# Smart EfficientZero Search for Atari

Stanford CS 224R: Deep Reinforcement Learning

**Matthew Harvill**
Department of Computer Science
Stanford University
mharvill@stanford.edu

## Abstract

For this project we investigate two main strategies for improving the performance of trained EfficientZero agents on three games in the Atari 100k benchmark. We first try using learned state representations and policy estimates to share Monte Carlo Tree Search (MCTS) visit information between similar states to improve agent performance in evaluation. Specifically, our implementation(s) measure state similarity based on combinations of cosine similarity of state representations and mean squared differences in normalized policies. According to our metric(s), we share similar states' visit information by merging their tree nodes into larger group nodes. We find that this method of sharing MCTS visit information between similar states significantly increases evaluation times without increasing agent performance, making it an impractical improvement strategy. For our second method we make two important observations. First, we notice that the tree traversal in state of the art (SOTA) AlphaZero-based systems is orders of magnitude faster than the policy and value networks' inference times. Second, since a single MCTS simulation proceeds until reaching an unvisited state, conducting multiple tree traversals in parallel allows us to explore not only wider, but also deeper in the search tree since we accumulate a larger pool of visited states. Indeed, we find that searching multiple trajectories at once improves agent performance by up to 35% on two of three Atari games, while increasing evaluation time by as low as 10%. Our implementation selects up to $N$ paths by branching on the top two actions at each visited state if their Predictive Upper Confidence Bound for Trees (PUCT) scores are within a small margin, essentially ensuring that each path will still contribute valuable information to the MCTS search in terms of exploration and exploitation. Unlike our state similarity-based method, this method improves performance, and it does not require well-trained representation and policy networks. This second feature makes our multiple path MCTS evaluation a viable strategy to use during training for any AlphaZero-based system, which we believe is an exciting line of future work given the success and widespread adoption of these systems.

# 1 Background

Using Monte Carlo Tree Search (MCTS) as a policy improvement operator in Reinforcement Learning (RL) was first introduced in AlphaZero Silver et al. (2017) for playing Chess and Go tabula rasa, but is still widely used due to its effectiveness. AlphaZero-based algorithms work by running MCTS simulations in each state $s_t$ and then selecting action $a_t$ according to the most visited action at the root state, $s_t$ according to the MCTS simulations. This process proceeds as shown in 1(a) for $T$ steps, or until a terminal state is reached. After all $T$ actions have been taken, the rewards and categorical policies gathered from 1(a) are backpropagated such that the value network is trained against the actual value of $s_T$ and a divergence loss is computed between the policy network and categorical visit count distribution. Thus, as described, MCTS is used as the policy improvement operator.

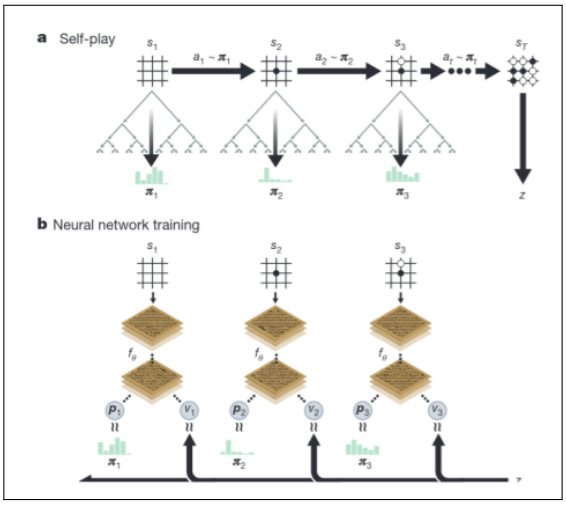



Figure 1: (a) AlphaZero simulation rollout.
(b) AlphaZero rollout statistics backpropagation.

One important detail to this process is that actions are selected according to PUCT scores from the following formula:

$$a^k = \arg\max_a \left\{ Q(s,a) + P(s,a) \frac{\sqrt{\sum_b N(s,b)}}{1 + N(s,a)} \left( c_1 + \log \left( \frac{\sum_b N(s,b) + c_2 + 1}{c_2} \right) \right) \right\}$$

where these scores balance exploration $N(s,a)$: the number of times action $a$ is taken in state $s$, and exploitation: $Q(s,a)$, the empirical Q-value, with an initial bias: $P(s,a)$, the policy network's estimate. Also constant values $c_1 = 1.25$ and $c_2 = 19,652$ are used according to the literature, and random Dirichlet noise is added to root action policy estimates in MCTS simulations to force exploration.

This process pertains to all AlphaZero-based systems, but since AlphaZero is closed-source and not designed for visual observations, this project builds on EfficientZero Ye et al. (2021), an open-source model-based variant of AlphaZero that achieves SOTA performance on the Atari 100k benchmark. EfficientZero is built on MuZero Reanalyze Schrittwieser et al. (2019), a model-based version of AlphaZero that corrects older off policy data to perform better in data-limited settings. EfficientZero also uses a self-supervised consistency loss between $s_{t+1}$, representations of the next observation, and $\hat{s}_{t+1}$, representations of the current observation passed through the dynamics network. While this project focuses on EfficientZero's search, these details still help to understand how EfficientZero, the backbone of this project, works.

## 2 Introduction

The MCTS process is vital to the performance of AlphaZero-based systems for two main reasons. First, it is used as the policy improvement operator, so the richer the information it provides, the quicker the policy will improve. Second, even with a fully trained AlphaZero-style system, it has been shown that performance can still be greatly improved by augmenting the trained policy network with MCTS simulations Silver et al. (2017). Given the importance of AlphaZero (and EfficientZero)'s MCTS process, this project attempts to improve EfficientZero's search in evaluation by leveraging the information learned by the state representation network.

Although AlphaZero's MCTS process is guided by the policy network implicitly with PUCT scores, in a set of simulations for some state, $s_t$, the MCTS visit information is still collected naively (similar states must be visited individually from scratch in the same search). While slightly different state-action pairs can lead to drastically different rewards, we believe that a sufficiently trained policy/value network can inform state representations to effectively predict similar state-action pairs, which is why we will only start sharing MCTS visit information after the policy/value network has achieved acceptable performance. Furthermore, we believe that RL problems with visual-based state spaces, like Atari games, will benefit greatly from this modification since they are likely to contain many similar states with the same optimal action responses.

In our early experiments we find that not only does leveraging state similarity information in MCTS degrade performance, but it also slows down evaluation by up to 20x. Given these issues and the fact that the search part of MCTS evaluation is roughly 110x quicker than model inference (getting new state representations, policy estimates, and value estimates), we decide to try different methods centered around the idea of evaluating multiple paths of the search tree in parallel. Like our similarity-based MCTS improvement methods, these methods attack our original objective of improving EfficientZero's MCTS process and prove to be much more successful.

## 3 Related Works

As discussed in 1, many variants of AlphaZero have emerged over the past few years including MuZero, EfficientZero, Sampled MuZero, GumbelZero, and more (Ye et al., 2021; Schrittwieser et al., 2019; Hubert et al., 2021; Danihelka et al., 2022). While MuZero expands on AlphaZero by learning a dynamics model and reward model of the environment, and EfficientZero expands on MuZero as mentioned in 1, Sampled MuZero improves on MuZero for problems with large discrete or continuous action spaces, by sampling from the policy distribution, with policy improvement guarantees. Also, GumbelZero (Danihelka et al., 2022) attempts to minimize the search process, and performs well when the number of MCTS rollouts is constrained. However, EfficientZero is not concerned with constrained simulations counts, and GumbelZero is not able to outperform AlphaZero in an unconstrained setting. While all of these methods improve on AlphaZero to address different issues, none of them take advantage of state-action similarities when performing expensive MCTS rollouts.

However, Swiechowski et al. (2018) modifies AlphaZero for playing HearthStone, a complex multiplayer game with partially-hidden states. Their modifications are relevant and interesting with respect to this project because they combine symmetric states into single nodes in the search tree. This method is similar to our proposed method based on state similarities except that our method combines similar states instead of identical symmetric states.

With respect to our second group of methods focused on evaluating multiple MCTS search paths in parallel, GumbelZero is specifically interesting to this project because it is also a SOTA system developed at DeepMind by some of the same members who designed AlphaZero, but unlike AlphaZero, it is open source. Therefore, after inspecting its source code, we confirm that current SOTA AlphaZero-based systems do not evaluate multiple MCTS searches in parallel, making our experiments relevant and exciting.

# 4 Methods

## 4.1 State Similarity-based Visit Information Sharing

The process for our state similarity-based methods of sharing MCTS visit information is depicted in 2. As shown, we see that similar states that would have separate nodes in the search tree by default (with disjoint visit information) are combined into a single node with shared visit information using our similarity-based methods. Since we are combining two states in the search tree, we ensure that the action probabilities (policy predictions) at each state are extremely similar for all of our similarity-based methods. Since this must be met in order to combine two states, we elect to keep the original policy predictions from the original similar state and use them for the group node. While there are other potential ways of sharing MCTS visit information between similar states, this method is similar to Swiechowski et al. (2018) and is straightforward to implement.

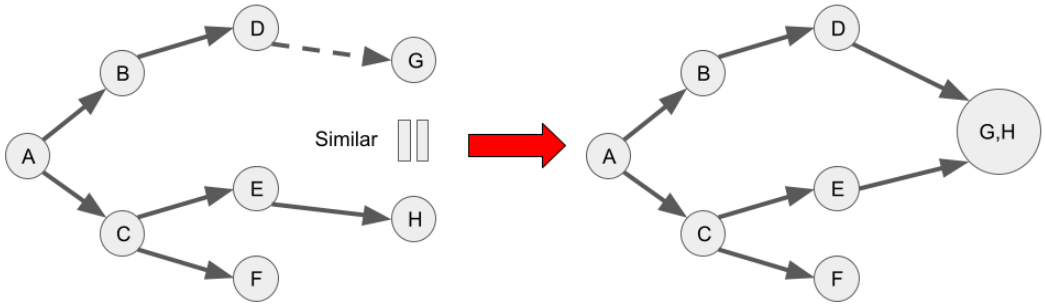



Figure 2: Example of visit information sharing process for similar states G and H.

The actual process for determining if two states are similar is done incrementally with each new simulation in a set of $N$ MCTS simulations. Once a MCTS simulation completes (an unvisited state is reached in the current set of $N$ simulations), this method compares the new state to the other $N - 1$ states from previous simulations ($N = 50$ by default). Therefore this method has $O(N^2)$ time complexity. Although $N$ is not prohibitively large, it is still large enough to cause noticeable slowdowns for these methods, as detailed later in 5.

Note that the only difference between 4.1.1 and 4.1.2 below is how state similarity is determined, while the process in 2 is identical.

### 4.1.1 Policy Similarity

For this method, our metric for determining similar states is based on the mean squared error (MSE) between the normalized policy predictions from each state. Specifically, two states are considered similar if the mean squared error between their normalized policy predictions is under $1e^{-5}$. Notice that we do not expect this method to perform well because it is easily possible that widely differing states have similar policy predictions. However, it is a nice baseline.

### 4.1.2 Policy and State Similarity

For this method, our metric for determining similar states uses the same MSE between normalized policy predictions as in 4.1.1 in conjunction with a cosine similarity between state representations that must be above $0.97$. Since the state representations for each observation are tensors of size $64$x$6$x$6$, this method is noticeably more computationally expensive than 4.1.1. However, it additionally incorporates state representation information, making it a much more viable method.

## 4.2 Parallel Path Selection

Our parallel MCTS search methods are illustrated in 3 and 4. Note that for each potential branching, only the top two actions are considered in any of our methods, but this is done recursively until $N$ desired branches are formed, from which the search continues until an unvisited state is reached. If

unvisited states are reached before $N$ branches can be formed, we do not attempt to search more branches. Instead we just proceed as we would if all $N$ had been searched. Also, as shown in solid blue in 3 and 4, the original best path from the default single path MCTS search is still visited.

#### 4.2.1 Naive Branching

Our first attempt is just a naive branching method that creates a new branch based on the second best action in a queue-like order starting from the root node until $N$ desired branches are formed. The PUCT scores of the actions are not considered except in determining the top two actions. For our example, this leads to the green branches in 3.

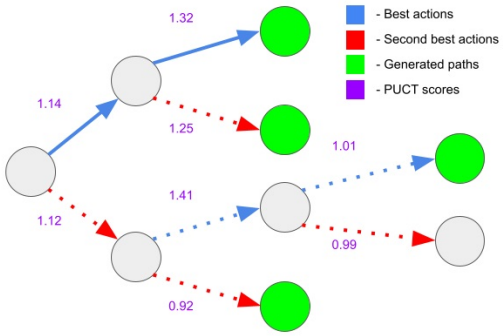



Figure 3: Example of naive MCTS expansion with $N = 4$ paths.

#### 4.2.2 Restricting Maximum Depth

In our early experiments we find that often times increasing the number of branches has diminishing returns. We hypothesize that this is related to compounding errors from the learned dynamics and reward models as search depth increases (which happens with a larger $N$). Therefore, for this method we end a simulation path once it reaches length 10, searching with the naive branching method from above. We decide to use a max depth of 10 because it is an empirical upper limit for the default method of using only one path. This method seems to have little to no effect on performance, prompting our next method.

#### 4.2.3 Backpropagating Single Best Path

Since MCTS is only meant to backpropagate the best actions according to PUCT scores, we also evaluate our naive branching method to visit multiple branches, but only backpropagate the visit statistics from the top branch (corresponding to the solid blue path in 3 and 4). We implement this method because it simplifies to the exact same updates as in the default MCTS search except that by searching multiple branches in parallel we can more efficiently use our GPU resources by evaluating more state representations. This allows us to search deeper into the tree (since we are more likely to have already evaluated nodes in the tree when we reach them), which should provide richer searches if our environment model is accurate.

#### 4.2.4 Dynamic Branching

The last method that we implement is similar to our naive method except that we do not always branch on the top two actions. Instead, we only branch if the difference in PUCT scores between the top two actions is $< 0.1$. Since PUCT scores with EfficientZero's choice of constants and setup range from $(0, 2)$, this is a relatively small margin. As shown in 4, for the same exact search tree as in 3, we now end up visiting different branches, since the difference in PUCT scores for the bottom-most nodes is too great.

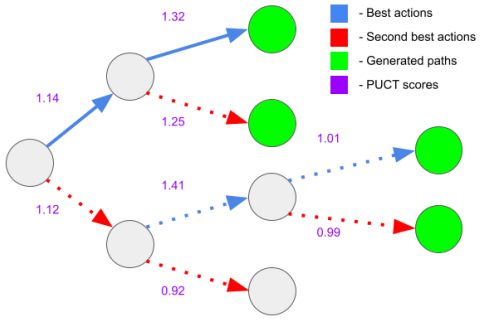
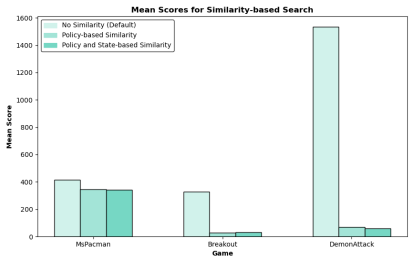



Figure 4: Example of dynamic MCTS expansion with $N = 4$ paths using a PUCT Margin of $\Delta = 0.1$ between top two actions.

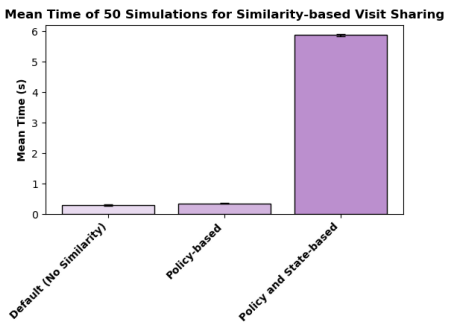
# 5 Results

## 5.1 State Similarity-based Visit Information Sharing

As shown in 5(a), our methods based on similar states perform worse than default across the board. We have two hypotheses for why this is the case. First, in determining our cutoffs for classifying two states as similar, we loosen the requirements since our earlier values average $< 1$ similar states per $N = 50$ simulations. By loosening our values, we achieve an average similar state count of 1 per 50 simulations. Even though our results are poor, these findings are interesting because they disprove our initial belief that there will be many similar states in a set of simulations. In addition to finding that there are not very many similar states, we also believe there are more sophisticated ways of sharing MCTS visit information. However, other methods that do not merge nodes in the tree require more overhead for updating multiple nodes at once. We decide not to implement one of these methods due to the fact that even if they perform well, the increased time spent determining similarities make them infeasible. The time slowdown (using a state-based similarity) is shown in 5(b). This 20x slowdown is for our fastest (vectorized) implementation of computing cosine similarity between states.




(a) State Similarity-based Visit Information Sharing Performance on MsPacman, Breakout, and DemonAttack




(b) State Similarity-based Time Comparison

Figure 5: Mean agent scores and time slowdowns across similarity-based MCTS visit information sharing methods.

Although these methods do not lead to better performance, they lead us to our second set of methods from the realization that we can perform more simulations per model inference. We also learn that there are not as many similar states as we initially believe, which we might be due to the representation network's improved ability at differentiating seemingly similar observations.

### 5.2 Parallel Path Selection

Before looking at the performance of our parallel path selection methods, we first look at the slowdown incurred. Since slowdown is an important drawback to our similarity-based methods, we highlight here in 6 that the slowdown incurred for searching more branches is minor even for up to 8x as many searches, making these methods a promising approach.

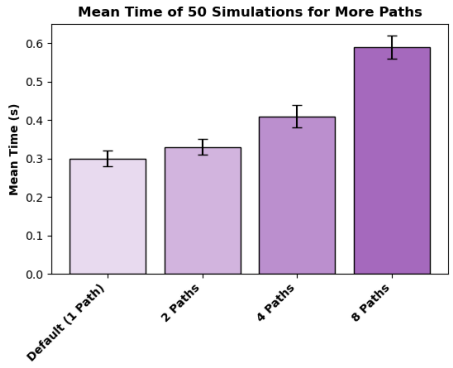



Figure 6: Clock times across maximum number of parallel paths searched.

#### 5.2.1 Naive Branching

Shown in 7 below, for MsPacman, naive branching increases performance with 8 branches, but decreases with 2 branches. Also, for Breakout, it performs significantly worse, and for DemonAttack it only increases performance for two branches. We notice that the high variance of performance with naive branching prevents us from extract a definitive answer to its effectiveness. This high variance is likely due to the fact that the variance in optimality of searched branches is higher, since we have no lower limit on the actions that are selected as long as they are in the top two.

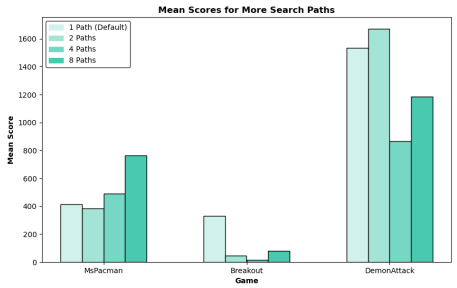


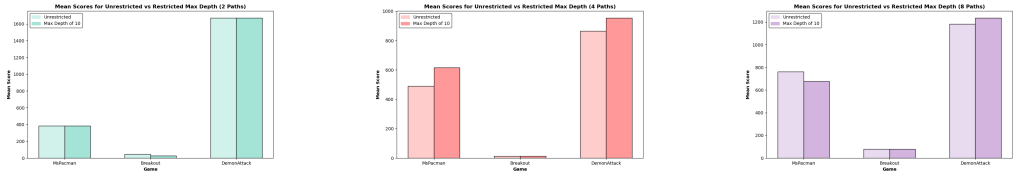

Figure 7: Mean agent scores across maximum parallel branch counts using naive branch selection.

#### 5.2.2 Restricting Maximum Depth

Our results for naive branching with a restricted maximum depth are also interesting. From 8, we see that there is almost no noticeable effect from restricting the maximum depth. This leads us to believe that after a certain depth, the information gained from rolling out more actions only has marginal effect on the resulting search statistics. Also, this hints that performance degradation with naive branching is likely due to updating visit information from more than a single best branch, not from searching deeper.

#### 5.2.3 Backpropagating Single Best Path

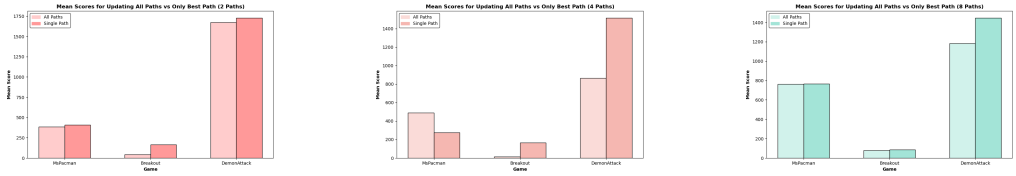
As shown below in 9, as a whole, only backpropagating the best path improves performance over backpropagating all of the visit information with naive branching. This confirms what we find




(a) 2 Naive Branches (b) 4 Naive Branches (c) 8 Naive Branches

Figure 8: Mean agent scores with naive branching for unrestricted vs restricted max search depth of 10 across different branching counts.

with restricting the maximum depth – that the drawback to naive branching is including all of the information from multiple search paths that are potentially far from optimal according to PUCT scoring.




(a) 2 Branches (b) 4 Branches (c) 8 Branches

Figure 9: Mean agent scores by only backpropagating the single best path in parallel search.

### 5.2.4 Dynamic Branching

Finally, with our dynamic branching method we see performance increases 10. Specifically, we see increases of up to $35\%$ on MsPacman and DemonAttack, while not suffering as much performance degradation as with naive branching on Breakout. We also see that the performance with dynamic branching has much less variance. We attribute this increased performance and lower variance to the fact that dynamic branching has a much tighter restriction on the paths it chooses to search. These results also suggest that it is possible to extract richer information using multiple parallel paths with MCTS according to PUCT.

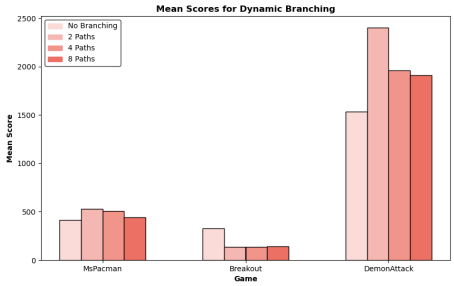



Figure 10: Mean agent scores across maximum parallel branch counts using dynamic branch selection.

After finding success with dynamic branching we also decide to investigate mean PUCT scores and number of PUCT evaluations for dynamic vs naive branching. These results are shown in 11. From this table, we see that the mean PUCT scores for these methods are very similar. We also see that PUCT scores are highest with only two branches, and degrade as more branches are visited. Since the mean PUCT scores for multiple paths are higher than default, these results suggest that with branching we are able to search paths that may have higher PUCT scores later on, but are not necessarily worth visiting and contributing to the statistics. This also makes sense for why naive and

dynamic branching have similar mean PUCT scores. These findings show that it is difficult to predict performance based solely on the PUCT scores evaluated with one of our search methods.

However, we also notice that the number of PUCT evaluations is much less with dynamic branching, but still higher than default. This confirms that the selectivity of branches from initial PUCT scores is much more important than the average PUCT scores across entire rollouts, and that by evaluating more states in the search tree we can improve performance over default search.

| | | 2 Branches | | 4 Branches | | 8 Branches | |
|---|---|---|---|---|---|---|---|
| | 1 Branch (Default) | Naive | Dynamic | Naive | Dynamic | Naive | Dynamic |
| Mean PUCT Scores | 0.882 | 1.057 | 1.067 | 1.010 | 1.010 | 0.957 | 0.961 |
| Number of Evals | 234 | 495 | 369 | 906 | 547 | 1534 | 666 |

Figure 11: Mean PUCT scores found and number of PUCT evaluations in parallel search for naive vs dynamic branching.

### 5.3 Search vs Inference Times

We briefly include the difference in search time vs model inference time during MCTS simulations since it is vital to our choice of parallel path search and a failure point for our similarity-based methods. From our testing, we find that the raw search averages about $4.5e^{-5}$ seconds and model inference averages about $5.1e^{-3}$ seconds, implying that search is roughly 110x faster.

## 6 Conclusion

For this project we improve performance on both MsPacman and DemonAttack with a parallel MCTS search using a dynamic action selection mechanism. We decide on this method after realizing that EfficientZero's model inference is orders of magnitude slower than the search. Although our method incurs a minor slowdown (10% for 2 paths, 33% for 4, and 100% for 8) due to a larger inference batch size, it provides a richer evaluation of the search tree, and empirically improves agent performance.

We also discover that using state similarities to improve MCTS degrades performance and is much slower. We believe the performance of these methods can be improved, but the potential gains are minimal since there seem to be very few similar states in most MCTS simulations. The most important reason why these methods are not worth pursuing is the significant (20x) slowdown incurred.

Since our most successful method using dynamic action selection with parallel search paths does not consider state similarities, it can be applied to any RL problem that uses policy-guided MCTS search. It is also relatively straightforward to implement, with minor tweaking based on hyperparameters including optimal search path count, $N$, and PUCT score margin, $\Delta$, making it a promising method for future investigation.

## 7 Future Work

Since our results from 11 imply that cumulative PUCT scores are less predictive of performance than the PUCT scores of actions closer to root nodes, we believe that our dynamic branching method shows more promise than some form of beam search using PUCT scores. Also, for this project we only have limited time and resources to test our methods during evaluation on fully trained EfficientZero models, so we believe that using dynamic branching during training is the most exciting direction forward with this research idea. Lastly, this implementation works out of the box with training using our current EfficientZero-forked codebase, making compute resources the only requirement to move forward in this direction.

## References


Ivo Danihelka, Arthur Guez, Julian Schrittwieser, and David Silver. 2022. Policy improvement by planning with gumbel. In *International Conference on Learning Representations*.

Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Mohammadamin Barekatain, Simon Schmitt, and David Silver. 2021. Learning and planning in complex action spaces. *CoRR*, abs/2104.06303.

Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, Timothy P. Lillicrap, and David Silver. 2019. Mastering atari, go, chess and shogi by planning with a learned model. *CoRR*, abs/1911.08265.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy P. Lillicrap, Karen Simonyan, and Demis Hassabis. 2017. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *CoRR*, abs/1712.01815.

Maciej Swiechowski, Tomasz Tajmajer, and Andrzej Janusz. 2018. Improving hearthstone AI by combining MCTS and supervised learning algorithms. *CoRR*, abs/1808.04794.

Weirui Ye, Shaohuai Liu, Thanard Kurutach, Pieter Abbeel, and Yang Gao. 2021. Mastering atari games with limited data. *CoRR*, abs/2111.00210.

# 8 Code

The adapted code for this project can be found here. The branches of interest are $\_Similarity$ (for our first family of similarity-based methods and $\_More\_Paths$ for our second family of parallel searching methods.

# 9 Appendix

Given our early experiments looking into reduced simulations we train an EfficientZero network on Breakout for 50k timesteps with only $N = 10$ simulations and 4 naive search paths. As shown in 12, our agent never averages more than 3 points earned, compared to around 100-200 for the default EfficientZero in the first 50k timesteps. We believe that these poor results do not predict poor performance for dynamic branching with the same number of simulations since the results below are obtained using naive branching and too few simulations to provide enough search information.

## 9.1 Preliminary Naive Branching Training Results

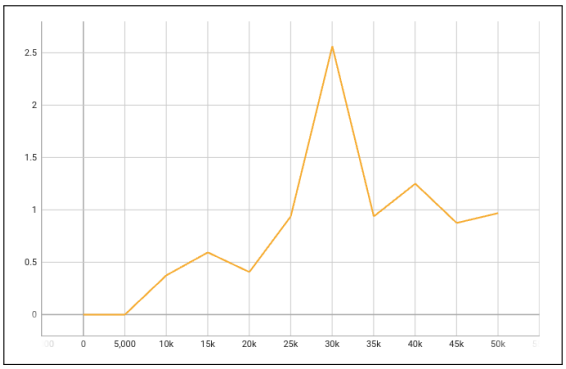



Figure 12: Mean agent scores during training for Breakout with only 10 simulations (default = 50) and 4 naive branches.