University of Warsaw
Faculty of Mathematics, Informatics and Mechanics

Mikołaj Błaż
Student no. 346862

Guiding tree search with policy-based reinforcement learning

Master's thesis
in COMPUTER SCIENCE

Supervisor:
dr hab. Piotr Miłoś
Institute of Mathematics
Polish Academy of Sciences

December 2019

Supervisor's statement

Hereby I confirm that the presented thesis was prepared under my supervision and that it fulfils the requirements for the degree of Master of Computer Science.

Date Supervisor's signature

Author's statement

Hereby I declare that the presented thesis was prepared by me and none of its contents was obtained by means that are against the law.

The thesis has never before been a subject of any procedure of obtaining an academic degree.

Moreover, I declare that the present version of the thesis is identical to the attached electronic version.

Date Author's signature

Abstract

Planning is a demanding task spanning multiple branches of Artificial Intelligence, including Reinforcement Learning and classical tree search algorithms. In this work, we examine a recently published Levin Tree Search (LevinTS) algorithm, which closely connects those two areas by using a policy to guide tree search procedure. Firstly, we experimentally evaluate LevinTS (and its slight modifications) behavior on the Sokoban environment. Secondly, we relax the requirement of providing a policy as an input to LevinTS by proposing a planning algorithm which combines LevinTS with reinforcement learning and can learn a good policy from a scratch. Experimental evaluation demonstrates that this method is comparable to learning from expert trajectories.

Keywords

planning, tree search, reinforcement learning, imitation learning, policy improvement, deep reinforcement learning, artificial intelligence

Thesis domain (Socrates-Erasmus subject area codes)

11.3 Informatics, Computer Science

Subject classification

I. Computing Methodologies
I.2. Artificial Intelligence
I.2.8. Problem Solving, Control Methods, and Search

Tytuł pracy w języku polskim

Przeszukiwanie drzew sterowane uczeniem ze wzmocnieniem opartym o politykę.

Contents

Chapter 1

Introduction

Planning is one of the core Artificial Intelligence challenges. It has received great attention in multiple fields, including reinforcement learning. Recent achievements in different tasks, like board games (e.g. AlphaGo [Silver]), demonstrate the potential of bringing together deep learning with planning and search techniques, like Monte-Carlo Tree Search.

One of primary approaches to planning is finding a path in some state space. Well-known examples of path finding algorithms include classical tree search methods, like depth-first search or breadth-first search. One of the most widespread extension of Dijkstra's algorithm for finding shortest path in graphs is A*. It is a search algorithm employed with a heuristic function, which guides the search and helps in reaching the goal state faster than an uninformed search.

Finding a good heuristic function can be challenging, in particular it can require domain expertise. A lot of work in the deep neural network field is devoted to replacing or enhancing hand-crafted descriptors of some entites with learnable representations ([Mahony]). It was achieved successfully in images domain and many others, including video games. The ability to act in such environments is one of reinforcement learning challenges. In a nutshell, the role of a heuristic is taken over by a policy, which guides an agent towards the goal.

Recently, [Orseau] tried to bridge a classical planning algorithm with reinforcement learning field by creating Levin Tree Search, a tree search algorithm which can be guided by a policy, which plays a role equivalent to a heuristic function in A*. Together with some theoretical guarantees (similar to those established in classical path finding), they demonstrated practical usefulness of the algorithm on the Sokoban environment. It is a puzzle requiring an agent to push a set of boxes onto goal locations, which is a common test-bed for planning methods.

In this work, we further explore this area of research. Firstly, we develop some modifications and variations of the base Levin Tree Search algorithm and evaluate quantitatively their effect on algorithm performance. Secondly, we connect this search method with policy learning algorithm, which allows to remove the requirement of providing an external policy to LevinTS. This novel combination proves to be successful on the same task of the Sokoban game.

Chapter 2

Background

2.1. Reinforcement Learning

Levin Tree Search is an algorithm guided by a policy, which is a central concept in reinforcement learning (RL). Therefore, before analyzing the algorithm, let us introduce some formalization. Reinforcement learning deals with the task of sequential decision making framed as an *agent* performing *actions* in an environment, transitioning from state to state and receiving rewards ([Sutton]). The goal of the agent is to maximize its expected cumulative reward. Formally, this task is represented by a Markov Decision Process (MDP) - a tuple $(\mathcal{S}, \mathcal{A}, T, R, \gamma)$ where:

- $\mathcal{S}$ - the state space of possible environment states,
- $\mathcal{A}$ - the action space of possible agent actions,
- $T : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ - the transition function describing the environment changes after certain actions taken in the given state[1].
- $R : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ - the reward function describing rewards granted to the agent after certain transitions[2].
- $\gamma \in [0, 1)$ - the discount factor (described in the next section).

We can describe the agent's behavior by a notion of a *policy*. A stochastic policy is a function $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$ mapping states[3] and corresponding actions to probabilities of their execution (conditioned by the current state). We can notice that the transition, reward and policy function depend only on the current state of the environment (and not the history of states). This Markov property is a common assumption in reinforcement learning and is therefore reflected in the definitions.

To sum up, a typical interaction loop looks as follows: given a state $s \in \mathcal{S}$, an agent samples an action $a \in \mathcal{A}$ from actions distribution $\pi(s, \cdot)$, determined by the agent's policy π. Then the environment state changes to $T(s, a) = s'$ and the agent receives a reward $R(s, a, s')$.

As for the learning algorithm, we can roughly divide RL methods into two categories: *model-free* and *model-based*. The first type of approaches try to directly represent the policy

[1] For simplicity we consider only the case of a deterministic transition function, i.e. each action leads only to one state. Stochastic transitions can be represented as a $T : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ function.

[2] The reward function should be bounded.

[3] Sometimes it useful to distinguish the states and *observations*, which describe what part of the state the agent can actually observe. Here we consider only the case of a fully-observable environment.

function (or a value function, introduced in the following section) and find an (nearly) optimal one without explicit knowledge about the environment dynamics. In model-based approaches, the transition (T) and reward (R) functions are either known or approximated, in order to perform planning to find the best sequence of actions. Examples of these two types will be presented in the next two sections.

2.2. Policy learning

The task of an RL agent is to find a policy which maximizes the expected return (also called a V-value function in [Francois-Lavet et.al.]) at any state, defined as:

$$V^{\pi}(s) = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k} \,|\, s_t = s, \pi\right] ,$$

where:

- $s_{t+1} = T(s_t, a_t)$ is the environment state at timestep $t+1$,
- $a_t \sim \pi(s_t, \cdot)$ is the action sampled at timestep t,
- $r_t = R(s_t, a_t, s_{t+1})$ is the reward at timestep t,
- $\gamma \in [0, 1)$ is the discount factor. We can see that it makes the immediate rewards preferable to the more distant ones.

The problem of finding (or approximating) an optimal policy is a central concern of reinforcement learning algorithms and in *model-free* methods it is approached by directly representing the policy or value function[4]. The policy or value function can be learned with many different methods. Recently, a lot of work is done in the field of Deep Reinforcement Learning, where those functions are approximated with neural networks.

In this work, we will use neural network policies and train them with imitation learning. Network parameters are learned in a supervised setting from expert trajectories: having a training dataset consisting of pairs $(s, a) \in \mathcal{S} \times \mathcal{A}$, which represent actions taken by experts in expert trajectories, we can train the network to predict 1 when presented an input (s, a) and 0 for input (s, a') if $a' \neq a$. Similarly to [Mnih], if the $\mathcal{A}$ space is discrete, we can actually make the network predict $\pi(s, a)$ for all $a \in \mathcal{A}$, given a state s on input.

2.3. Tree search

Having a model of an environment, we can make use of model-based methods related to path planning. A simple (and powerful) method to perform planning is a tree search. Formally, we define set of tree nodes as $\mathcal{N} := \mathcal{A}^* \cup \mathcal{A}^{\infty}$, i.e. (possibly infinite) sequences of actions. The environment starts in the initial state $s_0 \in \mathcal{S}$. For any $n \in \mathcal{N}$, let $S(n)$ denote the state of the environment after executing actions n (which is a sequence), starting from the s_0 state. Any sequence of action is uniquely represented by a tree node, but multiple action sequences can lead to the same environment state. Hence, for any state $s \in \mathcal{S}$ let $\mathcal{N}(s) := \{n \in \mathcal{N} : S(n) = s\}$ - all action sequences that lead to state s. Tree structure follows sequence prefixes: a node $n = (a_1, \ldots, a_k)$ has children $\mathcal{C}(n) := \{(a_1, \ldots, a_k, a) : a \in \mathcal{A}\}$. The tree root is denoted as n_0 and it as an empty sequence.

[4]Given an optimal value function, an optimal policy can be recovered from it.

Following [Orseau], we will consider the task of finding paths which lead to goal states $\mathcal{S}^g \subseteq \mathcal{S}$. This situation can be cast to RL formalism by defining a sparse reward: an agent receives a nonzero reward only after reaching a goal state ($R(s,a,s') \neq 0 \iff s' \in \mathcal{S}^g$). The search algorithm traverses a tree until it reaches one of the target nodes $\mathcal{N}^g \subseteq \mathcal{N}$, defined as $\{n : S(n) \in S^g\}$. When it comes to traversal order, there are many different alternatives. We will consider a broad class of search orders with *best-first* property.

2.3.1. Best-first search

The two famous and simple tree traversal algorithms, depth-first search and breadth-first search, are examples of uninformed search, in which the algorithm looks for target nodes with no external information.

However, suppose that we are given an arbitrary cost (or evaluation) function $\text{cost} : \mathcal{N} \to \mathbb{R}$, which provides information about nodes "goodness". We can extend this function also to states, by setting $\text{cost}(s) = \min_{n \in \mathcal{N}(s)} \text{cost}(n)$. The search has a *best-first* property, if for all states $s_1, s_2 \in \mathcal{S}$, if $\text{cost}(s_1) < \text{cost}(s_2)$, then s_1 is visited before s_2.

Algorithm 1: Generic best-first search

Input: cost function cost
Result: Flag indicating whether any target node was reached

$\mathcal{V} := \emptyset$
$\mathcal{F} := \{n_0\}$
while $\mathcal{F} \neq \emptyset$ **do**
  $n := \arg\min_{n \in \mathcal{F}} \text{cost}(n)$
  $\mathcal{F} := \mathcal{F} \setminus \{n\}$
  $s := S(n)$
  if $s \in \mathcal{S}^g$ **then**
    return true
  $\mathcal{V} := \mathcal{V} \cup \{n\}$ `// in basic algorithm version, V is not used`
  $\mathcal{F} := \mathcal{F} \cup \{\mathcal{C}(n)\}$
return false

The Algorithm 1 is an example of a general best-first search. Iteratively, a node with the lowest cost is chosen from the *fringe* (or *open*) set. After visiting, the node is added to the *visited* (or *closed*) set and its children are added to the fringe set. In some situations (depending on the cost function form) we can avoid visiting nodes corresponding to the same states multiple times. We will see such example in the next section.

Many well-known algorithms fall into this schema. If we denote $d(n)$ as the node's $n \in \mathcal{N}$ depth, then the aforementioned BFS and DFS algorithms can be obtained by setting, respectively, $\text{cost}(n) = d(n)$ and $\text{cost}(n) = \frac{1}{d(n)+1}$. If $g(n)$ is the weighted distance from node n to the root and $h(n)$ is a heuristic assessing distance of node n to the target node, then the three algorithms[5]: Dijkstra's search, greedy best-first search and A*, result from using the following cost functions: $\text{cost}(n) = g(n)$, $\text{cost}(n) = h(n)$ and $\text{cost}(n) = g(n) + h(n)$.

[5] To be exact, we obtain those algorithm versions without state cuts, i.e. which don't make use of the *visited* nodes set.

2.4. Levin Tree Search

Another algorithm from the same family, which we will focus on in this work, is Levin Tree Search (LevinTS). LevinTS is one of the two policy-guided algorithms proposed in [Orseau]. Its behavior depends on a policy π, which is given as an input to the algorithm. The policy guidance is reflected (only) in the cost function, which takes the following form:

$$cost(n) := \frac{d(n)}{\tilde{\pi}(n)} , \tag{2.1}$$

where

$$\tilde{\pi}((a_1, \ldots, a_k)) := \prod_{i=1}^{k} \pi(S((a_1, \ldots, a_i)), a_i) .$$

Put differently, $\tilde{\pi}(n)$ is the probability of executing sequence of actions n, while following the policy π from initial state. It follows that $\tilde{\pi}(n_0) = 1$ and $\forall n \in \mathcal{N} : \tilde{\pi}(n) = \sum_{n' \in \mathcal{C}(n)} \tilde{\pi}(n')$.

Algorithm 2: Levin Tree Search

Input: Policy π
Result: Flag indicating whether any target node was reached

1 $\mathcal{V} := \emptyset$
2 $\mathcal{F} := \{n_0\}$
3 **while** $\mathcal{F} \neq \emptyset$ **do**
4 $n := \arg\min_{n \in \mathcal{F}} \frac{d(n)}{\tilde{\pi}(n)}$
5 $\mathcal{F} := \mathcal{F} \setminus \{n\}$
6 $s := S(n)$
7 **if** $s \in \mathcal{S}^g$ **then**
8 **return** true
9
10 **if** *isMarkov*(π) **then**
11 **if** $\exists n' \in \mathcal{V} : (S(n') = s) \wedge (\tilde{\pi}(n') \geq \tilde{\pi}(n))$ **then**
  // state cut: no need to further expand n,
  // since s was already visited with higher probability
12 **continue**
13 $\mathcal{V} := \mathcal{V} \cup \{n\}$
14
15 $\mathcal{F} := \mathcal{F} \cup \{\mathcal{C}(n)\}$
16 **return** false

The whole LevinTS is written as Algorithm 2. It turns out that in the case of a Markovian policy (and, in this work, we consider only policies that depend only on a single state and a single action), we can limit the number of traversed tree nodes by performing state cuts. Authors of [Orseau] prove that under certain conditions - stated clearly in line 10 of LevinTS - we can discard traversing node children, while still maintaining the best-first property. Thanks to the best-first property, the search ends upon reaching the first goal state - the found path is optimal with respect to the given cost function.

The authors of LevinTS focus on its theoretical guarantees. We will not explore this subject further except for noting that this algorithm is guaranteed to find a solution (if one

exists) in a number of steps limited by[6]

$$\min_{n \in \mathcal{N}^g} \frac{d(n)}{\tilde{\pi}(n)}. \tag{2.2}$$

We will introduce some modifications and evaluate the algorithm performance in subsequent chapters.

2.5. Sokoban

All evaluations take place on Sokoban - a benchmark environment for planning tasks. An example level of this game is presented in Figure 2.1. In order to complete a level, the (green) agent must push all four boxes (yellow) onto target locations (red points). Neither the agent nor any box can occupy a cell with a wall (brick).

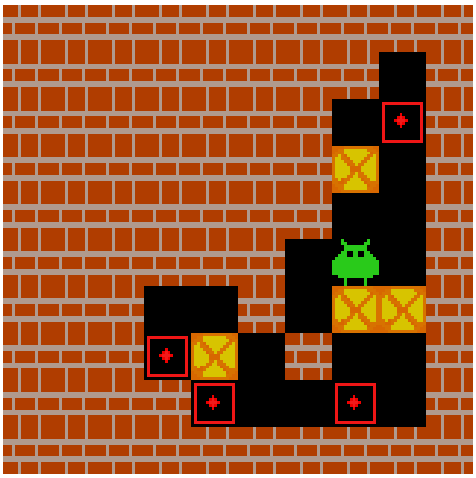

Figure 2.1: Example of a 10x10 Sokoban game.

Levels used both during agent training and LevinTS evaluation are procedurally generated (as in [Weber]) on the fly, creating diverse set ranging from easy to hard ones. In chapter 3 we report the ratio of boards solved by LevinTS, which we evaluated using a set of 1000 boards. The evaluation levels were *not* fixed, but their number was large enough to avoid significant variations in evaluation metrics. In chapter 4 we will use LevinTS with an iteratively improving policy and therefore we will employ a running average of solved boards as an evaluation metric. Finally, the size of the boards can vary. We use 8x8 boards in chapter 3 and 10x10 levels in chapter 4.

[6] Theorem 3 in [Orseau].

Chapter 3

Levin Tree Search with a fixed policy

3.1. Motivation

In its origin paper [Orseau], Levin Tree Search is presented as a quite powerful search algorithm, capable of solving all considered test levels (where the BFS algorithm managed to solve only 9% of them). It is inherently tied to a policy, so it is a natural idea to combine it with some policy learning algorithm. This will be the subject of chapter 4, but beforehand, we shall study the LevinTS algorithm itself. It is important to analyze where the good performance comes from. Is it the policy guidance, or is it the algorithm mechanism which guarantees an upper bound for the number of steps (equation 2.2)? We will try to examine these questions in this chapter, evaluating LevinTS with a fixed policy.

3.2. Input policy

The policy serving as a guidance to LevinTS used in this chapter is a neural network trained on expert trajectories. The trajectories were gather in advance with an exhaustive BFS search[1] to form a fixed dataset of moves. A validation set was held out in order to determine optimal training time. The network was trained to reproduce expert moves, which can be framed as a simple classification task of mapping an environment state to the correct action.

3.3. Policy modifications

As a policy guided algorithm, LevinTS depends strongly on the policy quality. An effective policy steers the search towards the goal, while a poor one does not bring much improvement. Indeed, if the policy is uniform (all actions are always chosen with equal probability), then the $\pi(n)$ factor in equation 2.1 depends only on the node depth. Consequently, the cost becomes an (increasing) function of node depth, so the nodes are expanded in breadth-first order – LevinTS reduces to BFS.

This mechanisms suggests that the policy *confidence* plays an important role in LevinTS. Given a policy, we can modify its confidence by a *temperature* parameter. Assuming that the policy probabilities are obtained by applying softmax functions to the logits, we can multiply the logits by a temperature factor to either sharpen (if temperature is bigger than 1) or smooth (if temperature is less than 1) the final policy distribution. A uniform policy is an extreme example of uncertain policy – it can be obtained from any policy by setting the temperature

[1] It guarantees finding the shortest available solutions

to 0. Conversely, very high temperature value results in a policy which assigns probability close to 1 to only one action (and approximately 0 to others). Concluding, the temperature gives us ability to evaluate a range of policies, with varying confidence degrees.

Another important property of the policy is its *sampling quality*. It can be measured by number of levels which can be solved using only sampling from the policy, i.e. choosing next action without any planning, just by sampling from the actions distribution. To obtain policies of different qualities, we took snapshots of the neural network from different epochs of training (ranging from the first to the final, tenth epoch).

The impact of policy confidence and quality on LevinTS performance is presented in Figure 3.1. Performance of LevinTS is measured as a ratio of solved levels, when given a budget of 1000 search steps. As already stated, temperature 0 makes the algorithm behave as BFS, whereas temperature 10000 is enough to make the policy almost deterministic. For each model checkpoint, there are two curves, depicting performance of LevinTS (with that policy as an input) and policy sampling quality for reference.

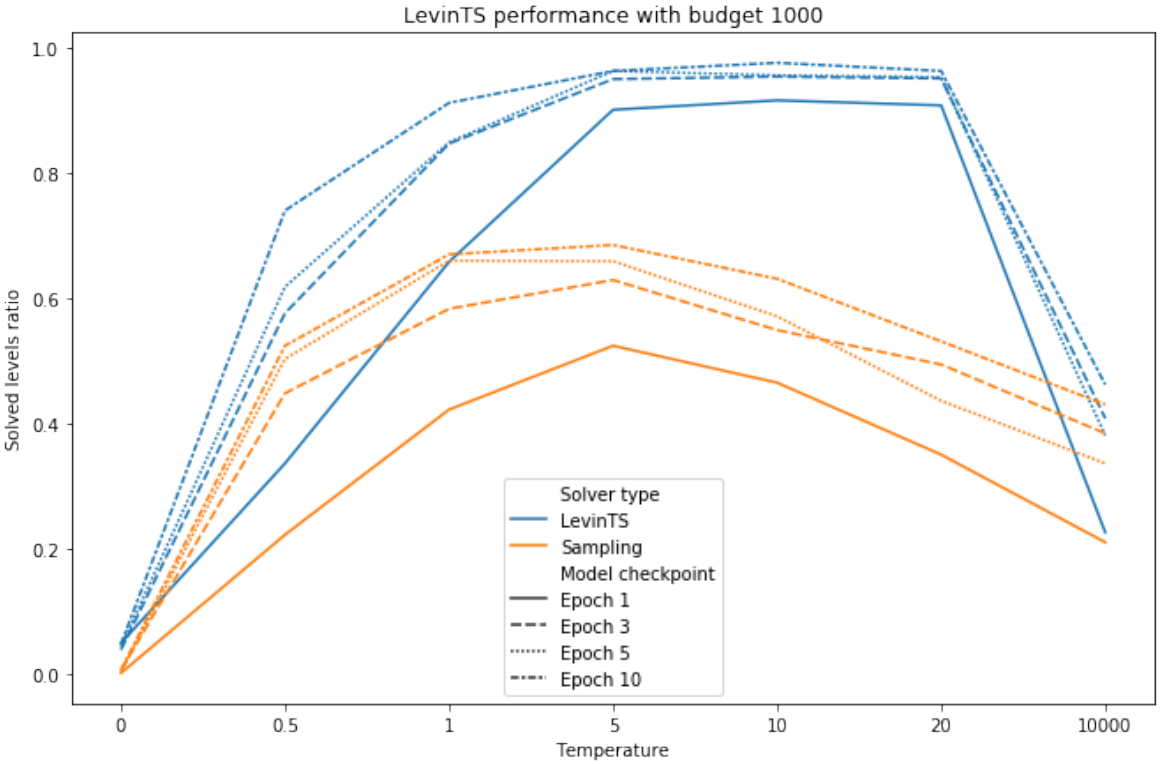

Figure 3.1: LevinTS performance in different temperatures and for different checkpoints, with policy sampling quality as reference

We can immediately notice that higher confidence improves LevinTS performance. Obviously, this is true to a certain degree, because a deterministic policy makes the Levin Tree Search behave very greedily. Interestingly, policy sampling quality for the best model peaks at temperature 1 – the temperature it was trained in. In such low temperatures, sampling quality differences between the checkpoints are reflected in LevinTS performance. For higher temperatures, those differences collapse, because almost all checkpoints reach nearly 100% performance, which is summarized also in the Table 3.1.

Temperature	0	0.5	1	5	10	20	10000
Epoch 1	5.0	33.7	65.9	90.2	**91.7**	90.9	22.7
Epoch 3	3.9	57.7	84.8	95.1	**95.5**	95.2	40.9
Epoch 5	4.4	61.9	85.0	**96.4**	95.7	95.4	38.2
Epoch 10	4.8	74.1	91.3	96.4	**97.7**	96.4	46.3

Table 3.1: LevinTS performance (in %) in different temperatures and for different checkpoints.

3.4. Algorithm modifications

Levin Tree Search algorithm is unambiguously defined by its cost function and the fact that it is a best-first search. Let us recall the cost formula characterizing LevinTS:

$$\text{cost}(n) := \frac{d(n)}{\tilde{\pi}(n)} \,.$$

We already explored the policy component of this function. The depth $d(n)$ factor is a key ingredient of LevinTS and was included in cost by its authors for two reasons: to ensure theoretical properties and to avoid greedy behavior of the search. The second argument is exemplified in [Orseau] with a search tree with an infinite path-like branch consisting of nodes with high $\pi(n)$ value. Without the $d(n)$ factor, search would never escape from that branch after entering it.

Keeping in mind the theoretical significance, we wanted to evaluate the $d(n)$ regularization usefulness experimentally, too. For this purpose, let us consider a slight modification of Levin Tree Search algorithm, with the cost function defined as:

$$\text{cost}(n) := \frac{r(d(n))}{\tilde{\pi}(n)} \,, \tag{3.1}$$

where $r : \mathbb{N} \to \mathbb{R}$ is a *balancing* function. If the balancing function grows rapidly, the search is penalized for visiting too deep tree areas, whereas a sub-linear balancing will encourage deeper tree exploration. We consider a few simple functions, like: identity, square, square root, log and a constant (which cancels out the depth factor completely).

Additionally, the evaluation takes place for different *level budgets* (number of allowed search steps) and with a policy with a fixed confidence (temperature 1) and quality (last epoch). The results are presented in Figure 3.2, along with the reference sampling quality. We can notice the obvious strong dependence of performance on level budget. Interestingly, for very low budgets sampling turns out to be a better strategy than searching. When it comes to balancing, we should keep in mind that optimal balancing depends strongly on the problem type. In our experiment, there are no substantial differences between most balancing types. Understandably, for small budgets broad search is not a good idea at all, since it prevents the search from finding the solution quickly. However, even for bigger budgets, removing the depth factor completely seems to be a slightly better alternative to the identity function.

3.5. Summary

The work presented in this chapter has an exploratory character. It aimed to analyze the Levin Tree Search algorithm from different perspectives and find the most influencing factors for its behavior, before applying it in a more complex setting of a policy improvement framework. Expectedly, the algorithm was quite sensitive to policy variations and level budget limiting.

Figure 3.2: LevinTS performance for different level budgets and balancing functions.

Interestingly, for the Sokoban environment, balancing the search with a depth factor did not play an important role. One of the significant findings of the analysis is the confirmation of LevinTS ability of improving even a low-quality policy - it will be vital in our further results. It was not clear from [Orseau], since their whole study was based on an already well-trained policy of approximately 60% quality.

The summary of all experiments conducted so far is presented in Figure 3.3. Four plots correspond to different network checkpoints. Different colors represent temperatures and different balancing functions are depicted as intervals surrounding plot lines. Finally, sampling quality is provided for reference as dotted lines. Notice increasing levels of quality for consecutive model checkpoints. Also, this way of presenting the results highlights the importance of the temperature parameter on LevinTS performance.

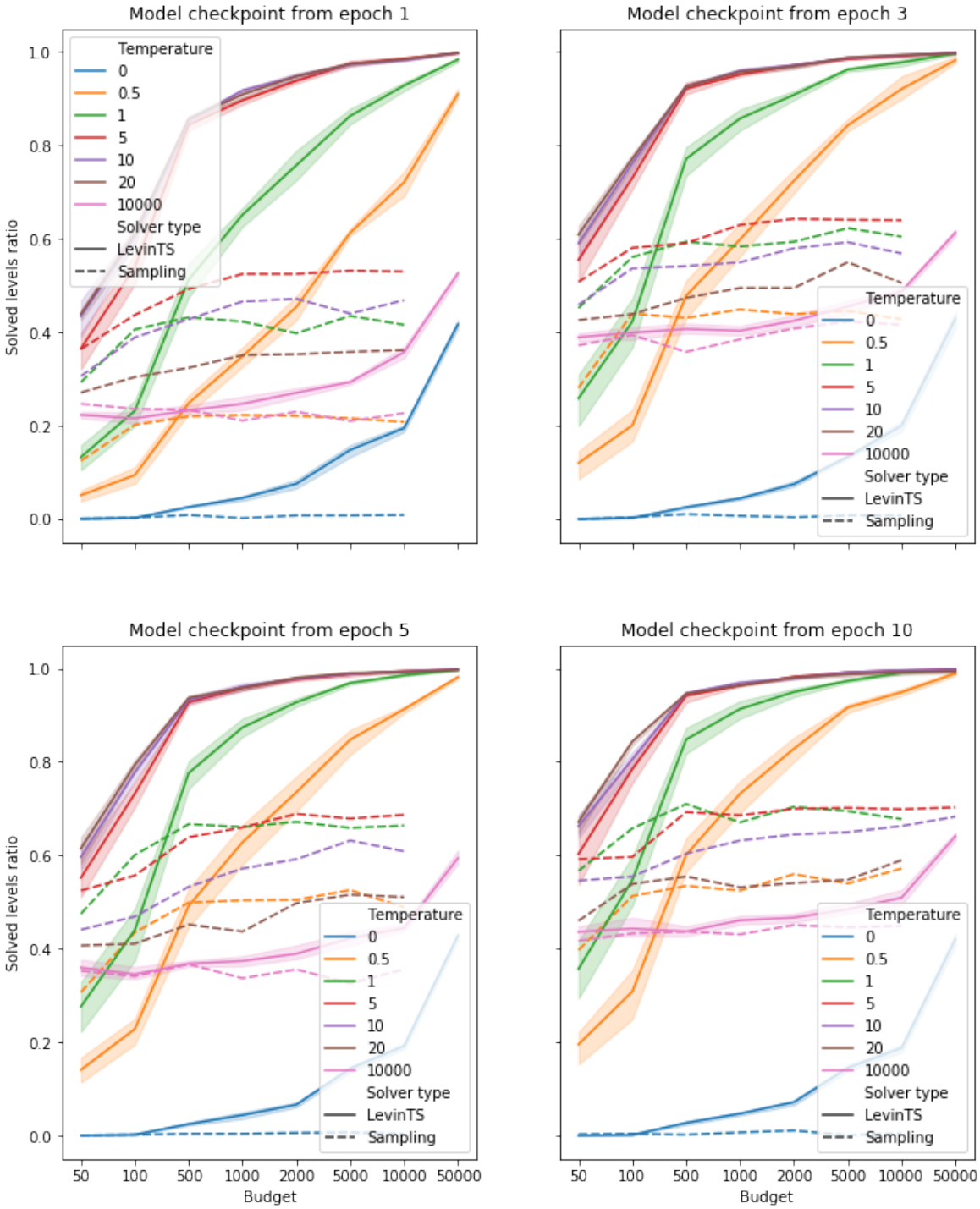

Figure 3.3: LevinTS performance (solid lines) and sampling quality (dotted lines) in different temperatures (colors), checkpoints (figures), level budgets (x-axis) and balancing functions (line intervals).

Chapter 4

Levin Tree Search with reinforcement learning

4.1. Motivation

In the previous chapter we explored the possibility of enhancing a *fixed* policy with Levin Tree Search algorithm. The necessity of providing an informative policy to LevinTS as an input is a demanding requirement. We would like to establish a self-sufficient procedure, which can relax this prerequisite. We can think of improving a given policy with LevinTS as a one step of a closed-loop system, which starts from a random policy and improves it iteratively. Such ideas have already been explored[1], mainly with Monte-Carlo Tree Search (MCTS) as a policy improvement component. However, when dealing with a situation of direct approximation of an optimal policy, LevinTS seems to be a very well suited tree search, operating directly with the policy as a guidance. At the time of writing, LevinTS has never been used in such end-to-end learning framework. We will examine its combination with imitation learning, which perfectly fits the image. This conjunction of tree search and policy learning results in a successful model-based reinforcement learning algorithm, which is proved experimentally in this chapter.

4.2. Algorithm

From a broad perspective, the algorithm idea is to alternately use the two components, LevinTS and imitation learning:

- With LevinTS, gather trajectories which will be used to learn a better policy.
- With those trajectories, improve the policy by imitation learning. The improved policy will be used again by LevinTS.

Those two steps are executed in a loop by the what we call the "workers" and the "server".

4.2.1. Workers

There may be multiple workers, each executing the same task independently. Their main goal is solving as many different Sokoban levels as fast as possible. The LevinTS is used as the

[1] One example of such system is AlphaZero ([Silver]). Another example, closer to our work, is a system described in [Anthony].

planning algorithm, with a neural network trained by the server (and updated periodically in each worker) as a policy. After finding a solution, it is delivered to the server in form of a trajectory from the initial to the end state, consisting of pairs $(s, a) \in \mathcal{S} \times \mathcal{A}$.

4.2.2. Server

The server is a single computing node with two roles: gathering solutions from the workers and training a neural network serving as a policy for LevinTS run on workers.

Replay buffer

As described in 4.2.1, each solution received from a worker has a form of trajectories. They are stored in a replay buffer[2] – a FIFO queue of limited size with a fixed number of the newest trajectories only.

Neural network

The neural network serves as a global policy (a single network shared between all workers). For a given environment state $s \in \mathcal{S}$, it predicts $\Pi(s) \in \mathbb{R}^4$ – four numbers, which can be interpreted as probabilities of performing four possible actions (going up, down, left or right[3]). Network architecture is relatively simple and consist of five convolutions layers followed by a dense one.

The network training follows the imitation learning procedure by treating solutions obtained with LevinTS as expert trajectories and using them to directly improve the policy. A single training iteration consist of several steps:

1. Sample a batch of transitions of length B by uniform sampling of B trajectories from the replay buffer, and then one transition within each trajectory. The result of this step is a batch of states $X \in \mathcal{S}^B$ and a corresponding batch of actions $y \in \mathcal{A}^B$.
2. Input X to the neural network and obtain batch of outputs $\Pi(X) \in \mathbb{R}^{B \times 4}$.
3. One-hot encode batch of expert actions y into $y' \in \mathbb{R}^{B \times 4}$ (since $|\mathcal{A}| = 4$).
4. Treat y' as labels and compute cross-entropy loss between $\Pi(X)$ and y'.
5. Preform gradient descent on network weights from computed loss and update weights accordingly.

4.3. Results

The resulting algorithm manages to learn a very well performing policy. When evaluated on the benchmark environment (10x10 Sokoban levels), it achieved 89% ratio of solved levels, meeting the performance of a baseline policy trained directly on expert trajectories (91%), and greatly outperforming the LevinTS run with a uniform policy (3%). Detailed results, further demonstrating usefulness of the developed algorithm, are presented below, with the greatest emphasis on three parameters analyzed in the previous chapter: temperature, balancing and level budget.

[2] This is a technique called *experience replay*, introduced in [Lin] and popularized by [Mnih].
[3] To refer it to previous notation, $\Pi(s) = (\pi(s, UP), \pi(s, DOWN), \pi(s, LEFT), \pi(s, RIGHT))$.

4.3.1. Temperature

This parameter, significant for the LevinTS with a fixed policy, turned out to be less important in policy improvement setting. The Figure 4.1 shows how the temperature affects the initial phase of learning. However, in the end, policies learning in all temperatures manage to achieve similar performance. The final performance, summarized in the Table 4.1, is similar regardless of the temperature.

A possible explanation is that our learning procedure finally causes the policy to saturate, meaning that it becomes confident and does not need sharpening its predictions with a temperature factor. In the previous chapter, the policy provided as an input to LevinTS was trained on a fixed dataset of trajectories, and therefore required the learning to be ended when the validation loss stopped improving. The loss value is not necessarily strictly correlated with the policy confidence, and it turned out to be the case with the fixed policy - its confidence required adjusting in order to ensure an optimal performance in combination with LevinTS.

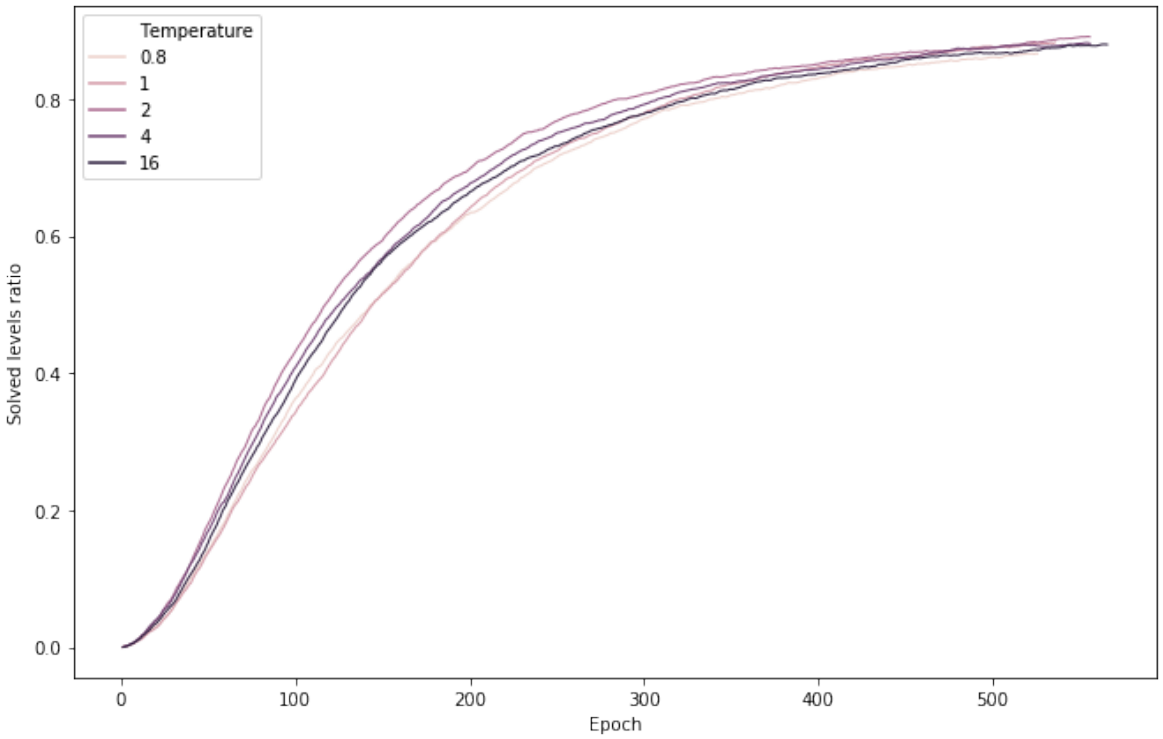

Figure 4.1: Comparison of learning process for different temperatures.

Temperature	0.8	1	1.5	2	4	16
LevinTS + RL performance	86.6%	88.2%	88.0%	**89.0%**	88.1%	87.9%

Table 4.1: Agent performance after learning in different temperatures.

4.3.2. Balancing

In contrast to the temperature, the balancing plays a relatively bigger role here than in LevinTS with a fixed policy setting. The Figure 4.2 reveals that this parameter affects the performance in all learning stages and the Table 4.2 confirms that it has a meaningful impact when the learning finishes.

We speculate that it might be caused by the fact that the balancing function has bigger consequences when LevinTS is combined with a learning algorithm. The balancing function influences the cost function, and in a single LevinTS run, its only implication is the tree exploration order and the time required to find a solution.

In LevinTS + RL algorithm, the trajectories (optimal in terms of the cost function) are collected and used for policy training. It means that the balancing function impacts not only the ability to solve single levels in a given budget, but also determines what type of trajectories are considered optimal and imitated by the policy. For example, a $r(d) = d^2$ function steers the search away from deep solutions and causes the replay buffer to contain less trajectories for levels requiring long solutions. It might inhibit the policy from learning to solve increasingly difficult levels in the training process.

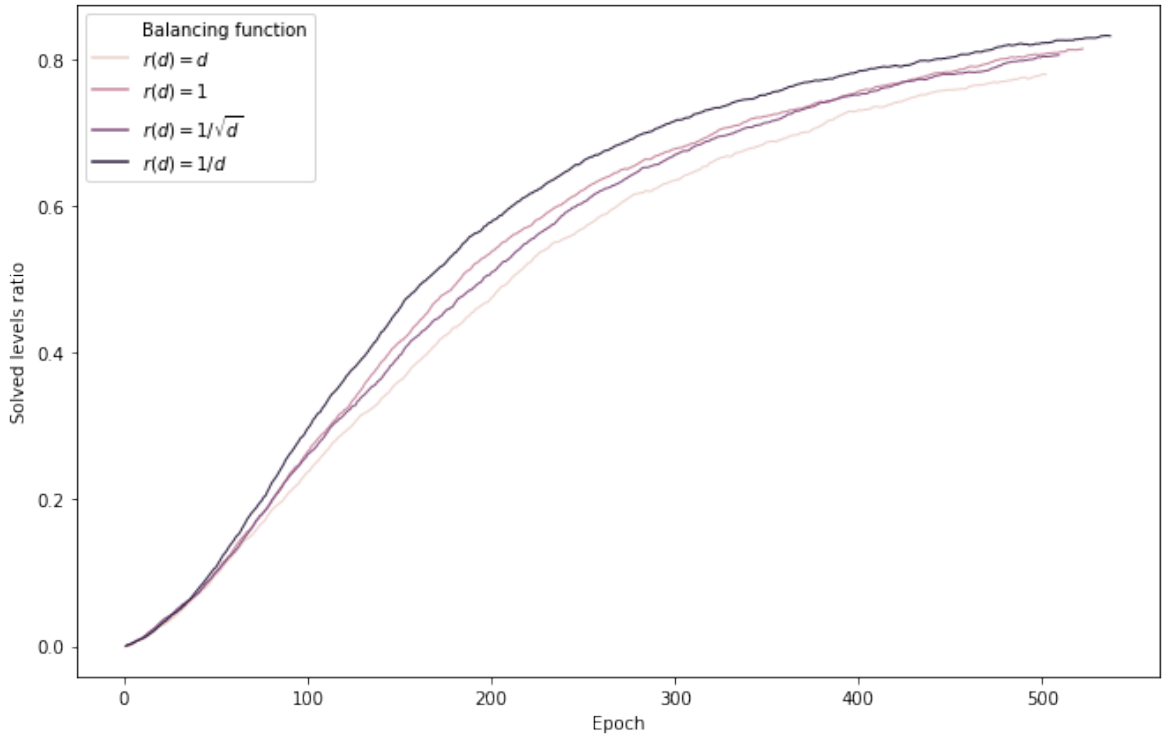

Figure 4.2: Comparison of the learning process for different balancing functions.

Balancing function	$r(d) = d$	$r(d) = 1$	$r(d) = 1/d$	$r(d) = 1/\sqrt{d}$
LevinTS + RL performance	78.0%	81.5%	**83.3%**	80.7%

Table 4.2: Agent performance after learning with different balancing functions.

4.3.3. Budget

Budget defines a number of allowed interactions with the environment for a single level. Naturally, it impacts solving efficacy more dramatically than the temperature or balancing. It is illustrated in the Figure 4.3, which confirms that bigger budgets give opportunity to solve more levels.

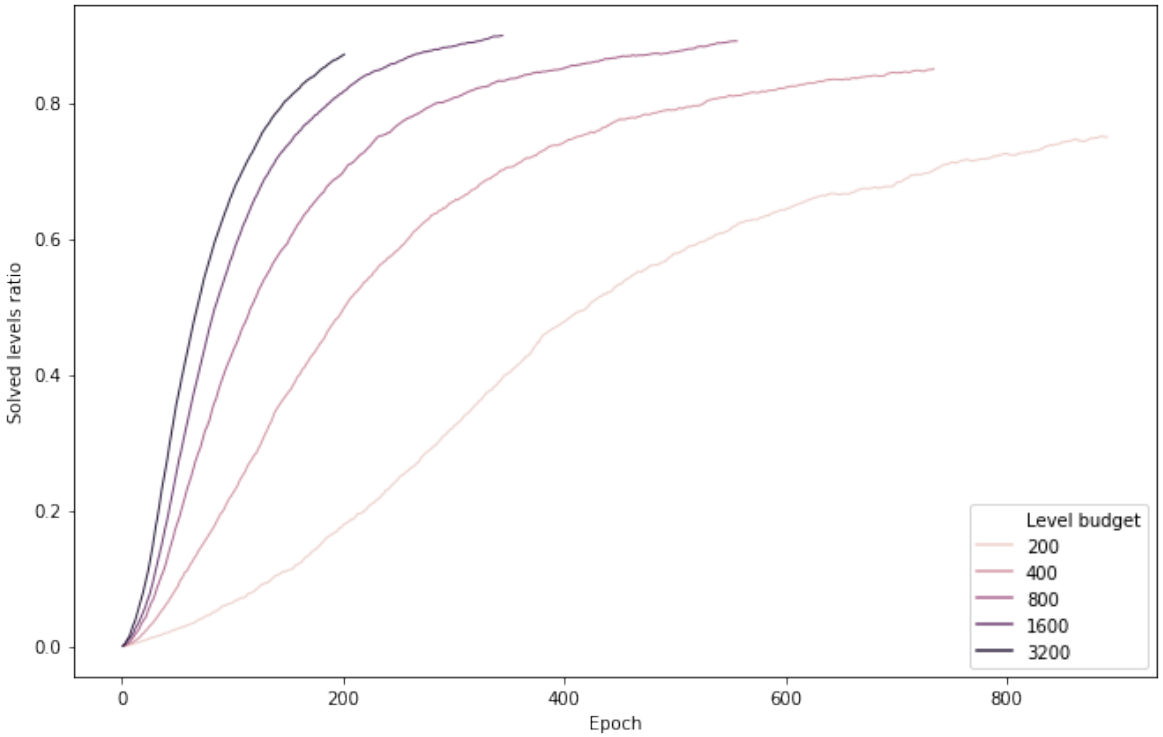

Figure 4.3: Comparison of learning process for different level budgets. Each agent was trained for 24 hours of wall time.

However, the figure's x scale represents epochs, which correspond to the number of processed levels. If we take into account the fact that allowing more environment interactions consumes more time, we notice that agents with a big budget actually can learn slower than with a low budget. In the figure 4.3, all agents are given the same amount of *wall* time. The best performing is the one with 1600 steps, and not with 3200 steps, which is summarized in the Table 4.3.

Level budget	200	400	800	1600	3200
LevinTS + RL performance	75.0%	85.0%	89.0%	**89.8%**	87.0%
LevinTS with uniform policy performance	0.7%	0.9%	3.3%	4.4%	**7.0%**

Table 4.3: Agent performance after learning for equal amount of wall time with different level budgets, accompanied with performance of LevinTS with a uniform policy for reference.

Another important detail included in the Table 4.3 is the performance of LevinTS with a uniform policy with different budgets. At the very beginning of learning, the agent's policy behaves close to a uniform one, so these numbers give estimates of solved levels ratio in the initial learning phase. Even with the ratio as low as 0.7%, the agent is able to take off the

ground and start learning a useful policy.

4.4. Hyperparameters

Besides LevinTS-specific parameters described above, a number of different hyperparameters were evaluated in search of optimal parameters setting. These included:

- learning rate,
- batch size,
- L2 regularization coefficient,
- label smoothing coefficient,
- replay capacity.

For each parameter, several experiments were run in order to select the most promising parameter value. Parameter values were chosen individually, since running a full grid search would be computationally unobtainable. Still, careful one-by-one tuning resulted in increasing the overall performance by approximately 10%. The best parameters set, described in detail in appendix A, was used to run the final experiments comparing LevinTS to the baselines and other work.

4.5. Comparison

In order to assess performance of our planning algorithm, we used two baselines of different nature. The first one was LevinTS run with the best performing policy from chapter 3 (trained on expert trajectories). Performance of this network is an upper bound for a policy trained without expert data, but trying to simulate perfectly solved levels. The second baseline and the lower bound for efficiency was LevinTS used with a uniform policy. As mentioned in section 3.3, this results in the search being equivalent to BFS.

We compared performance with respect to baselines by giving a fixed number of allowed node expansions (i.e. interactions with an environment), set to 800 for all methods. The results presented in the Table 4.4 show that our algorithm (LevinTS + RL) improves drastically over the baseline with a uniform policy. Also, it stays close to the upper bound performance of the baseline LevinTS with pretrained policy (which, in fact, is superior only if evaluated in temperatures grater than, e.g. 20).

Method	LevinTS + RL	LevinTS with uniform policy	LevinTS[1] with pretrained policy	LevinTS[20] with pretrained policy
Performance	89.0%	3.3%	66.1%	**91.3%**

Table 4.4: LevinTS policy improvement in comparison to baselines. LevinTS[T] means Levin Tree Search using policy with temperature T.

Comparing our method with other work in the field is not that straightforward due to various evaluation procedures. It is possible, however, to roughly relate efficiency of our algorithm to the original LevinTS paper ([Orseau]) results. Its authors assess performance of LevinTS (with a fixed policy) by computing the ratio of solved levels when given a budget of 100000 node expansions. Not surprisingly, it is capable of solving *all* given levels. However,

accompanying plots suggest that within a budget of 1000 steps, that version of LevinTS solves around 90-95% of levels. It must be reminded that this algorithm depends strongly on the policy sampling quality (which was equal to 67% in Orseau's work). Also, LevinTS in [Orseau] is evaluated only with a *fixed* policy, just like the baselines in the Table 4.4. Therefore, presented numbers serve only as a rough idea of different results.

4.6. Conclusions

Presented results - the Table 4.4 in particular - prove that our algorithm joining Levin Tree Search with policy learning is a successful combination. Even with a uniform initial policy (which is as weak as 3% in terms of our evaluation), it can produce an agent performing on par with an agent trained with an expert supervision.

We can think of the "LevinTS + RL" algorithm in two different ways. Firstly, it can be seen as an extension of pure Levin Tree Search requiring an input policy, which is not always easy to produce. With our framework, a good policy is a result of the algorithm, not its input.

On the other hand, we can look at LevinTS as a relatively simple and yet powerful policy improvement procedure. It relies on a policy (and only policy) to guide search, so it is an ideal companion for a policy-based RL algorithm, and can be employed in situations in which hand-crafted heuristics or value function approximators are difficult to produce.

4.7. Further work

We can see a few possible directions to explore. One of the most promising is adapting LevinTS parameters (like the temperature or budget) dynamically to agent behavior in different learning phases. For instance, setting a high temperature in initial phase could speed up agent exploration, since its policy is close to uniform at the beginning of learning. In later epochs, temperature is not that important, as the policy is already confident in predictions.

Another example of adaptive parameters is setting a high initial budget, so as to bootstrap policy learning. LevinTS requires at least *some* positive results when it comes to solved levels in order to start learning (but, as we have seen in section 4.3.3, even very low solved ratio is sufficient). Therefore, giving enough budget may be more important at the beginning, but not that essential with better policy in later phases.

Appendix A

Hyperparameters

The Table A.1 summarizes all hyperparameters used during neural network training. Provided values are the ones used during final evaluation in section 4.5. All parameters were varied and the best combination was selected with a partial grid search.

Parameter	Value	Description
Temperature	2	Logits multiplication factor that controls the final policy distribution smoothness.
Balancing function	$r(d) = \frac{1}{d}$	Function that influences the cost form. $r(d) = 1/d$ gives the search more depth-first rather than breadth-first character.
Learning rate	0.0002	The learning rate used by the RMSPProp optimizer.
Batch size	128	Number of training examples used at once to perform the backpropagation step.
L2 coefficient	0.0001	Coefficient of L2 regularization term in the loss function.
Label smoothing coefficient	0.005	Label smoothing allows to smooth ground truth action probabilities distribution (e.g. with 0.005 coefficient, a 1 becomes 0.995 in the distribution).
Replay capacity	10000	Size of the replay buffer which stores trajectories to be learned from.
Policy network train frequency	100	Number of gathered trajectories after which the policy network parameters are updated.
Solved trajectories ratio	1	Ratio of solved trajectories used for imitation learning[1].

Table A.1: Hyperparameter values used for final evaluation.

[1] Ratio lower than 1 would mean that also unsolved trajectories are used for imitation learning. Expectedly, it had detrimental effect on learning.

Bibliography

[Orseau] Laurent Orseau, Levi H. S. Lelis, Tor Lattimore, Théophane Weber, *Single-Agent Policy Tree Search With Guarantees*. 32nd Conference on Neural Information Processing Systems, Montreal, Canada, NIPS 2018.

[Bellman] Richard Bellman, *A Markovian decision process*. Journal of Mathematics and Mechanics: 679–684, 1957.

[Sutton] Richard S. Sutton, Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second edition, The MIT Press, Cambridge, MA, 2018.

[Francois-Lavet et.al.] Vincent Francois-Lavet, Peter Henderson, Riashat Islam, Marc G. Bellemare, Joelle Pineau. *An Introduction to Deep Reinforcement Learning*. Foundations and Trends in Machine Learning: Vol. 11, No. 3-4, 2018.

[Bojarski] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, Karol Zieba. *End to End Learning for Self-Driving Cars*. arXiv:1604.07316. 2016.

[Mnih] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin Riedmiller. *Playing Atari with Deep Reinforcement Learning*. NIPS Deep Learning Workshop, 2013.

[Lin] Long-Ji Lin. *Self-improving reactive agents based on reinforcement learning, planning and teaching*. Machine learning, 8(3-4):293–321. 1992.

[Anthony] Thomas Anthony, Zheng Tian, David Barber. *Thinking Fast and Slow with Deep Learning and Tree Search*. arXiv preprint arXiv:1705.08439, 2017.

[Silver] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, Demis Hassabis. *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm*. arXiv preprint arXiv:1712.01815. 2017.

[Mahony] Niall O' Mahony, Sean Campbell, Anderson Carvalho, Suman Harapanahalli, Gustavo Velasco-Hernandez, Lenka Krpalkova, Daniel Riordan, Joseph Walsh. *Deep Learning vs. Traditional Computer Vision*. Advances in Computer Vision Proceedings of the 2019 CVC. Springer Nature Switzerland AG, pp. 128-144, 2019.

[Weber] Théophane Weber, Sébastien Racanière, David P. Reichert, Lars Buesing, Arthur Guez, Danilo Jimenez Rezende, Adria Puigdomènech Badia, Oriol Vinyals, Nicolas Heess,

Yujia Li, Razvan Pascanu, Peter Battaglia, Demis Hassabis, David Silver, Daan Wierstra. *Imagination-Augmented Agents for Deep Reinforcement Learning.* arXiv preprint arXiv:1707.06203v2. 2018.