Ranking Policy Decisions

Anonymous Author(s)
Affiliation
Address
email

Abstract

Policies trained via Reinforcement Learning (RL) without human intervention are often needlessly complex, making them difficult to analyse and interpret. In a run with n time steps, a policy will make n decisions on actions to take; we conjecture that only a small subset of these decisions delivers value over selecting a simple default action. Given a trained policy, we propose a novel black-box method based on statistical fault localisation that ranks the states of the environment according to the importance of decisions made in those states. We argue that among other things, the ranked list of states can help explain and understand the policy. As the ranking method is statistical, a direct evaluation of its quality is hard. As a proxy for quality, we use the ranking to create new, simpler policies from the original ones by pruning decisions identified as unimportant (that is, replacing them by default actions) and measuring the impact on performance. Our experimental results on a diverse set of standard benchmarks demonstrate that pruned policies can perform on a level comparable to the original policies. We show that naive approaches for ranking policies, e.g. ranking based on the frequency of visiting a state, do not result in high-performing pruned policies. To the best of our knowledge, there are no similar techniques for ranking RL policies' decisions.

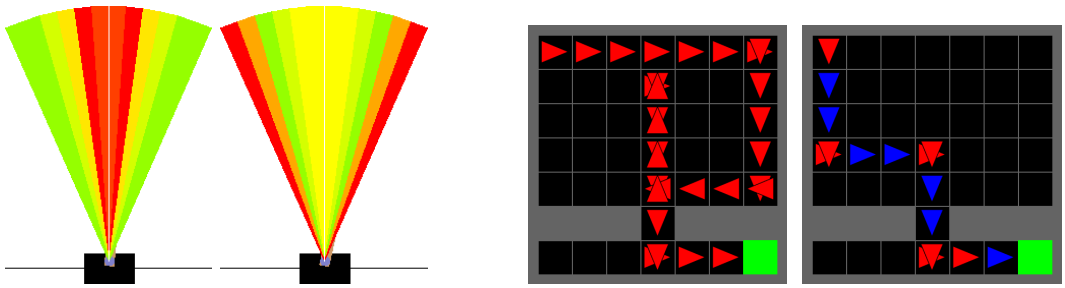
1 Introduction

Reinforcement learning is a powerful method for training policies that complete tasks in complex environments. The policies produced are optimised to maximise the expected reward provided by the environment. While performance is clearly an important goal, the reward typically does not capture the entire range of our preferences. By focusing solely on performance, we risk overlooking the demand for models that are easier to analyse, predict and interpret [17]. Our hypothesis is that many trained policies are *needlessly complex*, i.e., that there exist alternative policies that perform just as well or nearly as well but that are significantly simpler. This tension between performance and simplicity is central to the field of explainable AI (XAI), and machine learning as a whole [11]; our method aims to help by highlighting the most important parts of a policy.

The starting point for our definition of "simplicity" is the assumption that there exists a way to make a "simple choice", that is, there is a simple default action for the environment. We argue that this is the case for many environments in which RL is applied: for example, "repeat previous action" is a straightforward default action for navigation tasks.

The key contribution of this paper is a novel method for *ranking policy decisions* according to their importance relative to some goal. We argue that the ranked list of decisions is already helpful in explaining how the policy operates. We evaluate our ranking method by using the ranking to simplify policies without compromising performance, hence addressing one of the main hurdles for wide adoption of deep RL: the high complexity of trained policies.

Submitted to 35th Conference on Neural Information Processing Systems (NeurIPS 2021). Do not distribute.



(a) (b)

Figure 1: **(a)** CartPole, in a state where the cart and pole are moving rapidly. The heatmap represents the frequency of appearance of the possible pole angles (left) and the importance scores following SBFL (right). While it is more frequent for the pole to be centered, SBFL successfully identifies that the policy's decisions are more important when the pole is close to falling. **(b)** MiniGrid. Traces of executions with the original policy (left) and a *pruned policy* (right). States in which we take the default action are in blue. Both policies succeed, but pruning unimportant actions simplifies the policy.

We produce a ranking by scoring the states a policy visits. The rank reflects the impact that replacing the policy's chosen action by the default action has on a user-selected binary outcome, such as "obtain more than X reward". It is intractable to compute this ranking precisely, due to the high complexity and the stochasticity of the environment and the policy, complex causal interactions between actions and their outcomes, and the sheer size of the problem. Our work uses *spectrum-based fault localisation (SBFL)* techniques [21, 33], borrowed from the software testing domain, to compute an approximation of the ranking of policy decisions. SBFL is an established technique in program testing for ranking the parts of a program source code text that are most likely to contain the root cause of a bug. This ranking is computed by recording the executions of a user-provided test suite. SBFL distinguishes passing and failing executions; failing executions are those that exhibit the bug. Intuitively, a program location is more likely to be the root cause of the bug if it is visited in failing executions but less (or not at all) in passing ones. SBFL is a lightweight technique and its rankings are consistently highly correlated with the location of the root cause of the bug [33]. We argue that SBFL is also a good fit for analysing complex RL policies.

Our method applies to RL policies in the black-box manner, making no assumptions about the policy's training or representation. We evaluate the quality of the ranking of the decisions by the proxy of creating new, simpler policies (we call them "pruned policies") without retraining, and then calculate the reward achieved by these policies. Experiments with agents for MiniGrid (a more complex version of gridworlds) [7], CartPole [4] and a number of Atari games [4] demonstrate that pruned policies maintain high performance (similar or only slightly worse than that of the original policy) when taking the default action in the majority of the states (often 90% of the states). As pruned policies are much easier to understand than the original policies, we consider this an important step towards explainable RL. As pruning a given policy does not require re-training, the procedure is relatively lightweight. Furthermore, the ranking of states by itself provides an important insight into the importance of particular decisions for the performance of the policy overall.

The code for reproducing our experiments is available on GitHub[1], and further examples are provided on the anonymised website[2].

2 Background

2.1 Reinforcement learning (RL)

We use a standard reinforcement learning (RL) setup and assume that the reader is familiar with the basic concepts. An *environment* in RL is defined as a Markov decision process (MDP) and is denoted by $\langle S, A, P, R, \gamma, T \rangle$, where S is the set of states, A is the set of actions, P is the transition function, R is the reward function, γ is the discount factor, and T is the set of terminal states. An agent seeks

[1] https://github.com/AnonUser-532438/PolicyRankingAnon

[2] https://sites.google.com/view/ranking-policy-decisions/

to learn a policy $\pi : S \to A$ that maximises the total discounted reward. Starting from the initial state s_0 and given the policy π, the state-value function is the expected future discounted reward as follows:

$$V_\pi(s_0) = \mathbb{E}\left(\sum_{t=0}^{\infty} \gamma^t R(s_t, \pi(s_t), s_{t+1})\right) \tag{1}$$

A policy $\pi : S \to A$ maps states to the actions taken in these states and may be stochastic. We treat the policy as a black box, and hence make no further assumptions about π.

2.2 Spectrum-based fault localization (SBFL)

The reader is likely less familiar with spectrum-based fault localization (SBFL), as to the best of our knowledge, it has not yet been used in RL. We therefore give a detailed description. SBFL techniques [33] have been widely used as an efficient approach to aid in locating the causes of failures in sequential programs. SBFL techniques rank program elements (say program statements) based on their *suspiciousness scores*, which are computed using correlation-based measures. Intuitively, a program element is more suspicious if it appears in failed executions more frequently than in correct executions, and the exact formulas differ between the measures. Diagnosis of the faulty program can then be conducted by manually examining the ranked list of elements in descending order of their suspiciousness until the cause of the fault is found. It has been shown that SBFL techniques perform well in complex programs [3].

The SBFL procedure first executes the program under test using a set of inputs called the *test suite*. It records the program executions together with a set of Boolean flags that indicate whether a particular program element was executed by the current test. The task of a fault localization tool is to compute a ranking of the program elements based on the values of these flags. Following the notation from [21], the suspiciousness score of each program statement s is calculated from a set of parameters $\langle a^s_{ep}, a^s_{ef}, a^s_{np}, a^s_{nf} \rangle$ that give the number of times the statement s is executed (e) or not executed (n) on passing (p) and on failing (f) tests. For instance, a^s_{ep} is the number of tests that passed and executed s.

Many measures have been proposed to calculate the suspicious scores of program elements. In Equations (2a)∼(2d) we list a selection of popular and high-performing measures [1, 9, 15, 32]; these are also the measures that we use in our ranking procedure.

$$\text{Ochiai: } \frac{a^s_{ef}}{\sqrt{(a^s_{ef} + a^s_{nf})(a^s_{ef} + a^s_{ep})}} \tag{2a}$$

$$\text{Tarantula: } \frac{\frac{a^s_{ef}}{a^s_{ef}+a^s_{nf}}}{\frac{a^s_{ef}}{a^s_{ef}+a^s_{nf}} + \frac{a^s_{ep}}{a^s_{ep}+a^s_{np}}} \tag{2b}$$

$$\text{Zoltar: } \frac{a^s_{ef}}{a^s_{ef} + a^s_{nf} + a^s_{ep} + \frac{10000 a^s_{nf} a^s_{ep}}{a^s_{ef}}} \tag{2c}$$

$$\text{Wong-II: } a^s_{ef} - a^s_{ep} \tag{2d}$$

SBFL-based tools present the list of program elements in the order of their descending suspiciousness scores to the user. There is no single best measure for fault localization; different measures perform better on different types of programs, and it is best practice to use multiple measures [19].

While more sophisticated versions of SBFL exist [2, 5], in this work we prefer to stick to the simpler approach, which was sufficient for producing notable results.

3 Method: ranking policy decisions using SBFL

Inspired by the use of SBFL for localising the cause of a program's outcome, we propose a new SBFL-based method to identify the states in which decisions made by an RL policy are most important for achieving its objective. Our method is modular and is composed of two phases: (1) generating mutant policies and (2) ranking states based on the importance.

3.1 Definitions

Executions We apply the SBFL technique to a set of *executions* (sometimes called *trajectories* in the literature) of a given RL policy π with mutations. An *execution* τ of π describes a traversal of the agent through the environment MDP using the RL policy π and is defined as a sequence of states $s_0, s_1, \ldots$ and actions $a_0, a_1, \ldots$, where s_0 is an initial state and each subsequent state s_{i+1} obtained from the previous state s_i by performing action a_i, as chosen using $\pi(s_i)$. The last state must be a terminal state. As π or the environment can be stochastic, each execution of π may result in a different sequence of actions and states, and hence in a different τ. The set of all possible executions is denoted by $\mathcal{T}$.

Passing and failing executions An execution is either successful or failed. We define the success of an execution as a (binary) value of a given assertion on this execution. For example, the assertion can be that the agent reaches its destination eventually, or that the reward of this execution is not lower than 0.75 of the maximal reward for π. The assertion induces a Boolean function $C : \mathcal{T} \to \{0, 1\}$. We say that an execution τ is a *pass* if $C(\tau) = 1$, and is a *fail* otherwise. We use a binary condition for simplicity, as SBFL is designed to work with passing and failing executions. This condition can be relaxed [5], and we plan to investigate generalising this in future work.

Mutant executions We use SBFL to understand the impact of replacing actions by a *default action*. The choice of the default action d is context dependent and can be set by the user. For example, an obvious default action for navigation is "continue driving in the same direction". The default action can in principle be as basic as a single action, or as complex as a fully-fledged policy. In our experiments, we use the default action "repeat the previous action", defined as:

$$d(s_0, \ldots, s_i, a_0, \ldots, a_{i-1}) = a_{i-1} \tag{3}$$

Using the default action, we create *mutant executions*, in which the agent takes the default action d whenever it is in one of the *mutant states*. More formally, given a set of mutant states S_M, we act according to the policy π_{S_M} defined as:

$$\pi_{S_M}(s_0, \ldots, s_i, a_0, \ldots, a_{i-1}) = \begin{cases} \pi(s_i) & s_i \notin S_M, \\ d(s_0, \ldots, s_i, a_0, \ldots, a_{i-1}) & s_i \in S_M. \end{cases} \tag{4}$$

Decisions made in states in which the default action is a very good option are deemed less important. In these states, not following the policy would be less consequential. As the choice of d depends on the context and the user's goals, it is ultimately a user choice. If the user prefers not to define d, they can also set it to be a random action.

3.2 Generating the test suite and mutant executions on-the-fly

The naïve approach to generating a comprehensive suite of mutant executions for applying SBFL would be to consider all possible sets of mutant states S_M—that is, we would need to consider all possible subsets of the state space S. However, the state space of most RL environments is too large to enumerate, and enumerating all possible subsets of S is intractable even for simplistic environments.

We use two algorithmic techniques to address this problem: (a) we generate mutant executions *on-the-fly*, and (b) we use an abstraction function $\alpha : S \to \hat{S}$ to map the full set of states S to a smaller, less complex set of abstract states $\hat{S}$. Examples of these are in the supplementary material, and may for example include down-scaling or gray-scaling images that are input to the agent, or quantising a continuous state space. The set of possible mutations is then the set of subsets of $\hat{S}$, instead of the set of subsets of S. The test suite of mutant executions produced this way for π is denoted $\mathcal{T}(\pi) \subseteq \mathcal{T}$.

On-the-fly mutation We maintain a set $S_M \subseteq \hat{S}$ per execution. We begin each execution τ by initialising the set S_M with the empty set of states. At each step of the execution, upon visiting a state s, we check the current S_M. If $\alpha(s) \notin S_M$, we add $\alpha(s)$ to S_M according to the predefined *mutation rate* μ (and take the default action); otherwise, we use π to determine the action in this state. In case we re-visit an (abstract) state, we maintain the previous decision of whether to mutate state s or not. This way, the states that are never visited in any of the executions are never mutated; hence, we never

consider "useless mutations" that mutate a state that is never visited. We finish by marking τ as pass or fail according to $C(\tau)$. Note that a mutant execution may visit states not typically encountered by π, meaning that we are able to even rank states that are typically out of distribution. This might be especially important when trying to understand how the policy will behave in parts of the environment it is unfamiliar with.

Overall, our algorithm has five (tunable) parameters: the size of the test suite $|\mathcal{T}(\pi)|$, the passing condition C, the default action d, a *mutation rate* μ and the abstraction function α.

SBFL benefits from a balanced test suite of passing and failing executions [33], a ratio largely determined by the choices of μ and C. The choice of C depends on the context. In our experiments in Section 4.2, our goal was to find the states with highest impact on the expected reward. We set the condition to be "receive more than X reward" for some $X \in \mathbb{R}$, and chose X to yield a balanced suite (i.e. if there were too many failed runs, we lowered X). In most cases actions important for achieving at least X reward are important for maximising reward in general, so we found that this worked well. In the cases where some actions are not important for achieving X, but later important for getting even higher rewards, (e.g. states that only appear after achieving X reward) we would not have considered them important. The choice of μ is also significant. If it is too large, executions fail too often, and the behaviour in mutant executions is uninteresting. If μ is too small, we do not mutate states enough to learn anything, and in larger environment will fail to mutate many of the states we encounter. In our experiments, we selected μ manually, but this could easily be automated by a search algorithm.

3.3 Computing the ranking of the policy decisions

We now rank states according to the importance of policy decision made in these states, with respect to passing the condition. The ranking method is based on SBFL as described in Section 2.2. We first create the test suite of mutant executions $\mathcal{T}(\pi)$ as described above. We call the set of all abstract states encountered when generating the test suite $S_{\mathcal{T}} \subseteq \hat{S}$; these are the states to which we assign scores. Any unvisited state is given the lowest possible score by default.

Similarly to SBFL for bug localisation, for each state $s \in S_{\mathcal{T}}$ we calculate a vector $\langle a^s_{ep}, a^s_{ef}, a^s_{np}, a^s_{nf} \rangle$. We use this vector to track the number of times that s was unmutated (e) or mutated (n) on passing (p) and on failing (f) mutant executions, and we only update these scores based on executions in which the state was visited. In other words, the vector keeps track of success and failure of mutant executions based on whether an execution took the default action in s or not. For example, a^s_{ep} is the number of passing executions that took the action $\pi(s)$ in the state s, and a^s_{nf} is the number of failing executions that took the default action in the state s. Once we have constructed the vector $\langle a^s_{ep}, a^s_{ef}, a^s_{np}, a^s_{nf} \rangle$ for every (abstract) state, we apply the SBFL measures discussed in Section 2.2 to rank the states in π. This ranking is denoted by $rank : \hat{S} \to \{1, \ldots, |\hat{S}|\}$.

4 Experimental evaluation

Our goal is to demonstrate the applicability of our ranking method to a variety of standard environments and to provide evidence of the utility of the generated ranking. We aim to answer the following research questions:

RQ1: How can we measure the relative importance of decisions for achieving the reward? What is a good proxy for measuring this?

RQ2: Does the approach we present in this paper scale to large policies and complex environments?

RQ3: Can we use the ranking to explain policy decisions?

RQ4: Can the ranking procedure be used to determine when the policy has been sufficiently trained?

We answer these questions in Section 4.2 by performing extensive experiments on various environments and policies, including some very large ones.

4.1 Experimental setup

We experimented in several environments. The first is *Minigrid* [7], a gridworld in which the agent operates with a cone of vision and can navigate many different grids, making it more complex than a standard gridworld. In each step the agent can turn or move forward. We also used *CartPole* [4], the classic control problem with a continuous state-space. Finally, to test our ability to scale, we ran experiments in *Atari games* [4].

We use policies that are trained using third-party code. No state abstraction is applied to the gridworld environments (i.e., α is the identity). The state abstraction function for the CartPole environment consists of rounding the components of the state vector between 0 and 2 places, and then taking the absolute value. For the Atari games, as is typically done, we crop the game's border, gray-scale, down-sample to 18×14, and lower the precision of pixel intensities to make the enormous state space manageable. Note that these abstractions are not a contribution of ours, and were primarily chosen for their simplicity. For all experiments, we use "repeat the last action" as the default action.

Examples of some environments, and important states found in them, are given in Figs. 1a and 1b. Details about the state abstraction functions, policy training, hyperparameters, etc., are provided in Appendix A.

We define two additional measures to the SBFL ones in Eq. (2), for comparison. Eq. (5a) measures how frequently the state was encountered in the test suite. Eq. (5b) is a random ranking of the states visited by the test suite. We use the FreqVis measure as a baseline because we are not aware of any previous work with the same goals as ours for ranking policy decisions, and a naïve approach to determining importance may be simply looking at how frequently a state is visited.

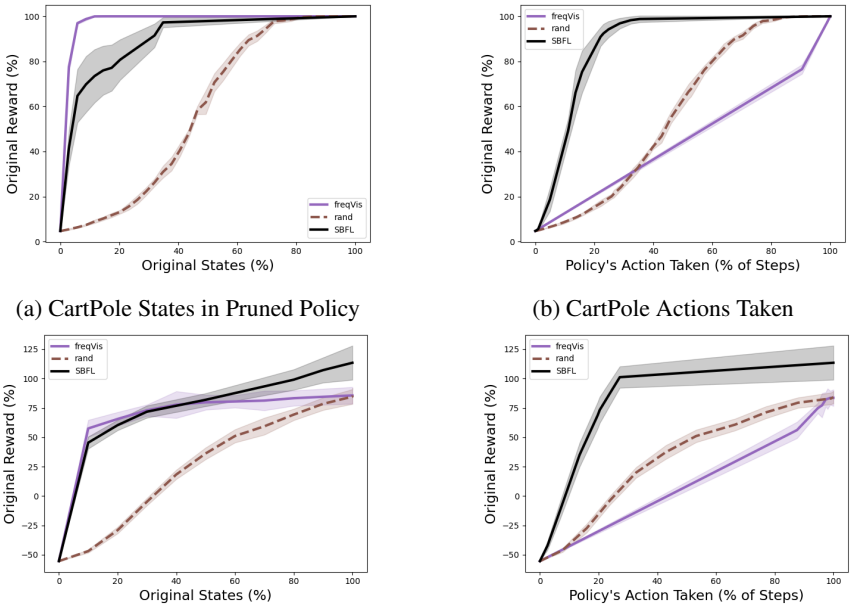
$$\text{FreqVis: } a^s_{ep} + a^s_{ef} + a^s_{np} + a^s_{nf} \qquad (5a) \qquad\qquad \text{Rand: } \sim \text{Unif}(0,1) \qquad (5b)$$

4.2 Experimental results

Performance of pruned policies The precise ranking of decisions according to their importance for the reward is intractable for all but very simple policies. To answer **RQ1**, in our experiments, we use the performance of *pruned policies* as a *proxy* for the quality of the ranking computed by our algorithm. In pruned policies, all but the top-ranked actions are replaced by the default actions. For a given r (a fraction or a percentage), we denote by $rank[r]$ the subset of r top-ranked states. We denote by π^r the pruned policy obtained by *pruning* all but the top-r ranked states. That is, an execution of π^r retains actions in the r fraction of the most important states from the original policy π and replaces the rest by default actions. The states that are in $rank[r]$ are called the *original states*. We measure the performance of the pruned policies for increasing values of r relative to the performance of the original policy π.

These results are shown in first four columns of Tab. 1, and some are represented graphically in Figs. 2a,c. SBFL in these stands for the *SBFL portfolio*, the combination of the four measures in Eq. (2), where the best result is taken at each point. The results show that the pruned policies obtained using the SBFL ranking can achieve a comparable performance that of π with less than 40% of the original decisions (and in some cases the number is as low as 20%). The performance of SBFL-based pruning is compared with random pruning in Eq. (5b) and FreqVis pruning in Eq. (5a). At first glance, it seems that the FreqVis pruned policies perform well in Figs. 2a,c. To better understand this, we show in the last two columns of Tab. 1 and in Figs. 2b,d how performance evolves with the proportion of steps in which the original policy is used over the default policy (i.e. not replaced by the default action). As shown in Figs. 2b and 2d, FreqVis does much worse by this metric, because it directly tries to make the pruned policy use the original policy as often as possible.

We observe that using our ranking method enables a significant pruning of the policies, while maintaining performance. To answer **RQ2**, we experimented with larger and more complex Atari environments. The results demonstrate that our framework is reasonably scalable, but also that the quality of the ranking is significantly improved if using a state abstraction. Results in the larger and more complex Atari environments show the effect of using a test suite that is too small: many states are not encountered while making the ranking. This means $rank[1]$, in which the original policy is used in *all* of the states discovered while making the ranking does not recover the performance of the original policy. Increasing the size of the test suite would help, but this is not a scalable solution. Instead, we use better *state abstractions*, which reduce the state space. In CartPole, this allows us to tackle a continuous domain. In Atari games, even the generic abstraction we use in all the games is



(a) CartPole States in Pruned Policy

(b) CartPole Actions Taken

(c) Krull States in Pruned Policy

(d) Krull Actions Taken

Figure 2: Performance of the pruned policies, measured as a percentage of the original reward. **(a)**,**(c)**: X-axis is the % of states where the original action is taken out of the set of all states encountered in the test suite. **(b)**,**(d)**: X-axis is the expected % of steps in which the original policy is followed over the default action during an execution of the pruned policy. See the supplementary material for more detail.

sometimes not enough ("NA" in Tab. 1). To show the potential utility of specialised abstractions, we create one for *Breakout*, in which we extract the coordinates of the ball and paddle. While the new abstraction does worse in terms of states pruned within the policy, it allows us to reach 90% of the policy's original performance with 13% fewer steps in which we use π over the default action.

SBFL ranking for better understanding the RL policy To answer **RQ3** directly, it requires a user study, which is out of scope for this paper. However, we base our conclusions on usefulness of the ranked list of decisions for understanding the policy on the existing research on the usefulness of the ranked list of program locations for understanding the causes of a bug in SBFL. While some studies suggest that the users typically do not go over the list of possible causes generated by SBFL linearly (and hence question the usefulness of the ranking) [22], the recent large-scale study demonstrates statistically significant and substantial improvements for the users who use an SBFL tool, and the results hold even for "mediocre" SBFL tools [34]. Based on this evidence, we suggest that the ranking can be used to explain policy decisions, as the ranked list itself would be helpful to identify the most important decisions. In addition, the pruned policies that we construct are simpler than the original policies while achieving a comparable performance, which can make identifying problems more straightforward. Examples are presented in the CartPole and Minigrid sections of the website.

Good vs bad policies Finally, we performed some initial work towards answering **RQ4**. Interestingly, our initial experiments demonstrate that the high performing and low performing policies agree on the 80% of actions in the top 10% of states. This suggests that the policy training (in CartPole) first picks the 'low-hanging fruit' by performing well in the most important states. The difference between a high-performing and a low-performing policy is mostly in the lower-ranked states. Full results for this experiment are available in Fig. 6 in the Appendix.

4.3 Comparison to existing work

Our results are difficult to compare to any existing work, as we are not aware of any methods with similar goals to our own; to rank the decisions made by policy according to their importance. However, some comparisons can be made based on the possible applications of our ranking, the most

Table 1: Minimum percentage of original states in pruned policies, and percentage of steps in which the original policy is used, before recovering 90% of original performance. Results are reported for SBFL portfolio ranking and (random) ranking. "X" denotes that the required reward was never reached, cf. Sec. 4.2.

Environment	% of original states restored in policy		% of steps that use π over default action	
MiniGrid	**49** ± 00	(99 ± 00)	**76** ± 00	(98 ± 01)
Cartpole	**31** ± 04	(65 ± 02)	**22** ± 04	(69) ± 02
Alien	X $\pm X$	(X $\pm X$)	X $\pm X$	(X $\pm X$)
Assault	**45** ± 07	(100 ± 00)	**93** ± 01	(100 ± 00)
Atlantis	**50** ± 00	(100 ± 00)	**99** ± 00	(100 ± 00)
BankHeist	X $\pm X$	(X $\pm X$)	X $\pm X$	(X $\pm X$)
BattleZone	**30** ± 00	(86 ± 07)	**84** ± 07	(84 ± 08)
Berzerk	**47** ± 12	(100 ± 00)	**88** ± 03	(100 ± 00)
Boxing	X $\pm X$	(X $\pm X$)	X $\pm X$	(X $\pm X$)
Breakout	**10** ± 00	(100 ± 00)	**54** ± 00	(99 ± 01)
Breakout (abs)	**40** ± 00	(85 ± 05)	**41** ± 00	(81 ± 06)
ChpperCmmnd	X $\pm X$	(X $\pm X$)	X $\pm X$	(X $\pm X$)
DemonAttack	**20** ± 00	(99 ± 01)	**98** ± 01	(99 ± 02)
Hero	**48** ± 04	(96 ± 03)	**86** ± 08	(96 ± 04)
IceHockey	**65** ± 20	(X $\pm X$)	**91** ± 08	(X $\pm X$)
Jamesbond	**30** ± 13	(68 ± 06)	**59** ± 20	(67 ± 06)
Krull	**75** ± 12	(99 ± 01)	**35** ± 21	(98 ± 02)
Phoenix	**30** ± 00	(92 ± 07)	97 ± 00	(**92** ± 07)
Pong	**21** ± 03	(79 ± 03)	**42** ± 01	(78 ± 03)
Qbert	**40** ± 00	(100 ± 00)	**84** ± 04	(100 ± 00)
Riverraid	**95** ± 05	(100 ± 00)	**99** ± 01	(100 ± 00)
RoadRunner	X $\pm X$	(X $\pm X$)	X $\pm X$	(X $\pm X$)
Seaquest	**48** ± 04	(94 ± 04)	92 ± 04	(**91** ± 05)
SpaceInvaders	**30** ± 00	(100 ± 00)	**93** ± 00	(100 ± 00)
StarGunner	**40** ± 00	(100 ± 00)	**99** ± 00	(100 ± 00)
YarsRevenge	X $\pm X$	(X $\pm X$)	X $\pm X$	(X $\pm X$)

obvious being interpretability. While the user study that would be required to make strong claims are outside of the scope of this work, we explore a comparison to motivate intuition. In Fig. 3, we show how a heatmap derived from our ranking differs from a conventional saliency map, based on taking the gradient of the policy's decision with respect to the state [10]. Our heatmap gives information about the general behaviour of the policy. In this case, points at which the agent needs to turn are redder (more important) than points where the agent is walking in a straight line. To understand why the downward-facing states in the right-most column are considered important, refer to our website for a full heatmap and explanation. On the other hand, the more conventional heatmap only shows information about a single decision (in this case, the one made from the initial state. We explore how other methods that create policy-level explanations differ from ours inSection 5.

5 Related work

There is a significant body of work in identifying the important parts of trained algorithms, but to the best of our knowledge none suggested to rank the states as we do. Prioritized experience replay [24] looks for the most important *transitions* for training. Saliency maps [10, 30, 35] identify the parts of the state that most influence influence the agent's decision. Sun et al. [27] have previously applied SBFL to visual feature importance for input images given to image classifiers. Other attempts include identifying the important parts of the *representation* of the policy by looking at the parameters of a trained model and pruning it to reduce its size [18]. None of these methods attempt to understand what the important decisions of the policy as a whole are.

Much of the recent work focuses on making deep learning models more interpretable [23, 20, 27]. Many approaches [10, 30, 35] to explaining deep reinforcement learning methods explain the decision made in a single state, without the context of the past or the future behavior. Iyer et al. [14] explain a

(a) (b)

Figure 3: Comparing a heatmap made using our ranking to conventional heatmaps. Colour from least to most important are blue, white and red. **(a)** Our heatmap based on the tarantula measure, showing all the states an agent encounters while walking to the goal, including the direction in which the agent was facing. For example, in the top left grid location, we show the importance of the state in which the agent is facing right, and the state in which it is facing down. **(b)** Conventional gradient-based heatmap, showing how important every part of the agent's observation is to its decision in the initial state. The former conveys information about the policy as a whole, while the second zooms in on a single decision.

single decision via an object-level saliency map by leveraging the pixel-level explanation and object detection. As these methods focus on single decisions, the explanation is typically not sufficient to understand the overall decision-making of the trained agent.

Other work has also attempted to explain entire policies, rather than individual decisions. Ehsan et al. [8] produce natural language explanations for state-action pairs, based on a human-provided corpus of explanations. Topin and Veloso [28] create a markov chain which acts as an abstraction of the policy, making it easier to reason about the policy. They create the markov chain by grouping states into abstract states based on how similarly the policy acts in different states; our method is substantially different in that it ranks states based on importance, rather than grouping similar states. While our method allows for the use of abstract states, it is not always necessary, and we are not contributing any specific abstraction function. The graphs are also only shown for relatively simply environments, and not tested on environments as complex as Atari games. Similarly, Sreedharan et al. [25] propose a method for creating a temporal abstraction of a policy by using bottleneck 'landmarks' in the policy's executions. A robot's behavior can be explained using operator-specified "important program state variables" and "important functions" [12]. We find that the policy-wide decision ranking in this paper is an easier and more general method for understanding the policy. For more work in this vein, we encourage the reader to consult the overview from Chakraborti et al. [6] of the rapidly growing field of Explainable AI Planning (XAIP).

There have been attempts to make more interpretable models, either from scratch [13], or by approximating a trained neural network [29]. In the latter case, our method may be useful for determining in which states the approximation must be most accurate.

6 Conclusions

We have applied SBFL-based ranking of states to reinforcement learning and demonstrated that this ranking correlates with the relative importance of states for the policy's performance. The ranking can be used to explain RL policies, similarly to the way ranked program locations are used to understand the causes of a bug. We evaluate the quality of our ranking by constructing simpler pruned policies, where only the most important decisions are made according to the policy, and the rest are default. Our experiments show that pruned policies have a comparable performance to the original policy, thus supporting our claim of the accuracy of SBFL-based ranking. Moreover, the pruned policies can be preferred by the users, as they are simpler. Our approach can be scaled with the use of abstractions, as demonstrated by the larger Atari environments.

In the future, we hope to explore different applications of the ranking, new measures, more nuanced hyperparameter selection, and relaxing the binary constraint over the assertion C. We do not expect our work to have any negative societal impacts, as it only serves to improve our understanding of the policies we train; the main concern is incorrectly increasing confidence in our policies.

References

[1] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. On the accuracy of spectrum-based fault localization. In *TAICPART-MUTATION*, pages 89–98. IEEE, 2007.

[2] Rui Abreu, Peter Zoeteweij, and Arjan J. C. Van Gemund. A new bayesian approach to multiple intermittent fault diagnosis. In *IJCAI*, pages 653–658, 2009. URL http://ijcai.org/Proceedings/09/Papers/114.pdf.

[3] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan JC Van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 82(11):1780–1792, 2009.

[4] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI gym. *CoRR*, abs/1606.01540, 2016. URL https://gym.openai.com.

[5] Nuno Cardoso, Rui Abreu, Alexander Feldman, and Johan de Kleer. A framework for automatic debugging of functional and degradation failures. In *ECAI*, pages 569–576. IOS Press, 2016. doi: 10.3233/978-1-61499-672-9-569. URL https://doi.org/10.3233/978-1-61499-672-9-569.

[6] Tathagata Chakraborti, Sarath Sreedharan, and Subbarao Kambhampati. The emerging landscape of explainable automated planning & decision making. In Christian Bessiere, editor, *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, pages 4803–4811. ijcai.org, 2020. doi: 10.24963/ijcai.2020/669. URL https://doi.org/10.24963/ijcai.2020/669.

[7] Maxime ChevalierBoisvert, Lucas Willems, and Suman Pal. Minigrid, 2018.

[8] Upol Ehsan, Brent Harrison, Larry Chan, and Mark O Riedl. Rationalization: A neural machine translation approach to generating natural language explanations. In *AI, Ethics, and Society*, pages 81–87, 2018.

[9] Alberto Gonzalez-Sanchez. Automatic error detection techniques based on dynamic invariants. *M.S. Thesis, Delft University of Technology*, 2007.

[10] Samuel Greydanus, Anurag Koul, Jonathan Dodge, and Alan Fern. Visualizing and understanding Atari agents. In *ICML*, page 1792, 2018.

[11] David Gunning and David W Aha. DARPA's explainable artificial intelligence program. *AI Magazine*, 40(2):44–58, 2019.

[12] Bradley Hayes and Julie A Shah. Improving robot controller transparency through autonomous policy explanation. In *HRI*, pages 303–312, 2017.

[13] Daniel Hein, Steffen Udluft, and Thomas A Runkler. Interpretable policies for reinforcement learning by genetic programming. *Engineering Applications of Artificial Intelligence*, 76: 158–169, 2018.

[14] Rahul Iyer, Yuezhang Li, Huao Li, Michael Lewis, Ramitha Sundar, and Katia Sycara. Transparency and explanation in deep reinforcement learning neural networks. In *AI, Ethics, and Society*, pages 144–150. ACM, 2018.

[15] James A Jones and Mary Jean Harrold. Empirical evaluation of the Tarantula automatic fault-localization technique. In *ASE*, pages 273–282. ACM, 2005.

[16] Cheol Kang. g6ling/Reinforcement-Learning-Pytorch-Cartpole, July 2020. URL https://github.com/g6ling/Reinforcement-Learning-Pytorch-Cartpole.

[17] Michael Lewis, Huao Li, and Katia Sycara. Deep learning, transparency, and trust in human robot teamwork. In *Trust in Human-Robot Interaction*, pages 321–352. Elsevier, 2021.

[18] Dor Livne and Kobi Cohen. PoPS: Policy pruning and shrinking for deep reinforcement learning. *IEEE Selected Topics in Signal Processing*, 2020.

[19] Lucia, David Lo, Lingxiao Jiang, Ferdian Thung, and Aditya Budi. Extended comprehensive study of association measures for fault localization. *Journal of software: Evolution and Process*, 26(2):172–219, 2014.

[20] Scott M Lundberg and Su-In Lee. A unified approach to interpreting model predictions. In *NIPS*, pages 4765–4774. Curran Associates, 2017.

[21] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. A model for spectra-based software diagnosis. *ACM TOSEM*, 20(3):11, 2011.

[22] Chris Parnin and Alessandro Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of ISSTA*, pages 199–209. ACM, 2011. doi: 10.1145/2001420.2001445. URL https://doi.org/10.1145/2001420.2001445.

[23] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. Why should I trust you? Explaining the predictions of any classifier. In *KDD*, pages 1135–1144, 2016.

[24] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. In *ICLR*, 2016.

[25] Sarath Sreedharan, Siddharth Srivastava, and Subbarao Kambhampati. Tldr: Policy summarization for factored SSP problems using temporal abstractions. In J. Christopher Beck, Olivier Buffet, Jörg Hoffmann, Erez Karpas, and Shirin Sohrabi, editors, *Proceedings of the Thirtieth International Conference on Automated Planning and Scheduling, Nancy, France, October 26-30, 2020*, pages 272–280. AAAI Press, 2020. URL https://aaai.org/ojs/index.php/ICAPS/article/view/6671.

[26] Felipe Petroski Such, Vashisht Madhavan, Rosanne Liu, Rui Wang, Pablo Samuel Castro, Yulun Li, Jiale Zhi, Ludwig Schubert, Marc G. Bellemare, Jeff Clune, and Joel Lehman. An Atari model zoo for analyzing, visualizing, and comparing deep reinforcement learning agents. In *IJCAI*, pages 3260–3267, 2019. doi: 10.24963/ijcai.2019/452. URL https://doi.org/10.24963/ijcai.2019/452.

[27] Youcheng Sun, Hana Chockler, Xiaowei Huang, and Daniel Kroening. Explaining image classifiers using statistical fault localization. In *ECCV*, LNCS. Springer, 2020.

[28] Nicholay Topin and Manuela Veloso. Generation of policy-level explanations for reinforcement learning. In *AAAI*, volume 33, pages 2514–2521, 2019.

[29] Abhinav Verma, Vijayaraghavan Murali, Rishabh Singh, Pushmeet Kohli, and Swarat Chaudhuri. Programmatically interpretable reinforcement learning. In *ICML*, pages 5045–5054, 2018.

[30] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. Dueling network architectures for deep reinforcement learning. In *ICML*, pages 1995–2003, 2016.

[31] Lucas Willems. lcswillems/rl-starter-files, August 2020. URL https://github.com/lcswillems/rl-starter-files. original-date: 2018-04-11T14:47:48Z.

[32] W Eric Wong, Yu Qi, Lei Zhao, and Kai-Yuan Cai. Effective fault localization using code coverage. In *COMPSAC*, volume 1, page 449, 2007.

[33] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE TSE*, 42(8):707–740, 2016.

[34] Xin Xia, Lingfeng Bao, David Lo, and Shanping Li. Automated debugging considered harmful: A user study revisiting the usefulness of spectra-based fault localization techniques with professionals using real bugs from large systems. In *ICSME*, pages 267–278. IEEE, 2016.

[35] Tom Zahavy, Nir Ben-Zrihem, and Shie Mannor. Graying the black box: Understanding DQNs. In *ICML*, pages 1899–1908, 2016.

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes]
   (c) Did you discuss any potential negative societal impacts of your work? [Yes]
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [N/A]
   (b) Did you include complete proofs of all theoretical results? [N/A]
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes]
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes]
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes]
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes]
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes]
   (b) Did you mention the license of the assets? [Yes]
   (c) Did you include any new assets either in the supplemental material or as a URL? [Yes]
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A]
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A]
5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]

```python
def cartpole_abs_func(n=-1):
    """
        Num Observation                 Min         Max
    0   Cart Position             -4.8            4.8
    1   Cart Velocity             -Inf            Inf
    2   Pole Angle                 -24 deg        24 deg
    3   Pole Velocity At Tip      -Inf            Inf
    """
    return lambda state: str([
        abs(round(state[0])),
        abs(round(state[1], 1)),
        abs(round(state[2]/4, 2)),
        abs(round(state[3], 1))])
```

(a) CartPole abstraction function

```python
def basic_abs(state):
    size = (18, 14)
    max_pix = 8
    gray = cv2.cvtColor(state, cv2.COLOR_RGB2GRAY)
    small = cv2.resize(gray, size, interpolation=cv2.INTER_AREA)
    round_pix = ((small / 255.0) * max_pix).astype(np.uint8)
    return str(round_pix) if ret_str else round_pix
```

(b) Atari Games generic abstraction function

Figure 4: Abstraction Functions in Python. "abs" is the absolute value, and "round" rounds the input to the number of decimal points given.

A Experiment Setup

Code for experiments can be found at the anonymous github repository: https://github.com/AnonUser-532438/PolicyRankingAnon.

A.1 Environments

MiniGrid We use the SimpleCrossingS9N1-v0 variant of the Minimalistic Gridworld Environment (MiniGrid) [7]. The environment always consists of a 7×7 grid in which the agent can navigate, with a wall separating the agent's starting position from the goal. There is always on hole in the wall. At each episode, the position of the wall and hole are randomized. The agent is rewarded for reaching the goal, and this reward is linearly annealed from 1 to 0.1 by the final step. After 322 steps, the episode ends. The state information given the agent is a cone of vision, allowing a single agent to

CartPole We use OpenAI's 'Gym' [4] implementation of the CartPole environment. The agent may push the cart left or right to balance the pole, and gains a reward of 1 at each step. If the pole falls too far to the side, or the cart moves too far to the side, the episode is terminated. The episode ends after 200 steps.

Atari Games To implement the Atari games, we use OpenAI's 'Gym' [4] package. We artificially limit the length of each episode to 600 steps to reduce running time of analyses.

A.2 Abstractions

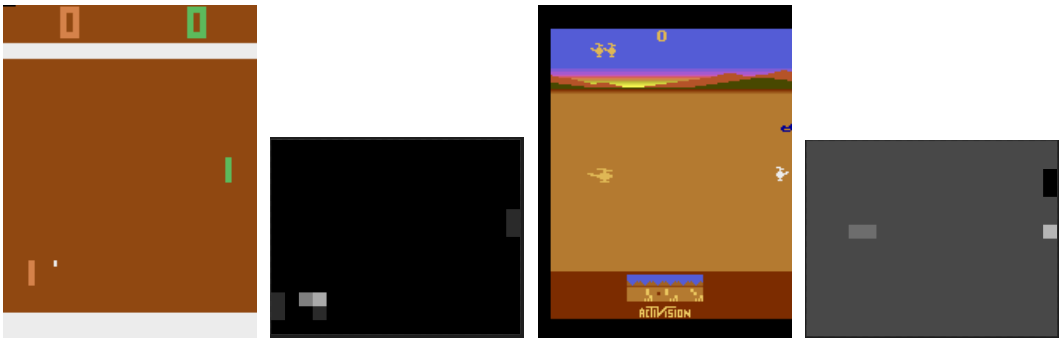
The functions used for abstracting the state space are shown in Figure 4.

MiniGrid No abstraction function is used in Minigrid experiments.

CartPole The state space is continuous in CartPole, and we discretise it by rounding each element of the state. The pole angle is first divided by 4, to further reduce the space. We finally take the absolute value to exploit CartPole's symmetry. It is worth noting that this would not work with any default action. For example, if the default action were 'move right', then states where the pole is leaning left (and moving right would make it fall over) would be more important than states where the pole is leaning right (and moving right would straighten it). We repeat the last action as our default action, and this treats both side symmetrically.

Atari Games For Atari games, the state is cropped to only the game area, and important parts of the state (e.g. character) are made bigger (so that they appear even in the abstract state) before applying the abstraction function. The abstraction function grayscales, downsamples to 18×14 pixels, and then forces the pixels to take integer values between 0 and 8. Examples of the Atari games abstract states are shown in Figure 5.

In the specialised Breakout abstraction, the pixel coordinates of the ball and paddle are isolated and rounded to the nearest 10.



(a) Pong before abstraction (b) Pong after abstraction (c) Chopper Command before abstraction (d) Chopper Command after abstraction

Figure 5: Atari Games Abstractions

Table 2: Expected reward for trained policy and random behaviour in each Atari game, constrained to 600 steps.

Game	Policy Reward	Random Reward
Alien	1500.0	178.0
Assault	929.1	214.2
Asteroids	301.0	564.0
Atlantis	14170.0	7950.0
BankHeist	553.0	12.0
BattleZone	12300.0	1400.0
Berzerk	811.0	140.0
Boxing	97.8	-0.1
Breakout	19.6	1.4
ChopperCommand	3510.0	600.0
DemonAttack	133.5	91.0
Hero	13414.0	272.0
IceHockey	0.2	-2.0
Jamesbond	290.0	15.0
Krull	1159.1	444.0
Phoenix	1766.0	390.0
Pong	6.7	-13.1
Qbert	7620.0	102.5
Riverraid	5485.0	1268.0
RoadRunner	30020.0	20.0
Seaquest	598.0	74.0
SpaceInvaders	724.5	164.5
StarGunner	10000.0	330.0
Tennis	-1.0	-8.3
YarsRevenge	21008.6	2304.6

A.3 Policies

Minigrid For MiniGrid, the policy was trained using code from the "RL Starter Files" github repository [31], and achieves an expected reward of 0.88.

CartPole For CartPole, the policy was trained using code from the "PyTorch CartPole Example" github repository [16], with expected reward 200. In our experiment including a low-performing policy, we halt training early to produce a policy with expected reward 97.

Atari Games For Atari games, policies were taken from Uber's Atari Zoo package [26]. Performance for each game is shown in Table 2.

Table 3: Hyperparameters for each enviroment.

Environment	$\lvert\mathcal{T}(\pi)\rvert$	μ	C	n_{test}
MiniGrid	5000	0.2	reward $\geq$ 0.8	100
CartPole	5000	0.4	reward $\geq$ 200	100
Alien	1000	0.2	reward $\geq$ 90.0	50
Assault	1000	0.2	reward $\geq$ 1.0	50
Atlantis	1000	0.2	reward $\geq$ 400.0	50
BankHeist	1000	0.2	reward $\geq$ 0.0	50
BattleZone	1000	0.2	reward $\geq$ 0.0	50
Berzerk	1000	0.2	reward $\geq$ 0.0	50
Boxing	1000	0.2	reward $\geq$ 0.0	50
Breakout	1000	0.2	reward $\geq$ 0.0	50
ChopperCommand	1000	0.2	reward $\geq$ 0.0	50
DemonAttack	1000	0.2	reward $\geq$ 0.0	50
Hero	1000	0.2	reward $\geq$ 910.0	50
IceHockey	1000	0.2	reward $\geq$ 0.0	50
Jamesbond	1000	0.2	reward $\geq$ 0.0	50
Krull	1000	0.2	reward $\geq$ 21.5	50
Phoenix	1000	0.2	reward $\geq$ 0.0	50
Pong	1000	0.2	reward $\geq$ 0.0	50
Qbert	1000	0.2	reward $\geq$ 75.0	50
Riverraid	1000	0.2	reward $\geq$ 80.0	50
RoadRunner	1000	0.2	reward $\geq$ 600.0	50
Seaquest	1000	0.2	reward $\geq$ 0.0	50
SpaceInvaders	1000	0.2	reward $\geq$ 0.0	50
StarGunner	1000	0.2	reward $\geq$ 0.0	50
YarsRevenge	1000	0.2	reward $\geq$ 176.0	50

A.4 Hyperparameters

The default action was "repeat the previous action" for all experiments. Table 3 shows all the other hyperparameters for the experiments. We also present n_{test}, the number of episodes over which each pruned policy is evaluated when performing experiments with pruned policies.

Typically, $|\mathcal{T}(\pi)|$ and n_{test} were chosen to be as high as possible within reasonable running times.

Minigrid and Atari Games In Minigrid and Atari games, $\mu = 0.2$ was empirically found to be a good balance between mutating enough states to gain information, and not mutating so much as to destroy the policy. C was chosen to balance the test suite between success and failure.

CartPole In CartPole, $C =$ reward ≥ 200 was chosen as it corresponds to checking whether the pole ever fell, or the cart went out of bounds. μ was set to balance the test suite.

A.5 Computational Resources

Our method does not require any re-training, an often expensive procedure. The primary cost is performing executions of the agent through the environment. To calculate the rankings, this means doing as many executions as are in the test suite. In our experiments, this was 500 for Atari games, which is far less than most training algorithms. The time taken to compute each rankings for each of the environments is shown in Table 4. For all of our experiments, we used an Intel(R) Xeon(R) Silver 4114 CPU @ 2.20GHz, with 188GB RAM, and a 16GB Tesla V100 GPU.

Producing the test suite takes time linear in the size of the test suite, and computing the ranking require sorting the set of all of the visited states S, which is $O(|S|log|S|)$.

Table 4: Average time to compute ranking in each environment.

Environment	Time (mm.ss)
MiniGrid	28.40
CartPole	09.40
Alien	34.51
Assault	34.41
Atlantis	32.50
BankHeist	31.54
BattleZone	34.52
Berzerk	21.35
Boxing	35.37
Breakout	31.19
ChopperCommand	33.08
DemonAttack	31.11
Hero	35.08
IceHockey	34.27
Jamesbond	32.45
Krull	28.20
Phoenix	31.14
Pong	31.41
Qbert	29.48
Riverraid	33.46
RoadRunner	34.37
Seaquest	31.18
SpaceInvaders	31.29
StarGunner	32.03
YarsRevenge	31.04

B Results

Additional results, including heatmaps and graphs showing each measure individually, can be found at https://sites.google.com/view/ranking-policy-decisions/home.

B.1 More Results from Policy Pruning

Table 5 is complementary to Table 1, and shows how many states were restored, or in how many states the policy was used over the default action, in order to reach 50% of the original policy's performance.

B.2 Producing pruned policy graphs

We evaluate the pruned policies as we progressively increase the number of states in which we take the original policy over the default action. We then plot the performance of the evaluated policies, with respect to either the percentage of states restored, or the percentage of steps in which the original policy is used.

Pruned policies do not improve monotonically Because of the complex interactions between the states, the policies do not monotonically improve as states are restored. For example, restoring a state could send the agent down a new trajectory in which other states have not yet been restored, and so the agent may fail. As a result, we show in each point the performance achievable for that percentage (of either states restored or policy's actions taken) *or less*. Suppose a policy q recovers 60% of the original policy's reward after having restored 35% of states. If there exists another policy which recovers 70% of reward with only 30% of states, then it is strictly better, and we should use this policy over q. As a result, we report having recovered 70% of the reward at both 30% and 35% of restored states.

Table 5: Minimum percentage of original states in pruned policies, and percentage of steps in which the original policy is used, before recovering 50% of original performance. Results are reported for SBFL portfolio ranking and (random) ranking. "X" denotes that the required reward was never reached, cf. Sec. 4.2.

Environment	% of original states restored in policy		% of steps that use π over default action	
MiniGrid	**35** ± 00	(85 ± 00)	**49** ± 00	(66 ± 04)
Cartpole	**6** ± 04	(46 ± 01)	**13** ± 01	(47) ± 01
Alien	X $\pm X$	(X $\pm X$)	X $\pm X$	(X $\pm X$)
Assault	**22** ± 04	(54 ± 05)	57 ± 10	(**51** ± 05)
Atlantis	**20** ± 00	(90 ± 00)	**87** ± 00	(92 ± 01)
BankHeist	**35** ± 05	(91 ± 03)	**62** ± 03	(83 ± 06)
BattleZone	**30** ± 00	(52 ± 04)	76 ± 01	(**42** ± 05)
Berzerk	**10** ± 00	(88 ± 03)	**74** ± 10	(80 ± 06)
Boxing	X $\pm X$	(X $\pm X$)	X $\pm X$	(X $\pm X$)
Breakout	**10** ± 00	(82 ± 04)	**54** ± 00	(67 ± 07)
Breakout (abs)	**40** ± 00	(61 ± 03)	**41** ± 00	(49 ± 05)
ChpperCmmnd	**23** ± 05	(83 ± 06)	**80** ± 05	(80 ± 10)
DemonAttack	**20** ± 00	(90 ± 05)	95 ± 06	(**84** ± 08)
Hero	**32** ± 05	(68 ± 04)	**45** ± 06	(58 ± 06)
IceHockey	**20** ± 04	(74 ± 05)	**66** ± 04	(67 ± 05)
Jamesbond	**20** ± 00	(40 ± 00)	**28** ± 07	(38 ± 01)
Krull	**19** ± 03	(64 ± 05)	**20** ± 05	(57 ± 05)
Phoenix	**20** ± 00	(78 ± 06)	**50** ± 00	(74 ± 07)
Pong	**10** ± 00	(50 ± 00)	**33** ± 02	(42 ± 01)
Qbert	**30** ± 00	(89 ± 03)	**78** ± 01	(78 ± 04)
Riverraid	**38** ± 04	(85 ± 05)	82 ± 02	(**80** ± 06)
RoadRunner	**28** ± 04	(85 ± 05)	**77** ± 04	(79 ± 07)
Seaquest	**30** ± 01	(70 ± 01)	**55** ± 09	(56 ± 02)
SpaceInvaders	**22** ± 04	(81 ± 03)	**63** ± 15	(75 ± 04)
StarGunner	**20** ± 00	(70 ± 00)	71 ± 13	(**69** ± 02)
YarsRevenge	**27** ± 04	(82 ± 04)	**65** ± 04	(70 ± 06)

SBFL portfolio ranking We present the SBFL portfolio ranking as the combination of the different SBFL measures presented. As we are free to use any of the measures, we use the best measure in each point. The SBFL curves show the best performance for any of the measures in each point.

Normalising scores For Minigrid and CartPole, the minimum score is 0, and so we simply show what percentage the pruned policy achieves of the original policy's reward. In Atari games, it is common to normalise reward with respect to the reward of a random policy. We use the figures in Table 2 to compute $100\times$(pruned score - random score) / (original score - random score).

B.3 Low-performance policies on CartPole

Figure 6 expands on results for **RQ4**. We use the rankings based on scoring states of the good policy using FreqVis and Tarantula. For each of these two rankings, we are able to look at the agreement between the two policies for the top X% of states, as X increases. The overall agreement in all states considered is about 64%. The results show that the policies disagree greatly in the 10% most frequently visited states, but beyond this frequency of visit is a poor predictor of agreement. On the other hand, we find that the policies agree in most top-ranked tarantula states.

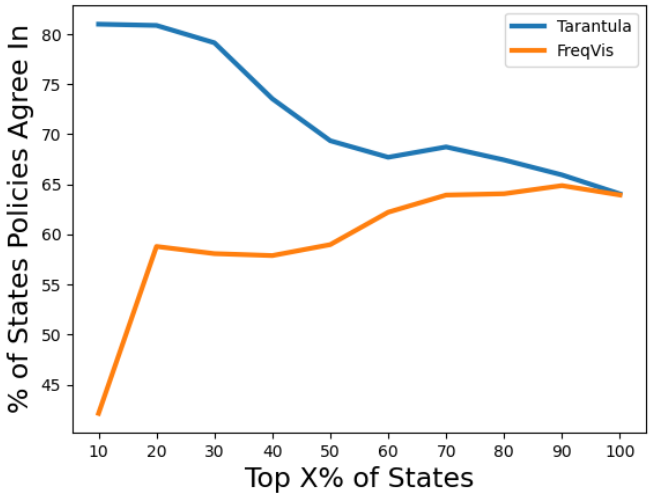


Figure 6: In CartPole, of the top X% of states ranked by either Tarantula or FreqVis, the percentage of agreement between a low and high-quality policy. Data taken from 1,000 runs through the environment.