Dual-Agent Deep Reinforcement Learning Approach to GPU Job Scheduling

Yiming Shao

A dissertation submitted to the Faculty of Graduate Studies

in partial fulfillment of the requirements

for the degree of

Master of Science

Graduate Program in

Electrical Engineering and Computer Science

York University

Toronto, Ontario, Canada

November, 2024

Abstract

Public cloud GPU clusters are increasingly used for distributed deep learning tasks, making the job scheduler critical for enhancing user experiences by minimizing job waiting and completion times. However, scheduling these tasks is inherently complex and NP-hard. Current approaches typically address job scheduling and GPU allocation separately, leading to suboptimal performance. Additionally, deep reinforcement learning (DRL)-based scheduling methods, while flexible, often overlook two challenges. Firstly, they often focus on minimizing the total job completion time across the job sequence and ignore the fairness in waiting times that users experience. Secondly, the speed of distributed training is significantly influenced by GPU communication costs, an often overlooked factor in existing approaches. To address this, we introduce *AttentiveSched*, a DRL-based scheduling framework that simultaneously optimizes job selection and GPU assignment to reach designed optimization objectives.

AttentiveSched considers the cluster topology to make more informed scheduling decisions. The two agents (job and GPU agents) use attention mechanisms to capture the global relationships in the input sequence. Through reward functions that address fairness, job completion time, and communi-

cation costs among GPUs, *AttentiveSched* achieves its objective and yields the best experimental results compared to multiple heuristics-based, meta-heuristics-based, and other DRL-based schedulers on real-world datasets.

Acknowledgements

Life has proven to be much harder than I ever anticipated, even though I have always known since childhood that it would not be easy. During my master's studies, which extended over three years—longer than expected—the world underwent significant changes, such as the COVID-19 pandemic and the rapid advancements in artificial intelligence with the emergence of large-scale models. Simultaneously, my own life transformed in profound ways. I began working on this topic during my final undergraduate year, and it has been a constant that has motivated me to persevere.

I would like to express my deepest gratitude to my supervisor, **Dr. Aijun An**. She showed immense patience and understanding during my most challenging times, especially when I felt overwhelmed. Her guidance and direction were invaluable, enabling me to complete this work independently.

Equally important are my mother and sister, whom I also wish to thank wholeheartedly. They faced difficult times alongside me, consistently encouraging me to overcome life's obstacles and providing me with unwavering, silent support.

I am also grateful to the current and former members of the Data Mining Lab at York University: **Hajer Ayadi**, **Junjie Deng**, **Hossein Pourmod-**

heji, and **Bon Ryu**. I greatly appreciated the relaxed atmosphere in the lab, where we supported each other by sharing new ideas, assisting with implementations, and proofreading.

Special thanks to **IBM Spectrum** and **IBM CAS** for their support and collaboration, which significantly contributed to the success of my research.

Finally, I would like to thank **York University**, where I have spent seven years since arriving in Canada. I have experienced the most significant periods of my life here, and the most substantial changes have occurred on this campus.

This may be the first time I have expressed, especially in writing, my reflections on these past seven years. As life continues, I am moving forward to a new chapter after the completion of my academic journey. I hope, everything can be better.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Recent years have seen rapid advancements in deep learning (DL) and its applications, which are characterized by the use of large training datasets and sophisticated models such as GPT-3 [6] and SAM [7]. These advancements have led to a substantial increase in the use of GPU clusters for distributed and parallel training of deep networks. As a result, GPU job scheduling has become increasingly crucial in managing computational resources for deep learning. However, job scheduling is a well-known NP-hard problem [8]. Finding the optimal schedule can, therefore, be computationally intensive, and it is often not feasible to compute an exact optimal solution within a reasonable time frame.

Due to this challenge, GPU cluster providers have relied on heuristic [9] or meta-heuristic [10] schedulers. These systems either utilize hand-engineered algorithms to select the next job to be scheduled and assign GPUs to the job, or employ search algorithms to find a good solution within a given solution space. However, heuristic methods heavily depend on the design of the heuristics and

are constrained by human design, often leading to sub-optimal solutions. The meta-heuristic methods often takes a long time to find a good solution.

Deep learning has demonstrated its ability to solve complicated real-world problems. To overcome the limitations inherent in human-designed heuristics, DL-based job schedulers [11], supported by Deep Reinforcement Learning (DRL), have emerged. These DRL-based schedulers have shown better performance compared to traditional heuristic methods [11]. However, most of these schedulers focus on minimizing the total job completion time across the job queue [12, 13] and ignore the long and unfair waiting times some users may experience. In addition, most schedulers do not explicitly consider the communication cost among the GPUs assigned to a job. Jobs that use GPUs located far away from each other have longer communication, and thus training time. These delays and prolonged training time negatively impact the user experience. Furthermore, most existing DRL-based GPU schedulers either only apply DRL to job selection, leaving GPU assignment to heuristics [11, 13], or they use a single DRL model to complete both job selection and GPU assignment [14, 15], which is hard to train due to the explosion of the action space that combines job and GPU selections.

1.1 Contributions

In this thesis, we propose a novel DRL-based GPU job scheduling framework that addresses both job selection and GPU allocation. The proposed scheduler uses two collaborative agents: one for job selection and the other for GPU allocation, to reduce the action space. We focus on ensuring fairness in terms of

waiting time standard deviation while maintaining competitive job completion time and makespan (i.e., the total time to finish all the jobs in an episode). We also tested our framework on minimizing makespan. Additionally, we consider the communication cost among GPUs as a factor in the GPU assignment. Our main contributions are as follows.

1. We propose a novel dual-agent DRL-based GPU job scheduling framework. The two agents (one for job selection and the other for GPU assignment) each have a linear action space and are collaboratively trained to reach certain scheduling objectives, with a primary focus on fairness. Additionally, the framework considers the communication costs between assigned GPUs to minimize job running time.

2. A salient feature of *AttentiveSched* is the use of CNNs and Transformers in its agent design. The CNN layer extracts local features in the input observations, while the Transformer extracts global dependencies among the elements in the input. To the best of our knowledge, we are the first to use a Transformer in both job and GPU agents for GPU job scheduling.

3. We demonstrate that *AttentiveSched* outperforms other schedulers in three categories (heuristic, meta-heuristic and DRL-based) by having the lowest sum of waiting time standard deviation and average job completion time on three real datasets. *AttentiveSched* also achieves best or second best makespan among all the evaluated schedulers.

4. We show a successful example of using DRL to solve an NP-hard combinatorial optimization problem.

1.2 Overview

The thesis is organized as follows:

- In Chapter 2, we introduce some background knowledge about job scheduling and different types of schedulers, including heuristic schedulers, meta-heuristic schedulers, and DRL-based schedulers. Additionally, we introduce some basic knowledge about deep reinforcement learning.

- In Chapter 3, we mathematically define the problem that will be described in this thesis, including its definition, the working process, and the objectives. We also prove the NP-hardness of the problem at the end of this section.

- In Chapter 4, we present the architecture of AttentiveSched and investigate all of its components. We formulate the reward function used by both agents and describe the training strategy.

  In Chapter 5, we cover the dataset, baselines, experimental results, ablation studies, and hyperparameter tuning strategy.

- In Chapter 6, we conclude this thesis and discuss some potential future directions.

Chapter 2

Related Works

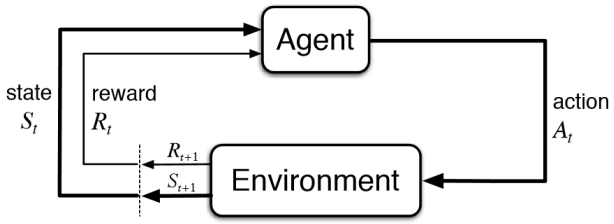

Figure 2.1: [1]Overview of reinforcement learning

2.1 Job Scheduling

Job scheduling [16] has been recognized as one of the most traditional optimization problems. The traditional version involves creating a schedule that defines the order and timing of job executions. The problem usually includes features such as the number of resources, which constrains the selection by the scheduler, job definitions, and a job queue that holds all jobs.

The scheduler typically has several behaviors:

- Ordering the tasks: Determining the sequence in which jobs should be executed based on a set of criteria such as job priority, arrival time, and deadlines.

- Resource Allocation: Assigning available resources to jobs in a manner that jobs may require some specific resources for execution.

As an optimization problem, job scheduling has specific objectives; for example, traditional objectives include minimizing the makespan, maximizing throughput, and minimizing waiting time.

The problem that this thesis aims to solve is a modern variant of job scheduling, which is job-GPU scheduling on a GPU cluster. The objective is to find a scheduler that can efficiently manage the training jobs submitted by users of a GPU cluster.

2.2 GPU Communication Cost

Unlike traditional job scheduling, in the GPU job scheduling problem, the choice of resources can significantly affect the running speed of jobs. This impact is largely due to the communication cost that arises from data transfers between devices, which are limited by the devices' bandwidth capacity [17]. The communication cost is determined by the network topology [18] used in the cluster, such as ring or star topologies. This introduces a problem: if a job requires multiple GPUs to complete its training, there could be an optimal solution that minimizes the communication cost between the selected GPUs or machines, depending on the topology structure.

Finding the optimal GPU selection can significantly reduce training time. Conversely, suboptimal selections can slow down the training, which is undesirable for customers and users of the GPU cluster, as training time is often associated with the rental fees of the cluster.

2.3 Heuristic Schedulers

Heuristic schedulers, some of the earliest types of job schedulers, are guided by specific knowledge about the nature of the jobs or the architecture of the system. Heuristics are generally simpler to implement compared to more modern schedulers. Common heuristic schedulers include algorithms like Shortest Job First, which prioritizes jobs with the shortest length; First In, First Out (FIFO) [19], which prioritizes jobs that arrived earlier; and TETRIS [9], which optimizes resource utilization in the system.

2.3.1 Advantages of Heuristic Schedulers

The primary advantage of heuristic schedulers is their ease of implementation. They can quickly adapt to new working environments, provided that the environment supplies sufficient information required by the heuristics. Furthermore, the inference time of heuristic schedulers is typically much shorter than other types, as they straightforwardly utilize a limited number of job features.

2.3.2 Limitations of Heuristic Schedulers

While heuristic schedulers are quick to use, they do not guarantee that the solution is globally optimal [12]. Typically, they lack the ability to utilize historical decisions or to be sensitive to potential future changes. Another drawback is that they are not sensitive to changes in the environment, such as new job features.

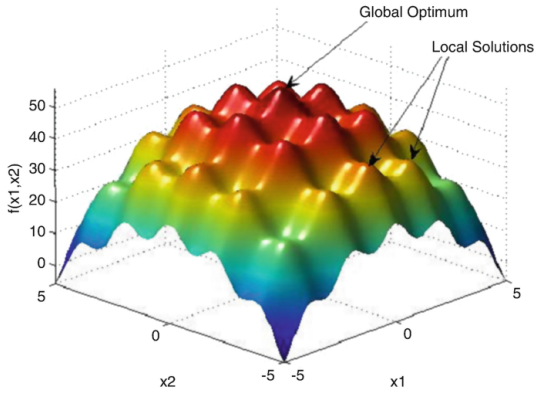

Figure 2.2: [2]Visualization of Meta-Heuristics

2.4 Meta-heuristic Schedulers

Meta-heuristic schedulers represent a sophisticated advancement over heuristic schedulers by aiming to discover optimal solutions within a given environment. Unlike heuristic schedulers, which are constrained by job features or system attributes, meta-heuristics employ a search technique that explores potential combinations to locate the best solutions. As illustrated in Figure 2.2, meta-heuristics such as Genetic Algorithms [20] and Multi-Objective Simulated Annealing (MOSA) [10], iterate through generations or steps to approximate the global optimum.

Different meta-heuristics use different strategies to generate solutions from the input sequence. For example, Multi-Objective Simulated Annealing (MOSA) utilizes a probabilistic method to find local optima by permitting the acceptance of worse solutions. Conversely, Multi Ant Lion Optimizer (MALO) [21], an enhancement of the Ant Lion Optimizer (ALO) [22], leverages random walks and strategic traps to identify the optimal solutions. Each generation or iteration in these meta-heuristics is evaluated through a fitness function specifically designed for the problem.

2.4.1 Advantages of Meta-Heuristic Schedulers

The primary advantage of meta-heuristic schedulers is their robustness against the specific features of jobs, letting them more capable of finding near-optimal solutions than simple heuristics.

2.4.2 Limitations of Meta-Heuristic Schedulers

However, the computational demands of meta-heuristics are significant; the time required to search for solutions is closely tied to the size of the input sequence, and a large dataset may result in unacceptably long search times. Furthermore, like heuristic approaches, meta-heuristics do not inherently adapt to future changes in the system, potentially limiting their effectiveness in dynamic environments.

2.5 DRL-based Schedulers

2.5.1 Deep Reinforcement Learning

Deep Reinforcement Learning (DRL) [23] is a subfield of machine learning that combines deep neural networks with reinforcement learning principles to tackle complex decision-making problems. Reinforcement learning [1] is often modeled as a Markov Decision Process (MDP) [24], historically applied in game theory. As illustrated in Figure 2.1, an MDP is defined by the tuple (S, A, P, R, γ), where:

- S: States representing the environment.
- A: Actions available to the agent.
- P: Transition probability $P(s' \mid s, a)$, which defines the probability of transitioning to state s' from state s after taking action a.
- R: Reward function $R(s, a)$, which evaluates the immediate reward received after transitioning from state s to state s' due to action a.
- γ: Discount factor, which quantifies the relative importance of future rewards compared to immediate rewards.

The primary objective of reinforcement learning is to discover a policy π that maximizes the expected return R given a state s.

Deep Reinforcement Learning (DRL) categorizes agents into two types: on-policy and off-policy. In on-policy reinforcement learning, the agent learns the value of the policy according to which it makes decisions; this policy is both evaluated and improved simultaneously. Proximal Policy Optimization

(PPO) [25] is a prominent example of on-policy learning. In contrast, off-policy reinforcement learning distinguishes between the policy used for collecting data (behavior policy) and the policy that is being optimized (target policy). A well-known example of off-policy learning is Q-learning [26].

As the application of Deep Reinforcement Learning expands, solving combinatorial optimization problems has become a prominent area of application. Combinatorial optimization problems, characterized as NP-hard, present significant challenges. DRL addresses these effectively due to its adaptive nature and ability to learn and optimize from ongoing interactions with the environment. These schedulers can potentially exceed the performance of traditional heuristic and meta-heuristic approaches and adapt to dynamic environments. Earlier implementations of DRL-based job schedulers, such as DeepRM [11] and RLTAPS [14], were trained in relatively small environments, managing short jobs and smaller amount of resources. More recent agents, such as DL^2 [13] and DAS [12], operate within larger-scale environments and incorporate more complex architectural designs compared to earlier agents. Specifically, DL^2 utilizes a Multi-Layer Perceptron (MLP) model, providing enhanced flexibility in GPU selection. Conversely, DAS introduces a dual-agent system, combining a transformer-based agent for job scheduling with a Gaussian process-based agent for machine allocation.

2.5.2 Advantages of DRL-based Schedulers

The advantage of DRL-based schedulers is related to the nature of DRL. They are sensitive to historical data and potential future changes in the environment, allowing them to perform better than traditional scheduling methods.

Moreover, trained models have a short inference time for decision-making, significantly reducing the time needed compared to meta-heuristic search processes.

2.5.3 Limitations of DRL-based Schedulers

Implementing DRL-based schedulers requires careful setup of the simulation environment and precise formulation of the reward functions. DRL-based schedulers have more hyperparameters compared to meta-heuristics to ensure effective training. Although the inference time of these models is short, a significant amount of training time is required before they can be deployed.

Chapter 3

Problem Definition

3.1 Job Queue and Job Characteristics

Consider a job queue $Q(t)$, defined as a dynamic set where t represents discrete time steps starting from $t = 0$. At each time step t, there is a probability $\tau \in [0, 1]$ that a new job will arrive at the queue. $Q(t)$ can contain up to n jobs.

Each job J_i in $Q(t)$ is characterized by a tuple of attributes:

$$(l_i, g_i, t_{w,i}, t_{a,i}, p_i)$$

where:

- $l_i \in \mathbb{Z}^+$ is the estimated length of the job, representing the number of time steps required for its completion, assuming there is no communication cost.
- $g_i \in \mathbb{N}, g_i \geq 1$, indicates the number of GPUs the job requests.

- $t_{w,i}$ is the waiting time of the job, measuring the number of time steps that J_i has been waiting in $Q(t)$ since its arrival.
- $t_{a,i}$ is the arrival time of the job.
- $p_i(t)$ represents the progress of job J_i, at each time step t. The progress increases by $\frac{1}{1+C_i}$ at each time step, where C_i is the communication cost associated with job J_i. A job is removed once $p_i(t) = l_i$.

3.2 GPU Cluster Configuration

The GPU cluster, denoted as S, is organized into R racks, each containing M machines, and each machine is equipped with G GPUs. Therefore, S comprises a total of $R \times M \times G$ GPUs.

Communication Cost Calculation in GPU Clusters

Define the cost coefficients for various levels of communication:

- c_{rack}: Inter-rack communication cost, applied when transferring data between different racks.
- c_{machine}: Inter-machine communication cost within the same rack, incurred when data is transferred between machines that are on the same rack.
- c_{gpu}: Inter-GPU communication cost within the same machine, which occurs when a job is alloacted multiple GPUs within a single machine.

For a given job J_i, the total communication cost C_i depends on the specific GPUs allocated to that job and is calculated as follows:

$$\begin{aligned} C_i = & \, c_{\text{rack}} \cdot (|\delta R_i| - 1) \\ & + c_{\text{machine}} \cdot \sum_{r \in \delta R_i} (|\delta M_{r,i}| - 1) \\ & + c_{\text{gpu}} \cdot \sum_{r \in \delta R_i, m \in \delta M_{r,i}} (|G_{r,m,i}| - 1) \end{aligned} \tag{3.1}$$

where:

- δR_i represents the set of unique racks that contain machines with GPUs allocated to job J_i.
- $\delta M_{r,i}$ denotes the set of unique machines within rack r that are involved in executing job J_i.
- $G_{r,m,i}$ indicates the set of GPUs within machine m, located in rack r, that are assigned to job J_i.

This formulation reflects the hierarchical structure of the GPU cluster and the allocation of resources specific to job J_i, as illustrated in Figure 3.1, which shows the various components and their interconnections.

3.3 Scheduling Process

The scheduling process involves dynamically assigning jobs from $Q(t)$ to the available GPUs based on the requirements of the jobs and the availability within cluster S.

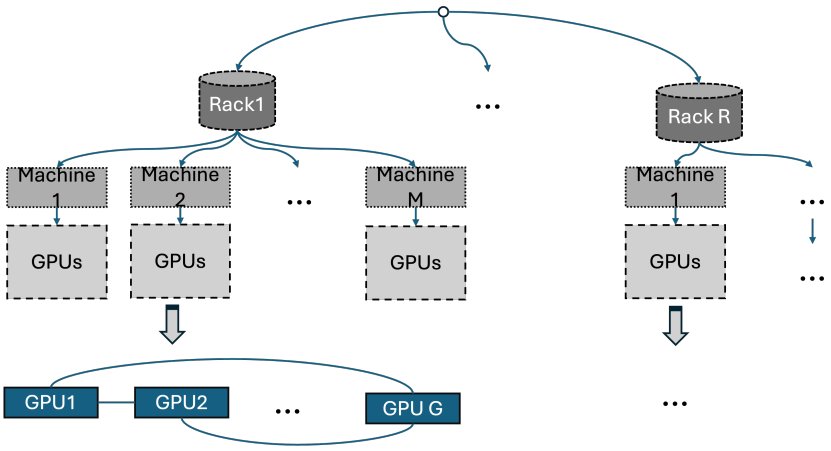

Figure 3.1: GPU Cluster Architecture

Illustration of the GPU cluster showing hierarchical communication paths.

Decision Variables

- $x_{i,t}$: Binary variable, $x_{i,t} = 1$ if job J_i is running at time step t, and $x_{i,t} = 0$ otherwise.
- $y_{r,m,g,i,t}$: Binary variable, $y_{r,m,g,i,t} = 1$ if GPU g in machine m of rack m within S is allocated to a job i at time t, and $y_{r,m,g,i,t} = 0$ otherwise.

Constraints

To effectively manage the GPU resources and ensure the integrity of the scheduling process, the following constraints are enforced:

$$c_{\text{gpu}} < c_{\text{machine}} < c_{\text{rack}} \tag{3.2}$$

$$\sum_{r=1}^{R} \sum_{m=1}^{M} \sum_{g=1}^{G} y_{r,m,g,i,t} = g_i \cdot x_{i,t}, \quad \forall i \text{ in } Q(t), \quad \forall t \tag{3.3}$$

$$\sum_{i=1}^{n}\sum_{r=1}^{R}\sum_{m=1}^{M}\sum_{g=1}^{G} y_{r,m,g,i,t} \leq R \times M \times G, \quad \forall t \tag{3.4}$$

$$\begin{aligned} &\text{If } x_{i,t_s} = 1, \text{ then } x_{i,t} = 1 \text{ for } t > t_s \text{until } p_i = l_i, \\ &\quad \text{at which point } x_{i,t} \text{ becomes } 0. \end{aligned} \tag{3.5}$$

Explanation of Constraints

- Constraint (3.2) ensures that the communication cost between GPUs on the same machine is lower than the cost between GPUs on different machines but within the same rack, which in turn is lower than the communication cost between GPUs on different racks.
- Constraint (3.3) ensures that if job J_i is scheduled at time t, then exactly g_i GPUs are allocated to it. If the job is not scheduled, no GPUs are allocated.
- Constraint (3.4) ensures that the total number of GPUs allocated at any time does not exceed the total number of GPUs available in the cluster.
- Constraint (3.5) ensures that a scheduled job continues to run without interruption once it starts and is removed from the scheduling queue once completed, as indicated by $p_i(t) = l_i$.

3.4 Objective

The scheduling objective is vary by the problem. The major objective covered in this thesis is minimizing the average job completion time while ensuring

fairness. Fairness is achieved by minimizing the variance in waiting times across all jobs, ensuring that jobs experience comparable waiting times.

$$\text{Minimize } (\alpha \cdot \text{Mean}(C) + (1 - \alpha) \cdot \text{SD}(W))$$

where W and C represent the waiting times and job completion times of all jobs, respectively. $\text{SD}(W)$ and $\text{Mean}(C)$ are the standard deviation of waiting times and the average of job completion times, respectively. The standard deviation is used here instead of variance because it has the same unit as the average. α is a weighting factor that balances the importance of minimizing the average waiting time against reducing variability.

3.5 Reduction from MKP to Job-GPU Scheduling

To establish that the Job-GPU Scheduling Problem is NP-hard, we reduce from the Multiple Knapsack Problem (MKP), a well-known NP-hard problem.

Mapping:

- Each *item* i_j in MKP corresponds to a *job* J_j in our scheduling problem. The weight w_j, representing the item's weight, corresponds to the number of GPUs required by the job J_j. The value v_j, representing the item's value, maps to $\frac{1}{n} \cdot jct_j$, where jct_j is the job completion time for job J_j.
- Each *knapsack* k in MKP, characterized by a specific capacity C_k, cor-

responds to a set of available GPUs in a time slot t_k.

- The *capacity* C_k of a knapsack is analogous to the total number of available GPUs G in the time slot t_k.

Objective Equivalence: In MKP, the objective is to maximize the total value of the items placed in knapsacks. This translates in our scheduling problem to minimizing a composite objective function:

$$\text{Minimize } (\alpha \cdot \text{mean}(JCT) + (1 - \alpha) \cdot \text{std}(WT))$$

Rewriting in terms analogous to MKP gives:

$$\alpha \cdot \sum_{k=1}^{m} \sum_{j=1}^{n} \left(\frac{1}{n} \cdot jct_j \right) x_{jk} + (1 - \alpha) \cdot \text{std}(WT)$$

where $v_j = \frac{1}{n} \cdot jct_j$.

Implications and NP-hardness: Since the translated objective includes the MKP components plus an additional term, std(WT), the problem remains at least as hard as MKP. Therefore, the Job-GPU Scheduling Problem is also NP-hard.

Chapter 4

Methodology

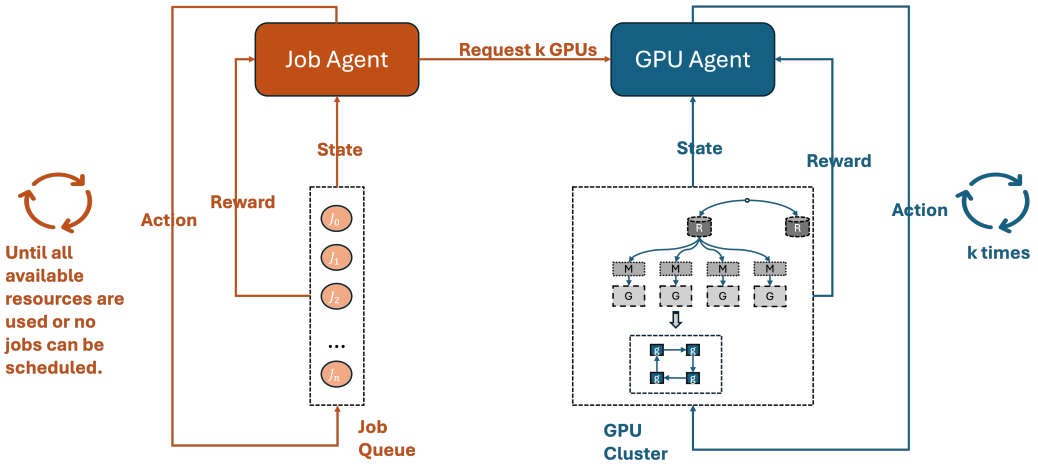

Figure 4.1: An overview of *AttentiveSched*.

AttentiveSched consists of an environment and two agents (the job agent and the GPU agent), as illustrated in Figure 4.1. The environment includes a job queue and a GPU cluster and provides specific states and rewards to the agents based on the actions they take. Both agents involve sequential decision-making, where a single time step is divided into a sequence of sub-steps. At each sub-step of the job agent, it selects a job from the job queue. The GPU

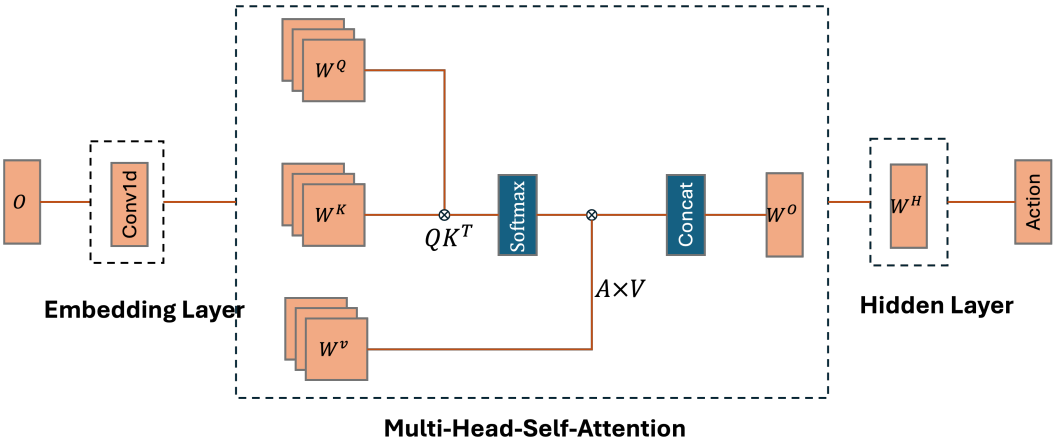

Figure 4.2: The network architecture of *AttentiveSched*

agent then allocates GPUs to the selected job through a sequence of sub-steps, during which it incrementally selects GPUs from the cluster until the number of GPUs required for the job is allocated. This process continues until either all available GPUs are used or all remaining jobs in the queue require a number of GPUs that exceeds the number of available GPUs.

4.1 Observations

Representing the state of the environment, the observations for the job agent and the GPU agent at time step t are:

- **Job Agent Observation** (O_t^j): The job agent's observation is defined as $O_t^j \in \mathbb{R}^{n \times u}$, where n is the number of job slots in the queue, and u represents the number of features of each job. The the ith job in the job queue is represented by a vector $(l_i, g_i, t_{w,i}, t_{a,i}, p_i)$, where l_i is the estimated length (in time steps) of the ith job, g_i indicates the number of GPUs requested by the job, $t_{w,i}$ is the waiting time since job arrival,

$t_{a,i}$ is the arrival time of the job, and $p_i(t)$ represents job progress.

- **GPU Agent Observation** (O_t^g): The GPU agent's observation is defined as $O_g \in \mathbb{R}^{z \times v}$, where z represents the number of GPUs in the cluster, and v denotes the number of features per GPU. Each vector representing a GPU, defined as $\langle r, m, g, s, q, l \rangle$, where r, m, g are the rack ID, machine ID, and GPU ID, to distinguish GPUs in the cluster, s is the status of the GPU, with 1 for available, 0 for occupied (by other jobs), and 0.5 for already selected by the current job (in a sequential process of selecting GPUs for the current job), q reflects the number of GPUs requested by the job currently being processed by the GPU agent, and l represents the expected running time of that job.[1] Note that although we represent an observation as a list or sequence of vectors, the use of the transformer in our agent network structure (described later) "discounts" the ordering in the input by using the self-attention mechanism.

4.2 Actions

AttentiveSched generates actions based on the observations. The process continues to schedule jobs from the queue until all GPUs are utilized or all feasible jobs (those requesting fewer GPUs than are available) have been scheduled. Both agents sequentially select jobs and GPUs. The rationale behind this extension is that, in practice, it is uncertain how many jobs will arrive at the

[1]Each GPU vector contains the information about the selected job (i.e., q and l). Such job information may appear to be redundant in GPU representations. However, putting the job information in each GPU representation helps the embedding layer (to be described later) to evaluate each GPU's suitability against the chosen job, which showed benefits in our preliminary experiments.

queue at each time point (in reinforcement learning, this time interval is represented as a timestep). Therefore, attempting to schedule all possible jobs in each timestep aligns more realistically with practical scenarios.

The reason why we use sequential sub-steps instead of scheduling all feasible jobs or allocating all GPUs within a single step is that the action space would be too large. For example, if a cluster has 32 GPUs and a job requests 8 GPUs, the number of potential actions is $\binom{32}{8}$, which is over 10 million. If the model selects only one GPU at a sub-timestep, the action space at a sub-timestep is reduced to $\binom{32}{1}$, thus making it more manageable.

- **Job Agent Action**: Denoted by a_j, this is the index of a job in the queue, where $a_j \in \{0, 1, \ldots, N\}$, and N is the number of job slots in the queue. Action N represents an idle action. At each time step, the job agent selects one job per sub-step until no further jobs can be scheduled.

- **GPU Agent Action**: For each selected job, the GPU agent outputs a sequence of k actions where k is the number of GPUs requested by the job. Each action is represented by a_g, which is the index of a GPU in the cluster, where $a_g \in \{0, 1, \ldots, Z\}$, and Z is the total number of GPUs. The GPUs are indexed from 0 to $Z - 1$. Action Z indicates an idle action, used when insufficient GPUs are available.

4.3 Model Architecture

Both the job and GPU agents use a transformer as its network structure and take a list of vectors as input. Their transformer structures are the same, but differ in the input and output layers to fit their particular input and actions.

Figure 4.2 shows their network structure. It includes an embedding layer to map each vector in the input list to a higher dimensional vector to allow richer and more detailed representation of the data. After all the vectors in the input sequence are converted, the list of converted vectors are inputted together to a transformer encoder that processes the embedded input. The transformer uses attention to "discount" the ordering of the elements (i.e., the jobs or GPUs) in the input sequence. Positional encoding is not used in our transformer. Finally, a projection layer is used to produce the output from the transformer encoder output.

The transformer model [27] has demonstrated advantages in learning inter-relationships and long dependencies within input data [28], suggesting its potential for solving deep reinforcement learning (DRL) tasks beyond its original domain of natural language processing (NLP). This capability provides confidence in its applicability to scheduling and optimization tasks. The performance of the transformer model compared to other common network structures is further discussed in Section 5.3.

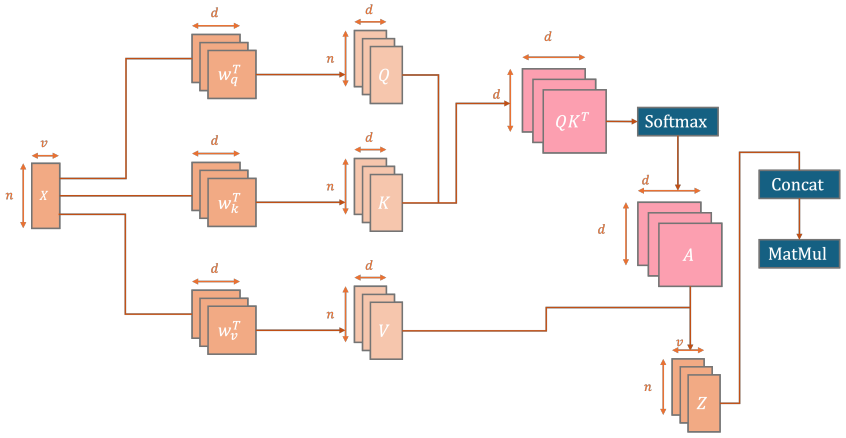

Figure 4.3: The architecture of Multi-Head Self Attention

Embedding Layer

A convolutional **embedding layer** is used as the embedding layer to enhance an input vector's feature representation. We use a one-dimensional convolutional network (Conv1D) [29] as the embedding layer because it can potentially capture the local relationship between features in each job or GPU vector. This layer is described by the following equation:

$$e_i = \text{ReLU}(\text{conv1d}(O_i) \tag{4.1}$$

where O_i represents the ith vector from the observation.The number of neurons in the embedding layer matches the hidden size of the self-attention mechanism in the transformer.

Multihead-Self-Attention Mechanism

The core component of the model is the self-attention mechanism, which allows each item in the input data to attend to all others. Using both a convolutional layer and a transformer in our approach combines the local pattern recognition abilities of CNNs with the global contextual understanding of transformers. The self-attention mechanism of the transformer is applied to process the embedded sequence, e^j, e^c, obtained from the convolutional layer.

In our work, the self-attention mechanism comprises multiple heads [27], each operating in parallel, allowing the model to learn different parts of the input data, as in figure 4.3. For head i, the learnable weights $\mathbf{W}_i^Q, \mathbf{W}_i^K$, and $\mathbf{W}_i^V$ are utilized to derive the query, key, and value of the input by multiplying e^j with these weights as follows:

$$\text{Q}_i = e \times \mathbf{W}_i^Q, \quad \text{K}_i = e \times \mathbf{W}_i^K, \quad \text{V}_i = e \times \mathbf{W}_i^V \tag{4.2}$$

We denote the output as Q_i, K_i, V_i.

Using the calculated variables Q_i, K_i, V_i, we can obtain the attention score for the head i. The computation is as follows:

$$\text{Attention}_i(\text{Q}_i, \text{K}_i, \text{V}_i) = \text{softmax}\left(\frac{\text{Q}_i \times \text{K}_i^T}{\sqrt{d_k}}\right) \times \text{V}_i \tag{4.3}$$

where d_k is the dimensionality of the key vectors, equal to the embedding size in e^j.

After computing the attention score for each head, we concatenate the attention scores from all heads and then pass them by a projection layer, which is a learnable weight, denoted as $\mathbf{W}^O$:

$$\begin{aligned}\text{MultiHead}(\text{Q}, \text{K}, \text{V}) =& \text{Concat}(\text{Attention}_1, \\ & \text{Attention}_2, \ldots, \text{Attention}_h) \times \mathbf{W}^O\end{aligned} \tag{4.4}$$

Hidden Layer

Subsequent to the projection layer, the output is fed into a hidden layer, which serves to further transform the combined attention vectors to match the action space. This layer is defined as follows:

$$\text{Output} = \text{ReLU}(\mathbf{W}^H \times \text{MultiHead}(\text{Q}, \text{K}, \text{V}) + \mathbf{b}^H) \tag{4.5}$$

where $\mathbf{W}^H$ and $\mathbf{b}^H$ are the learnable weights and biases of the hidden layer.

4.4 Rewards

Both agents of *AttentiveSched* can adapt to various scheduling objectives by properly formulating the reward functions. For the job agent in this thesis, the objective is balancing the fairness and ensuring the job completion time. The objective for the GPU agent, which is to minimize communication costs among selected GPUs.

4.4.1 Job Agent Reward: Fairness

Fairness is one of the objectives that job schedulers aim to achieve, yet people have different criteria for evaluating fairness, such as fees [30], and resource utilization [31]. In this thesis, fairness is measured by the standard deviation of jobs' waiting times in the queue [32], as described in Section 3.4. In order to avoid the agent to achieve better fairness but sacrifies the average waiting time, and a job's completion time consists of its waiting time and its running time. Thus, the reward function contains three parts, average job waiting time, the standard deviation of job waiting times, and the average job running time.

Let Q_t denote the job queue at timestep t, containing $|Q_t|$ waiting jobs. Each job $j \in Q_t$ requests a specific number of GPUs, denoted $gpu\_request_j$. The cluster has num_avl_gpus available GPUs.

Due to the sequential nature of job scheduling, the reward $R_{\text{Job}}$is categorized into two cases:

$$R_{\text{Job}} = \begin{cases} 0 & \text{if } \exists j \in Q_t : \text{gpu\_request}_j \leq \text{num\_avl\_gpus}, \\ R_t & \text{otherwise} \end{cases} \tag{4.6}$$

The reward function R_t is formulated as:

$$R_t = -\left(\alpha \cdot \mu_{w,t} + (1-\alpha) \cdot \sigma_{w,t} + \beta \cdot \mu_{r,t}\right) \tag{4.7}$$

where $\mu_{w,t}$ and $\mu_{r,t}$ are the average waiting time and average running time of all the jobs that have been submitted to the job queue by time t (both together measuring the job completion time), $\sigma_{w,t}$ is the standard deviation of the waiting times of these jobs (measuring the fairness), α is a weight factor that balances the trade-off between average waiting time and fairness, and β controls the emphasis on minimizing job running time.

Design Rationale $\sigma_{w,t}$ and $\mu_{r,t}$ are objectives that are to be minimized in the task. $\mu_{w,t}$ is added to the reward function to avoid job delays, since good fairness can also be achieved by delaying all jobs. α is used to balance between average waiting time and fairness, while β is used to balance between the term $\alpha \cdot \mu_{w,t} + (1-\alpha) \cdot \sigma_{w,t}$ and average job running time.

4.4.2 GPU Agent Reward

The GPU agent uses the same reward funcion that formulated in [33], besides the communication cost is calculated differently. Let j denote a selected job that requests k GPUs. The GPU agent executes a sequence of k sub-steps, each involving the selection of one GPU. The sub-step reward is 0, and the time-step reward is based on the communication cost. The reward is defined

as follows:

$$R_{GPU} = \begin{cases} 0 & \text{if } (n < k) \text{ or } (k = 1), \\ 1 - \frac{\text{Com\_cost}(j) - \min_k}{\max_k - \min_k} & \text{otherwise} \end{cases} \tag{4.8}$$

where n is the number of GPUs that have been selected so far by the agent, Com_cost(j) is the communication cost for the GPUs selected so far for job j, and $\min_k$ and $\max_k$ are the minimum and maximum possible communication costs that can be achieved by selecting k GPUs in the cluster, used for normalization purposes. The communication cost calculation is detailed in Equation 3.1 in Chapter3.

The computation for $\min_k$ and $\max_k$ is similar and can be performed offline in advance. For instance, when k is 2, the best cost is c_{gg}, achieved by using two GPUs in the same machine, and the worst cost is c_{rr}, occurring when each GPU is assigned to a different rack.

Design Rationale The reward function is designed to minimize the communication cost between the selected GPUs over the previous k sub-steps. With knowledge of the expected running time and the number of GPUs requested by the selected job, the agent is encouraged to select GPUs that are topologically close to each other.

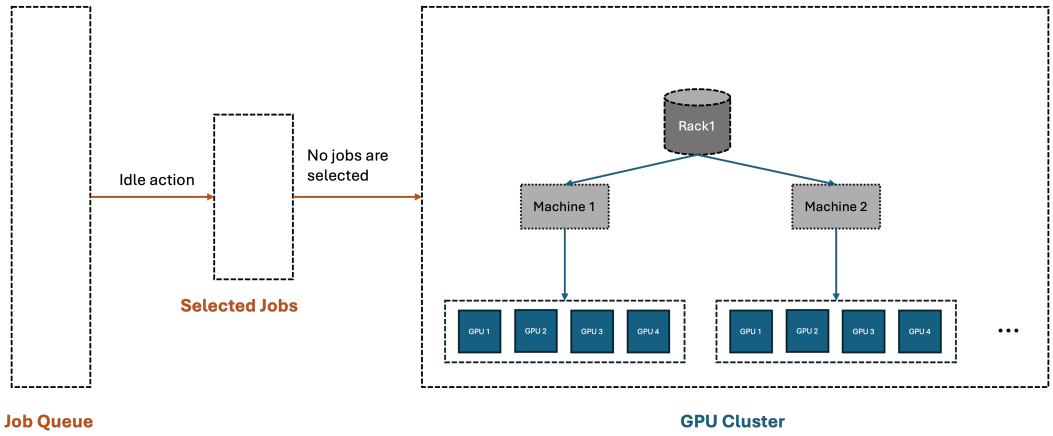

Figure 4.4: At time-step t_0, the job queue is empty.

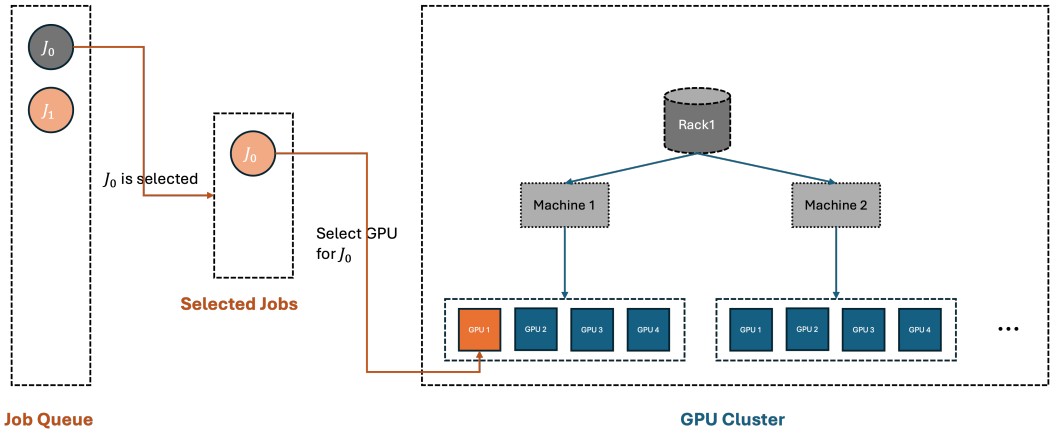

Figure 4.5: At time-step t_1, two jobs arrive, each requesting 2 GPUs. The job agent selects J_0, and the GPU agent assigns the first GPU to J_0.

4.5 Sequential Operation of Job Scheduling and GPU Allocation

The operation of our agents within the *AttentiveSched* framework is designed to handle job scheduling and GPU allocation efficiently, even in scenarios where the job queue might initially be empty. A series of figures demonstrates how *AttentiveSched* operates.

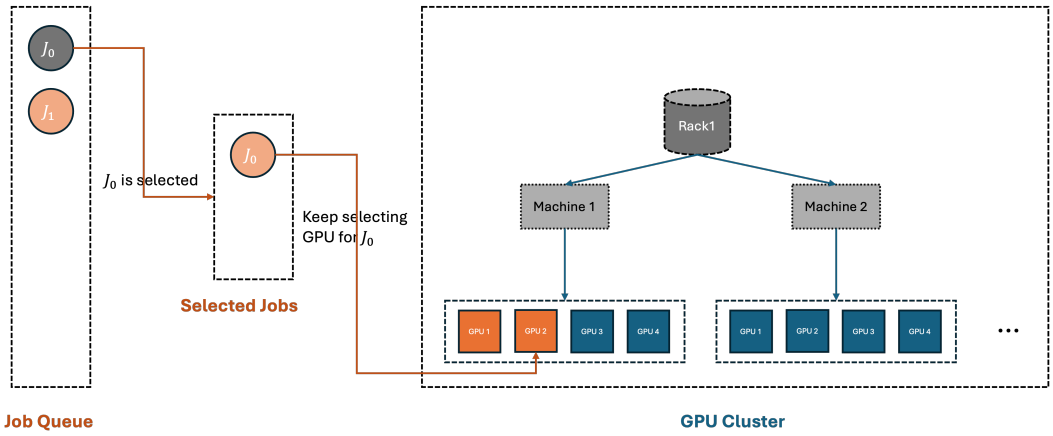

Figure 4.6: At time-step t_1, the GPU agent continues by selecting one more GPU for J_0.

At time-step t_0, as shown in Figure 4.4, the job queue is empty; the job agent takes an idle action, and the GPU agent does not operate. After the execution of the idle action, two jobs arrive in the queue.

At time-step t_1, the job agent begins scheduling jobs. As depicted in Figure 4.5, during sub-time-step 0, the job agent selects J_0, and the GPU agent assigns the first GPU to J_0. In Figure 4.6, the GPU agent continues by assigning a second GPU to J_0, completing its resource allocation.

Next, as shown in Figure 4.7, the job agent proceeds to select J_1. Since there are still enough GPUs available to run J_1, the GPU agent assigns one GPU to J_1. In Figure 4.8, the GPU agent assigns the second GPU to J_1, completing the resource allocation for J_1. The job scheduling and GPU assignment at t_1 are completed, and the jobs begin execution.

At time-step t_2, as illustrated in Figure 4.9, three new jobs arrive, each requiring more than 4 GPUs. Since the number of available GPUs is less than any of their requirements, the job agent takes an idle action, and the GPU agent does not operate.

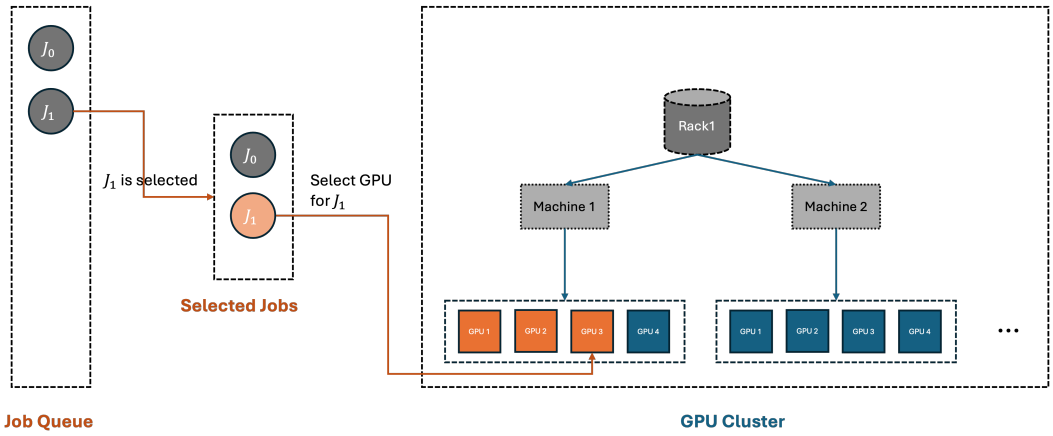

Figure 4.7: At time-step t_1, Since there are still enough GPUs available to run J_1, the job agent proceeds to select J_1. The GPU agent assigns one GPU to J_1.

4.6 Training

4.6.1 Optimizing with PPO

We use PPO [25] as our training algorithm for both agents. The update ratio between the current and the old policies is defined as:

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \tag{4.9}$$

where π_θ and $\pi_{\theta_{old}}$ represent the current and old policy parameters, respectively, and s_t, a_t denote the states and actions at time step t. To ensure stability, the algorithm avoids excessively large updates of policy parameters by clipping the ratio $r_t(\theta)$ within the range of $[1-\epsilon, 1+\epsilon]$, where ϵ is a hyper-parameter. As a policy gradient algorithm, PPO [25] utilizes the advantage function $\hat{A}_t$ to assess the quality of actions at time step t. In our work, we used GAE (Generalized Advantage Estimation) to smooth the learning pro-

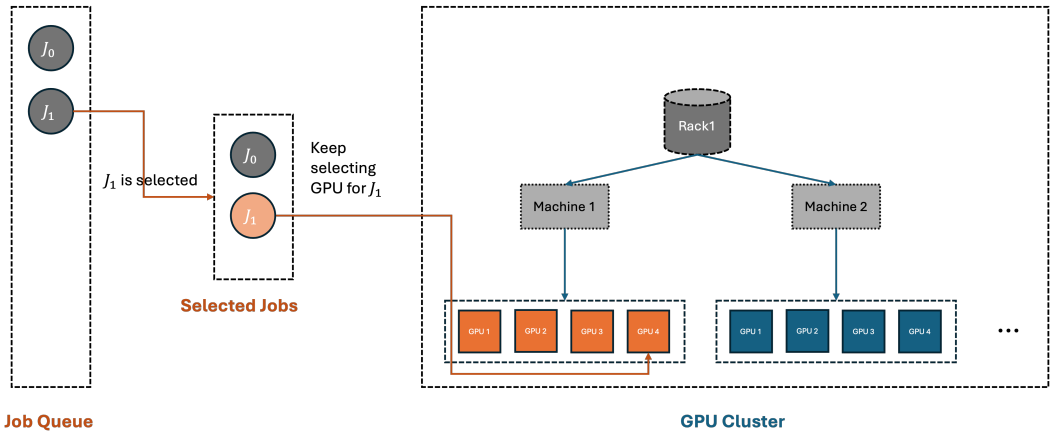

Figure 4.8: At time-step t_1, the GPU agent continues by assigning one more GPU to J_1. The job scheduling and GPU assignment at t_1 are completed, and the jobs begin execution.

cess, which is described by the equation:

$$\hat{A}_t = \sum_{k=0}^{\infty} (\gamma\lambda)^k \delta_{t+k} \tag{4.10}$$

where γ and λ are hyperparameters, and δ_t is the TD error at time step t, defined as:

$$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t) \tag{4.11}$$

where r_t represents the reward at time step t, and $V(s_t)$ is the estimated value of the state at time t.

Overall, the clipped surrogate objective function is given by:

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t) \right]$$

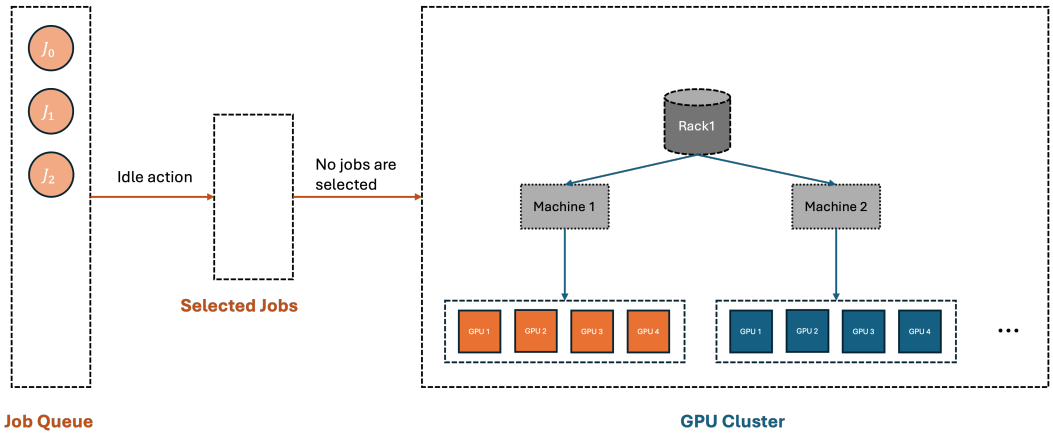

Figure 4.9: At time-step t_2, three jobs arrive, each requiring more than 4 GPUs. An idle action is taken because the number of available GPUs is less than any of their requirements.

4.6.2 Training Strategy

Each agent uses two networks of the same architecture for its policy and value heads, respectively. The strategy involves pretraining each agent independently before co-training them. During the pre-training phase, the job agent adopts the FirstFit strategy to select GPUs [34], while the GPU agent uses the FIFO method to schedule jobs [19]. After both agents have converged to a significant degree, they are co-trained together. The training efficiency of both strategies is examined in an ablation study. As with many PPO approaches, a rollout replay buffer [35] is used to store states, rewards, and actions throughout the training loop. The pseudocode for the independent training can be found in 1, and for the co-training in 2.

Algorithm 1 Proximal Policy Optimization (PPO) Training Loop for Independent Agent Training

Initialize policy parameters θ_0
Initialize entropy coefficient $\beta_0 = 0.01$
Initialize learning rate α_0
for iteration $= 0, 1, 2, \ldots$ **do**
    Run policy $\pi_{\theta_{\text{old}}}$ in environment for T timesteps
    Collect observations $\{s_t, a_t, r_t\}_{t=1}^{T}$
    Compute advantage estimates $\hat{A}_t$
    for epoch $= 1, 2, \ldots, K$ **do**
        Shuffle the batch of collected data
        for each minibatch **do**
            Compute the probability ratio:

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$$

            Compute policy entropy:

$$\mathcal{H}_t(\theta) = -\sum_a \pi_\theta(a|s_t) \log \pi_\theta(a|s_t)$$

            Compute surrogate loss with entropy term:

$$\mathcal{L}(\theta) = \hat{\mathbb{E}}_t \left[\min \left(r_t(\theta)\hat{A}_t,\ \text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}_t \right) + \beta \mathcal{H}_t(\theta) \right]$$

            Update policy parameters by maximizing $\mathcal{L}(\theta)$ with learning rate α
        end for
    end for
    Update old policy parameters: $\theta_{\text{old}} \leftarrow \theta$
    Update entropy coefficient β (decay from 0.01 to 0.0001)
    Update learning rate α using CosineAnnealingLR schedule
end for

Algorithm 2 Proximal Policy Optimization (PPO) Training Loop for Co-Training

Load policy parameters θ_0^j, θ_0^c from pre-trained Job Agent and GPU Agent
Initialize entropy coefficient $\beta_0 = 0.01$
Initialize learning rate α_0
for iteration $= 0, 1, 2, \ldots$ **do**
    Run policies $\pi_{\theta^j_{\text{old}}}, \pi_{\theta^c_{\text{old}}}$ in environment for T timesteps
    Collect observations $\{s_t^j, a_t^j, r_t^j, s_t^c, a_t^c, r_t^c\}_{t=1}^T$
    Compute advantage estimates $\hat{A}_t^j, \hat{A}_t^c$
    for epoch $= 1, 2, \ldots, K$ **do**
        Shuffle the batch of collected data
        for each minibatch **do**
            Compute the probability ratios for each agent:

$$r_t^j(\theta^j) = \frac{\pi_{\theta^j}(a_t^j|s_t^j)}{\pi_{\theta^j_{\text{old}}}(a_t^j|s_t^j)}$$

$$r_t^c(\theta^c) = \frac{\pi_{\theta^c}(a_t^c|s_t^c)}{\pi_{\theta^c_{\text{old}}}(a_t^c|s_t^c)}$$

            Compute policy entropy for each agent:

$$\mathcal{H}_t^j(\theta^j) = -\sum_a \pi_{\theta^j}(a|s_t^j) \log \pi_{\theta^j}(a|s_t^j)$$

$$\mathcal{H}_t^c(\theta^c) = -\sum_a \pi_{\theta^c}(a|s_t^c) \log \pi_{\theta^c}(a|s_t^c)$$

            Compute surrogate loss with entropy term for each agent:

$$\mathcal{L}^j(\theta^j) = \hat{\mathbb{E}}_t^j \left[\min \left(r_t^j(\theta^j)\hat{A}_t^j,\ \text{clip}\left(r_t^j(\theta^j), 1-\epsilon, 1+\epsilon\right)\hat{A}_t^j \right) + \beta \mathcal{H}_t^j(\theta^j) \right]$$

$$\mathcal{L}^c(\theta^c) = \hat{\mathbb{E}}_t^c \left[\min \left(r_t^c(\theta^c)\hat{A}_t^c,\ \text{clip}\left(r_t^c(\theta^c), 1-\epsilon, 1+\epsilon\right)\hat{A}_t^c \right) + \beta \mathcal{H}_t^c(\theta^c) \right]$$

            Update policy parameters for each agent by maximizing $\mathcal{L}^j(\theta^j)$ and $\mathcal{L}^c(\theta^c)$ with learning rate α
        end for
    end for
    Update old policy parameters: $\theta^j_{\text{old}} \leftarrow \theta^j$, $\theta^c_{\text{old}} \leftarrow \theta^c$
    Update entropy coefficient β (decay from 0.01 to 0.0001)
    Update learning rate α using CosineAnnealingLR schedule
end for

Chapter 5

Experiments

5.1 Experimental Design

5.1.1 Experimental Setup

GPU cluster We simulate a real GPU cluster that uses IBM Power8 servers, NVLinks to connect GPUs within the same server, InfiniBand (IB) to connect different servers, and Top of Rack Switch (TRS) with IB to connect racks. Since an NVLink [36] connection per a Tesla P100 GPU delivers 40 GB/s bidirectional interconnect bandwidth, we set the bandwidth between two GPUs on the same node to be 40 GB/s. Furthermore, since we have the IB overhead between intra-rack machines and IB and the TRS overhead between inter-rack machines, we assume their bandwidth is 4 GB/s and 2 GB/s, respectively. We define the communication cost as the inverse of the bandwidth: the time in seconds needed to transmit one GB. Thus, if GPUs are on the same machine, their communication cost is 1/40 s/GB. However, if they are on different machines but on the same rack, their communication cost is 1/4 s/GB, and if

they are on different racks, their communication cost is 1/2 s/GB. That is, we set c_{gpu}, $c_{machine}$ and c_{rack} in our communication cost computation to 1/40, 1/4 and 1/2, respectively.

In the experiment, the configuration of the cluster is: Each rack contains 4 machines, and each machine is equipped with 4 GPUs. The cluster consists of 4 racks, totaling 64 GPUs. The capacity of the job queue is set to 20, and it maintains all waiting jobs.

5.1.2 Datasets

Three datasets are used in the experiments: Microsoft DL Cluster Trace (111,883 jobs) [5], Alibaba Cluster Trace 2020 [3] (534 jobs), and Alibaba Cluster Trace 2023 [4] (185 jobs). A data generator is used to generate training data. This generator follows the same distribution of job lengths and numbers of requested GPUs as in the Microsoft DL Cluster Trace (MS Trace), creating the joint probability mass function (PMF) and sampling jobs with this probability distribution. 4,000 episodes are used in training, and each episode contains 200 jobs. A validation sequence of 200 jobs sampled from the MS Trace dataset is used to validate the agent's performance every 10 episodes for hyperparameter selection.

The trained model is tested on three datasets. Besides the MS Trace, the model is also tested on the Alibaba 2020 and 2023 traces. The model is fine-tuned on the job distributions of Alibaba 2020 and 2023 dataset to adapt to the data distribution before the testing on these two datasets.

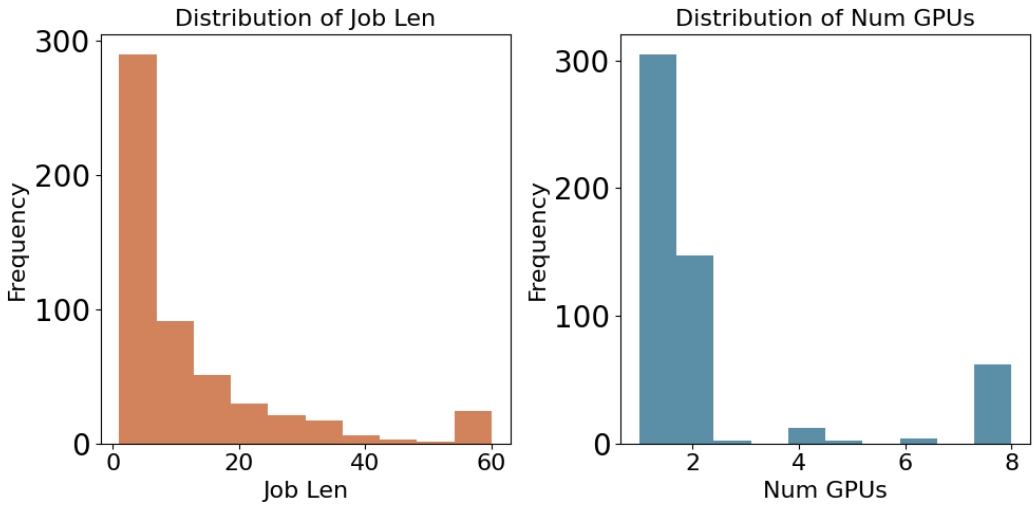

Figure 5.1: The distribution of Alibaba 2020 trace [3]

5.1.3 Baselines

We compare *AttentiveSched* with three heuristic, two metaheuristic, and four DRL-based schedulers. The schedulers are listed as follows:

- First-in-first-out (FIFO) [19]: Assigns higher priorities to jobs arriving earlier.
- Shortest Job First (SJF): Selects the job with the shortest estimated runtime, as used in Lyra [37].
- TETRIS [9]: Computes the dot product of normalized required job resources and available machine resources to make scheduling decisions.
- MOSA [10]: Extends traditional simulated annealing to handle multiple conflicting scheduling objectives.
- MALO [21] : A metaheuristic approach that leverages multiple adaptive learning techniques to optimize scheduling decisions.

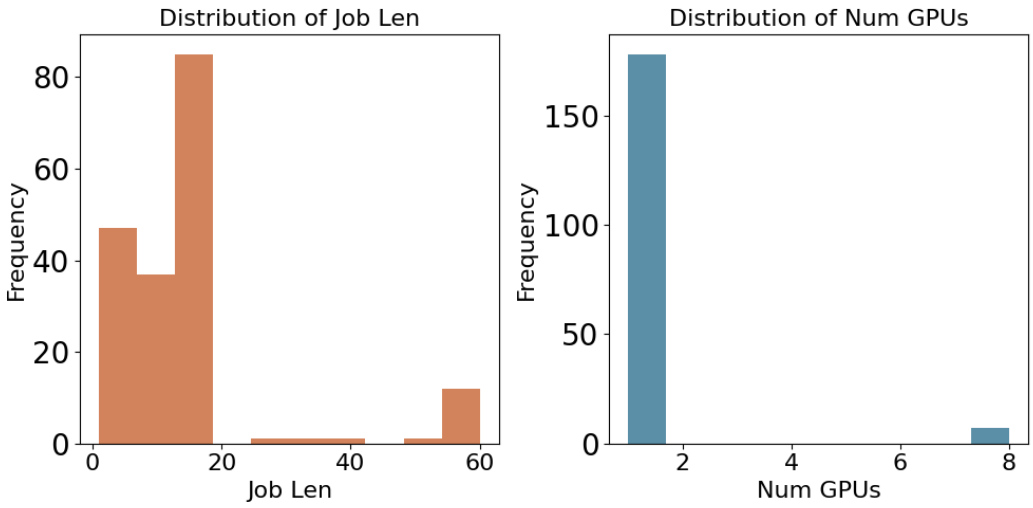

Figure 5.2: The distribution of Alibaba 2023 trace [4]

- RL-TAPS [14]: Utilizes DRL for job selection and GPU assignment to minimize average job slowdown.
- DL^2 [13]: Combines DRL for job selection with heuristic-based GPU assignment to minimize runtime interference. The policy neural network is initialized via offline supervised learning from a small set of historical job runtime traces.
- DeepRM [11]: Employs DRL for job scheduling and heuristics for resource assignment to minimize average job slowdown.
- DAS [12]: RL-based scheduler that manages distributed deep learning jobs on public clouds through two agents, target to minimize the completion time and training fee.

Schedulers without GPU assignment capabilities, such as FIFO [19], allocate the first k available resources, where k represents the number of requested GPUs.

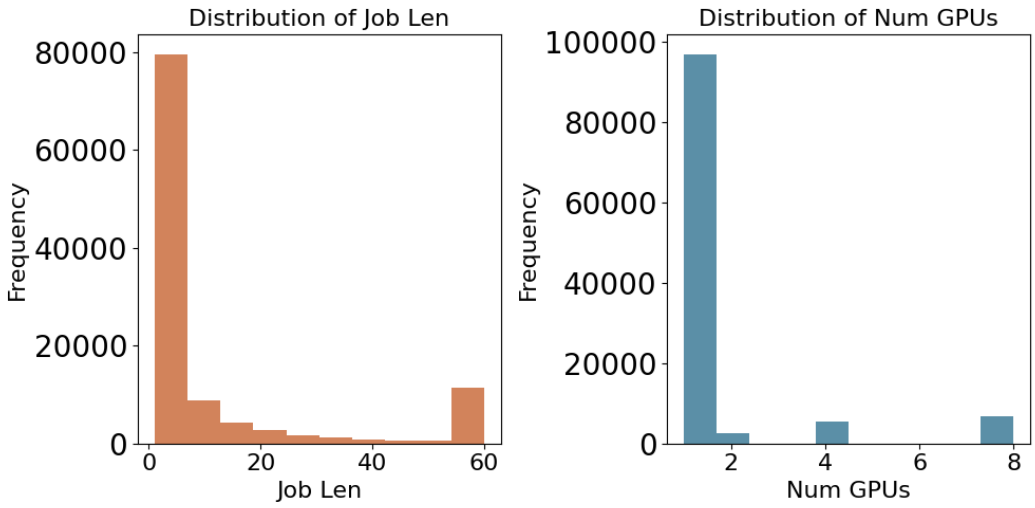

Figure 5.3: The distribution of Microsoft trace [5]

5.1.4 Hyperparameters

The model in both agents are configured with the following hyperparameters: $\alpha = 0.5$, $\beta = 0.2$, the embedding size in the embedding layer = 32, dropout rate = 0.1, the number of heads = 2, forward expansion = 4, hidden layer size = 256, and the activation function is ReLU. The value network has two layers of MLP, {512, 256}.

When training *AttentiveSched*, Adam is used as the optimizer, the initial learning rate is 0.0001 with the cosine annealing decay strategy[38], the mini-batch size is 512, the number of epochs is 3, $\gamma = 0.99$, $\lambda = 0.97$, policy clip $\epsilon = 0.2$, policy entropy decays from 0.01 to 0.0001 linearly, and the critic coefficient is 0.5, respectively. These hyper parameters were determined by using the validation set.

For meta-heuristics, both MOSA and MALO are configured to minimize the waiting time and the waiting time standard deviation in their fitness functions. During the solution searching, MOSA uses 1000 iterations, with a cool-

ing rate of 0.99 and an initial temperature of 100. In MALO, the number of iterations is 100, with 100 solutions being evaluated.

For DRL baselines, settings in the corresponding papers are used. $DL2$[13] uses a learning rate of 0.0001, a batch size of 256, a discount factor of 0.9, an entropy weight of 0.1, and the model contains two hidden layers with 256 neurons each. DeepRM [11] uses a learning rate of 0.001, a batch size of 64, RMSprop rho of 0.9, RMSprop epsilon of 1e-9, and the model contains one hidden layer with 20 neurons. Rl-TAPS [14] uses a learning rate of 0.001, a batch size of 64, and the model contains a hidden layer with 256 neurons. DAS [12] uses a learning rate of 0.00005, a batch size of 256, the epsilon clipping range is 0.2, the reward discount factor $\gamma$and weight of entropy loss λ are 0.98 and 0.03.

5.1.5 Evaluation metrics

We evaluate the scheduling performance using four key metrics: Job Completion Time (JCT), defined as the finish time of job j minus its arrival time; Job Waiting Time (WT), which is the time job j is scheduled minus its arrival time; Makespan (MSpan), representing the time taken by the agent to complete all jobs in an episode; and Objective, which combines measures of fairness and the average waiting time.

In our comparison, we aim to find the scheduler that has minimum sum of fairness and the average job completion time, which follows our objective.

Algorithms		MSpan	WT	JCT	Objective
Heuristics	SJF	530	7.37 ± 26.20	18.16 ± 50.74	44.36
	FIFO	519	12.42 ± 17.02	23.04 ± 31.76	40.06
	TETRIS	512	12.28 ± 15.44	22.70 ± 29.78	38.14
Meta-Heur	MOSA	540	12.98 ± 11.59	23.63 ± 28.18	35.22
	MALO	503	11.27 ± 13.43	21.87 ± 31.70	35.30
RL-based	DeepRM	533	11.88 ± 14.28	22.89 ± 34.53	37.17
	RL-TAPS	1344	39.62 ± 130.26	52.19 ± 142.85	182.45
	DL^2	502	11.20 ± 19.74	21.66 ± 33.52	41.40
	DAS	522	11.29 ± 14.62	21.66 ± 35.46	36.28
AttentiveSched		**499**	11.55 ± 12.60	21.73 ± 28.56	**34.33**

Table 5.1: Performance on Alibaba Cluster Trace 2020

Algorithms		MSpan	WT	JCT	Objective
Heuristics	SJF	230	3.06 ± 5.20	19.41 ± 28.29	24.61
	FIFO	255	4.26 ± 3.30	21.23 ± 29.39	24.53
	TETRIS	254	4.11 ± 3.06	20.77 ± 27.97	23.83
Meta-Heur	MOSA	203	4.55 ± 3.45	20.40 ± 23.43	23.85
	MALO	212	3.10 ± 3.40	19.61 ± 24.47	23.01
RL-based	DeepRM	211	4.0 ± 3.86	19.98 ± 23.88	23.84
	RL-TAPS	253	2.42 ± 4.93	19.59 ± 32.55	24.52
	DL^2	235	3.15 ± 4.83	19.50 ± 27.23	24.33
	DAS	220	3.79 ± 4.02	19.89 ± 25.19	23.91
AttentiveSched		**196**	3.17 ± 3.24	19.76 ± 24.87	**23.00**

Table 5.2: Performance on Alibaba Cluster Trace 2023

5.2 Results

Tables 5.3 (Microsoft Trace [5]), 5.2 (Alibaba Trace 2023 [4]), and 5.1 (Alibaba Trace 2020 [3]), compare *AttentiveSched* with baseline methods across three datasets. They present all metrics that we covered in Section 5.1.5.

Algorithms		MSpan	WT	JCT	Objective
Heuristics	SJF	187	2.34 ± 5.94	14.75 ± 29.23	20.69
	FIFO	**179**	3.68 ± 6.94	16.16 ± 28.93	23.10
	TETRIS	199	4.43 ± 6.08	16.69 ± 27.74	22.77
Meta-Heur	MOSA	221	5.28 ± 8.95	17.80 ± 30.20	26.75
	MALO	186	3.05 ± 5.95	17.78 ± 28.10	23.73
RL-based	DeepRM	191	3.92 ± 6.64	15.09 ± 27.76	21.73
	RL-TAPS	226	3.93 ± 8.78	15.66 ± 28.56	24.44
	DL^2	185	3.56 ± 6.97	15.95 ± 28.92	22.92
	DAS	186	2.91 ± 7.57	15.47 ± 29.42	23.04
AttentiveSched		181	2.51 ± 5.59	15.07 ± 26.33	**20.66**

Table 5.3: Performance on Microsoft Trace

5.2.1 Analysis and Findings

5.2.1.1 Analysis

Our analysis focuses on the objective column (sum of the fairness and the average job completion times). In addition, we also consider the makespan to determine if the scheduler is biased toward a specific metric.

In the Alibaba 2020 dataset, *AttentiveSched* achieves the best objective value, being 2% ahead of the second-best result, which is achieved by MOSA. *AttentiveSched* also has the lowest makespan among all schedulers, 1% ahead of the second-best makespan, achieved by DL^2.

In the Alibaba 2023 dataset, *AttentiveSched* once again achieves the best objective value, 0.5% ahead of the second-best, achieved by MALO. Although the objective values are very close, *AttentiveSched* also records the lowest makespan among all schedulers, 8% ahead of MALO and 3% ahead of the second-best makespan, achieved by MOSA.

In the MS trace dataset, *AttentiveSched* has an almost tied objective value with SJF, which is 0.1% ahead of it. However, the makespan is 3% ahead of

SJF and 1% worse than the best one, achieved by FIFO.

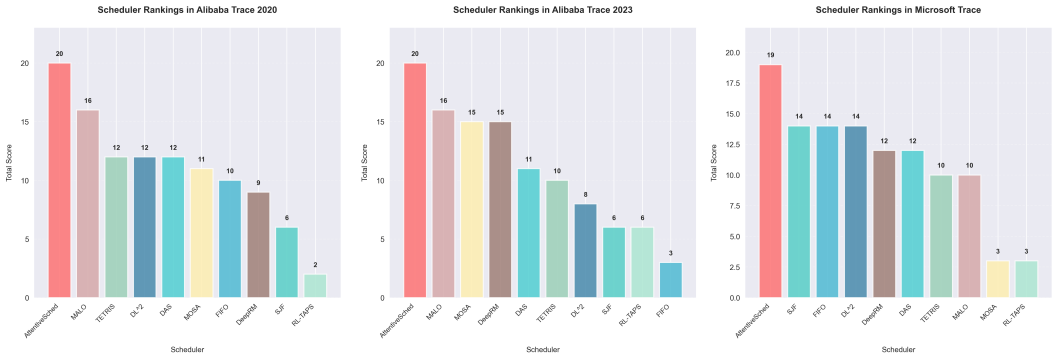

Figure 5.4: Local rankings of schedulers based on makespan and objective in each dataset

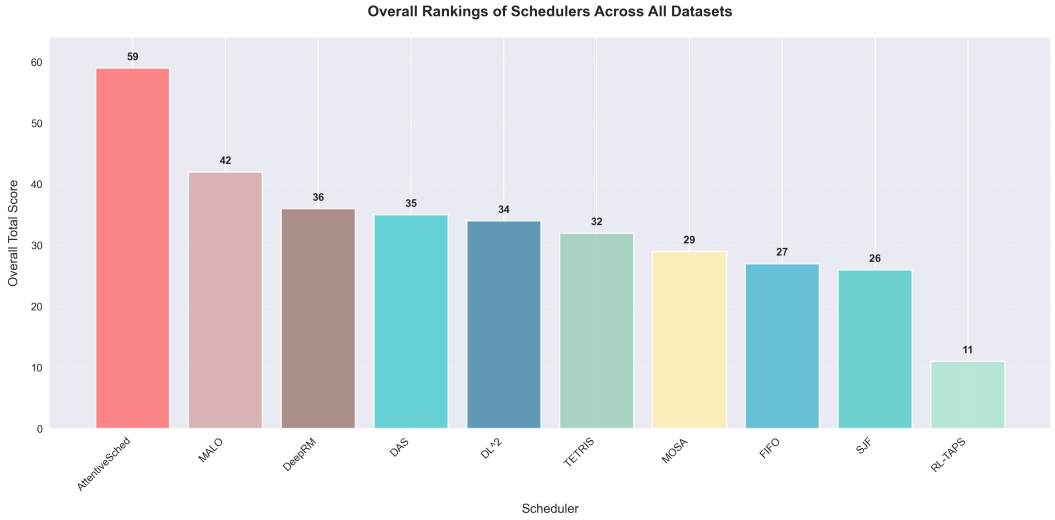

Figure 5.5: Global rankings of schedulers based on overall scores across all datasets

Overall, across the three datasets, *AttentiveSched* consistently achieves our objectives by exhibiting the lowest sum of fairness and average job completion time. Additionally, we ranked all schedulers based on their performance in makespan and objective within each dataset, as shown in Figure 5.4. Subsequently, we aggregated their scores to determine an overall ranking, as

illustrated in Figure 5.5. *AttentiveSched* attained the highest overall score, with MALO securing second place, and RL-TAPS obtaining the lowest score among all evaluated schedulers.

5.2.1.2 Findings

The experimental results and the rankings of the schedulers based on objective and makespan across the three datasets indicate that MALO achieved the second-highest overall performance among all schedulers. Following MALO, three DRL-based schedulers—DeepRM, DAS, and DL^2—ranked closely, with nearly identical scores of 36, 35, and 34, respectively. The heuristic-based schedulers demonstrated worse performance compared to the DRL-based schedulers and meta-heuristics, as they all ranked below fifth place. $RL - TAPS$ exhibited the lowest performance among all evaluated schedulers; this poor performance is attributed to its initial design for smaller-sized environments and the excessively large action space in this experiment, which convergence is not observed despite longer training periods.

5.3 Ablation Study

Algorithms	Makespan	WT	JCT	Objective
AttentiveSched	**196**	3.17 ± 3.24	19.76 ± 24.87	**23.00**
JobAgent + First-Fit	204	3.19 ± 3.4	19.88 ± 25.34	23.28
GPUAgent + FIFO	249	4.15 ± 3.22	20.26 ± 27.45	23.48

Table 5.4: Albaltion study on trying Job agent with First-Fit GPU assignment and GPU agent with FIFO job scheduler

Algorithms	Makespan	WT	JCT	Objective
CNN1D	**196**	3.17 ± 3.24	19.76 ± 24.87	**23.00**
MLP	197	3.08 ± 3.91	19.55 ± 25.03	23.46

Table 5.5: Albaltion study on using MLP and 1D convolutional layer as the embedding

Algorithms	Makespan	WT	JCT	Objective
Self-Attention	**196**	3.17 ± 3.24	19.76 ± 24.87	**23.00**
MLP	258	4.1 ± 6.16	20.69 ± 28.46	26.85
LSTM	210	3.22 ± 3.68	19.51 ± 25.9	23.19
RNN	215	3.53 ± 3.7	19.52 ± 25.12	23.22

Table 5.6: Albaltion study on using multiple models to process the embedded input

To evaluate the *AttentiveSched* framework more precisely, we conducted a series of experiments that examine various aspects of the framework, including testing the performance of a single agent independently, replacing components with popular models, training the agent on a different optimization objective, and training the agent using a different GPU topology. All experiments utilized the Alibaba 2023 dataset. In the final set of experiments, we compared the inference times of different schedulers to evaluate whether they significantly impact overall scheduling performance.

5.3.1 Independent Testing of Job Agent and GPU Agent

As shown in Table 5.4, we tested the independent performance of the Job agent and the GPU agent. The results indicate that their combined operation achieves better results than when they are used independently with heuristic methods.

Algorithms	Makespan	WT	JCT	Objective
Learnable weights	**196**	3.17 ± 3.24	19.76 ± 24.87	**23.00**
Average pooling	236	3.3 ± 4.13	20.14 ± 29.43	24.27

Table 5.7: Albaltion study on using average pooling to obtain the output layer

Reward Types	Makespan	WT	JCT	Objective
Fairness + Running Time	196	3.17 ± 3.24	19.76 ± 24.87	**23.00**
Fairness	226	3.21 ± 3.0	20.31 ± 25.67	23.31
Job Completion Time	**191**	3.65 ± 4.07	19.82 ± 22.21	23.89

Table 5.8: Albaltion study on trying different rewards

5.3.2 Performance of Replacing Components with Other Models

As described in Chapter 4, the *AttentiveSched* consists of three main components: an embedding layer, a multi-head self-attention mechanism, and a projection layer. We conducted four experiments to examine the effectiveness of each component: 1) replacing the 1D convolutional layer with an MLP [39] for embedding, 2) using an MLP [39] to process the embedded input, 3) using an LSTM [40] to process the embedded input, 4) using an RNN [41] to process the embedded input, and 5) substituting a learnable projection layer with average pooling to obtain the output.

5.3.2.1 Embedding Layer

We analyzed the impact of replacing the 1D convolutional layer with an MLP on performance. As shown in Table 5.5, with an equivalent amount of training time, the performance using a 1D CNN layer as the embedding layer is similar

Algorithms		MSpan	WT	JCT	Objective
Heuristics	SJF	210	3.43 ± 5.33	23.9 ± 26.56	29.23
	FIFO	235	4.68 ± 4.59	24.02 ± 28.07	28.61
	TETRIS	219	4.76 ± 4.7	23.83 ± 26.58	28.53
Meta-Heuristics	MOSA	218	5.63 ± 4.66	24.54 ± 25.39	29.20
	MALO	215	3.97 ± 4.58	23.14 ± 27.3	27.72
RL-based	DeepRM	218	4.16 ± 7.28	23.54 ± 29.33	30.82
	DL^2	208	5.24 ± 8.22	24.09 ± 26.07	32.31
	Rltaps	218	3.59 ± 7.19	23.83 ± 30.25	31.02
	DAS	214	3.95 ± 5.69	22.78 ± 27.56	28.47
AttentiveSched		**204**	5.05 ± 4.28	23.53 ± 23.14	**27.42**

Table 5.9: The experiment of Machine–level only topology on Alibaba 2023 Trace

to that of an MLP. However, the 1D CNN improves results by reducing the makespan by approximately 1%, the waiting time (WT) by 10%, and the job completion time (JCT) by 3%.

5.3.2.2 Self-Attention

Table 5.6 shows the performance of using an MLP and other popular sequence processing models to process the embedded input. With the same training duration, self-attention not only achieves better fairness but also significantly reduces makespan by 5% compared to the second-best model and 15% compared to the least effective model. It can be observed that the MLP performs poorly in processing sequential input compared to LSTM, RNN, and Transformer-like architectures.

Scheduler/Model	Setting 1	Setting 2	Setting 3	Setting 4	Setting 5
MOSA	0.63488	1.74138	4.78349	17.34027	109.30640
MALO	1.17520	5.44598	15.57050	69.32019	460.04321
SJF	0.00013	0.00042	0.00152	0.01038	0.02121
FCFS	0.00013	0.00040	0.00148	0.01041	0.02107
TETRIS	0.00013	0.00042	0.00147	0.00566	0.02517
DAS	0.15622	0.03986	0.20801	0.78448	6.49023
DeepRM	0.03741	0.01397	0.01844	0.03210	0.11049
DL^2	0.05299	0.03740	0.07608	0.12461	0.27881
RL-TAPS	0.02790	0.01888	0.01762	0.02974	0.09201
AttentiveSched	0.10198	0.16861	0.16095	0.24211	0.67073

Table 5.10: Inference times (in seconds) of different schedulers under various settings. Each setting corresponds to specific job queue and GPU cluster sizes: Setting 1: (10 jobs, 16 GPUs), Setting 2: (20 jobs, 32 GPUs), Setting 3: (30 jobs, 64 GPUs), Setting 4: (50 jobs, 128 GPUs), Setting 5: (100 jobs, 256 GPUs). All times are measured in seconds.

5.3.2.3 Projection Layer

Using learnable weights to reduce the dimensionality of the self-attention output has proven more effective than simply applying average pooling. The results can be found in Table 5.7.

5.3.3 Other Reward Functions

We evaluated *AttentiveSched* by altering the objective through replacing the reward function in Equation 4.7 with others. Two alternative rewards were tested. The first uses average job completion time as the reward, similar to the rationale described for the $\mu_{c,t}$ term in Chapter 4. Since the waiting time is no longer included in the reward function, we now define $\mu_{c,t}$ as the average job completion time up to timestep t. The second reward tested involves removing the $\mu_{c,t}$ term from the reward described in Equation 4.7.

The results in Table 5.8 show that using only fairness as the reward achieves a lower standard deviation of job waiting times (around 7%); however, this comes at the cost of approximately 13% longer makespan. The reward based on job completion time can lead the agent to complete jobs more quickly, disregarding fairness. This results in a lower makespan and narrower intervals for job completion times, considering both the average and its standard deviation.

5.3.4 Training on Another GPU Cluster Topology

In Section 5.1, the GPU cluster topology for all experiments conducted in this thesis is based on the IBM Power8 [42] server architecture, which has a rack-machine-GPU hierarchical structure. In a simplified GPU cluster, there is only machine-level communication cost. To test the generalization of *AttentiveSched*, we also examined it on a cluster topology that consists solely of machine-level communication costs, with $\frac{1}{4}$ used as the communication cost between machines. Given the change in topology, re-training of RL-based schedulers, including *AttentiveSched*, was required. The results are shown in Table 5.9. *AttentiveSched* still shows competitive performance against all baselines. Another observation is that the simpler topology improves the performance of Rltaps compared to the rack-machine-GPU structure. On average, DRL-based schedulers achieved the lowest makespan.

5.3.5 Inference Time Comparison

According to Table 5.10, we tested the inference or solution searching time of all baseline schedulers across various settings of GPU clusters and job queues, measuring them in seconds. From smaller to larger clusters, the solution

searching times for all heuristic schedulers are significantly less than those of other types, with all three having solution searching times of less than 0.1 second. The solution searching times for meta-heuristic schedulers are the longest, especially for MALO, which has a solution searching time of around seven minutes in the largest setting, consisting of 100 jobs in the queue and 256 GPUs in the cluster. DRL-based schedulers show acceptable inference times compared to the running times of jobs. Even in the largest setting, most of their inference times are below one second, except for DAS, which has a six-second inference time in the largest setting due to the extensive computation required by its Gaussian process. However, this is still relatively minor compared to the job running time. Overall, only meta-heuristic-based schedulers should consider solution searching time as a concern, and given the trend of increasing solution searching times, this duration could extend further if much larger clusters (e.g., thousands of GPUs) are configured.

5.4 Performance Tuning

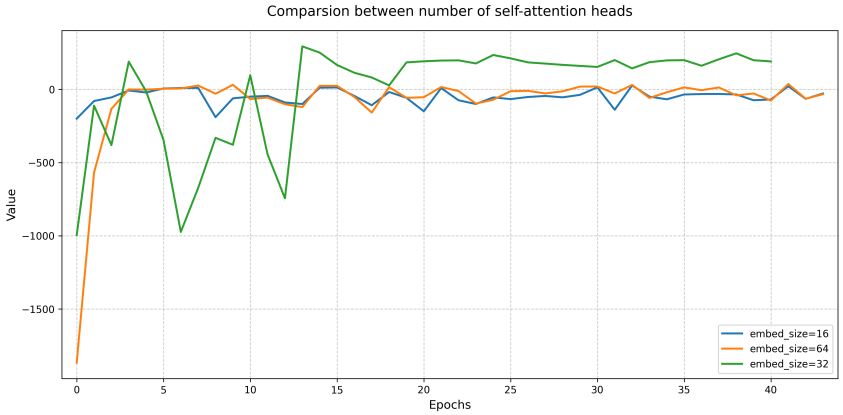

Figure 5.6: Comparison of different embedding sizes.

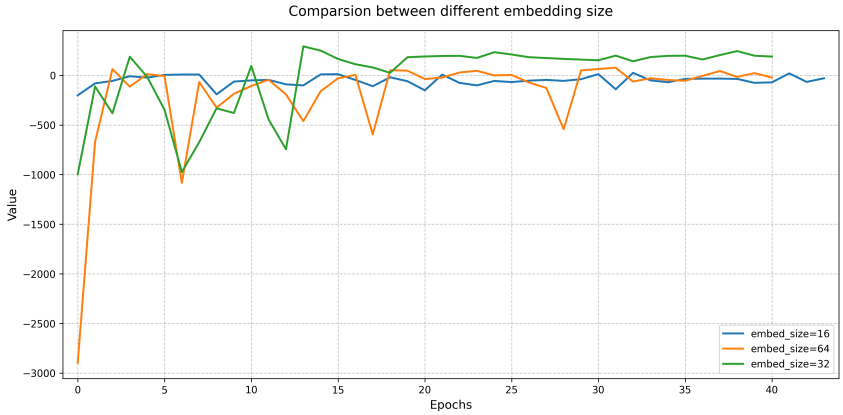

Figure 5.7: Comparison of different numbers of self-attention heads.

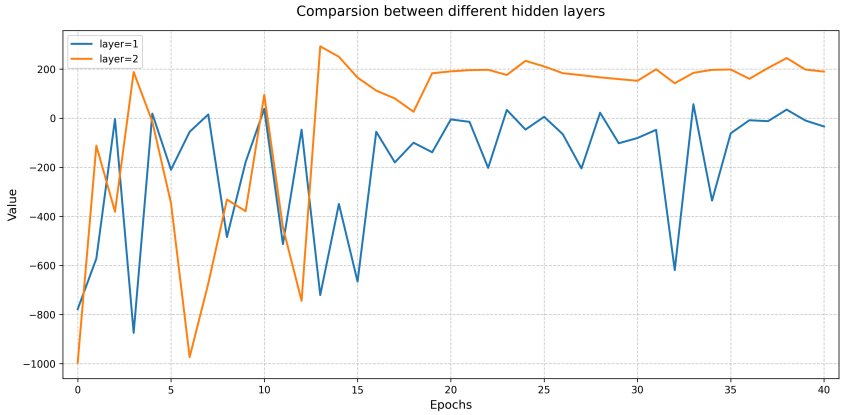

Figure 5.8: Comparison of different numbers of hidden layers.

Since *AttentiveSched* involves numerous hyperparameters, we conducted a series of experiments to determine the optimal settings for these parameters.

5.4.1 Reward Function Tuning

α and β are two hyperparameters in the reward function described in Equation 4.7. α is used to balance the weight between the average job waiting time and the standard deviation of job waiting times. β is used to balance the weight assigned to the average job completion time. We tuned α over the

range $[0, 1.0]$ and β over the range $[0.1, 1.0]$, incrementing each variable by 0.1, and tested all possible combinations. The combination that yielded the best results was $\alpha = 0.5$ and $\beta = 0.2$.

5.4.2 Model Tuning

5.4.2.1 Size of the Embedding Layer

The purpose of the embedding layer is to map the input features to higher dimensions. The size of the embedding is related to the number of features in the original input sequence. A size that is too small can lead to the loss of information, whereas a size that is too large can slow down training and may also cause overfitting. We tested embedding sizes of $\{16, 32, 64\}$. As demonstrated in the learning curve in Figure 5.6, an embedding size of 32 yielded the best results compared to the other sizes.

5.4.2.2 Number of Self-Attention Heads

The number of heads in the self-attention mechanism [27] is one of the most important hyperparameters. Each head can be trained to focus on different parts of the input sequence. A reasonable choice in the number of heads can improve the robustness of the model in dealing with noise and variants; conversely, an unsuitable number can lead to overfitting. The number of heads is related to the embedding size; with our embedding size set at 32, if we have 8 heads, each head would contain only 4 parameters, whereas with 4 heads, there would be 8 parameters per head. We tested $\{2, 4, 8\}$ heads. As shown in Figure 5.7, 4 heads provided the most robust learning curve among the three

options.

5.4.2.3 Number of Hidden Layers

After the projection layer, we need a few hidden layers to process the output to map to the action space. Additional hidden layers can not only align the output of the transformer model with the action size but also can learn relationships between each item. In our design, each layer contains 256 neurons. Since 3 layers obviously showed overfitting, we tested with $\{1, 2\}$ layers. As shown in Figure 5.8, one hidden layer was sufficient to provide a stable learning result.

Chapter 6

Conclusions and Future Directions

6.1 Summary of Contributions

This thesis proposed a reinforcement learning-based GPU job scheduler, *AttentiveSched*, that employs two agents to simultaneously optimize job selection and GPU assignment. *AttentiveSched* uses a CNN layer to extract local features in an input element, and a Transformer to extract global dependencies among the elements in the input. *AttentiveSched* achieves the lowest sum of waiting time standard deviation and average job completion time, while maintaining competitive makespan, in the comparison to several heuristic, meta-heuristic, and DRL-based schedulers on 3 test datasets with real GPU jobs from Microsoft and Alibaba DL Cluster Traces. This highlights the effectiveness of our attention-based DLR network design, reward function design and the consideration of the communication cost. In future work, we will investigate the

use of DRL for developing preemptive GPU job schedulers and explore more GPU cluster topologies.

6.2 Future Directions

There are several promising directions for future work based on the outcomes of this research:

- **Adding preemption:** With careful design of the pause/resume mechanism, a preemptive scheduler could naturally offer a broader range of effective scheduling solutions.
- **GPU elasticity:** The number of GPUs need not be constrained to the user's specification; when users do not specify the exact amount of GPUs they require, the scheduler could intelligently allocate an appropriate number of GPUs to jobs.
- **Incorporating more types of resources:** In addition to GPU resources, CPU jobs could also be considered within the cluster to broaden resource management and optimization.

Bibliography

[1] B. Jaeger and A. Geiger, "An invitation to deep reinforcement learning," *ArXiv*, vol. abs/2312.08365, 2023.

[2] J. M. Ponce-Ortega and L. G. Hernández-Pérez, *Metaheuristic Optimization Programs*, pp. 27–51. Cham: Springer International Publishing, 2019.

[3] Q. Weng, W. Xiao, Y. Yu, W. Wang, C. Wang, J. He, Y. Li, L. Zhang, W. Lin, and Y. Ding, "MLaaS in the wild: Workload analysis and scheduling in Large-Scale heterogeneous GPU clusters," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pp. 945–960, 2022.

[4] Q. Weng, L. Yang, Y. Yu, W. Wang, X. Tang, G. Yang, and L. Zhang, "Beware of fragmentation: Scheduling GPU-Sharing workloads with fragmentation gradient descent," in *2023 USENIX Annual Technical Conference (USENIX ATC)*, 2023.

[5] M. Jeon, S. Venkataraman, *et al.*, "Analysis of large-scale multi-tenant gpu clusters for dnn training workloads," *CoRR*, vol. abs/1901.05758, 2019.

[6] T. Brown, B. Mann, *et al.*, "Language models are few-shot learners," in *Advances in Neural Information Processing Systems*, vol. 33, pp. 1877–1901, Curran Associates, Inc., 2020.

[7] A. Kirillov, E. Mintun, N. Ravi, H. Mao, C. Rolland, L. Gustafson, T. Xiao, S. Whitehead, A. C. Berg, W.-Y. Lo, P. Dollár, and R. Girshick, "Segment anything," 2023.

[8] P. Sanders and J. Speck, "Efficient parallel scheduling of malleable tasks," in *2011 IEEE International Parallel & Distributed Processing Symposium*, 2011.

[9] R. Grandl, G. Ananthanarayanan, *et al.*, "Multi-resource packing for cluster schedulers," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, p. 455–466, 2014.

[10] T. Varadharajan and C. Rajendran, "A multi-objective simulated-annealing algorithm for scheduling in flowshops to minimize the makespan and total flowtime of jobs," *European Journal of Operational Research*, vol. 167, no. 3, pp. 772–795, 2005.

[11] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, "Resource management with deep reinforcement learning," in *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, Association for Computing Machinery, 2016.

[12] M. Xing, H. Mao, and S. e. a. Yin, "A dual-agent scheduler for distributed deep learning jobs on public cloud via reinforcement learning," in *Proc. of*

the 29th ACM SIGKDD Conf. on Knowledge Discovery and Data Mining, 2023.

[13] Y. Peng, Y. Bao, Y. Chen, C. Wu, C. Meng, and W. Lin, "Dl2: A deep learning-driven scheduler for deep learning clusters," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 8, pp. 1947–1960, 2021.

[14] B. Ryu, A. An, Z. Rashidi, J. Liu, and Y. Hu, "Towards topology aware pre-emptive job scheduling with deep reinforcement learning," in *Proc. of the 30th Annual International Conference on Computer Science and Software Engineering*, 2020.

[15] Y. Bao, Y. Peng, and C. Wu, "Deep learning-based job placement in distributed machine learning clusters with heterogeneous workloads," *IEEE/ACM Transactions on Networking*, vol. 31, no. 2, pp. 634–647, 2023.

[16] J. Krallmann, U. Schwiegelshohn, and R. Yahyapour, "On the design and evaluation of job scheduling algorithms," in *Job Scheduling Strategies for Parallel Processing*, pp. 17–42, Springer, 1999.

[17] S. A. Jyothi, A. Singla, *et al.*, "Measuring and understanding throughput of network topologies," in *Proc. of the Int'l Conf. for High Performance Computing, Networking, Storage and Analysis*, IEEE Press, 2016.

[18] H. Wang and Y. Chen, "Network topology description and visualization," in *2010 3rd International Conference on Advanced Computer Theory and Engineering(ICACTE)*, vol. 6, pp. V6–52–V6–56, 2010.

[19] D. Wang, J. Chen, and W. Zhao, "A task scheduling algorithm for hadoop platform," *J. Comput.*, vol. 8, pp. 929–936, 2013.

[20] K. Man, K. Tang, and S. Kwong, "Genetic algorithms: concepts and applications [in engineering design]," *IEEE Transactions on Industrial Electronics*, vol. 43, no. 5, pp. 519–534, 1996.

[21] L. Abualigah and A. Diabat, "A novel hybrid antlion optimization algorithm for multi-objective task scheduling problems in cloud computing environments," *Cluster Computing*, vol. 24, no. 1, p. 205–223, 2021.

[22] S. Mirjalili, "The ant lion optimizer," *Advances in Engineering Software*, vol. 83, pp. 80–98, 2015.

[23] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," 2016.

[24] C. C. White and D. J. White, "Markov decision processes," *European Journal of Operational Research*, vol. 39, no. 1, pp. 1–16, 1989.

[25] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *ArXiv*, vol. abs/1707.06347, 2017.

[26] M.-A. Chadi and H. Mousannif, "Understanding reinforcement learning algorithms: The progress from basic q-learning to proximal policy optimization," 2023.

[27] A. Vaswani, N. Shazeer, *et al.*, "Attention is all you need," in *Advances in Neural Information Processing Systems*, vol. 30, Curran Associates, Inc., 2017.

[28] T. Lin, Y. Wang, X. Liu, and X. Qiu, "A survey of transformers," *AI Open*, vol. 3, pp. 111–132, 2022.

[29] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems* (F. Pereira, C. Burges, L. Bottou, and K. Weinberger, eds.), vol. 25, Curran Associates, Inc., 2012.

[30] E. Bahel and C. Trudeau, "Stability and fairness in the job scheduling problem," *Games and Economic Behavior*, vol. 117, pp. 1–14, 2019.

[31] B. Li, M. Li, and R. Zhang, "Fair scheduling for time-dependent resources," in *Advances in Neural Information Processing Systems*, vol. 34, pp. 21744–21756, Curran Associates, Inc., 2021.

[32] M. Mähler, "Job scheduling under fairness aspects," in *Messung, Modellierung und Bewertung von Rechensystemen* (A. Lehmann and F. Lehmann, eds.), (Berlin, Heidelberg), pp. 46–60, Springer Berlin Heidelberg, 1991.

[33] J. Deng, A. An, *et al.*, "Gpu job scheduler with deep reinforcement learning," *Proceedings of the Canadian Conference on Artificial Intelligence*, May 2024. https://caiac.pubpub.org/pub/u6j20p40.

[34] A. Bertossi, L. Mancini, and F. Rossini, "Fault-tolerant rate-monotonic first-fit scheduling in hard-real-time systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 9, pp. 934–945, 1999.

[35] E. Liang, Z. Wu, *et al.*, "Distributed reinforcement learning is a dataflow problem," *CoRR*, vol. abs/2011.12719, 2020.

[36] A. Li, S. L. Song, *et al.*, "Evaluating modern gpu interconnect: Pcie, nvlink, nv-sli, nvswitch and gpudirect," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, pp. 94–110, 2019.

[37] J. Li, H. Xu, Y. Zhu, Z. Liu, C. Guo, and C. Wang, "Lyra: Elastic scheduling for deep learning clusters," in *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, (New York, NY, USA), p. 835–850, Association for Computing Machinery, 2023.

[38] I. Loshchilov and F. Hutter, "Sgdr: Stochastic gradient descent with warm restarts," *arXiv: Learning*, 2016.

[39] G. Singh and M. Sachan, "Multi-layer perceptron (mlp) neural network technique for offline handwritten gurmukhi character recognition," in *2014 IEEE International Conference on Computational Intelligence and Computing Research*, pp. 1–5, 2014.

[40] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, p. 1735–1780, Nov. 1997.

[41] A. Sherstinsky, "Fundamentals of recurrent neural network (rnn) and long short-term memory (lstm) network," *Physica D: Nonlinear Phenomena*, vol. 404, 2020.

[42] J. J. Cahill, T. Nguyen, *et al.*, "Ibm power systems built with the power8 architecture and processors," *IBM Journal of Research and Development*, vol. 59, no. 1, pp. 4:1–4:10, 2015.