Learning to Control the Cloud

Heimerson, Albin

2023

Document Version:
Publisher's PDF, also known as Version of record

Learning to Control the Cloud

Albin Heimerson

LUND
UNIVERSITY

Department of Automatic Control

PhD Thesis TFRT-1142
ISBN 978-91-8039-841-1 (print)
ISBN 978-91-8039-842-8 (web)
ISSN 0280–5316

Department of Automatic Control
Lund University
Box 118
SE-221 00 LUND
Sweden

Printed in Sweden by Media-Tryck.
Lund 2023

Abstract

With the growth of the cloud industry in recent years, the energy consumption of the underlying infrastructure is a major concern. The need for energy efficient resource management and control in the cloud becomes increasingly important as one part of the solution, where the other is to reduce the energy consumption of the hardware itself. Resource management in the cloud is typically done using relatively simple methods, with either local controllers or human operators, though as the complexity of the system increases, the need for more intelligent and automated controllers increases as well.

The cloud is a complex environment with many individual consumers sharing large pools of resources, scaling and moving their applications to satisfy their own objectives and requirements, while the cloud provider manages the underlying infrastructure to make efficient use of the hardware. This creates a dynamic environment with a highly variable load, and managing efficient resource usage while keeping the quality of service at an acceptable level is a complex task for such unpredictable environments. Both the consumers scaling their resources and the providers managing their infrastructure could benefit from intelligent automation.

By creating control strategies that take a larger context into account, it could allow for more informed decisions, and thus better control. A larger context makes the problem space more complex, and manually designing a controller becomes increasingly difficult. With the abundance of data available in many cloud systems, a data-driven approach seems like a natural choice. Reinforcement learning is a type of machine learning that is well suited for sequential decisions over time, and has been shown to be able to learn complex control strategies in many different domains. We explore the benefits and challenges of applying reinforcement learning methods to control different cloud systems according to complex objectives, and what usability concerns that show up in practice.

Starting off, we explore the combined control of cooling systems and load balancing in a datacenter. Cooling is a major energy consumer in datacen-

ters, giving us a natural objective for optimization, and the load balancing will affect the heat distribution in the datacenter, thus affecting the cooling. In a simple simulated environment, we apply reinforcement learning to control a mix of discrete and continuous control variables over both cooling and load balancing, with the objective to reduce energy consumption while adhering to temperature thresholds for the servers. We find that the controller is able to learn how to efficiently use the cooling system, improving on a baseline implemented using standard methods. Scaling this up and adding a more realistic air-flow simulation, we find that the gain from perfect placement is so small that it is simply generating noise in comparison to other factors in the cooling system. Instead, we focus on controlling the cooling system with the larger observational context, showing that it outperforms existing standard methods while also being able to adapt to changes in the system.

We then look at the problem of scaling a web services in a cloud environment, where a service is built from many interconnected microservices. These are typically scaled using local reactive controllers, but employing a proactive controller should improve the performance. By providing a reinforcement learning agent with a view over all the services, it implicitly learns how different jobs traverse the system, and use this to proactively scale services, keeping less resources in reserve, and still meeting response time requirements.

Moving from model-free control, we turn to using an existing fluid model of a microservice to create a controller. The fluid model is used to simulate trajectories for a load balancing controller, and using arbitrary loss functions over the trajectory, we can optimize the parameters of the controller using automatic differentiation. The resulting controller behaves well, though we only take a single gradient step to ensure stable updates, since the accuracy of the fluid model is reduced as the system moves away from the training data. We then show how an imperfect model can be extended with neural networks to capture unmodelled dynamics. For the fluid model, the increased accuracy from the extended model allows for more steps and thus faster policy convergence.

While we find that RL can indeed be used to create policies that improve on standard control methods, there are several usability concerns that arise when applying these methods to real systems. The main issue is the instability of the whole process, from exploration during training driving the system to bad states, to opaque function approximators making it difficult to ensure that the controller behaves as expected when deployed. While we discuss several methods to mitigate these issues, what actually works is highly dependent on the specific system and the requirements on the controller.

Acknowledgements

In my journey towards a PhD I have had the pleasure of being surrounded by many wonderful people. While I cannot list everyone here, I would like to thank all of them for providing many fond memories, and for helping me grow as a person and researcher.

I would like to thank the department of Automatic Control at Lund University for providing a great workplace and for supporting me throughout my PhD. And to Eva, Mika, Cecilia, Monica, Anders N and Anders B, thank you for always being there to help with all the practicalities of being a PhD student.

A special thanks go to my supervisor Johan Eker, who have been there to discuss the ideas I pursued, helping me find motivation when I needed it, and providing valuable feedback on my work. I have learned a lot from you, and I am grateful for the opportunity to work with you. And I thank my co-supervisor, Karl-Erik Årzén, for providing valuable feedback and guidance throughout the years.

I was often a bit alone in my research, and really appreciate the times I got to collaborate with other people at the department. Kristian and Bo, I appreciate being invited to investigate Covid spread models with you, it was a great experience and I learned a lot. And thank you Bagge, Mattias and Olof for engaging me with the Julia language, it became a source of much inspiration and even more procrastination during my PhD. And Johan R, I really enjoyed the papers we collaborated on, and I only wish we had found some overlap earlier.

I have also truly enjoyed the many activities organized by people at the department. Thank you Gautham for running the board games and floorball sessions, Felix for making the climbing sessions happen, Martin H, Martin M and Ylva for the disc golf sessions, and Martin GN for making me run faster than I thought I ever would. And to all other people participating and making these activities a joy, thank you.

Nils, we started this journey together, and it has been a pleasure to have you as a friend and colleague. And Martin H, I enjoyed sharing an office

with you, and I hope I get to do it again in the future.

Ericsson Research and RISE SICS North have both provided infrastructure for me to run experiments on, and I am grateful for that. I am also grateful for the opportunity to collaborate with people at both places, and I would like to thank all of them for the interesting discussions we have had.

I really appreciate the help I got with proofreading the thesis, so thank you Olle, Jonas, Ahmed, Johan G, Felix, Max NC and Ylva for all valuable feedback. And Martina, thank you for reviewing the thesis, and for giving good advice throughout my PhD.

And finally, to my family and friends outside of work, often providing a much needed respite from said work. Manfred, thank you for being a great friend and for all the fun times we have had together. And thank you Algot for taking my mind of work with games and other fun activities when I needed it. To my parents, Anneli and Jesper, and my siblings Malte and Alma, thank you for for supporting me throughout my life. Knowing you are there if I need it brings me comfort, and I am grateful for having you as my family. And finally, to Manon, for putting up with my antics and bringing me happiness, thank you for being in my life.

Financial Support

This work has received funding from Vinnova through the AutoDC project, from the Swedish Research Council through project VR 2017-04491, and from the European Union's Horizon 2020 research and innovation program under grant agreement 871259 (ADMORPH). Additionally, the author received a travel grant from the royal physiographic society in Lund for presenting the final publication included in the thesis at the 2023 IFAC World Congress in Japan. The author is a member of the ELLIIT Strategic Research Area at Lund University and is also affiliated with the Wallenberg AI, Autonomous Systems and Software (WASP) program.

Contents

Nomenclature

Terminology	Description
Action	A decision made by the agent to be enacted in the environment.
Agent	The entity that interacts with the environment.
Automatic differentiation	A technique for computing derivatives of functions implemented as computer programs.
Cloud computing	Computing resources provided as a service over the internet.
Environment	The system that the agent interacts with.
Experience replay	A technique for storing experiences and sampling them randomly for training.
Kubernetes	A container orchestration system for automating application deployment, scaling, and management.
Microservice	A small independently deployable and scalable software component that performs a specific, well-defined function within a larger application or system.
Model	Used to denote a dynamics model, not used for neural networks.
Policy	A mapping from states to actions.
Replicas	Separate instances of a single microservice.
Reward	A scalar value that the agent tries to maximize.
Scaling out/in	Increasing/decreasing the number of replicas in a microservice.
State	A set of values describing the current state of the environment.

Abbr.	Description
CFD	*Computational fluid dynamics* simulates fluid behavior.
CNN	*Convolutional neural network* is a neural network with convolutional layers.
CPU	*Central processing unit* is the primary processing component in a computer.
CRAH	*Computer room air-handler* is a cooling system for server rooms.
DC	*Datacenter* houses computing systems and hardware.
DE	*Differential equation* is an equation containing derivatives.
ELU	*Exponential linear unit* is an activation function.
FCFS	*First come, first served* is a queuing model.
GPU	*Graphical processing units* are designed for parallel processing.
IT	*Information technology* is an umbrella term for computing related technology.
MDP	*Markov decision process* is a framework for sequential decision-making.
ML	*Machine learning* are methods where computers learn from data.
MPC	*Model-predictive control* is an optimal control method.
NN	*Neural network* is a function approximator comprised of artificial neurons.
ODE	*Ordinary differential equation* has only one independent variable.
PID	*Proportional-integral-derivative* is a popular control method.
PPO	*Proximal policy optimization* is a popular method for deep RL, Section 3.3.
PS	*Processor sharing* is when the CPU is shared between tasks.
PUE	*Power usage effectiveness* is a measure of datacenter energy efficiency.
RL	*Reinforcement learning* see Chapter 3.
RNN	*Recurrent neural networks* are NNs that contains memory.
RR	*Round robin* is a queuing model.
SAC	*Soft actor-critic* is a popular method for deep RL, Section 3.3.
SLA	*Service level agreement* is a contract between a service provider and a customer.
SLO	*Service level objectives* are metrics for SLAs.
TD	*Temporal difference* is class of RL methods.
TPU	*Tensor processing units* are designed for parallel processing.
UDE	*Universal differential equation* is a differential equation that can fit any dynamics.
VM	*Virtual machine* is a virtualized computer.

Notation	Description
$\lvert\mathcal{X}\rvert$	Cardinality of the set $\mathcal{X}$.
$\mathbb{R}$	The set of real numbers.
$\mathbb{R}_+$	The set of non-negative real numbers.
$\mathbb{R}^{m\times n}$	The set of $m \times n$ matrices with real entries.
X^T	Transpose of matrix X.
$\boldsymbol{x}$	A vector of values.
x_i	The i'th component of $\boldsymbol{x}$.
$f_i(\boldsymbol{x})$	The ith component of the function $\boldsymbol{f}$ evaluated at $\boldsymbol{x}$.
$\langle x_1, x_2, \dots \rangle$	A tuple of elements.
$x \in_R \mathcal{X}$	x is sampled uniformly at random from the elements in $\mathcal{X}$.
$x \sim \mathcal{X}$	x is sampled from the distribution $\mathcal{X}$.
$\mathbb{E}_{\mathcal{X}}[f(x)]$	Expectation of $f(x)$ when x is sampled from $\mathcal{X}$.
$\mathbb{1}(e)$	Function yielding 1 if expression e is true, and 0 otherwise.
$J(\boldsymbol{x})$	Cost function for some parameters $\boldsymbol{x}$.
$\nabla_{\boldsymbol{x}} f(\boldsymbol{x})$	Gradient of f with respect to $\boldsymbol{x}$.
$[x_i \dots]$	A vector with all x_i for existing i.
$\langle \mathcal{S}, \mathcal{A}, \mathcal{T} \rangle$	State space, action space and transition probabilities of an environment.
$\langle \boldsymbol{s}, \boldsymbol{a}, r, \boldsymbol{s}' \rangle$	An experience tuple consisting of a state and action leading to a reward and a new state.
G_t	Cumulative discounted reward from time t and onward.
π	Policy for mapping states to actions, $a \sim \pi(s)$.
$Q^\pi(s, a)$	Action-value function for policy π taking action a from state s.
$V^\pi(s)$	State-value function for policy π from state s.
$\mathcal{D}$	Training data.

1

Introduction

The cloud is a term most people are familiar with, though few could readily define. Many online services we rely on in our daily lives, such as shopping, banking, social media, etc., all build on the cloud. The cloud has enabled much innovation by providing a flexible platform for creating and hosting services over the internet, and has become a key part of the infrastructure for many companies. While the cloud has become an integral part of our digital landscape, its inner workings often remain obscured from end users, who primarily experience the convenience and accessibility it provides.

At a basic level, every online service is running on a server, a computer that is connected to the internet. This computer runs software that allows it to respond to requests from other computers also connected to the internet. A typical request could be for files that make up a website, so that the computer sending the request can render it using a web-browser, and show the website to the user. As the number of users increases, the server needs to handle more requests, and at some point the server will not be able to handle them all in a timely manner. By adding more servers, and distributing the requests between them, we can handle more users. As the website grows, there might be users in different regions that want to access the website. These requests will have to travel further, resulting in a slower response for the user. Instead, servers can be added in more regions, and the requests can be distributed based on where the user is located.

The fundamental idea of the cloud is to abstract away the physical aspect of scaling and distribution of servers, providing a simple way to scale the number of running instances for an application based on the number of users, or move instances to anywhere in the world. This is enabled by virtualization technology, e.g., a *virtual machine* that is a software implementation of a computer, that can run on a physical computer. A server running on a virtual machine can be easily replicated, moved between physical computers, and even between datacenters in different regions.

In Figure 1.1 we show an example of a web application that is running in the cloud. From the user's perspective it just *exists* in the cloud, though

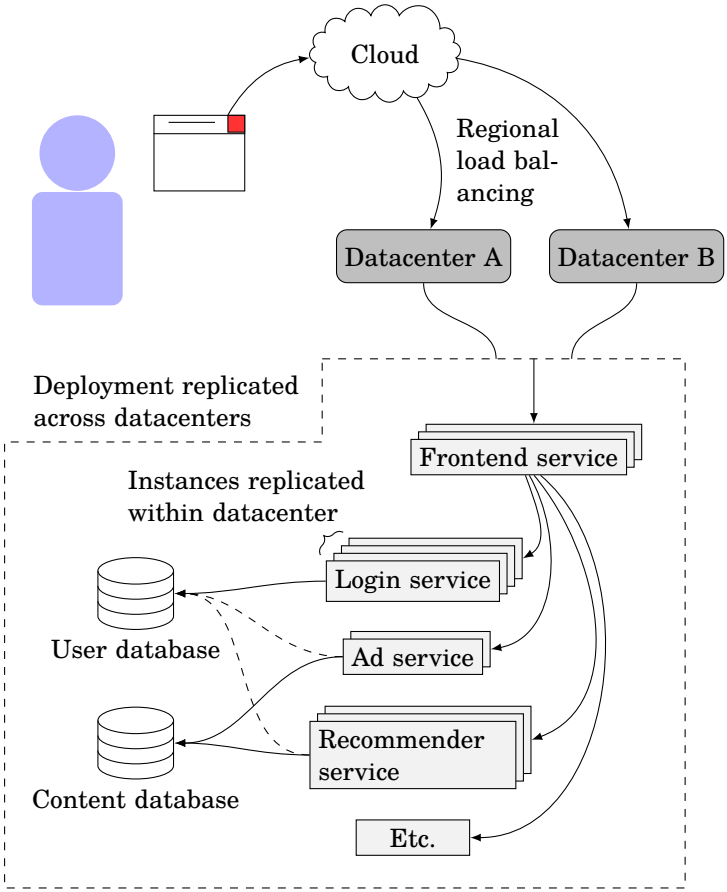

Figure 1.1 Example web app showing the abstractions of the cloud. The user interacts with the web app through their web browser, and in the background the requests will go through a complex network of services. There can be multiple instances of each service, both in the same datacenter and in different datacenters around the world. Selecting how many instances that should be running in each datacenter, and how to route the requests is a complex problem that we want to solve using data-driven methods with access to the larger context surrounding the application.

there are many things going on in the background to make this possible, and to provide a good user experience. The application could for example be a web-shop of some kind, comprising many internal services to provide all the functionality making up the website. As a user tries to load the main page, the user device will send out a request for the relevant files. Based on the IP-address of the user, the request can be sent to the datacenter that is located closest to the user. In this datacenter, the request will be forwarded to one out of potentially multiple instances that are running the frontend service of the web-shop. The frontend service collects the data required to put together the requested page by sending internal requests to different backend services, each responsible for generating some part of the required data. For a logged-in user the page might provide personalized recommendations and ads, and the frontend service will need to verify the user's credentials by checking with a login service which in turn will check with the user database. For an anonymous user the page might instead show some products and ads based on what is popular right now.

In addition to all that is shown here, there are typically layers of caching to be able to faster serve pages that are often visited as well as layers of security to protect the services from malicious actors. There are also many services that are not directly part of the web-shop, but are required to run the web-shop, such as monitoring services, logging services, and services that are responsible for the deployment of new versions of the web-shop. All services can run in multiple instances for each datacenter, and there might be multiple datacenters around the world. To provide a good user experience, while also keeping costs down, we need to carefully balance where to run the services, how many instances of each service to run, how to route the requests, etc.

This problem is typically solved using simple control strategies based on local information for each individual service, see Figure 1.2(a). So, for example, when deciding if we need more or less instances to run the ad service in a certain datacenter, it is common to look at the average utilization over the instances for that service in that datacenter and scale based on those values. While this works and is easy to implement, it typically requires keeping a large safety margin to ensure that we do not run out of capacity when there is a sudden increase in traffic.

In this thesis we take a more holistic approach, providing more context to the controller through information about the system and the environment it is running in. Looking at the scaling problem from this perspective, we want to scale the ad service based on many variables, e.g., utilization of the ad service, utilization of the frontend service, utilization of corresponding services in other datacenters, electricity price in the different regions, etc., see Figure 1.2(b). The goal is to utilize all available information that could possibly enhance the control policy. Addressing this problem through

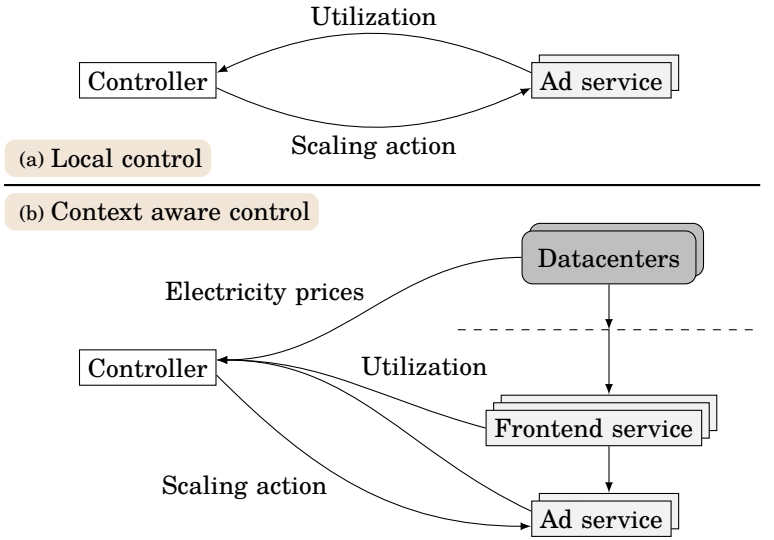

Figure 1.2 Example of scaling the ad service in a cloud application. In Figure 1.2(a) we show a controller that only depends on the state of the service it controls, while Figure 1.2(b) shows a controller that also takes into account the state of other services, and other parts of the environment.

conventional approaches becomes increasingly challenging when considering the larger context of the surrounding systems and environment. This thesis looks at learning-based methods, focusing on reinforcement learning, to overcome that complexity by automating the process of extracting the important variables and crafting a successful control strategy.

1.1 Contributions and Outline

The wider scope of this thesis is to investigate the challenges in applying learning-based methods for solving the problem of controlling the cloud and the infrastructure supporting it. While standard approaches in industry typically employ controllers that do something *good enough* in a local context, a larger context can provide more information to the controller, allowing for more optimized decisions. However, manually synthesizing a controller using conventional methods becomes increasingly challenging when considering the larger context of the surrounding systems and environment.

There are a few motivating factors for the learning-based approach we take in this thesis. We look at the cloud infrastructure from the perspective of both users and providers, and how good control of the infrastructure can

benefit both. This creates complex problems with multiple competing objectives and constraints. We also want to utilize a wide range of information from the surrounding system to make better decisions, creating a complex relationship between the control variables and the objectives. Learning-based methods can be used to extract important dependencies between the control variables and the objectives from data, and optimize a policy without requiring a prior understanding of the system. To do this well, large datasets are required, and cloud environments are a great source for collecting an abundance of data. Learning-based methods can also be adaptive to changes in the environment, something that can be beneficial in a dynamic environment such as the cloud.

Applying learning-based methods to these problems, our main contributions are:

- Showing that a holistic approach using a larger context can be beneficial to algorithms used for controlling the cloud and surrounding infrastructure.
- Demonstrating that reinforcement learning (RL) can learn good control policies in cloud control.
- Developing methods to impose known structure from the system onto policies, and showing that this can improve the efficiency of learning-based strategies.
- Demonstrating applicability of RL on both hardware and software infrastructure, from datacenter (DC) cooling to microservice autoscaling.
- Developing solutions using different learning-based methods, from standard model-free RL, to model-based methods relying on automatic differentiation.

We also provide open-source implementations of the algorithms used in the papers, to allow for reproducibility and for others to build on our work. The source code for the examples and simulations is linked in their respective chapters. In addition, we have contributed to open-source software that we used as part of the work in this thesis. The Julia packages ReinforcementLearning.jl and Hyperopt.jl were both extended to support our use cases, and DistributedEnvironments.jl was created to simplify distributed development and training in the Julia language.

Thesis Outline

Chapter 2 introduces the view of the cloud infrastructure that will be used throughout the thesis, as well as standard methods for controlling different

parts of the infrastructure. In Chapter 3 we introduce RL, going from basic concepts to more advanced methods, focusing on methods that are used in the rest of the thesis. Chapter 4 looks at the practical challenges of using RL for control, discussing problems that are inherent in the field and how they can be addressed. This chapter also provides related work on the topic of cloud control, focusing on how RL has been used in the field. Chapter 5 applies RL to the problem of controlling both the cooling and load balancing of a datacenter. This is extended in Chapter 6 to a more complex datacenter model, where a context-aware RL agent provides adaptive cooling decisions. Chapter 7 moves to the microservice application domain, where RL is used to proactively scale service chains in cloud applications. Chapter 8 also considers a microservice application, but look at the load balancing problem instead. By using an existing fluid model of the microservice application, we show how automatic differentiation can be used to easily optimize a load balancing policy for complex objectives. Chapter 9 looks at improving the model used in Chapter 8 by embedding a neural network in the existing model and learning the missing dynamics from data. Finally, Chapter 10 concludes the thesis and provides some future directions for the work.

Contributions from Each Chapter

Chapter 5 sets up an environment simulating the cooling system of a datacenter, relating how workload generates heat and how the cooling system can be controlled to remove the heat. The control strategy combines control of hardware and software infrastructure, using reinforcement learning to control both cooling and load balancing of the workloads simultaneously.

Chapter 6 extends the model used in Chapter 5 to both use a more complex air-flow simulation, and to include a larger number of servers and cooling units. The neural network (NN) is enhanced to improve learning in more complex environments, and we show how the RL agent can learn to adapt to changes in the environment.

Chapter 7 implements a proactive scaling strategy using RL for a microservice application where the call graph is unknown, and the state is not fully observable. This environment is challenging for a few reasons, e.g., many delayed effects and rewards, and required a couple of modifications to the standard RL setup for the agent to learn well, e.g., extending the state space to include temporally augmented state and restricting the action space to avoid local minima.

Chapter 8 attempts to find load balancing policies to optimize between resource usage and response time. First a fluid model is derived from log data collected in a microservice application. A cost function is defined based on the solution of the fluid model, taking into account transient and stationary queue lengths, as well as constraints on, e.g., response time percentiles.

Using automatic differentiation over the cost function, we can update the parameters for the load balancing policy using, e.g., gradient descent. A procedure to do this online is presented, and the performance of the resulting policy is evaluated in a real-world setting with an application distributed over three different Kubernetes clusters.

Chapter 9 improves on the fluid model used in Chapter 8 by embedding a neural network in the model. The neural network is used to learn some of the dynamics that are not captured by the fluid model, allowing the extended model to provide more accurate predictions, and thus improved policy updates. Additionally, we show that imposing constraints on the neural network based on understanding about the system behavior can in some cases drastically improve convergence speed of the learning process. We compare a few different ways of implementing this.

1.2 Publications

While this thesis is written as a monograph, most of the content is based on previously published work. This section describes the author's publications, how they connect to the chapters and the individual contributions.

Paper I

Heimerson, A., R. Brännvall, J. Sjölund, J. Eker, and J. Gustafsson (2021). "Towards a Holistic Controller: Reinforcement Learning for Data Center Control". In: *Proceedings of the Twelfth ACM International Conference on Future Energy Systems*. E-Energy '21. Association for Computing Machinery, New York, NY, USA, pp. 424–429. DOI: 10.1145/3447555.3466581.

This paper is the base for Chapter 5. The hypothesis is that a holistic controller that takes both room cooling and load balancing into account can improve the performance of the datacenter. We introduce a simulation model of the relevant parts of a datacenter, and define an RL agent that controls both load balancing and cooling systems simultaneously. We compare the RL agent to a standard control method, and show that the RL agent learns a policy that is more energy efficient.

The original idea was developed by all authors together, and all provided valuable feedback throughout the project. R. Brännvall provided the mathematical modelling of the DC, which was then implemented by A. Heimerson. The RL agent was designed and implemented by A. Heimerson, who also set up the experiments and collected all the data. While all authors contributed to the manuscript, A. Heimerson lead the work and produced a majority of the content together with R. Brännvall.

Paper II

Heimerson, A., J. Sjölund, R. Brännvall, J. Gustafsson, and J. Eker (2022). "Adaptive Control of Data Center Cooling using Deep Reinforcement Learning". In: *2022 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C)*. Online, pp. 1–6. DOI: 10.1109/ACSOSC56246.2022.00018.

This paper is the base for Chapter 6, and looks at extending the work from Paper I by creating a more realistic simulation of the air flow in the server hall based on [Sjölund, 2018]. An RL approach is compared to standard control methods, and is shown to be able to learn a competitive policy to control the room cooling in a context aware manner, as well as adapt to changes in the environment.

The idea is built on previous work and was developed by all authors. The connection to the fluid dynamics model of the DC as well as the thermal mass of the servers was implemented by J. Sjölund. A. Heimerson designed and implemented the RL agent, as well as designed the experiments and collected all the data. A. Heimerson and J. Sjölund wrote the majority of the manuscript, though remaining authors contributed with writing and valuable feedback.

Paper III

Heimerson, A., J. Eker, and K.-E. Årzén (2022). "A Proactive Cloud Application Auto-Scaler using Reinforcement Learning". In: *2022 IEEE/ACM 15th International Conference on Utility and Cloud Computing (UCC)*. Vancouver, Washington, USA, pp. 213–220. DOI: 10.1109/UCC56403.2022.00040.

This paper is the base for Chapter 7, where we look at the problem of smart scaling decisions in microservice applications consisting of multiple interconnected services. The idea is that simple metrics such as the load on the different services could be enough to understand the application graph, and thus be able to create proactive scaling strategies. We explore whether an RL agent can learn this from very simple data with no prior knowledge of the application or workloads, and compare the performance of the proactive RL strategy with common reactive scaling strategies.

The original idea was by J. Eker and A. Heimerson, and was further developed by all authors. Creating the microservice model was done by A. Heimerson with input from J. Eker, while the implementation of both model and RL agent was done by A. Heimerson. Setting up experiments and running them was done by A. Heimerson. The manuscript was mainly developed by A. Heimerson, with J. Eker contributing parts of the writing, and K.E. Årzén providing valuable feedback.

Paper IV

Heimerson, A., J. Ruuskanen, and J. Eker (2022). "Automatic Differentiation over Fluid Models for Holistic Load Balancing". In: *2022 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C)*. Online, pp. 13–18. DOI: 10.1109/ACSOSC56246.2022.00020.

This paper is the base for Chapter 8, and looks at the problem of load balancing in a microservice application. It introduces a general method for finding a cost optimizing controller based on an existing fluid model. By setting up a cost function that depends on values derived from the model, we can use automatic differentiation to find the gradient of the cost with respect to the controller parameters. We set up a multi-cluster microservice application, using an existing method to create the fluid model from log data. The load balancing parameters are then optimized using bounded gradient descent steps, and the approach is evaluated in a real-world setting. It is shown to reduce cost under disturbances while adhering to percentile constraints on the round-trip time.

A. Heimerson and J. Ruuskanen are both first authors and contributed equally to the work. J. Ruuskanen suggested the fluid model as a basis for a control strategy, and A. Heimerson provided the idea of using automatic differentiation. The idea was implemented by both first authors, with input from J. Eker. The experiments were conducted by both first authors, with A. Heimerson implementing the online controller and J. Ruuskanen setting up the experimental environment and modifying the online fitting of the fluid model. A. Heimerson and J. Ruuskanen contributed equally to the writing of the manuscript, with J. Eker writing smaller parts and providing valuable feedback.

Paper V

Heimerson, A. and J. Ruuskanen (2023). "Extending Microservice Model Validity using Universal Differential Equations". In: IFAC World Congress 2023. Yokohama, Japan.

This paper is the base for Chapter 9, and evaluate a method for extending the fluid model from Paper IV with an NN. The goal is to improve the accuracy of the model by learning the missing dynamics from data. Additionally, we show that imposing constraints on the NN based on understanding about the system behavior can drastically improve convergence speed of the learning process. In the end, the extended model show improved accuracy compared to the original fluid model, enabling better control decisions to be taken.

A. Heimerson instigated this project with the idea to improve on the fluid model by extending it with a NN. A. Heimerson also implemented and ran all the experiments. A. Heimerson wrote the manuscript, with J. Ruuskanen proofreading and providing valuable feedback.

Patent Application

Eker, J., A. Heimerson, and K.-E. Årzén (2023). "Device and method for scaling microservices". Pat. WO2023048609A1 (WO). Telefonaktiebolaget LM Ericsson (Publ).

The work that led to Paper III, and thus Chapter 7, also resulted in this patent application. It describes a procedure for using RL to proactively scale microservices in a distributed application.

The original idea was from J. Eker and A. Heimerson, and was further developed by all authors. The invention disclosure was written mainly by J. Eker, with feedback from A. Heimerson and K.E. Årzén. The experimental work implementing the procedure and showing the feasibility of the idea was done by A. Heimerson.

Publications Not in the Thesis

The author of this thesis has also contributed to the following publications, though they are not included in the thesis since they were deemed to be off-topic.

Soltesz, K., F. Gustafsson, T. Timpka, J. Jaldén, C. Jidling, A. Heimerson, T. B. Schön, A. Spreco, J. Ekberg, Ö. Dahlström, F. B. Carlson, A. Jöud, and B. Bernhardsson (2020). *On the sensitivity of non-pharmaceutical intervention models for SARS-CoV-2 spread estimation*. DOI: 10.1101/2020.06.10.20127324. preprint.

Soltesz, K., F. Gustafsson, T. Timpka, J. Jaldén, C. Jidling, A. Heimerson, T. Schön, A. Spreco, J. Ekberg, Ö. Dahlström, F. B. Carlson, A. Jöud, and B. Bernhardsson (2020). "Sensitivity analysis of the effects of non-pharmaceutical interventions on COVID-19 in Europe". *medRxiv*, p. 2020.06.15.20131953. DOI: 10.1101/2020.06.15.20131953.

Soltesz, K., F. Gustafsson, T. Timpka, J. Jaldén, C. Jidling, A. Heimerson, T. B. Schön, A. Spreco, J. Ekberg, Ö. Dahlström, F. Bagge Carlson, A. Jöud, and B. Bernhardsson (2020). "The effect of interventions on COVID-19". *Nature* **588**:7839 (7839), E26–E28. DOI: 10.1038/s41586-020-3025-y.

Bagge Carlson, F., M. Fält, A. Heimerson, and O. Troeng (2021). "ControlSystems.jl: A Control Toolbox in Julia". In: *2021 60th IEEE Conference on Decision and Control (CDC)*, pp. 4847–4853. DOI: 10.1109/CDC45484.2021.9683403.

2

Cloud Computing

Cloud computing is a broad concept, encompassing the technology and infrastructure that allows users to access and utilize computing resources without the need to own or manage physical hardware. It is a transformative paradigm that enables flexibility and cost-efficiency for services provided over the internet. For end users, the cloud is an increasingly prominent part of their daily routine where common services are provided online or through mobile applications.

With this increase in adoption, the energy consumption of the underlying DCs is a significant concern. According to [Koot and Wijnhoven, 2021], the total worldwide energy consumption from datacenters is expected to grow from 286 TWh in 2016, meaning around 1.15% of global energy consumption, to about 321 TWh or 1.86% in 2030. In parallel with this expected growth in the IT domain, a societal push towards a green electrification of the society is ongoing. Green electrical production from solar and wind power plants is increasing and introducing new challenges to the electric grid, as the production is difficult to control. Urbanization creates even more intensified power concentration in cities, which further challenges the power grid capacities. All in all, power availability will be an increased challenge in most places in the world. Cloud services and the underlying IT infrastructure need to find ways to reduce and adapt their energy and power demand in smart ways, to avoid expensive interruption of services and failures as well as enabling future growth. In addition to these challenges, the cloud also hosts more important infrastructure than ever before, e.g., in the form of health care, banking, and industrial control systems. The need for reliability and low latency is therefore added to the list of requirements, making for a complex multi-objective optimization problem.

This chapter provides the basics for hardware and software stacks in a cloud environment, specifically the parts that are interesting from a control perspective. Microservices are introduced as an integral part of how many web-based applications and services are built today to fully leverage the elasticity of the cloud. The chapter also provides an overview of related

work in the field of cloud control, standard methods as well as state-of-the-art algorithms and techniques.

2.1 Cloud Infrastructure

Cloud computing has introduced a shift in how internet services are hosted, and though "the cloud is just someone else's computer" is a well-used adage, there is more to the cloud than just using other peoples hardware. It provides several benefits compared to owning and managing the infrastructure in-house, and the cloud providers have become a large industry in their own right. Cloud providers typically run things at a larger scale, allowing a relatively low operational cost, both since the infrastructure surrounding the servers can be made more efficient with scale, but also since pooling resources can allow them to be more efficiently utilized over a large user base. The location of a large scale datacenter is also often motivated more by access to cheap electricity, internet and cooling, further driving the cost down. The elasticity gained from a large pool of resources that can be provisioned and scaled on-demand also gives the cloud users much flexibility compared to the much more arduous and time-consuming process of ordering and configuring new hardware when needing to scale. These are all important reasons why cloud computing became popular, though the term cloud as provided in the often cited definition from NIST [Mell and Grance, 2011] can also refer to smaller and private installations. Their definition focus more on the flexibility provided to the cloud users, and are categorized into five essential characteristics:

- *On-demand self-service* – Cloud users should be able to provision computing resources such as servers and storage automatically.
- *Broad network access* – It should be possible to access the resources using standard interfaces to allow for heterogeneous client platforms.
- *Resource pooling* – The different resources of the cloud provider are pooled to be assigned to different cloud users on demand.
- *Rapid elasticity* – It should be possible to rapidly scale resources according to demand at any time.
- *Measured service* – Cloud systems automatically collect metrics for the different services, allowing for monitoring and control by cloud provider and cloud user alike.

Using the services from a *cloud provider*, a *cloud user* can build their applications by running software on the provided infrastructure. The *end*

users are the consumers of the service, interacting with the *cloud application* through some network interface by sending *requests* that are processed by the application.

The cloud provider can offer their services at varying level of abstraction, ranging from access to virtualized hardware resources to access to providing a cloud application directly to the end user. A widely used categorization of these abstraction levels was also defined in [Mell and Grance, 2011].

- *Software as a Service* (SaaS) – Applications are provided through the cloud infrastructure to end users who have no part in managing the underlying infrastructure, e.g., a web-based email service or file synchronization service.
- *Platform as a Service* (PaaS) – The cloud user provides an application to run in existing environments, which together with the underlying infrastructure is handled by the cloud provider.
- *Infrastructure as a Service* (IaaS) – The cloud user provisions resources such as processing, storage and networks from the cloud provider, and deploys arbitrary software on the infrastructure to provide end users with some service.

Datacenters

All the energy used for the cloud is in some way consumed by the hardware that the cloud is running on, so considering a more energy efficient cloud without taking the hardware aspects into account would be negligent. With that said, this thesis is not concerned with trying to improve the hardware, but is instead interested in hardware from the perspective of control in conjunction with the software layers.

To describe how efficiently a DC uses energy, the power usage effectiveness (PUE) is commonly used in industry. It is based on the ratio of the power consumption between the full DC and only the IT equipment.

$$\text{PUE} = \frac{\text{Total DC energy}}{\text{IT equipment energy}} \tag{2.1}$$

PUE has been criticized as a metric since it is not defined well enough and can be reported with different overheads. It is also questionable that increasing the IT energy consumption should lead to a better PUE, assuming the cooling and other overheads are kept constant [Brady et al., 2013; Barroso et al., 2019]. With this said, it is still the most commonly used one and has been a driving factor in pushing the industry to improve the efficiency of their DCs.

In [Ni and Bai, 2017], 100 DCs were examined and their PUEs were found to range from 1.33 to 3.85, with an average of 2.13, thus showing an

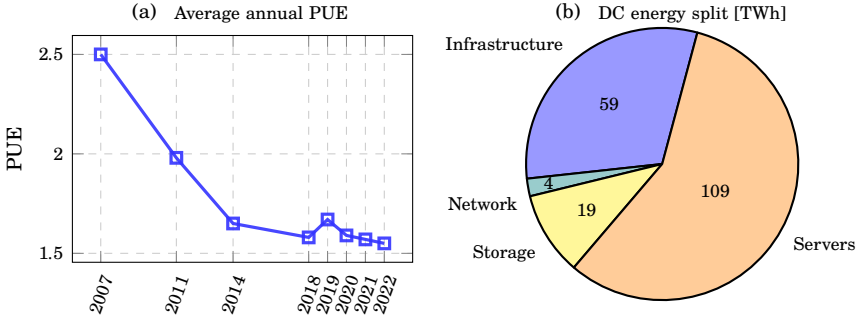

Figure 2.1 Figure 2.1(a) show average PUE based on survey results [Davis, 2022]. Figure 2.1(b) show the global energy demand in TWh from datacenters [Kamiya and Kvarnström, 2019], where all the non-IT energy usage such as cooling, power distribution systems and lighting, is included under the infrastructure slice.

overhead of more than 100%. The more recent study by [Davis, 2022] gives a slightly better outlook, though as seen in Figure 2.1(a) the improvements have drastically slowed. The main inefficiencies in non-IT infrastructure comes from power distribution system and cooling [Zhang et al., 2021; Barroso et al., 2019], which still share around a third of the total energy consumption as seen in Figure 2.1(b).

The layout of a DC can typically be divided into three parts, the server hall (IT equipment), the mechanical yard (cooling), and the electrical yard (power distribution). In Figure 2.2 we show an example of this layout, with some different components that are often found in a DC. While we will not consider all possible components and configurations in detail, it should give a brief overview of the more common setups that are relevant for the rest of the thesis.

The server hall contains cabinets called *racks*, specifically made to house IT-equipment such as servers, storage and networking equipment. A rack is mostly a frame with standardized slots for the different equipment, and can normally house around 20-50 pieces of equipment.

Power distribution units are used to distribute power to the racks, and *uninterruptible power supplies* are used to provide backup power from, e.g., diesel generators and batteries in case of power outages. Together they provide a continuous balanced supply of electricity to the DC while protecting sensitive equipment from electrical disturbances. Traditionally the power goes through a transformation where it is converted from alternating to direct current to feed into the battery circuit, and then converted back to alternating current for distribution in the datacenter. The double conversion

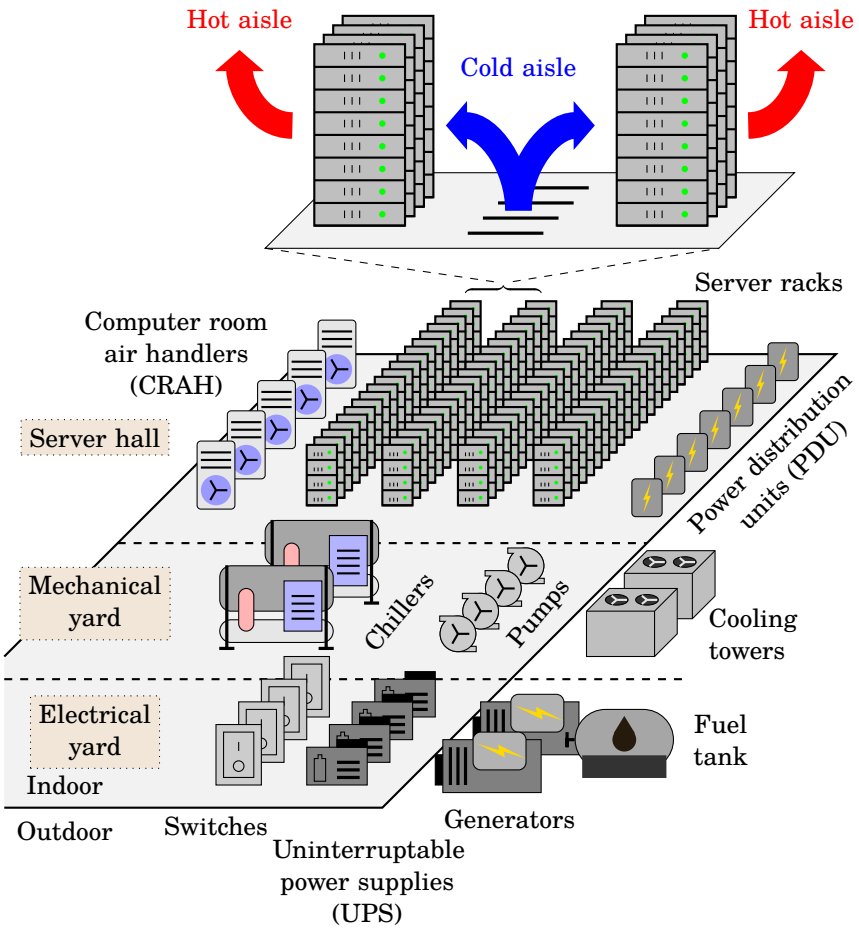

Figure 2.2 A simple sketch showing an overview of typical components related to a datacenter. At the top we see the hot and cold aisles with raised floor design, used to distribute cooling air to the racks. Below is the *server hall*, containing the IT equipment as well as parts of the cooling and power distributions systems. The *mechanical yard* contains the cooling systems with both chillers and free-cooling capabilities, while the *electrical yard* contains the power distribution systems and backup power systems.

of the electrical power provides stability against interruptions and disturbances, but comes at the cost of efficiency, and up to 15% of the energy can be lost in the conversion [Barroso et al., 2019].

The most common way of cooling the server hall is by air, providing simpler installation and maintenance compared to liquid cooling. In this case the racks are placed in rows to form *cold aisles* and *hot aisles*, see Figure 2.2. Cold air is provided from the air conditioning system, and is blown into the cold aisles where it passes through the racks and is heated up and circulated back. A common way to provide the cold air is through a raised floor system, where the cold air is blown under the floor and up through the floor tiles, and often the hot and cold aisles will use containment walls to prevent the hot air from mixing with the cold air. The cooling units help with both circulating the air in the server hall and cooling it, and depending on the type of cooling unit they are either self-contained with a built-in compressor (CRAC), or they are connected to a central cooling system (CRAH). CRAH units are usually more complex to set up compared to the CRAC, but provides more flexibility in how the cooling is done, and are therefore more common in large scale DCs. A CRAH unit is designed to blow air over chilled-water coils, where the chilled water can be supplied from a separate *chiller* unit or cold water plant. A chiller is a compressor based cooling unit that can be used to provide chilled water to the CRAH units. In *free-cooling*, cooling towers use outside air to cool the water, either through evaporation or through dry cooling. Evaporation uses an open-circuit system able to bring down the temperature more efficiently at the cost of a complex setup, while dry cooling use a closed-circuit heat exchanger which is simpler but less efficient. Free-cooling is often used in conjunction with chillers, only using the chiller to cool the water further if needed. There is also direct free-cooling, where the outside air is drawn directly into the server hall, but this is not so common due to the problem of contamination and humidity [Zhang et al., 2014]. A significant portion of the energy consumption in the cooling system comes from the chillers [Patterson, 2008], making free-cooling an important factor in reducing the cooling energy. Liquid-cooling is still an option used in specific cases, and can be provided either through cold-plates or through submerging the equipment in some non-conductive liquid. Though the heat exchange is much more efficient and require less flow and energy, it is not commonly adopted since dealing with the liquid makes for a complicated setup with a high maintenance burden.

Virtualization

One important factor of the cloud is the elasticity, i.e., the ability to dynamically scale resources up and down as needed. Scaling is often divided up into horizontal and vertical scaling, where horizontal scaling implies adding

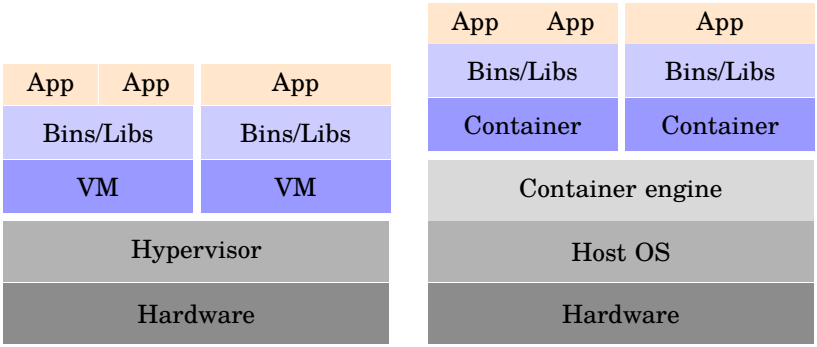

Figure 2.3 *Left:* A VM contains a virtualized version of the physical components, and runs a normal operating system on top of this to create a well isolated environment. *Right:* Containers are a more light-weight virtualization technology, sharing the host OS through a container engine.

or removing machines, while vertical scaling implies changing the amounts of available resources in an existing machine. They have different benefits, and can also be used in conjunction with each other to have the benefits from both. Vertical scaling can be easier to implement, since it does not require any changes to the application, but it is limited by the maximum amount of resources that can be added to a single machine and typically requires restarting when resized, thus causing downtime. This type of scaling is referred to as scaling up or down. Horizontal scaling needs to account for the fact that the application is running on multiple machines, requiring load balancing and state management, but can scale to a much larger extent and can scale without affecting currently running machines and thus requiring no downtime. This type of scaling is referred to as scaling out or in.

To be able to take advantage of the elasticity provided through the cloud, virtualization is a key technology used in both virtual machines (VMs) and containers to create isolated units of software that are agnostic to where they are running.

Virtual machines contain virtualized versions of the physical components such as CPU, memory and storage, and run a normal operating system on top of this to create a well isolated environment. This is visualized as the left stack in Figure 2.3, where the *hypervisor* is a layer between the VMs and the hardware. There are two types of hypervisors, type 1 running directly on the hardware, and type 2 running on top of an existing OS. The hypervisor provides an interface for VMs to interact with the hardware, and handles resource sharing between the VMs, allowing multiple VMs to run on the same physical machine. Since the physical components are virtual, they can

be configured to only use a sub-set of the available resources, and also be reconfigured over time to change the vertical scaling of the VM.

A *container* is a more light-weight virtualization technology, sharing the host operating systems (OS) through a *container engine* as seen to the right in Figure 2.3. The container engine helps keeping separate file systems, process spaces and networks stacks for each container, while leaving the host OS to handle systems calls. Thus, containers still have good isolation, though not as good as VMs, while having a smaller size. This makes them a good versioning tool for reproducible software deployments. These two technologies can also often be used together, where a VM is used to deploy the container engine to create a well managed and isolated environment for a cloud user, and the containers are used within the VM for deploying individual applications. Virtualized environments allow IT workloads to be moved between servers within, or sometimes even across, datacenters, allowing for flexibility in power and energy optimization.

Edge Computing

In recent years the number interconnected devices that communicate over the internet has grown tremendously, and connectivity seems to be added to all kinds of small devices and sensors making them part of the *Internet-of-Things* (IoT). These generate data which is often sent to some cloud for processing and storage [Naveen and Kounte, 2019], leading to large amounts of data being sent over the network.

While the cloud is typically based on large centralized datacenters, to fully be able to utilize the scalability and cost efficiencies that can be gained there, *edge computing* or *fog computing* are paradigms where the focus is to move computations closer to the user. Instead of sending all the data for storage and processing in the cloud, one could keep sensitive data locally, or pre-process data locally to send a reduced or anonymized set of it to the cloud. The computations can happen on smaller local DCs, just to reduce the latency and amount of data sent, or it can be done on private devices such as a phone or router to keep the data on devices that the user controls.

Low and predictable latency is also a driving factor in new use cases such as cloud based automation, IoT, XR (eXtended Reality), and VR (Virtual Reality), and edge computing has gained traction as a means of reducing this by bringing compute resources closer to the user. A combination of centralized cloud and edge compute nodes offers a distributed compute platform which allows for trade-offs between latency, cost, etc., and makes it possible to configure a cloud service to deliver solutions at the right service level at the right cost.

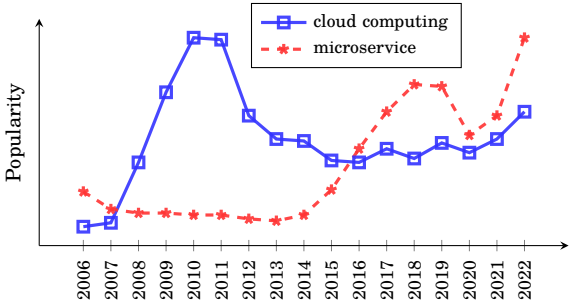

Figure 2.4 Google trends popularity [*Google Trends* 2023] for terms *cloud computing* and *microservices*. Both proportional to their own highest value, i.e., not showing relative popularity but rather how interest evolved over time.

Microservice Architecture

As the cloud keeps gaining in popularity and usage, the interest in cloud native applications, where microservices are the basic building blocks, have shown a rise in interest over recent years, see Figure 2.4.

Microservices [Cerny et al., 2018] are becoming a popular choice as the paradigm for cloud applications, as shown in the survey by [Swoyer and Loukides, 2020]. Microservices are a modern way of designing applications in the cloud, where small independent services are each responsible for some objective within a larger application. They are often built using containers, and can be deployed, tested and scaled individually of other parts of the application. This gives a large flexibility in the orchestration of individual services as well as some resiliency to failures, as a failure in one service does not necessarily affect the others. Scaling a microservice typically means deploying more instances, or *replicas*, of the service, and balancing the load between them. Microservices also mesh well with edge computing, where some parts of the application can be deployed on the edge, while others are deployed in the cloud depending on the needs of the application. Replicas of the same service can also be deployed in different locations, and the service can be configured to use the instance closest to the user. With this flexibility comes the complexity of control, and while doing local control on the individual services often suffice and can be relatively simple, looking to optimize the controllers in the global context still requires us to consider how the services interact as a whole. Software companies who employ a microservice architecture to build their applications have seen them explode in complexity, leading to a complicated interconnected system known as a *microservice death star* [Gan et al., 2019]. A small version of this is illustrated in Figure 2.5, though, real systems can have many more

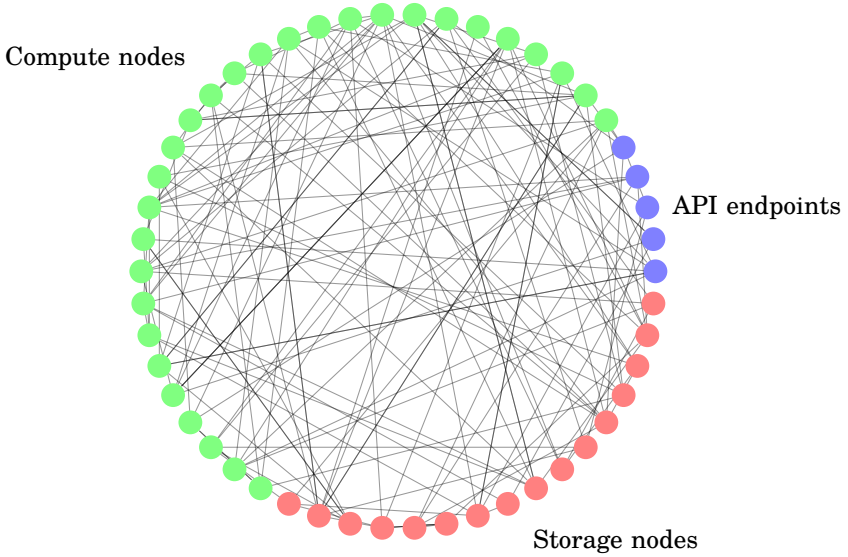

Figure 2.5 A simple illustration of why microservice applications can lead to an explosion in complexity that is known as a *microservice death star*. As the set of microservices providing API endpoints, data storage, login service, ad service, etc., continues to grow, the full system quickly becomes a complex mesh of connected services. Real examples of this from large cloud-based organizations can look much worse than this example.

services and connections.

Keeping track of all instances of services in a flexible environment where they are constantly being spawned and terminated is a challenge, and a *service mesh* can help with that. It is a dedicated infrastructure layer that provides communication and coordination capabilities for microservices within distributed applications. A service mesh is often implemented as a set of proxies alongside each microservice replica, and can be used to provide logging, service discovery, load balancing, security, routing, and more [Khatri and Khatri, 2020]. It complements the capabilities of microservices by handling networking concerns, and simplifies the management of distributed applications while improving reliability, scalability, and observability.

Common Tools

Kubernetes [*Kubernetes* 2023] (sometimes abbreviated as K8s) is a portable, extensible, open-source platform for managing containerized workloads and microservices at scale. One deployment unit in Kubernetes is called a *pod*, which is a group of one or more containers that share storage and net-

work resources. Kubernetes can easily be deployed on top of most cloud providers, as well as on-premise setups, creating a simple interface to manage cloud applications over different environments. In addition to container orchestration, Kubernetes provides some basic service-mesh features such as service discovery, load balancing, logging and autoscaling, to help with common tasks when deploying a cloud application. For more advanced features there are tools dedicated to specific tasks, such as Istio [*Istio* 2023] for service meshes and Prometheus [*Prometheus* 2023] for monitoring.

2.2 Controlling the Cloud

Applying control to the cloud is a broad topic, and there are many aspects that can be controlled, as well as many competing objectives when doing so. For a cloud provider it might be about managing the existing load on the datacenters to minimize hardware wear and energy consumption, while meeting the performance requirements of the cloud user. For a cloud user it might be about reducing how many resources are allocated to a service while still providing a service with acceptable performance to the end users. In both cases, there is a cost to providing the service, and there are performance requirements that need to be met.

Performance metrics. To quantify performance, is common to set up the requirements on the cloud service as a *service level agreement* (SLA) between cloud provider and user. The SLA is a contract that defines the expected performance of the service, and what happens when the service does not meet the requirements [Wieder et al., 2011]. The SLA can include multiple *service level objectives* (SLOs), each defining constraints on some metric that needs to be met. Which metrics are used depends on the service, but common ones include response time, throughput, uptime, etc.

Running costs. As a precaution against violating the SLOs, the service provider will typically over-provision resources. In the case of cooling control this might mean running the cooling systems at a higher capacity than what is needed to handle the current load, while for resource allocation in the cloud it might mean running more replicas than is needed to handle the current load. It will create a buffer to handle fluctuations in the workload, but it will also lead to higher costs for the service provider since they are paying for the resources. The ideal controller should be able to handle the fluctuations in the workload without violating the SLOs, while minimizing the amount of over-provisioning needed.

Cooling Control

Controlling the cooling equipment in a DC involves a number of different components, and the control strategies can vary greatly. In addition to control strategies, there are also many considerations regarding the physical infrastructure, such as the placement of the cooling equipment, the layout of the server hall, how to restrict the airflow and remove leaks.

The airflow in the server hall is mainly managed by the CRAH units, where the flow is often controlled by simple feedback loops based on keeping a constant temperature or pressure in the cold aisle. The temperature of the cooling water that is supplied to the CRAH units is typically controlled by the chillers, often using simple feedback loops to supply a constant temperature tuned to the expected load.

Operating conditions for the cooling systems can have a large impact on energy, as noted in [Barroso et al., 2019]. One example provided is that increasing the cold aisle temperature can drastically reduce the cooling energy, since the CRAH units can operate at higher temperatures and thus use less energy. Managing the airflow is also an important factor for energy efficiency, and here it can be something as simple as hot and cold aisle containment to reduce the mixing of hot and cold air. Cooling systems that are supported with free cooling capabilities also play a large role, since taking advantage of the lower outside temperatures can drastically reduce the use of chillers, and thus save a lot of energy.

While many DC operators do implement some of these improvements, others still use older strategies, where the main objective is to keep the servers cool enough to not overheat. Such strategies often resulted in simple control methods involving running the cooling systems at a constant high capacity that could handle the worst case scenario, leading to large amounts of wasted energy.

Resource Allocation

As demand comes and goes, the amount of resources allocated to a service should be adjusted accordingly. Scaling resources is not instant, and looking at horizontal scaling, adding a new container or VM typically take from a few seconds to a few minutes. If scaling happens fast in relation to the workload fluctuations, using a reactive approach, where resources are added when we notice that we are running close to capacity, is sufficient. However, if the workload changes quickly, or scaling is slow, a reactive approach will lead to either over-provisioning or under-provisioning as seen in Figure 2.6. During over-provisioning there are unused resources, wasting energy or costing money, while under-provisioning can lead to performance degradation or even service failure. For dynamic workloads, such as a stochastic behavior of users accessing some web service, the required resources can change

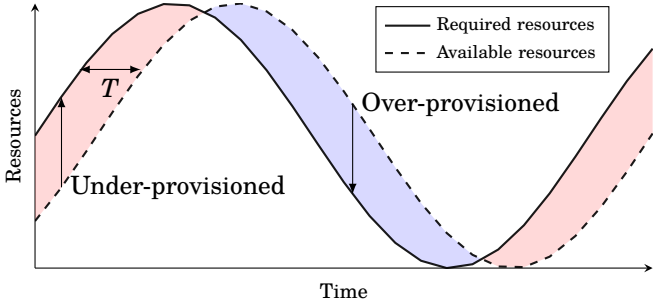

Figure 2.6 A visualization of scaling resources based on current requirements when the environment has a dynamic workload and scaling decisions take some time T to implement. It would lead to either over-provisioning (blue) or under-provisioning (red).

drastically over time.

Big cloud providers, such as Amazon Web Services, Google Cloud and Microsoft Azure, all provide some kind of autoscaling functionality for cloud resources. While vertical scaling do exist, it is typically horizontal scaling that is used, where the number of instances of a service is increased or decreased. While the implementation of automatic scaling will vary, the general idea typically comes down to following some target metric, e.g., CPU utilization, network traffic or other metrics, by reactively scaling resources based on manually defined thresholds of the metric. On top of this there might be a hysteresis mechanism to avoid jitter, since scaling decisions are not instant and thus typically incur a cost on the system. In some cases, predictive scaling methods using time-series forecasting are also offered, where the scaling decisions are based on predictions of the future workload.

Kubernetes also comes with a built-in *horizontal pod autoscaler* (HPA) [*Kubernetes HPA* 2023], implementing a simple and configurable solution for reactive scaling of resources. First it calculates a target number of replicas $\hat{r}_t$ based on the utilization metric value u_t, the target utilization value $\bar{u}$ and the current number of replicas r_t. If the relative change between the current and target replicas is less than some ϵ, it is discarded in favor of the old value to help to avoid jitter in the signal.

$$\hat{r}_t = \begin{cases} \lceil r_t \frac{u_t}{\bar{u}} \rceil & \text{if } \left|1 - \frac{u_t}{\bar{u}}\right| > \epsilon \\ \hat{r}_{t-1} & \text{else} \end{cases} \tag{2.2}$$

To avoid scaling in too fast, the maximum over a time-window W of previous target replica amounts is selected as the new replica amount.

$$r_{t+1} = \max_{k \in [0,W]} \hat{r}_{t-k} \tag{2.3}$$

Load Balancing and Scheduling

Load balancing and scheduling are closely related concepts, and in the context of managing workloads in a distributed environment, load balancing can be seen as a sub-problem of scheduling. While scheduling can entail decisions about both when and where to execute different tasks and what resources they should be assigned, load balancing focus mostly on the where, aiming to distribute the incoming load over existing resources according to some strategy. With the emergence of flexible cloud resources that are resized and moved around on the fly, these problems have become more complex and important than ever before.

Load balancing is a well studied problem, and the choice of strategy depends on the characteristics of the workload and the system as well as the desired outcome [Wang and Morris, 1985]. Some of the more common ones are *round-robin* (RR), *random* and *join-shortest-queue* (JSQ), as well as their weighted counterparts [Sharma et al., 2008; Lee and Jeng, 2011]. JSQ has in general better performance than the other two, and it can be near optimal in minimizing the total mean response time depending on the queue type [Gupta et al., 2007]. The problem, however, is that it requires knowledge about the current state in each instance at each decision, and is thus cumbersome to implement at scale. To handle this state knowledge drawback, improvements such as *power-of-d* [Azar et al., 1994; Bramson et al., 2010], *join-idle-queue* [Lu et al., 2011] and *join-the-best-queue* [Spicuglia et al., 2013] have been suggested.

While research in scheduling has traditionally been more tied to the operating system, the emergence of cloud computing made for a new domain with a slightly different set of challenges. The scheduling problem in the cloud is more complex since the resources are highly dynamic, and decisions are made on a higher level, though many of the same strategies can still be used. Some of the more common ones are *first-come-first-serve* (FCFS), *shortest-job-first* (SJF) and *shortest-remaining-processing-time* (SRPT) [Leung, 2004]. How to choose between these depend on what the objective is, where FCFS might be considered more fair, and SRPT typically provide a lower mean response time. The problem is similar to load balancing in that the more informed decisions require more information, where information such as remaining processing time is not always available.

3

Reinforcement Learning

> "Reinforcement learning empowers autonomous agents to learn, optimize, and decide, driving intelligent decision-making in complex environments"
>
> ChatGPT by OpenAI

This chapter is meant to give the reader a small introduction into the basics of learning-based control, specifically methods related to reinforcement learning (RL). While this should cover the concepts required to understand the rest of the thesis, it is by no means a complete coverage of the field, and many topics are only briefly touched upon or omitted entirely.

Starting with a small historical view of the top level concepts, it shows where RL fits in a larger picture of different methods for control. The ideas are then presented in approximately chronological order, describing important advancements and what each of them contributed to the field. Then Chapter 4 presents practical challenges that one is faced with when using RL, and how RL has previously been applied in the context of controlling cloud infrastructure.

3.1 Introduction

The idea of learning by trial-and-error is something most of us are familiar with, and it is something we do throughout our lives. By observing the effect of our actions, we can improve our future decisions. What we today know as reinforcement learning stems from a few different fields, where some of the earlier work comes from psychology with trial-and-error learning in animals [Sutton and Barto, 2018].

Another strong connection is found in the theoretically justified field of optimal control, with methods such as dynamic programming (DP) [Bellman, 1954] giving general strategies for solving problems containing certain structure. The Markov Decision Process (MDP) [Bellman, 1957] came around the same time, defining a general framework to model sequential decision-making problems with full information. Full information is not always feasible, and [Åström, 1965] describes an extension in the case of incomplete information, later to be called Partially Observable MDPs (POMDPs) and used in a variety of planning and machine learning (ML) applications.

Reinforcement learning as a field emerged a little while after that, and it was soon realized that the framework for optimal control, such as MDPs and DP, could be used in the context of RL as well. Optimal control is commonly considered when both model and cost function are known, though if either of them are not available the problem grows considerably harder. RL does not assume knowledge of either model or cost function in general, though there are certainly methods that incorporate more knowledge when available. [Sutton et al., 1992] frames RL as a computationally inexpensive approach for direct adaptive optimal control. *Direct* meaning that it is learning the policy by interacting with the actual environment instead of *indirect* which would be optimizing through a model.

Another discipline that shares similarities with RL is the field of adaptive control. [Åström and Wittenmark, 2008] defines an adaptive controller as "a controller with adjustable parameters and a mechanism for adjusting the parameters", a rather broad definition that could incorporate RL as well. Adaptive control tends to take a more structured approach, where the controller is designed to have certain properties and the parameters are adjusted to keep those properties as the environment changes. RL, on the other hand, does not really have adaptivity as its main goal, but rather achieve it as a byproduct of continuously learning while interacting with the environment.

While supervised learning is a popular learning-based approach, it is targeted for tasks where the desired output is known, and the learning process simply fits a function from input to output. This is not the case in RL, where the desired output is unknown, and the learning process must both figure out what the desired output is and learn a function to produce it.

Top level view. Figure 3.1 shows the typical interaction model used in RL. An *agent* observes the *state* of the *environment*, and decides on what *action* to take using some internal *policy*. The goal of the agent is to maximize the *reward* it receives from the environment, and it does so by updating its policy to try to improve the actions it takes.

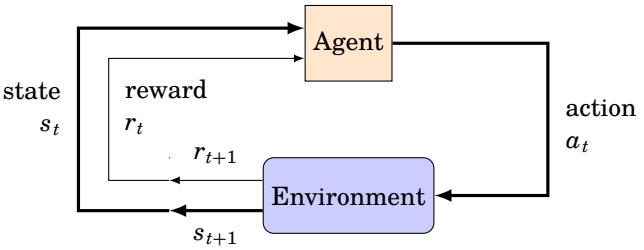

Figure 3.1 The interaction model commonly used in RL.

To make this more concrete, imagine a person trying to learn to juggle some balls. The agent is the person, or more accurately their mind, while the environment is the world around them, including the balls and the gravity, as well as their body. The goal is to keep juggling without dropping balls, which the reward should reflect by being positive when it is going well and negative if a ball is dropped. Each step involves the person observing the position and velocity of the balls and their own hands, and trying to choose how to best move their hands to avoid dropping the balls. If a ball is dropped, the person will likely try to adjust their policy to avoid making the same mistake again. Assuming they observed that there was little time to catch and throw the balls, a change in policy could be to try and throw the balls a bit higher to give them more time to react.

3.2 Markov Decision Processes and Dynamic Programming

A Markov decision process is a classical mathematical framework used to model sequential decision-making problems in uncertain environments, and is the standard formalization of the interaction between an agent and environment in the field of reinforcement learning (RL).

An MDP is a stochastic model that can be represented by the tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{T} \rangle$, where

- $\mathcal{S}$ is the set of states;
- $\mathcal{A}$ is the set of actions; and
- $\mathcal{T} : \mathcal{S} \times \mathcal{A} \times \mathbb{R} \times \mathcal{S} \to [0, 1]$ defines the transition probabilities of the environment as $\mathcal{T}(s, a, r, s') = \mathbb{P}(s_{t+1} = s', r_{t+1} = r \mid s_t = s, a_t = a)$, giving the conditional probability of ending up in state s' with reward r after taking action a from state s.

MDPs follow the Markov property, stating that the future is independent of the past given the present. This requires that the state space is *fully observable*, and that the transition probabilities only depend on the current state and action. So based on a state, the action can be selected to affect the next state and reward, and through affecting the next state it also affects future states and rewards. Selecting an action to maximize the expected future reward thus involves dealing with a trade-off; taking an immediate reward, or taking actions that might lead to higher rewards in the future.

A trajectory $\tau = \langle s_0, a_0, r_1, s_1, a_1, \ldots, s_N, r_N \rangle$ is a sequence of states, actions and rewards acquired from consecutive interactions between an agent and the environment. Sequences of values related to a single transition are denoted as *experiences*, $\langle s, a, r, s', a' \rangle = \langle s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1} \rangle$. The goal of the agent is to maximize the return G_t, the discounted sum of the received rewards, starting at the state s_t and following some policy π.

$$G_t = \sum_{k=t}^{\infty} \gamma^{k-t} r_{k+1} \tag{3.1}$$

The discount factor $\gamma \in (0, 1]$ is used to weigh future rewards against immediate rewards. For *episodic* environments, i.e., where the environment terminates after a finite number of steps, the rewards are equal to zero after the terminating step. For *continuing* environments, i.e., where the agent continues to interact with the environment indefinitely, the discount factor must be less than one to ensure that the return is finite. Using $\gamma < 1$ will affect the optimal policy, creating a more greedy approach that prioritizes immediate rewards over future rewards, and the implications of this should be considered based on the problem at hand.

As an example we introduce the MDP in Figure 3.2, which defines a simple environment with states $\mathcal{S} = \{S_1, S_2\}$ and actions $\mathcal{A} = \{A_1, A_2\}$. A_1 is deterministic and will always stay in the current state, while A_2 will attempt to switch state, and randomly succeed based on the transition probabilities. Some transitions give a reward, we have a reward of 10 for taking action A_2 from state S_1 and ending up in S_2 and a reward of 1 for taking action A_1 from state S_2. The optimal policy of this simple environment is not hard to calculate since there are only two possible non-zero rewards, each requiring different actions to be taken. So for the smaller reward of 1 we should take action A_1 from state S_2, resulting in a reward of 1 for each step. For the larger reward of 10 we should take action A_2 from state S_1. This action has two possible outcomes, either we succeed in switching state and receive a reward of 10 but are not in S_1 anymore, or we fail and stay in the same state with a reward of 0. The expected number of steps between each reward is given by the sum of the expectations for the two required transitions, $S_1 \to S_2$ and $S_2 \to S_1$, where individual

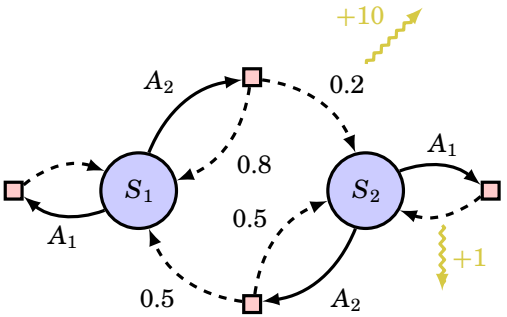

Figure 3.2 A simple MDP with states $\mathcal{S} = \{S_1, S_2\}$ and actions $\mathcal{A} = \{A_1, A_2\}$. A_1 is deterministic and will always stay in the current state, while A_2 will attempt to switch state, and randomly succeed based on the transition probabilities. Some transitions give a reward, we have a reward of 10 for taking action A_2 from state S_1 and ending up in S_2 and a reward of 1 for taking action A_1 from state S_2.

expectations are given by the inverse of the transition probabilities.

$$\mathbb{E}[S_1 \to S_2 \to S_1] = \frac{1}{0.2} + \frac{1}{0.5} = 7$$

As this gives an average reward per step of $10/7 \approx 1.43$, which is higher than the average reward per step of 1 for taking action A_1 from state S_2, action A_2 is the optimal action for accumulating the largest reward over time.

For a more structured approach to solve this we can instead turn to dynamic programming (DP) to find the optimal policy. Dynamic programming is a group of algorithms designed to find optimal policies given a perfect model of the environment in the form of an MDP. The key idea of DP is to structure the search for policies around the notion of value functions [Sutton and Barto, 2018]. A state-value function represents the expected future reward from some state s following some policy π, and is denoted $V^\pi(s)$. It can be defined based on the expectation of the return from (3.1), or equivalently by forming the *Bellman equation*, a recurrent relationship between

the value of the current state and a future state.

$$\begin{aligned} V^{\pi}(s_t) &= \mathbb{E}_{\pi,\mathcal{T}}\left[G_t\right] \\ &= \mathbb{E}_{\pi,\mathcal{T}}\left[\sum_{i=0}^{\infty} \gamma^i r_{t+i+1}\right] \\ &= \mathbb{E}_{\pi,\mathcal{T}}\left[r_{t+1} + \gamma \sum_{i=1}^{\infty} \gamma^{i-1} r_{t+i+1}\right] \\ &= \mathbb{E}_{\pi,\mathcal{T}}\left[r_{t+1} + \gamma V^{\pi}(s_{t+1})\right] \end{aligned} \tag{3.2}$$

The notation $\mathbb{E}_{\pi,\mathcal{T}}$ indicates that the expectation is over the action distribution $a_t \sim \pi(s_t)$, and the transition distribution $(s_{t+1}, r_{t+1}) \sim \mathcal{T}(s_t, a_t)$.

If the transition distribution $\mathcal{T}$ is known, we can use it to create an estimate $\hat{V}$ of the value function using a method called *iterative policy evaluation*. It works by iteratively updating the estimated values using the Bellman equation as an update rule, and can be shown to converge as long as $0 < \gamma < 1$ [Sutton and Barto, 2018].

$$\hat{V}_{k+1}(s) = \mathbb{E}_{\pi,\mathcal{T}}\left[r + \gamma \hat{V}_k(s')\right],$$

Turning to *policy improvement*, we instead update the policy based on the optimal actions for the current value function.

$$\pi(s) = \arg\max_a \mathbb{E}_{\mathcal{T}}\left[r + \gamma \hat{V}(s')\right] \tag{3.3}$$

The process of iteratively doing evaluation and improvement is called *policy iteration* [Sutton and Barto, 2018].

The optimal value function V^*, i.e., the value function assuming the optimal actions are taken, can be stated using the Bellman optimality equation.

$$V^*(s) = \max_a \mathbb{E}_{\mathcal{T}}\left[r + \gamma V^{\pi}(s')\right]$$

The optimal policy π^* can then be found according to (3.3).

A related value function is the action-value function $Q^{\pi}(s, a)$, denoting the expected value based on starting in state s_t and taking action a_t, and after that following policy π. It can easily be defined in terms of the state-value function, where the only difference from (3.2) is that the expectation is not over the action since it is already given.

$$Q^{\pi}(s, a) = \mathbb{E}_{\mathcal{T}}\left[r + \gamma V^{\pi}(s')\right]$$

The optimal action-value function is then defined as

$$Q^*(s, a) = \mathbb{E}_{\mathcal{T}}\left[r + \gamma \max_{a'} Q^*(s', a')\right],$$

and can for finite actions spaces be used to find the optimal policy as

$$\pi(s) = \max_a Q^*(s, a). \tag{3.4}$$

When the state and action spaces are finite, it allows for the use of *tabular* methods, where the value functions store individual values for each possible state, or state-action pair. Much of the theory of RL is built around these tabular methods, but for real-world problems it is often not feasible as state and action space are either just large, or infinite in the case of continuous variables. Dealing with infinite state and action spaces is often done by approximating the value function using a function approximator, and is covered in Section 3.3.

The MDPs also only treat the fully observable case, also something that does not hold for many real-world scenarios. To extend the definition to hold for *partially observable* problems, we need to include a latent observation of the state by creating the new tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \Omega, \mathcal{O} \rangle$ where

- $\mathcal{S}, \mathcal{A}, \mathcal{T}$ are the same as for the MDP;
- Ω is the set of possible observations; and
- $\mathcal{O} : \Omega \times \mathcal{S} \to \mathbb{R}$ is the *observation function* where $O(o, s) = \mathbb{P}(o_t = o \mid s_t = s)$ is the probability of observing o when the agent is in state s.

This complicates the problem since the agent does not know which state it is in, and can only generate a belief distribution based on the observation. This belief is a distribution over $\mathcal{S}$, and using the belief as the state, the POMDP can be reformulated as an MDP with infinite state and action spaces. Thus, it can be approached using the same methods as for MDPs with infinite state and action spaces, typically involving function approximation which is covered in Section 3.3.

3.3 Model-Free Reinforcement Learning

Methods that require the transition probabilities $\mathcal{T}$ are not always applicable, since in many cases there is no good model of the environment readily available. Instead, we can learn the value or policy function directly from collected experiences, making the function fit the observed data.

Monte Carlo Methods

Monte Carlo methods are a class of methods that learn directly from episodes of experience without any prior knowledge of the environment. It is one of the easiest ways to learn from experience, and is based on the idea of averaging sample returns to estimate the value function. One way

to make sure the returns are well-defined based on a finite trajectory is to assume that the environment is *episodic*, i.e., that the environment terminates in a finite number of steps. Given a trajectory τ of data collected up to the termination of the environment, the value function can be estimated for some state s as the average over returns G_t for t where $s_t = s$. This creates a noisy learner, though as the number of sample trajectories increase the estimate will converge to the true value function [Sutton and Barto, 2018].

Temporal Difference Learning

Temporal-difference (TD) learning is a class of incremental learning procedures that was introduced and analyzed in [Sutton, 1988]. Instead of learning by minimizing the error between the prediction and the observed outcome, e.g., Monte Carlo methods, TD methods use the error between the current and temporally successive predictions. This allows for estimating the expected return, the learning *target*, without needing the full trajectory, thus enabling the method to be used in both episodic and continuing environments. The simplest and most prevalent TD method is known as one-step TD, using the one-step return as the target for the value function.

$$\begin{aligned} G_{t:t+1} &= r_{t+1} + \gamma \hat{V}^{\pi}(s_{t+1}) \\ &= r + \gamma \hat{V}^{\pi}(s') \end{aligned}$$

The TD-error is defined as the difference between the target and the current estimate of the value function,

$$\delta = r + \gamma \hat{V}^{\pi}(s') - \hat{V}^{\pi}(s), \tag{3.5}$$

and is used to update the value estimate, weighted by the learning rate α.

$$\hat{V}^{\pi}(s) \leftarrow \hat{V}^{\pi}(s) + \alpha\delta$$

While (3.2) show that $r + V^{\pi}(s')$ is an unbiased estimate for $V^{\pi}(s)$, bootstrapping the estimates with $\hat{V}^{\pi}(s')$ introduces bias in the updates, and the target values will not be stationary as they depend on the current estimate of the value function. Though, with enough steps it will still converge to the true value function for π, and is in general faster than equivalent Monte Carlo methods [Sutton and Barto, 2018].

Instead of bootstrapping on a single step, we can generalize the return by taking n steps before using the value function estimate, thus getting an estimate that depends on n actual observations of the reward. This is known as the n-step return, and is defined as

$$G_{t:t+n} = r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^{n-1} r_{t+n} + \gamma^{n} \hat{V}^{\pi}(s_{t+n}).$$

For $n = 1$ we have the biased one-step returns method described above, while as n is increased the updates become less biased though having higher variance, approaching the return G_t used in Monte Carlo methods.

To balance the bias and variance of the possible returns, G_t^λ defines the return as a weighted average over all n-step returns.

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

Using G_t^λ for updating the value function creates a spectrum of methods called TD(λ). As λ is varied between 0 and 1, the methods range from TD(0) being biased by bootstrapping on a single step, to TD(1) being unbiased by using the full return G_t as the target.

Some well known methods based on TD-learning are SARSA [Sutton and Barto, 2018] and Q-learning [Watkins, 1989], both learning the action-value function rather than the state-value function. SARSA learns *on-policy*, and as such learns the action-value function for the policy used to collect the data for training. Q-learning is an *off-policy* algorithm, and the policy used to collect data does not really matter since it learns the optimal action-value function Q^*. Both are based on TD(0), though both can also be extended by using TD(λ) instead, providing a more complex estimate of the return.

Q-Learning

As an off-policy TD based algorithm, Q-learning [Watkins, 1989] became one of the early breakthrough in RL. Each update of the Q-table uses an experience tuple $\langle s, a, r, s' \rangle$, and is done similarly to TD(0).

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s', a') - Q(s,a) \right],$$

This algorithm directly tries to approximate the optimal action-value function Q^*, independent of which policy is being followed. To guarantee convergence we need to fulfill two criteria, (i) all state-action pairs are visited infinitely often, and (ii) the learning rate α_t decreases with time, but not too quickly. The formal definition of *not too quickly* is that the sum of all α_t is infinite, but the squared sum is finite [Watkins and Dayan, 1992]. A common approach to enforcing enough exploration is the epsilon-greedy policy, where the agent selects the action that provides the highest action-value estimate with probability $1-\epsilon$, and selects a random action with probability ϵ.

$$\pi(s) = \begin{cases} \arg\max_a Q(s,a) & \text{with probability } 1-\epsilon \\ a \in_R \mathcal{A} & \text{with probability } \epsilon \end{cases} \tag{3.6}$$

To show how Q-learning works in detail we look at the system in Figure 3.2, where there are two states and two actions, resulting in a Q-table with four entries. Starting with the initial value estimates as zero, learning rate $\alpha = 0.1$, discount factor $\gamma = 0.9$ and exploration factor $\epsilon = 0.05$, we can see how the table is updated as the agent interacts with the environment.

$$Q(s, a) = \begin{array}{c|c|c} & A_1 & A_2 \\ \hline S_1 & 0 & 0 \\ \hline S_2 & 0 & 0 \end{array}$$

Starting in state S_1, the policy goes the greedy route, though since both values are the same we randomize between the actions and take A_2. We are lucky and end up in S_2 with a reward of 10, leading to the table update

$$Q(S_1, A_2) \leftarrow 0 + 0.1 * (10 + 0.9 * 0 - 0) = 1 \Rightarrow \begin{array}{c|c|c} & A_1 & A_2 \\ \hline S_1 & 0 & 1 \\ \hline S_2 & 0 & 0 \end{array}$$

The policy acts greedily, but with the same value it randomly picks A_1, ending up in S_2 with a reward of 1.

$$Q(S_2, A_1) \leftarrow 0 + 0.1 * (1 + 0.9 * 0 - 0) = 0.1 \Rightarrow \begin{array}{c|c|c} & A_1 & A_2 \\ \hline S_1 & 0 & 1 \\ \hline S_2 & 0.1 & 0 \end{array}$$

The policy acts greedily and picks A_1, ending up in S_2 with a reward of 1.

$$Q(S_2, A_1) \leftarrow 0.1 + 0.1 * (1 + 0.9 * 0.1 - 0.1) = 0.199 \Rightarrow \begin{array}{c|c|c} & A_1 & A_2 \\ \hline S_1 & 0 & 1 \\ \hline S_2 & 0.199 & 0 \end{array}$$

The policy explores and randomly picks A_2, ending up in S_1 with no reward.

$$Q(S_2, A_1) \leftarrow 0 + 0.1 * (0 + 0.9 * 1 - 0) = 0.09 \Rightarrow \begin{array}{c|c|c} & A_1 & A_2 \\ \hline S_1 & 0.09 & 1 \\ \hline S_2 & 0.199 & 0 \end{array}$$

With more updates the estimates will improve, though since both policy and transitions are stochastic, the estimates will not converge to the optimal action-value function Q^* with this constant learning rate. We need the learning rate to approach zero, and using, e.g., $\alpha_t = \alpha/t$ it will converge as $t \to \infty$, approaching the optimal action-value function Q^*.

$$Q^*(s, a) \approx \begin{array}{c|c|c} & A_1 & A_2 \\ \hline S_1 & 13.56 & 15.07 \\ \hline S_2 & 12.10 & 12.33 \end{array}$$

Here we see that the optimal action is always to pick A_2, since it has the highest value in both states.

This makes for a simple learning algorithm that is easy to implement and understand. However, it is not very efficient as it requires many samples to converge, and it does not scale well to larger state spaces.

Function Approximation

Up until now the value functions have been simple lookup tables, with a value for each state, or state-action pair. Storing the table and visiting all the states becomes infeasible as the state space grows large, or impossible if there are continuous states. Instead of using a table, we want a function approximator that provides a continuous approximation of the function based on sparse observations. This will provide an approximate representation that is more space efficient, and that allows generalizing about unseen states.

The results from the tabular case for finite MDPs are no longer valid when using function approximators, and we can no longer expect to converge to the real value function or the optimal policy in general. Extending the theory to function approximation is an active field of research, and while there are some results for specific algorithms and settings [He et al., 2021; Dann et al., 2022], there is no general theory yet. Function approximation can still work in practice, but without any guarantees of converging to the true value function, and with a much less stable learning process that requires careful tuning of hyperparameters. Introducing function approximation also has the benefit of making RL more applicable to partially observable problems [Sutton and Barto, 2018].

Neural Networks

Though there are many approaches to function approximation, the most popular one in recent years is the artificial neural network (NN), driving many recent successes in deep learning. NNs are popular because they are very flexible function approximators, able to fit complex and high-dimensional functions with a relatively low amount of parameters thanks to the hierarchical learning structure in NNs [Goodfellow et al., 2016; Bengio, 2009].

In Figure 3.3 we see an NN with the input layer as well as two of the hidden layers drawn out. For each layer $\boldsymbol{z}_i$, the value from the previous layer $\boldsymbol{z}_{i-1}$ is propagated as a combination of an affine transform, and a non-linear *activation function* σ. The affine transform consists of a matrix $\boldsymbol{W}_i$ defining a linear map, and a vector $\boldsymbol{b}_i$ defining a translation. So for an input z_0 we can define z_i for $i > 0$ as

$$\boldsymbol{z}_i = \sigma(\boldsymbol{W}_i \boldsymbol{z}_{i-1} + \boldsymbol{b}_i). \tag{3.7}$$

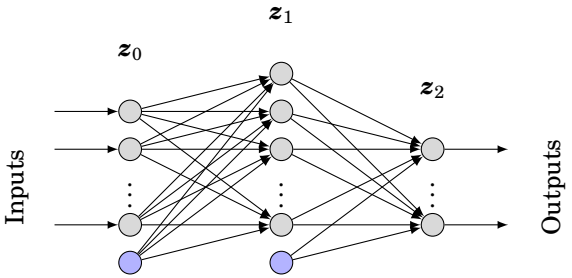

Figure 3.3 A layer consists of a number of values z_i in gray. The links between the layers consists of weights W_i (from gray nodes) and biases b_i (from blue nodes). The input propagates through the network according to (3.7).

The universal approximation theorem [Hornik et al., 1989] tells us that NNs can approximate any function to a given precision with enough parameters, though this is also true for many other types of function approximators. This even holds for a single layer, though stacking multiple layers after each other is where the real power of *deep* NNs comes out. In [Poggio et al., 2017] they discuss why deep NNs can deal with the *curse of dimensionality* for compositional functions, meaning the numbers of parameters needed to approximate higher dimensional functions grows at a *reasonable* pace.

To make notation easier, all the trainable parameters of a NN are typically collected into a single vector, e.g., $\boldsymbol{\theta} = [\boldsymbol{W}_i, \boldsymbol{b}_i]_{i=1}^{L}$ for an NN with L layers. The parameters $\boldsymbol{\theta}$ for the NN are updated by optimizing over a loss function J, which can, e.g., relate the current value function $\hat{V}_{\boldsymbol{\theta}}$ to the real value function V^{π}, evaluated over some data $\mathcal{D}$.

$$J(\boldsymbol{\theta}) = \frac{1}{2|\mathcal{D}|} \sum_{s \in \mathcal{D}} \left(V^{\pi}(s) - \hat{V}_{\boldsymbol{\theta}}(s) \right)^2, \tag{3.8}$$

For NNs it is common to use first order methods for optimization, since the number of parameters is typically large, and second order methods are computationally expensive. Using standard gradient based updates, the parameters are updated proportionally to the negative gradient using some learning rate α.

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \alpha \nabla_{\boldsymbol{\theta}_k} J(\boldsymbol{\theta}_k), \tag{3.9}$$

The update based on (3.8) then becomes

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \alpha \frac{1}{|\mathcal{D}|} \sum_{s \in \mathcal{D}} \left(V^{\pi}(s) - \hat{V}_{\boldsymbol{\theta}_k}(s) \right) \nabla_{\boldsymbol{\theta}_k} \hat{V}_{\boldsymbol{\theta}_k}(s). \tag{3.10}$$

The true function V^π that we aim to learn can in turn be estimated from data using different methods, e.g., based on Monte Carlo or temporal difference techniques.

Optimization algorithms. To make the gradient updates more efficient, it is common to use some variant of stochastic gradient descent. Standard stochastic gradient descent implies that each gradient update is based on a single data point, resulting in a stochastic approximation of the true gradient. What is more common is the case of *mini-batch* gradient descent, where each update is based on a subset $\mathcal{D}$ of the data, and the size of $\mathcal{D}$ is a hyperparameter. The benefit of both these methods are however similar. Using a smaller subset of the data can create more computationally efficient updates, and the stochasticity introduced can improve the learning process by both not getting stuck in local minima, and by adding some noise to the gradient, which can help in reducing overfitting. One of the more popular optimization algorithms based on stochastic first order gradients is Adam (Adaptive Moment Estimation) [Kingma and Ba, 2017], an adaptive method that adjusts the learning rate for each parameter based on the history of the gradient updates.

Activation functions. The choice of activation function σ is important for the performance of the NN [Sharma et al., 2020], where a few common options are shown in Figure 3.4. The choice of activation function can have a large impact on training, and common functions such sigmoid and tanh can have the problem of *vanishing gradients*, resulting in very slow learning. It happens since the functions have a small derivative for values not close to zero, and the gradient of a deep network will contain the product of many of these derivatives, resulting in a very small gradient. Rectified Linear Unit (ReLU) effectively avoids this problem through having one side be linear, though it can instead cause *dying neurons* from having exactly zero gradient for negative values, causing the neuron to never activate again. Leaky ReLU and exponential linear unit (ELU) [Clevert et al., 2016] are two approaches aiming to mitigate both these problems by having a non-zero gradient for negative values, while keeping the linear behavior for positive values.

TD-Gammon

In [Tesauro, 1992], TD(λ) is used to create a backgammon playing agent called TD-Gammon. From the board state, as well as some hand-crafted features based on expert knowledge, it uses an NN to estimate the expected value. Evaluating all possible actions and states two steps forwards, it selects the optimal action based on the value function estimate of all these states. Training by self-play allowed it to find unorthodox strategies that had a large impact on the backgammon community of the time, and

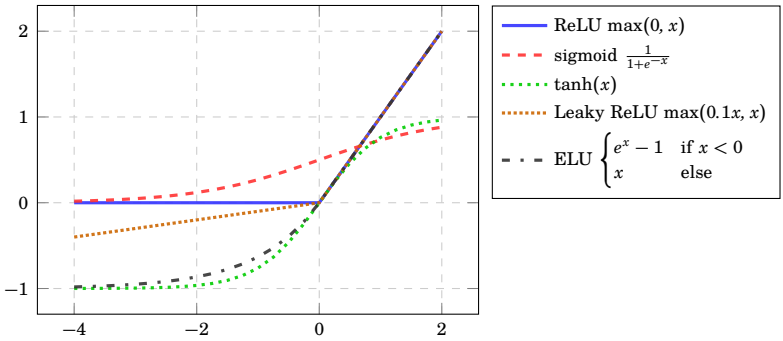

Figure 3.4 Common activation functions for NNs.

it substantially outperformed previous attempts at backgammon playing agents.

The success of TD-Gammon led to an increased interest in using NNs in RL, leading to further research in this area.

Deep Q-Learning

Another breakthrough came with Deep Q-learning in [Mnih et al., 2013; Mnih et al., 2015], where deep convolutional networks were used to learn the optimal action-value functions of different Atari games from high-dimensional pixel data. This was an influential paper which showed the viability of deep neural networks for difficult control tasks in environments with large and complex state and popularizing methods such as experience replay and target networks.

The approach is based on Q-learning as an off-policy method, allowing for learning from any recorded data, not necessarily recorded by the current policy. From this they employ experience replay, taking tuples of experiences $\langle s, a, r, s' \rangle$ from recorded data in randomized batches when training. This reduces the variance of updates since successive updates are not correlated anymore, improving the learning process.

Training the action-value function is done using updates similar to (3.10), with the gradient

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \sum_{s,a,r,s' \in \mathcal{D}} \left(r + \gamma \max_{a'} \hat{Q}_{\bar{\boldsymbol{\theta}}}(s', a') - \hat{Q}_{\boldsymbol{\theta}}(s, a) \right) \nabla_{\boldsymbol{\theta}} \hat{Q}_{\boldsymbol{\theta}}(s, a).$$

Here $\bar{\boldsymbol{\theta}}$ is a copy of the parameters that are only updated every few steps to follow $\boldsymbol{\theta}$, i.e., they will not affect the gradient, and the target value $r + Q_{\hat{\boldsymbol{\theta}}}(s', a')$ is constant for the same data. This is used in many following

works and known as *target networks*, keeping the target stationary in an effort to make the updates more stable.

They use an ϵ-greedy (3.6) policy to balance the *exploration* and *exploitation* of the agent, where ϵ is annealed over time to reduce the exploration as the agent learns.

Policy Optimization

So far the methods for control have been based on either action-value functions with a finite number of actions, or state-value functions where we know the dynamics of the system. Policy optimization allows us to learn a parameterized policy, that can output either discrete or continuous values, or probability distributions, and select actions without relying on a value function.

The basic idea is to have some performance measure $J(\boldsymbol{\theta})$ with respect to the parameters of the policy, e.g.

$$J(\boldsymbol{\theta}) = \mathbb{E}\left[V^{\pi_{\boldsymbol{\theta}}}(s_0)\right], \tag{3.11}$$

where increasing the value of J means we have improved the parameters of the policy. This assumes an episodic environment, with some starting position s_0, and a finite episode length.

The policy gradient theorem [Sutton and Barto, 2018] states that the gradient of (3.11) is proportional to

$$\begin{aligned}\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) &\propto \sum_s \mu(s) \sum_a Q^{\pi}(s,a) \nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a \mid s) \\ &= \mathbb{E}_{\mu}\left[\sum_a Q^{\pi}(s,a) \nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a \mid s)\right]\end{aligned} \tag{3.12}$$

where $\mu(s)$ is the on-policy state distribution.

In [Williams, 1992] REINFORCE is introduced, a policy gradient method using the Monte Carlo estimate for the episodic return.

$$\begin{aligned}\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) &\propto \mathbb{E}_{\mu}\left[\sum_a \pi_{\boldsymbol{\theta}}(a \mid s) Q^{\pi}(s,a) \frac{\nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a \mid s)}{\pi_{\boldsymbol{\theta}}(a \mid s)}\right] \\ &= \mathbb{E}_{\mu}\left[\sum_a \pi_{\boldsymbol{\theta}}(a \mid s) Q^{\pi}(s,a) \frac{\nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a \mid s)}{\pi_{\boldsymbol{\theta}}(a \mid s)}\right] \\ &= \mathbb{E}_{\mu,\pi}\left[Q^{\pi}(s,a) \frac{\nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a \mid s)}{\pi_{\boldsymbol{\theta}}(a \mid s)}\right]\end{aligned}$$

With G_t as the episodic Monte Carlo return estimate, the gradient update becomes

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha G_t \frac{\nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a_t \mid s_t)}{\pi_{\boldsymbol{\theta}}(a_t \mid s_t)}.$$

The gradient of the policy, $\nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a_t \mid s_t)$, denotes the direction in $\boldsymbol{\theta}$ for which we increase the probability of action a_t from state s_t. Then this is scaled by G_t, i.e., the better the return the more we increase the probability, as well as inversely scaled by the probability of the action, so if it is an action that have low probability of happening we give the update more weight. As a Monte Carlo method, REINFORCE has good theoretical properties, but in practice the high variance of the episode returns can be detrimental to the learning [Sutton and Barto, 2018].

An extension to this adds a baseline to the value estimate, i.e., $Q^{\pi}(s, a) - b(s)$, so instead of the return G_t we replace it by $G_t - b(s_t)$.

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha(G_t - b(s_t)) \frac{\nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a_t \mid s_t)}{\pi_{\boldsymbol{\theta}}(a_t \mid s_t)} \tag{3.13}$$

Here, $b(s_t)$ can be any function, though in practice the value function is often used to produce an *advantage* estimate, i.e., how much better the return was compared to the current expectation. This results in less variance in the learning since we reduce the size of the values in the return, i.e., the value we scale the gradient by.

While we assumed an episodic environment here, policy gradients can be shown to work in continuing environments as well, i.e., where the episode never terminates. Though the proof for continuing environments is more involved and requires the return to be zero in expectation [Sutton and Barto, 2018]. This can be done by defining value functions in relation to the average reward $\bar{r}(\pi)$, e.g., $V^{\pi}(s) = \mathbb{E}_{\pi,p}\left[r - \bar{r}(\pi) + V^{\pi}(s')\right]$, thus learning how much value we gain in relation to the average for each reward and having zero reward in expectation.

Actor-Critic

As we saw in the previous section, policy optimization has a quite natural dependency on value functions, and policy optimization and TD-learning actually work well together. Algorithms that combine the two are called *Actor-Critics*, and in using the combination they offer a trade-off between the different strengths of the two techniques. The *actor* is trained to find a policy using policy optimization, allowing for automatically learning appropriate probabilities for action distributions to balance exploration and exploitation as well as allowing for continuous actions. The *critic* is trained to provide a value estimate for the policy updates, creating a biased but lower-variance option of the episode returns from Monte Carlo methods. A standard way to do the updates is to, similarly to (3.13), use an advantage estimate, e.g., the TD-error (3.5) δ_t.

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha\delta_t \frac{\nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a_t \mid s_t)}{\pi_{\boldsymbol{\theta}}(a_t \mid s_t)} \tag{3.14}$$

One difficulty that is introduced with actor-critics is that we have two function approximations whose updates are separate, but not independent of each other. This can introduce instabilities in the updates since they both update towards a non-stationary target, created by the other network.

In the end, choosing the best approach for a problem is not always obvious, but there are some rules of thumb [Grondman et al., 2012] when selecting between actor-only, critic-only and actor-critic methods. For discrete action spaces, critic-only methods can be faster and more stable. For environments with continuous action spaces it is beneficial to have an actor, where if the problem is well estimated as a stationary MDP, we would use an actor-critic method, while if not, an actor-only method can be faster at keeping up with a changing environment.

Many popular model-free algorithms are based on the actor-critic approach, each with their own small modifications trying to make the learning process faster or more stable. We will cover a few of the more common ones, especially ones that are relevant for this thesis.

Natural Gradients

Looking at the simple gradient update presented in (3.9), we have a first order update

$$\Delta\boldsymbol{\theta} = \alpha \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}),$$

which has the problem of under- or over-shooting the optimal $\boldsymbol{\theta}$ since the step size is not adapted to the curvature of parameter space.

Trying to bound the size of the update can help, but selecting a good bound is not straightforward. Bounding the parameter updates by a euclidean norm can seem like a simple and reasonable choice at first, i.e.,

$$\Delta\boldsymbol{\theta} = \underset{\|\Delta\boldsymbol{\theta}\|<\epsilon}{\arg\max}\, J(\boldsymbol{\theta} + \Delta\boldsymbol{\theta}),$$

but the problem is that the euclidean distance in parameter space might not create a meaningful bound on the policy distribution.

The natural gradient [Amari, 1998] considers the curvature of the parameter space when calculating the update, resulting in more efficient and stable updates in many cases. In [Kakade, 2001] the natural gradient is applied to policy optimization to show how it can guarantee monotonic improvements of the policy, creating a more stable learning process.

To put a bound on the difference of two policies, we need to evaluate the difference in their distributions. The Kullback-Leibler (KL) divergence is a commonly used measure of the difference between two distributions, and is defined as

$$D_{KL}(\pi_{\boldsymbol{\theta}}, \pi_{\boldsymbol{\theta}+\Delta\boldsymbol{\theta}}) = \sum_x \pi_{\boldsymbol{\theta}}(x) \log\left(\frac{\pi_{\boldsymbol{\theta}}(x)}{\pi_{\boldsymbol{\theta}+\Delta\boldsymbol{\theta}}(x)}\right)$$

leading to the update rule

$$\Delta\boldsymbol{\theta} = \underset{D_{KL}(\pi_{\boldsymbol{\theta}}, \pi_{\boldsymbol{\theta}+\Delta\boldsymbol{\theta}})<\epsilon}{\arg\max} J(\boldsymbol{\theta} + \Delta\boldsymbol{\theta}).$$

The general approach for implementing this is to use the Fisher information matrix $F(\boldsymbol{\theta})$, which is the Hessian of the KL-divergence, to calculate the update [van Heeswijk, 2022]. This creates a second order method, which is computationally expensive, but also stable and efficient. Several RL algorithms build their update rule on this idea, though the exact implementation can differ, and often the Fisher information matrix is approximated instead of calculated exactly.

Proximal Policy Optimization

Proximal policy optimization (PPO) [Schulman et al., 2017b] came as a follow-up to trust region policy optimization (TRPO) [Schulman et al., 2017a], a method based on ideas from natural gradients, but solving a constrained optimization problem instead of calculating the exact natural gradient. TRPO has some nice theoretical properties, e.g., the constrained policy updates generate monotonic improvements to the policy, thus creating a stable learning process. However, the updates are still costly to calculate, and the method is complex to implement. PPO has similar stability properties as TRPO for the gradient updates, but is much simpler to implement and more resource efficient when it comes to calculating the updates.

PPO is an on-policy algorithm, meaning it can only use data collected by the current version of the policy. Instead of putting constraints directly on the policy updates, it is implemented by adding a term that penalizes policy divergence directly to the loss function. The ratio of the action probabilities between the updated policy and the old policy are defined as

$$k_{\boldsymbol{\theta}} = \frac{\pi_{\boldsymbol{\theta}}(a \mid s)}{\pi_{\bar{\boldsymbol{\theta}}}(a \mid s)},$$

where $\bar{\boldsymbol{\theta}}$ is the parameters used to collect the most recent training data, and are kept stationary while updating the policy. The basic objective of the policy is to increase the probability of taking good actions while decreasing the probability of bad ones. Doing this without any penalty could be implemented with

$$J_1(\boldsymbol{\theta}) = k_{\boldsymbol{\theta}}\hat{A},$$

where $\hat{A}$ is an estimate of the advantage function. Using the TD-error (3.5) as the basic advantage estimate, we create the n-step advantage estimate as

$$\hat{A}_t^n = \sum_{k=1}^{n} \gamma^{k-1}\delta_{t+k}$$

which can then, analogously to TD(λ), be combined using an exponentially weighted estimate over different horizons, creating the Generalized Advantage Estimation (GAE) [Schulman et al., 2018] which is used in PPO.

$$\hat{A}_t^{GAE} = (1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}\hat{A}_t^n$$

To also implement a penalty on the policy updates, a second objective is defined to enforce $1-\epsilon \leq k_{\boldsymbol{\theta}} \leq 1+\epsilon$, for some value of ϵ.

$$J_2(\boldsymbol{\theta}) = \max(1-\epsilon, \min(k_{\boldsymbol{\theta}}, 1+\epsilon))\hat{A}$$

The optimization objective for PPO is the minimum of these two objectives,

$$J(\boldsymbol{\theta}) = \min\big(J_1(\boldsymbol{\theta}), J_2(\boldsymbol{\theta})\big), \tag{3.15}$$

creating a gradient that will be restricted to zero if

- the policy has changed too much, i.e., $k_{\boldsymbol{\theta}} \notin [1-\epsilon, 1+\epsilon]$; and
- the change has improved the advantage estimate, i.e., $J_1(\boldsymbol{\theta}) > J_2(\boldsymbol{\theta})$.

Updating the policy using a stochastic gradient method means that we have two cases, batches of training data that was made worse by recent updates, i.e., for which $J_1(\boldsymbol{\theta}) < J_2(\boldsymbol{\theta})$, that will never be zeroed out, and batches of data that was improved by the recent updates, i.e., for which $J_1(\boldsymbol{\theta}) > J_2(\boldsymbol{\theta})$, that will be zeroed out if the policy has changed too much. This results in a larger effort towards improving the policy where it has become worse due to recent updates, in an effort to make sure that the policy always improves for all situations, but does not change too much before the updated policy has been verified in the environment.

The loss is then the combined optimization objective (3.15) with two additional terms for the value function loss and an entropy term, a measure of the randomness of the policy. The value function loss is needed if the policy and the value function share parameters, and the entropy term is added to encourage exploration. Both these terms are weighted by hyperparameters, and with a smaller entropy term the policy will be more deterministic, while a larger entropy term will encourage more exploration.

PPO is the algorithm used in Chapters 5 and 6, based on the supposed stability in the learning process as well as good implementations being available for the tools we were using.

Soft Actor-Critic

Soft Actor-Critic (SAC) [Haarnoja et al., 2018a; Haarnoja et al., 2018b] is an off-policy algorithm, allowing for a replay buffer $\mathcal{D}$ storing previous

experiences for reuse as training data. This makes for a more interaction efficient agent and also allows for learning from data collected by other algorithms. This can be beneficial in many real applications, since it might be easier and safer to collect initial data using standard algorithms, allowing the RL agent to train offline and reach some basic level of proficiency before starting to interact with the real environment.

SAC relies on something called a soft state-action value estimate, a modification of the standard state-action value function that estimates the expectation of the discounted future reward. The soft state-action value additionally takes entropy into account, and is defined as

$$Q_{\boldsymbol{\theta}}(s, a) = r + \gamma \mathbb{E}\left[V_{\bar{\boldsymbol{\theta}}}(s')\right]$$

where $V_{\bar{\boldsymbol{\theta}}}$ in turn is the soft state value function

$$V_{\bar{\boldsymbol{\theta}}}(s) = \mathbb{E}_{\pi_{\boldsymbol{\phi}}}\left[Q_{\bar{\boldsymbol{\theta}}}(s, a) - \alpha \log\left(\pi_{\boldsymbol{\phi}}(a \mid s)\right)\right].$$

Here $\boldsymbol{\theta}$ and $\boldsymbol{\phi}$ are parameterizations of the value and policy networks respectively, and $\bar{\boldsymbol{\theta}}$ is a target network to help stabilize the training. In addition to this, SAC implements the idea of double Q networks introduced in [van Hasselt, 2010], where the overestimation bias is reduced by training two Q networks and using the minimum when estimating the value function error.

The policy optimization in SAC combines the state-action value function driving the policy to take actions with high expected reward, with an entropy regularization term [Yves and Yoshua, 2006] that drives the policy to be more stochastic. The objective function is the expectation over these two, and is defined as

$$J(\boldsymbol{\phi}) = \mathbb{E}_{\pi_{\boldsymbol{\phi}}}\left[\alpha \log(\pi_{\boldsymbol{\phi}}(a \mid s)) - Q_{\boldsymbol{\theta}}(s, a)\right]$$

where α controls the balance between the entropy and reward objectives. In their second paper [Haarnoja et al., 2018b], they implement an automated tuning algorithm for α to automatically achieve a specific target entropy that can be set as a hyperparameter.

SAC is the algorithm used in Chapter 7, where, based on initial experiments it showed more stable learning, without much tuning of hyperparameters, compared to alternatives such as PPO and DDPG.

3.4 Model-Based Reinforcement Learning

There are many things that make RL difficult, and one of the main bottlenecks in the learning process is typically the *credit assignment problem* [Minsky, 1961]. It refers to the problem of assigning credit to actions

that led to a reward, when the reward is received after a long sequence of actions. The consequence is that it is very time-consuming to train large networks with many parameters using traditional model-free RL methods, and much of the literature typically employ relatively small NNs, that are faster and easier to train but limited in their capabilities [Ha and Schmidhuber, 2018b]. With a model of the environment, the agent can use the model to simulate the outcome of actions, and thereby quicker gain an understanding of the reward structure of the environment. The phrase "a model is worth a thousand datasets" from [Rackauckas et al., 2020] emphasizes that a robust model can provide immense value, and achieving equivalent generalization and sample efficiency without one can be challenging.

In model-based RL, the agent has an internal representation of the transition function $\mathcal{T}$ that defines the dynamics of the MDP. This is what we will call a *model*, and it can be learned or pre-existing, and is used to aid in creating a good policy. It can be used in different ways by the agent where one typical distinction is between using the model for *planning* or for *data generation* [Moerland et al., 2022]. Planning means the model is used by the policy to simulate possible outcomes of potentially interesting actions, allowing the policy to see what consequences to expect from different actions. When used for data generation, the agent use the model to generate synthetic data, which is then used to train a value or policy function in a model-free manner.

In Figure 3.5 we see a simple layout of a typical model-based RL agent, where the model can be trained on collected data to predict dynamics, and the policy and value functions can then be trained on some real data, as well as being supported by the model through either planning or synthetic data.

Learning a Model

In the control community, learning a dynamics model is known as *system identification* [Ljung et al., 2021], and essentially comes down to different methods of minimizing the prediction error of the model. In the ML community this problem would be part of the well-developed field of supervised learning. The benefit of supervised learning is that the target value to predict is often static, or at least from a static distribution. Compared to learning a value function with temporal difference updates, where the target value is not going to be stationary since it is also approximated by the value function, supervised methods give for more stable and directed gradient updates.

There are several important considerations when creating a model, some that are selected based on what environment that should be modelled, and some that depend more on how the model should be used [Moerland et al.,

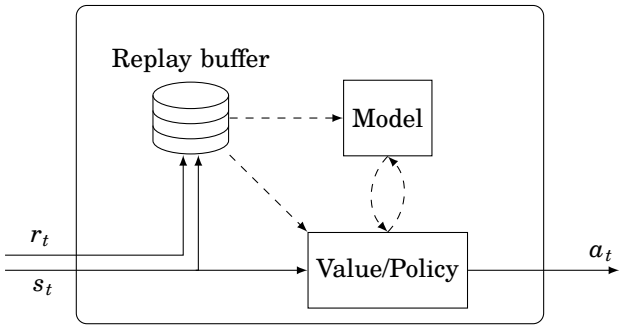

Figure 3.5 An example of a model-based RL agent. Real data is collected and used to train the model. The policy and value functions can learn from real data while also using the model for support, either for planning to generate better estimates or to generate synthetic data through simulated interactions.

2022]. In a *stochastic* environment it can be a large difference between a model that can predict the actual state distributions, compared to a model that just finds some type of *mean* related to the type of loss that is used. *Uncertainty* in the model due to limited data (*epistemic* uncertainty) is an important aspect, and is different from the uncertainty coming from the stochasticity of the environment (*aleatoric* uncertainty). A model that can capture information about uncertainty, both epistemic and aleatoric, can be very useful. Allowing for estimates of how certain the model is of a predicted state in the future when planning, as well as helping in driving exploration by identifying areas of uncertainty in the environment. *Partially observable* environments are also difficult to handle, and is different from stochasticity in that it can somewhat be mitigated by incorporating ways of accessing previous states. Examples of how to do this could be creating a *super state* that combines the states from a window of time, or using recurrent NNs (RNNs) as the model, where the networks have a type of short-term memory that can keep some data from previous interactions.

In [Ha and Schmidhuber, 2018b; Ha and Schmidhuber, 2018a] the authors present a method of learning motivated by our own cognitive system. Their simplified view is that first we compress the visual data to some efficient representation that is sent to a memory model able to predict future states based on historical ones, and based on the current state and our expectation of the future we can make an informed decision. The *world model* they present consists of the first two parts, the compression and memory, and are represented by a variational auto-encoder (VAE) and an RNN respectively. The VAE is trained on a dataset of images from the environment, while the RNN is trained on the latent representation from the

VAE, and both are trained in a supervised manner based on collected data. The efficiency of supervised learning enables the use of larger networks in the world model to fully capture the complex behavior of the environment. By offloading the extraction of important features to supervised learning, the RL part of the agent can focus on learning a small policy based on the extracted features, making for a more efficient learning process. In [Hafner et al., 2023] they use world models together with an actor-critic approach to learn state-of-the-art control for diverse tasks across multiple domains.

As mentioned, models can provide additional benefits by incorporating uncertainty in the predictions. This approach is taken by [Deisenroth and Rasmussen, 2011], where they use Gaussian processes to create a dynamics model, and [Chua et al., 2018] that implement ensembles of probabilistic NNs to capture both aleatoric and epistemic uncertainty. By embedding uncertainty about the dynamics in the model, the uncertainty can be propagated when planning to make more informed decisions, and making for sample-efficient learning. [Deisenroth and Rasmussen, 2011] show the impact uncertainty can have in driving exploration and learning by comparing the same models with and without the uncertainty, and without uncertainty performed significantly worse in terms of both learning efficiency and final performance.

Another important consideration is how we can use what we already know, or already have modelled for a related environment, when making a new policy. In [Moerland et al., 2022], transfer learning is promoted as one possible benefit of model-based RL. It refers to the ability to transfer knowledge learned in one task to another related task, and can be easier to do with a model compared to a policy or value function,

Using knowledge about the environment to impose structure on the model can also be beneficial for both the learning process and the resulting model. One example is to impose the structure of an ordinary differential equation (ODE) if we are looking for solutions to systems we expect to behave according to an ODE, and by letting the NN learn the dynamics of the system, we can use well established ODE solvers to simulate forward in time to find the solution. In [Chen et al., 2018] this idea is used to construct the neural ODE, where they learn the continuous dynamics function of an environment.

Embedding knowledge about the environment can also be beneficial for the learning process, and can be easy in some model-based approaches. The models created in [Rackauckas et al., 2020] extends scientific models with machine learning, allowing us to speed up the learning by providing known dynamics and only learning the residuals. This is used in chapter Chapter 9 to improve on the models used in Chapter 8, and doing so in a more sample efficient manner.

Using a Model

One way of using a model is to increase the sample efficiency of model-free algorithms by learning a policy or value function in part through simulated experiences from the model. This can be compared to experience replay, where the agent learns from a collection of previous experiences, where the collection of experiences can be seen as a non-parametric model. Dyna [Sutton, 1991] was one of the first model-based RL methods, implementing a model for synthetic data generation and learning the policy from both real and simulated interaction. In [Alvarez et al., 2020] they use neural ODEs to build a base model for use in RL, and show how they can use this to improve upon model-free performance by using it to generate extra data. A discussion of when a model could give a benefit compared to experience replay, and how it in some part comes down to the ability to generalize between experiences through using a model and planning, can be found in [Young et al., 2023].

For this thesis we will focus more on the case where the model is used for planning, an important part of decision-making that can provide insight into the consequences of different actions before they are done. To optimize the control action over a finite time horizon, in relation to some desired trajectory, is called trajectory optimization. It provides the ability to also put constraints on the state, which can be important especially in safety critical applications. If only the first step of the trajectory is executed, after which the optimization procedure is repeated with a receding horizon, it would lead to what is known as model predictive control (MPC), a method widely used in, e.g., robotics and control engineering among other fields. Learning-based MPC encompasses methods that extend MPC with learning in some manner, e.g., learning the model used for predictions in MPC, or use MPC for constraint satisfaction while using RL for optimizing hyperparameters for the objective [Hewing et al., 2020]. This can be considered model-based RL, and the method has been used in, e.g., [Kamthe and Deisenroth, 2018] where they create uncertainty-aware models for use in MPC, allowing for sample efficient learning that can account for constraints in a risk-averse manner.

Policy optimization algorithms come in many forms, and some can benefit from using a model for planning. Guided policy search [Levine and Koltun, 2013] uses a model for trajectory optimization, and then uses the resulting trajectory to train a policy network. Model-based value expansion (MVE) [Feinberg et al., 2018] is another method that uses a learned dynamics model for simulating the rewards over multiple steps forward, and using this to improve the value function estimates for policy gradient updates. Chapter 8 contains a type of model-based policy optimization, where gradients are calculated over the accumulated reward of simulated trajectories,

and used to update a simple policy.

When mixing with model-free methods, the ability to plan ahead can significantly increase the strength of the model-free policy, as long as the model provides good enough multistep predictions. In AlphaGo [Silver et al., 2016] the model is known, and they use both policy and value functions approximated by convolutional neural networks. And while the agent can play based on only them, it becomes many times stronger when used with Monte Carlo tree search to iteratively expand the search tree in collaboration with the policy and value functions, creating stronger value predictions for the actions that are likely to be of interest. Training this with both expert games and self-play resulted in the agent that beat the human world champion Lee Sedol in the game of Go.

Different environments might also be suited for different learners, and in [Young et al., 2023] they describe a few properties that can make environments more suited for model-based RL. For one, the environment should be easy to learn, e.g., have a simple factored structure that is easy to capture for a model. Environments where the return has a sharp dependence on the policy can make for a very uninformative signal under random exploration, while model-based RL can plan ahead and find good actions, as long as the model captures the dynamics well enough.

Using a model also introduces some challenges [Moerland et al., 2022]. For example, even if model-based RL is typically more sample efficient than model-free RL, it can be slower to train since each step can involve planning using the model and training of the model. If the model is learned, inaccuracies and uncertainties in the model are important to handle well. Model-based RL typically does not reach the same asymptotic performance as model-free methods, and the learning process can be sensitive to the quality of the model.

3.5 Other Topics in Reinforcement Learning

There are several other RL methodologies that, while not used in the thesis, would serve as interesting topics for further research. As will be discussed later, we do notice some shortcomings with many of the solutions we have used, and the ideas presented below could potentially serve as solutions to some of these issues.

Safe Reinforcement Learning

Safe RL [Garcıa and Fernandez, 2015; Gu et al., 2023] focus on ensuring safety, i.e., not taking any dangerous actions, neither during training nor execution of the agent. This is typically approached by creating optimization criteria and exploration procedures to make the agent avoid actions that

are deemed unsafe. While these bounds are typically more probabilistic in nature, and thus not as strict as the ones found in control theory, the agents also have the benefit of being able to learn from data and can thus be used in more complex environments and learn more optimal policies.

Robust Reinforcement Learning

Robust RL is a set of methods focused on creating policies that are robust to disturbances, perturbations and uncertainties in the environment [Moos et al., 2022]. The optimization criterion for robust RL is typically not to maximize the expected reward, but to penalize the worst-case reward under some uncertainty assumptions. This will create a more conservative policy, but one that can perform consistently over varying conditions and disturbances. There are many possible approaches to robust RL, one simple being to train an agent on a set of environments with slightly perturbed parameters, leaving the agent to learn a policy that is robust to these variations.

Offline Reinforcement Learning

Offline RL [Levine et al., 2020] focuses on the problem of learning from a dataset of experiences, without the ability to interact with the environment. One of the main reasons for offline RL is that exploration can be costly in a real environment, and learning only from data that is collected by a known safe algorithm, or even a human expert, can then provide an avenue for learning an agent fully offline. The key challenge is how to learn a good policy despite not being able to collect new data to correct potential errors or explore unseen states, as well as how to deal with the distribution shift between the data and the policy.

Hierarchical Reinforcement Learning

In hierarchical RL [Hutsebaut-Buysse et al., 2022] the goal is to decompose a complex task into smaller subtasks, which can then be solved individually. This makes the state and actions spaces smaller for each individual agent, and also makes it easier to transfer knowledge between tasks. On top of all these simple agent there can be a meta controller selecting some high level optimization goals which affect the behavior of the lower level agents.

Causal Reinforcement Learning

Causal ML [Kaddour et al., 2022] is the combination of causal inference and ML, giving more power to the ML algorithms by allowing them to reason about the causal structure of the data. In RL, *credit assignment problem* captures the importance of finding the causal relationship between action and reward, and while RL can be considered to have some causal features, the focus in the fields tend to be slightly different. While RL

focus on maximizing the cumulative reward, which will require an implicit understanding of the causal structure, causal inference typically focus on identifiability and inference based on the causal structure of the data. By making the causal inference more explicit in the model, we could potentially create RL algorithms that are more sample efficient and interpretable.

4

Reinforcement Learning in Practice

There are many practical challenges with RL that are not as prevalent in other ML methods. Many interesting achievements in RL come from games, e.g., Go and DotA 2, where simulations and self-play help drive massively parallelized learning. There are real world applications where RL has been applied successfully, such as stock trading, ad placement and protein folding. Physical systems such as robotics and heating systems are also an area of interest, often being trained in simulated environments before being deployed in the real world.

However, RL is notoriously difficult to train even in simulation, and there are multiple factors that make it a hard problem. One main factor is the *sample inefficiency*, requiring many interactions to properly learn a policy. This inefficiency is tied to many of the other reasons that makes RL difficult, among them the *credit assignment problem*, the *exploration-exploitation trade-off* and the *non-stationary targets* during training. In addition to this, RL using NNs suffer the same challenges as most other ML methods using NNs. The opaque representation creates a black-box structure for which the inner reasoning is hard to interpret, and it has plenty of hyperparameters to tune. Specifying a good reward function can also be a complicated task, where environments might have multiple objectives and constraints that are not obvious how they can be well incorporated.

When it comes to real world tasks, there are also a new range of problems that are not as prevalent, or present at all, in the simulated environments that are often used in RL [Dulac-Arnold et al., 2021]. For real systems there are often limited samples, it is not possible to quickly generate many interactions or running many environments in parallel to speed up the training. With this in mind, sample efficiency becomes a much more important factor. Unknown delays from sensors and actuators are likely to be more common in real systems, increasing the noise in the credit assignment

problem. Physical systems can often have safety constraints, where if the constraints are violated the system could be damaged or cause other problems. So learning while keeping safe must include some safety zone which can be hard to reason about well and embed in the reward function in a good way. Explainable policies also become more of an important question if the policy is going to be used to control anything that can actually cause damage or incur large costs if something goes wrong.

Conceptually, most basic RL algorithms are rather simple, and their state-of-the-art implementations take a number of design decisions to improve the performance of the agent. Not all of these decisions are theoretically motivated, or even well discussed in the publications, leading to an iterative process of trial-and-error as the user of these algorithms, when trying to figure out which configurations to use. Some of the more popular ideas are summarized in Section 4.1, while Section 4.2 covers some practical considerations for setting up training infrastructure and pipelines. Related work in learning for cloud infrastructure control is covered in Section 4.3.

4.1 Bag of Tricks

A *bag of tricks* is an expression denoting a set of techniques that can be used to try to reach our goal, in this case to make the RL agent learn well. As there are many sources of problems that show up in practice, making the learning process difficult and unstable, there are also many techniques that can be used to try to counter these problems. We will discuss some of the more common techniques to aid the learning process, and how they relate to the problems we have encountered in our work. While calling some of these *tricks* might be a bit unfair, as they are well known and theoretically sound techniques, it is a collection of techniques that are not directly related to the core of RL, but rather to the practicalities of applying it to real problems. For a more thorough discussion of some of these techniques and a comparison of their effect when applied to on-policy algorithms, see [Andrychowicz et al., 2020].

Inductive bias on NN structure. As previously mentioned, NNs are very versatile function approximators that scale well for high-dimensional data, though they can be difficult to train due to their black-box nature and sensitivity to hyperparameters. Some difficulties with training NNs can be countered by introducing good inductive biases into the network, or through the learning process. This can be seen clearly in the deep learning community, where large breakthroughs typically came from an insight that led to a specialized network structure that was very good at a type of problem. The most typical case is convolutional neural networks, where the insight that extracting important features using filters over adjacent values

leads to more efficient learning, and also leads to translational equivariance in the network. There are also RNNs that implement a *memory* in the network to improve learning of sequential data, and transformers that use attention to learn dependencies between different parts of the input. In Chapter 6, a simple structure is imposed on the NN based on the layout of the DC, aiming to reduce the number of parameters and make the learning problem easier. Chapter 9 introduces some novel structures for the output of the NN, and compares how the efficiency of the learning process is affected by the different structures.

Optimization. The optimization procedure can have a large impact on learning in NNs, and though the most common are first order stochastic gradient methods, there are many other interesting alternatives [Bottou et al., 2018]. Using parameter schedules such as learning rate decay can be important to achieve good convergence, and is often used in RL. While we have mainly relied on Adam [Kingma and Ba, 2017] for optimization, we have explored the effect of more complex methods. An interesting approach is to start with a first order method such as Adam, and switching to a second order method such as L-BFGS [Liu and Nocedal, 1989] as Adam starts to converge. This produces a fast and efficient initial phase of the learning, while pushing the loss down in the ed using a slower but more precise method. For the right problem, this can significantly improve the final loss without increasing the training time too much.

Target networks. An already mentioned concept that is commonly applied is *target networks*, where a separate version of the main value network is slowly following the real one. This will reduce the correlation between the estimates and the target values in the updates, by letting the target network be a separate and less noisy estimate of the target. The target parameters are updated to slowly follow the main network, either by periodically copying them or by using a soft update rule such as Polyak averaging [Polyak and Juditsky, 1992], where the target parameters take small steps towards the main network parameters at each update. These are used in Chapters 5 to 7, as part of the PPO and SAC agents.

Double Q networks. Double Q networks is a technique to reduce the overestimation bias in Q-learning by using two separate networks, one to estimate the best action and the other to estimate the target value from that action [van Hasselt et al., 2016]. By decoupling the action selection from the value estimation, the overestimation bias is reduced, and the agent can learn faster. This is a problem that becomes more prevalent in large-scale problems, where the value estimates can be more uncertain and thus more likely to overestimate the true value. Double Q networks are used in Chapter 7 for the SAC agent.

Gradient clipping. Gradient clipping is a way to constrain the size of the gradient steps in order to make the optimization parameters not change too much with each update, where too large steps could have a larger risk of accidentally destabilizing the learning process. It is commonly implemented using a norm on the gradient step, where it is clipped to a maximum size. In Chapters 5 and 6 this is used as part of the PPO agent, imposing the constraint on the probability ratio instead of directly on the gradient. And in Chapter 8 it is used to constrain how much the load balancing parameters can change in each step.

Normalization. Normalization techniques are shown in [Andrychowicz et al., 2020] to have larger impact on learning than gradient clipping. Most important was observation normalization, where the observations are normalized to provide approximately zero mean and unit variance. In addition to this they also found value function normalization quite beneficial, i.e., to allow the value function output to be zero mean and unit variance. Related to this is reward scaling, where just scaling the rewards to make the values be in a *reasonable* range for the value function can have a large impact on learning [Schaul et al., 2021]. All this relates to the idea of how NNs learn faster with well distributed data, and is something not only observed in RL. Normalization was used in Chapters 5 to 7, for both states, actions and rewards.

Reward shaping. Reward shaping [Hu et al., 2020; Grzes, 2017] on the other hand, is a process where the reward function is modified to better facilitate the learning process and can be a way to incorporate domain knowledge in the learning process. It can help to speed up learning and improve the quality of the final agent if it is embedded with the right values, though it can be tricky to reason about exactly what such changes do to the value function. Different problems will have different quirks when it comes to the reward, and one has to be careful not to accidentally change the reward in ways making the agent learn something that was not desired. We tried to use simple rewards, mapping as directly as possible to the business objective the agent should optimize. But Chapters 5 to 7 all implement some shaping decisions when it comes to how constraints should be implemented, or how multiple objectives should be combined.

Extending the state. Similar to providing a good reward function, it is important to make sure that the state observations give the agent enough information to learn the task. This is especially important in partially observable environments, where the agent only observe a subset of the state. In these cases, providing a way to access the evolution of the state by, e.g., stacking previous observations into a new state, or using RNNs to give the agent some memory of previous states, can be beneficial. Stacking

the state is used in Chapter 7, and was found to be important for the agent to learn the task.

Discretizing a space. A discrete space can come in different flavors, where the different values can have no inherent order, i.e., a categorical space, or where the values have an order, i.e., an ordinal or interval space. A categorical space could for example be the type of request, while an ordinal space could be the type of server to use, small, medium or large. An interval space would, in addition to order, have an equally spaced distance between values, e.g., the number of replicas of a microservice.

By learning a categorical policy for an action space based on ordinal or interval values, the inherent knowledge about the ordering of the actions are neglected, and the policy will have to learn this from scratch. Instead, we can learn a continuous policy, then the ordering of the actions are automatically there, and we just need to discretize the actions in some way. This is explored in [van Hasselt and Wiering, 2009], showing that continuous action that are rounded to the nearest integer can have benefits over learning a categorical policy. In Chapter 7 we use this idea for deciding the number of replicas in a microservice, also generating a continuous action that is simply rounded to the nearest valid integer in the interval action space.

Experience replay. Experience replay is used in many off-policy algorithms, keeping a history of experiences in a replay buffer. This allows for each experience to be used in many updates, and removes the correlation between successive experiences used in training, thus improving the efficiency. It can also help with stability since it decouples the current policy from the experience used in training, thus avoiding a too tight feedback loop which can cause instabilities in training [Mnih et al., 2013]. Experience replay is used in Chapter 7 for the SAC agent.

Prioritized experience replay [Schaul et al., 2016] makes the sampling happen with a probability proportional to the absolute TD-error, i.e., experiences that the current network is bad at predicting have larger chance of being included in the training, thus further improving the learning efficiency.

Policy initialization. Policy initialization, where the NN for the policy is initialized to have a "neutral" distribution, can significantly increase training speed [Andrychowicz et al., 2020]. Other ways of providing a good initial structure for the networks can also be found in the study, e.g., balancing the size between value and policy network where the value network should typically be bigger. A different approach to initializing the policy is through supervised learning on expert knowledge, where state-action pairs are used to train the policy to simply mimic the expert. Pre-training is another ap-

proach, where an agent can be trained in a simulated environment, or on offline data, after which it is transferred to the real environment where it can then already be proficient and can be further fine-tuned [Xie et al., 2022].

Hyperparameter tuning. Many of these improvements contain or introduce new parameters, which together with all hyperparameters that already exist for NNs make for a very high-dimensional space. These hyperparameters can have a large impact on the performance of the algorithm, and it is important to find good values for them. Grid search is one of the simpler methods to hyperparameter tuning, but in high dimensions can become inefficient and, e.g., Bayesian optimization [Hutter et al., 2019] or evolutionary algorithms [Kiran and Ozyildirim, 2022] will likely be more efficient. Chapters 5 to 9 all use hyperparameter tuning to some degree, and we found grid search and manual tuning to be sufficient in those cases.

4.2 Training

Training an RL algorithm is also a practical consideration that can present many challenges, both in terms of implementing the algorithmic details, but also in terms of setting up a good pipeline for running experiments.

When the agent does not train well it can be due to one of many reasons, and with the black-box nature of NNs together with RL being hard to interpret, it can be difficult to pinpoint the exact cause of a problem. It might be a problem with the implementation of the algorithm, or the implementation of the environment. The reward function might be incorrect, or just inefficient in the sense that it is not providing enough information to the agent. The state representation might be too complex, or not complex enough, and the same goes for the action space. It might simply be hyperparameters that are not tuned well, or that the agent needs to train for a little longer. Any of the tricks explained above, as well as many not mentioned, could be beneficial to make the agent train well. Creating a good pipeline to be able to run many experiments efficiently becomes very important, as well as having good tools to analyze the results.

With all these potential sources of problems, it is often a good idea to start as simple as possible, and gradually add complexity to the problem as we see that the agent is able to learn. For the work in this thesis, we often resorted to simplifying the algorithms and environments we wanted to use, to allow us to iteratively add complexity and have an easier time of finding the source of problems when they arose. In publications the focus is often on the final results, and with limited space it is not always possible to include all the details of how we got there.

One other practical problem that is often overlooked in academic work is the complexity of setting up a good training pipeline. With an algorithm and environment in mind for our RL problem, there are still choices both in terms of language and framework for the implementations, as well as what hardware to run it on.

At the time of writing, Python is by far the most common language for deep learning. And though there are other options, as in this thesis where we also use Julia [Bezanson et al., 2017] for parts of the work, there are vastly more tools and libraries available for Python. When it comes to RL algorithms they still rely on much of the same infrastructure as other deep learning algorithms, such as PyTorch [Paszke et al., 2019], TensorFlow [Abadi et al., 2016] and JAX [Bradbury et al., 2018] in Python, or Flux.jl [Innes et al., 2018] in Julia. RL specific frameworks are then built on top of these, such as stable baselines [Raffin et al., 2021] and RLlib [Liang et al., 2018] in Python, or ReinforcementLearning.jl [Tian, 2020] in Julia. Running RL workloads is typically time-consuming, and specialized hardware such as GPUs or TPUs, as well as large clusters of machines, are commonly used to help parallelize and speed up training. RLlib is part of the larger ecosystem Ray [Moritz et al., 2018], which is a collection of Python libraries providing different tools for handling distributed computations such as training and tuning of ML workloads.

For this thesis we began by looking at RLlib and Ray, which seemed like modern tools with good support for distributing work and a large collection of standard algorithms. While this was true, there were also some things that were not as good as we had hoped. The documentation was not always up to par, and the code was not always easy to follow. So when doing the standard things, and following their recipe for how to do it, everything was nice. But straying slightly from the standard use case, it could be quite difficult to figure out how to fit a different workflow into the framework. This was one of the reasons we decided to switch, moving to a much more basic framework, and trading some convenience and efficiency for the flexibility and control of doing things ourselves.

Deploying the clusters of machines that run the workloads can involve anything from configuring the hardware yourself, to provisioning virtual machines from a cloud platform. On top of these machines, there are often additional tools needed to orchestrate the infrastructure, where Kubernetes [*Kubernetes* 2023] is a popular choice with many tools that work with it. For example, Ray provide their own Kubernetes operator, which simplifies setting up a cluster of machines for running Ray workloads.

The effort required to set up an efficient pipeline for experimentation can vary a lot, depending on what tools are used in what part of the pipeline, and how well they work together. In this thesis we spent quite some time on setting up custom pipelines to learn about the different parts of the process,

and to be able to experiment with different setups.

Chapter 5 and Chapter 6 both use a similar software stack with RLlib and Ray on the top to handle parallel tuning of hyperparameters in the algorithms, though the hardware requirements were slightly different. In Chapter 5 we used a simple environment and small networks, and ran training on CPUs, while in Chapter 6 we used a more complex environment, requiring a GPU to execute. The networks were still relatively small, and kept on the CPU to maximize GPU time for the environment. A tool like Ray is very useful for this kind of work, as it handles the parallelization of training and tuning, while providing easy controls for resource scheduling and sharing. It also provides a convenient interface for logging and monitoring the basic metrics of the training, such as the reward and loss, as well as the hyperparameters used. The downside of using a tool like Ray, a full framework for distributed computing and RL, is that it can be hard to understand what is going on under the hood, and to customize parts of the training process.

In Chapter 7 we set up a cluster of machines on an OpenStack [*OpenStack* 2023] cloud, using Terraform [HashiCorp, 2023] and Ansible [Red Hat, 2023] to provision the machines and install the necessary software. The software implementation was done in Julia, using RL algorithms from ReinforcementLearning.jl, and parameter tuning using Hyperopt.jl [Bagge Carlson, 2018]. Training is then run from a single machine that acts as the control node, syncing the required code and data to the worker nodes, distributing the work over them, and collecting the results. A small package was built to support easier syncing of development code that frequently changed during experimentation, and additional functionality was contributed to both ReinforcementLearning.jl and Hyperopt.jl to better support our use case.

For Chapters 8 and 9 the implementation of the algorithm took most of the time rather than tuning parameters and training, so there was no need for a complex pipeline, and we manually parallelized the few experiments we had over local servers, running on CPU.

4.3 Related Work

While many of the widely applied methods in industry are based on simple heuristics and local information, there are also benefits to more complex methods that take a larger context into account. As manually crafting control strategies that take a larger context into account can quickly become infeasible as the complexity of the system grows, this is where learning-based methods can be useful. Cloud systems also provide easy data-collection, lending themselves well to data-driven methods that can automatically ex-

tract complex dynamics from the system. High level performance measures, often related to business goals such as minimizing energy consumption while adhering to different SLOs, are also often difficult to translate into low level control strategies, and learning-based methods can be used to optimize over these goals to find novel control strategies.

This thesis is mainly focused on how RL can fit this role, and what challenges are encountered when applying RL to the domain of cloud control. We will look at different control objectives related to what is explored in the thesis, such as DC cooling control, automatic resource scaling and load balancing. The rest of this section gives an overview of the current state-of-the-art in these areas, focusing on how RL fits into the picture.

Cooling Control

While there are many important tasks that can be achieved using the simple approaches mentioned in Section 2.2, there are also many benefits to methods that can incorporate more information and take a larger context into account.

A common method from the field of automatic control is LQR (linear quadratic regulator), a linear state-feedback controller minimizing a quadratic cost function. In [Garcia-Gabin et al., 2018] they create a linearized model of a DC and a cost function incorporating both total energy and maximum server temperatures to create a CRAH controller. While the traditional approach to find the optimal LQR controller rely on having a linear (or locally linearized) model of the system, [Yaghmaie et al., 2023] evaluate both model-free and model-building approaches for learning an LQR controller without having an initial model. Evaluating this on an idealized DC cooling scenario they conclude that both approaches learn stable policies quicker than standard RL methods, and find a feedback controller closer to the optimum. The drawback of LQR is that is assumes the system dynamics are accurately modelled by linear differential equations, and when that does not hold it might not perform well. By allowing non-linear policies using, e.g., NNs, we loose some interpretability and stability, but can create policies that are more flexible to a wide range of systems.

Many algorithms are developed and tested in simulation, since real systems can be difficult and expensive to test on. EnergyPlus [Crawley et al., 2001] is one simulation tool that can be used to simulate HVAC (Heating, Ventilation, and Air-Conditioning) systems in buildings, but has also been used to simulate DCs for training RL agents, minimizing cooling energy by finding optimal facilities setpoints [Li et al., 2020]. In [Lundin, 2021] they instead train an NN to model the dynamics of a datacenter, and use that as a model to train and validate RL agents against standard control strategies. In this thesis, the hardware experiments in Chapters 5

and 6 are done in simulation due to the risk, cost and complexity of running experiments on real hardware. The cloud experiment in Chapter 7 was also simulated for simplicity, but in Chapters 8 and 9 the experiments were done on real log-data from Kubernetes clusters running a simple service.

[Zhang et al., 2021] provides an overview of current research in DC cooling control, dividing it up between different ways of modelling power consumption of, e.g., CRAH units and chillers, as well as MPC and RL as the modern control strategies capable of dealing with these complex systems. The conclusion is, as was already discussed in Chapter 3, there is no optimal *one size fits all* solution, and it depends on the specifics of the problem at hand. Using ML to control a DC can often incur risks, since these are large and expensive systems where a malfunction can be costly. An important consideration is therefore how to create learning-based strategies that avoid problematic actions, which is the topic of *Safe RL* mentioned in Section 3.5.

One approach to ensure safety is to have a supervisor that approves any actions before they are enacted, assuming the supervisor is trusted to make the right decision. In [Gao, 2014] they take this approach, using a human operator to approve and enact the actions selected by the agent. By training an NN to predict future PUE in a supervised manner, this becomes similar to learning a Q-function, but with easier training since the target values are provided in the data. It is implemented for a Google DC, and the predictions are based on different features such as IT load, temperatures, humidity, as well as operational parameters. The trained NN can then be evaluated over different operational parameters to find which combination would provide the best PUE, and the operator implements the best one that is deemed safe. In [Gasparik et al., 2018] they describe how this was automated by both creating a better model with uncertainty, and by creating an automated supervisor that used the model and a number of safety constraints to decide if the action was safe or not.

Providing more energy efficient climate control using learning-based methods is not specific to DCs, and, e.g., [Li and Xia, 2015] successfully apply RL to a building HVAC system to balance energy consumption with thermal comfort. By using table based Q-learning with different levels of discretization, they balance the trade-off between performance and computational complexity, and improve on their control objective compared to constant temperature control.

In [Van Le et al., 2019] the authors focus on how to optimally utilize the free-cooling capabilities of an air-cooled DC. Using both MPC and RL, they control the air supplied to the servers by setting flows, temperatures and how outside air is mixed with recirculated air. A model of the gas-vapour mixtures is extended with an NN that is trained to estimate the power consumption based on system states involving mostly properties of the air,

which in part comes from the gas-vapour model, while also including the total IT-load in the state. Based on this model they compare the performance of an MPC approach and an RL agent controlling the temperature and humidity of the air supplied to the servers. Though both seem to improve on the performance of a simple hysteresis controller, the MPC approach has much higher computational cost for evaluating which action to take each step. In [Van Le et al., 2020] they further investigate RL for this problem, using different types of DQN to train RL agents on discretized subsets of the action space, outperforming simple baselines.

Active ventilation tiles are floor tiles that can be opened and closed to control the airflow in a DC with raised floor design. In [Hua et al., 2021] they implement both central algorithms, and multi-agent algorithms to control these tiles to minimize the cooling energy. For the multi-agent algorithm, they found that sharing a reward, e.g., the average reward of all agents, improved performance compared to independent rewards.

[Ran et al., 2019] combine the control of IT and cooling systems, using a variant of DQN to jointly optimize job scheduling and air flow in the DC. Similarly, [Chi et al., 2020] explore multi-agent RL is explored as a venue for joint optimization of workload scheduling and cooling, and propose an asynchronous optimization strategy to improve stability with different interaction frequencies of the agents. While we also do joint IT and cooling optimization using RL in Chapter 5, we consider a more complex cooling system with both free-cooling and chillers, making for a more complex problem. In [Wang et al., 2022] they model a more complex cooling chain, and create a reward function that drives the agent to minimize the energy consumption while keeping a constant temperature inside the IT-space. Compared to this, Chapter 5 allows for the temperature in the IT-space to vary, and include the energy from the server fans in the total energy, allowing the RL agent to find a better trade-off between energy consumption and temperature. Another difference is that the DC we model did not have containment, so there is recirculation to consider which add additional complications. Chapter 6 considers a similar problem as Chapter 5, just using a more realistic simulation model.

In [Lazic et al., 2018] they learn a coarse linear model of a DC cooling system from data, with the goal of controlling the air and water flow for the CRAHs to maintain a good temperature and differential air pressure, while minimizing energy consumption. The model is only based on different properties of the air, and they consider the IT-load as a disturbance. Using this simple model for MPC, the authors show how this strategy can improve on the performance of standard PID controllers when implemented on a large scale DC.

[Wan et al., 2023] develop an interesting model-based RL approach where they create a data-driven model, use MPC with this model to find

trajectories that are safe and energy efficient, and then use behavior cloning to train a policy that mimics the MPC controller. The model-based approach allows them to learn quicker, and also develop a risk model in conjunction to the transition model, allowing them to avoid actions that are deemed unsafe.

Automatic Resource Scaling

The simple algorithms presented in Section 2.2, based on local information and thresholding, work well enough in many cases. Though, in a large dynamic cloud environment, with complex objectives combining both resource costs and adhering to SLOs, it will generally be possible to improve on these simple algorithms.

To overcome the restriction of a local controller, work such as [Hasan et al., 2012; Borgetto et al., 2012] design their algorithms to take a larger context in consideration. In the first paper they do so by considering how compute, storage and network metrics correlate, to provide better scaling decisions over all those resources, while the second paper considers VM placement and migration in conjunction with turning physical servers on or off to save energy.

Another idea would be to aim for a proactive approach, where instead of just reacting to what resources would have been needed now, we predict what will be needed in the future. If these predictions can be made well enough it should allow for much more aggressive scaling, since we can start scaling before the resources are needed, and avoid under-provisioning. LSTMs (long short-term memory networks) is a popular type of NN used for time-series modelling. They work by having a memory state that is propagated to future predictions, so the NN can choose to keep information from previous states. This is used in [Kumar et al., 2018; Shahin, 2016] to model the CPU utilization, so they could predict future utilization based on historical values. In [Toka et al., 2020] they combine multiple predictive methods based on different techniques, including autoregressive methods and LSTMs, to predict the future workload. They show how running multiple methods that work in very different ways, and implementing scaling decisions based on the prediction from the one that performed best in the recent past, can improve the predictions.

In [Różańska and Horn, 2022] they use data-driven predictions to scale cloud resources proactively for compute intensive tasks with hard deadlines. They divide metrics into two categories, those that are influenced by the way the application is configured, and those without any strong relation to the configuration. Predictions of the future expected values are estimated separately for the two categories, with methods specific to the type of metric.

While predicting future workload can provide valuable information, it

is often not possible to accurately capture all behavior of the system, such as stochastic behavior of the users. And even though predictions often capture some average behavior, some over-provisioning will still be required to account for this variability.

Methods based on RL could ideally implement control that both take a larger context into account, and optimizes proactively to consider possible and expected future states. While RL does not necessarily model the dynamics to predict the future like an LSTM, the agent will learn to optimize over future reward according to the underlying MDP, and thus implicitly take future states and disturbances into account if possible.

One early method using Q-learning for automatic configuration of web services was proposed in [Bu et al., 2009], deciding parameters such as the number of servers and threads, and max number of concurrent clients. In [Xu et al., 2012] they use deep Q-networks to do vertical scaling to optimize a web system. Similar to the Dyna architecture mentioned in Section 3.4 they extend it with a model to improve data efficiency.

Horizontal scaling seems more common in the literature with, e.g., [Barrett et al., 2013; Ghobaei-Arani et al., 2018] using Q-learning scale a service horizontally based on information local to that service. Both craft a reward function that takes both resource costs and SLA violations into account.

While table-based Q methods can prove simpler in some ways, they are hard to scale, and they only handle discrete states. To alleviate this, [Bitsakos et al., 2018] explore deep Q-learning for horizontal scaling of a single service in a simple simulated scenario. Similarly, [Bibal Benifa and Dejey, 2019] use NNs with SARSA to scale slightly more complex services, though still only considering a single service but for heterogeneous workloads.

[Rossi et al., 2019] consider both vertical and horizontal scaling with the same agent, and bring forth the benefits of model-based approaches by comparing Q-learning, Dyna-Q and Q-based value iteration, showing that the model-based approaches can be more data efficient.

Microservice architectures today can consist of many services used to create a single application, where services depend on each other to provide the full functionality. This gives better opportunities for coordinated scaling, and specifically for scaling proactively in a more well-informed manner, since loads that propagate through the system create a temporal dependency between the services. This is considered in, e.g., [Millnert et al., 2018] where automatic scaling and admission control of time-sensitive service chains is done using techniques based in control theory and network calculus. The focus of the work was to determine the lowest upper bound on computing resources, to support workload variations without missing deadlines. [Yu et al., 2022] consider an unknown cloud application using multiple different services. By correlating request flows between services they extract the

service graph, and using Bayesian optimization they find optimal scaling decisions, minimizing cost and latency at the same time.

In Chapter 7 we similarly consider an unknown system of multiple services, and workloads that traverse over some chain of services. This makes for a hard RL problem since there is a sequence of decisions needed to be made for a job to succeed, and the reward is not known until the job has finished, creating delayed effects from the actions. We train an RL agent to implicitly learn the dependencies between the services, and use this to make proactive scaling decisions. While the user-facing service might still require a buffer since the random workload dictates the arrivals there, it should give a more stable approximation for the remaining services, allowing for smaller buffers without risking missing deadlines.

Load Balancing and Scheduling

The algorithms presented in Section 2.2 work well enough in many cases, but there are also benefits to more complex methods that can incorporate information that is typically not easy to include in the simple algorithms.

A common consideration for these problems is to maximize throughput or minimize response time by balancing or scheduling the tasks over a set of available resources. [Mao et al., 2016; Dong et al., 2020] apply RL to minimize response time using task scheduling, and use simulated scenarios for training and evaluation. To combine multiple competing objectives, [Wang et al., 2019] apply a multi-agent DQN approach. In [Shahab Samani and Stadler, 2022] the authors train an RL agent to do admission control and routing, showing that initial training on a simulated environment can be used to bootstrap the agent before deploying it in the real system.

By including energy consumption in the objective function we create a trade-off between energy and performance, creating a more complex problem. Here energy could just refer to some running cost, where heterogeneous hardware can have different properties. Similar to doing cooling control with awareness of the IT-systems, there are work on load balancing and scheduling with awareness of the non-IT systems such as cooling and power. An early attempt at this was [Liu et al., 2012] that do workload scheduling with awareness of the cooling and power systems. Using traditional modelling and optimization schemes, they schedule workloads when cooling is more efficient and electricity is cheaper, to reduce the energy cost of the datacenter. In [Townend et al., 2019] the authors present a scheduler that takes both physical and software layers into account when distributing containers on a Kubernetes cluster. They argue that improved models and metrics are needed to better understand the relationship between the two layers, in order to make energy efficient scheduling decisions. Another interesting approach is to look at the optimal load regime for servers, and try to keep as

many servers as possible in that regime by balancing the load appropriately, allowing servers with low load to enter a sleep state [Paya and Marinescu, 2017]. In [Xu et al., 2017] they authors apply geographical load balancing that takes electricity and delay into account. They assume that short-term estimates of workload and renewable energy production are available, and use this to make decisions on where to place the workload using standard optimization techniques. In [Cheng et al., 2018] the authors use DQN to do energy aware resource provisioning and workload scheduling in a large scale DC setting.

Taking heterogeneous hardware into account makes the problem more complex, and in [Baek et al., 2019] they use RL to automatically learn the best load balancing strategy for optimizing response time and energy usage in an edge cloud consisting of heterogeneous devices. [Kanbar and Faraj, 2022] combine scheduling and load balancing over multiple clouds, using NNs for classifying the workload, multi-criteria optimization for scheduling, and RL for load balancing. In Chapter 8 we approach a similar problem where we try to optimize the load balancing over servers located on the cloud and the edge, considering both cost of running the workload in different settings and the response time. We use a model-based method that regularly reevaluates the recent data, and try to come up with the optimal strategy for the current situation, rather than trying to learn a general strategy that works for all situations. In Chapter 9 we improve on the model to better capture some system dynamics, allowing for better predictions outside the training data, leading to faster convergence of the policy.

5

Holistic DC Control using Deep RL

The distribution of IT workload within a datacenter will affect where heat is generated from the physical equipment. As generated heat will in turn affect the cooling systems, it could be beneficial to have this information when controlling the cooling systems. Similarly, it could be beneficial to know the existing temperature distribution when distributing the IT workload. If more load is placed in areas that, by the layout of the DC, receives more cold air, the cooling systems could operate more efficiently.

While these systems clearly affect each other, the exact relationship is not as obvious, and controlling them in a coordinated manner is not trivial. We explore this holistic control problem using RL, deploying an RL agent to control both systems simultaneously. A simulated environment is also introduced, recreating the problematic aspects of a DC that we want to make sure our controller can handle. In Chapter 6 we develop a more complex environment to further explore the problem of holistic control using RL.

5.1 Thermal Model of a Datacenter

To train and evaluate the proposed RL agent, a thermal model of a DC is developed to capture the dynamics of the cooling system and the thermal interactions between the IT equipment and the environment. [VanGilder et al., 2018] provides a compact model of a DC cooling system, idealizing the system as a heat exchanger in series with a thermal mass. This compact model is developed further in [Healey et al., 2018], where components for the room, plenum, walls, floor, and ceiling are added to better represent the complete thermal mass of a DC.

The model created here, and later extended in Chapter 6, is based on a small DC pod seen in Figure 5.1, operated by RISE SICS North in Luleå,

Figure 5.1 Small DC pod at RISE SICS North in Luleå, Sweden.

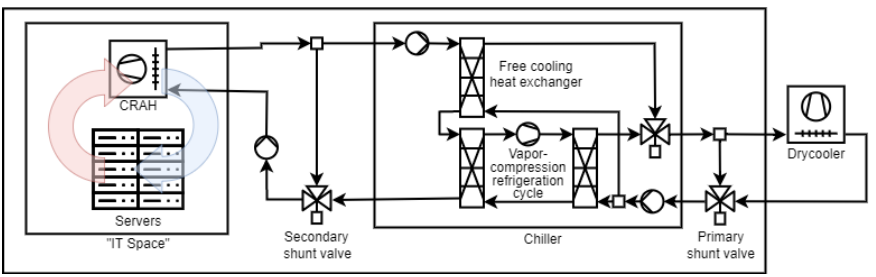

Figure 5.2 Conceptual heat rejection schematic of the physical model. "IT space" refers to the server hall in Figure 2.2, containing servers and CRAH units.

Sweden. It consists of 12 racks and 4 CRAHs, and is well instrumented to be usable for research purposes. The DC is cooled using a chiller with free-cooling capability, and the cooling system is designed to be able to operate in free-cooling mode for most of the year. Figure 5.2 shows a diagram of the infrastructure of interest, where CRAH units are transferring the heat generated in the server hall, into a water loop which in turn is connected to a chiller with free-cooling capability. The free-cooling functionality can be applied when the outdoor temperature is low enough to enable the dry-cooler to reject the heat from the IT equipment without using the compression-cycle. In compressor mode, a vapor-compressor cycle lowers the temperature obtained from the dry-cooler to chill the water loop connected to the CRAH so that the generated heat can be rejected. Operating the compressor in the chiller consumes a lot more power than operating the chiller in free cooling mode while also increasing the risk of a failure, and should be avoided if possible.

The simplified model presented in this section ignores the thermal mass and dynamics of equipment, facilities and cooling system, and only considers

Table 5.1 Variables used in the model.

Name	Description
p_s	IT-load on server s, $p_S^{idle/max} = [50, 400]$ W
p^{fan}	Total power used by fans
p^{comp}	Power used by compressor
Q_s	Air flow through server s, $Q_S^{min/max} = [0.001, 0.04]$ m^3/s
Q_c	Air flow through CRAH c, $Q_C^{min/max} = [0.1, 2.1]$ m^3/s
K_S^{fan}	Server fan power coefficient $K_S^{fan} = 7.9 \cdot 10^5$ W/(m^3/s)^3
K_C^{fan}	CRAH fan power coefficient $K_C^{fan} = 143$ W/(m^3/s)^3
$T_s^{in/out}$	Air temperature into/out from server s in degrees Celsius
$T_c^{in/out}$	Air temperature into/out from CRAH c, the outlet is a control variable that is restricted to $T_C^{min/max} = [18, 27]$°C
T^{amb}	Ambient (outdoor) temperature in degrees Celsius
T_s^{cpu}	CPU temperature in server s in degrees Celsius
T^{cpu}	CPU target temperature $T^{cpu} = 60$°C
C_v	Estimated volumetric heat capacity from [Sjölund et al., 2018] $C_v = 1183$ J/(Km^3)
R	CPU heat loss resistance $R = 0.33$
K_i	Integral action control coefficient $K_i = -8 \cdot 10^{-5}$ m^3/(s·K)

the heat and mass transfer associated with the cooling airflows.

Simple Model

Based on the conceptual heat rejection schematic in Figure 5.2, we create a model of the DC, aimed at capturing a few interesting dynamic properties of the cooling system. Each server generates heat based on the current load, which is picked up by the air and transported to the CRAH. The CRAH is connected to a chiller that can reject heat from the CRAH using either a compressor or free-cooling.

Recirculation and bypass flows are inefficiencies caused by hot air flowing back into the cold aisle or cold air bypassing the servers. While recirculation and bypass flows are avoided in most modern DCs using containment solutions, e.g., where cold and hot air are separated by lightweight walls, we choose to include these flows. One reason is that we specifically want to look at the problem of control in the context of complex air flows, which are more prevalent in DCs without containment. Though the main reason is that the DC the model is based on did not have any containment, and keeping the model close would simplify a potential future transfer to the real system.

The variables and constants used in the model are listed in Table 5.1, and will be further explained in the following sections. Generally, s and c will be used to index individual servers and CRAH units, while S and C will be used to denote common values over all servers and CRAH units.

Servers. The model contains 40 servers assumed to be identical. Heat generated by each server is modelled as proportional to the load $p_s(t)$ on the server, and the CPU temperature T_s^{cpu} depends on the generated heat as well as the inlet temperature T_s^{in} and air flow Q_s through the server.

$$T_s^{cpu}(t+1) = T_s^{in}(t) + \frac{1}{R}\frac{p_s(t)}{C_v Q_s(t)} \tag{5.1}$$

Here R captures the CPUs resistance to give off heat to the air, while C_v is the volumetric heat capacity of the air.

The servers contain an internal fan that we do not control, and we model it as an integral action controller that tries to keep the CPU temperature at a target temperature T^{cpu}. The target control action $\hat{Q}_s$ is calculated as

$$\hat{Q}_s(t+1) = Q_s(t) + K_i(T^{cpu} - T_s^{cpu}(t)), \tag{5.2}$$

with the actual flow Q_s being the value of $\hat{Q}_s$ constrained to the interval $[Q_S^{min}, Q_S^{max}]$. This creates a dynamically changing flow depending on both inlets and load that is not directly controllable, but can be influenced by changing the inlet temperature and the load on the servers.

Each outlet temperature from the servers are then modelled as capturing all the heat generated by the server under the previous step,

$$T_s^{out}(t+1) = T_s^{in}(t) + \frac{p_c(t)}{C_v Q_s(t)},$$

which is assumed to mix over all outlets, and create a single homogenous flow with temperature

$$T_S^{out}(t) = \frac{\sum_s Q_s(t) T_s^{out}(t)}{\sum_s Q_s(t)}.$$

CRAH, chiller and dry-cooler. This model only uses a single CRAH with two control variables, the flow $Q_c(t)$ and the outlet temperature $T_c^{out}(t)$. With this we have two ways of providing more cooling capacity, either by increasing the flow or by lowering the outlet temperature, affecting the surrounding systems in different ways, and consuming different amounts of power. The outlet temperature of the CRAH is assumed to be controlled directly, though in practice it is controlled by adjusting the temperature and flow of the water in the CRAH. This makes the model easier to implement,

not explicitly needing the water loop, and should not make much difference for the higher level objectives we have of holistic control.

The compressor is modelled to turn on depending on the relative temperature change needed in the CRAH. When turned on, it uses power proportional to the heat removed from the air in the CRAH.

$$p^{comp}(t) = \begin{cases} 0 \text{ if } T^{amb}(t) < T_c^{out}(t) \\ C_v Q_c(t) \left(T^{amb}(t) - T_c^{out}(t)\right) \text{ otherwise} \end{cases}$$

This introduces a non-linearity such that power consumption increases by a large factor for CRAH temperature setpoints below the ambient temperature.

The dry-cooler is modelled to remove all heat down to the ambient temperature. This is a simplification that gives us the main effect of the dry-cooler that we were after, where we have a form of free cooling which works for a limited range of temperatures.

Fan power. Fan power from the servers are typically included in the IT power when calculating the PUE. As we can select where in our model to include it, we choose to put it with the cooling power instead, as it is used to move air around in the DC. This is not important for the results, and is actually going to give a worse PUE, but it makes more sense to us to include it in the cooling power.

The total cooling power is then composed of both the fan power from servers and CRAH units and the compressor power. The fan power follows affinity laws derived from dimensional analysis [Buckingham, 1914],

$$p^{fan}(t) = K_C^{fan} Q_c(t)^3 + K_S^{fan} \sum_s Q_s(t)^3, \tag{5.3}$$

where the constants K_C^{fan} and K_S^{fan} are calculated as maximum power over the cube of the maximum flow for both server and CRAH fans. These numbers are specified by the manufacturer.

While model simplifies many parts of the system, it captures interesting dynamics in how the cooling system interacts with the servers and how the power consumption is affected by this.

Server hall. Simulating the air flow in the server hall is done using a model where the flow between CRAH and servers can either recirculate into the servers or bypass the servers depending on the flow difference between the CRAH and the servers. Recirculation means that a fraction η of the hot air from the servers flows back into the servers, and bypass means that a fraction μ of the cold air from the CRAHs flows directly back into the

CRAH.

$$\eta(t) = \max\left(0, 1 - \frac{Q_c(t)}{\sum_i Q_s(t)}\right)$$
$$\mu(t) = \max\left(0, 1 - \frac{\sum_i Q_s(t)}{Q_c(t)}\right) \tag{5.4}$$

By assuming a constant air pressure and good mixing, we approximate the temperatures of the server and CRAH inlets as

$$T_s^{in}(t+1) = (1-\eta(t))T_c^{out}(t) + \eta(t)T_S^{out}(t)$$
$$T_c^{in}(t+1) = (1-\mu(t))T_s^{out}(t) + \mu(t)T_c^{out}(t)$$

which is simply the weighted average of the temperatures of the air from the CRAH and the servers, where the weights are corresponding volumetric flows.

The assumption of constant air pressure does not hold precisely for the CRAH and servers, where the flow is assumed to be the same in and out while the temperature changes. This will result in slightly unrealistic dynamics where small amounts of mass are being created and removed in the servers and CRAH. We assume these effects are negligible compared to other dynamics, and that in regard to the RL agents ability to learn, the reduced complexity is worth the trade-off of having a slightly less realistic model.

5.2 Combined IT and Cooling Control

To control the presented model in a holistic manner, we need an algorithm that can handle a mix of discrete and continuous control variables, and that can learn from complex state variables without any initial understanding of the underlying dynamics. For this we develop an RL agent, controlling CRAH setpoints and load balancing decisions simultaneously based on the current state of the DC.

Reinforcement Learning Agent

We decide to use PPO [Schulman et al., 2017b], a state-of-the-art RL algorithm that performs well on many complex control problems, and has implementations that handle a mix of discrete and continuous control variables. While a few other algorithms were considered, e.g., DDPG and SAC, we found PPO to be well suited for this problem.

We use dense neural networks for both the policy function and value function, with no layers are shared between them. Each network takes

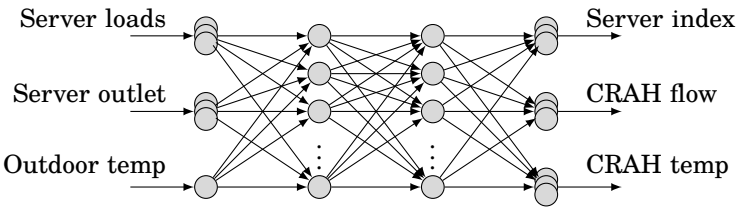

Figure 5.3 The policy network used for the RL agent. The value network has the same structure for the input and hidden layers, but a single output.

the state as input and feeds it through two hidden layers with 256 units each, and uses the tanh activation function. The value network has a single output with no activation, while the policy has three sets of outputs, see Figure 5.3.

State space. The state $\boldsymbol{s}$ is a vector consisting of the current server outlet temperatures and load, as well as load p^{job} and duration D^{job} of the incoming job.

$$\boldsymbol{s} = \left(\left[T_s^{out} \dots \right], \left[p_s \dots \right], p^{job}, D^{job} \right)$$

This is actually only a partial observation of the state, as, e.g., the current flows and temperatures inside the datacenter are unknown. As are the durations of the loads already distributed in the datacenter. To improve learning, the states are mapped using affine transformations between their estimated ranges and the range $[-1, 1]$, before being fed to the agent. The server outlet temperature is estimated to stay within the range $[15, 85]$°C, and the load is given in Table 5.1.

Action space. The action $\boldsymbol{a}$ consists of setting the CRAH temperature and flow setpoints, as well as the target server s for a potential job.

$$\boldsymbol{a} = \left(\left[T_c^{out} \dots \right], \left[Q_c \dots \right], s \right).$$

Each action is sampled from a set of distributions generated by the output of the policy network. The policy network has 4 outputs for the CRAH setpoints, and an additional one output per server for the load balancing. The 4 outputs are used as mean and logarithm of standard deviation for two Gaussian distributions, to sample the temperature and flow setpoints of the CRAH. The temperature and flow setpoints are sampled from the distribution, and constrained to the range $[-1, 1]$ before an affine transform is applied to map them to their respective ranges given in Table 5.1. The load balancing distribution uses the remaining outputs to generate a categorical distribution through the *softmax* function, which is then sampled to select a server.

Reward function. The reward r is designed to capture the energy cost of the cooling process, while penalizing the agent for the number of dropped jobs J_{drop} and for not adhering to the cold aisle server inlet threshold of 27°C, a common standard from industry [ASHRAE TC9.9, 2016]. The cold aisle temperature threshold is needed since our model does not capture the problems that appear as the servers run too warm, such as increased failure rates and reduced lifetime.

$$r = -\left(C_1\left(p^{fan} + p^{comp}\right)\Delta t + C_2 J_{drop} + C_3 \sum_s \mathbb{1}(T_s^{in} > 27) \right) \tag{5.5}$$

The reward is the negation of these three costs, weighted by $C_1 = 0.00001$, $C_2 = 10$ and $C_3 = 0.1$. The weights are tuned both for scaling the reward to a *reasonable* range to improve learning, and to punish overstepping the cold aisle temperature threshold or dropping jobs harder than the using energy, incentivizing the agent to prioritize those constraints.

5.3 Evaluating the RL Agent on the Simulated Model

Using the simulated model, we can evaluate the performance of the RL agent in a controlled environment. The code for both the model simulation and RL agent training is available on GitHub[1].

Training and Simulation Setup

We use Ray RLlib [Liang et al., 2018] to handle the training of the RL agent. The environment is implemented in Python using the OpenAI gym interface [Brockman et al., 2016], allowing RLlib to use it as a training environment. The PPO implementation in RLlib is adapted to fit with our environment, allowing for parallel training and hyperparameter search through Ray Tune [Liaw et al., 2018].

Each step in the environment, i.e., the time between an action and the subsequent observation, is 1 second. The data is logged through Ray to Tensorboard [Abadi et al., 2016] using callbacks that track the minimum, mean and maximum over a *soft horizon*. The soft horizon is used since the environment is continuing, meaning that there is no natural end to an episode. We run with a batch size of 200, and horizon length of 100, meaning that we collect 200 steps of data between each time we train and collect data over each 100 steps. Each batch is used to train the agent over 30 epochs using mini-batches of size 128. We effectively disabled the value function clipping parameter by setting it to a large number. Remaining training-parameters are left at their default values, e.g., using the optimization

[1] https://github.com/albheim/rldc-flowsim

algorithm Adam with learning rate 0.0004, a discount factor of 0.99 and a GAE parameter of 1.0. The entropy coefficient is left at zero to converge towards a more deterministic policy.

For the presented results we use the collected mean values, which we then further downsample by a factor 50 to make for a more readable plot.

Workload. The simulated IT workload is modelled to represent a simple 1-tier compute service, which could for example be an on-demand image recognition or inference task service. The service time and load of incoming jobs are assumed to be known, a simplification motivated by previous work showing how these types of values can be estimated [Ahmad et al., 2011]. Each job adds a load of 20 W and has a duration of 500 s with an arrival rate of one job per second, leading to an average load of 250 kW per server. Job queues or processor sharing are not modelled, and a job that is scheduled on a fully loaded server is simply dropped. This captures the essence of a more elaborate system where placing loads on full servers should be avoided.

Ambient temperature. We keep the outside temperature set to a constant value of $T^{amb}(t) = 20°\text{C}$, creating less learning obstacles for the RL agent. This also allows the baseline agent to be efficient over the whole sequence, setting a good performance measure for the RL agent to improve upon.

Baseline controller. We implement a baseline comparison using a load balancer placing incoming jobs on the server with the lowest load, and a cooling controller with a fixed setpoint for the CRAH temperature and flow of 22°C and 80% max flow respectively. These are both standard controllers found in industry, and provide a baseline that is easy to implement and interpret. The cooling temperature was selected to make the baseline agent not use the compressor, providing a good baseline for the RL agent to improve upon. For this simple setting the selected load balancer will actually be optimal when looking purely at the energy consumption of the server fans, as the inlet temperature is homogenous and an uneven in load will simply increase the required flow for one server while it is reduced for another, making for a higher power consumption since the power is proportional to the flow cubed (5.3).

We could compare to a more advanced controller, e.g., other RL algorithms, but it becomes difficult to compare the results as the baseline controller itself will be more complex with more parameters to tune, and also harder to know if we compare them fairly with both equally tuned and trained. Therefore, we choose to instead ensure that this simple controller is sufficiently competitive to make for an interesting comparison.

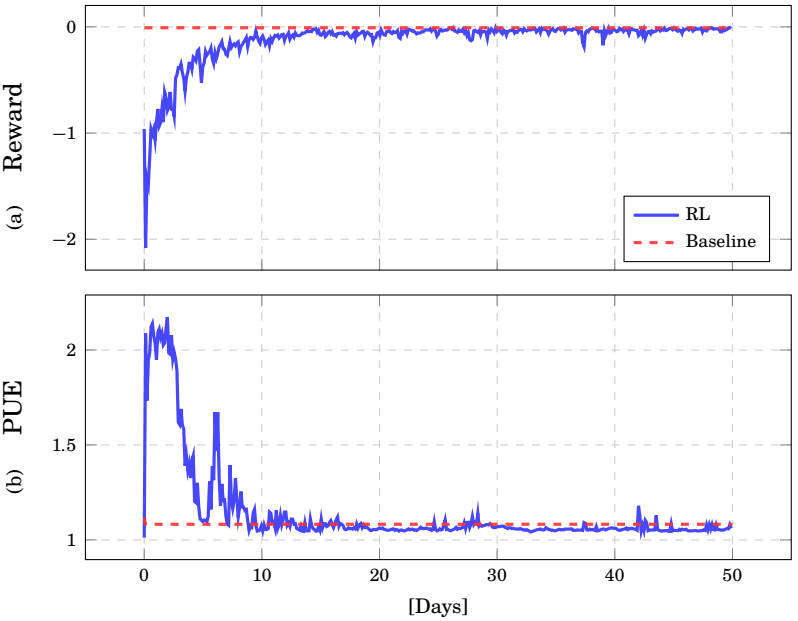

Figure 5.4 The reward Figure 5.4(a) and the PUE Figure 5.4(b) for both RL agent and baseline algorithm. The reward is what the RL agent will explicitly optimize for, while the PUE is what we want to indirectly minimize. The average PUE over the last 10 days is 1.059 for the RL agent and 1.082 for the baseline algorithm.

Results

Figure 5.4 shows the reward defined in (5.5) and PUE defined in (2.1), where the reward is what the RL agent will explicitly optimize for and the PUE is the metric we indirectly want to minimize. While the agent does seem to learn and really close in on the baseline algorithm for the reward in Figure 5.4(a), it does not quite reach the same level of performance. What we do see though is that the PUE in Figure 5.4(b) is on average lower for the RL agent than for the baseline algorithms. Averaging over the last 10 days, the baseline algorithm has a PUE of 1.082 while the RL agent has 1.059. It should be noted that these PUEs are very good, both due to assumptions made in the model, but also that the actual DC we model is developed to be very energy efficient. And though the absolute difference is small, the relative improvement in PUE is around 28% for the RL agent.

The main reason that the RL agent does not reach the same reward is because the baseline algorithm does not use the compressor which comprise a substantial amount of the cooling energy when activated. It also does not

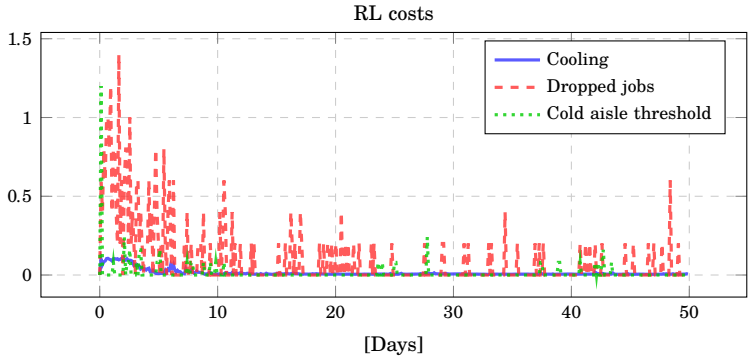

Figure 5.5 The three contributors to the total cost (negated reward) for the RL agent. Both dropped jobs and cold aisle constraints contribute a sparse but large cost, while the cooling power is a more continuous cost. Cooling power has a significant bump in the start coming from the compressor, though similar to the other costs the agent manages to reduce this over time. The baseline is not too interesting to look at, as it only has a small cooling power cost.

drop any jobs, and it keeps the cold aisle temperature threshold. So while the RL agent has to learn how to best balance all these different costs, the baseline algorithm already avoids all but the relatively small fan power cost. In Figure 5.5 the three individual cost terms from (5.5) are shown for the RL agent. While they all decrease over time as the agent learns, they still occasionally incur a cost.

Based on the frequency of the incurred costs, the load balancing seems to be more difficult to learn. This seems reasonable since the action is not as statically dependent on the input as the other two costs. There, compressor and cold aisle costs are quite directly dependent on the relationship between the CRAH temperature setpoint and either the ambient temperature or the threshold temperature respectively. For the RL agent it drops around 0.3% of the jobs over the last 20 days, a significant improvement from the first part, showing that it does learn some load balancing. The optimal load distribution for this simple case is to have it as even as possible, which the baseline algorithm achieves. Looking at the load distribution over the last 20 days, the mean and standard deviation over all the servers are 250 ± 83 W for the RL agent, and 250 ± 3.6 W for the baseline algorithm. So while the RL agent does manage to do load balancing well enough to mostly avoid dropping jobs, it does not yet manage to keep the load as even, and thus as good, as the baseline algorithm.

In Figure 5.6 we see the power distribution over the different components

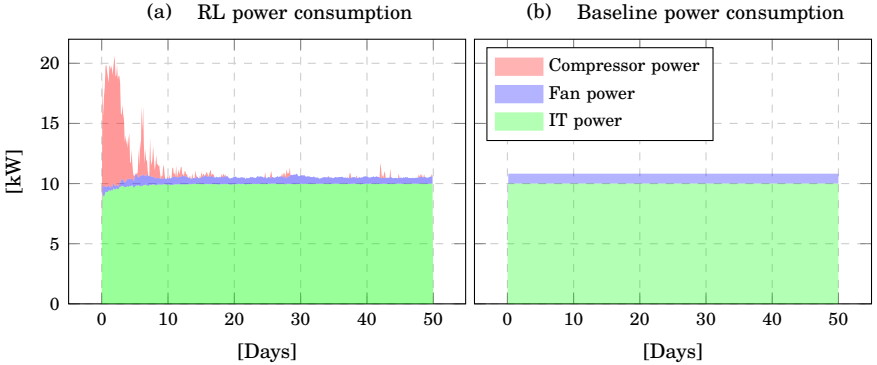

Figure 5.6 Power consumption distribution for the RL agent and baseline algorithm.

for the two algorithms. The baseline algorithm only uses the fans, while the RL agent also uses the compressor in the start, thus incurring a large energy cost. Over time the RL agent learns to control the temperature setpoints to keep the compressor mostly off. It only sparsely turns it on, likely due to exploration in the learning process. The RL algorithm also finds a lower fan power level, though as seen in Figure 5.5 this will occasionally incur a penalty on the cold aisle temperature. We also see that while the RL agent do miss a few jobs in the beginning, the total IT load is later on similar to the baseline. The shape of the RL power curve is (as expected) similar to the PUE curve in Figure 5.4(b), where the peak in the start comes from the compressor usage.

One more inefficiency to investigate is the recirculation and bypass flows. In Figure 5.7 we plot the difference of the CRAH flow and total flow through the servers, which is enough to decide whether we have recirculation or bypass flow according to (5.4). Both algorithms have mostly bypass-flows, though the RL agent evens out the flow a little more compared to the baseline algorithm. In our model it would seem like having neither bypass nor recirculation flow should be the best, since under free-cooling the inlet temperature does not affect energy usage, and only the fan speed will add to that from the CRAHs side. So reducing the fan speed while giving the same volume and temperature of air to the servers should always be better. A likely reason for this is that if the flow is too low the servers will start to recirculate air which will make for warmer inlet air and thus require higher server fan speeds. Since the load varies slightly, and the RL agent is not doing perfect load balancing, it is likely worth keeping a small buffer of extra flow on the side of the CRAH to avoid the servers having to increase their fan speed.

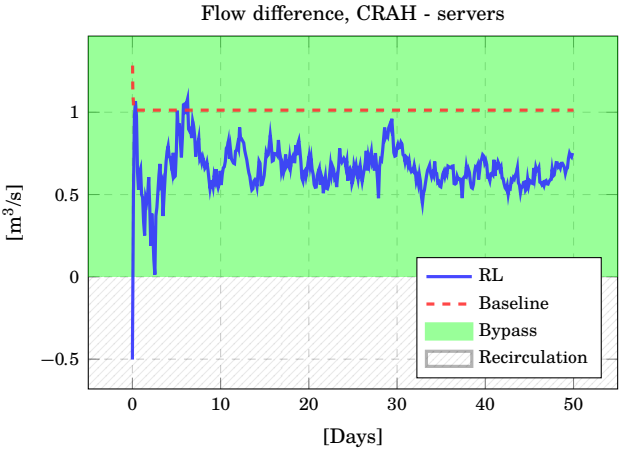

Figure 5.7 The difference between the CRAH flow and total flow through the servers, showing us whether we have recirculation or bypass flows. Both algorithms have bypass flows, meaning the CRAH flow is larger so some air flows directly back to the CRAH inlet.

Remarks on the Results

The RL agent showed that it could learn multiple disparate tasks, and while it did not improve on the reward compared to the baseline algorithm, it did manage to reduce the PUE of an already very efficient DC by 28%. The reward is a more complex objective, where multiple discontinuous parts make for a hard optimization problem for the RL agent.

We would like to see that a good policy in this environment could transfer to a more realistic environment. We do this to some extent in Chapter 6, where we transfer some structure and hyperparameters we found to work well in this environment, though we do not do any direct transfer learning.

With more complex flow dynamics and a varying outdoor temperature, the environment provides a more complex optimization task. An RL agent has the potential to learn all the complex dynamics and find an adaptive policy that can handle the varying conditions. A simple baseline like the one used here will likely not perform as well, leaving more room for the RL agent to improve.

It is also possible that the dense neural network model can be improved upon, something we look at in Chapter 6 by implementing features into the network that are based on the DC structure.

6

Adaptive DC Cooling using Deep RL

Based on the results from Chapter 5, we want to improve the model and scale it up, making for a much more complex environment. Here the RL agent has more opportunity to benefit from the learning-based approach in relation to standard control procedures. By adding a more realistic flow between the CRAH units and the racks we can now have hotspots among the server inlets, e.g., most of the warmer recirculated air ends up at a few server inlets, requiring them to have a larger flow to achieve the same cooling. Our hypothesis is that a controller that is aware of these conditions could utilize the knowledge to improve the cooling performance of the datacenter.

In this chapter we design an RL agent that outperforms common control procedures in a complex environment, showing the benefit of a data-driven approach that can adapt to changing circumstances.

6.1 Extending DC model with CFD

The model used in this chapter is based on much of the same ideas as in Section 5.1, though the model for air flow in the server hall is replaced with a more advanced algorithm based on computational fluid dynamics (CFD), originally presented in [Sjölund, 2018]. In addition to this, the model is scaled up to capture the full size of the DC pod it is based on. The layout can be seen in Figure 6.1, with 360 servers distributed over 12 racks in a hot aisle configuration with 2 CRAH units on each side of the room.

The proposed CFD method can be efficiently parallelized on GPUs, allowing the simulations to be run faster than real-time and be parallelized over many environments, making it feasible for use in an RL training scenario.

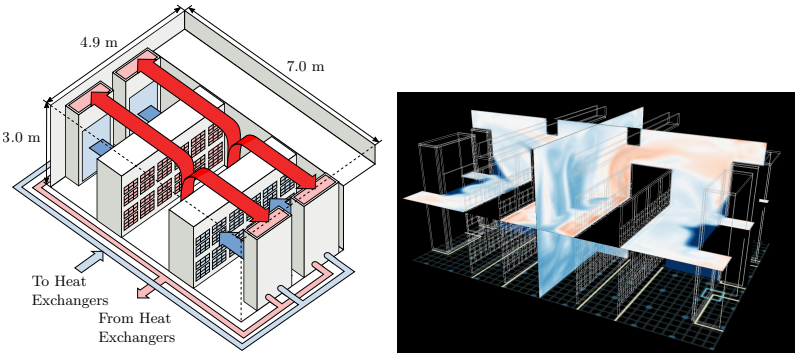

Figure 6.1 *Left:* Idealized hot and cold flows in the server hall, i.e., the "IT space" in Figure 5.2. *Right:* Slice of the CFD simulation of the server hall.

Server, CRAH, Chiller and Dry-Cooler

While the largest change is using CFD for simulating the flow of air in the server hall, there are also some changes to the other parts of the model. The variables and values from Table 5.1 are still used, though some equations and values are updated to better reflect the real system, and will be presented here.

Servers. For the servers, the main change is that they are extended with a model of their thermal mass based on the work in [VanGilder et al., 2013], creating a more dynamic temperature response to changes in load and inlet temperature. The air is assumed to flow across some heat source, raising the temperature based on the current load, and then across a thermal mass that can absorb or release heat depending on the relative temperatures. The temperature rise across the heat source, which is the CPU in this case, is calculated in the same way as in Section 5.1, though we present it in a slightly different way here.

$$\Delta T_s^{cpu} = T_s^{in} \frac{p_s}{C_v Q_s}$$

The model for CPU temperature and air flow is also the same, given by (5.1) and (5.2). A new state for the temperature of the thermal mass is introduced as

$$T_s^{it} = \frac{\tau_1}{\tau_1 + \Delta t} T_s^{it_{old}} + \frac{\Delta t}{\tau_1 + \Delta t} \left(T_s^{in} + \Delta T_s^{cpu} \right),$$

where τ_1 is a time constant deciding the heat uptake, Δt is the time step of the CFD simulation, and $T_s^{it_{old}}$ is the previous internal temperature. The

outlet temperature is then defined as

$$T_s^{out} = T_s^{in} + \Delta T_s^{cpu} + \frac{\tau_2}{\tau_1 + \Delta t}\left(T_s^{it_{old}} - T_s^{in} - \Delta T_s^{cpu}\right), \tag{6.1}$$

where τ_2 is a time constant deciding the heat release from the servers thermal mass. The time constants τ_1 and τ_2 are approximated to be around 10 minutes with $\tau_2 < \tau_1$ from the specification in [VanGilder et al., 2013], and measurements on the servers in the DC. This model is consistent with the one in Section 5.1 since setting $\tau_1 = \tau_2 = 0$, i.e., removing the effect of the thermal mass, will lead to the same equation for the outlet temperature.

Compressor and dry-cooler. The dry-cooler is modelled the same as in Section 5.1, where it removes all heat down to the ambient temperature, and then the compressor needs to remove the rest. The compressor will thus need to remove the energy corresponding to reducing the temperature from T^{amb} to the setpoint temperature T_c^{out} of the current volume flowing through the CRAH, and will do so with a coefficient of performance $K^{comp} \approx 3$. The total power used in the compressor is then a sum over the power used to reduce each CRAH to the corresponding setpoint temperature.

$$p_{comp} = \frac{C_v}{K^{comp}} \sum_{c=1}^{4} \max(0, Q_c(T_{amb} - T_{SP_c})) \tag{6.2}$$

Fan power. The power to the fans are derived in the same way as in Section 5.1,

$$p_{fan} = K_S^{fan} \sum_s Q_s^3 + K_C^{fan} \sum_c Q_c^3, \tag{6.3}$$

though the coefficient for the CRAH units is updated to $K_C^{fan} = 648$ W/(m^3/s)^3 to cover all flows in the CRAH and chiller, estimated to give us reasonable power consumption according to data.

Server Hall CFD Model

The CFD model that is used in this work was presented in [Sjölund, 2018] and defines three-dimensional boundary conditions for the room, servers, and CRAH units. The flows in the server hall seen in Figure 6.1 are modelled using the lattice Boltzmann methods, a class of CFD methods for fluid simulation, using the single relaxation time algorithm [Delbosc, 2015].

Lattice Boltzmann method. The lattice Boltzmann method in its simplest form is based on a uniform grid of statistical distribution functions, called lattice sites, representing density and velocity of fluid particle groups affected by different forces. Evolution of fluid flow over time is computed using the discrete lattice Boltzmann equation

$$f_i(\boldsymbol{x} + \boldsymbol{e}_i\Delta t, t + \Delta t) = f_i(\boldsymbol{x}, t) + \Gamma(f_i(\boldsymbol{x}, t)) + F\Delta t, \tag{6.4}$$

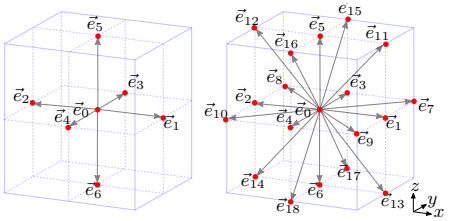

Figure 6.2 The lattice sites use either D3Q7 (left) or D3Q19 (right) discretizations for the lattice velocities $\boldsymbol{e}_i$.

where f is the distribution function of density, $\boldsymbol{x}$ is position, $\boldsymbol{e}$ is the flow directions given in Figure 6.2, Δt is the time step, Γ is the collision operator and F is a body force perturbation [Sjölund, 2018].

The collision operator Γ in (6.4) is implemented according to the Bhatnagar-Gross-Krook model, employing a single relaxation time τ to capture kinematic viscosity and return the perturbed system into a local equilibrium f_i^{eq} [Sukop and Thorne, 2006].

$$\Gamma(f_i(\boldsymbol{x}, t)) = -\frac{\Delta t}{\tau}\left(f_i(\boldsymbol{x}, t) - f_i^{eq}(\boldsymbol{x}, t)\right) \tag{6.5}$$

The macroscopic fluid density is the sum of all distribution functions,

$$\rho(\boldsymbol{x}, t) = \sum_{i=0}^{Q} f_i(\boldsymbol{x}, t),$$

while the macroscopic velocity is the lattice velocities weighted by the distribution functions,

$$\boldsymbol{u}(\boldsymbol{x}, t) = \frac{1}{\rho}\sum_{i=0}^{Q} f_i(\boldsymbol{x}, t)\boldsymbol{e_i}.$$

Buoyancy effects from natural convection are implemented using the Boussinesq approximation [COMSOL, 2023],

$$F_i = \pm\frac{\boldsymbol{g}\beta(T - T_0)}{2},$$

where the thermal expansion coefficient β is constant at reference temperature T_0 and $\boldsymbol{g}$ is gravitational acceleration. This force works along the directions aligned with gravity, i.e., $\boldsymbol{e_5}$ for D3Q7 and $\boldsymbol{e_6}$ for D3Q19 in Figure 6.2.

For simulating thermal evolution, a separate lattice T_i is used. The velocity lattice affects the temperature lattice through advection, while the

temperature affects velocity through buoyancy. Its evolution is described as [Delbosc, 2015]

$$T_i(\boldsymbol{x} + \boldsymbol{e}_i \Delta t, t + \Delta t) = T_i(\boldsymbol{x}, t) - \frac{\Delta t}{\tau_T} \left(T_i(\boldsymbol{x}, t) - T_i^{eq}(\boldsymbol{x}, t) \right),$$

where $T_i^{eq}(\boldsymbol{x}, t)$ is the equilibrium distribution function and τ_T the relaxation time.

Turbulence modelling. Turbulence was modelled by large eddy simulation and the turbulent eddy viscosity is calculated as

$$\nu_t = \frac{1}{6} \sqrt{\nu_0^2 + 18 C_s^2 \Delta^2 \sqrt{\mathcal{S}_{\alpha\beta} \mathcal{S}_{\alpha\beta}}}$$

where ν_0 is the kinematic viscosity for no turbulence model, $C_s = 0.1$ was chosen as the Smagorinsky constant, filter cutoff length Δ is set to unity and $\mathcal{S}_{\alpha\beta}$ is the local stress tensor [Delbosc, 2015].

The relaxation time in (6.5) is then replaced with

$$\tau = 3\nu + 0.5 = 3(\nu_0 + \nu_t) + 0.5.$$

Boundary conditions. There are three different types of boundary conditions used for the three-dimensional model.

Solid surfaces are defined using a no-slip condition that imposes zero velocity. This is implemented by reverting particles leaving the domain in the opposite directions.

Inlet/outlet conditions specify a constant flow velocity $\boldsymbol{u}_0$ and temperature T_0 in the direction of the surface normal $\boldsymbol{n}$. For the servers, the flow velocity is set based on the fan speed (5.2), and the temperature is set to (6.1). For the CRAH, the flow and temperature are part of the control variables.

Finally, the airflow into CRAHs and frontal server air intakes were modelled using a zero-gradient boundary condition, where the air was effectively removed from the simulation by setting the velocity and temperature gradients to zero.

6.2 Context-Aware Control using RL

Compared to the simulation model in Chapter 5, this experiment is a lot more complex. Both due to the more realistic model of the airflow in the DC, but also from having more servers which increases the size of the state and action spaces. While the agent is still an implementation of PPO [Schulman et al., 2017b], this environment required some modifications to the agent to make it more suitable for this environment.

Potential Design Modifications

Initial tests showed that the agent from Chapter 5 did not perform well. We therefore developed some different approaches, both to improve the learning experience for the agent, and to make the learning problem a little easier.

Convolutional Neural Networks. The first idea was to use a 2D convolutional neural network (CNN) over the front-panel layout of the server states. The assumption is that extracting the important features from a server can be done by just looking at the parameters from it and the immediate neighbors, and that this extraction looks similar for all servers. This should reduce the number of parameters needed in the network, and also make it easier for the network to learn how to extract the important features. The CNN output can then be joined with the other input states that do not belong to the server, and fed through a smaller dense network.

This idea can also be applied with a convolutional filter that only depends on a single server, since the most relevant information is probably available there.

Both these avenues were explored, and though it seemed to do a little better than the approach with dense networks, it was still unpredictable and inconsistent in learning. It was especially the load balancing that seemed difficult, and with poor load balancing the state and reward signal becomes very noisy. Thus, it also becomes more difficult for the RL agent to learn the CRAH setpoints, since the state and reward are less informative.

Manual feature extraction. Instead of using CNNs, we can create manual features based on the server layout. A simple feature is to simply average over a rack, and provide that as an additional signal which provides a more compressed and less noisy signal over server features. The problem is that this might not give enough information to do good load balancing, so it would either need to be used in combination with the server features, or we need to solve the load balancing problem differently.

Standard load balancing. As we noted in Chapter 5, the load balancing was harder to learn than the CRAH setpoints, and increasing the size of both state space and action space will only make it harder. Initial experiments showed that the load balancing was still problematic, not providing much benefit and mostly creating a noisy state for the RL agent to learn from. If we instead use a standard load balancing algorithm, we can remove the load balancing from the RL agent and let it focus on the CRAH setpoints. While this would remove the potential benefit of holistic control, we can still allow the RL agent to see the load distribution. This would allow the RL agent to learn how to control the CRAH setpoints in a context-aware manner, i.e., learning to adjust the CRAH setpoints based on the load distribution.

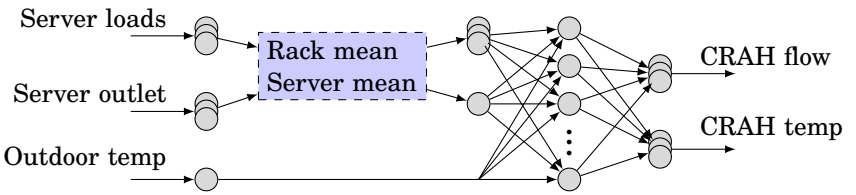

Figure 6.3 The network structure used for the policy. The policy has two layers, the first one with transformations to reduce the space and then a single dense layer before the output layer. The outputs of the policy network are used as mean and standard deviation for normal distributions, which are then sampled to get the CRAH setpoints.

Minimum flow load balancing. Based on the idea that having an even flow among the servers will provide the same cooling for less energy, we try load balancing based on the flow of the servers. Some simple experiments with either helping the RL agent to find this structure, or directly doing load balancing based on the minimum flow, showed no significant improvement. The minimum flow load balancing does reduce the variance in the flow, but it has no real effect on the total flow and energy consumption.

Reinforcement Learning Agent

To figure out what worked, and what hyperparameters to use, we did some initial testing and tuning on the simpler environment from Chapter 5. This provided significantly faster training times compared to running experiments on the full CFD based model, allowing for faster iteration when trying different approaches.

The best agent used a simple load balancer that places jobs on the least loaded server, and then a RL agent that controls the CRAH setpoints. The policy network is embedded with an initial layer doing some simple feature extraction, namely averaging the server metrics over each rack and over all servers. These features as well as the non-server states are then fed through a dense network with one hidden layer and the output layer, see Figure 6.3. The value network is just a normal dense network with 2 hidden layers acting on the full state. Both networks have dense layers with 64 units each using ELU activation functions, see Figure 3.4. For the value function, we use a clipping parameter of 1000 and a discount factor of 0.99.

The sizes of the policy and value network were established through hyperparameter tuning. One possible explanation to the difference in size is simply that the loss landscape is different for the two networks. The loss of the value function can be more complex, needing the increased flexibility of a larger network. Larger networks can also be more stable during learning, leaving a more robust target for the policy network to learn from. These are

just speculations based on observations from the hyperparameter tuning, and we have not done any additional experiments to validate them.

State space. The state $\boldsymbol{s}$ consists of room measurements of the server outlet temperatures T_s^{out}, the loads p_s for all servers, as well as the outdoor temperature T^{amb}.

$$\boldsymbol{s} = \left([T_s^{out} \dots], [p_s \dots], T^{amb}\right) \tag{6.6}$$

Similar to Chapter 5 we apply feature scaling to the states, using min-max normalization to the range $[-1, 1]$. The ambient temperature is approximated to stay in the range $[0, 30]$ in this experiment.

Action space. The actions are continuous flow and temperature setpoints for each of the 4 CRAH.

$$\boldsymbol{a} = \left([T_c^{out} \dots], [Q_c \dots]\right) \tag{6.7}$$

The actions are sampled from different normal distribution, where the mean and standard deviation for each action distribution is given as outputs from the policy network. That means a total of 16 outputs from the policy network, values for the mean and standard deviation for both temperature and flow control for each of the 4 CRAHs. The actions are, similar to the states, also scaled using min-max normalization from the range $[-1, 1]$ to the range of viable temperature and flow choices for the CRAHs.

Reward function. The reward r is designed to capture the energy cost of the cooling systems, as well as penalizing the server inlets for being above 27°C, similar to Chapter 5.

$$r = -\left(C_1\left(p^{fan} + p^{comp}\right) + C_2 \sum_s \max(0, T_s^{in} - 27)\right) \tag{6.8}$$

Compared to (5.5), the cold aisle cost is proportional to how much the temperature is above 27°C, instead of a binary penalty. This makes for a smoother reward function, which is easier to learn from, as well as being more in-line with a realistic objective. The weights $C_1 = 10^{-5}$ W^{-1} and $C_2 = 1$ K^{-1} are calibrated to put a relatively large penalty on the cold aisle temperature, while also scaling the reward to a reasonable range.

6.3 Evaluating the RL Approach

The RL agent is evaluated using the simulated environment in a few different scenarios, and the performance is compared to two baseline algorithms. The environment is parameterized to show interesting behavior. In particular, an outdoor temperature where it is not always possible to

use free-cooling to uphold the cold aisle thresholds, and a workload that is not always so high that the cooling system would be best of running at maximum capacity. We run both a scenario under normal operation, and a scenario where equipment malfunctions. The failure scenario involves a reduced cooling efficiency, exploring how the RL agent adapts to the changing circumstances.

All code except for the CFD simulation are publicly available and can be found on GitHub[1].

Training and Simulation Setup

The actual training was done using a set of Nvidia V100 GPU clusters running Ray [Moritz et al., 2018]. The GPUs were mainly used to power the CFD simulations, since that was the most computationally expensive part of the training, though they were also used for the RL training. Ray has excellent support for running large numbers of parallel jobs on distributed platforms, as well as managing resources such as GPUs and assigning how they should be used. With this said, Ray can be cumbersome to work with, and is not easy to adapt to non-standard workflows.

We use the PPO implementation from Ray RLlib [Liang et al., 2018], and train it using eight parallel environments. Parallel environments allow on-policy algorithms to collect a more varied batch of data, since each environment can have slightly different initial conditions, reducing the variance in the gradient estimates. Each step in the environment is 1 second, faster than many other works on similar problems. While these faster interactions does bring the risk of creating noisy actions, the problem with slower sampling is that can take a full time step to react to changes in the environment.

We collect 200 steps of data in each of our 8 environments between each time we train, creating a batch of 1600 samples to train on. This is used to train the agent using mini-batches of size 128 over 30 epochs. We only visualize the interactions from the first environment, and the data is averaged over 200 steps.

Ambient temperature. The outdoor temperature is based on historical weather data from Luleå, Sweden, where the DC pod is located. Figure 6.4 shows a linear interpolation over an hourly average of the temperature, retrieved from the local weather forecast SMHI [*SMHI* 2021], and is used as the outdoor temperature in the simulation.

Workload. The workload is similar to the one used in Chapter 5, though with slightly different parameters. In the simulation, each job adds a constant load of 20 W for 1.5 hours, and in each step of the simulation there

[1] https://github.com/albheim/rldc_rafsine

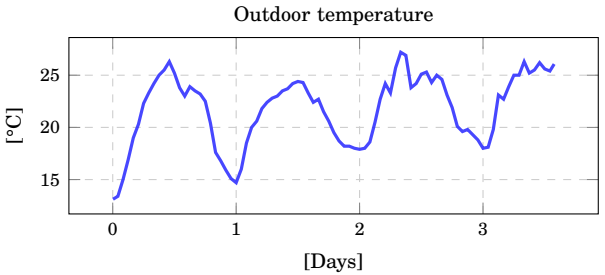

Figure 6.4 Temperature data for Luleå from SMHI [*SMHI* 2021] used in simulation.

is a 50% probability of a job arriving. The values were chosen to create a good average load for the DC, while still having enough variability to make the decisions interesting.

This workload can represent different kinds of services found in a datacenter, such as cloud services that start up for a while when the demand increases, as well as batch style jobs typical to high-performance computing environments. The static duration and energy requirement are trivial to extend to a more realistic workload, as is the arrival rate. In the current setup this would only affect the variability of the load that the RL agent sees, since load balancing is treated separately.

In Chapter 5 the agent dropped jobs when the server was at maximum capacity, but here we instead move those jobs to the server with the lowest load. Misplaced jobs will still incur a penalty, but it will not affect the total load of the system. By keeping the load constant between the different algorithms, we can more fairly compare the energy consumption of the cooling system.

Baseline controllers. We use the same type of baseline algorithm as in Chapter 5, but implementing two instances with different temperature setpoints for the CRAH units.

The baseline with 18°C setpoint is good at keeping the cold aisle temperature down, but will run the compressor more often.

The baseline with 22°C setpoint is much more energy-efficient, but will sometimes fail to adhere to the cold aisle threshold.

Using both as benchmarks for the RL agent should give a good comparison of how well it fares in each objective.

Normal Operation

Running the DC under normal conditions the three algorithms generate the reward in Figure 6.5, where the RL agent achieves the highest reward on

average. When the outdoor temperature in Figure 6.4 is below 18°C it allows for running the CRAH at minimum temperature with no compressor, and the 18°C baseline algorithm is more efficient here. With additional training, the RL agent might also learn to reduce the CRAH temperature to 18°C in these cases, but it is not unexpected that it is somewhat careful since lower temperatures can suddenly incur a large energy cost from the compressor.

While it can seem unfair to use an objective for which the RL agent is optimized as the measure of performance, it does show that the RL agent does learn what we asked it for, a policy that performs well according to the objective. And given that the RL agent could learn this, it would likely also be able to learn to optimize other objectives that are not too different.

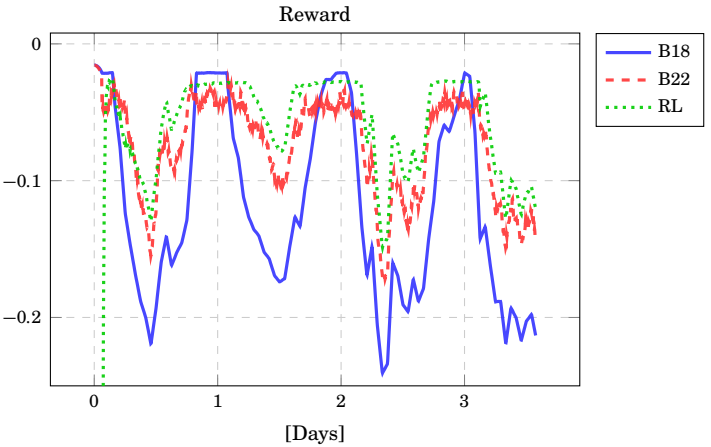

Figure 6.5 The reward (6.8) of the RL agent and the two baselines, B18 and B22. While the RL agent achieves a higher reward than the baselines most of the time, the performance varies a little with the outdoor temperature.

The total reward is based on both the cooling energy and the adherence to the cold aisle threshold, both of which are shown in Figure 6.6. The RL agent matches up quite well with the 22°C baseline when it comes to energy consumption, both outperforming the 18°C baseline. However, looking at the cold aisle loss in it is apparent that the 22°C baseline does not keep the cold aisle below the threshold, while the 18°C baseline incurs no loss and thus perfectly avoids the threshold. The RL agent balances the two objectives, keeping the cold aisle cost low after an initial learning period, while also keeping the energy cost low.

One thing to note is that the peaks in cooling energy cost happen in conjunction with the peaks in outdoor temperature, see Figure 6.4. These peaks are the same as in Figure 6.7, since the compressor runs more

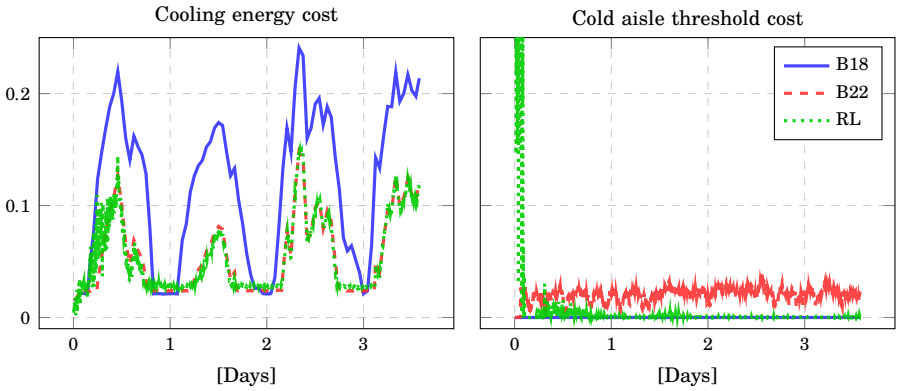

Figure 6.6 The costs that make up the reward (6.8), stemming from cooling power and the cold aisle threshold. In relation to the cost, the RL agent performs similar to each of the better agent in both cases, while the worse agent is showing a much higher cost.

when the outdoor temperature is high relative to the CRAH setpoint, thus adding a substantial energy cost. This shows how one of the most important problems is to learn to minimize the compressor usage, though this problem interacts with the problem of providing enough cooling to the DC.

In Figure 6.8 the inlet temperatures for all 360 servers are plotted for the three algorithms. Here we see how the RL agent tries to optimize against the boundary that was set, and while the 22°C baseline keeps many of the states around the same values as the RL agent, it violates the threshold by a couple of degrees on a few servers. The servers with higher inlet temperatures turn out to be in the upper parts of the rack. This is reasonable since the hot air flowing back to the CRAH goes above the servers, and re-circulation will thus come from above. The RL agent manages to balance the CRAH setpoints better to avoid these effects, and make the server inlet temperatures stay under the threshold.

As the workload maintains a relatively stable average load, it is no surprise that the PUE in Figure 6.9 replicates the shape of the cooling cost in Figure 6.6. It is clear that the 18°C baseline is worse when it comes to power consumption, and while the other two strategies are similar in power, the RL agent is better at adhering to the cold aisle temperature thresholds.

Adaptation to Disturbances

In addition to running the DC under normal conditions, we also conducted experiments to show how the agent continuously adapts to changes. We

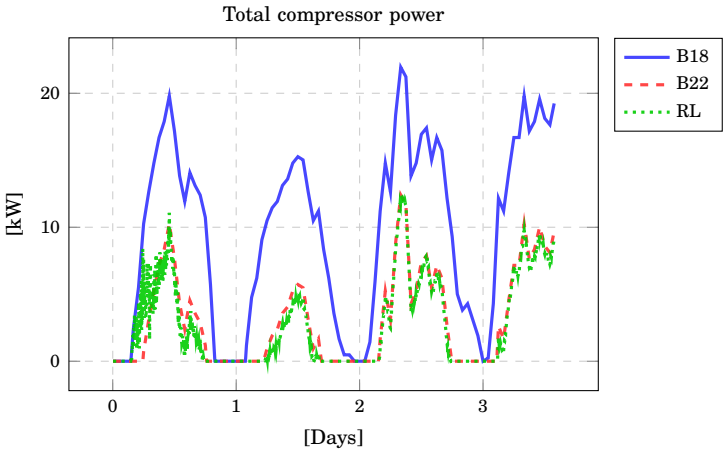

Figure 6.7 The energy used in the compressor for cooling (6.2) is a major part of the energy cost, and the RL agent learns to keep this low.

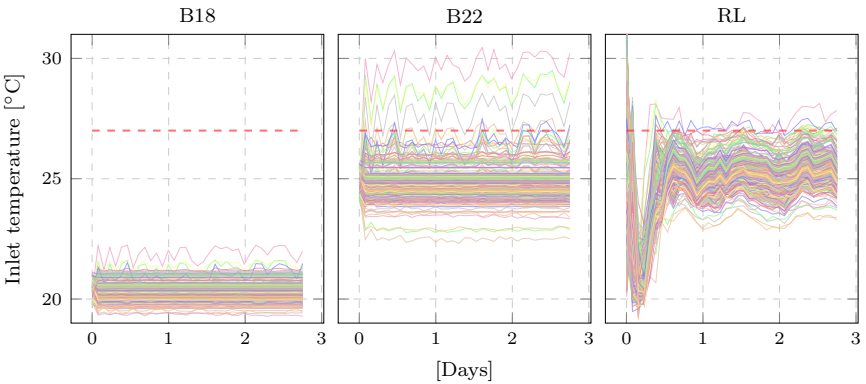

Figure 6.8 Temperature distribution of server inlets over the three algorithms. The red line is the 27°C threshold used in the loss calculations. The inlets that actually do go above the threshold tend to be in the upper part of the rack, which is reasonable since any re-circulation of hot air from the server will come from above.

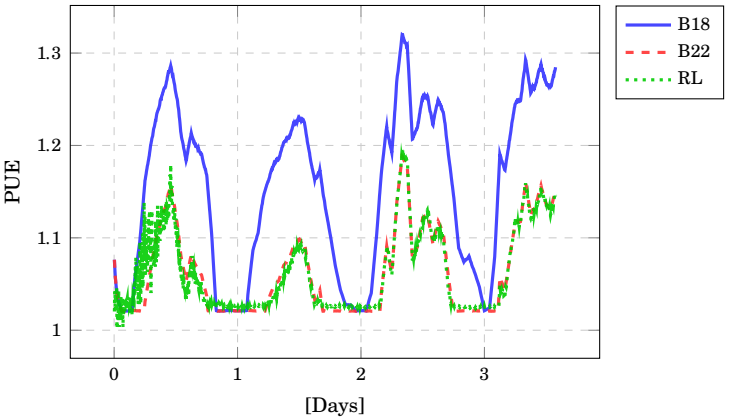

Figure 6.9 PUE for the RL agent and the two baselines.

do this by introducing an inefficiency in $CRAH_0$, resulting in the CRAH operating with only 80% of the original airflow, but using the same power.

The left figure in Figure 6.10 shows how the cold aisle threshold is temporarily broken by the RL agent when the inefficiency is introduced at day one, and how it manages to come back to a similar state as before within half a day. The right figure in Figure 6.10 shows how $CRAH_0$ loses efficiency, and how the flow setpoint is increased to make up for parts of the lost flow. There is also the neighboring $CRAH_1$, sharing the same cold aisle, which also increases its flow a bit to support the slightly broken $CRAH_0$. This is done without the RL agent getting any information that these CRAH units are in the same aisle and can support each other, and can be done since the agent is able to adapt over time and is aware of the shared context.

Summary and Discussion

We train an RL agent to control a DC cooling system in a more realistic environment, and show that it can learn to outperform standard control strategies. Compared to the baseline algorithms, it is able to adapt to changing circumstances, and is able to optimize for a more complex objective. Compared to manually designing the controllers, it is easy to give the RL agent more context that is not directly related to the objective, and let it learn how to use the information in a useful way. What might make RL less that optimal for these types of environments is that the learning process is typically very unstable, and the agent needs to explore to gain an understanding of the environment. This means that the agent might take actions that cause the system to go into dangerous states, possibly causing expensive failures to the physical systems.

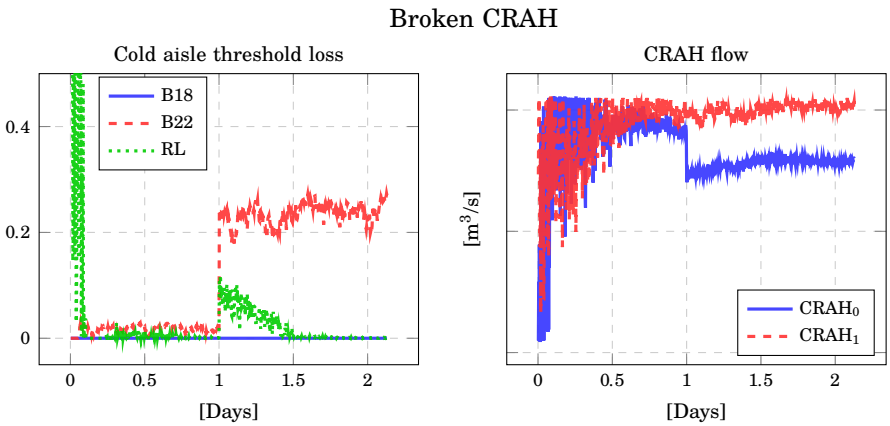

Figure 6.10 An experiment where one of the CRAH units ($CRAH_0$) experiences a problem that reduces its efficiency by 20%. The RL agent adapts to this change, pushing the loss from the cold aisle threshold back to a similar level as before the change. One noteworthy detail is that we also see an increased flow in $CRAH_1$, the CRAH unit that is next to the broken one.

Another problem is the scalability of the approach, where increasing the size of the DC will make both state space and action space grow, making it much harder to learn. Approaching the scalability problem, there are a few interesting techniques that could be used. One approach that we touched upon in this work is the idea of embedding knowledge in the network structure by, e.g., using convolutional layers for the server inputs. This could likely be extended further, making for more specialized structures that provide a good structure for learning this type of information. Another approach is to look at hierarchical RL, creating a disentangled controller structure that is more modular and scalable.

A future consideration is also to train an RL agent with only local knowledge to give a better insight into how much the RL agent actually gains from the larger context, and how much of the improvement is just from being able to optimize against the objective. This is not trivial, since it would entail multiple smaller agents that need to coordinate their actions, but it would give a better understanding of the benefit of the RL approach.

7

Proactive Cloud Autoscaling using RL

With the cloud becoming a more mature technology, and with 5G and edge computing becoming a reality, running timing-sensitive systems in the cloud is becoming a realistic possibility. One example is the digitalization of industrial automation systems, see Figure 7.1, which holds the promise of improving efficiency and resource utilization through data-driven innovations. Providing reliable and efficient computing resources with low latency is a key enabler for these systems to move to the cloud, though providing end-to-end timing guarantees with state-of-the-art technologies is typically not possible. There are numerous challenges when it comes to deterministic virtualization technologies, virtualized real-time networking, etc., as well as control plane issues related to how resources are allocated in a timely fashion. The focus of this chapter is time-sensitive services where each request is associated with a deadline, which is typically the case for, e.g., cloud-based control systems [Millnert et al., 2018; Skarin, 2021].

Problem Description

Microservice architecture is the predominant style for modern cloud applications, where a single microservice can typically be part of many applications, e.g., an authentication service which may be needed by different applications all parts of a larger system. Figure 7.2 shows a network of connected microservices that together form different applications in the cloud. Each cloud application handles different types of workloads and has a *call graph* spanning over a subset of the microservices, i.e., a path that the workload traverses as it is being processed. These graphs can easily grow quite complex, as the *microservice death star* example in Chapter 2 demonstrates. They can also be dynamic and data-dependent, making it even harder to infer behavior.

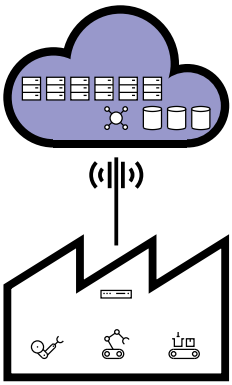

Figure 7.1 Connecting factories and other critical infrastructure to the cloud holds promises for optimal resource utilization and increased efficiency.

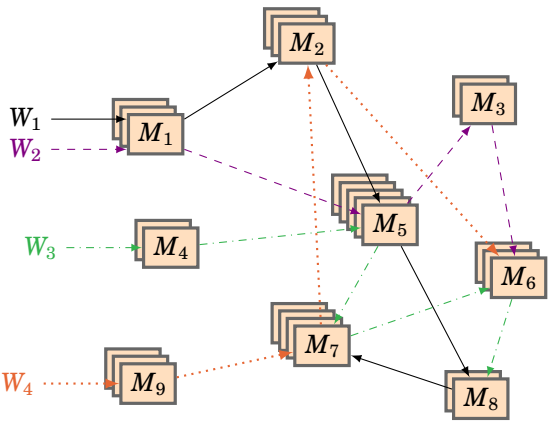

Figure 7.2 A set of microservices forming four cloud applications that receives different types of workloads, W_1 to W_4. Each application requires a different set of microservices, and though there is some overlap, the call graphs are different, i.e., the paths the workloads go through differ. Each microservice can be scaled independently, adding or removing replicas of the service.

Given the system in Figure 7.2, we consider how a simple reactive scaling strategy would work for it. The workload W_1 enters at M_1 and causes further calls to M_2, M_5, M_8, and M_7. If there is a rapid increase in demand, a reactive autoscaler would simply detect an increase in utilization at each of the involved microservices, one at a time. This means that the scaling of M_7 does not occur until the traffic reaches it, although it could have easily been predicted if the call graph was known ahead of time, leading to an unnecessary and unwanted delay in scaling.

A proactive scaling strategy could instead have the understanding that when workload W_1 increases, the capacity of microservices M_2, M_5, M_8, and M_7 can increase proactively.

Related Work

Automatic resource scaling has traditionally been done using reactive approaches based on utilization thresholds, though there are some more advanced techniques emerging in modern tools, see Section 2.2. There will always be a trade-off between the ability to serve a rapid increase in load while minimizing the buffer of unused allocated resources on standby; the tighter the resource allocation, the more cost-efficient the system will be.

Previous works on learning-based methods for automatic resource scaling are presented in Section 4.3, and although many of the methods are interesting, they typically simplify the problem in different ways we want to avoid, or they solve a slightly different problem. An example is simplifying the state space by considering a single microservice in isolation, which removes the possibility to scale based on the state of other microservices. Or scaling over long timescales to reduce noise from the actions, which also reduces the ability to be proactive since any reaction to a change will be delayed. Works such as [Bitsakos et al., 2018; Bibal Benifa and Dejey, 2019] use RL, but reduce the size of the action space by only allowing the algorithm to add or remove a single instance each step. This might be enough in many cases, but is certainly restricting the ability of the policy. RL is also applied in [Cheng et al., 2018; Dong et al., 2020], though they assume knowledge about the structure of the call graph. This can be extracted in some cases, but might not always be easily available, and there are benefits to be able to dynamically adapt to a changing structure. In [Kumar et al., 2018; Toka et al., 2020] they instead aim to model the future workload, and scale the system based on the model predictions. Given the inherent uncertainty in estimating future workload, we aim to instead rely on the fact that there are certain temporal dependencies in the internal service chains. These dependencies can be a more reliable source for predicting future load over the internal nodes, and can thus be used to make more informed scaling decisions.

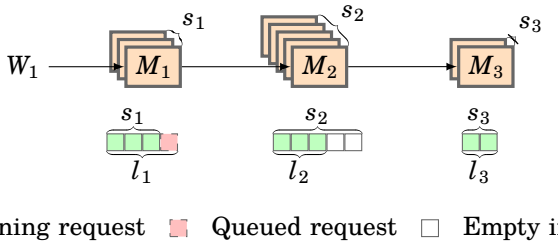

Figure 7.3 A single workload W_1 spanning 3 microservices M_i. The scale s_i denotes the number of replicas of each microservice, while l_i denotes how many requests are currently being processed or waiting to be processed. The workload W_1 has a path $P_1 = [M_1, M_2, M_3]$ associated with it, a service time on each microservice on the path $t_{j,i}$ and a deadline T_{d_j} for each request's total allowed execution time.

Our approach. Instead of explicitly modelling the call graph, and all the intricacies that come with that, we explore how well an RL agent can learn an implicit model given minimal data from the microservices. Some implicit representation of the underlying call graph will need to be learned in order to be able to act proactively, but this will be embedded in the agent's internal neural networks. This will effectively enable the policy to create feed-forward connections between the load of one microservice to the desired state of another based on how it estimates the workload behavior. For time-sensitive workloads with real-time constraints, the idea is that we should be able to reduce the spare capacity while still hitting the deadlines.

This requires an RL agent that acts on a short timescale and takes the whole service chain into account to be able to do proactive scaling. We do not assume knowledge about the workloads and do not try to predict the incoming loads, nor do we have any information on the call graphs. We also consider the latency, for example from boot time, when adding new resources.

7.1 Modelling a Microservice Application

We aim to create a simple simulated model that captures the important aspects of the microservice architecture described in Section 2.1. The environment and how it works should capture the scenario in Figure 7.2, though Figure 7.3 provides a better overview of the details of the model.

Each microservice M_i processes requests from an arrival queue in a first-in-first-out (FIFO) manner. The queue has a max length of q_{max}, and if the queue is full on a request's arrival, the request is dropped. The scale s_i denotes the number of replicas of M_i, and can take on any integer inside the

interval $[s_{min}, s_{max}]$ by booting or closing instances. Booting a new instance takes T_{boot} time, while closing takes T_{close} time. The total number of requests on M_i is l_i, representing both requests being worked on and requests in the queue. The utilization of M_i is then the load over available resources,

$$u_i = l_i / s_i. \tag{7.1}$$

Each workload W_j is assumed to traverse the graph one microservice at a time. They follow a service chain, defined by a path P_j, the order in which the microservices should be visited by requests from W_j. A deadline D_j defines the total time allowed for the request to traverse the path, while the service time $t_{j,i}$ is the time required to process a request from W_j on the microservice M_i.

Proactive Parameterization of the Simulation Model

Not all parameterizations of this type of environment will lend themselves to the problem we set out to solve. We specifically want to look at environments that, like Figure 7.2, has chains where it could be possible to scale quicker if scaling is done with respect to the full set of microservices.

Taking a simple scenario with a single workload running over three microservices as in Figure 7.3. The entry point M_1 will not be able to scale proactively without making guesses about random changes in arrivals, which is not the goal of this work. But for $M_{i>1}$, it is possible to start scaling before the workload reaches the microservice. To fully boot an instance $M_{i>1}$ proactively would require that

$$\sum_{k=1}^{i-1} t_{1,k} \geq T_{boot} \text{ for } i \in [2, 3], \tag{7.2}$$

i.e., that the total service time over the previous microservices is at least as long as the time to boot a new instance.

We also need to make sure the deadline is met to gain any value from finishing the request. If each microservice can immediately process the request, the response time will be $t_{1,1} + t_{1,2} + t_{1,3}$. However, if either of the microservices are fully utilized and scales reactively, it will instead take at least $t_{1,1} + t_{1,2} + t_{1,3} + T_{boot}$ time to finish. By setting this as the upper limit for our deadline, we can ensure that proactive scaling is required in order to avoid buffers on M_2 and M_3.

$$t_{1,1} + t_{1,2} + t_{1,3} + T_{boot} > D_1 \tag{7.3}$$

To also make sure it is actually possible to meet deadlines we want that the total work time is less than the deadline.

$$\sum_i t_{1,i} < D_1 \tag{7.4}$$

Selecting $t_{j,i} = T_{boot} = 1$ fulfills (7.2) for M_2 and M_3 allowing them to scale proactively, and (7.3) and (7.4) then gives that $3 < D_1 < 4$. Choosing, e.g., $D_1 = 3.5$ will thus make any reactive approach have to choose between either keeping a large enough buffer on all microservices or miss deadlines as the workload fluctuates. A proactive approach could instead scale M_2 and M_3 based on the state of microservices earlier in the service chain, allowing it to meet deadlines while not keeping a buffer anywhere but on the first microservice.

7.2 Proactive Control of Microservice Application

Proactive control entails some form of prediction of how the system will evolve, and then using this to make decisions. While there are proactive controllers that solely rely on time-series predictions, we want to give the controller more context, allowing it to make more reliable predictions and thus better decisions.

One way to achieve this is to model the system dynamics explicitly, and use that to predict how the system evolves. This is the approach used in Chapters 8 and 9, and while it can be efficient if possible, the creation of accurate enough models can present many challenges. We look at using model-free RL to implicitly capture the required understanding of the model dynamics. This does require a bit more data to train, but allows us to not assume any knowledge about the structure of the microservice application or underlying infrastructure, and can thus be more flexible.

Reinforcement Learning Agent

We used the Soft Actor-Critic (SAC) algorithm from [Haarnoja et al., 2018b], the version with automatic tuning of the entropy parameter α to match a given target entropy $\mathcal{H}$. It was chosen based on some initial testing of different algorithms, also including DDPG and PPO, where it seemed to perform well and be a little more robust to hyperparameters than the other algorithms.

State. The state observation consists of the current scale s_i and the utilization u_i for each microservice. What the agent actually observes is the state stacked with the previous state in time, to alleviate the problem of the state not being fully observable, and to allow for information about changes in state to be incorporated. This can also help in learning when there are delays in the environment, so that the agent can easier deduce what action led to what change in state, and ultimately which reward it should associate with the action.

$$\boldsymbol{s} = \big([s_i(t)\dots], [u_i(t)\dots], [s_i(t-1)\dots], [u_i(t-1)\dots]\big)$$

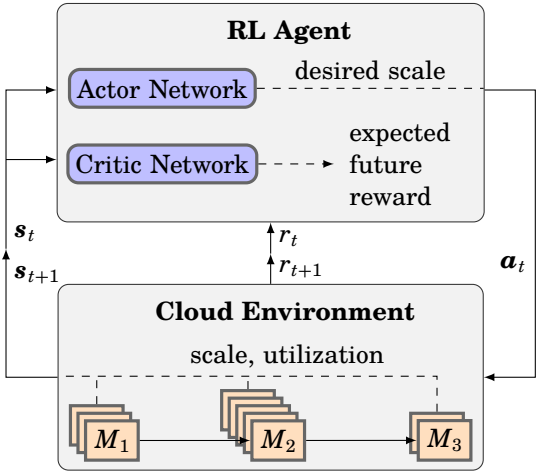

Figure 7.4 The RL agent contains both actor and critic networks, and interacts with the cloud environment by scaling the microservices and observing utilization and reward.

The state is normalized to the range $[-1, 1]$ before being fed to the neural networks.

Action. The controlled variable is the target scale s_i for each N microservices, and is an integer in the range between $s_{min} = 1$ and $s_{max} = 10$. The upper bound is simply set to something a little above what we expect to need. And though lower bound of zero could also be reasonable, experimental observations showed it made the RL agent more likely to get stuck in the local optimum of scaling in as much as possible.

While there are SAC implementations that handle discrete action spaces [Christodoulou, 2019], this environment provides an actions space with interval values. This means there is an inherent order and spacing, i.e., if the agent is in a state where it predicts scaling $s_i = 2$ is the best action, $s_i = 3$ is likely also a decent action, while $s_i = 10$ is probably not. For a categorical action space this knowledge is neglected, leaving the policy to learn this from scratch. Thus, we opt to use the original SAC with continuous actions, rounding the actions to the nearest valid integer. This is done to preserve the inherent ordering that exist in the action space, which can help the learning process. As an off-policy method it should also not affect the learning step that we round the actions, since the learning does not assume that the experiences used for training is from the current policy.

With continuous actions, the policy network outputs two values for each

action dimension $\boldsymbol{\mu}, \boldsymbol{\sigma} = \pi(s)$, the mean and logarithm of the standard deviation for a normal distribution. Each continuous action is then determined by individually sampling each distribution $\hat{a}_i \sim \mathcal{N}(\mu_i, \exp(\sigma_i))$, and scaling to the desired range. By rounding to the nearest valid integer we get an action in the space

$$\boldsymbol{a} \in [s_{min}, s_{min} + 1, \ldots, s_{max}]^N.$$

Reward. There is a value U_j to finishing a request of type W_j before the deadline D_j, and a cost C_i to running instances of type M_i. The number of requests that finished within the deadline during this step is denoted f_j, giving

$$r = \sum_{j}^{|W|} U_j f_j - \sum_{i}^{N} C_i s_i. \tag{7.5}$$

The coefficients U_j and C_i should be selected so that the values outweigh the costs for a single request, incentivizing the RL agent to finish requests. The coefficients are also used to scale the size of the reward to be manageable for the NN training.

7.3 Evaluating Proactive Scaling Approach

To evaluate the proactive RL approach we simply run it in the simulated environment and compare against a few other approaches. Both the environment and the RL agent were implemented in the Julia language, selected for its performance and ease of use for scientific computing. It was also a distinction compared to Ray which was used in Chapters 5 and 6, allowing for much more flexibility in the implementation.

The code for the RL agent and microservice simulation used to generate the results here is all published on GitHub[1].

Training Cluster Setup

We set up a cluster of virtual machines to run the experiments and hyperparameter tuning. Since the neural networks used are quite small, and the environment was not computationally heavy, we could run the training on CPUs.

We use the SAC algorithm provided in ReinforcementLearning.jl [Tian, 2020]. The original implementation only supported action spaces with a single dimension, so we extend the implementation according to [Haarnoja et al., 2018b] to support multidimensional action spaces.

[1] https://github.com/albheim/ServiceMeshControl/tree/thesis

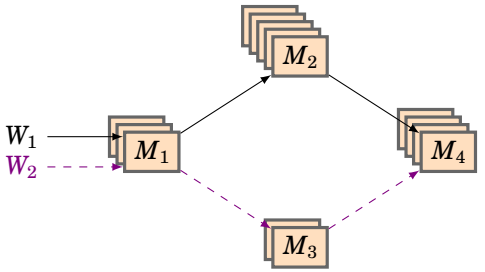

Figure 7.5 An environment with two workload chains traversing four microservices, some shared and some individual. Each microservice has a number of replicas and can be individually scaled.

To simplify the distributed workflow we created the package DistributedEnvironments.jl[2], automating the distribution of local development environments to remote workers.

Environment

To evaluate the algorithm we employ a slightly more complex version of Section 7.1, with multiple workload chains traversing four microservices, see Figure 7.5. Parameters are selected to accommodate proactive scaling as described in Section 7.1, and are the same to a large extent with service time $t_{i,j} = 1$ s, deadline $D_j = 3.5$ s, and boot time $T_{boot} = 1$ s. The scaling range is set to $s_{min} = 1$ and $s_{max} = 10$, and the max length of the queue is set to $q_{max} = 5$. Having a queue can make the problem a little harder, since it allows for some buffered requests to build up. With our setup, a single buffered request can make the system miss the deadline, making it problematic for the RL agent to *catch up* with queued work since it will not receive any reward for it. Given that T_{close} does not really introduce any interesting new dynamics, since the interesting delays were already introduced with T_{boot}, we simply set it to zero in these simulations. We assume that all microservices have the same cost $C_i = 0.5$.

Workload. We create a simple synthetic workload that generates requests according to a random walk over integers in a constrained range. This could for example be a set of control applications turning on and off randomly, adding a constant load while they turn on, removing it when they turn off, see Figure 7.6. It should create enough variation in the states to both make the problem interesting, and allow for robust and efficient learning.

So assuming we currently have an arrival rate of x_j new requests every step, then the workload will with a probability p_j pick a new random arrival

[2] https://github.com/albheim/DistributedEnvironments.jl

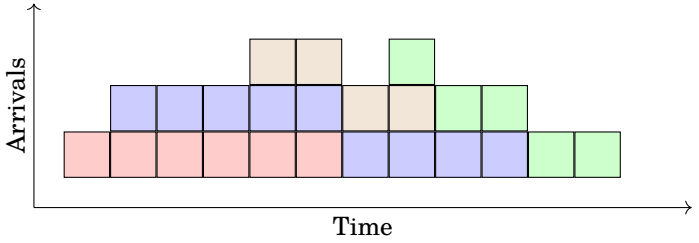

Figure 7.6 A visualization of a synthetic workload representing randomly arriving constant loads that are active over some duration.

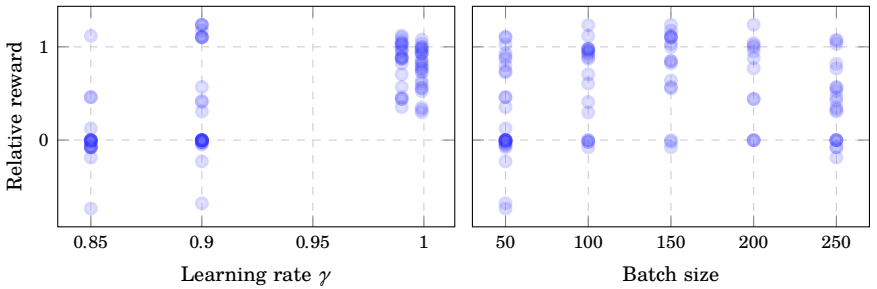

Figure 7.7 Hyperparameter optimization over many parameters, of which two are shown here. Each dot corresponds to a random set of values for multiple parameters, that had the specific value for the displayed parameter. The relative reward indicates the ratio of the average reward received compared to the simple agent presented in Section 7.3.

rate in the range $[\max(x_j^{min}, x-1), \min(x_j^{max}, x+1)]$. We assign the same value $U_j = 8$ to finishing requests from both workloads.

Hyperparameter Tuning

Though SAC is supposed to be robust to hyperparameters, some tuning still played an important role in making successful agents. We based our hyperparameter optimization on Hyperopt.jl [Bagge Carlson, 2018], though contributed some work to improve the functionality for distributed systems.

An example of the tuning is visualized in Figure 7.7, showing a scatter plot of how well different values of a parameter performed. The relative reward is the ratio of the average reward received compared to the simple agent presented in Section 7.3, and was used to make the results depend less on the random seed for the environment.

For the parameter γ we can see that $\gamma = 0.85$ neither shows a good average nor achieves the maximum value, and even though $\gamma = 0.9$ achieved

Table 7.1 Hyperparameters for RL agent using SAC.

Parameter	Value
optimizer	Adam
learning rate for optimizer	0.003
activation function	ELU
discount factor γ	0.93
target network smoothing τ	0.01
target entropy $\mathcal{H}$	10
value net hidden layers	4
value net hidden units	120
policy net hidden layers	2
policy net hidden units	20
mini-batch size	100
learning rate for α	$5 \cdot 10^{-5}$
update frequency	8
replay buffer size	$5 \cdot 10^{5}$
start algorithm	random action
steps with start algorithm	50000

the highest scores, it still has a large spread and most trials seem to end up with a reward of 0, so assuming we want a little robustness it is likely not the best choice either. The agents receiving zero reward indicates that they got stuck in a local optimum corresponding to scaling in all microservices to s_{min} to reduce the cost, and as the queues grow it will not receive any reward since all requests will miss their deadline. Exploration is difficult in this situation, where scaling out on a single service will not help since the reward is only given when a request finishes in time. For the agent to get out of this situation, it would require simultaneously scaling out on all services in a chain for long enough to empty all queues enough to finish a request in time, all while seeing a negative trend in the reward from the increased cost. Lower γ means a lower horizon over which the cumulative reward is considered, and it makes sense that this will prioritize minimizing the immediate costs over maximizing uncertain future rewards, thus easier ending up in this optimum. For the batch size parameter, something in the middle seem better in all ways, both best value and higher value for the main cluster of results.

The parameters used in the experiments presented here are shown in Table 7.1. Similar to in Section 6.2 we note that the network for the policy is smaller than the value functions network. It is difficult to know why, but one possibility is that the value function not as nicely behaved as the policy, and thus requires more parameters to model well.

Using random actions for the initial data collection seems to provide

slightly better results than using the simple agent, though it is not a large difference. It probably provides a bit more exploration, and as long as it provides small amounts of positive rewards, i.e., finishing requests in time, it might give a better dataset for the initial training.

Using ELU instead of the more common ReLU as activation function seems to provide better results in our experiments, though it is a minor difference. It is also a slower function, so if training time is a large problem it could be worth trying ReLU instead.

Baseline Algorithms

We compare the RL agent to a few baseline algorithms, both standard reactive ones and a proactive one that is as close to optimal as we can get with a simple method that does not know the future.

Simple. A basic reactive strategy is to look at the average utilization u_i over all machines for microservice M_i and if this is larger than some u_{max} a machine is added, if it is lower than some u_{min} a machine is removed. This only looks at utilization and can only scale by one machine per step, but it manages well with reasonably chosen limits and is simple to implement and understand. For these experiments $(u_{min}, u_{max}) = (0.5, 0.8)$ is used.

Kubernetes. The horizontal pod autoscaler in Kubernetes (2.2) and (2.3) is a slightly more advanced reactive strategy also based on thresholding, implementing some extra functionality to avoid jitter in the signal and erring on the side of caution when it comes to scaling in. We use the current utilization (7.1) as the scaling metric with a target utilization of $\bar{u} = 0.8$, and implement jitter reduction using $\epsilon = 0.1$ and a window length $W = 60$ s.

Oracle. We also introduce a scaling method which has more information about the state of the microservices than the other methods, though still no information about the future, i.e., the incoming load. It is called oracle to indicate that it has knowledge not available to the other algorithms.

The extra information help with proactively scaling the later microservices in the chain, and consists of the distribution of request types on a certain microservice, the structure of all the workload chains, the service times, and the boot times. With this, the oracle can calculate exactly when a request will arrive at a certain microservice, and how long before that it should start booting a new instance to process it. The first microservice in a request chain will still have to be reactive, and here the strategy is to always keep one extra instance running per request type.

For the oracle to work well there are some specific conditions which must be fulfilled, though the environments used for evaluation are designed to allow for the oracle to be optimal in the sense of proactive scaling for all

nodes but the first in a request chain. So while it will likely not be optimal in regard to (7.5), it is still a good benchmark for how well it is possible to scale proactively.

Results

The results from the two environments in Figure 7.3 and Figure 7.5 are presented here.

All data presented is sampled and smoothed to make it possible to visualize what is going on. The simulation environment logs data every simulated minute, which is then processed through a moving average over 1000 data points and downsampled by a factor of 1000 for nicer plotting.

The experiments presented use different random seeds from the ones used for hyperparameter optimization, ensuring that the agent was not overfitting the hyperparameters to the specific seed.

We show 50 days of data in the plots to give a view both of the initial learning phase and how well the agent stabilizes with time. The simulations are run over a longer time than shown here to verify that the RL agent does not enter catastrophic forgetting, a well-known phenomenon for neural networks learning in a sequential manner.

Two simple workload chains. For the two workload chains visualized in Figure 7.5, we have W_1 producing requests in the static range $[0, 4]$, and W_2 producing requests in the static range $[0, 2]$. Each have a probability 0.1 of changing each step.

The reward in Figure 7.8(a) shows how the different strategies perform over time. The RL agent quickly learns a good enough strategy to reach a reward close to the oracle scaler, though with a bit more noise. Much of the noise in the reward comes from the stochasticity of the workloads, and it can also be seen for the other strategies. However, the RL agent has some larger spikes in its reward which can be connected to failed requests, as seen in Figure 7.8(b). The failed requests are from either finishing after the deadline for the request, or from the request being dropped due to full queues. The larger spikes are a likely due to a combination of the RL agent having a stochastic policy and continuously learning and exploring how to improve the policy, and when trying to improve utilization it scales in a bit too much, leading to queues filling up and many following requests missing their deadline.

The RL agent does consistently fail a small part of requests, which can in part be attributed to the reward (7.5) that does not explicitly tell the agent to not miss any requests. So by keeping a higher utilization, the RL agent can earn enough reward that it might be worth failing a few requests. It might be possible that with more training, and reducing the stochasticity of the policy, the RL agent could learn to keep a high utilization while

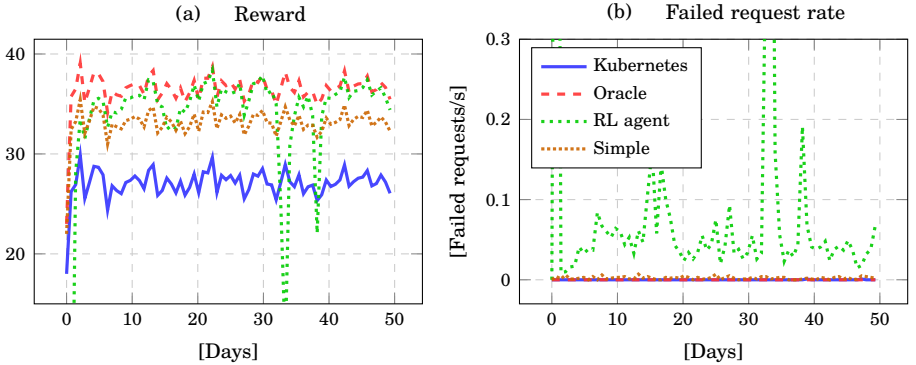

Figure 7.8 Figure 7.8(a) show the reward for the double chain environment in Figure 7.5. The larger dips in the RL agent reward can be connected to the failed requests in Figure 7.8(b) stemming from continually learning and exploring.

reducing the failed request rate. While the simple strategy also fails a few (barely visible) requests, this could be solved by changing the thresholds, though it would lead to a larger buffer on average. Neither the oracle nor the Kubernetes scaler fail a single request. This is expected since the oracle was designed to keep the lowest scale, while still guaranteeing no misses for this type of workload. The Kubernetes scaler looks at the maximally needed number of instances over a period, and as such will tend to have a larger buffer in favor of missing requests. This also leads to a lower reward for the Kubernetes scaler, and is the trade-off for being a safe strategy in that it promotes keeping a large buffer over risking scaling in too much and missing a request. The environment is certainly not optimal for the Kubernetes scaler, and we are not trying to claim it is a bad strategy, it is simply not optimal for the environment and objective we selected to be able to show the benefit of proactive scaling.

In Figure 7.9 the request rate for the different workloads as well as scale and utilization for the microservices are shown. The oracle scaler has a utilization around one on the microservices where it can scale proactively, while the utilization for the RL agent is not far behind in some cases, but it is significantly noisier. This is both due to continually learning and exploring, but also that the oracle scaler has information about the request distribution on each microservice, information that is not available to the other scaling strategies. The Kubernetes scaler is cautious about scaling in and tends to keep a larger buffer on average for this type of noisy loads, and will as such have quite low utilization.

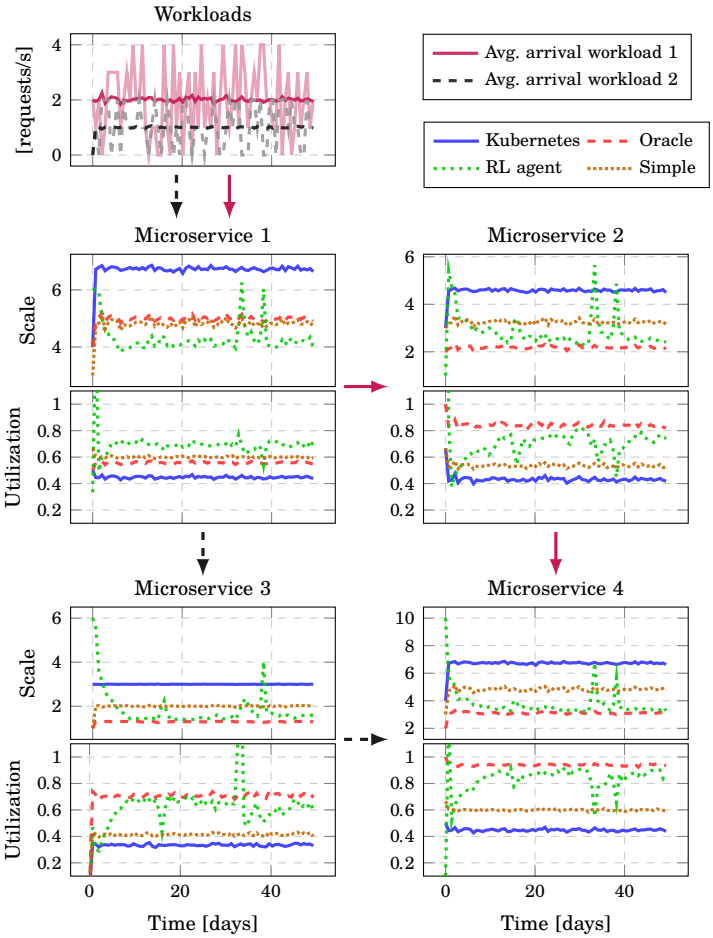

Figure 7.9 The double chain has two workloads (W_1 and W_2) with different average loads (top left) going through both shared microservices (M_1 and M_4) as well as individual ones (M_2 and M_3).

Looking at the utilization for the RL agent we see similar spikes for the first microservice as we see in Figure 7.8(b), showing how much of the failed requests likely stem from pushing the utilization a bit high for the entry point of the service chain. For microservice 2 and 3 we see that the utilization is quite high, but still contains some stochasticity. This is since the total load of microservice 1 is known to the agent, not the individual distributions of requests from workload 1 and 2. Microservice 4 has the highest utilization for the RL agent, stabilizing quite close to one. This should be expected since it has knowledge of the incoming load from microservice 2 and 3.

Adapting to new or changing workloads. Still based on Figure 7.5, we introduce workloads where the range of possible arrival rates change over time, thus changing the average load. A third workload W_3 is also introduced, with path $P_3 = [M_1, M_2, M_3]$. W_1 produces requests in the range $[0, 8]$ for the first third of the duration, and then $[0, 4]$ for the rest of the duration. W_2 produces requests in the range $[0, 2]$ for the full duration. W_3 produces no requests for the first two thirds of the duration, and then $[0, 2]$ for the rest of the duration. Each workload still have a probability 0.1 of changing each step.

The reward for this environment is shown in Figure 7.10(a), and we see that the RL agent quickly adapts to the change in workload as well as the new workload, keeping a reward on par with the oracle scaler. While the reward for the RL agent is a bit noisy for the same reason as before, stochastic workload and continual learning and exploring, it still manages to keep the reward high on average. There are some spikes from failed requests in Figure 7.10(b), and more failed requests in general compared to the other scaling strategies, but as discussed before this can be tuned with the objective function, and is not the main focus of this work.

Figure 7.11 shows the incoming requests for the different workloads, as well as the scale and utilization for the microservices. We see that the RL agent scales out and in as needed, and that the utilization is quite high for the microservices where it can scale proactively. Why it is not higher is due to the stochasticity of the workload, and the fact that the agent does not have information about the distribution of requests on the microservices. We also note that microservice 3 has quite low utilization until the new workload is introduced, then it increases. This is due to the fact that workload 2 will generate zero requests a significant amount of the time, and microservice 3 is not allowed to scale in that far leading to a lower utilization. When the new workload is introduced, the utilization increases since zero requests will happen less often.

Other experiments. While these two experiments help show that the RL agent can learn to scale proactively, and do so in a dynamic and changing

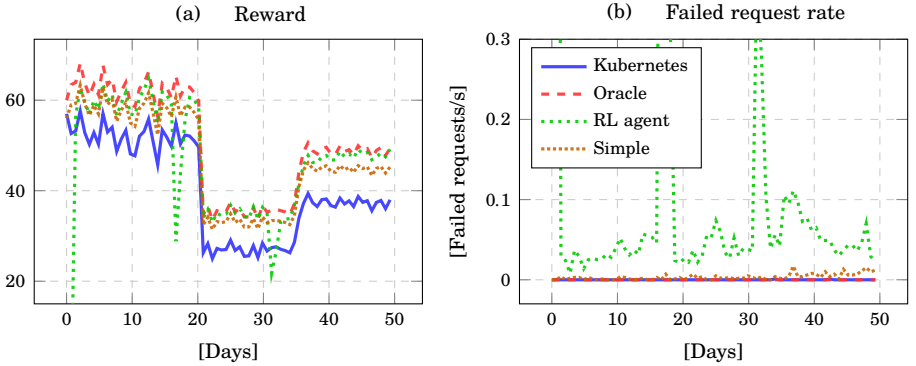

Figure 7.10 The reward under changing workloads. After a third of the time the average request rate for W_1 is halved, and after two thirds the new workload W_3 is introduced. The RL agent has no problems adapting to the changes, and thought the reward is a bit noisier, it is on par with the oracle scaler.

environment, we do have some additional results that we will just touch upon briefly as they do not particularly add to the main points of this work.

In an effort to reduce the failed requests, we increased the reward of a finished request compared to the cost of running an instance. This will shift the value function towards keeping a larger buffer, and should thus reduce the failed request rate. While the higher value for finished jobs did push the RL agent to be more careful by keeping a larger buffer, slightly reducing the rate of failed requests, it also made for a less pronounced proactive component in the strategy.

We also explore more complex workload paths with varying execution times, where proactive scaling was not always possible. The RL agent manages to learn a competitive policy, though, as expected, with less pronounced proactive scaling and thus fewer benefits compared to the reactive strategies.

Summary and Discussion

The environment presented here is difficult from a learning perspective in a few different ways. One is that it is not fully observable; so while we can see the utilization for a microservice, that does not tell us how much work there is left on the current requests, or what different types of requests populate a microservice. This makes it harder to understand the value of states, since the same state could have different values based on some hidden internal state. There are also delays, e.g., when booting new instances it takes time

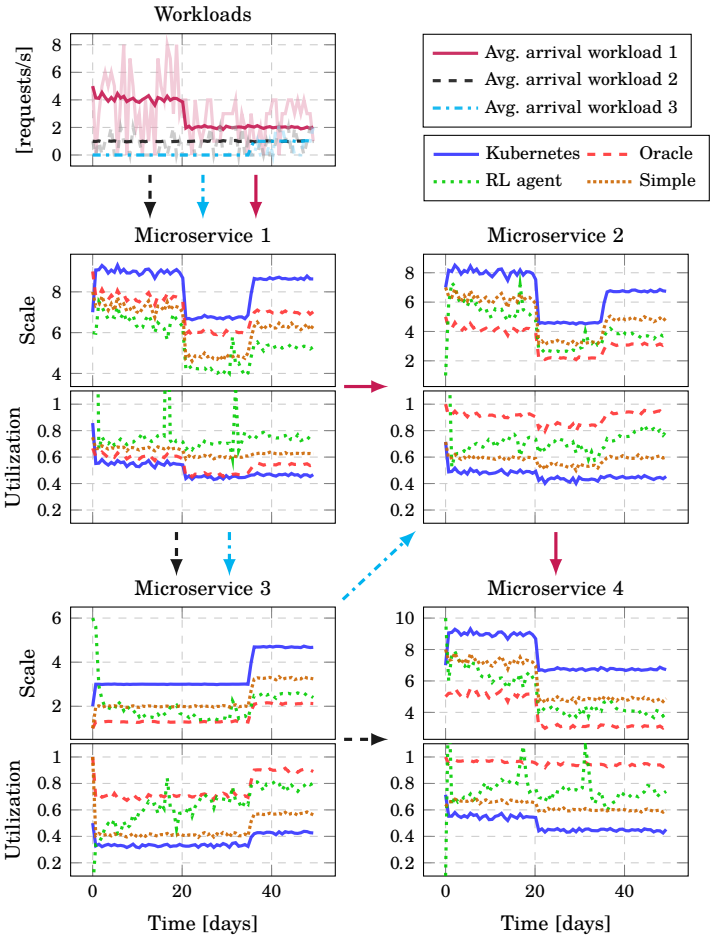

Figure 7.11 We now have three workloads with, two with changing average loads. The requests all have different paths through the set of shared microservices.

between the decision to scale before the state actually changes, and there is a delay between a change in the state that will lead to a request passing through and the request actually passing through to generate a reward.

In addition to the partial observability, the RL agent also has the stochasticity from the learning process and exploration, leading to much noisier actions. This is especially problematic when the environment, as in this case, has modes of operation where the distance between a good state and bad state can be small. This means a small change in the action can lead to a large change in the outcome. One way to make the RL policy more stable could be to reduce the target entropy over time, or simply acting based on the mean of the distribution instead of sampling from it. This would provide worse data for continuous learning, but if we assume that the environment is not changing and that we can stop training at some point, it could provide a more stable policy.

The reward also contain dynamics with different delays, where reducing the scale immediately reduces the running costs, but it can take time to get a request through to generate a positive reward. This especially holds true if the queues are backed up, and the RL agent will have to scale out multiple microservices for a while before seeing a reward. The tendency seemed much stronger when the environment was allowed to scale in to zero instances, and instead setting the minimum to one made it much less prevalent. The problem with backed up queues was also the reason the maximum for the queues was set rather low, introducing the problem of the buffer, but keeping it small to not let too much accumulate.

In the end, the aim was to create a proactive cloud scaler using reinforcement learning, achieving feed-forward scaling for service chains. We show how the RL agent is keeping utilization higher on microservices later in a service chain, indicating that the RL agent embedded some knowledge of the graph structure and learned the forward connection needed to do proactive scaling on the later microservices in the chain. We also show how the RL agent quickly adapts to changing or new workloads. The RL agent achieves a higher utilization than the reactive scaling strategies, but not really reaching the utilization of the oracle scaler for the nodes where it can scale proactively. This was done without giving the RL agent any prior knowledge of the environment or workload, learning only based on utilization values, which can often be easily extracted in a real system, as well as the reward function.

While we went with the approach of stacking previous states to mitigate the problems from the partially observable environment, a different approach could be to use a recurrent neural network to allow the agent to remember previous states. Comparing the effect and performance of these two approaches would have been interesting to explore, but was not the main focus of this work.

8

Load Balancing via Fluid Model Differentiation

One of the large benefits with the cloud is the flexible run-time management of applications, allowing for dynamic scaling and migration of resources for individual microservices in the application. These tools give us fine-grained control when balancing resource usage against the performance constraints from SLOs. With the current advances in edge computing, we also have the choice of whether to run the microservices in the cloud or at the edge. These options provide differences in, e.g., communication delays, resource availability, resource cost, and will thus affect the performance and cost of the application. Creating a run-time management system that can take these differences into account could therefore provide better routing strategies between the microservice replicas, i.e. replicated instances of a microservice, and thus better performance and cost for the application.

Introduction

In this chapter, we demonstrate how *automatic differentiation* can be used to learn load balancing parameters for a cloud application. We look at an application that consists of multiple microservices, where each microservice has a set of replicas that can be placed over multiple different sites. Figure 8.1 shows a distributed microservice where the incoming workload is distributed over replicas according to a weighted random scheme with probabilities p_i. Due to differences in communication delays, available capacity, etc., the access latency and the cost vary among each site. The incoming workload is constantly changing, meaning that any strategy must be able to adapt to variations in traffic.

Our method requires a model of the application from which performance metrics can be extracted, and a cost function and constraints that can be evaluated from these metrics. Using the fluid model introduced in [Ruuskanen et al., 2021a], the model parameters are fit to tracing data from an

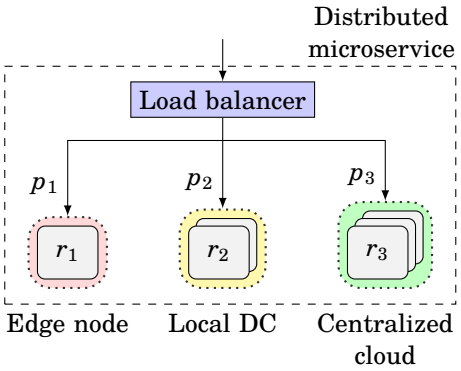

Figure 8.1 A distributed microservice with replica sets spanning multiple different sites. The load balancer distributes the incoming load randomly with probabilities p_i over the available replica sets. The microservice can also be part of a larger application of multiple communicating microservices.

application running live in a Kubernetes cluster. Every few minutes we extract a new model and calculate how the system is expected to evolve under current load balancing parameters. The evolved state is used to extract performance metrics, from which the cost function is evaluated. We then use automatic differentiation to calculate gradients of the cost with respect to the load balancing parameters, and update the parameters with a step that is bounded in the probability space. As both model extraction and automatic differentiation is relatively cheap we can do this in an online fashion, interacting with the application in real-time. This allows the model to adapt to changes in the application, and to changes in the environment, such as the edge cloud becoming more or less available. In an experimental evaluation, we show that this approach can improve the performance of the application and adapt to changes in the environment, all while supporting complex costs and constraints. We specifically focus on adapting the load balancer probabilities in order to minimize some cost function, but the technique itself is more general.

Queuing Networks

The fluid model used in this chapter is an approximative solution to the flows in a queuing network model of the application. We will here give a brief introduction to queuing networks and their use in modelling cloud systems, and compare our approach to previous work in this field.

Queuing theory is a popular method for modelling cloud systems, especially considering the queuing disciplines *first-come first-serve* (FCFS), *processor sharing* (PS) and *delay* (INF) [Balter, 2013; Ardagna et al., 2014].

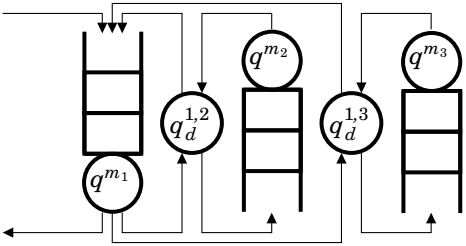

Figure 8.2 Multiclass PS queues q^{m_i} connected by INF queues $q_d^{j,k}$. Each arrow displays a class-to-class transition and is marked with the downstream class it connects to.

Dependencies between resources and servers can be modelled by joining multiple queues, forming a *queuing network*, see Figure 8.2. Each queue can also contain multiple classes of requests, e.g., different types of requests or different processing stages of the same request.

Unfortunately, queuing networks seldom have closed-form solutions, and evaluation by simulation is often prohibitively computationally expensive for real-time usage. Instead, important performance metrics are often approximated using different methods. For stationary solutions of product-form networks there is the mean-value analysis (MVA) and its extensions [Bolch et al., 2006]. Fluid models can produce transient solutions that do not require the assumptions of product-form networks, and they are fast to solve using modern methods. This makes them suitable for our online optimization approach.

Although *join-shortest-queue* (JSQ) and its extensions promises a good mean response time in a load balancing scenario, they are difficult to analyze under more general settings and costs. We thus restrict ourselves to study the weighted random strategy, which allows us to express the application as a queuing network with probabilistic routing and subsequently the optimization problem as finding the probabilities that minimizes our cost. This problem has been extensively studied in the queuing theory community, but in order to make it feasible only simple cost functions and performance metric constraints (if any) are typically considered for specific types of queues and networks. We will here state some notable results, and refer to the references within for a more encompassing description of this field.

Regarding open networks, response time minimization has been studied in, e.g., [Fratta et al., 1973] considering FCFS queues with link constraints using flow deviation, [Borst, 1995] considering FCFS queues over a weighted sum of response times, and [Guo et al., 2004] considering queues of either FCFS and PS discipline with constraints on response time variance. For

closed networks, throughput maximization has been studied for product-form networks in, e.g., [Kobayashi and Gerla, 1983] using gradient stepping as the gradient is readily obtained from the MVA algorithm [Bolch et al., 2006], and [Anselmi and Casale, 2013] using closed-form heuristics based on heavy-traffic limits which were later explored in [Wang and Casale, 2014] for heuristic weighting of the RR strategy. In [Hordijk and Loeve, 2000] it is shown that for product-form networks with general cost function and constraints on queue states, a Nash equilibrium can be obtained via deterministic routing. Further, in [Incerto et al., 2017; Incerto et al., 2018] *model-predictive control* (MPC) over the mean-field fluid model for a closed network was studied in order to minimize response time by tuning both routing probabilities and autoscaling.

Compared to these results, our approach of gradient stepping using automatic differentiation is more general. It allows for a very general class of cost functions and constraints, with the main requirement that all performance metrics used are obtainable from the fluid model. Further, the fluid model allows us to consider both transient and stationary metrics in the system, providing more flexibility in the cost function. However, in adopting such a general approach, we forgo any theoretical results on optimality bounds, feasibility, and convergence speed.

Automatic Differentiation

Automatic differentiation is a technique to evaluate the derivatives of functions defined by computer programs. The basic idea is to apply the chain rule to the code in order to reduce it to simpler expressions where the derivative of each individual operation is easily defined. There are a few different methods for automating this process, but for this work we choose an implementation using a type of *dual numbers* [Eastham, 1961].

Dual numbers provide a convenient way to propagate information about the value of an expression in conjunction with the derivative of the expression. So if we have a language where the higher level code is agnostic to type, and the lower level operations are defined for dual numbers, both the value and the derivative can be calculated in one go by supplying parameters as dual numbers.

Automatic differentiation distinguishes itself from numerical differentiation by being *exact* in the mathematical sense. While numerical differentiation approximates derivatives using finite differences, and thus suffers from the inaccuracies coming from those techniques, automatic differentiation will do a calculation corresponding to the exact mathematical expression. Numerical differentiation also has the disadvantage that higher order derivatives become computationally complex and involve larger errors, while gradients over multiple variables can cause an unnecessary overhead if the

primal evaluation of the function is expensive [Rall, 1981].

8.1 Microservice Application Model

Fluid Model

We adopt the fluid model for microservice applications introduced in [Ruuskanen et al., 2021a]. The model is based on a mean-field approximation of a simplistic queuing network representation of the application. In short, the queuing network models each replica to every service as a multiclass PS queue and each replica-to-replica delay as a multiclass INF queue. Each processing stage of a request across the entire microservice application is then modelled as a path over the classes in the network. Poisson arrivals are assumed, and the service time in each class is allowed to follow a *phase-type* distribution [Asmussen et al., 1996], representing the service time as the time-to-absorption in an internal Markov chain. The transient states of this chain are referred to as *phase states*. Although the individual replica models are simple, this fluid model can capture quite general graphs of microservices, and be completely extracted at run-time from commonly collected tracing data such as, e.g., arrival and service time for each request. However, due to the simplicity of the model, there can be signs of modelling errors when predicting too far from the operating point it was fit to.

For the fluid model we let $\mathcal{Q}$ be the set of queues, $\mathcal{C}$ the set of classes and $\mathcal{S}$ the set of phase states in the network. Each queue q represents a replica of some microservice, and has a unique set of classes $\mathcal{C}_q$. Each class within a queue represents different types of requests, or just different processing stages of the same request. And each queue/class pair $(q, c) \in (\mathcal{Q}, \mathcal{C}_q)$ is then assumed to have a unique set of phase states $\mathcal{S}_{q,c}$ to model the distribution of the service time for that class in that queue. Further, let k_q be the number of parallel processes working on requests in queue q. Let $\boldsymbol{\lambda} \in \mathbb{R}_+^{|\mathcal{C}|\times 1}$ be the Poisson arrival rates to each class and $\boldsymbol{P} \in \mathbb{R}^{|\mathcal{C}|\times|\mathcal{C}|}$ the class-to-class routing probability matrix. Finally, let $\boldsymbol{\Psi} \in \mathbb{R}^{|\mathcal{S}|\times|\mathcal{S}|}$, $\boldsymbol{B} \in \mathbb{R}^{|\mathcal{S}|\times|\mathcal{C}|}$, $\boldsymbol{A} \in \mathbb{R}^{|\mathcal{S}|\times|\mathcal{C}|}$ be the parameter matrices for the phase type distributions of each class, stacked into block diagonals in an appropriate order.

The *smoothed mean-field fluid model* from [Ruuskanen et al., 2021a] can then be introduced. It approximates the mean request population in each phase state $\mathbb{E}[\boldsymbol{X}(t)] \in \mathbb{R}^{|\mathcal{S}|\times 1}$ with the solution $\boldsymbol{x}(t) \in \mathbb{R}^{|\mathcal{S}|\times 1}$ to the following system of ODEs.

$$\frac{d\boldsymbol{x}}{dt} = \boldsymbol{W}^T \boldsymbol{\theta}(\boldsymbol{x}, \boldsymbol{z}) + \boldsymbol{A}\boldsymbol{\lambda} \tag{8.1}$$

The initial condition is $\boldsymbol{x}(0) = \boldsymbol{X}(0)$, and $\boldsymbol{W} = \boldsymbol{\Psi} + \boldsymbol{B}\boldsymbol{P}\boldsymbol{A}^T$. The processor

sharing function

$$\theta_i(\boldsymbol{x}, \boldsymbol{z}) = x_i g_{Q(i)}(\boldsymbol{x}, \boldsymbol{z}) \quad \forall i \in \mathcal{S}$$

defines a soft constraint on the evolution of the phase states. $Q(i)$ is a function mapping a phase state to its parent queue and

$$g_q(\boldsymbol{x}, \boldsymbol{z}) = \frac{1}{\left(1 + \left(k_q^{-1} \sum_{i \in \mathcal{S}_q} x_i\right)^{z_q}\right)^{1/z_q}} \qquad q \in \mathcal{Q} \tag{8.2}$$

is the smoothing function, where $\mathcal{S}_q$ is the set of phase states in queue q. The smoothing parameter z_q can be individually tuned for each queue q to improve accuracy in the model.

Microservice Application

We consider a cloud application subjected to requests from external users with an exponential inter-arrival time, i.e., Poisson arrivals, and where there are constraints on certain performance metrics, e.g., response time percentiles, via SLOs. The application consists of multiple microservices, each consisting of a set of replicas. The replicas can span multiple placement possibilities, e.g., machines, clusters or even different sites, each associated with their own communication delay and resource cost. The cost depends on things such as the cost of electricity, availability, required hardware, etc., and will be highly specific to the application and deployment. What we will assume, is that the cost is tied to *where* the requests are executed among the placement possibilities.

The end goal is to tune application management parameters to minimize the total cost of running the application, while not violating the constraints. In order to effectively determine such parameters, a model can be used to estimate the impact of them on both cost and constraints. Moreover, given such a model, it is possible to use automatic differentiation to differentiate the cost derived through this model with respect to these parameters. By updating the parameters using a gradient stepping scheme, we get a control strategy that drives the system towards a region of lower cost while avoiding constraint violations. In this chapter, we exemplify such a procedure by considering load balancing between the different replica sets in the application. Thus, our parameters to tune, or *control variables*, will be the set of all load balancing parameters.

An illustration of this kind of environment is presented in Figure 8.3. The load balancer is placed outside any site for conceptual understanding, but for a practical implementation, the load balancer would exist on all sites of the microservice to reduce unnecessary network traffic. This would also allow a more general approach where each individual replica set has its own parameters for load balancing, which could further reduce redundant traffic between sites.

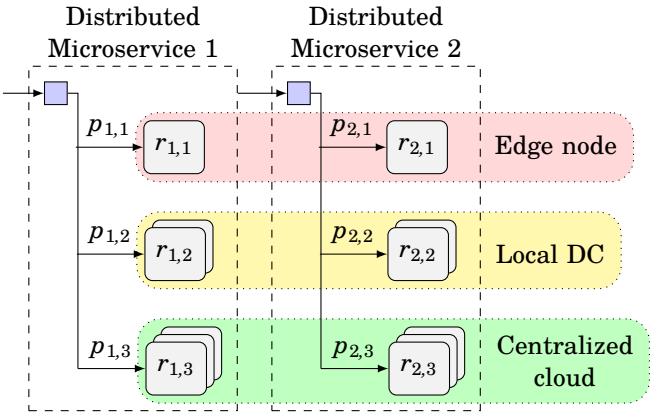

Figure 8.3 An application consisting of two distributed microservices, each with an internal load balancer and replicas $r_{i,j}$ spread over different sites. The control variables are the set of all routing probabilities for the load balancers, where $\boldsymbol{p}_i = [p_{i,1}, p_{i,2}, p_{i,3}]$ is the distribution for microservice i.

Control variables. Using the smoothed mean-field fluid model described in (8.1), the routing between the services is fully determined by the class-to-class transitions described in the routing probability matrix $\boldsymbol{P}$. As the model captures different request types as stand-alone classes, multiple non-zero values in the rows of $\boldsymbol{P}$ only occurs for load balancing purposes. Let $\boldsymbol{p}_i$ be a vector of the non-zero probabilities of a row in $\boldsymbol{P}$ which corresponds to a load balancer i. Let $\mathcal{P} = \{\boldsymbol{p}_1, \boldsymbol{p}_2, \dots\}$ be the set of all load balancing probability vectors in the system. The elements of $\mathcal{P}$ are our control variables, and by changing them we can affect the solution of (8.1). We denote this dependence as $\boldsymbol{x}(t \mid \mathcal{P})$.

Obtaining Desired Metrics

The cost and constraints for the application are based on different performance metrics. These metrics need to be derived from our model in order to create a differentiable mapping from control variables to costs and constraints. We base the cost on the mean number of requests present in each replica, and have a single constraint on a response time percentile.

Mean requests in replicas. As each replica is modelled by a single queue, the mean request present at time t can be approximated from the fluid model (8.1) by summing over all the phase states present in that queue.

$$x_q(t \mid \mathcal{P}) = \sum_{i \in \mathcal{S}_q} x_i(t \mid \mathcal{P}) \qquad q \in \mathcal{Q} \tag{8.3}$$

Further, let $\boldsymbol{x}_Q(t) \in \mathbb{R}_+^{|\mathcal{Q}|\times 1}$ be the vector of all modelled mean requests populations in each queue q in the network.

Response time percentiles. As shown in [Ruuskanen et al., 2021a], it is possible to obtain an approximation of the cumulative distribution function (CDF) of the response time given our fluid model. Let $\boldsymbol{\pi}(t) \in \mathbb{R}_+^{|\mathcal{S}|\times 1}$ be the probability vector of finding a request in the corresponding phase state at time t, and $\beta \in \mathbb{R}_+^{|\mathcal{C}|\times 1}$ the probability vector of the request entering the corresponding class at $t = 0$. The probability of remaining in the queuing network at time t can then be approximated with the following ODE,

$$\frac{d\boldsymbol{\pi}}{dt} = \boldsymbol{W}^T D^{g(\boldsymbol{x}^*,\boldsymbol{z})}\boldsymbol{\pi}(t), \qquad \boldsymbol{\pi}(0) = \boldsymbol{A}\beta \tag{8.4}$$

where $D^{g(\boldsymbol{x}^*,\boldsymbol{z})} \in \mathbb{R}_+^{|\mathcal{S}|\times|\mathcal{S}|}$ is a diagonal matrix with elements $D_{ii}^{g(\boldsymbol{x}^*,\boldsymbol{z})} = g_{Q(i)}(\boldsymbol{x}^*, \boldsymbol{z})$, i.e., (8.2) evaluated at the stationary solution $\boldsymbol{x}^*$ given some $\boldsymbol{z}$. As (8.4) is a linear system, it has a closed form solution. An approximation of the percentile φ_α can then be obtained by either bisection search over this closed-form solution, or by evaluating (8.4) and finding the t such that $\sum \boldsymbol{\pi}\,(t = \varphi_\alpha) = 1 - \alpha$. As the percentile and its approximation depends on $\boldsymbol{x}^*$, they also depend on the choice of $\mathcal{P}$ which we denote as $\varphi_\alpha\,(\mathcal{P})$.

8.2 Routing Optimization using Automatic Differentiation

The fluid model allows us to pose an idealized *optimal control* version of the original cost minimization problem, assuming a set of load balancing probability trajectories $\mathcal{P}(t)$ and some cost function $J_o(\cdot)$, as follows

$$\begin{aligned} \min_{\mathcal{P}(t)} \quad & \int_0 \infty J_o\,(t, \boldsymbol{x}_Q(t), \mathcal{P}(t))\,\mathrm{d}t \\ \text{subject to } & d\boldsymbol{x}/dt = (8.1) \\ & \sum \boldsymbol{p}(t) = 1 \;\forall t, \boldsymbol{p} \in \mathcal{P} \\ & \varphi_\alpha\,(\mathcal{P}(t)) \leq \varphi_{\lim} \;\forall t \end{aligned} \tag{8.5}$$

By minimizing over all $\mathcal{P}(t)$, we try to directly find the optimal load distribution that over time minimizes the selected cost. Although possible, the model will become less accurate the further $\mathcal{P}(t)$ is from what it was when recording the data used to fit the model, and thus the predicted optimum is also less accurate. Further, as cloud systems are inherently dynamic due to resource contentions, workload changes, migrations, or even malfunctions, the optimal $\mathcal{P}(t)$ will generally not be convergent as the system is subjected to both slow and abrupt changes over time. This necessitates an optimization scheme where we perform simultaneous online optimization and model

tuning in order to both update $\mathcal{P}(t)$ in a robust manner and adapt to changes in the system. To create such an algorithm, some adjustments to (8.5) are first needed.

Iterative model refitting & optimization. Using the model to predict the system behavior far from the operating point it was fit to will likely result in less accurate predictions. Less accurate predictions can lead to the system being driven into regions of high cost or constraint violations.

This can be remedied by continuously updating the model. But due to the fast timescale of the system dynamics compared to the time needed to gather enough data for an accurate model fitting using the scheme from [Ruuskanen and Cervin, 2022], robust online model tracking at the necessary speed becomes a non-trivial problem. Instead, one possible simple solution is to update $\mathcal{P}$ in discrete steps, where between each step the system is monitored in order to gather enough data to refit the model before deciding the next $\mathcal{P}_k$. By bounding how far $\mathcal{P}_k$ can move from $\mathcal{P}_{k-1}$, it is then possible to assure that the system moves within some vicinity of the current operating condition where the model is accurate, increasing robustness against accidentally violating the constraints. We denote step k as the period between t_{k-1} and t_k, the sample time as $h = t_k - t_{k-1}$, the data gathered during step k as $\mathcal{D}_k$, and the model re-fitted on $\mathcal{D}_k$ as $\dot{x} = F_k(\boldsymbol{x})$, where $F_k(\boldsymbol{x}) = \boldsymbol{W}^T\boldsymbol{\theta}(\boldsymbol{x}, \boldsymbol{z}) + \boldsymbol{A}\boldsymbol{\lambda}$ according to (8.1).

This iterative scheme will result in a slower control action, and thus a potentially slower convergence of the cost function minimization, than fully relying on the model to decide some trajectory $\mathcal{P}(t)$ in a single step. In fact, the system will reach a stationary operating condition before deciding the next $\mathcal{P}_k$ as this is required to reliably re-fit the model. But as the overall goal is to minimize cost of a running system over a potentially very long time horizon, this slowdown is acceptable as long as the system does not change too quickly.

Optimizing over weights. Optimizing directly over the probabilities becomes cumbersome, as we need to adhere to $\sum \boldsymbol{p} = 1 \ \forall \boldsymbol{p} \in \mathcal{P}_k$. Instead, it is possible to optimize over weight vectors $\boldsymbol{w}$ which can be converted to categorical distributions via the *softmax* function $S(\boldsymbol{w})$ [Goodfellow et al., 2016]. The softmax function maps vectors defined on $\mathbb{R}^n$, to vectors whose elements are allowed to take values in the interval $[0, 1]$ and that fulfills $\sum_i S_i(\boldsymbol{w}) = 1$. The i'th element of $S(\boldsymbol{w})$ is defined as

$$S_i(\boldsymbol{w}) = \frac{\exp(w_i)}{\sum_j \exp(w_j)}. \tag{8.6}$$

Further, the softmax function preserves the order of the vector element quantities, i.e., if $w_i \geq w_j$, then $S_i(\boldsymbol{w}) \geq S_j(\boldsymbol{w})$. We introduce the notation

$\mathcal{P}(\mathcal{W}_k)$ to convert the set of all weight vectors to a set of categorical distributions for load balancing, where $\mathcal{W}_k = \{\boldsymbol{w}_1, \boldsymbol{w}_2, \dots\}$ is the set of all weight vectors at time step $k+1$.

$$\mathcal{P}(\mathcal{W}_k) := \{S(\boldsymbol{w}) \ \forall \boldsymbol{w} \in \mathcal{W}_k\} \tag{8.7}$$

The set $\mathcal{W}_k$ will be decided based on data $\mathcal{D}_k$ that is gathered during step k, i.e., between $\{t_{k-1}, t_k\}$, using $\mathcal{P}(\mathcal{W}_{k-1})$ for load balancing.

Limiting the step size. A natural way of managing the step sizes would be to introduce some cost on the difference between $\mathcal{W}_k$ and $\mathcal{W}_{k-1}$. However, certain disturbances such as an increase in the load would increase the overall cost of the system, and thus change the relative step sizes if care is not taken. Instead, we will manage the step size limits by introducing the following constraint on the 2-norm on the change in probability over all weight vectors

$$\sqrt{\sum_{\boldsymbol{w} \in \mathcal{W}_{k-1}} ||S(\boldsymbol{w}) - S(\boldsymbol{w}^+)||_2^2} \leq d\mathcal{W}_{\text{lim}} \tag{8.8}$$

where $\boldsymbol{w}^+$ is the corresponding updated $\boldsymbol{w}$ in $\mathcal{W}_k$, and $d\mathcal{W}_{\text{lim}}$ is the maximum allowed step size.

Reworking the percentile constraint. Due to the dynamic nature of cloud systems, we cannot guarantee that any constraints based on performance metrics can actually be fulfilled at any time step. A disturbance might arise that pushes the system to an operating region where a constraint is violated. A constraint violation can also occur after a parameter update, as neither the model nor the robust stepping scheme is perfect.

The optimization algorithm must thus be able to handle such cases, and quickly drive the system back to a viable operating region. To do this, we can substitute the constraint by penalizing the cost function in the case of violation by, e.g., an additive cost function term $J_\varphi(\mathcal{P}(\mathcal{W}_k))$ using a *penalty function*. The important thing is that the penalization should be negligible as long as the constraint is not violated, and quickly grow as we approach and cross the constraint boundary. This will ensure that the gradient of the cost function points the parameters towards viable operating regions.

New cost function. At time step k, the refitted model $F_k(\boldsymbol{x})$ can then be used to determine the next $\mathcal{W}_k$ using the following new cost function

$$J_k(\mathcal{W}) = \int_0^{t_f} J_q(t, \boldsymbol{x}_Q(t_k + t), \mathcal{P}(\mathcal{W}))\, \mathrm{d}t + J_\varphi(\mathcal{P}(\mathcal{W})) \tag{8.9}$$

subjected to $\dot{\boldsymbol{x}} = F_k(\boldsymbol{x})$ where $\boldsymbol{x}(t_k)$ is set to the stationary solution of $\boldsymbol{x}(t \mid \mathcal{P}(\mathcal{W}_{k-1}))$, and the step size constraint (8.8). As we obtain transient

values from the fluid model, we can minimize over these given some arbitrary cost function $J_q(\cdot)$ from current time t_k over some time horizon t_f. In general, as the system will reach a stationary state before the next action is taken, we should let t_f be large enough to capture the stationary values of $\boldsymbol{x}_Q(t)$.

Further, the cost function only takes the next $\mathcal{W}_k$ into consideration, even if t_f would span multiple steps. This is done for simplicity, and since we are simply unsure how the model behaves when leaving the vicinity of $\mathcal{W}_{k-1}$. A prediction horizon could potentially be added together with a decreasing trust the further from $\mathcal{W}_{k-1}$ we move, to create an MPC-like problem formulation similar to [Incerto et al., 2017; Incerto et al., 2018]. But the resulting optimization problem would be problematic, as it would become quite intricate with no guarantees on convexity.

Cost-Optimizing Controller

In each step, we will use the cost function (8.9) to decide the next $\mathcal{W}_k$. As we consider no prediction horizon, and since the optimization problem is not convex with potentially multiple local minima, we will not try to optimize (8.9) until convergence in each time step. This would become unnecessarily costly, and only result in generating an optimal $\mathcal{W}_k$ given the step length constraint with no guarantees that it would actually move the system towards its global minimum.

Instead, a more direct approach is suitable. For demonstrative purposes, we use a simple single gradient step to decide $\mathcal{W}_k$, based on the gradient of (8.9). Using automatic differentiation, this gradient ∇J_k can be directly obtained, despite the dependence on the two ODEs (8.1) and (8.4). Together with some constant scaling factor α and a step size limiter c, we can then calculate the next $\mathcal{W}_k$ with the following gradient step update

$$\boldsymbol{w}^+ = \boldsymbol{w} - c\alpha\nabla_{\boldsymbol{w}} J_k\left(\mathcal{W}_{k-1}\right) \qquad \forall \boldsymbol{w} \in \mathcal{W}_{k-1} \tag{8.10}$$

where $\nabla_{\boldsymbol{w}}$ is the gradient with respect to the elements in $\boldsymbol{w}$, and $\boldsymbol{w}^+$ is the corresponding weight vector in $\mathcal{W}_k$. If (8.8) is not violated, then $c = 1$, otherwise it can be obtained as

$$c = \underset{c\in[0,1]}{\arg\min} \sqrt{\sum_{\boldsymbol{w}\in\mathcal{W}_{k-1}} \left\| S(\boldsymbol{w}) - S\left(\boldsymbol{w}^+ \mid c\right)\right\|_2^2} = d\mathcal{W}_{\lim} \tag{8.11}$$

using, e.g., bisection search.

The complete algorithm can thus be seen as an adaptable *gradient descent* scheme, where we after each step re-evaluate the model before calculating the gradient and deciding the next step to take. The resulting algorithm is summarized by the block diagram in Figure 8.4, and the pseudocode in Algorithm 1.

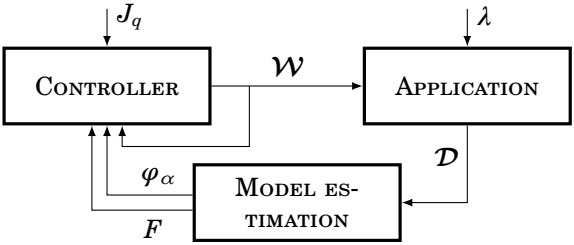

Figure 8.4 The controller sets some weights $\mathcal{W}$ for the application to run with. Data $\mathcal{D}$ is collected and used to fit model F as well as estimate the response time φ_α. These are then used by the controller to simulate the environment and update the old weights based on the gradient of the cost on the simulated state. Disturbances can act on both the controller, in the form of, e.g., changes in the cost J_q, and on the application, in the form of, e.g., changing workload λ.

8.3 Experimental Evaluation

To test and showcase the online optimization algorithm, two experiments were performed on a small distributed microservice application.

The application, alongside the implementation of our online optimization algorithm, is publicly available on GitHub[1].

Experimental Setup

A similar setup to the one studied in [Ruuskanen and Cervin, 2022] was considered, consisting of a simple microservice application deployed on a testbed of multiple Kubernetes clusters.

Kubernetes testbed. The federated application sandbox described in [Ruuskanen et al., 2021b] was used as a testbed for the application. The sandbox consists of clusters of virtual machines deployed on OpenStack [*OpenStack* 2023] in the Ericsson Research datacenter. To each cluster, 4 virtual machines, each with 4 virtual CPUs and 4 GB of RAM, were assigned. All virtual machines in each cluster are further connected via a cluster-specific isolated network. These networks are then connected to each other via a gateway, which enables network characteristics between clusters to be easily emulated using Tc-Netem [*Tc-Netem* 2023]. On each cluster, Kubernetes is then deployed along with the service mesh Istio [*Istio* 2023] to handle the application and to emulate a realistic cluster software stack. Istio further allows for easy extraction of the required tracing data for model fitting, and handling of cluster-to-cluster communication between microservices.

[1] https://github.com/JohanRuuskanen/ACSOS2022_code

Algorithm 1 Cost optimization procedure.

Algorithm 1a Control loop running N iterations of data collection and parameter updates. Data collection is run for a duration h.

Initialize $\mathcal{W}_0$
for $k \leftarrow 1$ to N **do**
  Set $\mathcal{P}(\mathcal{W}_{k-1})$ as load balancing strategy
  $\mathcal{D}_k \leftarrow \text{COLLECT\_DATA}(h)$
  $F_k, \varphi_{.95} \leftarrow \text{FIT}(\mathcal{D}_k)$
  $\mathcal{W}_k \leftarrow \text{UPDATE}(\mathcal{W}_{k-1}, F_k, \varphi_{.95})$
end for

Algorithm 1b Controller calculating $\mathcal{W}_k$ based on $\mathcal{W}_{k-1}$ and data $\mathcal{D}_k$.

function UPDATE($\mathcal{W}_{k-1}, F_k, \varphi_{.95}$)
  $\mathcal{W}_k \leftarrow \emptyset$
  for $\boldsymbol{w} \in \mathcal{W}_{k-1}$ **do**
    $\nabla_{\boldsymbol{w}} J_k \leftarrow \text{GRADIENT}(J_k(\boldsymbol{w} \mid F_k, \varphi_{.95}), \boldsymbol{w})$
    $\boldsymbol{w}^+ \leftarrow \text{LIMITED\_STEP}(\boldsymbol{w}, \nabla_{\boldsymbol{w}} J_k)$
    $\mathcal{W}_k \leftarrow \mathcal{W}_k \cup \{\boldsymbol{w}^+\}$
  end for
  return $\mathcal{W}_k$
end function

Application. A simplified version of the facedetection-as-a-service application shown in [Ruuskanen and Cervin, 2022] was used as an example application. It consists of two services, a *frontend* implementing a user interface and image preprocessing, and a *backend* implementing a face-detection algorithm. Both services are implemented in Python using Flask [*Flask* 2023] and Gunicorn [*Gunicorn* 2023]. We assume a structure as in Figure 8.5 where the user-facing frontend only exists on the edge close to its users, and the backend is distributed across multiple sites, emulated by our different clusters. Each frontend-to-backend connection is associated with a delay d_i, and each backend is also associated with a computation cost C_i. We will assume that the higher costs are associated with lower delays, in order to create a trade-off between cost and latency. Such scenarios could occur in, e.g., edge computing, where low-latency computations can be performed on the device or at geographically close edge datacenters, but at an increased cost, while off-loading computations to larger sites is cheaper but subjected to longer communication delays.

Cost-optimizing controller. Using the load generator from [Ruuskanen and Cervin, 2022], images are fed to the application as Poisson arrivals with

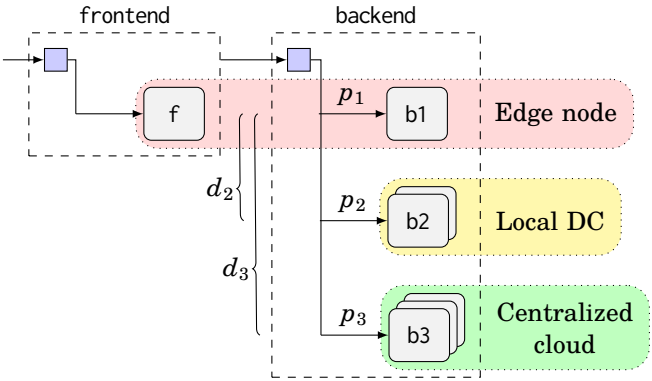

Figure 8.5 The incoming load passes through the frontend and is balanced over the replicas `bi` in the backend based on the probabilities p_i. The delay between the frontend and the replicas in the backend are d_i on average. The backends with lower delays are assumed to be more expensive to run, so a small edge node will be closer to the user, but will likely have higher relative costs. This creates an optimization problem where delay and cost has to be balanced.

rate λ. At every time step k, tracing data from Istio is collected to generate $\mathcal{D}_k$. The model F_k is fit to $\mathcal{D}_k$, after which the cost function $J_k(\mathcal{W})$ can be calculated. We do this using the Julia programming language, allowing us to take advantage of packages such as DifferentialEquations.jl [Rackauckas and Nie, 2017] for simulating the model, and ForwardDiff.jl [Revels et al., 2016] for automatic differentiation. With this the gradient ∇J_k of the cost function, as well as the next $\mathcal{W}_k$ can be effortlessly calculated.

Two Backends — Fixed Steps in an Offline Experiment

In a first experiment, we consider the example application with only two replica sets of the backend: `b1` that is deployed on the same cluster as the frontend, and `b2` that is deployed on a different cluster. All connections between the two clusters are given a Pareto-distributed additive delay with a 25ms mean, 5ms *jitter* (a Tc-Netem term roughly equating standard deviation) and 25% correlation between samples. Hence, requests load balanced to `b2` will experience an additive delay.

The probability constraint enables us to determine the load balancing probabilities directly by using a single parameter p_1, and then determining $p_2 = 1 - p_1$, removing the need for the weights and softmax function. This makes it feasible to collect data in a grid over $p_1 \in [0, 1]$, allowing us to run the agent offline against the recorded data with a fixed size step according to the grid that is in the direction of the gradient.

For the cost function, we let it be conditional on φ_α violating φ_{lim} for selecting either J_φ or J_q. J_φ is simply a scaled estimate of φ_α at stationarity, whereas J_q is 0 everywhere except at time t_f where it is a linear function of the state. We choose to look at the stationary values to make it easier to compare data.

$$J_k(p_1) = \begin{cases} C_\varphi \hat{\varphi}_\alpha(p_1) & \text{if } \varphi_\alpha > \varphi_{\text{lim}} \\ \boldsymbol{C}^T \boldsymbol{x}_Q(t_k + t_f \mid p_1) & \text{otherwise} \end{cases} \tag{8.12}$$

We use recorded data φ_α to decide if the constraint is broken. The actual cost is then implemented using a prediction from the model, $\hat{\varphi}_\alpha$, so it can be differentiated. We let $\alpha = 0.95$ to consider the 95th percentile of response times to the application and set $\varphi_{\text{lim}} = 0.55$ ms to create a scenario where it can be too slow under bad load balancing. $\boldsymbol{C}$ is set to zero for all queues except those related to either backend, where it is $\boldsymbol{C}_b = [3, 1]$ for the first and second backend respectively. While C_φ does not affect the gradient, since the loss is conditional and the step is fixed, we set $C_\varphi = 8$ to scale the two losses to be of similar magnitude for the visualization. For the model, we give each class in the queuing network 3 phase states, for a total size of $|\boldsymbol{\mathcal{S}}| = 24$.

We record data for the offline experiment by creating a grid over p_1 with steps of 0.05 between 0 and 1. For each value of p_1, data is recorded for $h = 300$s using an arrival rate $\lambda = 14$. The optimization algorithm is then run against the recorded data, starting from $p_1 = 1$, stepping along the grid in the gradient direction. The results are depicted in Figure 8.6, where Figure 8.6(a) shows the total mean requests present in the application, Figure 8.6(b) shows the response time percentiles, and Figure 8.6(c) shows the cost based on (8.12). The dashed red line shows the model prediction, while the blue (and green) lines represent values from recorded data. The cost in Figure 8.6(c) is conditional on (8.12), where the recorded values are plotted for both individual costs, while the model prediction is only plotted for the combined function. The apparent noisiness can be attributed to the noisy data used for refitting the model.

As can be seen, the algorithm manages to step in the direction of a decreasing cost and ultimately find the minimum. Starting at $p_1 = 1$, the system is passing all load to `b1` resulting in $\varphi_{.95} > \varphi_{lim}$ and the cost is then based on the response time curve. Moving in the direction of $-d\varphi_{.95}/dp_1$ the response time as well as the total queue length is decreasing since we offload `b1` by routing some load to `b2` instead. Crossing the threshold φ_{lim}, the cost switch to J_q. Even though we see that the total queue length starts growing as p_1 goes below 0.65, the cost for `b1` is higher than `b2` and thus the queue based cost is still decreasing with decreasing p_1. Before reaching $p_1 = 0$ though, we cross the φ_{lim} threshold again, finding the optimal value

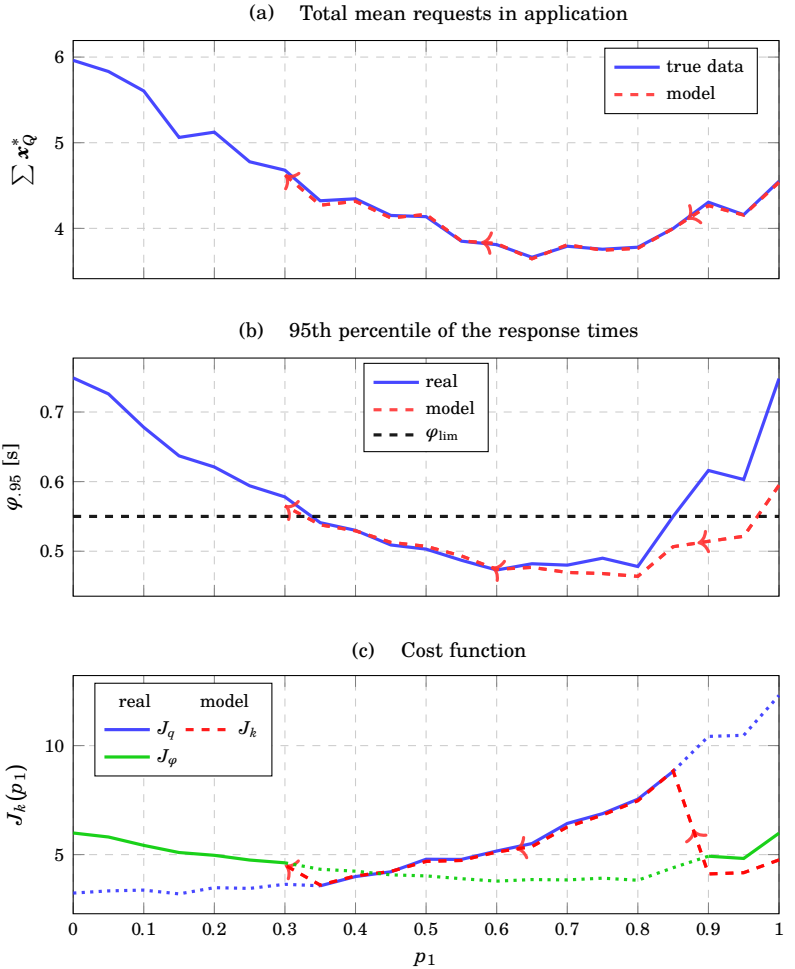

Figure 8.6 Results from the offline experiment with two backends. The values are plotted over the probability p_1. In Figures 8.6(a) and 8.6(b) the blue lines show the value from recorded data $\mathcal{D}$, while the red dashed line shows the corresponding prediction from the fitted model. In Figure 8.6(b) φ_{lim} shows when the cost (8.12) switches mode. The cost in turn can be seen in Figure 8.6(c), where the blue and green lines correspond to the queue cost respective to the cost on overstepping the response time threshold. The lines are filled where each cost is active, and dotted where they are inactive. The red dashed line shows the model's estimate of the full cost from both parts of the cost function.

of p_1 to be around 0.35.

Three Backends — Online Optimization Experiment

For the second experiment, we run everything live on the real application. We consider the same setup as in the first experiment, but also introduce a third backend replica (b3) deployed on a third cluster. All connections between the first and third clusters are given a Pareto distributed additive delay with a 50ms mean, 10ms jitter and 25% correlation between samples. As we now have more than one parameter to optimize over, we will resort to using the weight vectors in $\mathcal{W}$ as described in (8.7).

For the cost function, we again let J_q be a linear function of state at t_f and 0 for other times, to only consider stationary values and simplifying comparisons. However, the response time constraint is included by setting J_φ as an exponential penalty function based on the difference between φ_α and φ_{lim}. Again, we base the activation of the constraint on recorded data, as we would like it to be active when the real φ_α is near or above its limit. The cost itself is based on the predicted $\hat{\varphi}_\alpha$ to make it differentiable. The resulting cost function becomes

$$J_k(\mathcal{W}) = \boldsymbol{C}^T \boldsymbol{x}_Q\left(t_k + t_f \mid S\left(\mathcal{W}\right)\right) + C_\varphi e^{\mu(\varphi_{.95} - \varphi_{\text{lim}})} \hat{\varphi}_{.95}\left(S\left(\mathcal{W}\right)\right) \tag{8.13}$$

where we set $\alpha = 0.95$, $\varphi_{\text{lim}} = 0.6$ms, $\mu = 10$, $C_\varphi = 5$, $t_f = 5$ms and $\boldsymbol{C}$ to a zero vector except for the backend replicas where it is set to $\boldsymbol{C}_b = [3, 2, 1]$. Further, the gradient step is given $\alpha = 0.5$ and $d\mathcal{W}_{\text{lim}} = 0.15$. For the model, we give each class in the queuing network 3 phase states, for a total size of $|\mathcal{S}| = 33$.

The system is loaded with Poisson arrivals at $\lambda = 15$ for a total of 40 time steps, with the initial weight vector is set to $\boldsymbol{w}_0 = [2, 0, 0]$, giving $\boldsymbol{p}_0 \approx [0.78, 0.11, 0.11]$. Each time step is given a duration of $h = 300$s. We further let the running system be influenced by two disturbances. The first disturbance is introduced at time step 14, where the arrival rate is suddenly increase by 50% to $\lambda = 22.5$. The second disturbance is introduced at time step 27, where the costs to the backend replicas are changed to $\boldsymbol{C}_b = [1, 2, 3]$.

The results from this experiment can be seen in Figure 8.7. The blue shaded area shows where the first disturbance is active, and the red area shows where both the first and second disturbances are active. Figure 8.7(a) shows the total mean requests present in the application, Figure 8.7(b) shows the response time percentiles, and Figure 8.7(c) shows the cost based on (8.13). Both values from data (blue) and the corresponding fitted model (dashed red) are shown. Finally, Figure 8.7(d) shows the three load balancing probabilities from the frontend to b1 (p_1, blue), to b2 (p_2, dashed red) and to b3 (p_3, dotted green).

As can be seen, the online optimization algorithm manages to drive the system towards a load balancing setting of less cost and counteract the

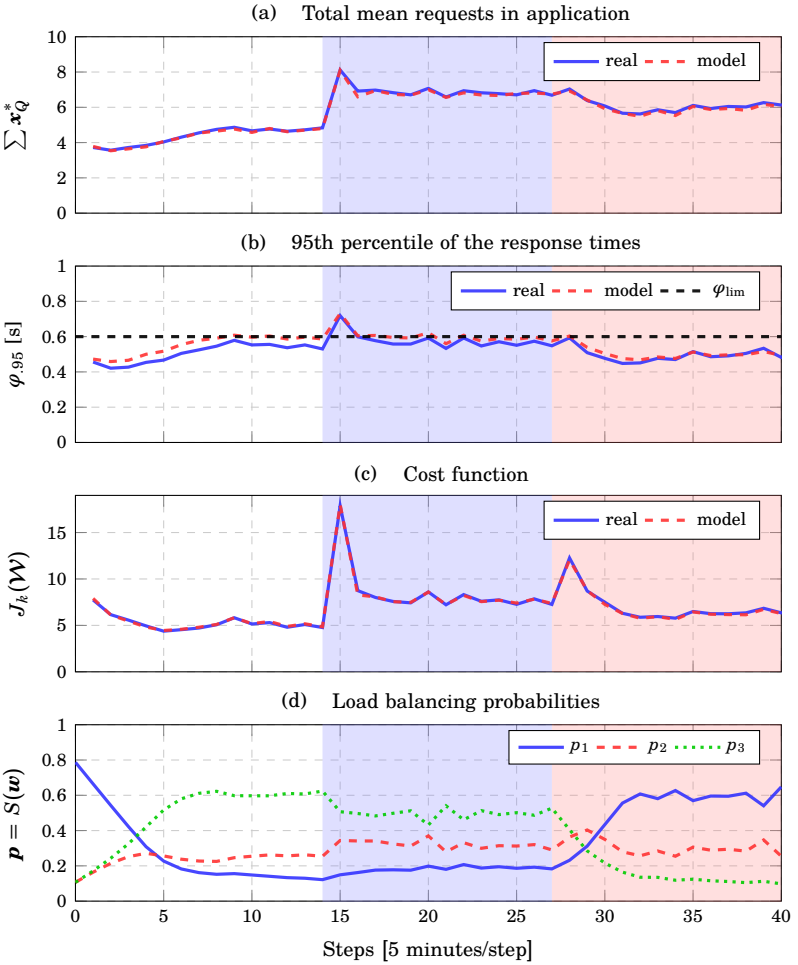

Figure 8.7 Results from the online experiment with three backends. The values are plotted over the simulated time steps, each being one sampling period of 300 seconds. In Figures 8.7(a) to 8.7(c) the blue lines shows the value from recorded data $\mathcal{D}$, while the dashed red line shows the corresponding value from the fitted model. Further, in Figure 8.7(d) the three lines corresponds to the three load balancing probabilities. Finally, the blue shaded area shows where the first disturbance is active on the arrival rate, while the shaded red area shows where the first and second disturbance is active on both arrival rate and queue length costs. A one time step lag can be seen on Figure 8.7(a), Figure 8.7(b), Figure 8.7(c) compared to Figure 8.7(d) as $\mathcal{D}_k$ is recorded using $\mathcal{W}_{k-1}$.

disturbances in quite a few steps. At a few steps, it seems as the algorithm steps upwards in cost, but this can be attributed to noise. At first, the system shifts load from b1 to b2 and b3 where costs are smaller. This decreases the cost, but also increases the total queue length and response time percentile, as b2 and b3 are associated with a higher site-to-site delay. The shift is mostly stopped when the percentile constraint is reached, but due to the simplicity of the gradient descent approach, the system experiences a slow final convergence. When the first disturbance in the form of a 50% increase in load is introduced, both the queue length and percentiles immediately increases. As the percentile constraint is now violated, the cost function spikes, which results in the gradient step aggressively moving the system back into a parameter configuration where the constraint is no longer violated. After the constraint is once again fulfilled, the cost function slowly settles to a minimum. At the activation of the second disturbance, in the form of shifting the costs of b1 and b3, the cost function increases while the queue length and percentile stays the same. The system then quickly shifts the load probabilities and reduce the cost until the effects of the penalty function becomes too large, and it settles to a slow final convergence. As can be seen in Figure 8.7(b), there are at times rather large gaps remaining between the percentile and its limit. This has to do with the values assigned to the penalty function. Increasing C_φ and μ would lead to more aggressive handling of violations and a tighter gap, but they were kept fairly low to yield a more presentable cost function.

8.4 Summary and Discussion

In this chapter, we have demonstrated how automatic differentiation can be used to optimize the load balancing parameters of a distributed microservice application. This is done by deriving an online optimization scheme to minimize some holistic cost by tuning the probabilities of random load balancers between replica sets. Although not guaranteed to find the global cost minimum, the algorithm is shown to reduce cost while adhering to constraints on the response time percentile in an experimental evaluation. The assumed microservice fluid model is fairly general, and the cost function can be arbitrarily defined as long as it depends only on metrics that the model can approximate. This enables our online optimization scheme to be quickly adapted for a multitude of different load balancing scenarios.

Automatic differentiation allows us to differentiate functions whose derivatives would be too difficult to explicitly derive. In our case, it is used to differentiate through an arbitrary cost function L based on the solution of two dependent ODEs from our fluid model. Other models, e.g., the common mean-value analysis, could be used as well, but we chose the fluid model

due to its transient values, validity for non-product-form networks and as it is quick to evaluate and differentiate using the Julia ecosystem. In our experiments, the model fitting and parameter update in Algorithm 1 takes about 10s, where the majority of the time goes to fitting the phase-type distributions using the expectation maximization algorithm, which can be made quicker by, e.g., moment matching. The differentiation step itself is quick, in the range of 100ms.

Further, the generality of automatic differentiation allows us to consider a whole range of other ways of optimizing an application, e.g. scaling and migration, as long as such an action can be represented in the used model. Also, given an expression for the gradient and higher order partial derivatives, more advanced optimization methods can be used to decide how to update the control variables. In this chapter we however chose to focus on optimizing load balancing using simple gradient stepping, to exemplify the procedure with a problem whose fluid model representation is simple, a comprehensible control variable update and a relatively easy experimental validation.

The approach however has some drawbacks. First, we provide no convergence guarantees as the performance is inherently dependent on the accuracy of the model. Identifying when a model is performing poorly is thus important to avoid driving the system into regions of high cost or constraint violations. A poorly chosen model can drive the system to regions of increasing cost and even constraint violations. It is thus important to keep track of model performance, in order to identify these situations before the system is driven out of bounds. Further, the generality of the approach is not only a strength since deriving a suitable cost function can require both time and expert knowledge.

We can compare this to Chapter 7, which while they are quite different in some sense, they both consider the same problem of optimizing a microservice application. In Chapter 7, we considered a model-free approach, where we used reinforcement learning to learn a policy for scaling the application. This approach is more general in the sense that it does not require a model of the system, but it is also more complex and requires more data to learn a good policy. Here we had a much easier process of creating a decent controller, but it is also more limited in the sense that it requires a model of the system, and that the control variables are limited to the probabilities of the load balancers. The restricted control policy is not an inherent limitation of the approach, but rather a consequence of the model used.

The dependence on a model is both a strength and a weakness, as it makes the approach more efficient but more limited. By combining a model with learning, we could potentially get the best of both worlds. This will be explored in Chapter 9, where we use the fluid model as a base with a

learned correction term to improve the accuracy of the model.

Restricting ourselves to selecting static load balancing parameters was both since the fluid model assumes the probabilistic routing, but also since it is interpretable, making for a controller which we can understand. By updating it often enough, it should provide both good performance and a good understanding of the system. The automatic differentiation approach is not limited to static load balancing parameters, and with a more flexible model we could potentially optimize over more complex policies. An NN could provide a more advanced policy able to update the load balancing parameters dynamically, but it would also bring in all the other problems of NN-based policies such as sample efficiency, interpretability and stability. While we did not explore that path in this chapter, it is a natural extension of the work presented here.

9

Improving Microservice Models

When modelling a system we want to capture the dynamics of the system as accurately as possible. Using hand-crafted models, we can often capture some dynamics of the system quite well, but it can be difficult to capture all the dynamics of the system. It typically creates models that generalize well since they are based on the underlying dynamics of the system, and are often easy to interpret. Using machine learning to fit a model, we have less control over what dynamics are captured, but it is possible to capture dynamics without fully understanding them. This creates models that are harder to interpret, and that might not generalize well outside the seen data, but can capture complex dynamics that are difficult to model by hand. By combining the two, taking known scientific models and embedding machine learning into them, we can have the flexibility from learning while not disregarding existing knowledge.

In this chapter we further explore the benefits of general automatic differentiation, and how we can use it to improve on the fluid model presented in Chapter 8. While the fluid model does a good job at representing the underlying system for the system parameters it was trained on, it does have a tendency to diverge from the true system dynamics as the system parameters change. This means that control parameter optimization, as done in Chapter 8, can be problematic if not retraining the model often enough. Using the same fluid model as a base, Section 9.1 show how to extend it by adding an NN that captures the dynamics related to the load balancing parameters that the fluid model is missing. In Section 9.2 we additionally discuss the how to impose bias on the network based on features of the model. These two approaches are then evaluated to see how they affect data efficiency and final performance in Section 9.3.

Data-Driven Discovery of Differential Equations

The success of deep learning is often accredited to a few different things. Top among those are the generality of neural networks (NNs) as efficient function approximators, as well as large amounts of data for training and the resources to run large scale training. Though, in addition to this, many of the large breakthroughs in deep learning community come from new ways of imposing inductive bias onto the model. One example being convolutional neural networks for image classification, with the idea that pixels close to each other were more likely to be related, or recurrent networks for time-series where the networks propagate information in one direction through time. Another idea is to impose the structure of an ODE when looking for solutions to systems that should behave as one, i.e., letting the NN model the dynamics of the system, and extracting the solution of the system using ODE solvers.

Using NNs to identify dynamics within an ODE has been done since the early 1990s, e.g. [Rico-Martínez et al., 1992; González-García et al., 1998; Long et al., 2018], as well as other function approximators such as Gaussian processes [Schober et al., 2014; Raissi and Karniadakis, 2018; Heinonen et al., 2018] or sparse regression methods [Brunton et al., 2016; Rudy et al., 2017].

All of this can be unified under the term Universal Differential Equations (UDEs), where the dynamics of the DEs are modelled using universal function approximators of some form, allowing them to capture arbitrary dynamics.

$$\frac{d\boldsymbol{x}}{dt} = f(\boldsymbol{x}, t, \boldsymbol{\eta}) \tag{9.1}$$

Here we assume some function f parameterized by $\boldsymbol{\eta}$ defining the dynamics of the system, though we can easily say that $f(\boldsymbol{x}, t, \boldsymbol{\eta}) = F(\boldsymbol{x}) + NN_{\boldsymbol{\eta}}(\boldsymbol{x})$, to both introduce a good bias with the known dynamics in F, and capture whatever is missing with $NN_{\boldsymbol{\eta}}$.

Recent interest in combining ODEs and NNs has led to work such as [Chen et al., 2018; Hasani et al., 2020; Rackauckas et al., 2020], showing that this approach can be used to efficiently solve a wide range of problems.

Similar techniques have previously been used to model microservice applications, e.g., [Yang et al., 2019] use pure NN models to capture the discrete dynamics of a microservice mesh and predict future states to optimize resource allocations.

In this chapter we attempt to keep the knowledge from an existing model, and only add a tiny NN modelling the missing dynamics. We specifically want to keep the existing microservice model from [Ruuskanen and Cervin, 2022], where the original model is good in most scenarios. There are however cases where it can have drastically reduced accuracy, for example when the

system parameter differs from where the model was fitted. This could be approached by training multiple versions of the original model, one model per parameter value, but as the system increase in size this can quickly become infeasible. Instead, we use a single base model extended by an NN, where the NN is trained to model how the dynamics change from these parameters by training on data collected from a small amount of the parameter space.

9.1 Extending a Fluid Model with Neural Networks

The fluid model is extracted from the system in the same way as in Chapter 8. The extracted model is then used to simulate the load in the system using (8.3), and to estimate the expected 95th percentile response time using (8.4). To predict outside the current parameter settings, the $\boldsymbol{P}$ matrix can be updated with desired load balancing values, and $\boldsymbol{\lambda}$ can be changed to reflect expected incoming load. And though we will not try to predict incoming load, and rather just use the most recent data as a good estimate when predicting, for verifying the models we will use the actual $\boldsymbol{\lambda}$ from the recorded data.

This updated model can now be simulated to generate expected loads and response time percentiles for a different parameter setting. Though the model does generalize quite well around the parameter setting where it was fitted, it will have an increased tendency for prediction errors as we move further away from the parameters used for the data collection.

To decrease these prediction errors, we add two parts to the model. An NN modelling the missing dynamics in the class population from the fluid model, and an NN modelling the error of the response time prediction.

Modelling the Missing Dynamics from the Fluid Model

We set up the extended model as an ordinary differential equation with two separate terms.

$$\frac{d\boldsymbol{x}}{dt} = F(\boldsymbol{x}, \boldsymbol{\lambda}, p) + NN_{\boldsymbol{\eta}}(\boldsymbol{x}, \boldsymbol{\lambda}, p) \tag{9.2}$$

The first term is given by the fluid model dynamics from (8.1),

$$F(\boldsymbol{x}, \boldsymbol{\lambda}, p) = \boldsymbol{W}^T \boldsymbol{\theta}(\boldsymbol{x}) + \boldsymbol{A}\boldsymbol{\lambda}, \tag{9.3}$$

while the second term is a universal function approximator, in this case an NN parameterized by $\boldsymbol{\eta}$. Both terms depend on the same inputs, the phase states $\boldsymbol{x}$, the arrival rate $\boldsymbol{\lambda}$ and the load balancing parameter p. We define a loss function, over some data $\mathcal{D}$, using the mean squared error between the

predicted steady-state class population $\hat{\boldsymbol{x}}_{\boldsymbol{C}}$ and the recorded average class population $\boldsymbol{x}_C^*$.

$$J_C(\boldsymbol{\eta}) = \frac{1}{|\mathcal{D}|} \sum_{p,\boldsymbol{\lambda},\hat{\boldsymbol{x}}_c \in \mathcal{D}} (\boldsymbol{x}_C^*(\boldsymbol{\lambda}, p) - \hat{\boldsymbol{x}}_C)^2 \tag{9.4}$$

The recorded class population $\boldsymbol{x}_C^*(\boldsymbol{\lambda}, p)$ is estimated similarly to the queue population in (8.3).

The steady state population $\hat{\boldsymbol{x}}_{\boldsymbol{C}}$ is estimated by simulating the ODE for some time t_f that is large enough that the system normally will have reached steady state, for this system $t_f = 2$ s was found to be sufficient. We use $\boldsymbol{\lambda}$ from recorded data when training, since this will allow a better training fit and will likely create a more accurate model since there will be less noise affecting the training loss.

When we instead use our models to estimate future behavior for different load balancing parameters, the actual $\boldsymbol{\lambda}$ is a stochastic variable and the value is not known. We can estimate it by the most recently recorded value, as that is likely a good approximation of the future value.

Modelling Response Time Prediction Error

The improvements to the fluid model will already make the estimate of the response time percentile (8.4) better, since we now have more accurate predictions for the class populations. But the main error source for the response time percentiles is the assumption made in the fluid model that each request receives the mean processor share, as mentioned in [Ruuskanen et al., 2021a], and this still remains. This assumption tends to be less true for larger utilization, due to increased variability in queue lengths, and results in the error of the estimate being correlated with the utilization.

Having the predicted $\boldsymbol{x}^*$, the phase state population in steady state, we can estimate the response time percentiles φ_α from (8.4) and set up the error of the estimate in relation to the actual response time percentiles from data, $\hat{\varphi}_\alpha$.

$$\varphi_{err} = \hat{\varphi}_\alpha - \varphi_\alpha(\boldsymbol{x}^*(\boldsymbol{\lambda}, p))$$

Since this error depends on the solution of (9.2), we can train an NN outside the ODE to estimate the prediction error based on utilization. We use a small NN, parameterized by $\boldsymbol{\mu}$ and using L_2 regularization to avoid overfitting, and train it using the mean squared error over some data $\mathcal{D}$.

$$J_{\varphi_\alpha}(\boldsymbol{\mu}) = \frac{1}{|\mathcal{D}|} \sum_{p,\boldsymbol{\lambda},\hat{\varphi}_\alpha \in \mathcal{D}} (\varphi_{err} - NN_{\boldsymbol{\mu}}(\boldsymbol{x}_Q^*(\boldsymbol{\lambda}, p)))^2 + \alpha \sum_{\mu \in \boldsymbol{\mu}} \mu^2$$

When using the model to predict the future behavior, we add the error estimate to the response time percentile estimate, $\varphi_\alpha(\boldsymbol{x}^*(\boldsymbol{\lambda}, p)) + \varphi_{err}$, to create a more accurate estimate.

9.2 Imposing Bias on the Neural Network

As discussed in Chapter 4, the design of the neural network can have a large impact on the learning efficiency and accuracy of the model, and imposing an inductive bias on the NN can be beneficial for learning. In this section we will look at ways of imposing bias on the NN structure, and how they can be combined with the fluid model.

Dense neural network. A simple baseline to capture the missing dynamics is to use a dense NN with no extra structure or constraints, just a few hidden layers with an output of size $|\mathcal{S}|$ using linear activations. Extending the fluid model described in Section 9.1 with an NN, we can see F, the already known dynamics from the model, as a type of bias on the network. Thus, it can be interesting to see how it performs without this bias, i.e., the NN by itself. We will refer to the fluid model extend by an NN as F+NN, and also compare it to both fluid model and NN by themselves, referred to as F and NN respectively.

Flow conservation. While a dense NN should be able to fit the error, it could be beneficial to give it some guidance by imposing certain structure on the NN. For example, a structure that enforces internal flow conservation could be beneficial since the fluid model is based on this assumption.

For this purpose we introduce

$$fc(\boldsymbol{z}) = \boldsymbol{z} - \frac{1}{|\mathcal{S}|}\sum_{i=1}^{|\mathcal{S}|} z_i, \tag{9.5}$$

which translates a vector so that it sums to zero. Applying fc to the output of the NN we can ensure that the internal flows sum to zero, i.e., no requests are created or destroyed by the NN, just rerouted.

$$\frac{d\boldsymbol{x}}{dt} = F(\boldsymbol{x}, \boldsymbol{\lambda}, p) + fc\left(NN_{\boldsymbol{\eta}}(\boldsymbol{x}, \boldsymbol{\lambda}, p)\right)$$

We will refer to this method as F+NN$_{mean}$.

Flow patterns from fluid model. An even more involved approach could constrain the NN to follow the flow pattern defined by the fluid model. With this we mean that the NN should only be able to alter flows between phase-states in the fluid model, and not create new ones. This is a stronger constraint than flow conservation, and could simplify learning even more since we enforce the structure that has already been proven good for the fluid model. Though if the fluid model is not general enough, this could also limit the accuracy of the model.

To do this we look at the flow pattern that is defined by the matrix $\boldsymbol{W}^T$ in (9.3). The column k of this matrix represent how the processed load from

phase-state k affects the system. The sum of the column is zero, where the only negative value is for row k since each replica will process requests based on the current load, and remaining non-zero values are positive and signify where the requests are sent. This holds for all but the columns that have flows that exit the system, where requests are simply disappearing from the system.

Assuming that the W^T is extracted from data recorded for a load balancing parameterization where all paths are relatively well-used, we should have all possible flows represented by some non-zero value in $\boldsymbol{W}^T$. If we assign a weight $w^k_{i,j}$ for each possible combination of source i (negative value) and sink j (positive value) within a column k in $\boldsymbol{W}^T$, as well as a weight $w^k_{i,0}$ for each source i that has no corresponding sink, the set of all these weights $\mathcal{W} = \{w^k_{i,j}\}$ then contains one parameter for each possible flow in the fluid model. We can then design the NN to have a final layer of size $|\mathcal{W}|$, where each output is mapped to a weight in $\mathcal{W}$. The effect on a phase-state i of the DE is then calculated by considering all flows in $\mathcal{W}$ that are either in to or out from i, consisting of $w^k_{i,\cdot}$ flowing out and $w^k_{\cdot,i}$ flowing in. Since the output of the NN is not constrained, the weights can both increase and reduce the flows. By defining any $w^k_{i,j}$ that is not in $\mathcal{W}$ to be zero, we can define the effect on the dynamics of phase-state i to be

$$z_i = \sum_{k=1}^{|\mathcal{S}|} \sum_{j=1}^{|\mathcal{S}|} w^k_{j,i} - w^k_{i,j}$$

where $\boldsymbol{z}$ can then be added to the dynamics of the fluid model. We will refer to this method as F+NN$_{\boldsymbol{W}}$.

When running the experiments, $\boldsymbol{W}^T$ typically have a few values near zero, likely stemming from numerical errors or other noise. Assuming as above that $\boldsymbol{W}$ is extracted from a load balancing parameterization where all paths are relatively well-used, we can find a threshold ϵ such that all values in $\boldsymbol{W}^T$ that have an absolute value smaller than ϵ are set to zero.

9.3 Evaluating NN based Model Extensions

We evaluate the different methods by comparing the sample efficiency when training the NN, and the accuracy of the resulting model. We then compare the performance metric predictions between a couple of the more interesting models to evaluate how well the dynamics are actually captured, and how well this achieves the goal of predicting outside the training data. Using the improved model we also train a load balancing policy using gradient descent, showing how the improved model can converge to a value close to the optimal without needing to refit the model.

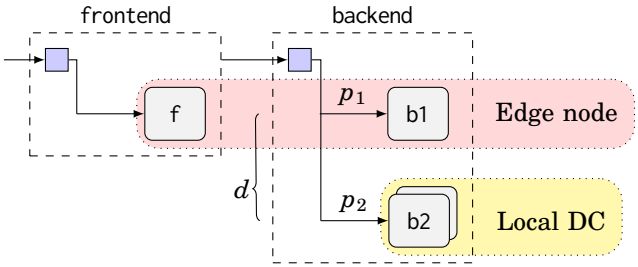

Figure 9.1 A simple distributed application with two microservices. The `frontend` only exists close to the user on the edge nodes, and the `backend` has replicas on both the edge nodes and the cloud. The backend has a weighted random load balancer with routing probabilities $p_1 = p$ and $p_2 = 1 - p$. There is a delay d between the edge and the cloud.

Data and Training

The application visualized in Figure 9.1 is the same as was used for the offline experiments in Chapter 8. We use the same dataset as was collected for those experiments, so the parameters for the applications are the same.

The class populations are extracted from 5-minute recordings of arrival and departure logs for each request, and this data is collected for all $p \in [0, 0.05, \ldots, 1]$. We use $p = 0.6$ as the *current* load balancing parameter, i.e., the data the fluid model is trained on. The NN can additionally use data collected at $p \in [0.1, 0.3, \ldots, 0.9]$, which we see as some historical data from previous experiences. This data together with data from $p = 0.6$ forms the training data $\mathcal{D}$ for the NN. The remaining data is only used to visualize how well the model captures the parameters in general. The historical data was selected in a sparse grid over the interval to have a chance of finding a good model for the missing dynamics.

While in this work, the cluster was only used for recording real data for the experiments, and all training and verification was done offline using the recorded data. This is not a restriction of the method in general, but was more convenient for experimentation. In Chapter 8 we show that the corresponding methods transfer well from offline evaluation to the live application.

The training is done using the SciML (Scientific Machine Learning) ecosystem in Julia, which has differential equation (DE) solvers that support automatic differentiation. This means that defining functions such as (9.4) which depend on the solution of a DE, and then using automatic differentiation to calculate the gradient of the loss with respect to the parameters, is possible. The NN optimization is done using the Adam algorithm, a standard first order method commonly used in deep learning. All code is publicly

available on GitHub[1].

Evaluating Model Designs

We compare the different extensions to the fluid model, presented in Section 9.2, to see how imposing bias on the NN affects the learning speed as well as the accuracy of the learned model. The objective is to capture the dynamics of the system better than the fluid model, and to be able to predict the system dynamics for a wider range of parameters than the fluid model was trained on. We have the metric defined in (9.4), which evaluates the prediction of the model on the training data $\mathcal{D}$ defined above, consisting of a range values for different load balancing parameters p. We also show the performance of the fluid model by itself for comparison, as well as a purely NN based model using the same amount of states in the UDE as for the fluid model.

For all models involving an NN, the networks are densely connected with three hidden layers of 20 units each, and exponential linear units for activation. While standard model has $|\mathcal{S}|$ outputs with linear activation from the NN, this is where the other models impose restrictions. The flow conservation model also has $|\mathcal{S}|$ outputs, but adds an additional layer subtracting the mean according to (9.5). The network using the structure of fluid model flows has $|\mathcal{W}|$ outputs, which are then used to calculate the final output as described in Section 9.2.

The different models are first evaluated on how the training loss evolves over time according to (9.4), and is shown in Figure 9.2. Each loss curve is an average over 5 trials with different seeds, though the 5 seeds are the same for all models.

The noisy training is likely due to the gradient flowing through the solution of the ODE, where a small change of the parameters might have a large effect on the solution, but the size of the gradient steps are based on the parameter space and not the solution space.

The NN model had some problems learning the dynamics and would sometimes produce an unstable system, sometimes crash the ODE solver, or not finding a good relationship between the dynamics and the load balancing parameter p, as can be seen in Figure 9.3. If trained for even longer and with larger networks, the NN model did improve the fit, though showing a larger tendency to overfit to data instead of capturing the dynamics.

Turning to the models that extend the fluid model, we see that they all start around the same loss as the fluid model by itself, and then improve from there. While F+NN and F+NN$_{mean}$ show similar performance, F+NN$_{\boldsymbol{W}}$ is able to improve the loss significantly faster, and also reaches a lower loss than the other two.

[1] https://github.com/albheim/IFAC2023_code/tree/thesis

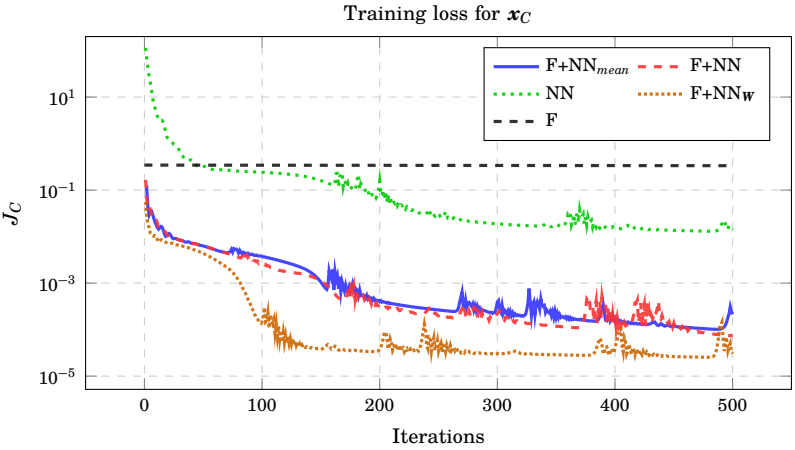

Figure 9.2 Training loss for the different models defined in Section 9.2, and the loss of the fluid model as reference.

To make sure this was consistent we ran additional experiments, both for larger networks and more training epochs. The inductive bias of both fluid model and constraints on the NN output still seems to be good, and significantly improves both sample efficiency and final loss for the time-horizons we trained over. In general, increasing the size of the networks did improve the models a little, though it also increased the training time by quite a lot. By training time we mean wall time per training step, and we do expect that this scales poorly with size since we currently use forward mode automatic differentiation, which is not very efficient for large networks. The sample efficiency might actually be slightly better for the larger NNs, up to some limit.

The high sample efficiency of F+NN$_{\boldsymbol{W}}$ seems reasonable since we have embedded the flow structure already, so the NN does not need to learn that, which likely provides a loss landscape that is easier to navigate. That it also has similar, or slightly better, performance than F+NN shows that the constraints are not restricting the model too much.

As the F+NN$_{mean}$ model performs similar to F+NN, we can conclude that the mean constraints are likely not restrictive enough to provide any additional benefit to the model.

Seeing this, we will evaluate the performance in more detail using only F+NN$_{\boldsymbol{W}}$ out of the extended models, and keeping both NN and F to compare against.

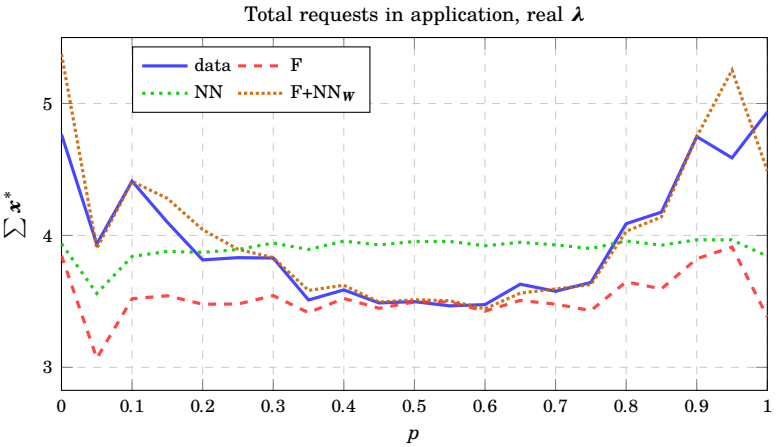

Figure 9.3 Predicted number of requests in the application using recorded $\boldsymbol{\lambda}$ for each evaluated p. Using the real $\boldsymbol{\lambda}$ allows for a better comparison of how well we fit the model dynamics instead of estimating the predictive capability on the stochastic workload. Given that F+NN$_{\boldsymbol{W}}$ seems to model the real data well, the model seems to have captured the dynamics well.

Evaluating Performance Metric Predictions

To start with, we look at how well the extended model predicts the total requests in the application by supplying the $\boldsymbol{\lambda}$ recorded for each p, see Figure 9.3. Basing the predictions on the real $\boldsymbol{\lambda}$ for each p makes for more accurate predictions, where the model can capture the actual dynamics of the application instead of noise from the input data.

The fluid model captures the real data well around $p = 0.6$ where it was trained, though as we move further away by updating p and $\boldsymbol{\lambda}$ in the model, we see the prediction accuracy decline. The NN model does not do too well, and seems to not have captured much of the dynamics in relation to p. The extended model is a lot more accurate and actually matches the data quite well, though we still see some artifacts, e.g., at $p = 0.95$ where the data decrease but the models increase. Since this can also be seen in the fluid model, it is likely an artifact from building upon it.

A use-case for these improvements is to find the load balancing parameters that optimize a cost function for the distributed application, as done in Chapter 8. We use the same cost function as was used for the offline experiments there, and evaluate the performance of the different models in Figure 9.4.

$$J(p) = \begin{cases} C_\varphi \varphi_\alpha \left(\boldsymbol{x}^*(\boldsymbol{\lambda}, p)\right) & \text{if } \varphi_\alpha > \varphi_{\text{lim}} \\ \boldsymbol{C}^T \boldsymbol{x}^*_Q \left(\boldsymbol{\lambda}, p\right) & \text{otherwise} \end{cases} \tag{9.6}$$

In Figure 9.4 we evaluate the fluid model, the NN model, and the F+NN$_W$ model. They are evaluated on the predicted request population, the predicted response time percentile and the predicted cost, and compared to the real data. The requests are based on a Poisson process, and the load balancer is random, so the real data show some stochastic behavior that the model does not predict. This is mainly from λ not being known for the values we predict, and instead estimated based on the most recent data, here being the arrival data recorded for $p = 0.6$. We can clearly see this difference when comparing the predictions for the total requests in Figure 9.3, using the individual estimates of λ for each p, and Figure 9.4(a), using the λ estimated for $p = 0.6$.

To estimate the cost function well the response time percentile plays a big role, as crossing it results in the cost function changing. As seen in Figure 9.4(b), neither the fluid model nor the NN model predicts crossing the response time threshold for low p, resulting in Figure 9.4(c) predicting a minimal cost for $p = 0$. This is the reason we retrained the model between each small change in p in Chapter 8, as predicting outside the training data could not be done reliably, and might push the system to a state where the constraints are broken. Collecting enough data to fit the fluid model takes at least a few minutes, so this method adds a large overhead to the optimization process. The extended fluid model F+NN$_W$ can, however, correctly predict crossing the response time threshold, and thus identifies a minimal cost that is similar to the real data. This allows for optimizing the control parameters over multiple steps without retraining the model, while still maintaining a good prediction accuracy.

Evaluating Control

To visualize this in a control scenario, Figure 9.5 shows how the control parameter is optimized using the model prediction. Using the model from Figure 9.4, we run gradient descent starting at $p = 0.6$ on the predicted values from the model. Exactly where the real minimum is will vary since the load is stochastic, but we can see that the found p ends up reasonably close to an optimal setting for the real data, instead of finding $p = 0$ as the fluid model would predict.

Summary and Discussion

By extending the models used in Chapter 8 with a small NN capturing the missing dynamics, the model's accuracy outside the current parameter settings is improved. This allows for updating the control parameters over more steps without retraining the model, while still maintaining a good prediction accuracy.

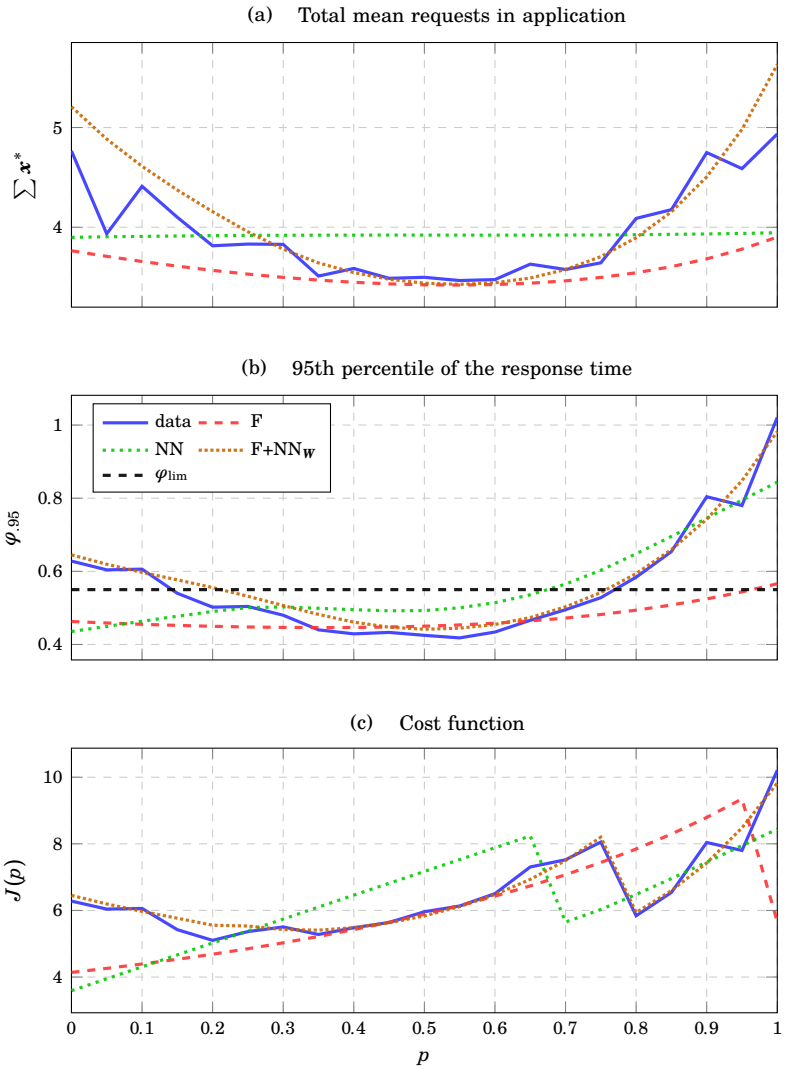

Figure 9.4 Here we see the recorded data in blue, together with predictions from a few different models. Predictions are based on $\boldsymbol{\lambda}$ for $p = 0.6$, and will thus not match the stochasticity of the varying load as was done in Figure 9.3. From the top we have the total number of requests, the 95th percentile response time and the cost function.

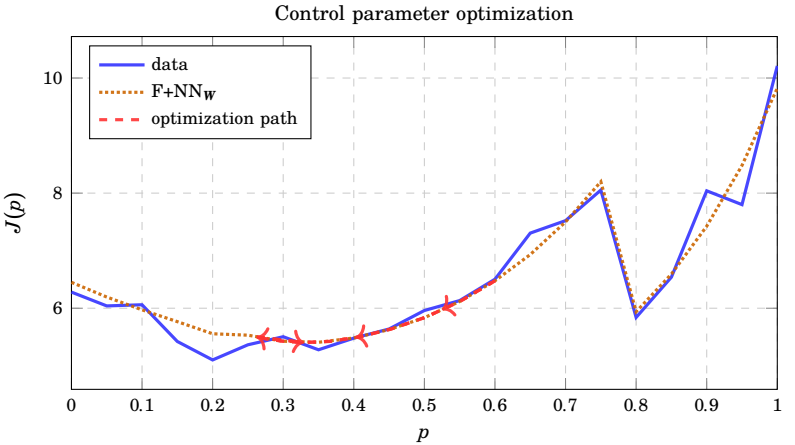

Figure 9.5 Gradient descent on p based on the loss of the trained F+NN$_W$ model. Starting at $p = 0.6$, it steps towards the minimum of the estimated loss. While it does overshoot the minimum a bit, it turns back and ends up at $p = 0.33$.

The act of embedding an NN in a DE can be seen as a type of inductive bias for the NN, and we also introduce additional ways of introducing inductive bias intro the network structure of the NN to compare with. The experiments show how the efficiency of training can increase by providing good inductive biases to the model, and how these models can perform better when it comes to capture correct dynamics.

Using the improved models, we do see that the control parameters can be more reliably optimized over multiple steps, ending up close to the actual optimal value with a single trained model. The fluid model would instead require many smaller steps where new data was collected and trained on in-between, resulting in slower convergence to good control parameters.

The method itself is more general than the specific use-case presented here, and both model improvement and control could similarly be done for a wide variety of systems that can be modelled using DEs.

10

Thesis Summary

This thesis has investigated the use of learning-based methods for control of datacenters and cloud infrastructure. The emphasis has been on reinforcement learning as a general framework for learning control policies, and how to approach the subject considering the many challenges that arise when applying RL to anything but small toy problems. In particular the thesis has presented the following contributions in each chapter.

- In Chapter 5 we apply a holistic strategy, controlling both cooling and load balancing using an RL agent, in a simulated datacenter. We show that the agent can learn to combine the control of both systems, and that it can improve on the PUE compared to a standard control system.

- In Chapter 6 we continue with context aware cooling control of a datacenter using RL, showing how the adaptive nature of RL can be beneficial in a changing environment.

- In Chapter 7 we show how microservice applications can benefit from proactive control, and how to design and implement an RL agent that can learn both an implicit representation of the service chains, and how to use that information to scale proactively.

- In Chapter 8 we apply automatic differentiation over a microservice fluid model to optimize load balancing parameters in a cloud application for some arbitrary costs and constraints.

- In Chapter 9 we look at improving an existing model by embedding NNs in the underlying ODE. We show how imposing structure on the NN can give additional benefits, and how the improved model can provide faster convergence when used for learning control policies.

10.1 Discussion

While we initially were optimistic about the potential of deep RL, trying to leave as much as possible to the agent to learn, we quickly realized that there are several practical complications with this approach. One problem is that RL is not sample efficient. Implementing the agent using black-box methods, while also trying to extend the context for the agent, quickly leads to a situation where the agent needs large volumes of data to learn anything. A second problem is that when the agent does learn, and performs well on average according to the given reward, it is often not robust and can randomly fail in ways that are difficult to predict. These two problems are related, since the agent needs to explore to learn, and the more it explores, the more likely it is to fail. Complex environments also present problems where, besides being more difficult to learn, they also make it harder for the one implementing the algorithm to evaluate what is going on and what is not working. As RL typically requires a lot of tuning and tweaking, this process is made more complicated when there is less insight into why and how certain things are happening.

With these problems in mind, we looked for ways of improving the agent's ability to learn. We did this by either applying common tricks as discussed in Chapter 4, or by embedding more information into the policy and the learning process. We explored different ways of embedding useful information, e.g., by imposing structure on the NN that makes up the policy, or by manually designing state transformations to provide more useful information. Both these approaches make use of our knowledge about the environment to reduce the load on the agent and the learning process. We also applied models as a means of providing more direct policy updates, aiding the learning process by providing more informative gradients.

Weaknesses and potential criticism. While we do believe that the methods presented in this thesis are interesting and useful, there are certainly many things that we might have done differently today with the knowledge we have now. Time and resource constraints have also limited what we have been able to do, as RL is a computationally intensive field, and all possible axes to explore create a combinatorial explosion of possible experiments.

We have shown that the RL methods can learn for the given objectives, often also outperforming some standard methods. However, one can argue that using the reward as a metric is unfair when one of the methods are explicitly optimizing for the reward. While we could come up with a different metric to compare the performance, that seems a bit unfair in the other direction, since part of the appeal of RL is that it can learn to optimize for arbitrary metrics as long as they can be expressed as a reward.

While there are other state-of-the-art RL methods published for similar

tasks, we mostly compare to simpler methods that are more standard in industry. One motivation for this is that it is not easy to make the comparison meaningful. For one, there is no real consensus on standard benchmark problems or datasets, making it difficult to directly compare one method to another. Using an agent that is not suited for the problem is not interesting, since it is unlikely to perform well, making for an unfair comparison that does not really say much. But when adapting an RL agent to a different problem, it can be unclear how much of the performance is due to the original agent and how much is due to adapting the agent well. The specific method from literature does not always make their code publicly available, and while the method might be described well, without the code there are often some implementation details that are not well-defined. So with slightly varying environments and objectives, and not always fully defined agents, it can be difficult to do a meaningful comparison. We saw more benefit in comparing to simpler methods that are standard in industry, providing a more stable baseline with a clear interpretation.

Some of our experiments could have benefitted from counterfactuals, e.g., what would have happened if we had not used a larger context for the RL agent. In some cases it was simply that we were focused on other objectives, while at other times it was not obvious how to do create a comparison that was fair and meaningful. For example, in Chapter 6 it would have been interesting to provide a comparison on how the agent performed without the larger context, but we were focused on the improvement compared to the baseline agents.

The problem of RL being opaque and compute-bound also forced us into iterative development. We repeatedly make small changes to the agent and train it, leaving us with large volumes of data that are not always comparable. While we have data convincing us that the methods work, we could have put more effort into making a convincing presentation of the results. Training an RL agent can generate very different policies, and it can be beneficial to present the results from multiple agents trained under slightly different circumstances, e.g., with different random seeds. While we did this in some experiments, presenting data from multiple agents can be difficult to do in a clear and concise way. Showing an average over many runs will likely show a much smoother learning experience, making the agent seem more stable than it actually is. And showing variance or individual runs can quickly become cluttered and hard to interpret when comparing multiple methods. We tend to show a single run, and try to explain the results based on our understanding of the agent and the environment. We also provide code to allow anyone to reproduce the results and draw their own conclusions.

Scalability is also a problem we could have addressed better, as it is not clear how well the methods used would scale to larger systems. For

example, we noted the increase in difficulty when moving from the smaller environment in Chapter 5 to the larger one in Chapter 6, not only from the larger scale but also from the increased complexity. Here we observed the benefit of hand-crafted features that reduce the size of the state. Reducing the complexity of the action space by removing the load balancing also had a large effect on the learning efficiency. Even though we did draft some ideas for how to create a more modular approach to learning in large systems, it was not further explored for this thesis.

10.2 Future Work

In this section we discuss some possible directions for future work. Apart from addressing the weaknesses mentioned above, there are also many other interesting topics that could be explored.

When it comes to deep RL methods, one of our main problems has been that neither the training process nor the final agent is very robust, and it is hard to know when something will fail unpredictably. As such, some methods mentioned in Section 3.5 could be of interest for a safer and more robust learning experience.

Another problem with deep RL is the lack of interpretability and guarantees on the policy. One possible approach to counter this is to create a parameterization of the policy function that is easier to understand and analyze. The work in [Lawrence et al., 2023] is an example of this, where the authors use the Youla-Kučera parameterization to refine the search space to be only stable policies, and optimize over them using RL.

Below we present a few interesting directions for future work connected to the different chapters.

Combining cooling and IT control in a DC. We still believe there can be a benefit to control cooling and load balancing together as done in Chapter 5, as these two systems do affect each other. Finding a method to control both aspects in a way that is scalable, and does not only result in the two parts creating noise for each other, could prove to be useful. Hierarchical RL could be interesting for this, allowing for some separation of concerns while still having a holistic top-level view.

Context aware cooling control in a DC. While we do improve on the target objective in Chapter 6, using an RL agent that is continuously learning and is aware of the IT load, this might not be the most interesting problem to solve. From discussions with industrial partners, one key factor that has been brought up is long term stability and autonomy. While autonomy could be achieved through continuous learning, stability is not really what RL is known for. With this in mind, it would be interesting to

look into robustness in RL, trying methods that create safer online learning and more robust policies. It would also be interesting to evaluate different methods on real world systems over longer periods of time, as the results from our short simulations might not be representative of the real world.

Proactive scaling of microservices. Similar to the previous chapters, the performance in Chapter 7 is good but rather stochastic, and it would be interesting to see if it is possible to create more robust policies. Another area we wanted to investigate was how to make the agent scale better to larger systems. One approach for better scaling is to make it modular, e.g., using a hierarchical RL approach. Another idea would be to assume the service graph can be extracted from system logs as is considered in Chapter 8, and use graph neural networks to process inputs from the extracted service graph in a more structured way. This would reduce the complexity of the problem for the agent, and thus scale better to larger systems.

Optimizing microservice load balancing parameters. The load balancing example in Chapter 8 was chosen to provide a simple model where the control variables were straightforward to understand, and which we could validate on a real system. However, the same method could equally well be used to optimize other control variables in a microservice application, e.g., the number of replicas for a service, as long as these variables can be represented in the model. The same method could also be used to optimize more complex controllers than a simple random load balancer, e.g., the parameters of a PID controller, or the weights of an NN-based policy.

One problem in Chapter 8 is that we only trust the model to make small gradient updates to the load balancing parameters between each update of the model. This makes for slow but safe updates, but it does not scale very well as the system grows. One approach could be to update the model more frequently, allowing for smaller steps but more often. Another solution is to make a better model that we can trust to take larger steps, and is part of what we attempt to do in Chapter 9.

Embedding NNs in an existing model. The techniques presented in Chapter 9 show how the efficiency and performance of NNs can be improved by providing part of the solution, or by imposing structural knowledge from the solution onto the NNs. One potential extension of these ideas could be to apply the same methods to a controller, where a known controller structure is extended with a small NN to further optimize the reward in an RL setting. There are also techniques that constrain the NN to be, e.g., Lipschitz continuous, which could provide further tools to reduce the variability and instability of an RL loop involving NNs, and maybe even provide some guarantees on the performance of the controller.

Bibliography

Abadi, M., A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mane, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viegas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng (2016). *TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems*. DOI: 10.48550/arXiv.1603.04467. preprint.

Ahmad, M., S. Duan, A. Aboulnaga, and S. Babu (2011). "Predicting completion times of batch query workloads using interaction-aware models and simulation". In: *Proceedings of the 14th International Conference on Extending Database Technology - EDBT/ICDT '11*. ACM Press, Uppsala, Sweden, p. 449. DOI: 10.1145/1951365.1951419.

Alvarez, V. M. M., R. Roşca, and C. G. Fălcuţescu (2020). "DyNODE: Neural Ordinary Differential Equations for Dynamics Modeling in Continuous Control". *arXiv e-prints*, arXiv–2009. arXiv: 2009.04278 [cs, eess, stat].

Amari, S. (1998). "Natural Gradient Works Efficiently in Learning". *Neural Computation* **10**:2, pp. 251–276. DOI: 10.1162/089976698300017746.

Andrychowicz, M., A. Raichuk, P. Stańczyk, M. Orsini, S. Girgin, R. Marinier, L. Hussenot, M. Geist, O. Pietquin, M. Michalski, S. Gelly, and O. Bachem (2020). *What Matters In On-Policy Reinforcement Learning? A Large-Scale Empirical Study*. arXiv: 2006.05990 [cs, stat]. preprint.

Anselmi, J. and G. Casale (2013). "Heavy-traffic revenue maximization in parallel multiclass queues". *Performance Evaluation* **70**:10, pp. 806–821. DOI: 10.1016/j.peva.2013.08.008.

Ardagna, D., G. Casale, M. Ciavotta, J. F. Pérez, and W. Wang (2014). "Quality-of-service in cloud computing: modeling techniques and their applications". *Journal of Internet Services and Applications* **5**:1, p. 11. DOI: 10.1186/s13174-014-0011-3.

ASHRAE TC9.9 (2016). *Data Center Power Equipment Thermal Guidelines and Best Practices*. URL: https://www.ashrae.org/File%20Library/Technical%20Resources/Bookstore/ASHRAE_TC0909_Power_White_Paper_22_June_2016_REVISED.pdf (visited on 2023-05-22).

Asmussen, S., O. Nerman, and M. Olsson (1996). "Fitting phase-type distributions via the EM algorithm". *Scandinavian J. Statistics* **23**:4, pp. 419–441. ISSN: 03036898, 14679469. JSTOR: 4616418.

Åström, K. (1965). "Optimal control of Markov processes with incomplete state information". *Journal of Mathematical Analysis and Applications* **10**:1, pp. 174–205. DOI: 10.1016/0022-247X(65)90154-X.

Åström, K. J. and B. Wittenmark (2008). *Adaptive Control: Second Edition*. Dover publications. ISBN: 978-0-486-46278-3.

Azar, Y., A. Z. Broder, A. R. Karlin, and E. Upfal (1994). "Balanced allocations". In: *Proceedings of the Twenty-Sixth Annual ACM Symposium on Theory of Computing*, pp. 593–602.

Baek, J.-Y., G. Kaddoum, S. Garg, K. Kaur, and V. Gravel (2019). "Managing Fog Networks using Reinforcement Learning Based Load Balancing Algorithm". In: *2019 IEEE Wireless Communications and Networking Conference (WCNC)*, pp. 1–7. DOI: 10.1109/WCNC.2019.8885745.

Bagge Carlson, F. (2018). *Hyperopt.jl : Hyperparameter optimization in Julia*. github. URL: https://github.com/baggepinnen/Hyperopt.jl.

Balter, M. (2013). *Performance Modeling and Design of Computer Systems : Queueing Theory in Action*. Cambridge University Press, Cambridge. ISBN: 978-1-107-02750-3.

Barrett, E., E. Howley, and J. Duggan (2013). "Applying reinforcement learning towards automating resource allocation and application scalability in the cloud". *Concurrency and Computation: Practice and Experience* **25**:12, pp. 1656–1674. DOI: 10.1002/cpe.2864.

Barroso, L. A., U. Hölzle, and P. Ranganathan (2019). *The Datacenter as a Computer: Designing Warehouse-Scale Machines, Third Edition*. Springer Nature. DOI: 10.1007/978-3-031-01761-2.

Bellman, R. (1954). "The theory of dynamic programming". *Bulletin of the American Mathematical Society* **60**:6, pp. 503–515. DOI: 10.1090/S0002-9904-1954-09848-8.

Bellman, R. (1957). "A Markovian Decision Process". *Journal of Mathematics and Mechanics* **6**:5, pp. 679–684. ISSN: 0095-9057.

Bengio, Y. (2009). "Learning Deep Architectures for AI". *Foundations and Trends® in Machine Learning* **2**:1, pp. 1–127. DOI: 10.1561/2200000006.

Bezanson, J., A. Edelman, S. Karpinski, and V. B. Shah (2017). "Julia: A Fresh Approach to Numerical Computing". *SIAM Review* **59**:1, pp. 65–98. DOI: 10.1137/141000671.

Bibal Benifa, J. V. and D. Dejey (2019). "RLPAS: Reinforcement Learning-based Proactive Auto-Scaler for Resource Provisioning in Cloud Environment". *Mobile Networks and Applications* **24**:4, pp. 1348–1363. DOI: 10.1007/s11036-018-0996-0.

Bitsakos, C., I. Konstantinou, and N. Koziris (2018). "DERP: A Deep Reinforcement Learning Cloud System for Elastic Resource Provisioning". In: *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 21–29. DOI: 10.1109/CloudCom2018.2018.00020.

Bolch, G., S. Greiner, H. de Meer, and K. S. Trivedi (2006). *Queueing Networks and Markov Chains*. John Wiley & Sons, Inc., Hoboken, N.J. DOI: 10.1002/0471791571.

Borgetto, D., M. Maurer, G. Da-Costa, J.-M. Pierson, and I. Brandic (2012). "Energy-efficient and SLA-aware management of IaaS clouds". In: *Proceedings of the 3rd International Conference on Future Energy Systems: Where Energy, Computing and Communication Meet*, pp. 1–10.

Borst, S. C. (1995). "Optimal probabilistic allocation of customer types to servers". *ACM SIGMETRICS Performance Evaluation Review* **23**:1, pp. 116–125.

Bottou, L., F. E. Curtis, and J. Nocedal (2018). "Optimization Methods for Large-Scale Machine Learning". *SIAM Review* **60**:2, pp. 223–311. DOI: 10.1137/16M1080173.

Bradbury, J., R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang (2018). *JAX: composable transformations of Python+NumPy programs*. Version 0.3.13. URL: http://github.com/google/jax.

Brady, G. A., N. Kapur, J. L. Summers, and H. M. Thompson (2013). "A case study and critical assessment in calculating power usage effectiveness for a data centre". *Energy Conversion and Management* **76**, pp. 155–161. DOI: 10.1016/j.enconman.2013.07.035.

Bramson, M., Y. Lu, and B. Prabhakar (2010). "Randomized load balancing with general service time distributions". In: SIGMETRICS '10. Association for Computing Machinery, New York, NY, USA, pp. 275–286. DOI: 10.1145/1811039.1811071.

Brockman, G., V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba (2016). *OpenAI Gym*. arXiv: 1606.01540 [cs]. preprint.

Brunton, S. L., J. L. Proctor, and J. N. Kutz (2016). "Discovering governing equations from data by sparse identification of nonlinear dynamical systems". *Proceedings of the national academy of sciences* **113**:15, pp. 3932–3937.

Bu, X., J. Rao, and C.-Z. Xu (2009). "A Reinforcement Learning Approach to Online Web Systems Auto-configuration". In: *2009 29th IEEE International Conference on Distributed Computing Systems*. IEEE, Montreal, Quebec, Canada, pp. 2–11. DOI: 10.1109/ICDCS.2009.76.

Buckingham, E. (1914). "On physically similar systems' illustrations of the use of dimensional equations". *Physical Review* **4**:4, pp. 345–376. DOI: 10.1103/physrev.4.345.

Cerny, T., M. J. Donahoo, and M. Trnka (2018). "Contextual understanding of microservice architecture: Current and future directions". *ACM SIGAPP Applied Computing Review* **17**:4, pp. 29–45. DOI: 10.1145/3183628.3183631.

Chen, T. Q., Y. Rubanova, J. Bettencourt, and D. Duvenaud (2018). "Neural ordinary differential equations". *CoRR* **abs/1806.07366**. arXiv: 1806.07366.

Cheng, M., J. Li, and S. Nazarian (2018). "DRL-cloud: Deep reinforcement learning-based resource provisioning and task scheduling for cloud service providers". In: *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 129–134. DOI: 10.1109/ASPDAC.2018.8297294.

Chi, C., K. Ji, A. Marahatta, P. Song, F. Zhang, and Z. Liu (2020). "Jointly Optimizing the IT and Cooling Systems for Data Center Energy Efficiency based on Multi-Agent Deep Reinforcement Learning". In: *Proceedings of the Eleventh ACM International Conference on Future Energy Systems*. ACM, Virtual Event Australia, pp. 489–495. DOI: 10.1145/3396851.3402658.

Christodoulou, P. (2019). *Soft Actor-Critic for Discrete Action Settings*. DOI: 10.48550/arXiv.1910.07207. preprint.

Chua, K., R. Calandra, R. McAllister, and S. Levine (2018). *Deep Reinforcement Learning in a Handful of Trials using Probabilistic Dynamics Models*. DOI: 10.48550/arXiv.1805.12114. preprint.

Clevert, D.-A., T. Unterthiner, and S. Hochreiter (2016). *Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)*. Version 5. DOI: 10.48550/arXiv.1511.07289. preprint.

COMSOL (2023). *The Boussinesq Approximation*. URL: https://www.comsol.com/multiphysics/boussinesq-approximation (visited on 2023-08-08).

Crawley, D. B., L. K. Lawrie, F. C. Winkelmann, W. F. Buhl, Y. J. Huang, C. O. Pedersen, R. K. Strand, R. J. Liesen, D. E. Fisher, M. J. Witte, and J. Glazer (2001). "EnergyPlus: Creating a new-generation building energy simulation program". *Energy and Buildings*. Special Issue: BUILDING SIMULATION'99 **33**:4, pp. 319–331. DOI: 10.1016/S0378-7788(00)00114-6.

Dann, C., Y. Mansour, M. Mohri, A. Sekhari, and K. Sridharan (2022). "Guarantees for Epsilon-Greedy Reinforcement Learning with Function Approximation". In: *Proceedings of the 39th International Conference on Machine Learning*. PMLR, pp. 4666–4689.

Davis, J. (2022). "Uptime Institute Global Data Center Survey 2022".

Deisenroth, M. P. and C. E. Rasmussen (2011). "PILCO: A Model-Based and Data-Efficient Approach to Policy Search". *Proceedings of the 28th International Conference on machine learning (ICML-11)*, pp. 465–472.

Delbosc, N. (2015). *Real-Time Simulation of Indoor Air Flow Using the Lattice Boltzmann Method on Graphics Processing Unit*. PhD thesis. University of Leeds.

Dong, T., F. Xue, C. Xiao, and J. Li (2020). "Task scheduling based on deep reinforcement learning in a cloud manufacturing environment". *Concurrency and Computation: Practice and Experience* **32**:11, e5654. DOI: 10.1002/cpe.5654.

Dulac-Arnold, G., N. Levine, D. J. Mankowitz, J. Li, C. Paduraru, S. Gowal, and T. Hester (2021). "Challenges of real-world reinforcement learning: definitions, benchmarks and analysis". *Machine Learning* **110**:9, pp. 2419–2468. DOI: 10.1007/s10994-021-05961-4.

Eastham, M. S. P. (1961). "2968. On the Definition of Dual Numbers". *The Mathematical Gazette* **45**:353, pp. 232–233. DOI: 10.2307/3612794.

Feinberg, V., A. Wan, I. Stoica, M. I. Jordan, J. E. Gonzalez, and S. Levine (2018). *Model-Based Value Estimation for Efficient Model-Free Reinforcement Learning*. DOI: 10.48550/arXiv.1803.00101. preprint.

Flask (2023). URL: https://flask.palletsprojects.com/en/2.3.x/ (visited on 2023-06-30).

Fratta, L., M. Gerla, and L. Kleinrock (1973). "The flow deviation method: An approach to store-and-forward communication network design". *Networks. An International Journal* **3**:2, pp. 97–133.

Gan, Y., Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou (2019). "An open-source benchmark suite for microservices and their hardware-software implications for cloud

& edge systems". In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '19. Association for Computing Machinery, Providence, RI, USA, pp. 3–18. DOI: 10.1145/3297858.3304013.

Gao, J. (2014). *Machine Learning Applications for Data Center Optimization*. URL: https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/42542.pdf (visited on 2023-05-23).

Garcia-Gabin, W., K. Mishchenko, and E. Berglund (2018). "Cooling Control of Data Centers Using Linear Quadratic Regulators". In: *2018 26th Mediterranean Conference on Control and Automation (MED)*, pp. 1–6. DOI: 10.1109/MED.2018.8442429.

Garcıa, J. and F. Fernandez (2015). "A Comprehensive Survey on Safe Reinforcement Learning". *Journal of Machine Learning Research* **16**:1, pp. 1437–1480.

Gasparik, A., C. Gamble, and J. Gao (2018). *Safety-first AI for autonomous data center cooling and industrial control*. Google. URL: https://blog.google/inside-google/infrastructure/safety-first-ai-autonomous-data-center-cooling-and-industrial-control/ (visited on 2023-04-27).

Ghobaei-Arani, M., S. Jabbehdari, and M. A. Pourmina (2018). "An autonomic resource provisioning approach for service-based cloud applications: A hybrid approach". *Future Generation Computer Systems* **78**, pp. 191–210. DOI: 10.1016/j.future.2017.02.022.

González-García, R., R. Rico-Martìnez, and I. G. Kevrekidis (1998). "Identification of distributed parameter systems: A neural net based approach". *Computers & chemical engineering* **22**, S965–S968.

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

Google Trends (2023). URL: https://trends.google.com/trends/ (visited on 2023-01-15).

Grondman, I., L. Busoniu, G. A. D. Lopes, and R. Babuska (2012). "A Survey of Actor-Critic Reinforcement Learning: Standard and Natural Policy Gradients". *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* **42**:6, pp. 1291–1307. DOI: 10.1109/TSMCC.2012.2218595.

Grzes, M. (2017). "Reward Shaping in Episodic Reinforcement Learning". ACM.

Gu, S., L. Yang, Y. Du, G. Chen, F. Walter, J. Wang, Y. Yang, and A. Knoll (2023). *A Review of Safe Reinforcement Learning: Methods, Theory and Applications*. DOI: 10.48550/arXiv.2205.10330. preprint.

Gunicorn (2023). URL: https://gunicorn.org/ (visited on 2023-06-30).

Guo, X., Y. Lu, and M. S. Squillante (2004). "Optimal probabilistic routing in distributed parallel queues". *PERFORMANCE EVALUATION REVIEW* **32**:2, pp. 53–54.

Gupta, V., M. Harchol Balter, K. Sigman, and W. Whitt (2007). "Analysis of join-the-shortest-queue routing for web server farms". *Performance Evaluation* **64**:9, pp. 1062–1081. DOI: 10.1016/j.peva.2007.06.012.

Ha, D. and J. Schmidhuber (2018a). *Recurrent World Models Facilitate Policy Evolution*. DOI: 10.48550/arXiv.1809.01999. preprint.

Ha, D. and J. Schmidhuber (2018b). "World Models". DOI: 10.5281/zenodo.1207631.

Haarnoja, T., A. Zhou, P. Abbeel, and S. Levine (2018a). "Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor". arXiv: 1801.01290 [cs, stat].

Haarnoja, T., A. Zhou, K. Hartikainen, G. Tucker, S. Ha, J. Tan, V. Kumar, H. Zhu, A. Gupta, P. Abbeel, and S. Levine (2018b). "Soft Actor-Critic Algorithms and Applications". arXiv: 1812.05905 [cs, stat].

Hafner, D., J. Pasukonis, J. Ba, and T. Lillicrap (2023). *Mastering Diverse Domains through World Models*. DOI: 10.48550/arXiv.2301.04104. preprint.

Hasan, M. Z., E. Magana, A. Clemm, L. Tucker, and S. L. D. Gudreddi (2012). "Integrated and autonomic cloud resource scaling". In: *2012 IEEE Network Operations and Management Symposium*, pp. 1327–1334. DOI: 10.1109/NOMS.2012.6212070.

Hasani, R., M. Lechner, A. Amini, D. Rus, and R. Grosu (2020). *Liquid Time-constant Networks*. arXiv: 2006.04439 [cs, stat]. preprint.

HashiCorp (2023). *Terraform*. Terraform by HashiCorp. URL: https://www.terraform.io/ (visited on 2023-05-19).

He, J., D. Zhou, and Q. Gu (2021). "Uniform-PAC Bounds for Reinforcement Learning with Linear Function Approximation". In: *Advances in Neural Information Processing Systems*. Vol. 34. Curran Associates, Inc., pp. 14188–14199.

Healey, C., J. VanGilder, M. Condor, and W. Tian (2018). "Transient data center temperatures after a primary power outage". In: Proceedings of the 17th InterSociety Conference on Thermal and Thermomechanical Phenomena in Electronic Systems, ITherm 2018, pp. 865–870. DOI: 10.1109/ITHERM.2018.8419583.

Heinonen, M., C. Yildiz, H. Mannerström, J. Intosalmi, and H. Lähdesmäki (2018). "Learning unknown ODE models with Gaussian processes". In: Dy, J. et al. (Eds.). *Proceedings of the 35th International Conference on Machine Learning*. Vol. 80. Proceedings of Machine Learning Research. PMLR, pp. 1959–1968.

Hewing, L., K. P. Wabersich, M. Menner, and M. N. Zeilinger (2020). "Learning-Based Model Predictive Control: Toward Safe Learning in Control". *Annual Review of Control, Robotics, and Autonomous Systems* **3**:1, pp. 269–296. DOI: 10.1146/annurev-control-090419-075625.

Hordijk, A. and J. A. Loeve (2000). "Optimal static customer routing in a closed queuing network". *Statistica Neerlandica* **54**:2, pp. 148–159.

Hornik, K., M. Stinchcombe, and H. White (1989). "Multilayer feedforward networks are universal approximators". *Neural Networks* **2**:5, pp. 359–366. DOI: 10.1016/0893-6080(89)90020-8.

Hu, Y., W. Wang, H. Jia, Y. Wang, Y. Chen, J. Hao, F. Wu, and C. Fan (2020). "Learning to Utilize Shaping Rewards: A New Approach of Reward Shaping". *Advances in Neural Information Processing Systems* **33**, pp. 15931–15941.

Hua, T., J. Wan, S. Jaffry, Z. Rasheed, L. Li, and Z. Ma (2021). "Comparison of Deep Reinforcement Learning Algorithms in Data Center Cooling Management: A Case Study". In: *2021 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pp. 392–397. DOI: 10.1109/SMC52423.2021.9659100.

Hutsebaut-Buysse, M., K. Mets, and S. Latré (2022). "Hierarchical Reinforcement Learning: A Survey and Open Research Challenges". *Machine Learning and Knowledge Extraction* **4**:1 (1), pp. 172–221. DOI: 10.3390/make4010009.

Hutter, F., L. Kotthoff, and J. Vanschoren, (Eds.) (2019). *Automated Machine Learning: Methods, Systems, Challenges*. The Springer Series on Challenges in Machine Learning. Springer International Publishing, Cham. DOI: 10.1007/978-3-030-05318-5.

Incerto, E., M. Tribastone, and C. Trubiani (2017). "Software performance self-adaptation through efficient model predictive control". In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 485–496. DOI: 10.1109/ASE.2017.8115660.

Incerto, E., M. Tribastone, and C. Trubiani (2018). "Combined vertical and horizontal autoscaling through model predictive control". In: Aldinucci, M. et al. (Eds.). *Euro-Par 2018: Parallel Processing*. Springer International Publishing, Cham, pp. 147–159. ISBN: 978-3-319-96983-1.

Innes, M., E. Saba, K. Fischer, D. Gandhi, M. C. Rudilosso, N. M. Joy, T. Karmali, A. Pal, and V. Shah (2018). *Fashionable Modelling with Flux*. DOI: 10.48550/arXiv.1811.01457. preprint.

Istio (2023). Istio. URL: https://istio.io/ (visited on 2023-05-23).

Kaddour, J., A. Lynch, Q. Liu, M. J. Kusner, and R. Silva (2022). *Causal Machine Learning: A Survey and Open Problems*. arXiv: 2206.15475 [cs, stat]. preprint.

Kakade, S. M. (2001). “A Natural Policy Gradient”. In: *Advances in Neural Information Processing Systems*. Vol. 14. MIT Press.

Kamiya, G. and O. Kvarnström (2019). *Data Centres and Energy – from Global Headlines to Local Headaches?* IEA. URL: https://www.iea.org/commentaries/data-centres-and-energy-from-global-headlines-to-local-headaches (visited on 2023-04-19).

Kamthe, S. and M. Deisenroth (2018). “Data-Efficient Reinforcement Learning with Probabilistic Model Predictive Control”. In: *Proceedings of the Twenty-First International Conference on Artificial Intelligence and Statistics*. PMLR, pp. 1701–1710.

Kanbar, A. B. and K. Faraj (2022). “Region aware dynamic task scheduling and resource virtualization for load balancing in IoT–fog multi-cloud environment”. *Future Generation Computer Systems* **137**, pp. 70–86. DOI: 10.1016/j.future.2022.06.005.

Khatri, A. and V. Khatri (2020). *Mastering Service Mesh: Enhance, Secure, and Observe Cloud-Native Applications with Istio, Linkerd, and Consul*. Packt Publishing Ltd. 606 pp. ISBN: 978-1-78961-194-6.

Kingma, D. P. and J. Ba (2017). *Adam: A Method for Stochastic Optimization*. DOI: 10.48550/arXiv.1412.6980. preprint.

Kiran, M. and M. Ozyildirim (2022). *Hyperparameter Tuning for Deep Reinforcement Learning Applications*. arXiv: 2201.11182 [cs]. preprint.

Kobayashi, H. and M. Gerla (1983). “Optimal routing in closed queuing networks”. *ACM Transactions on Computer Systems (TOCS)* **1**:4, pp. 294–310.

Koot, M. and F. Wijnhoven (2021). “Usage impact on data center electricity needs: A system dynamic forecasting model”. *Applied Energy* **291**, p. 116798. ISSN: 0306-2619.

Kubernetes (2023). Kubernetes. URL: https://kubernetes.io/ (visited on 2023-05-19).

Kubernetes HPA (2023). Kubernetes. URL: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/ (visited on 2023-05-26).

Kumar, J., R. Goomer, and A. K. Singh (2018). “Long Short Term Memory Recurrent Neural Network (LSTM-RNN) Based Workload Forecasting Model For Cloud Datacenters”. *Procedia Computer Science*. The 6th International Conference on Smart Computing and Communications **125**, pp. 676–682. DOI: 10.1016/j.procs.2017.12.087.

Lawrence, N. P., P. D. Loewen, S. Wang, M. G. Forbes, and R. B. Gopaluni (2023). *A modular framework for stabilizing deep reinforcement learning control*. arXiv: 2304.03422 [cs, eess]. preprint.

Lazic, N., T. Lu, C. Boutilier, M. K. Ryu, E. J. Wong, B. Roy, and G. Imwalle (2018). "Data Center Cooling using Model-predictive Control". In: *Proceedings of the Thirty-second Conference on Neural Information Processing Systems (NeurIPS-18)*. Montreal, QC, pp. 3818–3827.

Lee, R. and B. Jeng (2011). "Load-balancing tactics in cloud". In: *2011 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery*. IEEE, pp. 447–454.

Leung, J. Y.-T. (2004). *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. CRC Press. 1215 pp. ISBN: 978-0-203-48980-2.

Levine, S. and V. Koltun (2013). "Guided Policy Search". In: *Proceedings of the 30th International Conference on Machine Learning*. PMLR, pp. 1–9.

Levine, S., A. Kumar, G. Tucker, and J. Fu (2020). *Offline Reinforcement Learning: Tutorial, Review, and Perspectives on Open Problems*. DOI: 10.48550/arXiv.2005.01643. preprint.

Li, B. and L. Xia (2015). "A multi-grid reinforcement learning method for energy conservation and comfort of HVAC in buildings". In: *2015 IEEE International Conference on Automation Science and Engineering (CASE)*, pp. 444–449. DOI: 10.1109/CoASE.2015.7294119.

Li, Y., Y. Wen, D. Tao, and K. Guan (2020). "Transforming Cooling Optimization for Green Data Center via Deep Reinforcement Learning". *IEEE Transactions on Cybernetics* **50**:5, pp. 2002–2013. DOI: 10.1109/TCYB.2019.2927410.

Liang, E., R. Liaw, P. Moritz, R. Nishihara, R. Fox, K. Goldberg, J. E. Gonzalez, M. I. Jordan, and I. Stoica (2018). "RLlib: Abstractions for Distributed Reinforcement Learning". In: *International Conference on Machine Learning*, pp. 3053–3062. DOI: 10.48550/arXiv.1712.09381.

Liaw, R., E. Liang, R. Nishihara, P. Moritz, J. E. Gonzalez, and I. Stoica (2018). *Tune: A Research Platform for Distributed Model Selection and Training*. DOI: 10.48550/arXiv.1807.05118. preprint.

Liu, D. C. and J. Nocedal (1989). "On the limited memory BFGS method for large scale optimization". *Mathematical programming* **45**:1-3, pp. 503–528.

Liu, Z., Y. Chen, C. Bash, A. Wierman, D. Gmach, Z. Wang, M. Marwah, and C. Hyser (2012). "Renewable and cooling aware workload management for sustainable data centers". In: *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*. SIGMETRICS '12. Association for Computing Machinery, New York, NY, USA, pp. 175–186. DOI: 10.1145/2254756.2254779.

Ljung, L., T. Glad, and A. Hansson (2021). *Modeling and Identification of Dynamic Systems*. Studentlitteratur. ISBN: 978-91-44-15345-2.

Long, Z., Y. Lu, X. Ma, and B. Dong (2018). "PDE-Net: Learning PDEs from data". In: Dy, J. et al. (Eds.). *Proceedings of the 35th International Conference on Machine Learning*. Vol. 80. Proceedings of Machine Learning Research. PMLR, pp. 3208–3216.

Lu, Y., Q. Xie, G. Kliot, A. Geller, J. R. Larus, and A. Greenberg (2011). "Join-Idle-Queue: A novel load balancing algorithm for dynamically scalable web services". *Performance Evaluation* **68**:11, pp. 1056–1071. DOI: 10.1016/j.peva.2011.07.015.

Lundin, L. (2021). *Artificial Intelligence for Data Center Power Consumption Optimisation*. MA thesis. Uppsala University.

Mao, H., M. Alizadeh, I. Menache, and S. Kandula (2016). "Resource Management with Deep Reinforcement Learning". In: *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*. ACM, Atlanta GA USA, pp. 50–56. DOI: 10.1145/3005745.3005750.

Mell, P. and T. Grance (2011). *The NIST Definition of Cloud Computing*. NIST Special Publication 800-145. National Institute of Standards and Technology. URL: https://doi.org/10.6028/NIST.SP.800-145.

Millnert, V., E. Bini, and J. Eker (2018). "AutoSAC: automatic scaling and admission control of forwarding graphs". *Annals of Telecommunications* **73**:3, pp. 193–204. DOI: 10.1007/s12243-017-0597-0.

Minsky, M. (1961). "Steps toward Artificial Intelligence". *Proceedings of the IRE* **49**:1, pp. 8–30. DOI: 10.1109/JRPROC.1961.287775.

Mnih, V., K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller (2013). "Playing Atari with Deep Reinforcement Learning". arXiv: 1312.5602.

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

Moerland, T. M., J. Broekens, A. Plaat, and C. M. Jonker (2022). *Model-based Reinforcement Learning: A Survey*. DOI: 10.48550/arXiv.2006.16712. preprint.

Moos, J., K. Hansel, H. Abdulsamad, S. Stark, D. Clever, and J. Peters (2022). "Robust Reinforcement Learning: A Review of Foundations and Recent Advances". *Machine Learning and Knowledge Extraction* **4**:1 (1), pp. 276–315. DOI: 10.3390/make4010013.

Moritz, P., R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica (2018). "Ray: A Distributed Framework for Emerging AI Applications". arXiv: 1712.05889 [cs, stat].

Naveen, S. and M. R. Kounte (2019). "Key Technologies and challenges in IoT Edge Computing". In: *2019 Third International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC)*, pp. 61–65. DOI: 10.1109/I-SMAC47947.2019.9032541.

Ni, J. and X. Bai (2017). "A review of air conditioning energy performance in data centers". *Renewable and Sustainable Energy Reviews* **67**, pp. 625–640. ISSN: 1364-0321.

OpenStack (2023). OpenStack. URL: https://www.openstack.org/ (visited on 2023-05-19).

Paszke, A., S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala (2019). *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. DOI: 10.48550/arXiv.1912.01703. preprint.

Patterson, M. K. (2008). "The effect of data center temperature on energy efficiency". In: *2008 11th Intersociety Conference on Thermal and Thermomechanical Phenomena in Electronic Systems*, pp. 1167–1174. DOI: 10.1109/ITHERM.2008.4544393.

Paya, A. and D. C. Marinescu (2017). "Energy-Aware Load Balancing and Application Scaling for the Cloud Ecosystem". *IEEE Transactions on Cloud Computing* **5**:1, pp. 15–27. DOI: 10.1109/TCC.2015.2396059.

Poggio, T., H. Mhaskar, L. Rosasco, B. Miranda, and Q. Liao (2017). *Why and When Can Deep – but Not Shallow – Networks Avoid the Curse of Dimensionality: a Review*. DOI: 10.48550/arXiv.1611.00740. preprint.

Polyak, B. T. and A. B. Juditsky (1992). "Acceleration of Stochastic Approximation by Averaging". *SIAM Journal on Control and Optimization* **30**:4, pp. 838–855. DOI: 10.1137/0330046.

Prometheus (2023). Prometheus - Monitoring system & time series database. URL: https://prometheus.io/ (visited on 2023-05-24).

Rackauckas, C., Y. Ma, J. Martensen, C. Warner, K. Zubov, R. Supekar, D. Skinner, and A. J. Ramadhan (2020). "Universal differential equations for scientific machine learning". *CoRR* **abs/2001.04385**. arXiv: 2001.04385.

Rackauckas, C. and Q. Nie (2017). "Differentialequations.jl–a performant and feature-rich ecosystem for solving differential equations in julia". *Journal of Open Research Software* **5**:1, p.15.

Raffin, A., A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann (2021). "Stable-Baselines3: Reliable Reinforcement Learning Implementations". *Journal of Machine Learning Research* **22**:268, pp. 1–8. ISSN: 1533-7928.

Raissi, M. and G. E. Karniadakis (2018). "Hidden physics models: Machine learning of nonlinear partial differential equations". *Journal of Computational Physics* **357**, pp. 125–141.

Rall, L. B. (1981). *Automatic Differentiation: Techniques and Applications*. Springer.

Ran, Y., H. Hu, X. Zhou, and Y. Wen (2019). "DeepEE: Joint Optimization of Job Scheduling and Cooling Control for Data Center Energy Efficiency Using Deep Reinforcement Learning". In: *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pp. 645–655. DOI: 10.1109/ICDCS.2019.00070.

Red Hat (2023). *Ansible*. URL: https://www.ansible.com (visited on 2023-05-19).

Revels, J., M. Lubin, and T. Papamarkou (2016). "Forward-Mode Automatic Differentiation in Julia". arXiv: 1607.07892 [cs].

Rico-Martínez, R., K. Krischer, I. Kevrekidis, M. Kube, and J. Hudson (1992). "DISCRETE- vs. CONTINUOUS-TIME NONLINEAR SIGNAL PROCESSING OF Cu ELECTRODISSOLUTION DATA". *Chemical Engineering Communications* **118**:1, pp. 25–48. DOI: 10.1080/00986449208936084.

Rossi, F., M. Nardelli, and V. Cardellini (2019). "Horizontal and Vertical Scaling of Container-Based Applications Using Reinforcement Learning". In: *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pp. 329–338. DOI: 10.1109/CLOUD.2019.00061.

Różańska, M. and G. Horn (2022). "Proactive Autonomic Cloud Application Management". In: *2022 IEEE/ACM 15th International Conference on Utility and Cloud Computing (UCC)*, pp. 102–111. DOI: 10.1109/UCC56403.2022.00021.

Rudy, S. H., S. L. Brunton, J. L. Proctor, and J. N. Kutz (2017). "Data-driven discovery of partial differential equations". *Science advances* **3**:4, e1602614.

Ruuskanen, J., T. Berner, K.-E. Årzén, and A. Cervin (2021a). "Improving the mean-field fluid model of processor sharing queueing networks for dynamic performance models in cloud computing". *Performance Evaluation* **151**, p. 102231. DOI: 10.1016/j.peva.2021.102231.

Ruuskanen, J. and A. Cervin (2022). "Distributed online extraction of a fluid model for microservice applications using local tracing data". In: *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*. IEEE, pp. 179–190.

Ruuskanen, J., H. Peng, A. Åkesson, L. Larsson, and M. Kihl (2021b). *FedApp: a research sandbox for application orchestration in federated clouds using OpenStack*. arXiv: 2109.01480 [cs.DC].

Schaul, T., G. Ostrovski, I. Kemaev, and D. Borsa (2021). *Return-based Scaling: Yet Another Normalisation Trick for Deep RL*. arXiv: 2105.05347 [cs, stat]. preprint.

Schaul, T., J. Quan, I. Antonoglou, and D. Silver (2016). *Prioritized Experience Replay*. DOI: 10.48550/arXiv.1511.05952. preprint.

Schober, M., D. K. Duvenaud, and P. Hennig (2014). "Probabilistic ODE solvers with runge-kutta means". *Advances in neural information processing systems* **27**.

Schulman, J., S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel (2017a). *Trust Region Policy Optimization*. DOI: 10.48550/arXiv.1502.05477. preprint.

Schulman, J., P. Moritz, S. Levine, M. Jordan, and P. Abbeel (2018). *High-Dimensional Continuous Control Using Generalized Advantage Estimation*. DOI: 10.48550/arXiv.1506.02438. preprint.

Schulman, J., F. Wolski, P. Dhariwal, A. Radford, and O. Klimov (2017b). *Proximal Policy Optimization Algorithms*. arXiv: 1707 . 06347 [cs]. preprint.

Shahab Samani, F. and R. Stadler (2022). "Dynamically meeting performance objectives for multiple services on a service mesh". In: *2022 18th International Conference on Network and Service Management (CNSM)*. IEEE, Thessaloniki, Greece, pp. 219–225. DOI: 10.23919/CNSM55787.2022.9965074.

Shahin, A. A. (2016). "Automatic Cloud Resource Scaling Algorithm based on Long Short-Term Memory Recurrent Neural Network". *International Journal of Advanced Computer Science and Applications* **7**:12. DOI: 10.14569/IJACSA.2016.071236.

Sharma, S., S. Singh, and M. Sharma (2008). "Performance Analysis of Load Balancing Algorithms". *International Journal of Civil and Environmental Engineering* **2**:2, pp. 367–370.

Sharma, S., S. Sharma, and A. Athaiya (2020). "ACTIVATION FUNCTIONS IN NEURAL NETWORKS". *International Journal of Engineering Applied Sciences and Technology* **04**:12, pp. 310–316. DOI: 10.33564/IJEAST.2020.v04i12.054.

Silver, D., A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis (2016). "Mastering the game of Go with deep neural networks and tree search". *Nature* **529**:7587, pp. 484–489. DOI: 10.1038/nature16961.

Sjölund, J. (2018). *Real-Time Thermal Flow Predictions for Data Centers : Using the Lattice Boltzmann Method on Graphics Processing Units for Predicting Thermal Flow in Data Centers*. MA thesis. Luleå University of Technology.

Sjölund, J., M. Vesterlund, N. Delbosc, A. Khan, and J. Summers (2018). "Validated Thermal Air Management Simulations of Data Centers Using Remote Graphics Processing Units". In: *IECON 2018 - 44th Annual Conference of the IEEE Industrial Electronics Society*, pp. 4920–4925. DOI: 10.1109/IECON.2018.8591192.

Skarin, P. (2021). *Control over the Cloud: Offloading, Elastic Computing, and Predictive Control*. Department of Automatic Control, Lund University / Department of Automatic Control, Lund University. ISBN: 978-91-8039-093-4.

SMHI (2021). *SMHI - Sveriges meteorologiska och hydrologiska institut*. URL: https://www.smhi.se/data/meteorologi/ladda-ner-meteorologiska-observationer#param=airtemperatureInstant,stations=all,stationid=162870 (visited on 2021-09-28).

Spicuglia, S., L. Y. Chen, and W. Binder (2013). "Join the best queue: Reducing performance variability in heterogeneous systems". In: *2013 IEEE Sixth International Conference on Cloud Computing*. IEEE, pp. 139–146.

Sukop, M. C. and D. T. Thorne (2006). *Lattice Boltzmann Modeling: An Introduction for Geoscientists and Engineers*. Springer, Berlin, Heidelberg. DOI: 10.1007/978-3-540-27982-2.

Sutton, R., A. Barto, and R. Williams (1992). "Reinforcement learning is direct adaptive optimal control". *IEEE Control Systems Magazine* **12**:2, pp. 19–22. DOI: 10.1109/37.126844.

Sutton, R. S. (1988). "Learning to predict by the methods of temporal differences". *Machine Learning* **3**:1, pp. 9–44. DOI: 10.1007/BF00115009.

Sutton, R. S. (1991). "Dyna, an integrated architecture for learning, planning, and reacting". *ACM SIGART Bulletin* **2**:4, pp. 160–163. DOI: 10.1145/122344.122377.

Sutton, R. S. and A. G. Barto (2018). *Reinforcement Learning: An Introduction*. Second edition. Adaptive Computation and Machine Learning Series. The MIT Press, Cambridge, Massachusetts. 526 pp. ISBN: 978-0-262-03924-6.

Swoyer, S. and M. Loukides (2020). *Microservices adoption in 2020*. URL: https://www.oreilly.com/radar/microservices-adoption-in-2020/.

Tc-Netem (2023). URL: https://www.man7.org/linux/man-pages/man8/tc-netem.8.html (visited on 2023-06-30).

Tesauro, G. (1992). "Temporal Difference Learning of Backgammon Strategy". In: Sleeman, D. et al. (Eds.). *Machine Learning Proceedings 1992*. Morgan Kaufmann, San Francisco (CA), pp. 451–457. DOI: 10.1016/B978-1-55860-247-2.50063-2.

Tian, J. et al. (2020). *ReinforcementLearning.jl: A reinforcement learning package for the julia language*. URL: https://github.com/JuliaReinforcementLearning/ReinforcementLearning.jl.

Toka, L., G. Dobreff, B. Fodor, and B. Sonkoly (2020). "Adaptive AI-based auto-scaling for Kubernetes". In: *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, pp. 599–608. DOI: 10.1109/CCGrid49817.2020.00-33.

Townend, P., S. Clement, D. Burdett, R. Yang, J. Shaw, B. Slater, and J. Xu (2019). "Improving Data Center Efficiency Through Holistic Scheduling In Kubernetes". In: *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, pp. 156–15610. DOI: 10.1109/SOSE.2019.00030.

Van Le, D., Y. Liu, R. Wang, R. Tan, Y.-W. Wong, and Y. Wen (2019). "Control of Air Free-Cooled Data Centers in Tropics via Deep Reinforcement Learning". In: *Proceedings of the 6th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation*. BuildSys '19. Association for Computing Machinery, New York, NY, USA, pp. 306–315. DOI: 10.1145/3360322.3360845.

Van Le, D., R. Wang, Y. Liu, R. Tan, Y.-W. Wong, and Y. Wen (2020). "Deep Reinforcement Learning for Tropical Air Free-Cooled Data Center Control". arXiv: 2012.06834 [cs, eess].

VanGilder, J., C. Healey, Z. Pardey, and X. Zhang (2013). "A compact server model for transient data center simulations". In: vol. 119. ASHRAE Transactions PART 2, pp. 358–370.

VanGilder, J. W., C. M. Healey, M. Condor, W. Tian, and Q. Menusier (2018). "A Compact Cooling-System Model for Transient Data Center Simulations". In: *2018 17th IEEE Intersociety Conference on Thermal and Thermomechanical Phenomena in Electronic Systems (ITherm)*, pp. 707–715. DOI: 10.1109/ITHERM.2018.8419515.

Van Hasselt, H. (2010). "Double Q-learning". In: *Advances in Neural Information Processing Systems*. Vol. 23. Curran Associates, Inc.

Van Hasselt, H., A. Guez, and D. Silver (2016). "Deep Reinforcement Learning with Double Q-learning". *Proceedings of the AAAI conference on artificial intelligence* **30**:1.

Van Hasselt, H. and M. A. Wiering (2009). "Using continuous action spaces to solve discrete problems". In: *2009 International Joint Conference on Neural Networks*. IEEE, Atlanta, Ga, USA, pp. 1149–1156. DOI: 10.1109/IJCNN.2009.5178745.

Van Heeswijk, W. J. A. (2022). *Natural Policy Gradients In Reinforcement Learning Explained*. arXiv: 2209.01820 [cs, math]. preprint.

Wan, J., Y. Duan, X. Gui, C. Liu, L. Li, and Z. Ma (2023). "SafeCool: Safe and Energy-Efficient Cooling Management in Data Centers With Model-Based Reinforcement Learning". *IEEE Transactions on Emerging Topics in Computational Intelligence*, pp. 1–15. DOI: 10.1109/TETCI.2023.3234545.

Wang, R., X. Zhang, X. Zhou, Y. Wen, and R. Tan (2022). "Toward Physics-Guided Safe Deep Reinforcement Learning for Green Data Center Cooling Control". In: *2022 ACM/IEEE 13th International Conference on Cyber-Physical Systems (ICCPS)*. IEEE, Milano, Italy, pp. 159–169. DOI: 10.1109/ICCPS54341.2022.00021.

Wang, W. and G. Casale (2014). "Evaluating weighted round robin load balancing for cloud web services". In: *2014 16th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pp. 393–400. DOI: 10.1109/SYNASC.2014.59.

Wang, Y., H. Liu, W. Zheng, Y. Xia, Y. Li, P. Chen, K. Guo, and H. Xie (2019). "Multi-Objective Workflow Scheduling With Deep-Q-Network-Based Multi-Agent Reinforcement Learning". *IEEE Access* **7**, pp. 39974–39982. DOI: 10.1109/ACCESS.2019.2902846.

Wang, Y.-T. and Morris (1985). "Load sharing in distributed systems". *IEEE Transactions on Computers* **C-34**:3, pp. 204–217. DOI: 10.1109/TC.1985.1676564.

Watkins, C. J. C. H. and P. Dayan (1992). "Q-learning". *Machine Learning* **8**:3, pp. 279–292. DOI: 10.1007/BF00992698.

Watkins, C. J. C. H. (1989). *Learning from Delayed Rewards*. PhD thesis. King's College, Cambridge United Kingdom.

Wieder, P., J. M. Butler, W. Theilmann, and R. Yahyapour (2011). *Service Level Agreements for Cloud Computing*. Springer Science & Business Media. 368 pp. ISBN: 978-1-4614-1614-2.

Williams, R. J. (1992). "Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning". *Machine learning* **8**, pp. 229–256.

Xie, Z., Z. Lin, J. Li, S. Li, and D. Ye (2022). *Pretraining in Deep Reinforcement Learning: A Survey*. DOI: 10.48550/arXiv.2211.03959. preprint.

Xu, C.-Z., J. Rao, and X. Bu (2012). "URL: A unified reinforcement learning approach for autonomic cloud management". *Journal of Parallel and Distributed Computing* **72**:2, pp. 95–105. DOI: 10.1016/j.jpdc.2011.10.003.

Xu, Y., Y. Zhan, and D. Xu (2017). "Building cost efficient cloud data centers via geographical load balancing". In: *2017 IEEE Symposium on Computers and Communications (ISCC)*, pp. 826–831. DOI: 10.1109/ISCC.2017.8024629.

Yaghmaie, F. A., F. Gustafsson, and L. Ljung (2023). "Linear Quadratic Control Using Model-Free Reinforcement Learning". *IEEE Transactions on Automatic Control* **68**:2, pp. 737–752. DOI: 10.1109/TAC.2022.3145632.

Yang, Z., P. Nguyen, H. Jin, and K. Nahrstedt (2019). "MIRAS: Model-based Reinforcement Learning for Microservice Resource Allocation over Scientific Workflows". In: *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pp. 122–132. DOI: 10.1109/ICDCS.2019.00021.

Young, K., A. Ramesh, L. Kirsch, and J. Schmidhuber (2023). *The Benefits of Model-Based Generalization in Reinforcement Learning*. arXiv: 2211.02222 [cs]. preprint.

Yu, G., P. Chen, and Z. Zheng (2022). "Microscaler: Cost-Effective Scaling for Microservice Applications in the Cloud With an Online Learning Approach". *IEEE Transactions on Cloud Computing* **10**:2, pp. 1100–1116. DOI: 10.1109/TCC.2020.2985352.

Yves, G. and B. Yoshua (2006). "Entropy Regularization". In: Chapelle, O. et al. (Eds.). *Semi-Supervised Learning*. The MIT Press, pp. 151–168. DOI: 10.7551/mitpress/9780262033589.003.0009.

Zhang, H., S. Shao, H. Xu, H. Zou, and C. Tian (2014). "Free cooling of data centers: A review". *Renewable and Sustainable Energy Reviews* **35**, pp. 171–182. DOI: 10.1016/j.rser.2014.04.017.

Zhang, Q., Z. Meng, X. Hong, Y. Zhan, J. Liu, J. Dong, T. Bai, J. Niu, and M. J. Deen (2021). "A survey on data center cooling systems: Technology, power consumption modeling and control strategy optimization". *Journal of Systems Architecture* **119**, p. 102253. DOI: 10.1016/j.sysarc.2021.102253.