Using Reinforcement Learning to Size Tasks for Scientific Workflows

Nicolas Zunker

n.zunker@campus.tu-berlin.de

December 13, 2021

BACHELOR'S THESIS

Distributed and Operating Systems Chair

Technische Universität Berlin

Examiner 1: Prof. Dr. Odej Kao

Examiner 2: Prof. Dr. Volker Markl

Advisor: Jonathan Bader

Declaration

I hereby declare that I have created the present work independently and by my own without illicit assistance and only utilizing the listed sources and tools.

Hiermit erklaare ich, dass ich die vorliegende Arbeit selbststaandig und eigenhaandig sowie ohne unerlaubte fremde Hilfe und ausschliesslich unter Verwendung der aufgefaehrten Quellen und Hilfsmittel angefertigt habe.

Die selbständige und eigenständige Anfertigung versichert an Eides statt:

Berlin, den 13. Dezember 2021

Nicolas Zunker

Acknowledgment

I would like to thank my advisor Jonathan for his helpful advice, constructive feedback and assistance, Prof. Dr. Toussaint, Prof. Dr. habil. Kao and Prof. Dr. Zubow whose courses I enjoyed immensely and from whom I learned a lot, and my friends and family for their support.

NZ

Abstract

The growth of computational power and the increasing importance of digital data in scientific research has led to greater demand for computational resources by users who aim to process large datasets. Particularly in the natural sciences it has become common for scientists to break down their computing needs into a sequence of smaller tasks, a so called workflow. These workflows can then be run on a variety of different execution platforms, depending on the users needs.

Since workflows are composed of segregated inter-dependent tasks which can run independently of each other, the individual tasks which make up a workflow can be assigned a fraction of the computational resources available to the entire execution platform, and doing so intelligently could improve efficiency and performance.

This thesis aims to investigate how reinforcement learning can be applied to the allocation of resources to individual tasks in order to choose more efficient allocations. A reinforcement learning solution will be integrated into the source code of a popular scientific workflow management system and tested with several common bioinformatic workflows. Two different approaches (Gradient Bandits and Q-learning) will be used and their performance will be compared with both that of the default resource allocations and that of an approach based on [22, 25] which uses a feedback loop.

Ultimately the approaches presented in this thesis outperform the workflow's default configurations with regards to both memory and CPU efficiency: the q-learning approach assigned 7% less CPU hours, 31% less memory and was 7% faster, whereas the gradient bandit approach assigned 42% less CPU hours, 80% less memory and was only 4% slower. The feedback loop approach assigned 13% less CPU hours, 87% less memory and was 7% faster and thus performed worse than the gradient bandits with regards to CPU efficiency but better in terms of memory and speed.

Zusammenfassung

Die Relevanz von digitalen Daten und die damit verbundene Analyse von grossen Datensätzen in der wissenschaftlichen Forschung wächst kontinuierlich, und der Nutzerbedarf an Rechenressourcen steigt mit. Besonders in den Naturwissenschaften ist es üblich, dass Wissenschaftler die Analyse dieser Daten in kleinere Arbeitspakete aufteilen, so genannten Tasks, und die gesamte Sequenz dieser Tasks wird als Workflow bezeichnet. Die Workflows bestehen aus verschiedenen, eigenständigen Tasks die unabhängig voneinander ausgeführt werden können, daher muss ihnen nur ein Bruchteil der Rechenressourcen des Plattforms auf dem sie ausgeführt werden zugewiesen werden. Durch intelligente Allokationen ermöglicht dies eine größere Effizienz und Leistung der Workflows.

Die Forschungsfrage dieser Arbeit beinhaltet wie und mit welchem Effekt Reinforcement Learning für die Allokation von Rechenressourcen zu den Tasks eines Workflows angewandt werden kann. Hierzu werden zwei verschiedene Ansätze (Q-learning und Gradient Bandits) in den Quellcode einer Workflow Verwaltungssoftware integriert und mit fünf Bioinformatik Workflows getestet. Diese Ergebnisse werden verglichen mit der Leistung der Workflows unter normalen Konfigurationen, und einem Ansatz mit Hilfe von Rückkopplungsschleifen, der auf [22, 25] basiert.

Die in dieser Arbeit entwickelten Ansätze zeigten eine bessere Leistung als die Workflows mit voreingestellten Allokationen, sowohl im Bezug auf die Arbeitsspeichernutzung, als auch auf die CPU Nutzung. Der q-learning Ansatz allozierte 7% weniger CPU Stunden, 31% weniger Arbeitsspeicher und die Rechenleistung war 7% schneller. Der gradient bandit Ansatz allozierte 42% weniger CPU Stunden, 80% weniger Arbeitsspeicher und die Rechenleistung war nur 4% langsamer. Letztlich hat der Ansatz mit der Rückkopplungsschleife 13% weniger CPU Stunden und 87% weniger Arbeitsspeicher alloziert und war 7% schneller, und damit weniger effizient mit seiner CPU Allokationen als die gradient bandits aber besser in Bezug auf Arbeitsspeicher und Geschwindigkeit.

Table of Contents

Declaration I

Acknowledgment III

Abstract V

Zusammenfassung V

List of Figures X

1 Introduction **1**

1.1 Motivation and Problem Description 1

1.2 Goal of the Thesis 3

1.3 Structure of the Thesis 3

2 Background **5**

2.1 Scientific Workflows 5

2.2 Scientific Workflow Management Systems 6

2.3 Reinforcement Learning 9

2.3.1 Actions, States, Policies and Rewards 10

2.3.2 Gradient Bandits 11

2.3.3 Q-learning 14

3 Related Work **17**

3.1 Using Reinforcement Learning to Allocate Resources 17

3.2 Resource Allocations in Scientific Workflows 20

3.3 Points of Interest from these Works 22

4 Approach **23**

4.1 Integrating a Solution into a Workflow Manager 23

4.2 Gradient Bandits 25

4.2.1 CPU Bandit 25

4.2.2 Memory Bandit 29

4.3 Q-learning Agent 31

5 Evaluation **35**

5.1 Testing the Agents 35

5.2 Results and Discussion 37

5.2.1 Performance Comparison 37

5.2.2 Performance on a Workflow by Workflow Basis 43

5.2.3 Summary 44

6 Conclusion and Outlook **45**

6.1 Improvements 46

6.2 Implications and Further Areas of Research 47

A Example of a Task's Q-learning Agent **51**

Bibliography **53**

List of Figures

2.1 nf-core/shootstrap Workflow Represented as a Digraph 7

2.2 Agent Environment Interactions [20] 10

4.1 High Level Design: Integration of a Reinforcement-Learning Agent into Nextflow 24

4.2 Example of Two Bandits Converging Too Fast 28

4.3 Bandits with a Step Size Based on Their Historical Average Execution Time 29

5.1 Performance of the CPU and Memory Approaches 38

5.2 Memory Usage 39

5.3 CPU and Execution Time 42

5.4 Memory 42

A.1 Example of a Q-learning Agent 52

Chapter 1

Introduction

1.1 Motivation and Problem Description

The rising popularity of scientific workflows and the enormity of their constituent task's aggregate resource requirements [13] presents an opportunity to try and fine tune the resources allocated to these tasks to achieve better performance.

Since a scientific workflow is composed of many tasks which each have unique relationships between the resources they require, their input, and how it affects their performance, it becomes cumbersome or even futile to try and hand-pick the most efficient resource allocation for each individual task. In addition to this it is also quite common that the users may not have the requisite knowledge to properly size the tasks themselves and may make poor estimations [17] of what a given task's resource requirements are. When sizing tasks one also needs to find a balance between resource efficiency and improved execution time. Assigning a task more resources will usually make it execute faster but may increase costs. Additionally in an environment with limited resources assigning a task too many resources hampers other tasks and decreases the resources available to them. Finally it must also be considered that for a given use case the exact topology of the deployment scenario (i.e. whether it is being run on an individual computer, in a grid or cluster, or on a cloud computing platform) and the hardware infrastructure being used will also influence the performance of the workflow and its constituent tasks. When one considers all of these factors it becomes clear that fine tuning the resources for each of the tasks within such workflows "by hand" is difficult and complex.

However it is also obvious that sizing tasks properly is of critical importance. Specifically for large scale, distributed computations the resources being requested are so large that even small improvements in efficiency represent a large amount of reclaimed resources which could be allocated to other users or could reduce costs (in a scenario where computational resources are being paid for by the workflow user).

Thus this presents a scenario in which although better task sizing is beneficial for everyone it is not a problem with a simple solution.

At this point it becomes quite reasonable to begin to consider whether machine learning could provide a solution. There are many different methods which could be tried, however reinforcement learning presents two distinct advantages. Firstly it does not need to be trained with data from good allocations in the past because reinforcement learning agents 'learn on the go' by trying to discover the optimal policy for their goal through interaction with their environment. This is important because the problem of verifying a given resource allocation is optimal is exactly as hard as finding an optimal allocation, so gathering training data with examples of optimal allocations is ruled out. Secondly, reinforcement learning is adaptable. Because it only ever aims to learn an optimal policy though interactions, a reinforcement learning agent is constantly gathering feedback on its actions (even after it has reached a point where it could be considered 'optimal') and constantly trying to refine its approach to be as perfect as possible. Therefore if the environment or the problem changes and the old policy is no longer the best one, the agent will notice this and adjust its approach and learn a new policy to achieve its goal. Within the context of scientific workflows this is desirable because the tasks and the profile of the input data may change (over time or at once) and render the previous allocations of resources for the given task completely wrong. Indeed the entire system to which the workflow is deployed could change, for example migrating to new infrastructure or a different cloud computing provider. In all of these scenarios the way that resources are allocated to tasks would probably need to be adjusted and reinforcement learning is an approach which would be able to handle these changes and adapt to them.

Ultimately reinforcement learning presents an advantage for task sizing because it can handle the complex job of exploring different resource allocations and attempting to balance resource efficiency against faster execution. This is something which approaches such as linear regression or picking the average historical usage may struggle to do. The great advantage of reinforcement learning should be its ability to explore different allocations on its own and leverage them to increase performance.

1.2 Goal of the Thesis

The goal of this thesis is to use reinforcement learning to improve the efficiency and performance of scientific workflows by more accurately assigning resources to the individual tasks within the workflows. Specifically the CPU and memory usage will be examined using two different approaches: Gradient Bandits and Q-learning, and compared to the performance of the task's default configuration and a feedback loop approach which combines different aspects of the methods used in [22, 25].

1.3 Structure of the Thesis

The remainder of the thesis is structured as follows: in chapter 2 an overview and an explanation of the ideas and technologies relevant to this paper is given. Then in related work (chapter 3) some papers are discussed which addressed similar topics and served as influences for this thesis. After that the implementation of a solution to the problem is described in the approach (chapter 4) and finally in chapter 5 the results are presented and analysed. The thesis concludes in chapter 6 with a brief summary, a review of potential improvements and a look towards future research.

Chapter 2

Background

This chapter provides background information on scientific workflows and reinforcement learning. It explains their functionality to the extent to which they are related to this thesis. To begin with, in section 2.1 scientific workflows are discussed, then in section 2.2 the systems that manage them are introduced, and finally in 2.3 the reinforcement learning approaches relevant for this thesis are explained.

2.1 Scientific Workflows

Over the last two decades computation has emerged to become an integral part of scientific research [7],[4] and with the increased use of simulations and digital sensors the importance of digital data continues to rise[14]. This has led to unprecedented computing power requirements in the sciences. Indeed much of the effort of scientists is now invested in the analysing and processing of the data rather than the gathering of the data, and software costs have come to dominate capital expenditures for many large-scale experiments [1]. Some of the scientific fields

[1]Jim Gray. Jim Gray on eScience: A Transformed Scientific Method. 2007. url: http://languagelog.ldc.upenn.edu/myl/JimGrayOnE-Science.pdf.

which have particularly large computational needs are Biology, Astronomy, and Seismology [14].

To handle so much data it is usually processed in a sequence of small steps or tasks, often using command-line tools [25]. Since these tasks may work on the same data or a version of that data which has been processed by another task, there are of course temporal and logical dependencies between the tasks. Managing these complex interdependent structures can be quite difficult and has given rise to the concept of workflows, which are meant help model the constituent tasks and the sophisticated structure of their dependencies. The order of execution of the tasks and whether or not they can be run sequentially or concurrently is determined by these inter-task dependencies, [4],[11].

Workflows can be modelled in many different ways, ranging from simple scripting languages to graphs and mathematical models [19]. At their core, workflows consist of four simple components: the inputs, the tasks, the dependencies between tasks, and the outputs. This can easily be modelled as a graph with vertices to represent tasks and edges to represent dependencies. An example of this can be seen in 2.1 from the nextflow documentation [2].

2.2 Scientific Workflow Management Systems

The emergence of scientific workflows as a method for representing and managing these complex computations has lead to the emergence of scientific workflow mangement systems (SWMS, or scientific workflow managers) to manage the execution of scientific workflows and their constituent tasks. Today there are a litany of different SWMS's, such as Nextflow [8], Kepler [1], Pegasus [6] and others.

Because the resource requirements of scientific workflows can be immense [13], workflow managers must be able to interface with powerful execution platforms which usually use

[2]https://www.nextflow.io/docs/

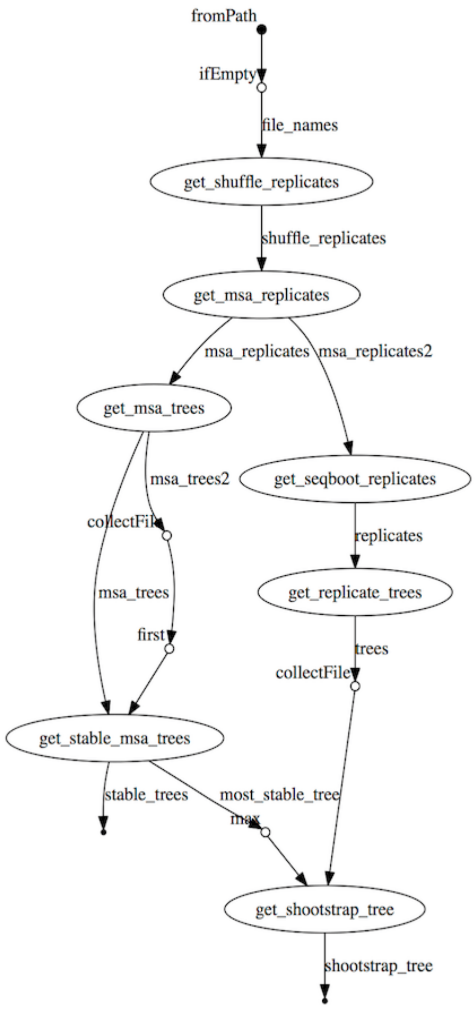


Figure 2.1: nf-core/shootstrap Workflow Represented as a Digraph

resource managers such as Kubernetes [3] or YARN [23]. A workflow manager will schedule a workflow's tasks with the resource manager (this will include assigning the task a fixed amount of resources), handle failures as configured by the user, manage the communication between tasks, and collect and store the results if the workflow's execution succeeds.

At this point it is important to clarify the distinct responsibilities of a workflow management system, an execution platform and a scientific workflow, with regards to the tasks. The user or creator of the workflow decides which tasks need to be executed and when they need to be executed relative to each other and the SWMS will schedule tasks with the execution platform according to this design. It is the execution platform which will ultimately decide when a task will actually be executed on physical hardware. In this transaction it is the SWMS which is acting as the middle man. Beyond which tasks to execute, and in which order, the user will also decide what the inputs to a task should be and the workflow manager will ensure that before the task is executed those inputs are available and that they are passed to the task.

Finally, with regards to resources, the user does not need to specify what the resources assigned to a task should be, the user may do so if they like, however it is not a requirement, and in the event that they do not assign any resources the SWMS will pick an assignment. The assignment of resources to a task is a request by the workflow management system to the execution platform, and, provided the platform can fulfil these requirements, the task should be executed in an environment where it has access to as many resources as it was assigned but not more. Tasks which exceed their resource requirements will often be aborted and this will be communicated to the resource manager which will then have to decided if it should try to run the task again, continue without the task, or abort the execution of the entire workflow- what action to take in this event can also be configured by the user.

2.3 Reinforcement Learning

Popularized in the book by Sutton and Barto [20], reinforcement learning can broadly be said to present a framework for a learning system or “agent” to find the optimal policy for achieving a given goal in an uncertain environment by interacting with the problem and the environment. Most importantly it is also able to adjust this policy “on the go”, meaning it can both learn a new policy if the challenge or the environment changes, and that it can be deployed without any training and it will improve as it gains experience. In this context the agent’s goal is always set by a reward function. Using reinforcement learning, the agent learns to maximize this function and thus, hopefully, achieve the desire of its designers. To put it simply, reinforcement learning is a “computational approach to learning from interactions” [20]

Central to all reinforcement learning is the idea of exploration versus exploitation. Since the agent initially knows nothing about its environment it must attempt to learn through exploration. By trying different things and receiving different rewards the agent can construct a policy that always makes the optimal choice. But in order to know what a good or an optimal choice is the learning system must also occasionally make the wrong choice so that it can learn not to make it again. Trying different things is ”exploration” and using the knowledge gained from this to make the right choice is“exploitation”. An agent cannot simultaneously explore and exploit. This dichotomy is at the core of reinforcement learning. The agent must always make the choice between exploring more to potentially discover an even better policy and eventually yield even better rewards in the future, or using its current policy to increase its immediate rewards.

Beyond the dichotomy mentioned above, the other challenge presented to the designers of a reinforcement learning agent is the question of how to translate the overarching goals into a singular reward. A reinforcement learning agent run ad infinitum should converge to an optimal policy which maximises reward, and thus the onus is on the designer to pick a reward function which, when maximised, will help them to achieve their goals.

2.3.1 Actions, States, Policies and Rewards

For the purposes of this thesis only q-learning and gradient bandits will be discussed. In order to understand these approaches the concepts of states and actions must be introduced. An agent interacts with its environment by performing actions. These actions may have a tangible effect on the agent and the environment and the agent receives feedback about its actions. This feedback is given to the agent in the form of rewards. The agent also maintains an internal representation of itself (either independently or in relation to the environment), this representation is considered the agent's state. An agent may change states either as a result of its own actions or as a result of changes in the external environment. A simple visual representation of these interactions can be seen in 2.2.

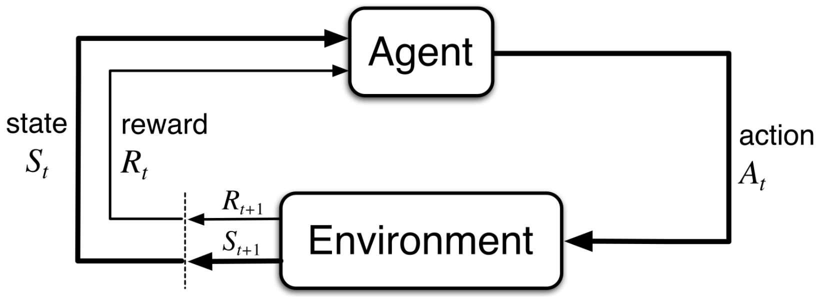


Figure 2.2: Agent Environment Interactions [20]

Policies and the question of how best to evaluate them are crucial to reinforcement learning. Within the context of states, actions and rewards, a policy is simply an instruction for which actions must be taken in which states to maximise the cumulative reward over time. Which action to take may depend on the value of the current state or the value of the state which the action may cause the agent to transition to. In order to achieve its goal an agent needs to develop a policy that maximizes its reward, then as the agent encounters new situations it simply follows the policy it has learned.

To evaluate a given policy it must be compared to the optimal policy. For any given problem and its reward function there exists at least one policy which maximizes the reward received and is considered optimal. At its core, reinforcement learning aims to enable the agent to continuously refine its current policy so that it approaches the optimal policy.

2.3.2 Gradient Bandits

In this thesis two specific types of reinforcement learning are considered. First are gradient bandits. The "Bandit Problem" is a sequential learning problem, where each round the bandit has to decide which action to take by pulling one of N levers [9]. The analogy used by Sutton and Barto is of a room with several levers, and the question of which lever to pull (the pulling of a certain lever may also be called the action). From the bandit's perspective pulling any of the levers yields a certain reward and it is the bandit's task to find a policy for pulling levers which yields the maximum reward. Approaches to solving the bandit problem present frameworks for solving problems in which an actor repeatedly returns to a static situation with the same choices.

Applying this to the case of scientific workflow managers and the sizing of tasks one can consider the levers, or the choice, as the resource configuration. The bandit is asked repeatedly to allocate a certain amount of resources for a task (equivalent to pulling one of the levers) and must find the best policy for maximising the reward (i.e. the best "lever" or resource allocation).

Gradient bandits solve the bandit problem by using the gradient of the reward function to learn a preference $H_t(a)$ for each of the available actions (or levers). This preference $H_t(a)$ is updated after each round t. Using gradient ascent the bandits take small steps in the direction of the ideal preference for each lever to maximize rewards. These steps are taken via stochastic gradient ascent, for which the formula is :

$$H_{t+1}(a) = H_t(a) + \alpha \frac{\delta E[R_t]}{\delta H_t(a)} \tag{2.1}$$

Here $E[R_t]$ is the expected reward function and α is the step size.

This formula aims to increment the preference for an action proportional to the increment's effect on performance. The preferences influence the bandit's probability of choosing a given action. After choosing an action and receiving a reward, the preferences for all of the actions are updated to try and follow the gradient of the reward.

In practice however the expected reward for each action is not known- if it were known the problem would be trivial and the agent could be configured to always pick the maximizing action. Instead the expected reward function and its gradient must be approximated over time. This leads to the formula for updating the preferences proposed by Sutton and Barto. For a preference $H_{t+1}(A)$ after taking action A at timepoint t (denoted by A_t) and receiving reward R_t , where $\hat{R}$ is the average of all the rewards so far and $\pi_t(a_t)$ is the probability of picking action a_t, its preferences are updated as follows:

$$H_{t+1}(A_t) = H_t(A_t) + \alpha(R_t - \hat{R})(1 - \pi_t(A_t)) \qquad \text{and} \tag{2.2}$$

$$H_{t+1}(a) = H_t(a) - \alpha(R_t - \hat{R})\pi_t(a) \qquad \forall a \neq A_t \tag{2.3}$$

In this formula α is called the step size and it is a parameter which may be a constant or a function of the time t, but it must be greater than 0. It has been proven in [20] that this formula eventually approximates formula 2.1 for gradient ascent.

After updating its preferences, the gradient bandit then uses these preferences to inform the probability of choosing the next action at time $t + 1$. This can be done, for example, using a softmax distribution, and indeed in the rest of this thesis the softmax formula in 2.4 is what is used to update gradient bandits.

$$\pi_t(a) = \frac{e^{H_t(a)}}{\sum_{i=1}^{n} e^{H_t(i)}} \tag{2.4}$$

At the start (timepoint $t = 1$) the gradient bandit's preferences are all the same: $H_1(a_1) = 0, \forall a$. It then chooses actions based on the probabilities determined by the preferences for each action, updates its preferences with the reward it received according to 2.2, and uses 2.4 to update the probabilities, which in turn will determine the next action chosen. Through this method the gradient bandit will begin with a natural inclination to explore, and as it figures out which actions yield the greatest rewards, the probability of picking those actions will increase and the bandit will choose those actions more often, thus naturally striking a balance between exploitation and exploration.

It should be mentioned, however, that this balance is dependent on the step size α. For large values of α the bandit will spend less time exploring and more time exploiting. But despite this, if the nature of the rewards received for a given action begin to change, for example if an action goes from consistently receiving quite large rewards to consistently receiving poor rewards, then the formula in 2.2 will naturally begin to decrease its preference for that action and the bandit will begin exploration again- regardless of step size. This is one of the strengths of reinforcement learning: it can adapt to changes in the environment or the reward function.

Finally, some terminology- when the preference for an action has become so large that the probability of picking this action approaches 1, then the gradient bandit can be said to have "converged" to this action.

2.3.3 Q-learning

While the bandits in the previous section do not have a concept of state and must only learn about the actions available to them, stateful agents must maintain an internal state $s \in S$ and choose actions $a \in A$ which will yield new rewards and may cause them to transition to new states. The transition to a new state may also be independent of the action picked. The set of all possible states which the agent may conceivably find itself in and the set of available actions which the agent may potentially take are denoted as S and A respectively. Within this context of states and actions, a policy $\pi(s)$ must pick the agent's action a based on its current state s.

Q-learning was first proposed in [24] and it provides a simple way for an agent to act optimally in controlled Markovian domains [24]. An explanation of Markov domains or policy evaluation is beyond the scope of this thesis. Q-learning will therefore be explained within the context of the simplified agent-environment interface mentioned earlier (subsection 2.3.1 and figure 2.2).

Q-learning is an off-policy temporal difference control algorithm. An in depth explanation of temporal difference is also out of the scope of this thesis but a simplified explanation will come later in this section. Off-policy refers to the fact that q-learning is able to approximate the optimal action-value function independent of the policy being followed. The fundamental formula underpinning q-learning is called one step q-learning:

$$Q_{t+1}(s,a) = Q_t(s,a) + \alpha \times (r + \gamma \times max_{\hat{a}}(Q_t(\hat{s},\hat{a})) - Q_t(s,a)) \tag{2.5}$$

In this formula the function $Q_t(s,a)$ is a state-action value function. It reflects the value (which is based on both the historic rewards up to timepoint t and the expected future rewards) of taking action a in state s. When an agent takes action a it will receive reward r and transition to state $\hat{s}$. After this the agent must update its $Q_t(s,a)$ function to $Q_{t+1}(s,a)$.

If the old value function for s and a was $Q_t(s, a)$ then the update to this function is the difference between the old valuation, and the reward plus the discounted expected value of the new state. The expected value of the new state is $Q_t(\hat{s}, \hat{a})$, where $\hat{a}$ is the action which maximises $Q_t(\hat{s}, a)$ and $\gamma \in [0, 1]$ is the discount. This is called the temporal difference:

$$(r + \gamma \times max_{\hat{a}}(Q_t(\hat{s}, \hat{a})) - Q_t(s, a) \tag{2.6}$$

The temporal difference reflects the difference between the current approximation of the value function $(Q_t(s, a))$ and the optimal value function $(r + \gamma \times Q_t(\hat{s}, \hat{a}))$. If the current approximation is too optimistic then 2.6 will be negative and if Q_t is too pessimistic (too small) then the temporal difference will be positive. Q_t is then adjusted by the temporal difference multiplied with the step size α to yield Q_{t+1}. This is what equation 2.5 does. The only requirement for this iterative process to converge to the optimal value function is that all (s, a) pairs continue to be updated [20].

Using a value function built from equation 2.5 an agent can approximate an optimal policy by simply using a greedy approach and always picking the action a which maximises $Q(s, a)$, when it is in state s (and then updating its value function according to 2.5). Building on this idea, the epsilon greedy q-learning algorithm can be constructed. The pseudocode for this algorithm can be seen below.

This algorithm introduces random exploration with a probability ϵ and follows the q-learning algorithm the rest of the time. With sufficient exploration and an epsilon that eventually decreases to zero this algorithm should approximate an optimal value function and can greedily select those actions which maximise the Q function it is learning. This is the algorithm which will be used by all q-learning agents implemented in this thesis.

A final note: since the requirement for convergence to the optimal value function is that all pairs are visited, during early runs whenever the q-learning agent encounters a situation

```
Initialise $Q(s, a) = 0$ , $\forall(s, a)$
Initialise start state $s$
while True do
    if $random < \epsilon$ then
        choose action $a$ randomly
    else
        choose action $a = max_a(Q(s, a))$
    end
    Take action $a$, receive reward $r$, transition to state $\hat{s}$
    Update $Q$: $Q_{new}(s, a) = Q(s, a) + \alpha \times (r + \gamma \times max_{\hat{a}}(Q(\hat{s}, \hat{a})) - Q(s, a))$
end
```

Algorithm 1: Epsilon Greedy Q-learning Pseudocode

with an action it has never chosen before, it must choose this action. Only if all the available actions have been tried at least once can it use the greedy approach of selecting the maximising action.

Similar to subsection 2.3.2 the q-learning agent also has a step size parameter α, and the agent also has to balance exploration against exploitation. However for the q-learning agent its exploration can be directly influenced by the ϵ parameter, whereas the gradient bandit's exploration is implicitly tied to its preferences.

Chapter 3

Related Work

This chapter discusses other works in related areas and considers the nuances of their approaches and how they differ from the one presented in this thesis. First there will be a discussion of select pieces of literature which use reinforcement learning approaches for allocating resources in 3.1, and then three papers related specifically to resource management within the context of scientific workflows will be considered in 3.2. All of the papers mentioned here served as important references and inspiration for this thesis.

3.1 Using Reinforcement Learning to Allocate Resources

SmartYARN [26] applies a q-learning approach to balance resource usage optimization against application performance. The need to balance reducing costs by using less resources against the need to increase runtime by using more resources is one of the central considerations for any client of a cloud computing platform. In this paper the researchers used the performance of the application under a certain resource configuration as the state-space, while the action-space was made up of just 5 actions: increasing or decreasing one unit of CPU or memory, or keeping the previous allocation. In the end the researchers found that the agent was able to achieve 98% of the optimal cost reduction and generally performed at the optimal level, finding the optimal allocation the vast majority of the time.

This paper served as the inspiration for the q-learning approach used in this paper and is also one of the few works discussed in this section which considers the central issue of cost versus performance when assigning resources. One significant difference however is that SmartYARN only seeks to optimise individual jobs which do not depend on each other, whereas for scientific workflows the jobs (or tasks) are part of a larger workflow and depend on each other and may need to wait for other tasks to complete before they are ready to be executed.

[18] tackled a similar problem: configuring the resource allocations of virtual machine's (VM's) using reinforcement learning. Their approach modelled the problem as a continuing discounted Markov Decision Process and solved it with q-learning. The states were always a triple of CPU Time credits (used for scheduling cpu time), virtual cpu's (vCPU's), and memory. The agent's available actions were increasing, decreasing or leaving the allocations, with only one resource allowed to be changed per action. As a reward the ratio of the achieved throughput to a reference throughput was used. The key trick used by the researchers was to use model based reinforcement learning. By modelling states the agent is able to simulate or predict the reward it can expect from previously unseen action-state pairs, whereas the classic q-learning approach used in this paper requires the agent to experiment with each action-state pair. This means the model based agent is able to learn much faster and can enter its exploitation phase earlier than the static agent which must explore the action-state space for a long time, especially when there are large state spaces. Ultimately however the researchers found that the static agent performed quite well and achieved high levels of throughput in its own regard, although the model-based approach consistently outperformed the static one.

This paper's problem and its solution to it are again quite similar to those of this thesis but there is a significant difference because it schedules the resources of virtual machines as opposed to the resources of individual jobs. Another difference is the use of model based reinforcement learning to limit the time spent exploring the state action space to approximate a value function and spend more time exploiting this function to achieve better results.

Finally there is the DeepRM [15] paper on resource management using deep reinforcement learning. Here the researchers used a policy gradient reinforcement learning algorithm combined with a neural network. The state-space consisted of the resources of the cluster and the resource requirements of arriving jobs, encoded as so called ”images” which could be fed into the neural network. The actions available to the agent were simply to choose a job and allocate resources to it. In order to speed up the process the agent was given the option to schedule up to M jobs in a row, thus enabling it to leap forward by as many as M timesteps instead of always progressing by a single timestep. The reward given to the agent was the negative average slowdown. In the end the researchers' approach outperformed both the shortest job first metric and a very strong, heuristic based algorithm called Tetris [12]. A key reason that the agent was able to achieve this increased performance was that it learned a policy to maximize the number of small jobs it completed. While the approach taken in the paper was still better with large jobs it was significantly better with small jobs because the network learned to always keep a small amount of resources available to quickly service any small jobs which might arrive, whereas the other approaches inevitably scheduled all the available resources so that small jobs also had to wait.

The similarities here are relatively few since this paper also had to concern itself with the decision of which job to allocate and when, whereas this thesis only considered how much of a given resource to allocate to a task. Beyond that there is also the fact that a neural network was employed and the decision to use the resources of the system to model the state of the agent. Though this is an idea which could be interesting as a topic of further research if one were to expand on this thesis. What is most interesting however is the policy that the neural network was able to learn. It was able to maximise throughput by always reserving a small amount of resources for small tasks which re-occur often. This kind of advanced strategy is something that could be built into an algorithm but the ability of a reinforcement learning agent to learn such a strategy shows their potential for impressive results within the field of scheduling and resource assignment.

3.2 Resource Allocations in Scientific Workflows

In [2] the researchers used cluster profiling to build node groups with similar performance profiles and map a task to a node group based on historical performance data. A 3 step system is used. In step 1 the cluster profiles are built based on microbenchmarks and hardware analysis tools. Then in step 2 the tasks are labelled in the SWMS based on their monitoring data. Finally, in the third step dynamic task-resource allocation is performed based on the information from the previous steps. Using this approach the authors were able to achieve a mean reduction of job runtimes of 19.8% compared to popular schedulers, and a reduction of 4.54% compared to the shortest-job-fastest-node (SJFN) heuristic.

This target of that paper is of course much more similar to that of this thesis than any of the previously mentioned papers, however the obvious difference is that this thesis approaches task resource assignment with a reinforcement learning approach. One other similarity is that this paper used the exact same SWMS (nextflow) as this thesis. Beyond that it would be an interesting area of research to try and combine the topic of that paper with the topic of this thesis and try and apply reinforcement learning to task resource assignment amongst node groups with similar performance profiles.

Next, in [25] a feedback based resource allocation system for the scheduling of scientific workflows is designed. Their approach uses an online feedback-loop to improve resource usage predictions and thus more accurately allocate resources to the constituent tasks of a scientific workflow. Tasks within scientific workflows were assigned memory based on percentile or linear regression predictors. One such percentile predictor, called PC50, worked by using the median memory usage of a task as its prediction. This approach was significantly outperformed by the *LR mean* approach which was a linear regression predictor that predicted memory usage based on a linear regression that relates a task's input data to its memory usage. This prediction is then offset by the standard deviation between the predicted and the true peak memory usage.

This paper addresses a very similar problem to that of this thesis but attempts to use linear regression to predict memory usage whereas this paper is focused on using reinforcement learning. Of course it also does not attempt to predict CPU usage or assign CPUs to tasks. The PC50 approach and *LR mean* approaches ultimately served as inspiration for the feedback loop approach with which the agents in this thesis are compared. The feedback loop approach used in this thesis takes the PC50 predictor and adds an offset based on standard deviation, much like *LR mean*, to try and predict memory usage.

Finally there is [22] which presents a job sizing strategy for tasks in scientific workflows. The researchers used equations based on integrals over the resource usage to either minimise waste or maximise throughput. When gathering training data they assigned tasks the maximum allowable value for all of their resources. This was repeated until the task's resource usage statistics began to converge. Then they used their analytical solution to assign resources and if the task failed or was killed it was retried again with the maximum available resources. Ultimately an increase in throughput of between 10 and 400 was achieved.

The approach in this paper also served as inspiration for the feedback loop approach used here. The idea to assign the maximum resources available and observe a task's performance is an interesting strategy for determining how to size a task. It could be considered an alternative form of exploration to that in reinforcement learning because it determines a task's resource requirements based on the task's usage when the maximum possible amount of resources are available as oppposed to exploring different allocations. It is critical to note that an assumption is made that a task's resource usage does not depend on the available resources. Lastly unlike in [25], which only considers memory, [22] looks at CPUs and disk space and how to assign these resources to tasks within scientific workflows.

3.3 Points of Interest from these Works

All of the papers discussed in section 3.1 managed to utilize reinforcement learning in an administrative fashion to improve the performance of jobs in their respective computing environments. A common theme among the papers was to limit the action space to increasing, decreasing or keeping the current allocation or resources and storing the current allocation as the agent's state. This approach is also used here for the q-learning agent. The majority of the papers also utilized neural networks and it is also worth noting that most of them used runtime in their reward functions.

In section 3.2 the topic of resource management in scientific workflows was looked at and the papers mentioned all tended to use more analytical approaches. Indeed there is relatively limited literature regarding combining machine learning and scientific workflow management. A consistent theme in the papers about sizing tasks in scientific workflows is the idea that a task's resource requirements are relatively static and can be learned over time using linear regression or profiling. One topic which is not really addressed in these papers is the relationship between resource assignment and performance, because assigning a task more resources (which may be less efficient) can speed up performance. This is an important topic in this thesis and the papers [25] and [22] do not mention runtime, while [2] does mention it but does not deal with task sizing. In this thesis the effect of task sizing on workflow performance is an delicate tradeoff which must be balanced correctly to both reduce resource wastage and maintain reasonable runtimes and improve (or at least not harm) workflow performance.

The papers [22, 25]in section 3.2 both served as direct inspiration for the feedback loop approach in this thesis which is compared with the reinforcement learning agents. The approach functions by combining the training process from [22] with the *LR mean* allocation scheme from [25].

Chapter 4

Approach

This chapter covers how a solution was integrated into a scientific workflow manager, as well as the specific parameters and reward functions of the reinforcement learning agents. In section 4.1 the software architecture of the approach and its integration with the nextflow source code will be discussed, and then in 4.2 and 4.3 respectively the details of the gradient bandit and the q-learning agents will be considered.

4.1 Integrating a Solution into a Workflow Manager

Nextflow [8] is a robust scientific workflow management system written primarily in groovy. It supports various execution platforms and has a large variety of tools to help users analyse the performance of their workflows. Nextflow is free open source software and for this thesis a fork of the 20.12.0 version was used [1]. In order to integrate a reinforcement learning approach with the nextflow source code a simple structure was designed which externalised the storage of data from previous tasks and workflows so that when a task needed to be scheduled the reinforcement learning agent could retrieve historical data about the task from a database to

[1]https://git.tu-berlin.de/zunkernicolas/nextflow-extension

inform its decision. The structure can be seen in the following figure.

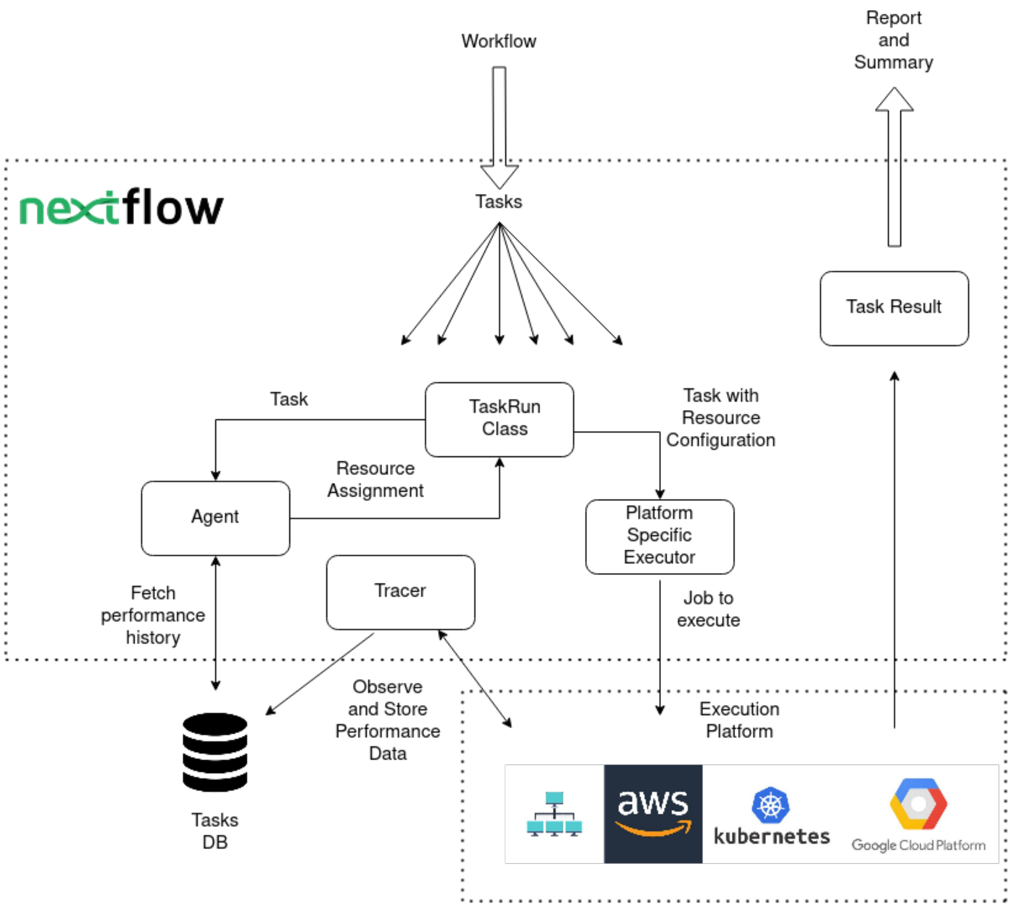


Figure 4.1: High Level Design: Integration of a Reinforcement-Learning Agent into Nextflow

Although the code for the reinforcement learning agents is logically a part of nextflow, when discussing the above design they will be referenced separately. The interaction between nextflow, the agents, and the external actors (the user, the database and the execution platform) is as follows. First, before a task is ready to be scheduled it is sent to a reinforcement-learning agent specific to that task, for the purposes of this thesis that was either a q-learning agent or a gradient bandit. Next, the task's agent would then retrieve historical performance data and any other relevant data (i.e. what state the agent was last in) from the database. This database is external to nextflow and the creation and management of this database is not performed by nextflow or the agent which only use the database to save or read data beyond a workflow's lifetime. Then, using the data, the agent selects a new resource allocation for the task and overwrites the task's default configuration. After this, nextflow uses its custom executor for the given execution platform to prepare the task and pass it on to that platform. The task is then executed on the platform. It is important to note that the execution platform will also have its own system for managing, scheduling and executing jobs and processes. As the task is executed nextflow's tracer module will gather performance data, i.e. peak CPU usage and peak RSS [2]. Finally, once the task is finished all of this data is stored in the database for the agent to use the next time the task comes. It is important to note that there is one agent for each unique task. These agents are called whenever their task needs to be executed and they use the database to receive rewards for their previous actions.

Now that the structure of the agent's environment has been explained and the relationship between collecting data and assigning a task resources is clear, it is time to delve into the specific approaches tried.

4.2 Gradient Bandits

In the course of this thesis gradient bandits were used to allocate both CPUs and memory. The way they did this will be explained here.

4.2.1 CPU Bandit

Design

The CPU Bandit, as it was nicknamed, had a very simple set of actions to choose from which were based on the number of CPUs available to the system. The bandit then learned how many CPUs to allocate to its task. Its reward was a very straightforward function:

[2]https://www.nextflow.io/docs/

$$reward = -t \times (1 + cpus - cpu\_usage/100) \tag{4.1}$$

In this equation cpu_usage is a value in percent of the number of single core CPUs used by the task. If a task were assigned 4 CPUs and only effectively used two of them then cpu_usage would be 200%. $cpus$ is the number of CPUs assigned to the task and t is how long the task took to run. Put in other words this equation punishes the agent with a negative reward equal to the amount of time it ran plus the unused CPU time (execution time multiplied by the number of unused CPUs). The idea is to provide an incentive for the agent to try to minimise the amount of time where any of the allocated CPU's are not being used, and by punishing it with the amount of time that the task ran for, the agent is also encouraged to try to reduce the amount of time taken for the task to complete. This also means that, should a task have 100% usage then its penalty is not 0 but is just the execution time. The reason for including time in the reward function is that if the agent has no concept of time it will always allocate the least amount of resources possible, since that is guaranteed to minimise the amount of resources wasted, and ultimately the tasks and their workflows would all be incredibly slow.

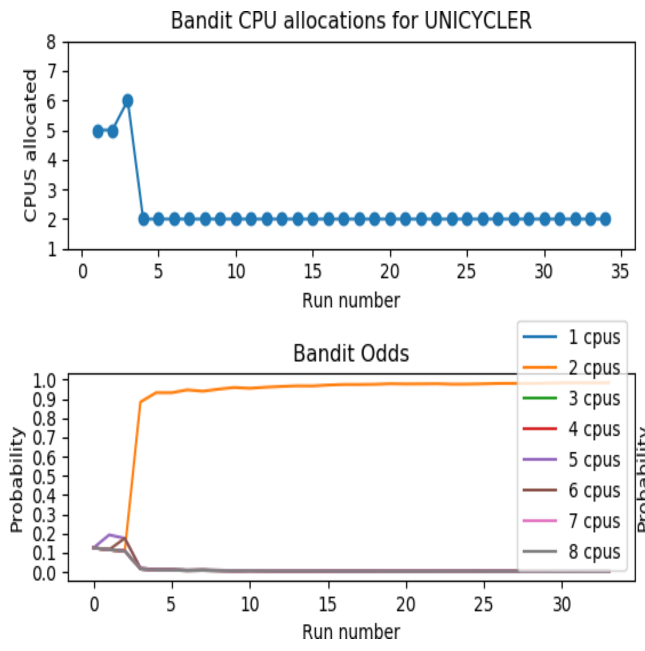
Picking the Right Step Size

The most pressing issue encountered with the CPU bandit approach was related to the step size. The initial step size α was 0.1, which would be an appropriate step size for a bounded reward function with a mean reward that is around 10 or is known to have a standard deviation less than that (these values are based on the example given in [20] where they use a step size of 0.1 for a reward function with a statistical mean of 4). However the reward function used in 4.1 has the bounds $[\,-time \times (cpus + 1)\,,\,-time\,]$ and as such is unbounded, since $time$ can be arbitrarily large and the standard deviation which was observed for the rewards was also too large. Therefore whenever tasks had a runtime > 10 seconds, α was too large.

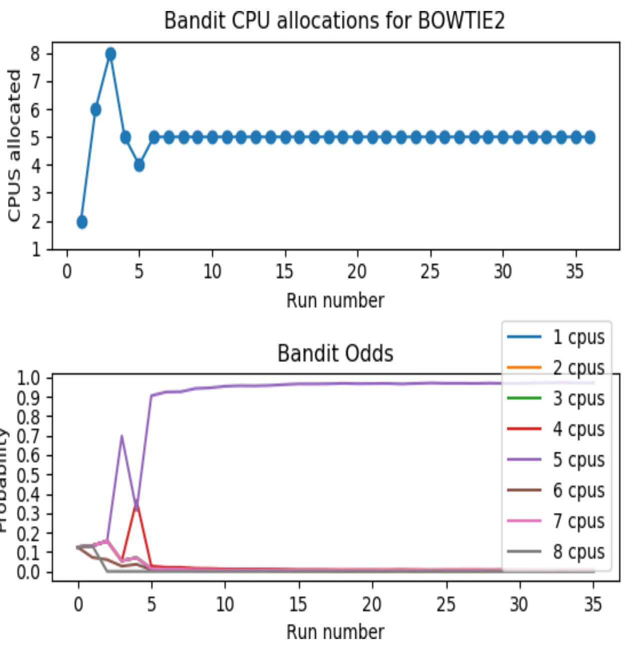
The danger in using a constant step size which is too large for the reward function is that when the bandit receives a reward which is considerably larger than the previous average the "step" taken in the direction of that action will also be too large and the bandits preferences will be updated such that the given action is always preferred and no other actions are tried any more. This causes the bandit to cease exploration too early. This effect was a common problem for tasks which took longer than 20 seconds to complete and was especially pronounced for tasks with runtimes of 100 seconds and more. For these tasks the inherent volatility of the reward function relative to a step size of 0.1 meant that the bandits were converging very early and often had not explored the other actions at all. To provide a concrete example: if a bandit's third allocation of CPUs received a reward of -150 and the first two allocations had had an average reward of -200, then for a step size of 0.1 the preference for the new allocation would increase by approximately $50 \times 0.1 = 5$. Now because of the exponential nature of the soft max distribution function used on the preferences, the probability of picking that action would increase by a large amount and then the bandit is in danger of converging to that allocation without any more exploration. This is best demonstrated graphically, as can be seen in figures 4.2a and 4.2b.

These figures show two gradient bandits which were exhibiting the behaviour described above. Each figure is made up of 2 graphs which display the action chosen by the bandit in the top graph and the probabilities for all of the actions in the bottom graph. As can be seen in the two examples, after receiving a relatively large reward, the agent's probability of choosing a given action rises far too quickly and no other actions are attempted thereafter. This is occurring because the step taken by the bandit in the direction of a single action's preference is too large and the resulting probability of choosing that action grows so much that no other actions are explored any more.

The solution to this problem is to adjust the step size so that it is not a constant value but is instead a function which uses the average time that it usually takes the task to complete.



(a) Bandit 1



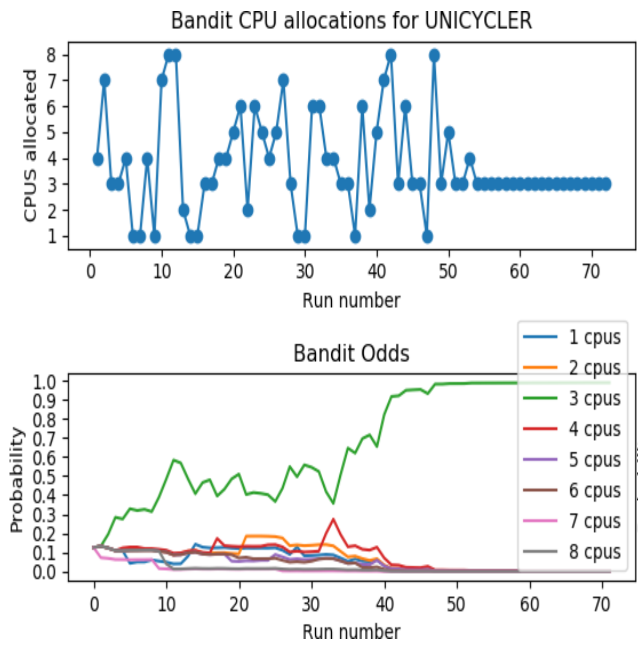
(b) Bandit 2

Figure 4.2: Example of Two Bandits Converging Too Fast

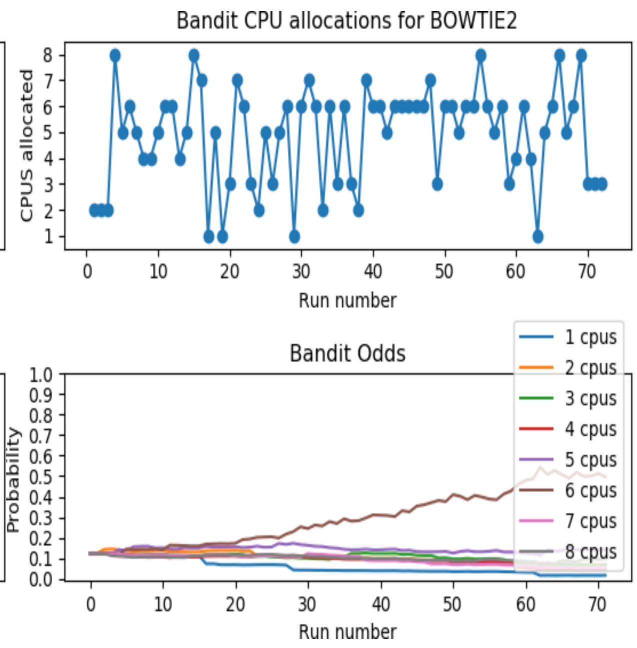
This way tasks which have longer average runtimes are given smaller step sizes. The new value for α is thus $1/avg(time)$.

The next two figures 4.3a, 4.3b show the same two tasks and their bandits but with the new step size. As can be seen the exploration phase is considerably longer and many different CPU allocations (actions) are tried before settling on one. Specifically in the case of the *UNICYCLER* task's bandit we can see that although allocating 3 CPUs seems to develop a strong preference early on (perhaps because it performed abnormally well once or perhaps because it genuinely is the best performing allocation) it does not become so preferred as to be dominant until much later and until that point the bandit continued to try other options.

It should be noted that these changes mean that for optimal performance of the bandits there should already be some historical data available about the task's and their runtimes. It is not a necessity because over time the bandit can gather this data itself, but it is preferable.



(a) Bandit 1



(b) Bandit 2

Figure 4.3: Bandits with a Step Size Based on Their Historical Average Execution Time

4.2.2 Memory Bandit

The other gradient bandit approach used for assigning resources to tasks was called the memory bandit and its available actions were memory assignments. Beyond the obvious, the memory bandit was different from the CPU bandit in two respects. Firstly, the amount of memory available is several order of magnitudes larger than the amount of cpus. Thus, in order to limit the number of available actions to a manageable amount, the original amount of memory assigned to the task m (in the default configuration) had to be split up into n chunks each of size $c = m/n$. The bandit is then presented a actions, each of which assigns $a \times c$ amounts of memory to the task. a can be chosen as $1.5 \times n$ to give a good balance between considering allocations that are larger and smaller than the default configuration, however since it is known in advance that user estimates are usually quite poor one could also choose $a < n$ to encourage memory efficiency. In addition to this, by choosing larger or smaller n values the size of the memory chunks can be adjusted. Secondly, and this is the biggest difference between the two bandits, a task which is assigned too little memory and overuses it will be killed. This means that there must be a special penalty for such cases.

The reward function used for the memory bandit, after assigning *memory* bytes of memory is as follows:

$$reward = \begin{cases} -2 \times memory/c, & \text{if the task ran out of memory} \\ -1 \times unused\_chunks & \text{otherwise} \end{cases}$$

Where $unused\_chunks = (memory - peak\_rss)/c$ and $peak\_rss$ is the peak resident set size (RSS), in bytes, of the task during its execution.

An obvious alternative for this function is to simply use the memory usage: $reward = peak\_rss/mem$, and $\alpha = 0.01$ since the reward function's range is [0,100]. However during testing this was found to be too small for tasks which have a low memory footprint. For example a task which uses 1% of the assigned memory when assigned n chunks, and thus receives a reward of 1%, will only yield a maximum reward of n% for the optimal allocation of just one memory chunk and for such small differences a gradient bandit will struggle to recognize the superior choice, even though it should be obvious. This is especially damaging to performance because these low usage tasks are precisely the ones where memory efficiency can be optimized the most and as such the reward function was changed to one which penalises the agent for the amount of unused memory. Of course to avoid the problem the CPU bandit had with its step size, the amount of unused memory in bytes could not be used and therefore the number of unused memory chunks was used instead. To return to the example of a task with 1% usage that task's rewards would now be $-n$ at worst and -1 at best, which makes it much easier for the bandit to determine good allocations. Finally in the case where a task has been killed the penalty is equal to double the number of memory chunks assigned because the agent must (at the very least) assign the same amount of memory again or must assign even more memory.

The actual implementation of the memory bandit is slightly adjusted so that whenever a task fails with one memory allocation and is retried with a new one, the new allocation must be greater than the previous one. If the bandit does pick a smaller allocation then it is ignored and the new allocation is either doubled or the default configuration for that task is used (if double the new allocation is also less than the failed allocation). This prevents the workflow from being broken off if a given task fails too many times due to repeated poor allocations.

Finally, to return to a discussion of step size similar to that in 4.2.1, the memory bandit can be assigned a constant step size $\alpha = 1/n$. This is is because for n memory chunks, the bandits reward function is bounded by $[-n, 0]$.

4.3 Q-learning Agent

Similar to the approach with the gradient bandits, there was one q-learning agent per unique task. The q-learning agent's state was always the current allocation of CPU and memory for the given task, and as its set of actions it could choose between incrementing or decrementing the amount of CPUs or memory, or it could do nothing. To limit the state space the maximum and minimum number of CPUs and memory as well as the amount by which they were incremented or decremented was based on the default allocation given to the task by the developers of the workflow (provided they did not exceed the resources of the execution platform). When the task was scheduled by the agent for the first time it would start in the default configuration's state. Just like the memory bandit in section 4.2.2 the possible memory states of the reinforcement learning agent are also chunks of memory.

For the q-learning agent a different reward function was used than for the CPU bandit, primarily because it also had to incorporate memory but also as part of an attempt to experiment with different reward functions and approaches.

$$r = -max(0.1, unused\_cpus) \times (t/avg\_t) \times (mem \times (1 - max(0.75, mem\_use))) \qquad (4.2)$$

Here the *mem* variable refers to the memory allocated to the process and *mem_use* is the value of of the peak RSS of the task divided by the memory assigned to it. *avg_t* is a constant value which is determined at runtime based on the historical average execution time for the task. Lastly, $unused\_cpus = cpus - cpu\_usage/100$ where *cpus* and *cpu_usage* are the same as in 4.1.

Since division by the task's average time is incorporated into the reward function it did not need to be incorporated into the step size as with the gradient bandit (see 4.2.1) and the issues associated with that were avoided. This function is effectively a product of the number of unused CPUs, the slow-down factor and the amount of unused memory. However there are some slight modifications. The *max* function is used to set an artificial floor for the penalty incurred by the unused CPUs and unused memory. Tasks which use more than 90% of the available CPUs are given the same reward as tasks which use exactly 90% in an attempt to prevent the agent from deciding to assign each task 1 CPU in order to minimise the number of unused CPUs. Additionally the floor for unused memory is capped at 25% in a similar manner. This is done to discourage the agent from allocating too little memory because tasks which use too much memory will of course be killed and have to start over, which can have a detrimental effect on performance. This reward function is of course negated in order to turn it into a penalty function so that the agent will seek to minimise its penalty by minimising the number of unused CPUs, the slowdown and the amount of unused memory.

The q-learning agent also has 3 other parameters- the step size α, epsilon ϵ and the discount γ. Step size was set to 0.1 and the discount was 1.0. Epsilon is adjusted over time- at first it is 0.5, to encourage exploration, then after 50 runs it is decreased to 0.25 and after 100 runs it is 0.1 to discourage exploration but leave room for the bandit to still occasionally try other actions and visit different states.

Exploring the State-Action Space

It should be noted here that unlike the gradient bandits from before, which had to decide from c or n number of actions for the CPU and memory bandits respectively, the q-learning agent must learn the value function for $(c-2)*((m-2)*a+2*(a-1))+2*((m-2)*(a-1)+2*(a-2))$ state-action combinations. Here c is the number of possible CPU states, m is the number of possible memory states and $a = 5$ is the number of possible actions. Now since the q-learning algorithm from 1 needs to try each action in every state combination at least once, and explores with probability ϵ then for arbitrary examples of $c = 4$ and $m = 3$ it will take, at the very least, 46 different runs just to have tried every action in every state once. This shows how much more complex the q-learning approach is than the gradient bandit approaches. Ultimately however the theoretical benefit posed by q-learning is that when the agent learns, for example, not to increment the amount of CPUs when in state s (where s is also the number of CPUs currently allocated), the agent is thus also learning to avoid all states/allocations $\{s_c \in S \mid s_c > s\}$ which allocate more CPUs than s. While this seems trivial, the gradient bandit is unable to see such connections and cannot tell that a task which has low CPU usage for s CPUs can be expected to have even lower usage for $s_c > s$ CPUs.

Chapter 5

Evaluation

This chapter looks at the set-up for the experiments and presents and discusses the results achieved. In section 5.1 the environment in which the agents were tested is explained and then in the section 5.2 the results are presented and discussed.

5.1 Testing the Agents

In order to evaluate the performance of these agents they were tested on a 32-core, 125GB RAM machine against 5 different workflows from the popular bioinformatics framework nf-core [1]. Each workflow is first run 10 times without any reinforcement learning agents active. This was done to have something to compare the results to later but also so that there was historical data about the average execution time of each of the tasks, since the reinforcement learning agents need this information. After that the agents are tested. The gradient bandits are run 50 times for each workflow and the q-learning agent 100 times. As has been explained in 4.3 the q-learning agent needs considerably longer to explore its extensive state-action space and as such needs to be trained for longer. However even though it is run twice as

[1]https://nf-co.re/

many times as the gradient bandit, the q-learning agent is still perhaps under-trained since the state-action space it needs to explore is so much larger than the action-space which the gradient bandit explores.

The five workflows used are the following: 1) *eager* - a pipeline for genomic NGS sequencing data, 2) *mhcquant* - a workflow for quantitative processing of data dependent (DDA) peptidomics data, 3) *nanoseq* - which is an analysis pipeline for Nanopore DNA/RNA sequencing data, 4) *viralrecon*- which can be used to perform assembly and intra-host/low-frequency variant calling for viral sample, and 5) *metaboigniter* - a pipeline for pre-processing of mass spectrometry-based metabolomics data.

The input data used for the workflows is based on custom combinations of different input files provided by the workflow's maintainers. Generally for each pipeline there are full example datasets and short example datasets as well as configurations for each of them. The short datasets are all configured to only assign either 1 or 2 CPUs and are thus are not really feasible for comparison, however the full datasets, whilst featuring realistic configurations, often take far too long to run. For reference, the full datasets will often take 6 hours or more to run once whereas the short examples never take more than 5 minutes. In order to build good pipelines with more realistic runtimes, configurations and input data, but without running too long, the short examples and the full ones need to be combined. The combinations are designed to strike a balance between having large input data and realistic task configurations but also being short enough that it is feasible to run the workflows hundreds of times. Each workflow's set of inputs is built individually but the overarching tactic used was to start with the command line arguments to the pipeline from the short examples and combine those with the configuration and the input data from the full dataset. After this the list of input files is reduced until the pipeline completes in a reasonable amount of time. By following this process, configurations and inputs were found for all five of the pipelines mentioned above such that they were completing after around 10 to 40 minutes.

Nextflow inherently keeps track of the total number of resources available and the resources already assigned to tasks and it will not schedule more resources than the system has left. The tasks are all run inside of docker containers with the exact amount of CPUs and memory requested by the task. Since the tasks are all being run on the same machine it would have been possible to ignore the sum of the resource requests and over-assign resources so that all of the tasks compete with each other, but this does not reflect situations where pipelines are being run on clusters with resource managers, and as such this was left unchanged.

For the final comparison in the next section the 10 runs with the default configurations are compared with the last 10 runs of each of the agents and 10 runs using a feedback loop approach. The feedback loop approach functions as follows: first, during training, the workflows are run 10 times and each task is assigned the maximum number of CPUs and memory available. Then, in the next 10 runs the task are assigned the floor of the average number of CPUs used during the training and the mean amount of memory used plus the standard deviation. Should a task fail, then it is retried with the maximum amount of memory ever used by that task during the training time, and should it fail again then it is retried with double that value. This feedback loop approach has been mentioned in chapter 3 and it is loosely based on the work from [22, 25].

5.2 Results and Discussion

5.2.1 Performance Comparison

In this section the performance of the gradient bandit and the q-learning approaches over their last 10 runs will be compared with each other and with the performance of the default configuration and the approach using a feedback loop.

The figures 5.1 and 5.2 show how these approaches performed. The left graph in the first figure shows the total amount of CPU hours used by all of the tasks across all of the workflows

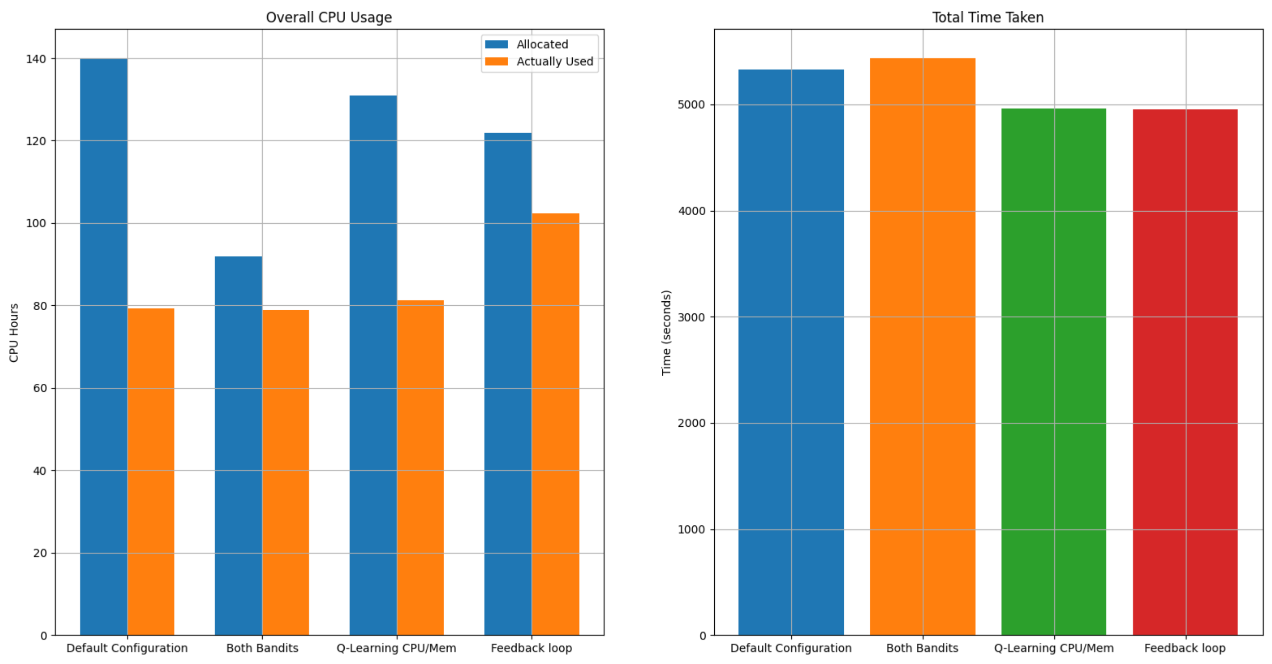


Figure 5.1: Performance of the CPU and Memory Approaches

and the amount of CPU hours that were actually used (also called the effective CPU hours). Effective CPU hours are the CPU usage multiplied by the execution time whereas the CPU hours are just the number of CPUs assigned multiplied by the execution time. For example a task which was assigned 3 CPUs, ran for 1 hour and had 200% CPU usage would have used 3 CPU hours but only 2 effective CPU hours. These values are calculated for each individual task and then summed up over all of the workflows.

The right graph in the first figure shows the cumulative amount of time it took for all of the workflows to run to completion. Additionally in 5.2, the memory usage is shown. The unit measured in the right graph of 5.2 is the gigabyte hour and the graph displays the total amount of gigabyte hours allocated across all of the workflows as well as the actual amount of gigabyte hours used (based on each task's peak RSS and execution time- this is the minimum amount of gigabyte hours which could have been assigned without the task being killed). The left graph of that figure shows the average memory usage across all of the tasks, where memory usages is considered the task's peak RSS value divided by the total memory assigned to it.

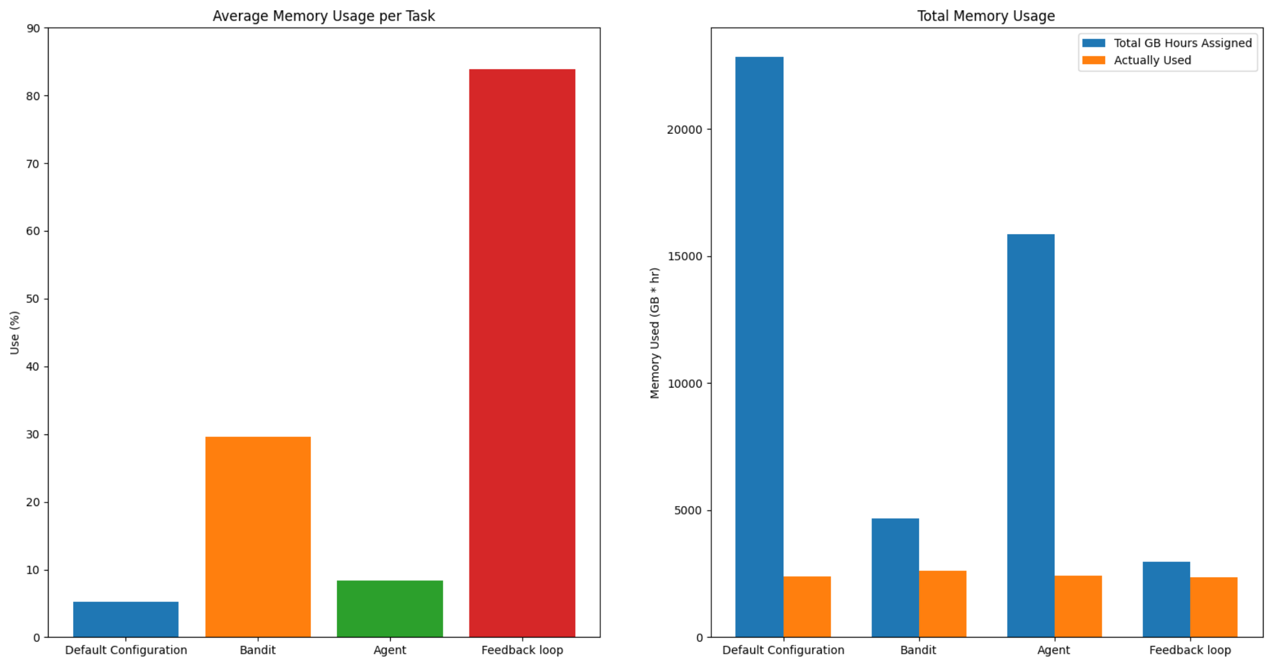


Figure 5.2: Memory Usage

From these graphs it is clear to see that all of the reinforcement learning approaches managed to improve CPU efficiency and allocate less CPU hours than the default configuration. In the case of the gradient bandit it was particularly effective at keeping the overall CPU hours assigned to an absolute minimum, however it seems that this efficiency came at the cost of speed, and it performed worse than its q-learning counterpart in this aspect and took slightly longer than the default configuration. The q-learning approach was faster than the default configuration and was more efficient with regards to CPU usage. As can be seen, both of the reinforcement learning approaches are able to assign less CPU hours without having a detrimental effect on execution time (in fact the q-learning approach assigns less CPU hours but executes faster). Looking at the reward function 4.2 for the q-learning agent the "CPU usage"considered by the function is not the task's usage of the task but rather $max(90, CPU\_usage)$. As such agents for tasks which exhibit very high CPU usage ($> 90\%$) receive no additional reward for increased efficiency (since efficiency can always be increased by decreasing the total number of CPUs assigned) so their rewards depend entirely on the execution time and they are thus encouraged to find resource allocations which decrease runtime. This most likely explains why the gradient bandit is more efficient but the q-learning agent is faster.

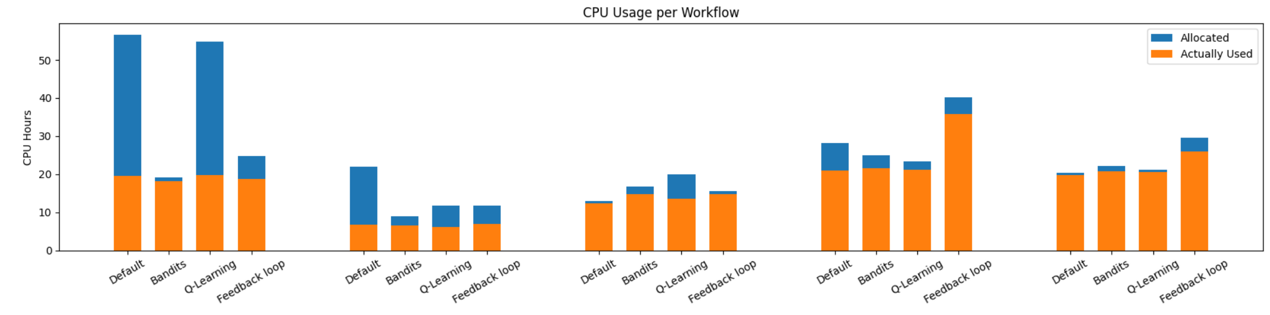
When compared with the feedback loop, the gradient bandit is 11% slower but uses 25% less CPU hours, whilst the q-learning agent performs worse in every aspect but achieves similar values for CPU hours assigned and execution time (it assigned 7% more CPU hours and was only 0.03% slower). Ultimately the fact that the feedback loop approach assigns less CPU hours than the q-learning agent but is still faster indicates that the q-learning approach has not found the right balance between increased resource usage at the cost of faster execution, however the fact that the gradient bandits assigned proportionally less CPU hours than the factor by which it was slower than the feedback loop approach indicates it is able to balance reduced execution times for efficient CPU allocations.

Interestingly, the feedback loop ultimately wound up increasing the effective CPU hours used by the workflows, something which had remained constant under the other 3 configurations, and the feedback loop approach was actually quite efficient in terms of the total amount of CPU hours assigned- relative to the increased effective usage. It can only be speculated as to why it used more actual CPU hours than the other approaches but it may be down to the fact that certain highly parallelizable tasks will be assigned incredibly large numbers of CPUs (as an example a task with 2800% CPU usage during training was assigned 28 CPUs), far beyond the point of diminishing returns, and eventually lead to the workflow using more effective CPU hours. This would indicate that assigning more resources may actually increase their usage, something which the feedback loop approach assumes is false.

Concerning memory allocations, both of the reinforcement learning approaches proved to be better than the default configuration. One aspect which stands out is how poor the default configurations are. Indeed on average they are using just 5% of the assigned memory on a task by task basis . On the other hand the q-learning approach is not much more efficient, however it still reduces the wasted memory by a large amount (5000 gigabyte hours saved compared to the default configuration). Yet this amount is dwarfed by the other two approaches which save close to 15000 gigabyte hours and reduce the memory assignments by 80 and 87 % respectively. The gradient bandit approach performed better than the q-learning approach by a significant margin and it assigned less than half as much as either the default configuration or the q-learning agent. The feedback loop approach is clearly the best, assigning the least amount of memory hours overall and showing very high memory usage for all of the tasks.

Ultimately the performance of the q-learning agent is down to the fact that the number of chunks of memory it could choose from was very low (in order to have less states so it could be trained faster), and thus it had a fairly limited extent to which it could reduce the memory usage. The gradient bandit suffers from this too because ultimately it also only had a fixed number of memory chunks to choose from and could only increase or decrease a task's memory usage by a multiple of the size of the memory chunk. The feedback loop does not have this problem and can assign memory at the most granular level possible. It therefore follows that it would naturally be able to outperform these two approaches which can only assign discrete chunks of memory. Another factor to consider is that both of the reinforcement learning approaches also had the option to assign more memory than the default configuration. This was clearly unnecessary, and because all of the tasks were using much less memory than the default amount it would have been possible to change the agents options to only be chunks of memory which added up to less than the default amount. This would have increased their performance.

This same problem which affected the q-learning agent's memory allocation performance is also reflected (to an extent) in the CPU allocations, where the q-learning agent is inefficient, relative to the other two approaches, because it was given a limited number of CPU allocation options to choose from in order to keep its state-actions space small.



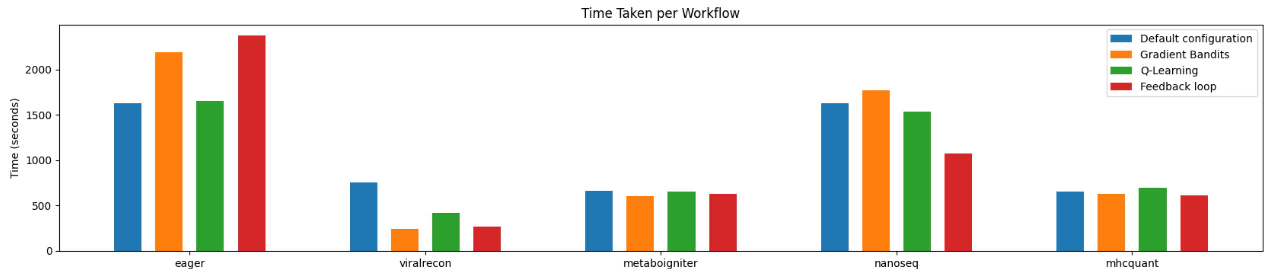


Figure 5.3: CPU and Execution Time

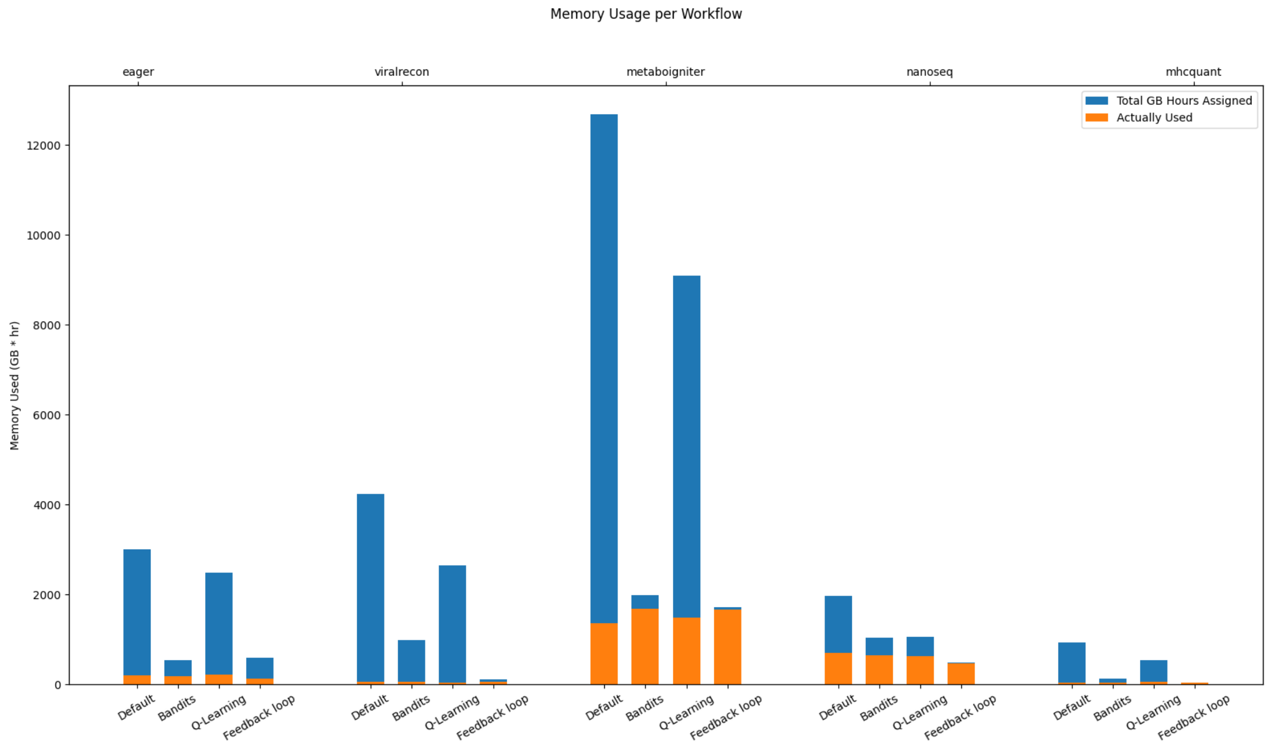


Figure 5.4: Memory

5.2.2 Performance on a Workflow by Workflow Basis

In this section the results are shown by workflow, the first figure, 5.3 shows the CPU usage and execution time in two separate graphs, with the bars grouped together by workflow and labelled with the approach. The workflows in the top graph of figure 5.3 correspond to the labels in the bottom graph. The second figure, 5.4, in a similar fashion, shows the memory usage.

It is quite difficult to find any outliers or patterns in this data and really the performance of the different approaches varies from one workflow to another, with each approach at one point or another being either the slowest or the fastest. Worth noting is of course the *eager* workflow. While the CPU assignments and base usage are quite similar for most of the other workflows, with *eager* the default configuration and the q-learning agent assign noticeably more CPU hours than are needed, however they achieve a faster execution time as a result. This indicates firstly that the authors of the workflow were most likely aware that there were tasks in this workflow which may have low CPU usage for periods of times but their execution speed still scales with more resources. Secondly this indicates that the q-learning agent was able to learn this and make the decision to assign more resources, even if the usage is low, because they decreased execution time. This is a direct result of the q-learning agent's reward function, which is geared slightly more towards speed. Lastly it also indicates the need to account for Input/Output (I/O) intensive tasks. These tasks will generally have lower CPU usage because they must wait for system calls to complete read and write operations but may still benefit from increased CPUs because they are quite CPU intensive during their "off" time between I/O operations.

Beyond this it can also be seen in the *nanoseq* and *mhcquant* workflows that the feedback loop assigns more CPU hours than any other workflow and it's effective usage is a bit higher. This may indicate, as was speculated in 5.2.1, that these workflows contain highly parallelizable tasks whose usage depends on the resource assignment.

5.2.3 Summary

In conclusion all of the reinforcement learning approaches were able to intelligently size the tasks in the workflows and in the end the overall number of CPU hours which were wasted was reduced. In addition to this the agents were also able to significantly reduce the memory usage compared to the default configuration but struggled to reach the near-optimal performance of the feedback loop. The gradient bandit took slightly longer on average but was much more efficient for that, whilst the q-learning agent struck a better balance between resource efficiency and increased performance and the q-learning approach was ultimately able to outperform the default configurations in both resource efficiency and runtime.

While it could be argued that the q-learning agent is therefore the “better” option it is hard to say exactly what the right approach is because it also depends on the desires of the user. Many scientists may be prepared to accept slightly longer runtimes if the “greediness” of their workflows is decreased and they are able to free up resources for others to use, or reduce their own costs (if they are paying for CPU hours). Indeed from a financial perspective the gradient bandit wastes significantly less CPU hours than the extra time it takes to complete, which is fairly marginal. However in a situation where the user is paying for the entire system and not for resource usage it may make more sense from a financial perspective to use the q-learning agent approach since it reduces the amount of time for which the system needs to be up by reducing the overall execution time.

Chapter 6

Conclusion and Outlook

This thesis has looked at the use of reinforcement learning to size tasks in scientific workflows. It has provided background information on scientific workflows, scientific workflow managers, the gradient bandit problem, and q-learning in chapter 2. It has discussed related work in chapter 3 and then incorporated gradient bandits and q-learning into a popular workflow manager in chapter 4. The performance of this approach was then tested against that of the default configurations and that of a feedback-loop based approach. In the evaluation this thesis found that the results had vastly improved on the default configuration and performed similarly with the feedback loop approach for runtime performance whilst achieving greater CPU efficiency but worse memory efficiency.

This rest of this chapter briefly looks at what could be improved upon in section 6.1 before section 6.2 looks forwards and asks what implications this could have and what avenues might prove interesting for further research.

6.1 Improvements

Immediately the most obvious way in which these approaches could be improved upon would be to incorporate read and write calls into the reward function. When a thread reads or writes characters from or to a device it must perform a system call and wait for it to complete, and during this time the thread cannot use the CPU. If a program has a high degree of parallel operations but also has to read and write lots of characters then it would probably have lower CPU usage (because of the system calls) but simultaneously would benefit from being assigned more CPUs. However because of the nature of the bandit and the agent's reward functions it is unlikely that this task would be assigned more resources since read and write calls are not considered.

Moving along this train of thought, it could also be prudent to incorporate the size of a task's input file into the reward function for the memory bandit or the q-learning agent. A task's memory footprint is most likely quite closely coupled with the size of the input file and this could prove important information for a reinforcement learning agent.

A different issue which was relevant to both approaches was the length of time needed to train the agents. The gradient bandit was trained for about 40 runs and the q-learning agent took 90 runs before they were "tested" across the next 10 runs. For large scale workflows this would take a considerable amount of time. Of course the agents are able to learn on the go and the performance during the training time is not so much worse as to make the workflows unusable during that time, so it is hard to say how much of a problem this could be.

Specifically for the q-learning agent there is room for improvement by incorporating replay buffers and eligibility traces. These would enable the agent to update its value function for the states and actions which led it to its current state and this is could help to learn an optimal Q function faster. This is not done in the current implementation and could have a significant impact on performance. Additionally an epsilon of $\epsilon = 1/t$ could be used to decrease the agent's exploration proportional to the time. Additionally it may be more prudent to separate the memory assignments from the CPU assignments and instead co-locate a q-learning agent for CPU with one for memory. This is what was done for the gradient bandit and it would make sense for the q-learning agent too, the relationship between CPU and memory assignments can be split up as they are not hugely dependent on each other, and because it could then explore a larger state-action-space, and thus be trained faster or achieve better results by having more fine-grained allocation options.

Finally, considering the testing set-up it would be an improvement to use kubernetes clusters with different machines to more accurately simulate the scenario where a workflow is performed on a larger distributed system with different nodes (with potentially heterogeneous resources). Such a set-up may also change results as the cumulative resources of the system would be greater, which may allows the agent's decreased resource usage to increase throughput even more and thus speed up the workflows to a greater degree. It would also be better to use random inputs chosen from several different sets of possible inputs as opposed to repeating the workflows with the same input data. By training the agents using the same inputs there is a danger of overfitting the agents' resource assignments. This may be less of a danger for the CPU assignments, because the character of a task's CPU usage is less dependent on the size of the inputs, but repeatedly using the same inputs for an agent which sizes tasks for memory does present a real danger of overfitting. This is because the memory footprint of a task is much more dependent on the input to the task. This recommendation is not particularly relevant for those workflows which tend to have similarly sized inputs, but it could make a difference for workflows with different usage patterns.

6.2 Implications and Further Areas of Research

The obvious implications of these findings are that, much like some of the scheduling problems mentioned in chapter 3, this is also an area in which reinforcement learning has interesting potential. Additionally it has also shown that the resource configurations in scientific workflows have room for improvement and are another area which should be investigated further.

Nextflow and other scientific workflow managers represent a different class of actor in the interactions between user, application and execution platform. These workflow managers do not manage the execution platform, nor do they create the workflows which are executed. They are a kind of middle man and thus present a unique perspective on user-executor interactions. Normally the interactions in such environments are just between the user and the execution platform. The user has control over what process it wants to perform and how many resources to ask for, while the execution platform has control over when and how to execute the process. However in the context of a scientific workflow and a workflow manager, the manager sits in between the user and the execution platform and has no control over the tasks that must be executed nor over the execution. It is only capable of scheduling and sizing the tasks. This position requires a slightly different approach to common problems such as scheduling and resource assignment but it also presents an interesting avenue to research novel approaches to these familiar problems.

Now of the resources most important to both the process being executed and the user, the disk space must rank as one of the most important. Although it was not covered in this thesis, sizing a task's disk space is another area where intelligent resource management could prove useful. It could most likely be solved in a fashion similar to the approach used for memory and in fact it would be quite simple to co-locate a gradient bandit approach for disk space alongside the memory and CPU bandits. Incorporating this into a q-learning agent would be trickier since the agent already has to explore a large state-action space for just memory and CPU, and by adding disk space into this it could become a very long process to try and train such an agent. Ultimately it may prove better to have separate q-learning agents for the CPU and the memory/disk-space (since these two are more likely to be closely related).

If one considers the current trends in machine learning and reinforcement learning agents then what could also be an interesting avenue for research would be to use deep reinforcement-learning [16], which uses a deep neural network to learn a policy. This could allow for the kind of fine-grained memory assignments that made the feedback loop perform so well.

Following up on this idea, one more interesting area for further research would be incorporating a machine learning approach into a scientific workflow management systems' internal scheduler. Whilst the reinforcement learning agents used in this thesis had no knowledge of the other tasks and of their interdependencies in the workflow, a scheduler would have the full understanding of a workflow's digraph and could perhaps improve performance to an even greater degree by intelligently choosing when to assign a task slightly more or less resources than usual, depending on which tasks must wait for the current one to complete.

Appendix A

Example of a Task's Q-learning Agent

This appendix chapter contains a graphical representation of the q-learning agent for one of the tasks from the *metaboigniter* workflow. The task was not chosen for any particular reason except that it only occurs once per workflow run and was thus only trained 100 times (other tasks occur multiple times within the same workflow and are thus trained more often).

There are 4 graphs in A.1 the top left graph is a scatter plot which indicates what action the agent took at what point in time. The bottom left graph indicates what the agent's CPU and memory allocation (its state) was at that point in time. Meanwhile the top right graph shows the agent's cumulative rewards over time, as well as a straight line which represents what the cumulative reward would look like if the reward was consistent. It should be noted that because the reward function is negative, whenever the line's slope is steep the agent is performing poorly and receiving low rewards, whilst near-horizontal slopes show that the agent is doing well and limiting its punishment (receiving higher rewards). Finally the bottom right graph is a simple scatter plot of what rewards the agent received, with a constant line to show the average reward.

This figure is useful both to indicate that the agent is actually trying to learn favourable actions, as well as to represent what states the agent is exploring.

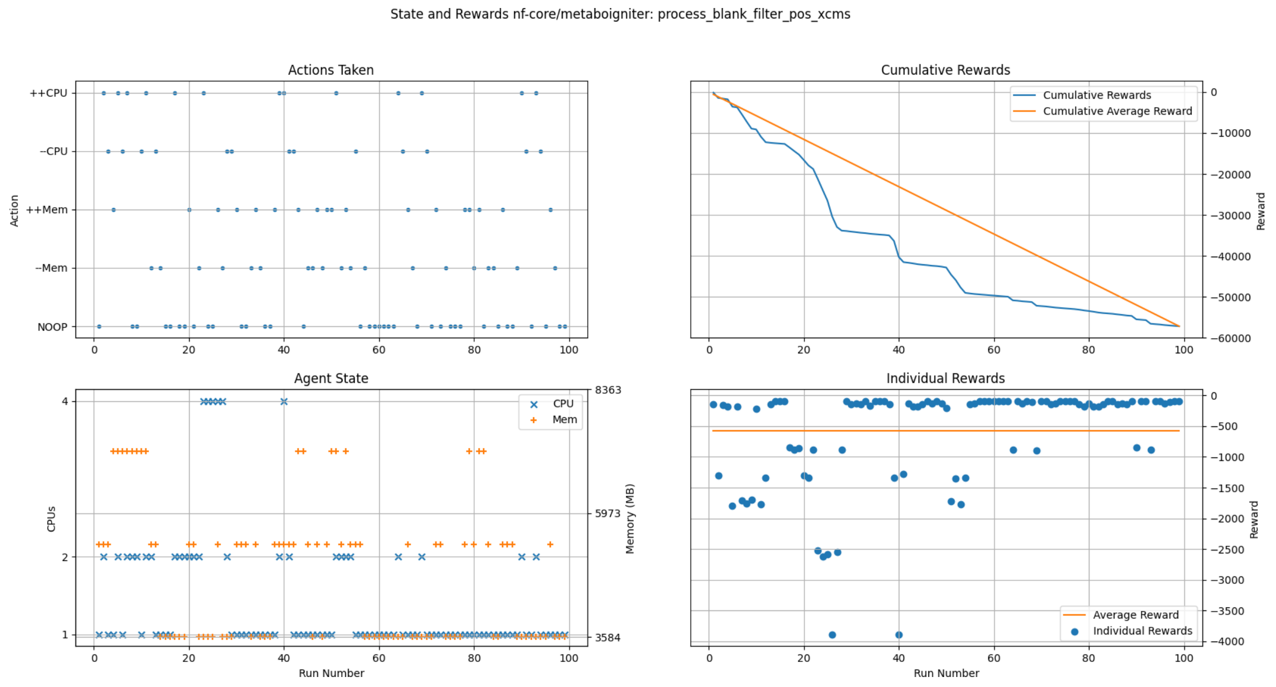


Figure A.1: Example of a Q-learning Agent

Bibliography

[1] Ilkay Altintas et al. "S04—introduction to scientific workflow management and the Kepler System". In: *Proceedings of the 2006 ACM/IEEE conference on Supercomputing - SC '06* (2006). DOI: 10.1145/1188455.1188669.

[2] Jonathan Bader et al. "Tarema: Adaptive Resource Allocation for Scalable Scientific Workflows in Heterogeneous Clusters". In: *arXiv preprint arXiv:2111.05167* (2021).

[3] Brendan Burns et al. "Borg, Omega, and kubernetes". In: *Communications of the ACM* 59.5 (2016), 50–57. DOI: 10.1145/2890784.

[4] Marc Bux and Ulf Leser. *Parallelization in Scientific Workflow Management Systems*. 2013. arXiv: 1303.7195 [cs.DC].

[5] Mukoe Cheong et al. "SCARL: attentive reinforcement learning-based scheduling in a multi-resource heterogeneous cluster". In: *IEEE Access* 7 (2019), pp. 153432–153444.

[6] Ewa Deelman et al. "Pegasus: A framework for mapping complex scientific workflows onto Distributed Systems". In: *Scientific Programming* 13.3 (2005), 219–237. DOI: 10.1155/2005/128026.

[7] Ewa Deelman et al. "Workflows and e-science: An overview of workflow system features and capabilities". In: *Future Generation Computer Systems* 25.5 (2009), 528–540. DOI: 10.1016/j.future.2008.06.012.

[8] Paolo Di Tommaso et al. "Nextflow enables reproducible computational workflows". In: *Nature Biotechnology* 35.4 (2017), 316–319. DOI: 10.1038/nbt.3820.

[9] Qin Ding, Cho-Jui Hsieh, and James Sharpnack. "An efficient algorithm for generalized linear bandit: Online stochastic gradient descent and thompson sampling". In: *International Conference on Artificial Intelligence and Statistics*. PMLR. 2021, pp. 1585–1593.

[10] Jianqing Fan et al. "A theoretical analysis of deep Q-learning". In: *Learning for Dynamics and Control*. PMLR. 2020, pp. 486–489.

[11] Yolanda Gil et al. "Examining the challenges of scientific workflows". In: *Computer* 40.12 (2007), 24–32. DOI: 10.1109/mc.2007.421.

[12] Robert Grandl et al. "Multi-resource packing for cluster schedulers". In: *ACM SIGCOMM Computer Communication Review* 44.4 (2014), pp. 455–466.

[13] Gideon Juve and Ewa Deelman. "Resource Provisioning Options for Large-Scale Scientific Workflows". In: *2008 IEEE Fourth International Conference on eScience*. 2008, pp. 608–613. DOI: 10.1109/eScience.2008.160.

[14] Chee Sun Liew et al. "Scientific Workflows: Moving Across Paradigms". In: *ACM Computing Surveys* 49.4 (Dec. 2016). ISSN: 0360-0300. DOI: 10.1145/3012429. URL: https://doi.org/10.1145/3012429.

[15] Hongzi Mao et al. "Resource management with deep reinforcement learning". In: *Proceedings of the 15th ACM workshop on hot topics in networks*. 2016, pp. 50–56.

[16] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *nature* 518.7540 (2015), pp. 529–533.

[17] A.W. Mu'alem and D.G. Feitelson. "Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling". In: *IEEE Transactions on Parallel and Distributed Systems* 12.6 (2001), pp. 529–543. DOI: 10.1109/71.932708.

[18] Jia Rao et al. "Vconf: a reinforcement learning approach to virtual machines auto-configuration". In: *Proceedings of the 6th international conference on Autonomic computing*. 2009, pp. 137–146.

[19] Matthew Shields. *Control- Versus Data-Driven Workflows*. Ed. by Ian J. Taylor et al. London: Springer London, 2007, pp. 167–173. ISBN: 978-1-84628-757-2. DOI: 10.1007/978-1-84628-757-2_11. URL: https://doi.org/10.1007/978-1-84628-757-2_11.

[20] Richard S. Sutton, Francis Bach, and Andrew G. Barto. *Reinforcement learning: An introduction*. MIT Press Ltd, 2018.

[21] Lauritz Thamsen et al. "Mary, Hugo, and Hugo*: Learning to schedule distributed data-parallel processing jobs on shared clusters". In: *Concurrency and Computation: Practice and Experience* 33.18 (2021), e5823.

[22] Benjamin Tovar et al. "A job sizing strategy for high-throughput scientific workflows". In: *IEEE Transactions on Parallel and Distributed Systems* 29.2 (2017), pp. 240–253.

[23] Vinod Kumar Vavilapalli et al. "Apache Hadoop Yarn". In: *Proceedings of the 4th annual Symposium on Cloud Computing* (2013). DOI: 10.1145/2523616.2523633.

[24] Christopher J. Watkins and Peter Dayan. "Q-learning". In: *Machine Learning* 8.3-4 (1992), 279–292. DOI: 10.1007/bf00992698.

[25] Carl Witt, Dennis Wagner, and Ulf Leser. "Feedback-Based Resource Allocation for Batch Scheduling of Scientific Workflows". In: *2019 International Conference on High Performance Computing Simulation (HPCS)*. 2019, pp. 761–768. DOI: 10.1109/HPCS48598.2019.9188055.

[26] Yu Xu et al. "Cost-efficient negotiation over multiple resources with reinforcement learning". In: *2017 IEEE/ACM 25th International Symposium on Quality of Service (IWQoS)*. IEEE. 2017, pp. 1–6.