Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea Magistrale in Informatica

REINFORCEMENT LEARNING IN ROGUE

Relatore:
Chiar.mo Prof.
ANDREA ASPERTI

Presentata da:
DANIELE CORTESI

Sessione I
Anno Accademico 2018/2019

To Mirta and Giovanni

Introduction

Reinforcement learning (RL) is a machine learning framework that involves learning to interact with an environment in such a way that maximizes a numerical reward signal, without human supervision. This is potentially better than developing hard-coded programs that interact with the environment, due to the thorough knowledge of its mechanisms that the latter approach requires. No such understanding is needed in RL, that can nonetheless produce surprisingly good policies exclusively by trial-and-error, although it can certainly be included when available. Moreover, RL has been able to devise *better* and *novel* strategies than those previously known: e.g. in the game of Backgammon [44] it learned unprecedented opening moves good enough to be adopted by professional players. In problems encompassing large state spaces — i.e. exhibiting an intractable amount of configurations, like Backgammon or Chess board positions — some approximation method is fundamental. Neural networks (see section 1.2.3) are powerful nonlinear function approximators inspired to biological brains that are often preferred for this task and we will be the basis of our work.

Recently RL has attracted a lot of attention due to the results attained by Mnih et al. [29] in Atari 2600 games. The authors developed a novel Q-learning algorithm and had a neural network, that they call deep Q-network (DQN), learn to play with fixed hyper-parameters the games available on the platform from almost raw pixels and using only the score as a reward, achieving expert human level play on many of them and super human levels on several others. This break through inspired a considerable amount of en-

deavors, summarized in [26], that improved on the results either by enhancing DQN or by entirely different approaches.

This work aims at automatically learning to play, by RL and neural networks, the famous Rogue, a milestone in videogame history that we describe throughly in chapter 2. It introduced several mechanics at its core, spawning the entire rogue-like genre, like procedural (i.e. random) level generation, *permadeath* (i.e. no level replay) and other aspects discussed in section 2.2, that make it very challenging for a human and constitute an interesting RL benchmark. Related works on this game started with Pedrini's thesis [34], to which our own can be considered a spiritual successor, and continued with [3, 4]. Several problems emerged there that we successfully addressed here, enabling us to obtain much better results and investigate new scenarios where we encountered further obstacles. Games are possibly the most popular testing ground for RL methods, because they are designed to challenge human skills and are often simplified simulations of reality. Many environments exist for training RL agents to play games, such as [6, 8] for arcade games, [8] for some continuous control tasks, [22] for the famous DOOM and very recently [46] for StarCraft II (please see section 1.3 for more details and other RL applications). Rogueinabox [34, 3] is the sole environment for Rogue we are aware of, see section 2.3 for more details and rogue-like environments.

In the following, we introduce the RL theoretical background in chapter 1 and then present Rogue in chapter 2. We proceed to discuss our experiments in chapter 3 and then draw the conclusions.

Contents

List of Figures

List of Tables

Chapter 1

Reinforcement Learning

Reinforcement learning (RL) is a machine learning setting involving an agent interacting with an *environment* that changes in relation to the *actions* performed. The agent receives a numerical *reward signal* for each action taken and seeks to maximize the cumulative reward obtained in the long run, despite the uncertainty about the environment. Since actions affect the opportunities available at later times, the correct choices require taking into account their indirect and delayed consequences, which may require foresight or planning. The RL framework is an abstraction of the problem of goal-directed learning from interaction, in which all relevant details are reduced to the environment *states*, the *actions* performed and the *rewards* consequently received, which define the goal of the problem. RL is strongly linked with psychology and neuroscience: of all forms of machine learning, it is the closest to the way that humans and other animals learn [41].

The main characteristics of RL are:

- being a closed-loop problem, such that the actions taken influence the later inputs, available actions and rewards;
- not having direct instructions as to what actions to take;
- the consequences of actions, and subsequent rewards, play out over extended periods of time.

RL differs from *supervised learning* in that there is no set of labeled examples provided by an external supervisor. In essence, it uses training information that *evaluates* its actions rather than *instructing* them by providing correct samples. Such feedback indicates how good the action taken is, but not whether it's the best or worst one possible. It would also be impractical to form a set of such examples that is both correct and representative of all the situations in which the agent is expected to act optimally. Moreover, RL is renown for producing optimal behaviors that were previously unknown, such as in the games of Backgammon [44] and Go [38, 39].

RL is also different from *unsupervised learning*, as it tries to maximize a reward signal instead of uncovering a hidden structure in unlabeled data.

To make the distinction even clearer, there is an important issue that arises only in RL: the trade-off between *exploration* and *exploitation*. An agent must in fact prefer actions it has tried in the past that were found to be effective in producing reward, however to discover such actions it has to try ones it has not selected before. The point is that neither exploration nor exploitation alone are sufficient to succeed at the task: they must be combined, typically by progressively shifting the focus from exploration to exploitation.

1.1 Elements of RL

In this section we present the main elements of the RL framework.

1.1.1 Agent

The entity continually interacting with the environment. The first selects actions according to a *policy* and the the second responds by presenting new observations and rewards. The agent can either be a complete organism or a component of a larger system. The boundary between the agent and the environment is usually drawn very close to the agent: for example, if it has arms then they are considered part of the environment and in general so

is everything that cannot be arbitrarily changed by the agent. The agent-environment boundary represents the limit of the agent's *absolute control*, but not of its knowledge, in fact it may know to any degree how the rewards are computed as a function of its actions.

1.1.2 Environment

In general a Partially Observable Markov Decision Process (POMDP) [33, 30]. Formally, a MDP is a tuple $(\mathcal{S}, \mathcal{A}, p)$, where:

- $\mathcal{S}$ is the set of possible states;
- $\mathcal{A}$ is the set of possible actions, with $\mathcal{A}(s_t)$ denoting the actions available at time t in state $s_t \in \mathcal{S}$;
- $p : \mathcal{S} \times \mathbb{R} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$ is the probability function of state transitions, such that $p(s, r|s_t, a_t)$ denotes the probability of a *next state-reward* pair following a a *state-action* pair at time t. Often $p(s|s_t, a_t)$ is used to denote the probability of transitioning to state s without considering a specific reward.

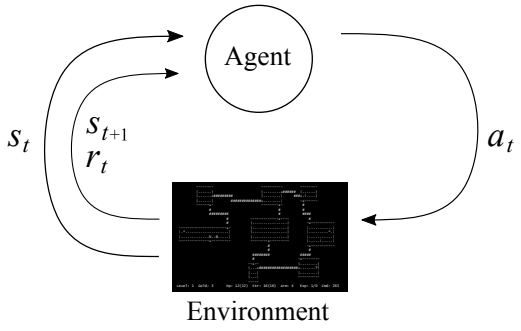


Figure 1.1: Markov Decision Process dynamics

The interaction between the agent and the environment is divided in discrete time steps $t = 0, 1, 2, \ldots$ At each time step the agent receives a representation of the environment state $s_t \in \mathcal{S}$, selects an action $a_t \in \mathcal{A}(s_t)$

according to a policy π and then receives the next state and reward s_{t+1}, r_t sampled from $p(\cdot, \cdot|s_t, a_t)$. We define a state-action *trajectory* the sequence $s_t, a_t, s_{t+1}, a_{t+1}, \ldots, s_T$ of states observed and actions taken from time t to T. The steps are not required to refer to fixed intervals of time and likewise the actions may be low or high level controls, such as motor voltages or the decision of a destination, respectively. We will solely consider the *episodic case* in which the interaction with the environment naturally breaks down in episodes, i.e. in many independent finite trajectories ending in a final state. The opposite case, called *continuing case*, is also studied and described in [41].

Most of RL theory is developed assuming the *Markov Property*, an attribute that the states presented by the environment have if they contain all relevant information for predicting future states, e.g. the position of all pieces in a chess game. The algorithms are nonetheless successfully applied often times even in its absence: people can make very good decisions in non-Markov tasks, e.g. poker, so arguably this should not be a severe problem for a RL agent.

1.1.3 Model

A system that is able to predict how the environment will behave. It can be used for planning, i.e. deciding the sequence of future actions considering possible future situations before they are actually experienced. Methods for solving RL problems that use models and planning are called *model-based*, while simpler methods that are exclusively trial-and-error learners are referred to as *model-free*. The latter are much more effective than they may appear, which makes RL very powerful indeed: complex sequences of actions that maximize the future cumulative reward received can be learned without any prior knowledge of the environment dynamics. In this work will focus on model-free approaches and will not discuss the model-based alternative, which is explained in [41].

1.1.4 Policy

The definition of the agent's way of behaving at a given time. It is a mapping, or a probability distribution, from perceived states of the environment to actions to be taken. Formally, a policy is a function $\pi : \mathcal{A} \times \mathcal{S} \rightarrow [0, 1]$ and the probability of selecting action a in state s_t at time step t is denoted $\pi_t(a|s_t)$.

1.1.5 Reward signal

The definition of the goal in a RL problem. On each time step, the environment sends the agent a number representing the reward. The objective of the learning agent is to maximize the total reward received in the long run. Importantly the process generating the reward must be unalterable by the agent and its definition should be devised with care: we must reward only *what* we want the agent to achieve, but not *how* to achieve it. If subgoals are rewarded, e.g. taking enemy pieces in a chess game, the agent might learn just to accomplish such subgoals instead of what we really want to achieve, e.g. winning the chess game.

1.1.6 Value function

The total amount of reward the agent can expect to accumulate over the future, starting from a given state. This represents an indication of the *long-term* desirability of states, e.g. one may yield a low immediate reward but be regularly followed by states that yield high rewards, or the opposite. More formally, the value function corresponds to the *expected return* of a state s. The return is defined as $G_t = \sum_{k=0}^{T} \gamma^k r_{t+k}$ where T is the final time step and $\gamma \in [0, 1]$ is a *discount factor*, which regulates how strongly future rewards are taken into consideration. Hence a value function is defined as:

$$v_\pi(s) = \mathbb{E}_\pi[G_t|s_t = s] \tag{1.1}$$

which denotes the expected return of starting in s and then following the policy π. This is called *state-value function* for policy π. The learning agent should seek actions that bring about states of highest value and not highest reward, because these are the actions that will obtain the greatest amount of reward over the long run. Values must be estimated and re-estimated from the sequences of observations an agent makes over its entire lifetime: as such a method for efficiently estimating values is crucial.

Another important function is q_π, the *action-value function*, which is defined as:

$$q_\pi(s,a) = \mathbb{E}_\pi[G_t|s_t = s, a_t = a] \tag{1.2}$$

The q and v functions are related in the following way:

$$v_\pi(s) = \sum_a \pi(a|s) q_\pi(s,a) \tag{1.3}$$

$$q(s,a) = \sum_{s',r} p(s',r|s,a)[r + \gamma v_\pi(s')] \tag{1.4}$$

State value functions satisfy a recursive relationship, known as the **Bellman equation**:

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[G_t|s_t = s] \\ &= \mathbb{E}_\pi[r_t + \gamma G_{t+1}|s_t = s] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)[r + \gamma \mathbb{E}_\pi[G_{t+1}|s_{t+1} = s']] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)[r + \gamma v_\pi(s')], \ \forall s \in \mathcal{S} \end{aligned} \tag{1.5}$$

An analogous equation can be derived for action value functions. This rule forms the basis of many ways to approximate v_π via update or backup operations, that transfer value information back to a state from its successors, or to state-action pairs from subsequent pairs.

1.1.7 Optimal policy and value functions

A policy π is defined to be better than or equal to another policy π', i.e. $\pi \geq \pi'$, if and only if $v_\pi(s) \geq v'_\pi(s)$ $\forall s \in \mathcal{S}$. A policy that is better or equal

to all other policies is defined to be an *optimal policy* and is denoted by π_*. All optimal policies have the same optimal state and action value functions, defined as: $v_*(s) = max_\pi\, v_\pi(s)$ and $q_*(s,a) = max_\pi\, q_\pi(s,a)$. These equations can also be written without referencing any policy, in a form known as the **Bellman optimality equations**:

$$\begin{aligned} v_*(s) &= \max_a \mathbb{E}[G_t | s_t = s, a_t = a] \\ &= \max_a \mathbb{E}[r_t + \gamma G_{t+1} | s_t = s, a_t = a] \\ &= \max_a \mathbb{E}[r_t + \gamma v_*(s_{t+1}) | s_t = s, a_t = a] \\ &= \max_a \sum_{s',r} p(s', r | s, a)[r + \gamma v_*(s')] \end{aligned} \tag{1.6}$$

$$q_*(s,a) = \sum_{s',r} p(s', r | s, a)[r + \gamma \max_{a'} q_*(s', a')] \tag{1.7}$$

Once we have q_* it's easy to formulate an optimal policy: simply taking one of the actions that maximizes $q_*(s, \cdot)$ will do. In literature, those actions are called *greedy actions* and the policy is referred to as a *greedy policy*. Greedy does not imply optimal in general, however since q_* is the estimate of the future return, and not of the future one-step reward, then a greedy policy always selects the action that maximizes the expected cumulative reward.

Using exclusively v_* requires complete knowledge of the environment's dynamics, i.e. $p(s', r|s, a)$, that would in principle allow to solve v_* (and also q_*). However having this kind of information is rare and usually the state space of the problem is so large, e.g. in chess, that it would require thousand of years. This is actually what the *value iteration* [41] algorithm does: the procedure is divided in iterations, each of which computes a more precise estimate of v_* by a formula directly derived from equation (1.6).

In practice, q_* or v_* are often estimated with increasing precision *during* the interaction with the environment, using the gathered experience. This kind of learning is usually referred to as *on-line learning*. Moreover, an optimal policy may not even require that the value of all states or state-action pairs be estimated, only the fraction that is frequently encountered.

In order to maintain a balance between exploration and exploitation in online learning, an ϵ-*greedy* policy is often used: such a policy selects the greedy action with respect to the current estimation of q_* or v_* with probability $1-\epsilon$ and a random action otherwise. When greedy actions are chosen, then we are *exploiting* our current knowledge of the environment, otherwise we are *exploring*.

1.2 RL algorithms

In this sections we will describe some the most well known RL algorithms. Some of them can framed in a *generalized policy iteration* (GPI) scheme, while others in the policy gradient framework.

In GPI the processes of policy *evaluation* and *improvement* are alternated. The policy evaluation process computes v_π, or more generally brings its estimate at a given time closer to its true value. The policy improvement process makes the current policy π greedy with respect to the updated estimate of v_π. The two processes pull in opposing directions, because policy improvement makes the value function incorrect for the new policy, while policy evaluation causes π to no longer be greedy. Their interaction however results in the convergence to optimality.

In policy gradient methods, the mapping π is learned directly via some *gradient ascent* technique on $\mathbb{E}[G_t]$. This has some advantages over GPI that we discuss in section 1.2.6.

1.2.1 On-policy vs Off-policy

Algorithms are said to be *on-policy* if they improve the same policy that is used to interact with the environment, and *off-policy* if the policy improved, called *target policy* is different than the one used to make decisions, called *behavior policy*, which generates the training data and may be arbitrary in general. In both cases, the policy used for the interaction is required to take all actions with a probability strictly higher than zero and thereby is often

ϵ-*greedy*. In fact, exploration can only stop in the limit of an infinite number of actions in order to be sure there are no actions that are actually better than those favored at a given time. On-policy learning produce policies that are only near-optimal, because they can never stop exploring. Off-policy learning is more powerful and includes on-policy methods as a special case, but it usually suffers from greater variance and is slower to converge.

Importance sampling

Off-policy methods usually make use of *importance sampling*, a technique for estimating expected values under one distribution given samples from another, which is used to weights the returns according to the relative probability of their trajectories occurring under the target and behavior policies, called *importance-sampling ratio*. The probability of a state-action trajectory $s_t, a_t, s_{t+1}, a_{t+1}, \dots, s_T$ occurring under policy π is $\prod_{k=t}^{T-1} \pi(a_k|s_k)p(s_{k+1}|s_k, a_k)$. Let μ be the behavior policy, then the importance-sampling ratio is:

$$\rho_{t:T-1} = \frac{\prod_{k=t}^{T-1} \pi(a_k|s_k)p(s_{k+1}|s_k, a_k)}{\prod_{k=t}^{T-1} \mu(a_k|s_k)p(s_{k+1}|s_k, a_k)} = \prod_{k=t}^{T-1} \frac{\pi(a_k|s_k)}{\mu(a_k|s_k)} \tag{1.8}$$

which has the interesting property of being independent of the environment dynamics. We also define $\rho_t = \rho_{t:t} = \frac{\pi(a_t|s_t)}{\mu(a_t|s_t)}$.

Importance sampling is necessary to compute an estimate of v_π that is correct in relation to the policy π and *unbiased*, however it can be the source of high variance, because the product in (1.8) is potentially unbounded.

1.2.2 Bootstrapping and temporal-difference learning

A RL algorithm is said to *bootstrap* if the updates it makes are based on estimates. This kind of update is at the heart of *temporal-difference* (TD) learning, a technique for estimating state or action value functions. The target for the update of the simplest TD method, called *TD(0)* or *one-step TD*, is $r_t + \gamma V(s_{t+1})$ where V is the estimate of the value function with respect

to a policy at a given time. The complete TD(0) update is:

$$V(s_t) \leftarrow V(s_t) + \alpha[r_t + \gamma V(s_{t+1}) - V(s_t)] \tag{1.9}$$

where $\alpha \in [0, 1]$ is called *step-size parameter* that controls the rate of learning and the starting value of V is arbitrary. The difference $\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$ is called the *TD error* and arises in various forms throughout RL. This form of TD evaluates every state $s \in \mathcal{S}$ and we refer to it as the *tabular case*. Since the target of the update involves the estimate of V, we say that TD(0) *bootstraps* and that it is *biased*. This bias is often beneficial, reducing variance and accelerating learning. The bootstrapping of one step TD methods enables learning to be fully on-line and incremental, since each update only requires a single environment transition.

The generalization of one-step TD is called *n-step* TD learning, and is based on the next n rewards and the estimated value of the state n steps later. The update rule thus involves *n-step* returns, defined as:

$$G_{t:t+n} = \sum_{k=0}^{n-1} \gamma^k r_{t+k} + \gamma^n V_{t+n-1}(s_{t+n}) \tag{1.10}$$

After n steps are made, the *n-step* TD update rule can be applied:

$$V_{t+n}(s_t) \leftarrow V_{t+n-1}(s_t) + \alpha[G_{t:t+n} - V_{t+n}(s_t)] \tag{1.11}$$

The value of n that results in faster learning in practice depends on the problem and involves a trade-off: higher values of n allows a single update to take into account more future actions, at the cost of performing the first n without actually learning anything.

1.2.3 Function approximation and neural networks

In real world problems, we cannot hope to store a value for every single state in the state space, as is required by tabular methods. This is because state sets sizes are often exponential in the space required to represent a single state, which is the case for the game we want to focus on, Rogue.

In these cases we must to resort to *function approximation*, a scalable way of generalizing from spaces much larger than computational resources. The tools of choice for function approximation are artificial neural networks, particularly deep convolutional neural networks, that are behind many of the recent successes in numerous fields of machine learning, especially computer vision [40, 42] and RL [29, 35, 28, 47]

Neural networks (NNs) are one of the most used methods of nonlinear function approximation. NNs are networks of interconnected units, the *neurons*, inspired to biological nervous systems. To each interconnection is associated a real number: together these are the trainable parameters of NNs and are often called *weights*. The neurons are logically divided in layers, connected to those immediately preceding and following. The first layer is called *input* layer, the last *output* layer and in between there can be an arbitrary number of *hidden* layers: if these are present, the NN is referred to as a Deep Neural Network (DNN).

The most used layers are:

Dense or Fully Connected (FC) computes $x_{i,j} = \sigma(\sum_k W_{j,k} x_{i-1,k})$ where $x_{i,j}$ denotes the output of the j-th neuron of the i-th layer and W its associated weight matrix. The function σ is the component that introduces nonlinearity in NNs: usually a *rectified linear unit* (ReLU) is preferred [32], defined as $\sigma(x) = max\{0, x\}$. The number of parameters of this kind of layer equals to input dimension times output dimension, so care should be taken when used on large inputs.

Convolution employs one or several *windows*, also called *filters*, that scan the input by moving the window on top of it, producing an output for each location. The window is moved according to *stride* values and its weights, referred to as *kernel*, are very low in quantity: only the size of kernel times the number of filters. Convolutions are known to be very powerful feature extractors and are especially good at 2-dimensional image processing, where they can learn to identify characteristics such as edges or other very localized patterns.

Recurrent layers are made of units that, in addition to the input from the preceding layer, also receive their own output at the previous step, called internal state. Via a built-in learned mechanism that combines the two, the units may "forget" (part of) the internal state and consider to a certain degree the input. Recurrent layers can thus be considered a form of learned memory of past inputs. The Long-Short Term Memory (LSTM) [18] was developed specifically to deal with the *vanishing gradient* problem [17], which is a real issue with a naive implementation of a recurrent network, because due to the *chain-rule* the gradient might involve a large number of factors close to zero multiplied together.

In this work we will describe several network architectures mainly by using images: refer to the legend in figure 1.2 for the meaning of each component.

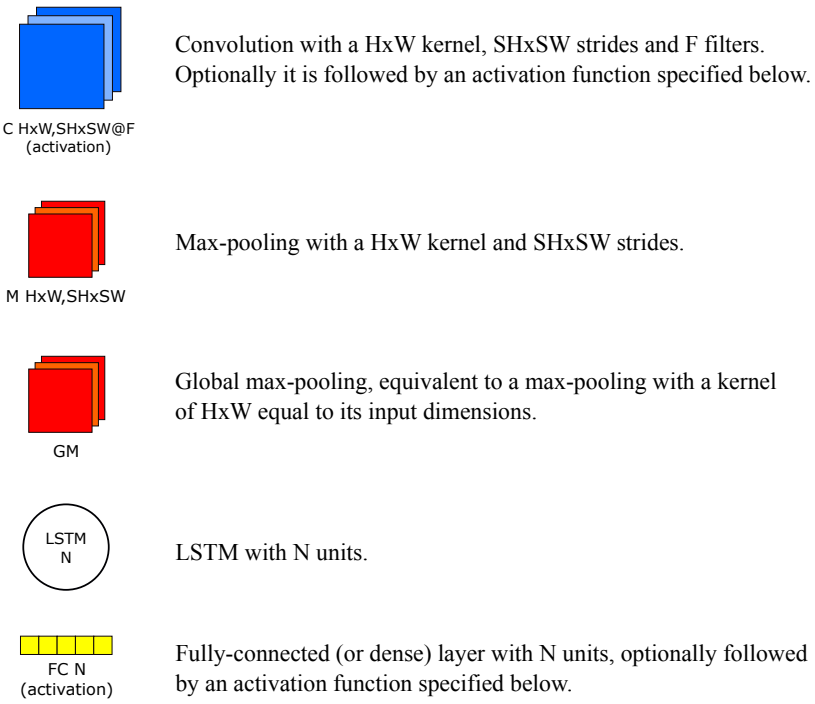


Figure 1.2: Neural network legend

1.2.4 The Deadly Triad

We face an important issue however if we combine function approximation, bootstrapping and off-policy training, known as the *deadly triad* [41]. The combination produces a well known danger of instability and divergence, due to the following factors: the sequence of observed states presents correlations, small updates in (action-)value function estimate may result in a significant change in the policy and thereby alter the data distribution (because an update could change which action maximizes the function estimate), and finally (action-)values and target values are correlated due to bootstrapping. The deadly triad was successfully addressed in [29], although only empirically without any theoretical guarantee, paving the way to a fair amount of work, summarized in [26].

1.2.5 Q-learning and DQN

Q-learning [49, 48] is an off-policy GPI TD method for estimating $Q \approx q_*$ demonstrated to converge to the optimal solution, at least in the tabular case, so long as all actions are repeatedly sampled in all states and the action-values are represented discretely, constituting one of the early breakthroughs in RL. We present tabular one-step Q-learning in Algorithm 1. Since the interaction with the environment is carried out by an ϵ-*greedy* policy on Q and its update rule actually evaluates a completely greedy policy, the algorithm classifies as off-policy.

A more recent breakthrough was achieved with *Deep Q-Networks* (DQNs) in [29], combining Q-learning with nonlinear function approximation and dealing with the deadly triad issue (section 1.2.4). In that work, deep convolutional neural networks were used to approximate the optimal action-value function q_* and play several Atari 2600 games on the Arcade Learning Environment (ALE) [6] directly from (almost) raw pixels, in many cases reaching and surpassing expert human level scores. The results were very remarkable because no game-specific prior knowledge was involved beyond the prepro-

cessing of frames (that only consists in converting colors to luminance values, stacking the last 4 frames due to artifacts of the old Atari platform and down-scaling them to save computational resources) and the very same set of parameters was used across all games. The main elements that enabled the success, addressing the instability issues, were *experience replay* and periodic updates of target values. Experience replay is a biologically inspired mechanism in which a number of state-action transitions $(s_t, a_t, r_t, s'_{t+1})$ are stored and sampled for training, randomizing over the data hence reducing correlations in the observation sequence and smoothing over changes in the data distribution. The periodic update of target values, opposed to an immediate one, is implemented by using two separate parameter (or weight) vectors θ_i and θ_i^-. The two vectors represent, respectively, the DQN parameters used to select actions and those used to compute the target at iteration i. The target parameters θ_i^- are only updated with θ_i every C steps, adding a delay that reduces correlation with the targets and making divergence more unlikely. The authors also found that clipping the TD error term to be in $[-1, 1]$ further improved the stability of the algorithm. Pseudo-code is shown in Algorithm 2.

The results of DQN were improved in [35] by prioritizing experience replay, so that important experience transitions could be replayed more frequently, thereby learning more efficiently. The importance of experience transitions were measured by TD errors, such that these were proportional to the probability of being inserted in the experience memory buffer. The authors also made use of importance sampling to avoid the bias in the update distribution.

1.2.6 Policy gradient and REINFORCE

We turn now the attention to methods that directly learn a *parametrized policy* without consulting (action-)value functions, called *policy gradient methods*. As with DQN, we will denote the parameters with θ and write $\pi(a|s;\theta)$ for the probability of taking action a in state s with parameters θ. The most

Algorithm 1 Tabular one-step Q-learning pseudo-code, adapted from [41]

initialize Q arbitrarily and $Q(\text{terminal-state}, \cdot) = 0$
for each episode **do**
    initialize state s
    for each step of episode, state s is not terminal **do**
        $a \leftarrow$ action derived from $Q(s, \cdot)$ (e.g. ϵ-greedy)
        take action a, observe r, s'
        $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma\, max_{a'}\, Q(s',a') - Q(s,a)]$
        $s \leftarrow s'$

Algorithm 2 DQN pseudo-code, adapted from [29]

initialize replay memory D with capacity N
initialize action-value function Q with random weights θ
initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
for each episode **do**
    initialize state s_1
    for each step t of episode, state s_t is not terminal **do**
        $$a_t \leftarrow \begin{cases} \text{random } a & \text{with probability } \epsilon \\ argmax_a\, Q(s_t, a; \theta) & \text{otherwise} \end{cases}$$
        take action a_t, observe r_t, s_{t+1}
        store transition (s_t, a_t, r_t, s_{t+1}) in D
        sample random minibatch of transitions (s_j, a_j, r_j, s_{j+1}) from D
        $$y_j \leftarrow \begin{cases} r_j & \text{if } s_{j+1} \text{ is terminal} \\ r_j + \gamma\, max_{a'}\, \hat{Q}(s_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$$
        perform gradient descent step on $(y_j - Q(s_j, a_j; \theta))^2$ w.r.t. θ
        every C steps reset $\hat{Q} \leftarrow Q$, i.e. set $\theta^- \leftarrow \theta$

important advantage over (approximate) TD methods is that the continuous direct policy parametrization does not suffer from sudden changes in action probabilities — one of the issues of the deadly triad — enabling stronger the-

oretical convergence guarantees [41]. Moreover, policy gradient techniques can express arbitrary stochastic optimal policies, which is not natural in GPI, and π can approach determinism, while ϵ-greedy action selection always has an ϵ probability of selecting a random action.

The REINFORCE [52] algorithm is a well known and simple *Monte-Carlo* policy gradient method. Monte-Carlo algorithms are the extreme of *n-step* methods on the opposite side of one-step TD. These methods always use the length of the episode as n and do not bootstrap. As such they are unbiased, but in practice exhibit greater variance and are slower to converge. REINFORCE employs the following update rule, derived from the *policy gradient theorem* [41]:

$$\theta_{t+1} = \theta_t + \alpha\gamma^t G \nabla_\theta log\ \pi(a_t|s_t, \theta_t) \tag{1.12}$$

Since the algorithm updates the same policy used for interacting with the environment, it belongs to the on-policy category.

(1.12) can be generalized to include an arbitrary baseline, as long as it does not vary with the action a:

$$\theta_{t+1} = \theta_t + \alpha\gamma^t [G - b(s_t)] \nabla_\theta log\ \pi(a_t|s_t, \theta_t) \tag{1.13}$$

The baseline can significantly reduce the variance of the update and thereby speed up the learning process. Commonly, a learned estimate of the value function, $V(s; \theta_v)$, is the choice for baseline. We denote its parameters with θ_v to indicate at the same time that in general they can be independent from the policy parameters θ, but in practice are shared to some degree. For example, θ and θ_v could denote the weights of a neural network with two separate output layers, one for π and the other for V, branching from a common structure of hidden layers, i.e. all non-output layers are shared. This is the case in the recent literature, e.g. A3C and ACER (figure 1.3), as well as in all our architectures (figures 3.1 and 3.3).

We present the pseudo-code of REINFORCE with baseline in Algorithm 3.

Algorithm 3 REINFORCE with baseline pseudo-code, adapted from [41]

initialize policy and state-value parameters θ and θ_v arbitrarily
for each episode **do**
    generate a trajectory $s0, a0, r0, \ldots, s_T, a_T, r_T$ following $\pi(\cdot|\cdot;\theta)$
    for each step t of episode **do**
        $G_t \leftarrow$ return from step t
        $\delta \leftarrow G_t - V(s_t, \theta_v)$
        $\theta_v \leftarrow \theta_v + \beta\delta\nabla_{\theta_v} V(s_t, \theta_v)$ ▷ $\beta \in [0, 1]$ is a step-size parameter
        $\theta \leftarrow \theta + \alpha\gamma^t\delta\nabla_\theta log\ \pi(a_t|s_t, \theta)$

1.2.7 Actor-Critic and A3C

Policy gradient methods that learn a value function and use bootstrapping are called *actor-critic*, where actor references the learned policy while critic refers to the learned value function. With bootstrapping, actor-critic methods re-introduce the TD error in the update rule. This enables learning to be fully on-line, on the contrary of REINFORCE, which as Monte-Carlo method must experience an entire episode before learning can begin.

A simple one step actor-critic algorithm would use the following update rule:

$$\theta_{t+1} \leftarrow \theta_t + \alpha\gamma^t(r_t + \gamma V(s_{t+1};\theta_v) - V(s_t;\theta_v))\nabla_\theta log\ \pi(a_t|s_t,\theta) \tag{1.14}$$

The actor-critic framework has been the center of attention of the latest RL endeavors [28, 37, 47, 19]. The Asynchronous Advantage Actor-Critic (A3C) algorithm [28] is a particular method that improved the state-of-the-art results of its time on Atari 2600 games and other tasks using several parallel actor-critic learners independently experiencing the environment, a component that stabilizes learning without the need for experience replay. The parallel nature induced a faster learning time with less resources, using moderately powerful CPUs instead of very powerful GPUs. The algorithm selects actions using its policy for up to t_{max} steps or until a terminal state is reached, receiving up to t_{max} rewards from the environment since its last update. Then the gradients for n-step updates are computed for each of the

state-action pairs encountered since the last update. Each n-step update uses the longest possible n-step return: a one-step update for the last state, a two-step update for the second last state, and so on. The accumulated updates are applied in a single gradient step. This is done in a number of parallel threads, each interacting with an independent instance of the environment with a local copy (θ' and θ'_v) of a set of global parameters (θ and θ_v), asynchronously updating the latter at each iteration in an intended non thread-safe way in order to maximize throughput. A3C parameterize the policy π and the baseline V with a neural network with two output layers, see figure 1.3.

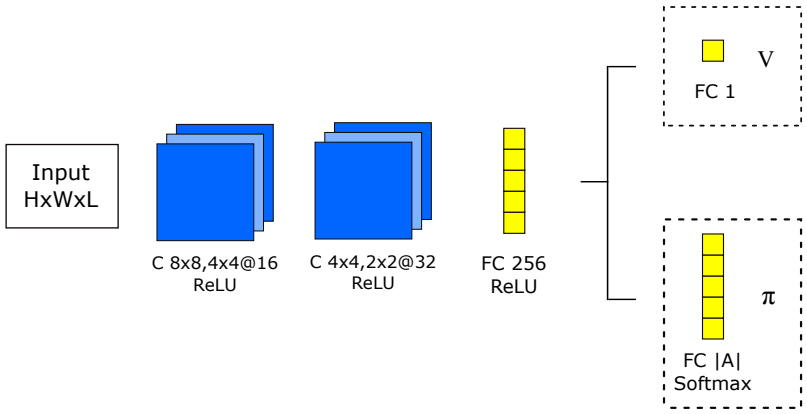


Figure 1.3: A3C neural network

The A3C policy update rule is:

$$\theta_{t+1} \leftarrow \theta_t + \nabla_\theta log\ \pi(a_t|s_t,\theta)A(s_t,a_t;\theta,\theta_v) + \beta \sum_{i=0}^{k-1} \nabla_\theta H(\pi(\cdot|s_{t+i};\theta)) \quad (1.15)$$

$$A(s_t,a_t;\theta,\theta_v) = \sum_{i=0}^{k-1} \gamma^i r_{t+i} + \gamma^k V(s_{t+k};\theta_v) - V(s_t;\theta_v) \quad (1.16)$$

where:

- k varies from state to state and is upper bounded by t_{max};

- A is an estimate of the advantage function $a_\pi(s_t, a_t) = q_\pi(s_t, a_t) - v_\pi(s_t)$, expressing the advantage of taking action a_t in state s_t and then acting according to π;

- H is the Shannon's entropy function, that the authors found to be particularly helpful on tasks requiring hierarchical behavior, encouraging exploration and preventing premature convergence to suboptimal policies, with β controlling the strength of the entropy regularization term.

The pseudo-code of an A3C actor-critic learner is shown in Algorithm 4.

A2C is the *synchronous* version of A3C, employed in [37] with the same or better results than A3C. Synchronous means that the trajectories experienced by the parallel actor-learners are collected and a single parameters update is computed. This allows a better exploitation of the parallel computing power of a GPU, reducing the wall-clock time of training, but not the actual number of training steps required to achieve a certain performance measure.

A3C and A2C display an issue known as *sample inefficiency*: they require a great amount of experience steps to reach a fixed performance score, much more than, e.g., DQN. Even if at the end of the training they find better policies, simulations steps can be expensive and sample efficiency become crucial, even more so when agents are deployed in the real world.

1.2.8 ACER

The Actor-Critic with Experience Replay (ACER) algorithm [47] combines the A3C framework with experience replay, *marginal importance weights*, the *Retrace* target [31] to learn Q, a technique the authors call *truncation with bias correction trick* and a more efficient version of Trust Region Policy Optimization (TRPO) [36], improving on the sample efficiency of A3C. Due to experience replay, ACER classifies as an off-policy algorithm.

Algorithm 4 A3C pseudo-code for an actor-learner thread, adapted from [28]

▷ assume global shared parameter vectors θ and θ_v
▷ assume global shared counter $T = 0$
▷ assume thread-specific parameter vectors θ' and θ'_v
initialize thread step counter $t \leftarrow 1$
repeat
    reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$
    synchronize thread-specific parameters $\theta' \leftarrow \theta$ and $\theta'_v \leftarrow \theta_v$
    $t_{start} \leftarrow t$
    get state s_t
    repeat
        sample $a_t \sim \pi(\cdot|s_t; \theta')$
        perform a_t, observe r_t, s_{t+1}
        $t \leftarrow t + 1$
        $T \leftarrow T + 1$
    until terminal s_t or $t - t_{start} = t_{max}$
    $R \leftarrow \begin{cases} 0 & \text{if } s_t \text{ is terminal} \\ V(s_t; \theta'_v) & \text{otherwise} \end{cases}$
    for $i \leftarrow t - 1$ down-to t_{start} **do**
        $R \leftarrow r_i + \gamma R$
        $d\theta \leftarrow d\theta + \nabla_{\theta'} log\ \pi(a_i|s_i; \theta')(R - V(s_i; \theta'_v)) + \beta \nabla_{\theta'} H(\pi(\cdot|s_i; \theta'))$
        $d\theta_v \leftarrow d\theta_v + \frac{\partial}{\partial \theta'_v}(R - V(s_i; \theta'_v))^2$
    perform asynchronous update of θ using $d\theta$ and of θ_v using $d\theta_v$
until $T > T_{max}$

In ACER, a neural network parameterize π and Q, instead of V as in A3C, because the method is based on marginal value functions [13]. They still make use of V as the baseline, which can be simply computed given π and Q as per equation (1.3). Without TRPO, the ACER update rule would be:

$$\theta_{t+1} \leftarrow \theta_t + \bar{\rho}_t \nabla_\theta log\ \pi(a_t|s_t;\theta)[Q^{ret}(s_t,a_t) - V(s_t;\theta_v)] + \mathop{\mathbb{E}}_{a\sim\pi}\left(\left[\frac{\rho_t(a)-c}{\rho_t(a)}\right]_+ \nabla_\theta log\ \pi(a|s_t;\theta)[Q(s_t,a;\theta_v) - V(s_t;\theta_v)]\right)$$

where:

- Q^{ret} is the *Retrace* target [31];
- ρ_t is referred to as *marginal importance weight* and is expected to cause less variance than a complete importance sampling ratio, since it does not involve the product of many potentially unbounded factors;
- $\rho_t(a) = \frac{\pi(a|s_t;\theta)}{\mu(a|s_t)}$, where μ is the policy that was used to take the action, which may differ from π during experience replay;
- $\bar{\rho}_t = min\{c, \rho_t\}$ is the *truncated importance weight*, with c being its maximum value. This clipping ensures that the variance of the update is bounded;
- $[x]_+ = max\{0, x\}$. This term in the lower part of the equation ensures that the estimate is unbiased and activates only when $\rho_t(a) > c$ and is at most 1;
- The above two points make up the *truncation with bias correction trick*.

With TRPO, the update is corrected so that the resulting policy does not deviate too far from an *average policy network* representing a mean of past policies. The authors decompose the policy network in two parts: a distribution f and a deep neural network that generates its statistics $\phi_\theta(s)$, such that the policy is completely characterized by $\phi_\theta : \pi(\cdot|s;\theta) = f(\cdot|\phi_\theta(s))$.

Algorithm 5 ACER pseudo-code for an actor-learner, adapted from [47]

▷ assume global shared parameter vectors θ, θ_v and θ_a
▷ assume ratio of replay r

repeat
  call ACER on-policy
  $n \leftarrow \text{Poisson}(r)$
  for n times **do**
    call ACER off-policy
until Max iteration or time reached

function ACER(on-policy?)
  reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$
  synchronize thread-specific parameters $\theta' \leftarrow \theta$ and $\theta'_v \leftarrow \theta_v$
  if not on-policy? **then**
    sample trajectory $\{s_0, a_0, r_0, \mu(\cdot|s_0), \cdots, s_k, a_k, r_k, \mu(\cdot|s_k)\}$
  else
    get state s_0
  for $i \leftarrow 0$ to k **do**
    compute $f(\cdot|\phi_{\theta'}(s_i))$, $Q(s_i, \cdot; \theta'_v)$ and $f(\cdot|\phi_{\theta_a}(s_i))$
    if on-policy? **then**
      sample $a_i \sim f(\cdot|\phi_{\theta'}(s_i))$
      perform a_i, observe r_i, s_{i+1}
      $\mu(\cdot|s_i) \leftarrow f(\cdot|\phi_{\theta'}(s_i))$
    $\bar{\rho}_i \leftarrow min\left\{1, \frac{f(a_i|\phi_{\theta'}(s_i))}{\mu(a_i|s_i)}\right\}$
    $$Q^{ret} \leftarrow \begin{cases} 0 & \text{if } s_t \text{ is terminal} \\ \sum_a Q(s_k, a; \theta'_v) f(a|\phi_{\theta'}(s_k)) & \text{otherwise} \end{cases}$$
  for $i \leftarrow k$ down-to 0 **do**
    $Q^{ret} \leftarrow r_i + \gamma Q^{ret}$
    $V_i \leftarrow \sum_a Q(s_i, a; \theta'_v) f(a|\phi_{\theta'}(s_i))$
    $g \leftarrow min\{c, \rho_i(a_i)\} \nabla_{\phi_{\theta'}(s_i)} log\ f(a|\phi_{\theta'}(s_i))(Q^{ret} - V_i)$
      $+ \sum_a \left[1 - \frac{c}{\rho_i(a)}\right]_+ f(a|\phi_{\theta'}(s_i)) \nabla_{\phi_{\theta'}(s_i)} log\ f(a|\phi_{\theta'}(s_i))(Q(s_i, a_i; \theta'_v) - V_i)$
    $k \leftarrow \nabla_{\phi_{\theta'}(s_i)} D_{KL}[f(\cdot|\phi_{\theta_a}(s_i))||f(\cdot|\phi_{\theta'}(s_i))]$
    $d\theta \leftarrow d\theta + \frac{\partial_{\phi_{\theta'}(s_i)}}{\partial \theta'}(g - max\left\{0, \frac{k^T g - \delta}{||k||_2^2}\right\} k)$
    $d\theta_v \leftarrow d\theta_v + \nabla_{\theta'_v}(Q^{ret} - Q(s_i, a_i; \theta'_v))^2$
    $Q^{ret} \leftarrow \bar{\rho}_i(Q^{ret} - Q(s_i, a_i; \theta'_v)) + V_i$
  perform asynchronous update of θ using $d\theta$ and of θ_v using $d\theta_v$
  update the average policy network: $\theta_a \leftarrow \alpha\theta_a + (1 - \alpha)\theta$

The update is constrained by the KL divergence between the distribution derived from the current and the average policy. The parameters θ_a of the average network are updated *softly*, whenever the policy is changed, by the rule $\theta_a \leftarrow \alpha\theta_a + (1-\alpha)\theta$. We present ACER pseudo-code in Algorithm 5.

1.3 RL applications

Videogames possibly represent the major RL domain and are certainly an important AI test bed, but they are not by any stretch the only application of the RL framework. In this section we outline some notable fields in which RL has been used (for a more thorough description see [26]):

Games are useful AI benchmarks as they are often designed to challenge human cognitive capacities. Many RL environments exist for games, both for discrete and contiuos actions, such as the Arcade Learning Environment (ALE) [6], featuring arcade Atari 2600 games, OpenAI Gym [8], also featuring Atari games but also others involving continuous actions, VizDoom [22], that allows interacting with the popular Doom videogame, one of the fathers of First Person Shooter (FPS) games and a StarCraft II environment [46], a very popular Real-Time Strategy (RTS) videogame. The game of our focus is Rogue, that we extensively describe in chapter 2.

Robotics [23] offers an important and interesting platform for RL: the real-world challenges of this domain pose a major real-world check for RL methods. Robotics usually involves controlling torque's at the robot's motor, a task with continuous actions, harder than discrete actions domains.

Natural Language Processing (NLP) where deep learning has recently been permeating and RL has been applied, for instance, in language tree-structure learning, question answering, summarization and sentiment analysis.

Computer vision where RL has been used, e.g., to focus on selected sequence of regions from image or video frames for image classification and object detection.

Business management can benefit from RL, for example, in commercially relevant tasks such as personalized content or ads recommendation.

Finance offers delicate tasks suitable for RL such as trading and risk management.

Healthcare presents interesting and important tasks, e.g. personalized medicine, dynamic treatment regimes and adaptive treatment strategies, where issues that are not standard in RL arise.

Intelligent transportation systems where RL can be applied to important and current tasks such as adaptive traffic signal control and self-driving vehicles.

Chapter 2

Rogue

In this chapter we describe Rogue, the videogame of our focus, and why it is an interesting problem for Reinforcement Learning (RL).

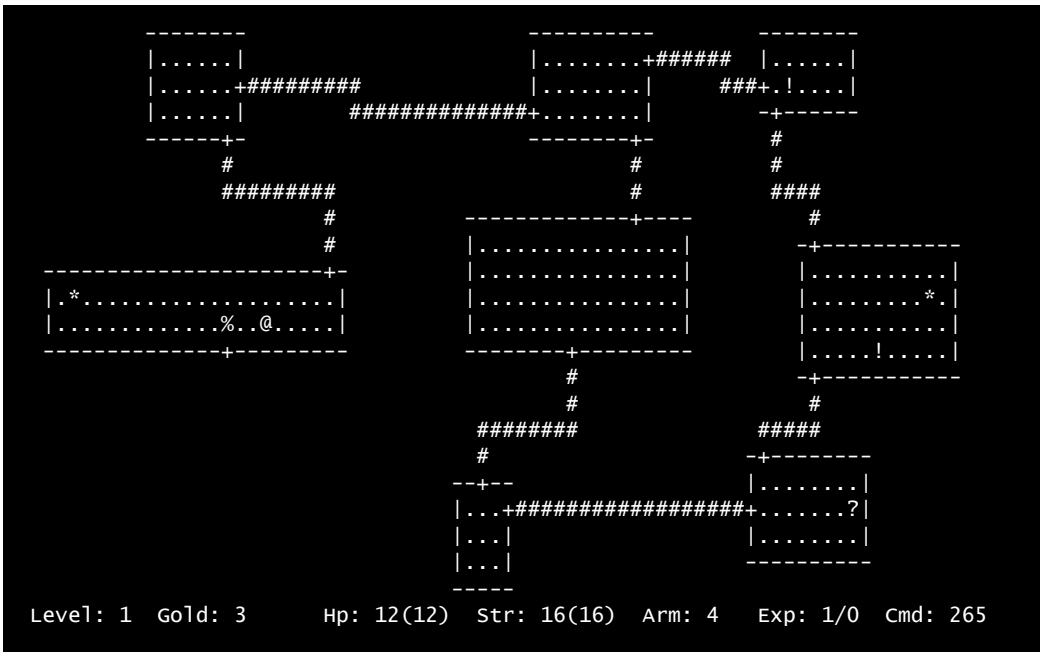


Figure 2.1: A Rogue screenshot

2.1 A game from the 80ies

Rogue, also known as *Rogue: Exploring the Dungeons of Doom* is a dungeon crawling video game, father of the rogue-like genre, by Michael Toy and Glenn Wichman and later contributions by Ken Arnold. Rogue was originally developed around 1980 for Unix-based mainframe systems as a freely-distributed executable.

In Rogue, the player controls a character, the rogue, exploring several levels of a dungeon seeking the *Amulet of Yendor*, located on a specific level. The player must fend off an array of monsters that roam the dungeons and, along the way, they can collect treasures that can help them, offensively or defensively, such as weapons, armor, potions, scrolls and other magical items. Rogue is *turn-based*, taking place on a grid world represented in ASCII characters, allowing players unlimited time to determine the best move to survive, while the world around is frozen in time. Rogue implements *permadeath* as a design choice to make each action meaningful: should the player-character lose all their health from combat or other means, the character is dead, and the player must restart a brand new character and cannot reload from a saved state. The dungeon levels, monster encounters, and treasures are procedurally generated on each playthrough, so that no game is the same as a previous one. As the first game presenting these as its core mechanics, Rogue is an important milestone in videogames history.

2.2 A difficult RL problem

There are many factors that make Rogue difficult and interesting for RL, some of which have already been mentioned in the preceding section. Some of these make it so hard that it is not conceivable to deal with them at the current level of technology and state-of-the-art methods. The problem has already been studied in previous work [34, 3, 4], for completeness we summarize here the challenges offered by the game:

POMDP nature Rogue is a Partially Observable Markov Decision Process (see section 1.1.2). The layout of each level of the dungeon is initially unknown and partially hidden, and is progressively discovered as the rogue crawls the dungeon. Solving partially observable mazes is a notoriously difficult and challenging task [50, 41, 21] Deep learning approaches were investigated in [51, 21], however the considered problems were different and simpler than the challenges offered by Rogue and in the case of [21] the authors focused on *imitation learning* rather than RL. Imitation learning is very akin to supervised learning, in which a policy is learned from examples produced by another policy that is generally supposed to be optimal (e.g. human expert actions).

Procedural generation and no level-replay Rogue dungeons are procedurally generated: whenever a new game is started (e.g. when the player dies) the levels will be randomly generated and different from previous ones. Replaying a previously experienced dungeon is thereby forbidden, unlike most videogames that allow restarting the same level without any alteration when losing. The procedural generation, even if it has constraints (e.g. the number of rooms is at most nine), means that level-specific learning can't be deployed with good results. It has been shown in [43] that simple convolutional networks can only learn to navigate sufficiently small (8×8), completely observable 2-dimensional grid mazes and are not able to generalize in larger spaces. The authors argue that *learning to plan* seems to be required for this kind of task.

Complex mechanics The game offers many different challenges:

- exploring the dungeon searching for the Amulet;
- finding and descending the stairs to the next level;
- discovering hidden areas, which may even conceal the stairs or the Amulet;
- fighting hostile monsters, avoiding death;

- collecting items, such as food to avoid starving and weapons to improve the chances of surviving fights;
- using the gathered items through the interaction with an inventory menu.

Learning to successfully engage in all of these activities in a completely end-to-end unsupervised way is a very difficult task, especially the discovery of hidden areas and a sensible use of the inventory, possibly beyond the current state-of-the-art.

Memory and Attention Both are important machine learning topics and both seems to be important for Rogue.

Memory is needed, e.g., to remember whether the Amulet was recovered because after that in order to win the game the stairs should be ascended instead of descended, which are different actions. Another scenario that requires memory is the discovery of hidden areas: suppose that a corridor terminates in what appears to be a dead-end, but actually continues into a room. If the wall is searched with a specific action it will reveal the continuation of the corridor, however the number of times the action should be performed is stochastic, usually requiring at most 10 attempts. Long-Short Term Memory (LSTM) [18] neural network units seem to be the natural choice for tackling these kind of issues and we will employ them in this work. LSTM units have been used in other RL tasks with good results, some examples are [51, 16, 19].

Attention is the ability to focus on specific parts of interest while ignoring others of lesser relevance, typical of human cognition. In Rogue, the part of the screen immediately surrounding the player is the one that most intuitively requires attention, especially for deciding the next short term action. Attention has been extensively investigated in recent works, such as [20, 15, 45], and seems to be a central topic for the future of machine learning.

Sparse rewards Rogue has no frequently increasing player score: there is a status bar in the lower part of the screen, showing values such as the current dungeon level, health and gold, however they vary sporadically. The quantity of gold recovered can serve as a score measure, however it is only minor with respect to the real objective of the game - recovering the Amulet of Yendor - that determines whether the game is won or lost. A game that was won is obviously better than one that was lost, even if more gold was attained in the latter.

Being an environment with sparse rewards is a trait shared with *Montezuma's Revenge*, renown as one of the most complex Atari 2600 games. Neither DQN, A3C or ACER were able to devise effective policies for this game, where complex sequences of actions must be learned without reinforcement before attaining any variation in score and, thereby, reward. Devising some sort of *intrinsic motivation* seems to be required for these kind of task, i.e. a problem independent reward added to the environment's own extrinsic reward. A successful and theoretically based approach to this issue is presented in [5], where the authors employ a count-based exploration bonus, designed to be useful in domains with large state spaces where a state is rarely visited more than once. Good results were achieved even in Montezuma's Revenge using this approach.

2.3 Rogueinabox

In this work we develop RL agents in Python that interact with the game via the *Rogueinabox* library[1], developed in [34, 3] and updated in [4]. This is a modular and configurable environment, that allows the use of custom state representations and reward functions. To our knowledge, this is the sole AI environment for Rogue, while for rogue-likes we are aware of [9] for Desktop Dungeons and [24] for Nethack, an evolution of Rogue.

[1] `https://github.com/rogueinabox/rogueinabox_lib`

We contributed ourselves to the library, mainly by:

- Implementing in the Rogue source code the customization of several options with command line parameters:
  - Whether to enable monsters, implemented in [2] with a compile-time flag;
  - Whether to enable hidden areas;
  - Random seed;
  - Amulet level;
  - Number of steps before the rogue is affected by hunger;
  - Number of traps;
- Realizing an agent wrapper base-class and an implementation that records all game frames on file;
- Refactoring several aspects;
- Allowing the users customize some library behavior;
- Documenting most of the code;

Chapter 3

Learning to play Rogue

In this chapter we describe our objectives, the simplifications we introduced to the problem, our attempts to create an agent capable of learning to play the game, the resulting policies and our evaluation metrics.

Our implementations use the Tensorflow library [1] for Python[1], we will link to each of them in the respective sections. In all of our experiments we employ the RMSProp optimizer[2], possibly the most popular gradient descent algorithm in RL, used in [29, 28, 19, 43, 47]

3.1 Problem simplification

Due to the complex mechanics outlined in section 2.2 we introduced some simplifications so that the problem becomes approachable with current state-of-the-art methods. In particular:

1. We initially limited ourselves to find the stairs of the first level and descend them, instead of looking for the Amulet of Yendor; given our good results we later expanded on this, see section 3.2;

[1] `https://docs.python.org/3.5/library/index.html`

[2] `http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`

2. We disabled monsters and hunger, so that fighting and inventory management were not part of the problem;

3. We disabled hidden areas, aware of the difficulty of discovering them in an end-to-end way, due to the unpredictable number of search actions required to uncover them;

4. We drastically limit the number of actions available to our agents: movement by one cell in the four cardinal directions and interaction with the stairs (descent/ascent). The game actually encompasses a much wider spectrum of commands, such as moving in a direction until an obstacle, inspecting the inventory, equipping items, eating food, drinking potions, etc.

What we were left with are randomly generated partially observable mazes, that because of Rogue's no-replay can only be experienced once. The resulting task is still challenging enough to be interesting (see section 2.2), but not so difficult as to be unapproachable.

3.2 Objectives

In Rogue the player wins the game when, after descending a fixed number of levels, they recover the Amulet of Yendor and climb back up through all levels, although these are not the same that were descended. By default the amulet is located at level 26 and we deemed this challenge too difficult for learning agents, even with the simplifications previously described. Instead, we formulate and tackle the following objectives:

1. Descend the first level;

2. Descend until the tenth level;

3. Recover the Amulet from early levels.

3.3 Descending the first level

We set our first objective to develop an agent capable of reliably finding and descending the stairs of the first level. By default this level has no hidden areas, even if they are enabled.

Previous work [3, 4] employed DQN (section 1.2.5) and were able to descend the stairs in the first level in 23% of games. In comparison, a completely random agent attains 7%. They faced many problems, mainly their agent did not seem able to learn to backtrack when it found itself at a dead-end and had a hard time getting away from walls once it got next to them. We overcame these issues with different algorithms, state representations and reward functions. We devised two different approaches, that we describe in this section.

3.3.1 Evaluation criteria

When developing several methods to solve a task it's important to establish a well-defined set of metrics under which each different effort becomes comparable. We expand on the criteria used in [4] and base our evaluation on the following statistics:

1. The average number of episodes in which the agent is able to descend the stairs; when this happens, we declare the episode won and reset the game;
2. The average number of steps taken when climbing down;
3. The average return;
4. The average number of tiles seen;

Our emphasis is on the first two points, but we also keep an eye on the others. In all of our experiments, we average these values over the most recent 200 games played at a given training step. Since we always employ several parallel actors, in the statistics we display in the various plots the

values are the average *over the averages* of each actor. In these and all other evaluation metrics, when victory conditions are not met within the maximum amount of actions established, then the episode is considered lost and the game reset.

Whenever two methods achieve the same results, we prefer the one that employs the smallest number of non-learned features or mechanisms. When they are equal even in that regard, we prefer the most *sample efficient* one, i.e. that which produces such results in a lower amount of training steps.

3.3.2 Partitioned A3C with cropped view

We describe here our first approach[3], that was published in [2] in collaboration with Francesco Sovrano — who developed most of the code — and accepted to LOD 2018[4], the Fourth International Conference on Machine Learning, Optimization, and Data Science. The method can be summarized in the following points:

1. The A3C algorithm;
2. The use of *situations*, a technique we developed to partition different category of states such that for each category a specific policy is learned, parameterized by a specific neural network, a *situational agent*;
3. A *cropped view*, i.e. a representation of the Rogue screen centered on the player and comprising only a portion of fixed size of their immediate surroundings, cropping out everything outside of it;
4. A neural network with a *Long-Short Term Memory* (LSTM) layer;
5. A reward signal not only encouraging descending the stairs, but also the exploration of the level and punishing actions that result in the rogue not moving.

[3] `https://github.com/Francesco-Sovrano/Partitioned-A3C-for-RogueInABox`
[4] `https://lod2018.icas.xyz/`

Neural network architecture

In previous work [34, 3, 4] some forms of handcrafted non-learned memory was used, such as a long-term *heatmap* with color intensities proportional to how many times the rogue walked on each tile and a short-term *snake-like* memory, representing the most recent rogue positions. These were provided as input to the neural network, instead of using any type of recurrent neural unit, which isn't very satisfying from a machine learning perspective.

In this work we decided to forgo those kind of handcrafted memories and employ in all of our neural network models a Long-Short Term Memory (LSTM) layer, that we described in section 1.2.3. When using a recurrent unit in RL we must take care how to perform backpropagation. There are two important matters to consider: preserving the *temporal sequence* of the steps — that both A3C and ACER pseudo code do not do (they actually reverse it) — and which initial recurrent hidden state to use for gradient descent. In our implementations we ensure the temporal preservation and use different initial states for backpropagation in A3C and ACER. In the former case, we store the recurrent state before each n-step update and then employ it for gradient descent. In the latter case, we always use an hidden state completely filled with zeros. Both approaches seem to perform equally well and it would be interesting to test which one actually achieves better results under identical circumstances. In appendix A some code snippets on this issue can be found.

The architecture we used is shown in 3.1. The network is fundamentally an extension of the structure used in A3C (figure 1.3): it consists of two convolutional layers followed by a fully connected (FC or dense) layer to process spatial dependencies, then a LSTM layer to process temporal dependencies. From here the structure branches into value and policy output layers. The total number of parameters of this model is almost 3 millions.

The network input is the state representation that we describe later in this section, while the LSTM also receives as input a numerical *one-hot* representation of the action taken in the previous state, concatenated to the obtained

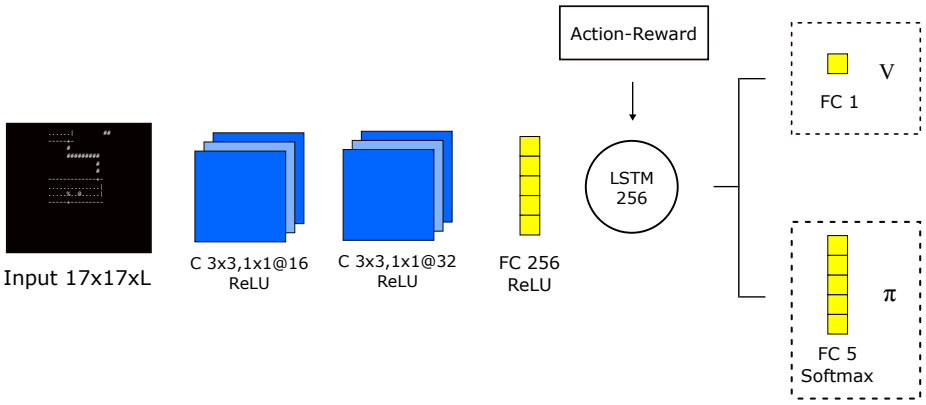


Figure 3.1: Neural network architecture for partitioned A3C

reward, inspired by [19]. Suppose that action a_j was taken, then the one-hot vector representing it is x such that $x_j = 1$ and $x_i = 0\ \forall a_i \in \mathcal{A}, i \neq j$. This entails our network is actually computing π and V as a function of (s_t, a_{t-1}, r_t) rather than just s_t. Later we will employ ACER on the same conditions as A3C: there the model will compute its output exclusively on s_t, showing that the richer A3C input is not influential.

Situations

With the term *situation* we denote a subset of environment states sharing a common characteristic, used to discriminate which situational agent should perform the next action. Such an agent is parameterized by the neural network model we just described and is completely independent from the others and do not share any parameter. The only way they interact is by contributing to the same return: the update rule for each situational agent takes into account the rewards received by the others that acted later. Please see the pseudo-code in Algorithm 6 for more details. The situations partition the task in a way that is reminiscent of hierarchical models [50, 10, 25], where generally a top-level model selects which sub-level model should interact with the environment, that at a later point returns control to the top-level system.

We experimented with three sets of situations, the first, that we call *s4*

Algorithm 6 Partitioned A3C pseudo-code for an actor-learner thread

▷ assume global shared counter $T = 0$
▷ assume global shared parameter vectors θ_z and $\theta_{z,v}$ for each situation z
▷ assume thread-specific parameter vectors θ'_z and $\theta'_{z,v}$
initialize thread step counter $t \leftarrow 0$
repeat
  for each situation z **do**
    reset gradients: $d\theta_z \leftarrow 0$ and $d\theta_{z,v} \leftarrow 0$
    synchronize local parameters $\theta'_z \leftarrow \theta_z$ and $\theta'_{z,v} \leftarrow \theta_{z,v}$
  $t_{start} \leftarrow t$
  get state s_t
  repeat
    compute situation z_t from s_t
    sample $a_t \sim \pi(\cdot|s_t; \theta'_{z_t})$
    perform a_t, observe r_t, s_{t+1}
    $t \leftarrow t + 1$
  until terminal s_t or $t - t_{start} = t_{max}$
  $T \leftarrow T + (t - t_{start})$
  $$Rt \leftarrow \begin{cases} 0 & \text{if } s_t \text{ is terminal} \\ V(s_t; \theta'_{z_t,v}) & \text{otherwise, } z_t \text{ computed from } s_t \end{cases}$$
  for $i \leftarrow t - 1$ down-to t_{start} **do**
    $R \leftarrow r_i + \gamma R$
    $$d\theta_{z_i} \leftarrow d\theta_{z_i} + \nabla_{\theta'_{z_i}} log\ \pi(a_i|s_i; \theta'_{z_i})(R - V(s_i; \theta'_{z_i,v})) + \beta \nabla_{\theta'_{z_i}} H(\pi(\cdot|s_i; \theta'_{z_i}))$$
    $$d\theta_{z_i,v} \leftarrow d\theta_{z_i,v} + \frac{\partial}{\partial \theta'_v}(R - V(s_i; \theta'_{z_i,v}))^2$$
  for each situation z **do**
    perform asynchronous update of θ_z using $d\theta_z$ and of $\theta_{z,v}$ using $d\theta_{z,v}$
until $T > T_{max}$

listed below from higher to lower priority:

1. The rogue stands on a corridor;
2. The stairs are visible;
3. The rogue is next to a wall;
4. Any other case.

The situations are determined programmatically and are not learned. When multiple conditions in the above list are met, the one with higher priority will be selected. For instance, if the stairs are visible but the rogue is walking on a corridor, the situation is determined to be (1) rather than (2), because the former has higher priority.

We define the second set of situations *s2* as :

1. The stairs are visible
2. The stairs are not visible

and the third set, *s1*, has no situations at all or, equivalently, a single situation.

State representation

The state is a 17×17 matrix corresponding to a cropped view of the map centered on the rogue (i.e. the rogue position is always on the center of the matrix). This representation has the advantage to be sufficiently small to be fed to FC layers and implicitly represents the position of the player. Moreover, in principle it could be used for any 2-dimensional maze arbitrarily larger than Rogue.

In our experiments we adopted two variations of the above matrix. The first, called *c1*, has a single channel, resulting in a $17 \times 17 \times 1$ shape, filled with the following values:

4 for stairs

8 for walls

16 for doors and corridors

0 everywhere else

The second variation, called *c2*, is made of two channels: one entirely dedicated to the stairs and the other to the rest of the environment. The state representation thereby has shape $17 \times 17 \times 2$, with the following values:

First channel

8 for walls

16 for doors and corridors

0 everywhere else

Second channel

4 for stairs

0 everywhere else

Previous work [34, 3, 4] did not use a cropped view, but rather employed a representation of the entire map not centered on the rogue. This is similar to what we make use of in section 3.3.3 (where we shall describe it in more details), albeit at a slightly higher level of abstraction and with the addition of handcrafted memory that we discussed earlier in the subsection related to the neural network.

Reward signal

We designed the following reward function:

+1 when stepping on a door for the first time from the inside of a room

+1 when one or more doors are discovered

+10	when the agent descends the stairs
-0.01	when the agent remains stationary, e.g. tries to move into a wall
0	otherwise

The reward values are chosen such that a significant amount of it can be gained only by descending the stairs, while the rest for exploring the dungeon. Each floor contains at most 9 rooms and each room no more than 4 doors, thus only about $\frac{2}{3}$ of the cumulative reward is awarded for exploring, while negative rewards are enough to teach the agent not to take useless actions but not significantly affect the balance between level exploration and stairs descent.

In previous work [34, 3, 4] a similar reward function was used, that differs in encouraging the discovery of new tiles (more general than our version, but less sparse) and a small negative "living" reward, given for each step.

Hyper-Parameters

Each episode lasts at most 500 steps/actions and may end sooner if the agent achieves success (i.e. descends the stairs). The rogue cannot die, since monsters are disabled and the default number of steps before hunger begins to affect the player is 1300. Most hyper-parameters values we used are from [27], an Open-Source implementation of [19], upon which we based our own. The values are summarized in table 3.1. The learning rate is annealed over time according to the following equation: $\alpha = \eta \cdot \frac{T_{max}-T}{T_{max}}$, where T_{max} is the maximum global step, and T is the current global step.

Parameter	Value
parallel actors	16
episode max length	500
entropy β	0.001
discount factor γ	0.95
batch size t_{max}	60
initial learning rate η	0.0007
rms decay	0.99
rms momentum	0
rms epsilon	0.1
rms clip norm	40

Table 3.1: Hyper-parameters for partitioned A3C with cropped view

Results

Agent	*s1-c2*	*s2-c2*	*s4-c1*	*s4-c2*
Success rate	0.03%	98%	96.5%	97.6%
Avg return	16.16	17.97	17.66	17.99
Avg number of seen tiles	655	386	366	389
Avg number of steps to succeed	2	111	108	110

Table 3.2: Partitioned A3C with cropped view final results

We plot our results in figure 3.2 and summarize them in table 3.2. Our best agent[5] shows remarkable skill in searching for the stairs and descending them. In particular, it has little troubles in backtracking through already explored parts of the labyrinth when coming at an impasse and rarely displays "uncertainty" (e.g. stepping left and then immediately right). Rarely does not mean never, of course: in certain dead-end scenarios the agent starts moving erratically without an obvious destination, remaining in the same room and exceeding the maximum number of steps per episode. We noted however that in a few cases, if given more time, it would eventually enter a corridor and successfully retrace its steps and find the stairs.

[5]A video of our agent playing is available at `https://youtu.be/1j6_165Q46w`

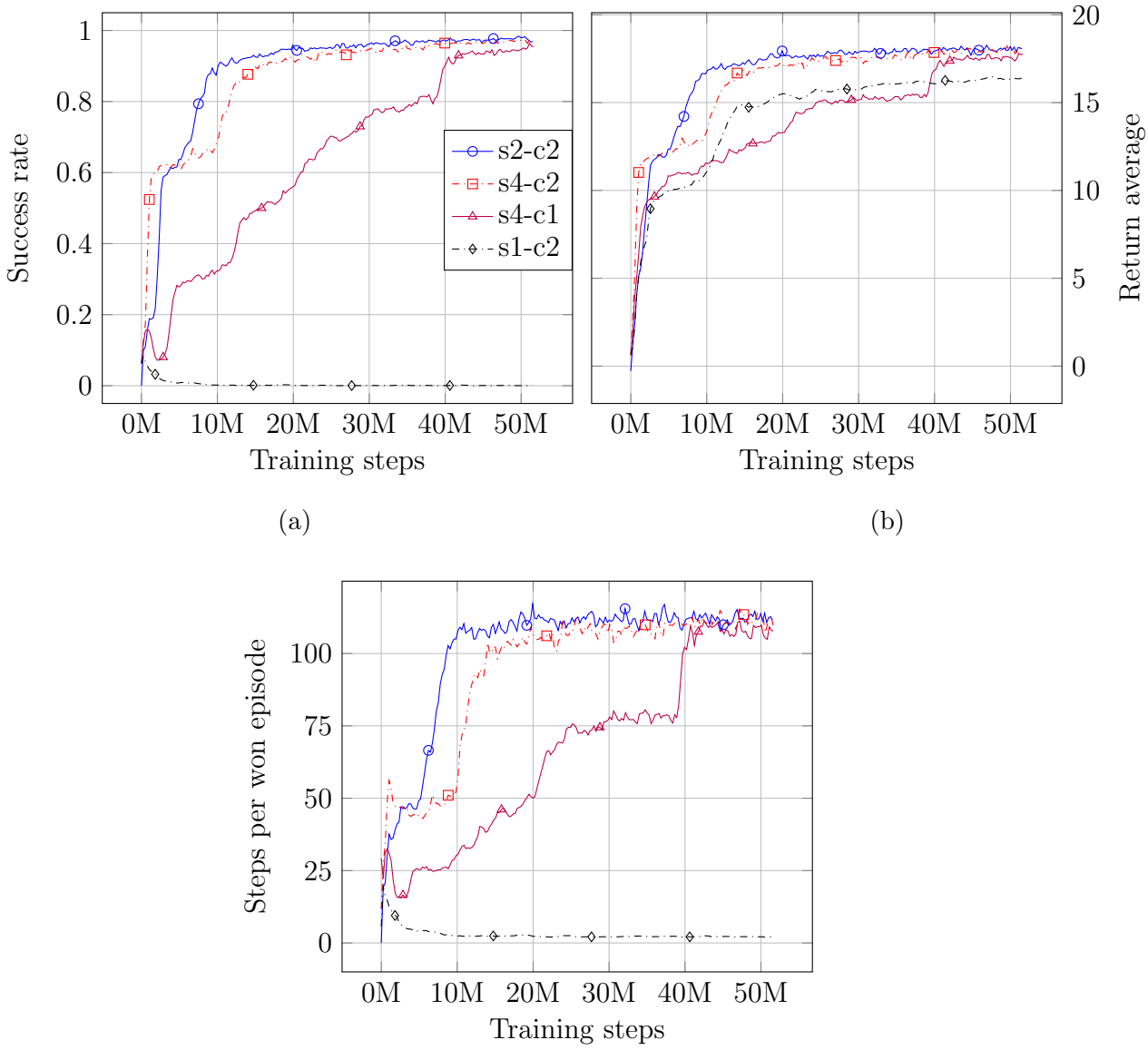


Figure 3.2: Results of partitioned A3C with cropped view

The most successful set of situations proved to be *s2*, showing that the other situations in *s4* only produce noise. Set *s1* on the other hand completely failed at the task; as can be seen in figure 3.2a the configuration *s1-c2* generated a policy that completely ignored the stairs. Generally we noted that all

situational agents not assigned to the "stairs are visible" situation learned to completely avoid the descent action: this is a reasonable outcome, because that action is never useful in those circumstances, however we believe this was exacerbated by the negative reward for standing still. Even being so small, it gives an immediate feedback for the descent action when performed anywhere else than on top of the stairs. This, combined with the easier rewards attainable by stepping on new doors and discovering rooms is the cause of the failure of *s1*. In section 3.3.3 we in fact show that rewarding only stairs descent leads *s1* to good results without changing anything else.

The experiment with 4 situations resulted in the development of the peculiar inclination for the agent of walking alongside walls, that is not observed with 2 situations, possibly exploiting what the neural network assigned to the "next to a wall" situation learned. This is arguably due to the combination of the easier subtask of following walls with respect to walking without anything around and the reinforcement of this behavior whenever a reward is obtained for following it, which frequently happens when stepping on a door for the first time and then proceeding to a new room. Moreover, we noted in all agents the tendency of taking a couple of steps in all doors of small rooms, an artifact possibly due to the reward function.

Finally, state representation *c2* induced faster learning, but not a significantly higher success rate.

Keras implementation issues

In order to increase readability and consistency with previous work we attempted to implement our agents with the Keras [11] library, however we were unable to reach the same scores.

We faced multiple issues during the translation, mainly:

- Keras' high level interface made us unable to abstract from the underlying tensor library (Tensorflow), due to undocumented exceptions raised when using threads;

- The number of training steps performed per hour halved with respect to Tensorflow;

- The learning was completely unsatisfactory using the native RMSProp optimizer implemented in Keras (with the same parameters employed in Tensorflow). The agent learned to prefer a single action over all others, obviously with very poor results;

- Even when making use of Tensorflow's optimizer, the agent only learned to explore the level without descending the stairs. It would get stuck when, in order to visit new rooms, it needed to backtrack through already visited portions of the map and it learned to ignore the stairs almost completely. The differences in the code are only in the thread bootstrapping phase and in the neural network model implementation. If we converted the Tensorflow's learned weights to Keras and let the agent play, it would achieve the same scores. We then turned to compare the single updates computed by Keras and Tensorflow on random data. We noted that they differ by an absolute value of $\sim 10^{-2}$, however this difference would not increase over any number of consecutive updates. Moreover the relative order of the learned action probabilities would remain the same.

Due to time constraints we could not figure out where the problem lies. Since recently Tensorflow's API has become much more readable and Keras-like, and due to the faster experienced training time, we decided to stick with that implementation.

3.3.3 Towers with A3C and ACER

In this section we present our second approach[6] to descending the first level. The main motivation behind this effort was to resolve some of the unsatisfying aspects of the previous solution. In particular:

[6] `https://github.com/rogueinabox/openai_acer`, commit tagged v0.8

1. The programmatic non-learned situations seem to be too much of a facilitation for the agents;

2. The cropped view, which also removes some difficulty from the task, specifically by not requiring to learn any generalization from the position of the rogue, which is always the center of the state representation;

3. The reward signal seems to be too informative and the source of artifacts in the learned policy, that we discussed in the results of section 3.3.2.

Our aim was to work out these issues while still achieving a comparable success rate. The first step was to deal with points (1) and (3). By evaluating A3C with the neural network structure described in figure 3.1, using no situations and only rewarding stairs descent, we obtained a 91% success rate (figure 3.4a).

This is a promising result, that inspired us to proceed to address issue (2) while simultaneously improve the performance of the agent. This is more delicate of a facet: the convolutions in the neural network previously employed do not reduce the dimensionality of the input, they actually increase it due to the number of filters. Since the convolutions are immediately followed by a FC layer, enlarging the input size in order to encompass the entire screen would further increase the already substantial number of parameters, requiring much more training steps, time and memory. To face this problem, we decided to employ the network architecture used in [34, 3, 4], a structure with three towers that we describe later in this section.

The main ingredients of this approach can be described as follows:

1. The A3C and ACER algorithms;

2. A state representation encompassing the *entire map* that is not centered on the rogue;

3. The *three towers* neural network architecture, that is able to process the above mentioned state with a limited number of parameters and training time;

4. A reward function that only encourages descending the stairs.

ACER

We decided to experiment with this algorithm because we expected that given the complete state representation A3C would have required more training steps to achieve good success rates. ACER was designed to be more sample efficient (see section 1.2.8), while still remaining very similar to A3C. We base our implementation on the OpenAI baselines repository [14], that employs a synchronous version of ACER (which is to ACER what A2C is A3C, see section 1.2.7), in order to fully exploit our GPU. In order to test if ACER would actually reduce the required training steps, we performed a couple of experiments:

1. We attempted to reproduce the results we described earlier in this section, i.e. the 91% success rate by employing a cropped view and rewarding only descending the stairs. We were unable to reproduce them, as we show in 3.4a: the ACER agent would reach $\sim 55\%$ success rate much more quickly than A3C, however it would not progress further. In order to verify if this was due to different optimizer hyper-parameters (that we describe later), since we first used the default values proposed in the OpenAI's implementation. We tested both with the same values we used for A3C and also with intermediate values. In the first case, the agent would not learn anything, i.e. the success rate would remain below 10% in the first 20 or so million steps. We did not let it proceed further than that because our main reason to employ ACER is to have faster training. In the second case, we obtained the same training progression described earlier: quickly reaching $\sim 55\%$ success rate then indefinitely oscillating below that. We are baffled by these

results, especially considering those we will describe in the next point. Because of time constraints and limited resources we were unable to further experiment on this, however it remains a point of interest.

2. We then compared A3C and ACER with the complete state representation the three towers network model. ACER learned much faster and reached remarkable results under the hyper-parameters we will describe later in this section, in part derived from the OpenAI's implementation. We show this comparison in 3.4b: A3C learns very slowly, but surely, approaching a 90% success rate at 100 million training steps. ACER on the other hand learns much more quickly achieving, 90% as soon as 19 million steps, then 97% at 36 millions and finally 98% at 76 million steps. We expected a similar outcome in the previous case as well: the issue remains unsolved.

Neural network architecture

The *three towers* model is shown in 3.3 (for ACER). The state representation is processed in three separate and independent branches, each of which we call a *tower*. This architecture was devised and first used in [34, 3], for completeness and since it is an important part of our work, we shall describe it in this work as well. The three towers can be divided in two categories:

Local vision tower It comprises a couple of convolutional layers without any non-linear activation function, then a max-pooling the size of the entire 22×80 input to extract the maximum value for each filter. The intent of this structure is to provide a form of attention that the model will learn where to focus via high convolutional activations. We employ a single tower of this kind.

Global vision tower It is made of an initial max-pooling layer, followed by two convolutional layers, each employing the ReLU activation function, then a final max-pooling layer that extracts the maximum value for each convolutional filter. This kind of tower is meant to capture

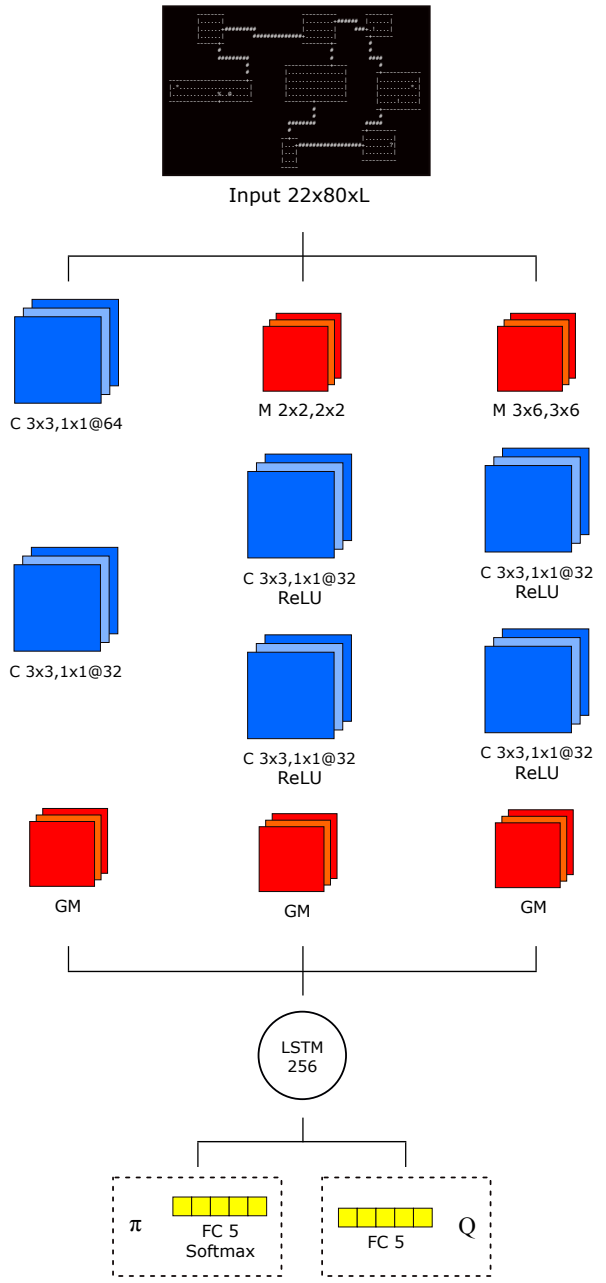


Figure 3.3: Three towers neural network architecture for ACER

high level features of the entire map, e.g. in the case of Rogue the positions of key points such as the doors and the stairs. We make use of two towers of this type, with a different initial max-pooling window.

The output of the towers is merged and then fed to an LSTM. In the A3C version, as in partitioned A3C, the LSTM is also provided with the concatenation of a *one-hot* representation of the previous action taken and the reward obtained. In ACER however we do not provide this input. The network then branches again into a policy and, in the case of A3C, a value layer, while in the case of ACER an action-value layer. The total number of parameters of this model is $\sim$ 406 thousands, which is lower than what we needed in partitioned A3C by a factor of ~ 7.

State representation

The state representation we used is slightly more elaborated than the one described in the previous section. As mentioned earlier, it encompasses the entire map and is not centered on the rogue, requiring some abstraction on its position to be learned. Just like in partitioned A3C, the state is implemented as matrix, this time with shape 22×80 (i.e. the dimensions of the screen in ASCII characters, minus the top and bottom rows, that provide text messages with information that we don't need yet). We experimented with several variations, each composed of a different number of layers. Our intent was to provide the model with a representation of the Rogue screen that is as close as possible to what a human see. Under this perspective, the first version we employed is composed of two channels, that we call *f2*, with the following values:

First channel

1 for walls

10 for doors and corridors

0 everywhere else

Second channel

10 for stairs

1 for the rogue position, that would overwrite the stairs if the rogue is exactly on top of them (just like in the game screen)

0 everywhere else

Since the performance we obtained with *f2* was very disappointing (please see the results at the end of this section for more details), we elaborated two similar representations, *f4* and *f5*, the first filled as follows:

First channel

1 for walls

0 everywhere else

Second channel

1 for doors and corridors

0 everywhere else

Third channel

1 for floor tiles

0 everywhere else

Fourth channel

10 for stairs

1 for the rogue position (overwriting the stairs if they are below the rogue)

0 everywhere else

f5 is analogous, however the fourth layer is dedicated to the stairs and a fifth is used for the rogue position.

Previous work [34, 3, 4] used a quite similar level of representation, with a slightly higher level of abstraction. The authors used a channel marking tiles

the rogue can walk on, another highlighting the position of doors and two layers equivalent to ours, marking the position of the rogue and of the stairs. In addition to this, one of the handcrafted forms of memory we mentioned in section 3.3.2 was supplied as a supplementary channel.

Due to time and resources constraints, we were unable to thoroughly test all the state representations presented, nor to search for the most compact one that would achieve high performance. We anticipate that *f5* lead to the best results. This is not completely satisfying, because all those different layers are not the way the state is presented by the game, especially not the channel dedicated to the stairs, which makes them visible even if the rogue is on top of them. The visibility issue anyhow is also present in the cropped view of partitioned A3C, so in this sense we are not providing the model a more informative input. Even if *f5* might facilitate learning, we deem this to be a minor simplification with respect to the cropped view and the non-learned situations of partitioned A3C. We also bring forward another consideration: in all of the state representations we built every "pixel", i.e. matrix cell, is important because it stands for an entire, sometimes independent, entity: e.g. a door, the rogue or a floor tile that can be walked on. The same is not true for Atari 2600 games for instance: each pixel is just a part of a larger entity, such as a space ship or a ball, that are made of many adjacent pixels. We are not certain how this difference could affect learning, given a single layered state representation. Possibly this should be taken into account when designing the neural network: under this perspective the fact that we need 5 layers to achieve good results may be the indication that the three towers model is not adequate, either in its entirety or in some parts of it, e.g. the number of convolutional filters or kernel sizes.

Reward signal

In these experiments we used the most simple reward function possible: a positive reward (with value 10) for descending the stairs and zero for everything else. This choice follows mainly two considerations:

1. We wanted to assess how well the agent could do if awarded only whenever it did what we wanted it to learn: finding and descending the stairs;

2. We did not want to introduce any bias in the agent behavior.

For the task at hand this reward proved to be adequate, as can be seen from the results discussed later in this section.

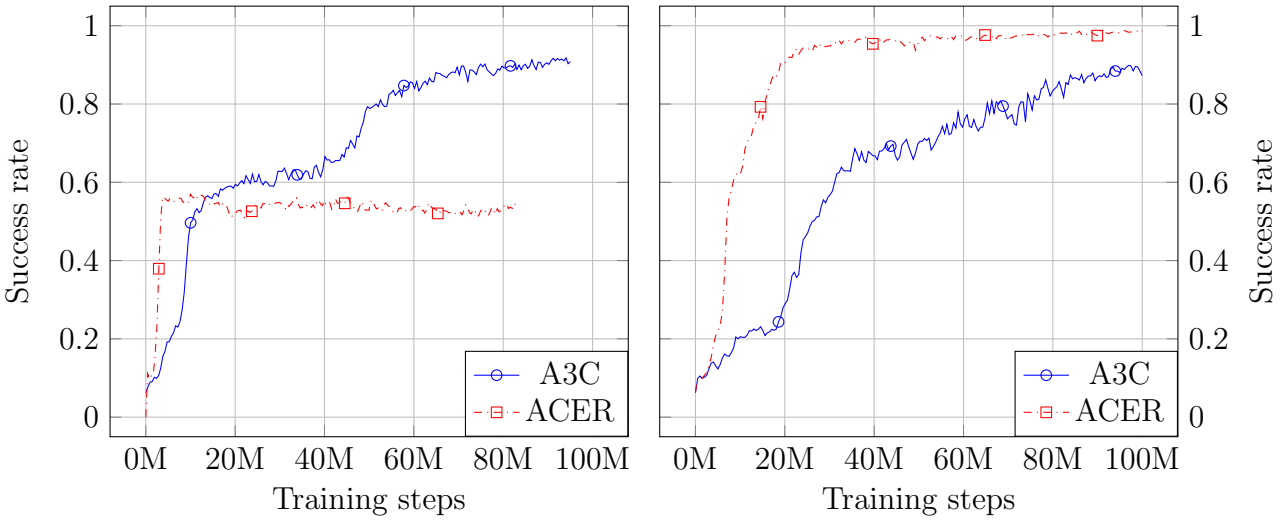
Hyper-Parameters

The hyper-parameters selection is totally unvaried for the A3C implementation, while there are some differences for ACER especially in the optimizer parameters, that correspond to the defaults from the OpenAI's implementation, due to the poor results we discussed earlier in the ACER subsection. Episode length and termination condition are unvaried from partitioned A3C, section 3.3.2. and so is the learning rate linear annealing. The values are summarized in table 3.3.

Parameter	Value
parallel actors	16
episode max length	500
entropy β	0.01
discount factor γ	0.99
batch size t_{max}	60
ACER memory buffer size	50000
ACER replay ratio r	4
ACER importance weight clipping c	10
ACER average policy α	0.99
ACER KL divergence constraint δ	1
initial learning rate η	0.0007
rms decay	0.99
rms momentum	0
rms epsilon	$1e^{-5}$
rms clip norm	10

Table 3.3: Hyper-parameters for ACER

Results



(a) Cropped view (state *c2*)

(b) Towers and complete map (state *f5*)

Figure 3.4: Results of A3C and ACER without situations and rewarding only stairs descent

We show our results in the following plots: in figure 3.4a A3C vs ACER with cropped view, in figure 3.4b A3C vs ACER with *f5* and the three towers network. In table 3.4 the final scores are summarized.

Agent	A3C-*c2*	ACER-*c2*	A3C-*f5*	ACER-*f5*
Success rate	90.84%	55.25%	87.25%	98.69%
Avg number of seen tiles	356	284	354	385
Avg number of steps to succeed	115	99	113	119

Table 3.4: Final results of A3C and ACER without situations and rewarding only stairs descent

The first figure presents the baffling results we discussed in the ACER section previously. All our attempts at varying hyper-parameters toward the values used for A3C resulted in the same ACER learning progression, or

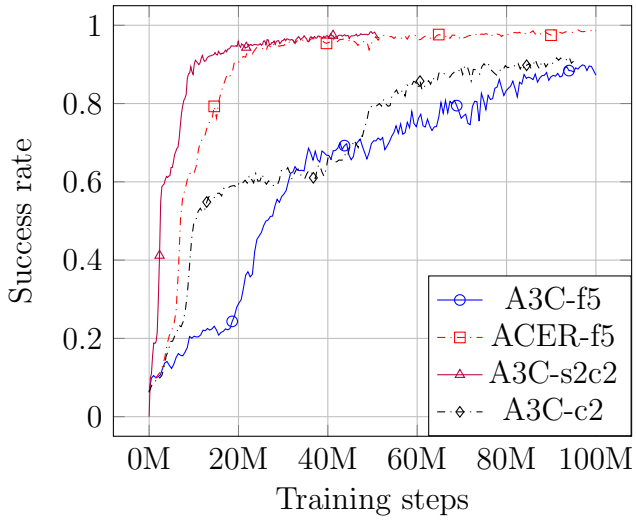


Figure 3.5: Comparison between our previous best results; here *-f5 means towers with complete map representation *f5*, *-c2 stands for cropped view state *c2* and *-s2c2 partitioned with situations set *s2*, cropped state *c2* and rewarded exploration

no progression at all (always under 10%) with the same A3C parameters. We could not figure out the reason behind this, especially considering the other results we obtained. This may be an interesting subject of research, if anything to gain more insight on the algorithms.

The second figure displays the learning speed of ACER relative to A3C: if the latter approaches 90% at 100 million training steps, the former has already reached that mark by 19 million steps, then proceeds to 97% by 36 millions and, finally, to 98% 76 million steps. With these results we match what we achieved with partitioned A3C and cropped view, foregoing all the facilitations we were unhappy about that we discussed at the beginning of this section. The price we paid is a slightly richer state representation, which doesn't provide more information than the cropped view (except everything outside of it), but arguably makes it easier to process by a neural network. As we mentioned, due to time and resource constraints we were unable to thoroughly test many different state representations. We experimented A3C

with *f2*, but found it could not achieve more than 12% within 30 million steps. Coupled with *f4* the algorithm performed slightly better and we suspect that it could eventually achieve interesting results, however we decided to interrupt it and focus on *f5*. This remains an interesting area to improve upon: as we mentioned in the state representation subsection earlier, these difficulties may be due to an inadequate neural network structure that is unable to extract relevant features, maybe because of an insufficient number of convolutional filters or layers.

The learned policy is able to backtrack its steps in many cases, just like the situational agents of partitioned A3C. The bias in the agent's behavior we were concerned about in the previous section, i.e. taking a step in all corridors of small rooms, is no more. It is however replaced by another, more understandable, bias: the tendency to attempt the descent action very often, even when they are not visible. Being the only source of reward, this was probably a predictable outcome.

3.4 Descending until the tenth level

The good results we achieved pushed us to test how far our agent could crawl in the dungeons. We thus set up a new experiment in which we allowed the agent to descend as far as the tenth level.

Here another important simplification that we mentioned in section 3.1 comes into play: from the second level onwards some areas of each level may be hidden in such a way that a stochastic number of actions is required to uncover them. The sections that could be hidden might also enclose the stairs, requiring the player to learn to habitually search (with a dedicated action) what appears to be a dead-end to a corridor or a room without visible doors. We deemed this to be too hard to learn for our model and thereby disabled the hidden areas.

An interesting aspect that we will evaluate is how well our agent deal with *dark rooms* and *labyrinths* (figure 3.6). The former are rooms where

only the cells *immediately surrounding* the rogue are displayed on the screen. The walls of such rooms are not shown when it is first entered, however they remain visible once discovered for the first time. The same holds for stairs and items, but not for floor tiles or monsters. The labyrinths are exactly what the name suggests: dense webs of corridors with numerous branches and dead-ends.

3.4.1 Evaluation criteria

The metrics we use for this task are the natural extension to those we described in section 3.3.1. We average the number of levels descended per episode, how many times each one is climbed down and the number of steps it took over the most recent 200 episodes played at a given training step. When the tenth level is reached, the episode is won and we reset the game.

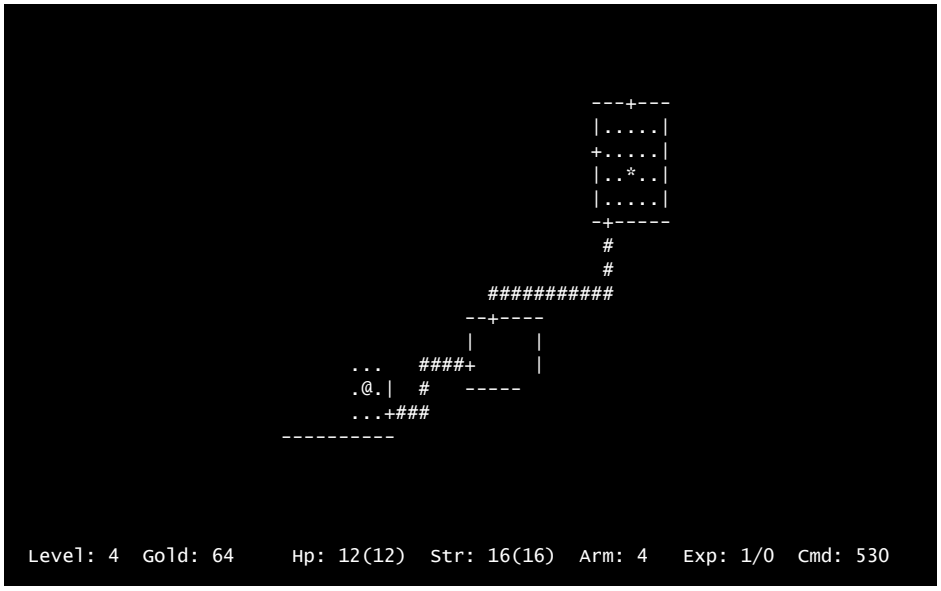
3.4.2 Towers with ACER

We employ the same configuration that lead to very good results in section 3.3.3, i.e.:

1. The ACER algorithm;
2. The *three towers* neural network model;
3. The *f5* state representation,
4. The reward function that directly encourages only descending the stairs;
5. The hyper-parameters of the previous section, except the maximum number of steps per episode — that we set to 1200 — since on average it took ~ 120 steps to descend the first level and $120 \cdot 10 = 1200$.

State representation

We noted that due to implementation details, the *f5* state representation would have retained displaying those tiles in dark rooms that are hidden



(a)

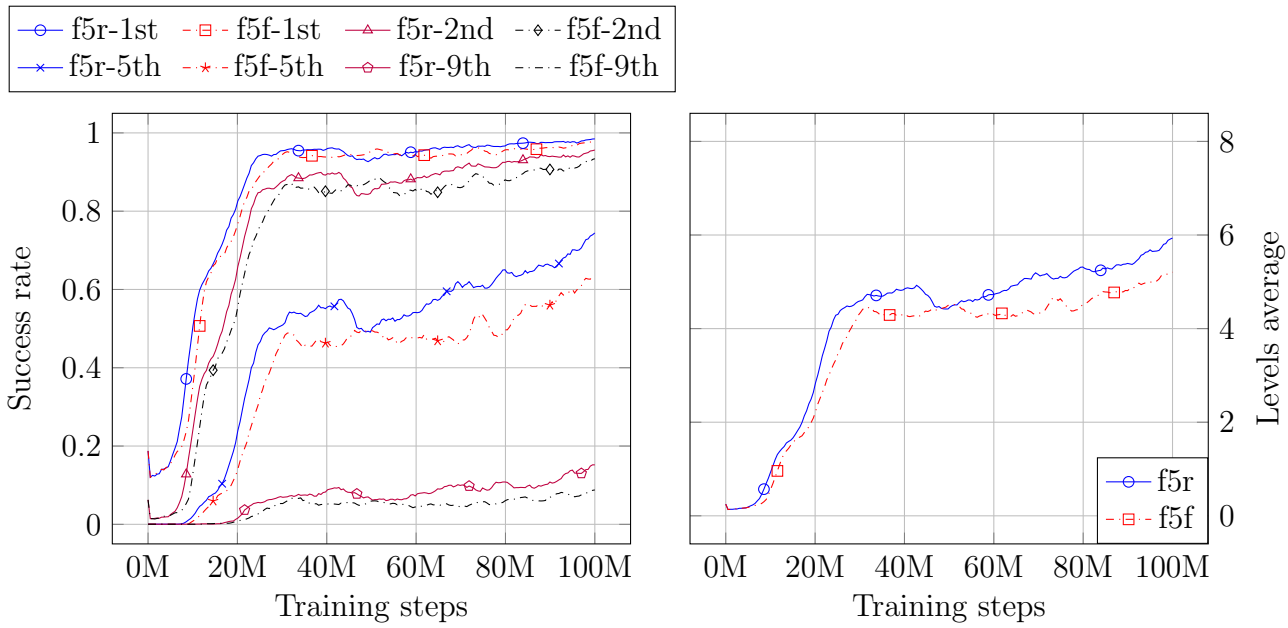
(b)

Figure 3.6: (a) Dark rooms (b) A labyrinth (surrounded by dark rooms)

when the rogue moves away. We thus decided to test to what degree this would have influenced the agent's behavior, with respect to a representation that instead hide such tiles like in the game screen. The first will be denoted

as *f5r*, where *r* stands for *remember*, and *f5f*, where the second *f* stands for *forget*.

Results



(a) *-*N*th stands for the average number of times the *N*th level is descended

(b) Average number of levels descended per episode

Figure 3.7: Results of ACER with Towers descending until the 10th level

The results are shown in figure 3.7 and table 3.5. The scores are not what could be expected: given the 98% success rate on the first level, one could estimate that reaching the tenth, thereby descending 9 levels, would happen $0.98^9 \sim 83\%$ of the times. Our agent however really struggled to get there, finding its way to the tenth level on average in only 15 games out of 100 when aided by *f5r* and almost 9 when using *f5f*. We speculate that the substantial difference from the expectation of 83% is due to dark rooms and labyrinths. In order to look at the issue more clearly, we tested how many times our agent would fail due to those reasons. That is, we checked the number of levels where it was not able to find and descend the stairs and

Success rate									
Level	1st	2nd	3rd	4th	5th	6th	7th	8th	9th
f5r	98.47%	95.62%	91.37%	84.41%	74.44%	61.12%	44.75%	28.25%	15.19%
f5f	97.72%	93.41%	86.19%	76.56%	63.62%	47.03%	29.91%	17.34%	8.78%
random	15.50%	3.00%	–	–	–	–	–	–	–

Avg number of steps to succeed									
Level	1st	2nd	3rd	4th	5th	6th	7th	8th	9th
f5r	131	263	405	546	667	771	856	923	958
f5f	140	287	434	581	705	805	873	922	954
random	350	668	–	–	–	–	–	–	–

Average levels descended	
State	Value
f5r	5.9
f5f	5.2
random	1.9

Table 3.5: Final results of ACER descending until the tenth level

how many steps it took there. We show these statistics in figure 3.8 and table 3.6. The numbers tell us that labyrinths are quite hard and only 48% of them are overcome, while levels with dark rooms are easier and successfully navigated $\sim$ 85% of the times. Even though the aren't enough labyrinths levels to justify the poor performance on reaching the tenth level, we can see that the number of steps taken there and where dark rooms are present is higher than the average number required when they are absent. They are also much higher than for the first level, where 120 were enough on average. Given these numbers, we expected that the agent could do better if given more than 1200 actions and in table 3.7 we show statistics for 2000, 3000 and even 10000 maximum steps. The latter is unreasonably high, it only serves as a sort of asymptotic analysis of the behaviors and we won't consider it in the following discussion. From the table its clear that simply incrementing their number, without any re-training, greatly and positively impacts on the agent performance. *f5r* with 2000 maximum steps outperforms the naive estimate

Level	1st	2nd	3rd	4th	5th	6th	7th	8th	9th
DRs	0%	20.72%	37.34%	48.79%	50.64%	60.95%	62.42%	63.78%	62.98%
Ls	0%	2.26%	2.62%	2.42%	3.17%	2.72%	5.07%	3.11%	2.42%
DR+Ls	0%	0.62%	1.67%	3.96%	5.86%	4.90%	9.15%	9.78%	9.00%

(a) Level by level statistics; we denote dark rooms with DRs, labyrinths with Ls and levels containing both of them with DR+Ls

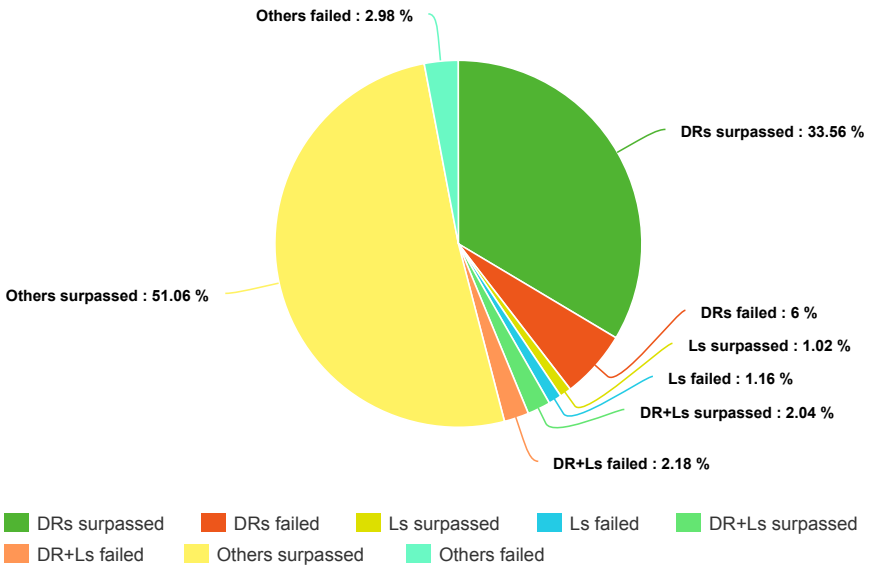
Statistic	Value
Avg steps per level	170
Avg steps per level with dark rooms	234 (+38%)
Avg steps per level with labyrinths	447 (+163%)
Avg steps per level with both	428 (+152%)

(b) Steps statistics

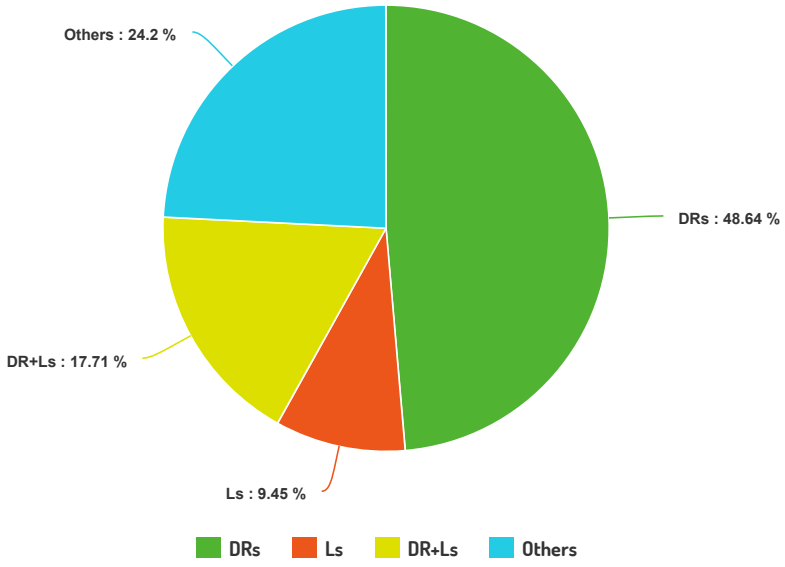
Table 3.6: Dark rooms and labyrinths statistics produced by *f5r* in 1000 episodes

of a 0.98^i success rate on the i-th level until the sixth, where it starts to deteriorate. The difference from 2000 to 3000 steps is especially significant for *f5f*, while for the other state representation was beneficial only for the seventh level onwards.

In the various plots and tables we can see the difference between state representation *f5r* and *f5f*. We did not expect it to be as significant as to determine such a gap in the success rate, although the discrepancy in the average number of levels descended per episode is not as pronounced: 0.7 with both 1200 and 3000 maximum steps and 1.5 with 2000. In any case this proves that dark rooms provide yet another difficult and interesting challenge in Rogue. It is not clear if *f5r* only speeds up learning or if it allows our agent to achieve results it could not attain otherwise. It is plausible that given more training time our model could have reached better results, since all scores from the second level onwards are in an ascending trend. Perhaps training exclusively on each of these levels can shine light on this matter.



(a) Levels statistics



(b) Failures statistics

Figure 3.8: Dark rooms and labyrinths statistics produced by *f5r* in 1000 episodes; we count a failure in dark rooms or labyrinths only if the agent took at least 60 steps in that level; we denote dark rooms with DRs, labyrinths with Ls and levels containing both of them with DR+Ls.

Success rate with 2000 maximum steps									
Level	1st	2nd	3rd	4th	5th	6th	7th	8th	9th
f5r	99.00%	99.00%	97.50%	93.00%	91.00%	83.50%	76.00%	68.50%	54.50%
f5f	98.00%	94.00%	88.00%	78.00%	67.50%	58.50%	46.50%	36.00%	26.50%
random	20.50%	1.50%	–	–	–	–	–	–	–
Success rate with 3000 maximum steps									
Level	1st	2nd	3rd	4th	5th	6th	7th	8th	9th
f5r	99.50%	98.50%	96.50%	94.00%	90.00%	83.00%	78.50%	73.00%	64.50%
f5f	99.00%	96.50%	93.50%	88.00%	82.00%	75.50%	67.00%	59.00%	50.00%
random	19.50%	1.50%	0.50%	–	–	–	–	–	–
Success rate with 10000 maximum steps									
Level	1st	2nd	3rd	4th	5th	6th	7th	8th	9th
f5r	99.00%	98.50%	97.00%	92.50%	90.00%	87.00%	83.50%	75.50%	71.00%
f5f	99.00%	97.00%	91.00%	86.00%	81.50%	75.50%	64.50%	56.00%	49.00%
random	37.50%	13.00%	3.50%	1.50%	1.00%	–	–	–	–

Average levels descended		
Max steps	State	Value
2000	*f5r*	7.6
	f5f	5.9
	random	2.2
3000	*f5r*	7.8
	f5f	7.1
	random	2.2
10000	*f5r*	7.9
	f5f	7.0
	random	5.7

Table 3.7: Results of ACER descending until the tenth level with more maximum steps per episode, sampled over 1000 episodes

3.5 Recovering the amulet from early levels

Our last efforts had us test if our agent could find the amulet and then win the game if the jewel was placed in an early level, specifically the first and second. After recovering the relic, the agent has to find and ascend the stairs on each level traversed, including the first; if it is able to do this, the

game is won.

We argue (and later show) that this task would become *very hard* should the amulet be placed on lower levels and the agent only rewarded for recovering it and winning, like we do. In fact, it would require to learn complex and long sequences of actions without any reinforcement, i.e. finding and descending the stairs of all levels above the amulet. This becomes evident in section 3.5.3.

For these new endeavors we persist with our ACER configuration, slightly tweaking it by:

1. Adding a layer in the *f5* state representation, showing the position of the amulet with a 1 and 0 in all other positions, that we call *f6*;

2. Positively rewarding stepping on the amulet and winning the game; anything else is rewarded with 0.

Here a simplification that was irrelevant in the previous efforts should have come into significant play: when the rogue reaches the amulet level, the descent action is turned into ascent until the end of the episode. Rogue allows descending further than the amulet level, however climbing up is permitted only when possessing the amulet; thereby, it is possible to permanently prevent any chance of winning a game. This solution that we introduced is probably unneeded if the amulet is placed in the first level, as the agent would, most likely, eventually learn to ignore the descent action. When however the jewel is placed on lower levels it should be of crucial aid, preventing difficulties that we are not ready to deal with yet. We will actually show in the experiment of section 3.5.3 that this simplification does not even have the chance to play any role.

3.5.1 Evaluation criteria

We expand on the metrics outlined in section 3.3.1 simply by averaging the number of times the amulet is recovered, in addition to all previous statistics.

3.5.2 First level

The main point of interest of this experiment is especially to assess how much harder this task is with respect to simply finding and descending the stairs and to evaluate the navigation skills of our agent once it has recovered the amulet and already discovered the stairs location. With respect to these issues, we obtained interesting and partially unexpected results.

Here we set the maximum amount of steps per episode back to 500.

Results

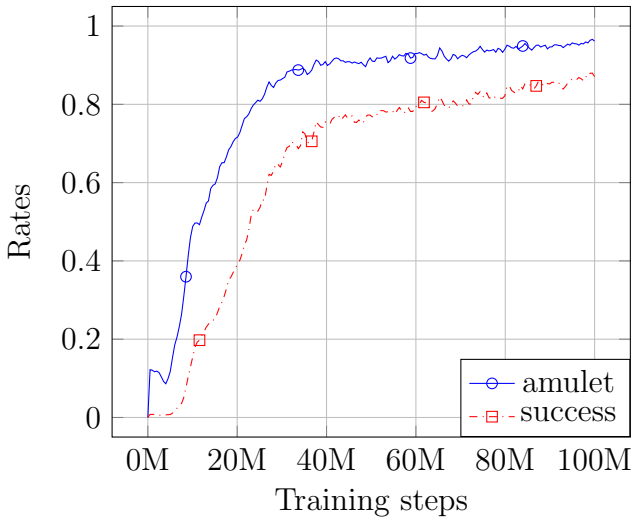


Figure 3.9: Results of ACER recovering the amulet from the first level

We show the results of this experiment in figure 3.9 and table 3.8. As expected this task is harder than simply descending the first level, even more so than what we thought. Intuitively one could estimate that first discovering the amulet and then the stairs would be close to locating the latter two consecutive times. If this was true, given a 98% success rate for finding the stairs of the first level, the success rate for this new task would have been similar to $0.98^2 \sim 96\%$. Our experiment tells a different story however: because our agent is able to accomplish its goal only $\sim$ 87% of

Statistic	ACER	random
Success rate	87.06%	2.00%
Amulet recovered rate	96.16%	14.50%
Pathfinding to stairs rate	90.54%	13.79%
Pathfinding to undiscovered stairs rate	94.75%	6.21%
Pathfinding to found stairs rate	86.95%	44.00%
Stairs found before amulet rate	52.10%	17.24%
Avg number of steps to succeed	205	274
Avg number of seen tiles	486	106

Table 3.8: Final results of ACER recovering the amulet from the first level

the times, though even if the learning curves were still in a slowly ascending trend when we interrupted the training. This discrepancy is probably due to a new skill required for this endeavor: pathfinding to a known location. We see from table 3.8 that the agent has discovered the stairs position before the amulet in $\sim 52\%$ of cases. In this kind of scenario it is able to succeed $\sim 87\%$ of the times after recovering the amulet, while in the opposite case — when it has yet to uncover the stairs — it is able to win almost 95% of the games. Regarding its behavior, sometimes it shows some difficulty in stepping toward the right direction and on some occasions it is not able to reach the stairs in time. For instance, when the path to the stairs is significantly longer than as the crow flies (like in figure 3.10), requiring first to visually get further from them, the agent always attempts many times to cross the wall before actually going for the correct path. This imperfect pathfinding is an indication of where to focus the attention in future work and is a symptom of something defective, either in the algorithms or in the neural network architecture.

3.5.3 Second level

This experiment is intended to show the limit of our learning configuration and set the bar for future work. We expected terrible results and were not surprised in this regard.

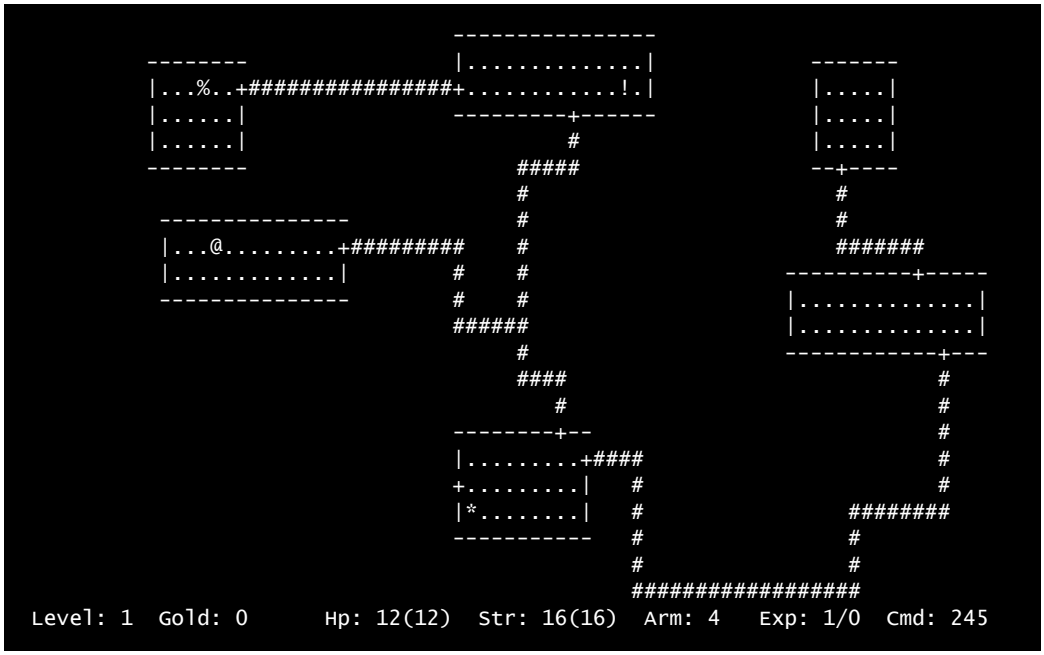


Figure 3.10: ACER agent attempting to cross walls to reach the stairs after having recovered the amulet

Here we set the maximum amount of steps per episode to 1000, that are more than enough for the task.

Results

Statistic	ACER	random
Success rate	0.06%	0%
Amulet recovered rate	2.06%	1.00%
First level descent rate	13.34%	12.50%
Second level ascent rate	0.06%	0%
Avg number of steps to succeed	59	–
Avg number of seen tiles	143	137

Table 3.9: Final results of ACER recovering the amulet from the second level

The results are shown in figure 3.11 and table 3.9. It appears that the agent is not able to learn at all, even though it is sometimes able to recover

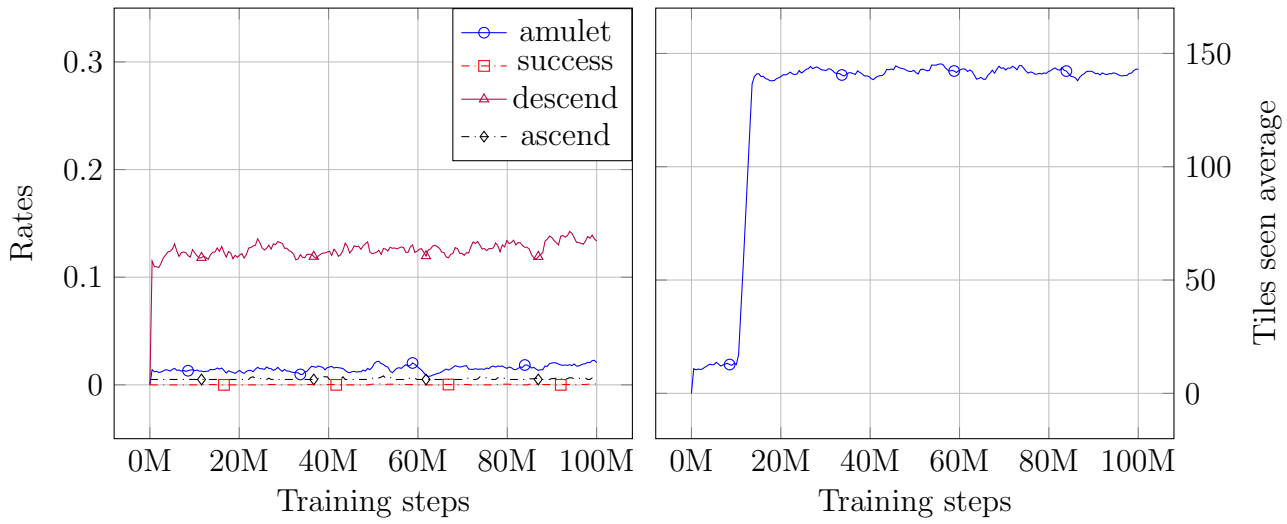


Figure 3.11: Results of ACER recovering the amulet from the second level

the amulet. This most likely happens when the rogue starts the game in the room holding the stairs of the first level, and then the amulet is found in the room the agent finds itself after descending the stairs. This is arguably not enough to learn to effectively navigate and search the rogue's surrounding. The problem is that the probability of quasi randomly recovering the amulet from the second level is not enough to teach the agent anything useful, in fact it keeps moving around haphazardly even after 100 million training steps: not surprisingly it does not perform significantly better than a completely random agent. This issue is mostly due to the extremely sparse reward, but may also be affected by the learning algorithm and the n-step length of 60, which may be too low in this scenario.

This experiment was meant to show that dealing with this task requires either further tuning of ACER, perhaps by prioritizing experience replay as in [35], or some form of *intrinsic reward* to compensate the discouraging sparsity and guide the exploration of rooms, or both. Regarding intrinsic reward [5] seems to be particularly promising, as the authors achieved unprecedented results in Montezuma's Revenge, an Atari 2600 game well known for its complexity and reward sparsity. Future work will have to address this issue

if keeping the reward function as simple and reasonable as possible remains a priority.

Conclusions

In this work we resolved the problematics encountered by Asperti et al. [34, 3, 4] and established new boundaries to what has been shown to be possible in Rogue via RL.

In section 3.3 we immediately addressed the arisen issues — namely the low descent rate and the troubles backtracking in cul-de-sac scenarios — via partitioned A3C and cropped view, then dealt with its unsatisfying facets by employing ACER and the three towers neural network. In both cases we achieved a 98% performance, while Asperti et al. obtained 23% and a completely random agent 7%.

In section 3.4 we tested how far our agent could go, setting the objective to reach the tenth level. The lower stages presented new challenges, i.e. dark rooms and labyrinths, that made the task harder than we expected. Our best results, involving a state representation that retain hidden tiles in dark rooms and 3000 maximum steps per episode, were reaching the tenth level 64.5% of the times and an average of 7.8 levels descended per episode.

Finally in section 3.5 we investigated recovering the amulet from the first and second level, awarding only victory and looting the jewel. We obtained a good 87% success rate in the former case, albeit slightly worse than expected due to pathfinding issues that manifested for the first time in this scenario. In the second case we had the terrible results we expected: our agent could not perform significantly better than a completely random policy. This last experiment mainly serves to prove that the task is indeed as hard as we thought and sets the bar for future efforts on this front.

Future work

There is certainly a lot more work yet for future endeavors. Even if all the issues faced in this work were resolved, the complexity of the game will surely challenge researchers for many years to come. Here we summarize some points that can be addressed to directly improve on this work, some natural extensions and other interesting directions:

- Test the complete state representations of section 3.3.3 more thoroughly;
- Experiment different neural network architectures, which may produce more capable agents with respect to pathfinding and traversing dark rooms and labyrinths; e.g. have convolutions decrease the input size with larger strides instead of or in addition to max-pooling;
- Experiment with some form of curriculum training [7]: e.g. separately training on each level may ascertain whether *f5r* only speeds up learning with respect to *f5f* or if it actually allows to reach otherwise unattainable results;
- Experiment with monsters, that will surely introduce further complications to the tasks; arguably, dealing with them may not be very significant unless the agent is also made able to gather and use weapons, otherwise the best solutions might just be to run away; this however requires interacting with the inventory, which is definitely one of the most complicated and difficult elements of the game to learn by RL;
- If reducing the complexity and parameters number of the neural network model becomes a priority, [12] proposes to use a method based on Vector Quantization and Sparse Coding to encode state representation and a very small network that focuses exclusively on approximating the policy rather than also extracting features.

Appendix A

ACER code

In this section we present some code snippets from our ACER implementation, derived from OpenAI's baselines [14].

A.1 Main loop

Here we show the main code controlling the program execution flow.

```
def learn(policy, env, flags):

    nenvs = env.num_envs
    ob_space = env.observation_space
    ac_space = env.action_space
    model = Model(policy=policy, ob_space=ob_space,
                  ac_space=ac_space, nenvs=nenvs,
                  num_procs=nenvs, flags=flags)

    runner = Runner(env=env, model=model,
                    nsteps=flags.nsteps, nstack=flags.nstack)
    if flags.replay_ratio > 0:
        buffer = Buffer(env=env, nsteps=flags.nsteps,
                        nstack=flags.nstack, size=flags.buffer_size)
    else:
        buffer = None
    nbatch = nenvs*flags.nsteps
    acer = Acer(runner, model, buffer, flags.log_interval, flags.stats_interval)

    ...  # further bootstrapping

    acer.tstart = time.time()
    for acer.steps in range(start_steps, flags.total_timesteps, nbatch):

        # on policy training
        acer.call(on_policy=True)

        # off policy training
        if flags.replay_ratio > 0 and buffer.has_atleast(flags.replay_start):
```

```
30              n = np.random.poisson(flags.replay_ratio)
31              for _ in range(n):
32                  acer.call(on_policy=False)  # no simulation steps in this
33
34          ...  # periodic save logic
```

A.2 Environment interaction and training

In the following snippets we show the interaction with environment. First we display a higher level n-step interaction followed by backpropagation.

```
1   class Acer():
2
3       ...
4
5       def call(self, on_policy):
6           runner, model, buffer, steps = self.runner, self.model, self.buffer, self.steps
7           if on_policy:
8               enc_obs, obs, actions, rewards, mus, dones, masks = runner.run()
9               self.episode_stats.feed(rewards, dones)
10              if buffer is not None:
11                  buffer.put(enc_obs, actions, rewards, mus, dones, masks)
12          else:
13              # get obs, actions, rewards, mus, dones from buffer.
14              obs, actions, rewards, mus, dones, masks = buffer.get()
15
16          ...  # reshaping
17
18          # here model.inital_state equals to:
19          # np.zeros((nenv, lstm_units*2))
20          names_ops, values_ops = model.train(obs, actions, rewards, dones,
21                                              mus, model.initial_state, masks, steps)
22
23          ...  # stats
```

Second we show the step-by-step environment interaction code, where we take care to preserve the *temporal sequence* of steps in order to correctly perform backpropagation later.

```
1   class Runner(object):
2
3       ...
4
5       def run(self):
6           # self.obs refers to the last environment observations
7           # stored in earlier calls
8           enc_obs = np.split(self.obs, self.nstack, axis=3)  # list of obs steps
9           mb_obs, mb_actions, mb_mus, mb_dones, mb_rewards = [], [], [], [], []
10          for _ in range(self.nsteps):
11              actions, mus, states = self.model.step(self.obs, state=self.states,
12                                                     mask=self.dones)
13              mb_obs.append(np.copy(self.obs))
14              mb_actions.append(actions)
15              mb_mus.append(mus)
16              mb_dones.append(self.dones)
17              obs, rewards, dones, _ = self.env.step(actions)
```

```
18              # states information for statefull models like LSTM
19              self.states = states
20              self.dones = dones
21              self.update_obs(obs, dones)
22              mb_rewards.append(rewards)
23              enc_obs.append(obs)
24          mb_obs.append(np.copy(self.obs))
25          mb_dones.append(self.dones)
26
27          enc_obs = np.asarray(enc_obs, dtype=np.uint8).swapaxes(1, 0)
28          mb_obs = np.asarray(mb_obs, dtype=np.uint8).swapaxes(1, 0)
29          mb_actions = np.asarray(mb_actions, dtype=np.int32).swapaxes(1, 0)
30          mb_rewards = np.asarray(mb_rewards, dtype=np.float32).swapaxes(1, 0)
31          mb_mus = np.asarray(mb_mus, dtype=np.float32).swapaxes(1, 0)
32
33          mb_dones = np.asarray(mb_dones, dtype=np.bool).swapaxes(1, 0)
34
35          # Used for statefull models like LSTM's to mask state when done
36          mb_masks = mb_dones
37          # Used for calculating returns. The dones array is now aligned with rewards
38          mb_dones = mb_dones[:, 1:]
39
40          # shapes are now [nenv, nsteps, []]
41
42          return enc_obs, mb_obs, mb_actions, mb_rewards, mb_mus, mb_dones, mb_masks
43
44      def update_obs(self, obs, dones=None):
45          if dones is not None:
46              self.obs *= (1 - dones.astype(np.uint8))[:, None, None, None]
47          self.obs = np.roll(self.obs, shift=-self.nc, axis=3)
48          self.obs[:, :, :, -self.nc:] = obs[:, :, :, :]
```

Appendix B

Three towers analysis

Here we discuss some interesting aspects of the main neural network architecture we employed: the three towers model in figure 3.3. In particular, we were curious of where the network focuses its attention, i.e. the location on the map where the max-pooling layers extract the maximum values from the last convolution of each tower. It should be noted however that when the maximum value is extracted all information on its position on the map is lost, unless the rogue is inside the receptive field centered on it. In this case a neuron could specialize in detecting the player character in specific coordinates of the field and some other feature. In the opposite circumstance the agent may be aware of the *presence* of significant entities, but not of their *precise location*. We call the towers, from left to right of the figure, global vision tower (GVT), local vision tower 1 (LVT1) and local vision tower 2 (LVT2).

We found that GVT generally produces the highest outputs very close to the location of the rogue and of the stairs when they are visible, and often on interesting (sometimes distant) spots such as unexplored doors. The values close to the rogue usually give an idea of where the agent will step next. For unknown reasons some channels also seem to output high values near the corners and sometimes in a vertical line on a side of the map.

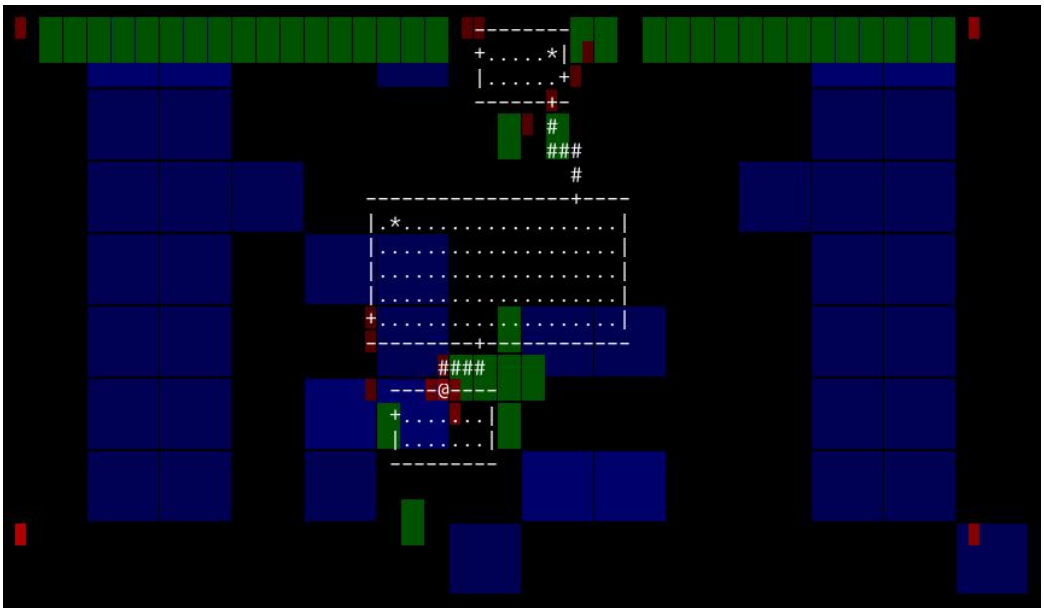
On the other hand, we realized that many of the LVTs channels produce

entirely single-valued output, e.g. all zeros. This phenomenon involves a number of channels inversely proportional to the amount of map explored and we are unsure of its significance. In figure B.1, to avoid noise, we have decided to show only the locations extracted from non single-valued channels. Of these, some had more than one position with the highest value.

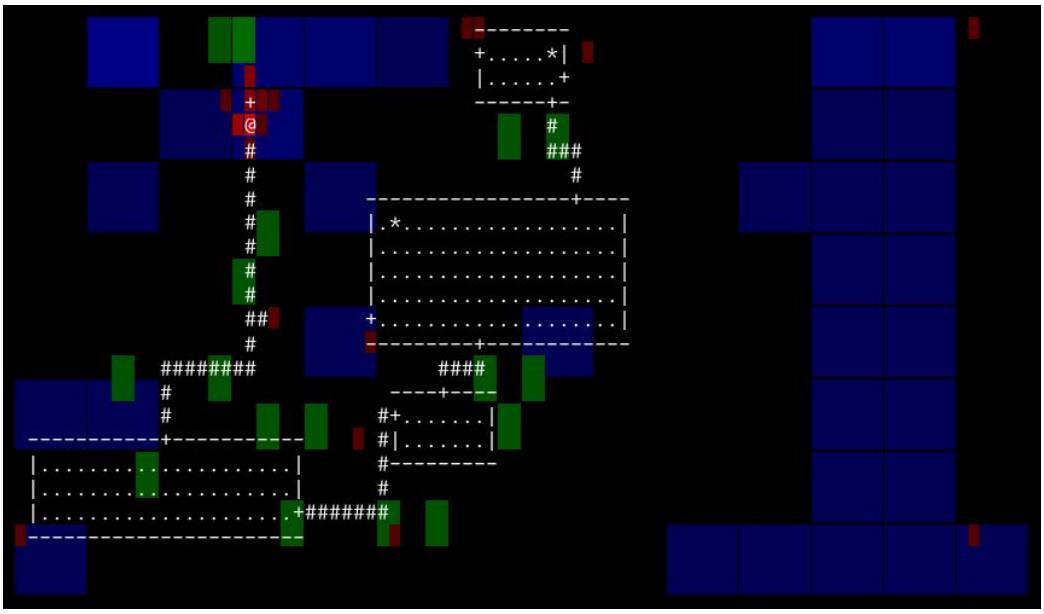
The areas marked by LVT1 are the most difficult to explain. They are generally locations where the agent has previously been, such as corridor corners that have just been turned, or uninteresting spots like room corners without doors and points close to them in unreachable sections of the map. This interpretation is not entirely consistent though, since LVT1 also sometimes highlights the location where presumably a room will be discovered in the immediate future.

Finally, we found LVT2 to be similar to GVT, albeit on a larger scale in both space and time. This tower produces the highest outputs on areas toward which the agent is currently directed and on far, unexplored territories that could be interesting in the future. Again, however, when the maximum is extracted from the convolutions output the information on its location is lost — unless the rogue is inside the receptive field — and only the knowledge of its presence is preserved: it is unclear how the network can use it so effectively.

In figure B.1 we can observe all the aspects we mentioned and note that no tower is free from noisy output, highlighting parts of the map that are not easily interpretable, such as the corners of the map. All in all, the most reasonable insight we gain from this analysis is that the neural network has redundant parameters and it could conceivably be reduced in size without a significant impact on its performance. Perhaps entirely removing tower LVT1, and possibly LVT2, or maybe altering the global max-pooling layers such that their output has shape $3 \times 3 \times C$ instead of $1 \times 1 \times C$ in order to retain some information on the position of features, would prove to be interesting experiments: we leave them for future work.



(a)



(b)

Figure B.1: Three towers focus; in red we show GVT, in green LVT1 and in blue LVT2; we use lighter colors for coordinates that had the highest value in multiple channels of the same tower

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Andrea Asperti, Daniele Cortesi, and Francesco Sovrano. Crawling in rogue's dungeons with (partitioned) a3c. *arXiv preprint arXiv:1804.08685*, 2018.

[3] Andrea Asperti, Carlo De Pieri, and Gianmaria Pedrini. Rogueinabox: an environment for Roguelike learning. *International Journal of Computers*, 2, 2017.

[4] Andrea Asperti, Carlo De Pieri, Mattia Maldini, Gianmaria Pedrini, and Francesco Sovrano. A modular deep-learning environment for Rogue. *WSEAS Transactions on Systems and Control*, 12, 2017.

[5] Marc Bellemare, Sriram Srinivasan, Georg Ostrovski, Tom Schaul, David Saxton, and Remi Munos. Unifying count-based exploration and

intrinsic motivation. In *Advances in Neural Information Processing Systems*, pages 1471–1479, 2016.

[6] Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

[7] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 41–48. ACM, 2009.

[8] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[9] Vojtech Cerny and Filip Dechterenko. Rogue-like games as a playground for artificial intelligence–evolutionary approach. In *International Conference on Entertainment Computing*, pages 261–271. Springer, 2015.

[10] Nuttapong Chentanez, Andrew G Barto, and Satinder P Singh. Intrinsically motivated reinforcement learning. In *Advances in neural information processing systems*, pages 1281–1288, 2005.

[11] François Chollet et al. Keras. `https://keras.io`, 2015.

[12] Giuseppe Cuccu, Julian Togelius, and Philippe Cudre-Mauroux. Playing atari with six neurons. *arXiv preprint arXiv:1806.01363*, 2018.

[13] Thomas Degris, Martha White, and Richard S Sutton. Off-policy actor-critic. In *International Conference on Machine Learning*, pages 457–464, 2012.

[14] Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, and Yuhuai Wu. Openai baselines. `https://github.com/openai/baselines`, 2017.

[15] Karol Gregor, Ivo Danihelka, Alex Graves, Danilo Rezende, and Daan Wierstra. Draw: A recurrent neural network for image generation. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 1462–1471, Lille, France, 07–09 Jul 2015. PMLR.

[16] Matthew Hausknecht and Peter Stone. Deep recurrent q-learning for partially observable mdps. *CoRR, abs/1507.06527*, 2015.

[17] Sepp Hochreiter. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 6(02):107–116, 1998.

[18] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[19] Max Jaderberg, Volodymyr Mnih, Wojciech Marian Czarnecki, Tom Schaul, Joel Z Leibo, David Silver, and Koray Kavukcuoglu. Reinforcement learning with unsupervised auxiliary tasks. *arXiv preprint arXiv:1611.05397*, 2016.

[20] Max Jaderberg, Karen Simonyan, Andrew Zisserman, and koray kavukcuoglu. Spatial transformer networks. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 2017–2025. Curran Associates, Inc., 2015.

[21] Peter Karkus, David Hsu, and Wee Sun Lee. Qmdp-net: Deep learning for planning under partial observability. In *Advances in Neural Information Processing Systems*, pages 4697–4707, 2017.

[22] Michał Kempka, Marek Wydmuch, Grzegorz Runc, Jakub Toczek, and Wojciech Jaśkowski. Vizdoom: A doom-based ai research platform for

visual reinforcement learning. In *Computational Intelligence and Games (CIG), 2016 IEEE Conference on*, pages 1–8. IEEE, 2016.

[23] Jens Kober and Jan Peters. Reinforcement learning in robotics: A survey. In *Reinforcement Learning*, pages 579–610. Springer, 2012.

[24] Jan Krajicek. BotHack. `https://github.com/krajj7/BotHack`, 2015.

[25] Tejas D Kulkarni, Karthik Narasimhan, Ardavan Saeedi, and Josh Tenenbaum. Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation. In *Advances in neural information processing systems*, pages 3675–3683, 2016.

[26] Yuxi Li. Deep reinforcement learning: An overview. *arXiv preprint arXiv:1701.07274*, 2017.

[27] Kosuke Miyoshi. Replicating the UNREAL algorithm. `https://github.com/miyosuda/unreal`, 2017.

[28] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*, pages 1928–1937, 2016.

[29] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[30] George E Monahan. State of the art - a survey of partially observable markov decision processes: theory, models, and algorithms. *Management Science*, 28(1):1–16, 1982.

[31] Rémi Munos, Tom Stepleton, Anna Harutyunyan, and Marc Bellemare. Safe and efficient off-policy reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 1054–1062, 2016.

[32] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

[33] Christos H Papadimitriou and John N Tsitsiklis. The complexity of markov decision processes. *Mathematics of operations research*, 12(3):441–450, 1987.

[34] Gianmaria Pedrini. Rogueinabox: a rogue environment for AI learning. Framework development and agents design. Bachelor's thesis, 2017.

[35] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2016.

[36] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International Conference on Machine Learning*, pages 1889–1897, 2015.

[37] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[38] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[39] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas

Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017.

[40] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2015.

[41] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

[42] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabinovich, et al. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–9, 2015.

[43] Aviv Tamar, Yi Wu, Garrett Thomas, Sergey Levine, and Pieter Abbeel. Value iteration networks. In *Advances in Neural Information Processing Systems*, pages 2154–2162, 2016.

[44] Gerald Tesauro. Td-gammon: A self-teaching backgammon program. In *Applications of Neural Networks*, pages 267–285. Springer, 1995.

[45] Alexander Vezhnevets, Volodymyr Mnih, Simon Osindero, Alex Graves, Oriol Vinyals, John Agapiou, et al. Strategic attentive writer for learning macro-actions. In *Advances in neural information processing systems*, pages 3486–3494, 2016.

[46] Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John Agapiou, Julian Schrittwieser, et al. Starcraft ii: a new challenge for reinforcement learning. *arXiv preprint arXiv:1708.04782*, 2017.

[47] Ziyu Wang, Victor Bapst, Nicolas Heess, Volodymyr Mnih, Remi Munos, Koray Kavukcuoglu, and Nando de Freitas. Sample efficient actor-critic

with experience replay. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2017.

[48] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[49] Christopher John Cornish Hellaby Watkins. *Learning from delayed rewards*. PhD thesis, King's College, Cambridge, 1989.

[50] Marco Wiering and Jürgen Schmidhuber. Hq-learning. *Adaptive Behavior*, 6(2):219–246, 1997.

[51] Daan Wierstra, Alexander Foerster, Jan Peters, and Juergen Schmidhuber. Solving deep memory pomdps with recurrent policy gradients. In *International Conference on Artificial Neural Networks*, pages 697–706. Springer, 2007.

[52] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. In *Reinforcement Learning*, pages 5–32. Springer, 1992.

Acknowledgements

I am very grateful to Andrea Asperti for his guidance, support and ideas over the course of the months I dedicated to this work. I also want to thank Carlo De Pieri and Gianmaria Pedrini, authors of the Rogueinabox library and previous efforts on the subject, who were the source more ideas and help, and Francesco Sovrano, whom I collaborated with in the first experiment and who gave me good advices.

Finally I want to thank my friends for the fun times and my mother and father for their help and the sacrifices they made for me.