Towards Topology Aware Pre-Emptive Job Scheduling with Deep Reinforcement Learning

Bon Ryu
York University
Toronto, ON
bonryu@eecs.yorku.ca

Aijun An
York University
Toronto, ON
aan@eecs.yorku.ca

Zana Rashidi
York University
Toronto, ON
zrashidi@eecs.yorku.ca

Junfeng Liu
IBM Canada
Markham, ON
jfliu@ca.ibm.com

Yonggang Hu
IBM Canada
Markham, ON
yhu@ca.ibm.com

ABSTRACT

We present a topology aware Deep Reinforcement Learning (DRL) scheduler that simultaneously chooses jobs to run and elastically allocates resources to them for Distributed Deep Learning data parallel jobs in a multi-GPU, multi-machine cluster. This work addresses multiple limitations in the state-of-the-art methods: 1) Not sufficiently accounting for the bandwidth sharing between multiple jobs running simultaneously in a cluster, 2) Using overly simply heuristics to solve the resource allocation problem, 3) Pretending that job speed is not affected by the topology of allocated resources in simulation environments. This DRL method calculates unique job speeds by taking advantage of a graph representation of the cluster topology. This enables modeling realistic sharing of inter and intra machine bandwidths such as QPI speed, CPU-GPU speed, GPU-GPU speed, Infiniband card to Top-of-Rack Switch, etc. Our neural network model is trained using the REINFORCE algorithm which is a policy gradient method. The model outputs a multiple softmax designed to represent an assignment table that specifies the resource allocation of GPU's to Jobs. Using this design we can dynamically choose/change which GPUs to assign to which jobs at discrete time steps. Our simulation experiments show that our method can outperform baseline schedulers that use heuristics for job picking and resource allocation.

CCS CONCEPTS

• **Computing methodologies** → **Planning and scheduling**; **Reinforcement learning**; Parallel computing methodologies.

KEYWORDS

GPU job scheduling, reinforcement learning, neural networks

ACM Reference Format:
Bon Ryu, Aijun An, Zana Rashidi, Junfeng Liu, and Yonggang Hu. 2020. Towards Topology Aware Pre-Emptive Job Scheduling with Deep Reinforcement Learning. In *Proceedings of the 30th Annual International Conference on Computer Science and Software Engineering (CASCON '20)*. Toronto, ON, IBM Corp., USA, 10 pages.

CASCON '20, Nov 10–13, 2020, Toronto, Canada

1 INTRODUCTION

Job scheduling is an important part of running a computing cluster, which could be a massively parallel supercomputing centre, a cloud data centre, or an on-site group of servers at a university research lab, or even possibly a small cluster of edge devices. Jobs must be scheduled both in terms of when they should run, as well as what resources to assign them. Colloquially, job scheduling refers to both tasks. In this paper, scheduling a job in time will also be referred to as job picking, and resource to job assignment will be referred to as resource assignment or resource allocation. In this paper, we focus on the problem of both job picking and assigning resources to data parallel Deep Neural Network (DNN) jobs that are meant to run on GPU-capable cloud-based data centres. In short, we present a Deep Reinforcement Learning (DRL) method that leverages the modeling of intra and inter machine network topology and decides which jobs to run by elastically assigning GPU resources to them.

Most schedulers use heuristics (such as the shortest job first) for GPU resource scheduling. Heuristic-based methods can produce good solutions in some situations, but they often lead to a solution far from an optimal one in many other situations. Recently, deep reinforcement learning has been used for GPU resource scheduling [8], which uses a DNN as the policy function for reinforcement learning to learn a scheduler by interacting with the environment. However, these DRL-based methods fail to consider the topology of the resources within the cluster. Further, these schedulers usually solve the job picking problem and do not deal with resource allocation which is crucial in order to design an effective scheduler. Preemption is also another issue not dealt with in schedulers currently in use. We discuss some of these in detail below.

Currently, there exists no "intra" and "inter" machine topology aware DRL-based scheduler that can schedule and assign resources to multiple jobs on a multiple machine, multiple GPU cluster. This includes most of the intra and inter machine bandwiths such as QPI speed, CPU-GPU speed, GPU-GPU speed, Infiniband card to Top-of-Rack Switch speeds, etc. Both the topology of resources allocated to a job and the resulting bandwidth sharing between multiple jobs affect the unique speeds of jobs. In other words, job speeds and resource assignments (aka allocations) have a non-linear relationship. Schedulers that fail to sufficiently taking into account

topology in their decision making, will fail to make optimal job picking and resource to job assignment decisions, resulting in sub-optimal scheduling performance. This study addresses this issue through detailed modelling of intra and inter network topology, bandwidth sharing, and unique job speeds. Unlike heuristic methods, DRL schedulers can in theory avoid modeling the topology by training their policy function (modeled by a neural network function approximator) directly on bare metal of one's cluster, but one loses the ability to use simulations to initialize the training of the neural network (NN) function approximator. In fact, the model parallelization work by [9] used DRL to train a NN scheduler on bare metal on a single machine for single jobs at a time, but would need time consuming re-training on bare metal if the topology or cluster machines changes.

Another issue not thoroughly explored in literature are preemption strategies for DDL. Preemption means that resource allocation of a running job can be changed after the job has started but not yet completed. For example, in the no-preemption case, the resources (i.e. GPUs) allocated to a job does not change, and a job cannot be paused/resumed once started. In what we call *partial preemption*, the number of GPUs assigned to a job can be changed during runtime but the job is not allowed to be totally paused. In what we call *full preemption*, jobs can also be paused/resumed but the number of GPUs assigned upon resuming can be different than before the job was paused. Full preemption allows the most elasticity of job scheduling, and better/fuller use of resources, potentially leading to a shorter makespan for a set/sequence of jobs.

In this study, we address both intra and inter network topology considerations, job scheduling, and resource assignment/allocation. We make some initial investigations into using a DNN as a full preemption scheduler that is trained with RL, and compare this to no-preemption heuristic baselines. To hasten the development and investigation of DRL to solve DNN job scheduling/resource allocation problem, this work remains in a simulation environment. Still, our initial findings compel us to believe full preemptive schedulers could can make better/fuller use of resources than schedulers that can only perform no preemption at all.

Our contributions are summarized as follows. We propose a reinforcement learning based GPU resource scheduler, called **RL-TAPS** (Reinforcement Learning based Topology-Aware Preemptive Scheduler), that considers the topology of the underlying infrastructure. **RL-TAPS** solves both the job selection and resource allocation problems at the same time. Furthermore, our method allows for preemption. Our use case in this paper is data parallel distributing deep learning jobs across multiple GPUs in multiple machines, although **RL-TAPS** can be applied to other job types with different resources. The results from both streaming and non-streaming scenarios show the superiority of our method compared to three baseline methods.

The paper is organized as follows: In section 2 we discuss related work and section 3 gives a brief background in reinforcement learning and the policy gradient algorithm. In section 4 we describe our proposed method and we report the evaluation results in section 5. Finally we share related insights and discuss future work.

2 RELATED WORK

2.1 Mathematical Programming vs Heuristics

It is quite useful to note for context that many combinatorial optimization problems have traditionally been solved by formulating a mathematical program and using algorithmic methods such as simplex, or branch and bound, to solve them. If mathematical programming approaches are too slow, then a custom heuristic would be designed. The general problem of which fraction of which resource to assign to which job can be formulated as a Mixed Integer Non-Linear Programming (MINLP) problem. By constraining decision variables to be discrete (i.e we assign whole GPUs to jobs), we can formulate an Integer Non-Linear Programming (INLP) problem. We can further relax the problem to be an Integer Linear Programming (ILP) problem by making some big assumptions such as pretending to know the job completion times ahead of time or setting the number of GPUs assigned per job to be equal and fair.

Two notable recent works in literature for scheduling and assigning resources for data parallel, parameter server (PS) based, Distributed Deep Learning (DDL) jobs on clusters, are Tiresias [7] and Optimus [11]. With respect to the resource assignment problem (aka job/device placement), the authors of Tiresias first tried to formulate an ILP program that minimized network bandwidth usage. Even despite assuming equal resource assignment to linearize the problem, their ILP solution was too slow. The authors of Optimus merely described their resource assignment problem as an INLP problem that minimized Job Completion Times. They state that the problem is NP-hard, and decided instead to use a heuristic for assigning resources to jobs. Both Tiresius and Optmius ultimately used heuristics for both time based job scheduling and the resource assignment. For added context, even prior to learning of these works, we attempted to solve the general MINLP problem with mathematical programming, but failed to find a formulation that produced fast nor close to optimal results.

Popular cloud-based cluster computing schedulers such as Yarn and Slurm use very simple heuristics for time based job scheduling, such as DRF [5], First in First Out (FIFO), Shortest Job First (SJF), etc., or some combination of these. They also use very simple heuristics for resource allocation. These heuristics do not properly consider network topology nor do they adjust resource assignment to existing jobs to better utilize resources. We believe that heuristic methods for job scheduling and resource allocation, will be ultimately inferior to the potential benefits that DRL can provide, namely due to the ability of DRL to solve highly non-linear and complex problems.

2.2 DRL-based schedulers

There have been a resurgence in the study of using Neural Networks to solve optimization problems with the creation of pointer networks by [14]. Vinyals et al. [14] used existing algorithmic mathematical programming approaches to supervise the training of their pointer networks. More recently though, Bello et al. [1] built upon the work of [14] by using Deep Reinforcement Learning (DRL) to train pointer networks. Deep Reinforcement Learning (DRL) can train a Deep Neural Network (DNN) function approximator to solve complex non-linear decision problems without explicitly labeled data, by instead using scalar reward signals to guide the iterative

training of NNs. The first known cases of using DRL for job-shop scheduling problems, however, were by Zhang and Dietterich in 1995 and 1996 [16–18].

Some recent works have begun to use DRL for GPU resource management [4, 8, 9]. Mao et al. [8] used a DRL policy gradient method to choose jobs in a queue in a simple simulation environment with no network topology considerations. Their input layer for their simple Multi Linear Percepteron (MLP) policy network was a rectangular grid that represented the resources as contiguous columns, with each row representing a discrete time snapshot. Thanks to the simplicity of their design and availability of their code on github, this work has become very influential to subsequent investigations into using DRL for resource management for not just the field of cluster scheduling but also networking. [4] was inspired by [8] for example to pipe the input into a convolutional neural network (CNN) to choose a job, and then subsequently use a small NN to choose one of two heuristic placement algorithms. Both of these works ([4, 8]) are applicable only to data parallel jobs, as is our current work, which is also based on the work of Mao et al. [8].

Device placement/allocation for model parallelism of DL jobs is a much more complicated problem, which has been investigated on a single job and single machine basis by [9, 10]. Those authors brilliantly used sequence to sequence pointer networks to take a computation graph as input, and output a sequence of devices to assign to the nodes of the computation graph. Perhaps in the future, a DRL based method of automatically performing model and data parallelism simultaneously will be developed. For the time being, however, our current work attempts to address some of the many unsolved issues in DRL for data parallel DDL.

3 BACKGROUND

We give a short background on reinforcement learning and the policy gradient algorithm that will be used in the proposed method.

3.1 Reinforcement Learning

In reinforcement learning, an agent interacts with an environment and learns to take actions such that it maximizes some performance measure. The environment is represented via a state space and the agent sees a state s_t at each time step t. It then takes an action a_t based on the current state and is rewarded r_t by the environment while transitioning to the next state s_{t+1}. An episode is defined as a sequence of triples of state, action, reward, $((s_t, a_t, r_t))_{t=1..T_{max}}$, where T_{max} is the maximum length of the episode, and is finite.

Reinforcement learning is different from other forms of learning in that the agent has no explicit labelled data about the environment and other variables (i.e. states, reward, action). The agent collects data via interacting with the environment and learns optimal action sequences through experience.

3.2 Policy Gradients

In the theoretical development of the policy gradient algorithm for the episodic case, one first starts with the objective,

$$J(\theta) \doteq v_{\pi_\theta}(s_0), \tag{1}$$

where s_0 is the start state of an episode, θ are the parameters of a function π_θ, the policy determined by the parameters. A policy function π is a mapping from states S to actions A and can be defined as a probability distribution over actions a given a state s. Since the number of state and actions pairs can be very prohibitively large, function approximators such as neural networks are used to represent a policy. Thus a policy defined through a neural network with parameters θ is written as π_θ. v_{π_θ} is the performance measure called "true value function" and is the only function that can inform us of what actions to take (in transitioning between states) to get the highest possible performance. Theoretically, it is difficult if not impossible to determine v_π. In practice, it is common to approximate the objective as the expected return, $\mathbb{E}_{\pi_\theta}[G_t]$, where $G_t = [\sum_t^{T_{max}} \gamma^t r_t]$ [13, 15]. The discount factor γ is set between 0 and 1 and is used to put more weight on immediate rewards and discount later rewards. For the episodic case it is commonly set to 1 for simplicity. The REINFORCE algorithm [15] then uses the following update to the parameters:

$$\theta \leftarrow \theta + \alpha \mathbb{E}_{\pi_\theta} \left[G_t \nabla_\theta \log \pi_\theta(s_t, a_t) \right] , \tag{2}$$

where we are taking the expectation over many trials and time steps, sampled using π_θ. While we use the original formulation of G_t, it can be tailored to the problem being solved.

4 PROPOSED METHOD

In this section, we describe the problem we are solving, our proposed approach to solving the problem, and the simulation environment we train and test our proposed model.

4.1 Problem statement

Given a cluster of computers, each with one or more GPUs, and a set of deep learning jobs including the ones that are currently running on the cluster and the ones that were submitted and are waiting for GPU allocation, the goal is to determine:

(1) which jobs should be chosen to run.
(2) which GPUs should be assigned to jobs chosen to run.

so that the average job slowdown among all the jobs is minimized. A job's slowdown is defined as the difference between its finish and enter time, divided by its estimated time to run alone.

Note that we do not separate job selection and job allocation in this problem definition, and consider them simultaneously so that jobs are selected only if they can lead to an allocation that achieve a better outcome. Also, we allow preempting the already running jobs and elastically changing their resource allocations to improve overall performance.

4.2 Topology Awareness

Our scheduler is topology-aware in two ways: 1) intra and inter machine bandwidths in the cluster's topology are modelled in detail as part of the environment, and affect the observed reward signal; 2) the current GPU to job assignments (partially representing the cluster topology) are presented as input to an NN policy function. An NN input design that incorporates topology bandwidth information is left for future work. Even without bandwidth information directly incorporated into the input, the NN can in theory find GPU to job assignment combinations that make better use of the cluster topology and thus increase the reward signal.

4.3 State representation

The state S describes the status of the cluster in terms of GPU usage and the jobs in the job queue. Assume that the cluster has a set of X machines, $\{n_i | i = 1..X\}$, with a total number of M GPUs, $\mathbf{G} = \{g_i | i = 1 \dots M\}$. Also assume that the job queue can take up to N jobs and each element of the queue is called a job slot. Each job slot can hold one job, and the slots are sequentially filled up as jobs arrive. We represent the set of N job slots as $\mathbf{J} = \{j_i | i = 1 \dots N\}$.

Similar to [8], we use a set of matrices to represent the state, shown in the top part of Figure 1. Firstly, X two-dimensional matrices are dedicated for the **cluster representation**, one matrix per machine. Next, N two-dimensional matrices represent the N job slots in the job queue, one matrix per job slot. Finally, there is a matrix to represent a backlog.

The state representation in Figure 1 shows of a small cluster of $X = 4$ machines (labeled $n1$ to $n4$) and a job queue with $N = 3$ job slots (labelled $j1$ to $j3$). The first two job slots are occupied by a running job. The job in the third slot is waiting. In Figure 1, the number of columns of the matrix for machine n_i is the number of GPUs it has (e.g., n_1 has 4 GPUs). The number of columns of the matrix for a job slot j_i is the maximum number GPUs a job can request (e.g., 6 GPUs in the figure). In general, the rows of the matrices represent the time dimension, and the columns represent the resource dimension. Each row of the matrices is a representation of the entire environment along the columns (cluster usage, job resource request, etc.) during a specific discrete span of time. The multiple rows then allows for the presence of sequential information to be represented in the state.

The highlighted cells in the matrices representing machines indicate that the corresponding GPUs are occupied by the current running jobs. The highlighted columns in a matrix representing a job slot in the queue indicate the number of GPUs requested by the job, $gpusreq(j)$. The time span of each row of the image is calibrated to have a constant value in seconds (see Appendix A). Thus if we know the amount of computation a job has left, and its estimated speed, we can determine the number of times steps required for this job. In a job slot, the number of highlighted rows of a job is the number of time steps it is expected to run for if the job is 1) running alone, 2) is allocated exactly the number of GPUs it requests, and 3) there is zero communication time between GPUs. In the cluster representation, the number of highlighted rows of a job is the number of time steps left for it to complete, given its current unique topology-dependent speed, and the calibrated time span of the row. Job modelling is further explained in section 4.6.2.

When a job arrives at the cluster, but there are no empty job slots, the job gets appended to the small backlog matrix (illustrated at the right most end of the top part of Figure 1), for which a single job is represented by a single unit.

When a job is assigned GPUs by **RL-TAPS**, we paint the corresponding GPU columns in the cluster representation with the same color. In our full preemptive method, **RL-TAPS**, jobs that arrive to the job slots remain there as long as they are not finished. They can be not-started, paused, or currently running. Only when the jobs are finished are they removed. This way, currently running, paused, and not-started jobs in the queue are all candidates for being chosen by **RL-TAPS**. In contrast, in the RL simulation for

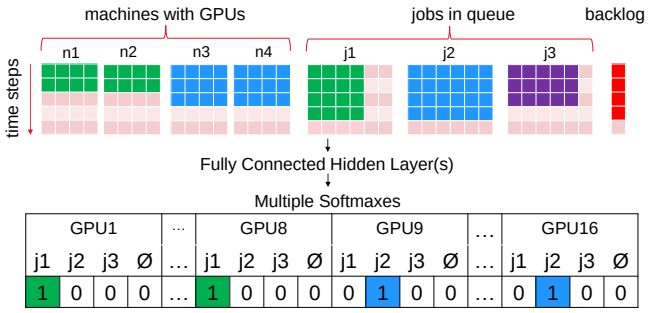

Figure 1: The architecture of the NN function approximator. Above there are three jobs in the queue. The first two are running, and the third is waiting. This assignment is not optimal but possible, depending on the level of training.

no-preemption schedulers, jobs that are chosen (and thus assigned a static number of GPUs), are removed from their jobslot, so they only occupy the cluster representation and are no longer candidates for being chosen.

To simplify the "painting" of the cluster representation, we do not stack jobs vertically in the cluster representation. In other words, each column of the cluster representation is assigned to at most one job. This state representation will be used as the input to the neural network policy function.

4.4 Actions and Rewards

4.4.1 Actions, A. The NN takes multiple simultaneous actions using a multiple softmax approach. If there are M GPUs in the cluster, there are M output softmaxes, and all softmaxes share the previous hidden layer. Each softmax, as shown in Figure 1, consists of $N + 1$ units, where N is the number of job slots. The corresponding GPU-to-job assignment table is shown on the right hand side of Figure 2. In this study, each GPU can be assigned to at most one job slot, or not assigned to any job slot, i.e., the null job slot $\varnothing$. The decision of which job to assign a single GPU to, is done using a single softmax. With multiple GPUs in a cluster, we have multiple softmaxes, one per GPU. Note that if the job slot corresponding to a currently-running job is not assigned to any GPUs, it is paused.

In this manner, we can solve the problem of choosing a job and assigning resources to it, simultaneously. Furthermore, this approach is naturally pre-emptive. A job that has not finished, remains in its slot, and can be paused or resumed simply by not assigning or assigning GPUs to it. If a job is not assigned any GPUs, then it is not chosen. Thus, job picking and resource assignment are performed at the same time.

4.4.2 Topology-Aware Reward, R. In RL, a scalar signal for reward is needed to guide the training of the policy function (a neural network in our case). It can also be helpful to subtract penalties from the reward to discourage bad decisions. We describe exactly how we formulate rewards and penalties. The main measure of reward we use is computational throughput in units of FLOPS (Floating Point Operations Per Second), which we interchangeably refer to as speed. In general, we calculate the aggregate sum of speeds of all jobs.

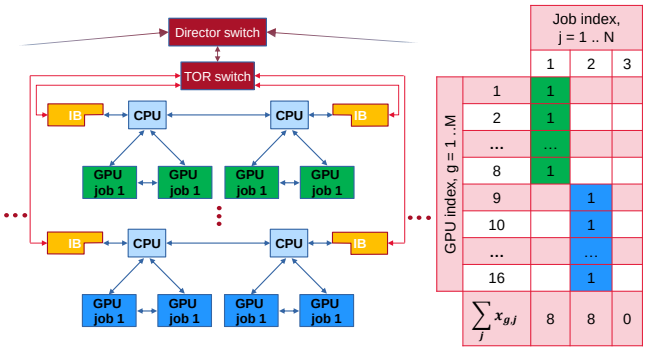

Figure 2: (Left) A rack is shown with its Top of Rack Switch (TRS), and two of its machines. The vertical ellipsis mean that there exists additional machines. Horizontal ellipsis on either side mean there may exist additional racks. The dark red arrows pointing to the DS indicate connections from two additional TRS's on either side. (Right) Example assignment of GPUs 1 to 8 for job 1 and GPUs 9 to 16 for job 2.

Rather than considering the instantaneous speed of a job, we prefer for simplicity to think about the average speed that a job experiences while interacting with other running jobs. Since time steps are not instantaneous but rather span two time points, it is suitable to consider the average speed of a job. The job's average speed at a discrete time step contributes to the total reward for the time step. This contribution is expressed as,

$$R_{t(j)} = v_{t(j)} = \frac{d_{m(j)}|\mathrm{G}_{(j)}|}{tt_{m(j)} + rt_{(j)}}, \tag{3}$$

where $d_{m(j)}$ is the computational "distance" of job j per minibatch. I.e. $d_{m(j)} = d_{ex(j)}m_{(j)}$, where $d_{ex(j)}$ is the computational "distance" of job j on a single example, and $m_{(j)}$ is the minibatch size. The computational distance is defined as the number of FLOPs (Floating Point Operations)[1]. $\mathrm{G}_{(j)}$ is the subset of GPUs assigned to job j. $|\mathrm{G}_{(j)}|$ is thus the number of GPUs assigned to job j. In the denominator, $tt_{m(j)}$ is the training time of a minibatch for job j. $rt_{(j)}$ is the time it takes for a reduction operation at the end of a minibatch such as gradient averaging across the GPUs of a job. Since reduction time is independent of the minibatch size, but is still characteristic of a job running in job slot j, $rt_{(j)}$ lacks a m subscript.

For each time step t of an episode i, we calculate a reward value for the time step as a sum of the throughputs (equation 3) across currently running jobs, in addition to any penalties associated with that time step.

$$R_t^i = \sum_j R_{t(j)} + Penalties \tag{4}$$

The superscript episode index, i, is omitted on the RHS of equations 3 and 4 for simplicity.

There were two main penalties designed to help aid the neural network to train.

[1]We use the term distance as proxy for computation (FLOPs not FLOPS) to intuitively use kinematic equations such as $\Delta d = vt$

(1) The cost of a job sitting idle in a jobslot or backlog, whose magnitude is calculated as

$$C_{t(j)} = \frac{d_{m(j)}gpusreq(j)}{tt_{m(j)} + srrt_{(j)}}, \tag{5}$$

where $srrt_{(j)}$ is called the single rack reduction time and is an estimate of the job's reduction time if the job running alone on a single rack using $gpusreq(j)$ GPUs.

(2) The cost of fewer than requested GPUs being assigned to the job in a jobslot, whose magnitude is calculated as

$$C_{t(j)} = \frac{d_{m(j)}[gpusreq(j) - |\mathrm{G}_{(j)}|]}{tt_{m(j)} + rt_{(j)}}, \tag{6}$$

The first penalty applies to both **RL-TAPS** and no-preemption methods. Without this penalty, **RL-TAPS** would fail to learn to decrease slowdown by running multiple jobs simultaneously. This is because running one job at a time with high resource usage could also increase throughput but comes at the expense of longer idle time for waiting jobs. The second penalty only applies to **RL-TAPS** and it is necessary for training the NN to learn to assign GPUs close to the number that is requested. Assigning more GPUs to a job than requested can be beneficial and is counted in equation 3.

4.5 Policy Function Design and Training

We use a neural network to represent the policy function.

4.5.1 Neural Network Structure. The overall NN architecture is shown in Figure 1. Its input, described in section 4.3, represents the state of the environment. The input is connected to the output using a single fully connected hidden layer. The output layer of the NN consists of multiple softmaxes (one for each GPU), all sharing the previous hidden layer. Each softmax represents a probability distribution over job slots given a GPU. The bottom part of Figure 1 illustrates the output layer of the NN. Each softmax corresponds to a decision to assign a GPU to a job slot, with the highest probability element of the softmax corresponding to the most favoured job slot. In addition, a GPU may not be assigned to any job slot. Thus, a null job slot $\varnothing$ is used to represent such a situation.

At a given time step t, let A_g be the event that a GPU g is assigned to one of $N+1$ job slots, which corresponds to the action with the highest probability outputted by GPU g's softmax, $p(A_g|s)$. We use A_t to represent the event that events A_g for all GPUs occur simultaneously, that is, $A_t = \bigcap_{g=1}^{M} A_g$. Assuming independence among A_g's, we can use the product rule of probability to express a single policy function as:

$$\pi_\theta(A_t, S_t) = p(A_t|S_t) = \prod_{g=1}^{M} p(A_g|S_t) \tag{7}$$

where S_t is the given input at t to the NN. Note the capitalized states, actions, and rewards (S_t, A_t, R_t) signify that they are sampled using the policy. Such a single function is needed in the policy gradient training algorithm described below.

4.5.2 Training Algorithm. As described in the background section, the REINFORCE algorithm is a result of direct policy differentiation where the goal is to maximize the expected return. However, the policy gradients suffers from high variance. This is in part alleviated

by using the "REINFORCE with baseline" algorithm, which calls for subtracting an appropriate baseline, b_t from the return:

$$\theta \leftarrow \theta + \alpha \mathbb{E}_{\pi_\theta} \left[(G_t - b_t) \nabla_\theta \log \pi_\theta(s_t, a_t) \right] \tag{8}$$

where the choice of b_t should leave the expectation of the gradient on the RHS unchanged. The expected reward G_t and baseline b_t are computed over a set of episodes generated based on a set of training job sequences. In our training algorithm, b_t is the average of the return over multiple episodic simulations,

$$b_t = \frac{1}{E} \sum_{i=1}^{E} G_t^i, \tag{9}$$

where $i = 1..E$ is an index for episode. The expectation on the RHS of equation (8) is taken simply by averaging across all job sequences and episodes involved in calculating the gradient prior to a parameter update.

In order to generate multiple episodes per job sequence during training, a softmax for a GPU is treated as a probability distribution, from which a sample is taken to determine which job to assign the GPU to. During inference, a GPU is assigned to the job slot with highest probability outputted by the corresponding softmax.

At each time step t of an episodes, the NN would be required to make a decision of which GPUs to assign to which job slot, with the option of GPUs not be assigned to any slot (i.e. the null job slot). Then the actions are implemented by the scheduler, followed by a calculation of the reward for the current step t. Then finally the time step t is advanced by 1, and a new job can arrive for this new time step.

We trained the policy gradient method with two kinds of training loops. First we trained using a method akin to the usual mini-batch gradient based training method in a static dataset scenario. This training loop is shown in Algorithm 1. Second, we trained the NN in a streaming data scenario.

In both scenarios, multiple sequence of jobs, representing job arrivals, are made before running finite length episodic simulations. Each job sequence represents an arrival of one job per time step t. Each epoch involves multiple job sequences.

A single job sequence is used to generate multiple episodes. At each step of an episode i, as usual, we calculate a cumulative discounted reward, the return G_t^i. An aggregate baseline for each time step, b_t, is calculated by averaging v_t^i across all episodes. A gradient, $\Delta\theta$, is accumulated across multiple job sequences, episodes, and time steps,

There are three main differences between [8] and our study with respect to the training algorithm. Firstly, in contrast, we test the performance of our model on separate test scenarios. Secondly, [8] employed epoch based learning, in which NN updates occur once per epoch. In our work, we update the NN multiple times per epoch. Thirdly, we additionally test the performance of our method in the face of changing simulation data.

The training loop for minibatch style training is shown in algorithm 1. Training and test job sequences are pre-made and remain static, and thus we refer to this as the **Non-Streaming training** method.

Algorithm 1 was modified in two small ways to simulate **Streaming Data training**. Firstly, before shuffling the job sequences, one would simply re-initialize a new set of J training job sequences every few epochs. Secondly, a new testing job sequence was generated for every epoch. This approach was used to investigate the potential for the NN to train with new incoming job sequences.

Algorithm 1: Minibatch Training with Static Job Sequences

```
Initialize network parameters θ
Initialize J training job sequences
Initialize batchsize B
for each epoch:
    Shuffle training job sequences
    for job sequence l = 1..J:
        for episode i = 1..E:
            Generate episode sequence ((S_t^i, A_t^i, R_t^i))_{t=1..T_i}
            for t = 1..T_max:
                G_t^i ← Σ_{k=t}^{T_max} γ^{k−t} R_k^i
        for t = 1..T_max:
            b_t ← (1/E) Σ_{i=1}^{E} G_t^i
            for episode i = 1..E:
                Δθ ← Δθ + (G_t^i − b_t) ∇_θ log π_θ^i(A_t, S_t^i)
        if l mod B is 0:
            θ ← θ + α (1/BE) Δθ
```

4.6 Simulation Environment

4.6.1 Modeling Cluster Topology. The RL scheduler is trained on a simulated environment. A neural network designed in Theano is used as a function approximator. All code is written in Python. The graph representation of the cluster was modelled using the NetworkX python package.

The main benefit of building a simulation environment is it enables quick testing of different ideas, without having to wait for real DL jobs to run on a cluster.

Intra and inter machine network topology is simulated by modeling the cluster as a graph. We assume that each machine in the cluster is an IBM Power8 Minsky box (model S822LC [2]), and that the cluster can consist up to 4 racks. The main idea was to model elements such as CPUs, GPUs, network cards and switches as nodes in the cluster and the communication links between them as edges. The full list of nodes modelled is shown in Table 1. Each machine consists of 4 GPUs, 2 CPUs, and 2 Infiniband cards. Two GPUs are connected to each CPU. This type of configuration was described in [3], which benchmarked the use of IBM's PowerAI library for performing DDL on a 256 GPU cluster of IBM Power8 Minsky boxes.

The following edge weights between the nodes were modelled: default bandwidth, run-time bandwidth, number of jobs per edge. The node and edge types modelled are described in table 1, and a visual representation of the nodes and edges are shown in a graphical representation of a portion of the cluster on the left hand side of Figure 2.

The main idea behind topology modeling is to count the number of jobs using each edge in the graph, that is, the number of jobs per edge. To do so, we need to keep track of the links used by GPU to GPU pair paths. With a large number of resources, it is infeasible

Table 1: Nodes and Edges modelled

Nodes	CPU, GPU, Infiniband card (IBC), Top-of-Rack Switch (TRS), Director Switch (DS) [1]
Edges (Default BW in GB/s)[2]	CPU-CPU (38.5), CPU-GPU(40), CPU-IBC(1000), IBC-TOR(12.5), TRS-DS(6.25)
Edges Weights [3]	Default BW, Run-Time BW,[4] Number of Jobs per Edge

[1] Only one Director Switch was simulated.
[2] BW = Bandwidth
[3] Each edge has three weights
[4] Run-Time BW = Default BW / Number of Jobs per Edge

to list all possible combinations of GPU-GPU assignments ahead of time. Thus we pre-compute the shortest paths between each unique GPU-GPU pair. During run time, for each job, we build up a set of edges, which is the set summation of all edges in the paths of all unique GPU-GPU pairs being used by a job. Obviously, if a job was only assigned a single GPU, that job would not contribute to the job count of any edge. The "Number of Jobs per Edge" edge weight value is used in determining the effective speed of simulated jobs. The speed of jobs become important in the reward formulation which is described in section 4.4.2.

4.6.2 Job Modelling. As already mentioned, our job picking and resource allocation method is meant for DNN type jobs. Here we discuss the attributes of the simulated DNN jobs that our scheduler tries to schedule as they arrive in our simulation, not the attributes of the NN function approximate used in our DRL method. Symbols representing modelled job attributes are suffixed with "(j)".

Let us assume for simplicity that the time it takes for a minibatch of data to feed into a GPU to be negligible compared to the time it takes for a job to complete one minibatch of computation. Furthermore, consider a job that runs on a single GPU, which has a Floating Points Operations Per Second (FLOPS) rating of v_{P100}. $P100$ is an NVIDIA GPU model. Let us think about the complexity of a DNN model, i.e. the Floating Point Operations (FLOPs not FLOPS) of a single forward pass on a single data example, as a distance per example, $d_{ex(j)}$.

The speed of a single GPU job, $v_{t(j)}$, at a given time step t during the simulation, can be expressed as follows and is roughly equivalent to speed of the GPU.

$$v_{t(j)} \approx \frac{d_{m(j)}}{tt_{m(j)}} \approx 0.9 v_{P100}, \tag{10}$$

where the 0.9 on right hand side is to simulate the fact that the actual FLOPS observed is less than the advertised FLOPS.

To use realistic values of attributes of DNN jobs, a table of 35 Convolutional Neural Network (CNN) architectures and their properties such as model complexity ($d_{ex(j)}$), gradient size (same as the memory footprint of all model parameters), etc. were conveniently obtained from [12]. During job initialization, it is randomly assigned a CNN architecture's model complexity, and gradient size.

Also during job initialization, the training time of a job's minibatch, $tt_{m(j)}$, is obtained from equation 10, where v_{P100} is a constant, $d_{m(j)} = d_{ex(j)} m_{(j)}$ is known because $d_{ex(j)}$ is simply the complexity a of model as reported by [12]. $m_{(j)}$ is a job's assigned mini-batch size randomly chosen as one of $\{32, 64, 128, 256, 512\}$. Finally, borrowing from [8], we sample from uniform distributions (see section 5.1) to assign a job a length $len(j)$ and number of resources (only GPUs in our case) requested by the job, $gpusreq(j)$.

During the simulation, a job finishes once it has undergone a total amount of computation, which we refer to as "total computational distance":

$$d_{tot(j)} = len(j) \times gpusreq(j) \times d_{cell}, \tag{11}$$

where $len(j)$ and $gpusreq(j)$ multiply to give the number of colored cells of a job slot (see figure 1), and d_{cell} is a calibrated computational distance associated with a single cell, in units of FLOPs. The details of this calibration is included in the Appendix A. The purpose of the calibration process is two fold. One, it ensures that the input image can represent the entirety of the computational distance of the job, whether that be in the queue, or in the cluster portion of the input. Secondly, the calibration associates a time value in seconds, t_{step}, with a row of the input image. Every time the simulation advances one step, the computational distance in FLOPs each job has "travelled" can be computed uniquely as $v_{t(j)} \times t_{step}$.

4.6.3 Reduction Time Measurement and Formulation. The reduction time of a job, $rt_{(j)}$ is modelled using a combination of measured data and a heuristic. In our modeling, it is expressed as follows:

$$rt_{(j)} = srrt_{(j)}(1 + (r-1)/5)s_{(j)} \tag{12}$$

The right hand side of equation 12 consists of three terms. The first, $srrt_{(j)}$, is the single rack ring reduction time, derived from ring reduction measurements on a single rack for up to 16 GPUs. The second factor in parentheses is a simple heuristic to model the increase in reduction time due to using increasing number of racks, r. The third, $s_{(j)}$ is a scale factor that accounts for bandwidth sharing. These three terms are explained in more detail below.

Single Rack Ring Reduction Time, $srrt_{(j)}$: Ring reduction measurements were taken for up to 4 machines on a single rack, for a total of 16 GPU's. It was measured for 100 MB and 500 MB gradient sizes, and the measurements are shown in Figure 3. The measurements were fitted with square root functions. In order to estimate the reduction time for different gradient sizes and GPU counts, we interpolate linearly between the two curves by drawing a vertical line at the desired GPU count. At 0 MB, we set the reduction time to be 0, and for all gradient sizes greater than 100 MB, we simply use the equation of the line between the 100 MB and 500 MB curves. We use the two fitted curves in Figure 3 to account for different number of GPUs allocated to a job, and interpolated between them as explained above, to account for different gradient sizes.

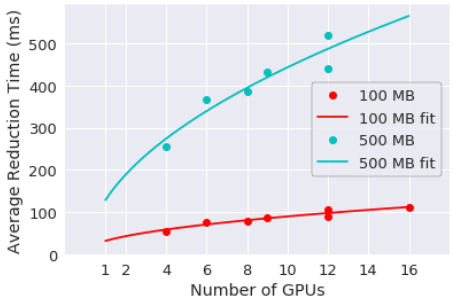

Figure 3: Single Rack Reduction Time measurements

Extrapolating Reduction time for Multiple Racks: Since at the time of measurement and data collection, reduction time measurements between racks were not available, we model between rack reduction times with the second term on the right hand side of 12. For example, if we require the reduction time of using 8 GPUs on 2 racks instead of 1 rack, we multiply the right hand side by $(1 + (2 - 1)/5) = 1.2$. Thus, for 2, 3, and 4 racks, this factor would be 1.2, 1.4 and 1.6 respectively.

Scale factor, $s_{(j)}$: Finally, the scale factor, $s_{(j)}$ is expressed as

$$s_{(j)} = \frac{limbw_{single(j)}}{limbw_{multi(j)}}, \tag{13}$$

where the numerator, $limbw_{single(j)}$, represents the limiting bandwidth of a job j if that job was running alone in the cluster. The denominator, $limbw_{multi(j)}$, represents the limiting bandwidth of a job j while there are multiple jobs running in the cluster. Given that a job j is assigned a set of GPUs, let $paths_{(j)}$ be the collection of shortest paths between unique GPU-GPU pairs among the assigned GPUs. Further, let $\mathbf{E}_{(j)}$ be the set of all undirected edges in $paths_{(j)}$. $limbw_{single(j)}$ and $limbw_{multi(j)}$ are defined as,

$$limbw_{single(j)} = \min_{e \in \mathbf{E}_{(j)}} \text{Default BW}(e) \tag{14}$$

$$limbw_{multi(j)} = \min_{e \in \mathbf{E}_{(j)}} \text{Run-time BW}(e). \tag{15}$$

To help remember the meaning of $limbw_{single(j)}$ and $limbw_{multi(j)}$, one can call them "single job limiting bandwidth" and "multiple job limiting bandwidth", respectively.

The main reason for using the scale factor is that it is impossible to pre-measure reduction times for the huge number of combinations of GPU to job assignments. The scale factor helps account for the difference between a job's reduction time due to sharing of bandwidth with other jobs during the RL simulation, versus the reduction time the job would have if it was running alone. This is needed since the reduction time measurements were carried out for one reduction process at a time on a cluster of four Minksy Boxes.

5 EVALUATION

We describe specifics of the cluster topology, dimensions of the NN, and the performance measures used for evaluation. Also, we briefly describe the non-topology aware baseline methods of comparison. Finally we present the results of our method.

5.1 Experimental Setup

We tested our method on a simulated cluster with a total of 8 machines on 4 racks, with 2 machines on each rack (which can be written down as [2,2,2,2]). Each machine has 4 GPUs, thus there is a total of 4 GPUs x 8 machines = 32 GPUs. Thus $M = 32$. Normally, one would keep as many machines on the same cluster as possible. However, as a proof of concept, we wished to investigate the effects of topology on the performance of our method compared to the non-topology aware baselines, while allowing a NN to train within a reasonable amount of time.

The number of job slots we used in the input image is $N = 10$, and the backlog could hold 60 jobs. The maximum $gpusreq(j)$ was limited to 12 per job, to limit the number of columns needed to represent a jobslot in the input image.

The number of hidden units used for the hidden layer was set to 1.5 times the total number of output units, rounded up. The total number of output units is $32 \times 11 = 352$. Updates to the NN parameters were done with an Adam optimizer and an initial learning rate of 0.001.

During each episode, one job per time-step would arrive for the first 400 consecutive time-steps. The maximum episode length, T_{max} was set to 1000. The number of episodes, E, per job sequence was set to 20. The number of job sequences, J, was different for the minibatch and streaming training experiments.

The length of the job, $len(j)$, and its resource request, $gpusreq(j)$, were sampled from uniform distributions. Roughly half the jobs had $len(j)$ between 1 and 3, and the other half between 6 and 10. Similarly, roughly half of the jobs had $gpusreq(j)$ between 1 and 7, and the other half between 8 and 12.

5.1.1 Non-Streaming vs Streaming Training. For non-streaming data training, a static set 200 training and 200 testing job sequences were created before the start of an epoch.

For streaming data training, a fresh set of 60 training job sequences were used every 10 epochs. At the end of every epoch, a fresh set of 60 testing job sequences were used for testing.

For both training scenarios, the training and testing performances were plotted for 50 epochs, every epoch.

5.2 Performance Measures

5.2.1 Mean Reward. To display the reward across all E episodes and J job sequences of a single epoch, the first return, G_1^i of each episode i is collected for every job sequence. Notice that the usual formula for the first return, G_1^i, is already an aggregate (a discounted sum) of all rewards of a single episode. In this paper, we refer to **Mean Reward** as the mean of all G_i^i across all job sequences and episodes of an epoch:

$$Mean\ Reward = \frac{1}{J}\frac{1}{E}\sum_j \sum_i G_1^i. \tag{16}$$

where job sequence index j is omitted in the return for simplicity.

5.2.2 Slowdown. In scheduling studies, it is also instructive to plot the slowdown. Slowdown for a single job is defined as,

$$Slowdown(j) = \frac{finishtime(j) - arrivaltime(j)}{len(j)}, \tag{17}$$

where $finishtime(j)$ is the time step at which a job finished. If by time step T_{max} there exists unfinished jobs, then those jobs are assigned T_{max} as the finish time. $arrivaltime(j)$ is the time-step at which a job arrives.

To associate an aggregate slowdown for an entire epoch, we calculate the **Mean Slowdown** by simply taking the average job slowdowns across all jobs that arrived during an epoch. This mean is taken across all job sequences and all episodes of an epoch.

5.3 Evaluation baselines

To compare with baseline job picking and resource allocation methods, we used the following static schedulers from [8]: Random, Shortest-Job-First (SJF), and Tetris. These schedulers only performed job picking, by selecting one job at a time among the jobs in the slots. The Random and SJF job pickers are self-explanatory. Tetris is based on [6]. Its implementation in our environment works by picking the job whose resource request $gpusreq(j)$, when multiplied by the number of available GPUs, leads to the largest value. For resource allocation for these schedulers, the behaviour of heuristic resource allocator from [8] was preserved. This baseline resource allocator simply numbered the resources of a certain type consecutively, then picked the first x resources to assign, where x is the number resources of a certain type requested by a job. The allocator would search for free resources in the cluster representation of the input image from left to right starting at the top of the image. For a given job, its resource allocations across the GPUs were required to begin at the same time step. Although this baseline resource allocator is not topology aware, we made sure to use the capabilities of our simulator to calculate the unique speed of jobs scheduled by these schedulers.

The Total Rewards and Average Slowdowns were also computed for these baseline schedulers. The extra penalties explained in section (4.4.2) were of course not used because they are not applicable to static resource allocators.

5.4 Performance Results

The results of measuring Mean Reward and Mean Slowdown for the non-streaming data and streaming data training experiments are shown in figure 4 and 5, respectively. The figures show that mean reward and slowdown performance of **RL-TAPS** is clearly better than the baseline schedulers. The baseline schedulers performed poorer because they do not learn to, nor even heuristically, take into account topology. In order for an NN to take into account topology, there must be some signal to guide it to understand that certain values of units in the input correspond to desirable choices made by the output. In the case of our prototype, this signal only came in the form of a topology-sensitive reward signal and current GPU to job assignments. To our surprise, the NN was able to learn even though the bandwidth usages of the topology were not explicit features in the input.

The non-streaming training method showed slightly better performance than the streaming training method with respect to the mean reward, but the performance with respect to mean slowdown were similar. This is good news as the end goal is to first train **RL-TAPS** in a simulation environment, but deploy and continue to train in a real cluster. The big benefit of detailed topology modelling

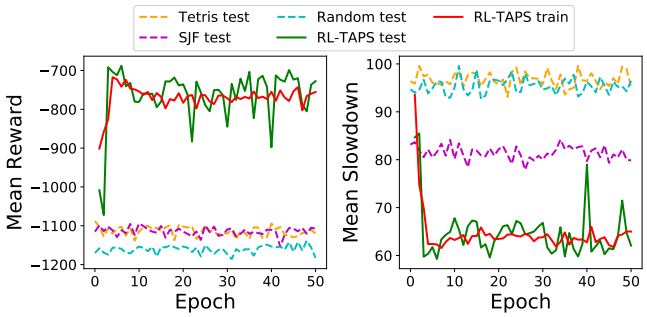

Figure 4: Mean Reward (top) and Slowdown (bottom) for the non-streaming data training scenario (see 5.1.1).

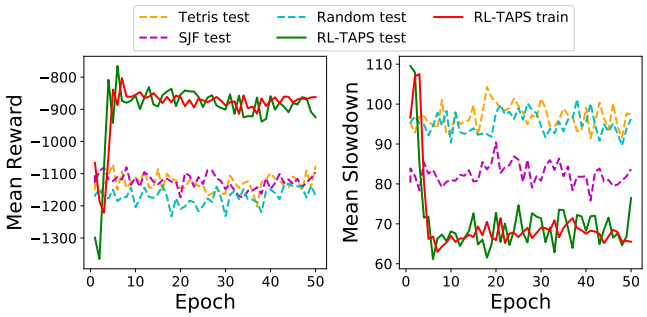

Figure 5: Total Reward (left) and Slowdown (right) for the streaming data training described (see5.1.1).

is that training can start in simulation and thus the scheduler could potentially be useful when it is deployed, without having to wait months to be trained.

6 DISCUSSION

Training NNs with the Policy Gradient method is often fraught with difficulties such as unstable policy parameters. Interesting and also concerning is that most of the learning in our experiment happens very early on and quickly plateaus. Further investigation is needed with regard to whether learning is stopping prematurely, or actually progressing well very quickly. With respect to the environment modelling, one limitation is that we have not yet modelled the cost of pausing and restarting DDL type jobs. Given the strong performance of **RL-TAPS** thus far, however, we are hopeful that **RL-TAPS** will still be capable of outperforming the no-preemption methods. Making this improvement may require some change to the input or the RL simulation. Another limitation currently is that the NN input design makes simulating large cluster sizes prohibitive. The jobslot representations are quite wasteful as many units of the input may end up with zeros. A fully connected hidden layer to a large input layer is not scalable. The large output search space also contributes to the high parameter count of our NN (roughly 1 million). Nonetheless, once trained, our NN completes a single inference step in less than 0.1 seconds on CPU. If incorporated into a production scheduler, **RL-TAPS** will not require many resources.

7 CONCLUSION AND FUTURE WORK

We proposed a deep reinforcement learning based scheduler that simultaneously selects jobs and assigns resources to them. Job selection and resource allocation are non-linearly related and this scheduler addresses the challenge associated with attempting to simultaneously solve these two dependent combinatorial optimization problems. Our scheduler **RL-TAPS** considers the topology of the environment and allows for full preemption. We evaluated the performance of **RL-TAPS** in different scenarios and compared it against various baselines demonstrating the efficiency and effectiveness of our method.

Going forward, we wish to address the scalability issue of the NN. In the short term, we will try reformulating the input into something that is more compact, with room to incorporate topology bandwidth information. There is also the need to solve additional assignment problems such as which reduction algorithm to assign to a job for both intra-machine and inter-machine communication. For example, Nvidia's NCCL library as well as IBM's DDL library consists of various different gradient reduction (i.e. averaging) algorithms for both within and between machine communication. The question of how to solve multiple simultaneous assignment problems without exploding the search space must be investigated.

In addition to the full preemption scheduling of **RL-TAPS**, we wish to explore other preemption methods mentioned in section 1. Designing a full-preemption NN was less complicated then designing an NN that can handle partial-preemption decisions due to the requirement of enforcing constraints in the latter. Currently, to our knowledge, there exists no NN architecture method that would allow one to enforce inequality constraints. The combinatorial optimization techniques with NNs to date have all avoided tackling such problems. We approached this issue in our work through penalties, but **RL-TAPS** fails to enforce them strictly. Likely this problem may require novel NN ideas to solve, and would be a very interesting endeavour.

REFERENCES

[1] Irwan Bello, Hieu Pham, Quoc V. Le, Mohammad Norouzi, and Samy Bengio. 2017. Neural Combinatorial Optimization with Reinforcement Learning. *ArXiv* (Jan. 2017). http://arxiv.org/abs/1611.09940 arXiv: 1611.09940.

[2] Alexandre Caldeira, M. Kahle, Gerard Saverimuthu, and K. C. Vearner. 2015. IBM power systems S822LC technical overview and introduction. *IBM Red Paper* (2015).

[3] Minsik Cho, Ulrich Finkler, Sameer Kumar, David Kung, Vaibhav Saxena, and Dheeraj Sreedhar. 2017. PowerAI DDL. (Aug. 2017). https://arxiv.org/abs/1708.02188

[4] Giacomo Domeniconi, Eun Kyung Lee, and Alessandro Morari. 2019. CuSH: Cognitive ScHeduler for Heterogeneous High Performance Computing System. In *DRL4KDD 19: Workshop on Deep Reinforcement Learning for Knowledge Discover.* 7.

[5] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. 2011. Dominant resource fairness: fair allocation of multiple resource types. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation (NSDI'11).* USENIX Association, Boston, MA, 323–336.

[6] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. 2014. Multi-resource packing for cluster schedulers. In *Proceedings of the 2014 ACM conference on SIGCOMM - SIGCOMM '14.* ACM Press, Chicago, Illinois, USA, 455–466. https://doi.org/10.1145/2619239.2626334

[7] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. 2019. Tiresias: A {GPU} Cluster Manager for Distributed Deep Learning. 485–500. https://www.usenix.org/conference/nsdi19/presentation/gu

[8] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. 2016. Resource Management with Deep Reinforcement Learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks (HotNets '16).* ACM, New York, NY, USA, 50–56. https://doi.org/10.1145/3005745.3005750

[9] Azalia Mirhoseini, Anna Goldie, Hieu Pham, Benoit Steiner, Quoc V. Le, and Jeff Dean. 2018. A Hierarchical Model for Device Placement. (Feb. 2018). https://openreview.net/forum?id=Hkc-TeZ0W

[10] Azalia Mirhoseini, Hieu Pham, Quoc V. Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. 2017. Device Placement Optimization with Reinforcement Learning. *arXiv:1706.04972 [cs]* (June 2017). http://arxiv.org/abs/1706.04972 arXiv: 1706.04972.

[11] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. 2018. Optimus: An Efficient Dynamic Resource Scheduler for Deep Learning Clusters. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys '18).* ACM, New York, NY, USA, 3:1–3:14. https://doi.org/10.1145/3190508.3190517 event-place: Porto, Portugal.

[12] Samuel. 2020. albanie/convnet-burden. https://github.com/albanie/convnet-burden original-date: 2017-08-04T10:11:16Z.

[13] Richard S. Sutton. 2018. *Reinforcement learning: an introduction* (second edition. ed.). The MIT Press, Cambridge, Massachusetts.

[14] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. 2017. Pointer Networks. *arXiv:1506.03134 [cs, stat]* (Jan. 2017). http://arxiv.org/abs/1506.03134 arXiv: 1506.03134.

[15] Ronald J. Williams. 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Mach Learn* 8, 3 (May 1992), 229–256. https://doi.org/10.1007/BF00992696

[16] Wei Zhang. 1996. Reinforcement learning for job-shop scheduling. (1996).

[17] Wei Zhang and Thomas G. Dietterich. 1995. A reinforcement learning approach to job-shop scheduling. In *IJCAI*, Vol. 95. Citeseer, 1114–1120.

[18] Wei Zhang and Thomas G. Dietterich. 1996. High-performance job-shop scheduling with a time-delay TD (λ) network. In *Advances in neural information processing systems.* 1024–1030.

A INPUT CALIBRATION

To integrate the simulation with the reduction time measurements, we associate a time span in seconds, $t_{rows(c)}$, with each row of the input image, and a computational distance for per cell, d_{cell}. The table below shows how $t_{rows(c)}$ is calibrated as the time in seconds it would take a vgg-vd-19 model to complete 1000 minibatch iterations using a single GPU, over the horizon of the input image.

Table 2: Formulas to derive $t_{row(c)}$ and d_{cell}

subscript c subscript f vgg-vd-19	calibration variable means final or total calibration DNN model
$d_{ex(c)}$	20×10^9 FLOPs, complexity of model
$m_{(c)}$	256, minibatch size of model
$d_{m(c)} = d_{ex(c)} m_{(c)}$	computational distance per minibatch
$it_{f(c)}$	1000, total number of job iterations
$g_{(c)}$	1, number of columns to represent the job
$d_{f(c)} = it_{f(c)} d_{m(c)} g_{(c)}$	total computational distance of the job
$v_{(c)} = v_{P100}$	GPU speed
$t_{f(c)} = \frac{d_{f(c)}}{v_{(c)}}$	total job run time in seconds
$rows_{(c)}$	10, number of rows (horizon)
$n_{(c)} = rows_{(c)} g_{(c)}$	number of highlighted cells for job in jobslot
$d_{cell} = \frac{d_{f(c)}}{n_{(c)}}$	calibrated computational distance per cell
$t_{row(c)} = \frac{t_{f(c)}}{rows_{(c)}}$	calibration time of each row in seconds