to -> Efficient Zero v2

I. Mu Zero 로 오기까지

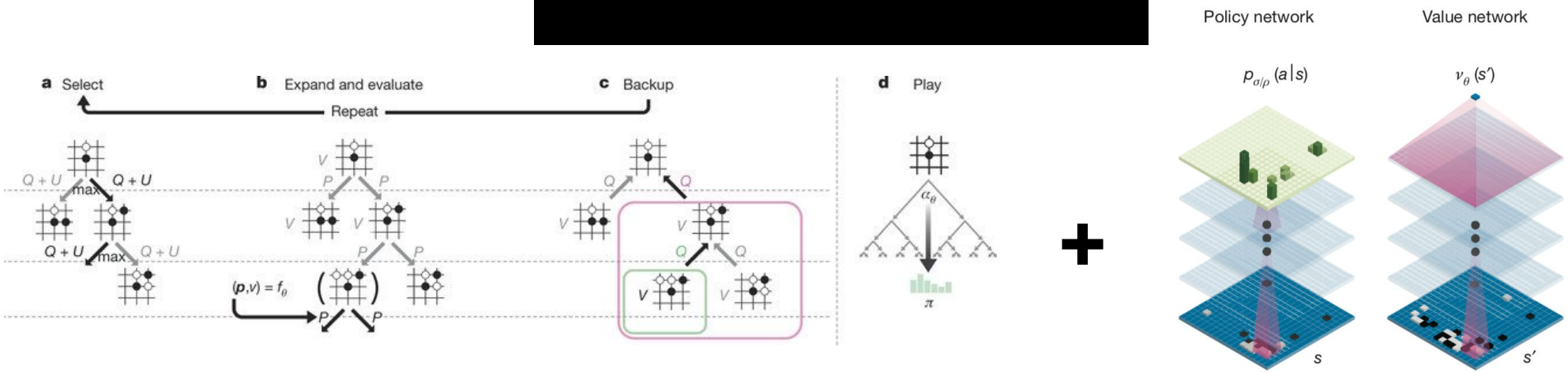
II. 문제는?

III. Efficient Zero(2021)

IV. Efficient Zero v2(2024)

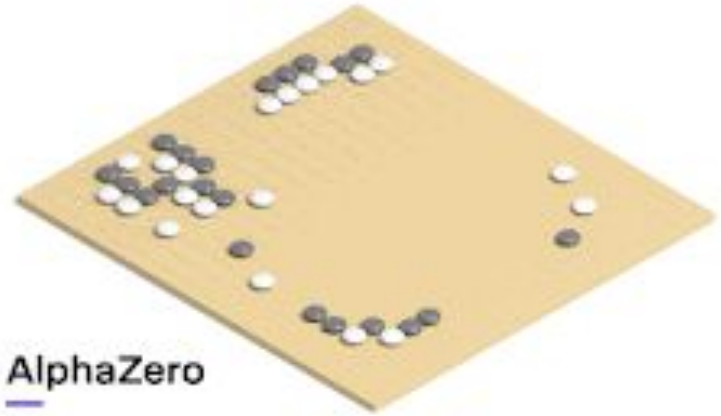
V. GPU 와 MCTS

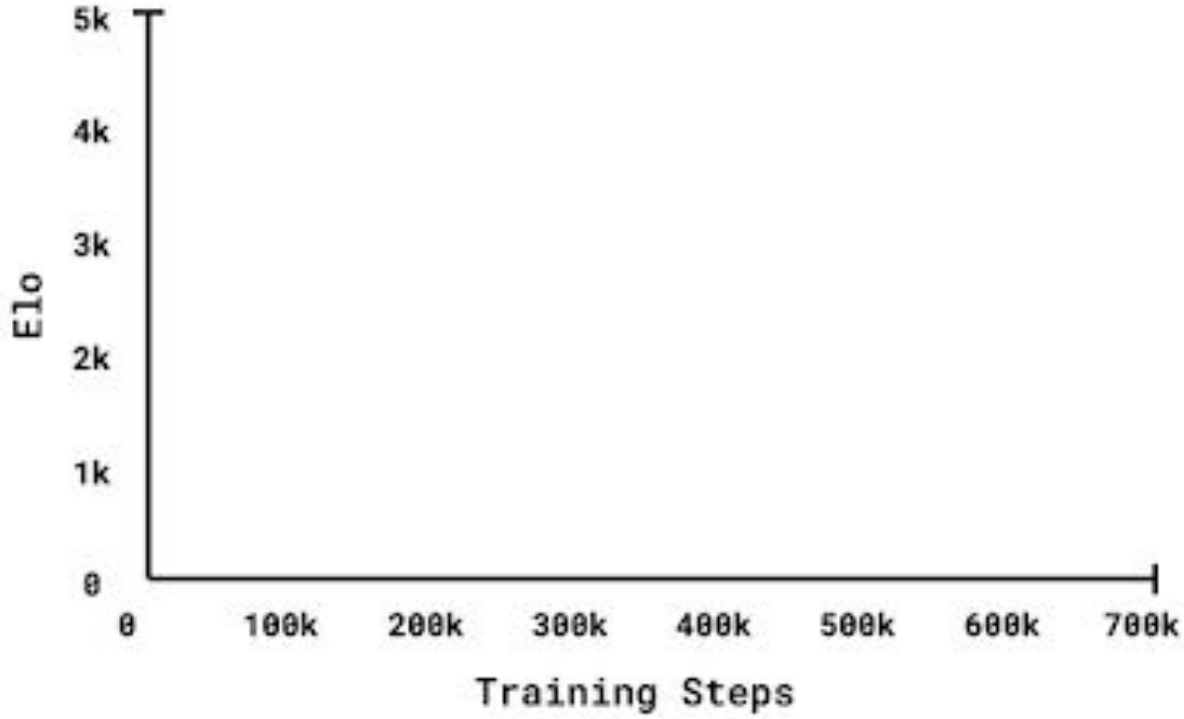
Mu Zero 로 오기까지 - AlphaGo, AI 시대의 신호탄



- 2016년 AlphaGo는 Policy/Value 네트워크와 MCTS의 결합으로 바둑에서 인간 최고수를 압도
- 사람들의 기보를 학습에 사용. 이로 인해 완전히 AI 만으로 인간을 넘어섰다고 보기는 어려울수 있음
- 하지만, 이 사건은 강화학습과 탐색 알고리즘의 결합이 갖는 잠재력을 보여줌

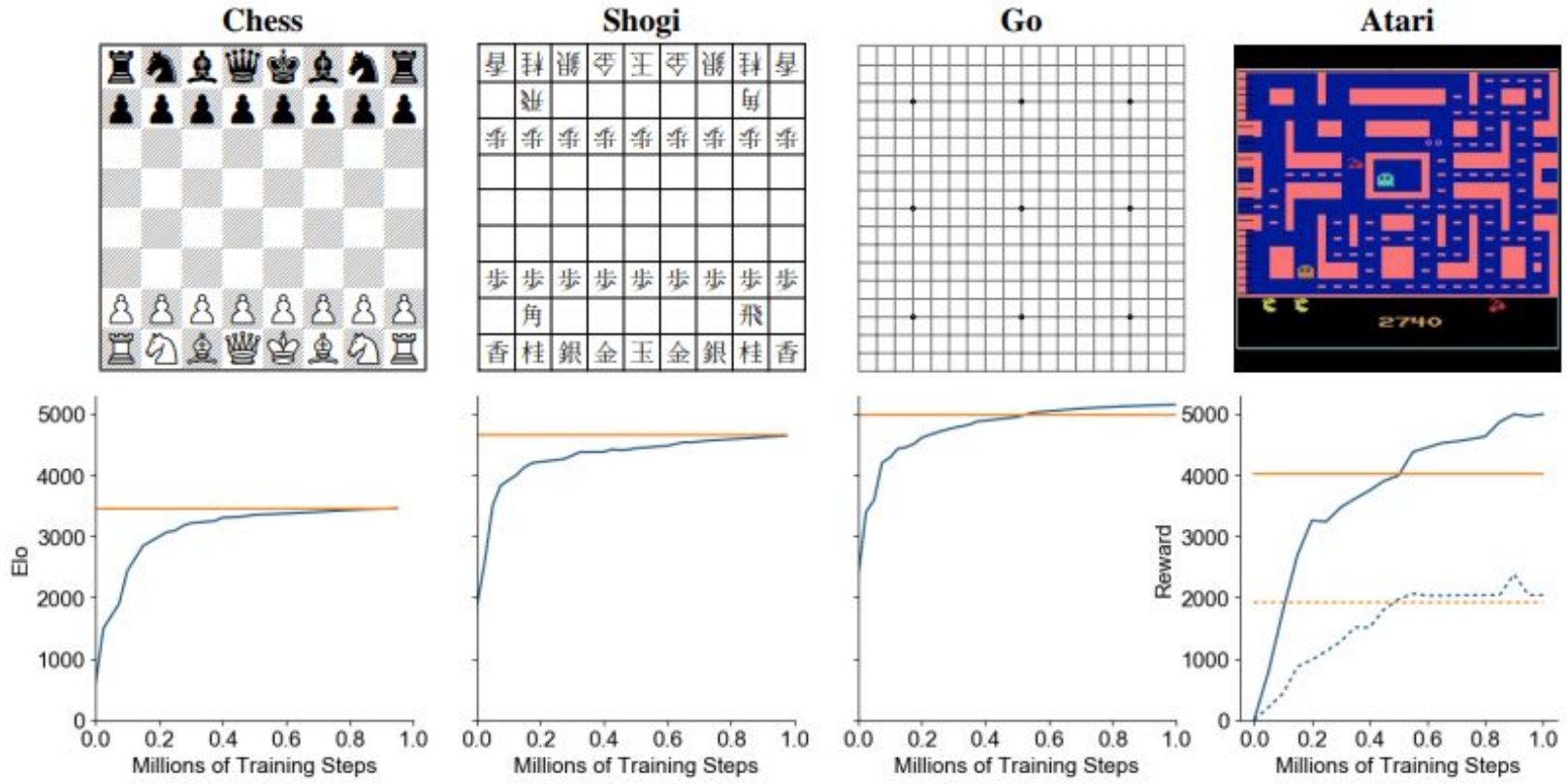
Mu Zero 로 오기까지 - AlphaZero, 바둑을 넘어서 모든것으로





- AlphaZero 는 사람의 도메인 지식이 없이 Self-Play 만으로 학습이 가능하게 만들었음
- 도메인 지식이 필요 없으므로, 바둑, 장기, 체스 등 어떤 종류의 게임도 학습 가능하게 만들수 있었음
- 사람 없이, 인공지능 자체로 완전히 사람을 넘어서는 알고리즘의 등장
- 하지만, 풀고자 하는 문제의 MCTS에 올릴수 있는 완벽한 시뮬레이터 필요

Mu Zero 로 오기까지 - MuZero, 이제부턴 상상의 힘으로!



- AlphaZero 로 Atari 를 풀기 위해서는 AlphaZero를 위해 Atari 를 로직적으로도, 그래픽적으로도 완벽히 재현한 코드가 있어야 하지만 그러기 위해서는 비용이 너무 많이 들 뿐더러 현실적이지도 않음.
- MuZero 는 MCTS에 올릴수 있는 완벽한 시뮬레이터가 제공되지 않아도 됨 MCTS 내에서 수행되는 샘플링은 오직 '월드 모델' 즉 상상에서만 진행됨. 이로 인해 Atari 같은 게임도 현실적인 범위에서 풀수 있음.

문제는? - 데이터 사용량

the cost of a large number of environmental interactions. For example, AlphaZero [39] needs to play 21 million games at training time. On the contrary, a professional human player can only play around 5 games per day, meaning it would take a human player 11,500 years to achieve the same amount of experience. The sample complexity might be less of an issue when applying RL

21M / 5 game / 365 day = 11,500 year

- AlphaZero 는 어떠한 도메인 지식 없이도 매우 뛰어난 성능을 보임
- 그렇다 치더라도 엄청난 수의 게임을 시도해야 함
- 21M 게임을 시도
  - 현실적인 횟수로 프로 바둑 기사는 하루 5회의 게임 가능
  - 이를 365일로 나누면
  - 하루도 쉬지 않고 게임해서 11,500년
- 그만큼 대량의 샘플을 탐색하며 모으는것은 엄청나게 고도의 엔지니어링이 필요한데다 매우 오랜 시간과 자원이 필요

문제는? - 연산 비용

computations. For example, MuZero needs 64 TPUs to train 12 hours for one agent on Atari games. The high computational costs pose problems both for the future development of such methods as well as practical applications.

TPU v4 pod	us-central2	오클라호마	US$3.22
TPU v3 pod	europe-west4	네덜란드	US$2.00

3.22$ x 64 x 12h = 2,458$ = 450만원

- MuZero 는 사실상 거의 넘을수 없는 Sota의 성능을 달성
- 또한 AlphaZero 와 비교해서 탐색해야 하는 샘플수 자체도 크게 줄어듬
- 하지만 Atari 한 게임을 학습하는데에 필요한 자원은 무지막지함
  - MuZero 의 발표 시기상 TPU v4 를 사용한다고 가정
  - 현재 TPU v4의 1시간 가격은 3.22$ 이지만 과거 시점은 더 높음
  - 이때 TPU 64 개를 12시간동안 사용해 학습
- 계산상 한 게임을 학습하는데 필요한 금액은 450만원을 훨씬 넘어섬

문제는? - 문제 정의

computations. For example, MuZero needs 64 TPUs to train 12 hours for one agent on Atari games. The high computational costs pose problems both for the future development of such methods as well as practical applications.

- 대부분의 연구원은 1개 case의 Eval 에 450만원 이상을 지출할수 없음
- 연구를 진행하려면 코드를 작성하면서 수십 수백번을 테스트 필요
- 또한 Atari 는 알고리즘의 성능을 평가하는데 많이 사용되지만, 실용적 문제들에 비교해 상대적으로 쉬운 문제들에 가까움
- 더 어려운 문제는 훨씬 더 많은 자원을 사용할것은 명백함
- 도저히 실용적으로 사용할수 없는 수준

- 이제 우리는 현실적으로 RL으로 문제를 풀기 위해서
  - 적은 데이터
  - 적은 연산량
- 으로 SOTA에 가까운 성능을 낼수 있을 알고리즘이 필요

Efficient Zero(2021) - 개요

- Efficient Zero 는 MuZero 의 틀에서 3가지를 개선
  - 자기 지도 일관성 학습
  - Value prefix 학습
  - Off-Policy 보정

- 달성
  - 100K 개의 샘플만으로 학습
    - DQN이 50M 샘플으로 학습하는 성능을 500배 적은 데이터로 학습
  - GPU 4개로 7시간을 학습
    - MuZero 에 비하면 한없이 적은 자원과 시간을 사용

Abstract

Reinforcement learning has achieved great success in many applications. However, sample efficiency remains a key challenge, with prominent methods requiring millions (or even billions) of environment steps to train. Recently, there has been significant progress in sample efficient image-based RL algorithms; however, consistent human-level performance on the Atari game benchmark remains an elusive goal. We propose a sample efficient model-based visual RL algorithm built on MuZero, which we name EfficientZero. Our method achieves 194.3% mean human performance and 109.0% median performance on the Atari 100k benchmark with only two hours of real-time game experience and outperforms the state SAC in some tasks on the DMControl 100k benchmark. This is the first time an algorithm achieves super-human performance on Atari games with such little data. EfficientZero's performance is also close to DQN's performance at 200 million frames while we consume 500 times less data. EfficientZero's low sample complexity *and* high performance can bring RL closer to real-world applicability. We implement our algorithm in an easy-to-understand manner and it is available at https://github.com/YeWR/EfficientZero. We hope it will accelerate the research of MCTS-based RL algorithms in the wider community.

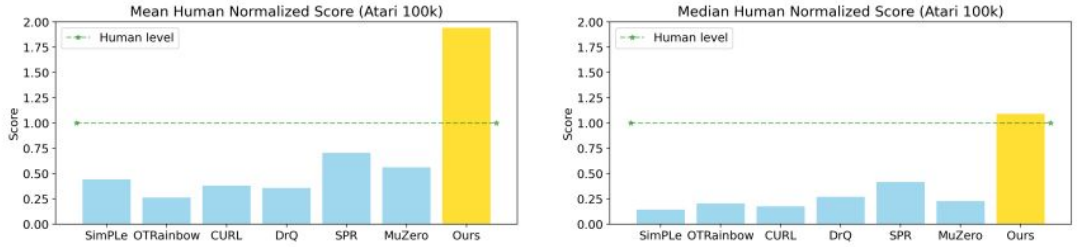


Figure 1: Our proposed method EfficientZero is 176% and 163% better than the previous SoTA performance in mean and median human normalized score and is the first to outperform the average human performance on the Atari 100k benchmark. The high sample efficiency and performance of EfficientZero can bring RL closer to the real-world applications.

Efficient Zero(2021) - 자기 지도 일관성 손실

- MuZero 는 오직, reward 와 value, policy 신호만으로 월드 모델을 학습
  - 이는 모델이 적은 샘플로 학습되기 위해서 충분한 gradient를 생성하지 못함
- Efficient Zero 는 자기지도학습 으로 월드 모델을 좀 더 적은 샘플로 학습할수 있음

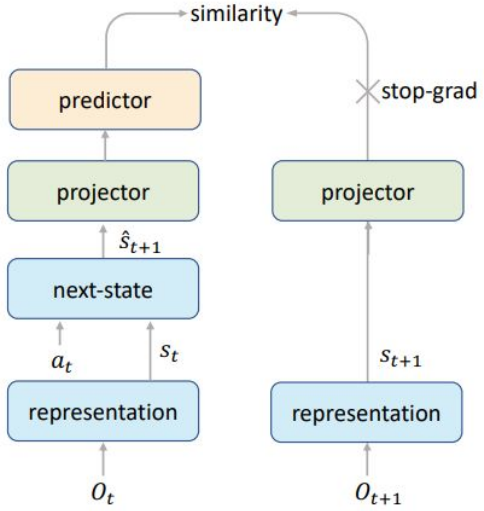
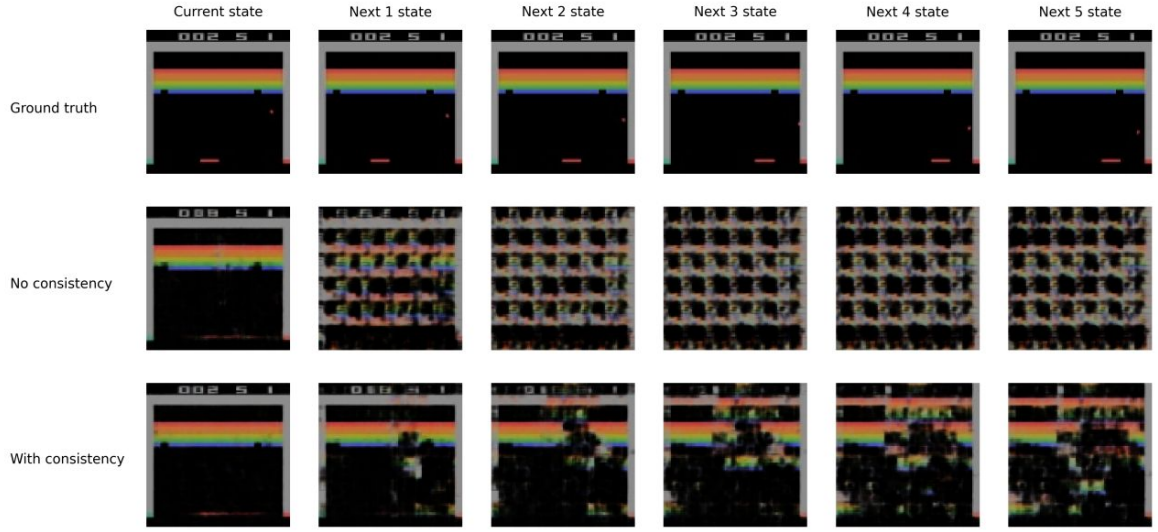


Figure 2: The self-supervised consistency loss.

Game	Full	w.o. consistency	w.o. value prefix	w.o. off-policy correction
Normed Mean	**1.943**	0.881	1.482	1.475
Normed Median	**1.090**	0.340	0.552	0.836



- 자기지도학습 으로 월드 모델을 가이드 하지 않았을 경우 매우 큰 폭으로 성능이 떨어짐
- 또한 월드 모델의 레이턴트 공간에서 프레임을 재구성할때
  - 자기지도학습이 없을경우: 전혀 재구성되지 않음
  - 자기지도학습이 있을경우: 재구성이 정상적으로 가능함
- 자기지도학습이 월드 모델이 적은 데이터로 환경의 동역학적 일관성을 담아내도록 강제하고, 일반화를 도움

Efficient Zero(2021) - Value Prefix 학습

- MuZero 는 한 step 씩 환경의 reward 를 예측하여 여러 step 의 reward 를 벨만 방정식으로 합산하여 계산
- 하지만 모든 값은 정확히 근사되지 않고. 많은 step 을 합산할수록 누적되는 오차가 많아짐
- 또한 학습된 근사함수인 월드 모델은 reward 가 정확히 어느 시점에 추가되는지. 정확히 근사하기 어려움
- 이렇게 여러번 reward 를 계산하고 합산하는 방식은 구조적으로 한계를 가지고 있음

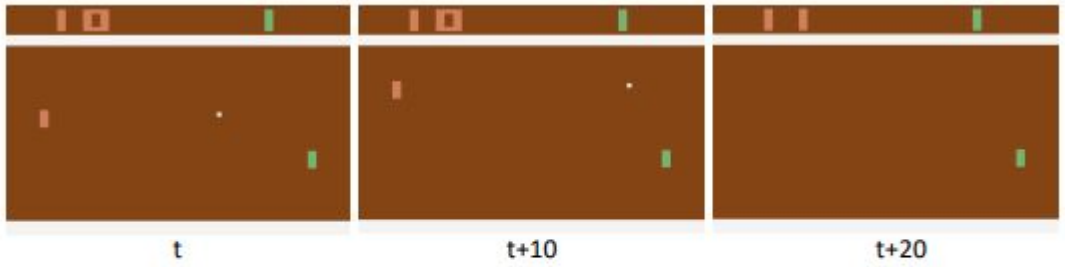
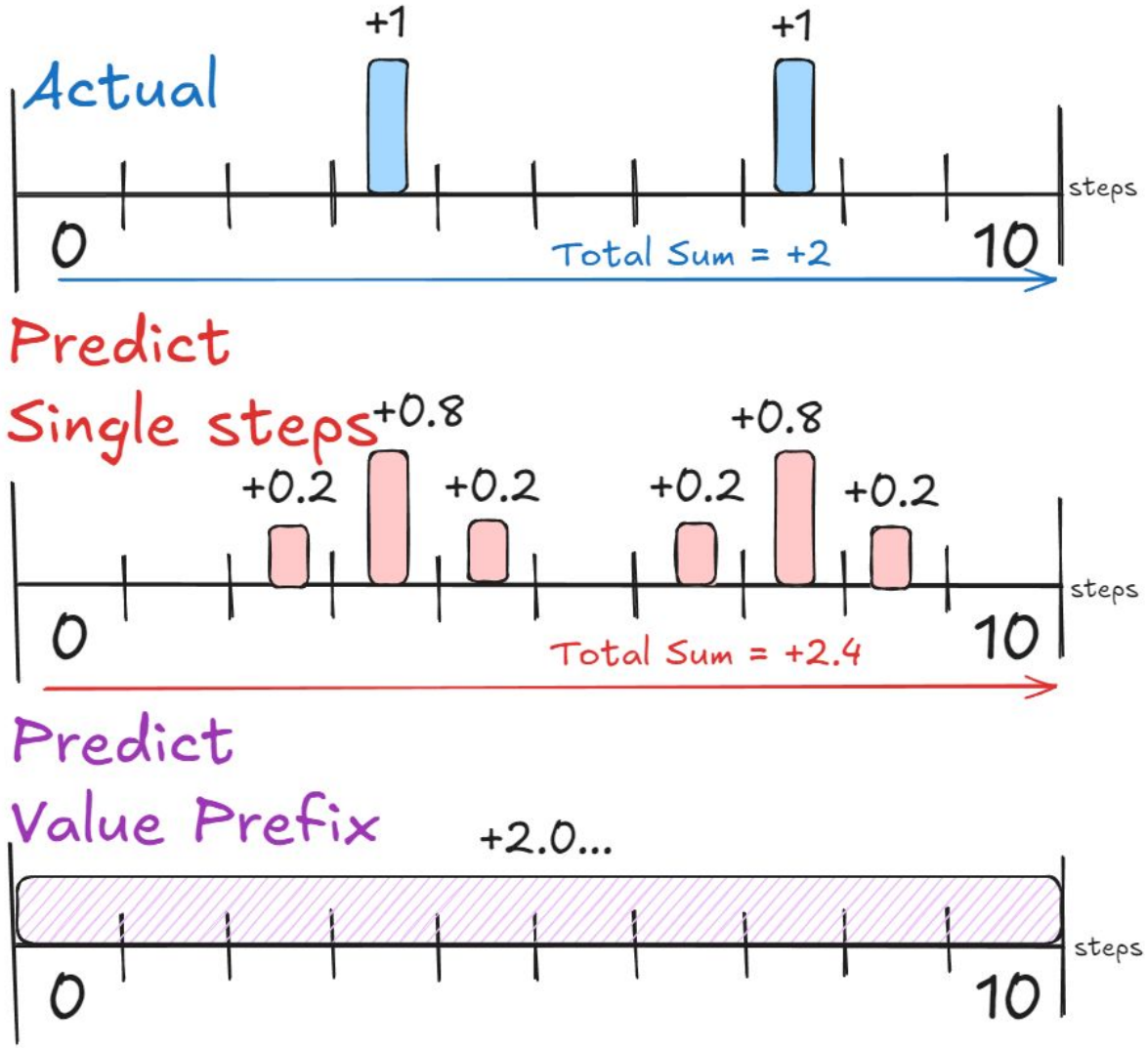


Figure 3: A sample trajectory from the Atari Pong game. In this case, the right player didn't move and missed the ball.



- EfficientZero 는 이를 Value prefix 라는 구조로 파훼
- 여러 Step 의 레이턴트 공간을 입력으로 받아. 해당 step들 내에서 나올 reward 합산을 한번에 예측하도록 작성

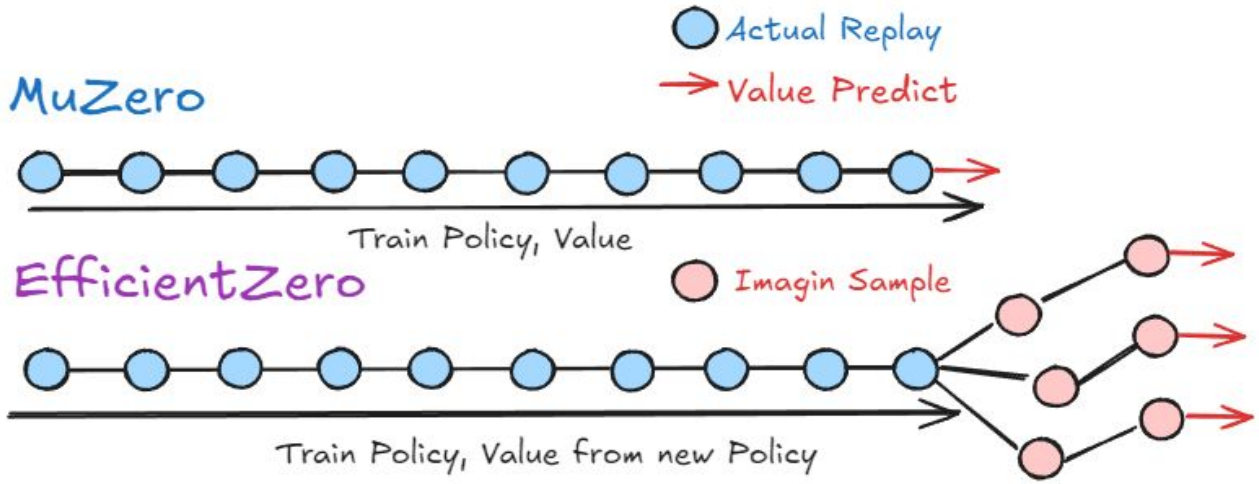
Efficient Zero(2021) - Off policy 보정

- 기존의 MuZero Reanalyze(Replay buffer 를 써서 샘플 효율을 높이는 버전) 는 학습할 정보를 Replay buffer 의 리플레이를 불러와 더 정확히 학습

- 하지만 Replay buffer 에서 불러와진 리플레이는 낡은 정책에서 만들어진 정보

- 초기 정책의 리플레이는 후기 정책으로 고도화 된 뒤에는 상대적으로 부정확한 추정값을 낼 가능성이 높음

- 그렇기에, EfficientZero 는 리플레이 시점의 정보를, 지금 시점의 정책으로 월드 모델에서 시뮬레이션해 최신 정책으로 행동하면 예상되는 끝단 Value를 새로 만들어내 학습시킴



$$z_t = \sum_{i=0}^{l-1} \gamma^i u_{t+i} + \gamma^l \nu_{t+l}^{\text{MCTS}}$$

Efficient Zero(2021) - 아키텍쳐 도식

- Efficient Zero 는 연산 자원을 최대한 효율적으로 사용하기 위해 여러 프로세스(Ray worker)를 동원해 동시에 작업들을 처리
- Data worker
  - MCTS 를 수행해 최선의 행동을 선택해 환경을 효과적으로 탐색하며 데이터를 수집하는 worker
- CPU Rollout worker
  - Replay에 담긴 정보를 정리하고 Batch 화 해 Reanalyzing 작업을 보조
- GPU batch worker
  - 과거 정보를 불러와 Reanalyzing 하여 실제 모델을 학습할 새로운 Policy. Value Target 정보 생산
- Learner
  - GPU batch worker 가 생산한 정보를 학습하여 갱신된 네트워크를 Data worker로 전송

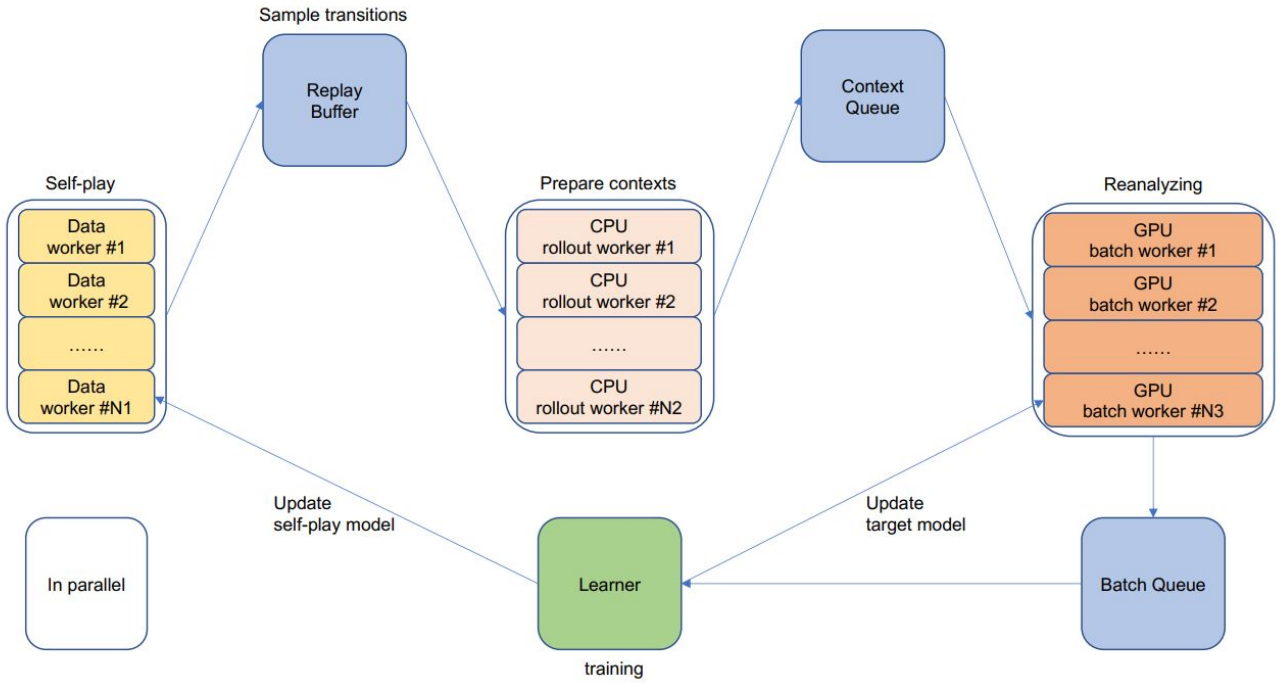


Figure 6: Pipeline of the EfficientZero implementation.

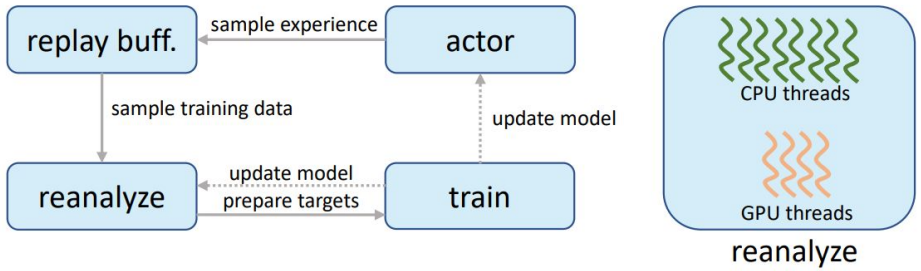


Figure 9: EfficientZero implementation overview.

Efficient Zero(2021) - Atari 100K & DM Control 100k

- 실제 시간으로 환산시 2시간의 플레이 정보인 Atari 100K
  - 최초로 사람수준의 성능 달성
  - 이에 근접한 성능을 내기 위해서 DQN은 500배 많은 데이터를 수집해야 함
  - 26개 게임중 14개의 게임을 사람보다 잘함

- 연속적 액션 제어 환경인 DM Control 100k
  - 이산 액션 알고리즘이기 때문에, 각 연속 제어 차원 마다 5등분 하여 이산화 하여 학습
  - 물론 제어 차원이 늘어날수록 액션 개수가 지수적으로 증가하기 때문에 연속 액션 차원이 3개 이하만 테스트 진행
  - 이때 픽셀 입력을 받아 행동함에도, Vector 입력을 받는 SAC 보다 더 나은 성능 달성

Table 1: Scores on the Atari 100k benchmark (3 runs with 32 seeds). EfficientZero achieves super-human performance with only 2 hours of real-time game play. Our method is 176% and 163% better than the previous SoTA performance, in mean and median human normalized score respectively.

Game	Random	Human	SimPLe	OTRainbow	CURL	DrQ	SPR	MuZero	Ours
Alien	227.8	7127.7	616.9	**824.7**	558.2	771.2	801.5	530.0	808.5
Amidar	5.8	1719.5	88.0	82.8	142.1	102.8	**176.3**	38.8	148.6
Assault	222.4	742.0	527.2	351.9	600.6	452.4	571.0	500.1	**1263.1**
Asterix	210.0	8503.3	1128.3	628.5	734.5	603.5	977.8	1734.0	**25557.8**
Bank Heist	14.2	753.1	34.2	182.1	131.6	168.9	**380.9**	192.5	351.0
BattleZone	2360.0	37187.5	5184.4	4060.6	14870.0	12954.0	**16651.0**	7687.5	13871.2
Boxing	0.1	12.1	9.1	2.5	1.2	6.0	35.8	15.1	**52.7**
Breakout	1.7	30.5	16.4	9.8	4.9	16.1	17.1	48.0	**414.1**
ChopperCmd	811.0	7387.8	1246.9	1033.3	1058.5	780.3	974.8	**1350.0**	1117.3
Crazy Climber	10780.5	35829.4	62583.6	21327.8	12146.5	20516.5	42923.6	56937.0	**83940.2**
Demon Attack	152.1	1971.0	208.1	711.8	817.6	1113.4	545.2	3527.0	**13003.9**
Freeway	0.0	29.6	20.3	25.0	**26.7**	9.8	24.4	21.8	21.8
Frostbite	65.2	4334.7	254.7	231.6	1181.3	331.1	**1821.5**	255.0	296.3
Gopher	257.6	2412.5	771.0	778.0	669.3	636.3	715.2	1256.0	**3260.3**
Hero	1027.0	30826.4	2656.6	6458.8	6279.3	3736.3	7019.2	3095.0	**9315.9**
Jamesbond	29.0	302.8	125.3	112.3	471.0	236.0	365.4	87.5	**517.0**
Kangaroo	52.0	3035.0	323.1	605.4	872.5	940.6	**3276.4**	62.5	724.1
Krull	1598.0	2665.5	4539.9	3277.9	4229.6	4018.1	3688.9	4890.8	**5663.3**
Kung Fu Master	258.5	22736.3	17257.2	5722.2	14307.8	9111.0	13192.7	18813.0	**30944.8**
Ms Pacman	307.3	6951.6	**1480.0**	941.9	1465.5	960.5	1313.2	1265.6	1281.2
Pong	-20.7	14.6	12.8	1.3	-16.5	-8.5	-5.9	-6.7	**20.1**
Private Eye	24.9	69571.3	58.3	100.0	**218.4**	-13.6	124.0	56.3	96.7
Qbert	163.9	13455.0	1288.8	509.3	1042.4	854.4	669.1	3952.0	**13781.9**
Road Runner	11.5	7845.0	5640.6	2696.7	5661.0	8895.1	14220.5	2500.0	**17751.3**
Seaquest	68.4	42054.7	683.3	286.9	384.5	301.2	583.1	208.0	**1100.2**
Up N Down	533.4	11693.2	3350.3	2847.6	2955.2	3180.8	**28138.5**	2896.9	17264.2
Normed Mean	0.000	1.000	0.443	0.264	0.381	0.357	0.704	0.562	**1.943**
Normed Median	0.000	1.000	0.144	0.204	0.175	0.268	0.415	0.227	**1.090**

Table 2: Scores achieved by EfficientZero (mean & standard deviation for 10 seeds) and some baselines on some low-dimensional environments on the DMControl 100k benchmark. EfficientZero achieves state-of-art performance and comparable results to the state-based SAC.

Task	CURL	Dreamer	MuZero	SAC-AE	Pixel SAC	State SAC	EfficientZero
Cartpole, Swingup	582± 146	326±27	218.5 ± 122	311±11	419±40	835±22	**813±19**
Reacher, Easy	538± 233	314±155	493 ± 145	274±14	145±30	746±25	**952±34**
Ball in cup, Catch	769± 43	246 ± 174	542 ± 270	391± 82	312± 63	746±91	**942±17**

Efficient Zero(2021) - 의의와 한계

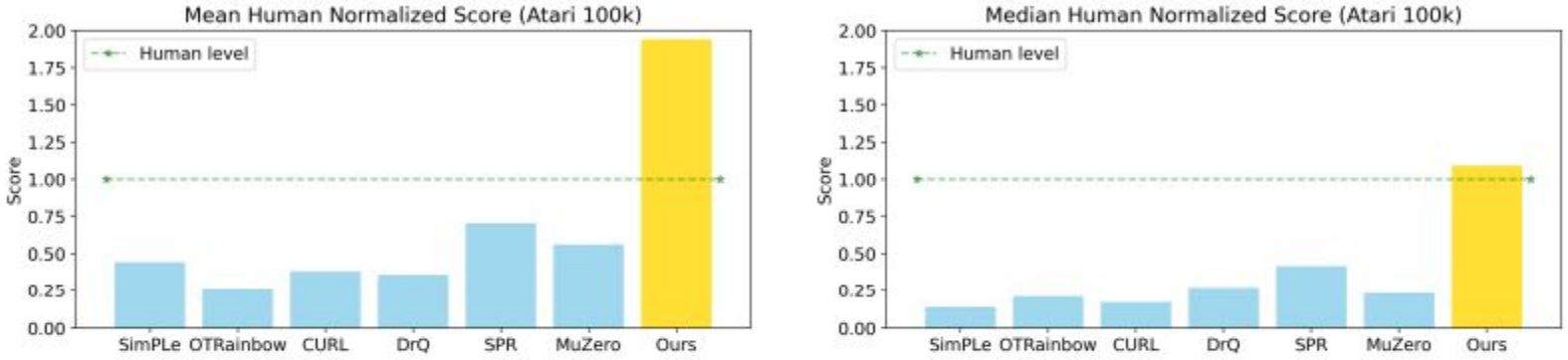


Figure 1: Our proposed method EfficientZero is 176% and 163% better than the previous SoTA performance in mean and median human normalized score and is the first to outperform the average human performance on the Atari 100k benchmark. The high sample efficiency and performance of EfficientZero can bring RL closer to the real-world applications.

- EfficientZero는 저데이터 환경에서 모델+탐색 접근의 가치를 입증
  - 아타리 100k 벤치마크에서 최초로 "사람 이상"인 알고리즘

- 하지만 연속 제어 태스크에서는 그대로 적용하기 어려운 구조적 한계가 존재
  - 차원의 저주로 3개의 연속 제어 차원 이하만 적용 가능. 샘플 효율성 측면에서는 SOTA 이지만,

    그 외는 이산 제어 알고리즘으로 연속 제어 환경이 '가능은 하다' 수준

    (액션의 개수가 N 개면 액션의 수가 5^N 의 크기이기 때문에 이를 커버하는 서치는 사실상 가능하지 않음)

Efficient Zero v2(2024) - 개요

- Efficient Zero V2 는 v1의 강점을 유지하면서 연속 제어와 저차원 관측을 포괄하는 범용 프레임워크를 제시
- 연속 제어
  - 기존 Efficient Zero 는 이산적 행동만 가능. 이 마저도 행동 개수가 늘어나면 탐색에 필요한 서치 개수가 크게 늘어나야 했음
  - 액션별로 순서를 매겨 트리를 확장하는 기존 MCTS와 다르게. 좀더 효율적 트리확장인 Gumbel Search 를 적용
  - -1 ~ 1 사이의 실수 N개를 예측하는 형태의 제어와 같은 문제에도 최고의 성능 달성
- 저차원 관측
  - 기존 Efficient Zero 는 역으로 "이미지들" 을 입력받고 행동을 결정하는 고차원 관측에 특화되어 있었음
  - 하지만 이제 여러 벡터들이나 수치적 값인 저차원 관측들을 사용하는 환경에서도 사용 가능해짐

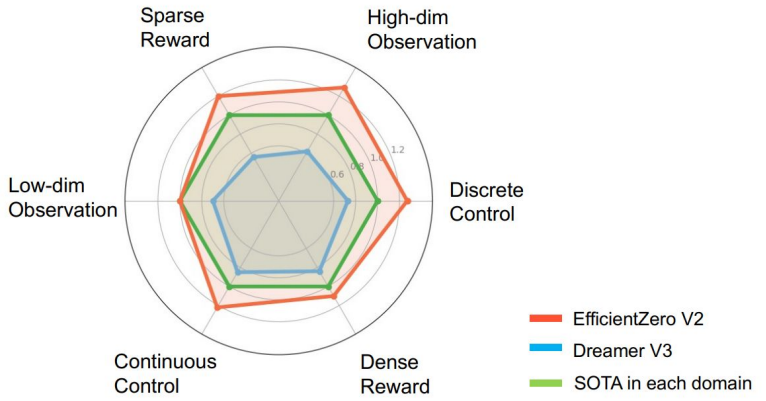


Figure 1. Comparison between EfficientZero V2, DreamerV3 and other SOTAs in each domain. We evaluate them under the Atari 100k, DMControl Proprio, and DMControl Vision benchmarks. We then set the performance of the previous SOTA as 1, allowing us to derive normalized mean scores for both EfficientZero V2 and Dreamer V3. EfficientZero V2 surpasses or closely matches the previous SOTA in each domain.

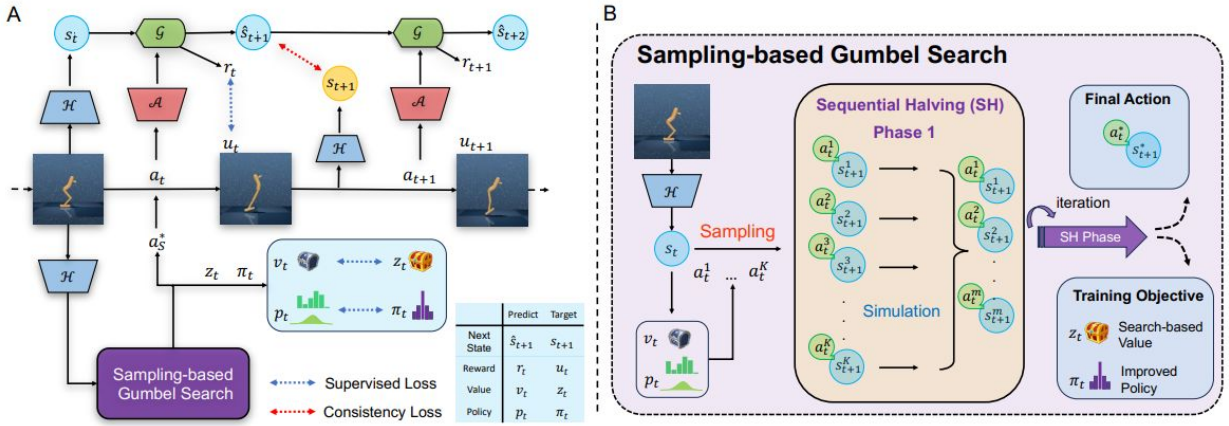


Figure 2. Framework of EZ-V2. (**A**) How EZ-V2 trains its model. The representation $\mathcal{H}$ takes observations as inputs and outputs the state. The dynamic model $\mathcal{G}$ predicts the next state and reward based on the current state and action. Sampling-based Gumbel search outputs the target policy π_t and target value z_t. (**B**): How the sampling-based Gumbel search uses the model to plan. The process contains action sampling and selection. The iterative action selection outputs the recommended action a_S^*, search-based value target (target value), and improved policy (target policy).

Efficient Zero v2(2024) - 먼저 / Gumbel Muzero

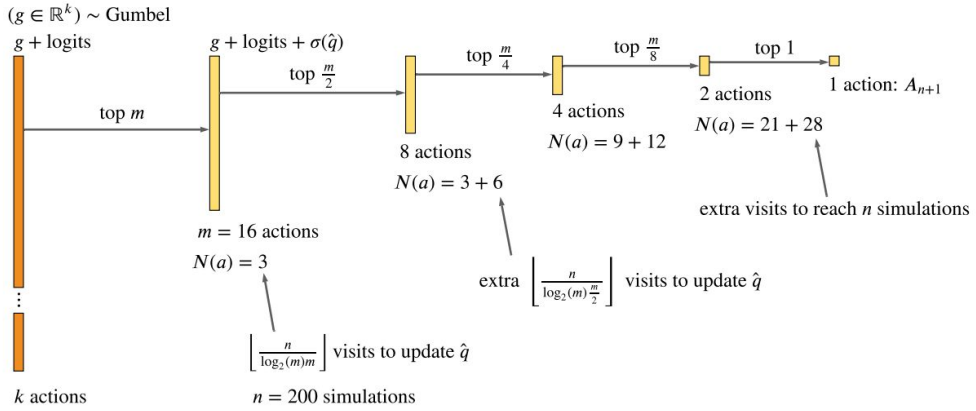


Figure 1: The number of considered actions and their visit counts $N(a)$, when using Sequential Halving with Gumbel on a k-armed stochastic bandit. The search uses $n = 200$ simulations and starts by sampling $m = 16$ actions without replacement. Sequential Halving divides the budget of n simulations equally to $\log_2(m)$ phases. In each phase, all considered actions are visited equally often. After each phase, one half of the actions is rejected. From the original k actions, only the best actions will remain.

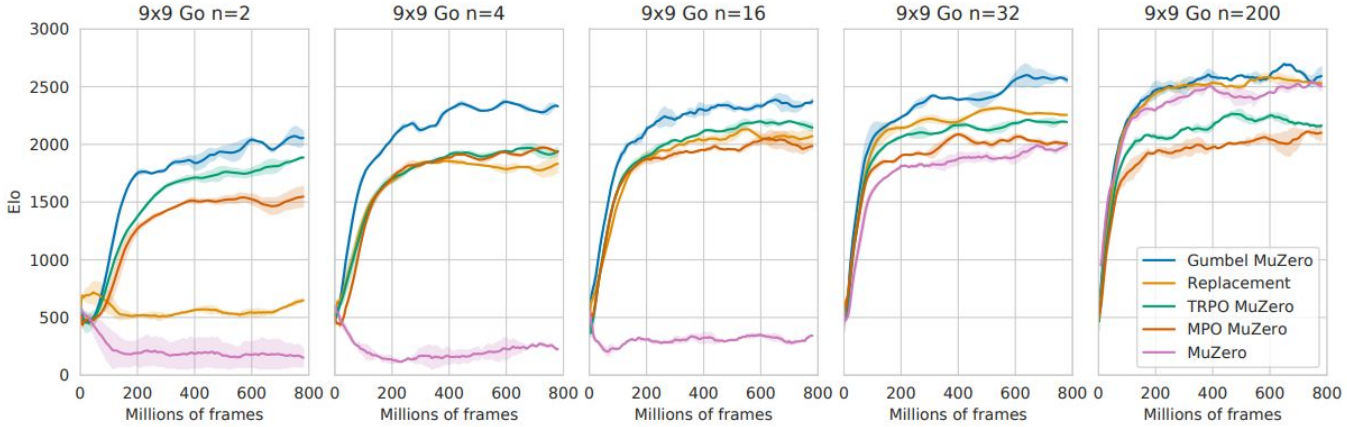


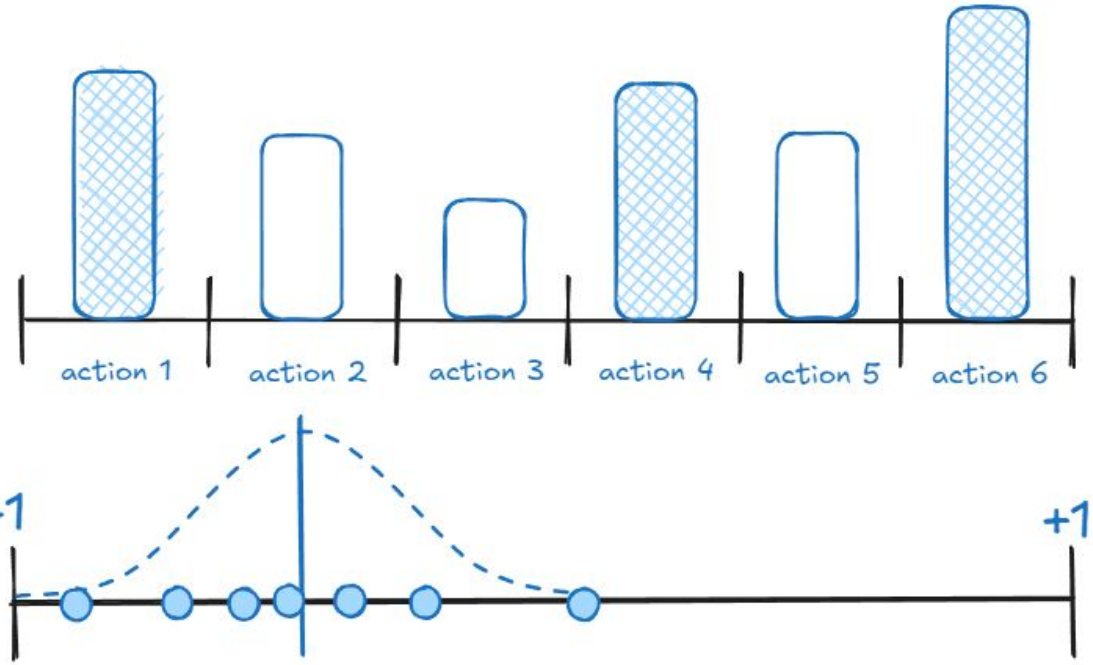
Figure 2: Elo on 9x9 Go, when training with $n \in \{2, 4, 16, 32, 200\}$ simulations. Evaluation uses 800 simulations. Shades denote standard errors from 2 seeds.

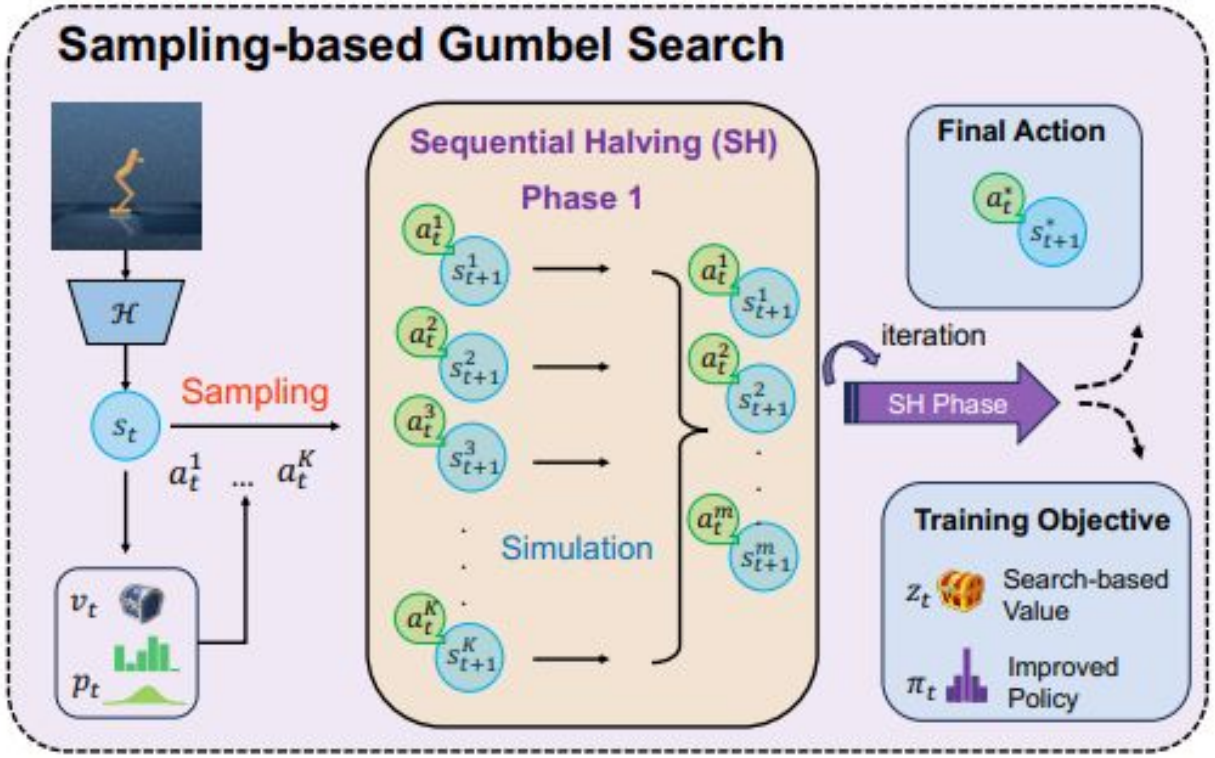
- 기존 '순수' MCTS 방법론은. 정말 대수의 법칙에 입각한 통계법칙으로 행동을 평가하고 학습하였음
- 이 방법의 문제는 결국 큰 액션 공간의 문제를 학습하려면 그에 따라 탐색해야 하는 샘플의 수가 기하급수적으로 많이 필요
- Gumbel Muzero 는 N 개의 행동을 선택할때. 가장 가치가 높아 보이는 M 개의 행동을 골라내고 거기서만 선택.
  - 이때 '순수' MCTS와 다르게 N 개의 행동은 절대 겹치지 않는 '비복원 추출' 을 사용(탐색 횟수를 낭비하지 않기 위해)
  - 또한 이러한 Gumbel 추출과 그 결과를 이용한 학습 방법은 "이론적"으로 정책의 개선을 보장함

    (기존 MCTS의 결과를 뉴럴넷에 학습하는건 정책이 이전보다 나아지는것을 보장하지 못했음)
- 이를 통한 성과로. 왼쪽 그림과 같이 한 행동을 하기 위해 MCTS의 노드를 2개만 확장하더라도 높은 성능 확보
- 노드 개수를 늘리면 노드가 많은 MCTS보다도 성능이 매우 높음

Efficient Zero v2(2024) - Sample Base Gumbel Search

- 이산 행동
  - 기존의 Gumbel MCTS 전략대로 사용
  - 전체 행동에서 상위 M 개를 추려 '비복원 추출' 시행
- 연속 행동
  - 정책 네트워크가 SAC 와 같이 평균과 표준편차를 뱉는 형태로 변경. 이후 M 개의 벡터를 분포에서 샘플링
  - 여기서 샘플링된 Action 들을 기반으로 Gumbel Search





Efficient Zero v2(2024) - Sample Base Gumbel Search

- 이러한 Gumbel MCTS 를 기반으로 하는 방법은.

  Gumbel Muzero 의 이산 행동에서 성능 증가를 이미 검증.

- 연속 공간을 해결하는 MCTS 기반 연구가 없던것은 아니였음

  Sample MCTS는 그 사례중 하나.
- Efficient Zero v2 의 S-Gumbel Search 는 Sample

  MCTS 에서 Gumbel 기반 MCTS로 변경한것과 거의 동일

- 이때 Sample MCTS 와 S-Gumbel Search 를 비교해 봤을때

  훨씬 작은 노드 수(8개) 만으로 큰 서치 수(50개) 로 Search 한

  Sample MCTS 보다 훨씬 높은 성능 달성.

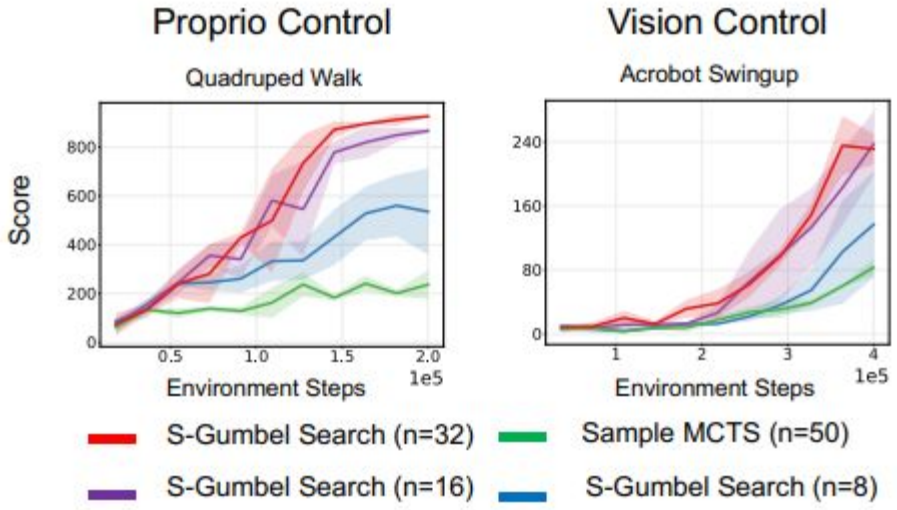


Figure 3. Ablation study of our search method, namely the sampling-based Gumbel search (S-Gumbel search). We compare it with our search method with different numbers of simulations (n=16, 8) and Sample MCTS (Hubert et al., 2021). Our method outperforms Sample MCTS, and increasing the number of simulations improves our method's performance on hard tasks.

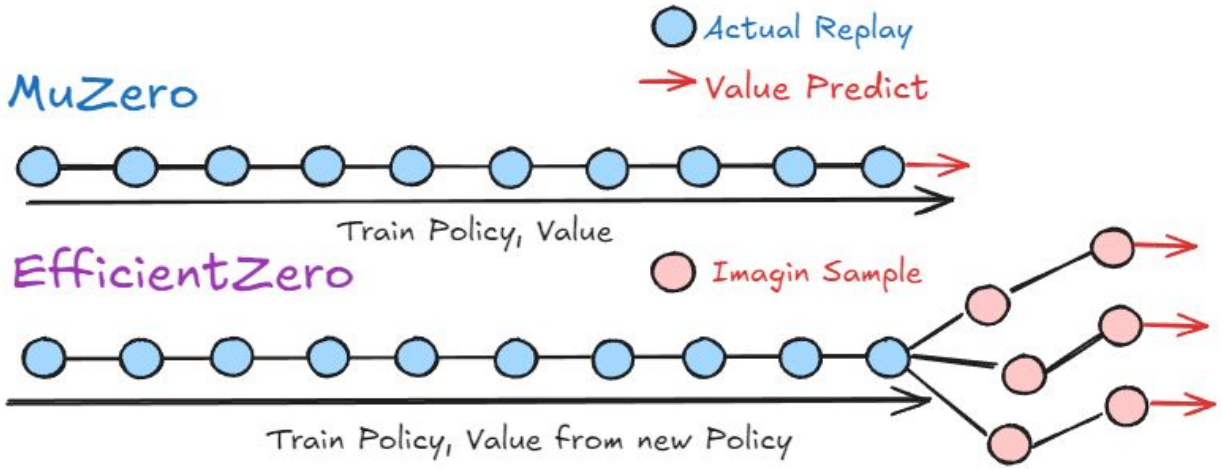
Efficient Zero v2(2024) - 서치 기반 벨류 추정

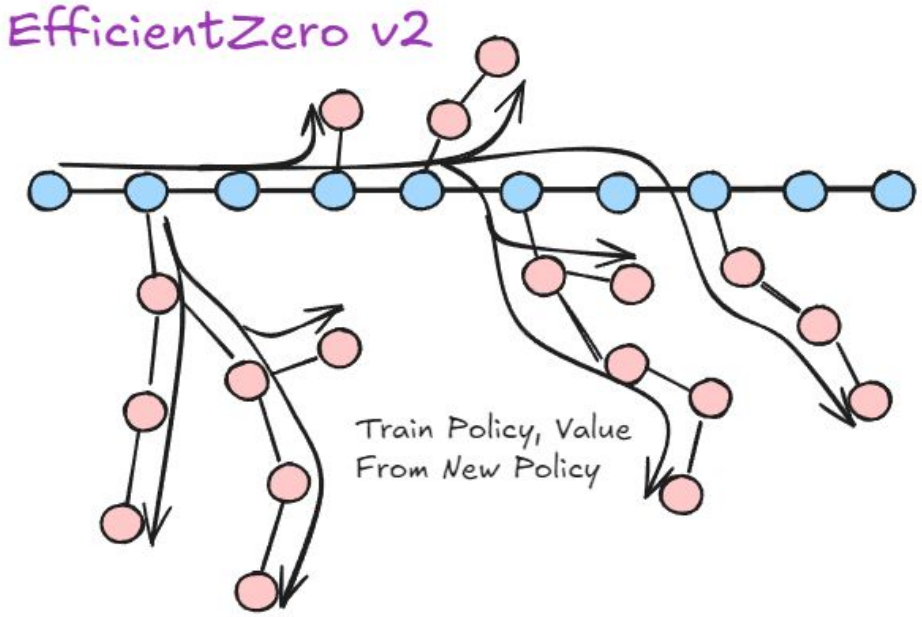
- Efficient Zero 의 Off policy 보정과 목적은 동일하지만 더욱 더 발전된 방법론인 서치 기반 밸류 주정 사용

- 기존의 Off policy 보정은 오래된 정책의 기록을 그대로 사용하되, 마지막에서 새로운 Policy에 맞는 Value 를 예측하기 위해 새로 Value 를 계산

- 서치 기반 벨류 추정은, 오래된 정책의 모든 시점에서 서치를 다시 수행한 뒤 Value 를 계산하여 학습
- 오래된 정책의 기록을 사실상 완전히 재해석하는 단계에 이름

- 오래되서 쓸모없는 기록이여도 오래 생각해 가능성을 보는것과 같은 이런 작업을 통해 이전 Efficient Zero 보다 더욱 높은 샘플 효율성을 달성





$$\hat{V}_n(s_0) = \sum_{t=0}^{H(n)} \gamma^t \hat{r}_t + \gamma^{H(n)} \hat{V}(\hat{s}_{H(n)})$$

Efficient Zero v2(2024) - 저차원 관측

- Efficient Zero v2 는 저차원 관측에서도 높은 성능을 위해. 저차원 관측에서는 기존의 이미지 입력용 Conv 를 대체해 일반적인 FC 레이어로 대체

- 또한. '입력만' 정규화하는 running mean block 을 추가
- 이는 Batch Norm 이지만 학습 가능한 파라미터가 없는것과 같음

- 이런 running mean block은 꽤나 생소한 구조로 보일수 있지만. Simba 와 같은 저차원 데이터의 아키텍쳐 논문에서 비슷한 동작을 하는 개념인 RS Norm 이 등장하기도 함

For the 2-dimensional image inputs, we follow the most implementation of the architecture of EfficientZero. For the continuous control task with 1-dimensional input, we use a variation of the previous architecture in which all convolutions are replaced by fully connected layers. In the following, we describe the detailed architecture of EZ-V2 under 1-dimensional input.

The representation function $\mathcal{H}$ first processes the observation by a running mean block. The running mean block is similar to a Batch Normalization layer without learnable parameters. Then, the normalized input is processed by a linear layer, followed by a Layer Normalisation and a Tanh activation. Hereafter, we use a Pre-LN Transformer style pre-activation residual tower (Xiong et al., 2020) coupled with Layer Normalisation and Rectified Linear Unit (ReLU) activations to obtain the latent state. We used 3 blocks and the output dim is 128. Each linear layer has a hidden size of 256.

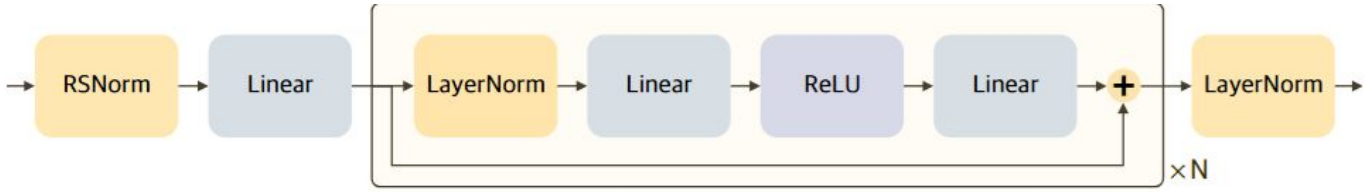


Figure 3: **SimBa architecture.** The network integrates Running Statistics Normalization (RSNorm), Residual Feedforward Blocks, and Post-Layer Normalization to embed simplicity bias into deep RL.

Running Statistics Normalization (RSNorm). First, RSNorm standardizes input observations by tracking the running mean and variance of each input dimension during training, preventing features with disproportionately large values from dominating the learning process.

Given an input observation $\mathbf{o}_t \in \mathbb{R}^{d_o}$ at timestep t, we update the running observation mean $\mu_t \in \mathbb{R}^{d_o}$ and variance $\sigma_t^2 \in \mathbb{R}^{d_o}$ as follows:

$$\mu_t = \mu_{t-1} + \frac{1}{t}\delta_t, \quad \sigma_t^2 = \frac{t-1}{t}(\sigma_{t-1}^2 + \frac{1}{t}\delta_t^2) \tag{3}$$

where $\delta_t = \mathbf{o}_t - \mu_{t-1}$ and d_o denotes the dimension of the observation.

Once μ_t and σ_t^2 are computed, each input observation $\mathbf{o}_t$ is normalized as:

$$\bar{\mathbf{o}}_t = \text{RSNorm}(\mathbf{o}_t) = \frac{\mathbf{o}_t - \mu_t}{\sqrt{\sigma_t^2 + \epsilon}} \tag{4}$$

where $\bar{\mathbf{o}}_t \in \mathbb{R}^{d_o}$ is the normalized output, and ϵ is a small constant for numerical stability.

While alternative observation normalization methods exist, RSNorm consistently demonstrates superior performance. A comprehensive comparison is provided in Section 7.1.

Efficient Zero v2(2024) - Atari 100K

Table 1. Scores achieved on the Atari 100k benchmark indicate that EZ-V2 achieves super-human performance within just 2 hours of real-time gameplay. Our method surpasses the previous state-of-the-art, EfficientZero. The results for Random, Human, SimPLe, CURL, DrQ, SPR, MuZero, and EfficientZero are sourced from (Ye et al., 2021).

Game	Random	Human	SimPLe	CURL	DrQ	SPR	MuZero	EfficientZero	DreamerV3	BBF	**EZ-V2 (Ours)**
Alien	227.8	7127.7	616.9	558.2	771.2	801.5	530.0	808.5	959	1173.2	**1557.7**
Amidar	5.8	1719.5	88.0	142.1	102.8	176.3	38.8	148.6	139	**244.6**	184.9
Assault	222.4	742.0	527.2	600.6	452.4	571.0	500.1	1263.1	706	**2098.5**	1757.5
Asterix	210.0	8503.3	1128.3	734.5	603.5	977.8	1734.0	25557.8	932	3946.1	**61810.0**
Bank Heist	14.2	753.1	34.2	131.6	168.9	380.9	192.5	351.0	649	732.9	**1316.7**
BattleZone	2360.0	37187.5	5184.4	14870.0	12954.0	16651.0	7687.5	13871.2	12250	**24459.8**	14433.3
Boxing	0.1	12.1	9.1	1.2	6.0	35.8	15.1	52.7	52.7	**85.8**	75.0
Breakout	1.7	30.5	16.4	4.9	16.1	17.1	48.0	**414.1**	31	370.6	400.1
ChopperCmd	811.0	7387.8	1246.9	1058.5	780.3	974.8	1350.0	1117.3	420	**7549.3**	1196.6
Crazy Climber	10780.5	35829.4	62583.6	12146.5	20516.5	42923.6	56937.0	83940.2	97190	58431.8	**112363.3**
Demon Attack	152.1	1971.0	208.1	817.6	1113.4	545.2	3527.0	13003.9	303	13341.4	**22773.5**
Freeway	0.0	29.6	20.3	**26.7**	9.8	24.4	21.8	21.8	0	25.5	0.0
Frostbite	65.2	4334.7	254.7	1181.3	331.1	1821.5	255.0	296.3	909	**2384.8**	1136.3
Gopher	257.6	2412.5	771.0	669.3	636.3	715.2	1256.0	3260.3	3730	1331.2	**3868.7**
Hero	1027.0	30826.4	2656.6	6279.3	3736.3	7019.2	3095.0	9315.9	**11161**	7818.6	9705.0
Jamesbond	29.0	302.8	125.3	471.0	236.0	365.4	87.5	517.0	445	**1129.6**	468.3
Kangaroo	52.0	3035.0	323.1	872.5	940.6	3276.4	62.5	724.1	4098	**6614.7**	1886.7
Krull	1598.0	2665.5	4539.9	4229.6	4018.1	3688.9	4890.8	5663.3	7782	8223.4	**9080.0**
Kung Fu Master	258.5	22736.3	17257.2	14307.8	9111.0	13192.7	18813.0	**30944.8**	21420	18991.7	28883.3
Ms Pacman	307.3	6951.6	1480.0	1465.5	960.5	1313.2	1265.6	1281.2	1327	2008.3	**2251.0**
Pong	-20.7	14.6	12.8	-16.5	-8.5	-5.9	-6.7	20.1	18	16.7	**20.8**
Private Eye	24.9	69571.3	58.3	218.4	-13.6	124.0	56.3	96.7	**882**	40.5	99.8
Qbert	163.9	13455.0	1288.8	1042.4	854.4	669.1	3952.0	13781.9	3405	4447.1	**16058.3**
Road Runner	11.5	7845.0	5640.6	5661.0	8895.1	14220.5	2500.0	17751.3	15565	**33426.8**	27516.7
Seaquest	68.4	42054.7	683.3	384.5	301.2	583.1	208.0	1100.2	618	1232.5	**1974.0**
Up N Down	533.4	11693.2	3350.3	2955.2	3180.8	**28138.5**	2896.9	17264.2	-	12101.7	15224.3
Normed Mean	0.000	1.000	0.443	0.381	0.357	0.704	0.562	1.945	1.120	2.247	**2.428**
Normed Median	0.000	1.000	0.144	0.175	0.268	0.415	0.227	1.090	0.490	0.917	**1.286**

- Efficient Zero v2 는 범용 알고리즘(이산, 연속 행동 알고리즘) 인 Dreamer v3 와 비교했을때 66 개 중 50개에서 우위를 드러냄
- 또한 이전인 Efficient Zero 비교하여서도 이산 공간 문제에서 큰 폭의 성능 증가를 보였음

Efficient Zero v2(2024) - DM Control Vision/Proprio

Table 2. Scores achieved on the Proprio Control 50/100k and Vision Control 100/200k benchmarks (with 3 seeds run for each) demonstrate that EZ-V2 consistently maintains sample efficiency, whether with proprioceptive or visual inputs. The tasks are categorized into easy and hard groups as proposed by (Hubert et al., 2021). The results of DreamerV3 are sourced from the official data (Hafner et al., 2023).

Benchmark	Proprio Control 50k				Vision Control 100k			
Task	SAC	TD-MPC2	DreamerV3	**EZ-V2 (Ours)**	CURL	DrQ-v2	DreamerV3	**EZ-V2 (Ours)**
Cartpole Balance	**997.6**	962.8	839.6	947.3	963.3	**965.5**	956.4	911.7
Cartpole Balance Sparse	993.1	942.8	559.0	**999.2**	999.4	**1000.0**	813.0	951.5
Cartpole Swingup	**861.6**	826.7	527.7	805.4	**765.4**	756.0	374.8	747.8
Cup Catch	949.9	**976.0**	729.6	969.8	932.3	468.0	947.7	**954.7**
Finger Spin	900.0	**965.8**	765.8	837.1	850.2	459.4	633.2	**927.6**
Pendulum Swingup	158.9	520.1	**830.4**	825.4	144.1	233.3	619.3	**726.7**
Reacher Easy	744.0	903.5	693.4	**940.3**	467.9	722.1	441.4	**946.3**
Reacher Hard	646.5	580.4	768.0	**795.4**	112.7	202.9	120.4	**961.5**
Walker Stand	870.0	**973.9**	767.3	953.6	733.8	426.1	939.5	**944.9**
Walker Walk	813.2	**965.5**	475.2	944.0	538.5	681.5	771.2	**888.8**
	Proprio Control 100k				Vision Control 200k			
Acrobot Swingup	44.1	**303.2**	62.8	297.7	6.8	15.1	67.4	**231.8**
Cartpole Swingup Sparse	256.6	421.4	172.7	**795.4**	8.8	81.2	392.4	**763.6**
Cheetah Run	**680.9**	614.4	400.8	677.8	405.1	418.4	587.3	**631.6**
Finger Turn Easy	630.8	**793.3**	560.5	310.7	371.5	286.8	366.6	**799.2**
Finger Turn Hard	414.0	**604.8**	474.2	374.1	236.3	268.4	258.5	**794.6**
Hopper Hop	0.1	84.5	9.7	**186.5**	84.5	26.3	76.3	**206.4**
Hopper Stand	3.8	**807.9**	296.1	795.4	627.7	290.2	652.5	**805.7**
Quadruped Run	139.7	**742.1**	289.0	510.6	170.9	339.4	168.0	**384.8**
Quadruped Walk	237.5	853.7	256.2	**925.8**	131.8	311.6	122.6	**433.3**
Walker Run	635.4	**780.5**	478.9	657.2	274.7	359.9	**618.2**	475.3
Mean	552.0	**740.9**	517.1	723.2	437.3	410.3	498.5	**726.1**
Median	633.3	**806.4**	543.4	800.4	324.9	330.6	484.5	**788.1**

- 연속 액션 공간에서도 비전/저차원 상태공간 모두에서 다른 기존 RL 알고리즘들을 압도하거나 준 SOTA에 가까운 성능을 확인
- TD-MCP2 를 저차원 문제에서 꺾을수 없었지만 TD-MCP2 는 한 행동을 결정하기 위해 9216개의 샘플을 탐색했고, Efficient Zero v2 는 32개만 탐색. 또한 이를 Flops 단위에서 비교하면 1000배정도 더 연산 효율성이 높았음

Efficient Zero v2(2024) - 의의와 한계

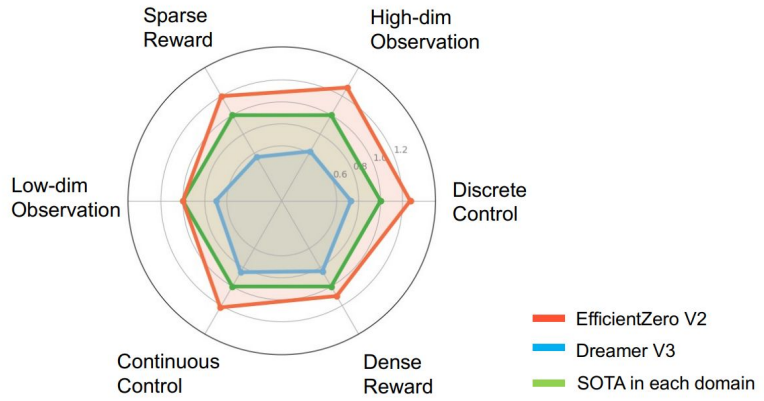


Figure 1. Comparison between EfficientZero V2, DreamerV3 and other SOTAs in each domain. We evaluate them under the Atari 100k, DMControl Proprio, and DMControl Vision benchmarks. We then set the performance of the previous SOTA as 1, allowing us to derive normalized mean scores for both EfficientZero V2 and Dreamer V3. EfficientZero V2 surpasses or closely matches the previous SOTA in each domain.

- Efficient Zero v2 는 위 그림처럼 샘플이 한정된 경우에서 6가지 속성의 문제들에서 SOTA 를 넘어서는 성능을 달성
- Dreamer v3 가 아주 강력한 '범용' 알고리즘으로써 주목을 받았지만 Efficient Zero v2는 많은 부분에서 Dreamer v3를 큰 폭으로 능가
- 하지만 Efficient Zero v2 에게도 한계는 존재
    - 모델 아키텍쳐 상 RNN 이나 LSTM, RSSM 등을 사용하는 모델들이 처리할수 있는 POMDP 문제를 풀어낼 수 없음
    - Gumbel MCTS의 파라미터에 대한 큰 민감도를 가져 학습 시간에 대한 병목과 성능 향상 폭이 제한되는 경우도 존재

GPU 와 MCTS - 나오지 않은 문제점

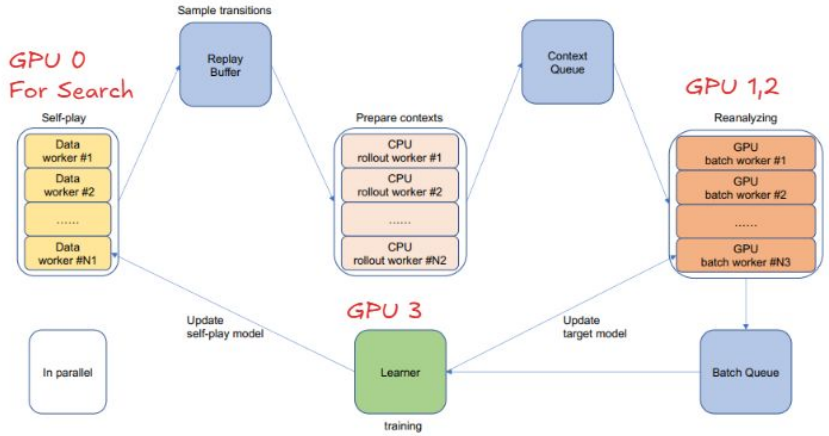


Figure 6: Pipeline of the EfficientZero implementation.

We think our open-source implementation of EfficientZero can drastically accelerate the research in MCTS RL algorithms. Our implementation is computationally friendly. To train an Atari agent for 100k steps, it only needs 4 GPUs to train 7 hours. Our framework could potentially have a large impact on many real-world applications, such as robotics since it requires significantly fewer samples.

ate the reanalyze module, we split the reanalyze computation into the CPU part and the GPU part, such that computation on CPU and GPU are run in parallel. We use a different number of actors between CPU and GPU to match their total throughput. To increase the throughput on GPU, we also collocate multiple batch computation threads on one GPU, as in Tian et al. [43]. We also implement the MCTS in C++ to avoid performance issues with Python on large amounts of atomic computations.

- Efficient Zero 의 학습 파이프라인 도식을 살펴보면, 위와 같이 매우 많은 worker 로 분산하고 병렬적으로 학습을 진행하였음
- 하지만 이게 "왜" 필요했는가? 를 생각해 보아야 함
- 하나의 MCTS 를 처리하는것은 연산이 많이 필요하지 않음, 하지만 Efficient Zero 에서 Reanalyzing 을 해야 하는 데이터는 엄청나게 많은데 이를 하나하나 따로 처리하는것은 GPU의 throughput 을 낭비하는것과 다름없음
- 그 결과 throughput 을 낭비하지 않고 최대한 적은 시간동안 학습을 진행하기 위해 몸을 비튼 결과가 위와 같은 매우 많은 worker 로 분리된 학습 구조

GPU 와 MCTS - 그래서 어떻게 해야할까?

A simple, full-GPU implementation of AlphaZero

This repository is the result of Andreas Spanopoulos' contribution to the redesign of the AlphaZero.jl open-source AlphaZero implementation in Julia. With this redesign, we achieved a remarkable **8x speedup in performance** by running the Monte Carlo Tree Search (MCTS) fully on GPU.

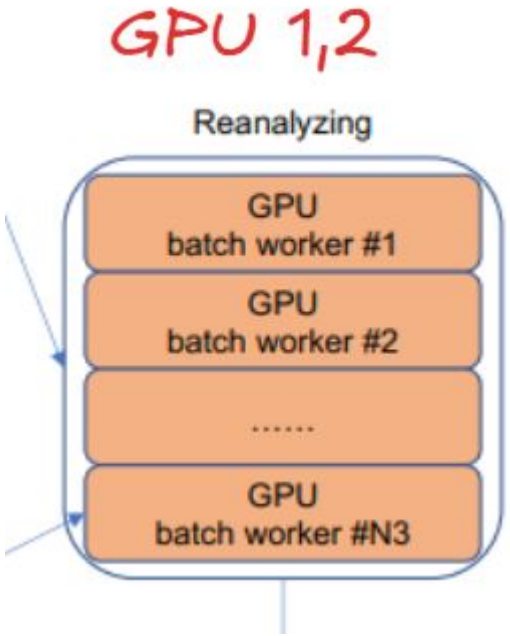
The aims of this redesign are:

1. **Code readability and maintainability.**
2. **Code modularity and extensibility.**
3. **Code performance.**

By meeting these aims, not only have we maintained high standards for the overall architecture and readability of the code of AlphaZero.jl, but we have also made a significant leap in performance, which will be discussed in detail in the evaluation section.

The redesign is based on the original implementation by Jonathan Laurent, who is the main author of AlphaZero.jl, and the mentor of this project.

In this redesign, the MCTS algorithm runs fully on GPU. By fully, we mean that the environment functions, the neural network inference, and the MCTS algorithm itself, all run on GPU, like in [Jones, 2021]. This allows us to parallelize the MCTS algorithm across a batch of environments, thus unlocking the full potential of the GPU, and creating training data much faster during the self-play phase.



- AlphaZero.ji 라는 프로젝트는 AlphaZero의 학습에서 모든 환경과 MCTS 를 GPU 로 넣어. 이전 구현과 비교하여 8배에 가까운 성능 향상을 달성 (해당 내용이 중요한 이유는. 같은 언어와 코드베이스에서 완전한 GPU 전환이므로 좀 더 엄밀한 증거가 됨)
- 이는 GPU 기반 MCTS 가 빨라서가 아니라 GPU 내에서 돌릴때. batch 화 하여 학습하는데에 유리했기 때문임
- Efficient Zero의 batch worker 는 cpp로 작성되어 있기에 MCTS 를 제각각 worker 에 분리하고 MCTS 의 연산을 수행하고. model 부분들만 모아서 batch 로 GPU 에 전송
- 이러한 구조가 Efficient Zero 의 연산 성능에 크리티컬한 보틀넥이 될것은 매우 자명한 문제임

GPU 와 MCTS - 그럼 JAX해

- AlphaZero.ji 와 동일하게 우리도 이를 JAX로 작성하여 Reanalizing 작업을 단일 batch화 된 MCTS 작업으로 변경할 수 있음

- 이를 통해 매우 많은 ray worker 들을 작성하여 분산 처리하고, 데이터를 다시 모아 batch 화 해 gpu의 throughput 을 최대한 늘릴 고민을 할 필요가 없음
- 애초에 MCTS 의 step 이 전부 vmap을 통한 batch 로 동작하기 때문에 GPU 를 효율적으로 쓰기 위한 프로세스간 데이터 전송이나 그런 문제를 전혀 고려할 필요가 없기 때문
- 또한, 리플레이 버퍼에서 대량의 데이터를 샘플링 하는 문제도 JAX 내부에서 처리가 가능함

GPU 와 MCTS - 근데 GPU 만 쓰면 다 되나?

- 그렇다고 JAX화 & GPU 내에서 처리 만이 능사는 아니다

- 해당 사례는, MCTX(Deepmind 의 JAX 네이티브 MCTS) 의 이슈 페이지에서 긁어온 내용
- 요약하자면
  - GPU의 코어 클럭 속도는 CPU에 비해 수십배 느림
  - 이때 MCTS의 트리가 깊어 짐에 따라 알고리즘이 직렬화 되는데, 이 경우 GPU 가 CPU에 비해 너무 불리함
  - GPU는 병렬화 할수록 강력하고, 직렬화 할수록 느려지기 때문
  - 해당 문제때문에 GPU 기반 MCTS의 성능 증가가 미미할 가능성을 배제할수 없음

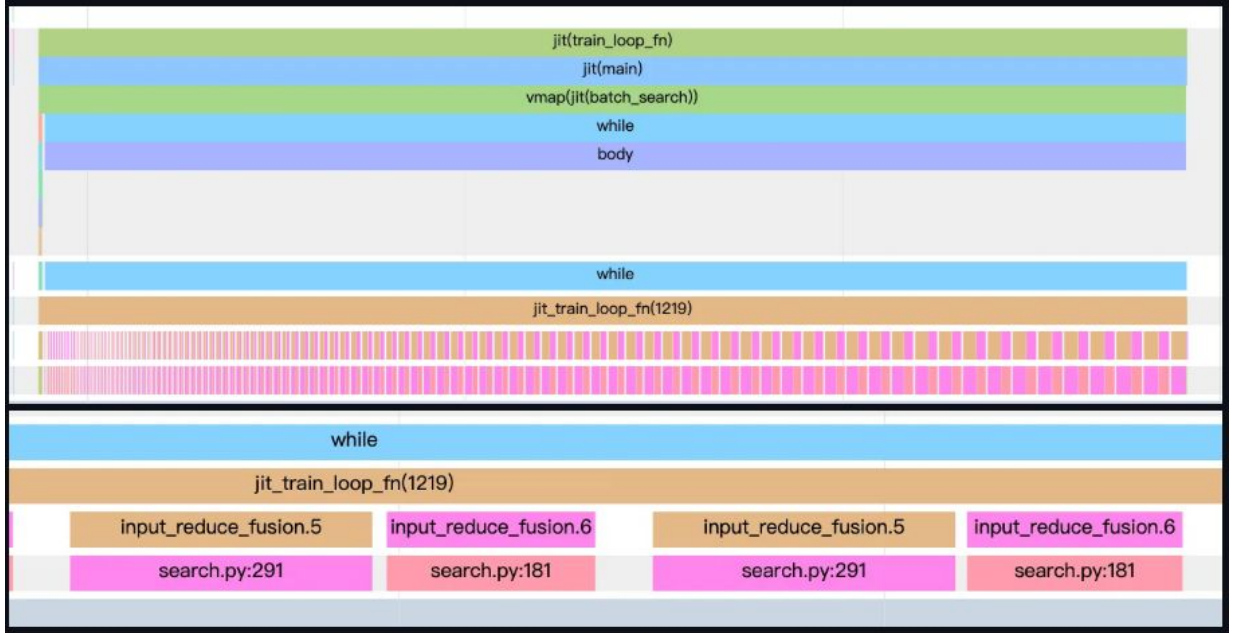
`loop_fn` in `search.py` becomes slow as the tree depth increases #107

I am using mctx to implement MuZero, where actions are selected at each node via a forward pass of a large neural network. Initially, the main bottleneck in terms of time cost is the model forward pass, which is expected and similar to cases where C++-based MCTS is used.

However, after a few steps of training, as the model starts to learn certain biases, the search tree becomes deeper. At this point, the bottleneck shifts to the `loop_fn` used in the selection and backpropagation phases.

Here's the trace of one search. You can see that most of the time is spent in `search.py:291` and `search.py:181`, which correspond to (`search.py` was slightly adjusted to fit my codebase, so the line number mismatches):

- `tree, _, _ = jax.lax.while_loop(cond_fun, body_fun, loop_state)` in `backward`
- `end_state = jax.lax.while_loop(cond_fun, body_fun, initial_state)` in `simulate`



My question is: is there any way to improve the time efficiency of this part?
In practice, even with a tree depth of only a few tens, the performance is heavily affected by the huge gap in efficiency between `jax.lax.fori_loop` on GPU and CPU (with GPU being thousands of times slower).

GPU 와 MCTS - 느려져? 빠르게 만들면 되잖아?

- 직렬화 때문에 느려져? GPU에서도 빠르게 알고리즘 고치면 되잖아?

- 왼쪽 연구인 Nested Monte Carlo Search 는 해당 GPU의 특성을 분석해 MCTS 가 GPU에서 동작할때 느려지는 부분에 맞춰 알고리즘을 새로 변경
- 해당 변경 이후 싱글 CPU 코어와 비교해 GPU에서 돌리는 서치의 속도가 420 배 정도 향상 되었다고 주장

- 이 외에도 GPU 의 하드웨어 적 차이 문제를 해결하려는 많은 알고리즘적 변경 시도가 있음

- 그렇다면 JAX 기반 MCTS 코드에 GPU용 알고리즘 변경들을 이식할수는 없을까?
- 해봐야 알겠지만, 험난할 것으로 예상

GPU for Monte Carlo Search

Lilian Buzer[1] and Tristan Cazenave[2]

[1] LIGM, Universite Gustave Eiffel, CNRS, F-77454 Marne-la-Vallee, France
[2] LAMSADE, Universite Paris Dauphine - PSL, CNRS, Paris, France

Abstract. Monte Carlo Search algorithms give excellent results for some combinatorial optimization problems and for some games. They can be parallelized efficiently on high-end CPU servers. Nested Monte Carlo Search is an algorithm that parallelizes well. We take advantage of this property to obtain large speedups running it on low cost GPUs. The combinatorial optimization problem we use for the experiments is the Snake-in-the-Box. It is a graph theory problem for which Nested Monte Carlo Search previously improved lower bounds. It has applications in electrical engineering, coding theory, and computer network topologies. Using a low cost GPU, we obtain speedups as high as 420 compared to a single CPU.

Keywords: Monte Carlo Search · GPU · Playouts.

감사합니다