UNIVERSITÀ DEGLI STUDI DI SIENA

Dipartimento di Ingegneria dell'Informazione
e Scienze Matematiche

Real World Problems through Deep Reinforcement Learning

Luca Pasqualini

Ph.D Thesis in Information Engineering and Science

Supervisor

Prof. Marco Maggini

Co-supervisor

Prof. Maurizio Parton

Siena, 2022

Abstract

Reinforcement Learning (RL) represents a very promising field in the umbrella of Machine Learning (ML). Using algorithms inspired by psychology, specifically by the Operant Conditioning of Behaviorism, RL makes it possible to solve problems from scratch, without any prior knowledge nor data about the task at hand. When used in conjuction with Neural Networks (NNs), RL has proven to be especially effective: we call this Deep Reinforcement Learning (DRL). In recent past, DRL proved super-human capabilities on many games, but its real world applications are varied and range from robotics to general optimization problems.

One of the main focuses of current research and literature in the broader field of Machine Learning (ML) revolves around benchmarks, in a never ending challenge between researchers to the last decimal figure on certain metrics. However, having to pass some benchmark or to beat some other approach as the main objective is, more often than not, limiting from the point of view of actually contributing to the overall goal of ML: to automate as many real tasks as possible.

Following this intuition, this thesis proposes to first analyze a collection of really varied real world tasks and then to develop a set of associated models. Finally, we apply DRL to solve these tasks by means of exploration and exploitation of these models. Specifically, we start from studying how using the score as target influences the performance of a well-known artificial player of Go, in order to develop an agent capable of teaching humans how to play to maximize their score. Then, we move onto machine creativity, using DRL

in conjuction with state-of-the-art Natural Language Processing (NLP) techniques to generate and revise poems in a human-like fashion. We then dive deep into a queue optimization task, to dynamically schedule Ultra Reliable Low Latency Communication (URLLC) packets on top of a set of frequencies previously allocated for enhanced Mobile Broad Band (eMBB) users. Finally, we propose a novel DRL approach to the task of generating black-box Pseudo Random Number Generators (PRNGs) with variable periods, by exploiting the autonomous navigation of a state-of-the-art DRL algorithm both in a feedforward and a recurrent fashion.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

One of the main goal of computer science as a whole is to automate all the processes of gathering and elaboration of various kind of information. By automating these processes, multiple problems can be solved in a way that requires no human involvement and in a really fast way, thanks to the advances in materials and technology. Sometimes, however, there are problems whose solutions are hard to automate. Thanks to *machine learning*, a multitude of real world problems have been solved without the need for humans to actually code a solution, i.e. an algorithm to solve them.

Machine learning research can either be *theoretical* or *practical*, with the latter focusing more on applying existing methods to some kind of tasks to solve them, while the former has the never ending goal of improving the stability and the overall performance of the algorithms employed. This thesis is very grounded into the real world and as such is very practical in nature.

In Section 1.1 we analyze the core reasons from this work and what this thesis and the research behind it aim to achieve, while in Section 1.2 we detail the general concepts common to all chapters of the thesis. In Section 1.3 we report the main questions guiding our research, while in Section 1.4 we describe the overall structure of the thesis. Finally, in Section 1.5 we detail all the research papers published within the period of this research and which are also directly involved in the work presented in the thesis.

1.1 Motivations

Current research and literature in the machine learning field is very often focused on competing on some benchmarks, in a never ending challenge between authors to the last decimal figure on one or more metrics appropriate to

the task at hand. While beating competitors' approaches on various datasets could be an appealing perspective for a researcher, it being the sole focus of research could be quite limited from the point of view of contributing to the overall goal of our field's research: to *automate* as many problems as possible. By automate we mean the task of automating the creation of the automated processes of gathering and elaborating the information that constitute the core of computer science as a whole.

The approach of this thesis is completely opposite to the usual collection of metrics computed within various toy tasks. Our goal is to lay *not that* far from the real world. Indeed, our goal is to apply known algorithms and methods to real world problems, investigating if it is possible to solve them with the least possible amount of prior knowledge. In this journey we'll play a lot with modeling appropriate simulations to real world problems in order to be solved by machine learning methods. While these models are naturally simplifications of the real tasks, they are realized with the goal of being as consistent to the real world as possible. By employing specific algorithms to each one of these simulations and by adapting these algorithms were required, we aim to propose one or multiple way to solve these real world tasks completely from scratch. Are you ready to tag along on our journey?

1.2 General concepts

A very promising field of machine learning is *reinforcement learning*, inspired by psychology, specifically by the Operant Conditioning of Behaviorism. With reinforcement learning it is really possible to train computers from scratch, without prior knowledge nor data about the task at hand. Of course, while this makes reinforcement learning capable of solving tasks of really varied nature, it also makes it a very hard field to work in, and to successfully apply its methods to a certain task is no easy feat.

All the research presented in this thesis revolves around reinforcement learning, in a way or another. The way reinforcement learning is involved depends on the task at hand: it could be in conjunction with supervised learning, working alongside some kind of heuristic or just applied as is. Multiple algorithms or versions of the same algorithm are also employed. While algorithms

change, the general ideas remain the same, and they are all described in 1.2.1.

To correctly apply algorithms to multiple problems, a certain standardization is required. Custom solution might be required on certain tasks, but to have a clear standard and a proved functioning base is really helpful to make sure results are consistent. More often than not, proving that a reinforcement learning method can work on a certain task is already a huge portion of the study itself. Because of that, a framework comprising many reinforcement learning algorithms has been developed during the period of this research. This framework has been applied to all tasks described in this thesis, with the sole exception of the one described in the first chapter. This framework is described in 1.2.2.

1.2.1 Reinforcement Learning (RL)

For a comprehensive, motivational and thorough introduction to RL, we strongly suggest reading from 1.1 to 1.6 in [1].

RL is learning what to do in order to accumulate as much reinforcement as possible during the course of actions. This very general description, known as *the RL problem*, can be framed as a sequential decision-making problem as follows.

Assume an *agent* is interacting with an *environment*. When the agent is in a certain situation - a *state* - it has several options - called *actions*. After each action, the environment will take the agent to a next state, and will provide it with a numerical *reward*, where the pair "state, reward" may possibly be drawn from a joint probability distribution, called the *model* or the *dynamics* of the environment. The agent will choose actions according to a certain strategy, called *policy* in the RL setting. The RL problem can then be stated as finding a policy maximizing the expected value of the total reward accumulated during the interaction agent-environment.

To formalize the above description, see Figure 1.1 representing the agent-environment interaction.

At each time step t, the agent receives a state $S_t \in \mathcal{S}$ from the environment, and then selects an action $A_t \in \mathcal{A}$. The environment answers with a numerical reward $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$ and a next state S_{t+1}. This interaction gives raise to a

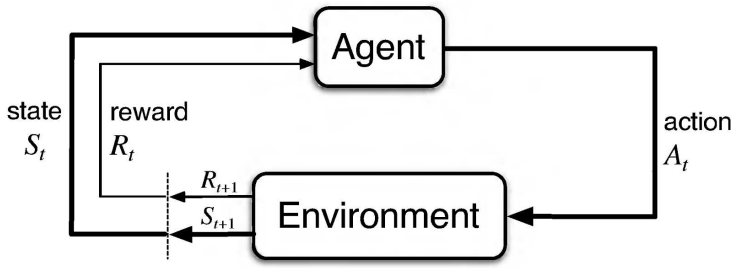


Figure 1.1: ©[1, figure 3.1] The agent-environment interaction is made at discrete time steps $t = 0, 1, 2, \ldots$. At each time step t, the agent use the state $S_t \in \mathcal{S}$ given by the environment to selects an action $A_t \in \mathcal{A}$. The environment answers with a real number $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$ called reward, and a next state S_{t+1}. Going on, we obtain a trajectory $S_0, A_0, R_1, S_1, A_1, R_2, \ldots$

trajectory of random variables:

$$S_0, A_0, R_1, S_1, A_1, R_2, \ldots$$

In the case of interest to us, $\mathcal{S}, \mathcal{A}$ and $\mathcal{R}$ are finite sets. Thus, the environment answers the action $A_t = a$ executed in the state $S_t = s$ with a pair $R_{t+1} = r, S_{t+1} = s'$ drawn from a discrete probability distribution p on $\mathcal{S} \times \mathcal{R}$, the model (or dynamics) of the environment:

$$p(s', r|s, a) := p(s', r, s, a) := \Pr(S_{t+1} = s', R_{t+1} = r|S_t = s, A_t = a).$$

Note the visual clue of the fact that $p(\cdot, \cdot|s, a)$ is a probability distribution for every state-action pair s, a.

Figure 1.1 implicitly assumes that the joint probability distribution of S_{t+1}, R_{t+1} depends on the past only via S_t and A_t. In fact, the environment is fed only with the last action, and no other data from the history. This means that, for a fixed policy, the corresponding stochastic process $\{S_t\}$ is Markov. This gives the name Markov Decision Process (MDP) to the data $(\mathcal{S}, \mathcal{A}, \mathcal{R}, p)$. Moreover, it is a *time-homogeneous* Markov process, because p does not depend on t. In certain problems the agent can see only a portion of the full state, called *observation*. In this case, we say that the MDP is *partially observable*. Observations are usually not Markovian, because the non

observed portion of the state can contain relevant information for the future. In this paper we model a PRNG as an agent in a partially observable MDP.

When the agent experiences a trajectory starting at time t, it accumulates a *discounted return* G_t:

$$G_t := R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}, \qquad \gamma \in [0, 1].$$

The return G_t is a random variable, whose probability distribution depends not only on the model p, but also on how the agent chooses actions in a certain state s. Choices of actions are encoded by the policy, i.e. a discrete probability distribution π on $\mathcal{A}$:

$$\pi(a|s) := \pi(a, s) := \Pr(A_t = a|S_t = s).$$

A discount factor $\gamma < 1$ is used mainly when rewards far in the future are less and less reliable or important, or in *continuing* tasks, that is, when the trajectories do not decompose naturally into *episodes*.

The average return from a state s, that is, the average total reward the agent can accumulate starting from s, represents how good is the state s for the agent *following the policy* π, and it is called *state-value* function:

$$v_\pi(s) := E_\pi[G_t|S_t = s].$$

Likewise, one can define the *action-value* function (known also as *quality* or *q-value*), encoding how good is *choosing an action a from s and then following the policy* π:

$$q_\pi(s, a) := E_\pi[G_t|S_t = s, A_t = a].$$

Since the return G_t is recursively given by $R_{t+1} + \gamma G_{t+1}$, the RL problem has an optimal substructure, expressed by recursive equations for v_* and q_*. If an accurate description of the dynamics p of the environment is available and if one can store in memory all states, then dynamic programming iterative techniques can be used, and an approximate solution v_* or q_* to the Bellman optimality equations can be found. From v_* or q_* one can then easily recover an optimal policy: for instance, $\pi_*(s) := \operatorname{argmax}_{a\in\mathcal{A}} q_*(s, a)$ is a deterministic optimal policy.

However, in most problems we have only a *partial knowledge of the dynamics*, if any. This can be overcome by *sampling trajectories* $S_t = s, A_t = a, R_{t+1}, S_{t+1}, A_{t+1}, R_{t+2}, \dots$ to *estimate* the q-value $q_\pi(s,a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a]$, instead of computing a true expectation. Moreover, in most problems *there are way too many states to store them in memory*, or just to go through every state just once. In this case, the estimate of $q_\pi(s,a)$ must be stored in a parametric function approximator $q_\pi(s,a;w)$, where w is a parameters vector living in a dimension much lower than $|\mathcal{S} \times \mathcal{A}|$. Due to their high representational power, Deep Neural Networks (DNN) are nowadays widely used as approximators in RL: the combination of DNN with RL is called Deep Reinforcement Learning (DRL).

Iterative dynamic programming techniques can then be approximated, giving a family of algorithms known as *Generalized Policy Iteration* algorithms. They work by sampling trajectories to obtain estimates of the true values $\mathbb{E}_\pi[G_t|S_t = s, A_t = a]$, and using supervised learning to find the optimal parameters vector w for $q_\pi(s,a;w)$. This estimated q-value is used to find a policy π' better than π, and iterating over this evaluation-improvement loop usually gives an approximate solution to the RL problem.

Generalized Policy Iteration is *value-based*, because uses a value function as a proxy for the optimal policy. A completely different approach to the RL problem is given by Policy Gradient (PG) algorithms. They estimate directly the policy $\pi(a|s;\theta)$, without using a value function. The parameters vector θ_t at time t is modified to maximize a suitable scalar performance function $J(\theta)$, with the gradient ascent update rule:

$$\theta_{t+1} := \theta_t + \alpha\widehat{\nabla J(\theta_t)}.$$

Here the *learning rate* α is the step size of the gradient ascent algorithm, determining how much we are trying to improve the policy at each update, and $\widehat{\nabla J(\theta_t)}$ is any estimate of the performance gradient $\nabla J(\theta)$ of the policy. Different choices for the estimator corresponds to different PG algorithms. The vanilla choice for the estimator $\widehat{\nabla J(\theta_t)}$ is given by the Policy Gradient Theorem, leading to an algorithm called REINFORCE and to its baselined derivatives, see for instance [1, Section 13.2 and forward]. Unfortunately, vanilla PG algorithms can be very sensitive to the learning rate, and a single update with a large α can spoil the performance of the policy learned so far.

Moreover, the variance of the Monte Carlo estimate is high, and a huge amount of episodes are required for convergence. For this reason, several alternatives for $\widehat{\nabla J(\theta_t)}$ has been researched.

In the thesis we mostly use PG algorithms. The two main algorithms we use are described in Chapter 3. We saw that PG algorithms, and Proximal Policy Optimization in particular, performs quite well on all of our tasks. At the end of our journey we will also discuss what we think are the reasons for this.

1.2.2 Framework

The framework used for all RL algorithms, with the sole exception of the study presented in Chapter 2, is USienaRL[1].

This framework allows for environment, agent and interface definition using a preset of configurable models. While agents and environments are direct implementations of what is described in the RL theory, interfaces are specific to this implementation. Under this framework, an interface is a system used to convert environment states to agent observations, and to encode agent actions into the environment. This allows to define agents operating on different spaces while keeping the same environment. By default an interface is defined as pass-through, i.e. a fully observable state where agents action have direct effect on the environment.

While there exists multiple frameworks for RL research available in literature, we developed a custom solution from scratch in order to better control and understand the underlying algorithms. In various occasions, algorithms are unstable because of numerical problem due to hyperparameters or certain reward values. Having designed and coded from scratch all algorithms employed in this thesis, while them being still part of a cohesive, standardized and tested framework, allows us to better understand which hyperparameters need tuning or which algorithm is better suited to solve certain kind of tasks.

Moreover, each task is modeled into an environment designed to be compatible with this framework. Reward are designed with both the algorithms stability and the goal of the task in mind. This process is known in literature

[1] Available on PyPi and also on GitHub: `https://github.com/InsaneMonster/USienaRL`.

as reward engineering. Reward engineering and reward transmission back and forth between the environments and the algorithms is completely managed by the framework.

1.3 Research questions and contributions

Research always start from one or more questions. In the case of this thesis, since we are focusing on applying RL to the real world, we can define one main question: *is RL capable of solving real world problems?*

Naturally, this question is both too broad and too complex to be answered in just one breath. It also depend on the task at hand! We can decompose this question into multiple questions, each one focused on a specific problem. In what follows, all these questions are reported, as well as the proposed contributions we made to answer each one of them. Each contribution defines a research path we followed, and we discuss in detail, chapter by chapter, all of them throughout the thesis.

1. *AlphaGo is a powerful software capable of playing Go at superhuman strength. While incredibly powerful, it doesn't know how to play to maximize its score. In the real world, many passionate players are eager to learn from the software but it is hard to learn a definite strategy when all the final set of moves are not optimal, usually even bad. Is it possible to make use of AlphaGo superhuman strength to learn from it in a complete way? Is it possible to take the score into account directly while using the AlphaGo algorithm?*

2. **Leela Zero Score** is an AlphaGo-like software we propose to study the behaviour of the machine player when the algorithm is trained on scores instead of binary outputs. Developing this software also allowed us to verify a well known statement among the researchers of this field: "using a score doesn't work".

3. *One of the most human skills of them all is creativity. Machines do not know how to create, while humans, often, do. With respect to machine learning, Language Generation Models are used to generate sentences given a context. Poetry, while being just text in a way, it has to adhere*

to a specific set of rules, like rhyme schemes or a specific metric. In order to achieve this, poets start from a draft, iteratively refining it until the desired meaning is conveyed in the required form. Is it possible for an algorithm to refine a poem? Can creativity be injected into this process by letting the algorithm to find a good solution by itself, from scratch?

4. **A generate and revise framework** for poetry is what we propose to mimic the human creative process, i.e. to generate poems that are repeatedly revisited and corrected, as humans do, in order to improve their overall quality. The task of revising poems according to their desired rhyme scheme is framed in the context of RL, injecting creativity into the process by exploring a solution from scratch. We also show how this approach is general and not only limited to fixing a poem rhyme scheme.

5. *Recent advances in telecommunication technology and protocols, like the advent of 5G networks, generated a large amount of possible interesting avenues for research. One of them is to find out how to efficiently manage the coexistence of multiple types of traffic, each one with its own set of stringent requirements, incompatible one another. Is it possible to use an algorithm to automatically find one optimal, or at least good, solution? What kind of RL algorithms can achieve this result and how should the task be modeled as a MDP to be efficiently solved?*

6. **A resource slicing model** is proposed as modeled task, where ultra-reliable low-latency communications and enhanced Mobile BroadBand traffics have to coexist on a time-frequency resource grid. We address the problem of devising a suitable MDP for the task and of training a good DRL agent, employing a state-of-the-art PG algorithm. We show that the policy devised by the DRL agent never violates the latency requirement of URLLC traffic and, at the same time, manages to keep the number of eMBB codewords in outage at minimum levels, when compared to other state-of-the-art schemes.

7. *In the field of cryptography, RL has little to now use. In most real world applications, where security is important but there are not enough resources to employ true-random numbers, pseudo-random numbers are*

employed in their place. Machine learning is often used to break the deterministic algorithms used to generate such pseudo-random numbers, or at the very least to imitate one of them. Is it possible to use the learning from scratch approach of RL in the field of cryptography, specifically by generating from scratch pseudo-random numbers generators? If so, which algorithms have greater performances and how should this task be modeled as a MDP to be solved?

8. **Two innovative approaches to the task of generating pseudo-random numbers generators** is what we propose to address this task. Specifically, we first train a RL agent to learn a policy to solve an N-dimensional navigation problem. In this context, N is the length of the period of the sequence to generate and the policy is iteratively improved using the average score of an appropriate test suite run over that period. Then, we also show how the aforementioned approach can be improved by learning a policy to solve a partially observable MDP, where the full state is the period of the generated sequence and the observation at each time step is the last sequence of bits appended to such state. Both approaches are innovative and they lay the foundation of a research path not undertaken by anyone before.

To sum up, we take multiple real world task to analyze. We define appropriate system models and, according to the model, we study how algorithms behave. Through their behaviour, we assess if solving the problem by means of RL is a feasible. If it is, we analyze how good of a solution we get compared to other state-of-the-art approaches, like heuristics or other machine learning methods.

In the following section, we organize each contribution into chapters, each chapter representing one leg of our journey into the application of RL to real world tasks.

1.4 Thesis structure

In the following Chapters the thesis will try to answer the research questions described in Section 1.3, detailing each one of the introduced contributions.

1. In Chapter 2 a practical study of the AlphaGo algorithm from a score perspective is presented. Indeed, AlphaGo and all of its derivatives can play with superhuman strength because they are able to predict the win-loss outcome with great accuracy. However, Go as a game is decided by a final score, and in final positions AG plays sub-optimal moves: this is not surprising, since AG is completely unaware of the final score, all winning final positions being equivalent from the win-rate perspective. This can be an issue, for instance when trying to learn the "best" move or to play with an initial handicap. Moreover, there is the theoretical quest of the "perfect game". No empirical or theoretical evidence can be found in the literature to support the folklore statement that "using a score instead of a win-rate doesn't work". We present Leela Zero Score an AG-like software built to support or disprove this "doesn't work" statement. For simplicity, we will keep the following discussion focused on the sole game of Go, but we keep in mind that our approach is also applicable to all other games where similar conditions apply, i.e. having a score as result of a match.

2. In Chapter 3 we analyze the human creative process of writing, proposing an approach to emulate it through a machine. When humans write something, their text is usually revisited, adjusted, modified, rephrased, even multiple times, in order to better convey meanings, emotions and feelings that the author wants to express. Amongst the noble written arts, Poetry is probably the one that needs to be elaborated the most, since the composition has to formally respect predefined meter and rhyming schemes. We discuss a framework to generate poems that are repeatedly revisited and corrected, as humans do, in order to improve their overall quality. We frame the problem of revising poems in the context of RL and we compare two PG algorithms over the task. Our model generates poems from scratch and it learns to progressively adjust the generated text in order to match a target criterion. We evaluate this approach in the case of matching a rhyming scheme, without having any information on which words are responsible of creating rhymes and on how to coherently alter the poem words. The proposed framework is also general and, with an appropriate reward shaping, it can be applied to other text

generation problems.

3. In Chapter 4 we present a RL solution to a novel and very relevant research issue in the field of 5G and beyond 5G (B5G) networks: how to manage the coexistence of different types of traffic, each with very stringent but completely different requirements. We propose a DRL algorithm to slice the available physical layer resources between ultra-reliable low-latency communications (URLLC) and enhanced Mobile BroadBand (eMBB) traffic. Specifically, in our setting the time-frequency resource grid is fully occupied by eMBB traffic and we train the DRL agent to employ PPO, a state-of-the-art DRL algorithm, to dynamically allocate the incoming URLLC traffic by puncturing eMBB codewords. We show that the policy devised by the DRL agent never violates the latency requirement of URLLC traffic and, at the same time, manages to keep the number of eMBB codewords in outage at minimum levels, when compared to other state-of-the-art schemes.

4. Finally, in Chapter 5 we propose an innovative approach to the field of cryptography, specifically to the task of generating statistically uncorrelated numbers, i.e. Pseudo-Random Numbers (PRNs). They are generate through specific algorithms known as Pseudo Random Number Generators (PRNGs). Test suites are used to evaluate PRNGs quality by checking statistical properties of the generated sequences. These sequences are commonly represented bit by bit. Machine learning techniques are often used to break these generators, i.e. approximating a certain generator or a certain sequence using a NN. But what about using machine learning to *generate* PRNs generators? We first propose a RL approach to the task of generating PRNGs from scratch by learning a policy to solve an N-dimensional navigation problem. In this context, N is the length of the period of the sequence to generate and the policy is iteratively improved using the average score of an appropriate test suite run over that period. This approach lays the foundation of our study on PRNG and relies on a feedforward NN operating a fully observable MDP. Then, we also propose a more advanced approach to the same task, by learning a policy to solve a partially observable MDP, where the full state is the period of the generated sequence and the observa-

tion at each time step is the last sequence of bits appended to such state. We use a LSTM architecture to model the temporal relationship between observations at different time steps, by tasking the LSTM memory with the extraction of significant features of the hidden portion of the MDP's states. We also show that modeling a PRNG with a partially observable MDP and an LSTM architecture largely improves the results of the fully observable feedforward approach.

1.5 List of Publications

In the following, a list of the research contributions produced during the period of this research is provided.

Peer reviewed journal papers

1. **Pasqualini, L.** and Parton, M. (2020). "Pseudo random number generation through reinforcement learning and recurrent neural networks". *Algorithms (13), 11, 307*
   Candidate Contribution: approach conceptualization and modeling, design and development of the experimental campaign

Peer reviewed conference papers

1. **Pasqualini, L.** and Parton, M. (2020). "Pseudo random number generation: A reinforcement learning approach". *Procedia Computer Science, 170, pp. 1122–112* (**IWSMAI 2020**)
   Candidate Contribution: approach conceptualization and modeling, design and development of the experimental campaign

2. Zugarini, A., **Pasqualini, L.**, Melacci, S. and Maggini, M. (2021). "Generate and Revise: Reinforcement Learning in Neural Poetry". (**IJCNN 2021**)
   Candidate Contribution: conceptualization, design and development of RL solution, joint design and development of the experimental campaign

3. Saggese, F., **Pasqualini, L.**, Moretti, M. and Abrardo, A. (2021). "Deep Reinforcement Learning for URLLC data management on top of scheduled eMBB traffic". (**accepted for publication at 2021 GLOBECOM - IEEE Global Communications Conference**)
   Candidate Contribution: conceptualization, design and development of RL solution, joint design and development of the experimental campaign

4. Meloni, E., **Pasqualini, L.**, Tiezzi, M., Gori, M., and Melacci, S. (2021). "SAILenv: Learning in Virtual Visual Environments Made Simple". In *International Conference on Pattern Recognition* (**ICPR 2020**)
   Candidate Contribution: design and development of the visual environment: coding, modeling, texturing, user experience

5. Meloni, E., Tiezzi, M., **Pasqualini, L.**, Gori, M., and Melacci, S. (2021). "Messing Up 3D Virtual Environments: Transferable Adversarial 3D Objects". (**ICMLA 2021**)
   Candidate Contribution: design and development of the visual environment: coding, modeling, texturing, user experience

Chapter 2

AlphaGo Score Targeting through Reinforcement Learning

The first leg of our journey into the applications of reinforcement learning in real world tasks could not refrain to start from the study of very first application breaking through all news media around the world: AlphaGo (AG). As software, AG is *just* playing Go at superhuman level strength, but the same approach to learning from scratch can be applied to a large variety of board games, like chess. AG and all of its derivatives can play with such superhuman strength because they are able to predict the win-loss outcome with great accuracy. However, Go as a game is decided by a final score, and in final positions AG plays sub-optimal moves: this is not surprising, since AG is completely unaware of the final score, all winning final positions being equivalent from the win-rate perspective. This can be an issue, for instance when trying to learn the "best" move or to play with an initial handicap. Moreover, there is the theoretical quest of the "perfect game". Thus, a natural question arises: is it possible to train a successful AG-like DRL agent to predict scores instead of win-rates?

No empirical or theoretical evidence can be found in the literature to support the folklore statement that "this doesn't work". In this chapter we present Leela Zero Score (LZS), an AG-like software built to support or disprove the "doesn't work" statement. LZS is built on the open source solution known as Leela Zero (LZ), and is trained to predict scores instead of win-rates. For simplicity, we will keep the following discussion focused on the sole game of Go, but we keep in mind that our approach is also applicable to all other games where similar conditions apply, i.e. having a score as result of a match.

2.1 Introduction

The game of Go has been a landmark challenge for AI research since its very beginning. It is very suited to AI, with the importance of patterns and the need for deep exploration, and very tough to actually solve, with its whole-board features and subtle interdependencies of local situations. Nowadays, AI has reached superhuman level in the game of Go with the well-known DeepMind algorithm AlphaGo Zero [2] (AGZ), a zero-knowledge evolution of AG [3]. More generally, any perfect information two-player zero-sum game like Go can be tackled efficiently by DeepMind algorithm AlphaZero [4] (AZ).

In perfect information two-player zero-sum games, where the win/lose outcome is given by a final score difference, maximizing this score difference is still an open and important question. In fact, AG derivatives play suboptimal moves in the endgame. The open-source clean room implementation of AGZ known as LZ [5] is also known to play suboptimal moves, see Section 4.4 in [6].

This phenomenon is rooted in the win/lose reward in the RL pipeline. Giving a reward of 1 (win) or 0 (lose) at the end of the game means that AG derivatives maximize the winrate instead of the actual score difference. It is a folklore statement that replicating the AG pipeline using score instead of the binary outcome as a primary target is unsuccessful. A qualitative argument is that score is unlikely to be a successful reward, because a single point difference may change the winner, thus inducing instability in the training. As a matter of fact, the two most recent and successful RL approaches to score maximization in the game of Go, that is, KataGo [7] and SAI [6] have taken different routes. KataGo does include score estimation, but only as a secondary target: the value to be maximized is a linear combination of winrate and expectation of a nonlinear function of the score difference, not the score difference itself. In SAI, the winrate is modeled as a two-parameters family of sigmoids $\sigma_{\alpha,\beta}$: while α can be seen as the final score difference, α and β are learnt indirectly by training $\sigma_{\alpha,\beta}$ against the classical binary reward.

Still, humans do use score estimations instead of winrate estimations while playing score-based games. Therefore, the question remains: if a DRL agent can learn how to maximize its winrate, why should it not be possible to learn how to win by maximizing the final score? The implications from a DRL

training process perspective are far from trivial.

2.2 Score as Target

This discussion aims to provide a sound and quantified direct evidence of the limitations of training a DRL agent directly on the score difference. To this aim we train an instance of LZ on the 9×9 board, using score as a target. We name this instance LSZ. We show that the training is successful, but it does converges prematurely to a player weaker than a corresponding AGZ-like player.

In this section, we first introduce the open source engines LZ and SAI in 2.2.1. Then, we describe the changes we made to LZ to obtain LZS in 2.2.2, as well as our training techniques. In Section 2.3, we show our results and we run both a qualitative and a quantitative analysis. Finally, in Section 2.4 we describe all implications of the aforementioned results.

2.2.1 Leela Zero and SAI

Free and open source, LZ [5] is a Go program with no human provided knowledge, known as one of the most faithful reimplementations of the system described in AlphaGo Zero [2]. For all intents and purposes, it is considered an open-source AGZ. The agent plays using MCTS without Monte Carlo playouts and a deep residual convolutional neural network stack.

Released on 25 October 2017, the neural network powering the agent of LZ is trained by a distributed effort, which is coordinated at the LZ website. Lacking the computational power required to train AGZ, members of the community provided computing resources by running the client, which generates self-play games and submits them to the server.

The self-play games are used to train newer networks. Newer networks are then matched one against the other in multiple games and promoted according to a process known as gating. These games are known as matches. Through gating, newer networks losing to the previous better ones are discarded, speeding up the training process.

The training process of LZ ended on 15 February 2021. LZ is available at `https://zero.sjeng.org/` and `https://github.com/leela-zero/`

`leela-zero`.

SAI is a fork of LZ described thoroughly in [8, 9, 6, 10]. Unlike LZ, SAI was trained also on the 9×9 board, and, for the purposes of this work, SAI 9×9 can be considered an AGZ-like software.

2.2.2 Leela Zero Score

The overall architecture of the open-source engine LZ was replicated in our setting. The main difference is in the target. To implement a score instead of a binary target during the training and inference phases, the following issues were addressed:

- The board size: LZ has an hard-coded board size 19×19, while LZS was implemented on a 9×9 board for efficiency purposes.
- The outcome: instead of binary flag, i.e. 0 or 1, the target became an integer number, in the range $s \in [-N^2, N^2]$, N being the Go board size.
- The winrate: the role of a probability between 0.0 and 1.0 in choosing the best possible move according to the MCTS tree was taken by the expected score, which is a number $s_e \in [-N^2, N^2]$, where N is the Go board size.
- The heuristics: during the training, LZ employs a set of heuristics during self-plays to avoid useless sets of actions at the end of the match. For example, there is no point for a player to keep playing if it wins by passing. This heuristic doesn't work anymore in our new setting: to maximize score, the agent should keep playing so long it has a chance of increasing its current score.
- To minimize the modifications to the existing MCTS code in the LZ software, the outcome s and the expected score s_e were both normalised to $\hat{s} = \frac{s}{N^2}$ and $\hat{s}_e = \frac{s_e}{N^2}$ in $[-1.0, 1.0]$.

Scaling down the board from size $n \in \mathbb{N}$ to size ρn with $\rho < 1$ yields several benefits:

- Average number of legal moves at each position scales down by ρ^2.

- Average length of games scales down by ρ^2.
- The number of visits in the MCTS tree scales down by ρ^4, which can be inferred from the previous two.
- The number of layers in the residual convolutional neural network stack scales by ρ.
- The fully connected layers at the end of the neural network stack are smaller. This grants increased training and inference speed.

To summarize the performance benefits of scaling down the board dimension from 19×19 to 9×9, we can estimate a total speed improvement for self-play games in the order of ρ^9.

The heuristics of LZ were updated. In a state where passing causes a win for the current player with score $\hat{s}$ (where the score is evaluated by Tromp-Taylor rules[1], the UC tree is visited to check for better alternatives, i.e. if the following condition is verified for at least one node:

$$\hat{s}_e > \hat{s}$$

where $\hat{s}_e$ is the expected score computed by LZS network introduced previously, and $\hat{s}$ is the current score. If this is not the case, the agent passes like it did in LZ. On the other hand, if one or more than one better moves are found, the best one is chosen according to $\hat{s}_e$.

2.2.3 Training

Our training was composed by multiple phases, inspired from the original LZ training process, as well as general knowledge inferred from SAI training. Specifically, the phases of each training cycle were as follows.

- Self-play. A set of 2000 self-plays per cycle, where the network plays Go against itself using the modified LZ engine with the following parameters: no playouts, a variable amount of visits v, increased randomness while playing the first 15 moves, a set of 6 threads, a batch size of 5,

[1] `https://senseis.xmp.net/?TrompTaylorRules`

noise network randomization and specific heuristics for passing during the game.

- Training. The network is trained over a window of self-plays. Specifically, the most recent self-plays are downloaded and added to the previously downloaded ones, if available. Then, positions from all the self-plays within a variable window w are extracted and used as data, in a supervised learning fashion with a training, validation and test set.

- Gating. A set of 400 matches played between the new trained network and the current best network using the modified LZ engine is played. Parameters are set as follows: no playouts, $v = 100$, no randomness in the moves, a set of 6 threads, a batch size of 5, no noise network randomization and no heuristics for passing during the game.

- Promotion. Depending on the results of the gating step, the old best network is maintained or replaced. Specifically, if at least 55% of the 400 matches are won by the candidate network, it is promoted to be the new best network. It's important to note that while we train the network to maximize its score, through gating we are assessing its capabilities in winning the game. In other words, the most desirable outcome is to obtain a network good at winning through score maximization.

After the end of each cycle, a new cycle starts. When a network is promoted to be the new best network, the current training generation number increases. At some values of the generation number g, the hyperparameters $v(g)$ and $w(g)$ are updated, where $v(g)$ is the amount of visits in MCTS tree during self-plays, and $w(g)$ is the size of the training window, as follows:

$$v(g) = \begin{cases} 100 & \text{if } g \leq 15 \\ 150 & \text{if } 15 < g \leq 31 \\ 250 & \text{if } 31 < g \leq 47 \\ 400 & \text{if } 47 < g \leq 63 \\ 600 & \text{if } 63 < g \leq 79 \\ 850 & \text{otherwise} \end{cases}$$

$$w(g) = \begin{cases} 4 & \text{if } g \leq 15 \\ 8 & \text{if } 15 < g \leq 31 \\ 12 & \text{if } 31 < g \leq 47 \\ 16 & \text{if } 47 < g \leq 63 \\ 20 & \text{if } 63 < g \leq 79 \\ 24 & \text{otherwise} \end{cases}$$

2.3 Results

In Figure 2.1 we show training results in terms of uncalibrated Elo rating.

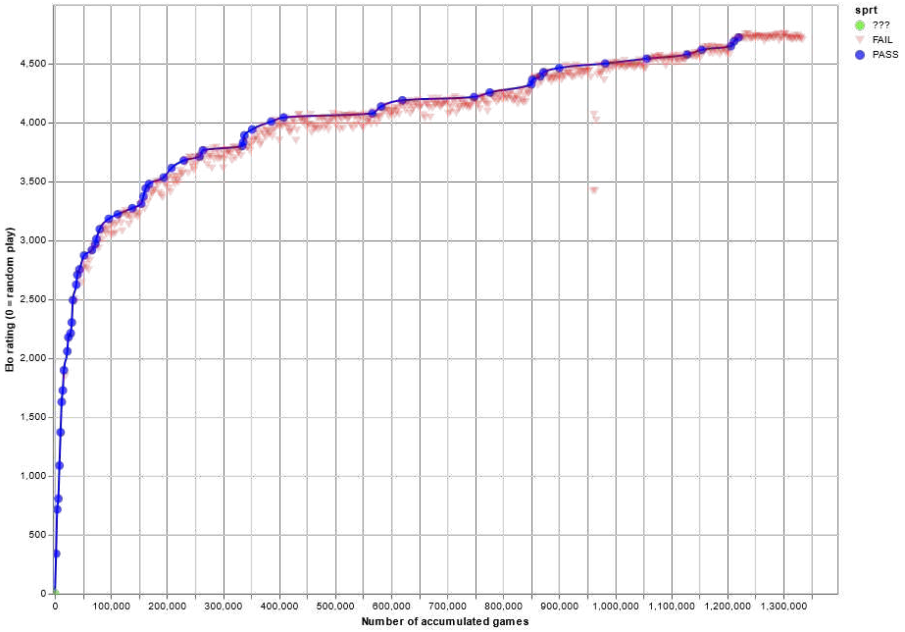


Figure 2.1: Uncalibrated Elo ratings of various LZS networks during training, expressed w.r.t. the amount of self-plays.

On the x-axis the amount of self-plays are shown. On the y-axis the uncalibrated Elo is reported. Like in LZ [5], the blue circle represents matches won by the new trained network, i.e. when a promotion happens, while pink

triangles show when such matches fail to promote the last trained network. The Elo estimate is uncalibrated, because it is based on setting to 0 the Elo of the first network, which was chosen arbitrarily as a random network. A calibrated estimate of the Elo rating is described in the next sections.

However, for the purposes of the training, the uncalibrated estimate was sufficient, as it allowed to assess when the training process stalled: when the new trained network was not able to beat the previous best network for an empirically-chosen amount of cycles, we scaled up the network size. Starting from a base network with 2 residual convolutional layers of 64 filters, from now on referred to as 2×64, we scaled to:

- 4×128 at $g = 1$, after 2000 self-plays.
- 8×160 at $g = 25$, after 150000 self-plays.
- 10×192 at $g = 43$, after 720000 self-plays.
- 12×256 at $g = 50$, after 908000 self-plays.

Similarly, we changed the learning rate l_r of training at certain times during the entire process, with the goal of reducing stalls. Such times are selected empirically, depending on the size of the network and the current generation. As a simple rule of thumb, we reduced the learning rate when we trained bigger networks at reasonably high generation numbers and we saw very little improvements from one set of 2000 self-plays to the next. Specifically, we trained with $l_r = 0.005$ for $g < 44$ and with $l_r = 0.0005$ for $g > 45$.

Based on the expectation from the SAI 9×9 run, a stopping rule for the training was decided a priori, that the run would have been stopped when no promotion was obtained after 40 cycles, i.e., 80000 self-plays. The rule was met after 1400000 self-plays.

To assess the strength of LZS, we picked the best network and run both a qualitative and a quantitative evaluation.

2.3.1 Qualitative Evaluation

Fifteen games were played between LS and Carlo Metta, a strong amateur player[2]. Ten games were played with 400 visits for each move, five games were played with 20000 visits for each move.

A thorough analysis of such games shows that training has been successful in producing a consistent player, which, however, exhibits some unusual characteristics when compared to other artificial agents. The match ended with a score of 14-1 in favor of the human player: although LZS found itself in a position of clear advantage several times, it was only able to win one game, one of those with 20000 visits. LZS showed some peculiar and not always desirable features. LZS certainly has a direct and aggressive style. It does not seem to admit sacrificing few stones for a better final results, not to foresee sacrifice on the opponent's side. This is clearly in contrast with the flexibility shown by other artificial agents.

Another striking situation occurred several times: when in balanced positions, LZS attempted to further increase the score difference, rather than settling for a narrow victory, in such an aggressive and self-delusional way that it resulted in an inevitable defeat. It may be argued that this phenomenon was a direct effect of LZS training scheme.

2.3.2 Quantitative Evaluation

For a quantitative evaluation of the strength of LZS, we need a calibrated Elo rating.

Elo ratings are computed with a maximum likelihood optimization algorithm, similar to Rémi Coulom's Bayes Elo [11]. To get a sensible anchoring, we selected one out of four LZS promoted networks, and compared their strength against a panel of 12 SAI networks of similar strength. All the matches in this evaluation step were played with the following settings.

- LZS: $v = 400$, no randomness in the moves, a set of 6 threads, a batch size of 5, no noise network randomization and no heuristics for passing during the game.

[2]Player profile on the European Go Database `https://www.europeangodatabase.eu/EGD/Player_Card.php?&key=14713996`.

- SAI: $v = 400$, no randomness in the moves, a set of 6 threads, a batch size of 5, no noise network randomization and no heuristics for passing during the game, $\lambda = 0$ and $\mu = 0$.

LZS and SAI played black at alternate times.

To select the panel of SAI networks, we first estimated the Elo of LZS. We first considered the panel used in [6]. LZS lost all games against the second weakest net of the panel, which had a Elo of 3500. We therefore chose a second, weaker panel, namely, SAI_{2000}, SAI_{2500}, SAI_{3000}, and SAI_{3500}, where SAI_x is a SAI network of the principal run in [6] having Elo rate approximately x.

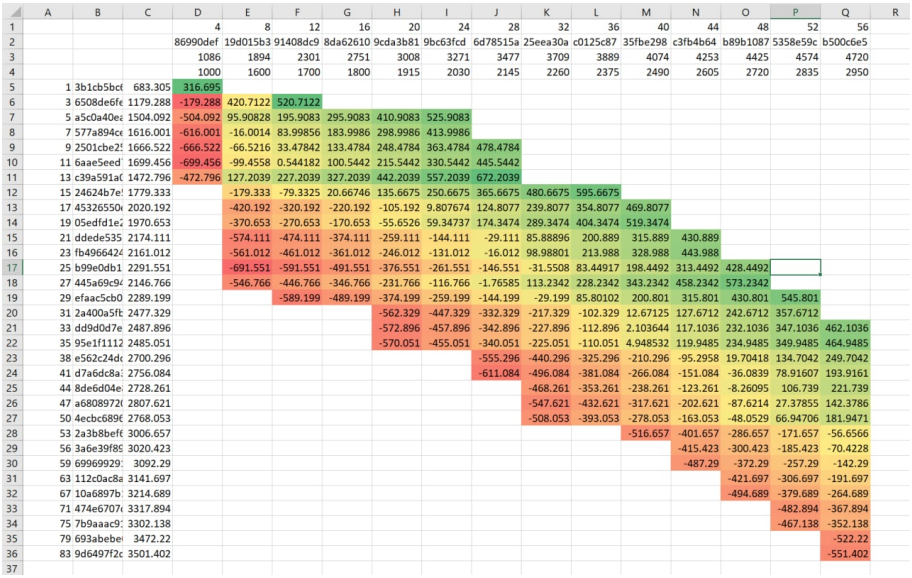
After 200 games between LZS and the preliminary panel, LZS had the following winrates:

$$w_r(N) = \begin{cases} 0.985 & \text{against } \text{SAI}_{2000} \\ 0.925 & \text{against } \text{SAI}_{2500} \\ 0.480 & \text{against } \text{SAI}_{3000} \\ 0.155 & \text{against } \text{SAI}_{3500} \end{cases}$$

Based on this results, we finally chose as calibration panel a set of 32 SAI networks, whose Elo ranged from 683 to 3501. The matches were organized as follows: starting from the first, we chose the next LZS network in the sample; then, we had the chosen LZS network play 20 games against a set of SAI networks of appropriate strength. Some combinations were not required, because the strength difference would make the results useless from a statistical point of view; we saved the results and iterated to the next LZS network. See Figure 2.2.

The total amount of matches was 207. The results of the matches were fed into the Elo algorithm.

In Figure 2.3 we compare the LZS run with the first SAI run. SAI grew at a steadier pace than LZS. This run of SAI was cut at around 200000 self-plays, therefore the figure was cut at that level. In Figure 2.4 we compared LZS with the second run of SAI, which converged after a comparable number of self plays. This comparison showed that, during training, LZS had consistently lower values of Elo. Moreover, this calibrated version of Figure 2.1 confirmed that the last increase in the size of the network, at 900000 self-plays, had



				4	8	12	16	20	24	28	32	36	40	44	48	52	56
				86990def	19d015b3	91408dc9	8da62610	9cda3b81	9bc63fcd	6d78515a	25eea30a	c0125c87	35fbe298	c3fb4b64	b89b1087	5358e59c	b500c6e5
				1086	1894	2301	2751	3008	3271	3477	3709	3889	4074	4253	4425	4574	4720
				1000	1600	1700	1800	1915	2030	2145	2260	2375	2490	2605	2720	2835	2950
1	3b1cb5bc	683.305		316.695													
3	6508de6fe	1179.288		-179.288	420.7122	520.7122											
5	a5c0a40ea	1504.092		-504.092	95.90828	195.9083	295.9083	410.9083	525.9083								
7	577a894ce	1616.001		-616.001	-16.0014	83.99856	183.9986	298.9986	413.9986								
9	2501cbe2!	1666.522		-666.522	-66.5216	33.47842	133.4784	248.4784	363.4784	478.4784							
11	6aae5eed	1699.456		-699.456	-99.4558	0.544182	100.5442	215.5442	330.5442	445.5442							
13	c39a591a(	1472.796		-472.796	127.2039	227.2039	327.2039	442.2039	557.2039	672.2039							
15	24624b7e!	1779.333			-179.333	-79.3325	20.66746	135.6675	250.6675	365.6675	480.6675	595.6675					
17	45326550(	2020.192			-420.192	-320.192	-220.192	-105.192	9.807674	124.8077	239.8077	354.8077	469.8077				
19	05edfd1e2	1970.653			-370.653	-270.653	-170.653	-55.6526	59.34737	174.3474	289.3474	404.3474	519.3474				
21	ddede535	2174.111			-574.111	-474.111	-374.111	-259.111	-144.111	-29.111	85.88896	200.889	315.889	430.889			
23	fb496642(	2161.012			-561.012	-461.012	-361.012	-246.012	-131.012	-16.012	98.98801	213.988	328.988	443.988			
25	b99e0db1	2291.551			-691.551	-591.551	-491.551	-376.551	-261.551	-146.551	-31.5508	83.44917	198.4492	313.4492	428.4492		
27	445a69c9(	2146.766			-546.766	-446.766	-346.766	-231.766	-116.766	-1.76585	113.2342	228.2342	343.2342	458.2342	573.2342		
29	efaac5cb0	2289.199				-589.199	-489.199	-374.199	-259.199	-144.199	-29.199	85.80102	200.801	315.801	430.801	545.801	
31	2a400a5fb	2477.329						-562.329	-447.329	-332.329	-217.329	-102.329	12.67125	127.6712	242.6712	357.6712	
33	dd9d0d7e	2487.896						-572.896	-457.896	-342.896	-227.896	-112.896	2.103644	117.1036	232.1036	347.1036	462.1036
35	95e1f111(	2485.051						-570.051	-455.051	-340.051	-225.051	-110.051	4.948532	119.9485	234.9485	349.9485	464.9485
38	e562c24d(	2700.296								-555.296	-440.296	-325.296	-210.296	-95.2958	19.70418	134.7042	249.7042
41	d7a6dc8a:	2756.084								-611.084	-496.084	-381.084	-266.084	-151.084	-36.0839	78.91607	193.9161
44	8de6d04e:	2728.261									-468.261	-353.261	-238.261	-123.261	-8.26095	106.739	221.739
47	a6808972(	2807.621									-547.621	-432.621	-317.621	-202.621	-87.6214	27.37855	142.3786
50	4ecbc689(	2768.053									-508.053	-393.053	-278.053	-163.053	-48.0529	66.94706	181.9471
53	2a3b8bef(	3006.657											-516.657	-401.657	-286.657	-171.657	-56.6566
56	3a6e39f85	3020.423												-415.423	-300.423	-185.423	-70.4228
59	69969929:	3092.29												-487.29	-372.29	-257.29	-142.29
63	112c0ac8a	3141.697													-421.697	-306.697	-191.697
67	10a6897b:	3214.689													-494.689	-379.689	-264.689
71	474e6707(	3317.894														-482.894	-367.894
75	7b9aaac9:	3302.138														-467.138	-352.138
79	693abebe(	3472.22															-522.22
83	9d6497f2c	3501.402															-551.402

Figure 2.2: Match table between LZS networks and SAI networks, where hashes, estimated Elo ratings and relative differences are displayed.

not produced any relevant improvement in the next 400000 self-plays, thus confirming that LZS was converging prematurely at a lower level player.

2.4 Discussion

In this chapter we discussed a Go artificial player trained using score as target. The same pipeline was used as the pipeline of SAI and LZ, well known AG-like open source software, whose networks were instead trained with the traditional binary target. The training was successful, and produced a player with valid play but particular style. After calibration, the training proved to produce consistently weaker networks than the corresponding networks in the SAI training, and converged prematurely to a lower Elo level. Our results prove the folklore statement that using the score as target doesn't work as well as using win-rates, while still converging to a reasonable player with an interesting, while suboptimal, playstyle. We think that such playstyle could still be reasonably employed to train average level players with an opponent

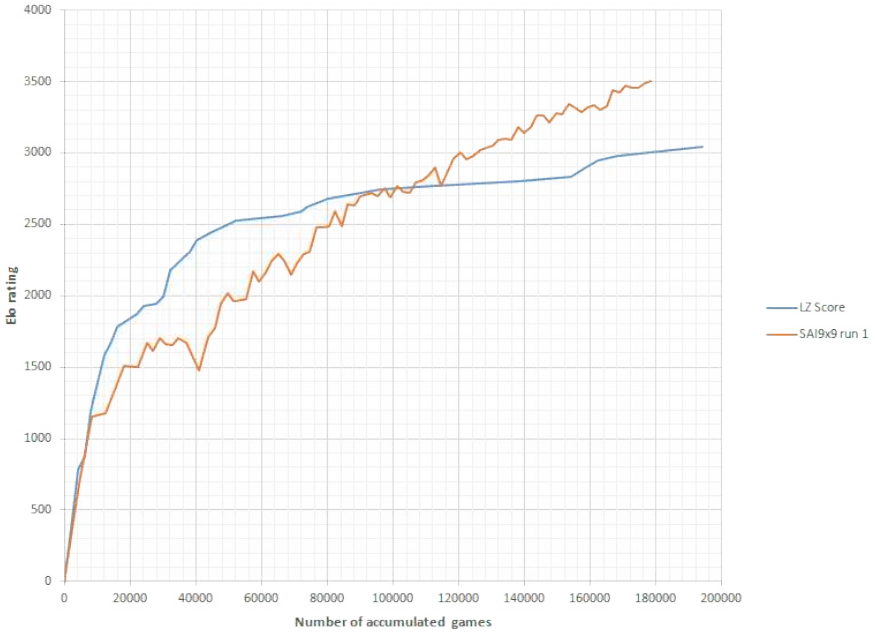


Figure 2.3: Calibrated Elo ratings of various LZS networks and associated SAI 9×9 networks in the first run, expressed w.r.t. the amount of self-plays.

whose strategy adapt to the players' needs.

Figure 2.4: Calibrated Elo ratings of various LZS networks and associated SAI 9×9 networks in the second run, expressed w.r.t. the amount of self-plays.

Chapter 3

Neural Poetry through Reinforcement Learning

The second leg of our journey into the real world applications of RL is oriented towards an artistic and creative task: poem generation and revision. Usually, writers, poets and singers do not create their compositions in just one breath. Text is revisited, adjusted, modified, rephrased, even multiple times, in order to better convey meanings, emotions and feelings that the author wants to express. Amongst the noble written arts, Poetry is probably the one that needs to be elaborated the most, since the composition has to formally respect predefined meter and rhyming schemes. Is it possible to add a revision process to a poem generation task, in order to better mimic human approach to writing? Moreover, is it possible to exploit RL to stimulate creativeness into such revision process?

In this chapter, we discuss a framework to generate poems that are repeatedly revisited and corrected, as humans do, in order to improve their overall quality. We frame the problem of revising poems in the context of RL and we compare two PG algorithms over the task. Our model generates poems from scratch and it learns to progressively adjust the generated text in order to match a target criterion. We evaluate this approach in the case of matching a rhyming scheme, without having any information on which words are responsible of creating rhymes and on how to coherently alter the poem words. The proposed framework is also general and it can be applied to other text generation problems through domain adaptation, for example by changing the reward function.

3.1 Introduction

Developing machines that reproduce artistic behaviours and learn to be creative is a long-standing goal of the scientific community in the context of Artificial Intelligence [12, 13]. Recently, several researches focused on the case of the noble art of Poetry, motivated by success of Deep Learning approaches to Natural Language Processing (NLP) and, more specifically, to Natural Language Generation [14, 15, 16, 17, 18, 19]. However, existing Machine Learning-based poem generators do not model the natural way poems are created by humans, i.e., poets usually do not create their compositions all in one breath. Usually a poet revisits, rephrases, adjusts a poem many times, before reaching a text that perfectly conveys their intended meanings and emotions. In particular, a typical feature of poems is that the composition has also to formally respect predefined meter and rhyming schemes.

Early methods on Poetry Generation [20] addressed the problem with rule-based techniques, whereas more recent approaches focused on learnable neural language models. The first deep learning solutions tackled Chinese Poetry. In [14], authors combined convolutional and recurrent networks to generate quatrains. Afterwards, both [16] and [15] proposed a sequence-to-sequence model with attention mechanisms. In the context of English Poetry, transducers were exploited to generate poetic text [17]. The generation structure (meter and rhyme) is learned from characters by cascading a module considering the context, with a weighted state transducer. Recently, in Deep-speare [18], the authors generated English quatrains with a combination of three neural models that share the same character-based embeddings. One network is a character-aware language model predicting at word level, another neural model learns the meter, and the last one identifies rhyming pairs. Generated quatrains are finally selected after a post-processing step from the output of the three modules. In [19], the authors focused on a single Italian poet, Dante Alighieri, by making use of a syllable-based language model, that was trained with a multi-stage procedure on non-poetic works of the same author and on a large Italian corpus.

With the aim of developing an artificial agent that learns to mimic this behaviour, we design a framework to generate poems that are repeatedly revisited and corrected, in order to improve the overall quality of the poem.

We frame this problem as a navigation task approached with RL, exploiting Proximal Policy Optimization [21] that, to our best knowledge, is not commonly applied to Natural Language Generation, despite being an improved instance of the more common Vanilla Policy Gradient. In the task of generating and progressively editing the draft of a poem until it matches a target rhyming scheme, we show that Proximal Policy Optimization leads to better results than Vanilla Policy Gradient. The agent is not informed about what a rhyme is and how to implement the considered scheme, making the task extremely challenging in a RL perspective. The agent generates a draft poem and it corrects the draft one word at a time. It not only understands that the ending words of each verse are the ones that are important with respect to the rhyming scheme, but that also other words of the poem might need to be adjusted to make the poem coherent with the rhyming words. Despite the application to poetry generation, the proposed framework is general and it can be applied to other text generation problems, provided an opportune reward shaping.

RL has been recently used in several Natural Language Generation applications, such as Text Summarization [22, 23, 24], Machine Translation [25] and Poem Generation [26, 27] as well. However, most of the proposed approaches exploit RL as a mean to make common evaluation metrics differentiable, such as BLEU and ROUGE scores [28]. Of course, these metrics can be computed only in those tasks in which the target text (ground truth) is available. In [26] the authors extended Generative Adversarial Networks (GANs) [29] to the generation of sequences of symbols, through RL. The GAN discriminator is used as a reward signal for a RL-based language generator, and, among a variety of tasks, their framework was applied to Chinese quatrains generation. In [27], a mutual RL scheme was used to improve the quality of the generated Chinese quatrains. Another RL-based approach is proposed in [27], where two networks simultaneously learn from each other using a mutual RL scheme, in order to improve the quality of the generated poems. In both works, different generic rewards were designed exploiting the simplest policy-based RL algorithm, i.e. Vanilla Policy Gradient. Surprisingly, Proximal Policy Optimization is less commonly used in the scope of Natural Language Generation, despite leading to a more robust and efficient RL algorithm [30].

Our generate-and-revise framework is related to retrieve-and-edit seq2seq approaches [31, 32, 33, 34, 35], where text generation reduces to an adaptation/paraphrasing of the retrieved template(s) related to the current input. The refinement process can be optimized with standard seq2seq learning algorithms because of the presence of revised targets. In our generate-and-revise instead, we neither start from retrieved templates, nor we have reference revisions. That is why we cast the problem as a navigation task and exploit RL to learn a revision policy that adjusts draft poems in order to improve their quality.

This chapter is organized as follows. The neural models are described in Section 3.2, while the RL-based poem revision dynamics is detailed in Section 3.3. Results are reported in Section 3.4 and, finally, we discuss them in Section 3.5.

3.2 Generate and Revise Poems

Our framework is rooted on the idea that creating a poem is a multi-step process. First, the draft of a new poem is generated. Then, an iterative revision procedure is activated, in which the initial draft is progressively edited. We model this problem by means of a *generator*, that creates the draft, and a *reviser*, that edits the draft up to the final version of the poem. The reviser is structured as an iterative procedure that, at each iteration, identifies a word of the poem which does not suit well the context in which it is located, and substitutes it with a better word. At each step the reviser has to decide both *which* word to replace and *with what*. A straightforward approach to implement this idea is to design a RL agent that jointly addresses both the tasks. Thus, given an m-word poem with vocabulary size $|V|$, the agent has to choose among a large number of actions, i.e. $|V| \cdot m$, due to usually large $|V|$ (in the order of tens of thousands in our experiments). Therefore the problem quickly becomes extremely hard to tackle.

We keep the idea of exploiting a RL-based approach, but we decouple the problem implementing the reviser with two learnable models, namely the *detector* and the *prompter*, each of them responsible of one of the two aforementioned tasks, i.e., detecting a word to substitute (detector), and suggesting how

to change a target word (prompter), respectively. The *generator*, the *detector*, and the *prompter* are based on neural architectures, trained from scratch with appropriate criteria, while the *detector* is fully developed by means of RL. The whole scheme is sketched in Fig. 3.1. The structure of this module allows us to reduce the action space of the RL procedure to (up to) N words in the poem, making it independent on $|V|$. The prompter identifies the words in V that are most compatible with the surrounding context. In early planning stages, also an overall character-level process was considered, but it was discarded because it hindered convergence to meaningful sentences.

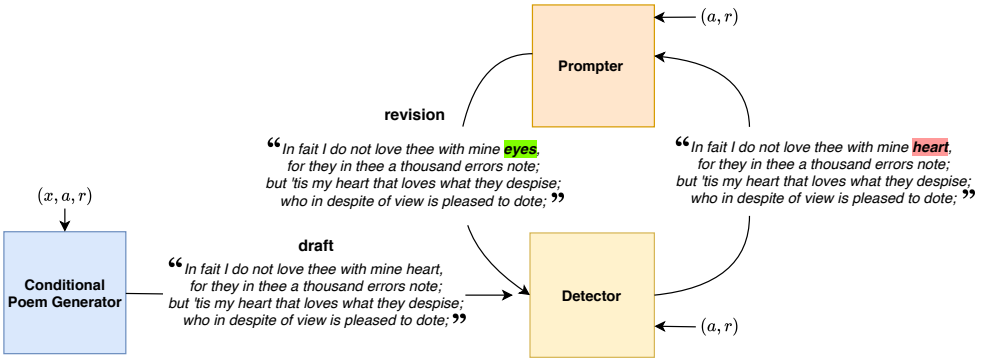


Figure 3.1: Overall Generate and Revise scheme on an example poem. The conditional poem generator (light blue module) produces a draft poem, which is iteratively revised by the detector (pale yellow) - Prompter (light orange) modules until satisfaction of certain criteria. At each step the detector identifies the word to replace, *heart* highlighted in red, while the prompter is responsible for finding the substitute, *eyes* highlighted in green.

In the following, the generator (Section 3.2.1), the detector (Section 3.2.2) and the prompter (Section 3.2.3) will be described in detail, whereas the RL-dynamics of the detector are presented in Section 3.3.

3.2.1 Conditional Poem Generator

The poem generation procedure is an instance of Natural Language Generation based on a learnable Language Model (LM). Before considering the specific details of Poetry, we describe the LM used in this discussion. Let us consider

a sequence of tokens $(w_1, \ldots, w_{n+m})$ taken from a text corpus in a target language. For convenience in the description, let us divide the tokens into two sequences $\boldsymbol{x}$ and $\boldsymbol{y}$, where $\boldsymbol{x} = (x_1, \ldots, x_n)$ and $\boldsymbol{y} = (y_1, \ldots, y_m)$. The former ($\boldsymbol{x}$) is the context provided to the text generator from which to start the production of new text. More generally, $\boldsymbol{x}$ is a source of information that conditions the generation of $\boldsymbol{y}$ ($\boldsymbol{x}$ could also be empty). The goal of the LM is to estimate the probability $p(\boldsymbol{y})$, that is factorized as follows,

$$p(\boldsymbol{y}) = \prod_{i=1}^{m} p(y_i | y_{<i}, \boldsymbol{x}), \tag{3.1}$$

being $y_{<i}$ a compact notation to indicate the words in the left context of y_i. Notice that when the sequence $\boldsymbol{x}$ has size $n = 0$, we fall back to the traditional LM formulation [36]. The text generation is the outcome of sampling the next sequence $\boldsymbol{y}$ from (3.1). Machine Translation, Text Summarization, Text Continuation, Poem Generation, and in general any sequence-to-sequence problem in NLP can be formulated as in (3.1). The way $p(y_i|y_{<i}, \boldsymbol{x})$ will be related to the input sequence $\boldsymbol{x}$ depends on how strongly $\boldsymbol{x}$ is informative with respect to $\boldsymbol{y}$. Problems in which the source sequence significantly biases the generation outcome are referred as *non-open-ended* text generation, in contrast to *open-ended* text generation, where the source sequence loosely correlates with the output $\boldsymbol{y}$ [37].

Poem Generation is an instance of *open-ended* text generation. When starting to generate a novel poem from scratch, there is, of course, no source input sequence. After having generated a few verses or when starting from a few given verses, the next-verses generation can be conditioned using them (i.e., $\boldsymbol{x}$ contains previously given verses, while $\boldsymbol{y}$ is about the verses to be generated), but there is still a huge degree of freedom in the possible verses that can be generated, due to the intrinsically creative nature of Poetry. There might be several features to further constrain the LM with information that does not come from the input text, and, in this discussion, we consider two important features, that are the author a and the target rhyme scheme r. We update (3.1) by introducing the information on a and r,

$$p(\boldsymbol{y}) = \prod_{i=1}^{m} p(y_i | y_{<i}, \boldsymbol{x}, a, r). \tag{3.2}$$

that is the reference equation on which our poem generation is based.

We model the distribution in Equation 3.2 by means of a sequence-to-sequence neural architecture with attention. Our LM is a variant of [38], similar to the one proposed in [18], and it is based on an encoding-decoding scheme. The encoder is responsible of creating a compact representation of $\boldsymbol{x}$, while the decoder yields a probability distribution over the words in V given the outcome of the encoder, and the conditioning signals a and r leading to $p(y_i|y_{<i}, \boldsymbol{x}, a, r)$.

Encoding. The encoder of $\boldsymbol{x}$ computes a contextual representation of each word x_j of the input sequence $\boldsymbol{x}$ (n words), by means of a bidirectional Long Short Term Memory (bi-LSTM). The output of this module is the set $H_x = \{\boldsymbol{h}_1, \ldots, \boldsymbol{h}_n\}$, being $\boldsymbol{h}_j$ the contextualized representation of the j-th word. In detail, at each time step j, the bi-LSTM is fed with the concatenation of the word embedding $\boldsymbol{w}_j \in \mathbb{R}^d$ associated to x_j, and $\boldsymbol{u}_j \in \mathbb{R}^r$, a character-based representation of x_j. We indicate with $\overrightarrow{\boldsymbol{h}}_j$, $\overleftarrow{\boldsymbol{h}}_j$ the internal states of the bi-LSTM processing the sequence of augmented word representations,

$$\overrightarrow{\boldsymbol{h}}_j = \overrightarrow{LSTM}_{\text{encx}}([\boldsymbol{w}_j, \boldsymbol{u}_j], \overrightarrow{\boldsymbol{h}}_{j-1}),$$
$$\overleftarrow{\boldsymbol{h}}_j = \overleftarrow{LSTM}_{\text{encx}}([\boldsymbol{w}_j, \boldsymbol{u}_j], \overleftarrow{\boldsymbol{h}}_{j+1}),$$

where $\overrightarrow{LSTM}$, $\overleftarrow{LSTM}$ are the functions computed by the LSTMs in the two directions. The final representation of the j-th word of the input sequence is $\boldsymbol{h}_j = [\overrightarrow{\boldsymbol{h}}_j, \overleftarrow{\boldsymbol{h}}_j]$. Overall, the encoder outputs $H_x = \{\boldsymbol{h}_1, \ldots, \boldsymbol{h}_n\}$. The char-based representation $\boldsymbol{u}_j$ is obtained by processing the word characters with another bi-LSTM. We augment $\boldsymbol{w}_j$ with a char-based representation to better encode sub-word information, that is crucial to capture rhyming schemes and meter in the poems.

Decoding. The decoder is responsible of returning the distribution $p(y|y_{<i}, \boldsymbol{x}, a, r)$ at each time index i, and, when used to generate text, to sample a word from p. We stack two recurrent layers. First an LSTM that computes at each time step i a representation $\boldsymbol{z}_i$ given the previous word y_{i-1} merged with author (a) and rhyme scheme (r) information encoded in form of embeddings $\boldsymbol{a}$ and $\boldsymbol{r}$, obtaining:

$$\boldsymbol{z}_i = \text{LSTM}_{\text{dec}}([\boldsymbol{w}_{i-1}, \boldsymbol{u}_{i-1}, \boldsymbol{a}, \boldsymbol{r}], \boldsymbol{z}_{i-1}),$$

where $\boldsymbol{w}_{i-1}$ is the word embedding of y_{i-1} and $\boldsymbol{u}_{i-1}$ is the character-aware word representation shared with the encoder. Thanks to the inputs $\boldsymbol{a}$ and $\boldsymbol{r}$, the state $\boldsymbol{z}_i$ includes author-specific and rhyme-scheme-specific information. This allows the system to generate text that is oriented toward the given author style and the target rhyme scheme. The second recurrent layer is a Gated Recurrent Unit (GRU) cell [39] that progressively fuses $\boldsymbol{z}_i$ with the context data in H_x, in order to create a further vector $\boldsymbol{q}_i \in \mathbb{R}^d$ that compactly includes all the conditioning signals of (3.2). First, an attention mechanism [38] is applied over the encoding of the words of $\boldsymbol{x}$, i.e, on their contextualized representations collected in H_x, yielding an attention-based representation $\boldsymbol{c}_i$ of $\boldsymbol{x}$,

$$\boldsymbol{c}_i = \text{attn}(\boldsymbol{z}_i, H_x).$$

Then, the concatenation of $\boldsymbol{c}_i$ with the representation $\boldsymbol{z}_i$ of the triple $(y_{<i}, a, r)$ is processed by a GRU cell,

$$\boldsymbol{q}_i = \text{GRU}([\boldsymbol{c}_i, \boldsymbol{z}_i], \boldsymbol{z}_{i-1}),$$

Finally, the distribution $p(y|y_{<i}, \boldsymbol{x}, a, r)$ is obtained through a linear projection of $\boldsymbol{q}_i$ with the transposed embedding matrix $E' \in \mathbb{R}^{d \times |V|}$, and then applying the softmax function. The model is trained to maximize $p(\boldsymbol{y})$ on a text corpora of poems (see Section 3.4).

Generation. Poems are generated sampling from p. As a matter of fact, the sampling strategy plays a crucial role in the quality of the generated text, and it has been recently shown to have a major impact in Natural Language Generation [40]. We preferred nucleus (top-p) sampling, with $p = 0.9$, to generate quatrains over multinomial and top-k sampling. We indicate with $\boldsymbol{o}$ the sequence of words sampled from p that will consitute a draft poem. The drafts poems generated by the model will be then revised by the joint work of detector and prompter modules.

3.2.2 Detector

Once we have generated a draft poem using the model of Section 3.2.1, a detection module learns to select the next word of the draft that needs to be

revised. The detector is a neural model that yields a probability distribution $\pi(o_i|\boldsymbol{o}, a, r)$ over the N words of the poem. Of course, in order to detect which words to replace, it is important to take into account the author and rhyme information. In detail, the words of poem $\boldsymbol{o}$ are encoded by a network that is analogous to the encoder of $\boldsymbol{x}$ in Section 3.2.1. The word representations, collected in H_o, are processed by an attention mechanisms attn_{det}, building a compact embedding of the whole poem that is also function of the author and of the rhyme scheme. Then, a Multi-Layer Perceptron (MLP) with softmax activation in the output layer returns the probability over the N words,

$$\pi(o_j|\boldsymbol{o}, a, r) = \text{MLP}_j(\text{attn}_{det}([\boldsymbol{a}, \boldsymbol{r}], H_o)) \tag{3.3}$$

being MLP_j the j-th output unit. Multinomial sampling applied to π leads to the selection of the word(s) that should be replaced. This module is trained by RL, as we will describe in Section 3.3.

3.2.3 Prompter

The role of the prompter module is to provide valid candidates to replace the word previously selected by the detector of Section 3.2.2. The prompter module solves the problem of modeling language given the left-right contexts of each word, that can be formulated following an approach similar to the one exploited by the conditional LM of (3.2). Thus, given an author a and a rhyme scheme r, we use a neural model to learn the following distribution from data,

$$p(\boldsymbol{o}) = \prod_{i=1}^{N} p(o_i|o_{<i}, o_{>i}, a, r), \tag{3.4}$$

being $o_{<i}$, $o_{>i}$ the words in left and right context of o_i, respectively. Once $\text{p}(\boldsymbol{o})$ has been learnt, we can sample $p(o_i|o_{<i}, o_{>i}, a, r)$ to get one or more candidate words for replacing the selected one.

The prompter network follows the context encoding schemes of [41] and [42]. In particular, the words of poem $\boldsymbol{o}$ are encoded by a network that computes representations of the left and right contexts around each target word, discarding the target word itself. Differently from the encoding of $\boldsymbol{o}$ in Sec-

tion 3.2.2, here the final representation of the j-th word is then $[\overrightarrow{\boldsymbol{h}}_{j-1}, \overleftarrow{\boldsymbol{h}}_{j+1}]$.[1] This representation is concatenated with the author embedding $\boldsymbol{a}$ and the rhyme scheme embedding $\boldsymbol{r}$, followed by a learnable linear layer with softmax activation that projects the concatenated vector to the space of vocabulary indices. Including $\boldsymbol{a}$ and $\boldsymbol{r}$ in the prompter module is crucial in order to allow the network to learn how to revise a target word in function of the poet and rhyme scheme. Candidate(s) for replacing the selected word are sampled from $p(o_i|o_{<i}, o_{>i}, a, r)$, as discussed in the Poem Generator of Section 3.2.1. In this case we used top-k sampling ($k = 50$) to have a large pool of candidates. The prompter is trained to maximize $p(\boldsymbol{o})$ on a text corpora of poems (Section 3.4).

3.3 Revision as a Navigation Task

Once the poem generator and the prompter modules have been trained, the task of revising a generated poem consists in detecting which words to change and letting the prompter replace them. If we assume to change one word at a time, we can easily consider this task as a decision process in the space of the dictionary words V. Each decision defines which word to change at that a given step, and the prompter replaces it with a suitable candidate. The sequence of decisions is the *policy* of an agent whose goal is to improve the text, according to a given reward function. Text revision stops when a satisfying score has been reached. This task may be cast as a navigation problem, where the current *state* of the agent is identified by the sequence of the words in the current text revision. This allows us to reformulate the problem as a RL task where the navigation space is the environment [43], while the decisions are identified by actions executed by the agent in the environment[2].

As described in Chapter 1, specifically in Section 1.2.1, a RL problem can be framed as a sequential decision-making problem in which, at each step t, the agent observes a state $S_t \in \mathcal{S}$ from the environment, and then selects an action $A_t \in \mathcal{A}$. The environment yields a numerical reward $R_{t+1} \in \mathcal{R}$ and then it moves to the next state S_{t+1}. This interaction gives raise to a *trajectory*

[1] In our implementation, we used the same LSTMs when encoding data in the detector and in the prompter module.

[2] For a comprehensive introduction to RL, see sections from 1.1 to 1.6 in [1].

of random variables. In our task, since words are elements of the vocabulary V, we have that $\mathcal{S}$ is the space of the poems of length N with words from V for the target author a and with rhyming scheme r, $\mathcal{A}$ is the set of indices of the word positions in the poem plus the do-nothing action, while $\mathcal{R} \subset \mathbb{R}$. To define a reward function we use the shortest path problem formulation. The agent aims at reaching the final text revision in the least amount of steps. Conventionally this means that the reward R_t is defined as a negative number for each state not at the goal state position and a positive number or zero when the goal state is reached. Formally, if $\boldsymbol{o}_t$ is the poem revision at step t, we have

$$A_t = \hat{A}_t \in \bigcup_{g=1}^{N+1} \{g\} \tag{3.5}$$

$$S_{t+1} = (\boldsymbol{o}_{t+1}, a, r) \tag{3.6}$$

$$R_{t+1} = \begin{cases} 1 & \text{if } S_{t+1} = S_f \\ -1 & \text{otherwise} \end{cases} \tag{3.7}$$

where S_f is the goal state in which the text is not revised anymore, i.e. when the poem rhyme scheme matches the target rhyme scheme.

The natural connection between the modules presented in Section 3.2 and the RL-based setting is easily established once we redefine (3.3) as the probability of an action in the state described by the triple $(\boldsymbol{o}, a, r)$, that perfectly suits the definition in (3.6), yielding a *policy* function. Using DNNs to approximate the RL-related functions, as we do in the case of the probability distribution over the action space π, is a pretty common approach in nowadays RL-based problems (see, e.g., [43]). In the following descriptions, we compactly rewrite (3.3) adding the symbol θ to refer to the network weights that are learned by means of the RL procedure, i.e., $\pi(\cdot|\cdot;\theta)$. Policy Gradient methods are suitable for navigation tasks, as shown in [44], especially when the states' space becomes large [45]. In such spaces often off-policy algorithms (like Q-Learning) are indeed observed to be unable to converge. In this discussion, we compare two on-policy RL algorithms: Vanilla Policy Gradient (Section 3.3.1) and Proximal Policy Optimization (Section 3.3.2).

3.3.1 Vanilla Policy Gradient

Vanilla Policy Gradient (VPG) [46] is an on-policy RL algorithm whose aim is to learn a policy without using q-values as a proxy. This is obtained increasing the probabilities of actions that lead to higher return, and decreasing the probabilities of actions that lead to lower return. Actions are usually sampled from a multinomial distribution for discrete actions' spaces and from a normal distribution for continuous action spaces. VPG works by updating policy parameters θ via stochastic gradient ascent on policy performance over a buffer built from a certain number of trajectories,

$$\theta \longleftarrow \theta + \alpha \nabla_\theta J(\pi(\cdot|\cdot;\theta))$$

where $J(\pi(\cdot|\cdot;\theta))$ denotes the expected finite-horizon undiscounted return of the policy and ∇_θ its gradient with respect to θ ($\alpha > 0$). In order to compute $J(\pi(\cdot|\cdot;\theta))$, the algorithm requires to evaluate further actions for each state s in the buffer. In this chapter, we use Generalized Advantage Estimation (GAE) [47] to compute such actions, and the obtained rewards are saved and normalized with respect to "when" they are collected (the so called rewards-to-go). These are solutions reported in literature to be stable and to improve overall training performance of the model.

3.3.2 Proximal Policy Optimization

Proximal Policy Optimization (PPO) [21] is another on-policy RL algorithm which improves upon VPG. It is considered the state-of-the-art in policy optimization methods and it is a modified version of Trust Region Policy Optimization (TRPO) [48]. Both methods try to take the biggest possible improvement step on a policy using the currently available data, without stepping "too far" and making the performance collapse. This is done by maximizing a surrogate objective, subject to a constraint on policy update quantity, where such constraint depends on the KL-divergence between the old policy and the new policy after the update. Specifically, PPO uses a clipped objective to heuristically constrain the KL-divergence,

$$\max_\theta \mathbb{E}[\min(\rho_t \bar{A}_t, \text{clip}(\rho_t, 1-\epsilon, 1+\epsilon) \cdot \bar{A}_t),$$

where $\rho_t = \frac{\pi(A_t|S_t;\theta_t)}{\pi_{\theta_{old}}(A_t|S_t;\theta_{t-1})}$ is a policy ratio, $\text{clip}(\rho_t, \cdot, \cdot)$ clips ρ_t in the interval defined by the last two arguments, ϵ is an hyperparameter (we set $\epsilon = 0.2$), $\bar{A}_t$ is the estimated advantage function at time step t and A_t, S_t are respectively the action and the state at time step t. In the implementation used in this chapter, when parameters θ are updated over a buffer of trajectories, the update process is early stopped if the constraint is not respected, thus avoiding the new policy to step "too far" from the previous one.

3.4 Results

We collected poems in English language from the Project Gutenberg[3] using the GutenTag tool [49] to filter out non-poetic work and collections. We also discarded non-English contents that occasionally appeared in the retrieved documents. Poems are organized in stanzas, according to their XML-based description. Each stanza was then divided into quatrains, if not already in such format, and we assigned a rhyming scheme to each stanza, from a fixed dictionary of rhyming schemes. Rhymes were automatically detected with the Pronouncing library[4] and a few additional heuristic rules to cover most of the undetected rhymes. Long poems without any rhyming pattern were discarded as well. We used the meta-information about the author to define the authorship of the stanza, when available. We considered the most 768 frequent authors, the rest was marked as unknown. Overall, we obtained $757,891$ quatrains, divided in three sets of the sizes $684,100$, $36,006$ and $37,785$, respectively, used to train, validate and test the models. We limited the word vocabulary to the most frequent $50,000$ words, assigning an embedding of size 300 for all the models in all the experiments. The maximum sequence length of a quatrain has been set to 50, longer verses were truncated.

We define multiple experimental settings and tasks in order to evaluate the quality of the each module proposed in this work, up to the entire system that includes all the modules and the full pipeline of generation and iterative revision.

[3] `https://www.gutenberg.org/`

[4] `https://pypi.org/project/pronouncing/`

3.4.1 Conditional Poem Generator

While the proposed generator of Section 3.2.1, follows an established neural architecture, the innovative elements we introduce in this work are about the poem-related conditional features, *author* and *rhyme scheme*, and their use in Poetry with character-aware representations. We considered the task of generating a quatrain $\boldsymbol{y}$ given the context sequence $\boldsymbol{x}$ that is the previous quatrain, where the rhyme scheme is a symbol indicating the rhymes of an eight-verse poem. We considered the 50 most frequent rhyme schemes of size eight. The architecture hyper-parameters were commonly selected by choosing the best configuration on the validation set for the vanilla (i.e., not conditioned by author and rhyme) generator. The bi-LSTM state encoding the context sequence $\boldsymbol{x}$ was set to 512, as the state of the decoder $LSTM_{enct}$, and the GRU cell as well. Author and rhyme embedding sizes were set to 128 and 256, respectively.

We compared in terms of generation perplexity a model trained with or without any of the newly introduced conditional features, reporting results in Table 3.1. The conditional features allows the LM to be more accurate, that is an important result considering the open-ended challenging nature of the poem generation task.

Table 3.1: Perplexity measured on the validation (Val) and test (Test) sets of the poem generator, trained with or without conditional features.

	Val	Test
Vanilla Generator	52.98	59.78
Conditional Generator	**51.40**	**54.86**

3.4.2 Prompter

A similar analysis was followed to evaluate the quality of the prompter model of Section 3.2.3. In particular, we trained a prompter model on single quatrains, enforcing it to learn how to predict a word given its context. We used a bi-LSTM state of 1024 units. Again, the role of the new conditional features is what we are mostly interested in and, observing results of Table 3.2, we

can see that they improve the suggestion quality. This result is in line with the case of Section 3.4.1, confirming the importance of further poem-related information.

Table 3.2: Perplexity measured on the validation (Val) and test (Test) sets of the prompter module, trained with or without conditional features.

	Val	Test
Vanilla Prompter	14.09	14.78
Conditional Prompter	**12.90**	**13.40**

3.4.3 Revision as a Navigation Task

In order to show the quality of the detector module and that approaching text correction as shortest path problem is feasible, we created "corrupted" poems from real poems in the dataset by replacing one or more words in random positions with words sampled from the entire vocabulary V.[5] The agent operates in an environment where each episode starts with a corrupted poem, and it has to learn to reconstruct the original not-corrupted poem, selecting at each step which word to change. In this artificial setting we assume that, once the agent picks which word to substitute, a perfect prompter (oracle) will replace it with the ground truth, i.e. the word originally positioned there in the real poem. This means that after each agent action, the selected position will be either replaced with the original word, in case of a corrupted word, or nothing will be changed, in case of a correct word. The navigation terminates when the goal state is reached, that occurs after all the corrupted words are removed from the poem.

The MLP predicting actions has a single hidden layer of size 512. We performed different experiments over this poem reconstruction environment, using a PPO-based agent. Each experiment differs in the number of poems that the agent has to fix, and the number of words perturbed in the poem. We considered {1, 10, 100} poems that, at the beginning of each episode, are randomly "corrupted" by altering 1 or 3 (referred to as "multiple") words in

[5]Frequent words are sampled as replacement more often than rare ones.

the original poem. Please note that even in the simplest case, the experiment with one poem only and a single perturbed word, the number of generated "corrupted" poems is huge, $|V|^{|x|}$, where $|x|$ is the poem length.

The PPO-agent is trained for 10 volleys in each experiment, with $1,000/20,000/200,000$ episodes in the experiments with $1/10/100$ poems, respectively. We set the maximum episode length to 10 steps. Hence, the reward varies between $[-10, 1]$ where 1 corresponds to the case in which the agent immediately identifies the "corrupted" word (when there is only 1 corrupted word), and -10 indicates a full failure. Results are shown in Table 3.3. The "Volley 0" column defines the average total reward at the end of the first volley, while the "Volley 9" column defines such value at the end of the last volley.

Table 3.3: Results of the experiments with the PPO-based agent on poem reconstruction task of Section 3.4.3. The averaged total rewards after the first volley and the last volley are reported, respectively.

	R Volley 0	R Volley 9
1 Poem	-7.866	-0.425
1 Poem (multiple)	-9.092	-3.126
10 Poems	-7.793	0.293
100 Poems	-8.110	-6.389

The reward value improves during the training volleys, while increasing the number of poems makes the problem exponentially more complex. Due to the dynamic perturbation of the poems on each volley, the policy learned by the agent is not tied to a set of perturbed words, but it is general over the set of poems the agent is trained on. However, independently from the amount of poems, results confirm that the revision problem can be framed into a shortest path problem and addressed by RL using PPO, in a varying amount of time.

3.4.4 Generate and Revise Poems

Now we consider the complete system in which all the modules are active as in Fig. 3.1. We focused on the task of generating poems and progressively revising them, in which the agent goal is to substitute words so that the poem matches a target rhyme scheme. Episodes begin with poems generated by

the conditional generator. This task is significantly more challenging than the previously described ones, since there is no ground truth for generated poems, and words replacements are provided by the prompter model described in Section 3.2.3. Therefore, we let the model free to change any word in the quatrain, without restricting the agent actions to words at the end of each verse. Basically, the agent does not know that rhymes are related to the ending words of some verses, while the only information it receives is the reward (or penalty) signal that tells if the poem fulfills the target rhyming scheme or not.

We ran several experiments comparing PPO with VPG, varying in each experiment the number of poems to revise in the environment in {10, 100, 200, 500, 1,000}. We set to the number of training steps per volley at 10,000 for the experiment with 10 poems, and we increase it to 100,000 in the other experiments. Additionally, we considered another experiment, indicated as `dynamic`, in the most difficult scenario, i.e., where the environment spawns new, unseen, artificially generated quatrains at each episode. In such a case we report results of PPO only, because using VPG always resulted in a failure. An episode ends either when the target rhyme scheme is matched or after 30 steps, that corresponds to the maximum episode length. Therefore, the reward of an episode ranges in the interval $[-30, 1]$. Differently from our previous work [19], we do not carry out human evaluations, since rhyme matching can be quantitatively measured through the reward. Indeed, the reward is a direct way to assess the revised poem quality, because it is proportional to the number of steps needed for adjusting the target rhyme scheme. In particular, from Equation 3.7 we can observe that the number of revising steps in an episode (i.e. where reaching the goal state S_f) is equivalent to $|R_f| + 2$.

Results are presented in Table 3.4. We can see that, while the agent improves the reward in all the experiments with PPO, learning with VPG is not stable, and performs poorly. The superiority of PPO over VPG is also illustrated in Fig. 3.2, where we can see the instability of VPG in contrast to the steady progresses of PPO. Even if the task is very challenging, the model is able to strongly improve the average R score, thus indicating that it is actually moving the right steps in progressively fixing the rhymes. We also report in Table 3.5 and in Table 3.6, two examples of draft revisions obtained

with the agent trained with PPO in the `dynamic` environment.

Table 3.4: VPG vs PPO: Reward on the experiment of Section 3.4.4 with 10, 100, 200, 500 and 1000 poems. PPO is also evaluated with an environment that continuously generates new drafts (`dynamic`).

	N poems	R first Volley	R last Volley
VPG	10	-18.752	-14.630
PPO		-10.239	**-1.186**
VPG	100	-19.415	-19.264
PPO		-15.598	**-5.200**
VPG	200	-20.950	-18.432
PPO		-15.323	**-3.757**
VPG	500	-21.191	-19.623
PPO		-15.043	**-7.780**
VPG	1,000	-26.150	-21.179
PPO		-11.579	**-9.733**
PPO	`dynamic`	-14.796	-12.415

3.5 Discussion

In this chapter we discussed an innovative way of implementing the notion of creativity in a machine. Considering the task of automatically generating new poems, we proposed a model that implements the human-like behaviour of writing a draft and revising it multiple times. We proposed to create drafts that are conditioned to author and rhyme information, while the revision process is built around an iterative procedure that can be described as a navigation problem and solved with RL and PPO, that significantly outperformed VPG. Multiple experiments confirmed that the proposed approach is feasible and that it allows the machine to learn how to revise text, even if it is not explicitly instructed on which portion of text it should revise. The proposed framework is also general enough to be eventually applied to other text generation tasks.

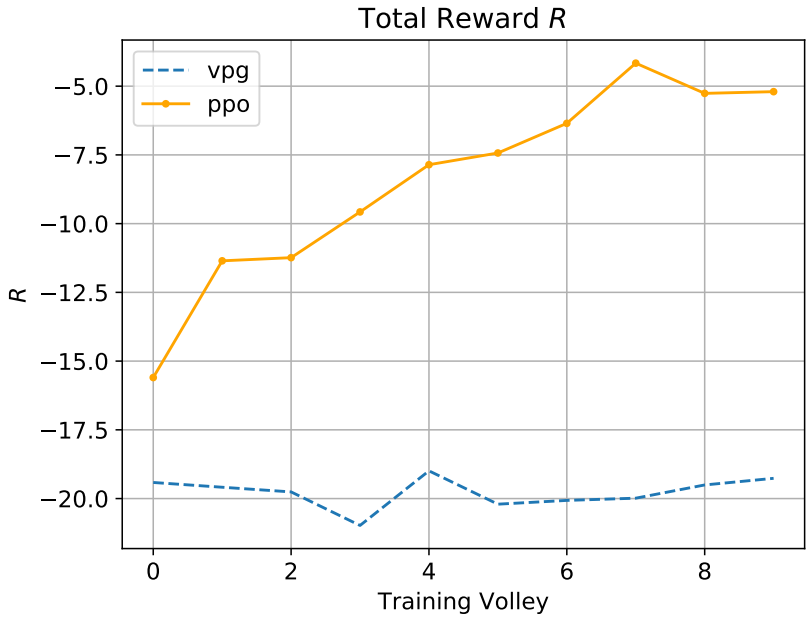


Figure 3.2: Rewards yielded by using PPO and VPG with respect to the number training volleys, in the experiment of Section 3.4.4 with 100 poems in the environment.

Table 3.5: Two examples of generated poems with generate and revise approach given a target rhyme scheme, before the revision iterative steps.

Rhyme scheme	Draft
AABB	*the mist that made us sweat and* ***ache***
	with toil, from doing good or ill,
	the hour when we were led to play
	the children of the people's ***brood****,*
ABBB	*and when, above, the winter's snow*
	has risen in the wintry ***sky***
	and leaves their path to cloud's decay,
	*and life is spent, and life is drear****,***

Table 3.6: Two examples of generated poems with generate and revise approach given a target rhyme scheme, after the revision iterative steps.

Rhyme scheme	Revision
AABB	*the mist that made us sweat and* ***chill***
	with toil, from doing good or ill,
	the hour when we were led to play
	the children of the people's ***way****,*
ABBB	*and when, above, the winter's snow*
	has risen in the wintry night ***away***
	and leaves their path to cloud's decay,
	and life is spent, and life is drear ***today***

Chapter 4

Resource Slicing through Reinforcement Learning

With the advent of 5G and the research into beyond 5G (B5G) networks, a novel and very relevant research issue is how to manage the coexistence of different types of traffic, each with very strict but completely different requirements. This bring us to the third leg of our journey: what could be a better real world application of RL than learning how to manage such complex traffic?

In this chapter we propose a DRL algorithm to *slice* the available physical layer resources between URLLC and eMBB traffic. Specifically, in our setting the time-frequency resource grid is fully occupied by eMBB traffic and we train the DRL agent to employ PPO, a state-of-the-art DRL algorithm, to dynamically allocate the incoming URLLC traffic by puncturing eMBB codewords. Assuming that each eMBB codeword can tolerate a certain limited amount of puncturing beyond which is in outage, we show that the policy devised by the DRL agent never violates the latency requirement of URLLC traffic and, at the same time, manages to keep the number of eMBB codewords in outage at minimum levels, when compared to other state-of-the-art schemes.

4.1 Introduction

Resource slicing of different kinds of traffic is a key enabler for 5G and B5G networks, allowing the coexistence on a common infrastructure of different services with different requirements such as eMBB and URLLC [50]. The two kind of traffic have different quality-of-service (QoS): eMBB users require high throughputs, while URLLC has strict low-latency and reliability constraints [51]. In particular, URLLC traffic is characterized by short packets

that need to be transmitted and decoded in less than 1 ms [52], so that conventional channel-aware scheduling is generally not possible.

Addressing the problem of URLLC-eMBB scheduling, [51] compares the performance of different techniques in the uplink of a 5G system and lays the ground for the subsequent literature using either *puncturing*, orthogonal multiple access (OMA) and non-orthogonal multiple access (NOMA). Immediate scheduling of URLLC packets in combination with hybrid automatic repeat request (HARQ) is another approach investigated in [53]. In [54] eMBB codewords are punctured to accomodate URLLC traffic and the throughput loss for eMBB packets is evaluated under different models. In [55] the authors propose a resource allocation scheme for URLLC-eMBB traffic based on successive convex approximation and semidefinite relaxation of the general optimization problem.

Because of its ability of finding very good to optimal policies for systems that dynamically change through time [1], *Reinforcement Learning* (RL) is a natural choice to address the random dynamics of URLLC traffic. Usually, RL is employed in its *Deep* (DRL) formulation, where a multi-layer NN is employed to extract features from states hardly enumerable in the simpler tabular approaches. Accordingly, in [56], and [57] the authors propose two RL algorithms based on Q-learning to multiplex eMBB and URLLC traffic employing OMA and NOMA, respectively.

In [58] and [59], DRL approaches based on PG algorithms are proposed to choose the resources punctured by the URLLC transmission. The former requires *a-priori* information to the number of resources to be punctured, the latter relies on the instantaneous channel state information (CSI).

Most of the recent literature [51, 54, 56, 57, 58, 59] assumes that the URLLC packets are transmitted as soon as they arrive. However, as long as they satisfy their latency constraints, URLLC packets can tolerate a certain amount of delay and this (minimum) tolerable delay can be exploited to give the scheduler some degree of freedom. In this chapter, we address the slicing problem by allowing some slack for URLLC transmissions to minimize their impact on eMBB traffic. We assume a system where all resources are already allocated to eMBB traffic, so that every time there is a URLLC transmission an eMBB packet needs to be punctured. The URLLC scheduler

is a DRL agent, which select the resources for the URLLC packets with the goal of minimizing the outage of the eMBB traffic due to puncturing. To enforce slice isolation, the control planes of the different slices are kept to a minimum degree of interaction [54], and the URLLC the eMBB schedulers are two independent entities. Accordingly, the only information the URLLC scheduler needs to possess is the robustness of each eMBB codeword to puncturing. Moreover, because of the strict latency requirements, we assume that the URLLC scheduler does not have instantaneous channel state information. While our proposed codeword model resembles the *threshold model* described in [54], it retains two important differences. First, we consider a more realistic non-homogeneous situation where different puncturing policies can be adopted at different times. Then, we consider a threshold per codeword rather than per user.

This chapter is organized as follows. After discussing the system model from a telecommunication perspective (Section 4.2), we discuss the proposed DRL agent and our training approach (Section 4.3). We present our results in Section 4.4, then we explore an expanded approach also considering the reliability of URLLC packets over multiple frequencies (Section 4.5). Finally, we discuss all results in Section 4.6

4.2 Low-Latency Traffic on Narrow-Band System Model

We consider a single cell scenario in which one base station (BS) serves a set of downlink user equipments (UE) with different requirements. We denote as U and E the number of URLLC and eMBB users, respectively. The set of UEs belonging to the URLLC and eMBB slices are referred to as $\mathcal{E}$ and $\mathcal{U}$, respectively. We consider a single coherence interval as time horizon, where the channel can be considered constant. The time axis is divided into Σ equally spaced time slots of fixed duration. To accommodate URLLC traffic, with its strict latency requirements, slots are further divided into M minislots[1]. As for the frequency domain, the system bandwidth is divided into F orthogonal

[1]In 3GPP, the formal term for a "slot" is eMBB Transmit Time Interval (TTI), and a "minislot" is a URLLC TTI [54].

frequency resources (FR)[2].

We consider two different schedulers, one for each type of traffic, which operate separately and independently of each other. The *eMBB scheduler* is responsible for assigning time and frequency resources to eMBB users: each eMBB codeword can occupy any fraction of the total available number of minislots and FRs. As customary, eMBB scheduling is operated at the slot boundaries. At the same time, the *URLLC agent* operates on a per minislot basis with the possibility of puncturing some of the resources already assigned to eMBB users, if needed. In the following, we present a detailed description of how we model the traffics, and their coexistence.

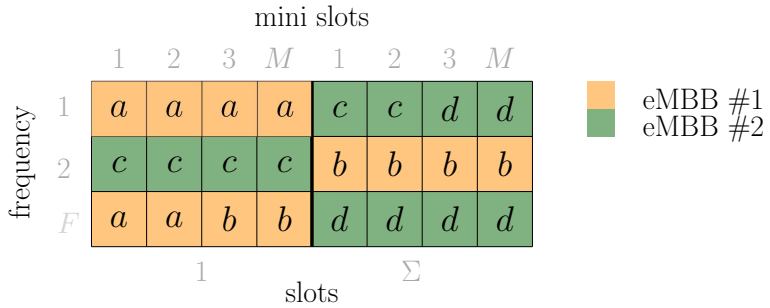


Figure 4.1: Toy example of the resource allocation and codeword placement for the eMBB users, $F = 3$, $\Sigma = 2$, $M = 4$. Resources are allocated at slot boundaries, while codewords are $a, b \in \mathcal{W}_1$, $c, d \in \mathcal{W}_2$ and $|a| = |b| = |c| = |d| = 6$.

4.2.1 The eMBB Scheduler

In this chapter, we do not explicitly address the eMBB scheduling problem, but, rather, we assume that a proper radio resource allocations has been performed somehow and we can focus on the coexistence of URLLC traffic on top of eMBB. Nevertheless, we need to describe the main principles of the eMBB scheduling policy, which is to maximize a rate-dependent utility function, not considering any latency. Hence, radio resources are allocated to the set of active users on a slot basis following the OMA paradigm. Moreover, since

[2]With "frequency resources" we refer to the abstract concept of bandwidth available in an OFDM system and we may refer to resource blocks or subcarriers, indifferently.

there is enough time to exchange channel quality information (CQI) before each scheduling decision, it is reasonable to assume perfect knowledge of CSI at the BS. Therefore, eMBB resource allocation can be performed following conventional methods such as the water-filling algorithm [60].

The scheduler has to further take into account that the eMBB packets might share the radio resources with URLLC traffic and in such event they should carry enough redundancy to be punctured without losing the entire packet. We denote as $\mathcal{W}$ the codebook at the BS. The BS will then select a subset $\mathcal{W}_e \subset \mathcal{W}$ containing all the codewords of user e. A single codeword intended for user e is denoted as $w \in \mathcal{W}_e$. The length in symbols $|w|$ of a codeword is always a multiple of the minislot length, i.e., each codeword spans an integer number of minislots. Finally, we denote with $w_{t,f}$ the codeword transmitted on the radio resource f during minislot t and with $\mathcal{W}_t = \bigcup_{\nu=1}^{F} w_{t,\nu}$ the set of all codewords transmitted during the minislot t. Figure 4.1 shows a toy example of a possible resource allocation and codeword placement for two eMBB users.

4.2.2 The URLLC agent

Generally speaking, the QoS requirements of an URLLC user $u \in \mathcal{U}$ in a wireless network are specified as follows: a packet of size N_u bits must be successfully delivered to the receiver within an *end-to-end delay* of no more than $T_u^{\max}$ seconds with a probability of at least $1 - \epsilon_u$ [53]. Moreover, a URLLC packet may randomly arrive at the BS at any moment. In this chapter we will concentrate on the edge delay, i.e. the delay computed as the difference between the time the scheduler receives the packet and the time the packet is transmitted. This choice is justified by the fact that the backhaul delay is generally negligible [61], while UL queuing delay and transmission delay can be taken into account by reducing the value of the tolerable latency $T_u^{\max}$. Without loss of generality, we define the tolerable latency in terms of the maximum number $l_u^{\max}$ of minislots that can be waited before exceeding the latency constraint.

To simplify the description of the problem, we will focus on URLLC packets of fixed length corresponding to a single minislot. However, the considered framework can be easily extended to the case where a packet occupies more

than one minislot. The packets are generated following memory-less packet arrival distributions: Bernoulli with arrival probability p_u and Poisson with mean value λ_u. The packets are stored in a first-in first-out (FIFO) queue $\mathcal{Q}$ of infinite length. The URLLC agent is responsible for taking the decision whether the oldest packet in the queue should be transmitted or not in the current minislot, and onto which frequency resource in the grid.

Owing to the strict latency constraint, the CSI of URLLC users cannot be estimated. Hence, power adaptation during transmission is not possible and ARQ re-transmission mechanisms can be hardly acceptable. Accordingly, reliability can be expressed in terms of outage probability for a fixed predefined transmit power. As in previous works in literature [54], we assume that the URLLC transmit power is large enough so that the outage probability remains under an acceptable threshold.

4.2.3 URLLC and eMBB Coexistence

Coexistence of eMBB and URLLC is achieved by superposition coding or puncturing [50]-[54]. In this chapter we consider a puncturing strategy, where the BS decides to use a certain resource for URLLC traffic regardless of any eMBB user already occupying it. To avoid any interference between the two types of traffic, the eMBB codeword is punctured, i.e., the transmit power of the eMBB user on the specific resource is set to zero. To tolerate puncturing, we assume that each eMBB codeword employs an inner erasure code with rate $1 - C_w/|w|$ [51], that allows to correct up to C_w erased minislots. The eMBB scheduler is in charge of determining the *class* C_w for each codeword. The class assignment is performed on a codeword basis, i.e., the BS can assign codewords with different C_w to the same user. The assignment of higher C_w to different codewords encompasses the possibility of employing a more robust transmission mechanisms to prevent outage even in the presence of puncturing, as discussed in Section 4.2.1. Of course, the higher C_w the lower the transmission rate. Note that the algorithm implemented by the eMBB scheduler may be unknown by the URLLC traffic agent, as long as the latter is informed of the codewords class by the former.

4.3 The DRL Agent

RL is usually employed to solve a MDP defined over a real world task. It is common knowledge in literature that RL is especially effective when paired with parametric function approximators for the policy $\pi(a|s)$, specifically multi-layer NNs (hence the name *deep*). That holds true especially for each task where the state space is too complex to be represented in tabular form, making it impossible for the algorithm to compute the estimated value of each state in a reasonable time [1]. Since the simulation's state observed by the proposed URLLC agent at each time step t is of combinatorial complexity, as shown in section 4.3.1, in this chapter we follow the DRL paradigm when training the agent, according to literature best practices.

4.3.1 System Model as a MDP

The application of RL to our task requires to formulate the URLLC scheduling problem as a MDP. Despite the task at hand being inherently not episodic, for convenience of operation we truncate it in multiple episodes of length T minislots, corresponding to the whole coherence interval of the channel. A minislot $t \in \{1, \dots, T\}$ represents a time step in the episode. At the beginning of each episode, resource allocation and codeword placement for eMMB users is performed and then, at each time step, new URLLC packets are generated according to a certain distribution.

The DRL action consists in deciding whether the first URLLC packet in the queue should be transmitted in the current minislot or not and on which FR. The *possible actions* at time step t are collected in the set $\mathcal{A}_t = \{0, 1, \dots, F\}$, where 0 means no transmission, while otherwise the action indicates the FR index for transmitting the URLLC packet. If the URLLC queue is empty, the only possible action is 0.

The *state* at each time step t is then represented by the set $\mathcal{S}_t = \{\mathcal{S}_t^{(u)}, \mathcal{S}_t^{(e)}\}$, where $\mathcal{S}_t^{(u)}$ and $\mathcal{S}_t^{(e)}$ collect the URLLC and eMBB information at step t, respectively. In particular, the 2-dimensional state $\mathcal{S}_t^{(u)}$ is

$$\mathcal{S}_t^{(u)} = \{Q_t, \Delta_t\} \tag{4.1}$$

where Q_t represents the length of the URLLC queue at step t, while $\Delta_t = l_u^{\max} - l_t^{\text{old}}$ represents the difference between the tolerable latency and the

latency of the oldest packet in the queue at step t. The F-dimensional state $\mathcal{S}_t^{(e)}$ collects for each of the F frequency channels the variable $s_t(f)$, which tracks if the codeword transmitted on channel f is in outage ($s_t(f) = -1$) or not ($s_t(f) \geq 0$). A non-negative $s_t(f)$ stores the residual number of times that the codeword can be punctured without being in outage. Let $\rho_t(w)$ denote the number of times the codeword w has been punctured from the beginning of the episode, $s_t(f)$ is computed as

$$s_t(f) = \max\left\{C_{w_{t,f}} - \rho_t(w_{t,f}), -1\right\}, \tag{4.2}$$

remembering that $w_{t,f}$ is the codeword placed on resource f and minislot t. Once a codeword is in outage, its state variable is set to -1 and does not change anymore regardless of the times is further punctured.

4.3.2 Reward Function

In a RL problem choosing the reward is an empirical process: a good reward function should capture the essence of the task at hand. In this case the objective is to minimize the number of eMBB codewords in outage while keeping the latency of URLLC packets below the given threshold. With this goal in mind, we introduce the eMMB penalty function $e_t(w)$

$$e_t(w) = \begin{cases} -1, & \mathcal{C}_w - \rho_{t-1}(w) \geq 0 \cap \mathcal{C}_w - \rho_t(w) < 0, \\ 0, & \text{otherwise}, \end{cases} \tag{4.3}$$

which takes value -1 only if the chosen action causes the outage of the codeword w. Furthermore, since $\Delta_t < 0$ signals the violation of the latency constraint, we introduce the following URLLC penalty function

$$L_t = \begin{cases} 0, & \Delta_t \geq 0, \\ -\frac{\alpha T}{F+1}, & \Delta_t < 0. \end{cases} \tag{4.4}$$

The heuristic value $-\frac{\alpha T}{F+1}$ is empirically chosen so that the violation of the latency constraint for an URLLC packet results in a larger negative contribution than the outage penalty for eMBB traffic. In the worst case, every punctured resource will lead to an outage event and the total maximum contribution of the eMBB penalty is $-T$. However, T is a large value which can

lead to numerical instabilities within the learning process, and it is not highly probable that an eMBB outage event occurs every step. Hence, we normalize T by $(F+1)/\alpha$ ($\alpha \geq 1$) which is a term that accounts for to the fact that the larger is the number of frequency resources, the less probable are outage events. The value of α is devised by means of trial and error, according to best practices of reward engineering in RL [1]. Finally, the reward at time t can be expressed by

$$R_t = \sum_{w \in \mathcal{W}_t} e_t(w) + L_t, \tag{4.5}$$

Eventually, when $\Delta_t < 0$ the episode is considered finished.

4.3.3 Algorithm and Neural Network Architecture

Among the possible DRL techniques, we consider the PG algorithm PPO [21], described in this thesis in Chapter 3, specifically in Section 3.3.2. PPO aims at taking the biggest possible improvement step on a policy without ending too far from the previous one, thus avoiding the risk of performance collapse. While we know that many papers in literature focus on well known methods as Deep Q-Learning (DQL) or VPG, both presents huge drawbacks w.r.t. PPO. Specifically, DQL cannot scale to large action spaces required by complex tasks as the one at hand, while VPG is extremely unstable during training. PPO is considered state-of-the-art for PG methods in current RL literature, by being an efficient approximation of TRPO [48], the latter being proved to guarantee a monotonic improvement over the expected discounted reward w.r.t. the training steps.

PPO is an actor-critic algorithm [21], where two different neural networks are required. To this respect, we consider two completely separated subnetworks, one for the *value function* (the critic, with output estimated value of current state) and one for the *policy function* (the actor, with output the current strategy). Both policy and value function subnetworks have three dense layers with 128, 64, and 32 neurons, respectively. All of them operate a rectified linear activation function (ReLU). Furthermore, the policy subnetwork has a dense fourth layer with $F+1$ neurons to choose the actions, while the value subnetwork has a dense fourth layer with 1 neuron and no activation to estimate the value. Finally, all layers are initialized using Xavier initialization.

4.4 Results

We consider a scenario, where the slot duration and the coherence time of the channel are set to 1 and 10 ms, respectively. Each slot is further divided in $M = 14$ minislots. The number of frequency resources is $F = 12$. The length of an episode corresponds to the coherence time of the channel so that the number of time slots for each episode is $\Sigma = 10$, for a total of $T = 140$ minislots. We consider a single URLLC user, i.e. $U = 1$, and the number of eMBB users is $E = 10$. We further set the maximum delay constraint to $l_u^{\max} = M/2 = 7 = 0.5$ ms. We consider only codewords of class $C_w \in \{0, 1\}$, i.e., codewords that can be punctured zero or one times before being in outage.

Regarding PPO, we use an instance of PPO-Clip with a clip ratio equal to 0.2, and an early stopping strategy if the mean KL-divergence of the new policy from the old one grows beyond a given threshold, set as $1.5 \cdot 10^{-2}$, as described in [62]. To reduce the variance of the states' values stored into the various trajectories we feed the NN with, we use the generalized advantage estimation approach with $\gamma_{\text{GAE}} = 1$ and $\lambda_{\text{GAE}} = 0.97$, as proposed in [47]. The value of α in (4.4) is 3.

To have a fair performance comparison, we consider four alternative URLLC scheduling algorithms:

- *Aggressive.* The URLLC packet is transmitted immediately on a randomly chosen frequency.
- *Threshold Proportional (TP).* The URLLC packet is transmitted immediately on the frequency resource occupied by the codeword with the highest puncturing threshold, given by (4.2). TP has almost optimal performance when the URLLC is forced to transmit immediately upon arrival, i.e., $l_u^{\max} = 1$, and in case of low average URLLC load [54].
- *TP-lazy.* As long as $\Delta_t > 0$, the packet is transmitted only if $\sum_{w \in \mathcal{W}_t} C_w - \rho_t(w) \geq \sum_{w \in \mathcal{W}_{t+1}} C_w - \rho_t(w)$, i.e. if the present state is somehow better (or equal) than the next one. If $\Delta_t = 0$, the transmission is forced in the present minislot. In any case, the choice of the frequency is made according to the TP scheme. This heuristic combines the advantage of the TP transmission policy with the possibility of waiting before puncturing eMBB resources.

- *TP-smart.* This heuristic immediately transmits the URLLC packet. If there is an eMBB codeword already in outage the scheduler transmits on the resources occupied by that codeword, otherwise TP-smart acts as TP. Note that this scheme results optimal, in terms of impact on eMBB codewords, for immediate transmissions in this scenario.

Table 4.1: Total reward versus activation probability p_u.

p_u	0.1	0.2	0.3	0.4	0.5	mean
aggressive	-4.60	-9.04	-14.01	-18.55	-23.34	-13.91
TP	-1.31	-3.23	-6.71	-9.85	-15.78	-7.38
TP-lazy	-1.18	-3.18	-5.72	-9.25	-14.63	-6.79
TP-smart	-0.77	-1.64	-2.34	-3.28	-4.23	-2.43
PPO	-0.48	-1.18	-1.955	-2.69	-3.77	-2.03

During the *learning phase* of the PPO agent, the parameters related to eMMB resource allocation and URLLC traffic generation are randomized on a per episode basis. While this is not mandatory to train a functioning agent, it is crucial to help the agent to generalize. In other words, the agent is trained to learn a generalized strategy that is not specific either for a particular eMMB allocation policy or a particular URLLC traffic load. While this is not mandatory to train a functioning agent, it is crucial to help the agent to learn a generalized strategy that is not specific either for a particular eMMB allocation policy or a particular URLLC traffic load. Since the MDP formulation considered in this chapter models a not episodic task, to compute the expected discounted reward we set $\gamma = 0.99$. To further simulate a continuous task, we initialize each episode with a random number of URLLC packets in the queue. The number of packets generated in this way is always smaller than $l_u^{\max}$ to avoid that the episode starts with $\Delta_t < 0$. After training the agent with Bernoulli distribution of packets, we show the results obtained by running the RL-based agent in *inference mode*, providing comparisons with the considered heuristic schemes.

Table 4.2: Average number of URLLC packets not served before the end of the episode.

p_u	0.1	0.2	0.3	0.4	0.5
TP-lazy	0.597	1.222	1.790	2.416	3.015
PPO	0.030	0.068	0.081	0.136	0.210

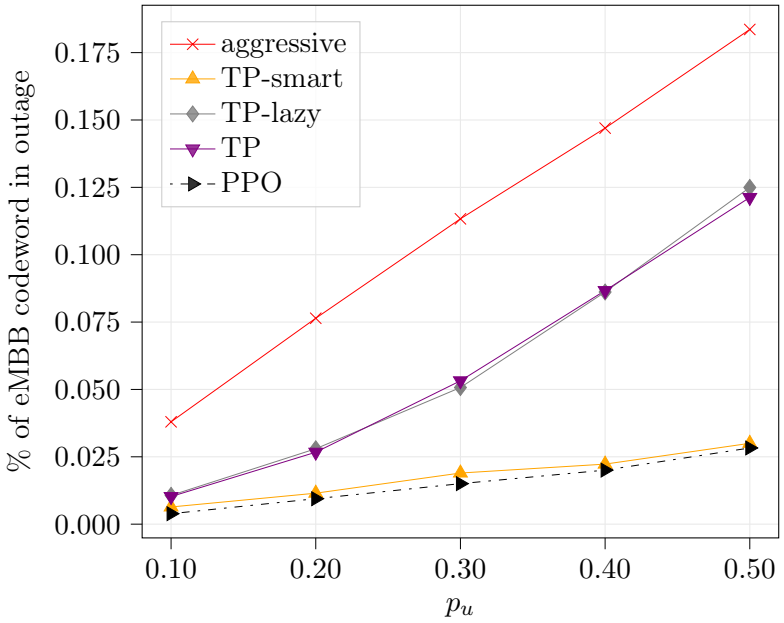


Figure 4.2: Percentage of eMBB codeword in outage versus activation probability p_u, $T = 1400$.

4.4.1 Bernoulli Distribution

Here we discuss the results obtained for the Bernoulli distribution.

Table 4.1 shows the total episode reward $\sum_{t=1}^{T} R_t$ as a function of p_u, for $T = 140$. It is worth noting that the PPO agent trained as proposed in this chapter, outperforms all the other schemes for every value of p_u. In Table 4.2, we show the average number of packets remaining in the URLLC queue at the end of each episode for different p_u. The results for aggressive, TP-smart and TP are omitted since the URLLC packets are promptly transmitted upon

arrival. At the opposite, with the TP-lazy scheme a non-negligible amount of traffic remains unserved at the end of an episode. We can see that the PPO agent is able to devise a new policy in-between the two. It is worth noting that while TP, TP-lazy, aggressive and TP-smart are all designed to to *never violate the URLLC latency constraints* for Bernoulli distribution, the PPO agent learns this on its own.

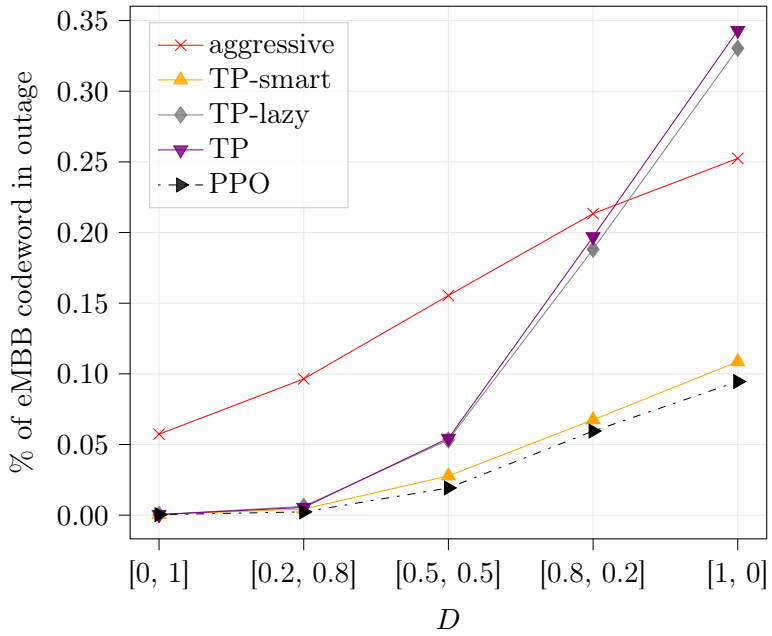


Figure 4.3: Percentage of eMBB codewords in outage versus the different percentage of classes of codeword for probability of activation $p_u = 0.3$.

The subdivision of the task into episodes may somehow distort the correct evaluation of the algorithms' performance. To simulate a longer time horizon is of critical importance to the real word task. To address this issue, we repeated our tests scaling up the length of each episode by one order of magnitude, *without retraining the agent.*

Fig. 4.2 shows the percentage of eMMB codewords in outage at the end of each episode for the various schemes, with $T = 1400$, while the class of each codeword is again randomly chosen. Also in this case, the PPO agent outperforms all the other schemes.

Finally, Fig. 4.3 shows the percentage of eMBB codewords in outage for different compositions of eMBB codewords classes $D = [\Pr\{\mathcal{C}_0\}, \Pr\{\mathcal{C}_1\}]$ and $p_u = 0.3$. The PPO agent outperforms all the heuristic schemes, including the one we considered optimal at one step. Among the other things, we can see that PPO has good performance even when $D = [1, 0]$, i.e. there are only codewords without an inner erasure code.

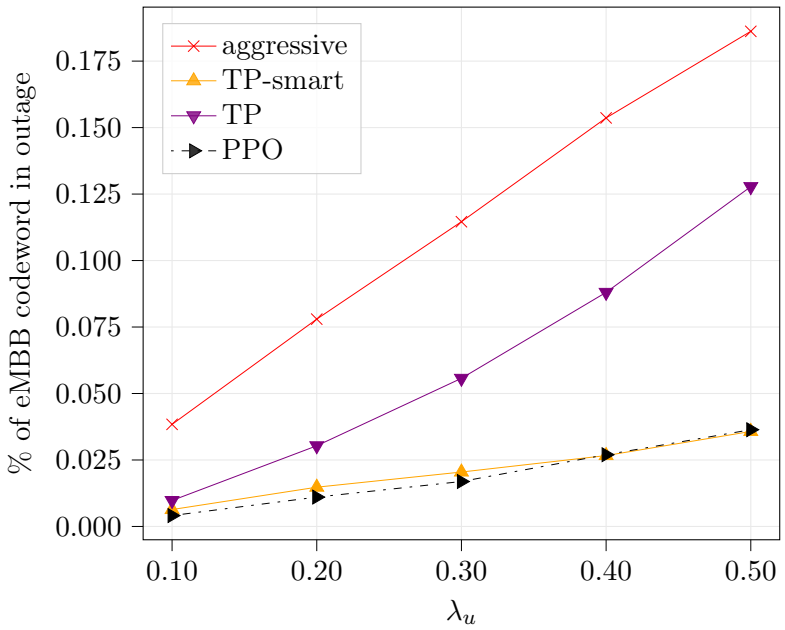


Figure 4.4: Percentage of eMBB codeword in outage versus Poisson rate λ_u.

4.4.2 Poisson Distribution

In this section, we discuss the results obtained for the Poisson distribution, collected for $T = 140$. To assess the generalization capabilities of PPO, we use the same network trained on the environment with Bernoulli distribution.

In Fig. 4.4, we show the performance of the various schemes in terms of eMBB codewords' outages. Also in this case, the proposed PPO agent attains best performance. It is worth noting that TP-lazy heuristic scheme is omitted since it is not able to deal with the bursts of packets generated under the

Poisson assumption, thus consistently violating URLLC latency requirements. All other schemes, *PPO included*, never violate them.

4.5 Towards Reliability and Multi-Frequencies Communication

The narrow-band system model is by construction limited by the amount of frequencies the scheduler can transmit over concurrently. While having only one frequency is really helpful to simplify the real world model to be easier to approach for RL methods, it does exclude a very important requirement of URLLC traffic: reliability.

In order to properly fulfill the reliability requirement, the agent must select a number of resources able to support the communication rate given by the number of bits to be transmitted. Assuming that N informative bits are encoded and transmitted onto the set $\mathcal{F}_u \subseteq \mathcal{F}$, having cardinality $|\mathcal{F}_u| = F_u$, an ϵ_u-reliable transmission occurs if the probability of outage p_u is lower or equal ϵ_u. Having assumed a parallel channel environment, the probability of outage can be expressed by [63]

$$p_u(N, F_u, \Gamma_u) = \Pr\left\{ \sum_{f \in \mathcal{F}_u} \log_2\left(1 + \gamma_u(f)\right) \leq \frac{N}{\Delta_f T_m} \right\} \leq \epsilon_u \tag{4.6}$$

where $\frac{N}{\Delta_f T_m}$ [bit/s/Hz] represents the spectral efficiency of the transmission onto F_u frequencies. In relation (4.6), we highlighted the URLLC outage probability dependence on N, F_u, and Γ_u. We remark that Γ_u is given by the path loss of URLLC user, its receiver noise figure, and the available power of the transmitter. To transmit the highest number of bits, the best solution is employing all the available power for the transmission. Assuming a fixed fraction of power is reserved for URLLC communication, the value of Γ_u is user dependent and cannot be optimized further by the agent.

Having assumed i.i.d. instantaneous SNR realization with mean value Γ_u, the maximum N depends on the cardinality of set $\mathcal{F}_u$. Higher the cardinality F_u, higher will be the maximum number of bits transmissible. Selected the

value of F_u, the maximum number of bits can be evaluate solving

$$N^*(F_u, \Gamma_u) = \max \{N \,|\, p_u(N, F_u, \Gamma_u) \leq \epsilon_u\} \tag{4.7}$$

which provides a unique solution, due to the monotonicity of the outage probability [64, Proposition 1]. Therefore, the relation between F_u and $N^*(F_u)$ is an injective function: giving the former will give a precise information on the latter. At every mini-slot, the URLLC agent is responsible for taking the decision whether the oldest packet should be transmitted or not in the current mini-slot, and onto which frequencies the transmission should be make.

The optimization problem (4.7) cannot be solved in a closed form, taking into account that the analytic formulation of the outage probability in a parallel channel environment is not known; bounds exist, e.g. [65, 66], but they are not easy to be addressed in an optimization problem, involving non-convex special functions. However, it is easy to find the value of $N^*(F_u, \Gamma_u)$ by means of numerical simulations. Let us denote the estimated outage probability obtained through Monte Carlo simulation as

$$\hat{p}_u(N, F_u, \Gamma_u) = \lim_{n\to\infty} \frac{1}{n} \sum_{i=1}^{n} \mathbb{1}\left\{ \Delta_f T_m \sum_{f\in\mathcal{F}_u} \log\left(1 + \gamma_u(f)\right) \leq N \right\} \tag{4.8}$$

where $\mathbb{1}(\cdot)$ is the indicator function. Keeping fixed the values of F_u and Γ_u, is it possible to find N^* using an iterative exhaustive search method: at iteration t, we select a value of $N^{(t)}$; we estimate the outage probability as (4.8); if $\hat{p}_u(N^{(t)}, F_u, \Gamma_u) \leq \epsilon_u$, we select a new value of $N^{(t+1)} = N^{(t)} + 1$ repeating the process; else, if $\hat{p}_u(N^{(t)}, F_u, \Gamma_u) > \epsilon_u$, we set $N^*(F_u, \Gamma_u) = N^{(t-1)}$. Repeating this procedure for all possible values of interest of F_u and Γ_u, we obtain a lookup table comprehending all the optimal $N^*(F_u, \Gamma_u)$.

Given the aforementioned lookup table, it is possible to define a greedy algorithm to approach this kind of scenario, described in the following subsection. We remark that the DRL approach described in 4.5.2 is also able to reduce the size of the entry in the lookup table.

4.5.1 Greedy Algorithm

The aim of this algorithm is to empty the queue $\mathcal{Q}$ as fast as possible, puncturing the minimum number of resources. Moreover, we also aim to minimize

the impact on the eMBB scheduled codeword.

Defining as $Q(t)$ the length of the queue in bits at mini-slot t, the number of resources selected for the communication is given by

$$F_u = \begin{cases} \min \{F_u \mid N^*(F_u, \Gamma_u) \geq Q(t)\}, \text{ if } \exists N^*(F_u, \Gamma_u) \geq Q(t), \\ F \text{ otherwise.} \end{cases} \tag{4.9}$$

In other words, we selected the entry of the lookup table that guarantees that the queue is emptied by the largest amount of bits.

To reduce the impact of the eMBB codeword, we select the frequencies to be punctured taking into account the puncturing state of the codewords and their duration at mini-slot t. We remark that every codeword already in outage can be punctured without reducing the overall performance of the network. Beside codewords in outage, every $f \in \mathcal{F}$ having protection is an high priority frequency to be punctured. On the other hand, if two codewords are not protected and not in outage, the priority is given to the resources having lower remaining duration. This choice is made according to the system behaviour: all codewords contain the same information; hence, causing outage will impact in the same way the overall eMBB traffic, no matter the codeword chosen. However, causing outage on a longer codeword will free more resources to be punctured in the following mini-steps, giving space to avoiding further outages in the future. It is worth remarking that the same procedure can be applied when the codewords transport different number of data bits. In that case, the duration could be weighted by taking into account the number of informative bits contained in a codeword.

4.5.2 Multi-Frequencies DRL Agent

In a multi-frequencies setting, the DRL action consists in deciding how many bits of URLLC packets in the queue should be transmitted in the current mini-slot and over how many frequencies. Even if we are to transmit one packet at a time as in the mono-frequency approach, transmitting it over multiple frequencies causes the size of the action set to grow according to a combinatorial rule. Specifically, the size of $\mathcal{A}$ is 2^F, with F the amount of frequencies and $\mathcal{A}$ the action set defined as follows

$$\mathcal{A} = \bigcup_{f=1}^{F} \{1_f, 0_f\}$$

where 1_f is the action of transmitting over f frequency and 0_f is the action of not transmitting over f frequency. It is evident that no transmission at time step t is equal to

$$a_t = \{0_0, 0_1, 0_2, \ldots, 0_F\}$$

While transmitting over all frequencies at time step t is equal to

$$a_t = \{1_0, 1_1, 1_2, \ldots, 1_F\}$$

If the URLLC queue is empty, the only possible action is $\{0_0, 0_1, 0_2, \ldots, 0_F\}$. Since we also need to define how many bits of the URLLC packets to transmit at each time step t, it is necessary to rethink this formulation. Specifically, we explored two possible solutions:

- Employ a continuous action space
- Employ a hierarchical approach

Note that we keep both the NN and the DRL algorithm used to train it unchanged from the approach previously described. For the reward we keep the same overall concept but we apply some minor changes, as follows. The new eMMB penalty function now depends on $\mathcal{W}_e$, i.e. the entire set of codewords for user e. We can define it as $e_t(\mathcal{W}_e)$ is now defined as

$$e_t(\mathcal{W}_e) = \begin{cases} -\hat{o}_N, & \hat{o} > 0, \\ 0, & \text{otherwise}, \end{cases} \tag{4.10}$$

where $\hat{o}$ is the number of codewords $w \in \mathcal{W}_e$ in outage because of the action at time step t. In this context $\hat{o}_N$ is the normalized amount of outages w.r.t. the amount of unique codewords available in $\mathcal{W}_e$ at time step t

$$\hat{o_N} = \frac{\hat{o}}{card(\mathcal{W}_e(t))}$$

For what concerns the URLLC penalty function we define

$$L_t = \begin{cases} 0, & \Delta_t \geq 0, \\ -10, & \Delta_t < 0. \end{cases} \tag{4.11}$$

The heuristic value 10 is empirically chosen so that the violation of the latency constraint for an URLLC packet results in a larger negative contribution than the outage penalty for eMBB traffic without steering too much the agent towards a completely aggressive policy. Like in the narrow-band setting, the reward at time t can be expressed by

$$R_t = \sum_{t \in T} e_t(\mathcal{W}_e) + L_t, \tag{4.12}$$

where we show the dependency of the eMBB penalty function from the entire set of codewords of user e at time step t. Finally, we only terminate the episode when $t = T$.

4.5.3 MDP with Continuous Action Space

If we employ a continuous action space, the DRL action is now defined as

$$\mathcal{A} = \bigcup_{f=1}^{F} \{n_f\}$$

where n_f is the action of transmitting a certain amount n of information over f frequency. If $n = 0$ no transmission occurs over that frequency. n is a real number corresponding to fraction representing a certain amount of bits. While the amount of bits is always an integer number, it is easier to represent for the DRL agent using a real value. The value n can indeed be easily converted to the amount of bits at any time, according to the size of the URLLC packets and the size of the queue.

The *state* at each time step t is then represented by the set

$$\mathcal{S}_t = \{\mathcal{S}_t^{(u)}, \mathcal{S}_t^{(e)}, \mathcal{S}_t^{(c)}, \mathcal{S}_t^{(x)}\},$$

where $\mathcal{S}_t^{(u)}$ and $\mathcal{S}_t^{(e)}$ collect the URLLC and eMBB information at time step t, respectively, while $\mathcal{S}_t^{(c)}$ and $\mathcal{S}_t^{(x)}$ collect the codewords encoding and

expiration information at time step t. In particular, the 2-dimensional state $\mathcal{S}_t^{(u)}$ is

$$\mathcal{S}_t^{(u)} = \{Q_t, \Delta_t\} \tag{4.13}$$

where Q_t represents the length of the URLLC queue at step t, while $\Delta_t = l_u^{\max} - l_t^{\text{old}}$ represents the difference between the tolerable latency and the latency of the oldest packet in the queue at step t. The F-dimensional state $\mathcal{S}_t^{(e)}$ collects for each of the F frequency channels the variable $s_t(f)$, which tracks if the codeword transmitted on channel f is in outage ($s_t(f) = -1$) or not ($s_t(f) \geq 0$). A non-negative $s_t(f)$ stores the residual number of times that the codeword can be punctured without being in outage. Let $\rho_t(w)$ denote the number of times the codeword w has been punctured from the beginning of the episode, $s_t(f)$ is computed as

$$s_t(f) = \max\left\{C_{w_{t,f}} - \rho_t(w_{t,f}), -1\right\}, \tag{4.14}$$

remembering that $w_{t,f}$ is the codeword placed on resource f and mini-slot t. Once a codeword is in outage, its state variable is set to -1 and does not change anymore regardless of the times is further punctured.

The F-dimensional state $\mathcal{S}_t^{(c)}$ represents each eMBB codeword on each frequency using one-hot encoding. This allows the DRL agent to see how the eMBB codewords are distributed over the frequencies. Finally, the F-dimensional state $\mathcal{S}_t^{(x)}$ defines how many time steps are left until each codeword expires over all frequencies.

4.5.4 Hierarchical MDPs

If we opt for an hierarchical approach, we need to divide the problem into two tasks. The first task is to schedule over the frequencies, i.e. the agent choosing over which frequencies to transmit. The second task is to schedule over the URLLC packets in the queue, i.e. the agent choosing how much of the queue to transmit in a certain time step. It is possible to define an heuristic which determines how many packets of portions of packets to transmit depending on how many frequencies are used. Because of that, the second agent task is to decide how many frequencies to transmit over at each time step. Finally, we have two different agents, each one with its own action space and state

space. This subdivision allows us to keep both action spaces simpler than the continuous one, i.e. they are discrete action spaces.

Frequencies Scheduler The frequencies scheduler is a MDP where the agent schedules the URLLC packets over a definite set of frequencies F, i.e. decide over which frequencies $\hat{F}$ to schedule among the entire set of frequencies F at a certain time step t. Specifically, $\hat{\mathcal{F}} \subset \mathcal{F}$ and the cardinality of $\hat{\mathcal{F}}$ is defined at the beginning of each episode. For this agent it is impossible to stay idle at any time step t^{fs}, and it exists in a virtual time-frame w.r.t. the time steps of the general MDP. Specifically, its episode is as long as the amount of frequencies it needs to schedule over, i.e. the cardinality of $\hat{F}$:

$$T^{fs} = |\hat{\mathcal{F}}|$$

where T^{fs} is the length of the episode for the frequencies scheduler. The action set of the frequencies scheduler is very similar to the action set presented in in the narrow-band approach, specifically the action consists in deciding on which frequency to transmit at each time step. The *possible actions* at time step t are collected in the set $\mathcal{A}_t^{fs} = \{1, \dots, F\}$, where the number indicates the frequency index for transmitting the URLLC packet.

The *state* at each time step t^{fs} is then represented by the set

$$\mathcal{S}^{fs}_{(t,t^{fs})} = \{\mathcal{S}^{(e)}_{t^{fs}}, \mathcal{S}^{(x)}_{t}\}$$

where $\mathcal{S}^{(e)}_{t^{fs}}$ collects the eMBB information at time step t^{fs} and $\mathcal{S}^{(x)}_{t}$ collect the expiration information at time step t. Please note that the expiration information remains the same for the entire episode, thus it only depends on the general time step t, while eMBB information changes during the episode, thus depending on virtual time step t^{fs} Both $\mathcal{S}^{(e)}_{t^{fs}}$ and $\mathcal{S}^{(x)}_{t}$ are defined as in the continuous action space approach described above.

Packets Scheduler The packets scheduler is a MDP where the agent schedules how many frequencies to use at a certain time step t. Given the amount of frequencies F, it is possible to deterministically compute the amount of packets to transmit at the associated time step. The episode and time-frame of

the packets scheduler coincide with the episode and time-frame of the general MDP, and we can think of this MDP as the general MDP.

The *possible actions* of the packets scheduler at time step t are defined as:

$$\mathcal{A}_t^{ps} = \{0, 1, \ldots, F\}$$

where 0 means to not transmit any URLLC packet, while $1, \ldots, F$ means to transmit over 1 or more, up to F, frequencies. Packets to transmit are then computed accordingly. When this decision is made, the frequencies scheduler acts for as many virtual time steps t^{fs} as the value chosen by the packets scheduler action. For example, if the action $a_t = 3$ then the URLLC packets are to be transmitted over 3 frequencies. As a consequence, the frequencies are chosen by the frequencies scheduler by acting for a total of 3 virtual time steps t^{fs}.

The *state* at each time step t is then represented by the set

$$\mathcal{S}_t^{ps} = \{\mathcal{S}_t^{(u)}, \mathcal{S}_t^{(e)}, \mathcal{S}_t^{(x)}, \mathcal{S}_t^{(r)}\}$$

where $\mathcal{S}_t^{(u)}$ and $\mathcal{S}_t^{(e)}$ collect the URLLC and eMBB information at time step t, respectively, while $\mathcal{S}_t^{(x)}$ collect the expiration information at time step t. All the first three elements in the set $\mathcal{S}_t^{ps}$ are defined as in the continuous action space approach described above, while the $\mathcal{S}_t^{(r)}$ represent the maximum number of bits transmissible for each number of channel selected or employing all frequencies. The former state is a vector of size F and takes the name of *full rate*, while the latter is a single number, taking the name of *max rate*.

4.5.5 Results

While interesting in theory, the continuous action space approach described in 4.5.3 does not work well in practice. Multiple experiments show how hard is for the DRL algorithm to converge to a reasonable solution, even if largely suboptimal. Indeed, from our experiments it is evident that PPO, and PG algorithms in general, are very unstable when the action space is continuous.

In literature, continuous action spaces are often used for navigation or mechanical tasks where outputs are constrained within really small real values and where even a minimal variation of one of the real values in the actions

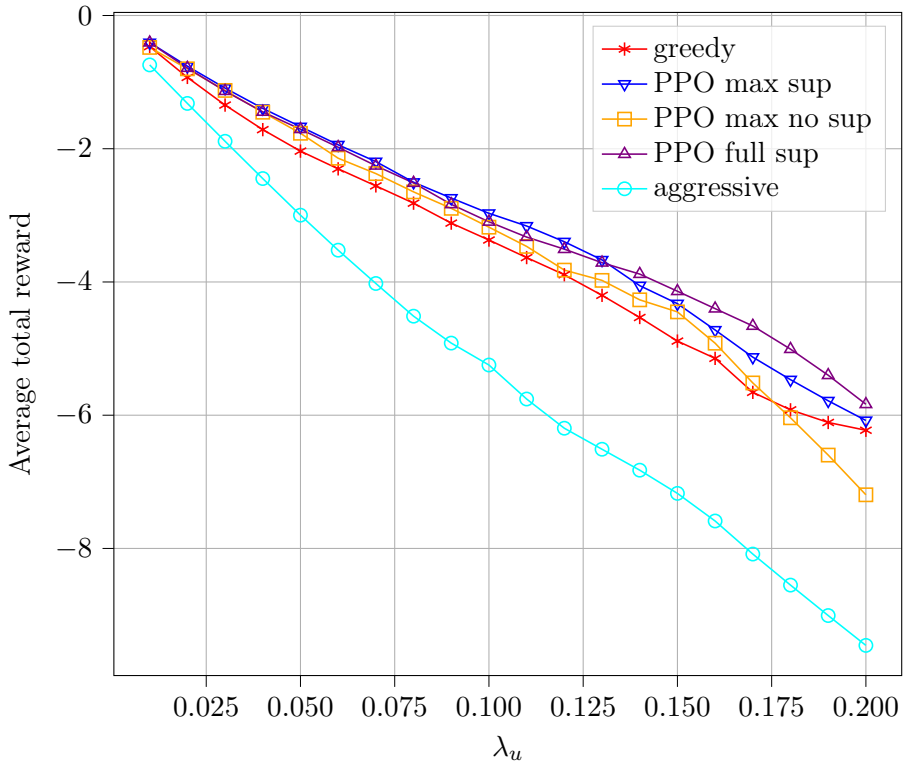


Figure 4.5: Multi-frequencies setting: average total reward versus arrival rate λ_u for regular training period

vector causes a relevant consequence for the environment. Often, such values also change gradually. In our setup, the real values within the actions vector vary a lot at each time step, for they represent the amount of bits in the queue to transmit over each frequencies. Beside that, unstable normalization is required to make sure the amount of bits transmitted never exceeds the amount of bits in the queue. These two factors alone make the continuous approach we presented in this chapter unable to solve the task.

In our experiments, we compare the performance by means of reward of two heuristics, the *aggressive* heuristic inspired from the one described for the narrow-band setting, and the *greedy* heuristic, described in 4.5.1, with multiple variants trained by PPO algorithm. All DRL agents are trained for

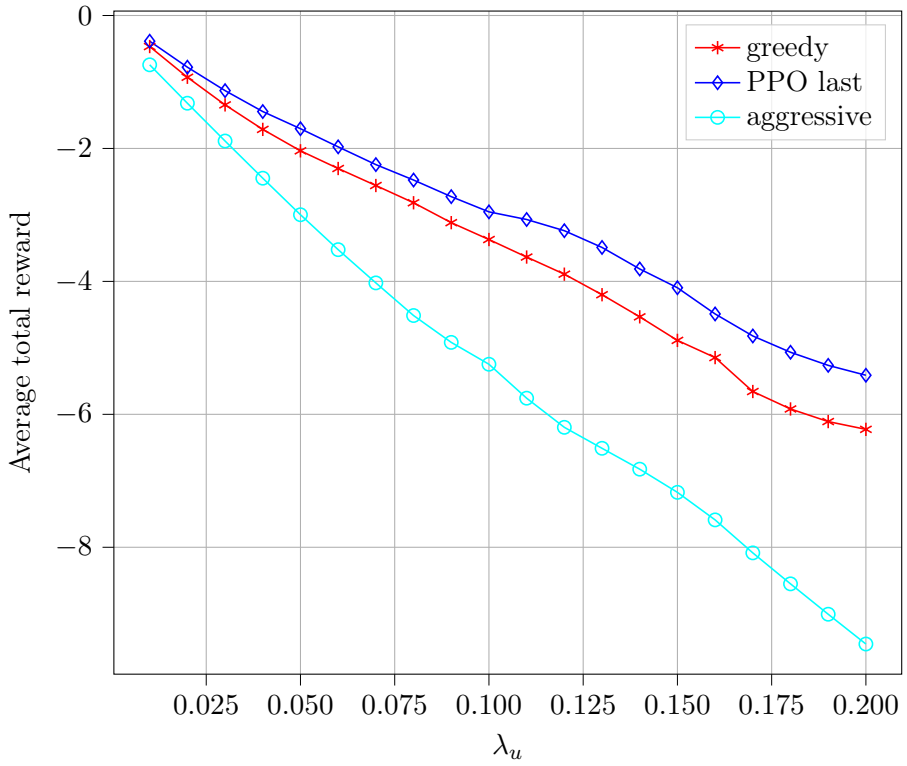


Figure 4.6: Multi-frequencies setting: average total reward versus arrival rate λ for really long training period

100 training volleys, 2000 episodes per volley, updating the model every 1000 training episodes.

We show in Figure 4.5 the average total reward obtained by various DRL variants and the two heuristics w.r.t. the arrival rate of new URLLC packets. Arrival rates are kept to low values for a dedicated URLLC set of frequencies should be used for higher arrival rates. We can see that adding some supervisions during training can help improve performance both with full rate state and with max rate state. We can also see that such performance increase is not that significative. Finally, we can see that all proposed variants overall outperforms the heuristics: the aggressive by a large margin while the greedy one by a smaller margin. In Figure 4.6, we compare the performance of a max state DRL agent trained without supervision for a much higher amount

of episodes. Specifically, we trained the agent for 150 training volleys, 10000 episodes per volley, updating the model every 5000 training episodes. We can see from the plot that increasing training episodes is really effective in devising a much better policy overall. This is to be expected for tasks with high variance of states and rewards w.r.t. actions like the one considered in this chapter.

4.6 Discussion

In this chapter we discussed a DRL approach to train an agent acting as a scheduler able to dynamically manage the coexistence of the URLLC traffic on top of the eMBB traffic. The agent is trained using PPO, a state-of-the-art DRL algorithm, and once trained it can decide where to execute puncturing with average time complexity within the range of $[0.3, 3.0]$ milliseconds, depending on the underlying hardware. It also supports parallelization by means of autonomous decisions over multiple simulations at once. The trained RL agent overall outperforms all the other schemes on multiple performance metrics, being capable of noteworthy generalization over the complementary task of never violating URLLC latency requirements while minimizing eMBB codewords' outages. The discussed approach is highly scalable with respect to the length of each simulation and the arrival packet distribution, without retraining the agent. This is of critical importance since the real world task we modeled is inherently not episodic and the URLLC packets' arrival distribution is not known a priori or it could be subject to changes. Beside that, we studied how to scale our DRL approach to a multi-frequencies setting, where reliability is of primary concern. Since we proved a continuous DRL approach not able to converge, we propose a hierarchical DRL approach, consisting of two agents: a frequencies scheduler and a packets scheduler. The former is tasked to choose the amount of frequencies and the latter is tasked to schedule each packet in the queue to a certain frequency, according to the amount chosen by the frequencies scheduler. We compared the combined DRL agent, trained with various hyperparameters, to an agressive heuristic and a greedy heuristic. Finally, we proved our approach to be able to obtain better performances overall w.r.t. the heuristics.

Chapter 5

Pseudo Random Number Generation through Reinforcement Learning

The fourth and last leg of our journey brings us in the field of cryptography, specifically to discuss a novel way to combine PRNGs and RL. PRNGs are algorithms produced to generate long sequences of statistically uncorrelated numbers, i.e. PRNs. These numbers approximate the properties of random numbers and are widely employed in mid-level cryptography and in software applications. Test suites are used to evaluate PRNGs quality by checking statistical properties of the generated sequences. These sequences are commonly represented bit by bit. Machine learning techniques are often used to break these generators, i.e. approximating a certain generator or a certain sequence using a NN. But what about using machine learning to *generate* PRNs generators?

In this chapter we first propose a RL approach to the task of generating PRNGs from scratch by learning a policy to solve an N-dimensional navigation problem. In this context, N is the length of the period of the sequence to generate and the policy is iteratively improved using the average score of an appropriate test suite run over that period. This approach lays the foundation of our study on PRNG and relies on a feedforward NN operating a fully observable MDP. Then, we also propose a more advanced approach to the same task, by learning a policy to solve a partially observable MDP, where the full state is the period of the generated sequence and the observation at each time step is the last sequence of bits appended to such state. We use a LSTM architecture to model the temporal relationship between observations at different time steps, by tasking the LSTM memory with the extraction of significant features of the hidden portion of the MDP's states. We also show that modeling a PRNG with a partially observable MDP and an LSTM architecture largely improves the results of the fully observable feedforward approach.

5.1 Introduction

Generating random numbers is an important task in cryptography [67], and more generally in computer science. Random numbers are used in several applications, whenever producing an unpredictable result is desirable: for instance games, gambling, encryption algorithms, statistical sampling, computer simulation and modeling, and many others.

An algorithm generating a sequence of numbers approximating properties of random numbers is a PRNG. A sequence generated by a PRNG is "pseudo-random" in the sense that it is generated by a deterministic function: the function can be extremely complex, but the same input will give the same sequence. The input of a PRNG is called seed, and to improve the randomness the seed itself can be drawn from a probability distribution. Of course, assuming that drawing from the probability distribution is implemented with an algorithm, everything is still deterministic at a lower level, and the sequence will repeat after a fixed, unknown, number of digits, called *period* of the PRNG.

While true random numbers sequences are more fit to certain applications where true randomness is necessary - for instance, high-end cryptography - they can be very expensive to generate. In most applications, a pseudo-random number sequence is good enough.

The quality of a PRNG is measured by the randomness of the generated sequences. The randomness of a specific sequence can be estimated by running some kind of statistical test suite. In this chapter the National Institute of Standards and Technology (NIST) statistical test suite for random and pseudo-random number generators [68] is used to validate the PRNG.

In literature, NNs have been used for predicting the output of an existing generator, that is, to break the key of a cryptography system. There have also been limited attempts at generating PRNGs using NNs by exploiting their structure and internal dynamics. For example, the authors of [69] use recurrent NNs dynamics to generate pseudo-random numbers. In [70], the authors use the dynamics of a feedforward NN with random orthogonal weight matrices to generate pseudo-random numbers. Neuronal plasticity is used in [71] instead. In [72] a generative adversarial network approach to the task is presented, exploiting an input source of randomness, like an existing PRNG

or a true random number generator.

A PRNG usually generates the sequence incrementally, that is, it starts from the seed at time $t = 0$ to generate the first number of the sequence at time $t = 1$, then the second at $t = 2$, and so on. Thus, it is naturally modeled by a deterministic MDP, where state space, action space and rewards can be chosen in several ways. In this chapter we discuss a novel approach to a DRL pipeline that can be used on this MDP to train a PRNG agent. This DRL approach works without data nor external inputs and without employing any structural dynamics, requiring only a feedfoward NN to operate. This is a probabilistic approach that generates pseudo-random numbers with a "variable period", because the learned policy will generally be stochastic. This is also a defining feature of this RL approach.

Finally, we observe how this approach has an action set with size growing linearly with the length of the sequence. This is a severe limiting factor, because when the action set is above a certain size it becomes very difficult, if not impossible, for an agent to explore the action space in a reasonable time. We then overcome the above limitation by proposing a different MDP formulation, using a partially observable state. By observing only the last part of the sequence, and using the hidden state of a LSTM NN to extract important features of the full state, we significantly improve the results obtained with the first feedforward approach.

This chapter is organized as follows. We describe the feedforward approach and the associated fully observable MDP in Section 5.2.1, while the recurrent approach and its associated partially observable MDP are shown in Section 5.2.2. In Section 5.2.3 we detail the reward function used for the MDP, which is shared by the two approaches. In Section 5.2.4 we describe the recurrent NN architecture used to model the environment state and the algorithm employed to train it. Finally, in Section 5.3 we present the results of our experiments, which are then discussed in Section 5.4.

5.2 Pseudo Random Number Generation

The main idea is quite natural: since a PRNG builds the random sequence incrementally, we model it as an agent in a suitable MDP, and use DRL to train

the agent. Several "hyperparameters" of this DRL pipeline must be chosen: a good notion of states and actions, a reward such that its maximization gives sequences as close as possible to true random sequences, and a DRL algorithm to train the agent.

5.2.1 Binary Formulation and fully observable MDP

While easier said than done, the problem of generating PRNs seems at first glance suitable for RL. There is however one caveat: the naive approach that comes to mind, that is, using as state the last generated number, is inherently not Markovian: whatever reward we use to measure the randomness of the sequence, it must depends on the whole sequence that was generated before. On the other hand, using as state the whole sequence, and increasing the length of the sequence by appending a new number is calling for the curse of dimensionality[73]!

We address this issue by a completely different approach. We keep the sequence length fixed, say N. At each time step the agent can now fully observe the state of the environment, that is, the entire sequence codified in binary form, for a total length of B bits. This state is Markovian by definition. The goal of this decision task is to perturb the sequence without changing its length, and this can be obtained by setting a certain fixed binary value at some position in the sequence. Actions will then be given by setting the value of various bits at certain positions in the sequence. They will be described more in detail below.

Finally, we model the task as finite-horizon episodic, by fixing a termination time T. Thus, the state S_T is the output sequence of the PRNG, i.e. the sequence of B PRNs, represented in their binary form. A fixed length output however violates one requirement of PRNGs, that is, a PRNG has to be able to generate a (possibly) infinite amount of numbers. To solve this problem, we can think of the generated sequence as the period of the PRNG. By concatenating multiple output of the same PRNG it is possible to obtain a (possibly) infinite amount of numbers. Please note that this will produce PRNs with "variable period", because the learned policy will generally be stochastic, as we will show in the next section. This is a feature of this RL approach.

As usual, we start generating PRNs from a number known as the seed.

According to our definition of state, we set all B bits of the sequence corresponding to the starting state equal to the seed value, e.g. 0. This is in general an B-dimensional navigation task.

Formally, we can model our task as a fully observable MDP in the following way, given the bit length of the sequence B. The state space is given by all possible bit sequences of length B:

$$\mathcal{S} := \{(b_1, b_2, \ldots, b_B) : b_n \in \{0, 1\}\}.$$

Action 1_n is the action of setting the n^{th} bit to 1, and 0_n is the action of setting the n^{th} bit to 0. The action space is then:

$$\mathcal{A} = \bigcup_{n=1}^{B} \{1_n, 0_n\}$$

This finite MDP formulation has $|\mathcal{S}| = 2^B$ and $|\mathcal{A}| = 2B$, and is called Binary Formulation (BF). This is the formulation used in the feedforward solution and our first proposed approach.

5.2.2 Recurrent formulation and partially observable MDP

The main problem with BF is the fact that the size $2B$ of the action set grows linearly with the increasing length B of the sequence. Above a certain size, it is no longer possible to learn a policy with high average score. If the size is big enough, no policy can be learned at all because it is almost impossible for an agent to explore an action space so huge in a reasonable time. Consider that for PRNGs a sequence of 1000 bits is quite short, while for a RL problem 2000 actions are way too much.

We overcome this limitation of BF by hiding a portion of the full pseudo-random sequence, letting the agent see and act only on the last N bits. This removes the correlation between the final length of the sequence and the number of actions, at the cost of introducing a temporal dependency among states, breaking Markovianity and making the resulting MDP partially observable. This new problem is solved by approximating the hidden portion of the state with the hidden state of a recurrent NN with memory, see Section 5.2.4 for details. We call this new approach Recurrent Formulation (RF), and it is the second approach we discuss.

Let $N \in \mathbb{N}$ be the number of bits at the end of the full sequence that we want to expose, or, in other words, let the observations space be the set $\mathcal{O} := \{0,1\}^N$. We want the agent to be able to change freely the last N bits, that is, the action space $\mathcal{A}$ coincides with $\mathcal{O}$. We also fix a predetermined temporal horizon T for episodes. This means that the size of the action space is $|\mathcal{A}| = 2^N$ for a generated sequence of $T \cdot N$ bits. Both $|\mathcal{A}|$ and length of S_T depend on N, but they do not depend on each other. Increasing the length of the generated sequence can then be done by increasing the horizon T, leaving constant the action space size.

For example, with $N = 3$, action and observation sets are:

$$\mathcal{O} = \mathcal{A} = \{[000]; [001]; [010]; [100]; [011]; [101]; [110]; [111]\}$$

Continuing the example, assume at $t = 0$ we start from a random initial state $S_0 = O_0 = [001]$, and that $A_0 = [111]$. Now the full state is $S_1 = [001111]$, but only the last 3 bits $O_1 = [111]$ are observed by the agent. If the agent now chooses $A_1 = [101]$, the full state becomes $S_2 = [001111101]$, and so on. If we set the episodes length to 100, at the end of the episode the full state is a 300 bits sequence.

Clearly, this formulation can work only if the policy approximator $\pi(\cdot|\cdot;\theta)$ can preserve some information from the time series given by the past observations. Recurrent NNs can model memory, and for this reason are one of the possible approaches to process time series and, more in general, temporal relationships among data. This is very useful in RL environments where, from the point of the view of the agent, the Markov property does not hold, as it is typically the case in many partially observable environments. This approach was suggested in [74], see [75, 76] for examples of use cases far different from ours.

State-of-the-art recurrent NNs for this kind of problems are the Gated Recurrent Unit (GRU) ones, and the LSTM ones. Typically, GRU performs better than LSTM, but for this particular formulation we have experienced good performance with LSTM. Thus, we use LSTM layers to approximate the policy network, see 5.2.4 for details on the NN architecture.

5.2.3 Reward Function through NIST Test Suite

The NIST statistical test suite for random and pseudo-random number generators is the most popular application to test the randomness of sequences of bits. It has been published as a result of a comprehensive theoretical and experimental analysis and may be considered as the state-of-the-art in randomness testing for cryptographic and not cryptographic applications. The test suite has become a standard stage in assessing the outcome of PRNGs shortly after its publication.

The NIST test suite is based on statistical hypothesis testing and contains a set of statistical tests specially designed to assess different pseudo-random number sequences properties. Each test computes a test statistic value, function of the input sequence. This value is then used to calculate a P-value that summarizes the strength of the evidence for the sequence to be random. For more details, see [68].

If S_t is the sequence produced by the agent at time t, the NIST test suite can be used to compute the average P-value of all eligible tests run on S_t. Some tests return multiple statistic values: in that case, their average is taken. If a test is failed its value in the average is set to zero. Some tests are not eligible on certain too short sequences, and in this case they are not considered for the average. This average $\text{avg}_{\text{NIST}}(S_t)$ is used at the end T of each episode as a reward function for the MDP:

$$R_t = \begin{cases} \text{avg}_{\text{NIST}}(S_t) & \text{if } t = T \\ 0 & \text{otherwise} \end{cases}$$

This reward strategy is shared by BF and RF. Note that, since P-values are probabilities, rewards belong to $[0, 1]$, and that NIST test suite accuracy grows with the tested sequence length.

The NIST test battery is run with a framework called NistRng[1]. This framework allows us to easily run a customizable battery of statistical set over a certain sequence. The framework also computes which tests are acceptable over certain sequences, i.e. due to their length. Acceptable tests for a certain sequence are called eligible tests. Each test returns a value and a flag stating

[1] Available on PyPi and also on GitHub: `https://github.com/InsaneMonster/NistRng`.

if the test was successfully exceeded or not by the sequence. If a test is not eligible with respect to a certain sequence it cannot be run and it is skipped.

5.2.4 Algorithm and Neural Network Architecture

In this discussion we use the PG algorithm PPO, described in [21] and considered state-of-the-art in PG methods. We already introduced PPO in this thesis, in Chapter 3, specifically in Section 3.3.2, and we also employed it with positive results in Chapter 4. PPO tries to take the biggest possible improvement step on a policy using the data it currently has, without stepping too far and making the performance collapse.

The PPO used in this chapter is an instance of PPO-Clip as described by OpenAI at [62], with only partially shared value and policy heads. More details on the NN architecture in Section 5.2.4. We use 0.2 as clip ratio, and early stopping: if the mean KL-divergence of the new policy from the old grows beyond a threshold, training is stopped for the policy, while it continues for the value function. We used a threshold of $1.5 \cdot K$, with $K = 1\text{e}{-2}$.

To reduce the variance, we used Generalized Advantage Estimation as in [47] to estimate the advantage, with $\gamma = 1$ and $\lambda = 0.95$. Saved rewards R_t were normalized with respect to when they were collected (called rewards-to-go in [62]).

PPO is an actor-critic algorithm. This means that it requires a NN for the policy (the actor) and another NN for the value function (the critic). Moreover, since RF is a partially observable MDP formulation, we need a way to maintain as much information as possible from previous observations, without exponentially increasing the size of the state space. We solve this problem with LSTM layers [77].

The NN used for RF starts with two LSTM layers, with bias = 1 for the forget gate. After the LSTM layers, the network splits in two different subnetworks, one for the policy and one for the value function. The policy subnetwork has three dense layers with 256, 128 and 64 neurons respectively, stacked. All of them have ReLU activation. After this, a dense layer with 2^N neurons provides preferences for the actions, which are turned into probabilities by a softmax activation. This is the policy head. The value function subnetwork starts exactly as the policy one (but with different weights): three

dense layers with 256, 128 and 64 neurons respectively, stacked, all with ReLU activation. At the end, a dense layer with 1 neuron and no activation for the state value. This is the value head. All layers have Xavier initialization.

BF uses two different NNs for the policy and the value function. The policy network has three dense layers with 256, 512 and 256 neurons respectively, stacked. All of them have ReLU activation. After this, the policy head is the same as in RF: a dense layer with 2^N neurons and softmax activation. The same for the value function network: three dense layers with 256, 512 and 256 neurons respectively, stacked, with ReLU activation. After this, the value function head is the same as in RF: a dense layer with 1 neuron and no activation. All layers have Xavier initialization.

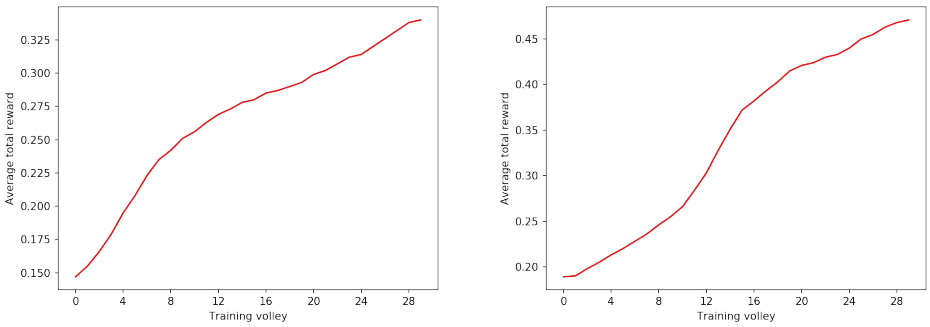
5.3 Results

Our experiments consists on multiple sets of training processes of various BF and RF agents. The goal of these experiments is to measure the performance of the new RF agents and compare it with the results achieved by BF agents.

We consider 3 different agents, all trained by PPO-Clip described in Section 5.2.4 with an actor-critic NN described in Section 5.2.4.

The agent based on the formulation RF is called π_{RF}, and similarly we denote by π_{BF} the agent based on BF, that we use as a baseline. We introduce also a third agent, a variation of π_{BF} denoted by $\hat{\pi}_{BF}$ and called "wanderer": this agent is forced to move as much as possible within the environment by masking out all actions that would keep the agent in the same state. In other words, at time-step t the agent $\hat{\pi}_{BF}$ is forbidden from setting a certain bit to the same value it has at the previous time step $t-1$.

In the experiments, we optimize a PPO-Clip loss with GAE advantage estimation. Two separated policy loss and value loss are estimated by Adam with mini-batch samples of 32 experiences drawn randomly from a buffer. The buffer is filled by 500 episodes for π_{BF} and $\hat{\pi}_{BF}$, and by 1000 episodes for π_{RF}. Once the buffer is full, a training epoch is performed: thus, for instance, an agent π_{BF} building sequences of length $B = 200$ by episodes of length $T = 100$ will start training when the buffer is filled with $500 \cdot 100 = 50,000$ experiences, and the epoch will end after $[50,000/32] = 1562$ training steps. Learning rates



(a) π_{BF} with $B = 80$ **(b)** $\hat{\pi}_{BF}$ with $B = 80$

Figure 5.1: Experiment on BF with $B = 80$. The learning curve is different and the average total reward is better with $\hat{\pi}_{BF}$. Volleys are composed of 1000 episodes each and the fixed length of each trajectory is $T = 40$ steps.

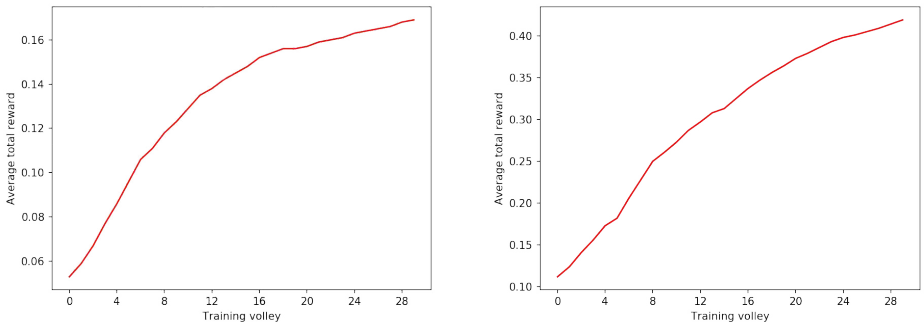
are 3e−4 for the policy and 1e−3 for the value. At the end of the epoch, the buffer is emptied, and the RL pipeline goes on by experiencing new episodes.

We present experimental results in the form of plots. The performance metric used is the average total reward across sets of epochs called *volleys*. In this chapter, a volley is made from two epochs, that is, the plots represent a moving average over a window of two epochs.

Figures 5.1a, 5.1b, 5.2a, 5.2b and 5.3a, 5.3b describe three experiments with π_{BF} and $\hat{\pi}_{BF}$ over sequences of different lengths: $B = 80$ bits, $B = 200$ bits and $B = 400$ bits respectively. Length of episodes is $T = 40$, $T = 100$ and $T = 200$ respectively.

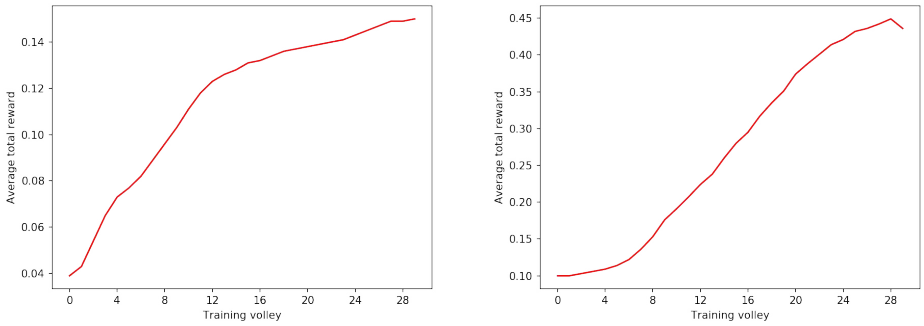
For very short sequences of 80 bits, $\hat{\pi}_{BF}$ has better performance at the end yet very similar performance at beginning of training. Making the sequence longer, $B = 200$ bits, allows the wanderer agent to perform much better than the baseline: starting from slightly different performance at the beginning, $\hat{\pi}_{BF}$ has a much higher average total reward at the end of the training process. This trend is confirmed with sequences of $B = 400$ bits.

Figures 5.4, 5.5 and 5.6 describe three experiments with π_{RF}. In this case the length of the trajectory T directly influences the length of the sequence



(a) π_{BF} with $B = 200$ **(b)** $\hat{\pi}_{BF}$ with $B = 200$

Figure 5.2: Experiment on BF with $B = 200$. Despite the similar learning curve, there is a huge difference in the achieved average total reward per episode between $\hat{\pi}_{BF}$ and π_{BF} at the end of the training process. Volleys are composed of 1000 episodes each and the fixed length of each trajectory is $T = 100$ steps.



(a) π_{BF} with $B = 400$ **(b)** $\hat{\pi}_{BF}$ with $B = 400$

Figure 5.3: Experiment on BF with $B = 400$. The difference in the achieved average total reward per episode between $\hat{\pi}_{BF}$ and π_{BF} is similar to the case with $B = 200$, while the learning curve is different. Volleys are composed of 1000 episodes each and the fixed length of each trajectory is $T = 200$ steps.

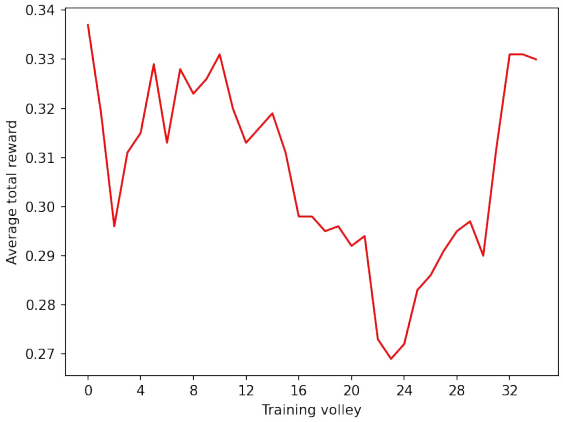


Figure 5.4: Average total rewards during training of π_{RF} with $N = 2$ and $T = 100$. Volleys are composed by 2000 episodes each.

generated by π_{RF}. We keep $T = 100$ constant across π_{RF} experiments. This value is chosen experimentally as giving best performance while keeping inference time acceptable. To obtain sequences comparable with the ones generated by BF agents, we choose to append $N = 2$, $N = 5$ and $N = 10$ bits respectively, resulting in final sequences of length $2 \cdot 100 = 200$ bits, $5 \cdot 100 = 500$ bits and $10 \cdot 100 = 1000$ bits respectively. The seed S_0 of the PRNG is drawn from a standard multivariate normal distribution of dimension $N = 2$, $N = 5$ and $N = 10$ respectively.

For $N = 2$, training is not successful. We do not have a clear explanation for this fact. A possible reason is that we are trying to represent the non observable portion of the state with a very low-dimensional input. However, this is not completely true, because in theory the inputs from every time-step is preserved in the hidden state of the LSTM. This issue could be related to the difficulties that recurrent NNs have shown with gradient descent optimizers, see [78].

For $N = 5$, that is, sequences of 500 bits, π_{RF} vastly outperforms π_{BF} and, albeit by a narrower margin, $\hat{\pi}_{BF}$ with sequences of 200 bits. Thus, we have better performance with 150% additional bits in the sequence. For $N = 10$,

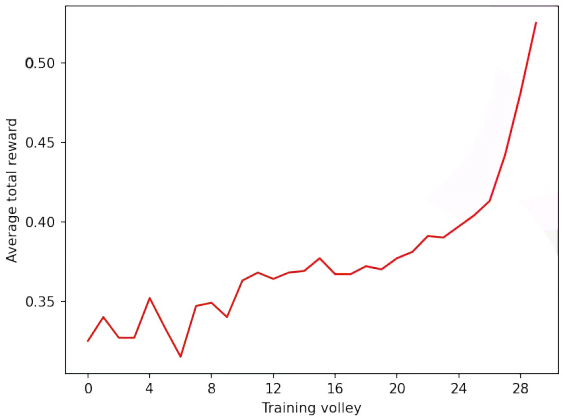


Figure 5.5: average total rewards during training of π_{RF} with $N = 5$ and $T = 100$ steps. Volleys are composed by 2000 episodes each.

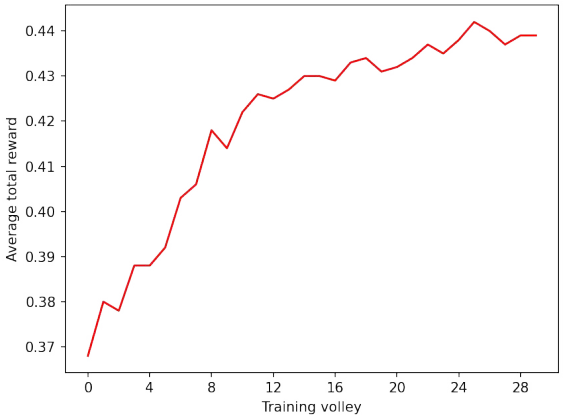


Figure 5.6: average total rewards during training of π_{RF} with $N = 10$ and $T = 100$ steps. Volleys are composed by 2000 episodes each.

that is, sequences of 1000 bits, π_{RF} vastly outperforms π_{BF} with $B = 400$. The wanderer agent in this case performs similarly, but still π_{RF} produces sequences that are 150% longer than the ones produced by $\hat{\pi}_{BF}$.

(a) NIST score 0.4 **(b)** NIST score 0.43 **(c)** NIST score 0.53

Figure 5.7: A graphical representation of 3 sequences of 1000 bits generated by the same trained π_{RF} with their NIST score. Images are obtained by stacking the 1000 bits in 40 rows and 25 columns, then ones are converted to 10×10 white squares and zeros to 10×10 black squares. The resulting image is smoothed.

We now want to test the hypothesis that, in order to generate PRNGs by DRL, modeling the MDP as in RF is much better than modeling it as in BF. To this aim, we compare the average total rewards per episode of random agents operating in RF and in the baseline BF.

We call these agents ρ_{RF} and ρ_{BF} respectively.

This comparison is performed to exclude the possibility that PPO is performing better with RF, but maybe different algorithms would perform better with BF. Using a random agent means removing the algorithm from the equation.

Figures 5.8a, 5.8b, 5.9a, 5.9b, and 5.10a, 5.10b describe three comparisons between the random agents producing short, medium and long sequences respectively. As before, short means 80 bits in BF and 200 bits in RF, medium means 200 bits in BF and 500 bits in RF, long means 400 bits in BF and 1000 bits in RF. In each comparison, the average total rewards of ρ_{RF} is higher than ρ_{BF}. From these results, we can assess that RF is a better formulation than BF for the PRNG task.

Finally, figures 5.7a, 5.7b and 5.7c represent graphically three sequences of 1000 bits generated by the same agent π_{RF} after training, with NIST scores 0.4, 0.43 and 0.53 respectively. The 1000 bits are stacked on 40 rows and 25 columns, then ones are converted to 10×10 white squares and zeros to 10×10 black squares, and the resulting image is smoothed.

5.4 Discussion

In this chapter we discussed multiple ways to automatically generate PRNGs, a task of interest and a currently open field of research. Both our proposed approaches use RL to build a PRNG from scratch and, to the best of our knowledge, this is a novel approach. While they are not currently able to directly compete in a production environment with deterministic approaches used in literature, both approaches present promising results, which we think can be used to lay down a new research path combining RL and PRNGs. Both proposed approaches present the following interesting features:

- They require no input data, so that the generated PRNG is always a novel algorithm, and the trained agent explores solutions possibly unknown to a human-generated or a supervised-learning-generated algorithm. In other words, the generated PRNG is not an imitation of some other algorithm and the generated sequence is not an imitation of a pre-existing PRNs or TRNs sequence.

- Each time a training process is run, the resulting PRNGs are likely to be different from each other. This is an inherent property of RL as a whole: the policy learned is one of the very many stochastic optimal policies. Moreover, one can change the model hyperparameters and/or the training algorithm to increase diversity between generated PRNGs.

- For a single starting state, multiple solutions can be obtained after one training process since RL policies can be stochastic. Indeed, given a certain starting state, the same agent will usually follow different trajectories, leading to different output sequences. In short, we obtain a non-deterministic PRNG given a single seed. This is a novel property which is not present in current state-of-the-art PRNGs.

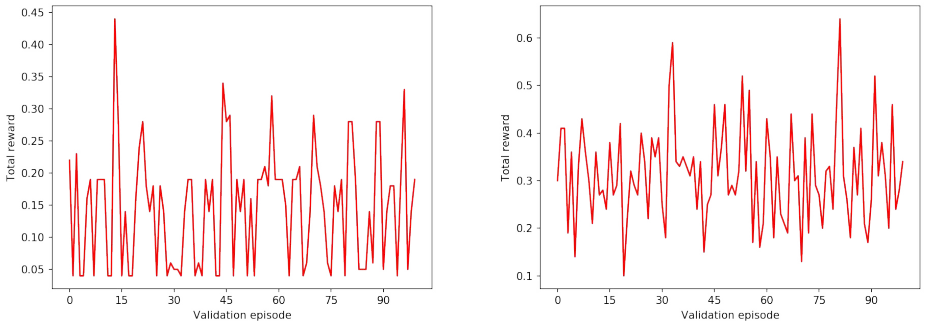
- Given that NNs are black-box approximators, the policy of the RL agent is black-box. Since the PRNG is the algorithm given by that policy, it is also black-box. This grants the nice property of having no human insights in the inner functioning of the PRNG, thus providing (potentially) increased security from a cryptographic standpoint.

The first proposed approach (BF), models a fully observable MDP. The main limitation of this approach is the dimensionality B of the state, i.e. the period of the PRNG. Our experiments show that, at least on average hardware, is very complex to successfully train an agent on longer sequences, thus obtaining a PRNG with longer period.

This limitation is partially overcome by the second approach (RF) we discussed, which uses a partially observable MDP to model the task. Experiments show that RF agents trained with PPO obtains at the same time a higher average NIST score and longer sequences, thus improving BF in two different ways. We use a PPO instance with a hidden state of an LSTM to encode significant features of the non observed portion of the sequence: as far as we know, this is also an original idea for PRNG. Moreover, experiments with a random agent show that RF is a better MDP modeling when compared with BF, that is, when RF is compared with BF, obtains a higher average NIST score with longer sequences. All this means that RF scales better to PRNG with longer periods.

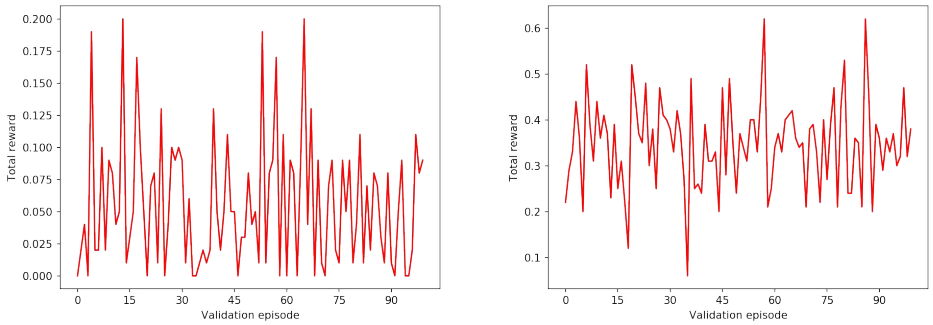
However, while RF is a serious improvement over BF, the action space size grows as 2^N, where N is the bit-length of the appended sequence. Since DRL does not scale well to discrete and large action sets, see for instance [79], this is a limitation for RF. In our experiments, we have found that $N > 10$ is not feasible for RF. Moreover, the vanishing gradient is an obstacle to increasing the episodes length T with recurrent NNs, as shown in [78]. In our experiments we have been unable to train RF with $T = 200$, so we consider $T = 100$ an upper bound for RF with PPO and the LSTM architecture described in Section 5.2.4. Since the PRNG period is $T \cdot N$, we can say that RF with all the hyperparameters described in this chapter does not scale well over 1000 bits.

Another problem of this approach is the sparse reward, which in general makes difficult for a RL agent to be trained, whatever the setting.



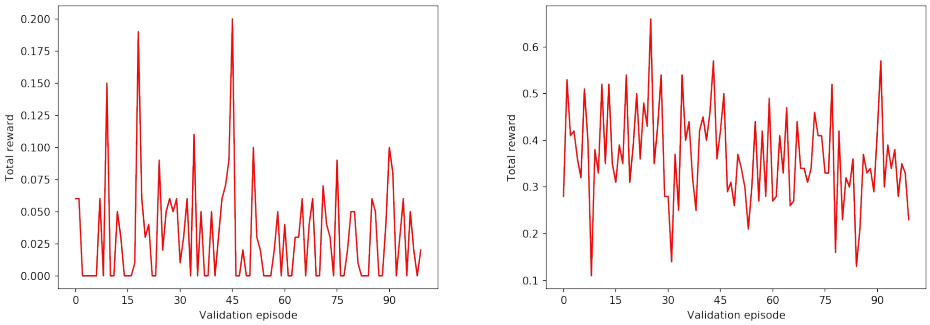
(a) ρ_{BF} with $B = 80$, episodes length $T = 40$.**(b)** ρ_{RF} with $N = 2$, episodes length $T = 100$.

Figure 5.8: Average total rewards of a random agent on BF and RF for short sequences



(a) ρ_{BF} with $B = 200$, episodes length $T =$ 100.**(b)** ρ_{RF} with $N = 5$, episodes length $T = 100$.

Figure 5.9: Average total rewards of a random agent on BF and RF for medium sequences.



(a) ρ_{BF} with $B = 400$, episodes length $T = 200$.

(b) ρ_{RF} with $N = 10$, episodes length $T = 100$.

Figure 5.10: Average total rewards of a random agent on BF and RF for long sequences.

Chapter 6

Conclusions and Future Works

We have arrived to the end of our journey into the application of RL to real world tasks. We discussed various problems and we studied multiple ways to solve them. We saw that each task presented its own challenges, often very difficult to overcome if not by means of a specific modeling.

The often abused general notion that RL is just a collection of computationally expensive methods to solve all kind of tasks, seldom happens to be true. Indeed, even when using state-of-the-art methods and algorithms, RL requires the task to be modeled with specific MPDs in order to be solved within reasonable time and with reasonable performance. This is even more true when the underlying hardware is not top tier. While there is for sure a great variation from task to task, we found some common issues that need to be addressed before to apply any state-of-the-art algorithm or method.

The first common problem is the definition of a meaningful state space for the MDP. Without repeating the thorough introduction of Section 1.2.1, the state space defines what is and what is not a possible state for the environment the RL agent is acting upon. In all the problems we discussed, the state space was defined by one or more vectors, hence being defined as continuous state space. In the context of DRL, DNNs are really powerful at discovering on their own the features of such complex data. However, just like with common supervised techniques, junk data and uncorrelated data do make the process slower and decreases the performance of the resulting approximator. Moreover, while it is a good practice to always normalize inputs to NNs, normalization could reduce the range of meaningfully different states, especially in certain contexts. For example, when in Chapter 3 we discussed how to navigate in the state space of a text in order to revise a poem, we actually had to move the DRL agent in the space of the embeddings of the text. This was necessary because using as agent inputs the normalized id values of each word

would make very different words appear to be similar from the point of view of the algorithm. This is especially true for large dictionaries like our own, where words can have really high integer ids. A similar problem is the one we faced in Chapter 5, when at first we used as state space for the DRL PRNG the entire sequence of pseudo-random bits. For large sequences, the problem was unsolvable because it was impossible for the DNN to extract meaningful features from the sequence. This effect was also increased by the fact the sequence was incrementally more uncorrelated because of the task itself, making the problem unapproachable by DRL techniques. It's worth noting that there are some real world tasks where full observability of the MDPs is impossible, for example due to noisy sensors or incomplete information about the environment state. In these settings, RL could be applied only with robust algorithms able to deal with partial observability, as the definition of a proper state, action space and reward function for an RL agent, as described throughout this thesis, may not be enough to guarantee optimal performance or even convergence.

The second common problem is the definition of a meaningful action space for the MDP. As explained in Section 1.2.1, the action space defines what can be done by the RL agent on the environment at each time step. It is also possible to mask or clip part of this state at certain time steps, to prevent impossible actions only when specific conditions are met. As a general rule, we found action spaces defined through a vector, i.e. continuous action spaces to be very hard to work with. This is especially true when the elements within that vector are supposed to assume only certain values, like 0 or 1. In Chapter 4, for example, we discussed the very bad performances of the continuous action space approach to the complex task of allocating URLLC packets over multiple eMBB frequencies at once. Like the other works presented in this thesis, we always employed discrete action spaces, enumerating all actions. While modeling MDP with discrete action spaces proves to be very effective in stabilizing the algorithms performance, it also presents a lot of challenges if the resulting action space is of combinatorial complexity: no state-of-the-art algorithm can find an optimal policy if the amount of actions to be explored is too large! In the case of the resource slicing task over multiple frequencies at once, since the continuous approach was not an option due to its terrible per-

formances, we decomposed the problem into two different MDPs, both with discrete action spaces, thus reducing their size to reasonable levels.

Finally, a third common problem is to be found in the definition of the reward function. This is a problem well known in literature, and it is usually defined as reward engineering. Without explaining again what the reward function is, since it is well detailed in Section 1.2.1, we think that using rewards as distributed as some toy tasks well known in literature is usually the best way to go. Indeed some of these tasks, like finding the shortest path in a two dimensional grid or keeping the agent at a certain state, are effective way to define broader reward functions groups. For example, the tasks discussed both in Chapter 3 and Chapter 5 have specific reward functions whose broader properties follow that of a shortest path problem. On the other hand, the problem discussed in Chapter 4 is faced through a reward function whose distribution closely match that of an agent with the simple goal of keeping its current state.

More in general, we believe that most real world tasks can be solved through RL if they can be defined as MDPs with appropriate states and actions space for N-dimensional navigation tasks, where the reward function is shaped to train the agent into one of two possible policies: finding another goal position in the least possible amount of steps or keeping a certain goal position for the highest possible amount of steps. In this context, N is the size of the state, which we assume continuous for most, if not all, the modeled tasks. We strongly suggest, instead, to use discrete action spaces or to resort to a divide et impera strategy, as in Chapter 4, for action spaces of combinatorial size.

In the following, a brief summary of the key contributions of the thesis will precede some intuitions on relevant avenue for further work.

6.1 Summary of Contributions

- In Chapter 2 we discussed a Go artificial player trained using score as target. The same pipeline was used as the pipeline of SAI and LZ, well known AG-like open source software, whose networks were instead trained with the traditional binary target. The training was successful, and produced a player with valid play but particular style. After

calibration, the training proved to produce consistently weaker networks than the corresponding networks in the SAI training, and converged prematurely to a lower Elo level. Our results prove the folklore statement that using the score as target doesn't work as well as using win-rates, while still converging to a reasonable player with an interesting, while suboptimal, playstyle.

- In Chapter 3 we discussed an innovative way of implementing the notion of creativity in a machine. Considering the task of automatically generating new poems, we proposed a model that implements the human-like behaviour of writing a draft and revising it multiple times. These drafts are conditioned to author and rhyme information, while the revision process is built around an iterative procedure that can be described as a navigation problem and solved with RL. Multiple experiments confirmed that the proposed approach is feasible and that it allows the machine to learn how to revise text from scratch, even if it is not explicitly instructed on which portion of text it should revise.

- In Chapter 4 we discussed a DRL approach to train an agent acting as a scheduler able to dynamically manage the coexistence of the URLLC traffic on top of the eMBB traffic. The trained RL agent overall outperforms all the other schemes on multiple performance metrics, being capable of noteworthy generalization over the complementary task of never violating URLLC latency requirements while minimizing eMBB codewords' outages. The discussed approach is highly scalable with respect to the length of each simulation and the arrival packet distribution, without retraining the agent. This is of critical importance since the real world task we modeled is inherently not episodic and the URLLC packets' arrival distribution is not known a priori or it could be subject to changes. Furthermore, we discussed an extended approach to scale the DRL agent to reliable traffic over multiple frequencies. We tried to solve the task both through a continuous action space approach and a hierarchical approach, finding out the first was unable to solve the task, while the second one outperformed all considered heuristics.

- Finally, in Chapter 5 we discussed multiple ways to automatically gen-

erate PRNGs. Both our proposed approaches are novel and use RL to build a PRNG from scratch. The first one uses a feedforward DRL approach over the entire period of the generator, while the second one uses a recurrent approach to extract the features of the already processed bits of the sequence, thus reducing the action space size to reasonable levels for longer periods. Both approaches present promising results, and they both disclose some interesting features. For example, they requires no input data, so that the generated PRNG is always a novel algorithm and the trained agent explores solutions possibly unknown to a human-generated or a supervised-learning-generated algorithm. Moreover, each time a training process is run the resulting PRNGs are likely to be different from each other and even for a single starting seed state multiple policies can be obtained. Finally, given that NNs are black-box approximators, the policy of the RL agent is black-box. Since the PRNG is the algorithm given by that policy, it is also black-box. This grants the nice property of having no human insights in the inner functioning of the PRNG, thus providing (potentially) increased security from a cryptographic standpoint.

6.2 Issues and avenues of research

In the following we analyze the open issues on most contribution's topics and we discuss what we think are the possible avenues of research.

Neural Poetry In Chapter 3 we proved that a human-like approach to poems generation is possible through the appropriate framework. We exploited RL to inject the notion of creativity into the machine, by letting the RL agent explore the state space of the embeddings of the text. The policy of the detector agent was iteratively refined by training with the state-of-the-art algorithm PPO. Finally, we suggested that the proposed approach is general enough to scale to other text related tasks. However, we think there is an issue in applying our approach to different related tasks. This issue is to find a good reward for the algorithm. We employed rhyme schemes since they can easily be computed and a discrete reward can be assigned depending on the

amount of matching words at the end of each line. Many other tasks, like for example producing a prose text, are really hard to quantify for they involve a qualitative and often subjective analysis. We think that focusing on researching new ways to quantify various qualities of texts could open wide paths to follow to really apply what's introduced in this contribution to a broader set of problems.

Resource Slicing In Chapter 4 we proposed a DRL approach to train an agent acting as a scheduler able to dynamically manage the coexistence of the URLLC traffic on top of the eMBB traffic. We proved our approach to be both scalable to multiple frequencies and able to outperform all other heuristics on multiple performance metrics. We think that the natural next step is to use real traffic data instead of simulated. Beside that, we always limited the study to single user, and we think it could be interesting to apply the discussed approaches to multi-user settings.

Psuedo Random Number Generation In Chapter 5 we proposed a novel approach to the task of generating PRNGs from scratch. We also discussed a recurrent approach to make the action space size of the MPD independent on the length of the generated PRNs sequence. We compared the two approaches and with appropriate experiments we proved that the problem is feasible in both cases. We also show that the recurrent approach vastly outperforms the other. We think, however, that the path is very open for further research and advancements. To the best of our knowledge, employing RL in this field to solve this task is a completely novel approach, making it evident that our promising result are very far from finding a solution employable in practice. We hope our work will inspire future research which combines PRNGs and RL. Finally, in the following we identify a set of interesting issues and possible improvements::

- Reward function sparsity: by giving rewards only at the end of each episode, the amount of steps of each episode cannot be *that* high. Moreover, RL algorithms rarely work well with very sparse reward functions. This poses a significant limitation to the length of the generated period. In our opinion, finding a less sparse reward function could lead to

significative performance improvements.

- Architectural limitations: even with a less sparse reward function, there is only *so much* a recurrent NN can remember. The upper bound given by the episodes length could be overcome, or at least mitigated, by other architectures capable of maintaining a memory for a time longer than recurrent neural networks, for example attention models.

- Period stacking: an approach similar to the recurrent one could be used to intelligently stack generated periods. For example, one could train a policy to stack sequences generated by one or multiple PRNGs, even with different sequence lengths. This would generate PRNGs with very long periods without a sensible drop in quality. This approach could use a mixture of state-of-the-art PRNGs, DRL generators like the ones proposed, and so on. A challenge of this approach is how to measure the change in the NIST score when appending a random sequence to another.

- Starting seeds: multiple seeds are the one of the core features of PRNGs. To allow for a significant amount of seeds to be concurrently supported, the variance they introduce during learning has to be reduced somehow. We believe that processing somehow the vector representation (for example with convolutional filters) could be a promising path to follow.

Bibliography

[1] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[2] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton *et al.*, "Mastering the game of Go without human knowledge," *Nature*, vol. 550, no. 7676, p. 354, 2017.

[3] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, p. 484, 2016.

[4] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel *et al.*, "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play," *Science*, vol. 362, no. 6419, pp. 1140–1144, 2018.

[5] Gian-Carlo Pascutto and contributors, "Leela Zero," 2018, http://zero.sjeng.org/home [Online; accessed 17-August-2018]. [Online]. Available: http://zero.sjeng.org/home

[6] F. Morandin, G. Amato, M. Fantozzi, R. Gini, C. Metta, and M. Parton, "SAI: A sensible artificial intelligence that plays with handicap and targets high scores in 9×9 go," in *ECAI 2020 - 24th European Conference on Artificial Intelligence, 29 August-8 September 2020, Santiago de Compostela, Spain, August 29 - September 8, 2020 - Including 10th Conference on Prestigious Applications of Artificial Intelligence (PAIS 2020)*, ser. Frontiers in Artificial Intelligence and Applications, G. D. Giacomo, A. Catalá, B. Dilkina, M. Milano, S. Barro, A. Bugarín, and J. Lang, Eds., vol. 325. IOS Press, 2020, pp. 403–410. [Online]. Available: https://doi.org/10.3233/FAIA200119

[7] D. J. Wu, "Accelerating Self-Play Learning in Go," *arXiv:1902.10565*, 2019.

[8] F. Morandin, G. Amato, R. Gini, C. Metta, M. Parton, and G. Pascutto, "SAI a Sensible Artificial Intelligence that plays Go," in *International Joint Conference on Neural Networks, IJCNN 2019 Budapest, Hungary, July 14-19, 2019*, 2019, pp. 1–8. [Online]. Available: https://doi.org/10.1109/IJCNN.2019.8852266

[9] A. authors, "SAI, a Sensible Artificial Intelligence that plays Go," Preprint available in the supplemental material. A reduced version of this preprint is to be published in the proceedings of a conference on neural networks, 2018.

[10] Gianluca Amato and contributors, "SAI: a fork of Leela Zero with variable komi," https://github.com/sai-dev/sai [Online; accessed 10-Nov-2019]. [Online]. Available: https://github.com/sai-dev/sai

[11] R. Coulom, "Bayesian Elo rating," 2010, http://www.remi-coulom.fr/Bayesian-Elo/ [Online; accessed 10-Nov-2019]. [Online]. Available: https://www.remi-coulom.fr/Bayesian-Elo/

[12] M. A. Boden, "Chapter 9 - creativity," in *Artificial Intelligence*, ser. Handbook of Perception and Cognition, M. A. Boden, Ed. San Diego: Academic Press, 1996, pp. 267 – 291. [Online]. Available: http://www.sciencedirect.com/science/article/pii/B978012161964050011X

[13] S. Colton, G. A. Wiggins *et al.*, "Computational creativity: The final frontier?" in *European Conference on Artificial Intelligence (ECAI)*, vol. 12. Montpelier, 2012, pp. 21–26.

[14] X. Zhang and M. Lapata, "Chinese poetry generation with recurrent neural networks," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2014, pp. 670–680.

[15] Q. Wang, T. Luo, D. Wang, and C. Xing, "Chinese song iambics generation with neural attention-based model," in *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*. AAAI Press, 2016, pp. 2943–2949.

[16] X. Yi, R. Li, and M. Sun, "Generating chinese classical poems with rnn encoder-decoder," in *Chinese Computational Linguistics and Natural Language Processing Based on Naturally Annotated Big Data*. Springer, 2017, pp. 211–223.

[17] J. Hopkins and D. Kiela, "Automatically generating rhythmic verse with neural networks," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics*, vol. 1, 2017, pp. 168–178.

[18] J. H. Lau, T. Cohn, T. Baldwin, J. Brooke, and A. Hammond, "Deep-speare: A joint neural model of poetic language, meter and rhyme," in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics*, vol. 1, 2018, pp. 1948–1958.

[19] A. Zugarini, S. Melacci, and M. Maggini, "Neural poetry: Learning to generate poems using syllables," in *International Conference on Artificial Neural Networks*. Springer, 2019, pp. 313–325.

[20] S. Colton, J. Goodwin, and T. Veale, "Full-face poetry generation." in *International Conference on Computational Creativity (ICCC)*, 2012, pp. 95–102.

[21] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[22] R. Paulus, C. Xiong, and R. Socher, "A deep reinforced model for abstractive summarization," *arXiv preprint arXiv:1705.04304*, 2017.

[23] Y.-C. Chen and M. Bansal, "Fast abstractive summarization with reinforce-selected sentence rewriting," in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics*, vol. 1, 2018, pp. 675–686.

[24] S. Narayan, S. B. Cohen, and M. Lapata, "Ranking sentences for extractive summarization with reinforcement learning," in *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, vol. 1, 2018, pp. 1747–1759.

[25] L. Bentivogli, M. Negri, M. Turchi *et al.*, "Machine translation for machines: the sentiment classification use case," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, 2019, pp. 1368–1374.

[26] L. Yu, W. Zhang, J. Wang, and Y. Yu, "Seqgan: Sequence generative adversarial nets with policy gradient," in *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[27] X. Yi, M. Sun, R. Li, and W. Li, "Automatic poetry generation with mutual reinforcement learning," in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, 2018, pp. 3143–3153.

[28] A. Belz and E. Reiter, "Comparing automatic and human evaluation of nlg systems," in *11th Conference of the European Chapter of the Association for Computational Linguistics*, 2006.

[29] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Advances in neural information processing systems*, 2014, pp. 2672–2680.

[30] Y.-L. Tuan, J. Zhang, Y. Li, and H.-y. Lee, "Proximal policy optimization and its dynamic version for sequence generation," *arXiv preprint arXiv:1808.07982*, 2018.

[31] K. Guu, T. B. Hashimoto, Y. Oren, and P. Liang, "Generating sentences by editing prototypes," *Transactions of the Association for Computational Linguistics*, vol. 6, pp. 437–450, 2018.

[32] Z. Cao, W. Li, S. Li, and F. Wei, "Retrieve, rerank and rewrite: Soft template based neural summarization," in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics*, vol. 1, 2018, pp. 152–161.

[33] Y. Li, X. Liang, Z. Hu, and E. P. Xing, "Hybrid retrieval-generation reinforced agent for medical image report generation," in *Advances in neural information processing systems*, 2018, pp. 1530–1540.

[34] J. Weston, E. Dinan, and A. H. Miller, "Retrieve and refine: Improved sequence generation models for dialogue," *arXiv preprint arXiv:1808.04776*, 2018.

[35] N. Hossain, M. Ghazvininejad, and L. Zettlemoyer, "Simple and effective retrieve-edit-rerank text generation," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, 2020, pp. 2532–2538.

[36] Y. Bengio, R. Ducharme, P. Vincent, and C. Jauvin, "A neural probabilistic language model," *Journal of machine learning research*, vol. 3, no. Feb, pp. 1137–1155, 2003.

[37] A. Holtzman, J. Buys, L. Du, M. Forbes, and Y. Choi, "The curious case of neural text degeneration," *arXiv preprint arXiv:1904.09751*, 2019.

[38] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.

[39] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," *arXiv preprint arXiv:1412.3555*, 2014.

[40] A. See, A. Pappu, R. Saxena, A. Yerukola, and C. D. Manning, "Do massively pretrained language models make better storytellers?" in *Proceedings of the 23rd Conference on Computational Natural Language Learning (CoNLL)*, 2019, pp. 843–861.

[41] O. Melamud, J. Goldberger, and I. Dagan, "context2vec: Learning generic context embedding with bidirectional lstm," in *Proceedings of The 20th SIGNLL Conference on Computational Natural Language Learning*, 2016, pp. 51–61.

[42] G. Marra, A. Zugarini, S. Melacci, and M. Maggini, "An unsupervised character-aware neural approach to word and context representation learning," in *International Conference on Artificial Neural Networks*. Springer, 2018, pp. 126–136.

[43] V. M. Babu, U. V. Krishna, and S. Shahensha, "An autonomous path finding robot using q-learning," in *2016 10th International Conference on Intelligent Systems and Control (ISCO)*. IEEE, 2016, pp. 1–6.

[44] M. Knudson and K. Tumer, "Policy search and policy gradient methods for autonomous navigation," in *and Learning Agents Workshop at AAMAS 2010*, 2010.

[45] L. Pasqualini and M. Parton, "Pseudo random number generation: a reinforcement learning approach," *Procedia Computer Science*, vol. 170, pp. 1122–1127, 2020.

[46] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Advances in neural information processing systems*, 2000, pp. 1057–1063.

[47] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel, "High-dimensional continuous control using generalized advantage estimation," *arXiv preprint arXiv:1506.02438*, 2015.

[48] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization," in *International conference on machine learning*, 2015, pp. 1889–1897.

[49] J. Brooke, A. Hammond, and G. Hirst, "Gutentag: an nlp-driven tool for digital humanities research in the project gutenberg corpus," in *Proceedings of the Fourth Workshop on Computational Linguistics for Literature*, 2015, pp. 42–47.

[50] S. E. Elayoubi, S. B. Jemaa, Z. Altman, and A. Galindo-Serrano, "5G RAN slicing for verticals: Enablers and challenges," *IEEE Commun. Mag.*, vol. 57, no. 1, pp. 28–34, 2019.

[51] P. Popovski, K. Trillingsgaard, O. Simeone, and G. Durisi, "5G Wireless Network Slicing for eMBB, URLLC, and mMTC: A Communication-Theoretic View," *IEEE Access*, vol. 6, pp. 55 765–55 779, 2018.

[52] C. She, C. Yang, and T. Q. S. Quek, "Radio Resource Management for Ultra-Reliable and Low-Latency Communications," *IEEE Commun. Mag.*, vol. 55, no. 6, pp. 72–78, 2017.

[53] A. Anand and G. de Veciana, "Resource Allocation and HARQ Optimization for URLLC Traffic in 5G Wireless Networks," *IEEE J. Sel. Areas Commun.*, vol. 36, no. 11, pp. 2411–2421, 2018.

[54] A. Anand, G. de Veciana, and S. Shakkottai, "Joint Scheduling of URLLC and eMBB Traffic in 5G Wireless Networks," *IEEE/ACM Trans. Netw.*, vol. 28, no. 2, pp. 477–490, 2020.

[55] J. Tang, B. Shim, and T. Q. S. Quek, "Service Multiplexing and Revenue Maximization in Sliced C-RAN Incorporated With URLLC and Multicast eMBB," *IEEE J. Sel. Areas Commun.*, vol. 37, no. 4, pp. 881–895, 2019.

[56] M. Elsayed and M. Erol-Kantarci, "AI-Enabled Radio Resource Allocation in 5G for URLLC and eMBB Users," in *2019 IEEE 2nd 5G World Forum (5GWF)*, 2019, pp. 590–595.

[57] Y. Li, C. Hu, J. Wang, and M. Xu, "Optimization of URLLC and eMBB Multiplexing via Deep Reinforcement Learning," in *2019 IEEE/CIC International Conference on Communications Workshops in China (ICCC Workshops)*, 2019, pp. 245–250.

[58] Y. Huang, S. Li, C. Li, Y. T. Hou, and W. Lou, "A Deep-Reinforcement-Learning-Based Approach to Dynamic eMBB/URLLC Multiplexing in 5G NR," *IEEE Internet Things J.*, vol. 7, no. 7, pp. 6439–6456, 2020.

[59] M. Alsenwi, N. H. Tran, M. Bennis, S. R. Pandey, A. K. Bairagi, and C. S. Hong, "Intelligent resource slicing for eMBB and URLLC coexistence in 5G and beyond: A deep reinforcement learning based approach," *IEEE Trans. Wireless Commun.*, pp. 1–1, 2021.

[60] P. He, L. Zhao, S. Zhou, and Z. Niu, "Water-Filling: A Geometric Approach and its Application to Solve Generalized Radio Resource Allocation Problems," *IEEE Trans. Wireless Commun.*, vol. 12, no. 7, pp. 3637–3647, 2013.

[61] G. M. S. Rahman, M. Peng, K. Zhang, and S. Chen, "Radio Resource Allocation for Achieving Ultra-Low Latency in Fog Radio Access Networks," *IEEE Access*, vol. 6, pp. 17 442–17 454, 2018.

[62] OpenAI, "Proximal Policy Optimization," OpenAI web site, 2018, https://spinningup.openai.com/en/latest/algorithms/ppo.html.

[63] D. Tse and P. Viswanath, *Fundamentals of Wireless Communication*. USA: Cambridge University Press, 2005.

[64] F. Saggese, M. Moretti, and P. Popovski, "Power minimization of downlink spectrum slicing for embb and urllc users," 2021, Online: https://arxiv.org/abs/2110.14544.

[65] B. Bai, W. Chen, K. B. Letaief, and Z. Cao, "Outage exponent: A unified performance metric for parallel fading channels," *IEEE Trans. Inf. Theory*, vol. 59, no. 3, pp. 1657–1677, 2013.

[66] J. P. Coon, D. E. Simmons, and M. D. Renzo, "Approximating the outage probability of parallel fading channels," *IEEE Commun. Lett.*, vol. 19, no. 12, pp. 2190–2193, 2015.

[67] A. M. Gergely and B. Crainicu, "A succinct survey on (pseudo)-random number generators from a cryptographic perspective," in *2017 5th International Symposium on Digital Forensic and Security (ISDFS)*, 2017, pp. 1–6.

[68] L. E. Bassham III, A. L. Rukhin, J. Soto, J. R. Nechvatal, M. E. Smid, E. B. Barker, S. D. Leigh, M. Levenson, M. Vangel, D. L. Banks *et al.*, "Sp 800-22 rev. 1a. a statistical test suite for random and pseudorandom number generators for cryptographic applications," 2010.

[69] V. Desai, R. Patil, and D. Rao, "Using layer recurrent neural network to generate pseudo random number sequences," *International Journal of Computer Science Issues*, vol. 9, no. 2, pp. 324–334, 2012.

[70] J. M. Hughes, "Pseudo-random number generation using binary recurrent neural networks," Ph.D. dissertation, 2007.

[71] H. Abdi, "A neural network primer," *Journal of Biological Systems*, vol. 2, no. 03, pp. 247–281, 1994.

[72] M. De Bernardi, M. Khouzani, and P. Malacaria, "Pseudo-random number generation using generative adversarial networks," in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 2018, pp. 191–200.

[73] R. Bellman, "Dynamic programming," *Science*, vol. 153, no. 3731, pp. 34–37, 1966.

[74] S. Duell, S. Udluft, and V. Sterzing, "Solving partially observable reinforcement learning problems with recurrent neural networks," in *Neural Networks: Tricks of the Trade*. Springer, 2012, pp. 709–733.

[75] L. Wang, W. Zhang, X. He, and H. Zha, "Supervised Reinforcement Learning with Recurrent Neural Network for Dynamic Treatment Recommendation," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, ser. KDD '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 2447–2456. [Online]. Available: https://doi.org/10.1145/3219819.3219961

[76] S. Chakraborty, "Capturing Financial markets to apply Deep Reinforcement Learning," *arXiv:1907.04373*, 2019. [Online]. Available: https://ideas.repec.org/p/arx/papers/1907.04373.html

[77] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[78] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE transactions on neural networks*, vol. 5, no. 2, pp. 157–166, 1994.

[79] T. Zahavy, M. Haroush, N. Merlis, D. J. Mankowitz, and S. Mannor, "Learn what not to learn: Action elimination with deep reinforcement learning," in *Advances in Neural Information Processing Systems*, 2018, pp. 3562–3573.