ΠΑΝΕΠΙΣΤΗΜΙΟ
ΘΕΣΣΑΛΙΑΣ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

Σχεδιασμός και Υλοποίηση Αλγορίθμων Τεχνητής Νοημοσύνης για το Interval Scheduling Problem

Γεώργιος Διαμαντόπουλος

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

ΥΠΕΥΘΥΝΟΣ

Νικόλαος Τζιρίτας
Επίκουρος Καθηγητής

Λαμία 30 Ιουλίου έτος 2021

ΠΑΝΕΠΙΣΤΗΜΙΟ
ΘΕΣΣΑΛΙΑΣ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

Σχεδιασμός και Υλοποίηση Αλγορίθμων Τεχνητής Νοημοσύνης για το Interval Scheduling Problem

Γεώργιος Διαμαντόπουλος

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

ΥΠΕΥΘΥΝΟΣ

Νικόλαος Τζιρίτας
Επίκουρος Καθηγητής

Λαμία 30 Ιουλίου έτος 2021

UNIVERSITY OF
THESSALY

SCHOOL OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE & TELECOMMUNICATIONS

Design and Implementation of Artificial Intelligence Algorithm for the Online Interval Scheduling Problem

Georgios Diamantopoulos

FINAL THESIS

ADVISOR

Nikolaos Tziritas
Assistant Professor

.

Lamia July 30 year 2021

«Με ατομική μου ευθύνη και γνωρίζοντας τις κυρώσεις (1), που προβλέπονται από της διατάξεις της παρ. 6 του άρθρου 22 του Ν. 1599/1986, δηλώνω ότι:

1. Δεν παραθέτω κομμάτια βιβλίων ή άρθρων ή εργασιών άλλων αυτολεξεί ***χωρίς να τα περικλείω σε εισαγωγικά*** *και χωρίς να αναφέρω το συγγραφέα, τη χρονολογία, τη σελίδα. Η αυτολεξεί παράθεση χωρίς εισαγωγικά χωρίς αναφορά στην πηγή, είναι λογοκλοπή. Πέραν της αυτολεξεί παράθεσης, λογοκλοπή θεωρείται και η παράφραση εδαφίων από έργα άλλων, συμπεριλαμβανομένων και έργων συμφοιτητών μου, καθώς και η παράθεση στοιχείων που άλλοι συνέλεξαν ή επεξεργάσθηκαν, χωρίς αναφορά στην πηγή. Αναφέρω πάντοτε με πληρότητα την πηγή κάτω από τον πίνακα ή σχέδιο, όπως στα παραθέματα.*

2. Δέχομαι ότι η αυτολεξεί ***παράθεση χωρίς εισαγωγικά****, ακόμα κι αν συνοδεύεται από αναφορά στην πηγή σε κάποιο άλλο σημείο του κειμένου ή στο τέλος του, είναι αντιγραφή. Η αναφορά στην πηγή στο τέλος π.χ. μιας παραγράφου ή μιας σελίδας, δεν δικαιολογεί συρραφή εδαφίων έργου άλλου συγγραφέα, έστω και παραφρασμένων, και παρουσίασή τους ως δική μου εργασία.*

3. Δέχομαι ότι υπάρχει επίσης περιορισμός στο μέγεθος και στη συχνότητα των παραθεμάτων που μπορώ να εντάξω στην εργασία μου εντός εισαγωγικών. Κάθε μεγάλο παράθεμα (π.χ. σε πίνακα ή πλαίσιο, κλπ), προϋποθέτει ειδικές ρυθμίσεις, και όταν δημοσιεύεται προϋποθέτει την άδεια του συγγραφέα ή του εκδότη. Το ίδιο και οι πίνακες και τα σχέδια

4. Δέχομαι όλες τις συνέπειες σε περίπτωση λογοκλοπής ή αντιγραφής.

Ημερομηνία: 30/7/2021

Ο Δηλών.

*(1) «Όποιος εν γνώσει του δηλώνει ψευδή γεγονότα ή αρνείται ή αποκρύπτει τα αληθινά με έγγραφη υπεύθυνη δήλωση
του άρθρου 8 παρ. 4 Ν. 1599/1986 τιμωρείται με φυλάκιση τουλάχιστον τριών μηνών. Εάν ο υπαίτιος αυτών των πράξεων
σκόπευε να προσπορίσει στον εαυτόν του ή σε άλλον περιουσιακό όφελος βλάπτοντας τρίτον ή σκόπευε να βλάψει άλλον, τιμωρείται με κάθειρξη μέχρι 10 ετών.»*

ΠΕΡΙΛΗΨΗ

Το Online Interval Scheduling Problem έχει πρόσφατες εφαρμογές στον τομέα του cloud computing για την βελτιστοποίηση της ανάθεσης διεργασιών σε εικονικές μηχανές για επεξεργασία. Με την μεγαλύτερη κατανάλωση ενέργειας να γίνεται για την επεξεργασία δεδομένων και την ψύξη, ένας πιο βέλτιστος προγραμματισμός των διεργασιών μπορεί να ελαττώσει το busy time και ως συνέπεια την κατανάλωση ενέργειας. Στο non-preemptive clairvoyant online interval scheduling η αναπαράσταση των διεργασιών γίνεται με ένα ημιανοιχτό διάστημα της ώρας άφιξης και αναχώρησης και επιπλέον τις απαιτήσεις της διεργασίας σε πυρήνες. Ένας επιπλέον περιορισμός του online προβλήματος είναι το ότι δεν έχουμε γνώση για τις μελλοντικές αφίξεις διεργασιών. Αυτή η δουλειά βασίζεται στον Threshold Based Categorisation (TBC) αλγόριθμο ο οποίος δουλεύει ως ένα βήμα προεπεξεργασίας για τους παραδοσιακούς αλγόριθμους προγραμματισμού διεργασιών. Ο TBC βασίζεται σε μια τιμή κατώφλι T η οποία χρησιμοποιείται για να καθαρίσει αν δύο διεργασίες είναι αμοιβαία αποκλειόμενες και ως συνέπεια δεν μπορούν να μπουν στην ίδια μηχανή. Βασικό για την καλή απόδοση του αλγορίθμου είναι ή επιλογή της τιμής T. Για την επίλυση αυτού του προβλήματος 2 πράκτορες ενισχυμένης μάθησης, ο Q-Learning και ο Deep Q-Learning, υλοποιήθηκαν και εκπαιδευτικάν σε πραγματικά δεδομένα ώστε να μάθουν να επιλέγουν το βέλτιστο T βάση την παρελθοντικές και αναμενόμενες μελλοντικές απαιτήσεις του server. Η απόδοση των πρακτόρων μετρήθηκε σε δεδομένα που οι πράκτορες δεν είχαν ξανά δει και τα αποτελέσματα επιβεβαιώνουν ότι οι πράκτορες όντως προσαρμοζονται στις διάφορες απαιτήσεις και πετύχουν καλύτερα αποτελέσματα από τον TBC με σταθερό T και τον First Fit.

ABSTRACT

The Online Interval Scheduling Problem has recently been applied in the field of cloud computing for optimising the packing of incoming tasks into virtual machines (VM) for processing. With the largest part of the power consumption of data centers arising from processing and cooling, a more optimal packing reduces the busy time of the system and thus the total power consumption. Specifically in the non-preemptive clairvoyant online interval scheduling problem, arriving jobs are represented by the semi-open interval of their arrival and departure time and their processing requirements. An additional constraint of the online problem is the restriction of knowledge for future job arrivals. This work relies on the Threshold Based Categorisation (TBC) algorithm which works as a preprocessing step to traditional bin packing algorithms. TBC relies on a threshold value which is used to determine whether two tasks are mutually exclusive and thus can not be packed in the same bin. Critical for the performance of this algorithm is the selection of a fitting threshold value. To solve this problem two reinforcement learning agents are proposed, specifically a Q-Learning and a Deep Q-Learning agent, which are trained on real world data to learn to select optimal thresholds based on the server's past and expected load. The agent's performance is evaluated on previously unseen data and the experimental results show that the agents learn to adjust to many different workloads and outperform both the TBC with static threshold strategy and First Fit.

Table of Contents

ΠΕΡΙΛΗΨΗ I
ABSTRACT III

CHAPTER 1 Introduction 2

(Section 1.1 Online interval scheduling) 2

(Section 1.2 Threshold Based Categorisation) 2

(Section 1.3 Proposed Algorithm) 3

CHAPTER 2 Related Work 4

CHAPTER 3 System Model 6

CHAPTER 4 Agent Architecture 7

(Section 4.1 Q-learning) 8
(subsection 4.1.a Introduction to Q-learning) 8
(subsection 4.1.b Temporal Differences) 9

(Section 4.2 Reward Function) 10

(Section 4.3 TD and Unexplored State Action Pairs) 11

(Section 4.4 State Definition) 11

(Section 4.5 Deep Q-Learning) 15
(subsection 4.5.a Introduction to deep Q-learning) 15
(subsection 4.5.b Training the q network) 16

(Section 4.6 Training the agents) 17

CHAPTER 5 Experimental results 18

(Section 5.1 Simulator) 18
(Section 5.2 Results) 18

CHAPTER 6 Conclusion 22

REFERENCES 23

CHAPTER 1 Introduction

(Section 1.1 Online interval scheduling)

The Interval Scheduling problem or Bin Packing problem is an optimization problem regarding the packing of varying size items into a single or many bins with the goal of packing as many items as possible into one bin or using the least number of bins to pack all items. Depending on the formulation, this problem can be used for optimization in a plethora of applications such as reducing thrashing when allocating memory to tasks in an operating system [19] or for packing cargo in aircrafts[20]. The focus of this work is on a specific formulation that applies to large cloud servers with the goal of minimizing power consumption. This specific problem has recently attracted much attention from the research community [1], [2], [3], creating an influx of new algorithms. Such environments have massive computational capabilities which come coupled with equally massive energy requirements. With the highest power consumption attributed to cooling and processing [9],[10] improving the optimality of the algorithms used in such environments is critical for reducing the need for both aforementioned factors. The benefits of such a reduction are both environmental and monetary by simultaneously reducing the carbon footprint and the cost of running a data center.

The specific formulation for the cloud server case is the following. Jobs are described by a semi-open interval denoting the job's starting and departing time as well as the job's demand in CPU cores. Each job that arrives must be packed in a bin at its arrival time (strict time) and remain there until it has finished processing without its execution being paused (non-preemptive scheduling). The online version of this problem introduces the constraint that there is no knowledge of future job arrivals and thus is closer to the real-world problem in which we only become aware of the jobs when they arrive. The goal of the algorithms solving this problem is to minimize the total busy time. The busy time of a bin is simply the duration of its operation, from the initialization time until the last job is processed. The total busy time of the system is the sum of the busy times of every bin that was opened in order to process all the jobs.

(Section 1.2 Threshold Based Categorisation)

If we define local parallelism as the number of jobs that run in parallel in the same bin it is easy to correlate higher local parallelism with reduced busy time. But that is not always the case. In [8], the authors proved that packing jobs of similar size together, in dense workloads, reduces busy time overall as demonstrated in figure 1 workload 2. To exploit this, the authors proposed Threshold Based Categorisation (TBC), an algorithm which works as a preprocessing step to any traditional bin-packing algorithm such as First Fit [11]. The algorithm works using a parameter called the Threshold (T) which as shown experimentally, outperforms the state-of-are algorithms, and improves over the worst case of First Fit especially in dense workloads. This parameter T heavily

influences the algorithm's performance, and choosing an optimal value largely depends on the situation.

(Section 1.3 Proposed algorithm for dynamic threshold)

This work focuses on designing and implementing a machine learning based approach for choosing an optimal T value. For this task, two different agents, a Q-learning [17], [12] agent, and a Deep Q-learning [18] agent are employed to update the T values in order to minizine the busy time. The agents can choose between the TBC algorithm with any threshold (in the defined range) and the First Fit algorithm. Allowing the agent to use First Fit when the arriving jobs are sparse, and TBC with an appropriate T on dense workloads. Here sparse and dense are used as an example, in reality the decisions are taken using a more complex set of metrics.

CHAPTER 2 Related Work

To my knowledge there is yet for work to appear in the literature in which the online or the offline interval scheduling problem is tackled using machine learning techniques. The current state of the art approaches for optimizing both the online and the offline interval scheduling problem mostly rely on traditional algorithms and some heuristics approaches all focused around improving the competitive ratio which is the ratio of the online algorithms performance to the best case performance.

There are many works in the literature focusing on the offline version of the problem but are omitted here since the approaches for solving the offline version of the problem are significantly different that those for the online. In [4], the authors compute and compare the competitive ratios for the traditional packing algorithms of First Fit, Best Fit and Any Fit and also propose Hybrid First Fit which defines a variable beta and classifies tasks with size larger that $\frac{1}{beta}$as large and the rest as small with the goal of packing large and small tasks separately. Finally they prove that the algorithm achieves a competitive ratio of $\frac{5}{4}m + \frac{19}{4}$ when m is not known and $m + 5$ when m is known, where m is the ratio of the durations of the largest and smallest task.

In [5], the authors consider the same problem as this work, the Clairvoyant Online Interval Scheduling problem. The authors improve the previous upper bound of $O(\frac{logm}{loglogm})$ to $O(\sqrt{logm})$. The two algorithms proposed by the authors are the Hybrid Algorithm and the Classify-Duration-First-Fit. The Hybrid Algorithm uses two types of bins: general bins (GN) and classify by duration bins (CD). At any moment, based on the load the algorithm can decide whether to pack the task in a GN or CD bin. Finally, the Classify-Duration-First-Fit algorithm keeps groups of bins. Each group can only accommodate tasks that belong in the same category. Each task is classified into a category based on its duration

In [6], the authors consider the online flexible tasks scheduling problem which is similar to the online interval scheduling problem with the main difference being that tasks are described by their arrival time, starting time, and processing time allowing the scheduler to pack the task anytime between its arrival and starting time. Of course this allows for further optimization of the scheduling. The authors propose algorithms for both the clairvoyant and non-clairvoyant case, Batch and Batch+ for the former and Classify-By-Duration Btach+ (CDB+) and Profit for the latter. The Batch and Batch+ algorithms work iteratively and rely on starting the execution of as many tasks as possible in parallel. In each iteration the task with the shortest deadline is marked with a flag, and all tasks in that iteration begin processing at the same time as the flag task. Batch+ arguments Batch by additionally starting the execution of tasks arriving during the processing of the initially started jobs. CDB+ is used in the clairvoyant case where task durations are known; the algorithm first, given a category size, creates bins and each arriving task is put into a category based on its duration; finally every category is packed separately using Batch+. The Profit algorithm is similar to Batch+ with the

main difference being that when the flag tasks start their execution the algorithm packs the tasks that have arrived before the flag task (profitable tasks) with the arriving tasks being packed in the same iteration. The process is repeated for the non-profitable tasks of that iteration. Finally the competitive ratio of Batch+ and profit are computed at $m + 1$,where m is the length of the taks with the maximum length, for the former and at $4 + 2\sqrt{2}$ for the latter.

In [7], the authors consider two versions of the online clairvoyant interval scheduling problem, minimizing the busy time and maximizing the tasks packed while not exceeding a predetermined busy time. The algorithm BucketFirstFit(A) is a general case greedy algorithm. Each job is assigned to a category based on its duration on arrival with each category being packed on a separate set of bins. The parameter 'A' denotes the number of categories. Finally, the authors prove that for a=4 the competitive ratio of the algorithm is $5logm$ where m is the length of the taks with the maximum length.

This work heavily relies on the TBC algorithm [8] .The TBC algorithm is designed to be used in combination with traditional bin packing algorithms as a pre-processing step. TBC's main functionality is to only allow non-mutually exclusive tasks to be packed together in the same bin. Two tasks with durations dur1, dur2 are considered non-mutually exclusive if their duration ratio is less than the mutual exclusivity threshold (T).

$$\frac{max(dur1, dur2)}{min(dur1,\ dur2)} < T \quad \text{eq. 1}$$

The above can be extended to apply to the bin case. Using the maximum duration (max_dur) and the minimum duration (min_dur) in that bin, a task with duration (dur) can be packed in that bin if it satisfies the above for both the largest and smallest task in the bin.

$$\frac{max(max_dur, dur)}{min(max_dur,\ dur)} < T \quad \text{and} \quad \frac{max(min_dur, dur)}{min(min_dur,\ dur)} < T \text{ eq. 2}$$

The satisfaction of the eq.2 guarantees that there is no other task in the bin for which eq. 1 is not satisfied and thus the bin belongs to the non mutually exclusive set of bins for the current task.

Using the above, for every task, we can compute a subset of bins that only contain non-mutually exclusive tasks and thus are the most optimal candidates for the task to be packed in. In the case where there is no bin satisfying the above equation, a new VM is opened for the task to be packed in.

CHAPTER 3 System Model

Formally, the problem can be described by a set of jobs $J = \{J_1, J_2 \ldots J_n\}$ denoting the jobs arriving at a large parallel system that need to be scheduled to a bin for processing. A job J_n is described by a semi-open interval I = [arival_time, departure_time) and its processing requirements in CPU cores, denoted by R_n. The processing requirements of each job are known on arrival (clayvorant case of online interval scheduling). Finally, for each job J_n in the set of incoming jobs, the job's duration D_n can be calculated by subtracting the job's arrival time from its departure time. The scheduler is responsible for opening bins according to the packing algorithm to accommodate all jobs in J. Each bin V_m can be described as a set of time intervals $TI_m = \{TI_1, TI_2 \ldots TI_k\}$, and a capacity C_m which denotes the number of CPU cores available to that bin. Each TI_m represents a time interval in which the current CPU cores used in that bin stay the same. When a job is packed into a bin, the intervals are updated in such a way to represent the new usage of CPU cores caused by the job. As a constraint, interval's CPU cores usage must not exceed the capacity C of the bins at any point in time. In the case where no bin can accommodate a job, a new bin is opened. Finally, when a bin is done processing all of its assigned jobs it is shut down by the scheduler.

The goal is to pack a workload J, such that to minimize the busy time of the system which is the total time the system is under load. The busy time of a bin V_l can be calculated by subtracting the closing time from the opening time. Formally, the total busy time of the system BT_{sys} can be calculated using the following

$$BT_{sys} = \sum_{i=1}^{total\ bins} BT_{V_i}$$

Where BT_{V_i} is the total busy time of the i-th bin in the system.

CHAPTER 4 Agent Architecture

The TBC algorithm heavily relies on the threshold for optimizing lower bounds. Although the use of a static threshold can yield better performance over traditional algorithms in some cases, especially in the offline version of Interval Scheduling where there is knowledge about the load and more sophisticated algorithms are possible, this approach is not very effective. The alternative to using a static threshold is the use of an algorithm that is tasked with dynamically adjusting the threshold.

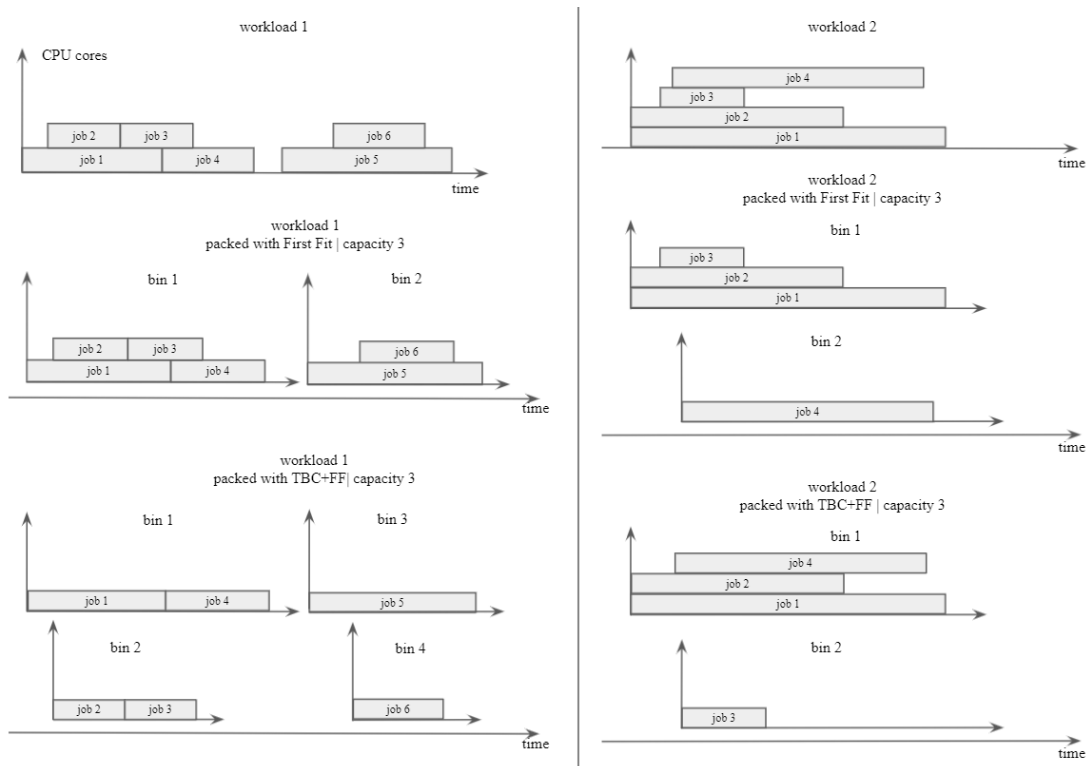


Figure 1: Packing of two simple workloads using the First Fit algorithm and the TBC algorithm (assuming appropriate threshold to achieve illustrated packing). Workload 1 is sparse, thus First Fit achieves optimal packing while Workload 2 is dense and TBC achieves optimal packing.

Figure 1 demonstrates the differences between First Fit and TBC. In workload 1, when applying the TBC algorithm, jobs 1 and 4 are mutually exclusive with jobs 2 and 3 and thus are packed in different bins; the same is also true for jobs 5 and 6. On the other hand, First Fit packs these jobs together. Workload 1 also demonstrates why the TBC algorithm is not suitable for sparse workloads since choosing to split a sparse workload only increases the busy time without getting the benefits. On the other hand, workload 2 shows how the TBC algorithm can achieve better packing results in dense sets. By summing the total busy times of bins 1 and 2 for both cases we can see that TBC's total busy time is the sum of duration of job1 and job3 while First Fit's is the sum of duration of job1 and job4. Because the duration of job4 is greater than that of job3, TBC result in an inferior packing against First Fit.

Even in favorable workloads for the TBC algorithm an initial spike in busy time can be observed. For example, in the workload 2 shown in figure 1 the busy time achieved by TBC before job4 arrives is higher than that of First Fit, but that 'sacrifice' is rewarded later when job4 arrives and bin1 still has available capacity for it. Therefore, in this scenario TBC outperforms First Fit. Based on the aforementioned scenarios the goal is to design reinforcement learning agents (see next subsections) that dynamically adjust the threshold for packing jobs on different machines based on their durations.

(Section 4.1 Q-learning)

(subsection 4.1.a Introduction to Q-learning)

The first agent used to solve this problem is a Q-learning [17] agent which uses the Temporal Differences [12] update rule. Q-learning is a model-free algorithm, which means that there is no model of the environment. A typical model of an environment consists of the Transition Probability Distribution (TPD) and a Reward Function. The Transition Probability Distribution is a table containing all the probabilities for every possible transition from each state and the reward function, in the model point of view, is one in which all rewards of possible state action pairs are known. In such problems the optimal solution can be found using Dynamic Programming or an approximation of the optimal solution can be found with the use of heuristic algorithms. With the above absent, model-free algorithms are implemented such as to learn the environment through experience but with no guarantees for optimality.

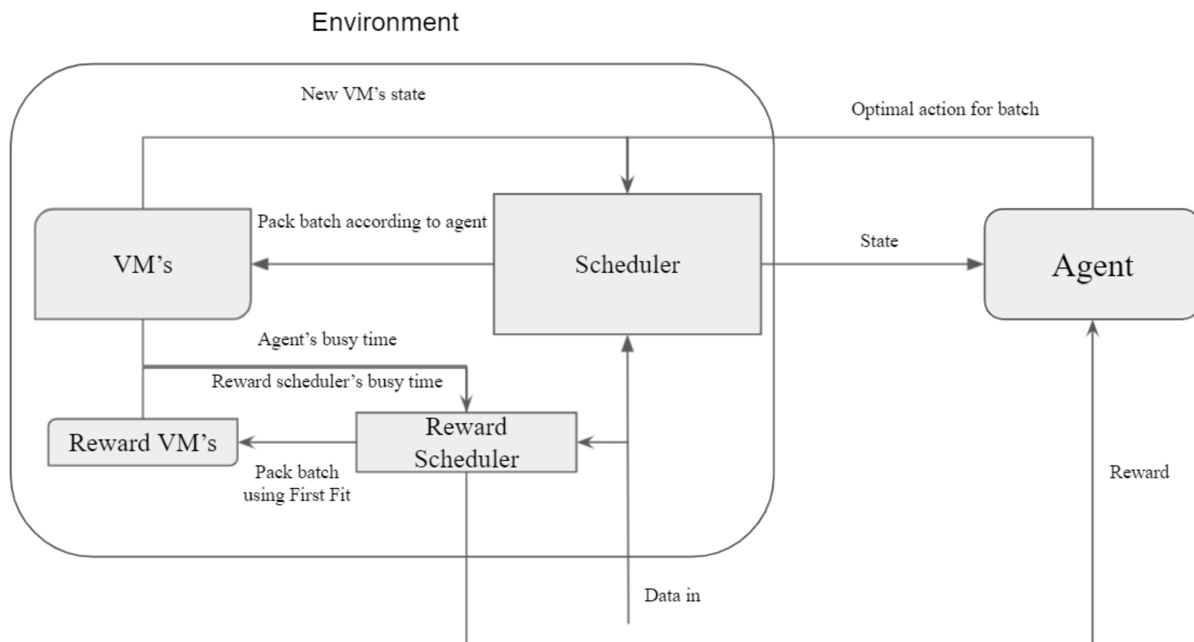


Figure 2: Overview of the algorithm. The agent acts on the environment by selecting an action and gets rewarded according to the optimality of that action.

The environment (Fig. 2) is a model of the real-world problem used for the agent to explore and gain experience on the problem. Each time the agent makes a decision it directly affects the environment and based on the change this action caused, the agent

gets a reward. Specifically for this problem, the environment is a class that contains the scheduler and other auxiliary metrics used to calculate the reward and provide the agent with the next state.

Q-learning learns the environment through the maximization of the Q function. The Q function is computed at each time step to calculate the expected reward of each action. The environment in which the agent is placed provides the agent with the current state S_t and with the reward of the action taken in the previous time step A_{t-1} (see fig 2). The agent then, using the expected rewards of all possible actions calculated by the Q-function, will choose the action A_t which corresponds to the maximum future reward.

The Q-values are stored in an N_1 x N_2 x … x N_p x M matrix Q where N_1 is the range of discrete values of the metric representing state 1 the state, N_2 is the range of discrete values of the metric representing state 2 and so on, P is the number of different metrics that are used to represent the state, and M is the number of all possible actions. Each state can be represented as a P dimensional vector S with S_1 being the value of the metric that represents state 1, S_2 being the value of the metric that represents state 2, and so on. At each time step t, the agent first updates $Q[(S_{t-1}, A_{t-1})]$ Q-value to be closer to the R_{t-1} plus the expected future reward using the TD learning update which will be explained in detail in the next subsection. Finally, before the training can begin the Q-table must be initialized. There are many methods of initializing the Q-values such as an optimistic initialization in which the Q-table is initialized with high Q-values thus promoting exploration, using random values, or all 0's or 1's or any number heavily depending on the reward function. For this problem, the initialization chosen was all 0's which will be discussed later in this chapter. Figure 3 illustrates the architecture of a q-learning agent.

(subsection 4.1.b Temporal Differences)

The update of Q-values happens using Temporal Differences (TD) learning. In TD, updates take place after every time step. This constant update can help the agent stay up to date and adapt to new workloads. The TD update rule for Q-learning is the following.

$$Q_{new}(S_t, A_t) = Q_{old}(S_t, A_t) + LR * (R_t + G * max(Q_{future}(S_{t+1})) - Q_{old}(S_t, A_t))$$

Where LR is the learning rate, R is the reward and G is the discount factor. The learning rate defines the impact the update will have on the old value. Choosing a balanced LR is important since both a very large and a very small LR can prevent convergence. One advantage of this update rule is its ability to maximize long-term rewards. This is achieved by taking into account the max expected future reward. Consider the case of chess. Generally, a move that allows the agent to capture the opponent's queen will have a high reward. But in the edge case where capturing the opponent's queen allows a checkmate for the opponent (losing the game for the agent), such a move should be avoided. An agent which does not take into account expected future rewards would fail

to get good results in chess and generally in any problem which requires long-term decision making. The TBC algorithm has a similar property, that is, choosing a lower T value can temporarily increase the busy time over the First Fit algorithm since a new bin will be opened to pack a mutually exclusive task that could have been packed in an existing bin. An agent acting on current results will never choose to use the TBC algorithm, since from its point of view it would never maximize the reward. Q-learning with TD overcomes this hurdle by using the max future reward in the update rule and a result reflecting future rewards on current actions.

If the above formula is studied carefully, one can deduce that for continuous serial updates the discount factor G is raised to 2nd power for the second call, to the 3rd for the third call, and so on. By lowering the value of G we effectively reduce how much the agent will take into account future rewards. G is bounded in the range of [0,1]. Considering the extreme cases, a G value of 1 would make the agent try to maximize the reward given after an infinite time or put differently, taking into account reward after an infinite amount of time with the same weight as the current reward. The opposite is true for a G value of 0 , which would make the agent completely disregard the future and only maximize the current reward.

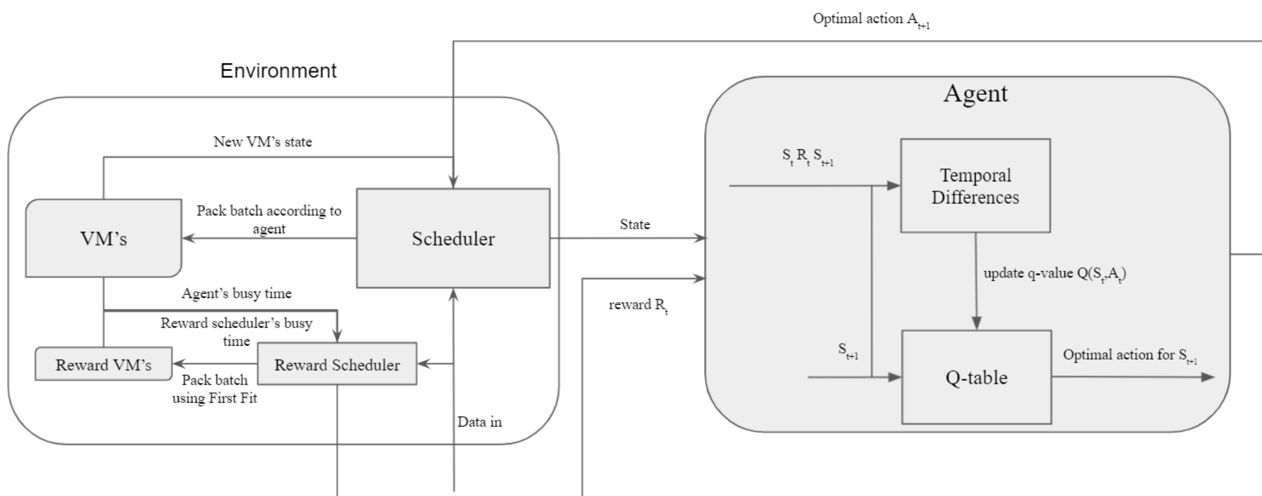


Figure 3: Overview of the architecture of a q-learning agent.

(Section 4.2 Reward Function)

Choosing a reward function for this type of problem is a difficult task since the final goal is not known. In traditional problems such as chess, the reward function is simple, e.g. +1 for winning, 0 for draw, and -1 for defeat. Considering this specific problem, rewarding the agent using the optimal packing is impossible since the problem is NP-Complete. Therefore, a different approach must be adopted to solve this problem.

The reward function that is chosen is based on the performance of the agent's packing as compared to the packing of the jobs by the First Fit algorithm. To calculate this value a secondary scheduler is placed in the environment which packs the same jobs as the

agent in parallel. When a new batch arrives and the agent is ready to receive its reward for the previous task, the total busy time achieved by the main scheduler and the reward scheduler are measured and their difference is given as the reward. Formally, for state S_t in which a batch B_n is packed, the reward R_t is calculated using the following formula

$$R_t = BT_{reward\ sch} - BT_{main\ sch}$$

Using this formula, in case the agent outperforms the First First algorithm it is rewarded with the positive busy time difference, in case the agent achieves the same performance with the First Fit algorithm it is rewarded 0, and finally if the agent is performing worse than the First Fit algorithm, the agent is rewarded with the negative busy time difference. Such a reward function steers the agent towards the right direction and does not limit its performance in any way since the better it performs the higher the agent's reward.

(Section 4.3 TD and Unexplored State Action Pairs)

When using the TD rule, more specifically when calculating the max expected future reward, an unexplored state can be chosen. Consider the simple case of a Q-learning agent with two possible actions A1 and A2. When trying to calculate the max expected reward of state S_{n-1}, $\max(Q(S_n))$ is calculated. If we assume that, $Q(S_n, A1) = -10$ and $Q(S_n, A2) = 0$ the max expected future reward will be $Q(S_n, A2)$ which is correct if the state $Q(S_n, A2)$ has been visited before and its Q-value represents reality. However, if we consider the case where the state has not been visited and its true value is not yet calculated, the risk of $Q(S_n, A2) < -10$, for example, $Q(S_n, A2) = -100$, is possible hence causing $Q(S_{n-1}, A_{n-1})$ to diverge from its true value. To solve this problem a secondary matrix $V_{explored}$ with the same dimensions as the Q-table is created. This matrix holds information about whether a state has been visited. Then when choosing the max expected future reward a conditional max function is used which returns the maximum of only explored state-action pairs or 0 in the case where no future state is explored.

$$Q_{new}(S_t, A_t) = Q_{old}(S_t, A_t) + LR * (R_t + G * cond_max(Q_{future}(S_{t+1}), V_{explored}(S_{t+1})) - Q_{old}(S_t, A_t))$$

(Section 4.4 State definition)

The states are the agent’s inner representation of the environment and are the sole information used by the agent to make decisions. Defining the states of the environment is a critical part of the design to ensure convergence and therefore learning. When choosing states the goal is to minimize dimensionality while maximizing useful information about the environment. Dimensionality in this context refers to the

dimensions and the size of each dimension of the Q-table. The maximization of useful information is a bit less strictly defined. In more simple problems the most useful and complete metrics to describe the environment are straightforward, for example, a smart thermostat trying to keep the temperature of a room at a constant temperature set by the user would need information about the indoor temperature of the house and maybe the outside temperature and the humidity. In more complex problems though, such simple metrics alone will rarely suffice.

Before explaining the state's metrics in depth, some more information about the structure of the data must be given. To model the progression of time in the agent, the workload is split into batches each of duration H, where H is how often the agent will update the threshold. Formally, a workload J split into batches can be denoted as the set $J = \{B_1, B_1, \ldots B_n\}$ where batch B_1 contains jobs that arrived in the first time window of duration H, B_2 contains jobs that arrived in the second time window of duration H and so on.

To calculate the new state of the environment when the new batch B_n arrives, the jobs of the previously packed batch B_{n-1}, and the first job of the new batch B_n are used. The decision is taken when the first job of the new batch has arrived. The derived state is used to calculate the optimal action to pack B_n. The first batch B_0 is packed using the First Fit algorithm since no information about the state is available. One thing to clarify here is that the packing strategy stays the same for the whole batch.

Sparsity is the ratio of the sum of the duration of the M_n jobs in a batch B_n divided by the total busy time of the jobs when packed in a bin with infinite capacity with all empty intervals (idle time) removed.

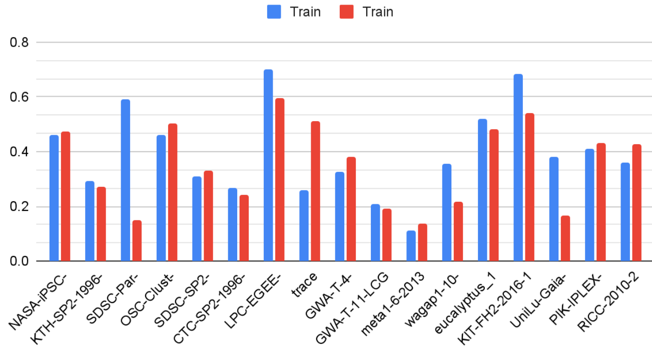
$$Spar_{B_n} = \frac{\sum_{i=1}^{Mn} durB_i}{BusyTime(B_n)}$$

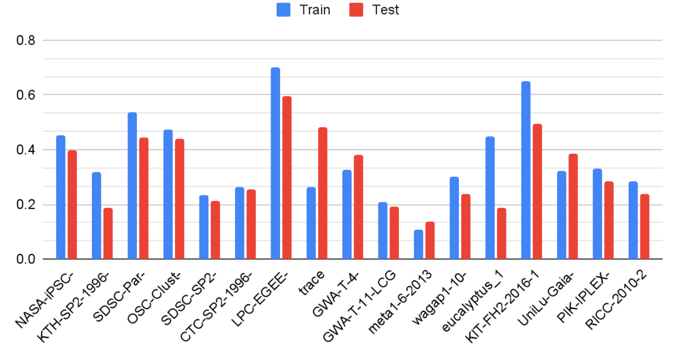
by definition $\sum_{i=1}^{m} durB_i \geq BusyTime(B_n)$

hence $0 < Spar_{Bn} \leq 1$

Sparsity was chosen because it heavily affects the packing strategy since sparse sets are best packed using First Fist due to the fact that not enough jobs arrive close enough to each other for TBC to achieve a more optimal packing. Furthermore, using the TBC algorithm in sparse datasets can further increase the busy time as illustrated in figure 1.

One limitation of the Q-learning algorithm is that states can not be continuous due to the nature of the Q-table. So every continuous variable must be converted into a discrete one. In the case of sparsity, the continuous range of (0,1] is converted into 11 discrete states from 0 to 10 with a step of 1 with the conversion happening by rounding down to 1 digit precision and multiplying by 10. Figure 4 shows the mean sparsity of batches for every workload.





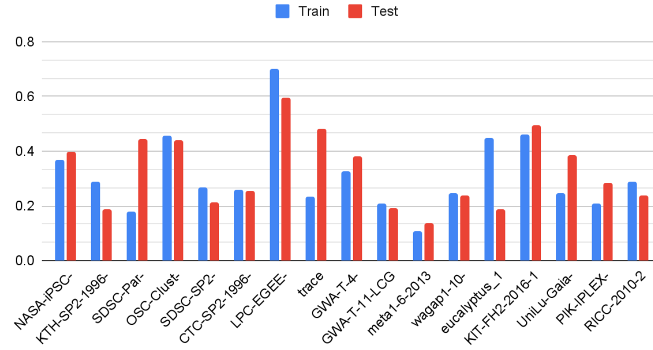


Figure 4: Mean sparsity of batches for all workloads for all VM capacities

The L2 norm of the duration distribution is a way of measuring the ‘uniformity’ of the duration distribution of a batch. If the L2 norm is applied on a vector that represents a distribution of a random variable X the result represents the ‘uniformity’ of the distribution. Studying the extreme cases, assuming the vector V_{uniform} = [0.33, 0.33, 0.33] the scaled L2 norm of V_{uniform} will be almost 0, while the scaled L2 norm for $V_{\text{one-hot}}$ = [1,0,0] which is a one-hot vector will be 1.

To calculate the L2 norm first we must calculate the distribution of durations in the batch. Initially, the upper and lower bounds of durations in the batch are calculated. Using these bounds, M bins are calculated and the jobs are assigned to the bins whose value is the closest to its duration. The vector of size M containing the sums of each bin divided by the total number of jobs represents the distribution of durations. The following formula is used to calculate the L2 norm

$$L2 = \sqrt{\sum_{i=1}^{M} \left|x_i\right|^2}$$

Where x is summing over the distribution vector for the current batch. This value is initially bounded by [1/sqrt(M), 1]. To scale these results in the range of (0,1] the following is used

$$sL2 = \frac{L2 * \sqrt{M} - 1}{\sqrt{M} - 1}$$

This, in combination with sparsity, gives the agent a wider picture of the workload. This again is a continuous variable in the range of (0,1] and converted into a discrete variable using the same method as sparsity, by rounding down to 1 digit precision and multiplying by 10. Figure 5 shows the mean L2 of batches for every workload.

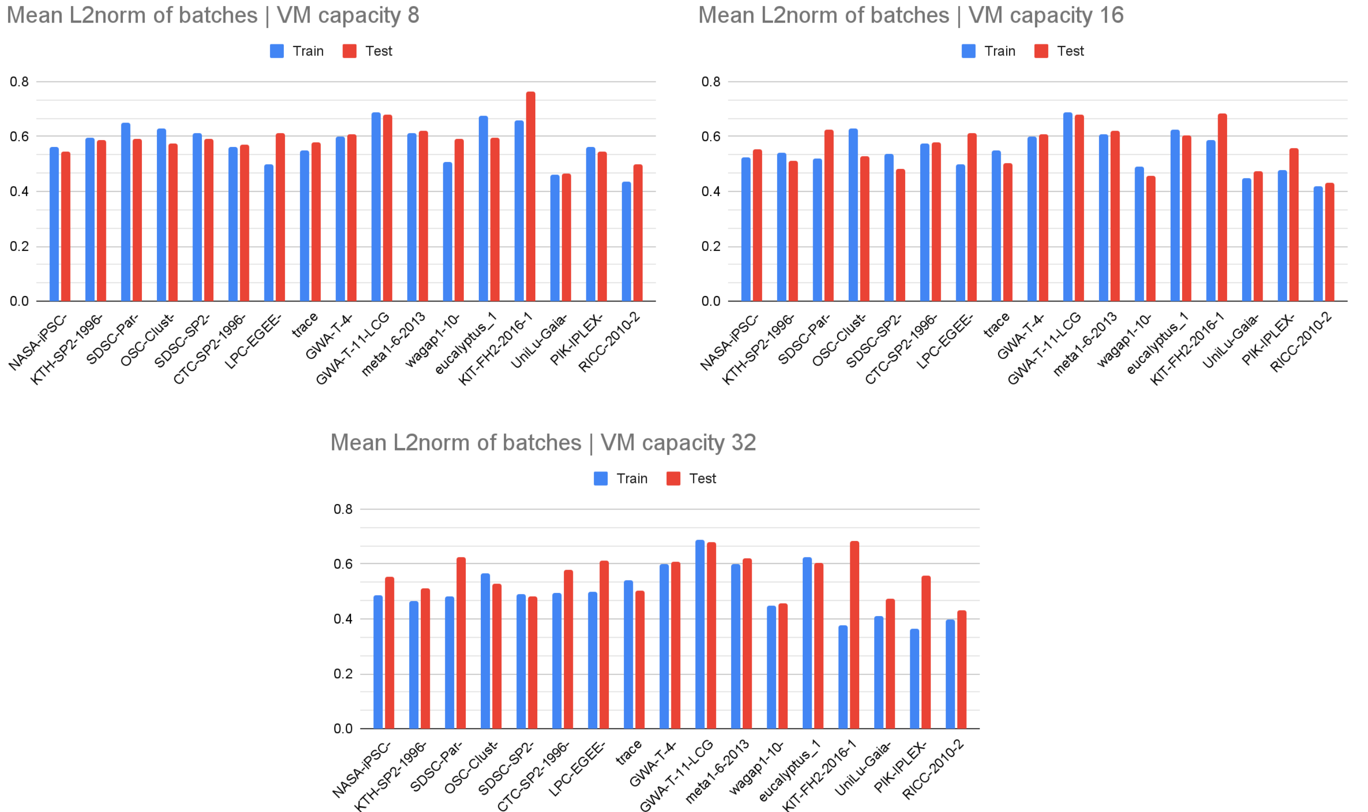
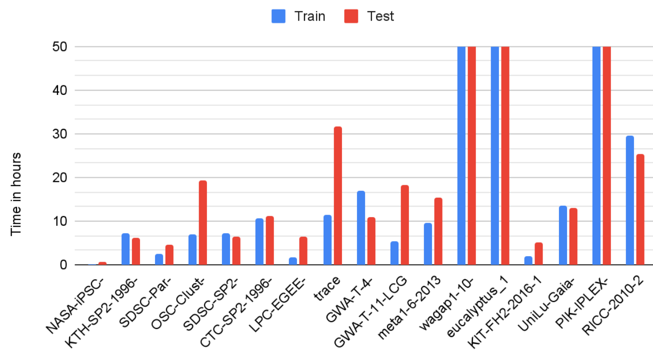
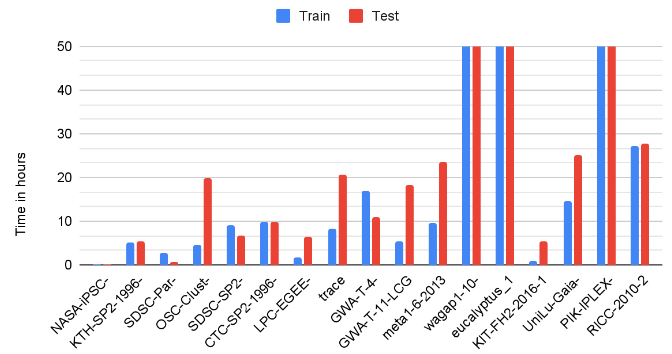


Figure 5: Mean L2 norm of batches for all workloads for all VM capacities.

The third state defining metric is active bins, which is simply the number of active bins at the time of the arrival of the first job in B_n. This variable is by its nature discrete, only taking integer values and its range is from [0,10] with values more than 10 being rounded down to 10.

The final metric used is overlap, which measures whether the previous batch B_{n-1} overlaps with the next batch B_n calaculated using the latest departing job of batch B_{n-1} and the first job of B_n . This metric is binary and hence discrete. Figure 6 shows the mean overlap of batches for every workload.





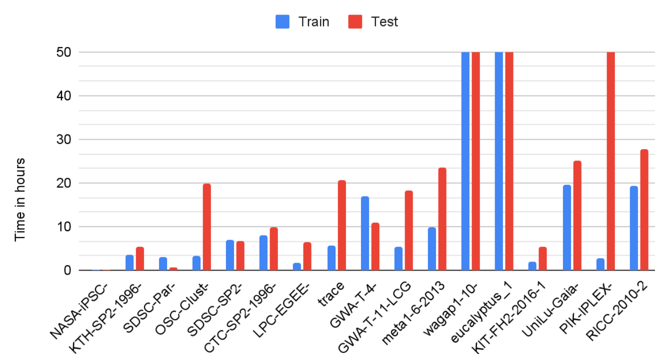


Figure 6: Mean overlap of batches for all workloads for all VM capacities

(Section 4.5 Deep Q-Learning)

(subsection 4.5.a Introduction to deep Q-learning)

Deep Q-learning[18] is a method similar to Q-learning with the main difference being the replacement of the Q-table with a Neural Network. Ignoring the training for now and considering a trained agent this variation not only allows for continuous variables to be used as state metrics but also for an agent able to generalize better and make more accurate decisions for unseen data due to the strong generalization power of neural networks. Of course, training such an agent requires higher computational capabilities due to the complexity of the neural network and inference requires higher computational power. But in cases where the use of continuous and/or a large number of state metrics are required to describe the environment or in cases where exploring a large number of states is not possible, the traditional Q-learning algorithm does not work but the Deep Q-learning algorithm is a viable option. Additionally, for an algorithm aimed at the cloud, an increase in the required computational power of this scale is marginal.

(subsection 4.5.b Training the Q network)

Two Neural Networks take the place of the Q-table - the policy network and the target network. The target network is used by the agent to infer actions by using the current state as input and the policy network is trained on the data that the agent produces by initially randomly choosing actions and receiving rewards. The target networks' weights are periodically updated to match the policy network; this approach of using two networks is used since it provides a more stable training.

The input of these networks is an N-dimensional vector V_{in} where N is the number of metrics describing the states and whose values are the state metrics for the current state S_n. The output of the network is the M-dimensional vector V_{out} where M is the action space. With the V_{out} calculated, the optimal action is $A[argmax(V_{out})]$ where A is the action values vector.

To train the policy network, a replay memory technique is utilized. Initially, the un-trained target network makes random decisions and is rewarded accordingly by the environment. With the values state, action, next state, and reward, a training dataset can be compiled. The training set consists of a state vector S_t which is the input and a vector V_{target} of which is the target output of the network for the state action pair (S_t, A_t). If the output of the target net when inferring the action for state S_t is V_{out}, V_{target} consists of the updated Q-value for action A_t, calculated using the reward the agent received in the TD update rule. Formally

$$V_{target}[A_t] = R_t + G * \max(TargetNet(S_{t+1})) - V_{out}[A_t]$$

Where R_t is the reward, G is the discount factor, and $\max(TargetNet(S_{t+1}))$ is the max future Q-value obtained by running the next state through the target network. One can easily see the similarities between the update rule of regular Q-learning and Deep Q-learning. With V_{target} calculated, the pair (S_t, V_{target}) can be pushed into replay memory and used for training. In Fig 4. the architecture of a Deep Q-learning agent is also shown visually.

Training the Policy Network is just like training any other Neural Network. A loss function is used to calculate the output error which is then propagated backward and using an optimizer function in combination with the gradients calculated on the forward pass by the auto-grad algorithm of PyTorch[21] the new weights and biases are updated to reduce the output error. Of course, the auto-grad algorithm is turned off for inference to reduce computational complexity. In further detail, the loss function used is Mean Squared Error which simply calculates the average of the squared difference of each element of V_{target} and V_{out} , the optimizer is RMSProp [22] which utilizes techniques such as momentum to help escape local minima (in non-convex loss functions) and adjustable learning rate to promote faster convergence. The architecture of the networks consists of the input layer of size equal to the number of states, 3 hidden layers of size 1024 with the ReLU[16] activation function and an output layer that outputs the Q-values for each

possible action. The state metrics, the reward function, and the other common parameters between the two algorithms not mentioned here stay the same.

(Section 4.6 Training the agents)

Training these algorithms is computationally expensive, especially for the Deep Q-learning algorithm. Additionally the problem is by nature sequential prohibiting any parallelization of the training process. An ε-greedy (epsilon-greedy) training algorithm was used for the agents. The ε-greedy algorithm is used to determine how an agent will act. Each time the agent is given a new state and tasked with choosing an action from a list of possible actions, the ε-greedy algorithm determines whether that action will be an exploration or an exploitation action based on a probability ε. Exploitation is defined as picking the action with the largest q-value and has a probability of 1- ε while exploration is defined as choosing a random action hence exploring the environment and has a probability of ε. Considering the extreme cases, an agent with an epsilon of 0, so an agent that never explores, will only choose the current best action, possibly getting stuck at a local minimum. On the other hand, it is easy to see how an agent with an epsilon value of 1 will fail to converge.

The agents are trained in an episodic manner. For every episode, the agent is called to pack the same part of a workload. After the training is done, the agent's performance is evaluated on the next part of the workload which has not been used for training.

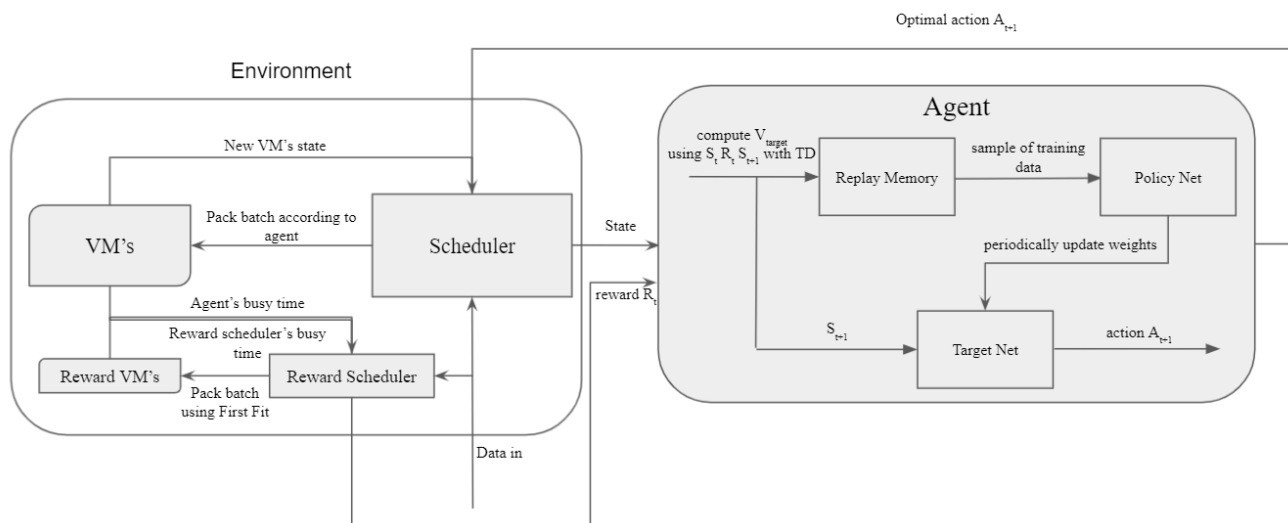


Figure 7: Overview of the architecture of a Deep q-learning agent.

CHAPTER 5 Experimental Results

(Section 5.1 Simulator)

The scheduler is the core part of the model. It is the scheduler that is tasked with packing the jobs, keeping track of machines (creating new machines, deleting empty machines), and the busy time. The model was built using an object-oriented approach. The machine, the scheduler, and the jobs are all modeled as classes.

The Job class contains information about the start time, end time, duration, demand, and the timestamp of the job's arrival and departure. The job class also includes many properties that mainly manipulate the timestamps to extract information about the day and time of arrival and completion.

The Machine class has a few main components, most notably a list of all active jobs in the machine, a list of all active intervals, and the Interval Maker. The Machine has information about its start time to keep track of its busy time. The most notable methods of the Machine class are the update method and the add method. The update method is called by the scheduler for all machines every time a new job arrives. The update methods can be thought of intuitively as the way for the model to move forward in time. It makes all the necessary checks and updates currently active jobs, adds new ones along with their respective time intervals in the lists, and keeps track of whether the machine is empty and should be shut down. The add method works using the interval maker class.

The interval maker's sole purpose is the creation of intervals. When the scheduler assigns a new job to a machine, the machine then proceeds to call the add methods of interval maker. The add method first removes past intervals and then in one pass (O(n) time complexity, where n is the number of active intervals) finds the suitable start point and endpoint for the interval, creates the interval, and updates the requirements for all intermediate intervals.

Finally, the scheduler class handles everything related to the packing of jobs and keeps track of the state of the system. It keeps a list of all active machines and its methods are the bin packing algorithms and the run methods which take a workload, part of a workload, or single job and simulate it running on the cloud server.

(Section 5.2 Results)

To validate the performance of the agent, real-world datasets from [13], [14], and [15] were used. The datasets consist of instances of jobs each with information about arrival time, duration, and demand in CPU cores which are parsed, discarding any job that has a demand larger than or equal to the max capacity of the VM's. As a consequence, when comparing the results of the same sets with different capacities, variations between the same sets are expected. Furthermore, the data are split into D single-core jobs where D

is the demand of the original job. The agents were trained for 3 different VM capacities: 8 cores, 16 cores, and 32 cores.

To compare the algorithms the normalised busy time is used. To normalise the busy time, the workloads were packed using the First Fit algorithm as well as the TBC with static threshold and TBC adjusted with Q-Learning and Deep Q-Learning + FF. For each algorithm the percent difference over the First Fit algorithm was calculated and the normalised busy time was calculated using the following formula $NormBT = 100 + PD$ where PD is the percent busy time difference of the algorithm as compared to First Fit. The agents were trained on 5 thousand jobs and tested on the next 5 thousand jobs for each workload. The agents both used a Gamma value of 0.8, an epsilon of 0.05 (5% percent change to explore), and a learning rate of 0.01. The Deep Q-Learning agents' replay memory has a capacity of 10.000 and is cleared upon filling up.

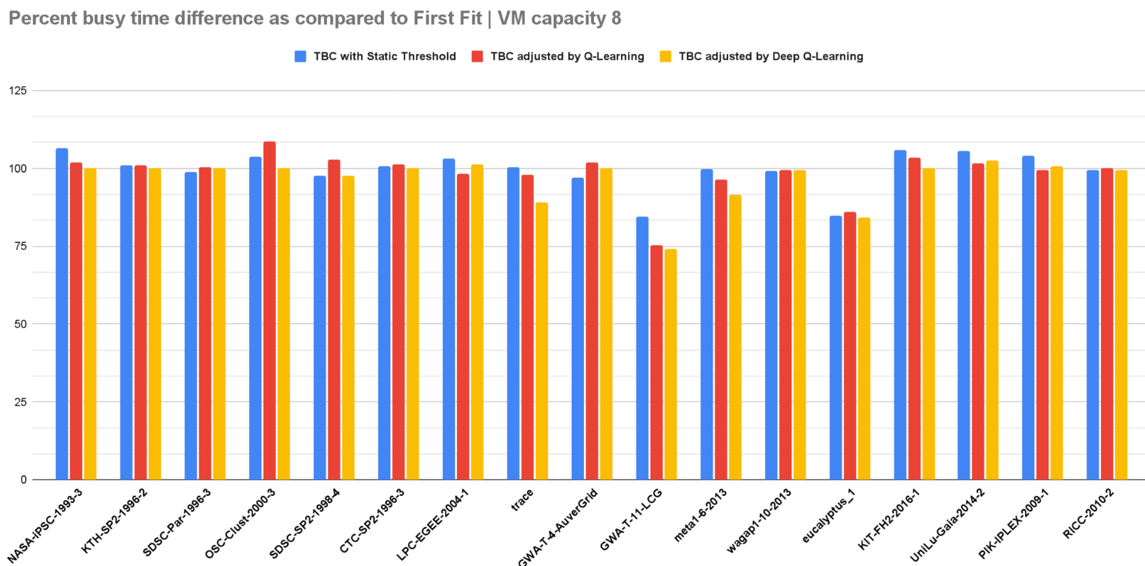


Figure 8: Performance of TBC with static threshold, Q-Learning and Deep Q-Learning adjusting threshold as compared to First Fit for 17 workloads with VM core capacity 8. Lower percentage means lower busy time and thus better performance.

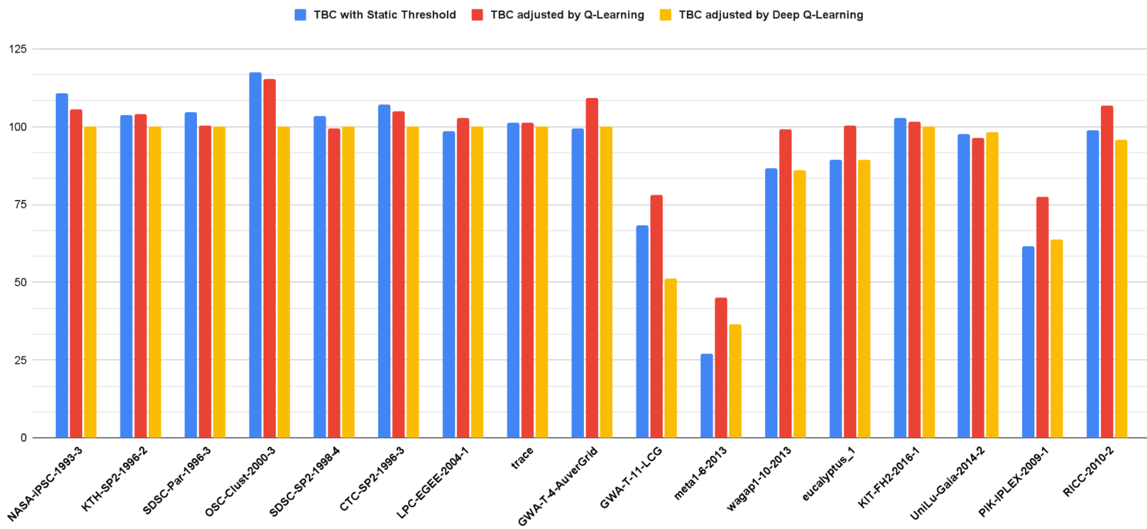


Figure 9: Performance of TBC with static threshold, Q-Learning and Deep Q-Learning adjusting threshold as compared to First Fit for 17 workloads with VM core capacity 16. Lower percentage means lower busy time and thus better performance.

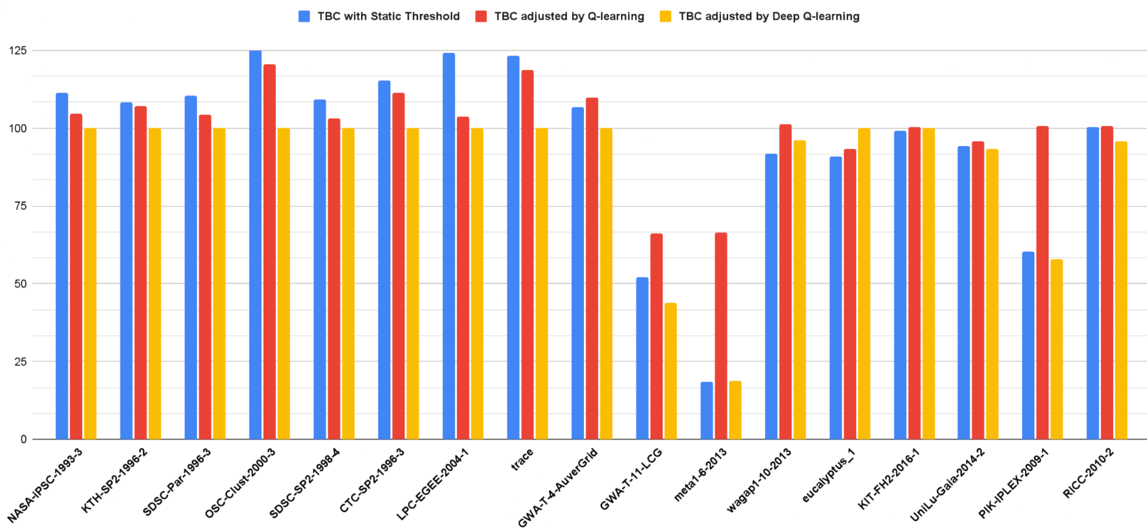


Figure 10: Performance of TBC with static threshold, Q-Learning and Deep Q-Learning adjusting threshold as compared to First Fit for 17 workloads with VM core capacity 32. Lower percentage means lower busy time and thus better performance.

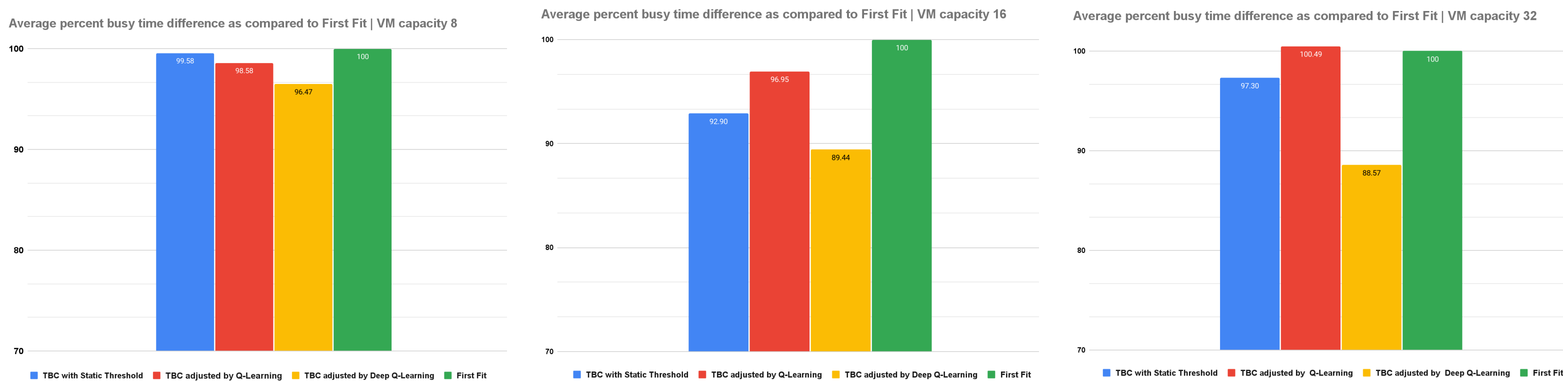


Figure 11: Average percent difference in busy time as compared to First Fit for all the datasets for 8, 16, and 32 cores capacity.

As shown in the above results the Deep Q-Learning agent managed to consistently achieve a better performance than both the TBC algorithm with a static threshold and the Q-Learning agent. Furthermore, when compared to First Fit, the Deep Q-Learning managed to match the performance of First Fit in cases where First Fit seems to be the optimal approach and outperform it when a more optimized packing was possible. In such cases where the optimal strategy seems to be First Fit, the static threshold method by nature will not be able to achieve better results. Focusing on the Q-Learning agent, we can see that while on average it outperformed the static algorithm for capacities of 16, and 32 cores, it failed to match the performance of the First Fit algorithm in the worst case, specifically sparse workloads. The main disadvantages of the Q-learning algorithm and the reason for its poor performance as compared to the Deep Q-learning algorithm is its lack of generalisation. The deep Q-learning algorithm uses a deep neural network which is designed for generalisation, this in turn allows the Deep Q-learning agent to make informed decisions on unseen states based on similar cases seen in the training. On the other hand, the Q-learning algorithm when faced with an unexplored state will choose a random action. This is especially obvious in the cases where First Fit seems to be the optimal packing in which the generalised Deep Q-learning agent will have no trouble selecting First Fit in an unseed case but the Q-learning will choose a random action resulting in inferior performance.

CHAPTER 6 Conclusion

In this work, two reinforcement learning based algorithms for the online interval scheduling problem were proposed. These algorithms are based on the model free Q-learning and Deep Q-learning reinforcement learning agents which excel in maximising long-term reward tasks due to the temporal difference update rule. These agents work in combination with the TBC algorithm which itself is a preprocessing step to any traditional bin packing algorithm (in this problem First Fit was used). The goal of the agent is to minimize the total busy time of the system when processing a workload and thus the total power consumption. The agents were trained and evaluated using real-world data and the Deep Q-Learning agent, due to its great generalisation capabilities, outperformed both the First Algorithm and the TBC algorithm with a static threshold. In the future, algorithms capable of predicting future workloads, similar to work proposed in [13] but also in the form of an LSTM recurrent neural network [23], can be implemented in order to provide the agent with extra information resulting in a more optimized and faster adapting agent.

REFERENCES

[1] M. Flammini, G. Monaco, G. L. Moscardelli, H. Shachnai, M. Shalom, T. Tamir, and S. Zaks. Minimizing total busy time in parallel scheduling with application to optical networks. Handbooks in operations research and management science, 411(40-42):3553–3562, 2010

[2] G. B. Mertzios, M. Shalom, A. Voloshin, P. W. Wong, and S. Zaks. Minimizing total busy time in parallel scheduling with application to optical networks. Theoretical Computer Science, 562:524–541, 2015

[3] W. Tian, Q. Xiong, and J. Cao. An online parallel scheduling method with application to energy-efficiency in cloud computing. The Journal of Supercomputing, 66:1773–1790, 2013.

[4] Li Y, Tang X, Cai W. Dynamic bin packing for on-demand cloud resource allocation. IEEE Transactions on Parallel and Distributed Systems. 2015 Jan 19;27(1):157-70.

[5] Ren R, Tang X. Online flexible job scheduling for minimum span. InProceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures 2017 Jul 24 (pp. 55-66).

[6] Azar Y, Vainstein D. Tight bounds for clairvoyant dynamic bin packing. ACM Transactions on Parallel Computing (TOPC). 2019 Oct 15;6(3):1-21.

[7] Shalom M, Voloshin A, Wong PW, Yung FC, Zaks S. Online optimization of busy time on parallel machines. Theoretical Computer Science. 2014 Dec 4;560:190-206.

[8] In private communication with Nikos Tziritas and Panos Oikonomou

[9] Wang D, Ren C, Govindan S, Sivasubramaniam A, Urgaonkar B, Kansal A, Vaid K. ACE: abstracting, characterizing and exploiting peaks and valleys in datacenter power consumption. ACM SIGMETRICS Performance Evaluation Review. 2013 Jun 17;41(1):333-4.

[10] M. Dayarathna, Y. Wen and R. Fan, "Data Center Energy Consumption Modeling: A Survey," in IEEE Communications Surveys & Tutorials, vol. 18, no. 1, pp. 732-794, Firstquarter 2016, doi: 10.1109/COMST.2015.2481183.

[11] X. Tang, Y. Li, R. Ren, and W. Cai. On first fit bin packing for online cloud server allocation. In IEEE International Parallel and Distributed Processing Symposium (IPDPS), pages 323–332, 2016

[12] Sutton RS, Barto AG. Reinforcement learning: An introduction. MIT press; 2018.

[13] Yao Lu, John Panneerselvam, Lu Liu, Yan Wu, "RVLBPNN: A Workload Forecasting Model for Smart Cloud Computing", Scientific Programming, vol. 2016, Article ID 5635673, 9 pages, 2016.

[14] D. G. Feitelsoni, D. Tsafrir, and D. Krakov. Experience with using the parallel workloads archive. Journal of Parallel and Distributed Computing, 74(10):2967– 2982, 2014

[15] D. Klusacek and V. Chlumsky. Evaluating the impact of soft walltimes on job scheduling performance. Workshop on Job Scheduling Strategies for Parallel Processing, Springer, pages 15–38, 2018.

[16] Li Y, Yuan Y. Convergence analysis of two-layer neural networks with relu activation. arXiv preprint arXiv:1705.09886. 2017 May 28.

[17] Watkins, C.J.C.H., Dayan, P. Q-learning. Mach Learn 8, 279–292 (1992).

[18] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602.

[19] Reuben M. Reducing the Thrashing Effect Using Bin Packing. Bar Ilan University. Department of Mathematics and Computer Science.; 2004Eliiyi U, ELIIYI DT.

[20] Applications of bin packing models through the supply chain. International Journal of Business and Management Studies. 2009 Jan;1(1):11-9.

[21] Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L. and Lerer, A., 2017. Automatic differentiation in pytorch.

[22] Kurbiel, T. and Khaleghian, S., 2017. Training of deep neural networks based on distance measures using RMSProp. arXiv preprint arXiv:1708.01911.

[23] Gers, F.A., Eck, D. and Schmidhuber, J., 2002. Applying LSTM to time series predictable through time-window approaches. In Neural Nets WIRN Vietri-01 (pp. 193-200). Springer, London.