Mirage: Towards Low-interruption Services on Batch GPU Clusters with Reinforcement Learning

Qiyang Ding*
The University of Texas at Austin

Pengfei Zheng*
University of Wisconsin, Madison

Shreyas Kudari
The University of Texas at Austin

Shivaram Venkataraman
University of Wisconsin, Madison

Zhao Zhang
Rutgers University

ABSTRACT

Accommodating long-running deep learning (DL) training and inference jobs is challenging on GPU clusters that use traditional batch schedulers, such as Slurm. Given fixed wall clock time limits, DL researchers usually need to run a sequence of batch jobs and experience long interruptions on overloaded machines. Such interruptions significantly lower the research productivity and QoS for services that are deployed in production. To mitigate the issues from interruption, we propose the design of a proactive provisioner and investigate a set of statistical learning and reinforcement learning (RL) techniques, including random forest, xgboost, Deep Q-Network, and policy gradient. Using production job traces from three GPU clusters, we train each model using a subset of the trace and then evaluate their generality using the remaining validation subset. We introduce Mirage, a Slurm-compatible resource provisioner that integrates the candidate ML methods. Our experiments show that the Mirage can reduce interruption by 17-100% and safeguard 23%-76% of jobs with zero interruption across varying load levels on the three clusters.

ACM Reference Format:
Qiyang Ding*, Pengfei Zheng*, Shreyas Kudari, Shivaram Venkataraman, and Zhao Zhang. 2023. Mirage: Towards Low-interruption Services on Batch GPU Clusters with Reinforcement Learning. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '23), November 12–17, 2023, Denver, CO, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3581784.3607042

1 INTRODUCTION

DL practitioners and researchers are increasingly leverage large-scale GPU clusters to train large ML models. For example, some scientists are training the 345 million parameter BERT [12] or the 175 billion parameter GPT-3 [8] models for various DL tasks and these large networks can take $O(10^3)$ GPUs for days and even months to train [40, 52]. Others deploy ML models to perform inference for classifying celestial objects and detect Type Ia supernovae on streaming data in real-time [18]. These jobs are long-running as they continuously process incoming data. As a result, computing centers are experiencing large increases in execution time and wait time, a trend that is expected to continue.

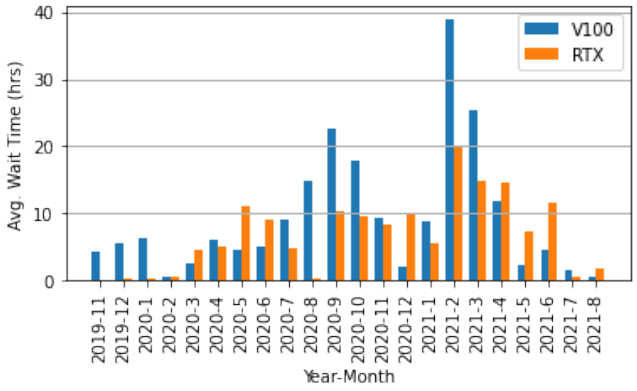


Figure 1: Average Queue Wait Time on the V100 (TACC Longhorn) and RTX (TACC Frontera) GPU Cluster.

In an effort to ensure responsiveness and fairness, computing centers enforce a fixed wall clock time limit for jobs running on GPU clusters. For example, TACC Longhorn has a 48 hour limit, while the NERSC Perlmutter supercomputer has a 12 hour limit. Since the runtime of DL training is much longer than these limits (e.g.,pre-training the 20 billion GPT-NeoX models takes 96 Nvidia A100 GPUs for 30 days [5]), scientists resort to running consecutive jobs. That is, just before the time limit expires, they checkpoint model training and then resume training by submitting a second job. Specifically, this is usually done in an automatic way by submitting an array of jobs to the scheduler, e.g., SLURM, which does not start accumulating the priority of a dependent job until the previous one completes [39]. We find that users can experience long wait time by submitting consecutive jobs reactively (e.g., up to 40 hour wait time on a V100 cluster in February 2021, as shown in Figure 1). Thus, the existing *reactive* approach inevitably introduces interruptions, limits the quality of service (QoS) and hurts responsiveness.

A straightforward way to facilitate long-running services is to dedicate a group of compute nodes and relax the wall clock time, such as the real-time queue on NERSC Perlmutter [29]. This approach requires policy and scheduler changes system-wise and it is prune to low machine utilization, responsiveness, and unfairness [17]; thus the use of such queues is limited to special requests. A second way is to predict the queue wait time, as in classic works [30, 31, 45], and then to submit subsequent jobs proactively as advised by the prediction. However, an empirical study showed

*Equal contribution to this work

SC '23, November 12–17, 2023, Denver, CO, USA

ACM ISBN 979-8-4007-0109-2/23/11.
https://doi.org/10.1145/3581784.3607042

that prior approaches only have an average accuracy of 20-72%, and at least 91% of predictions are incorrect [46]. The inaccurate queue wait time prediction can be attributed to the randomness in job arrival and completion time (a detailed study is in §3). In this work, we take a different approach to reducing the interruption by designing a *proactive* job provisioner that learns the queue state changes using historical job traces via reinforcement learning techniques. Researchers have examined RL methods for job scheduling in High-performance Computing [24, 33, 51] to maximize global performance metrics such as machine-wise utilization or fairness among all users. In contrast, the scope of this work is to minimize the overall time-to-solution for an individual user.

Specifically, we formulate the proactive resource provisioning problem using the *reinforcement learning* (RL) paradigm. The proactive resource provisioner functions as the agent and our environment includes the cluster and jobs. The machine states and queue states form the environment at a specific time and the provisioner can extract these states from the scheduler (e.g., Slurm). The action that the resource provisioner (agent) takes is to either submit the job or do nothing. Our reward can be determined by the resulting interruption or overlap for a series of actions taken during job execution.

In particular, we investigate Deep Q-Network (DQN) and policy gradient methods using two network architectures, transformers [48] and Mixture-of-Experts, as the state-action value function and the policy function, respectively. We design Mirage, a flexible RL framework with pluggable network architecture and RL methods. Mirage interacts with a low overhead Slurm simulator which we have validated to be sufficiently close to production Slurm deployments.

To examine the effectiveness and generality of the candidate methods, we evaluate Mirage using traces from *three* production GPU clusters. This includes 20-month long traces from TACC Longhorn (a 88-node V100 cluster) and Frontera (a 84-node RTX cluster) and a five-month long trace from Lonestar6 (a 76-node A100 cluster). We refer to these three clusters as V100, RTX, and A100. Following standard ML practice, we partition the job traces of each cluster into training and validation sets with a ratio of 80:20. For each cluster, the RL method is trained on the training partition and then evaluated on the validation range to examine generality. Our results show that, for HPC workloads, under a medium to high level of cluster load, Mirage safeguards 23%-72%, 35%-72%, and 40%-60% jobs to have zero interruption on the V100, RTX, A100 cluster, respectively. We also observe that Mirage significantly outperforms the reactive and average baseline strategies in reducing the average interruption by 25-53%, 21-44%, 77-100%, when machines are heavily loaded across the three job traces.

By comparing across all eight methods (see §6), the MoE+DQN (Mixture-of-Experts with Deep Q-Learning) and transformer+PG (policy gradient agent with transformer) methods outperform others. When the machine is heavily loaded, transformer+PG has a 16.9-55.9% lower interruption compared to MoE. Its aggressiveness also pays a 1.8-2.6x higher overlap when the machines are lightly loaded. Given the balanced performance, Mirage uses MoE+DQN as its default model. We also provide an option for users to use transformer+PG if they work on a heavily loaded machine.

We would like to note that one should not misunderstand the effectiveness scope of the examined RL methods. These methods are trained with a specific cluster job trace, and they are only effective on that cluster. Our generality lies in the methods used but not in the models, in the sense that one need to train the models with the particular job trace of the machine that one wants to run the proactive provisioner on.

The applicability of Mirage is beyond DL training and inference service. Scientists can also use this tool for long-running simulations or data analysis workflows for shorter experiment turnaround. Mirage, the trained models, Slurm simulator, and model training code are all available at https://github.com/zhaozhang/Mirage. The main contributions of this paper include:

- The design and evaluation of DQN and policy gradient methods with transformer and MoE network architecture that significantly reduces interruptions caused by heavy machine load.
- The flexible Mirage framework with pluggable RL methods and network architectures.
- Slurm simulator that can support job trace sampling and replaying with low overhead.
- Evaluating the generality of the proposed method on three distinct GPU clusters.
- An open source implementation of the simulator and RL-based provisioner.

The rest of the paper is organized as following: We discuss the trend of DL jobs and basics of RL in §2. We analyze the job traces and present the results in §3. The formalization of resource provisioning as RL is discussed in §4 and the implementation details are in §5. We present and discuss the experiment design and results in §6. A review of related RL research and its application in HPC is presented in §7. Finally, we conclude in §8.

2 PRELIMINARIES

Deep learning applications have been increasingly popular on GPU clusters. In this section, we discuss the long-running training and inference service demanded by scientists in supercomputers and present the preliminaries for **deep Q-learning [25], policy gradient [43] and mixture of experts [50]**.

2.1 Long Running Deep Learning Applications

As scientists and DL practitioners investigate more complex problems, they often utilize larger models and datasets. For example, the 345 million parameter BERT [12] model takes 5 days to train with 8 NVIDIA A100 GPUs on the English Wikipedia dataset [37]. The 175 billion parameter GPT-3 model is estimated to take 34 days to train with 1,024 NVIDIA A100 GPUs [28]. Scientists are also deploying trained models for real-time data processing or object detection. Some examples include transient celestial object detection [18], point-scanning electromagnetic imaging super-resolution [13], and disease detection in digital agriculture [26]. These applications require running long inference jobs as a service, which is problematic since modern GPU clusters frequently impose 48- or 72-hour limits to ensure responsiveness.

2.2 Deep Q-Learning

Reinforcement learning (RL) is a machine learning method based on trial and error. At a time step t, an RL agent takes an action A_t to react to the current system state S_t, and thereby, shifts the system to the next state S_{t+1}. The agent receives an immediate reward R_t for triggering the transition, and this process of making decisions and collecting rewards repeats to until a terminal step T. The RL agent is trained to learn an optimal that maximizes the accrued rewards within the time horizon ($t = 1, \ldots, T$). When the learned policy is non-deterministic and the state transition is stochastic, the trajectory of produced states, actions and rewards is uncertain and we denote it with sequence of random variables $S_1, A_1, R1$, ..., S_T, A_T, R_T.

At time step t, we use function $Q_\pi(S, A)$ to indicate the the expected return (cumulative reward) starting from t. π itself is function that maps an encountered state to its decided action, and the discounting factor $0 \leq \gamma \leq 1$ translates future rewards to present values.

$$Q_\pi(s, a) = E_\pi[\sum_{k=1}^{T} \gamma^k R_{t+k+1} | S_t = s, A_t = a] \tag{1}$$

with Monte Carlo simulations, one can rollout different trajectories from a policy, and then, compute an average return for each visited state as a sample estimate of $Q(S, A)$. This would converge if every state and action is visited infinite times. Learning can also happen online by bootstrapping with the Q values of the next state in the sequence„ which is referred to as Q-learning [25]:

$$Q_{k+1}(s, a) \leftarrow Q_k(s, a) + \alpha(R + \gamma \max_{a'} Q_k(s', a') - Q_k(s, a)) \tag{2}$$

s' is the next state transitioned from s, and k represents the iteration of learning that updates Q. The Q-values are captured in a tabular manner but this could get cumbersome for large state-spaces. Recent studies [42, 43] use a deep neural network with weights θ, namely a deep Q network (DQN), to represent Q and its loss function and gradients for the k-th iteration are defined as below. Rather than computing the full expectation, the loss function can be optimized with mini-batch Stochastic Gradient Descent (SGD).

$$L(\theta_{k+1}) = E[(R + \gamma max_{a'} Q(s', a'; \theta_k) - Q(s, a; \theta_{k+1}))^2] \tag{3}$$

$$\begin{aligned} \nabla_{\theta_{k+1}} L(\theta_{k+1}) = E[R + \gamma max_{a'} Q_k(s', a'; \theta_k) \\ -Q(s, a; \theta_i) \nabla_{\theta_k} Q(s, a; \theta_{k+1})] \end{aligned} \tag{4}$$

2.3 Policy Gradient

Deep Q-Networks (DQN) is a value-based RL method that is known for a few limitations. First, DQN lacks a direct representation of the policy, and policy inference requires solving an optimization problem $\text{argmax}_a Q(s, a)$ and this challenges when the action space is high-dimensional. Second, DQN does not automatically trade off exploit and exploration, and usually relies on human-crafted strategies such as the ϵ-greedy strategy [25] to guarantee exploration. However, manually tuning the ϵ threshold is difficult. To overcome these limitations, policy gradient [42, 43] is widely used; it parameterizes a policy with a deep neural network and directly outputs the probability of the action to take. Moreover, policy gradient can autonomously arbitrate exploit and exploration.

We briefly formulate policy gradient as below. First, let τ denote a random trajectory that comprises of a sequence of state, action, reward triples, i.e., $s_1^\tau, a_1^\tau, r_1^\tau, \ldots, s_T^\tau, a_T^\tau, r_T^\tau$, to until the terminal time step T; let $\pi_\theta(\tau)$ denote the probability distribution of τ when it is realized with a stochastic policy π. π is implemented using a deep neural network with parameters θ; let $r(\tau)$ represent the cumulative reward $\sum_{t=1}^{T} r_t$ and let $J(\theta)$ represent the expectation of $r(\tau)$, i.e., $J(\theta)$=$\mathbb{E}[\sum_{t=1}^{T} r_t^\tau]$. Note that, unlike value based methods, policy gradient directly maps a state s_t to its decided action a_t with (softmax) probability $\pi_\theta(a_t|s_t)$. The policy gradient theorem states that

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta(\tau)}[\nabla_\theta log \pi_\theta(\tau) r(\tau)] \tag{5}$$

, which can be approximated with Monte Carlo rollout,

$$\nabla_\theta J(\theta) \approx \frac{1}{N}(\sum_{t=1}^{T} \nabla_\theta log \pi_\theta(a_t^\tau | s_t^\tau)(\sum_{t=1}^{T} r_t^\tau) \tag{6}$$

, and the policy network can be update with gradient ascent and learning rate α, i.e., $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$, which iteratively converge to a (local) maximum. Note that although policy gradient solves some of the issues of deep Q-learning, it also incur some other issues; its learned policy may converge to a local optimum [38], and in real-world applications, it is found to suffer from high-variances [35]. As there arguably is no clear winner, we adopt both the DQN and the policy gradient method.

2.4 Mixture of Experts (MoE)

Scaling up model capacity is one of the key success of deep learning, as increased model capacity usually renders higher predictive power. There are three learning strategies to enhance model capacity, i.e., scaling up a monolithic model [21], ensemble learning [10] and MoE (Mixture of Experts) [50]. We try all these three different strategies to build and train RL provisioners, and evaluate them in Section 4. Note that we increase model capacity but cautiously prevent over-fitting using cross-validation.

Scaling up a monolithic model. One strategy to increase model capacity is to train a huge, monolithic deep neural network that may include hundreds to thousands of layers and billions of parameters [21]. However, training a huge monolithic model is a herculean task. First, training efficacy deteriorates as gradient vanishes when back-propagated through a deep architecture. Residual bypasses and advanced activation functions mitigate this issue but with no complete solution. Second, a huge, complex model has a higher degree of curvature and contains more saddle points and local minima, and gradient descent is more prone to stuck. Third, training a huge neural network is computationally prohibitive.

Ensemble learning. The other strategy to level up model capacity is ensemble learning such as Random Forest (RF) [7] and Gradient Boosting Decision Trees (GBDT) [14], which average the outputs of multiple learners, each trained on a sub-datasest. However, aggregating a large (infinite) number of weak learners (each underfits the complex data and incurs high biases), increases model complexity but does not effectively increase model capacity. Thus, RF or GBDT usually have difficulties in modeling complex supervised learning tasks such as language modeling. Moreover, classic ensemble learning adopts a static mixture weights for its member learners.

Table 1: Stats of the Job Traces of V100 and RTX.

	V100	RTX	A100
node count	88	84	76
Start Time	11/04/2019	12/04/2019	11/01/2022
End Time	08/20/2021	08/19/2021	03/31/2023
Orig. Job Count	189,899	375,095	49,997
Filtered Job Count	65,017	175,090	24,779

However, learners usually perform differently for varied input regions, and the mixture weights should be adaptively different for different input samples.

Mixture of Experts (MoE). We believe the philosophy that instead of training a complex, monolithic model or an ensemble of many weak models, one should train and combine several models of intermediate complexity [15]. Following this philosophy, the MoE deep neural networks [50] has achieved remarkable success. An MoE neural network contains many experts that share the same network architecture, each of which is trained on a partition of the training samples. There is a mixture (or gating) layer that adaptively routes inputs to their best-fit experts, and the best-fit experts' out are aggregated (averaged) as the final output. This MoE scheme scales up model capacity with no significant increase of computation overhead. We adopt the softmax gating layer [41], which computes the weighted average of the Q-values across E different DQN experts.

$$Q(s,a) = \sum_{e=1}^{E} G_\theta(e) Q_e(s,a) \quad , G_\theta(\cdot) = softmax(x \cdot W_\theta) \tag{7}$$

3 TRACE ANALYSIS AND DATA CLEANING

We collected job traces from three production GPU clusters in a national computing center. V100 has 88 compute nodes, each with four Nvidia V100 GPUs. RTX has 84 compute nodes, each with four Nvidia RTX 5000 GPUs. A100 has 76 compute nodes, each with three Nvidia A100 GPUs. Specifically, we collect the fields of *JobID, JobName, UserID, SubmitTime, StartTime, EndTime, Timelimit, NumNodes*. The time span of the job traces are 21, 20, and 4 months, respectively. These job traces reflect significantly different types of workloads in job arrival time, execution time, node count, node hour consumption, and job queue wait time. We first analyze the differences quantitatively and then discuss how the job trace data is cleaned before training.

3.1 Difference in Job Traces

As shown in Table 1, the V100, RTX, and A100 traces have 65,017, 175,090, and 17,570 jobs respectively. Figure 2 depicts the job count distribution over time. The average job count is $2,955 \pm 1,289$, $8,378 \pm 20177$, and $4,377 \pm 659$ per month for the V100, RTX, A100 cluster respectively. As we can see from the figure, there is no clear pattern of job arrival at a month granularity. It is also worth noting that there are 96,780 short jobs (less than 30 secs) on RTX. We do not remove such short jobs from our job traces, as they reflect the real machine usage at that time.

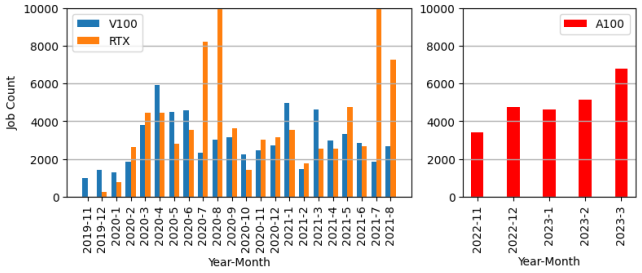
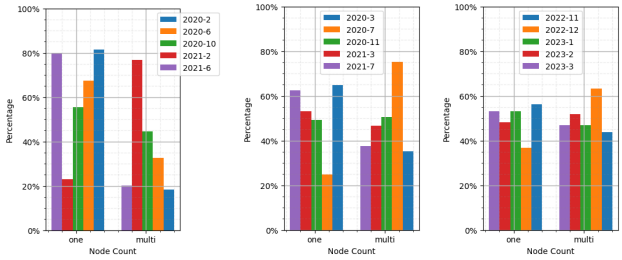


Figure 2: Job Arrival Distribution on the V100, RTX, and A100 Clusters.

For the node count distribution, the average nodes per job is 2.5, 1.3, and 1.6 on V100, RTX, and A100, respectively. Although multi-node jobs take only a small portion of total job count, their share in node hour consumption is more significant as shown in Figures 3(a), 3(b), and 3(c). For example, in 2021-2 on V100, the percentage of multi-node jobs is 23.4%, but they take 76.9% of the total node hours. Similarly in 2017-11, 12.0% of jobs are multi-node but take 82.5% of total node hours. This observation aligns with the trend of the time-consuming multi-node DL training.



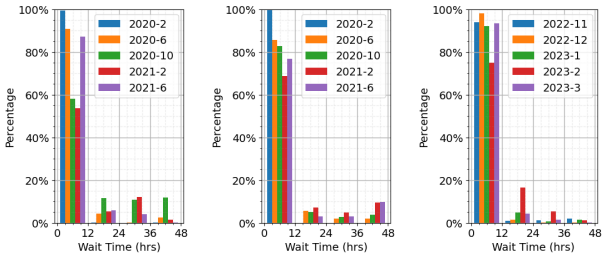
(a) V100 (b) RTX (c) A100

Figure 3: Distribution of Node Hour Consumption with Node Count on the V100, RTX, and A100 Cluster.

Finally, the statistic most directly relevant to this research, the queue wait time, is depicted in Figure 4. The average queue wait time on the V100 and RTX clusters is shown in Figure 1. The distribution from individual months are presented in Figure 4(a), Figure 4(b), and Figure 4(c). In 2020-10 and 2021-2, $\sim 30-41\%$ of the jobs on V100 have to wait for longer than 24 hours. On RTX, the percentage of jobs waiting for longer than 24 hours is $\sim 12-24\%$. On A100, 92-98% of jobs experienced a wait time less than 12 hours across the five-month period except 2023-2, where 26% of the jobs are waiting for more than 12 hours, and 3% are waiting for longer than 36 hours. Long wait time significantly limits the service provided by batch GPU clusters for long running DL training and inference jobs.

3.2 Data Cleaning

We manually filter out jobs that 1) request more nodes than the available and 2) sub-jobs that are submitted within one Slurm job. In the early-production phase of these clusters, all nodes are in



(a) V100 (b) RTX (c) A100

Figure 4: Distribution of Queue Wait Time on the V100, RTX, and A100 Cluster.

the same partition, then they are split into multiple partitions to address the needs of development and test and production run. Since our job traces are collected in the production partition, there are early jobs that request more nodes than that are available in the partition. So we remove those jobs from the job traces. There are also jobs, though recorded in the Slurm database, that are sub-jobs within one Slurm job. These jobs have an identical prefix followed by the sub-job id. So we combine them as one single job with the starting time of the first sub-job as the start time, and the end time of the last sub-job as the overall end time. Table 1 summarizes the statistics of the job traces.

The machine downtime, e.g., maintenance, leaves blank time ranges in job traces. We take the blank time ranges as no users submit any jobs during that time. Since the scheduled maintenance is one day every one or two months, it does not impact model training significantly. Slurm allows users to submit jobs with dependencies, and it releases the next job to the queue upon the finish of the previous job. This dependency is not reflected in the job traces, so we take jobs with dependencies as independent jobs submitted at different time. This approach does not change the queue wait time of the subsequent jobs.

4 REINFORCEMENT LEARNING FOR PROACTIVE PROVISIONING

Following the advances in deep supervised and unsupervised learning, reinforcement learning (RL) has embraced deep architectures and has achieved success in solving complicated, human-level control problems such as playing Atari and Go. These recent innovations motivate us to design a smart resource provisioner using RL techniques, and particularly in our framework, deep Q network [34] (DQN) and policy gradient (PG) [43], which we show can help HPC practitioners perform adaptive, proactive resource provisioning.

We discuss the advantages and disadvantages of DQN and PG in Section 2.3, and implement an RL framework with both DQN and PG policies (cf., Figure 7). Similar to the policy framework in [1], instead of building independent value and policy networks, we build the framework with a dual-head architecture, i.e., the V-head or the Q-value head and the P-head or policy gradient head. The two heads share the same foundation model (a transformer or an MoE-transformer model) while have different embedding and output layers. For the Q-head, the embedding layer encodes state-action pairs and the output layer maps their Q-values. For the P-head, the embedding layer encodes only states and the output layer maps to a the probability of different actions, followed by a sampling layer that accordingly samples actions.

To distill high-level features from the cluster state variables and the state transitions, we propose using transformer, which is validated to be effective for a wide-spectrum of ML application including both language modeling and generation (GPT-3 transformer [32]), and computer vision (ViT, i.e., vision transformer [16]). Particularly, we leverage transformer's multi-head attention mechanism to model long-range dependence, predict how cluster states (e.g., queue length and cluster business) transition over time, and thereby, manage interruption (or overlap) proactively.

Note that in addition to RL-based implementations, we also build Mirage with classical statistical models including random forest and gradient boosting decision tree, though results in the experiment section declares a significant gain for the RL models over the statistical model.

4.1 Encoding Queue, Server and Job States.

We represent the system state at each instant t with an m-dimensional vector v_t. By default, m is 40 and the vector comprises the following variables to encode queue, server and job information.

(a) Queue State: First, queue state includes the number of queued jobs currently waiting for scheduling ($var1$). Second, it includes summary statistics (i.e., 0th, 25th, 50th, 75th, and 100th percentiles) of the sizes (number of requested nodes) of the queued jobs ($var2 - var6$). Third, queue state includes summary statistics of the ages (time since submission) of the queued jobs ($var7 - var11$). Fourth, it includes summary statistics of the runtime limit (maximum duration that each job is allowed to run in the cluster) of the queued jobs ($var12 - var16$).

(b) Server State: First, server state includes the number of actively running jobs across all servers in the cluster ($var17$). Second, it includes summary statistics of the sizes ($var18 - var24$), elapsed runtime ($var25 - var29$) and runtime limit ($var30 - var34$) of all running jobs.

The model only considers submission time prediction for a single successor job, after the submission of its predecessor job. For example, suppose a user job J is partitioned as four 48 hour long sub-jobs $J1$, $J2$, $J3$ and $J4$. The model only considers optimizing the submission time of the successor sub-job $J2$ after the predecessor sub-job $J1$ is submitted. When $J2$ is submitted per model's decision, $J2$ becomes the predecessor and $J3$ becomes the successor, and so on, until $J4$ is submitted. Therefore, the model maintains a current Predecessor-Successor pair for each group of chained sub-jobs, and at each instant, the state representation also includes the status quo.

(c) Predecessor job state and ***(d) Successor job information:*** The predecessor states includes the size ($var35$), time limit ($var36$), queue time ($var37$), and elapsed runtime ($var38$). The successor information includes its size ($var39$), and time limit ($var40$). The successor job has not entered the cluster yet and thus only static information is included. It should be noted that we consider jobs' internal state, such as say an ML job's epoch progress to be private to the user. That is, the agent does not know or require users

to annotate or expose any job-specific internal information for training.

4.2 Encoding Workload and Cluster History

The state matrix. The 40-dimensional vector v_t is an abstraction of the queue, cluster and job states at an instant t, while k consecutive vectors $v_t, v_{t-1}, \ldots, v_{t-k}$ encode both workload (job arrival and completion) and cluster (the queue and server) history of length k. Existing RL work, such as playing Altari games [25], adopts a similar approach in constructing model input, within each the last four video frames (last four snapshots of video game) are taken as model input. In parallel, our model input is a $k \times m$-dimensional state matrix S_t at each instant t. By default, we set k as 144, and periodically record v_t at a default interval of 10 minutes which corresponds to the workload and cluster history of the last 24 hours back-traced from t.

4.3 The State Space and the Action Space

Unifying the state and action space for DQN and PG network. The action space for our RL provisioner consists of only two different actions: submit (submission) and no-submit (no-submission). The DQN network takes as input both the state matrix and the action to query, while the PG network takes as input only the state matrix. To render a unified model input, we add an ordinal variable that represents the action to query. For the ordinal action variable, 1 represents submission, -1 represents no-submission, for which the DQN network realizes it as an essential input, while the action is only a placeholder and is always 0 when inputted to the PG network. When training the foundation transformer model, we flatten the state matrix to a long vector and concatenate with the additional ordinal action variable. The flattened states (by default) contains 5761 (144×40+1) variables (cf., Figure 5).

4.4 Policy Serving

Deterministic Policy. For a learned DQN policy, the neural network is a value function $Q(S_t, a_t)$ that predicts, at the current instant t, the expected future gains of submitting versus not submitting the successor sub-job, given the current state and the history encoded in the state matrix S_t. For each targeted predecessor-successor pair, after the predecessor is submitted, the DQN model is periodically invoked at an interval of 10 minutes, and will submit the successor sub-job only when the Q-value of submission is larger than that of no-submission. Non-deterministic policy. For a learned PG policy, the network outputs directly the probability of submission versus no-submission, and the action to take is randomly sampled from this output binomial distribution (cf., 5).

4.5 Shaping the Reward

Different HPC practitioners can have their own views of the penalty of overlap and (or) interruption between the predecessor and successor sub-jobs. Performance-sensitive users may consider interruption to have a much larger penalty than overlap, while resource-waste-aversion users may consider overlap to have a larger penalty than interruption. We set two user-configurable penalty coefficients, e_I and e_O , for users to configure their penalty for interruption and overlap, respectively. Given this, we first set the time horizon of actions. Suppose the current instant is t', at which the predecessor sub-job is submitted, and after that, the agent's trained DQN or PG policy produces a sequence of actions $a_{t'+1}, \ldots, a_{t'+T}$. We can see that $a_t = 0$ for all $t' + 1 \leq t < t' + T$, and $a_{t'+T} = 1$ as the terminal action for the successor sub-job is submission, while any other actions prior to a_{t+T} are all non-submission actions.

After the successor job is submitted at instant $t' + T$, the agent closely monitors the state of the successor job and when it is dequeued for running at some future instant $t' + T'$, the outcome of how much the overlap or interrupt is revealed. We use r_I to denote an outcome interruption and r_O to denote an outcome overlap.

With the settings above, we shape the reward R_t for actions a_t as below. Note that we use negative penalty to equally represent reward, which means a reward of zero is the best possible reward, and the larger the interruption or overlap penalty is, the smaller the reward.

$$R_t = \left\{ \begin{array}{l} -e_i \cdot r_i, \text{if INTERRUPT} \\ -e_o \cdot r_o, \text{if OVERLAP} \end{array} \right\}, t' + 1 \leq t \leq t' + T \qquad (8)$$

If the observed interruption is small, the previous actions are rewarded to making the right decisions. That is, the previous no-submission decisions are credited for not causing overlap, and the final submission decision is credited for being proactive and timely. A similar argument applies when the observed overlap is small.

4.6 Network Architecture

The foundation model - transformer. We build the foundation model with a transformer (Figure 5)) model. Transformers are the state-of-the-art method for sequence modeling and prediction. For example, in NLP tasks like language modeling, a transformer model takes as input a sequence of previous words to predict the sequence of words that follow. This resembles our proactive scheduling task where we parse a history of system (e.g., queue and node) snapshots and forecast the change in system states. The difference lies in that, for language models, the transformer decoder deciphers the intermediate representation into predicted words, while for proactive scheduling, the decoder translates the intermediate representation of future system states into queuing delay, which can be used to compute the resulting interruption (or overlap). Furthermore, system state has both long-range dependencies (e.g., periodic, weekly jobs) and short-range dependencies (e.g., bursty arrival of jobs) over the time horizon. Thus, not all system snapshots in a history window provide dependent, relevant features to forecast the status quo and future. The multi-head self-attention module filters out irrelevant snapshots in history and identifies ones that contribute to prediction. Self-attention is the key technique in Transformer or BERT to improve upon traditional sequence modeling and achieve state-of-the-art results for classification tasks, including language modeling, machine translation and sentiment classification. Thus we propose using self-attention as the key enabler to accurately predict future system states and reinforcement learning rewards.

The V-head and the P-head output layers. The deep foundation model learns intermediate representation of the policy internal values and the it is either followed by a single linear map to reduce to a scalar output (i.e., the Q-value) of the V-head, or followed

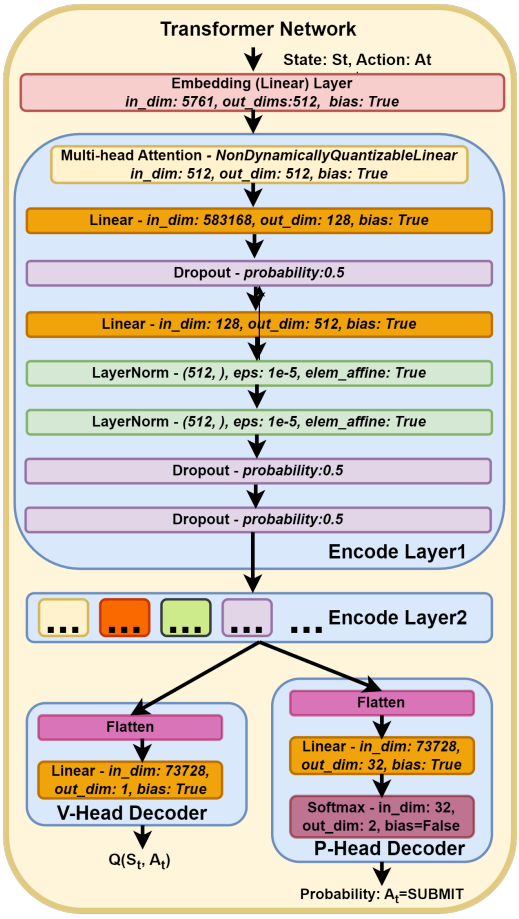


Figure 5: Dual-head Network Architecture - Transformer

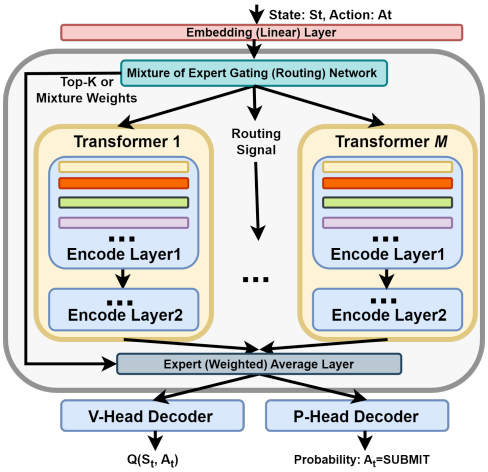


Figure 6: Architecture with the MoE Foundation Model

by a linear decision layer plus a softmax layer, which output the distribution of action sampling for the P-head (cf., Figure 5).

Hyperparmeter tuning. The hyperparameters that define the model structure, such as the number of multi-attention heads per transformer encoder layer, are tuned with RayTune [23], an auto-ML engine for hyper-parameter tuning. Figure 5 shows the tuned hyperparameters for all network layers. This set of tuned hyperparameters is repeatedly used as the default throughout this study.

4.7 The MoE foundation model

As discussed in Section 2.4, we further augment the capacity of the transformer foundation model with the MoE scheme. Figure 6 shows the MoE architecture and the workflow to combine experts. We implement and evaluate both the Top-1 sparse MoE [41] and the weighted average dense MoE [41] and found that the Top-1 scheme exhibit inferior provisioning performance in comparison to the weighted average MoE, though it enjoys less computational overhead due to sparse activation. Throughout this study, we omit the results of the Top-1 scheme for brevity.

The intuition behind MoE is similar to that of piecewise polynomial fitting such that individual polynomials focus to fit a small region (partition) of the input space (samples), and all the polynomials integrate to fit a complex function over the entire input space (complete set of samples). As cluster workloads and machine busyness change from weeks to weeks and months to months, we temporally split the cluster log records (training samples) to M (10 by default) fractions to train different expert transformers. Different experts focus on optimizing provision policy for different load levels and different workload mixes.

4.8 Experience Replay

Online RL usually fails for complicated control tasks if there exists correlation between consecutive training samples, which explodes the variance of gradient updates and distorts the a policy's value estimates. We also employ experience replay [4] to break data correlations. Experience instances, each of which includes the submission or no-submission action taken by the trained policy, as well as its resulting reward (interruption or overlap), are stored in a memory pool. The memory pool is random as the experience instances are collected from different time points for each simulation trial, and are randomly shuffled across different simulation trials. Experience instances are then grouped into random mini-batches to perform training of the value (or policy) networks.

4.9 Training DQN and PG

The training procedure consists of two phases. For the first phase, we train the foundation model offline while for the second phase, we train the V-head and the P-head decision layers online by running Slurm simulation experiments.

4.9.1 Offline training. **a) Sample collection.** We first collects a series of training samples; each episode instantiates an (simulated) experimentation episode that first submits a predecessor sub-job to the cluster at the current instant, and then, submit the successor sub-job at 7 averagely split sample points in a time range between start time (2 days warm-up after the simulation start time) and the end time of predecessor job. The episode lasts until the successor's delayed reward (i.e., interruption or overlap) can be observed. A single sample, i.e., a (state, action, reward) triple, is collected at the end of the episode and stored into the experience memory pools. When a job is partitioned into k sub-jobs, it can maximally generate

$k-1$ training samples (predecessor-successor pairs) for training. **b) Foundation model training.** In the offline phase, we pre-train the transformer or MoE-transformer foundation model with supervised learning; using training samples randomly sampled from the memory pool. Each sample includes a state and an observed reward. The input to the foundation model is the flattened state (cf., Section 4.3 and Figure 5), while the output is the observed reward. We use the Adam optimizer [6].

4.9.2 Online training. **a) Online DQN training.**: Online DQN training executes on-policy RL training [25] using the gradient formula in Equation 4. Training samples are collected with actions decided by the learned Q-value function. The DQN policy may never submit the successor and this leads to an infinite episode. To prevent such an extreme case, we add a small probability $\epsilon > 0$, with which, the DQN leaner randomly chooses actions regardless of the Q-values (i.e., ϵ-greedy strategy [25]). **a) Online PG training.**: Online PG training executes on-policy RL trainig using the gradient formulate in Equation 6. Note that though the V-head and P-head share the same foundation model, these two heads are trained independently.

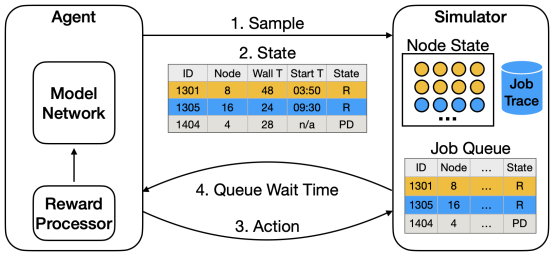


Figure 7: Architecture Overview.

5 SYSTEM DESIGN AND IMPLEMENTATION

In this section, we discuss the overall Mirage design and its implementation details.

5.1 System Architecture

The overall system architecture of Mirage consists of two major components: the agent and the simulator, as shown in Figure 7. It is designed to be generic for both model-free and model-based learning and handles value function approximation and model training.

At a high level, the agent trains the proactive provisioner and the simulator simulates the Slurm scheduler using the provided job traces. The simulator implements the priority-based scheduling algorithm with back-filling [49]. and mimics the behavior of the production Slurm deployments used in our GPU clusters. The simulator loads the jobs trace that we collected over a 20-month period and exposes an interface of *sample(), step(), and submit()*.

For each episode, the agent and the simulator will be initialized and run up to the time when the first job should be submitted. Then the agent submits the first job by calling *submit()*, calls *sample()* to acquire the cluster state. After that, the agent calls *step()* to instruct the simulator to move forward for a period of time, then collects the cluster state again. With sufficient states, the agent evaluates the rewards of submitting the second job, and makes a decision. Once a decision has been made, the agent calls *submit()* to submit the second job. Then the simulator runs till the second job starts and returns its final queue wait time that can be used to compute the reward.

The agent has a reward processor that receives the queue wait time from the simulator and converts it to the reward as required by the RL algorithm.

5.2 The Slurm Simulator

In order to train the agent, and validate the results, we develop a low-overhead Slurm simulator that supports Slurm's core scheduling logic, i.e., backfilling and priority scheduling. There are two main roles fulfilled by our simulator. One is to simulate the workload to create a virtual cluster environment, and the other is to provide support for offline learning and online learning of agents. Since the time taken for each iteration of learning depends on the simulation speed, it is crucial that we have a low-overhead, yet high-fidelity simulator design. Our current build simulates a one month workload within one minute.

Our simulator includes three components, a cluster abstraction, a scheduler abstraction, and a simulator module that interacts with the agent. The cluster abstraction is used to consume the workload, while the scheduler determines which jobs will be submitted to the cluster based on its policy. The simulator module exposes a rich API for running workloads and allows controlling a number of factors such as running the simulator a specific length of time or running until all the jobs have been finished etc. The ability for users to customize such factors makes our simulator design general enough to support both offline and online learning.

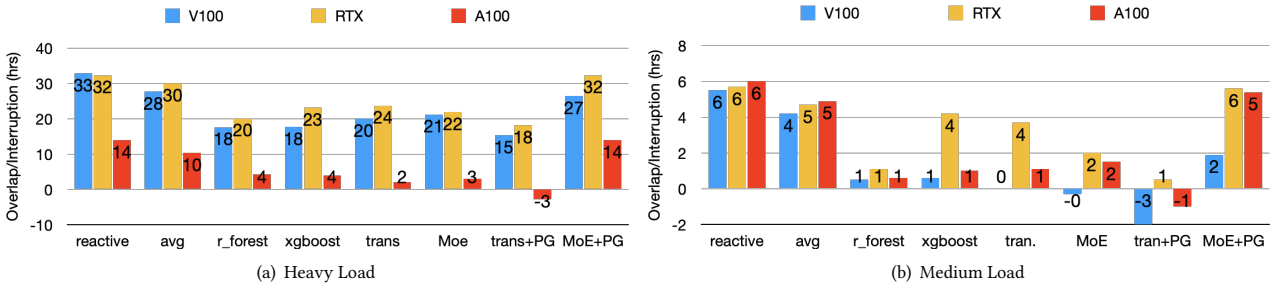
We evaluate simulation fidelity using an identical workload trace, and compare our simulator with the standard Slurm simulator [3, 44], which implements the exact Slurm scheduling logic that is used in real-word HPC clusters. We use 5 randomly sampled weeks as input and find that the difference in makespan across the five runs is less than 2.5%. We also compute the differences in job completion time (JCT) and find that the geometric mean of the difference is no more than 15% across all runs. Further, our simulator has a 3-26× lower overhead than the standard Slurm simulator.

6 EXPERIMENTS AND RESULTS

To validate the effectiveness and generality of Mirage, we conduct experiments using the job traces of the V100, RTX, and A100 GPU clusters. We partition each trace in 80:20 ratio for training and validation. Specifically, the training split is 11/2019-02/2021 and the validation split is 03/2021-07/2021 on both V100 and RTX. For the A100 GPU cluster, the training split is 11/2022-02/2023 and the validation split is 03/2023.

Our evaluation considers the following scenarios:

- For each trace, training Mirage with the training data for single node jobs, then using on the validation subset to examine the generality of the proposed method.



(a) Heavy Load

(b) Medium Load

Figure 8: Average Interruption of a Pair of 48-hour Single-node Jobs on the V100, RTX, and A100 GPU Clusters.

- For each trace, training and validating Mirage with 8-node jobs to study the effectiveness of our proposed method on multi-node jobs.
- Comparing the RL-based techniques with the statistical methods and the heuristic baselines of reactive provisioning.

We implement Mirage using PyTorch 1.11 [36] and use Ray [27] to enable distributed data preprocessing. The training is executed on the Lonestar6 supercomputer at Texas Advanced Computing Center.

We use two heuristic baselines: first is **reactive**, which submits the subsequent job upon the completion of the current one. ***It is worth noting that, the reactive baseline is what researchers usually use as a common practice [39].*** The second baseline is **avg**, which is derived by monitoring the average queue wait time T_{avg} and submitting the second job T_{avg} time units before the first job finishes. The second group of methods are ensemble methods, i.e. **XGBoost** [9] and **Random Forest** [7]. XGBoost combines multiple decision trees and leverages gradient boosting to make predictions. Random Forest also combines multiple decision trees to output a single output via averaging or voting. The third group of methods are RL-based methods. We treat transformer and MoE as foundation models, and examine both the DQN and policy gradient learning methods. So there are four combinations in total: **{transformer, MoE}** $\times$ **{DQN, PG}**.

For all experiments, we present the evaluation results under different cluster loads. We characterize cluster busyness with the reactive queue wait time measured in the baseline and define three categories: longer than 12 hours (high load), between two and 12 hours (medium load), and within two hours (light load).

6.1 Single-node Evaluation

In this experiment, we train each model by uniformly sampling the training time range and ran a pair of consecutive single-node jobs for 48 hours. Then we use the same sampling method in the validation range to examine generality. Single-node jobs are commonly for DL inference service. Figure 8 shows the average interruption/overlap of all techniques on the three GPU clusters.

As shown in Figure 8(a), when the cluster is heavily loaded, i.e., the reactive queue time exceeding 12 hours, random forest, XGboost, transformer, MoE, MoE+DQN and transformer+PG all show improved interruption with an average reduction of 44.1%, 33.7%, and 84.7% on the V100, RTX, and A100 cluster compared to the reactive baseline, respectively. MoE+PG is not as effective as other ensemble learning and RL methods. By comparing the training loss of MoE+PG with transformer+PG, MoE+PG overfits with equally low training loss with transformer+PG but a much lower validation performance.

We see a similar pattern when the clusters are with medium load in Figure 8(b). The ensemble learning techniques, DQN techniques, and transformer+PG show sigfinicant interruption reduction. Across the three clusters, Transformer+PG has the lowest interruption followed by MoE.

6.2 Multi-node Evaluation

In this experiment, we examine the effectiveness of candidate methods using a pair of eight-node jobs, which is a representative scale across the three clusters. Figure 9 illustrates comparison with heavy and medium loads.

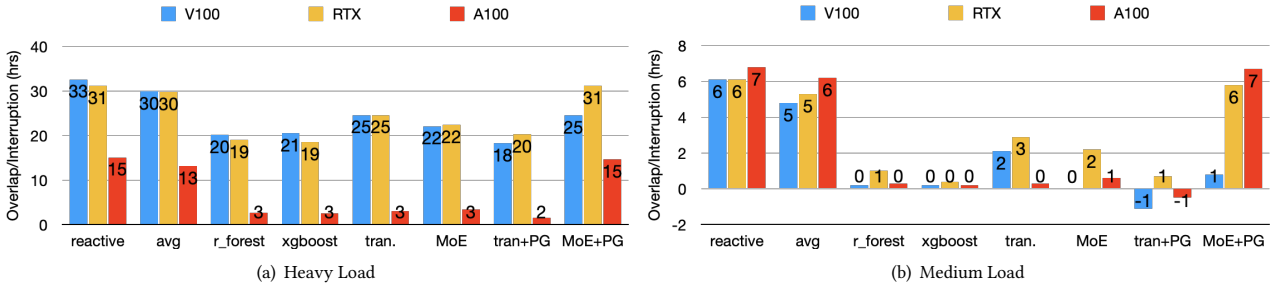
With heavy load as shown in Figure 9(a), the XGBoost and Random Forest methods show promising reduction of interruption by 37.5%, 40.0%, and 82.5% across the three clusters. MoE+DQN reduces the interruption by 32.2%, 28.2%, and 77.5%, which is slightly behind the ensemble learning methods. Transformer+PG has an improvement by 43.9%, 34.9%, and 90.1%, which shows the best results if we compare with an average.

With medium load as shown in Figure 9(b), We observe that ensemble learning methods almost completely eliminate the interruption. Transformer+PG shows a similar result. All these interruption elimination is at a cost of overlap when the machines are lightly loaded. We will discuss this in next section.

Now, comparing the effectiveness of MoE+DQN and transformer+PG methods across clusters, we see that they are more effective on the A100 cluster than on the V100 and RTX clusters. This is because the A100 job trace has less noisy jobs (those request hours of wall clock time but run for only 30 seconds), as we discussed in §3.1. The experiments in §6.1 and §6.2 with the A100 cluster clearly show the effectiveness of the RL-based methods, especially with a clean job trace.

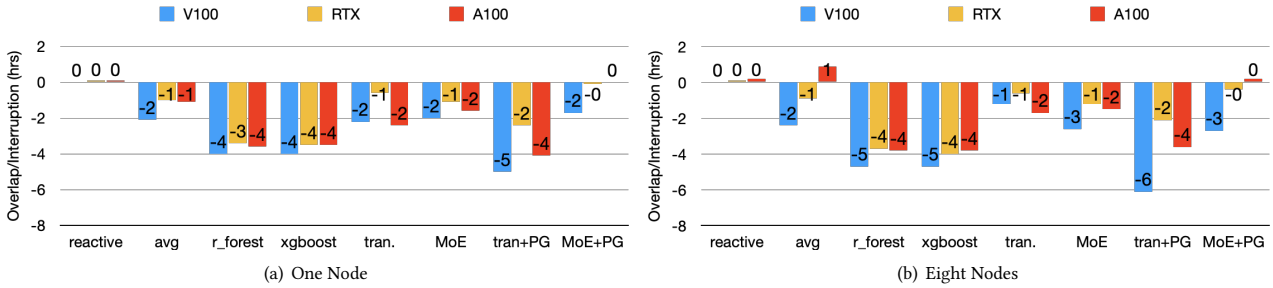
6.3 Overlap with Low Machine Load

Although the ensemble learning and RL-based methods can efficiently reduce the interruption when the machine is heavily or medium loaded. They inevitably introduce overlap between jobs when the machine is lightly loaded, where jobs can easily get compute nodes without a significant long wait time. Figure 10 shows the overlap of all methods with 1-node and 8-node jobs across the three



(a) Heavy Load (b) Medium Load

Figure 9: Average Interruption of a Pair of 48-hour Eight-node Jobs on the V100, RTX, and A100 GPU Clusters.



(a) One Node (b) Eight Nodes

Figure 10: Average Overlap of a Pair of 48-hour Eight-node Jobs on the V100, RTX, and A100 GPU Clusters with Light Load.

clusters. Given the 48-hour wall clock time, a few hour overlap is not a big problem for the proactive provisioner, as the subsequent job needs to load data, checkpoints, and libraries. Once it is ready, the current job can be released, and the subsequent job will resume from the latest checkpoint, so there is no redundant computation or wasted node hours. However, we want the proactive provisioner to be intelligent to avoid overlap if possible. With this standard, we observe that the ensemble methods and the transformer+PG method introduce a ~2X long overlap as the MoE+DQN method.

To balance the interruption reduction and overlap across machine load level, Mirage uses the MoE+DQN as its default model. We leave transformer+PG as a option for users as an aggressive provisioner, which will be more effective when the machine load is high.

7 RELATED WORKS

Facilitating long-running services on batch GPU clusters is relatively new in the area, due to the recently rising adoption of machine learning and deep learning in scientific research. National computing centers are deploying several large scale GPU dense supercomputers, such as Perlmutter at NERSC, Polaris at ALCF, and Frontier at OLCF. Similarly, industry leaders such as Microsoft, Meta, and Tesla deploy their own GPU supercomputers for internal DL-driven research [2, 20, 47]. All these machines have $O(10^4 - 10^5)$ high end GPUs. Most of these machines use batch schedulers such as Slurm and Cobalt to manage compute resources. Several NSF (National Science Foundation) funded AI institutes are devoted to enabling AI/ML/DL in application domains such as agriculture and ecology. Supporting inference as long-running services is as important as model training, and it is challenging for existing schedulers to support such services with low interruption.

On the other hand, reinforcement learning has made breakthroughs that match or exceed human capability in many areas such as game playing [43], Tokamak reactor control [11], and protein folding [19]. In one way or another, researchers formulate these problems as stochastic control problems then use the classic agent-environment paradigm and evaluate a set of RL techniques of SARSA, deep Q networks, online/offline policy, actor-critic and many more. In the field of high performance computing and distributed computing, researchers have explored the feasibility of RL in scheduling problems [24, 33, 51] to maximize certain performance metrics and file system configuration [22] to adapt to online I/O workloads. In this paper, we view the resource provisioning problem as a control problem and train a transformer-based neural network that predicts the expected interruption/overlap, then we use the deep Q network and policy gradient methods to make job submission decisions.

8 CONCLUSION AND FUTURE WORK

We examine a suite of ensemble learning and reinforcement learning methods to build Mirage, a proactive resource provisioner towards facilitating long-running DL training and inference services on batch GPU clusters with low interruption/overlap. Mirage is trained and validated with months-long job traces on three productive GPU clusters. Our experiment shows that Mirage can enable 23%-76% more jobs with zero interruptions, especially when the queue wait time is long. Mirage effectively reduces the interruption by 17-100% across the three clusters compared to the reactive baseline. In future, we will examine the generality of the proposed methods on GPU cluster with a much larger size.

9 ACKNOWLEDGEMENT

Work of Pengfei Zheng was supported by Computing Research Association. Qiyang Ding and Zhao Zhang are supported by the NSF ICICLE AI Institute (OAC-2112606).

REFERENCES

[1] [n. d.]. Clipped Proximal Policy Optimization. https://intellabs.github.io/coach/components/agents/policy_optimization/cppo.html.

[2] [n. d.]. Introducing the AI Research SuperCluster — Meta's cutting-edge AI supercomputer for AI research.

[3] 2022. Slurm Simulator. https://github.com/ubccr-slurm-simulator/slurm_simulator.

[4] Sander Adam, Lucian Busoniu, and Robert Babuska. 2011. Experience replay for real-time reinforcement learning control. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 42, 2 (2011), 201–212.

[5] Sid Black, Stella Biderman, Eric Hallahan, Quentin Anthony, Leo Gao, Laurence Golding, Horace He, Connor Leahy, Kyle McDonell, Jason Phang, Michael Pieler, USVSN Sai Prashanth, Shivanshu Purohit, Laria Reynolds, Jonathan Tow, Ben Wang, and Samuel Weinbach. 2022. GPT-NeoX-20B: An Open-Source Autoregressive Language Model. (2022).

[6] Sebastian Bock and Martin Weiß. 2019. A proof of local convergence for the Adam optimizer. In *2019 international joint conference on neural networks (IJCNN)*. IEEE, 1–8.

[7] Leo Breiman. 2001. Random forests. *Machine learning* 45 (2001), 5–32.

[8] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165* (2020).

[9] Tianqi Chen and Carlos Guestrin. 2016. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. 785–794.

[10] Noel Codella, Quoc-Bao Nguyen, Sharath Pankanti, David Gutman, Brian Helba, Allan Halpern, and John R Smith. 2016. Deep Learning Ensembles for Melanoma Recognition in Dermoscopy Images. *arXiv preprint arXiv:1610.04662* (2016).

[11] Jonas Degrave, Federico Felici, Jonas Buchli, Michael Neunert, Brendan Tracey, Francesco Carpanese, Timo Ewalds, Roland Hafner, Abbas Abdolmaleki, Diego de Las Casas, et al. 2022. Magnetic control of tokamak plasmas through deep reinforcement learning. *Nature* 602, 7897 (2022), 414–419.

[12] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[13] Linjing Fang, Fred Monroe, Sammy Weiser Novak, Lyndsey Kirk, Cara R Schiavon, B Yu Seungyoon, Tong Zhang, Melissa Wu, Kyle Kastner, Alaa Abdel Latif, et al. 2021. Deep learning-based point-scanning super-resolution imaging. *Nature Methods* 18, 4 (2021), 406–416.

[14] Jerome H Friedman. 2001. Greedy function approximation: a gradient boosting machine. *Annals of statistics* (2001), 1189–1232.

[15] Geoffrey Hinton. [n. d.]. Mixture of Experts. https://www.cs.toronto.edu/~hinton/csc321/notes/lec15.pdf.

[16] Kai Han, Yunhe Wang, Hanting Chen, Xinghao Chen, Jianyuan Guo, Zhenhua Liu, Yehui Tang, An Xiao, Chunjing Xu, Yixing Xu, et al. 2022. A survey on vision transformer. *IEEE transactions on pattern analysis and machine intelligence* 45, 1 (2022), 87–110.

[17] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*.

[18] Željko Ivezić, Steven M Kahn, J Anthony Tyson, Bob Abel, Emily Acosta, Robyn Allsman, David Alonso, Yusra AlSayyad, Scott F Anderson, John Andrew, et al. 2019. LSST: from science drivers to reference design and anticipated data products. *The Astrophysical Journal* 873, 2 (2019), 111.

[19] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, et al. 2021. Highly accurate protein structure prediction with AlphaFold. *Nature* 596, 7873 (2021), 583–589.

[20] Jennifer Langston. 2020. Microsoft announces new supercomputer, lays out vision for future AI work. https://blogs.microsoft.com/ai/openai-azure-supercomputer/

[21] Guohao Li, Matthias Müller, Bernard Ghanem, and Vladlen Koltun. 2021. Training graph neural networks with 1000 layers. In *International conference on machine learning*. PMLR, 6437–6449.

[22] Yan Li, Kenneth Chang, Oceane Bel, Ethan L Miller, and Darrell DE Long. 2017. CAPES: Unsupervised storage performance tuning using neural network-based deep reinforcement learning. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*. 1–14.

[23] Richard Liaw, Eric Liang, Robert Nishihara, Philipp Moritz, Joseph E Gonzalez, and Ion Stoica. 2018. Tune: A research platform for distributed model selection and training. *arXiv preprint arXiv:1807.05118* (2018).

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

[25] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. 2015. Human-level control through deep reinforcement learning. *nature* 518, 7540 (2015), 529–533.

[26] Sharada P Mohanty, David P Hughes, and Marcel Salathé. 2016. Using deep learning for image-based plant disease detection. *Frontiers in plant science* 7 (2016), 1419.

[27] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. 2018. Ray: A distributed framework for emerging {AI} applications. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 561–577.

[28] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Anand Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. 2021. Efficient large-scale language model training on gpu clusters. *arXiv preprint arXiv:2104.04473* (2021).

[29] NERSC. [n. d.]. NERSC Queues and Charges. https://docs.nersc.gov/jobs/policy/#-limits-and-charges.

[30] Daniel Nurmi, John Brevik, and Rich Wolski. 2007. QBETS: Queue bounds estimation from time series. In *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 76–101.

[31] Daniel Nurmi, Anirban Mandal, John Brevik, Chuck Koelbel, Rich Wolski, and Ken Kennedy. 2006. Evaluation of a workflow scheduler using integrated performance modelling and batch queue wait time prediction. In *SC'06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. IEEE, 29–29.

[32] OpenAI. 2020. GPT-3: Language Models are Few-Shot Learners. https://github.com/openai/gpt-3.

[33] Alexandru Iulian Orhean, Florin Pop, and Ioan Raicu. 2018. New scheduling approach using reinforcement learning for heterogeneous distributed systems. *J. Parallel and Distrib. Comput.* 117 (2018), 292–302.

[34] Ian Osband, Charles Blundell, Alexander Pritzel, and Benjamin Van Roy. 2016. Deep exploration via bootstrapped DQN. *Advances in neural information processing systems* 29 (2016).

[35] Matteo Papini, Damiano Binaghi, Giuseppe Canonaco, Matteo Pirotta, and Marcello Restelli. 2018. Stochastic variance-reduced policy gradient. In *International conference on machine learning*. PMLR, 4026–4035.

[36] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[37] J Gregory Pauloski, Qi Huang, Lei Huang, Shivaram Venkataraman, Kyle Chard, Ian Foster, and Zhao Zhang. 2021. KAISA: an adaptive second-order optimizer framework for deep neural networks. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[38] Thomas Pierrot, Valentin Macé, Felix Chalumeau, Arthur Flajolet, Geoffrey Cideron, Karim Beguir, Antoine Cully, Olivier Sigaud, and Nicolas Perrin-Gilbert. 2022. Diversity policy gradient for sample efficient quality-diversity optimization. In *Proceedings of the Genetic and Evolutionary Computation Conference*. 1075–1083.

[39] SchedMD. [n. d.]. Multifactor Priority Plugin. https://slurm.schedmd.com/priority_multifactor.html#age.

[40] Or Sharir, Barak Peleg, and Yoav Shoham. 2020. The cost of training nlp models: A concise overview. *arXiv preprint arXiv:2004.08900* (2020).

[41] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. 2017. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538* (2017).

[42] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. 2016. Mastering the game of Go with deep neural networks and tree search. *Nature* 529, 7587 (2016), 484–489.

[43] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. 2018. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science* 362, 6419 (2018), 1140–1144.

[44] Nikolay A. Simakov, Martins D. Innus, Matthew D. Jones, Robert L. DeLeon, Joseph P. White, Steven M. Gallo, Abani K. Patra, and Thomas R. Furlani. 2018. A Slurm Simulator: Implementation and Parametric Analysis. In *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*,

Stephen Jarvis, Steven Wright, and Simon Hammond (Eds.). Springer International Publishing, 197–217.

[45] Warren Smith, Valerie Taylor, and Ian Foster. 1999. Using run-time predictions to estimate queue wait times and improve scheduler performance. In *Workshop on Job scheduling strategies for Parallel Processing*. Springer, 202–219.

[46] Ozan Sonmez, Nezih Yigitbasi, Alexandru Iosup, and Dick Epema. 2009. Trace-based evaluation of job runtime and queue wait time predictions in grids. In *Proceedings of the 18th ACM international symposium on High performance distributed computing*. 111–120.

[47] Tesla. 2021. Ahead of 'Dojo,' Tesla Reveals Its Massive Precursor Supercomputer. https://www.hpcwire.com/2021/06/22/ahead-of-dojo-//tesla-reveals-its-massive-precursor-supercomputer/.

[48] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NeurIPS)*. Curran Associates Inc., Red Hook, NY, USA, 6000–6010.

[49] Andy B Yoo, Morris A Jette, and Mark Grondona. 2003. Slurm: Simple linux utility for resource management. In *Workshop on job scheduling strategies for parallel processing*. Springer, 44–60.

[50] Seniha Esen Yuksel, Joseph N Wilson, and Paul D Gader. 2012. Twenty years of mixture of experts. *IEEE transactions on neural networks and learning systems* 23, 8 (2012), 1177–1193.

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

[52] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. 2022. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068* (2022).

Appendix: Artifact Description/Artifact Evaluation

ARTIFACT DOI

https://doi.org/10.5281/zenodo.8084694

ARTIFACT IDENTIFICATION

The artifact of this paper includes 1) the code repo, 2) three GPU cluster job traces, and 3) the trained models.

The proactive provisioner has the components of a SLURM simulator and an RL agent. The RL agent learns to submit a pair of 48-hours jobs to minimize the interruption between them. Depending on the state-action value function and policy function selection, the training procedure produces neural networks that functions as the value function and policy function, respectively.

With the cod and data, we expect any experienced HPC practitioner can reproduce our experiment results and to extend our solutions.

REPRODUCIBILITY OF EXPERIMENTS

The experiment workflow includes the case of RTX as described in the paper. Reviewers can replace the job traces by switching the input job trace.

The total estimation of the training may take an A100 GPUs 8 days.

The expected results are in text form, which is different than the result section in the paper. We would like to do an update once the paper is accepted.

Please contact zzhang@tacc.utexas.edu to obtain a copy of the job traces. Then follow https://github.com/zhaozhang/Mirage/blob/master/README.md to reproduce the results.

ARTIFACT DEPENDENCIES REQUIREMENTS

i) The comprehensive workflow comprises three integral components: the generation of training data, the training of the model, and the validation of the model. These components necessitate distinct hardware resources. The process of training data generation and model validation predominantly rely on CPU resources, with the potential requirement for expansive DRAM resources if there is a need to generate or validate a larger number of sample points concurrently. On the other hand, the training of the model necessitates GPU resources, though it currently supports utilization of a single GPU exclusively. For instance, we employ the Epyc 7763 for tasks necessitating CPU resources and the Nvidia A100 for tasks demanding GPU resources. The highest attainable DRAM size is 256GB of DDR4 Memory.

ii) The empirical investigations presented in this manuscript were conducted on Rocky Linux 8.6, underpinned by the Linux kernel version 4.18.0.

iii) The experimental environments were established and scrutinized using Python versions 3.9.7 and 3.10.8. Additional software libraries that are requisite for these environments encompass numpy, ray, torch, matplotlib, pandas, sklearn, and xgboost. Any further imported modules derive exclusively from Python's built-in module set.

iv) The core idea of Mirage is to train the resource provisioner using RL methods. The job traces of the three GPU clusters are essential to this work. Unfortunately, the job trace data is proprietary, so it is not going be public. However, we made it temporarily available for AD/AE evaluation. The job trace data is at https://utexas.box.com/s/pme8jt5l10k7te7cydexdu98991d4n6y.

ARTIFACT INSTALLATION DEPLOYMENT PROCESS

i) It is necessary for the user to install Python version 3.9.7 or later, as well as all dependent modules. The codebase, developed in Python, does not require any compilation.

ii) Instructions to run the experiments can be found in the README.md file. The process begins with offline data generation or baseline creation, continues with model training, and ends with validation. Here we will include two examples for getting reactive baseline and MoE model. {MIRAGE_ROOT} is the local Mirage path and we use cluster Lonestar6 as the following examples of reproducing the experiment results.

Before running examples, the following operations are necessary: 1. Create directory {MIRAGE_ROOT}/src/data and {MIRAGE_ROOT}/src/experiment 2. Copy {MIRAGE_ROOT}/src/model/moe to {MIRAGE_ROOT}/src/moe

MoE training example:

cd {MIRAGE_ROOT}/src/top/ && python3 offline_data_gen.py -parallel -num_samples 684 -num_probe 7 -interval 4 -od {MIRAGE_ROOT}/src/data/ls6/train_data_684 -workload filtered-ls6.log -start_time 2022-11-01T00:00:00 -warmup_len 2 -workload_len 5 -node 1 -baseline default python3 {MIRAGE_ROOT}/script/pickle_merge.py -wd {MIRAGE_ROOT}/src/data/ls6/train_data_684 -out batch_ls6_684_7.pickle

cd {MIRAGE_ROOT}/src/model/moe/ && python3 train.py -wd ../../data/ls6/ -n moe_ls6 -parallel -nd ls6_684_7 -mix_epoch 300 -sample_window 144

cp {MIRAGE_ROOT}/src/data/model/ {MIRAGE_ROOT}/experiment/ls6_train_moe/model/

cd {MIRAGE_ROOT}/src/top && python3 online_validate.py -num_validate 156 -interval 4 -workload filtered-ls6.log -workload_len 5 -start_time 2023-03-01T00:00:00 -od {MIRAGE_ROOT}/experiment/ls6_validate_moe/result -m {MIRAGE_ROOT}/experiment/ls6_train_moe/model/moe_moe_ls6.pt -warmup_len 2 -sample_window 144 -parallel -mt moe -node 1

Reactive baseline generation example:

cd {MIRAGE_ROOT}/src/top/ && python3 offline_data_gen.py -parallel -num_samples 156 -num_probe 7 -interval 4 -od {MIRAGE_ROOT}/experiment/ls6_offline_gen_baseline_reactive/baseline_reactive -workload filtered-ls6.log -start_time 2023-03-01T00:00:00 -warmup_len 2 -workload_len 5 -node 1 -baseline baseline_reactive

```
    python3 {MIRAGE_ROOT}/script/pickle_merge.py -wd {MI-
RAGE_ROOT}/experiment/ls6_offline_gen_baseline_reactive/baseline_reactive
-out baseline_reactive_ls6_merge.pickle
```