UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

MACHINE LEARNING OPTIMIZATIONS OF THE EASY BACKFILL SCHEDULER FOR HPC SYSTEMS

A THESIS

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

Degree of

MASTER OF SCIENCE

By

ENZO DUREL
Norman, Oklahoma
2025

MACHINE LEARNING OPTIMIZATIONS OF THE EASY BACKFILL SCHEDULER FOR HPC SYSTEMS

A THESIS APPROVED FOR THE
SCHOOL OF COMPUTER SCIENCE

BY THE COMMITTEE CONSISTING OF

Dr. Chongle Pan (Chair)

Dr. Ronald Barnes

Dr. Andrew H. Fagg

Acknowledgments

First of all, I would like to thank Dr. Barnes who agreed to guide me during my research, as well as the two other members of my committee, Dr. Pan and Dr. Fagg.

I would also like to thank all the people who encouraged me during this period, such as my family and my close friends.

Table of Contents

List Of Tables

List Of Figures

Abstract

The goal of this research is to implement machine learning techniques to improve task scheduling algorithms in multi-core architectures. In this study, multi-core architectures are represented by High Performance Computing (HPC) systems, with a particular focus on enhancing the *EASY Backfill* scheduling algorithm.

To achieve this objective, a supervised learning model was developed to estimate the user-provided execution time, followed by a reinforcement learning model designed to refine and stabilize these predictions.

The supervised model, based on the *TabTransformer* architecture, was trained using data in the *Standard Workload Format (SWF)* such as HPC2N and CEA-Curie. These two datasets contains around 500,000 jobs collected during 5 years. The model was implemented within *Batsim*, an HPC resource management simulator.

After implementation, the supervised model demonstrated improved performance compared to the baseline but tended to be overly aggressive, often underestimating job runtimes. To mitigate this issue, the reinforcement learning model was introduced to predict a dynamic padding factor α, which increased the job success rate from 36% to 83% while maintaining stable performance across key metrics such as makespan, mean turnaround time improved by 61%, mean waiting time improved by 57%, and mean slowdown improved by 46%.

Chapter 1

Introduction

1.1 Background and Motivation

Optimization is a broad and well-known topic in mathematics. It consists of finding the best possible solution within a space of possibilities. This principle can be applied to almost everything in our world and is an integral part of computer science. Since solutions are not always trivial for certain problems, many optimization methods exist, including the use of machine learning, which is the main subject of this work.

Task scheduling is a complex problem in computer science: how can we determine the optimal execution order of programs in order to satisfy certain criteria, such as execution time and latency, within an environment influenced by numerous parameters, including computing power, available resources, and user activity.

It is within this framework that the idea of this research emerges: to explore various machine learning techniques with the goal of optimizing and relatively improving task scheduling on computer architectures featuring multiple processors.

1.2 Research Objectives

Many studies have already explored the use of machine learning for task scheduling. However, few have attempted to implement multiple machine learning solutions within well-known dynamic schedulers.

To simulate a multi-core architecture, I chose to use models of supercomputer simulation. Software such as Slurm, used on supercomputers like OSCER, allows for the management of tasks on these complex machines. Indeed, these systems consist of nodes, computing units, often with heterogeneous architectures, on which a user can execute a program.

These nodes are a valuable resource in environments where many users wish to run programs simultaneously. This is where Slurm comes into play: it incorporates certain dynamic scheduling algorithms that ensure the proper functioning of the system by deciding the order in which tasks are executed.

The most well-known and widely used algorithm in these systems is called Backfill. The objective of this research is therefore to improve this scheduling algorithm through the use of machine learning techniques.

First, we will discuss a supervised model capable of improving the prediction of user program execution times. Next, we will enhance the behavior of the scheduling algorithm through the integration of a reinforcement learning model. Finally, we will combine these methods to obtain promising results.

1.2.1 Research Questions

The following work will try to answer these research questions:

- RQ1: *Can a supervised learning model improve the accuracy of user-provided runtime estimates in real HPC workloads, and does this improved prediction positively impact scheduling performance?*

- RQ2: *Can reinforcement learning identify an effective padding factor that improves key scheduling metrics (slowdown, waiting time, turnaround time) compared to the baseline EASY scheduler?*

- RQ3: *Can a hybrid scheduling strategy that combines learned runtime estimates with RL-optimized padding provide measurable performance gains, and what trade-offs emerge between job timeliness (slowdown) and system reliability (success rate)?*

1.2.2 Contributions

This thesis makes the following contributions:

1. A supervised runtime prediction model (UTEM) based on the TabTransformer architecture, trained on large HPC traces to improve the accuracy of user runtime estimates.

2. A reinforcement-learning-based padding factor optimizer, implemented using the Cross-Entropy Method (CEM), to adjust the aggressiveness of EASY Backfilling through episodic policy search.

3. A hybrid ML-assisted scheduling framework, integrating UTEM predictions with CEM-optimized padding inside the Batsim simulator to evaluate real scheduling behavior.

4. A comprehensive experimental evaluation on HPC2N and CEA-Curie workloads showing significant improvements in slowdown, waiting time, and turnaround time, with explicit analysis of these results.

5. A discussion of limitations and future directions, including the need for dynamic, state-dependent padding strategies (e.g., sliding-window updates) and the extension to more expressive RL policies.

Chapter 2

Literature Review

2.1 Machine Learning for Task Scheduling

One might wonder why use machine learning in the context of task scheduling, even though many dynamic algorithms have already proven their effectiveness. I saw this approach of integrating machine learning techniques as complementary to modern algorithms. Machine learning has the ability to learn patterns from known data and adapt to new environments. In this case, replicas of workload data from supercomputers.

To establish a viable machine learning-based solution for task scheduling, we must first study the different machine learning techniques applicable in this field. I therefore spent the first major part of this work studying task scheduling, as well as the various possible solutions involving machine learning.

In the field of task scheduling, machine learning can generally be used in two different ways:

- **Predictive optimization:** predicting and refining known data that will help the scheduler plan tasks (execution time, waiting time, memory usage, etc.)
- **Prescriptive optimization:** defining a new scheduling logic based on or by creating rules

It is quite natural, in the first case, to consider using a supervised model, and in the second, to use reinforcement learning.

2.1.1 Supervised Learning

Supervised learning in task scheduling, within the context of the Backfill algorithm, can be expressed as a model that, based on data from numerous tasks, predicts or refines a key characteristic used by the scheduling algorithm.

For example, in the work *"Improving Backfilling by using Machine Learning to predict Running Times"* of Eric et al. (2015), the authors implement simple learning methods to estimate execution times from HPC traces. The results show that the backfilling policy associated with a machine learning model outperforms the classical algorithm (according to the "bounded slowdown" criterion). This paper also highlights the known correlation between the user's estimated runtime and both the job's characteristics and the user's historical data.

There are many ways to integrate this supervised approach. Here are some that have been explored:

- Predicting the execution time of a job.
- Estimating the reliability of the prediction and influencing the scheduler's behavior accordingly.
- Classifying jobs by duration type. This approach was recently implemented in the work of Yonghao et al. (2024), where they created a classifier that determine if a job is short or long. This approach shows improvements for their regression model.

Moreover, the objectives of optimization depend on the research framework. Here is a list of the main objectives I have encountered:

- Reduction of the average waiting time.
- Reduction of the ratio between (waiting time + execution time) and execution time alone.
- Improvement of resource utilization (memory, CPU). This last objective is difficult to implement and depends on the system. As a result, developing a generic model becomes challenging.
- Preservation of job submission order. I consider this characteristic very important in a multi-user environment. Ensuring fairness is difficult to achieve while also taking user history into account in the prediction.

The model developed is a **TabTransformer**, a recent approach that adapts Transformer architectures for tabular data environments. This approach is promising, and many studies have shown improvements compared to other machine learning and deep learning methods: "TabTransformer leverages the self-attention mechanism to effectively capture intricate patterns and dependencies within tabular data" (Wang et al. 2024)

I chose this architecture for its novelty and strong potential, particularly in HPC environments. As shown in Fengxian (2023), "The results show that the proposed method achieves an average accuracy of 0.892, with 15.2% MAPE, and outperforms other methods in terms of prediction performance and training time." Furthermore, this approach provided a pedagogical opportunity to deepen my understanding of Transformer-based architectures.

Transformers are both contextual and highly parallelizable, which is especially interesting for HPC workloads that can contain millions of job traces. Their parallel nature is an advantage, as tree-based algorithms such as Random Forest were often preferred for their parallel execution capabilities.

2.1.2 Reinforcement Learning

Scheduling on HPC systems has become increasingly complex over the years. These new complexities require new solutions, more efficient than simple scheduling heuristics such as FCFS or SJF combined with the Backfill algorithm. This can notably be seen in the study conducted for the CQGym and DeepRM simulators: "the increasing complexity of computing systems and the highly dynamic nature of workloads have placed a tremendous burden on manually designed and tuned scheduling heuristics." (Avery and Savakis 2023)

This is where reinforcement learning (RL) agents come into play, interacting with the system. They observe states (such as queue length, resource utilization, and waiting times), take actions (such as job selection, backfill replacement, or safety computations), and receive rewards (for example, negative job slowdown, idle core penalty, or fairness metrics). The DRAS (Fan et al. 2022) study demonstrates that DRL-based scheduling agents can improve results by approximately 45%.

Reinforcement learning can be perfectly integrated into an HPC scheduling system as follows:

- **State:** Information about the queue, available resources, and program characteristics.

- **Action:** Which parameter to optimize (job-core mapping, best runtime factor, etc.).
- **Reward:** Improvement in bounded slowdown, reduced waiting time, better resource utilization.

Each action and each scheduling decision step influences the future state of the system. This definition of the problem corresponds to a *Markov Decision Process (MDP) (Puterman 1994)*, which forms the basis of reinforcement learning problem formulation (Sutton and Barto 2018).

Recent studies have shown that an agent can not only improve key performance metrics of the system, such as total execution time, but also control other parameters to achieve a balance within the system depending on the optimization objective. For instance, in the RLBackfilling paper of Kolker-Hicks et al. (2023), the RL agent demonstrates control over the aggressiveness of backfilling, dynamically balancing utilization and fairness in scheduling.

The very concept of the Backfill algorithm is to execute jobs in order of arrival while allowing, when possible, smaller jobs to be executed earlier without delaying longer ones. The key principle is therefore to control the aggressiveness or frequency with which the algorithm decides to run jobs it considers "small."

2.1.3 Cross Entropy Method (CEM)

The Cross-Entropy Method, introduced by Rubinstein (1997), is a population-based stochastic optimization technique. CEM is designed for optimizing low-dimensional continuous parameters in problems where the quality of each candidate solution can be evaluated independently. It searches for the best candidates within a distribution

and then updates the sampling distribution by shifting the mean and variance of a Gaussian toward high-performing candidates.

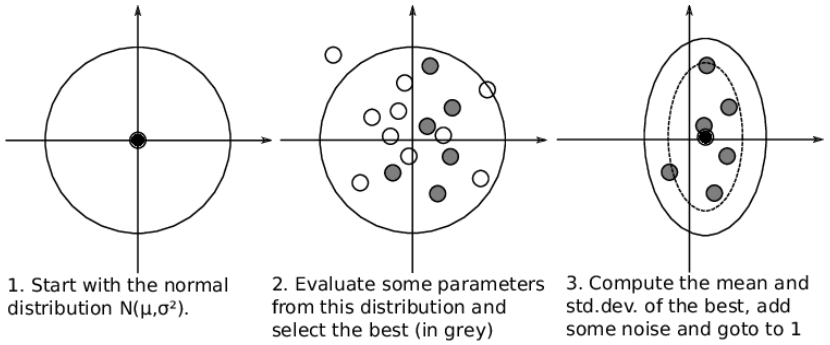

Figure 2.1: Schematic View of the Cross Entropy Method, adapted from Marin and Sigaud (2012)

Figure 2.1 illustrates the iterative structure of the Cross-Entropy Method. At each iteration, a population of candidate solutions is sampled from the current Gaussian distribution. The objective function is evaluated for each candidate, and the top-performing (elite) samples are selected. The mean and variance of the sampling distribution are then updated toward these elite values, progressively concentrating probability mass around high-quality solutions.

This method is suitable for optimizing one or only a few parameters. This is ideal since only a single parameter, α, needs to be optimized. The method provides stable training and is easy to implement.

CEM maintains a Gaussian distribution over the policy parameter α. In general CEM uses a covariance matrix, but in this problem, α is one-dimensional, so the covariance matrix reduces to scalar variance σ^2:

$$\alpha_i \sim \mathcal{N}(\mu_k, \sigma_k^2), \quad i = 1, \ldots, N.$$

At each iteration, we generate N candidates $\alpha_1, \alpha_2, \ldots, \alpha_N$ sampled from this distribution. In this case, these α_i correspond directly to the values α_i. We then compute the reward obtained for each candidate, $reward(\alpha_i)$. Next, we keep the M best candidates, i.e., those achieving the highest rewards. Let N be the number of sampled candidates and M the number of elite candidates ($M < N$). Let $R(\alpha_i)$ be the reward associated with candidate α_i. The elite set is

$$E_k = \{\alpha_i \mid \alpha_i \text{ is among the top M candidates ranked by R}(\alpha_i)\}.$$

The distribution is then updated by computing the mean and standard deviation of the selected candidates:

$$\mu_{k+1} = \frac{1}{M} \sum_{\alpha_i \in \mathcal{E}_k} \alpha_i,$$

$$\sigma_{k+1} = \sqrt{\frac{1}{M} \sum_{\alpha_i \in \mathcal{E}_k} (\alpha_i - \mu_{k+1})^2}.$$

These steps are repeated until:

$$\sigma_k < \varepsilon \qquad \text{or} \qquad k > K_{max},$$

where K_{max} is a maximum number of iterations. It is used to prevent non-convergence in cases where the reward landscape is flat or multimodal. When the reward distribution is flat, the variance may not decrease significantly even though further updates do not meaningfully change the estimated optimum.

2.1.4 Hybrid Learning

While supervised learning and reinforcement learning each have their own strengths, recent studies have shown that combining them can lead to more efficient and adaptive systems. This combination is often referred to as *Hybrid Learning*. The goal of this approach is to leverage the advantages of both paradigms, the accuracy and data efficiency of supervised learning, and the adaptability and exploration capabilities of reinforcement learning.

While prior research has explored RL for job scheduling and even RL for backfilling, few works explicitly combine runtime prediction and RL-based backfilling in a unified, simulator-driven framework.

It could addresses this gap by integrating a supervised runtime estimator with an improved backfill policy using RL, deploying and evaluating the combined model within Batsim (Dutot et al. 2017), ensuring realism and providing a modular design that enables offline training, online fine-tuning, and hybrid (predictor + RL) experimentation.

Hence, this work contributes to the evolution of data-driven, adaptive scheduling in HPC and aligns with the research trajectory established by RLBackfilling (Kolker-Hicks et al. 2023) and DRAS (Fan et al. 2022).

Chapter 3

Methods

3.1 HPC Scheduling

High Performance Computing (HPC) systems have become increasingly widespread with the rise of computing applications that demand ever greater resources. These systems are often composed of multiple computation points called *nodes*. These nodes are most often multi-core and heterogeneous in architecture. This work will focus only on clusters composed of heterogeneous CPU nodes. We consider a cluster composed of a finite set of compute nodes

$$\mathcal{M} = \{1, 2, \ldots, M\}.$$

Each node $m \in \mathcal{M}$ may have a different capacity. For simplicity, we represent the capacity of node m by a positive integer c_m, corresponding for instance to the number of CPU cores or an abstracted compute unit. We denote the total capacity of the system by

$$C_{\text{tot}} = \sum_{m \in \mathcal{M}} c_m.$$

At time t, let $\mathcal{R}(t)$ denote the set of running jobs and let $A_m(t)$ be the amount of capacity allocated on node m. The remaining capacity on node m is

$$f_m(t) = c_m - A_m(t),$$

and the global free capacity vector is $f(t) = (f_1(t), \ldots, f_M(t))$.

3.1.1 Properties

A workload consists of a finite set of independent jobs

$$\mathcal{J} = \{J_1, J_2, \ldots, J_N\}.$$

Workloads used during this thesis are files where one job is define per line, containing multiples attributs, features defining as the columns of the workloads.

Each job J_i is characterized by the tuple

$$J_i = (a_i, \mathbf{p}_i, r_i),$$

where:

- a_i is the submission (arrival) time;
- $\mathbf{p}_i \in \mathbb{N}^M$ is the resource demand vector, where $p_{i,m}$ is the amount of capacity required on node m (e.g., cores), with $0 \leq p_{i,m} \leq c_m$;
- $r_i \in \mathbb{R}_{>0}$ is the (unknown) actual runtime.

Usually, $\mathbf{p}_i$ is sparse, jobs may require capacity on a subset of nodes.

Jobs are non-preemptive: once started, a job occupies $\mathbf{p}_i$ continuously for r_i time units until completion.

Let b_i denote the start time assigned by the scheduler, and $c_i = b_i + r_i$ its completion time. The waiting time is

$$w_i = b_i - a_i.$$

One of the main characteristics of these systems lies in the existence of a resource manager — a small program that instruments and orchestrates the many programs executed within these multi-user systems. This is where the main management challenge arises: deciding which resources to allocate, how many, to whom, and for how long.

A schedule assigns to each job J_i:

- a start time $b_i \geq a_i$;
- an allocation pattern $\mathbf{p}_i$ over nodes such that the capacity constraints on each node are respected.

Formally, at any time t, the capacity constraint on node m is

$$\sum_{J_i \in \mathcal{R}(t)} p_{i,m} \;\leq\; c_m, \qquad \forall m \in \mathcal{M}, \tag{3.1}$$

where $\mathcal{R}(t)$ is the set of jobs i with $b_i \leq t < b_i + r_i$.

A schedule is *work-conserving* if, for any time t and any job J_i waiting in the queue, there exists no feasible allocation of $\mathbf{p}_i$ on the free capacity $f(t)$ that would allow J_i to start earlier.

In this context, an optimization problem naturally emerges: how to allocate each task in a way that optimizes certain objectives, such as execution time, average waiting time, or resource usage (memory, CPU, etc.).

Several performance metrics are traditionally used in HPC systems:

Makespan. The makespan is defined as

$$\text{makespan} = \max_i c_i.$$

Average waiting time. The average waiting time is defined as

$$\overline{w} = \frac{1}{N} \sum_{i=1}^{N} w_i.$$

Turnaround time. The turnaround time is defined as

$$T_i = w_i + r_i, \qquad \overline{T} = \frac{1}{N} \sum_{i=1}^{N} T_i.$$

Slowdown (bounded or unbounded). The bounded and unbounded slowdowns are defined as

$$\text{slowdown}(J_i) = \frac{w_i + r_i}{r_i}, \qquad \overline{\text{slowdown}} = \frac{1}{N} \sum_{i=1}^{N} \frac{w_i + r_i}{r_i}.$$

Fairness and predictability. HPC centers often require:

- respect for FCFS order (especially for the head-of-queue job),
- predictable job start times,
- avoidance of starvation of large or small jobs.

In practice, HPC centers choose one or more of the following objectives:

- minimize average turnaround time,
- minimize average slowdown,
- minimize makespan,
- maximize average utilization,
- minimize tardiness or bounded slowdown,

- optimize weighted combinations of these metrics under policy constraints (e.g., FCFS with backfilling).

3.1.2 Backfill Algorithm

To address these challenges, several algorithms have been implemented and used in such systems, notably the *Backfill* algorithm, or more specifically its most common version, *EASY Backfill* (Li and Zhao 2007), known for its simplicity of implementation. This algorithm relies on metadata provided by users that specify or predict the resource usage of their programs. The most important metadata in these systems include:

- **Node:** Which node or computation point will be used.
- **Estimated execution time:** The predicted running time of the program.
- **CPU:** Number of processors to use.
- **GPU:** Number of graphics processors, or the VRAM required by the program.
- **Memory:** Amount of RAM needed to execute the program.

The EASY Backfill algorithm is based on a FCFS policy with a *reservation* for the first job in the queue $J^{(0)}(t)$, combined with opportunistic execution of smaller jobs that do not delay this reservation.

Reservation time of the first job. At time t, let $J^{(0)}(t)$ be the head-of-queue job with parameters $(a_0, r_0, p_0, \hat{t}_0)$. The scheduler computes the earliest time $R_0(t)$ such that:

- at least p_0 nodes are available for the entire time interval $[R_0(t), R_0(t) + \hat{t}_0)$,
- $R_0(t) \geq t$.

Equivalently, $R_0(t)$ is the smallest $u \geq t$ such that

$$\forall \tau \in [u, u + \hat{t}_0), \quad M - \sum_{J_i \in \mathcal{R}(\tau)} p_i \ \geq \ p_0.$$

The interval $[R_0(t), R_0(t) + \hat{t}_0)$ defines the *reservation window* for $J^{(0)}(t)$.

Feasibility of a backfill job. Consider a job $J_k \in \mathcal{Q}(t)$ with parameters $(a_k, r_k, p_k, \hat{t}_k)$, where $J_k \neq J^{(0)}(t)$. The scheduler may start J_k at time t in backfill mode if and only if the following two conditions hold:

1. *Immediate resource feasibility:*

$$p_k \leq f(t),$$

   i.e., there are enough free nodes to start J_k at time t.

2. *No delay of the reservation:* assuming that J_k runs for at most its estimated time $\hat{t}_k$, its execution interval $[t, t + \hat{t}_k)$ must not violate the reservation of $J^{(0)}(t)$. In practice, this reduces to checking that the additional allocation of p_k nodes for at most $\hat{t}_k$ time units does not cause the number of free nodes to fall below p_0 at any point within the reservation window.

If both conditions are satisfied, J_k is *backfilled*: it is started immediately at time t and removed from $\mathcal{Q}(t)$, without changing the reservation time $R_0(t)$ of $J^{(0)}(t)$.

Algorithm. The EASY Backfill policy can now be summarized as follows at each decision time t:

1. Update $\mathcal{R}(t)$ and $\mathcal{Q}(t)$ based on job arrivals and completions.

2. While there exists a job in $\mathcal{Q}(t)$ that can start immediately on free nodes, try to start jobs according to:

(a) If $J^{(0)}(t)$ can start (i.e., $p_0 \leq f(t)$), then start $J^{(0)}(t)$ and remove it from the queue.

(b) Otherwise, compute the reservation time $R_0(t)$ for $J^{(0)}(t)$.

(c) For each job J_k in $\mathcal{Q}(t)$ with $k \geq 1$ (in arrival order), test the two feasibility conditions above. If both hold, start J_k at time t as a backfill job and update $f(t)$ accordingly.

3. Advance simulation time to the next event (arrival of a job or completion of a running job) and repeat.

By construction, EASY Backfilling preserves FCFS order for the head-of-queue job: no backfilled job is ever allowed to delay the reserved start time $R_0(t)$ of $J^{(0)}(t)$ under the assumption that each job runs no longer than its estimated walltime $\hat{t}_i$.

3.1.3 TensorFlow

Deep learning techniques have grown rapidly in recent years. For this work, I chose to use *TensorFlow*, a Python library for designing and implementing various machine learning techniques.

This library was chosen to ensure code consistency — both for implementing the user execution time prediction model (which I am more familiar with in TensorFlow) and for developing the deep reinforcement learning agent. The version used in this work is TensorFlow 2.19 (Abadi et al. 2015) and Keras 3.10 (Chollet et al. 2021).

3.1.4 Batsim

Selecting the right environment is crucial for developing machine learning solutions. Initially, I created my own simulator for multi-processor machines from scratch. I used

this environment to prototype and better understand the system in which I would be working. However, this approach had limitations, leading me to turn toward HPC systems.

At first, I intended to use *SLURM*, since it was used at the university and I was already familiar with it. However, it turned out that SLURM's internal development tools were limited — for instance, custom algorithm implementations would only have been possible in C.

For research purposes, it seemed more appropriate to choose an environment based on R or Python, where prototyping is easier.

My choice ultimately turned to *Batsim*. Batsim is a simulator built on top of *SimGrid*, another simulation and analysis framework for heterogeneous computing environments. Batsim also provides a Python framework called *pybatsim*, which simplifies the implementation of machine learning–based scheduling algorithms.

Batsim uses a discrete-event model. The simulation timeline is advanced by events such as:

- job submission events,
- job start events (triggered by the scheduler),
- job completion events,
- resource availability updates.

For each event involving scheduling decisions, Batsim pauses the simulation and sends a message to the external scheduler describing the current system state. After receiving the scheduler's decision, Batsim resumes the simulation and updates the relevant events.

3.1.4.1 Workload Description

Jobs are provided through a *workload file*, typically in JSON format. Each job specifies:

- a submission time;
- a duration profile (e.g., a `delay` profile with a fixed runtime);
- a resource request (e.g., number of cores or nodes);
- optional metadata inside a `json` field, used to store additional information such as user/group identifiers or predicted runtimes.

3.1.4.2 External Scheduler Interface

One of Batsim's key features is that it does *not* contain a built-in scheduler. Instead, the scheduler runs as a separate program and communicates with Batsim through a ZeroMQ-based protocol (Hintjens 2013). The messages exchanged include:

- **JOB_SUBMITTED**: a new job has arrived;
- **JOB_COMPLETED**: a job has finished execution;
- **REQUEST_DECISION**: Batsim requests scheduling actions;
- **EXECUTE_JOB**: scheduler instructs Batsim to start a job;
- **KILL_JOB**: scheduler requests termination of a job (if supported).

Here is an example of batsim message exchanged in batsim side:

```
1    [master_host0:Scheduler REQ-REP:(1114) 1678743.359054] [
   network/INFO] Received '{"now": 1678743.859054, "events":
   [{"timestamp": 1678743.859054, "type": "EXECUTE_JOB", "data
   ": {"job_id": "w0!397", "alloc": "80-87"}}]}'
```

Listing 3.1: Example of Batsim simulation message

Here are some examples of batsim message exchanged in scheduler side:

```
INFO:pybatsim.batsim.batsim:Message Sent to Batsim: {'now'
: 1678743.859054, 'events': [{'timestamp': 1678743.859054,
'type': 'EXECUTE_JOB', 'data': {'job_id': 'w0!397', 'alloc'
: '80-87'}}]}
```

Listing 3.2: Execute job message example from scheduler to Batsim simulator

```
INFO:pybatsim.batsim.batsim:Message Received from Batsim:
{'now': 1678773.885189, 'events': [{'timestamp':
1678773.885189, 'type': 'JOB_COMPLETED', 'data': {'job_id':
 'w0!395', 'job_state': 'COMPLETED_WALLTIME_REACHED', '
return_code': -1, 'alloc': '0-19 28-35 60-71'}}]
```

Listing 3.3: Job completed message example from scheduler to Batsim simulator

The scheduler thus fully controls:

- when jobs start,
- how resources are allocated,
- when the queue is updated,
- how priorities or heuristics are applied.

For research purposes, this separation between simulation and scheduling logic is one of the main reasons I used it.

3.1.5 Platforms

Batsim is implemented on top of SimGrid (Casanova 2001), and therefore depends on it for simulations. A system is defined in a *platform* file, which is an XML file describing the computing environment where the simulation will be executed.

A platform is defined by several XML tags, including:

- **zone:** Equivalent to a cluster.
- **host:** A computing resource at any level (processor, node, etc.).
- **link:** A connection between multiple hosts.
- **route:** A path between two hosts.
- **zoneRoute:** A path between zones.

3.1.6 Workloads

To simulate such an environment, we need datasets — workloads. Finding a suitable dataset for scheduling simulation can be challenging, as many are available online, but each serves a different purpose (analysis, optimization of specific criteria, etc.).

In the first part of my research, I spent time searching for a dataset — traces that would provide a solid basis for simulation. Generally, when simulating multi-architecture environments, research papers tend to use their own benchmarking systems (Bienia 2011) tailored to the specific optimization of a chosen system.

However, this approach is often complex and highly constrained. Solutions are developed for specific architectures and systems at a given time, while computing technologies evolve continuously. Research and its industrial implementation are often separated by several years, making it unwise to base a model on potentially obsolete systems.

Nevertheless, it is necessary to select an environment in which to simulate in order to move forward. A lot of different workloads are accessible on Internet, like Google Cluster (Reiss et al. 2011) or Azure Traces (GitHub 2025). However, I based my work on the efforts of the scientific community to standardize HPC datasets through the *Standard Workload Format (SWF)* (Talby et al. 2014).

This format "was defined in order to ease the use of workload logs and models" (Talby et al. 1999). It is a widely used representation for job logs collected from production HPC clusters. Each job is represented by a single line containing exactly 18 space-separated fields in a fixed order. Missing or unknown information is encoded using the value -1. Formally, an SWF workload is a finite sequence

$$\mathcal{W} = \{L_1, L_2, \ldots, L_N\},$$

where each line L_i corresponds to job J_i and is of the form

$$L_i = (F_{i,1}, F_{i,2}, \ldots, F_{i,18}).$$

An SWF line L_i is valid if it satisfies:

$$F_{i,1} \in \mathbb{N}, \qquad F_{i,2} \geq 0, \qquad F_{i,4} \geq 0,$$
$$F_{i,5}, F_{i,8} \geq 1, \qquad \text{and} \qquad F_{i,3}, F_{i,6}, F_{i,7}, F_{i,9}, F_{i,10} \in \{-1\} \cup \mathbb{R}_{\geq 0}.$$

Here are some lines from a SWF workload format file:

```
1 272428 31690439 0 45  63 -1 -1  63   840 -1 1 274 167 -1 -1 4
    -1 -1
2 272429 31690542 0  6 512 -1 -1 512 86400 -1 1 47  151 -1 -1 4
    -1 -1
3 272430 31690547 0 50  28 -1 -1  28   840 -1 1 274 167 -1 -1 4
    -1 -1
```

Listing 3.4: SWF example lines from HPC2N workload

Many workload traces are available on their website, ranging from 1993 to 2015. Additionally, a data-cleaning effort has been made for almost all datasets to remove

inconsistent lines. Nevertheless, some columns may still contain missing values, represented by -1.

Most simulators can read these files directly or offer tools to convert them into their own format — which is the case for Batsim. It provides a conversion program from SWF to JSON, producing a `workload.json` file:

$$\texttt{workload.json} = \{\text{“jobs”} : [J_1, J_2, \ldots, J_N]\}.$$

Each job J_i is represented as a JSON object:

$$J_i = \{\text{“id”},\ \text{“subtime”},\ \text{“walltime”},\ \text{“res”},\ \text{“profile”},\ \text{“json”}\}.$$

A job J_i converted from SWF into Batsim JSON takes the form:

```
{
  "id": 272428,
  "subtime": 31690439,
  "walltime": 840,
  "res": { "ncores": 63 },
  "profile": { "type": "delay", "duration": 45 },
  "json": {
      "userid": 274,
      "groupid": 167
  }
}
```

Listing 3.5: Batsim JSON job definition example

The two files, `platform.xml` and `workload.json`, thus provide all the data required to run a simulation in Batsim. Once the simulation is running, we connect to it via *pybatsim* through a TCP interface, and then execute a scheduling algorithm

implemented in Python. The figure 3.1 shows the full environment of the Batsim and pyBatsim.

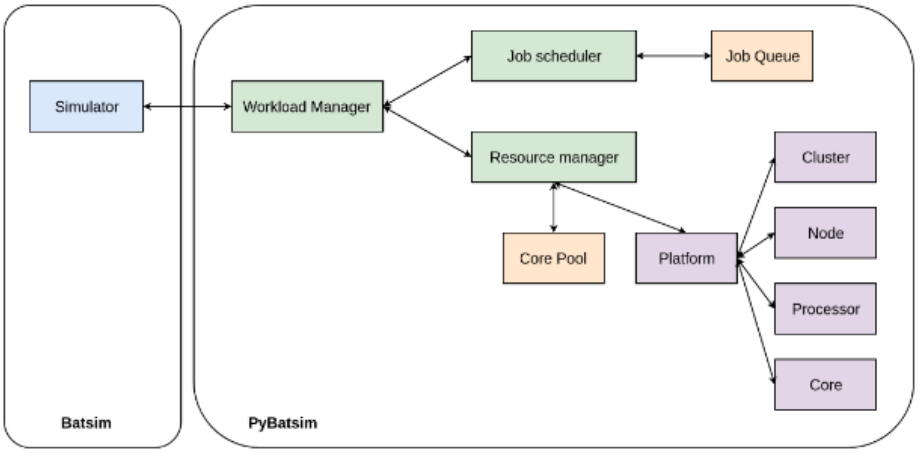

Figure 3.1: Batsim/Pybatsim Environment

Pybatsim provides a program to convert a workload in SWF format into the JSON format readable by Batsim. Unfortunately, the provided program is incomplete and does not correctly transform all fields from the SWF file into the JSON output.

I therefore rewrote the conversion script to make it more readable and modular, allowing easier selection of which fields to include. In my model, I needed to add the user ID and the task's group ID — both essential because the embedding layers in my model depend on them.

More specifically, the SWF format serves as a basic starting point for quickly prototyping a model. However, most studies show that they enrich their input data for training. Many are based on prior work demonstrating correlations between predicted execution times and user behavior. It thus seemed essential to introduce an intermediate step between SWF and JSON conversion.

This intermediate step would generate a new dataset format called *ESWF (Extended Standard Workload Format)*, where the first fields correspond to those of SWF, and

additional fields are customized according to the model's needs. This enriched dataset would then be used to train the model and converted into Batsim-compatible JSON format.

Custom fields are accessible via a `dict_json` attribute, which provides the JSON configuration of the task. This attribute is already integrated into PyBatsim, minimizing the required modifications to the tool itself.

This approach differs from newer formats such as *MWF*, which modify the original SWF fields directly, explained in Julita and Marco (2021).

3.2 User Time Estimation Model (UTEM)

In HPC systems, the time estimated by the user is a key decision factor for the task scheduler, especially in the Backfill algorithm, which seeks to fill resource gaps with short tasks that do not slow down the overall operation of the system.

In this context, I propose a supervised model that estimates, on behalf of the user, the execution time of a program.

3.2.1 Workloads summary

Log	**Jobs**	**Duration**
HPC2N	202,871	42 Months
$HPC2N_{10000}$	10,000	3 Months
CEA-Curie	312,826	20 Months

Table 3.1: The logs datasets of the sample computing clusters

3.2.2 Data Pre-processing

To develop a model, it is essential to understand the data being used. Studies have shown that users' historical parameters have a strong correlation with their behavior, and consequently, with their estimated execution times.

To address this, I relied on a recent study of Yonghao et al. (2024) where "previous studies have shown a robust correlation between user behavior and job runtime" and computes new features from SWF data. Machine learning methods have been widely used to predict the execution time of a task, and this work follows that line of research. Here are the different features used:

Characteristics ID	**Meaning**
submit_time	Time when job got submitted
req_time	User requested time
req_nodes	Number of nodes required
req_memory	Number of memory amount required
status	SWF status field
user_id	User id
group_id	Group id
think_time	Time interval between two job submissions
week	Week of the year submitted

Table 3.2: Fields computed from workloads

Some columns from SWF format has been deleted during the pre-processing process, either because information is unavailable, which is -1 in SWF datasets, or because it is meaningless for the model prediction.

Numerical features include log-transformed requested time, requested nodes and think time with week, submit time, status, requested memory. All numeric features are standardized using a StandardScaler from sklearn framework.

Categorical variables (such as user ID and group ID) are converted to integer indices via a top-k mapping. Let X be a categorical variable with domain $\mathcal{C}$, and let $f(c)$ denote the empirical frequency of category $c \in \mathcal{C}$. The top-k mapping selects the subset

$$\mathcal{C}_k = \{c \in \mathcal{C} \mid f(c) \text{ is among the } k \text{ highest frequencies}\}.$$

Each category $c \in \mathcal{C}_k$ is assigned a unique integer index, while all categories in $\mathcal{C} \setminus \mathcal{C}_k$ are mapped to a single special index representing an "other" category.

After this mapping, each categorical feature is represented through a learned embedding vector, using an embedding dimension of 20 for user IDs and 5 for group IDs. These numbers have been chosen by looking at the statistical analysis of the datasets in the parallel workload archive (Talby et al. 1999). Only few users and groups influence mainly the system.

The prediction target is the log-transformed job runtime (log1p(run time)). The preprocessing outputs a numeric feature matrix, indexed categorical features and a log runtime target.

3.2.3 Model Architecture

The model inputs are divided into two categories: numerical data and categorical data. Numerical variables are combined into a dense vector and then projected into a fixed-dimensional latent space to match the size of other model representations. Each categorical variable is converted into a dense embedding vector through a dedicated embedding layer, which learns a specific vector representation for each category value.

Each categorical input x_i with vocabulary size V_i is represented by a learned embedding matrix $E_i \in \mathbb{R}^{V_i \times d}$, and the embedding is

$$e_i = E_i[x_i].$$

The model then forms a sequence of tokens representing all job features. Each token corresponds to a variable (numerical or categorical) expressed in the same latent space. Additionally, this sequence is enriched by a *Column Embedding*, which helps identify the nature of each token (user, group, queue).

Once preprocessed, the inputs are passed through the Transformer encoder blocks using the *MultiHeadAttention* layer. This layer enables the model to learn contextual relationships among the different features (for example, user behavior on a specific queue type, job size, requested resources, etc.). This is the key advantage of the Transformer over traditional deep learning models, which usually process features independently.

Given a sequence of tokens $X \in \mathbb{R}^{n \times d}$, the self-attention layer computes

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right) V,$$

where

$$Q = XW^Q, \quad K = XW^K, \quad V = XW^V.$$

Each Transformer block contains an Feed-forward network (FFN)

$$\text{FFN}(x) = W_2\, \sigma(W_1 x + b_1) + b_2.$$

The complete model design is available in Appendix 1 and is divided into three main parts: input processing (A.1), Transformer block (A.2), and output layer (A.3). This model is the result of iterative experimentation throughout this research. A table of the neural architecture is also available in the appendix.

3.2.4 Training and Prediction

The model was trained using the Huber loss function. Several metrics were also used for result analysis, including the MAE and MAPE functions.

The Huber loss is a compromise between the Mean Squared Error (MSE) and the Mean Absolute Error (MAE). It behaves quadratically for small errors and linearly for large ones. This makes it particularly relevant in HPC environments, where execution times are highly heterogeneous—either very short or very long. The model thus becomes more robust to outliers and exhibits more stable learning behavior. Huber loss is defined as

$$HUBER_\delta = \begin{cases} \frac{1}{2}(y - \hat{y})^2, & \text{if } |y - \hat{y}| \leq \delta \\ \delta\left(|y - \hat{y}| - \frac{1}{2}\delta\right), & \text{otherwise,} \end{cases} \tag{3.2}$$

where y current predictions and $\hat{y}$ is the actual runtimes.

The MAE measures the mean absolute deviation between the actual and predicted values. In HPC scheduling, MAE is useful for quantifying the average deviation between predicted and real execution times. MAE is defined as

$$\text{MAE} = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}_i|, \tag{3.3}$$

where y_i the current prediction and $\hat{y}_i$ is the actual runtime of job i.

The MAPE expresses the average percentage error relative to the real execution time, providing a relative interpretation of model accuracy. For example, a MAPE of 20% means that, on average, predictions deviate by 20% from the actual execution times. MAPE is defined as

$$\text{MAPE} = \frac{100}{N} \sum_{i=1}^{N} \left| \frac{y_i - \hat{y}_i}{y_i} \right|, \tag{3.4}$$

where y_i the current prediction and $\hat{y}_i$ is the actual runtime of job i.

3.3 RL Model

Reinforcement learning has been studied for many years in the context of task scheduling, both in HPC and non-HPC systems. It enables an agent to explore an environment and improve its performance based on the actions it takes.

Most studies approach reinforcement learning as a way to create an entirely new scheduling policy. In this work, where multiple scheduling techniques are implemented, I chose to adopt a different approach. Rather than replacing the existing scheduling algorithm, I use the *EASY Backfill* algorithm and integrate an auxiliary optimization method to assist it. The goal is to improve and automate certain decision-making processes.

3.3.1 Padding Factor Selection

To achieve this, I designed an agent that optimizes a multiplicative factor applied to the execution time estimated by the supervised prediction model. The objective is to find the best factor that modifies the behavior of the Backfill algorithm, defined as

$$\tilde{t}_{run}(j) = \alpha \cdot \tilde{t}_{UTEM}(j), \tag{3.5}$$

where $\tilde{t}_{UTEM}(j)$ is the predicted execution time of job j from the *User Time Estimation Model (UTEM)*, and α is a multiplicative adjustment factor determined dynamically by the optimization method.

A larger α leads to a more conservative policy (jobs are assumed to take longer, filling fewer idle slots), while a smaller α leads to a more aggressive policy (allowing more backfilled jobs to be executed). The agent's objective is to learn the best trade-off between system utilization, job slowdown, and fairness by dynamically adjusting α.

In this work, a full workload trace is treated as one episode during training. Because the evaluation of each candidate padding factor is based on a full scheduling episode and does not involve step-by-step interaction or temporal credit assignment, the optimization problem does not constitute a reinforcement learning problem. This choice was made because the primary objective of the RL component is to explore how the padding factor influences overall scheduling performance at the scale of an entire trace. However, I do not consider this a limitation of the method itself. For example, reinforcement learning, and CEM, can also be applied using sliding-window episodes, where each episode corresponds to a fixed number of consecutive jobs (e.g., 1,000). Such a formulation provides more frequent updates and better reflects real-world online scheduling. The episodic formulation used here was therefore a deliberate simplification for the purpose of focusing on padding-factor optimization rather than exploring multiple RL variants.

The Cross-Entropy Method used here is not separate from reinforcement learning. It is a population-based policy search algorithm and therefore part of the RL family. In this thesis, I did not abandon reinforcement learning; rather, I selected CEM

because the optimization target—a single global padding factor—makes population-based search more stable and appropriate than temporal-difference methods. Future work could extend this RL formulation to dynamic or state-dependent padding strategies.

3.3.2 Cross-Entropy Method Implementation

To optimize the padding factor α used by the EASY Backfill scheduler, this work implements a lightweight variant of the Cross-Entropy Method (CEM), fully integrated into the Python scheduling environment. The CEM optimizer maintains a Gaussian search distribution $\mathcal{N}(\mu, \sigma^2)$ over the scalar parameter α. At each iteration, a population of N candidate values $\{\alpha_i\}_{i=1}^{N}$ is sampled from the current distribution and evaluated over one full scheduling episode in the simulator. Each episode executes the same discrete–event loop as Batsim, where candidate solutions influence job feasibility through the padded runtime prediction $\hat{t}_i = \alpha_i \cdot f(j)$, with $f(j)$ the model-based runtime estimate.

For each sampled α_i, the simulator computes a cumulative reward that combines system utilization, slowdown, backfilling overruns, and a strong penalty whenever the padding factor causes a delay of the head-of-queue job J_0. After evaluating all N candidates, the method selects the top M "elite" values according to their total reward. The distribution parameters are then updated as

$$\mu \leftarrow \frac{1}{M} \sum_{\alpha_i \in E} \alpha_i, \qquad \sigma \leftarrow \sqrt{\frac{1}{M} \sum_{\alpha_i \in E} (\alpha_i - \mu)^2},$$

where E denotes the elite set. This update step gradually shifts the search distribution toward padding factors that improve scheduling performance while maintaining the EASY Backfill guarantee that J_0 is never delayed. Iterations terminate once σ becomes

sufficiently small, indicating convergence to a near-deterministic estimate of the optimal padding factor $\alpha^\star$. The resulting value is then integrated into the scheduler as the fixed padding parameter used during the final evaluation in Batsim.

3.3.3 Markov Decision Process (MDP) Formulation

The problem can be modeled as a Markov Decision Process only if all three conditions hold. The first one is Full Observability or Markov Property. The state defined must contain all information needed to predict next state and to compute the reward at current time. Secondly, the second criterion, Stationary Transition Dynamics, states that the environment dynamics must not depend on time, only on the current state and action. Finally, the reward depends only on the current state, the current action and the next state.

Although the present work optimizes only a single global parameter and does not use a full state-dependent RL method, the MDP formulation is included here to provide the theoretical basis for potential future extensions involving full policy-learning approaches.

3.3.3.1 MDP Definition

I model the interaction between the scheduler and the computing cluster as a Markov Decision Process (MDP) $\mathcal{M} = (\mathcal{S}, \mathcal{A}, P, R, \rho_0, \gamma, T)$, where:

- $\mathcal{S}$ is the state space,
- $\mathcal{A}$ is the action space,
- $P(s_{t+1} \mid s_t, a_t)$ is the transition kernel,
- $R(s_t, a_t, s_{t+1})$ is the reward function,

- ρ_0 is the distribution of the initial state s_0,
- $\gamma \in [0, 1]$ is the discount factor (here $\gamma = 1$),
- T corresponding to the time when all jobs in the trace have been completed.

State space $\mathcal{S}$. At decision step t, the environment returns a continuous observation vector $s_t \in \mathbb{R}^{36}$ that summarizes the current cluster and queue configuration.

$$s_t = \left[f_t^{\text{free}},\, f_t^{\text{window}},\, f_t^{\text{qlen}},\, \overline{f}_t^{\text{req}},\, \overline{f}_t^{\text{pred}},\, \overline{f}_t^{\text{age}},\, \mathbf{f}_{t,1:K}^{\text{req}},\, \mathbf{f}_{t,1:K}^{\text{pred}},\, \mathbf{f}_{t,1:K}^{\text{age}} \right] \in \mathbb{R}^{36},$$

where f_t^{free} is the number of free nodes, f_t^{window} is the reservation window for J_0, f_t^{qlen} is the queue length, and the remaining components correspond to means and job features over the top K jobs in the queue (requested procs, prediction, age which is the difference between cluster time and job submit time).

Only jobs whose submission time is less than or equal to the current simulation timestamp are included in the state representation. Future arrivals are never exposed to the agent, even though Batsim internally stores such information. The state therefore reflects exactly the information available to a real scheduler at decision time: the currently running jobs, the free resources, and the subset of jobs that have already arrived in the queue.

During the experiment K was equal to 10.

Action space $\mathcal{A}$. The scheduler's action at each decision step is the *padding factor* α_t. The action space is the bounded interval

$$\mathcal{A} = [\alpha_{\min}, \alpha_{\max}] \subset \mathbb{R},$$

with $\alpha_{\min} = 1$ and $\alpha_{\max} = 15$ in our experiments.

Given an action $a_t = \alpha_t$, the scheduler uses $\alpha_t \hat{r}(j)$ as the effective walltime estimate when deciding whether a job can be safely backfilled without delaying J_0.

Transition kernel P. Given a state s_t and an action $a_t = \alpha_t$, the transition to the next state s_{t+1} is fully determined by the Easy Backfilling scheduler and the simulator.

In this work, I approximate the true (very high-dimensional) cluster state with the 36-dimensional feature vector s_t defined above, and treat the resulting process as an MDP.

Initial state distribution ρ_0 and horizon T. The initial state is fully determined by the simulator. At time 0, no job is running and all jobs submitted at timestamp 0 are in the queue. The simulation ends when all jobs in the workload trace have completed.

Control objective. Given a policy $\pi(a_t \mid s_t)$, the return of an episode is the undiscounted cumulative reward. The reinforcement learning objective here is to find a policy $\pi^\star$ that maximizes the expected return.

In this work, I restrict π to a simple one-parameter family that selects a constant padding factor α for the entire episode.

Reward function. It is also possible to modify the behavior of the algorithm by changing the parameters of the defined reward function

$$\begin{aligned} R_t = w_u \cdot \Delta\text{Utilization}_t - w_s \cdot \Delta\text{AvgSlowdown}_t \\ - w_d \cdot \mathbf{1}\{J_0 \text{ delayed at t}\} - w_o \cdot OverrunRate_t, \end{aligned}$$

where:

- **Utilization** represents the change in total resource usage (e.g., number of occupied nodes) since the previous step. We use the notation $U_t := U(t)$ for the utilization evaluated at event time t and U_{t-1} for the utilization just before processing the event at time t.

$$\Delta\text{Utilization}_t = U_t - U_{t-1}. \tag{3.6}$$

- **AvgSlowdown** is the mean slowdown ratio averaged over completed jobs. Let $\mathcal{C}_t$ be the set of jobs whose completion events are processed at time t. It is defined as

$$\text{AvgSlowdown}_t = \begin{cases} \dfrac{1}{|\mathcal{C}_t|} \displaystyle\sum_{j \in \mathcal{C}_t} \text{slowdown}(j), & \text{if } |\mathcal{C}_t| > 0, \\ 0, & \text{otherwise.} \end{cases} \tag{3.7}$$

- **Delay penalty** applies when the first queued job J_0 is delayed; higher weights make this a stronger constraint. This logic is formally defined as

$$\mathbf{1}\{J_0 \text{ delayed at } t\} = \begin{cases} 1, & \text{if } \alpha \text{ postpones the earliest feasible start of } J_0, \\ 0, & \text{otherwise.} \end{cases} \tag{3.8}$$

- **OverrunRate** represents the fraction of backfilled jobs whose runtime exceeded their predicted padded execution time. We consider that a job overruns its padded prediction if

$$r_j > \tilde{t}_j(\alpha). \tag{3.9}$$

Let $\mathcal{B}_t$ be the set of backfilled jobs that complete in $(t-1, t]$. The overrun rate at time t is defined as:

$$\text{OverrunRate}_t = \begin{cases} \dfrac{1}{|\mathcal{B}_t|} \left| \left\{ j \in \mathcal{B}_t \,\middle|\, r_j > \tilde{t}_j(\alpha) \right\} \right|, & \text{if } |\mathcal{B}_t| > 0, \\ 0, & \text{otherwise.} \end{cases} \tag{3.10}$$

where $\hat{r}(j)$ is job j predicted runtime, and by $\tilde{t}_j(\alpha)$ the padded estimate used by the scheduler.

The model is trained using an environment based on the OpenAI Gym library. It is trained directly on datasets in the SWF format with first user time prediction, then the UTEM model time prediction and integrated into the Batsim simulation ecosystem. It was trained with weights $w_u = 1.0$, $w_s = 0.25$, $w_d = 25.0$ and $w_o = 0.1$. These hyperparameters encourage policies that increase overall utilization while avoiding large slowdowns, delaying the first job in the queue, or incurring frequent backfill overruns.

The reward R_t is thus a weighted combination of four quantities:

- $\Delta\text{Utilization}_t$: encourages better resource usage,
- $\Delta\text{AvgSlowdown}_t$: penalizes increased slowdown and unfairness,
- $\mathbf{1}\{J_0 \text{ delayed at } t\}$: heavily penalizes head-of-queue delays,
- OverrunRate_t: penalizes unsafe underestimation of runtimes.

The weights (w_u, w_s, w_d, w_o) are chosen to reflect the typical priorities of HPC centers: high utilization is desirable, but not at the expense of fairness for the first job in the queue or of frequent runtime overruns. The resulting scalar reward provides the agent with a meaningful signal for evaluating and improving the padding factor α over full scheduling episodes.

3.3.3.2 MDP Assumptions

Using an MDP formulation requires more than defining a state, an action, and a reward. We must verify that the scheduling problem, as represented through the features available to the agent, satisfies the assumptions of reinforcement learning. In particular, the question is whether the state representation used in this work is sufficiently Markovian for the padding-factor control objective.

Markov Property. A process is Markovian if the current state contains all the information that is necessary to predict the next state and the immediate reward defined as

$$P(s_{t+1} \mid s_1, \ldots, s_t, a_t) = P(s_{t+1} \mid s_t, a_t).$$

In a real HPC system, the full cluster state is extremely high-dimensional and includes information that the agent cannot observe (internal scheduling metadata, execution internals from Batsim, detailed network or memory behavior, etc.). However, the agent in this work does not attempt to learn the entire scheduling policy; it only adjusts a *single scalar parameter*, the padding factor α_t. This parameter affects scheduling decisions exclusively through the padded execution time $\alpha_t \hat{r}(j)$.

EASY Backfilling relies only on:

- the number of free nodes,
- the reservation window of the first job J_0,
- the queue length,
- the requested resources, predicted runtimes, and ages of the next jobs in the queue.

These fields are those included in s_t. For this specific control problem, no other information is required. All hidden internal details maintained by Batsim (true runtime counters) influence the observable state only through changes in queue structure, job completions, and resource availability, all of which are captured in s_{t+1}.

Thus, *conditional on* (s_t, α_t)*, the simulator deterministically produces* s_{t+1}, making the reduced process Markovian with respect to the agent's decision variable.

Stationary Dynamics. The transition dynamics are also stationary:

- job arrivals are fixed by the SWF trace,
- EASY Backfilling uses deterministic, time-invariant logic,
- Batsim processes events using a deterministic event-based engine,
- the effect of α_t on the padded runtime is constant across time.

Therefore, $P(s_{t+1} \mid s_t, a_t)$ does not depend on t.

Reward Dependence. The reward is constructed to depend only on (s_t, a_t, s_{t+1}):

- utilization changes come from the difference between s_t and s_{t+1},
- slowdown is computed from jobs completed during the transition,
- the delay of J_0 is fully determined by the decision at time t,
- the overrun rate is observable once Batsim reveals the true runtime when a job finishes.

This satisfies the MDP reward definition.

Although the internal state of an HPC simulator is larger than what the agent observes, the padding-factor control problem depends only on the decision-relevant

subset of features, all of which are included in s_t. The system's dynamics are stationary, and the reward depends exclusively on the transition induced by the current action.

For these reasons, the problem can be treated as an *approximate MDP*.

A problem is defined as an MDP rather than a POMDP (Kaelbling et al. 1998), which generalize MDP definition, if something essential is hidden form the agent, such as unknown true runtimes or queue features incomplete for example.

3.4 Batsim Implementation

For implementation, the *EASY Backfill* algorithm was used as the base, as it remains the most popular scheduling algorithm in HPC systems due to its simplicity and intuitive logic. One of the main goals of this work is not to make the scheduling system more complex, but to provide additional tools that make it more automated and adaptive.

3.4.1 Pre-processing

The only pre-processing step required is the conversion of the SWF workload format into the JSON format that can be read and interpreted by Batsim. This conversion ensures that all relevant job attributes — such as submission time, requested resources, and user information — are preserved for use by both the supervised and padding factor models.

3.4.2 UTEM Inference

The trained supervised model is loaded into the Batsim ecosystem using TensorFlow and Keras libraries. It is also necessary to load the corresponding artifacts (numerical and categorical input mappings) in order to perform inference correctly.

I implemented a custom *RuntimePredictor* class that constructs the numerical and categorical input vectors from the workload file, computes the predicted walltime for each newly submitted job, and sends this job to the Batsim instance for scheduling and execution.

3.4.3 CEM Scheduler

To integrate the CEM model, the EASY Backfill algorithm was modified to include the adaptive adjustment mechanism.

During each scheduling event, only the jobs that have already arrived are inspected, and their features are passed to the RL state encoder. Jobs with future submission timestamps are ignored until they enter the system.

In the final version, the supervised runtime prediction model is incorporated, providing more accurate input for the CEM agent. The resulting scheduler computes the optimal α factor based on the different workloads. This hybrid integration of supervised and padding factor optimization techniques enables a more intelligent and automated scheduling system that adapts dynamically to workload conditions in real time.

Chapter 4

Results and Discussions

4.1 User Time Estimation Model Results

During training, I used *Weights & Biases (WandB)*, an online platform that helps track, analyze, and interpret machine learning experiments. (Biewald 2020)

The model was trained on the **HPC2N** and **CEA-Curie** workloads, two larges and cleaned SWF dataset. I created a smaller workload $HPC2N_{10000}$ which contains only the first 10000 jobs in the dataset. The purpose was to find if the model can predict well with a tiny workload. These two workloads are commonly used in the literature, which makes it easier to compare the performance of different models under similar conditions.

4.1.1 Time-Aware Cross-Validation

Runtime predictions is a time-dependent problem: job arrivals follow temporal patterns, the state could evolve over time. For these reasons, the classical cross-validation technique is not applicable in our context because of (i.i.d.). It would violating causality and leads to information leakage, as noted in the time-series literature Bergmeir et al. (2018); Cerqueira et al. (2020).

So, I adopt a time-aware cross-validation which keep the chronological order of the workload trace. For fold k we define

$$\text{Train}_k = \{j_1, \ldots, j_{T_k}\},$$
$$\text{Validation}_k = \{j_{T_k+1}, \ldots, j_{T_k+V}\},$$
$$\text{Test}_k = \{j_{T_k+V+1}, \ldots, j_{T_k+V+H}\},$$

where T_k increases across folds, H is the number of jobs in testing subsets and V the number of jobs in validation subsets. This is crucial because it ensures that each test set follows its corresponding training set in time. It prevents information leakage from future job behaviors. It also provides a more reliable and realistic assessment of generalization performance.

4.1.2 Training

The model has been initially training for 400 epochs and 5 folds for one hour of execution on my personal laptop. During training, I used two *callbacks*:

- **Early Stopping:** Training stops automatically when no further improvement is observed. The patience parameter was set to 15 epochs.
- **Learning Rate Reduction on Plateau:** When a plateau is detected in validation metrics, the learning rate is reduced to stabilize and improve convergence. It was configured with a patience of 5, a factor of 0.5 and a minimum learning rate of $3e-5$.

These two callbacks make the run go about a 100 epochs. During the remaining part of this section, only training on $HPC2N_{10000}$ has been shown for more clarity because of the dataset size. I tuned the attention dropout, the FFN dropout and the embedding dropout defined as

$$d_a \in \{0.05, 0.075, 0.1, 0.15\},$$

$$d_{ffn} \in \{0.05, 0.075, 0.1, 0.15\},$$

$$d_e \in \{0.05, 0.075, 0.1, 0.15\}.$$

The different sizes of each hidden layer. I've also worked on the number of embedded categorical features for the user id and the group id. I've also tunned the number of attention heads such as

$$h \in \{2, 4, 8\}.$$

I have also tuned the delta parameter from the Huber loss function, it helps to control robustness to outliers. The delta parameter has been test with

$$\delta \in \{0.2, 0.5, 0.8, 1, 2\}.$$

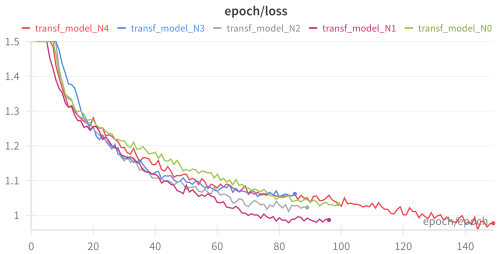

Figure 4.1: UTEM training Huber loss as a function of epochs.

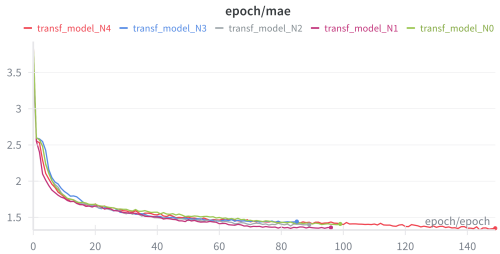

Figure 4.2: UTEM training MAE as a function of epochs.

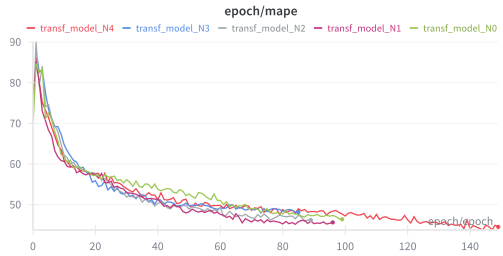

Figure 4.3: UTEM training MAPE as a function of epochs.

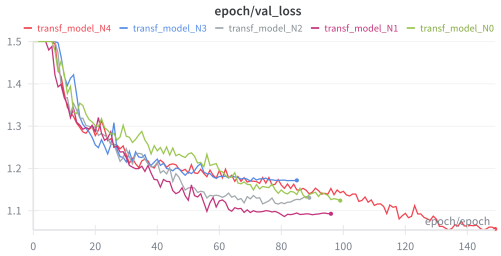

Figure 4.4: UTEM validation Huber loss as a function of epochs.

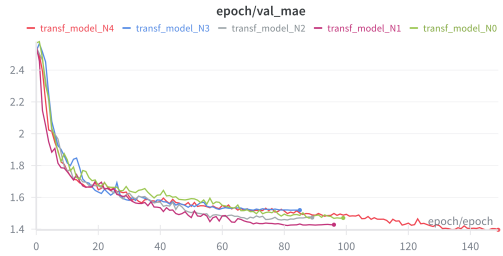

Figure 4.5: UTEM validation MAE as a function of epochs.

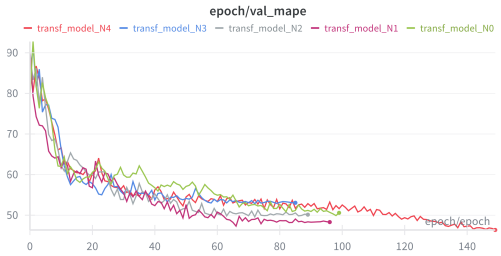

Figure 4.6: UTEM validation MAPE as a function of epochs.

As shown in the figures 4.1, the model learns effectively during training. The validation loss in figure 4.4 follows a similar trend to the training loss, indicating good generalization and stable learning.

We can see that metrics show in figures 4.2, 4.3, 4.5 and 4.6 are evolving similarly as the loss function.

After training, I evaluated the model on each fold test set, yielding the following mean results per dataset:

Workload	Testing Loss	Testing MAE	Testing MAPE (%)
$HPC2N_{10000}$	1.19	1.55	59.05
HPC2N	1.17	1.52	57.97
CEA-Curie	1.14	1.49	55.51

Table 4.1: UTEM Testing Results

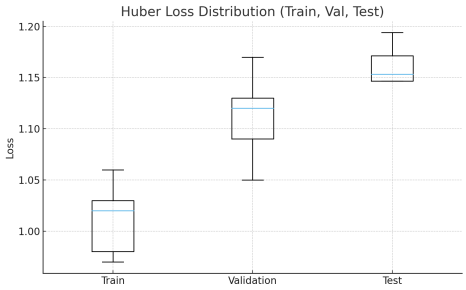

Figure 4.7: UTEM cross-validation Huber loss distribution across models.

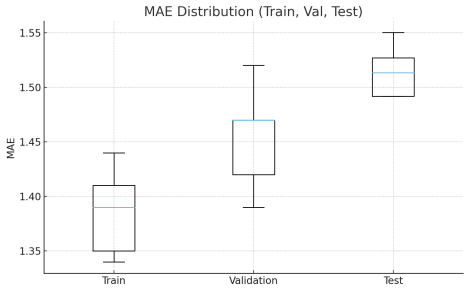

Figure 4.8: UTEM cross-validation MAE distribution across models.

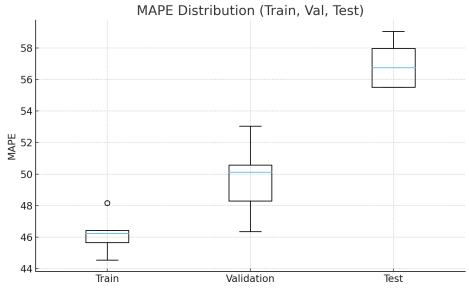

Figure 4.9: UTEM cross-validation MAPE distribution across models.

For comparison, I also computed the MAE and MAPE for user-provided estimated times without using the model. The errors were significantly larger — strongly biased by a few extreme outliers, such as jobs with predicted runtimes of 432,000 seconds that actually completed in 2 seconds. Interestingly, these jobs were still marked as *completed*, suggesting that some users intentionally overestimate runtimes or forget to update their estimates before submission.

4.1.3 Batsim

When integrated into the Batsim simulation environment, the results revealed that approximately **64.42%** of jobs were killed due to underestimated runtimes produced by the model. This highlights a major issue of over-aggressive predictions, which the reinforcement learning model aims to mitigate through the adaptive padding factor α introduced earlier.

4.1.4 Analysis

Overall, the supervised model demonstrates good learning capability but tends to predict runtimes that are too short. Similar findings in the literature suggest developing a secondary model that estimates the confidence or uncertainty of the primary prediction. However, in this work, I address this limitation using a reinforcement learning model that dynamically adjusts predictions to achieve a better balance between accuracy and robustness.

I statistically analysed the error distribution on the HPC2N workload. Let r_i the actual runtime of job i and $\hat{r}_i$ the corresponding user estimate. For each job we define the absolute error and absolute percentage error as

$$e_i = |\hat{r}_i - r_i|, \qquad \text{APE}_i = \frac{|\hat{r}_i - r_i|}{r_i}. \tag{4.1}$$

Given a set of N jobs, we compute the mean absolute error (MAE) and mean absolute percentage error (MAPE) as

$$\text{MAE} = \frac{1}{N}\sum_{i=1}^{N} e_i, \qquad \text{MAPE} = \frac{1}{N}\sum_{i=1}^{N} \text{APE}_i. \tag{4.2}$$

Table 4.2: Descriptive statistics of user runtime estimate errors on the HPC2N workload.

Metric	Mean	Std. deviation	95% CI
MAE (seconds)	17,019.87	44,317.88	[16,827.02, 17,212.72]
MAPE (ratio)	1.1087	1.9673	[1.1001, 1.1173]

Table 4.2 summarises the descriptive statistics for the user estimate errors on HPC2N. On average, user-specified runtimes deviate from the actual runtime by more than one hundred percent. This tells a tendency of users to overestimate requested runtimes.

4.2 CEM Scheduler

The objective of the CEM-based scheduler is to learn the optimal scaling factor α that improves the overall success rate of job executions while maintaining acceptable performance across other metrics such as turnaround time, waiting time, and slowdown.

The following figures illustrate the success rate and other metrics as functions of the scaling factor α:

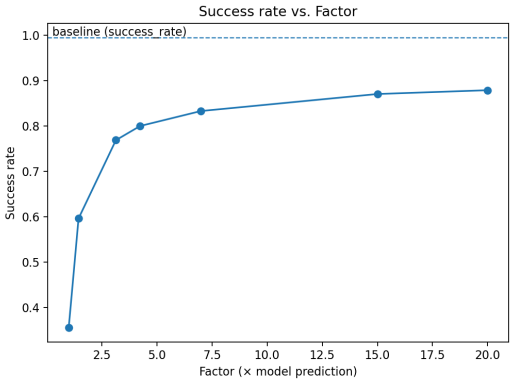

Figure 4.10: Success Rate vs Factor

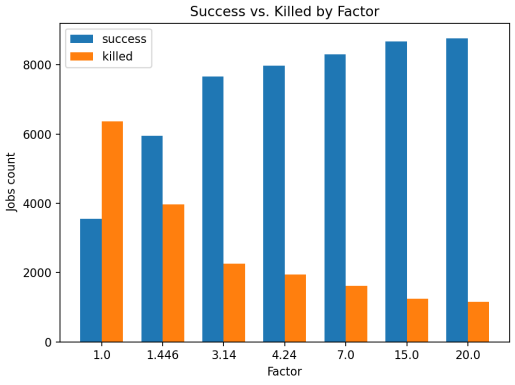

Figure 4.11: Jobs Killed vs Factor

The results show in figures 4.10 and 4.11 that the success rate increases rapidly for small factors but struggles to reach 100% even for larger values of α. Nevertheless, as the factor grows, the success rate improves substantially, while other metrics (such as slowdown and waiting time) do not deteriorate significantly, making the model clearly superior to the baseline.

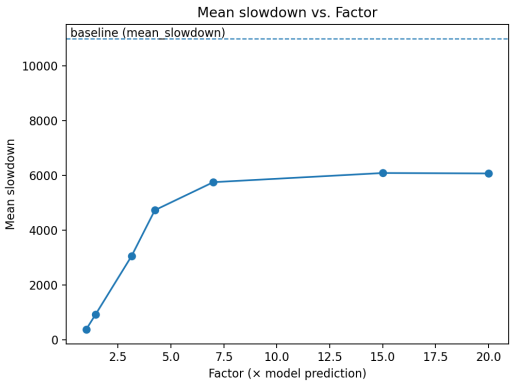

Figure 4.12: Mean Slowdown vs Factor

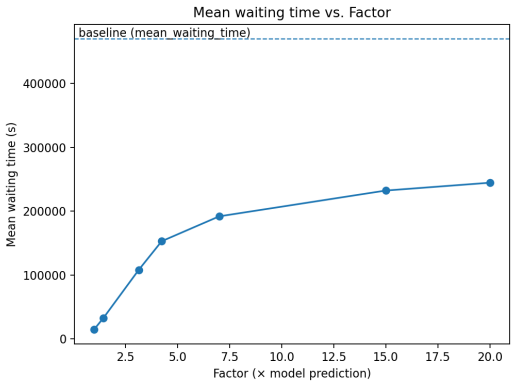

Figure 4.13: Mean Waiting Time vs Factor

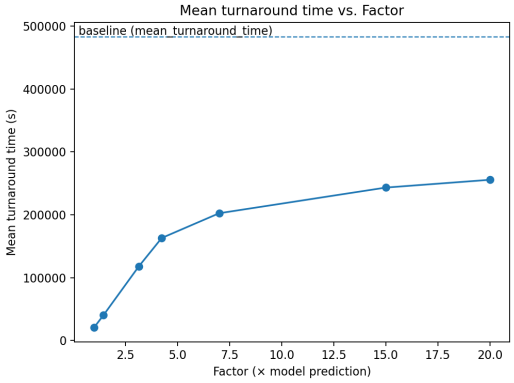

Figure 4.14: Mean Turnaround Time vs Factor

As shown in the figures4.12, the mean slowdown remains two to three times better than the baseline, while the mean waiting time improves by a factor of three to five in figure 4.13. The same improvement ratio is observed for the mean turnaround time in figure 4.14, demonstrating that the reinforcement learning scheduler achieves significantly better performance across key metrics without compromising efficiency.

4.3 Batsim Performance

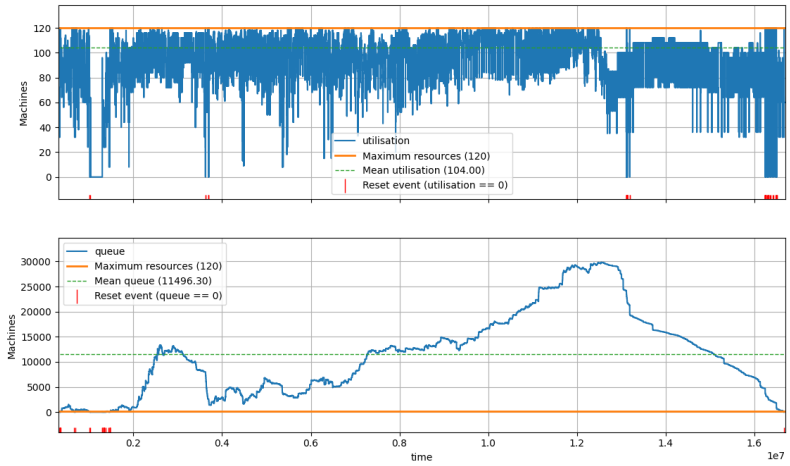

Figure 4.15: System Resources Utilization without models. The x-axis represents the time and the y-axis shows machine utilization (nodes and resources)

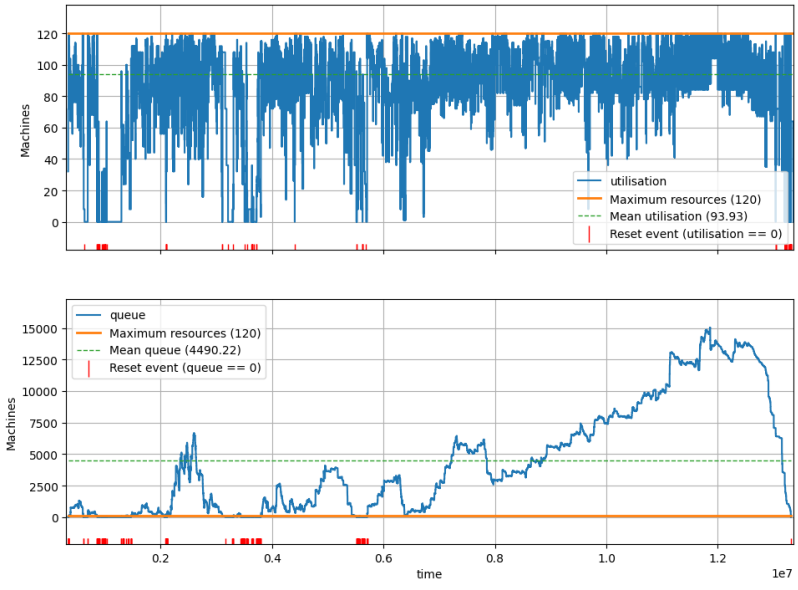

Figure 4.16: System Resources Utilization with models. The x-axis represents the time and the y-axis shows machine utilization (nodes and resources)

We can observe in these figures 4.15 and 4.16 the overall machine utilization. At the beginning of the simulation, the model performs better by using fewer resources and releasing them more quickly. During peak utilization periods, the model also tends to allocate all available resources, in contrast to the baseline case where only about 100 out of 120 nodes were actively used.

Overall, the total queued resources never exceed 15,000 with the model, compared to 30,000 without it, resulting in an approximate improvement of 100% in resource efficiency.

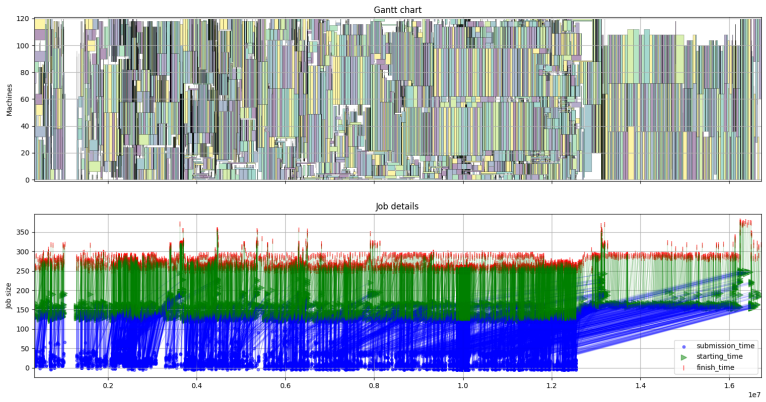

Figure 4.17: Gantt chart of jobs scheduled without models. The x-axis represents time. The first y-axis shows nodes assigned. The second y-axis shows job size.

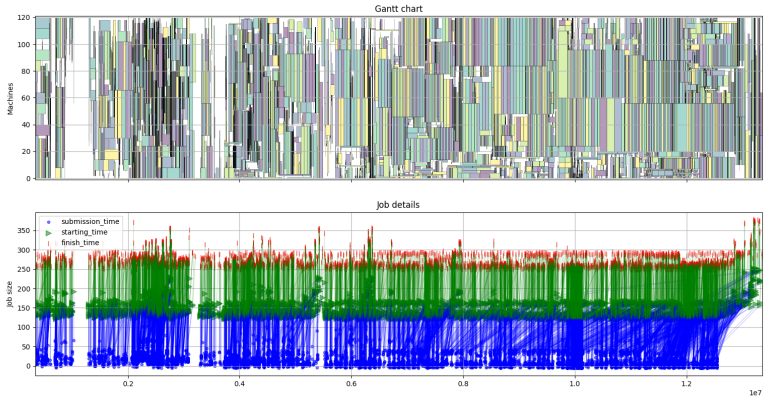

Figure 4.18: Gantt chart of jobs scheduled with models. The x-axis represents time. The first y-axis shows nodes assigned. The second y-axis shows job size.

In the Gantt chart in figure 4.18, we can observe that the model's predictions help the scheduler to handle the final jobs more efficiently. In contrast, in the first chart in figure 4.17, the larger queue suggests that the scheduler struggles to allocate resources and schedule jobs promptly. This is particularly evident from the significant gap between the job submission time and its actual start time, indicating longer waiting periods and lower scheduling efficiency.

However, we could also assume that the model struggles to schedule the final jobs, which are significantly larger, as can be seen in the first Gantt chart. If this assumption is correct, the behavior is likely due to the fact that such large jobs did not appear during the initial part of the workload and were introduced suddenly near the end, making them more difficult for the model to handle effectively.

However, we can also see that the model performs better at the beginning of the workload. The job arrival chart shows a more balanced and padded scheduling pattern, with additional gaps indicating that resources were managed more flexibly and efficiently during the early stages.

4.4 Statistical Evaluation of Scheduling Results

4.4.1 Descriptive Performance Comparison

To assess the performance and robustness of the developed scheduler, we compare it against a baseline EASY Backfill scheduler that relies on user-provided runtime estimates. Both schedulers are evaluated on the test sets of the same HPC2N workload and under the same set of random seeds. For each seed, we record four aggregate performance metrics: mean slowdown, mean waiting time, mean turnaround time, and overall success rate.

Let x_j^{base} denote the value of a metric for the baseline scheduler under seed j, and x_j^{CEM} the corresponding value for the CEM + UTEM scheduler. We define the paired difference

$$d_j = x_j^{\text{base}} - x_j^{\text{CEM}}, \tag{4.3}$$

so that positive values indicate that the CEM + UTEM scheduler achieves a lower value for the metric. For each metric, we compute the sample mean and standard deviation of these paired differences,

$$\bar{d} = \frac{1}{n}\sum_{j=1}^{n} d_j, \qquad s_d = \sqrt{\frac{1}{n-1}\sum_{j=1}^{n}(d_j - \bar{d})^2}, \tag{4.4}$$

where n is the number of seeds. Assuming approximate normality of the paired differences, a two-sided 95% confidence interval for the mean difference is given by

$$\bar{d} \pm t_{0.975,n-1} \cdot \frac{s_d}{\sqrt{n}}. \tag{4.5}$$

To complement the confidence interval analysis, we also report the effect size using Cohen's d for paired samples,

$$d_{\text{Cohen}} = \frac{\bar{d}}{s_d}. \tag{4.6}$$

Table 4.3 reports the mean performance across all seeds for both schedulers, while Table 4.4 summarizes the observed paired differences, their variability, confidence intervals, and effect sizes.

Table 4.3: Mean performance of the baseline EASY Backfill and the CEM + UTEM scheduler on the HPC2N workload.

Metric	Baseline (mean)	CEM + UTEM (mean)
Slowdown	10,876.83	5,799.29
Waiting time (s)	469,336.48	199,616.86
Turnaround time (s)	482,854.19	188,003.29
Success rate	0.9871	0.8043

Table 4.4: Paired differences (baseline minus CEM + UTEM) for the HPC2N workload.

Metric	Mean difference	Std. dev.	95% CI	Cohen's d
Slowdown	5,077.54	306.14	[4,794.41, 5,360.67]	16.59
Waiting time (s)	269,719.62	3,092.38	[266,859.65, 272,579.60]	87.22
Turnaround time (s)	294,850.91	11,510.40	[284,205.56, 305,496.26]	25.62
Success rate	0.1829	0.0221	[0.1624, 0.2033]	8.26

We formally evaluate statistical significance using two-sided paired t-tests in Section 4.4.2.3. The descriptive analysis presented here shows that the CEM + UTEM scheduler substantially reduces slowdown, waiting time, and turnaround time relative

to EASY, while also exhibiting a lower success rate. The decrease in success rate is expected: the baseline scheduler uses heavily overestimated walltimes, which minimizes job kills at the expense of resource utilization and queueing delay. In contrast, the CEM + UTEM scheduler relies on more realistic runtime estimates and therefore schedules more tightly, exposing a small fraction of jobs to the risk of exceeding their reserved time.

Even with this trade-off, the magnitude of the improvements in slowdown, waiting time, and turnaround time indicates that the CEM + UTEM scheduler provides a significantly more efficient and responsive scheduling policy.

4.4.2 Hypothesis Testing

To provide rigorous support for the research questions, the performance of the supervised prediction model and the scheduling policies is evaluated using formal statistical hypothesis testing, in addition to the descriptive statistics presented in the earlier sections.

4.4.2.1 Runtime Prediction: UTEM vs. User Estimates

Let e_i^{user} and e_i^{UTEM} denote the absolute runtime errors for job i based on the user-provided estimate and the UTEM prediction, respectively. For each job, we define the paired difference

$$d_i = e_i^{\text{user}} - e_i^{\text{UTEM}},$$

so that positive values indicate lower error under UTEM.

To evaluate whether the mean difference in errors differs from zero, we apply a two-sided paired t-test with hypotheses

$$H_0 : \mathbb{E}[d_i] = 0, \qquad H_1 : \mathbb{E}[d_i] \neq 0.$$

This formulation tests for any systematic difference between the two predictors, without assuming in advance that UTEM improves accuracy. The statistical significance is assessed jointly with descriptive metrics such as MAE and MAPE, as well as 95% confidence intervals for the mean error difference. Across all workloads considered, the paired t-test rejects H_0 with $p \ll 0.01$, and the positive mean difference confirms that UTEM yields lower error on average than user estimates.

4.4.2.2 Scheduler Comparison: CEM + UTEM vs. EASY

Let x_j^{base} and x_j^{CEM} denote the measured value of a performance metric (slowdown, waiting time, turnaround time, or success rate) under the EASY scheduler and the CEM + UTEM scheduler, respectively. For each metric and simulation seed, we define the paired difference

$$d_j = x_j^{\text{base}} - x_j^{\text{CEM}}.$$

A two-sided paired t-test is used to evaluate whether the mean paired difference differs from zero:

$$H_0 : \mathbb{E}[d_j] = 0, \qquad H_1 : \mathbb{E}[d_j] \neq 0.$$

This testing framework does not assume that CEM + UTEM must outperform the baseline; instead, the direction of the effect is determined from the sign of the sample mean $\bar{d}$ whenever the null hypothesis is rejected. For each metric, we report $\bar{d}$, its 95% confidence interval, and Cohen's d.

Consistent with the confidence intervals shown in Table 4.4, the paired t-tests reject H_0 for slowdown, waiting time, and turnaround time with $p < 10^{-4}$, indicating statistically significant mean reductions relative to EASY. The success rate exhibits a statistically significant mean decrease ($p < 10^{-4}$), reflecting the trade-off introduced by a more aggressive backfilling policy.

4.4.2.3 Period-Based test

To further evaluate the statistical significance of the improvements obtained by the CEM-optimized padding factor, we conduct a period-based paired t-test. The seven independent simulations run under both the CEM-enhanced scheduler and the baseline EASY policy are treated as matched evaluation periods. For each period k and each performance metric, we compute the paired difference

$$d_k = x_k^{(\text{baseline})} - x_k^{(\text{CEM})}.$$

Table 4.5: Period-based paired t-test comparing CEM scheduler and EASY baseline.

Metric	**Mean Diff.**	**Std Dev**	t**-statistic**	p**-value**
Slowdown	-5077.55	302.88	-42.00	$p \ll 0.001$
Waiting Time	-2.692×10^5	2.692×10^3	-257.40	$p \ll 0.001$
Turnaround Time	-2.937×10^5	1.199×10^4	-64.99	$p \ll 0.001$
Success Rate	-0.183	0.0199	-17.20	$p < 0.001$

The results show that the CEM-adjusted scheduler achieves statistically significant improvements over the EASY baseline for all primary response-time-related metrics.

Slowdown, waiting time, and turnaround time exhibit strongly negative mean differences, confirming consistent reductions across periods. The success rate shows a statistically significant mean decrease, illustrating the expected trade-off associated with tighter padding-factor optimization. Overall, the statistical analysis demonstrates that the CEM-optimized padding factor provides robust and systematic performance gains while introducing a controlled impact on job completion reliability.

4.5 Limitations and Future Work

Traditional HPC schedulers such as FCFS and EASY rely on user estimates. These estimates are often inaccurate, which leads to job overruns and inefficient backfilling. The approach here was to incorporate a learned runtime predictor that provides more reliable estimates.

The proposed hybrid approach (Machine Learning and Reinforcement Learning) outperforms traditional schedulers, heuristic padding strategies, and random policies. The neural predictor provides more accurate estimates of job durations while the Cross-Entropy Method makes the whole scheduling more safer in system view. Unlike fixed heuristics or ML-only which based their research only on loss metrics, my approach implements a reproducible way to train agents through a simulator.

In the present study, each workload trace was treated as a single episode. While this simplifies experimentation, a sliding-window episode structure—evaluating the policy every fixed number of jobs would be more aligned with real-world online schedulers and would enable more frequent updates of the CEM. This approach would also reduce return variance and accelerate convergence. The method presented in this thesis does

not preclude this extension, rather, it focuses on establishing the viability of padding-factor optimization. Exploring sliding-window episodes constitutes a natural direction for future reinforcement-learning-based schedulers.

The goal of this work was not to give a state of the art regression model, but to implements multiples machines learning techniques that makes the scheduling policy better. We can found very promising other regressor model such as Transformer using time encoded data related to the user behavior. My main contribution is more showing end-to-end scheduling improvements.

This work presents several limitations. First, the prediction model achieves only moderate accuracy, which reduces its potential impact on scheduling performance. Second, the padding factor is optimized offline and remains fixed during the simulation, which limits the system's ability to adapt to variations in the workload.

A limit is visible concerning the trade-off between timeliness and reliability. While the CEM method reduces average slowdown (from 10,876.83 to 5,799.29), this improvement comes at the cost of a lower success rate (from 0.9871 to 0.8043). The more aggressive backfilling increases the risk of violated runtime predictions, which in turn leads to job cancellations or requeueing. The scheduler prioritizes faster job completions at the expense of completion reliability. Future work should investigate mechanisms, such as adaptive padding or confidence-aware backfilling.

For future work, several directions can be explored: improving the prediction model, implementing an adaptive or online optimization of the padding factor, and extending the agent beyond a single parameter in order to model a richer scheduling policy. These developments would allow for a more detailed evaluation of the approach's potential in diverse HPC environments.

Chapter 5

Conclusion

For this study, several machine learning techniques were implemented in HPC systems for task scheduling, with a particular focus on improving the *EASY Backfill* scheduling algorithm.

A supervised deep learning regression model was developed using *TabTransformers* to estimate the execution time of jobs, replacing the user-provided runtime estimates. The implemented model achieved a MAPE score of 57%, which is a promising result in HPC workload environments, where prediction errors typically range between 40% and 80%.

In addition, a padding factor optimization model was implemented to make the supervised model's predictions more accurate and less aggressive. This approach significantly increased the job success rate from 39% to 82% by dynamically adjusting an α padding factor.

There is always something we can improve. In particular, future work could explore the use of additional machine learning techniques to further optimize task scheduling in HPC systems. For example, by implementing classification-based approaches or more advanced reinforcement learning methods capable of defining entirely new scheduling policies.

Reference List

Abadi, M., A. Agarwal, P. Barham, E. Brevdo, Z. Chen, and Coauthors, 2015: Tensorflow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org, Accessed: 2025-11-11, https://www.tensorflow.org/.

Avery, A., and A. Savakis, 2023: Deeprm: Deep recurrent matching for 6d pose refinement. *2023 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, 6206–6214, https://doi.org/10.1109/CVPRW59228.2023.00660, accessed: 2025-11-08.

Bergmeir, C., R. J. Hyndman, and J. M. Benítez, 2018: Machine learning methods for time series forecasting with discrete dependent variables. *Statistical Science*, **33 (3)**, 414–434, accessed: 2025-11-11.

Bienia, C., 2011: Benchmarking modern multiprocessors. *Proceedings of the 2011 IEEE International Symposium on Workload Characterization (IISWC)*, IEEE, 1–2, accessed: 2025-11-12.

Biewald, L., 2020: Experiment tracking with weights and biases. Software available from wandb.com, Accessed: 2025-11-11, https://www.wandb.com/.

Casanova, H., 2001: Simgrid: a toolkit for the simulation of application scheduling. *Proceedings First IEEE/ACM International Symposium on Cluster Computing and the Grid*, 430–437, https://doi.org/10.1109/CCGRID.2001.923223, accessed: 2025-11-08.

Cerqueira, V., L. Torgo, and I. Mozetič, 2020: Evaluating time series forecasting models: an empirical study on cross-validation and hyperparameter tuning. *Machine Learning*, **109**, 1997–2028, accessed: 2025-11-11.

Chollet, F., and Coauthors, 2021: Keras: The python deep learning library. *Journal of Machine Learning Research*, **22 (55)**, 1–6, URL https://keras.io, accessed: 2025-11-11.

Dutot, P.-F., M. Mercier, M. Poquet, and O. Richard, 2017: Batsim: A realistic language-independent resources and jobs management systems simulator. *Job Scheduling Strategies for Parallel Processing*, N. Desai, and W. Cirne, Eds., Springer International Publishing, Cham, 178–197, accessed: 2025-11-08.

Eric, G., G. David, R. Valentin, and T. Denis, 2015: Improving backfilling by using machine learning to predict running times. *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 1–10, https://doi.org/10.1145/2807591.2807646, accessed: 2025-11-08.

Fan, Y., and Coauthors, 2022: Dras: Deep reinforcement learning for cluster scheduling in high performance computing. *IEEE Transactions on Parallel and Distributed Systems*, **33 (12)**, 4903–4917, https://doi.org/10.1109/TPDS.2022.3205325, accessed: 2025-11-08.

Fengxian, C., 2023: Job runtime prediction of hpc cluster based on pc-transformer. -, URL https://link.springer.com/article/10.1007/s11227-023-05470-2, accessed: 2025-11-08.

GitHub, 2025: Azure public dataset: Azure lmm inference trace 2025. https://github.com/Azure/AzurePublicDataset/tree/master, Accessed: 2025-11-13.

Hintjens, P., 2013: *ZeroMQ: Messaging for Many Applications*. O'Reilly Media, accessed: 2025-11-11.

Julita, C., and D. Marco, 2021: Modular workload format: Extending swf for modular systems. *Job Scheduling Strategies for Parallel Processing*, D. Klusáček, W. Cirne, and G. P. Rodrigo, Eds., Springer International Publishing, Cham, 43–55, accessed: 2025-11-08.

Kaelbling, L. P., M. L. Littman, and A. R. Cassandra, 1998: Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, **101 (1–2)**, 99–134, https://doi.org/10.1016/S0004-3702(98)00023-X, accessed: 2025-11-11.

Kolker-Hicks, E., D. Zhang, and D. Dai, 2023: A reinforcement learning based backfilling strategy for hpc batch jobs. *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Networking, Storage, and Analysis (SC-W '23)*, Association for Computing Machinery, 1316–1323, https://doi.org/10.1145/3624062.3624201, accessed: 2025-11-08.

Li, B., and D. Zhao, 2007: Performance impact of advance reservations from the grid on backfill algorithms. *Sixth International Conference on Grid and Cooperative Computing (GCC 2007)*, IEEE, 456–461, accessed: 2025-11-21.

Marin, D., and O. Sigaud, 2012: Towards fast and adaptive optimal control policies for robots: A direct policy search approach. Accessed: 2025-11-08.

Puterman, M. L., 1994: *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, New York, NY, accessed: 2025-11-11.

Reiss, C., J. Wilkes, and J. L. Hellerstein, 2011: Google cluster data. *Google Research*, https://github.com/google/cluster-data, Accessed: 2025-11-21.

Rubinstein, R. Y., 1997: Optimization of computer simulation models with rare events. *European Journal of Operational Research*, **99 (1)**, 89–112, https://doi.org/ https://doi.org/10.1016/S0377-2217(96)00385-2, URL https://www.sciencedirect.com/science/article/pii/S0377221796003852, accessed: 2025-11-13.

Sutton, R. S., and A. G. Barto, 2018: *Reinforcement Learning: An Introduction*. 2nd ed., MIT Press, Cambridge, MA, accessed: 2025-11-11.

Talby, D., D. G. Feitelson, J. P. Jones, and Coauthors, 1999: The standard workload format (swf) for parallel job scheduling. -, accessed: 2025-11-08, https://www.cs.huji.ac.il/labs/parallel/workload/swf.html.

Talby, D., D. G. Feitelson, J. P. Jones, and Coauthors, 2014: Experience with using the parallel workloads archive. -, accessed: 2025-11-18, https://www.cs.huji.ac.il/w~feit/papers/PWA14JPDC.pdf.

Wang, X., Y. Qiao, J. Xiong, Z. Zhao, N. Zhang, M. Feng, and C. Jiang, 2024: Advanced network intrusion detection with tabtransformer. *Journal of Theory and Practice of Engineering Science*, **4 (03)**, 191–198, https://doi.org/10.53469/jtpes.2024.04(03).18, URL https://centuryscipub.com/index.php/jtpes/article/view/527, accessed: 2025-11-11.

Yonghao, C., Y. Huaqiang, H. Fengyao, and H. Jianshu, 2024: Regression-classification parallel prediction: An online learning optimization for job scheduling backfill algorithm. *Proceedings of the 2024 8th International Conference on High Performance Compilation, Computing and Communications*, Association for Computing Machinery, New York, NY, USA, 35–40, HP3C '24, https://doi.org/10.1145/3675018.3675775, accessed: 2025-11-08.

1 Appendix A - Model Architecture

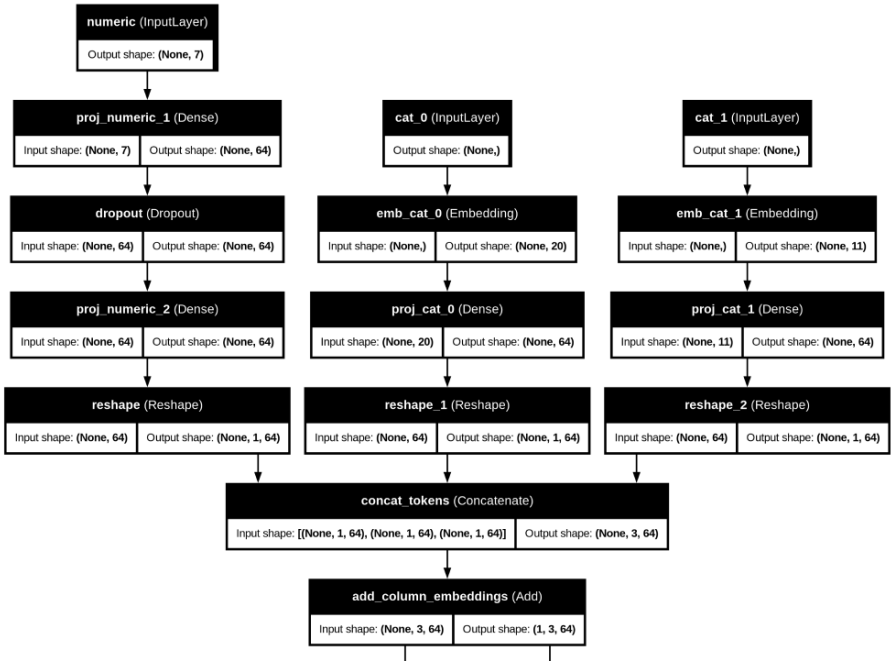

Figure A.1: Model Input Processing

Component	**Value**
Base hidden dimension	64
Number of Transformer layers (depth)	4
Number of attention heads	4
Feedforward dimension (FFN)	128
Dropout rate (all Dropout layers)	0.1
Layer Normalization Rate	1e-6
Number of tokens	3
Dense activation function	ReLU
Loss	Huber
Optimizer	Adam
Initial Learning Rate	1e-3

Table A.1: UTEM Model Architecture Summarize

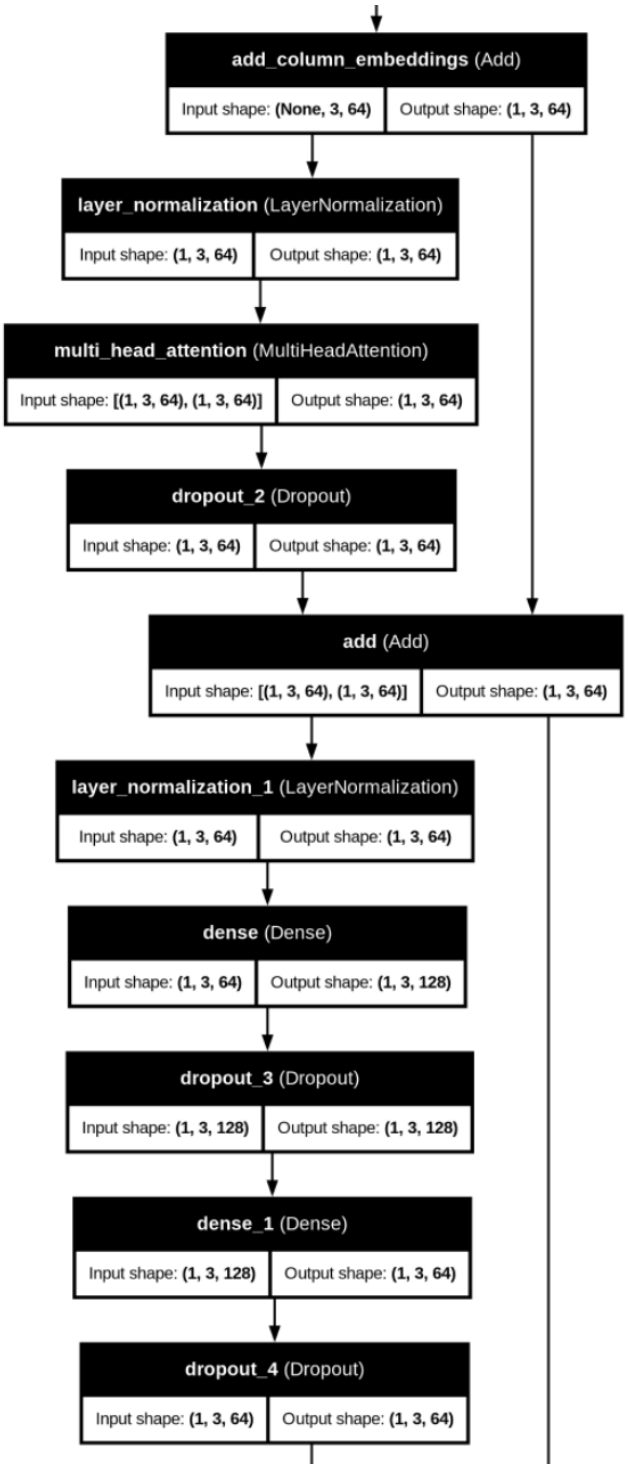

Figure A.2: Model Transformer Block

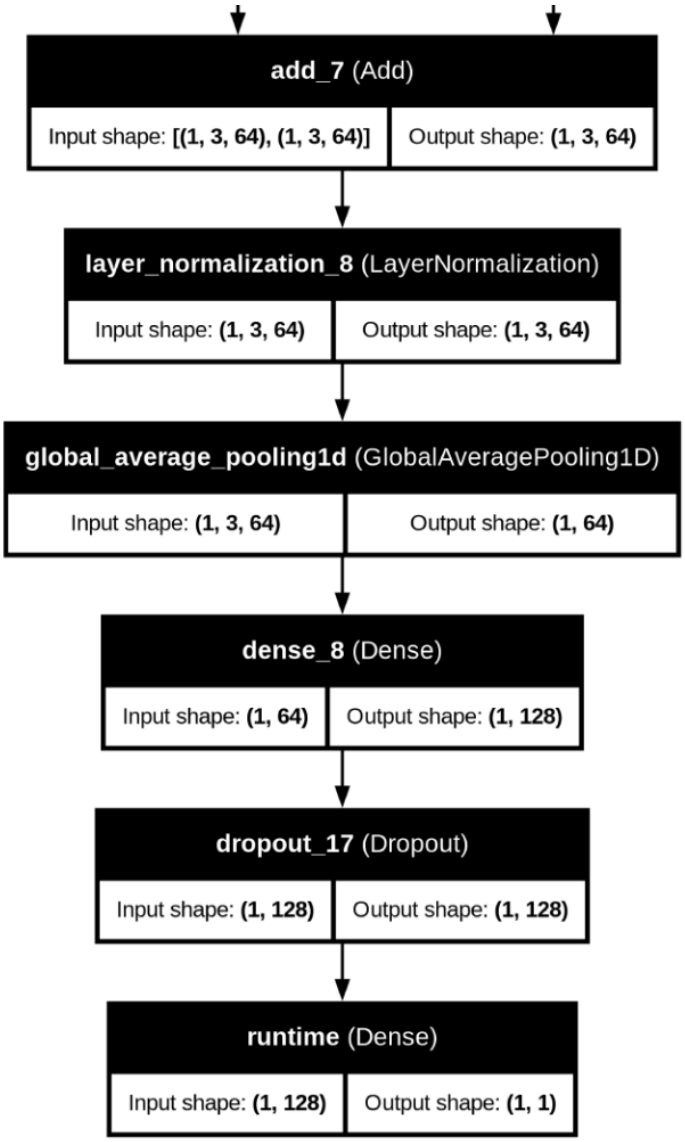

Figure A.3: Model Output

2 Appendix B - HPC2N Seth Platform

```
<?xml version='1.0'?>
<!DOCTYPE platform SYSTEM "https://simgrid.org/simgrid.dtd">
<platform version="4.1">
  <zone id="AS_SETH" routing="Full">

    <!-- master node -->
    <cluster id="master"
             prefix="master_host"
             suffix=""
             radical="0-0"
             speed="1Gf"
             bw="4GBps"
             lat="2us">
      <prop id="role" value="master"/>
    </cluster>

    <!-- Seth compute nodes.
         120 nodes; each has 2 single-core CPUs (Athlon MP2000+, 1.667 GHz).
         Peak ~ 2 flops/cycle => 1.667e9 * 2 = 3.334e9 flop/s per core.
     -->
    <cluster id="seth"
             prefix="linux"
             suffix=""
             radical="0-119"
             speed="3.334e9f"
             core="2"
             bw="4GBps"
             lat="2us" />

    <!-- Backbone link between clusters -->
    <link id="backbone" bandwidth="1.25GBps" latency="500us" />

    <!-- zone Route -->
    <zoneRoute src="seth" dst="master" gw_src="linuxseth_router" gw_dst="master_hostmaster_router">
      <link_ctn id="backbone"/>
    </zoneRoute>

  </zone>
</platform>
```