A2C	Advantage Actor Critic
A3C	Asynchronous Advantage Actor Critic
CNN	Convolutional Neural Networks
CPU	Central Processing Unit
DQN	Deep Q-Network
DRL	Deep Reinforcement Learning
GPU	Graphical Processing Unit
IP	Internet Protocol
ML	Machine Learning
NN	Neural Network
RL	Reinforcement Learning
SGD	Stochastic Gradient Descent
SOTA	State of the art
SWA	Stochastic Weight Averaging
TCP	Transmission Control Protocol
TD	Temporal Difference
UDP	User Datagram Protocol

Contents

Part I

Distributed Reinforcement Learning Background

1 Reinforcement Learning

Reinforcement learning is the idea of being able to assign credit or blame to all the actions you took along the way while you were getting that reward signal.

– Jeff Dean

1.1 Introduction

Reinforcement Learning is considered the third subset of Machine Learning alongside supervised learning and unsupervised learning [1]. Reinforcement Learning's origins date back to the 1950s when researchers still referred to it as "trial-and-error learning" and focused on its development purely for engineering purposes [1]. It was not until the mid-1960 that the terminology "Reinforcement Learning" began to spread through the literature, and not until the 1980's that researchers achieved significant progress in the field with the discovery of temporal-difference learning and effective actor-critic methods.

1.2 Uniqueness of Reinforcement Learning

Reinforcement Learning is inherently different from both Supervised and Unsupervised learning, for it does not start from an initial "dataset" asset from which information can be extracted but instead revolves, in its most basic form, around an agent interacting in a specific environment [1]. Most peculiarly, this learning concept is fairly intuitive and closely related to our approach to learning as human beings and our interaction with the outside world [1]. When experiencing the world around us, we first interact with it with a particular action decided by us. We receive subsequent feedback and elaborate on it by associating the feedback received with the action that led to it. The more positive the feedback received, the more the specific action taken is "reinforced" and more likely to be taken in the future in a similar situation. Vice-versa was the feedback to be negative, and the opposite would naturally happen.

Reinforcement Learning founds its ideas in a conceptually similar way, with an agent that "lives" in an environment, experiences the different states of the environment, takes actions in order to interact with the environment, receives feedback (also called reward) as a result of every action, and gradually learns from all that information. It is not easy to provide a general

definition of RL that does not make assumptions about the precise form of the underlying structural elements that constitute it. However, we could refer to it as an "area of Machine Learning, that deals with sequential decision-making. A key aspect of RL is that an agent learns a good behavior. This means that it modifies or acquires new behaviors and skills incrementally." [2]. A definition such as this is relatively abstract. In this research, I have focused mainly on a narrower subset of the field, and I will now dissect the components to make them more concrete.

1.3 Ingredients of RL

1.3.1 Discrete Markov Decision Process

The first step towards concreteness is to formalize a model of the environment mathematically. In the vast majority of RL settings, a Discrete Markov Decision Process (MDP) is chosen as a representation, a model which was first widely explored in the early 1960s by Ronald Howard's 1960 book, "Dynamic Programming and Markov Processes" [3]. MDPs, for short, are discrete-time stochastic control processes, in this setting, useful when modeling environments where outcomes are partially the output of the action taken by the agent and partly by a random factor. Their structure is often represented by a tuple of four elements [4]:

$$(S,\ A,\ P_a(s, s'),\ R_a(s, s'))$$

S represents the State Space, the set of all possible states characterizing an environment. In other words, is the set of all scenarios an agent could encounter when interacting with the environment. A is the Action Space, potentially made by A_s elements, since some actions could be specific to some states and could be discrete as well as continuous. $P_a(s, s')$ is known as the State transition function, assigning a probability of going from state s to s' subject to taking action a while in state (it is an object that helps to deal with the stochasticity of the system). $R_a(s, s')$ instead is the Immediate reward function, assigning a reward coming from taking action a to go from state s to state s'.

The probabilistic nature of the state transition function serves as a generalization, as it is true that the transition function in many problems is deterministic, yet even in those types of problems it helps us introduce the "exploration vs exploitation tradeoff", which is an im-

portant topic discussed in a section of this chapter on its own.

1.3.2 Policy

Up to now, the mentioned machinery allows in a way to define the "rules of the game" by modeling and making assumptions about some characteristics of the environment. The next key mechanism delivers decision-making abilities to the agent, which can change throughout the interactions and captures the feedback obtained to improve and learn. In this field, a tool used to tell the agent how to behave is referred to as a policy. A policy is generally represented by π, and it aims at maximizing the expected stream of future discounted rewards that would come if that same decision-making process was to be maintained in future time steps [4]. More concretely, the policy can be seen as a function that maps s_t to a probability distribution over actions of the kind $P(a_t|s_t)$ through which the action to be taken next can be sampled from. For this research, I will consider only test cases using a discrete set of actions consistent across states. At this point, a question comes naturally. Should one sample from a distribution of possible actions, or perhaps is it better to choose always the actions maximizing the future streams of rewards? This question introduces the differences between on-policy and off-policy algorithms, which I will discuss more in detail in a later section.

But for now, in simple terms, if the algorithm followed the second heuristic blindly during training, its learning process would be slowed down quite a bit. That is because by being sure about a positive reward given by an action, the exploration of other actions would cease to exist, and possibly not all possible states would be explored. Using such algorithm in an applied setting would then lead to an "ignorant" policy, leading to sub-optimal decisions. That is why, in off-policy algorithms, the target policy is decoupled from the behavioral policy. Instead, in on-policy methods, usually this problem is solved with another approach, namely one that allows for non-greedy actions to be selected through an ϵ-greedy strategy, entropy regularization [5] or a draw from a distribution over actions.

1.3.3 Exploration vs Exploitation Tradeoff

The phenomena just explained is known as the exploration vs. exploitation tradeoff, and the quest for achieving the right balance of the two for optimal learning is a big topic in the field [1]. Many of the approaches used to tackle it inject some randomness while taking actions, avoiding the exploitation of the current policy found by the algorithm, which could impede

the exploration of the rest. One of the simplest and most common approaches of injecting some randomness in the actions taken by the agent is referred to as ϵ - Greedy approach, where the agent chooses exploitation with probability ϵ and exploration with probability 1 - ϵ. Notably, (particularly in off-policy) some other methods used to incentivize exploration are the Upper Confidence Bound [6], Boltzmann Exploration (a.k.a. Softmax Exploration) and Thompson Sampling [7].

1.3.4 V - Function

The policy up to this point has been defined as the function dictating the behavior of the agent in the environment, yet that tells us only part of the story. Other concepts need to be introduced to concretely get towards learning that policy, for example, some learned metric to measure how "valuable" a certain state is. This metric comes from deriving a Value Function from a specific policy π, which is nothing but a function evaluating the quality of a certain state, assuming we are committing to a certain policy π. In notation, we can express it with $V^{\pi}(s)$. In the aforementioned MDP setting we can write it as $V^{\pi}(s) = E_{\pi}\{R_t \mid s_t = s\} = E_{\pi}\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s\right\}$ [1]. As shown in the formula, the true value function based on a policy π considers all the future states visited while following that policy and their discounted rewards, and can therefore be considered a "forward-looking" measure, since only future states are considered in the evaluation while the past is ignored.

1.3.5 Q - Function

Although useful as a measure of "goodness" of a certain state, the V function in some cases might not be flexible enough, for instance when the transition function between states is not known. In that case, one would need to estimate some other function, taking into consideration the joint quality of taking action a at state s, which would allow ignorance about the transition function. Therefore if the Q function is in some sense a more detailed function that could yield better results, why is it not the only function estimated to learn a policy? The answer to that question lies in the dimensionality of the two since estimating $Q(s, a)$ with respect to $V(s)$ would be considerably more difficult proportionally to the dimensionality of a (the number of actions an agent can take could be substantially large, and potentially infinite in the case of continuous actions). In terms of efficiency with a lower sample or when dealing with a known transition function, learning $V(s)$ is preferable.

1.3.6 Relations between V and Q

It is, therefore, clear now that the V function evaluates a single state, whereas the Q function evaluates a state and a possible action available in that state jointly. At first glance they might seem somewhat related to one another, and in fact, these two fundamental RL measurements share a strong mutual nexus. The connection can be described as follows: $V^{\pi}(s) = \sum_{a \in A} \pi(a \mid s) * Q^{\pi}(s, a)$ [4]. This interesting fact has practical applications in training a Neural Network to approximate Q and V, especially in the case of Dueling Architectures (such as the famous Actor-Critic family of algorithms in RL). A team of pioneers in these approaches explained it best in their paper [8]: "Dueling Network Architectures for Deep Reinforcement Learning", one of their important results was the following "the advantage of the dueling architecture lies partly in its ability to learn the state-value function efficiently. With every update of the Q values in the dueling architecture, the value stream V is updated – this contrasts with the updates in a single-stream architecture where only the value for one of the actions is updated, the values for all other actions remain untouched".

1.3.7 Off-policy vs On-policy

Given this premise, a wide variety of algorithms has been studied to learn this decision-making function. That set of algorithms is partitioned into two subsets: Off-policy and On-policy algorithms. A relevant distinction can be made here: we can indicate the current policy as the policy used by the agent when making decisions and the learned policy as the one capturing the learning effect of the agent's experiences. If the two coincide, meaning the current policy is the same as the learned policy, the algorithm is referred to as on-policy, whereas if that is not the case, the algorithm can be labeled as off-policy. It is clear that given their functioning, on-policy algorithms generally lean more towards exploitation rather than exploration and vice versa, making off-policy learning more appropriate for scenarios in which a more substantial "creativity" of the algorithm is needed. Nevertheless, this comes at the burden of a slower convergence time since on-policy algorithms converge much faster to a solution [1] (with a higher associated probability of being stuck in a local minima). In any case, choosing one over the over is a matter of the specific problem at hand, and no set of algorithms dominates over the other in absolute terms.

1.3.8 Some other useful terms

Before going deeper into the nitty-gritty of the different algorithms on which this research has been based, some more notation needs to be introduced. I have mentioned how an agent at a given time t in state s takes action a, receives a reward r, and goes to another state s'. This whole process on behalf of the agent is called a step and, in tuple notation, can be written as (s_t, a_t, r_t). The agent is initialized at $t = 0$ and interacts with the environment until a finalization condition is met, represented by the termination of the episode. Furthermore, the time-indexed process of steps taken from an agent during the whole duration of the episode, is called trajectory and can be formalized as follows: $\tau = \{(s_t, a_t, r_t)\}_{t=0}^{T}$.

2 Computer networks & parallelization

Everybody who learns concurrency thinks they understand it, ends up finding mysterious races they thought weren't possible, and discovers that they didn't actually understand it yet after all.

– Herb Sutter

2.1 Motivations behind Distributed Reinforcement Learning

It is clear by now that the most powerful algorithms in the field run on machines computationally capable of handling the workload required to interact with the environment and optimize the parameters of the function approximator (Neural Network). It has also been shown by Mnih et Al.[5] that it is not only possible but also advantageous from different points of view to split the workload to utilize the hardware of a single machine fully. The argument they made for this claim is two-sided, and consists on one front of the full potential utilization of a machine (creating different threads to handle asynchronously and in parallel different agents living in different environments), and on the other exploiting the very same nature of A2C to decorrelates the gradient updates, leading to a better training of the Neural Network. While their proposed method displays the improvement of A2C by using a machine with 16 CPU cores, an extension to an even more significant number of cores comes to mind to achieve even more benefits from the parallelization of the work. Unfortunately, even the most advanced commercial machines eventually face an architectural upper bound on the maximum number of cores they can host due to size and heat generation. A solution to that problem could be building an interconnected cluster of several machines, yet that setting is far from being ordinary and thus unlikely to be used in training algorithms since GPUs at the same level of hardware specialization would offer superior performances. The framework I present in this work aims at building on top of the previous solutions proposed by researchers. It provides a novel framework using SWA, allowing DRL training using minimal computational resources (without being constrained to the architectural boundaries of hardware design) and allowing for massive scalability. In the following few sections, I will go over the relevant ingredients that combined constitute the flavor of this framework.

2.2 Multithreading & Multiprocessing

First and foremost the difference between multithreading and multiprocessing should be addressed, as they are both used extensively in different ways in the mentioned papers as well as in the architecture of this new framework I will propose. A process in computing is a set of instructions that are being executed by a computer. Therefore it can be thought of as a program or command that is running on the computer and using some of its available system resources. Whereas a thread is instead defined as the steps taken and the order in which the steps are performed by a running program. Every thread runs code from its starting location in a predefined sequence for a given set of inputs. Multiple threads can be used to ameliorate application performance by running different application tasks simultaneously. Since threads are therefore capable of handling in parallel smaller tasks defined by the process, the upper bound on the possible number of coexisting threads is naturally much higher than the one of simultaneous processes. Given that the framework has been fully written in Python, which is subject to the Global Interpreter Lock, a program's thread cannot access resources belonging to a different program. Hence the necessity of using multiple processes, instead of multiple threads, to handle the parallelization (sharing weights, exchanging gradients, sharing a replay memory).

2.3 Elements of Computer Networking

The most straightforward thought when more computational power is needed, yet the single machine has reached its limit, is to integrate a cooperative system of machines sharing the same protocol to propagate information across the system. Fortunately, the Internet provides us with every tool needed to work on that task. However, some more concepts more familiar to the world of Informatics and Network Protocols need to be introduced.

2.3.1 IP Addresses & Ports

An IP address is a collection of numbers that uniquely identifies a machine inside a network. They are divided into IPv4 and IPv6 [9], being different because of the number of bits they can use to express numbers (32 and 128, respectively). They are the building block of the system proposed in this research, since every machine involved in the system will need one,

functioning as a point of contact for the other nodes. A port, instead, is a number chosen by the user and working as an endpoint for a service, useful in case of contact between two machines. The possible port numbers are allocated using 16 bits, taking 65536 possible values. The first 5000 ports are generlly reserved by the operative system, and the rest is configurable for custom application without fear of overwriting default behavior. This concept, along with the IP, will reveal itself useful when instantiating a connection between devices to exchange information.

2.3.2 WAN & LAN

The difference between WAN and LAN defines the scope and speed of our network of computational nodes. The Wide Area Network (WAN) is the worldwide network of telecommunications where devices can share information on a global scale. In contrast, the Local Area Network (LAN) is a more protected environment and geographically bound environment, where devices are connected to the same physical location. It covers a narrower distance while simultaneously providing a higher speed of information transmission across the net (developments in technology have also seen the emergence of Wireless Local Area Networks, WLAN). The experiments provided in this research will focus on a WLAN setting, although little effort is required to change the settings to generalize it to WAN.

2.3.3 TCP & UDP

Having established the concept of IP addresses and the different Area Networks, the last key element allowing the whole communication process to work is a Transmission Protocol for sending and receiving the information. The two most used ones are the Transmission Control Protocol (TCP) [10] and the User Datagram Protocol (UDP) [11]. As it often happens, no protocol outperforms the other in absolute terms, but rather each has its pros and cons. UDP provides superior speed compared to TCP, as it lessens the overhead and metadata of the bytes passed through, also not requiring the acknowledgment of the connection on behalf of the receiver. The latter could cause some packets to be lost or arrive in a different order than the intended one. TCP, instead, is a slower yet more reliable protocol, which at the expense of overhead in the transmission metadata offers the stability of the connection, error checking and packet reordering (were they to arrive in a different order at the destination). In the experiments provided, the choice of TCP was motivated by the limited number of computa-

tional machines connected to the system and the low resource setting, making the time of information transmission derisory compared to the computational time of each device. This design choice also allowed for the implementation of a WebSocket protocol using TCP as its backend transmission protocol. Websockets fit the job because, after an initial handshake between the devices, they allow for the creation of finite messages that can be exchanged (instead of a continuous stream of bytes provided by raw TCP).

3 Literature Review & History of DRL

Research is to see what everybody else has seen, and to think what nobody else has thought.

– Albert Szent-Gyorgyi

3.1 Before Deep Reinforcement Learning

In their most basic and straightforward forms, Reinforcement Learning algorithms can be conceptualized as a table serving as a mapping from states to actions. This setting is, of course, rather intractable when scaled up due to the intractability coming from more complex environments and higher dimensional states and actions. The introduction of function approximators was a fundamental improvement in dealing with these problems [12] [1], especially for what concerns on-policy algorithms. Another breakthrough brought forward in the field came in 1992 when RJ Williams [13] introduced an important class of algorithms called REINFORCE, using backpropagation for their optimization. Another important paper published in the same year tackled a similar issue combining $TD(\lambda)$ and artificial neural nets to solve the game of backgammon [14]. These papers were, therefore, important documents laying the foundations of this field, providing the world respectively with a class of algorithms dealing with both immediate and delayed reinforced tasks and TD methods powered by some primitive forms of artificial neural networks.

3.2 Deep Q-Learning

While interest in the area was high, integrating Neural Networks as function approximators to function with RL objectives was not as straightforward. Different researchers published about this difficulty and made theoretically sound arguments highlighting the instability in results as an outcome of combining the two [15] [16], perhaps the most notorious work being published by Tsitsiklis & Roy in their paper "An Analysis of Temporal-Difference Learning with Function Approximation" [17]. After the failure of TD-Gammon methods in other settings (such as checkers and similar games), efforts in the field had been fading out under the false belief that RL methods had encountered their functional performance and stability limitations. Even if progress was slow over those same years, eventually, major improvements

came with the publishing of Playing Atari with Deep Reinforcement Learning by Mnih et Al. on behalf of DeepMind technologies [18]. Their work made it possible to melt the former congestion by integrating other advances in related fields, such as Convolutional Neural Networks in Computer Vision and Speech recognition and analysis. Their ground-breaking paper used a Deep version of Q-Learning with visual sensory inputs to solve seven Atari 2600 games with a similar input a human would have when playing the games. Their implementation proved to be a proper step forward, mastering several of the proposed games with their complex control policies, matching expert-level human performance on the same ones, and outperforming others using solely raw pixels as inputs. An important characteristic of the algorithm emphasized in their research is its sample-efficient. Given its Off Policy nature and the information retrieved by the agent interacting with the environment is policy-unrelated and can therefore be stored and reused for later updates using random sampling. The latter helps drastically in reducing the "correlation of updates", historical problem of DRL coming from the fact that given a trajectory of an agent its step-wise tuple components are likely to be sequentially correlated to one another, producing correlated gradient updates. In the case of RL, if the agent is facing for some time a particular situation in which the same action is reinforced over all the others sequentially, it might lead to an over-reinforcement and, thus, a worse learning experience. Although a significant step forward, the first version of this algorithm did not come flawless: DQN systematically overestimates the true Q-values because it uses a max function to choose the actions, which is conceptually sound but may lead to slower training and weaker policies in practical terms.

3.3 Gorila

Building on the DQN paper, another group of researchers sought to capitalize on the ground breaking algorithm and tried to come up with a method allowing for the parallelization of its training process. They aimed to reduce training wall-clock time while keeping (or even improving) the results obtained. The result of their research is now known as Gorila (Generalized Reinforcement Learning Architecture) presented in their paper "Massively Parallel Methods for Deep Reinforcement Learning" [19]. Gorila uses a distributed replay memory, allowing different agents to run in parallel on different machines, using a client-server structure for exchanging information with the computing client sending gradients, and the general server

backpropagating them. In turn, the researchers tested this newly devised architecture on 49 different Atari games using GPU-powered machinery for training. The results were promising: Gorila outperformed vanilla DQN in a large variety of Games (41 out of 49), setting a new standard for the DRL field, while the algorithms' training was highly parallelized, fulfilling its promises.

3.4 A3C

In 2016, another paper was published, bringing another important breeze of innovation into the world of DRL. The team of researchers representing Google DeepMind, captained by Mnih, developed an asynchronous method for Deep Reinforcement Learning, which they have decided to name "A3C", short for Asynchronous Advantage Actor-Critic [5]. Their work did not elaborate a new algorithm but rather a framework to allow Actor Critic algorithms to be trained on non-specialized hardware. The latter was achieved using a single device and exploiting the mechanisms of multithreading and multiprocessing. Their framework consisted in using a machine equipped with several CPU cores, allowing for the creation of different environments and different agents in each of the cores, each interacting in its isolated copy of the environment and having its own policy, which contributed to the training of a more general policy at the machine level. This approach might seem similar to the one proposed in the Gorila paper, used to power their "massively distributed" version of DQN. However, there are significant differences that the authors themselves point out. First of all, Mnih et Al.'s approach was, as mentioned, limited to a single machine and therefore subject to the impossibility of hosting a large number of cores used for computations. Even advanced customer hardware rarely has a number of cores larger than 16 because of architectural challenges in the design brought up by heat generation. Nevertheless, this limitation in the number of cores was compensated by the superior speed in information transmission, which is almost negligible within the machine, and the possibility of limiting the resources to CPUs, thus not using more specialized hardware.

Given the high probability of blocking with such a system concerning gradient accumulation, the authors deviated from the approach previously used in Gorila, which consisted of sequential gradient updates, and opted for asynchronous updates instead. This choice was backed by previous research in the field by Niu et Al. [20] that developed Hogwild!, a lock-free ap-

proach used to parallelize SGD assuming sparse optimization, achieving nearly optimal rates of convergence.

The results brought forward by the Google's team democratized the training of DRL algorithms with more modest resources, achieving the same level of testing performance as specialized hardware. Furthermore, Actor-Critic methods were able to set a new performance standard in the field by outperforming the competitors present back then.

3.5 IMPALA

Yet another improvement in the field of Distributed RL came in 2018 when Espeholt et Al. [21] revolutionized the distribution of work to computational nodes at the conceptual level. The result of their work has been named IMPALA, and provides a framework able to tackle the extensive amount of data produced by the agents and the extensive training time required. The conceptual shift at the very foundation of the project is the decoupling of learning and acting: in A3C we had computational nodes which hosted agents which shared gradients with the central server. In IMPALA the information shared between nodes and the central server does not consist of gradients but rather of trajectories of experience, after which a GPU is used in the central server for the "aggressively parallel" processing of the trajectories through mini-batches. The latter is implemented by using a novel Actor-Critic algorithm introduced in their paper: V-Trace. V-Trace uses Truncated Importance Sampling on previously computed temporal difference values, which resulted in a better algorithm in terms of data efficiency and performance. As it often happens, improvements on one side come at a cost on the other side. In this case, the downside consists in storing all the information of all processes on a single machine, which requires the RAM to be sufficiently large in order to accommodate all the data generated by all learners, mining the principle of scalability as the number of actors gets large. Not only that, but this approach requires, once again, powerful GPUs capable of crunching all the data coming from all the different nodes.

3.6 The insurgence of Distributed DRL

After the observable improvements brought in the field by the previous algorithms, more research has been done to study and to better understand the world of DRL. A particular

focus was placed on solving the two issues still troubling the field: the tradeoffs between sample efficiency, stability of the results, and computational simplicity. In even more recent years, similar frameworks to the one I bring forward have been proposed and are now the state of the art when it comes to distributed DRL.

Youjie et Al. in 2019 [22] for instance, have studied the latency time and effectiveness of these systems when it comes to network communication in the context of gradient accumulation, providing a new architecture software-hardware named "iSwitch". Their solution uses a UDP transmission protocol with a custom-made message structure and special hardware designed purposely for optimization through the experiments. Although their research is extensive and precise, their system provides every computational node with an NVIDIA Titan RTX GPU and ad-hoc hardware custom-made for the research, which is a very far setting from the focus on low-level resources provided in my research. Furthermore, Terhani et Al. [23] in 2021 have instead researched a Federated DRL System, motivated by the future projections of networks' efficiency in communication due to new technologies. Insofar as the communication, their approach is more similar to mine: their network's nodes exchange weights instead of training data in order to alleviate the memory strain on a central node, yet their work does not even mention some aspects like SWA, nor it poses as much emphasis on certain aspects such as the parallelization within the single node of the network, which plays a vital role to boost the speed and efficiency of the system as a whole, and poses as well some other interesting questions in the architectural phase.

3.7 Stochastic Weight Averaging

Another key component of my work is the extensive use of Stochastic Weight Averaging in the algorithm's convergence process. SWA is a technique introduced in the literature in 2018 by Izmailov et Al. [24], approach involving continuously weighted averaging the trajectories of SGD in weight space. Their paper shows that it leads to wider optima and thus an improved generalization of the final model at a modest added computational cost, which consists of barely taking the average of two sets of the Network's parameters. The primary purpose of my experiments will involve studying the behavior of the two parameters that regulate SWA, which will occur every time information is shared along the network of WebSockets. One of the first applications of this method to RL has been released in 2018, where researchers used

it to improve stability in A2C [21]. Their approach consisted of 5 different nodes sequentially averaging the parameters, which achieved the desired stability results in different test settings, such as several Atari games and CartPole. An even closer application in the context of SWA to my framework is SWAP [25], Stochastic Weight Averaging in Parallel, which is a similar procedure using parallel computations for the averaging of the set of weights, allowing for parallelization of the process. Although siblings, my framework will not make use of parallel procedures to carry out the computations but instead will focus on an asynchronous version of the latter to accommodate the machines' heterogeneous computational capabilities and various latencies in the parameters' transmission.

Part II

Framework Design and Experiments

4 Framework Architecture

As an architect, you design for the present, with an awareness of the past for a future which is essentially unknown.

– Norman Foster

4.1 Genkidama

With the rise in popularity of the area of DDRL (Distributed Deep Reinforcement Learning), in order to facilitate research and incentivize deployment in the industry, versatile and configurable software needs to be developed. This idea motivates the framework I propose, Genkidama, an open-source version of a distributed architecture exploiting SWA for the training of DRL algorithms in the form of a library entirely written in Python. To the best of my knowledge, no configurable setup nor study has been published tuning the SWA hyperparameters in a DRL context. While some other distributed DRL training tools are currently present, such as RLlib [26], moolib [27] and others, this library differs from them as it revolves entirely around the use of SWA for the training through a lightweight system of server-clients using WebSockets. In this section, I will go through the architectural choices that motivate the fabric of the library, strictly linking them to their role in the training process for the algorithms. The main modules in which it is divided are Sockets, Neural Networks Models, Optimizers, and Replay Buffer. These modules work in conjunction with other stand-alone scripts, the main ones being CoreOrchestrator and SingleCore, for defining the algorithms' body and behavior and handling the machine's multiprocessing. Following up on the latter, I will go through all the components in more detail, which will later be combined to compose the mechanics of the single machine and the network of multiple machines.

4.2 WebSockets

The module Sockets outlines critical components necessary for the transmission of data between the computational machines (clients) and the server, as well as the address and port of where they will be hosted. An initial class GeneralSocket defines the feature that both client and server need to handle with the same behavior, which is the ability to listen for a message from its counterpart and store it in a variable of its own. Since the object of the communi-

cation is going to be (bytes-encoded) parameters of a neural net, which can be quite heavy, the necessity arises for a transmission protocol capable of reordering the packets of data in case they arrive in a different order than the one in which they have been sent. The previous situation becomes more common as the weight of the parameters increases or the physical distance between the clients and the server is large [28]. I have identified the solution to that problem to be a TCP protocol implemented through a sock.SOCK_STREAM object provided by the socket python library since it allows for a high degree of maneuverability. As previously stated when introduced, the TCP protocol brings home more safety than the UDP protocol at the cost of transfer speed since the latter drops the metadata, but at the same time becomes more unsafe in its transmission and should be used in a more controlled environment [28]. The shift in configuration is relatively easy to make, and it is configurable directly through the library. From the initial common behavior, the classes responsible on the two fronts diverge in class Child and class Parent, each focusing on their specific tasks. The process starts from the initialization of some Children, which are told to listen in their corresponding IP addresses at a designated port. Since the communication from each Child is unique, and it is one towards the server, the WebSocket server is handled by the main process. Once all Children have been initialized, the common Parent is initialized by feeding it the addresses and ports of the computational Children; only now the first stage of the communication process begins. To make sure that a connection has been established after the initialization of a network, I have built another handshake protocol on top of the first one provided by TCP. At initialization, the Parent tries to handshake every client IP provided at its designated port, checking that the structure of the Neural Network used in both corresponds in order to avoid potential disagreement at a later stage. In order to handle multiple Children, the Parent node spawns multiple threads, each of which shares access to the common Neural Network instance and its parameters. The subsequent communication follows a continuous process of Encoding - Transmission - Decoding, where the task of encoding is a mapping from weight space to byte space of the kind $E : W \longrightarrow B$ and D, and naturally E is an invertible function, of which inverse describes the decoding $D = E^{-1}$. At each interaction, the current weights of the specific Child are sent to the Parent, which updates its current weights by considering the incoming and current set of weights. If we refer to the Parent's weights at time t as $W_{P,t}$ and similarly to the Child's weights as $W_{C,t}$, we have that $W_{P,t} = (1 - \alpha)\ W_{P,t-1} + \alpha\ W_{C,t-1}$. In turn, the parent responds by sending the entirety of its parameters to the Child, which are then

integrated in a similar fashion: $W_{C,t} = (1 - \beta)\ W_{C,t-1} + \beta\ W_{P,t-1}$.

4.3 Neural Network Models

This module helps create the Neural Networks that will be used as function approximators when training the DRL model. They should inherit from class GeneralNeuralNet the methods allowing it to interact directly in the procedures of encoding and decoding its parameters, and also torch.nn.Module for what concerns the more NN-specific behavior, such as architecture, forward passes, choosing the actions to be taken, choices of loss functions, and random initialization of weights.

4.4 Optimizers

Another interesting feature that I have decided to introduce is the possibility of creating a custom shared version of SOTA optimizers. This allows for the opportunity to share parameters and define ad-hoc behavior during the optimization process, in order to fully use and combine information coming from the frequent pull and push of parameters at the individual machine level. This is achieved by defining a class inheriting from torch.optim.Optimizer, choosing the appropriate optimizer and manipulating its self.param_groups attribute.

4.5 Replay Buffer

This module has been introduced to facilitate the implementation of several algorithms which make use of shared memory for later sampling batches for gradient updates. It is important to note that this Replay Buffer is not shared between server and clients but is contained in every single Child machine, although shared across processes, it is accessible by all CPU cores present in the machine, but not in the external network. Its structure is lightweight and consists of a simple tensor T_{CxNxM}, where C represents the dedicated number of cpu cores inside the machine for computational purposes. N instead, represents the size of each single replay buffer in each CPU core, and M the number of components describing a step of the agent in the environment, therefore including vectors of $(state, action, reward)$. This Tensor is once again shared across processes, distributing global read and local write access to each CPU core to its own “memory share”, thus avoiding the risk of simultaneous writes to the

same pointer in memory. It already comes with several functions allowing storage of new data and sampling methods with/without replacement from shared memory.

4.6 Single Machine

The behavior of the single machine is contained in the CoresOrchestrator and in the SingleCore scripts, and the A algorithm inspires it. A CoreOrchestrator object is first initialized by the machine, which captures the various parameters used in the process and initializes a Neural Network with weight values randomly initialized by a standard normal. The Orchestrator then considers the computational CPU resources allocated for the training and organizes the various CPU cores to each handle one in parallel. Each process (or interchangeably CPU core) creates its own copy of the environment and has its independent agent playing in it to gather data and perform the Network's gradient updates. In order to avoid heavy asymmetries in the learning process, I have decided to implement a coordination system using semaphores, which limits the full efficiency of the parallelization process, but makes the system more manageable as communication with other machines is later introduced. This system consists of a semaphore placed at several key points of the training process, namely at the very beginning and subsequently at every episode's end. In more concrete terms, the semaphore is nothing but a vector S of booleans shared in memory, which starts as being completely shut down. As the i-th CPU core fires up the engines (in the case of the start) or has finished playing its current episode, its respective element activates, becoming a 1. Only subsequently, every process waits until all the elements of the vector have been activated (i.e., when all other CPU cores have finished their current task). It is important to note that every process shares its copy of the Neural Network, and every *batch_size* steps it performs an optimization step of the orchestrator network using the batch of collected data. It only at *batch_size * 2* steps then proceeds to pull back the Orchestrator's weights to the local ones. The weight are then sent to the Parent node, updated through a weighted average with strength equal to a configurable parameter alpha. Once pushed, the Child pulls back the parameters from the Parent node, using another configurable parameter beta which acts in the weighted average. Lastly, a topic deserving some words is the way the optimization process works. Since weighted updates of the Orchestrator Network are asynchronously coming from each process, and the processes are upper bounded by the number of CPU cores, I thought it was not only possible but also

suitable to implement Hogwild! style updates. They consist of shared simultaneous write access in updating the Orchestrator's weights, with no waiting time whatsoever is involved since no blocking happens. Although the theoretical validity of Hogwild! to the day assumes sparse weight updates, empirical results have shown the algorithm to perform just as well in RL settings [5].

Algorithm 1: Algorithm for the single machine (single episode)

Initialize process step counter $t = 0, T = 0$
Initialize a Neural Network O for the orchestrator with weights θ_O
Initialize a Neural Network C for the considered CPU core with weights θ_C
Reset gradients of O and C
while $T < T_{max}$ **do**
    $t_{start} = t$
    Get state s_t
    Initialize an empty buffer B
    while $t \,\%\, batch\_size \mathrel{!}= 0$ *or not terminal* s_t **do**
        Random sample $\phi \sim U(0, 1)$
        if $\phi > \epsilon$ **then**
            Take $a_t \sim \pi_{\theta_L}(a_t|s_t)$
        else
            $a_t \sim U(\{A_t\})$
        Observe r_t and s_{t+1}
        Store (a_t, r_t, s_{t+1}) in B
        $t \longleftarrow t + 1$
        $T \longleftarrow T + 1$
    $$\mathrm{R} = \begin{cases} 0 & \text{if } s_t \text{ terminal} \\ \mathrm{V}_{\theta_C}(s_t) & \text{otherwise} \end{cases}$$
    for $i \in \{len(B), \ldots, 0\}$ **do**
        $R \longleftarrow r_i + \gamma R$
        $B[i][r_i] \longleftarrow R$
    Perform batch-update of θ_O using B
    if $t \,\%\, batch\_size * 2 == 0$ *or terminal* s_t **then**
        $\theta_C \longleftarrow \theta_O$

4.7 Network of Multiple Machines

Thus far, I have discussed the functioning of individual machines. Yet, this same algorithm has already been implemented with successful results: my addition to the topic is the extension to a minimal resource setting using a network of computational nodes. The protocol for exchanging information, previously explained with higher generality in section SOCKET SECTION, consists of initialization by a central node (Parent) of a Neural Network equal in architecture to the others.

As a common denominator, the update of the Network is controlled by different threads spawned by the primary process in the Parent machine. Therefore, when receiving the new information (whether gradients or weights), it incorporates them and proceeds to propagate them back to the single machines and implemented in a cascade fashion to be both the Orchestrator and subsequently to the CPU cores' weights. Unlike the behavior of the single machine, the threads on the Parent node do not use Hogwild! for two main reasons: first and foremost, the object here is the set of weights, not anymore the accumulated gradients, and secondly, since they are exchanged at every episode, it is unlikely that the updates will be sparse, a characteristic that would mine the efficacy of convergence.

Algorithm 2: Algorithm for the WebSocket network of Machines

```
Initialize a Parent Neural Network N_P with parameters θ_P
Initialize in every worker a Child Neural Network N_C with parameters θ_C = θ_P
for Child i do
    Initialize a Websocket and listen at some port p
    Wait for handshake with the Parent
while Socket connection is active do
    for Child i, asynchronously do
        Play 1 episode using the above Algorithm for the single machine
        θ_P ←(1 − α) ∗ θ_P + α θ_{C_i}
        θ_{C_i} ←(1 − β) ∗ θ_{C_i} + β θ_P
    ; if Child i sends end message then
        Parent terminates socket connection with Child
```

A pictorial representation perhaps can better explain the flow of the Neural Network's weights from both fronts, Parent and Children.

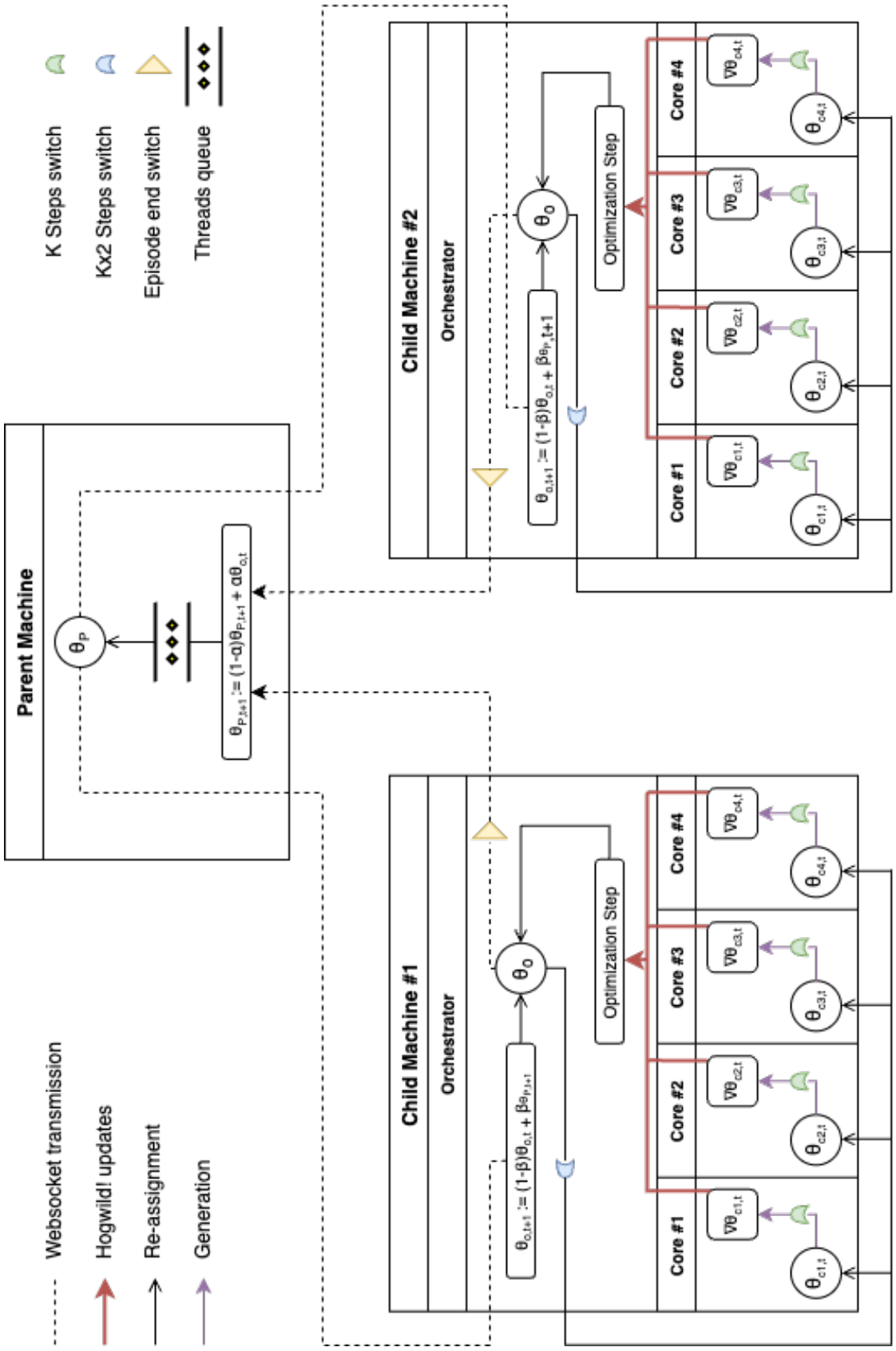

Figure 4.1: Flow of NN weights during training, both within the single Child and between Parent and Children. This example shown is precisely the one used in the experiments, using two computational nodes with 4 CPU cores each.

5 Experiments

Invention requires two things: One, the ability to try a lot of experiments, and two, not having to live with the collateral damage of failed experiments.

– Andy Jessy

5.1 Testing Environment

To test the capabilities and performances of this new tool, I have decided to test it against one of the most classical Reinforcement Learning environments, which has been cherry-picked due to its characteristics. Since most of the experimentation of Genkidama is going to take place in extremely low-resources machines (a cluster of Raspberry Pis, which are equipped with low-performance 4 cores CPUs), I have decided to consider an environment that is not CNN based. In fact, to solve most games nowadays, a Convolutional Neural Network is used as a feature extractor, and then its outputs are fed as inputs to the Reinforcement Learning algorithm. My approach will instead tackle Lunar Lander v2 [29], a game developed by OpenAI for its Gym, which consists of a 2D spaceship that aims to land safely in a lunar environment. The lander has access to thrusters, which are subject to noise but can be fired by consuming fuel to stabilize and direct the lander. The state is composed of 8 components:

i. Horizontal position in screen coordinates
ii. Vertical position in screen coordinates
iii. Horizontal velocity
iv. Vertical velocity
v. Angle
vi. Angular velocity
vii. Left leg contact to the ground (boolean)
viii. Right leg contact to the ground (boolean)

Whereas the action space is composed of four possible actions:

i. Do nothing

ii. Fire main engine located at the bottom of the ship

iii. Fire left engine

iv. Fire right engine

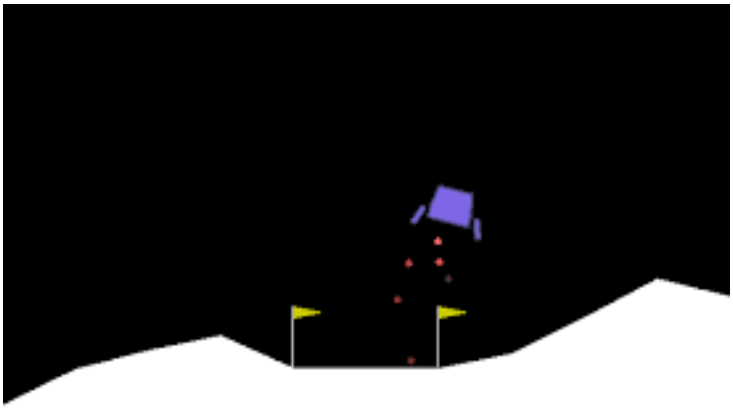

Figure 5.1: A frame of the game LunarLanderv2 by OpenAI.

The initial state of the lander is always at the center of the screen, at coordinates $(0, 0)$ and random initial velocity. The reward system for this game consists in attributing a penalty for firing engines (thus consuming fuel) of -0.3 for each frame. In contrast, the reward is +10 for each leg that touches the ground, and the final reward for landing can be either be -100 for crashing, 0 for landing outsize of the flag zone, or +100 for a perfect landing in between the flags.

5.2 Solving LunarLander with A3C

I will try to implement a more classical version of the A3C algorithm with some tweaks in order to accommodate an ϵ-greedy exploration strategy (not common in on-policy algorithms such as A3C, yet helps here in preventing overfitting without compromising the nature of the algorithm, being just 1 in 100 actions random). The next step consists in fine-tuning the hyperparameters of the model, which in the majority of the circumstances requires a familiarity with the environment and some good old trial and error. As typical in the world of Reinforcement Learning, the hyperparameters to tune are many, and often the algorithm's

performance is highly susceptible to their correct tuning. In this first section, I will analyze the hyperparameters of A3C when solving LunarLanderv2. Specifically, I will look at the batch size used to make the Network's updates and the parameter γ which is the reward discount factor.

5.2.1 Batch size

Batch size is a hyperparameter defining the number of samples to be used when making gradient updates. In a more general definition, it controls the "error" the algorithm is making when computing the gradient in the training process. If the batch size is extremely small and pushed to the extreme, for example, one sample at a time, the updates will be extremely noisy. If, instead, the batch size is extremely large, as large as the whole dataset to be precise, it would take us back to normal gradient descent. Now, these are general definitions that do not apply that well to the optimization realm of on-policy algorithms. First of all because one does not have a fixed dataset but instead it is changing continuously as the agent experiences more of the environment. Secondly becondly because by being on-policy the algorithm should learn the current policy, making experience replay unfeasible and still an open research field in the context of sample efficiency. Since sample efficiency is extremely low in A3C, it is possible to perform a batch update only after having gathered the information, which are then thrown away. Given the characteristics of the environment, it is clear that the most important feedback is awarded for a correct landing between the flag. The latter needs to be emphasized heavily since the episode could run for as many as 1000 steps. Intuitively, the idea should be that the agent should think of the landing phase well before the lander is close to the ground to prepare for a softer and more precise landing. I will fix the reward to be $\gamma = 0.999$ and proceed with testing batch sizes of $30, 60, 120$.

As later shown in the results (5.2 and respective table), the algorithm learns better with larger batch sizes. Although the difference between a batch size of 60 and 120 is almost unnoticeable from the graphs, the convergence of the 120-sized batch leads to somewhat more stable results when comparing the two final distributions of rewards at the last epoch. The increase in batch size also has an important implication regarding the algorithm's training time since an increase in batch size corresponds to a linear reduction in computational time. Again, this fact comes logically from the observation that batch size and frequency of gradient updates in on-policy (sample inefficient) algorithms coincide perfectly. By drawing some

conclusions in this section, I can state that, ceteris paribus, the best size for the batch updates amongst the considered ones in 120, both in terms of testing efficacy and in training efficiency.

5.2.2 Discount factor γ

The discount factor is an omnipresent hyperparameter in reinforcement learning since it is embedded into both the fundamental values that need to be estimated, the V-function and the Q-function [4] (as shown below).

$$Q_\pi(s,a) = \mathbb{E}_\pi\left[G_t \mid S_t = s, A_t = a\right] = \mathbb{E}_\pi\left[\sum_{j=0}^{T} \gamma^j r_{t+j+1} \mid S_t = s, A_t = a\right]$$

$$V_\pi(s) = \mathbb{E}_\pi\left[G_t \mid S_t = s\right] = \mathbb{E}_\pi\left[\sum_{j=0}^{T} \gamma^j r_{t+j+1} \mid S_t = s\right]$$

Since $\gamma \in [0,1]$, the quantities above converge to a finite value. Intuitively γ can be considered a measure of importance of future states when considering the quality/value of a single state. In these experiments, I will consider three different levels of gamma: $0.95, 0.99, 0.999$, which might not seem too far off from one another, but in the context of discounting a batch size of 120 steps, it could mean weighting the last value for significantly different values. In this specific context, the values chosen for these hyperparameters are high, justified by the idea that the most meaningful reward comes at the end, and ideally, its effect on the learning process should be highly emphasized.

As noticeable from the analysis on γ, the initial intuition of raising the value towards 1 is correct. As γ is lowered, the learning process becomes progressively more unstable, eventually leading to a less stable output in the .99 case and terrible results in the case of .95. To see this in perspective let's assume we are running an episode of Lunar Lander, and the lander performs poorly, crashing and thus getting a reward of -100. In the process of actualizing the rewards, since a batch size of 120 is considered, we would be taking into consideration the learned information coming from the crash at the following level of importance.

γ	0.95	0.99	0.999
$-100 * \gamma^{120}$	-0.212	-29.939	-88.687

Table 5.1: Actualized values of different levels of γ 120 steps in the future.

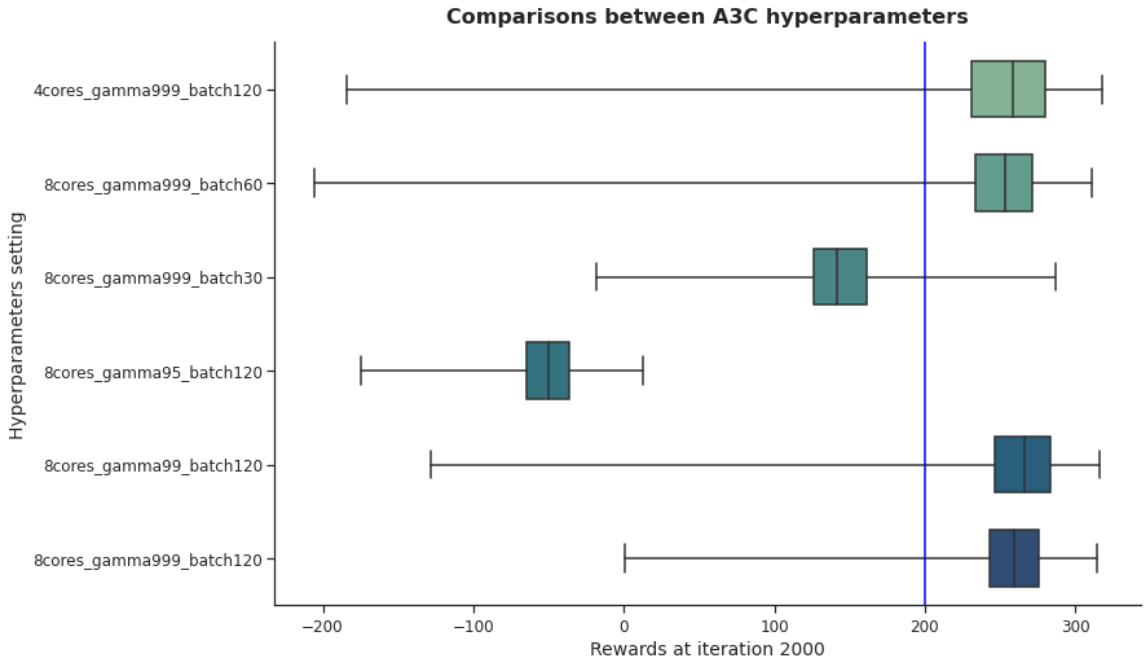

Figure 5.2: Plotting of the performance of different sets of A3C hyperparameters at the 2000^{th} run of the algorithm, playing LunarLander-v2 (500 evaluation episodes).

As highlighted in the plot and conjectured above, the algorithm in this setting is susceptible to the discounting parameter γ, where even a small reduction from .99 to .95 leads to a catastrophic drop in performance. The correctly tuned version of the algorithm running on only four cores seems to perform similarly to the tuned one with eight cores, yet presents a visibly higher variability in producing the results, thus is to be considered less stable. Table 5.2 summarizes the results of this first round of experiments in a pictorial representation, using all the data coming from each run after the algorithm's convergence. I will summarize them also numerical form below, given the importance that this run will later play in comparison with its distributed Genkidama counterpart.

	μ	σ
8 cores, gamma .999, batch 120	**255.9**	**36.9**
8 cores, gamma .99, batch 120	254.0	59.7
8 cores, gamma .99, batch 60	243.7	58.0
4 cores, gamma .999, batch 120	230.0	87.1
8 cores, gamma .99, batch 30	149.1	45.0
8 cores, gamma .95, batch 120	-51.3	21.3

μ is mean, σ is standard deviation

Table 5.2: Results derived from using the last checkpoint of each model and playing 500 episodes each to gather statistical results.

The model with the best joint performance in terms of both mean and standard deviation comes from the eight cores machine with $\gamma = 0.999$ and a batch size of 120. The exciting comparison comes when looking at the performance of the four cores one, which is able to solve the game, yet carries almost twice the standard deviation of its better-performing sibling. The golden question now becomes whether Genkidama performs worse, just as good, or perhaps better than its non-distributed cousins, a question which I am going to explore in the following sections.

5.3 Solving Lunar Lander with Genkidama

In order to test this framework against the more classical version of the algorithm, I am going to consider the best hyperparameters coming from the experiments above as a baseline. In this section, I will focus on a study involving the two central parameters involved in the Stochastic Weight Averaging procedure, taking place while the Children exchange weights with the Parent and vice-versa. While the testing above was carried out using a single multi-core machine, this part of the test will consider computationally inferior hardware and will spawn only four processes in each. Specifically, the hardware consists of two Raspberry Pis (model 3B+), and each of the two will spawn a WebSocket connection to communicate externally. At the same time, four parallel internal processes will handle the policy updates and

update the network's weights.
Given the structure of the library, which is meant to live in both the world of industry and research, I have implemented a system of semaphores in order to coordinate the different cores within the same machine, but I have deliberately chosen not to implement one at the Sockets' level. This decision comes from the fact that given a particular CPU, its cores are likely to be homogeneous among themselves, sharing a similar (if not exactly equal) efficiency in computation. This is not true when it comes to a network of communicating machines, where a coordination semaphore would have likely implied slowing down every machine at the speed of the lowest one, to guarantee the same amount of updates from every connected machine. This architectural choice will have implications in the experiments, insofar as one of the RaspberryPis has been subject to more usage previous to the experiments, which in turn is going to result into slower computations and less frequent weights updates (hence a heavily imbalanced contribution in the whole process).

5.3.1 Alpha & Beta

As intuitively one would think, lowering α, the parameter behind the strength of "absorption" of the weights from the central node, would result in slower learning. This effect will logically lead to modest adjustments of the model's parameters, where the set of weights θ_{t+1} of the neural network presents only a marginal movement past θ_t. Not only that, but pushing the boundaries of testing further, I will analyze different levels of α and β jointly to observe their behavior, as described in the previous section. The comparisons I will consider will focus initially on three choices of the two parameters, namely the values in the sets $P_\alpha = \{0.1, 0.5, 0.9\}$ and $P_\beta = \{0, 0.5, 0.9\}$, and I'm going to jointly grid search them, meaning a complete learning run composed of 2000 weights updates is going to be carried out for each tuple $\{(\alpha_i, \beta_j)\}_{i \in P_\alpha, j \in P_\beta}$.

The following few pages show, therefore, the results obtained by the overall nine runs of the algorithm in a tabular fashion. The results include 500 episode runs of a given checkpoint of the model saved on the go during the training process. The resulting mean μ and the associated standard deviation σ are analyzed since the goal is to learn about the whole learning process and not only the final results.

Number of updates	α	0.1					
	β	**0**		0.5		0.9	
		μ	σ	μ	σ	μ	σ
100		-112.4	46.0	-141.5	68.9	-148.5	78.2
200		-102.1	44.9	-124.1	46.7	-135.7	63.1
300		-86.0	41.6	-116.3	48.4	-126.5	57.5
400		-73.4	49.5	-109.0	51.2	-124.8	56.2
500		-53.0	49.3	-110.8	47.7	-119.9	50.8
600		-46.8	55.9	-102.9	48.6	-119.8	60.3
700		-27.5	58.9	-101.3	52.5	-114.8	51.4
800		-10.5	52.9	-97.9	48.9	-110.0	54.7
900		4.4	54.4	-95.5	56.7	-106.1	50.9
1000		16.2	64.5	-81.7	49.3	-100.2	47.3
1100		33.4	63.6	-75.4	51.3	-99.6	50.9
1200		46.2	57.1	-72.7	56.7	-94.8	53.4
1300		177.1	114.9	-72.1	62.3	-96.6	55.7
1400		175.4	113.2	-58.1	64.4	-88.8	49.8
1500		215.2	86.3	-44.0	55.0	-84.6	54.4
1600		221.3	76.6	-42.3	60.6	-82.9	52.2
1700		197.4	102.3	-35.4	60.0	-76.6	53.1
1800		225.3	88.5	-29.3	61.9	-77.6	54.3
1900		245.6	67.0	-21.8	59.6	-75.3	53.5
2000		248.6	59.1	-18.4	59.5	-70.5	56.7

μ is mean, σ is standard deviation

Table 5.3: Table showing the learning paths of Genkidama at different levels of alpha and beta.

Number of updates	α	0.5					
	β	0		**0.5**		0.9	
		μ	σ	μ	σ	μ	σ
100		-113.3	41.6	-123.8	48.8	-124.9	50.7
200		-100.4	53.0	-108.5	47.3	-112.5	48.0
300		-53.7	57.7	-99.9	43.8	-104.9	50.1
400		-37.4	47.9	-81.5	57.2	-93.0	40.6
500		-6.9	48.9	-56.8	63.0	-71.8	49.9
600		-9.3	56.5	-38.0	63.9	-54.6	48.5
700		-10.1	58.2	-21.6	62.3	-35.6	54.0
800		23.0	50.5	-14.4	62.9	-26.1	59.9
900		53.6	53.2	6.2	49.3	-20.5	60.2
1000		62.5	49.2	6.8	53.8	-17.2	62.4
1100		92.3	56.6	18.6	50.4	5.9	53.6
1200		95.7	60.8	47.1	44.9	14.5	53.0
1300		95.9	43.0	60.3	50.1	28.0	52.9
1400		129.6	41.4	86.3	48.7	45.9	62.8
1500		110.2	93.8	96.8	46.3	49.0	68.2
1600		126.3	38.3	102.3	44.6	75.3	71.9
1700		109.9	126.8	116.0	47.9	101.0	87.6
1800		153.0	83.1	111.0	40.9	186.8	108.3
1900		168.6	70.8	119.2	40.0	197.3	98.4
2000		102.4	124.1	120.5	41.3	209.8	94.7

μ is mean, σ is standard deviation

Table 5.4: Table showing the learning paths of Genkidama at different levels of alpha and beta.

Number of updates	α	0.9					
	β	**0**		0.5		0.9	
		μ	σ	μ	σ	μ	σ
100		-123.4	47.9	-123.0	58.9	-113.7	55.5
200		-111.7	41.0	-97.9	57.0	-97.9	47.4
300		-96.0	43.7	-86.0	59.9	-70.3	47.2
400		-46.2	57.0	-57.5	65.9	-46.2	55.9
500		-37.0	58.3	-33.6	64.6	-19.7	50.2
600		-32.5	72.3	-7.3	59.7	-4.3	50.5
700		20.3	36.4	-8.4	60.8	8.2	47.7
800		-0.2	57.1	20.4	46.6	15.7	46.4
900		64.5	54.1	32.5	53.8	35.5	47.4
1000		10.9	53.8	49.7	63.0	50.6	51.3
1100		183.1	103.2	77.3	61.0	72.0	51.5
1200		201.9	94.8	89.2	66.4	93.5	51.1
1300		32.8	54.7	108.0	51.3	104.9	54.5
1400		226.7	62.9	181.2	105.4	112.3	46.6
1500		243.3	73.6	205.6	97.5	117.5	42.2
1600		237.9	70.4	218.4	94.3	217.8	79.3
1700		247.9	63.7	220.4	88.5	224.3	74.7
1800		247.3	67.1	223.5	92.5	221.8	78.0
1900		241.6	78.5	225.3	85.8	223.8	89.0
2000		257.2	39.9	237.2	71.7	231.8	78.9

μ is mean, σ is standard deviation

Table 5.5: Table showing the learning paths of Genkidama at different levels of alpha and beta.

As the evidence suggests, the framework seems to be working correctly and in respect of the basic logic of convergence in the training of a NN. Given the definition of "solution" for LunarLanderv2, there seems to be at least one combination of parameters for each level of alpha that can solve the environment, at times, quite consistently (for instance, in the case of $(\alpha = .1, \beta = 0)$. In that case, with β being set to 0, there is no "pulling back" of weights on behalf of the Children node. In other words, they are left to their natural descent and the weights $\theta_{P,t}$ are continuously attracted by $\theta_{C,t+1}$. In the graph below, the data is shown through a series of box and whiskers plots. More than half of the combinations of (α, β) achieve a comparable solution in terms of solving the game by its definition (mean > 200, or in pictorial terms to the right of the blue line), averaging results close to 250. Only the combination $(\alpha = 0.9, \beta = 0)$ leads to a relatively better "worst case scenario".

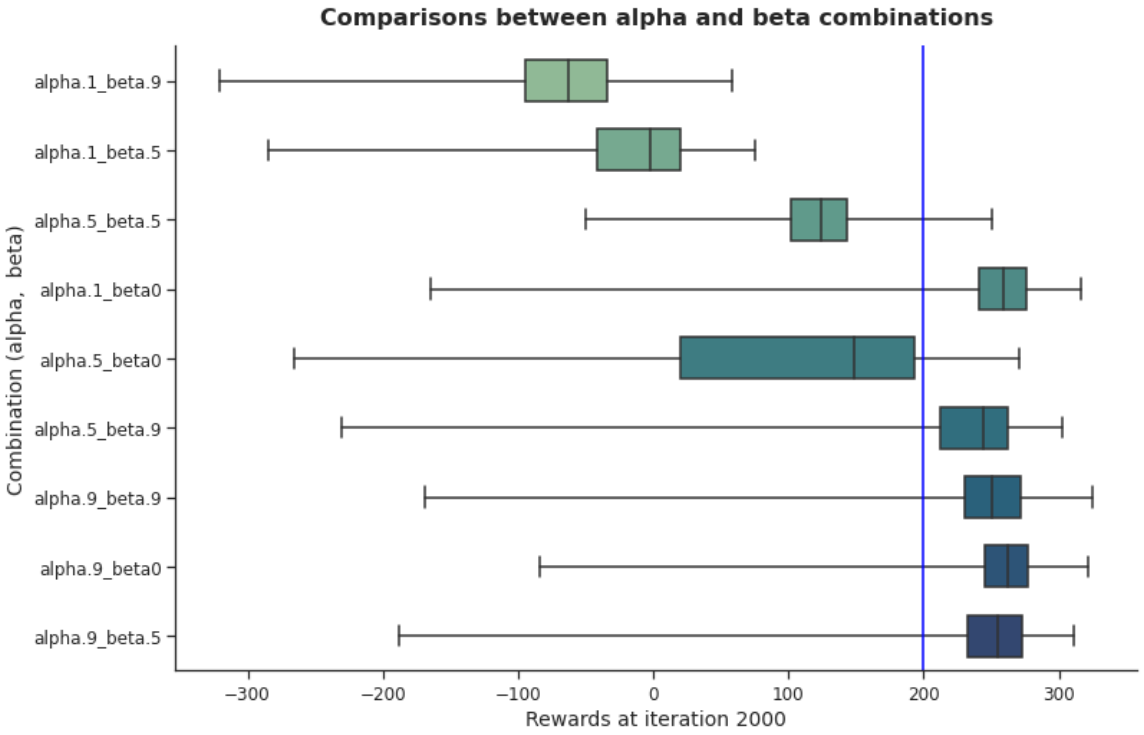

Figure 5.3: Plotting of the Genkidama results at the 2000^{th} run of the algorithm, playing LunarLander-v2 (500 evaluation episodes).

A clear pattern is noticeable in the levels of $\alpha = 0.1$, where the increase of β causes the results to worsen. This first fact has intuitive meaning since α guarantees slow learning on behalf of the Parent's parameters, whereas β regulates the strength that keeps the Child parameters close to the Parent ones. The combination of low alpha levels and high levels of betas signifies slow learning for both Parent and Children.

Besides this, the results are not crystal clear for high values of α, where further experi-

mentation is needed to determine its behavior, although they all seem to get to a solution of the environment.

5.4 A3C vs Genkidama

Having established that each succeeds at solving the game, it is now time to make a comparison to understand where they stand with respect to one another in terms of performance, defined jointly by the mean as the average reward and the standard deviation as the stability attained. With some statistical imagination, we can think of the cumulative reward coming from an episode being a draw from the distribution of possible rewards, given a game G with setting S, a model M, and its weights θ. This twist allows to write $r_{G_S,M_\theta} \sim P(R \mid G_S, M_\theta)$, and by having diffent weights coming from different learning approaches (i.e. the different algorithms) I can sample rewards from them and plot them as a density as such:

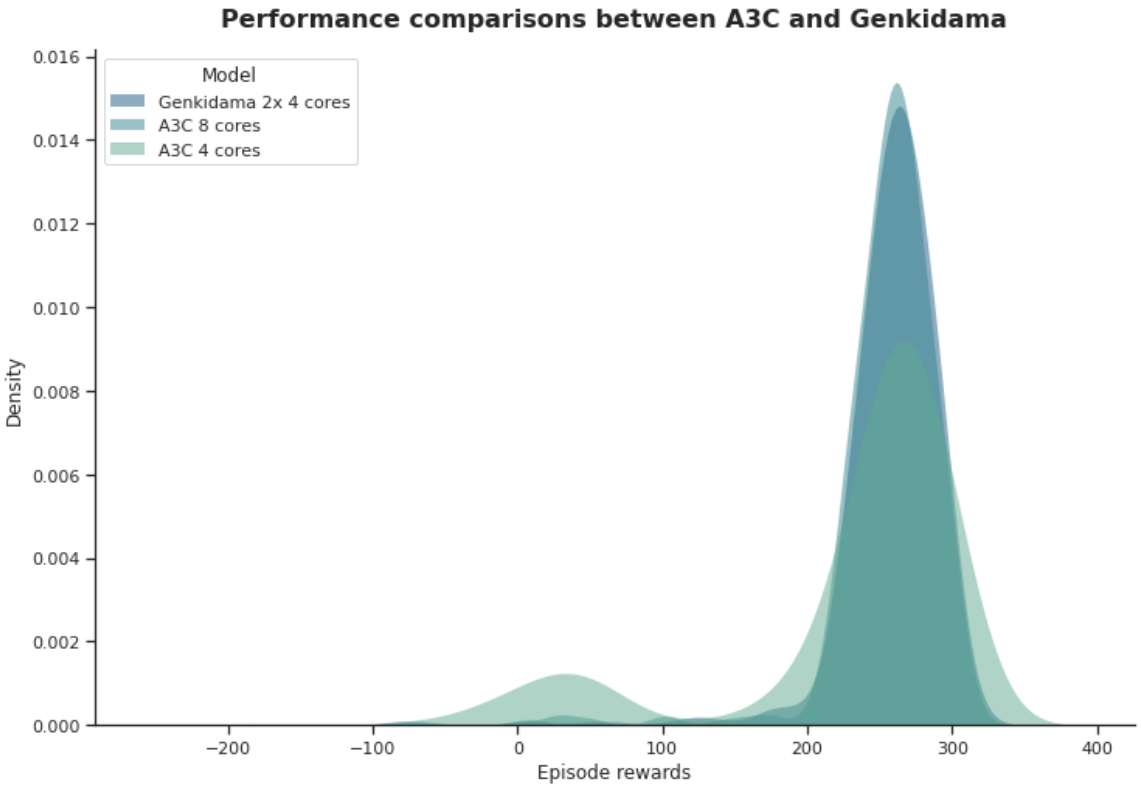

Figure 5.4: Plotting the distribution of 500 sampled rewards from A3C and Genkidama.

It clearly emerges that the distributions generated by A3C with 8 cores (the best performing model of the A3C set) and a 2x4 cores Genkidama ($\alpha = 0.9, \beta = 0$) almost coincide. This has a significant meaning, for the same result achieved using a superior 8-cores chip Apple M1 has been achieved by coordinating two utterly inferior hardware through Genkidama. Of

course, this pictorial representation of the densities gives us the first intuition of the claims, I will hence back them up with a table of the precise results that truly matter when discussing outcomes:

	μ	σ
Genkidama 2x4 cores	257.2	39.9
A3C 8 cores	255.9	36.9
A3C 4 cores	230.0	87.1

μ is mean, σ is standard deviation

Table 5.6: Results derived from using the last checkpoint of each model and playing 500 episodes each to gather statistical results.

Table 5.6 presents the evidence supporting the superiority of Genkidama 2x4 cores and A3C 8 cores with respect to A3C 4 cores. The first two are characterized by a significant improvement on stability and mean reward, being so close to one another that they can be safely considered comparable in their performances.

Part III

Future work & Conclusions

6 Future work

Change is the end result of all true learning.

– Leo Buscaglia

Although the positive results brought forward in my work, I have only scraped the surface, and several other experiments can be carried out. I will list below some interesting ideas for future research:

- Increase in the size of the system: as multiple children nodes are added to the network (total > 2) they could potentially improve the effects of Stochastic Weight Averaging, leading to even more stable results.

- Introducing off-policy algorithms: In this project's designing and coding phase, I have also included another important element, namely the shared replay memory to be used within each machine. In this research, due to time constraints, I have not had the chance to use it, yet I believe it could be highly beneficial for improving the performance of off-policy algorithms.

- A further study of the parameters α and β: I have explored solely the middle values and the ones at the extremes in their domain $[0, 1]$, but likely a better joint tuning of the two for better results is possible. It would be highly interesting also to study their generalization to other settings, i.e. whether the same levels of α and β hold performant when the problem at hand changes.

- Quantization: it is a tool growing in popularity which in this case could be highly beneficial to the scaling of this framework. Quantization allows the change in bits representation of the floating point numbers used in the Neural Network, it has been studied in the RL world [30] showing that classical DRL in Atari does not lose performance when quantized down to 8/16 bits while gaining a speedup of 50% in training time on a RaspberryPi. In this case, quantization could also be beneficial in thinning out the weight exchanged by the nodes (cutting down communication time considerably).

- Hogwild! on asynchronous SWA: It would be interesting to see whether Hogwild! would be beneficially implementable on the parent node in the process of SWA. This would of course, assume that the update of the set of weights is both sparse and preferably in a

slow learning setting (low α), so as not to lock the weight, causing blocking when the number of nodes scales up significantly.

- Pushing the low resource settings: The experiments proposed turned out to work on low resource hardware, but a further study is necessary to check whether the computational requirements in terms of hardware can be pushed further to even smaller, cheaper, and computationally inferior devices.

7 Conclusions

There will come a time when you believe everything is finished; that will be the beginning.

– Louis L'Amour

In this document, I have started the journey in the land of Deep Reinforcement Learning, showing how recent the real progress in the field is. It is clear that the new advances all point towards one direction, which is the distributed approach for the training of the latter, bringing many positive and desirable features, by now necessary to address the main concerns of RL. I have then walked through the path of Distributed Reinforcement Learning, exploring state of the art, and surprisingly found that a unified framework using SWA and allowing training in minimal resource settings does not yet exist. This simple observation gave birth to the idea motivating the manufacturing of Genkidama, a framework filling those gaps that I have designed and codified. More importantly, I have shown its potential by comparing its distributed A3C version to its classical vanilla counterpart. Furthermore, the experiments I presented showed how this generalization of A3C comes with the possibility of massive parallelization through the distributed system. All of this comes at the same performance level as the original one (for the same number of overall CPU cores used) while using substantially inferior hardware for the computations. The behavior of this new framework is still to be vastly explored, yet these results are a strong motivation, showing a glance of its true potential and encouraging further research.

Bibliography

(1) Sutton, R. S.; Barto, A. G., *Reinforcement Learning: An Introduction*, Second; The MIT Press: 2018.

(2) Francois-Lavet, V.; Henderson, P.; Islam, R.; Bellemare, M. G.; Pineau, J. An Introduction to Deep Reinforcement Learning, Foundations and Trends in Machine Learning: Vol. 11, No. 3-4, 2018, 2018.

(3) Howard, R. A. Dynamic Programming and Markov Processes, Cambridge, MA, 1960.

(4) Laura Graesser, W. L. K. Foundations of Deep Reinforcement Learning: Theory and Practice in Python, 2019.

(5) Mnih, V.; Badia, A. P.; Mirza, M.; Graves, A.; Lillicrap, T. P.; Harley, T.; Silver, D.; Kavukcuoglu, K. Asynchronous Methods for Deep Reinforcement Learning, 2016.

(6) Auer, P. Using Confidence Bounds for Exploitation-Exploration Trade-Offs, 2003.

(7) Thompson, W. R. On the Likelihood that One Unknown Probability Exceeds Another in View of the Evidence of Two Samples, 1933.

(8) Wang, Z.; Schaul, T.; Hessel, M.; van Hasselt, H.; Lanctot, M.; de Freitas, N. Dueling Network Architectures for Deep Reinforcement Learning, 2015.

(9) Cisco Systems, I. IPv6 Addressing White Paper, 2008.

(10) IETF RFC 9293 Transmission Control Protocol (TCP), 2022.

(11) Postel, J. RFC 768 User Datagram Protocol (UDP), 1980.

(12) Arulkumaran, K.; Deisenroth, M. P.; Brundage, M.; Bharath, A. A. A Brief Survey of Deep Reinforcement Learning, 2017.

(13) Williams, R. J. Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning, USA, 1992.

(14) Tesauro, G. Temporal Difference Learning and TD-Gammon, 1995.

(15) Baird, L. Residual Algorithms: Reinforcement Learning with Function Approximation, 1995.

(16) Boyan, J.; Moore, A. Generalization in Reinforcement Learning: Safely Approximating the Value Function. 1994.

(17) Tsitsiklis, J.; Van Roy, B. An analysis of temporal-difference learning with function approximation, 1997.

(18) Mnih, V.; Kavukcuoglu, K.; Silver, D.; Graves, A.; Antonoglou, I.; Wierstra, D.; Riedmiller, M. Playing Atari with Deep Reinforcement Learning, 2013.

(19) Nair, A.; Srinivasan, P.; Blackwell, S.; Alcicek, C.; Fearon, R.; Maria, A. D.; Panneershelvam, V.; Suleyman, M.; Beattie, C.; Petersen, S.; Legg, S.; Mnih, V.; Kavukcuoglu, K.; Silver, D. Massively Parallel Methods for Deep Reinforcement Learning, 2015.

(20) Niu, F.; Recht, B.; Re, C.; Wright, S. J. HOGWILD!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent, 2011.

(21) Espeholt, L.; Soyer, H.; Munos, R.; Simonyan, K.; Mnih, V.; Ward, T.; Doron, Y.; Firoiu, V.; Harley, T.; Dunning, I.; Legg, S.; Kavukcuoglu, K. IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures, 2018.

(22) Li, Y.; Liu, I.-J.; Yuan, Y.; Chen, D.; Schwing, A.; Huang, J. Accelerating Distributed Reinforcement Learning with In-Switch Computing, Phoenix, Arizona, 2019.

(23) Tehrani, P.; Restuccia, F.; Levorato, M. Federated Deep Reinforcement Learning for the Distributed Control of NextG Wireless Networks, 2021.

(24) Izmailov, P.; Podoprikhin, D.; Garipov, T.; Vetrov, D.; Wilson, A. G. Averaging Weights Leads to Wider Optima and Better Generalization, cite arxiv:1803.05407Comment: Appears at the Conference on Uncertainty in Artificial Intelligence (UAI), 2018, 2018.

(25) Gupta, V.; Serrano, S. A.; DeCoste, D. Stochastic Weight Averaging in Parallel: Large-Batch Training that Generalizes Well, 2020.

(26) Team, T. R. Raylib, 2022.

(27) Research, M. Meta Moolib, 2022.

(28) Stevens, W. R. TCP-IP Illustrated, Vol. 1: The Protocols, 1994.

(29) Brockman, G.; Cheung, V.; Pettersson, L.; Schneider, J.; Schulman, J.; Tang, J.; Zaremba, W. OpenAI Gym, 2016.

(30) Krishnan, S.; Chitlangia, S.; Lam, M.; Wan, Z.; Faust, A.; Reddi, V. J. Quantized Reinforcement Learning (Qua{RL}), 2020.