Saarland University
Faculty of Mathematics and Computer Science
Department of Computer Science

Masterthesis

Using Reinforcement Learning for Low-Latency High-Throughput Request Scheduling

Submitted by
Safya Alzayat

Submitted
September 2022

Reviewers
Dr. Jonathan Mace
Dr. Antoine Kaufmann

Abstract

Managing resources through scheduling is a core part of distributed systems. Schedulers are designed to optimize for a particular goal such as latency. Many of today's schedulers use heuristics to try to meet those goals, however, designing heuristics that are robust to different workloads and edge cases is a challenging task especially if the scheduler is optimizing for multiple goals. Even though manually tuning the heuristics is possible, it's a tiresome and error-prone process for system administrators.

In this thesis, we designed a reinforcement learning scheduler for model serving systems. We address key practical challenges, such as addressing low latency and high throughput of incoming requests. Our reinforcement learning scheduler is able to achieve the same SLO satisfaction as heuristic schedulers while using fewer resources.

Acknowledgments

First and foremost, I would like to express my gratitude to Allah the Almighty for his many blessings and helping me complete this thesis.

I would like to thank my advisor Jonathan Mace for his continuous support, feedback, and all the time he invested in this project. I would like to thank Antoine Kaufmann for reviewing this thesis. I would also like to thank Arpan Gujarati and Matheus Stolet for their help and support navigating this problem.

Finally, I would like to thank my parents, siblings, and my husband, they kept me going and none of this would have been possible without their support, encouragement, and prayers.

Contents

1 Introduction **9**
- 1.1 Motivation . . . 9
- 1.2 Problem Statement . . . 10
- 1.3 Thesis Organisation . . . 11

2 Background **13**
- 2.1 Scheduling . . . 13
  - 2.1.1 Service Level Objectives . . . 14
- 2.2 Model Serving . . . 14
  - 2.2.1 Clockwork [6] . . . 15
  - 2.2.2 Clockwork Scheduler . . . 15
- 2.3 Reinforcement Learning . . . 17
  - 2.3.1 Model Free vs Model Based . . . 18
  - 2.3.2 Discrete vs Continuous Action Space . . . 18
  - 2.3.3 Deep Reinforcement Learning . . . 19
- 2.4 Related Work . . . 20

3 Design **23**
- 3.1 Challenges of Designing an RL-based scheduler . . . 23
- 3.2 RL Agent Invocation . . . 24
- 3.3 State Space . . . 24
  - 3.3.1 Proposed State Space . . . 26
- 3.4 Action Space . . . 27
- 3.5 Reward Function . . . 28
  - 3.5.1 Sparse Reward Functions . . . 28
  - 3.5.2 Proposed Reward Function . . . 29
- 3.6 Learning Episodes . . . 30
- 3.7 Training the RL Agent . . . 31
- 3.8 Summary . . . 31

4 Implementation **33**
4.1 OpenAI GYM . . . 33
4.2 Stable Baselines . . . 34
4.2.1 Invalid Action Masking . . . 34
4.3 Discrete Event Simulator . . . 34
4.4 Training Setup . . . 35

5 Experiments **37**
5.1 Workload description . . . 37
5.1.1 Low SLOs Workload . . . 37
5.1.2 Azure Workload . . . 37
5.2 Training the RL agent . . . 38
5.3 Testing the RL agent . . . 39
5.3.1 Testing on Azure workload . . . 39
5.3.2 Testing on Low SLOs Workload . . . 40
5.4 Insights . . . 43
5.4.1 Comparing Different Reward Functions . . . 43
5.4.2 Training Workload Matters . . . 44

6 Discussion **47**

7 Conclusion **49**

Chapter 1

Introduction

1.1 Motivation

Scheduling is a fundamental part of systems especially distributed systems. A typical system setting is that there is a queue of requests to be scheduled on available resources such as CPUs or GPUs. These requests can take a different time to execute and in some cases, they might have different priorities. Scheduling is the process of making decisions about the execution order of requests on the available resources to enable efficient resource sharing [1]. Schedulers are usually optimizing for a high-level goal such as average or maximum request latency and heuristics are usually used to achieve this goal.

Network packet scheduling using heuristics is a well-established field where researchers have come up with algorithms that work in practice. There are algorithms that don't enforce fairness across different users such as Round-Robin (RR), Shortest-Job-First (SJF). As mentioned these algorithms do not enforce fairness and thus they can lead to starvation. Other algorithms such as Weighted Fair Queuing [15] and Deficit Round Robin [25] provide fairness.

Similar to network packet scheduling, requests in model serving systems have low latency, arrive at high throughput, and have predictable execution times. However, each request has an associated deadline based on user-defined SLOs after which the response is no longer useful to the user [6]. Moreover, different users can have different SLOs which makes scheduling requests more complicated.

Even though many heuristic algorithms were proposed, designing a heuristic scheduler that can efficiently schedule different workloads and quickly adapts when the workload changes is still a challenging task. There are numerous edge cases that the scheduler needs to handle. The problem of designing a scheduler becomes even more challenging if we want the scheduler to optimize for multiple goals simultaneously. Manually tuning the heuristics to adapt to

changing workloads or goals is possible, however, it's a tedious and error-prone process even for experienced system administrators [28].

An alternative approach to designing heuristic schedulers is to use a learning-based scheduler instead of handcrafting it. Due to the complexity of the problem, enumerating all the possible situations in a dataset to use in supervised learning is absurd. However, reinforcement learning doesn't depend on a pre-fixed dataset for the agent to learn, instead, the agent learns by interacting directly with the environment and observing the consequences of its actions which is passed as the reward. Deep reinforcement learning is an approximation function that can learn to schedule requests, this way we can train an agent to achieve the same goal without the complexity of designing heuristics.

There is some initial related work on using deep reinforcement learning schedulers in high-performance computing and data processing clusters [28, 12] where the reinforcement learning agent schedules jobs. These jobs usually have relatively long execution times (minutes or hours) and they arrive in the system at a low rate (tens per second).

In this thesis, we are focusing on low-latency, high-throughput scheduling which is close to a network packet or OS thread scheduling. In this setting, We have far greater concerns than cluster scheduling as we have to make decisions much more frequently and the state of the system is much larger.

1.2 Problem Statement

Considering the success of schedulers powered by deep reinforcement learning in high-performance computing and data processing clusters, we want to explore the possibility of using deep reinforcement learning in model serving systems where the requests have a low execution time (milliseconds) and arrive at a high rate (tens of thousands of requests per second).

In order to attain a learned schedule that can handle all different cases and goals, plenty of questions first need to be answered, such as can an RL agent learn to schedule requests? Can a learned schedule do better in some edge cases? How do we handle the ever-changing number of users and worker machines in the system?

In this thesis, we first tackle the question of whether a deep reinforcement learning agent can learn to schedule requests in the case of having one scheduling goal in the system such as scheduling requests so that they can satisfy their user-defined latency SLOs. After that, we discuss and provide some evidence of whether a reinforcement learning agent can learn to schedule requests to satisfy multiple scheduling goals.

1.3 Thesis Organisation

The rest of this thesis is organized as follows. Chapter 2 introduces some background about scheduling, model serving, and reinforcement learning. Chapter 3 presents the design of the deep reinforcement learning agent. The frameworks used and the implementation details are discussed in chapter 4. The RL agent is evaluated in chapter 5 and we discuss the possibility of using RL to schedule requests to satisfy multiple goals in chapter 6. In chapter 7, we conclude this thesis.

Chapter 2

Background

2.1 Scheduling

Pinedo [16] defines the scheduling problem as "Scheduling is a decision-making process that is used on a regular basis in many manufacturing and services industries. It deals with the allocation of resources to tasks over given time periods and its goal is to optimize one or more objectives.".

In distributed systems, there is a queue of requests waiting for a resource to be scheduled on. Once the request is allocated a resource it executes for a period of time until it completes, this period of time varies from one request to another. Moreover, different requests can have different priorities.

In most systems, there are multiple resources available for requests to execute on. In some systems, it is preferable to subdivide these resources (e.g. time division multiplexing) and execute several requests concurrently. Executing requests concurrently is often needed so that the system is fully utilized. However, other times we set up the problem so that only one request can consume a resource at any point in time to have predictable execution times.

In the case of having multiple resources requests may have data locality constraints where requests can only be executed on a specific resource, or it may also be the case that requests can execute on any available resource.

The scheduler has the task of choosing which request executes on which resource at what time according to an objective or goal [1]. An example of a simple scheduler is First In First Out (FIFO) where the scheduler executes requests as they arrive at the system. However, this scheduler doesn't enforce fairness can lead to starvation.

Schedulers usually aim to achieve more complex objectives. For example, different requests

might have different deadlines, making it preferable to prioritize the earliest deadline first; other times requests might belong to different users, making it preferable to give an equal share of resources to each user.

2.1.1 Service Level Objectives

A service level objective (SLO) is an agreement between the service provider and the user about a specific metric. The service provider guarantees a certain service level to the users. If the service provider failed to meet the agreed-upon SLO it is considered an SLO violation and there might be a monetary penalty on the service provider.

There are different types of SLOs, the most common SLO is latency SLO where the service provider guarantees that the user's requests will complete within a certain latency. A latency SLO example is 100 ms. In this case, the service provider is guaranteeing that any requests will complete and a response will be returned to the user in less than 100 ms.

Another type of SLOs is throughput SLO where the service provider guarantees that the user will receive a certain throughput. A throughput SLO example is 100 requests per second. In this case, the service provider is guaranteeing that the user will get a throughput of at least 100 r/s (if the user is sending enough requests). There are many other SLO types, however, in this thesis, we focus on latency SLOs.

2.2 Model Serving

Model Serving is a service provided by the cloud, similar to function as a service (FAAS) but instead of functions, machine learning models are hosted in the cloud. The model serving system provides applications or systems with an API to access their hosted models and perform inference requests on them. Exporting this API to the model serving system users facilitates incorporating AI into applications or systems.

From the model serving system point of view, the model serving system users upload their pre-trained deep neural networks (DNN) ahead of time. The users can then submit inference requests to the API provided by the model serving system. The model serving system's back-end manages the users' models and the hardware accelerator resources (GPUs) and provides timely responses to the user's inference requests. Upon receiving an inference request, the model serving system loads the appropriate model into GPU if the model was not already loaded, runs the DNN on the input provided by the user, and returns the resulting output to the user.

There are multiple model serving systems that have been recently proposed such as Tensorflow serving [14], Clipper [4], INFaaS [19], and Clockwork [6]. In this thesis, we implement

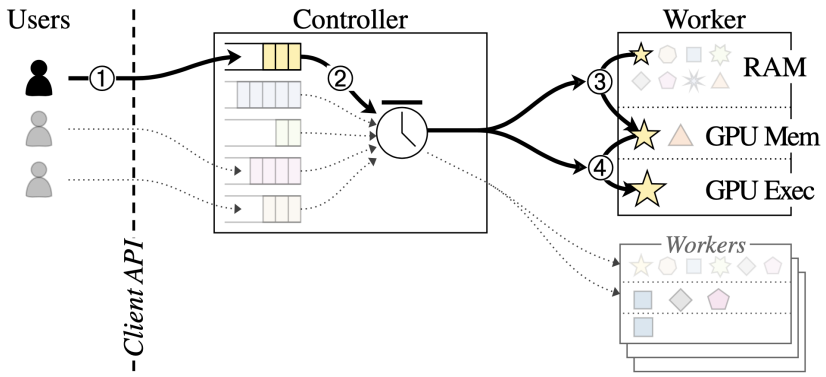

Figure 2.1: Clockwork's Architecture

our scheduler in Clockwork as it provides predictable execution times for inference requests and mitigates tail latency problems which means it can be accurately simulated.

2.2.1 Clockwork [6]

Clockwork is a distributed model serving system that specializes in serving deep neural networks. In figure Figure 2.1 we can see Clockwork's architecture. Clockwork has a central controller and a number of workers. Users send inference requests (1) which are queued on the controller. The controller has a centralized scheduler that maintains a global view of the system state, including all workers, and it's responsible for making decisions about when to execute each request and on which GPU (2). Each worker has a set of DNN models loaded into RAM and the workers have exclusive control over one or more GPUs. To execute a request on a GPU, the scheduler schedules a model to be loaded into GPU memory if it is not loaded (3) and schedules it to be executed on the GPU (4) and the workers will execute the schedules exactly as ordered.

2.2.2 Clockwork Scheduler

Clockwork has a central scheduler that proactively schedules requests on workers. The scheduler utilizes the available system state information, which includes all the requests in the system and the state of all workers and it can predict when requests will complete since DNNs have predictable execution times. The scheduler submits a minimal amount of work to keep the workers busy as delaying the decision will enable better scheduling decisions such as batching requests to the same model.

The scheduling logic is divided into two parts, one for scheduling inference requests on the GPUs and one to load requests from the main memory into the GPU memory. In this thesis,

we are focusing on scheduling inference requests so we will not go into details about the load scheduling logic.

To avoid having idle GPUs, the scheduler schedules an inference whenever the GPU has less than 5ms of outstanding work. To schedule an inference, the scheduler must select a model and a batch size. The scheduler prioritizes larger batch sizes for efficiency. The scheduler decides which model and batch size to schedule based on strategies. A strategy specifies a model, a batch size, and a latest timestamp which is calculated by subtracting the batch execution time from the deadline of the request. Each GPU has a strategy queue that contains strategies for the models loaded on this GPU and the queue is ordered by the latest timestamp. The scheduler continues to dequeue strategies until it finds a valid strategy that has sufficient requests to fill the specified batch size and the latest timestamp has not elapsed yet. Once a valid strategy is found the scheduler schedules an inference on the GPU for the model specified in the strategy and requests are dequeued to fill the batch specified in the strategy.

In summary, Clockworks scheduler makes the following decisions:

- Load/unload a model on a GPU
- Execute an inference request on a GPU for a model that is loaded on that GPU
- Choose the batch size to execute requests in, this way it can execute multiple inference requests from the same model simultaneously on a GPU (The model must be loaded on that GPU)

The scheduler can take any of the previous decisions with the following constraints:

- An inference request can only execute on a GPU that has the appropriate model loaded
- Only one inference request or batch of requests can execute at a time on the GPU
- GPUs are independent and execute concurrently
- Model loading and inference request execution are independent and can execute concurrently on the same GPU

Scheduling requests in this setting is non-trivial. Clockwork hosts multiple models and it strives to be fair across them. There are SLOs per request (possibly different per user) and we want to meet as many SLOs as possible. Lastly, the ability to batch requests gives an interesting trade-off, as we can delay executing requests to form a batch and get better utilization since bigger batches execute in less time per request on the GPU.

Using a simple heuristic such as FIFO would result in low SLO satisfaction in many cases. For example, if we had two models, model one has an execution time of 10ms and an SLO 100ms, and the second model has an execution time of 1ms and an SLO 10ms. If two

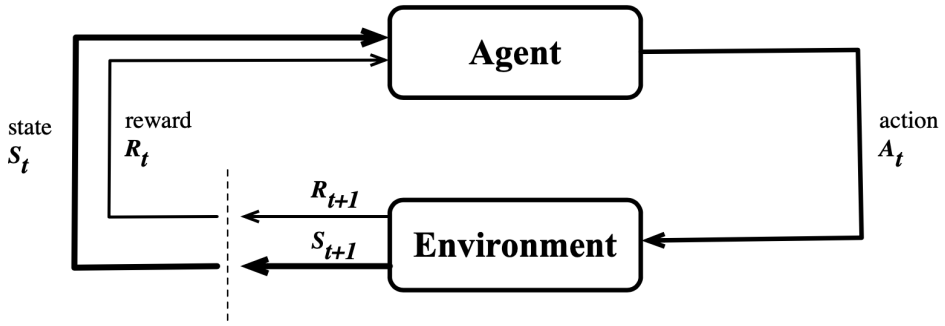

Figure 2.2: The reinforcement learning cycle [28]

requests one for model one and one for model two arrived at time t_0. The FIFO scheduler would either schedule model one or two depending on which model was enqueued first. If model one was scheduled first the request for model two would violate its SLO. On the other hand, Clockwork's scheduler is SLO-aware which means it would always choose model two first as it has an earlier deadline (t_{10}) compare to model one's deadline (t_{100}).

2.3 Reinforcement Learning

Reinforcement Learning (RL) is a machine learning training technique based on rewarding desired behavior and punishing undesired behaviors. In reinforcement learning an agent continuously interacts with an environment by observing the state S_t of the environment and the previous reward R_t and taking an action in the environment A_t, the environment is responsible for evaluating the action and giving a reward R_{t+1} based on how good/bad the action was. The reward $R_t + 1$ is returned to the agent alongside the new state S_{t+1} of the environment after performing the agent's action. This RL cycle is shown in figure 2.2

The goal of reinforcement learning is for the agent to learn to take actions to maximize the discounted rewards collected from the environment. The agent takes actions based on policies. These policies define the probability of taking a certain action in a given state.

Most of the reinforcement learning methods have good convergence on tabular problems. Tabular problems are problems where the state-action space is relatively small so that it can fit in an array or table.

Below is a list of important RL terminology

- State: The state of the environment. For example, in a request scheduling system, we may represent the state of the environment as a vector of queue sizes indicating the number of pending requests in each model queue.

- Action: The decision taken by the agent to be performed in the environment. For example, in a request scheduling system, actions can be a vector with values 0/1 for load/execute orders respectively for each model in the system.
- Reward: A feedback value given to the agent for every action. For example, if a request is completed within its SLO we give the agent a reward of +1.
- Policy: The decision making function. It maps states to actions. An example of a policy can be to choose an action randomly in any given state.
- Value Function: A function that maps states to real numbers. The value of a state is the expected cumulative reward achieved by starting from this state and following a specific policy.
- Model: The agent's view of the environment. However, not all RL agents use a model of the environment.

2.3.1 Model Free vs Model Based

Reinforcement learning algorithms can be either model free or model based. These terms refer to whether the agent builds a model of the environment and uses it to predict the environment's response. Model free algorithms learn by using real samples from the environment. On the other hand, model based algorithms use predictions of the next state and reward to calculate optimal actions.

Model based algorithms allow the agent to plan by thinking ahead since it can accurately predict the next state and reward if it performs a certain action, however, they are computationally more expensive and the agent will only be as good as the model learned. This becomes a problem when the environment is complex and learning a model for it becomes tricky.

2.3.2 Discrete vs Continuous Action Space

Environments can have either a discrete action space or a continuous action space. In discrete action spaces, there is a finite number of actions that the agent can take. For example, deciding which GPU to schedule requests on is a discrete action space. On the other hand in continuous action space environments, the actions are real-valued vectors. An example of continuous action spaces is when the agent needs to decide the speed or the angle of an object.

The action space type influences which RL algorithm can be used since not all algorithms support both action spaces.

2.3.3 Deep Reinforcement Learning

Deep reinforcement learning combines reinforcement learning with deep learning. In deep reinforcement learning, a deep neural network (DNN) is used as the policy to estimate the probability of taking a specific action given a state. The main reason for using a deep neural network is for problems that have an enormous state space which makes memorizing all the states in a tabular form infeasible [10].

By incorporating deep learning in reinforcement learning, reinforcement learning is now able to solve more complicated tasks with lower prior knowledge because of the DNN's ability to learn different levels of abstractions from data.

Deep reinforcement learning has been used in many different fields such as games [23], robotics [8, 27], video streaming [11], network traffic optimizations [3], and much more. There are many different deep reinforcement learning algorithms that have been proposed in the past few years such as DQN [13], TRPO [20], PPO [21], and more. PPO provides a better performance rate and convergence compared to other algorithms [21] and it was used in related scheduling papers [28].

Proximal Policy Optimisation (PPO)

PPO is a model free algorithm that supports discrete action space so it is suitable for our problem. PPO is an actor critic method where there are two main models an actor and a critic. The actor chooses an action for the agent based on the current policy and the critic evaluates the chosen action and updates the policy parameters based on how well the action was. Both the actor and critic are neural networks.

The actor takes as an input the state and outputs a list of probabilities, one for each action. The critic also takes the state and additionally it takes the reward given by the environment and outputs an estimation for the value of the state which is the expected cumulative reward starting from this state till the end following the current policy. The higher the state value the better. Figure 2.3 shows the actor-critic Architecture.

PPO uses the actor network to interact with the environment and saves all the observed experiences in the memory. Once the number of saved experiences reaches a certain pre-determined number the agent updates its networks using the collected experience. The agent iterates over the experience for a pre-determined number of epochs and once the training is over the memory is cleared and the agent resumes interacting with the environment using the updated networks

In order to improve the stability of the actor training, the policy update is limited at each training step so that the new policy is not far from the old policy. This way the policy keeps on improving without risking performance collapse.

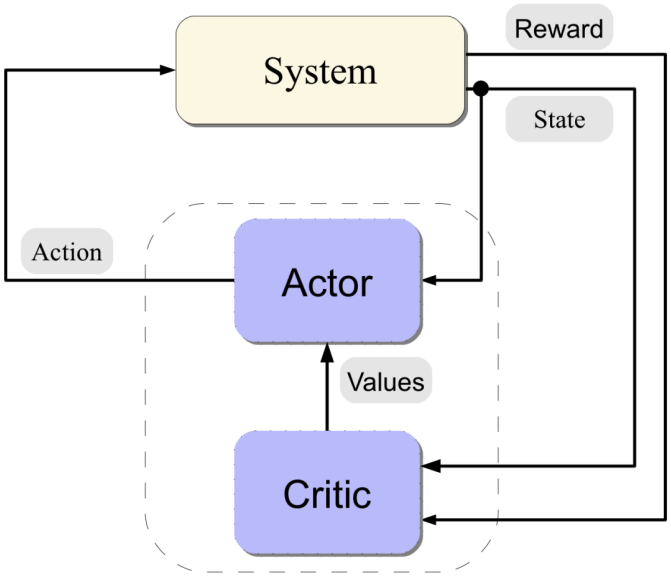

Figure 2.3: Actor-Critic Architecture [26]

2.4 Related Work

In this thesis, we focus on scheduling for model serving using deep reinforcement learning. Deep reinforcement learning has been studied for scheduling in the following contexts.

Zhang et al. [28] propose RLScheduler an automated high-performance computing (HPC) batch job scheduler that uses reinforcement learning. The authors used a kernel-based network as their policy network which is inspired by the kernel function in CNNs. The kernel network is a 3-layer fully connected network.

The RLScheduler implementation is based on the Proximal Policy Optimization (PPO) algorithm. The authors limit RLScheduler to only observe a fixed number (MAX_OBSV_SIZE) of jobs. If there are fewer jobs, they pad the vector with all 0s job vector; if there are more jobs, they use First Come First Server (FCFS) scheduling algorithm to sort all the pending jobs and select the top MAX_OBSV_SIZE jobs. RLScheduler trains its agent in simulation based on job traces.

Mao et al. [12] introduce an RL-based scheduler (Decima) that is responsible for making scheduling decisions for data processing cluster jobs. The authors used graph neural networks to overcome the classical fixed-size neural network input. The authors use the embedding types to represent the jobs in the system. The result of these embeddings is then fed into a softmax function to make the scheduling decision of which stage to schedule and the parallelism level of each stage.

The RL agent is trained in simulation based on job traces. A gradual training method is proposed where the episode length is gradually increased throughout the training process. The agent's initial policy is poor by definition which results in poor scheduling decisions that lead to an overloaded cluster which shouldn't be the case. Letting the agent continue training in an overloaded cluster wastes training time and thus earlier episodes where the policy is still poor are shorter. We use the idea of gradually increasing the episode length in this thesis.

Sheng et al. [24] propose the use of deep reinforcement learning to schedule tasks to VMs on the edge server. The authors are optimizing for the quality of experience of the users which is the ratio between the response time of tasks and the expected response time. The paper uses the REINFORCE algorithm as their deep RL agent. This paper is targeting a different setting where requests have a very low task rate (3 to 7 tasks per second) and the expected response is in a different order of magnitude (5 to 10 seconds)

Although all the above papers leverage reinforcement learning methods for scheduling, they are not solving the scheduling problem in a model serving setting where requests have low latency and high throughput.

Qin et al. [17] propose a swift model serving scheduling framework with a Region-based Reinforcement Learning (RRL) approach. Where they aim to identify the optimal parallelism configuration by estimating the performance of similar configurations with that of the known ones.

Even though the RRL paper uses RL for model serving scheduling it's not scheduling individual requests or deciding which GPU should serve which requests. Instead, they are focusing on systems parameter tuning (parallelism configuration) which is not the focus of this thesis.

Chapter 3

Design

In this thesis, we are interested in exploring the use of reinforcement learning in model serving systems where requests have low execution times and arrive at a high rate.

In realistic systems like Clockwork, scheduling logic can be complicated as requests have low latency, high throughput, and varying workloads. Moreover, in cloud systems, there can be multiple goals, and implementing heuristic schedulers to satisfy all goals is extremely challenging. While manually tuning heuristics to adapt to the changing workloads or goals is possible, it's an error-prone process even for system administrators with high experience [28].

Given the heuristics drawback, we believe deep reinforcement learning to be a good fit for the problem and can overcome the previous challenges as it learns from actual workload without relying on inaccurate assumptions [12].

We present a deep reinforcement learning based scheduler for Clockwork [6]. The RL scheduler is able to schedule inference requests on the available GPUs to satisfy their user-defined SLOs. The scheduler prioritizes using large batch sizes since they use fewer resources and will make the GPU available to serve other requests. We assume that all the DNN models are already loaded on the GPUs by an external module.

In this chapter, we discuss the challenges of designing an RL-based scheduler as well as all the design decisions we took.

3.1 Challenges of Designing an RL-based scheduler

Although RL has promise, there are several challenges we must address in designing an RL scheduler

- The input state space size affects the agent's ability to learn. If the state is too large the agent will not be able to learn a good policy and will take a long time to predict actions.
- The input state space and output actions needs to deal with the dynamic nature of users joining and leaving the system as well as the dynamic number of GPUs available to the system.
- Designing a good reward function is a crucial task as it is responsible for how the agent behaves [5].

3.2 RL Agent Invocation

Choosing the agent invocation frequency is an important design decision that requires balancing tradeoffs. On one hand, invoking the agent too frequently is computationally expensive as each invocation incurs a fixed cost. On the other hand, if the agent is not invoked frequently enough, requests will violate their SLOs.

There are two main approaches to trigger the agent invocation. First, event-based triggers, where the agent is invoked only when a specific predefined event happens, this event can be anything such as a request's arrival or a request completion. Alternatively, the agent triggering can be time-based, where the agent is invoked once every specific period of time.

Both approaches have their own drawbacks, for the event-based method, the main drawback is that finding the event on which the agent will be invoked is challenging. Invoking the agent on each request's arrival would be too expensive in the case of high-throughput workloads where tens of thousands of requests arrive every second. Time-based invocations, on the other hand, have the drawback that the agent might be invoked even when there are no pending requests which might happen in the case of light workloads.

Given the fixed costs of each agent invocation and the fact that we have no control over the arrival rate of requests, we chose to rely on the time-based invocation approach. Periodically invoking the agent guarantees that the number of agent invocations is controlled and does not burden the system.

As a simple future optimization, we can rely on a hybrid agent-invocation model where the periodic invocation is skipped if there are no pending requests to be scheduled.

3.3 State Space

The state is the input passed to the RL agent, the state should provide all the information that the agent needs to make a decision about which action to take.

Choosing what information to include in the state is a crucial task, as there is a trade-off between the state size and the time it takes the agent to make a decision as well as the agent's ability to learn a good policy.

Below is a list of all the information available in the system that we can possibly include in the state.

- Information about each model
  - The execution time for each supported batch size.
    - Not every possible batch size is supported. Choosing to batch a smaller number of requests in a bigger batch is not optimal. The agent could also consider scheduling a smaller batch if the requests are about to violate their SLOs as smaller batches execute in less time, so a request can violate its SLO if scheduled in a large batch but the SLO can be satisfied if the request is scheduled in a smaller batch.
  - On which GPUs is the model loaded
    - Models are not necessarily loaded on all GPUs, and it's not possible to do an inference on a model that is not loaded in the GPU.
- Information about each model queue
  - Number of pending requests on the controller
    - Including the number of pending requests is essential, if there are no pending requests we can't perform an inference for this model, also the number of pending requests is one of the factors for deciding which batch size to execute.
  - The time remaining till the deadline of each request
    - The agent needs to know the time remaining till the deadline. This information is needed so that the agent can know if the request is going to complete in time before its deadline or if it's going to violate its SLO. If the request is going to violate its SLO the agent shouldn't schedule the request as in this case it is wasting resources on a request that is going to violate its SLO in any case.
  - History information such as the number of requests that satisfied and violated their SLOs in the past
- Information about each GPU
  - Pending work on the GPU in milliseconds

    * We need to include the pending work on the GPU so that the agent can make load-balancing decisions. If the GPU has a lot of work scheduled already, it's not going to pick up the newly scheduled request soon. Having a request scheduled on a GPU for a long time can result in the request violating its SLO even though it could have been served by another GPU and could have completed in time to satisfy its SLO.

  - Which models are loaded on the GPU

Including all the aforementioned information for each request, model, and GPU is going to result in an enormous state. The big state will make it very challenging for the agent to learn anything useful and the agent will take a long time to predict an action.

3.3.1 Proposed State Space

The system can have thousands of models and including information about each model is going to result in a big state. The large state space would make taking a scheduling decision computationally expensive. However, different users have different workloads, and most of the models will not have pending requests all the time. Including a lot of inactive models (models that don't have any pending requests) in the state is going to enlarge the state without any added benefit, additionally, having a big state will make the learning process harder on the agent.

To reduce the state size, a potential solution would be to only include information about active models (models that have pending requests), however, the number of active models changes all the time which would result in a variable size input. As we discussed in the background section, most prior work on deep RL uses fixed-size inputs. Although there is work on graph neural networks (and others) on variable-size inputs, there is less prior work. In this work, we primarily investigate fixed-size input as a starting point.

To address this challenge we instead include a small fixed number of models in the state. Models that are included in the state are chosen based on which models are active and have requests that are expiring soon. If the state includes more models than those currently active in the system the state is padded.

Active Models in the state can still have a large number of pending requests, moreover, this number is not constant across models. Including fine-grained information about every request would still result in a large state space, and it would require a dynamic representation of the state space.

To address this challenge, we instead include the number of requests pending for each model and information about the oldest request for each model i.e. the request that will expire the soonest. The information we need to include is the laxity of the request, that is the amount

of time left till the request needs to start its execution or else it will violate its deadline. $Laxity = Deadline - Current\_time - Execution\_time$.

Finally, the number of GPUs in the system is not fixed since it depends on the available resources, some GPUs may be down for maintenance or reserved for other systems. Including information about every GPU in the system will result in a variable size input to the DNN, To overcome this, we include information about the GPU that the agent is making a decision for. This way we are always including only one GPU information and information about a fixed number of models, thus the state size is always fixed.

The new compressed state passed to the agent is as follows:

- Information about N fixed number of models
  - The number of pending requests for each model (when padding the state we use 0)
  - The laxity of the oldest request for each model (when padding the state we use 1000 which means it's not urgent)
- Information about the GPU we are doing a scheduling decision for
  - The amount of pending work in milliseconds

Given this new state, the environment needs to maintain a mapping of the models passed in the state to the agent and their ids in the environment so that we can carry out the actions that the agent takes on the correct models.

3.4 Action Space

The action is the output of the Reinforcement learning agent, it's the decision that the agent decided on according to its policy.

In the Clockwork setting, the scheduler which in our case is the RL agent is responsible for choosing a model to do an inference for and choosing the GPU the inference is going to be executed on. The agent is also responsible for choosing the batch size in which the requests are going to be executed in i.e. the number of requests that are going to be executed simultaneously in a batch on the GPU.

Moreover, sometimes it can be resource efficient to wait and not schedule requests so that more requests can arrive and be executed in a batch. To be able to achieve this we give the agent an option to take the action of waiting and not scheduling anything so that it can wait and make better scheduling decisions at the next scheduling point.

There are two different design choices that we can do.

- The agent takes actions for all GPUs at the same time.
- The agent takes actions for one GPU at a time.

We chose the second design where the agent takes actions for one GPU at a time since the number of GPUs can change over time due to GPUs being down for maintenance or reserved for other jobs. To overcome the varying number of GPUs we make the agent take a decision for one GPU at a time so that the DNN output size is fixed.

To recap, for each GPU in the system the agent makes a decision for each model passed in the state. The agent takes one action from each of the following categories.

- Null, Infer. The agent takes null if it chose to skip scheduling that model and infer if it chose to execute some requests from that model.
- Batch size. The agent chooses the number of requests to be scheduled in a batch. The supported batch sizes are 1,2,4,8,16.

3.5 Reward Function

The reward function plays a major role in whether the reinforcement learning agent learns the assigned task. A bad reward function can make the agent learn the wrong thing, it can make it avoid taking certain actions, or it can make the agent take random actions. The reward function needs to be correlated to the action taken by the agent so that the agent can develop a policy that will take actions to maximize the reward.

3.5.1 Sparse Reward Functions

Designing the reward function can be challenging depending on the complexity of the task given to the agent. The challenging part comes from the fact that some tasks require multiple actions from the agent to complete, and we can only judge whether the agent was successful or not after those multiple actions, so we don't know what reward to give to the agent. In this case, we can choose to not give any reward until the task is completed but this can result in a very sparse reward function.

Sparse reward functions are problematic for two main reasons. First, it takes the agent a long time to learn anything useful, as the agent can take multiple actions without any reward. The second more important problem is that the agent is faced with a credit assignment problem as it can't easily know which action resulted in this reward since there is a long sequence of actions leading to the reward.

An example to show why sparse rewards pose a challenge in our scenario: the task given to the agent is to schedule all requests arriving in the system such that they don't violate a 100

ms SLO. If we were to only give rewards once a request successfully finishes or if it violates its SLO, in the first 100 ms the agent may not get any negative rewards if it didn't schedule any request and if it got a positive reward it will have trouble associating this reward with the action that caused it.

3.5.2 Proposed Reward Function

In order to have a more dense reward function, we need to estimate a reward for every action the agent takes. Based on our estimation the agent is given an early reward that indicates how good/bad the action might be and later on after the task completes and we know the correct reward that should have been given to the agent we give the agent another reward to correct the initial early rewards. For example, if the agent was given a positive reward when it scheduled a request and that request violated its SLO we give the agent a negative reward greater than the early positive reward given so that the total reward for that request is negative.

Our reward function is composed of three parts and the unit we use in the reward function is the execution time on the GPU. We chose the GPU execution time as it is the cost of executing the request and we want to reduce wasting resources, it is also proportional to the request size so if we are executing a big model or a big batch size the execution time will be longer. Below we describe the reward.

- GPU reward function
  - Negative reward proportional to the amount of work pending on the worker
    - Without this reward, the agent would overload the worker with requests, and then all the request would violate their SLO while waiting to be executed on the GPU. Having a worker that has requests scheduled for thousands of milliseconds is a recipe for failure as no action the agent takes will result in requests successfully satisfying their SLO (assuming 100 ms SLO)
- Early reward function whenever a request is scheduled
  - Negative reward equal to its execution time on the GPU
    - If the request is going to be executed as part of a batch, each request gets a negative reward equal to the batch execution time divided by the number of requests in the batch
  - Positive reward equal to twice the execution time of batch size 1
    - Giving the early positive reward based on the batch size 1 execution time gives the agent a bigger total reward when the chosen batch size is bigger. If the agent chooses to schedule a bigger batch, the execution time per request

is going to be smaller so it's going to have a smaller negative reward and the positive reward is constant regardless of the batch size so the total reward given to the agent per request is bigger when the batch size is bigger

- Reward correction in case of SLO violation
  - Negative reward equal to three times the early positive reward that was optimistically given to the agent
    - This reward is given so that the accumulated reward given to the agent reflects how well the agent did for the whole task

3.6 Learning Episodes

In order for the agent to learn a task, it needs many training steps. The training step is the process of observing the state and taking an action based on the agent's policy, taking a "step" in the environment is done by applying the action, and returning the next state and reward to the agent. The agent updates its policy according to the reward given. The goal for the agent is to learn the optimal policy that is going to maximize its reward.

During training, the agent can reach an irrecoverable state by making bad decisions and after reaching this state no matter what action the agent takes it doesn't make a difference as it will just fail at completing the task. An example of an irrecoverable state is when the agent schedules thousands of milliseconds of work on the GPU. In this case, no request scheduled on that GPU will meet its SLO (assuming 100ms SLOs).

Restarting the environment gives the agent a chance to learn from its mistakes and try to avoid doing them again. By definition, the RL policies are initially bad and thus the agent makes poor decisions in the early training steps. The idea of restarting the environment and starting a fresh environment is done through episodes where the training steps can be divided into episodes. After each episode, the environment is reset so that the agent can resume learning in a fresh environment.

In episodic tasks such as playing chess, there is a clear end to the task or episode, which is the end of the game, once the game ends you start the next episode by restarting the game. However, in continuous tasks, there is no terminal state, for example, in our system requests will keep arriving at the system and the agent (scheduler) will continue to schedule them and there is no terminal state. Therefore, we choose the number of training steps in each episode.

Given the initially poor RL policies training the agent for a large number of steps in the earlier episodes is a wasted training time. Therefore, we make the earlier episodes shorter and gradually increase the episode length. This idea of gradually increasing the episode

length was initially proposed in the Decima paper (Discussed in section 2.4).

3.7 Training the RL Agent

Training an RL agent requires a large number of interactions with its environment. In order to train the agent offline, we need to be able to simulate the environment or the system that the agent is going to be used in. If the system can't be simulated the agent can train in the target environment directly. However, this approach poses several issues such as long training time and large financial costs.

In a typical system, it's hard to simulate things because there are lots of non-deterministic sources of interference. But one of the key design choices of Clockwork is predictability. This makes it possible to accurately simulate the system and thus be able to train the agent in the simulated system.

The simulator allows us to train and test the RL agent on a variety of experiments that don't have to be seen in real systems and without needing a cluster of GPUs. Moreover, training in simulations is faster and more efficient as we are not using real resources and executing requests. An additional advantage of building a python simulator is that we can easily use the available RL frameworks since they are mostly written in Python.

Designing an RL agent in simulations allows us to focus on designing an agent that can make good scheduling decisions without worrying about how long it takes to make each decision. In simulations, the time freezes while the agent is making a decision so we don't need to worry about requests timing out during that time. This way we can separate the problem of designing an RL agent into two parts: designing an agent that makes good decisions and designing an efficient agent that takes fast decisions. While both parts are important, in this thesis we focused on the first part of designing an agent that makes good decisions.

3.8 Summary

To recap, our proposed RL-based scheduler is invoked in a time-based manner, where we invoke the agent to make a decision every 1 ms. The agent is trained in episodes of gradually increasing length where the episode length is doubled every other episode and we set a maximum episode length.

The state passed to the agent as an input is a vector of length = *the number of models in the state* $\times$ *2 (num pending requests and laxity of oldest request) + 1 (the amount of pending work on the GPU we are making a decision for).* We chose the number of models to be included in the state to be 12 which makes the state vector length = 25. We chose 12 models as this is the maximum number of GPUs we schedule requests on. This number

is not hard-coded nor is it a limitation of our design it's just the largest number we used in our experiments as we were using the clockwork paper experiments as a reference and they only had 12 GPUs available.

The agent makes a decision for each GPU separately and the output actions vector has a length = *number of models in the state* $\times$ *2 (Null/Infer, Batch size)*. In our case, we use 12 models in the state so the actions vector has a length = 24. This allows the agent to make a decision for each model, it can either skip scheduling the model or decide to schedule requests to execute on the GPU we are making a decision for. Additionally, the agent decides the number of requests to be scheduled by choosing the batch size.

As a reward, the agent is given the accumulation of the following rewards

- For every request scheduled the agent is given a reward = $-$ *request execution time (batch execution time/number of requests in the batch).*
- For every request scheduled that is estimated to complete before its deadline the agent is given a reward = *2 * batch size 1 request execution time.*
- For every request that is completed and violated its deadline or timed out the agent is given a reward = $-$ *3 * request execution time*

Chapter 4

Implementation

4.1 OpenAI GYM

Gym [2] is a python library that provides a set of popular reinforcement learning environments that researchers can use to test their reinforcement learning models, this way we can have a way to standardize and benchmark results. Gym also allows custom environment creation so that the users can create a new environment that can be easily used by other researchers.

Gym provides different spaces that we can use to describe the state space and the action space. Below is a description of the spaces we use in this thesis.

- Multi Discrete Space: This space represents the Cartesian product of arbitrary Discrete spaces where there are multiple discrete spaces that the agent can choose one value from each space. Each value in the discrete spaces is one hot encoded so that the agent doesn't assume that there is a relation between the values, for example, there is no comparison relation between model id 1 and model id 2. We use this space to describe our action space since we want the agent to make multiple decisions such as whether to perform an inference or not for each model in the state space and which batch size to choose.

- Box Space: This space represents the Cartesian product of n closed intervals. Using the box space we can specify a lower and upper bound for each dimension we want to include in the space. We use the box space to encode our state space since our state consists of multiple continuous values such as the number of pending requests for each model and the laxity of each model as well as the amount of work pending on the worker.

4.2 Stable Baselines

Stable baselines [18] is a framework that provides a set of reliable implementations of popular reinforcement learning algorithms in PyTorch. In this thesis, we use stable baselines' implementation for the deep reinforcement learning agent.

4.2.1 Invalid Action Masking

Maskable PPO [7] is a deep reinforcement learning algorithm provided by stable baselines. Maskable PPO is an implementation of invalid action masking for the Proximal Policy Optimisation (PPO) algorithm [21]. Using this implementation the environment needs to provide a mask with each state, the mask shows whether each action is valid or not.

We use the invalid action masking due to the fact that for every state in our environment most actions are not valid. For example, we can not perform an inference for a model that does not have any pending requests, moreover, the agent should not choose a bigger batch size than the available requests as there is no point in choosing a batch size 16 for only one request.

There are two alternative solutions other than invalid action masking. One solution would be to allow the agent to choose any action and ignore this action in the environment if it is invalid. The other solution would be to give an instant negative reward for invalid actions so that the agent learns to avoid this action. However, due to the large percentage of invalid actions in our environment, both solutions resulted in poor convergence.

4.3 Discrete Event Simulator

Discrete event simulation is a model that allows us to represent complex systems in a flexible and intuitive way [9]. In discrete event simulators, events are enqueued in a queue and the time moves forward at discrete intervals from one event time to the next event time allowing us to skip the time in between these two events. An example would be if we want to simulate executing a request that takes 10 seconds, we will have an event that marks the start of the execution and another event with time start + 10 seconds to mark the end of the execution. The start event will be dequeued and after that, the end event will be dequeued and the simulation time will jump 10 seconds even though this will take milliseconds in real-time.

We built a discrete event simulator of clockwork in Python. The simulator simulates workloads arriving in the system, and it also simulates requests executing on the GPUs. We validated the simulator by running the clockwork paper experiments.

Below is a list of workloads that are implemented in the simulator

- Closed-Loop workload: One request is sent at the beginning of the simulation (Or at a specified time) and whenever a request finishes its execution and a response is ready another request is generated.
- Fixed rate workload: A fixed number of requests are generated every period of time. For example, In the case of a fixed request rate of 100 requests per second a request is generated every 10 ms.
- Open-Loop Poisson workload: Requests are generated according to the Poisson distribution. This is implemented by calculating when the next request should be generated using NumPy exponential generator.
- Poisson Trace Replay workload: Requests are generated in a Poisson distribution according to a trace that specifies the number of requests that should be generated for every period of time e.g. 1 sec. An example trace can be 100 0 300. This means 100 requests should be generated in the 1st second, no requests in the 2nd second, 300 in the 3rd second, etc.

Different workloads with different request rates can be combined for the same or different users. This variety of workloads allows us to design a wide range of experiments.

4.4 Training Setup

The RL agent training is done in episodes. Each episode is composed of a varying number of training steps where the number of training steps per episode is gradually increased throughout the training process (explained in section 3.6). After each episode the environment is restarted i.e. all workloads are reset, the events in the simulator queue are erased, and the state of the controller and GPUs are reset so that the agent has a fresh environment for the next episode. After every training episode, the agent is evaluated so that we can observe the agent's policy getting better.

The agent is saved periodically so that we can later load the agent and evaluate it on different experiment settings or even continue training it on the same or different workloads if it's needed.

The time taken to train the agent depends on the complexity of the task and how many training steps are needed to converge on a good policy.

Chapter 5

Experiments

5.1 Workload description

We use two workloads to train and test our reinforcement learning agent. All experiments are done in a discrete event simulator of Clockwork

5.1.1 Low SLOs Workload

In this experiment we use the ResNet50 model stats and we vary the number of model instances (m = 12 or 48) the cumulative request rate is also varied (r = 600 r/s, 1200 r/s, or 2400 r/s). We vary the SLO so that we can evaluate the lower limit at which requests can satisfy their SLOs. There are 6 GPUs on which requests can execute on.

For each workload, we evaluate workload satisfaction which is the percentage of requests that satisfied their SLOs. We start with an SLO of 3 ms which is a very tight SLO as ResNet50 has an execution latency of 2.7 ms, and we continually double the SLO till it reaches 96 ms which is a relaxed SLO. Each model receives requests in Poisson inter-arrival time distribution, and all models are independent so requests cannot be batched across models.

5.1.2 Azure Workload

In this experiment, we simulate 12 GPUs and replay a real workload trace of Microsoft Azure Functions (MAF) [22]. The trace records approximately 46,000 function workloads. We use 61 different DNN models and we duplicate each model 66 times, resulting in a total of 4,026 model instances. We configure all models to have a 100 ms SLO. We tune the traces to have a request rate of approximately 9500 r/s that is distributed across models.

5.2 Training the RL agent

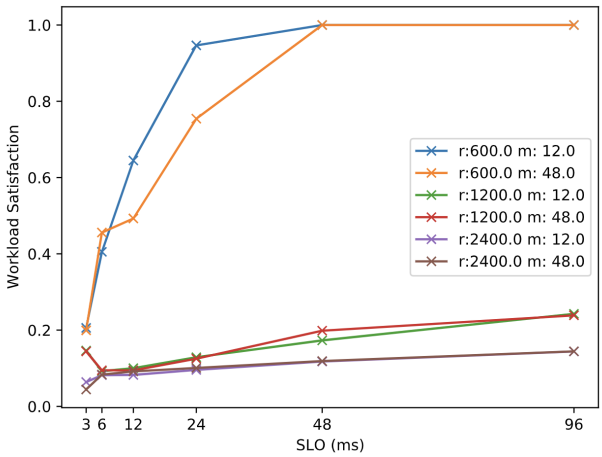

Figure 5.1: Random RL.

In order to train the RL agent we first need to find a problem that is not trivial that a random agent can do well. This way we can observe if the agent is making random decisions or if it is indeed learning to schedule requests. In figure 5.1 we ran a random agent on the low SLOs workloads which contains six different workloads varying the request rate (600 r/s, 1200 r/s and 2400 r/s) and the number of models (12 and 48). In this experiment, there are six GPUs available to schedule requests on. On the x-axis, we vary the SLO of each request between 3 ms SLO and 96 ms SLO. On the y-axis, we report the percentage of requests that satisfied their SLO. We ran this experiment since it contains different workloads some easier to schedule than others. From figure 5.1 we see that workloads with a rate of 600 r/s are so trivial that the random agent can successfully schedule them when the SLO is relaxed enough. However, heavier workloads are more complicated and need some logic so that they can satisfy their SLOs.

We chose a harder workload for the agent to train on, to be precise we trained the agent on the workload that has a request rate of 2400 r/s distributed among 48 models and specified the SLO of all models to be 24 ms. Figure 5.2 shows how the agent progresses during the training phase. The agent is tested after each training episode to measure the percentage of requests that satisfied their SLO. In figure 5.2 we can see that the agent was able to converge to a good policy that can schedule this workload in around 20 episodes. As mentioned in 3.6 episodes start with a small number of learning steps and they gradually increase in length, in this case, they start with 3000 learning steps and double every other episode with a maximum episode size of 60000.

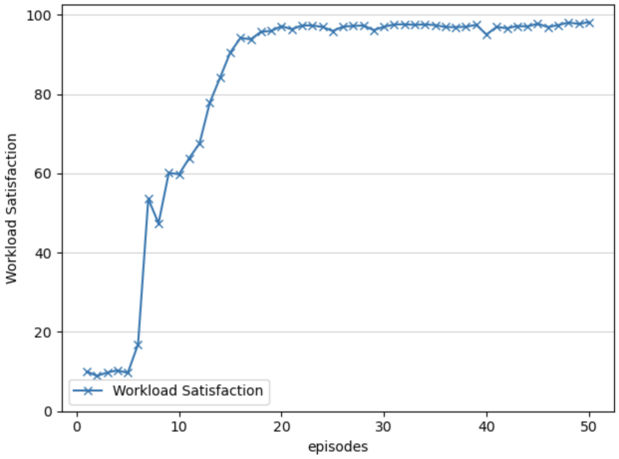

Figure 5.2: Training RL.

5.3 Testing the RL agent

In order to test the RL agent, we ran the trained agent from section 5.2 on different workloads. It's important the agent can schedule workloads different than the one it was trained on so that we can judge its generalization powers.

5.3.1 Testing on Azure workload

First, we tested the agent on the azure workload. In this workload, we replayed the MAF workload explained in 5.1.2 and we check to see if the trained agent is able to schedule

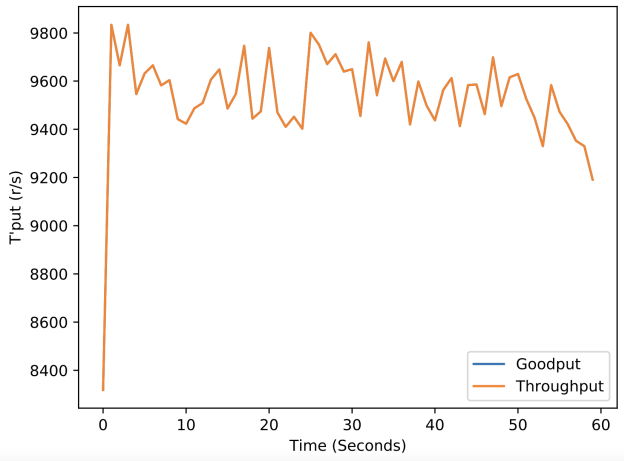

Figure 5.3: Testing on Azure Workload.

requests on the available GPUs in a way that can satisfy their SLOs. In this workload, the agent is scheduling a higher request rate, larger number of models, and scheduling requests on more GPUs than it was trained on. In figure 5.3 we can see that both the throughput and goodput (Number of requests that satisfied their SLOs per second) lines are identical which means that the agent was able to successfully schedule all the requests within their SLO.

In figure 5.4 we plot the average batch size that was chosen by the agent to schedule requests in. To compare the batch size chosen by the RL agent to the batch size chosen by the Clockwork heuristic scheduler, we ran the Clockwork scheduler on the same azure workload (Results are similar to figure 5.3) and plotted the average batch size chosen by the heuristic scheduler in figure 5.5. Comparing the batch size chosen by the agent 5.4 and the batch size chosen by the heuristic scheduler 5.5, we can see that the agent had a higher batch size than the heuristic scheduler 5.5. Choosing a higher average batch size means that the RL agent is using fewer resources on average than the heuristic scheduler.

5.3.2 Testing on Low SLOs Workload

Finally, we tested the trained RL agent in section 5.2 on the low SLOs workloads (described in 5.1.1 and 5.2) so that we can see how well can the agent schedule requests on different workloads with different SLOs than the one it was trained on.

Figure 5.6 shows that the agent was able to schedule requests in a way that satisfied the more relaxed SLO of 96ms for all workloads and it was able to satisfy most of the requests for SLO 24 ms and 48 ms. However, we can see that the agent was not able to get to 100% SLO satisfaction for any of the workloads with SLO 3ms and 6ms. The agent also had worse

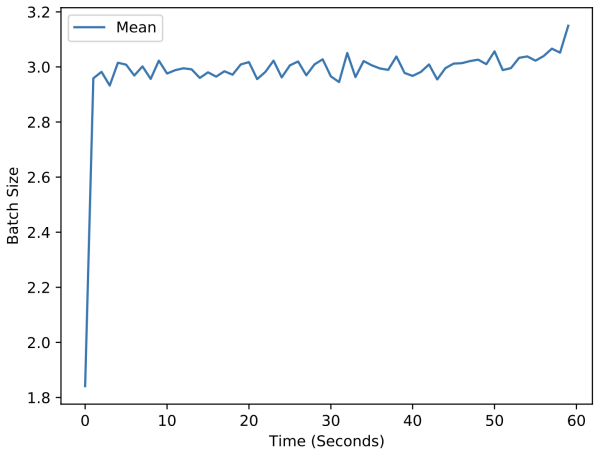

Figure 5.4: RL Batch Size.

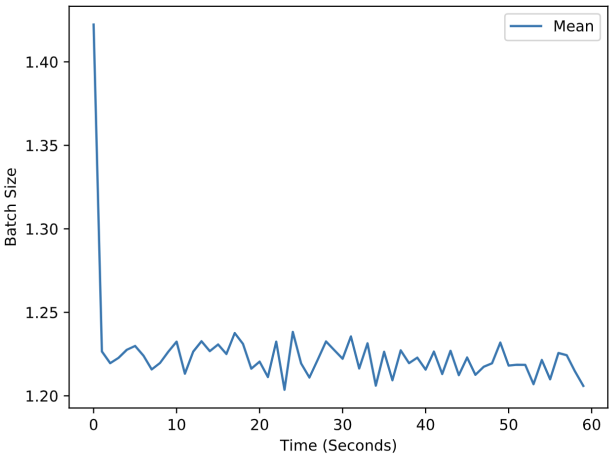

Figure 5.5: Heuristics Batch Size.

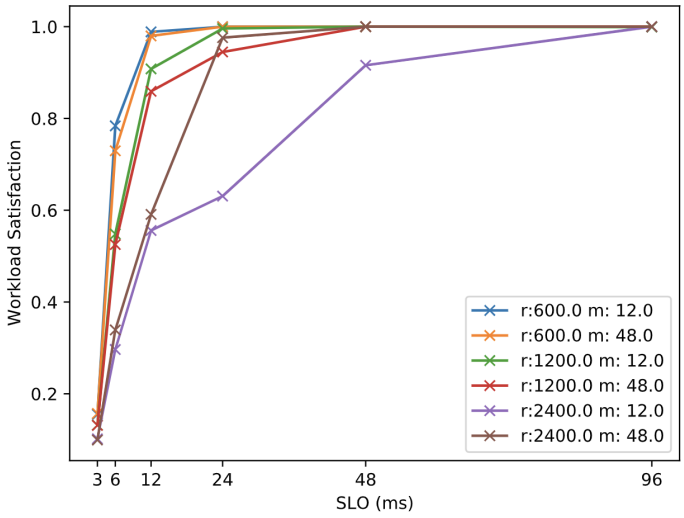

Figure 5.6: Testing RL on Low SLOs Workload.

results on the purple workload with a request rate 2400 r/s across 12 models than the brown workload with a request rate 2400 r/s across 48 models especially in the case of 24 ms SLO even though the purple workload is easier than the brown one as each model has a higher request rate so there are more batching opportunities. Our intuition here was that this is due to the fact the agent was trained on the brown workload so it was able to perform better on it even though it's harder.

In order to determine if the RL agent is doing well enough, we ran our heuristic baseline which is the Clockwork scheduler (explained in section 2.2.2) on the low SLOs workload (explained in 5.1.1 and 5.2). In figure 5.7 we see that the heuristic scheduler was able to satisfy all the requests for the 48 ms and 96 ms SLO, moreover, it was able to get 100% SLO satisfaction for the 600 r/s and 1200 r/s across 12 and 48 models on SLOs as small as 6 ms. This tells us that there is room for improvement and that the agent can be making better scheduling decisions, especially for the 2400 r/s and 12 models workload where there is the biggest SLO satisfaction gap between the agent and the heuristic scheduler (for the 24ms SLO there is a difference of approximately 40% between the agent and the heuristic schedulers).

For that reason, we decided to train the agent for an additional five episodes to tune the agent for each workload. In figure 5.8 we ran the low SLO experiment again with the same settings, however, for each workload, we used the new version of the RL agent that was trained for an additional five episodes on that specific workload.

Figure 5.8 shows that the additional training did indeed improve the agent scheduling decision and it was able to achieve higher SLO satisfaction. Using only five extra training

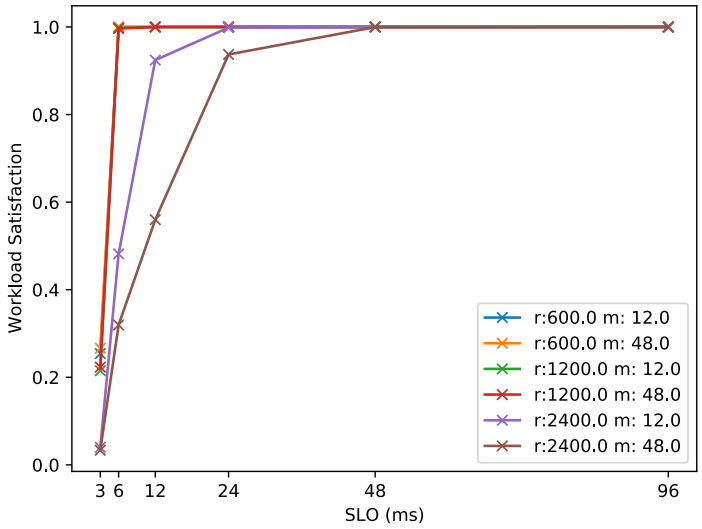

Figure 5.7: Testing Clockwork Scheduler on Low SLOs Workload.

episodes the agent was able to increase the SLO satisfaction of the purple workload (request rate 2400 r/s across 12 models) with SLO 24ms by more than 30% compared to figure 5.6. This indicates that the agent can adapt to new workloads with a few extra episodes.

However, we can see that the fine-tuned agent still makes poor scheduling decisions on extremely tight SLOs. We believe that this might be due to the fact that we are invoking

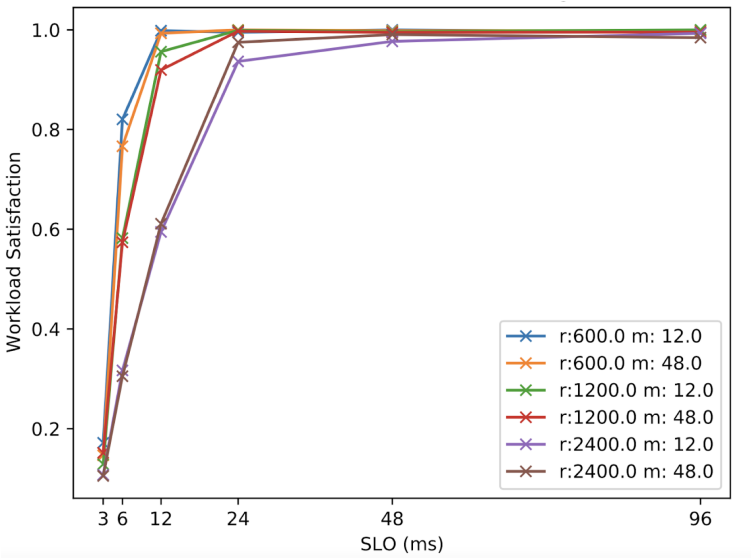

Figure 5.8: Training for additional 5 episodes for each workload.

the agent once every millisecond and this naturally results in it not being able to satisfy requests with very tight SLOs such as 3ms.

5.4 Insights

In this section, we discuss the effect of some of the design choices we took. We demonstrate how different reward functions can have drastically different results. Moreover, we observe how the training data controls the agent's generalization capabilities.

5.4.1 Comparing Different Reward Functions

Designing a good reward function is crucial to the agent's ability to learn the given task. In order to demonstrate how the reward function affects the learning capabilities of the agent, we compare three different reward functions.

- Reward function 1: A simple sparse reward function that gives the agent +1 whenever a request completes within its SLO and -1 whenever a request violates its SLO (completing after the SLO deadline or timing out).
- Reward function 2: A dense reward function that doesn't include domain knowledge about GPU overloading. We use a subset of the reward function explained in section 3.5. We include the early reward and the correction reward but not the GPU reward.

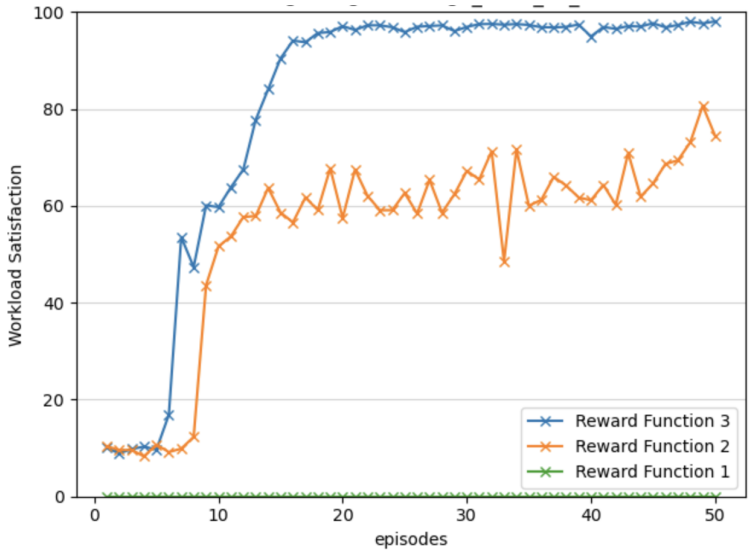

Figure 5.9: Training an RL agent on a workload with 2400 r/s distributed across 48 models with SLO 24 ms using different reward functions

- Reward function 3: Our chosen reward function explained in section 3.5.

In figure 5.9 we train three agents one for each reward function on the same workload described in section 5.2 which has a request rate of 2400 r/s across 48 models with SLO 24 ms and there are six GPUs available to schedule requests on. We can see that our reward function (Reward function 3) is the only reward function that is able to achieve a good SLO satisfaction.

Reward function 1 is too sparse for the agent to associate the reward with the action that caused it, which resulted in the agent not being able to learn anything. Reward function 2 converges to a lower SLO satisfaction than reward function 3 and it takes the agent much longer to reach this policy as first the agent needs to learn not to overload the GPUs and since this information is not directly included in the reward function it takes the agent a long time to learn this. On the other hand, our reward function (Reward function 3) is able to achieve 100 % SLO satisfaction in a small number of episodes.

5.4.2 Training Workload Matters

In section 5.2 we trained the RL agent on a relatively hard workload (2400 r/s across 48 models, SLO 24ms, and 6 GPUs available to scheduler requests on). Training the agent on an easier workload such as (2400 r/s across 48 models and SLO 96ms) results in the agent having worse generalization power.

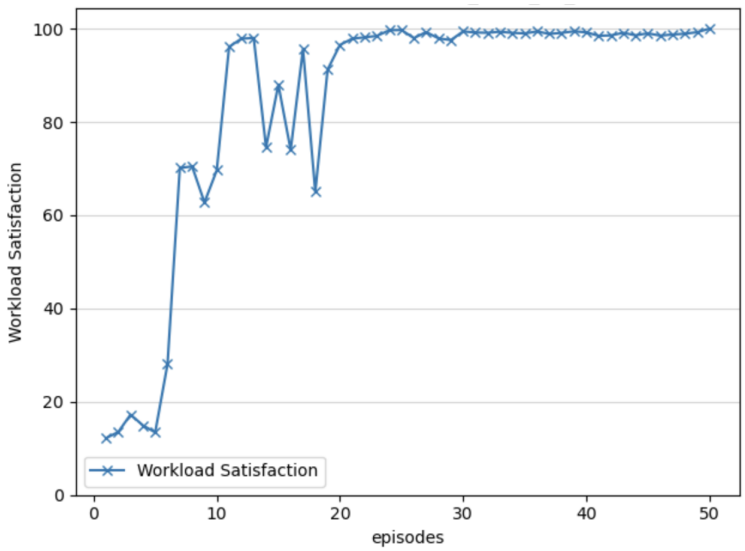

Figure 5.10: Training an RL agent on a workload with 2400 r/s distributed across 48 models with SLO 96 ms

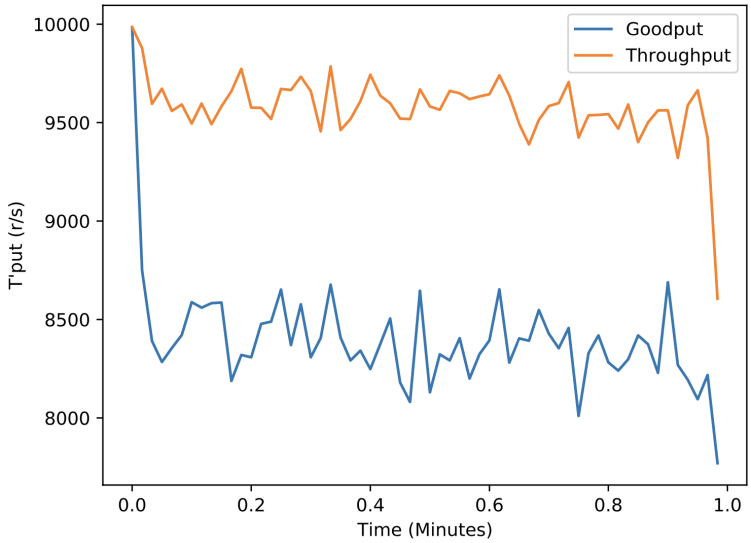

Figure 5.11: Testing an RL agent trained on a request rate 2400 r/s distributed across 48 models with SLO 96 ms on the azure workload

In figure 5.10 we plot the progression of the RL agent during the training phase. After each episode, the agent is tested on the same training workload and the percentage of requests that satisfied their SLO is reported. We can see that the agent is able to converge on a good policy that can schedule all requests in the training workload in less than 30 episodes.

To evaluate the generalization powers of this RL agent we test the agent on the azure workload (explained in section 5.1.2) with a request rate of around 9500 r/s across 4024 models and 12 GPUs available to schedule requests on. In figure 5.11 we see that the RL agent had poor performance and was not able to satisfy the given workload.

This experiment demonstrates the effect of the training data on the agent's policy and its ability to generalize to other workloads. Training the agent on a harder workload as in 5.2 resulted in the agent being able to generalize to the azure workload and satisfy all requests within their SLO as shown in figure 5.3.

Chapter 6

Discussion

So far we have discussed the single SLO scenario. However, scheduling requests that have a single SLO type is still a relatively easy problem. The complexity of scheduling requests becomes more apparent once there are multiple SLOs or multiple goals for different users as designing heuristics for multiple SLOs would be error-prone and tedious.

In this chapter, we provide some preliminary evidence that RL can be used to schedule requests to satisfy multiple SLOs simultaneously. We use artificial SLOs that will show if

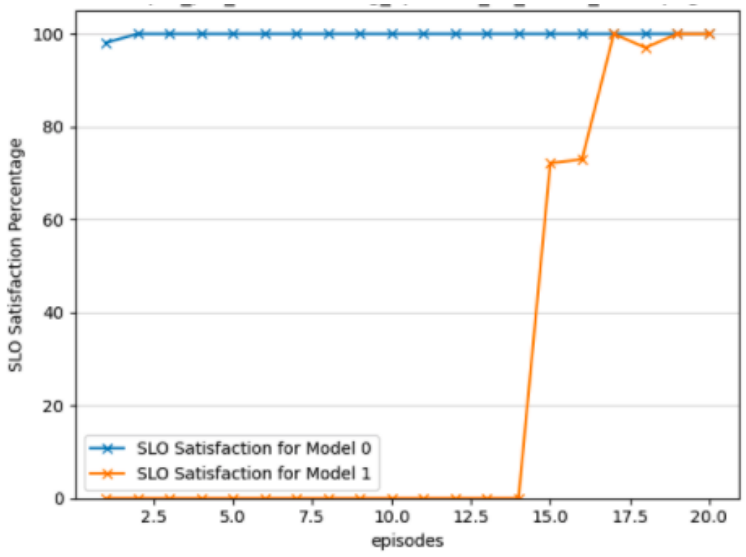

Figure 6.1: Training RL agent on two SLOs. Model 0 has an SLO of receiving a response in less than 100 ms, model 1 has an SLO of receiving a response between 100 ms and 1000 ms

the agent learns the intended behavior regardless of whether the SLO is realistic or not.

We did a synthetic experiment to show that the RL agent can learn to schedule requests with different SLOs. The experiment has two users each with a separate SLO that is mutually exclusive. One SLO is to schedule requests in less than 100 ms and the other SLO is to schedule requests after 100 ms and before 1000 ms, each model has a workload of 150 r/s which can be easily scheduled on one GPU. The RL agent is required to schedule requests from both models so as to satisfy their respective SLOs. In figure 6.1 we can see that model 0 can be trivially scheduled since the agent doesn't need any special knowledge to schedule it and it can be scheduled randomly. On the other hand, we can see that the agent struggled to schedule model 1 in the early training episodes before finally learning that it needs to wait before scheduling requests from model 1.

Training a reinforcement learning agent to schedule requests with different SLO types such as latency and throughput SLOs is a promising direction that has a lot of potential, however, this still needs more research regarding how to construct the different rewards for different SLOs and how to combine them. Due to time constraints, we were not able to completely investigate this problem and we leave this for future work.

Chapter 7

Conclusion

In this thesis, we explored the possibility of using deep reinforcement learning to schedule requests in a model serving system. First, we designed a reinforcement learning agent that is able to schedule requests and satisfy their latency SLO without putting a restriction on the number of models or the number of GPUs. We evaluated the agent on different workloads than what it was trained on and the agent was able to achieve comparable SLO satisfaction to the heuristic scheduler while using fewer resources. Finally, we provided some preliminary evidence that shows that reinforcement learning can be used to schedule requests to satisfy multiple SLOs simultaneously.

Bibliography

[1] Luiz F Bittencourt, Alfredo Goldman, Edmundo RM Madeira, Nelson LS da Fonseca, and Rizos Sakellariou. Scheduling in distributed systems: A cloud computing perspective. *Computer science review*, 30:31–54, 2018.

[2] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[3] Li Chen, Justinas Lingys, Kai Chen, and Feng Liu. Auto: Scaling deep reinforcement learning for datacenter-scale automatic traffic optimization. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 191–205, 2018.

[4] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. Clipper: A {Low-Latency} online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 613–627, 2017.

[5] Jonas Eschmann. Reward function design in reinforcement learning. *Reinforcement Learning Algorithms: Analysis and Applications*, pages 25–33, 2021.

[6] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving {DNNs} like clockwork: Performance predictability from the bottom up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 443–462, 2020.

[7] Shengyi Huang and Santiago Ontañón. A closer look at invalid action masking in policy gradient algorithms. *arXiv preprint arXiv:2006.14171*, 2020.

[8] Dmitry Kalashnikov, Alex Irpan, Peter Pastor, Julian Ibarz, Alexander Herzog, Eric Jang, Deirdre Quillen, Ethan Holly, Mrinal Kalakrishnan, Vincent Vanhoucke, et al. Scalable deep reinforcement learning for vision-based robotic manipulation. In *Conference on Robot Learning*, pages 651–673. PMLR, 2018.

[9] Jonathan Karnon, James Stahl, Alan Brennan, J Jaime Caro, Javier Mar, and Jörgen Möller. Modeling using discrete event simulation: a report of the ispor-smdm modeling good research practices task force–4. *Medical decision making*, 32(5):701–711, 2012.

[10] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM workshop on hot topics in networks*, pages 50–56, 2016.

[11] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. Neural adaptive video streaming with pensieve. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 197–210, 2017.

[12] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM special interest group on data communication*, pages 270–288. 2019.

[13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[14] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. Tensorflow-serving: Flexible, high-performance ml serving. *arXiv preprint arXiv:1712.06139*, 2017.

[15] Abhay K Parekh and Robert G Gallager. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM transactions on networking*, 1(3):344–357, 1993.

[16] Michael Pinedo and Khosrow Hadavi. Scheduling: theory, algorithms and systems development. In *Operations research proceedings 1991*, pages 35–42. Springer, 1992.

[17] Heyang Qin, Syed Zawad, Yanqi Zhou, Lei Yang, Dongfang Zhao, and Feng Yan. Swift machine learning model serving scheduling: a region based reinforcement learning approach. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–23, 2019.

[18] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021.

[19] Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. {INFaaS}: Automated model-less inference serving. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 397–411, 2021.

[20] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897. PMLR, 2015.

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

[22] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 205–218, 2020.

[23] Kun Shao, Zhentao Tang, Yuanheng Zhu, Nannan Li, and Dongbin Zhao. A survey of deep reinforcement learning in video games. *arXiv preprint arXiv:1912.10944*, 2019.

[24] Shuran Sheng, Peng Chen, Zhimin Chen, Lenan Wu, and Yuxuan Yao. Deep reinforcement learning-based task scheduling in iot edge computing. *Sensors*, 21(5):1666, 2021.

[25] Madhavapeddi Shreedhar and George Varghese. Efficient fair queuing using deficit round-robin. *IEEE/ACM Transactions on networking*, 4(3):375–385, 1996.

[26] Csaba Szepesvári. *Algorithms for Reinforcement Learning*, volume 4. 01 2010.

[27] Lei Tai, Giuseppe Paolo, and Ming Liu. Virtual-to-real deep reinforcement learning: Continuous control of mobile robots for mapless navigation. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 31–36. IEEE, 2017.

[28] Di Zhang, Dong Dai, Youbiao He, Forrest Sheng Bao, and Bing Xie. Rlscheduler: an automated hpc batch job scheduler using reinforcement learning. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2020.

Erklärung

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

Statement

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

Einverständniserklärung

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

Declaration of Consent

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

Saarbrücken, 23.09.2022 (Datum/Date) Safya (Unterschrift/Signature)