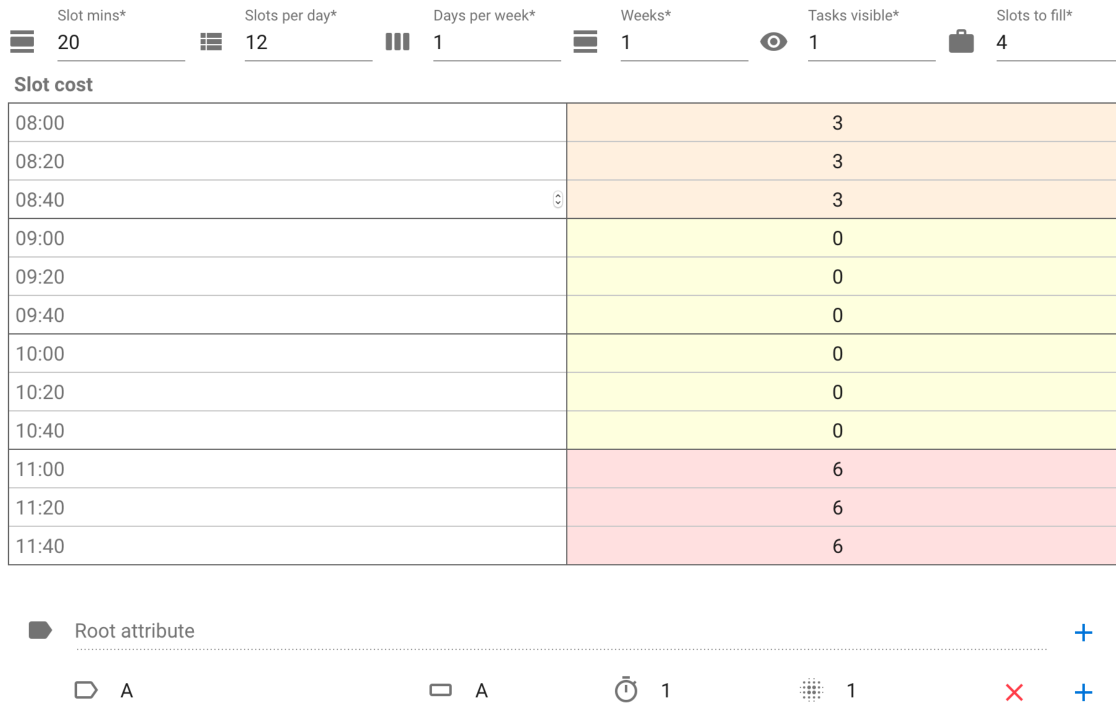
Environment B was more challenging than environment A since the state space increased substantially, as well as the reward interval. Agents learned the underlying environment rule, however the LSTM cells still seem a hindrance for performance.

Dummy environment C

Environment C was developed to check whether the agent can learn to group task together to get higher reward. The agents were expected to group the tasks without

spacing in any sequence of cheap slots. Figure 24 presents dummy environment C structure, and candidate RL agents are presented in Table 3.

Figure 32 - Dummy environment C configuration



Bottom fields: **Tag** - Attribute name; **Rectangle** – Abbreviation; **Clock** – Slot factor; **Pattern** – Conditional probability.

Table 3 - RL Agent A1, A4 and A5 configuration for dummy environment C

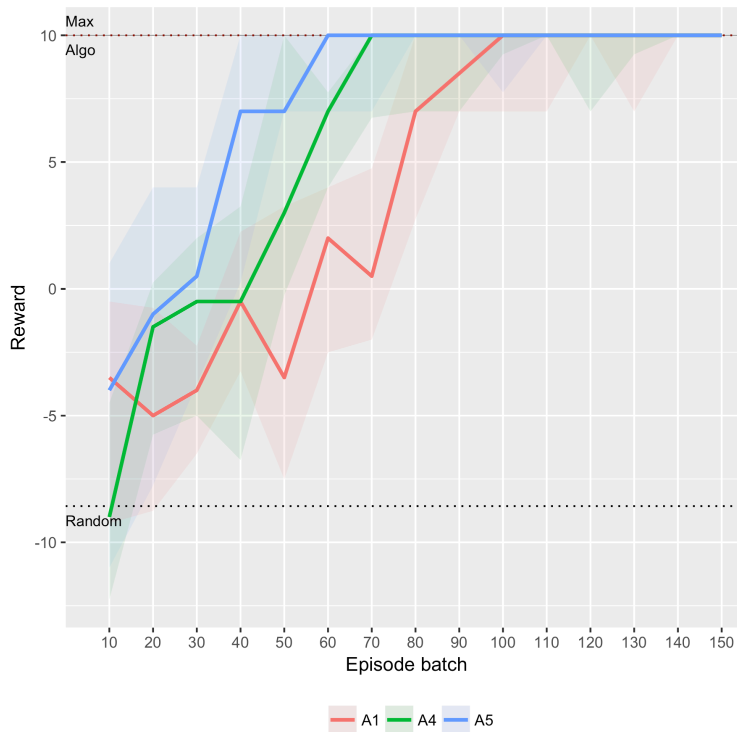
			Structure			
Name	α	Clip	Workers	Hidden	Nodes	RNN
A1	.0100	40	**2**	**2**	8	0
A4	.0100	40	**2**	**3**	8	0
A5	.0100	40	**3**	**2**	8	0

Varying parameters between the candidate RL agents are displayed using bold typeface.
α – Learning Rate; **Clip** – Gradient clip; **Hidden** – Number of hidden layers;
Nodes – Number of nodes per hidden layer; **RNN** – RNN size; **Workers** – Number of workers;

Quantitative analysis

Agents learned to perform optimally spanning from random reward levels to maximum reward in around 140 episodes as shown in Figure 33. Agent A4 and A5 learned the rule faster that agent A1.

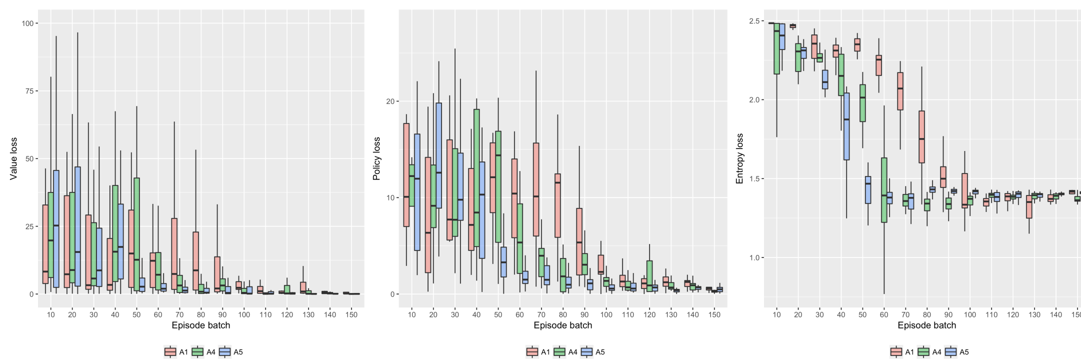
Figure 33 - Reward results on dummy environment C



Reward per training epoch for RL agents on environment C.
Lines plot percentile **50** and the shaded areas plot the space between the percentile **25** and **75** for each RL agent.

Figure 34 shows that the 3 agents minimized value, policy and entropy in a similar fashion. Initial loss variance is higher in agent A5 due to the increased number of workers. Agent A5 minimized losses faster than the other agents and led to the development and application of a consistent policy for environment C.

Figure 34 - RL Agent losses on dummy environment C



From left to right, value, policy and entropy losses across episodes for RL agents in dummy environment C.

Qualitative analysis

Figure 35 depicts the evolution of solution for agent A5 which progress in total reward along episodes. Interestingly, having arrived at different policies that return maximal reward, agents stick to that single optimal policy and consistently perform it.

Figure 35 - Solution evolution on dummy environment C for agent A5

08:00	-
08:20	A
08:40	-
09:00	-
09:20	-
09:40	A
10:00	-
10:20	A
10:40	-
11:00	-
11:20	-
11:40	A

08:00	-
08:20	A
08:40	A
09:00	A
09:20	A
09:40	-
10:00	-
10:20	-
10:40	-
11:00	-
11:20	-
11:40	-

08:00	-
08:20	-
08:40	-
09:00	A
09:20	A
09:40	A
10:00	A
10:20	-
10:40	-
11:00	-
11:20	-
11:40	-

All agents applied optimal policy consistently.
Left – solution at episode 10; **Center** – solution at episode 40; **Right** - solution at episode 70.

Discussion

The networks were kept small despite a substantial increase in state space from environment B to environment C. Note that in this environment, the size of input and output vectors (17 and 16, respectively) became larger the number of nodes per hidden layer (8). This increase in state space accounted for an increase in the number of training epochs until good solutions were found. Agent A5 effectively explored the environment by amassing varied observations brought by its 3 worker networks, leading to the discovery of an optimal rule faster, which is in accordance other reports [37]. It is interesting to note that varying experience boosts learning more than adding additional hidden layers. This means that when a network has a structure large enough to represent the optimal mapping functions from observation to action, increasing observation variability through multiple workers has substantial benefit over increasing network structure. Finally, even though there was more than one optimal solution, it was sufficient to learn only one policy to achieve consistently perfect score, and so the networks stick to a single policy.

Dummy environment D

This environment was created to assess whether the agents learn to space tasks by the amount specified by the slot factor parameter. It the case of this environment, it was expected that the agents in Table 4 learned to place 3 tasks with 1 slot spacing in a day, and 1 task in the other day, while avoiding the costly slots depicted in Figure 36.

Figure 36 - Dummy environment D configuration

Slot mins* 20 | Slots per day* 12 | Days per week* 2 | Weeks* 1 | Tasks visible* 1 | Slots to fill* 4

Slot cost

08:00	3	3
08:20	3	3
08:40	3	3
09:00	0	0
09:20	0	0
09:40	0	0
10:00	0	0
10:20	0	0
10:40	0	0
11:00	6	6
11:20	6	6
11:40	6	6

Root attribute +

A A 1 1 × +

Bottom fields: **Tag** - Attribute name; **Rectangle** – Abbreviation; **Clock** – Slot factor; **Pattern** – Conditional probability.

Table 4 - RL Agent A5, A6 and A7 configuration for dummy environment D

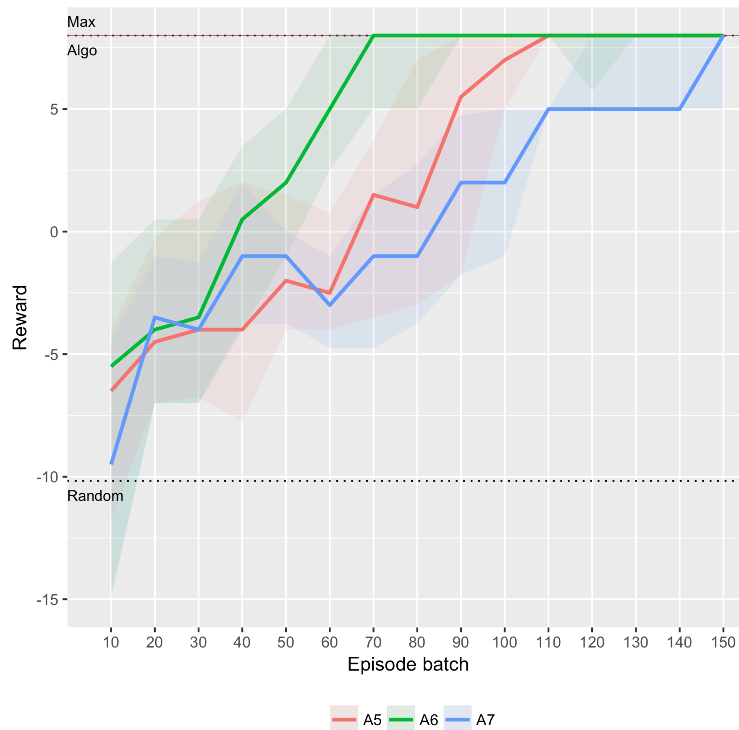
			Structure			
Name	α	**Clip**	**Workers**	**Hidden**	**Nodes**	**RNN**
A5	.0100	40	3	**2**	**8**	0
A6	.0100	40	3	**2**	**16**	0
A7	.0100	40	3	**3**	**8**	0

Varying parameters between the candidate RL agents are displayed using bold typeface.
α – Learning Rate; **Clip** – Gradient clip; **Hidden** – Number of hidden layers;
Nodes – Number of nodes per hidden layer; **RNN** – RNN size; **Workers** – Number of workers;

Quantitative analysis

Figure 37 shows that the three agents learned the rule well. Agent A6 reached optimal action by episode 90, while agents A5 and A7 reached optimal by episode 120 and 150. Agent A7 displayed less consistent performance when compared to the other agents. Figure 38 shows that while the three agents minimized value and policy in a similar fashion, agent A7 entropy loss decreased at a slower pace, displaying a significant drop around episode 110. Similar abrupt drops happened to the other agents sooner.

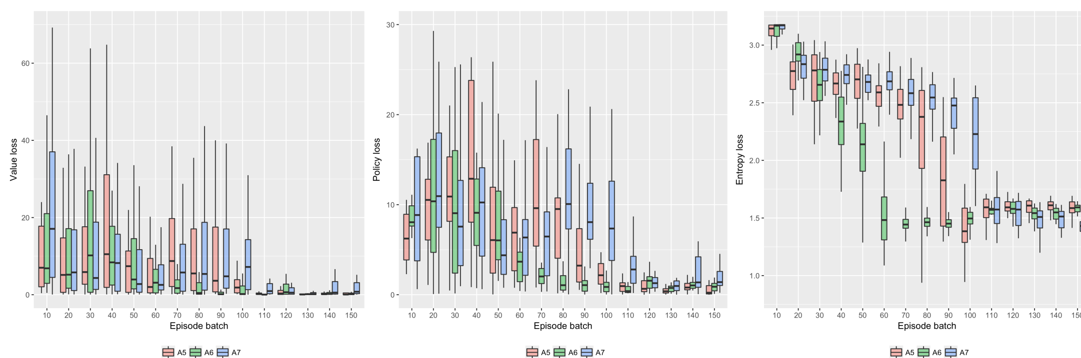
Figure 37 - Reward results on dummy environment D



Reward per training epoch for RL agents on environment D.
Lines plot percentile **50** and the shaded areas plot the space between the percentile **25** and **75** for each RL agent.

These drops were also observed in value and policy losses, for agents A5 and A7 that displayed abrupt drops in value loss variance in 100 and 110 respectively. Agent A6 displayed a smoother minimization of both value and policy losses, which accounts for the steady growth in reward per epoch it presented until convergence around episode 90.

Figure 38 - RL Agent losses on dummy environment D



From left to right, value, policy and entropy losses across episodes for RL agents in dummy environment D.

Qualitative analysis

The 3 agents arrived at optimal solutions shown in Figure 39, scheduling the 4 tasks in adjacent slots as intended. Again, agents displayed preference for a single optimal solution as soon as it was discovered, and consistently applied it across episodes.

Figure 39 - Optimal solutions on dummy environment D

08:00	-	-
08:20	-	-
08:40	-	-
09:00	A	-
09:20	-	A
09:40	A	-
10:00	-	A
10:20	-	-
10:40	-	-
11:00	-	-
11:20	-	-
11:40	-	-

08:00	-	-
08:20	-	-
08:40	-	-
09:00	-	-
09:20	-	A
09:40	A	-
10:00	-	A
10:20	-	-
10:40	-	A
11:00	-	-
11:20	-	-
11:40	-	-

08:00	-	-
08:20	-	-
08:40	-	-
09:00	A	-
09:20	-	-
09:40	A	-
10:00	-	A
10:20	-	-
10:40	-	A
11:00	-	-
11:20	-	-
11:40	-	-

All agents applied optimal policy consistently.

Discussion

Small hidden layer structure was enough to learn the rule. Interestingly, in environment D the number of possible states increased beyond the number of iterations before reaching the optimal rule. In fact, the complete state space is 12 951 and all agents consistently applied optimal policy, after visiting less than 1200 states with repetition. The addition of more nodes to the hidden layers benefitted the network more than incrementing the number of hidden layers, which implies that the increasing number of layers created excess complexity in the network that was unnecessary to represent the problem and required additional learning episodes to minimize error. The agents kept sticking to the single optimal state that they found, since for this environment it was still good enough to learn a single policy to reach maximum reward.

Dummy environment E

This environment sought to assess whether the agent learns to group the tasks by attribute. The expected behavior is to schedule tasks A and B in order in a single group. The size of the task buffer becomes relevant in this environment since different tasks may be observed, but only one will be known at a time, according to the unit process definition presented in Chapter 2. Figure 40 presents dummy environment E structure, and candidate RL agents are presented in Table 4. This environment poses additional challenges to the agents due following factors increasing state complexity:

- The observation vector doubles to represent the 2 task attributes A and B;

- The agent must learn to distinguish task attributes A and B;
- The task buffer is populated at random and all sequences are equally likely.

Figure 40 - Dummy environment E configuration

Slot mins*	Slots per day*	Days per week*	Weeks*	Tasks visible*	Slots to fill*
20	12	2	1	1	4

Slot cost

08:00	3	3
08:20	3	3
08:40	3	3
09:00	0	0
09:20	0	0
09:40	0	0
10:00	0	0
10:20	0	0
10:40	0	0
11:00	6	6
11:20	6	6
11:40	6	6

Root attribute

Tag	Rectangle	Clock	Pattern
A	A	1	0.5
B	B	1	0.5

Bottom fields: **Tag** - Attribute name; **Rectangle** – Abbreviation; **Clock** – Slot factor; **Pattern** – Conditional probability.

Table 5 - RL Agent A6, A8 and A9 configuration for dummy environment E

			Structure			
Name	α	Clip	Workers	Hidden	Nodes	RNN
A6	**.0100**	40	**3**	**2**	16	0
A8	**.0095**	40	**3**	**3**	16	0
A9	**.0090**	40	**4**	**2**	16	0

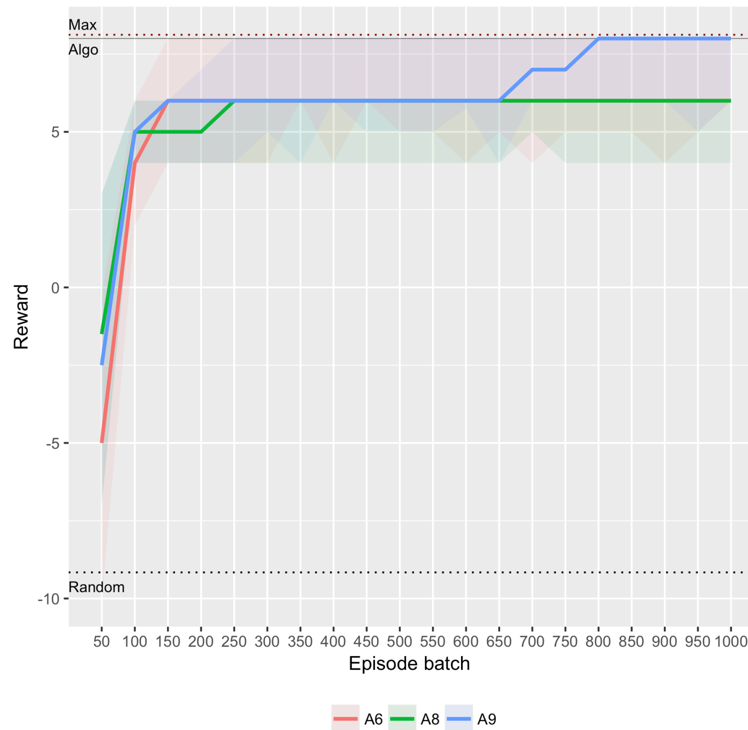
Varying parameters between the candidate RL agents are displayed using bold typeface.
α – Learning Rate; **Clip** – Gradient clip; **Hidden** – Number of hidden layers;
Nodes – Number of nodes per hidden layer; **RNN** – RNN size; **Workers** – Number of workers;

Quantitative analysis

Figure 41 shows small differences between the max reward heuristic and algorithm agent, which are due to, the fact that sequential slot ordering may not always deliver the best result for all possible task samples, and thus introduce a small variation. Regarding

the RL agents, they took significantly more episodes to converge, with only agent A9 having achieved consistently optimal results.

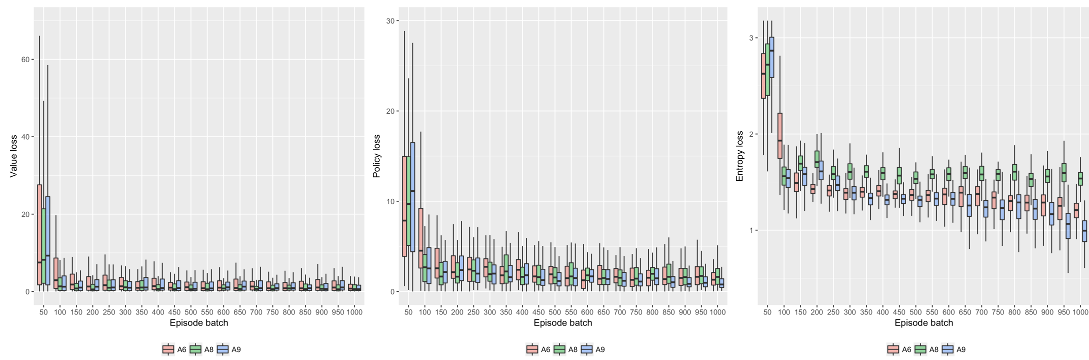
Figure 41 - Reward results on dummy environment E



Reward per training epoch for RL agents on environment E.
Lines plot percentile **50** and the shaded areas plot the space between the percentile **25** and **75** for each RL agent.

Figure 42 demonstrates that value and policy losses were largely minimized by the agents shortly after the beginning of training, however entropy suffered an abrupt drop in the first 150 episodes, after which it flattened for agent A8, and kept slowly dropping for agents A6 and A9, the latter in a steeper fashion. The fact that entropy losses keep dropping indicates that there is potential room for reward improvement by increasing certainty for exploitation rather than exploration, which explains the variability indicated by percentiles 25 and 50, specially for agent A9.

Figure 42 - RL Agent losses on dummy environment E



From left to right, value, policy and entropy losses across episodes for RL agents in dummy environment E.

Qualitative analysis

Figure 43 demonstrates possible optimal solutions found by the RL agents. Note that in this environment the agent must learn more than one optimal policy because optimal solution varies with sampled tasks. An optimal policy for a task buffer composed of A tasks, may not be the same for a mixed buffer. The order in which tasks appears also affects the possible optimal solutions given the tasks that were already scheduled.

Figure 43 - Examples of optimal on dummy environment E

08:00	-	-
08:20	-	-
08:40	-	-
09:00	-	-
09:20	-	-
09:40	-	B
10:00	-	B
10:20	-	B
10:40	-	B
11:00	-	-
11:20	-	-
11:40	-	-

08:00	-	-
08:20	-	-
08:40	-	-
09:00	-	-
09:20	B	A
09:40	B	A
10:00	-	-
10:20	-	-
10:40	-	-
11:00	-	-
11:20	-	-
11:40	-	-

08:00	-	-
08:20	-	-
08:40	-	-
09:00	A	-
09:20	A	A
09:40	A	-
10:00	-	-
10:20	-	-
10:40	-	-
11:00	-	-
11:20	-	-
11:40	-	-

To consistently maximize reward, the agent A9 learned more than one optimal policy.

Discussion

The complexity of environment increased substantially with comparison to the previous ones. Learning rates of A8 and A9 agents needed adjustment to avoid gradient overflows that appeared for higher learning rates. Once again it was demonstrated that network structure of layers much smaller that the observation vector can accurately map observations to optimal actions, which indicates that underlying rules are still simple, despite the increasing state space complexity which reached 62 103 possible states.

Furthermore, the addition of a fourth worker was necessary to achieve higher learning rates and stabilize learning, as was demonstrated by agent A9. The optimal policies were applied in more than half of the cases. This is explained in one hand by the additional variability introduced by random task buffer sampling which requires the network to learn more than one optimal solution, and in the other, by the additional challenge in minimizing entropy losses which depend on the complexity of the state space. Despite this fact, if such networks would be applied in production environment, where the agent will solely exploit what it learnt, rather than switching between exploitation and exploration in hopes to refine its strategy, a more consistent performance would be observed.

Dummy environment F

This final dummy environment tests learning of prior rules to schedule task of 4 types. The environment is presented in Figure 44 and the candidate agents in Table 6.

The following are the additional challenges posed by the environment:

- Task buffer sequences are longer and not equally likely;
- Task attributes are embedded in vectors of size 3 using floats obtained by PCA, instead of using binary values;
- For any given task buffer sequence, it is possible to deterministically reach maximum reward, however there are intermediate states that hinder maximum reward. The underlying general assumption to guide policy is: at every step, select the policy that maximizes total cumulative reward, so that selection of immediate action considers not only potential the immediate reward, but also the reward of policies for future tasks.

Figure 44 - Dummy environment F configuration

Slot mins*	Slots per day*	Days per week*	Weeks*	Tasks visible*	Slots to fill*
20	12	2	1	1	8

Slot cost

08:00	6	6
08:20	3	3
08:40	0	0
09:00	0	0
09:20	0	0
09:40	0	0
10:00	0	0
10:20	0	0
10:40	0	0
11:00	0	0
11:20	3	3
11:40	6	6

Root attribute

Tag	Rectangle	Clock	Pattern
A	A	1	0.25
B	B	2	0.25
C	C	1	0.25
D	D	2	0.25

Bottom fields: **Tag** - Attribute name; **Rectangle** – Abbreviation; **Clock** – Slot factor; **Pattern** – Conditional probability.

Table 6 - RL Agent A9, A10, A11 and A12 configuration for dummy environment F

			Structure			
Name	α	**Clip**	**Workers**	**Hidden**	**Nodes**	**RNN**
A9	.0001	**40**	**4**	**2**	**16**	0
A10	.0001	**20**	**12**	**2**	**64**	0
A11	.0001	**10**	**16**	**2**	**128**	0
A12	.0001	**5**	**16**	**3**	**128**	0

Varying parameters between the candidate RL agents are displayed using bold typeface.
α – Learning Rate; **Clip** – Gradient clip; **Hidden** – Number of hidden layers;
Nodes – Number of nodes per hidden layer; **RNN** – RNN size; **Workers** – Number of workers;

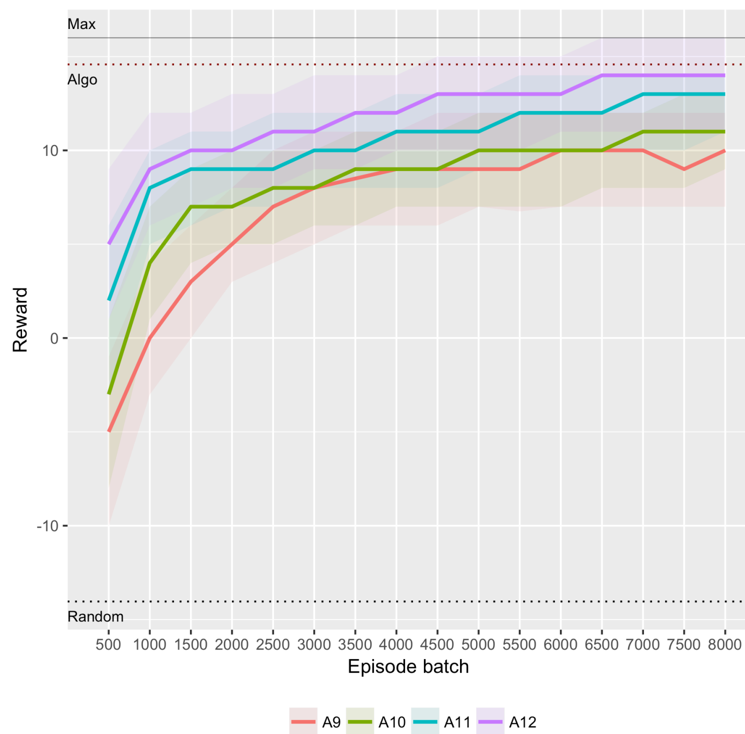
Quantitative analysis

Figure 45 demonstrates that the algorithm heuristic now underperforms the max reward heuristic, since the choice of slot for a task may impact tasks downstream on the task buffer, the algorithm will generally underperform when compared to the max reward heuristic that sorts the task buffer before scheduling. Regarding the RL agents, the four

took many more training episodes to train, but reached high scores, particularly the A12 agent, that applied optimal policies in 25% of the cases, and near optimal policies in more than 50% of the cases.

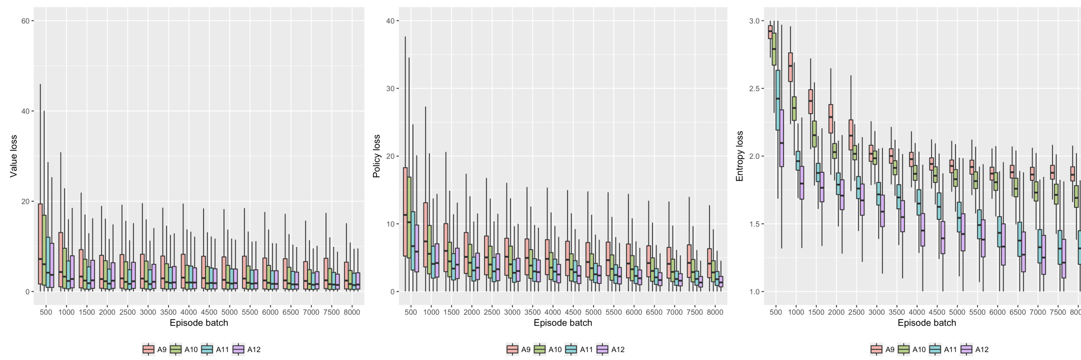
Figure 46 demonstrates that value and policy losses were minimized and converged to particularly small and less variant values for agents A11 and A12 which used 16 workers. All the agents minimized entropy, but still had room for convergence to steady state, despite having reached near optimal solutions for agent A12 around episode 6500. Agent A9 entropy loss flattened around episode 6500.

Figure 45 - Reward results on dummy environment F



Reward per training epoch for RL agents on environment F.
Lines plot percentile **50** and the shaded areas plot the space between the percentile **25** and **75** for each RL agent.

Figure 46 - RL Agent losses on dummy environment F



From left to right, value, policy and entropy losses across episodes for RL agents in dummy environment **F**.

Qualitative analysis

For most situations, agent A12 learned to maximize cumulative reward and find policies that do not limit future reward, and thus frequently achieving maximum reward through policies depicted in Figure 47. Note that learning multiple optimal policies paramount to reach sustained maximum reward.

Figure 47 - Optimal solutions for dummy environment F

08:00	-	-
08:20	-	-
08:40	-	-
09:00	-	B
09:20	C	-
09:40	C	D
10:00	C	-
10:20	C	D
10:40	-	-
11:00	-	D
11:20	-	-
11:40	-	-

08:00	-	-
08:20	-	-
08:40	-	-
09:00	C	-
09:20	C	-
09:40	A	D
10:00	A	-
10:20	A	D
10:40	-	-
11:00	-	D
11:20	-	-
11:40	-	-

08:00	-	-
08:20	-	-
08:40	-	-
09:00	-	D
09:20	A	-
09:40	A	D
10:00	A	-
10:20	D	D
10:40	-	-
11:00	D	-
11:20	-	-
11:40	-	-

Agent A12 learned more than one policy to reach maximum reward consistently.

Discussion

Even though the state spaces increased by 4 orders of magnitude to 176 986 730 potential states, however the agents learned to perform well. Decreasing the learning rate was once again necessary to prevent gradient overflows and stabilize learning. Because of the increasing complexity of state space, additional network structure was necessary for proper representation, and additional workers were necessary to increase learning variability and increase the odds of observing less likely task buffer sequences. The global increase in worker instances, hidden layers and layer nodes accounted for the results obtained by algorithm A12. There was still room for further training and improvement of the results as seen by the entropy loss evolution, however the results obtained by the best agent were demonstrated to perform as well as, or possibly better

than the heuristics, thus, being able to automatically capture the underlying structure of the scheduling problems, and effectively amassing such knowledge to guide decision and arrive at globally good quality solutions.

Real daily environment

This real environment tests whether the RL agent can perform on a real-world problem setting. The environment is presented in Figure 44 and the candidate agents in Table 6. The following are determinants of increasing state space complexity in the real environment relative to Dummy Environment F:

- The task buffer increased from 8 to 18;
- Task attributes increased from 4 to 5;
- The total number of slots increased from 24 to 72.

This implies an increase in state space by many orders of magnitude. The environment employs a set of real tasks attributes from the Portuguese Primary Care Setting, whose frequencies were estimated from a full-year of appointments at a Primary Care Clinic. The schedule is composed of 10 minute slots, that can be used to hold tasks that required either 2 or 3 of those slots, thus requiring 20 and 30 minutes respectively. The total number of appointments sums up to between 6 and 9 working hours, rendering common working time schedules for Primary Care Doctors. The slots structure is presented in Figure 48, and the task attribute structure in Figure 49. Table 7 presents the structure of the candidate agents that were used to perform the task. Because of overflow problems the activation functions of these algorithms a few of the common settings of changed as follows:

- Activation function – TanH was used to prevent gradient overflows;
- Loss regularization factor – 0.25 was used to prevent underfitting;
- Entropy regularization factor - 0.01 was used to prevent underfitting.

Finally, because of the real-world complexity of this environment, the performance of a human agent with expert knowledge on Primary Care scheduling was also assessed.

Figure 48 – Real daily environment slot structure

Slot mins*	Slots per day*	Days per week*	Weeks*	Tasks visible*	Slots to fill*
10	72	1	1	1	18

Slot cost

08:00	3
08:10	3
08:20	3
08:30	3
08:40	3
08:50	3
09:00	0
09:10	0
09:20	0
09:30	0
09:40	0
09:50	0
10:00	0
10:10	0
10:20	0
10:30	0
10:40	0
10:50	0
11:00	0
11:10	0
11:20	0
11:30	0
11:40	0
11:50	0
12:00	0
12:10	0
12:20	0
12:30	3
12:40	3
12:50	3
13:00	6
13:10	6
13:20	6
13:30	6
13:40	6
13:50	6
14:00	3
14:10	3
14:20	3
14:30	0
14:40	0
14:50	0
15:00	0
15:10	0
15:20	0
15:30	0
15:40	0
15:50	0
16:00	0
16:10	0
16:20	0
16:30	0
16:40	0
16:50	0
17:00	0
17:10	0
17:20	0
17:30	3
17:40	3
17:50	3
18:00	3
18:10	3
18:20	3
18:30	3
18:40	3
18:50	3
19:00	6
19:10	6
19:20	6
19:30	6
19:40	6
19:50	6

Figure 49 – Real daily environment task attribute structure

Root attribute

Attribute	Abbreviation	Slot factor	Conditional probability
Child	Ch	3	0.180
Pregnant	Pr	3	0.182
Hypertension	Ht	2	0.219
Diabetes	Dm	3	0.092
Adult	Ad	2	0.327

Slots of 10 minutes are assumed. Task time factor and probabilities were estimated from Primary Care data.
Tag - Attribute name; **Rectangle** – Abbreviation; **Clock** – Slot factor; **Pattern** – Conditional probability. Bottom fields:
Tag - Attribute name; **Rectangle** – Abbreviation; **Clock** – Slot factor; **Pattern** – Conditional probability.

Table 7 - RL Agent A13 and A14 configuration for Real daily environment

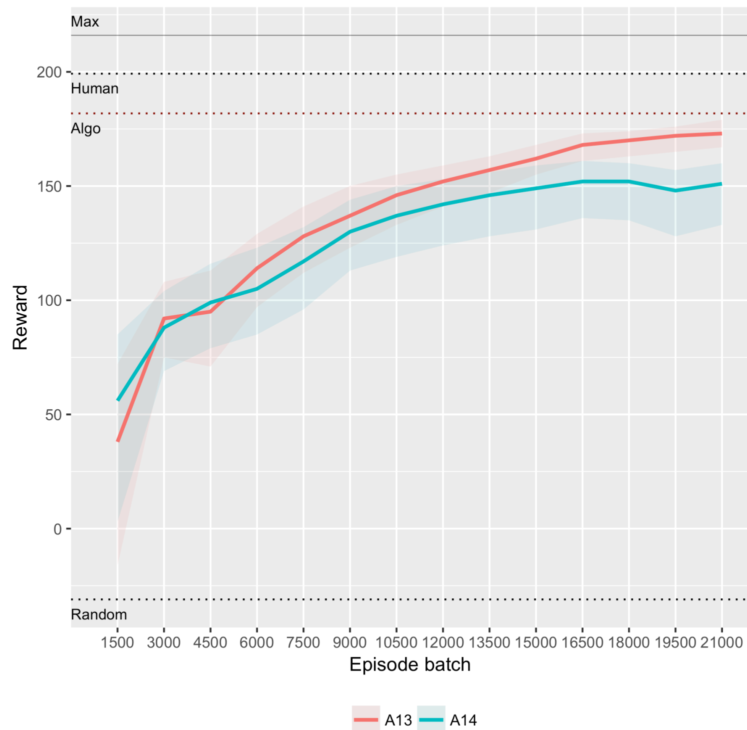
			Structure			
Name	α	**Clip**	**Workers**	**Hidden**	**Nodes**	**RNN**
A13	.0001	20	12	**3**	256	0
A14	.0001	20	12	**4**	256	0

Varying parameters between the candidate RL agents are displayed using bold typeface.
α – Learning Rate; **Clip** – Gradient clip; **Hidden** – Number of hidden layers;
Nodes – Number of nodes per hidden layer; **RNN** – RNN size; **Workers** – Number of workers;

Quantitative analysis

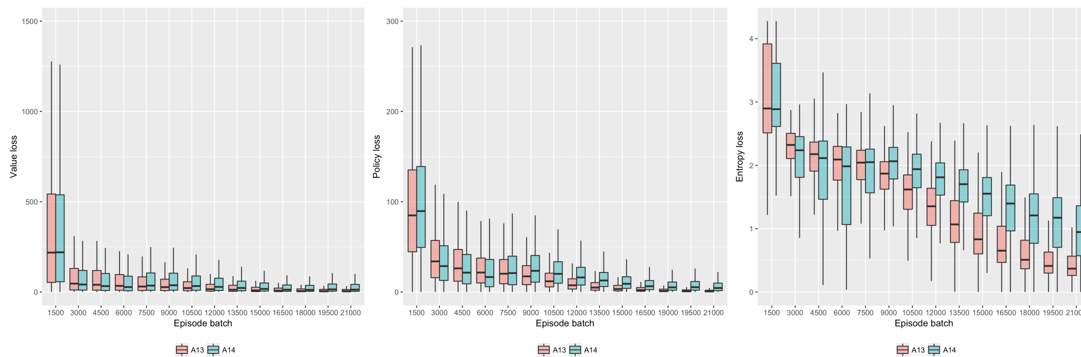
Figure 50 shows that the unit process effect of the scheduling problem in Primary Care makes the algorithm and the human agent significantly underperform relative to the max heuristic that executes under perfect information. In addition, we can see that both agents were able to achieve performance levels compared to the algorithm agent, in particular agent A13. Figure 51 indicates that RL Agents minimized the losses for value and policy, however entropy still did not level off for both algorithms, indicating that there was still room for improvement, mainly for algorithm A14. Finally, since state complexity grew notably for this environment, the number of training sessions to achieve good performance also grew significantly, surpassing the 21000 mark for the presented tests. In addition to the plotted results, the average computation time for a single instance of the RL agent *versus* the algorithm agent were respectively 120ms and 56000ms. This denotes an increase in speed of more than 400 times of the RL agent against the greedy heuristic of the algorithm agent.

Figure 50 - Reward results on real daily environment



Reward per training epoch for RL agents on environment F.
Lines plot percentile **50** and the shaded areas plot the space between the percentile **25** and **75** for each RL agent.

Figure 51 - RL Agent losses on real daily environment



From left to right, value, policy and entropy losses across episodes for RL agents in dummy environment **F**.

Qualitative analysis

Both algorithms A13 and A14 arrived at good solutions frequently, even though they did not know beforehand what type of tasks would they be required to schedule. The algorithm groups similar tasks together and leaves enough space between appointments. Some appointments are spaced far apart in a failed attempt make room for similar tasks that could have arrived in the future but that did not. Because leaving

empty slots did not yield high penalties, the algorithm prioritizes grouping and avoiding overlaps over blank spaces.

Figure 52 - Optimal solutions for real daily environment

08:00	-	08:00	-	08:00	-
08:10	-	08:10	-	08:10	-
08:20	-	08:20	-	08:20	-
08:30	-	08:30	Ad	08:30	-
08:40	-	08:40	-	08:40	Ch
08:50	Ht	08:50	Ch	08:50	-
09:00	-	09:00	-	09:00	-
09:10	-	09:10	-	09:10	Ch
09:20	-	09:20	-	09:20	-
09:30	Ch	09:30	Ch	09:30	-
09:40	-	09:40	-	09:40	-
09:50	-	09:50	-	09:50	Ad
10:00	Ad	10:00	Ad	10:00	-
10:10	-	10:10	-	10:10	Ad
10:20	Ad	10:20	Ad	10:20	-
10:30	-	10:30	-	10:30	-
10:40	-	10:40	-	10:40	-
10:50	Ad	10:50	Ad	10:50	Dm
11:00	-	11:00	-	11:00	-
11:10	Ht	11:10	Ht	11:10	-
11:20	-	11:20	-	11:20	Ch
11:30	-	11:30	-	11:30	-
11:40	Ad	11:40	Ad	11:40	-
11:50	-	11:50	-	11:50	Ch
12:00	Ad	12:00	Ad	12:00	-
12:10	-	12:10	-	12:10	-
12:20	Pr	12:20	-	12:20	Pr
12:30	-	12:30	-	12:30	-
12:40	-	12:40	-	12:40	-
12:50	Ch	12:50	Dm	12:50	Pr
13:00	-	13:00	-	13:00	-
13:10	-	13:10	-	13:10	-
13:20	-	13:20	-	13:20	-
13:30	Dm	13:30	Ch	13:30	-
13:40	-	13:40	-	13:40	-
13:50	-	13:50	-	13:50	-
14:00	Pr	14:00	Ht	14:00	Ch
14:10	-	14:10	-	14:10	-
14:20	-	14:20	-	14:20	-
14:30	-	14:30	-	14:30	-
14:40	Ad	14:40	Ad	14:40	Ch
14:50	-	14:50	-	14:50	-
15:00	Ht	15:00	Ch	15:00	-
15:10	-	15:10	-	15:10	Ch
15:20	-	15:20	-	15:20	-
15:30	-	15:30	-	15:30	-
15:40	Ht	15:40	Pr	15:40	Ad
15:50	-	15:50	-	15:50	-
16:00	-	16:00	-	16:00	Ch
16:10	-	16:10	-	16:10	-
16:20	-	16:20	Pr	16:20	-
16:30	-	16:30	-	16:30	Ch
16:40	-	16:40	-	16:40	-
16:50	Ht	16:50	Pr	16:50	-
17:00	-	17:00	-	17:00	Ad
17:10	Ch	17:10	-	17:10	-
17:20	-	17:20	-	17:20	Ad
17:30	-	17:30	-	17:30	-
17:40	-	17:40	-	17:40	-
17:50	-	17:50	-	17:50	Pr
18:00	Ch	18:00	-	18:00	-
18:10	-	18:10	-	18:10	-
18:20	-	18:20	-	18:20	-
18:30	-	18:30	Ch	18:30	-
18:40	-	18:40	-	18:40	-
18:50	-	18:50	-	18:50	-
19:00	-	19:00	-	19:00	-
19:10	-	19:10	-	19:10	-
19:20	-	19:20	-	19:20	-
19:30	-	19:30	-	19:30	-
19:40	-	19:40	-	19:40	-
19:50	-	19:50	-	19:50	-

Agent A13 arrived at nearly optimal solutions frequently.

Discussion

Even though the state spaces increased substantially RL agents learned to perform nearly the same level as the algorithm agent. Changing the activation function was crucial for stable learning, since other agents exhibited significant overflow problems around episode 6000 which rendered further learning impossible. To balance training stability and performance in the face of a different activation function, the number of worker instances was lowered to 12. This was shown to be the minimum required to avoid RL agent instability and consequential forgetting of the learning rules, which also became quite common in the longer training sessions. Further improving the number of instances would stabilize learning and reduce the overall episode count, however the total training time increased substantially due to the increased parallel work. It was also seen that training for A14 could be extended for further episodes to reach better results, as can be seen by the both the reward curve profile that did not level off, and from the entropy chart. To some extent, prolonging A13 training could possibly yield marginally better results.

Even though the RL agent performance approached the algorithm and the human performance but did not surpass it, it was seen that it takes 400 times less time than the algorithm agent to arrive at good enough solutions presented in the qualitative analysis, thus making it feasible to use in a real-time setting, where computation time is an important constraint. While the algorithm agent, decision time grows exponentially in relation to task buffer size, and linearly with slot buffer size, the RL agent grows at most linearly to the network, implying that for more complex environments, the algorithmic agent may become unfeasible due to high computation costs.

Chapter 6

Discussion and Future Work

In the present work, we have formulated the problem of task scheduling and applied it in the context of Primary Care appointment scheduling. We defined the requirements for feasible solutions for this problem, considering appointment scheduling models according to the unit process, and suggested RL agents as promising algorithms. Afterwards, we devised and developed a set of micro-services that compose an integrated platform for creation, development and testing of specific appointment scheduling environments, and studied the performance of RL agents in a set of predefined environments of increasing difficulty. Finally, we have seen that RL agents are able to learn the underlying rules that govern such environments that need to be considered to arrive at optimal or near optimal scheduling solutions and meet computation time constraints that simple heuristics failed to meet. Some issues were raised along this path which we present below.

Patient scheduling is a POMDP requiring only knowledge of the current state

The formulation of this type of patient appointment scheduling is a POMDP. Indeed, only knowledge of the present state is required to select an action and transition to the next. Using RNN seemed interesting since it could allow experience from the past states to influence the decision process in the next states. However, we have seen that assumption does hinder the ability of the RL agent to learn. Because the action to select depends only on the current state space, training the weights of the current network without linking to past states results in a better approximation of the function that maximizes learning.

Dummy environments demonstrated RL agent ability to maximize cumulative reward

The increasing difficulty of the environments that were created exposed the need for the model to learn to maximize global reward in place on immediate step reward. One factor that contributed to such learning was giving reward only at the end of each episode. That way the network had to create its own representation of the accountability of reward of each decision it took. This design decision seems good, since giving reward to the agent at every state may not lead to a generalized long-term

reward maximization strategy and let the agent focus on the relative benefit of each task, and possibly converge to local optima corresponding to policies that would prefer intermediate states for which not all tasks on the buffer were scheduled. In addition, this formulation of the problem as rewarding global action instead of providing reward along the steps is also aligned with the nature of the problem that the RL agent is intended to surpass, which is that of probabilistic decision given incomplete information. The RL agent demonstrated the ability of amassing such knowledge and take decision considering that overarching goal.

Decision space is much lower dimensional than state space

We have also seen that the network structure required to achieve optimal and near-optimal policy learning is small when compared to state space. This indicates that the network learns to represent rules and policies instead of intermediate spaces, since such spaces would could not possibly be represented in such small networks. We have also seen that too small networks develop suboptimal policies that consistently lead to a suboptimal result, and that small increments in layer numbers or duplication of nodes per layer was enough to represent optimal and near-optimal policies.

Increasing the number of worker networks stabilizes and speeds up learning

Finally, we have seen that incrementing the number of workers frequently yielded faster convergence to optimal policy. This in line with the findings that the increase in number of instance networks of the A3C algorithms drops training time in a linear way [37]. This makes utilization of hardware resources more efficient, and makes it possible to train RL agents with optimal performance in commodity hardware. In addition, such approach also decreases overall iteration development iteration time, making the development process more efficient than with other implementations.

Furthermore, while the real daily environment presented complexity levels real Primary Care daily schedules, further improvements are needed to completely model the scheduling problem for suitable used in production setting. Next, we present the steps and challenges that need to be considered in future work to develop RL agents for real world application.

Development of predefined schedule templates

We have devised a problem in which the RL agent learns to schedule tasks on an empty schedule. Even though the agent does need to learn actions for intermediate

schedules to arrive at the final solution, a well-trained RL agent could be use in practice to develop and suggest schedule templates, that could be used to effectively schedule appointments taking into consideration appointment frequencies and requirements. This would be useful because even though schedules need to be reviewed frequently, they should ideally change in a step-wise fashion and not in a continuous fashion, to better serve the needs of doctors and avoid the novelty and surprise that continuously updating schedule structure would create.

Training agents to act on partially filled schedules

After having created scheduling templates, scheduling task could be extended to take two steps. Before selecting a slot from the schedule, the RL agent should choose from a set of pre-filled schedules, that correspond to each working week. This two-step approach would have the benefit of providing the RL agent with additional opportunities to find the best scheduling solutions, and it would also map directly to the scheduling task performed by the doctors during appointments, in which the doctor needs to consider the problem of scheduling the patient in a slot, as well as spacing the appointments for the patient within the recommended or required timeframe.

Environment reward design

Environment reward plays a key role in the ability of the RL agent to learn. Poor reward design decisions may result in the agent not learning the underlying environment rules, of choosing suboptimal action. Because in real-world scenarios the number of slots is expected to be much larger than in the dummy environments presented, the number of steps per episode is bound to increase by one order of magnitude. This poses a challenge for the algorithm to weigh the accountability of each single action to reach the reward. As such, it might be important to consider intermediate checkpoints where reward is given at a specified number of steps of an episode and training is performed not only at the end, but also during the episode. In addition, a stale slot factor should be added to control the weight of the penalties incurred by leaving empty slots, which becomes relevant in real-world schedules composed of slots of short duration and tasks taking numerous slots.

Same-day appointments and multiple doctor schedules

The model can also be enriched to accommodate same-day appointments. Because the challenges of scheduling same-day appointments have mainly to do with how

same-day appointment slots are distributed across doctors to provision adequate demand during the day, considering this problem also requires development of scheduling algorithms that take on multiple schedules at once. That goal can be achieved using the number of weeks to model multiple separate schedules for each doctor, and current additional reward rules needed to be designed to compute bonus or penalties of solutions that provision same-day slots as required by the observed distribution of same-day appointment demand.

Lateness, no-shows and interruptions

An additional variable that can be taken into consideration when creating task attributes is the probability for lateness and no-show. Such probabilities would allow the model to design scheduling schemes that incorporate relevant factors that drive increasing waiting times and doctor and patient dissatisfaction. In addition, interruptions can be modeled as slot attributes that indicate the probability of interruption and interruption duration at a given time. That way the RL agent could also consider moving appointments that are prone to take longer away from the moments in which interruptions are likely to appear. Small slot costs, implemented in the current system could also be used to model the problem.

Assess transfer learning as means to adapt agents to small environment shifts

Because the task attribute probabilities and slot costs can suffer small changes overtime, it is worthwhile to assess how pre-trained algorithms adapt to these changes and how much time they require to achieve optimal performance in comparison to training the algorithms from scratch. More complex changes may need additional training since policies very different from the ones the RL agent learned may be needed to reach good results.

Online performance assessment

Finally, online agent performance assessment becomes necessary. That is because the agent is expected to learn during offline training phase, and then perform tasks in production without trying to learn new strategies. Provided that appointment attributes and distributions come from real-world data, it is expected that when exploiting, the RL agent performs equally well in both environments. However, if the agent can learn online in production environment, the balance between exploitation and exploration may hinder performance in real world, if the agents chooses to explore different actions.

Thus, not only does an RL agent performance needs to be measured when solely exploiting, to estimate the fitness for a production environment, but also, it should refrain from learning, and apply only the acquired knowledge. That way the performance of the agent is expected to increase and become more consistent.

RL agent algorithm upgrades

Recent advances have hypothesized that variations in the structure of the A3C algorithm may further increase its learning speed and overall performance. Namely, it seems that making A3C instances synchronous may yield faster learning than A3C, an algorithm named A2C [85].In addition, using trust region optimization leads to consistent improvement, and the use of a distributed Kronecker factorization method seems to improve sample efficiency and scalability, yielding and algorithm named Actor Critic using Kronecker-factored Trust Region [85]. Upgrading the current A3C algorithm with such methods may also improve the overall performance of RL agents in this setting.

Taking these considerations to extend the present work may lead to production-ready RL agents able to effectively replace doctors and clerks in scheduling appointments in the Primary Care setting, and ultimately improve the quality of provided healthcare.

References

[1] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, “Resource Management with Deep Reinforcement Learning.,” in *HotNets*, 2016, pp. 50–56.

[2] C. Delimitrou and C. Kozyrakis, “Quasar: resource-efficient and QoS-aware cluster management,” in *ACM SIGPLAN Notices*, 2014, vol. 49, pp. 127–144.

[3] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella, “Multi-resource packing for cluster schedulers,” *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 455–466, 2015.

[4] Y. Sun *et al.*, “Cs2p: Improving video bitrate selection and adaptation with data-driven throughput prediction,” in *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*, 2016, pp. 272–285.

[5] J. Dean and L. A. Barroso, “The tail at scale,” *Commun. ACM*, vol. 56, no. 2, pp. 74–80, 2013.

[6] S. Sahni and Y. Cho, “Complexity of scheduling shops with no wait in process,” *Math. Oper. Res.*, vol. 4, no. 4, pp. 448–457, 1979.

[7] M. C. Gombolay, R. Jensen, J. Stigile, S.-H. Son, and J. A. Shah, “Apprenticeship Scheduling: Learning to Schedule from Human Experts.,” in *IJCAI*, 2016, pp. 826–833.

[8] S. M. Kehle, N. Greer, I. Rutks, and T. Wilt, “Interventions to improve veterans’ access to care: a systematic review of the literature,” *J. Gen. Intern. Med.*, vol. 26, no. 2, p. 689, 2011.

[9] S. D. Pizer and J. C. Prentice, “What are the consequences of waiting for health care in the veteran population?,” *J. Gen. Intern. Med.*, vol. 26, no. 2, p. 676, 2011.

[10] T. Cayirli and E. Veral, “Outpatient scheduling in health care: a review of literature,” *Prod. Oper. Manag.*, vol. 12, no. 4, pp. 519–549, 2003.

[11] J. Goldsmith, “The hospital as we know it is too costly, too unwieldly, and too inflexible to survive. A radical prescription for hospitals.,” *Harv. Bus. Rev.*, vol. 67, no. 3, pp. 104–111, 1988.

[12] X.-M. Huang, “Patient attitude towards waiting in an outpatient clinic and its applications,” *Health Serv. Manage. Res.*, vol. 7, no. 1, pp. 2–8, 1994.

[13] A. Jackson, “A waiting time survey in general practice.,” *Aust. Fam. Physician*, vol. 20, no. 12, pp. 1744–7, 1991.

[14] C.-F. Chien, F.-P. Tseng, and C.-H. Chen, “An evolutionary approach to rehabilitation patient scheduling: A case study,” *Eur. J. Oper. Res.*, vol. 189, no. 3, pp. 1234–1253, 2008.

[15] J. Canet *et al.*, “Development and validation of a score to predict postoperative respiratory failure in a multicentre European cohort: A prospective, observational study,” *Eur. J. Anaesthesiol.*, vol. 32, no. 7, pp. 458–470, Jul. 2015.

[16] N. Liu, S. Ziya, and V. G. Kulkarni, “Dynamic scheduling of outpatient appointments under patient no-shows and cancellations,” *Manuf. Serv. Oper. Manag.*, vol. 12, no. 2, pp. 347–364, 2010.

[17] J. Patrick, M. L. Puterman, and M. Queyranne, “Dynamic multipriority patient scheduling for a diagnostic resource,” *Oper. Res.*, vol. 56, no. 6, pp. 1507–1525, 2008.

[18] J. Strahl and others, “Patient appointment scheduling system: with supervised learning prediction,” 2015.

[19] D. Gupta and B. Denton, “Appointment scheduling in health care: Challenges and opportunities,” *IIE Trans.*, vol. 40, no. 9, pp. 800–819, 2008.

[20] N. P. H. Trust and U. of Bristol, *Studies in the Functions and Design of Hospitals: The Report of an Investigation Sponsored by the Nuffield Provincial Hospitals Trust and the University of Bristol*. Oxford University Press, 1955.

[21] J. F. Rockart and P. B. Hofmann, “Physician and patient behavior under different scheduling systems in a hospital outpatient department,” *Med. Care*, pp. 463–470, 1969.

[22] M. Murray and C. Tantau, “Same-day appointments: exploding the access paradigm,” *Fam. Pract. Manag.*, vol. 7, no. 8, p. 45, 2000.

[23] Y. Arzi, “On-line scheduling in a multi-cell flexible manufacturing system,” *Int. J. Prod. Res.*, vol. 33, no. 12, pp. 3283–3300, 1995.

[24] S. Russell, P. Norvig, and A. Intelligence, “Artificial Intelligence, a modern approach,” *Prentice-Hall Englewood Cliffs*, vol. 25, p. 27, 1995.

[25] L. Bianchi, M. Dorigo, L. M. Gambardella, and W. J. Gutjahr, “A survey on metaheuristics for stochastic combinatorial optimization,” *Nat. Comput.*, vol. 8, no. 2, pp. 239–287, 2009.

[26] C. Blum and A. Roli, “Metaheuristics in combinatorial optimization: Overview and conceptual comparison,” *ACM Comput. Surv. CSUR*, vol. 35, no. 3, pp. 268–308, 2003.

[27] D. Sun and L. Lin, “A dynamic job shop scheduling framework: a backward approach,” *Int. J. Prod. Res.*, vol. 32, no. 4, pp. 967–985, 1994.

[28] S. Nakasuka and T. Yoshida, “Dynamic scheduling system utilizing machine learning as a knowledge acquisition tool,” *Int. J. Prod. Res.*, vol. 30, no. 2, pp. 411–431, 1992.

[29] M. Asada, S. Noda, S. Tawaratsumida, and K. Hosoda, “Purposive behavior acquisition for a real robot by vision-based reinforcement learning,” *Mach. Learn.*, vol. 23, no. 2, pp. 279–303, 1996.

[30] R. Kohavi and F. Provost, “Glossary of terms,” *Mach. Learn.*, vol. 30, no. 2–3, pp. 271–274, 1998.

[31] L. Busoniu, R. Babuska, B. De Schutter, and D. Ernst, *Reinforcement learning and dynamic programming using function approximators*, vol. 39. CRC press, 2010.

[32] R. Glaubius, T. Tidwell, C. Gill, and W. D. Smart, “Real-time scheduling via reinforcement learning,” *ArXiv Prepr. ArXiv12033481*, 2012.

[33] M. E. Aydin and E. Öztemel, “Dynamic job-shop scheduling using reinforcement learning agents,” *Robot. Auton. Syst.*, vol. 33, no. 2, pp. 169–178, 2000.

[34] R. S. Sutton and A. G. Barto, *Reinforcement learning: An overview*, vol. 1. MIT press Cambridge, 1998.

[35] V. Mnih *et al.*, “Human-level control through deep reinforcement learning,” *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[36] D. Silver *et al.*, “Mastering the game of Go with deep neural networks and tree search,” *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[37] V. Mnih *et al.*, “Asynchronous methods for deep reinforcement learning,” in *International Conference on Machine Learning*, 2016, pp. 1928–1937.

[38] Y. Li, “Deep reinforcement learning: An overview,” *ArXiv Prepr. ArXiv170107274*, 2017.

[39] “Log Analytics With Deep Learning And Machine Learning,” *Hackernoon*, 12-May-2017. .

[40] V. Nair and G. E. Hinton, “Rectified linear units improve restricted boltzmann machines,” in *Proceedings of the 27th international conference on machine learning (ICML-10)*, 2010, pp. 807–814.

[41] X. Glorot, A. Bordes, and Y. Bengio, “Deep sparse rectifier neural networks,” in *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, 2011, pp. 315–323.

[42] J. Schmidhuber, “Deep learning in neural networks: An overview,” *Neural Netw.*, vol. 61, pp. 85–117, 2015.

[43] A. Graves, M. Liwicki, H. Bunke, J. Schmidhuber, and S. Fernández, “Unconstrained on-line handwriting recognition with recurrent neural networks,” in *Advances in Neural Information Processing Systems*, 2008, pp. 577–584.

[44] H. M. Fayek, M. Lech, and L. Cavedon, “Evaluating deep learning architectures for Speech Emotion Recognition,” *Neural Netw.*, 2017.

[45] S. Hochreiter and J. Schmidhuber, “LSTM can solve hard long time lag problems,” in *Advances in neural information processing systems*, 1997, pp. 473–479.

[46] A. Graves, A. Mohamed, and G. Hinton, “Speech recognition with deep recurrent neural networks,” in *Acoustics, speech and signal processing (icassp), 2013 ieee international conference on*, 2013, pp. 6645–6649.

[47] L.-Y. Deng, *The cross-entropy method: a unified approach to combinatorial optimization, Monte-Carlo simulation, and machine learning*. Taylor & Francis, 2006.

[48] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning internal representations by error propagation,” California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[49] P. J. Werbos, “Backpropagation through time: what it does and how to do it,” *Proc. IEEE*, vol. 78, no. 10, pp. 1550–1560, 1990.

[50] Y. Dauphin, H. de Vries, and Y. Bengio, “Equilibrated adaptive learning rates for non-convex optimization,” in *Advances in neural information processing systems*, 2015, pp. 1504–1512.

[51] A. Y. Ng, “Feature selection, L 1 vs. L 2 regularization, and rotational invariance,” in *Proceedings of the twenty-first international conference on Machine learning*, 2004, p. 78.

[52] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout: a simple way to prevent neural networks from overfitting.,” *J. Mach. Learn. Res.*, vol. 15, no. 1, pp. 1929–1958, 2014.

[53] D. P. Bertsekas and J. N. Tsitsiklis, “Neuro-dynamic programming: an overview,” in *Decision and Control, 1995., Proceedings of the 34th IEEE Conference on*, 1995, vol. 1, pp. 560–564.

[54] R. S. Sutton, “Temporal credit assignment in reinforcement learning,” 1984.

[55] R. S. Sutton and A. G. Barto, *Reinforcement learning: An overview*, 2nd Edition (In Preparation)., vol. 1. MIT press Cambridge, 2017.

[56] E. A. Feinberg and A. Shwartz, *Handbook of Markov decision processes: methods and applications*, vol. 40. Springer Science & Business Media, 2012.

[57] A. Markov, “Extension of the limit theorems of probability theory to a sum of variables connected in a chain,” 1971.

[58] A. M. Schäfer, “Reinforcement learning with recurrent neural networks,” 2008.

[59] M. L. Puterman, *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

[60] D. P. Bertsekas, D. P. Bertsekas, D. P. Bertsekas, and D. P. Bertsekas, *Dynamic programming and optimal control*, vol. 1. Athena scientific Belmont, MA, 1995.

[61] R. S. Sutton, “Learning to predict by the methods of temporal differences,” *Mach. Learn.*, vol. 3, no. 1, pp. 9–44, 1988.

[62] N. Metropolis and S. Ulam, “The monte carlo method,” *J. Am. Stat. Assoc.*, vol. 44, no. 247, pp. 335–341, 1949.

[63] G. Tesauro, “Td-gammon: A self-teaching backgammon program,” in *Applications of Neural Networks*, Springer, 1995, pp. 267–285.

[64] M. Riedmiller, "Neural fitted Q iteration-first experiences with a data efficient neural reinforcement learning method," in *ECML*, 2005, vol. 3720, pp. 317–328.

[65] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization," in *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, 2015, pp. 1889–1897.

[66] V. Mnih *et al.*, "Playing atari with deep reinforcement learning," *ArXiv Prepr. ArXiv13125602*, 2013.

[67] H. Van Hasselt, A. Guez, and D. Silver, "Deep Reinforcement Learning with Double Q-Learning.," in *AAAI*, 2016, pp. 2094–2100.

[68] C. J. C. H. Watkins, "Learning from delayed rewards," King's College, Cambridge, 1989.

[69] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Mach. Learn.*, vol. 8, no. 3–4, pp. 229–256, 1992.

[70] T. Degris, M. White, and R. S. Sutton, "Off-policy actor-critic," *ArXiv Prepr. ArXiv12054839*, 2012.

[71] A. Nair *et al.*, "Massively parallel methods for deep reinforcement learning," *ArXiv Prepr. ArXiv150704296*, 2015.

[72] R. J. Williams and J. Peng, "Function optimization using connectionist reinforcement learning algorithms," *Connect. Sci.*, vol. 3, no. 3, pp. 241–268, 1991.

[73] M. Abadi *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *ArXiv Prepr. ArXiv160304467*, 2016.

[74] T. P. Vogl, J. Mangis, A. Rigler, W. Zink, and D. Alkon, "Accelerating the convergence of the back-propagation method," *Biol. Cybern.*, vol. 59, no. 4, pp. 257–263, 1988.

[75] J. Fink, "Docker: a software as a service, operating system-level virtualization framework," *Code4Lib J.*, vol. 25, 2014.

[76] P. Hintjens, *ZeroMQ: messaging for many applications*. O'Reilly Media, Inc., 2013.

[77] G. Bierman, M. Abadi, and M. Torgersen, "Understanding typescript," in *European Conference on Object-Oriented Programming*, 2014, pp. 257–281.

[78] M. Piispanen, "Modern architecture for large web applications," 2017.

[79] W. Reese, “Nginx: the high-performance web server and reverse proxy,” *Linux J.*, vol. 2008, no. 173, p. 2, 2008.

[80] M. Grinberg, *Flask web development: developing web applications with python*. O’Reilly Media, Inc., 2014.

[81] A. Thusoo *et al.*, “Hive: a warehousing solution over a map-reduce framework,” *Proc. VLDB Endow.*, vol. 2, no. 2, pp. 1626–1629, 2009.

[82] R Core Team, *R: A Language and Environment for Statistical Computing*. Vienna, Austria: R Foundation for Statistical Computing, 2013.

[83] “Dinghy faster, friendlier Docker on OS X,” 25-Sep-2017. .

[84] J. Watson, “Virtualbox: bits and bytes masquerading as machines,” *Linux J.*, vol. 2008, no. 166, p. 1, 2008.

[85] Y. Wu, E. Mansimov, S. Liao, R. Grosse, and J. Ba, “Scalable trust-region method for deep reinforcement learning using Kronecker-factored approximation,” *ArXiv Prepr. ArXiv170805144*, 2017.