AI in Games: Achievements and Challenges

Yuandong Tian
Facebook AI Research

Game as a Vehicle of AI

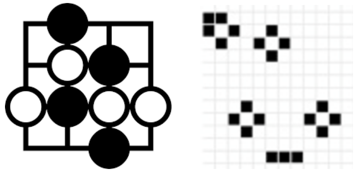
Infinite supply of *fully* labeled data

Controllable and replicable

Low cost per sample

Faster than real-time

Less safety and ethical concerns

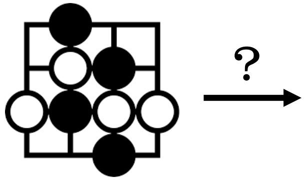


Complicated dynamics with simple rules.

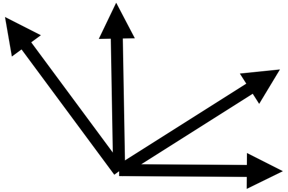
Game as a Vehicle of AI

Algorithm is slow and data-inefficient

Require a lot of resources.



Abstract game to real-world

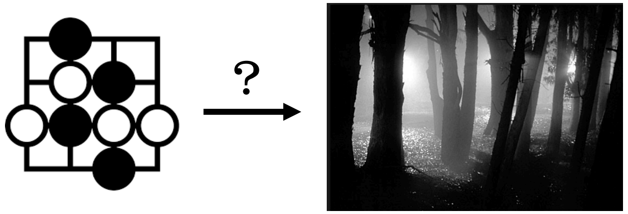


Hard to benchmark the progress

Game as a Vehicle of AI

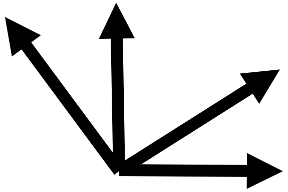
Algorithm is slow and data-inefficient

Require a lot of resources.



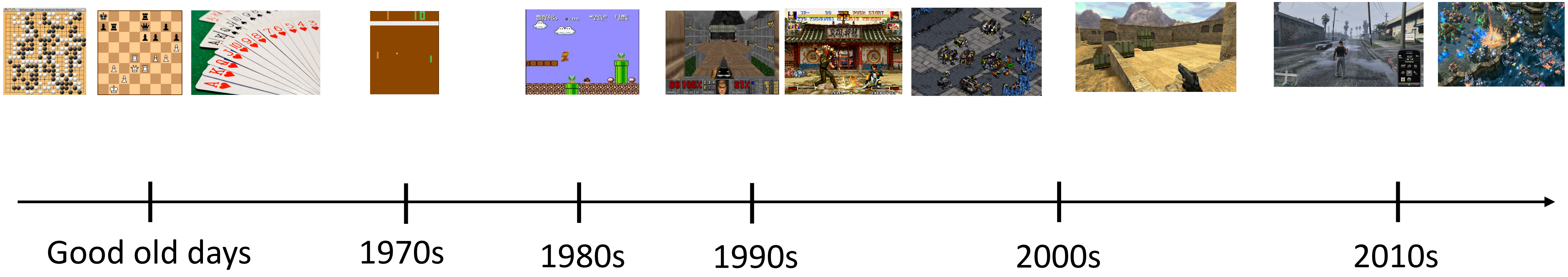
Abstract game to real-world

Better Games



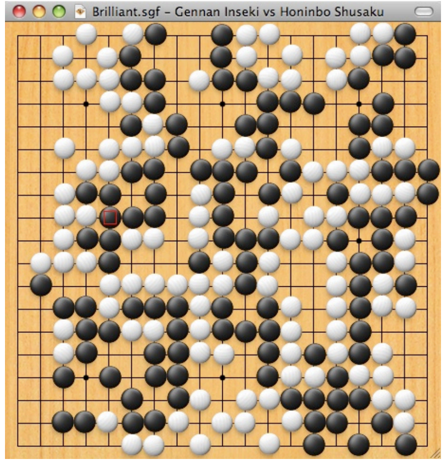
Hard to benchmark the progress

Game Spectrum

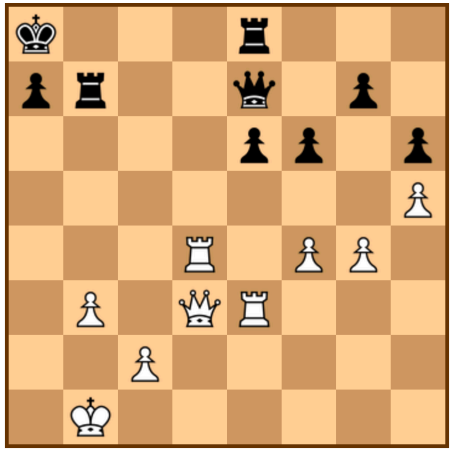


Game Spectrum

Good old days — 1970s — 1980s — 1990s — 2000s — 2010s



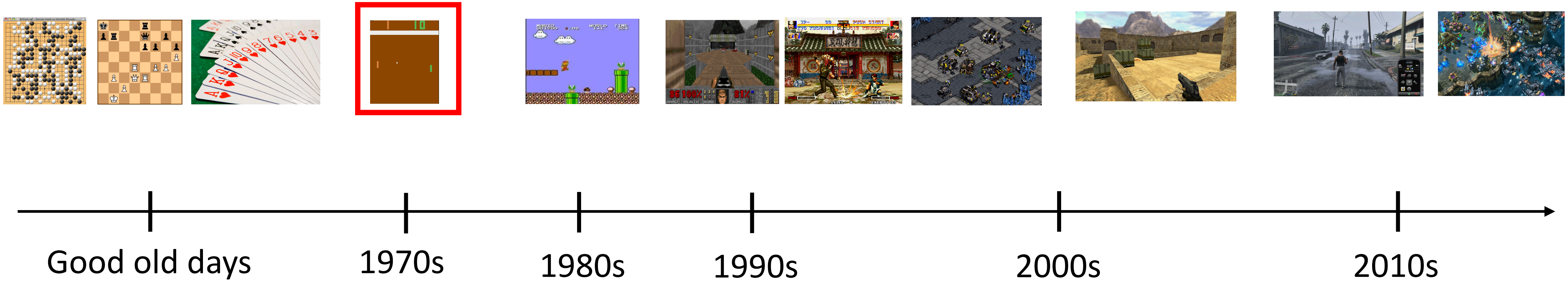
Go

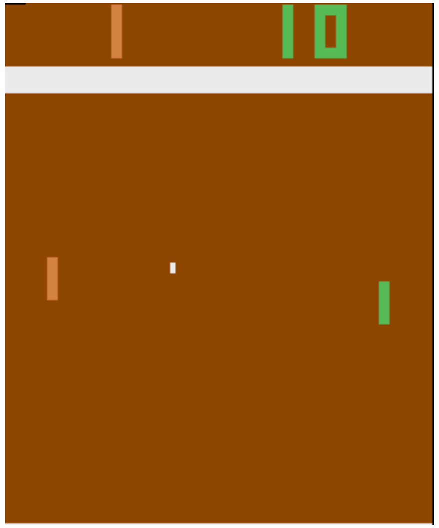


Chess

Poker

Game Spectrum

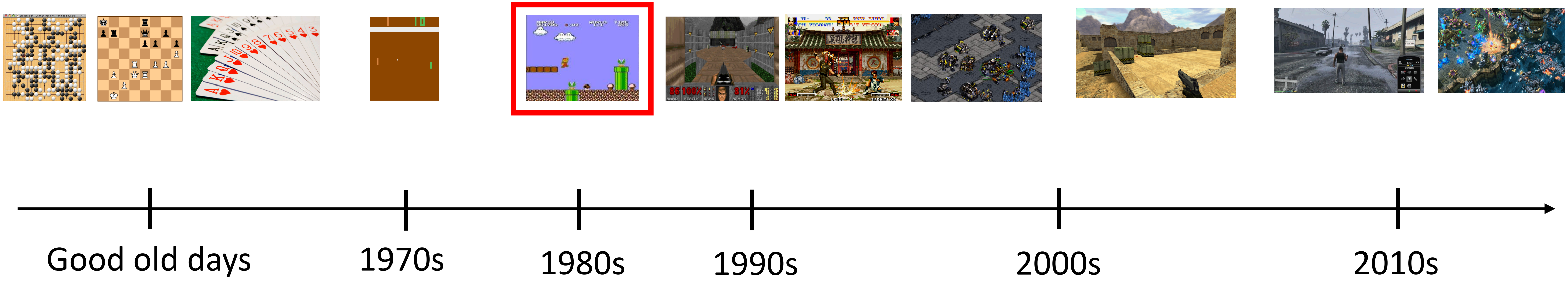




Pong (1972)

Breakout (1978)

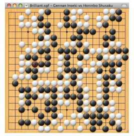
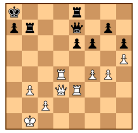
Game Spectrum

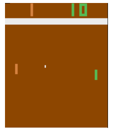


Super Mario Bro (1985)

Contra (1987)

Game Spectrum






Good old days 1970s 1980s 1990s 2000s 2010s

Doom (1993)

KOF’94 (1994)

StarCraft (1998)

Game Spectrum

Good old days | 1970s | 1980s | 1990s | 2000s | 2010s

Counter Strike (2000)

The Sims 3 (2009)

Game Spectrum

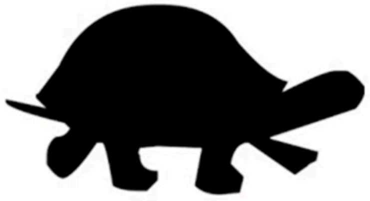
Good old days — 1970s — 1980s — 1990s — 2000s — 2010s

StarCraft II (2010)

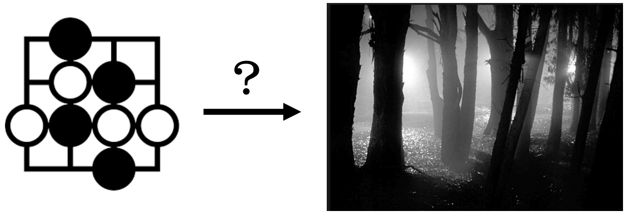
GTA V (2013)

Final Fantasy XV (2016)

Game as a Vehicle of AI

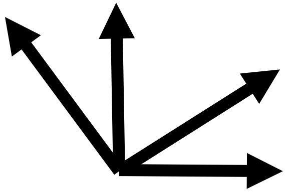


Algorithm is slow and data-inefficient



Abstract game to real-world

Require a lot of resources.



Hard to benchmark the progress

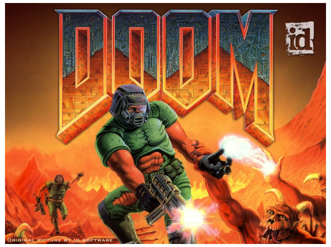
Better Algorithm/System

Better Environment

Our work

Better Algorithm/System

DarkForest Go Engine
(Yuandong Tian, Yan Zhu, ICLR16)



Doom AI
(Yuxin Wu, Yuandong Tian, ICLR17)

Better Environment

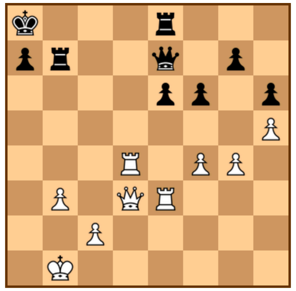
ELF: Extensive Lightweight and Flexible Framework
(Yuandong Tian et al, arXiv)

How Game AI works

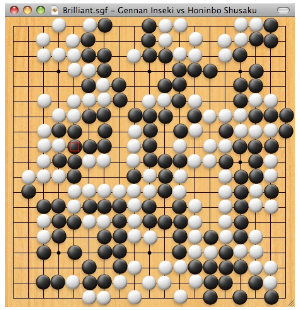
Even with a super-super computer,
it is not possible to search the entire space.

How Game AI works

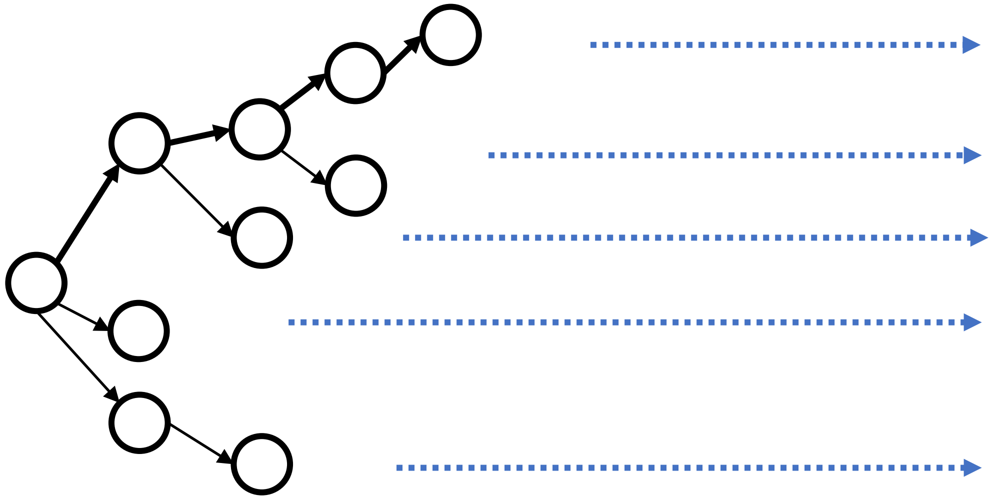
Even with a super-super computer,
it is not possible to search the entire space.



Lufei Ruan vs. Yifan Hou (2010)



Current game situation



Black wins

White wins

Black wins

White wins

Black wins

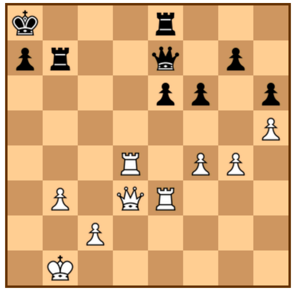
Extensive search ***Evaluate*** ***Consequence***

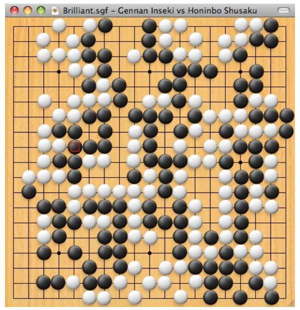
How Game AI works

How many action do you have per step?

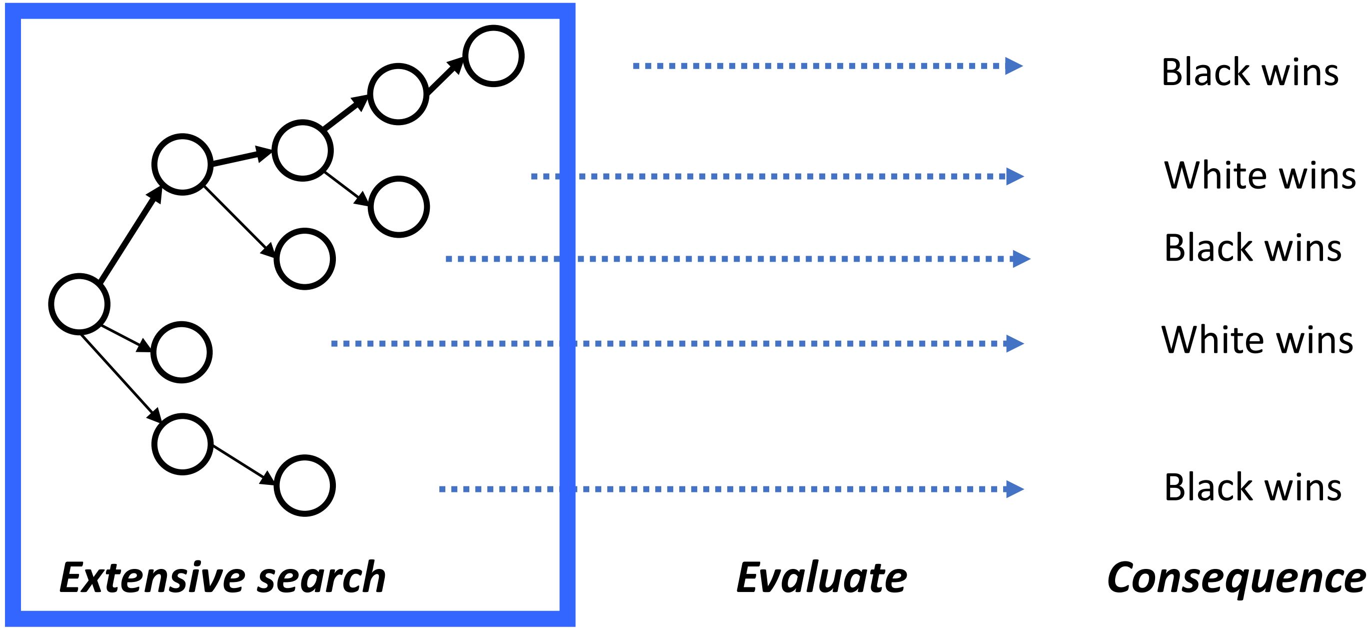
Checker: a few possible moves
Poker: a few possible moves
Chess: 30-40 possible moves
Go: 100-200 possible moves
StarCraft: 50^100 possible moves

Alpha-beta pruning + Iterative deepening [Major Chess engine]
Counterfactual Regret Minimization [Libratus, DeepStack]
Monte-Carlo Tree Search + UCB exploration [Major Go engine]
???





Current game situation

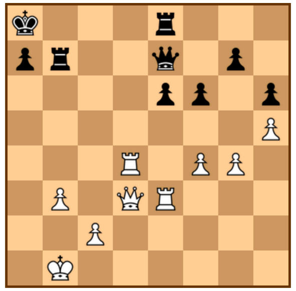


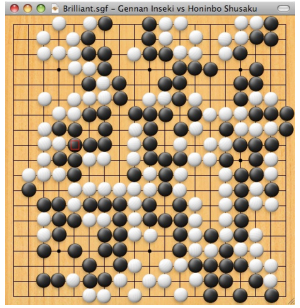
How Game AI works

How complicated is the game situation? How deep is the game?

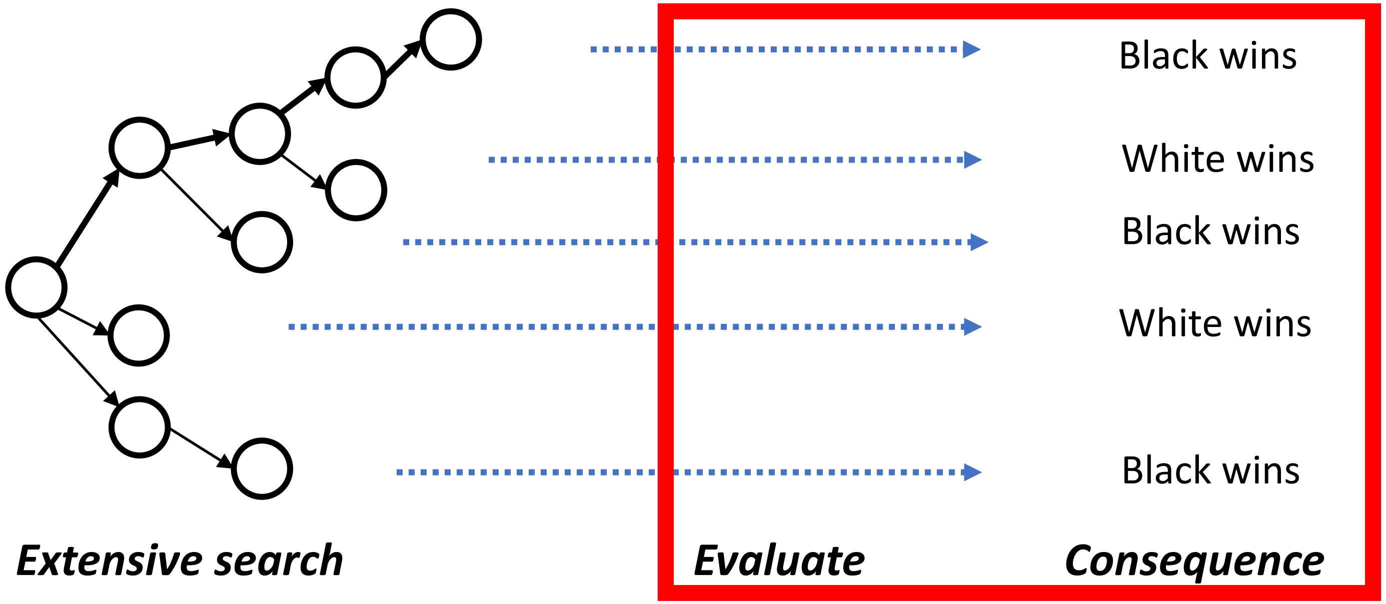
Chess
Go
Poker
StarCraft

Rule-based
Linear function for situation evaluation [Stockfish]
End game database
Random game play with simple rules [Zen, CrazyStone, DarkForest]
Deep Value network [AlphaGo, DeepStack]





Current game situation



Extensive search ***Evaluate*** ***Consequence***

How to model Policy/Value function?

Non-smooth + high-dimensional
Sensitive to situations. One stone changes in Go leads to different game.

Traditional approach

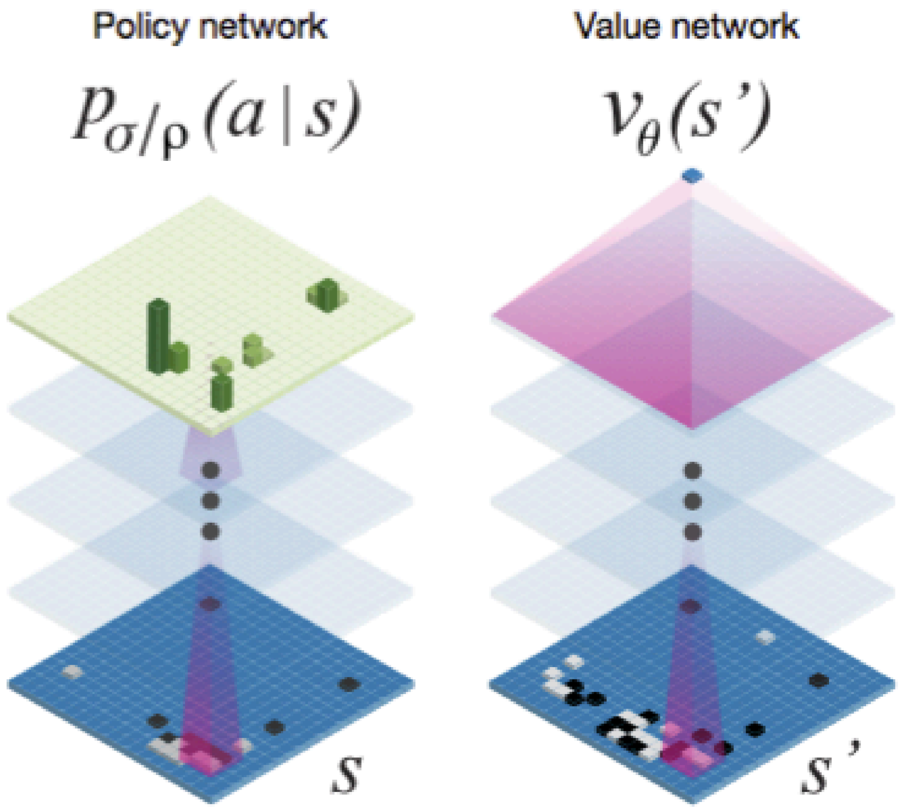
- Many manual steps
- Conflicting parameters, not scalable.
- Need strong domain knowledge.

Deep Learning

- End-to-End training
  - Lots of data, less tuning.
- Minimal domain knowledge.
- Amazing performance

Case study: AlphaGo

- Computations
  - Train with many GPUs and inference with TPU.
- Policy network
  - Trained supervised from human replays.
  - Self-play network with RL.
- High quality playout/rollout policy
  - 2 microsecond per move, ~~24.2% accuracy~~. ~30%
  - Thousands of times faster than DCNN prediction.
- Value network
  - Predicts game consequence for current situation.
  - Trained on 30M self-play games.



"Mastering the game of Go with deep neural networks and tree search", Silver et al, Nature 2016

AlphaGo

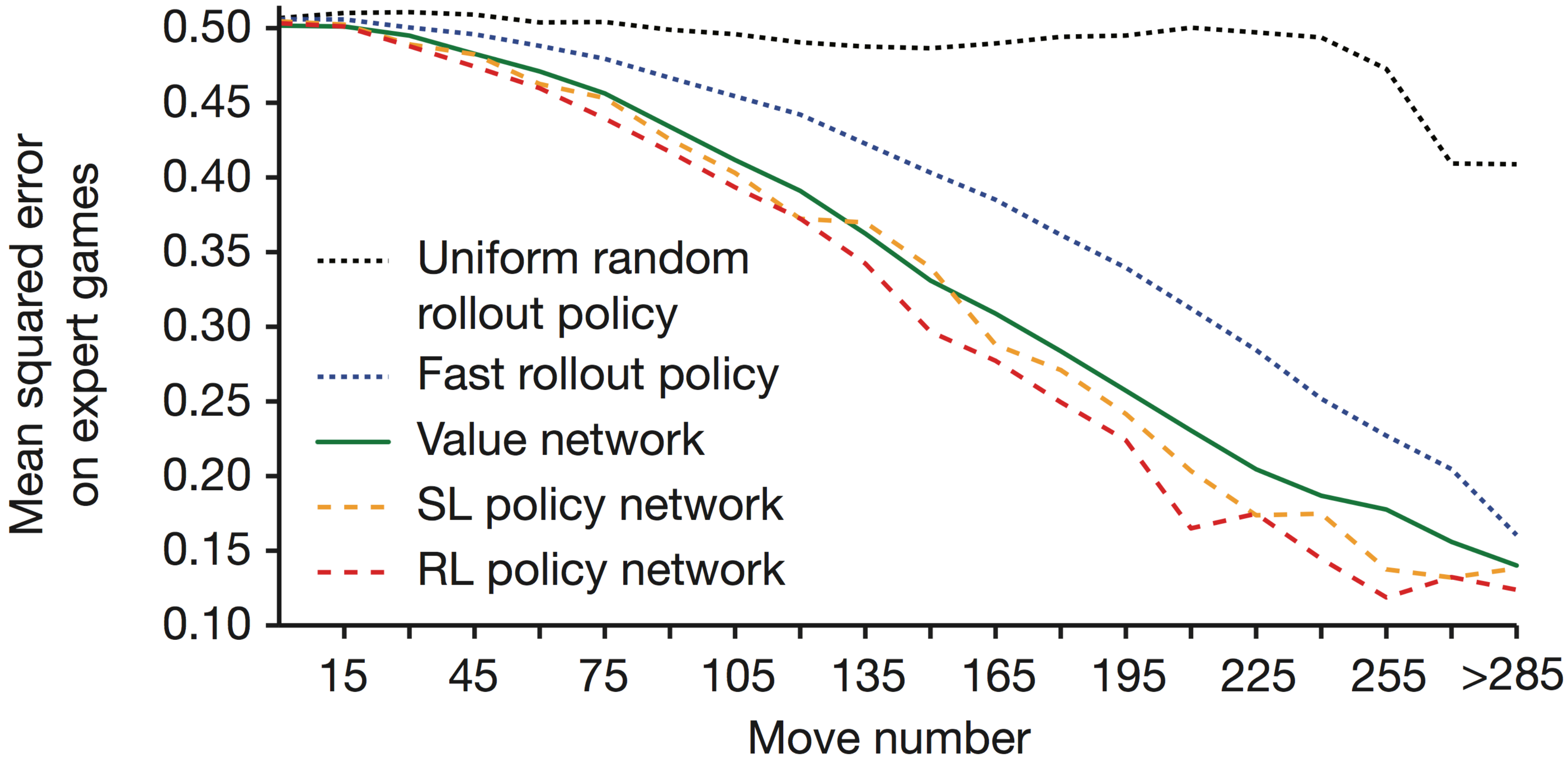
- Policy network SL (trained with human games)

Architecture			Evaluation				
Filters	Symmetries	Features	Test accuracy %	Train accuracy %	Raw net wins %	*AlphaGo* wins %	Forward time (ms)
128	1	48	54.6	57.0	36	53	2.8
192	1	48	55.4	58.0	50	50	4.8
256	1	48	55.9	59.1	67	55	7.1
256	2	48	56.5	59.8	67	38	13.9
256	4	48	56.9	60.2	69	14	27.6
256	8	48	57.0	60.4	69	5	55.3
192	1	4	47.6	51.4	25	15	4.8
192	1	12	54.7	57.1	30	34	4.8
192	1	20	54.7	57.2	38	40	4.8
192	8	4	49.2	53.2	24	2	36.8
192	8	12	55.7	58.3	32	3	36.8
192	8	20	55.8	58.4	42	3	36.8

"Mastering the game of Go with deep neural networks and tree search", Silver et al, Nature 2016

AlphaGo

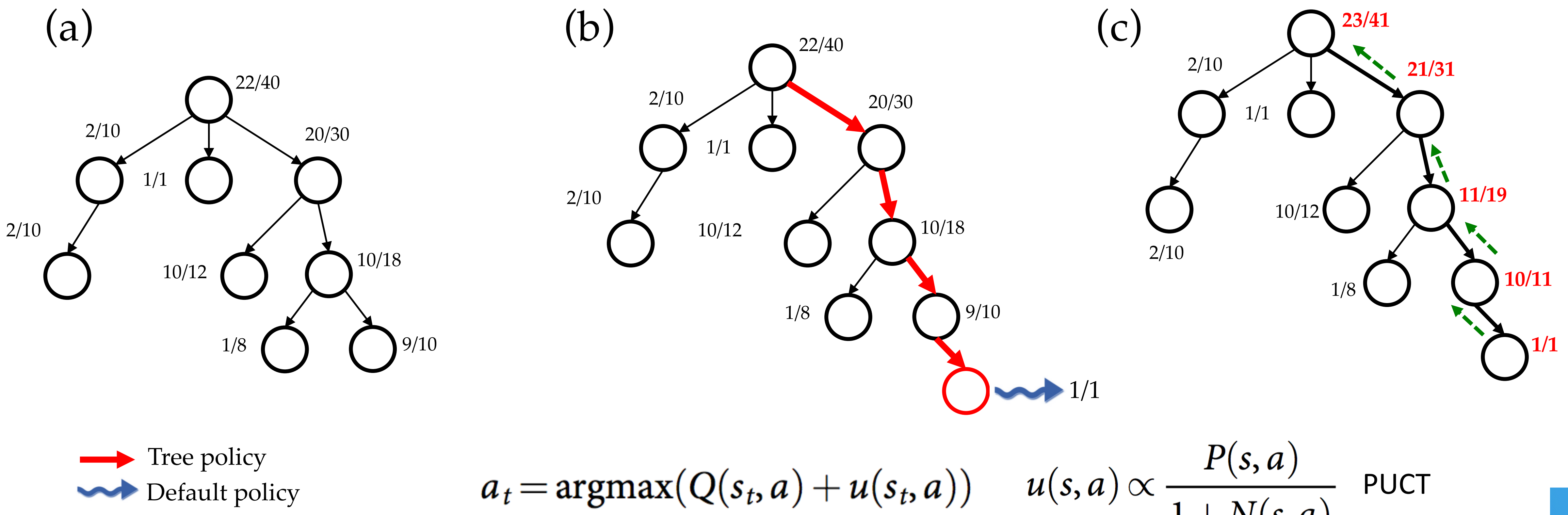
- Fast Rollout (2 microsecond), ~30% accuracy



"Mastering the game of Go with deep neural networks and tree search", Silver et al, Nature 2016

Monte Carlo Tree Search

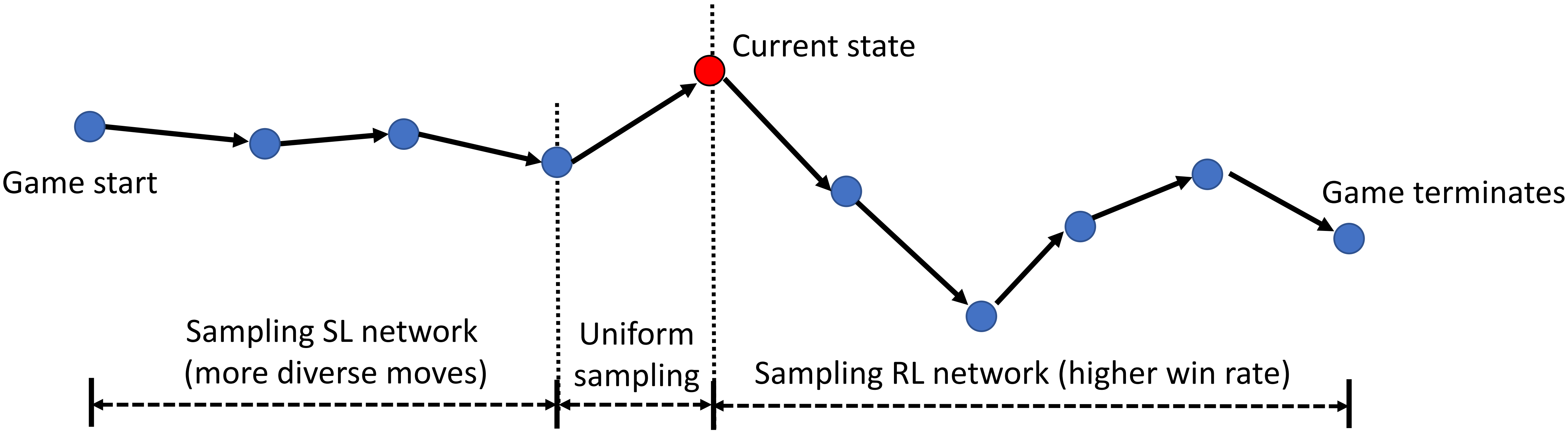
Aggregate win rates, and search towards the good nodes.



$$a_t = \underset{a}{\operatorname{argmax}}(Q(s_t, a) + u(s_t, a)) \qquad u(s,a) \propto \frac{P(s,a)}{1+N(s,a)}$$ PUCT

AlphaGo

- Value Network (trained via 30M self-played games)
- How data are collected?



"Mastering the game of Go with deep neural networks and tree search", Silver et al, Nature 2016

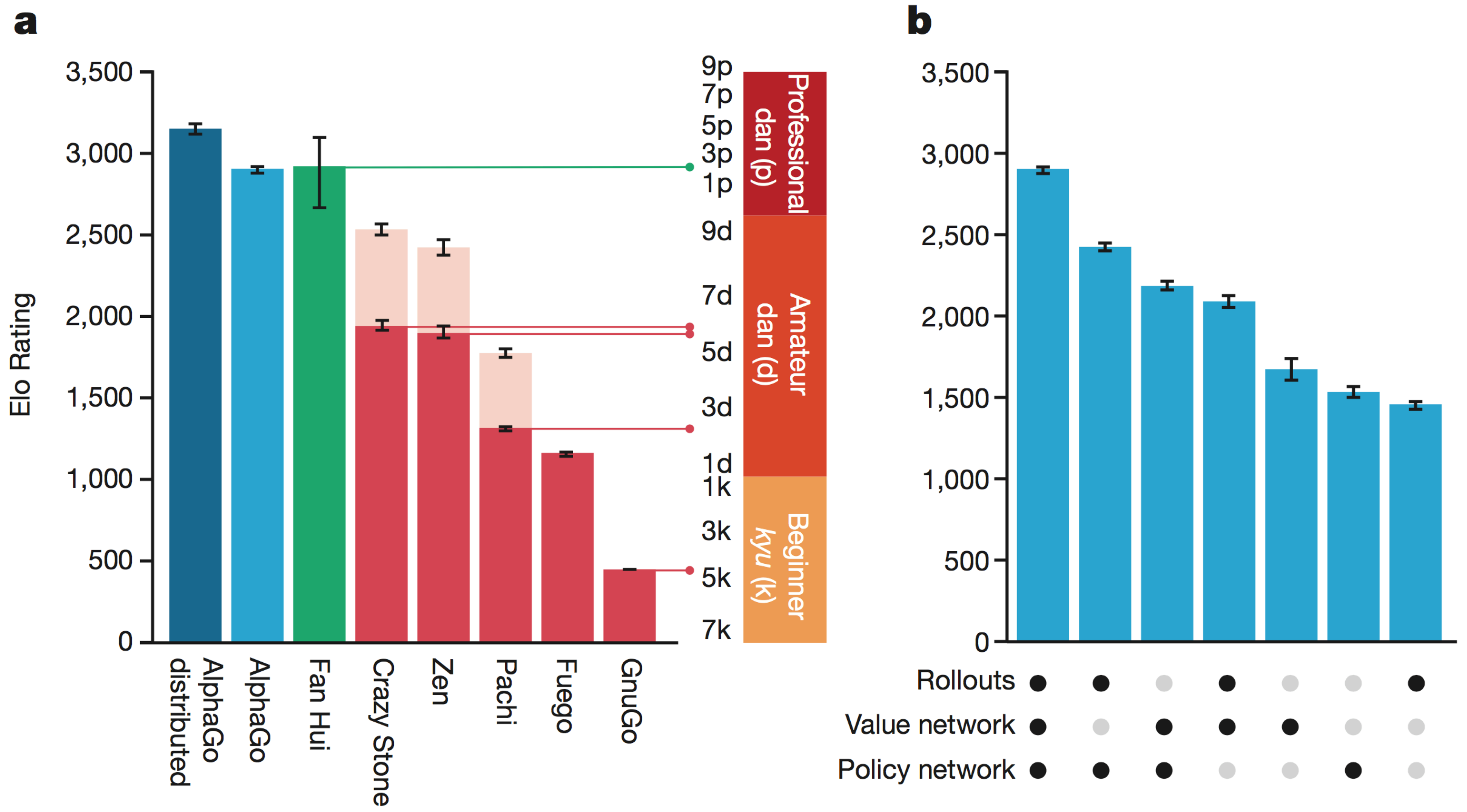
AlphaGo

- Value Network (trained via 30M self-played games)

Short name	Policy network	Value network	Rollouts	Mixing constant	Policy GPUs	Value GPUs	Elo rating
α_{rvp}	p_σ	v_θ	p_π	$\lambda = 0.5$	2	6	2890
α_{vp}	p_σ	v_θ	–	$\lambda = 0$	2	6	2177
α_{rp}	p_σ	–	p_π	$\lambda = 1$	8	0	2416
α_{rv}	$[p_\tau]$	v_θ	p_π	$\lambda = 0.5$	0	8	2077
α_{v}	$[p_\tau]$	v_θ	–	$\lambda = 0$	0	8	1655
α_{r}	$[p_\tau]$	–	p_π	$\lambda = 1$	0	0	1457
α_{p}	p_σ	–	–	–	0	0	1517

"Mastering the game of Go with deep neural networks and tree search", Silver et al, Nature 2016

AlphaGo



"Mastering the game of Go with deep neural networks and tree search", Silver et al, Nature 2016

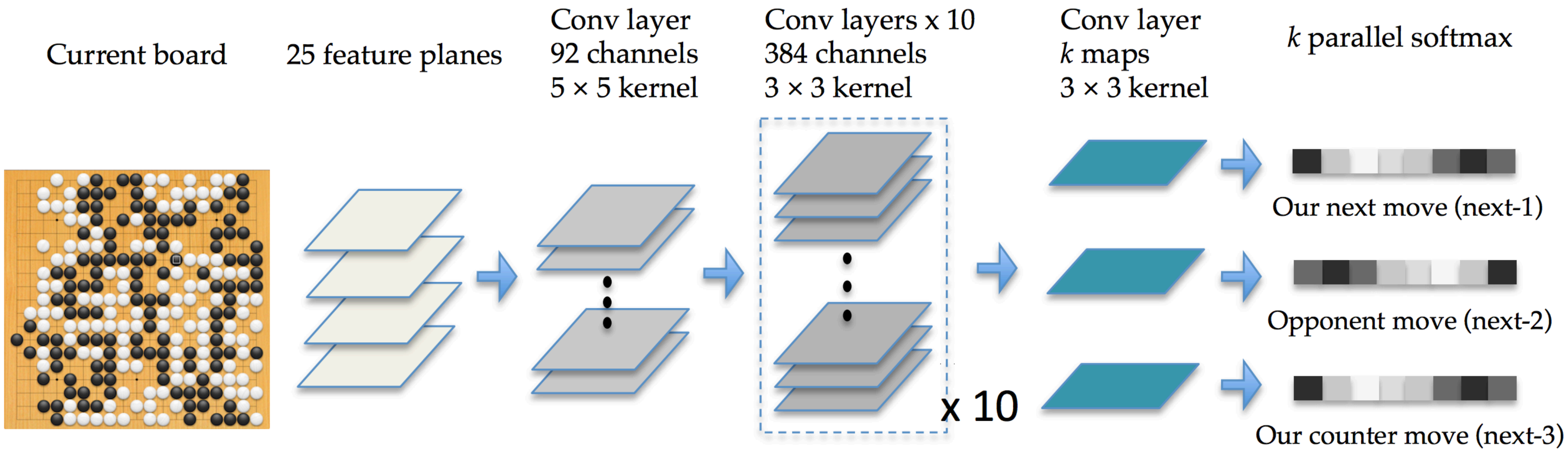
Our work

Our computer Go player: DarkForest

Yuandong Tian and Yan Zhu, ICLR 2016

- DCNN as a tree policy
  - Predict next k moves (rather than next move)
  - Trained on 170k KGS dataset/80k GoGoD, ***57.1%*** accuracy.
  - KGS 3D without search (0.1s per move)
  - Release 3 month before AlphaGo, < 1% GPUs (from Aja Huang)

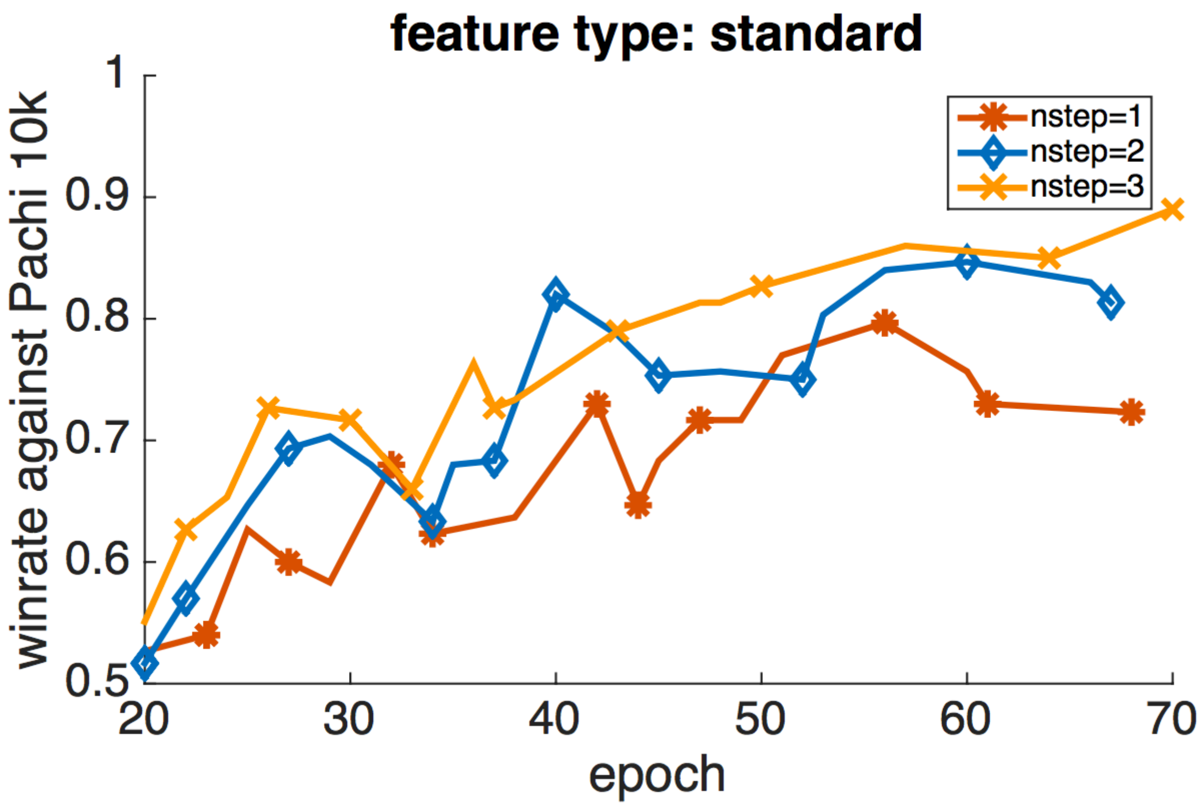
Yan Zhu



Our computer Go player: DarkForest

Name
Our/enemy liberties
Ko location
Our/enemy stones/empty place
Our/enemy stone history
Opponent rank

Feature used for DCNN



Pure DCNN

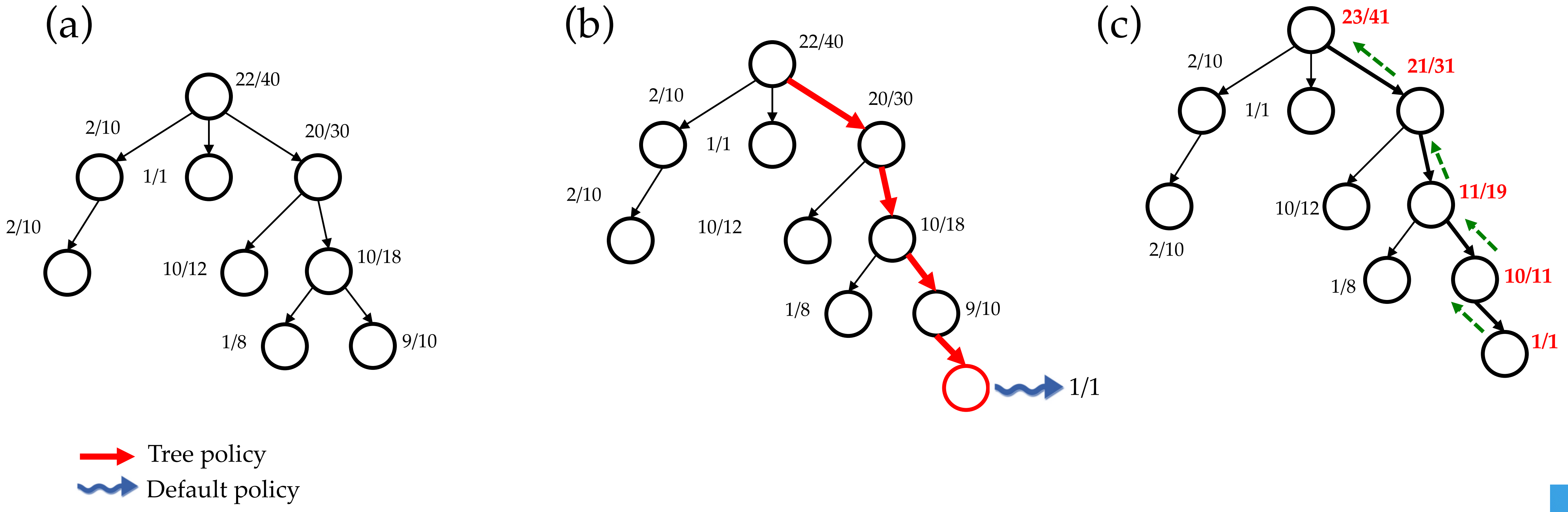
darkforest: Only use top-1 prediction, trained on KGS
darkfores1: Use top-3 prediction, trained on GoGoD
darkfores2: *darkfores1* with fine-tuning.

	GnuGo (level 10)	Pachi 10k	Pachi 100k	Fuego 10k	Fuego 100k
Clark & Storkey (2015)	91.0	-	-	14.0	
Maddison et al. (2015)	97.2	47.4	11.0	23.3	12.5
darkforest	98.0 ± 1.0	71.5 ± 2.1	27.3 ± 3.0	84.5 ± 1.5	56.7 ± 2.5
darkfores1	99.7 ± 0.3	88.7 ± 2.1	59.0 ± 3.3	93.2 ± 1.5	78.0 ± 1.7
darkfores2	$\mathbf{100 \pm 0.0}$	$\mathbf{94.3 \pm 1.7}$	$\mathbf{72.6 \pm 1.9}$	$\mathbf{98.5 \pm 0.1}$	$\mathbf{89.7 \pm 2.1}$

Win rate between DCNN and open source engines.

Monte Carlo Tree Search

Aggregate win rates, and search towards the good nodes.



DCNN + MCTS

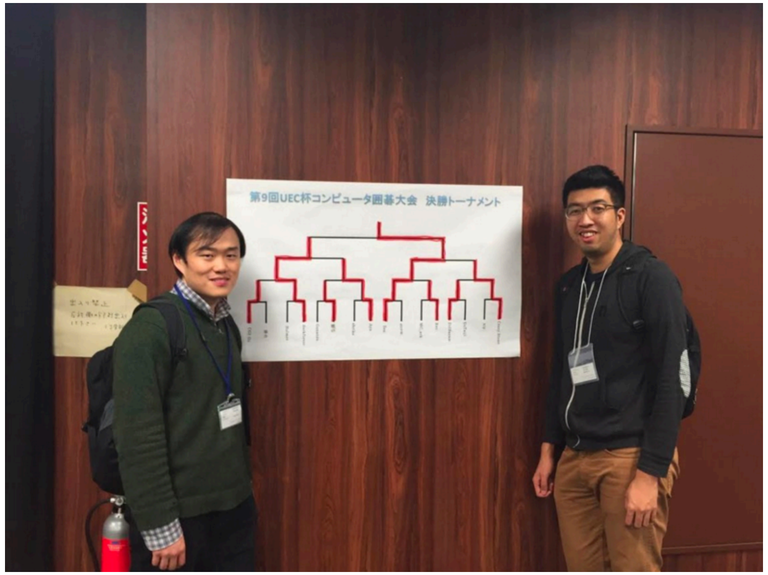
darkfmcts3: Top-3/5, 75k rollouts, ~12sec/move, KGS 5d

	darkforest+MCTS	darkfores1+MCTS	darkfores2+MCTS
Vs pure DCNN (1000rl/top-20)	84.8%	74.0%	62.8%
Vs pure DCNN (1000rl/top-5)	89.6%	76.4%	68.4%
Vs pure DCNN (1000rl/top-3)	91.6%	89.6%	~~79.2%~~ 94.2%
Vs pure DCNN (5000rl/top-5)	96.8%	94.3%	82.3%
Vs Pachi 10k (pure DCNN baseline)	71.5%	88.7%	94.3%
Vs Pachi 10k (1000rl/top-20)	91.2% (+19.7%)	92.0% (+3.3%)	95.2% (+0.9%)
Vs Pachi 10k (1000rl/top-5)	88.4% (+16.9%)	94.4% (+5.7%)	97.6% (+3.3%)
Vs Pachi 10k (1000rl/top-3)	95.2% (+23.7%)	98.4% (+9.7%)	99.2% (+4.9%)
Vs Pachi 10k (5000/top-5)	98.4%	99.6%	100.0%

Win rate between DCNN + MCTS and open source engines.

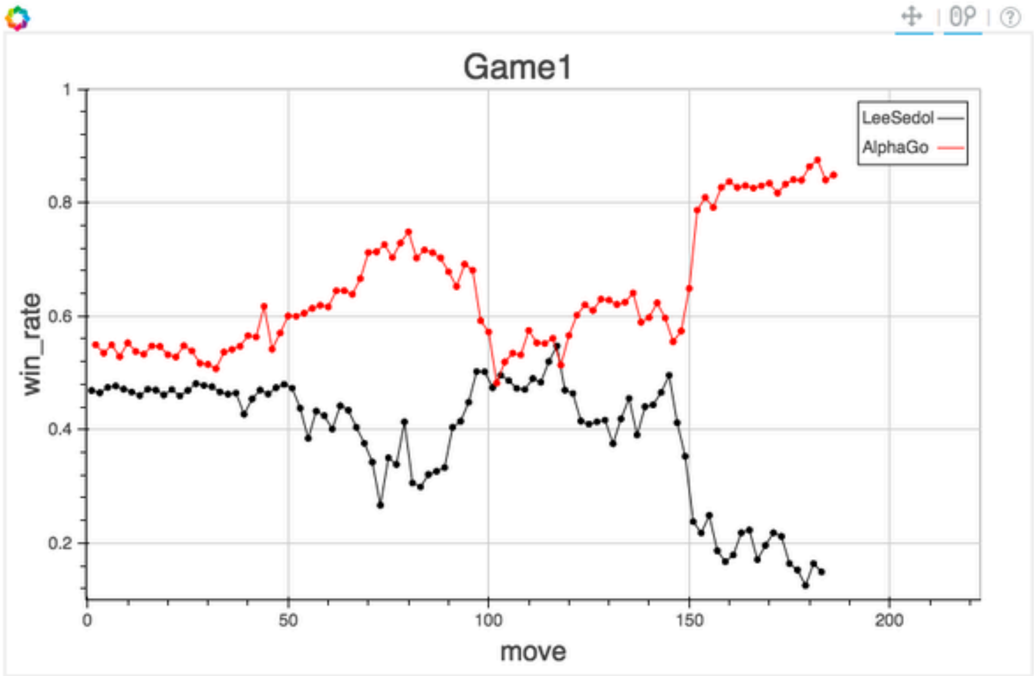
Our computer Go player: DarkForest

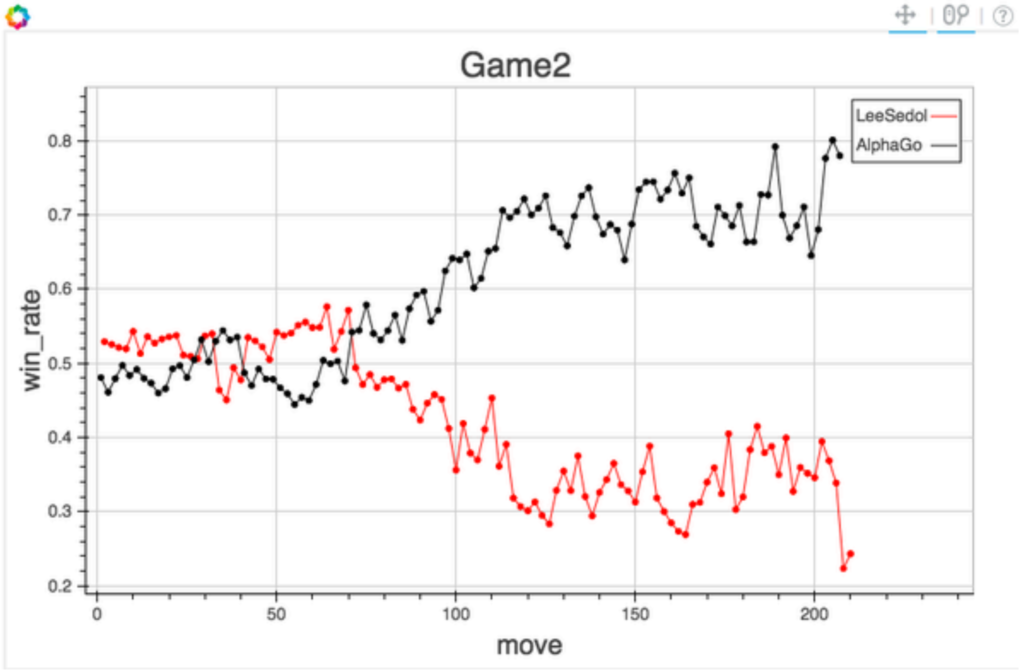
- DCNN+MCTS
  - Use top3/5 moves from DCNN, 75k rollouts.
  - Stable KGS 5d. Open source. https://github.com/facebookresearch/darkforestGo
  - 3rd place on KGS January Tournaments
  - 2nd place in 9th UEC Computer Go Competition (Not this time ☺)



DarkForest versus Koichi Kobayashi (9p)

Win Rate analysis (using DarkForest) (AlphaGo versus Lee Sedol)

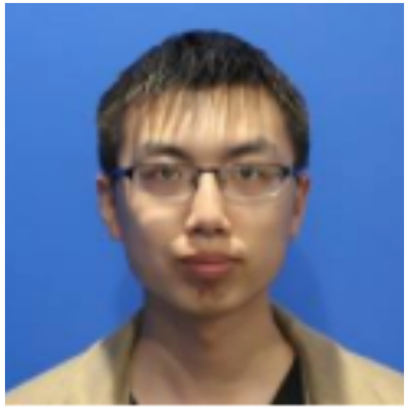




First Person Shooter (FPS) Game

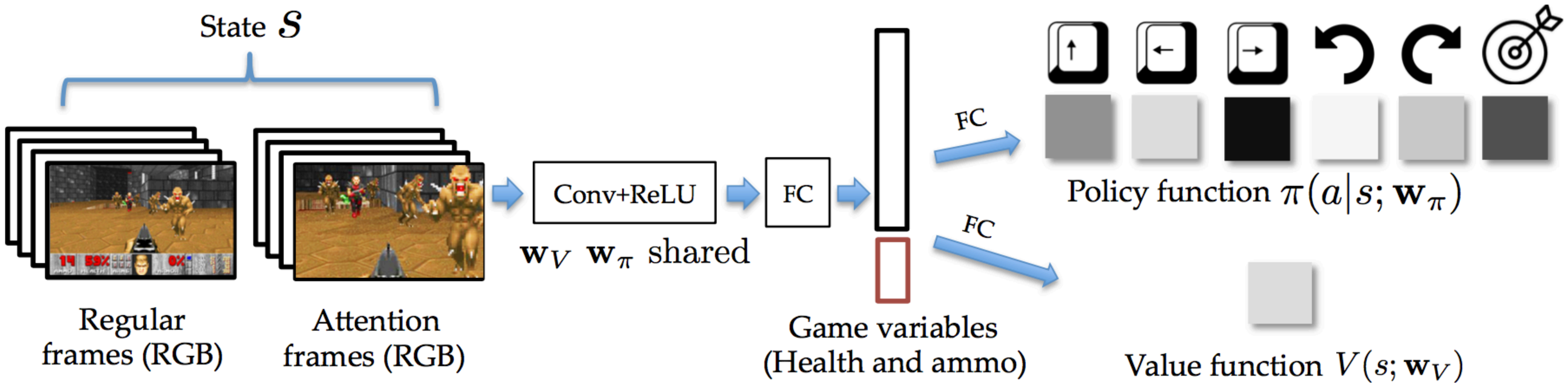
Yuxin Wu, Yuandong Tian, ICLR 2017

Play the game from the raw image!



Yuxin Wu

Network Structure



Simple Frame Stacking is very useful (rather than Using LSTM)

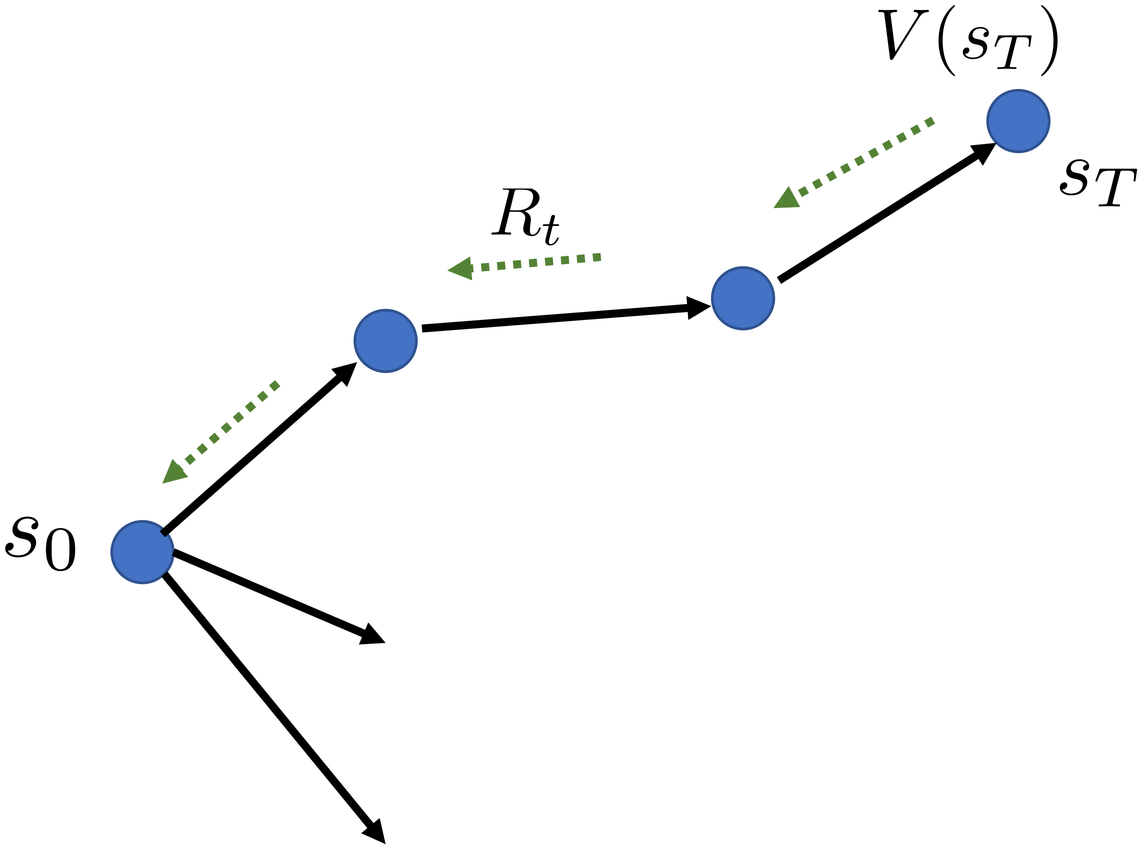
Actor-Critic Models

$$\nabla \log \pi(a|s_t)(R_t - V(s_t))$$

<u>*Update* Policy network</u> → Reward

<u>***Update* Value network**</u>

$$(R_t - V(s_t))\nabla V(s_t)$$

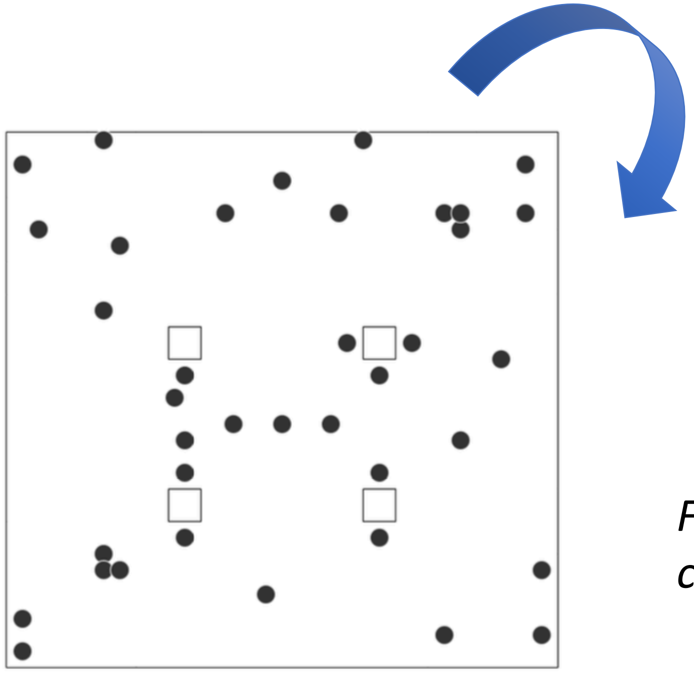


Encourage actions leading to states with high-than-expected value.

Encourage value function to converge to the true cumulative rewards.

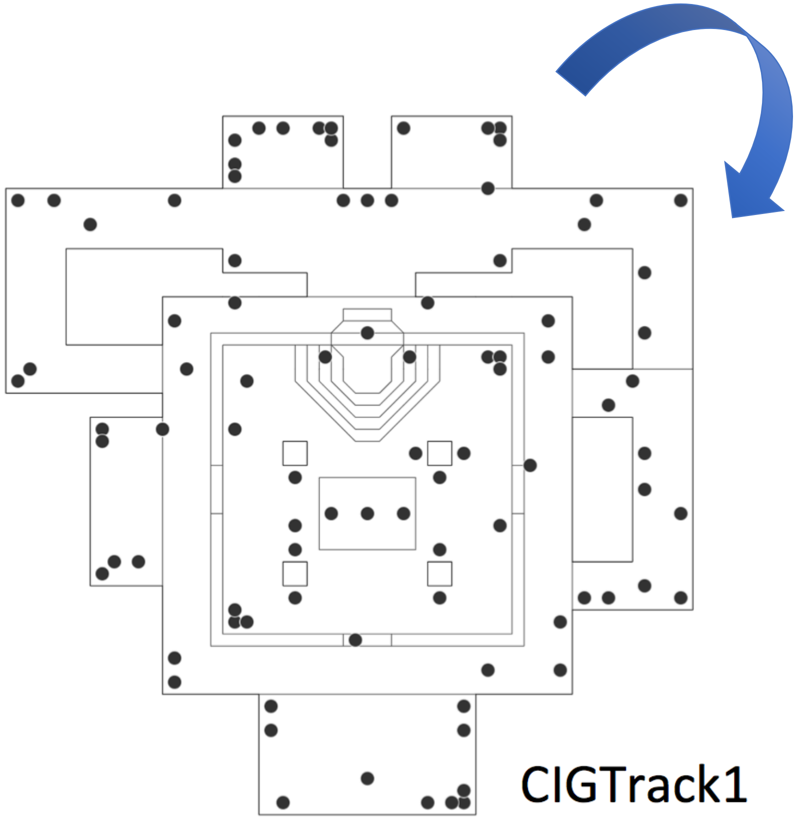
Keep the diversity of actions

Curriculum Training



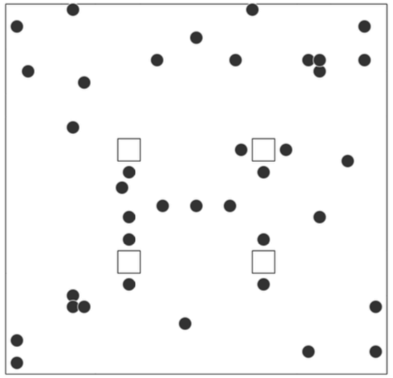
FlatMap

From simple to complicated



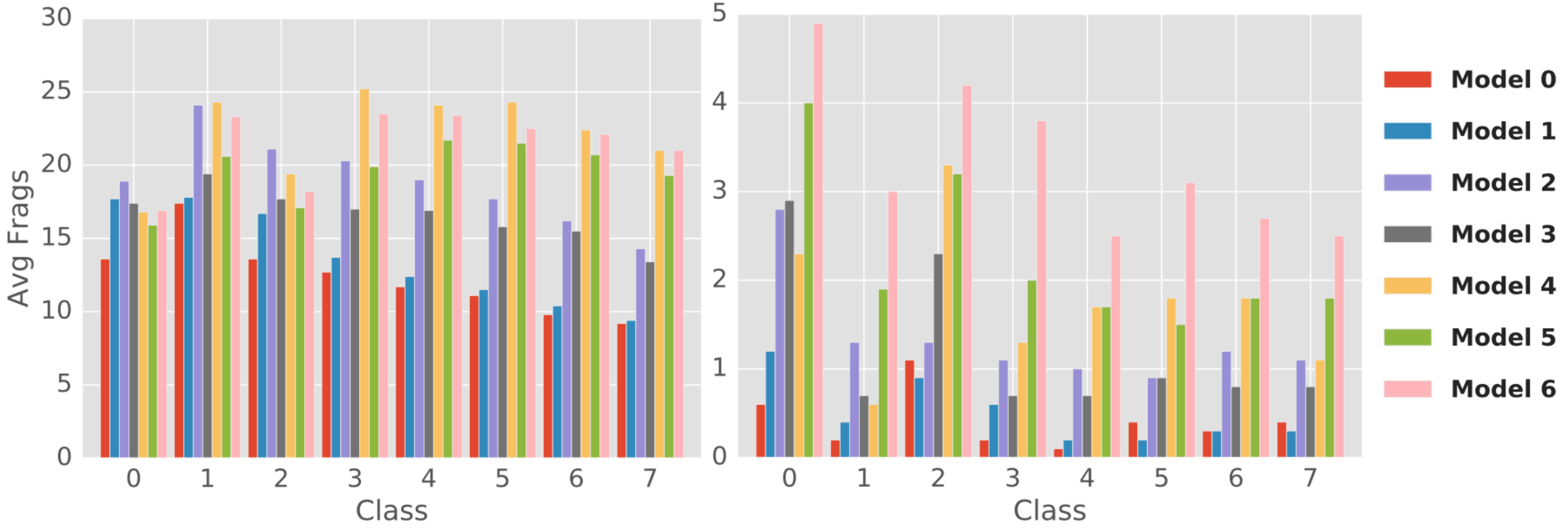
CIGTrack1

Curriculum Training



FlatMap

	Class 0	Class 1	Class 2	Class 3	Class 4	Class 5	Class 6	Class 7
Speed	0.2	0.2	0.4	0.4	0.6	0.8	0.8	1.0
Health	40	40	40	60	60	60	80	100



VizDoom AI Competition 2016 (Track1)

We won the first place!

Rank	Bot	1	2	3	4	5	6	7	8	9	10	11	Total frags
1	F1	**56**	**62**	n/a	**54**	**47**	43	**47**	**55**	**50**	**48**	**50**	**559**
2	Arnold	36	34	**42**	36	36	**45**	36	39	n/a	33	36	413
3	CLYDE	37	n/a	38	32	37	30	46	42	33	24	44	393

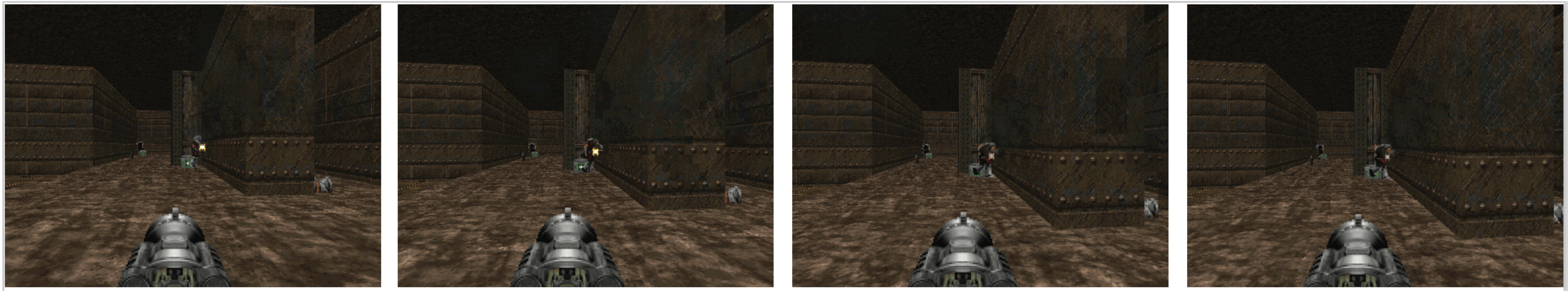
Videos:

https://www.youtube.com/watch?v=94EPSjQH38Y
https://www.youtube.com/watch?v=Qv4esGWOg7w&t=394s

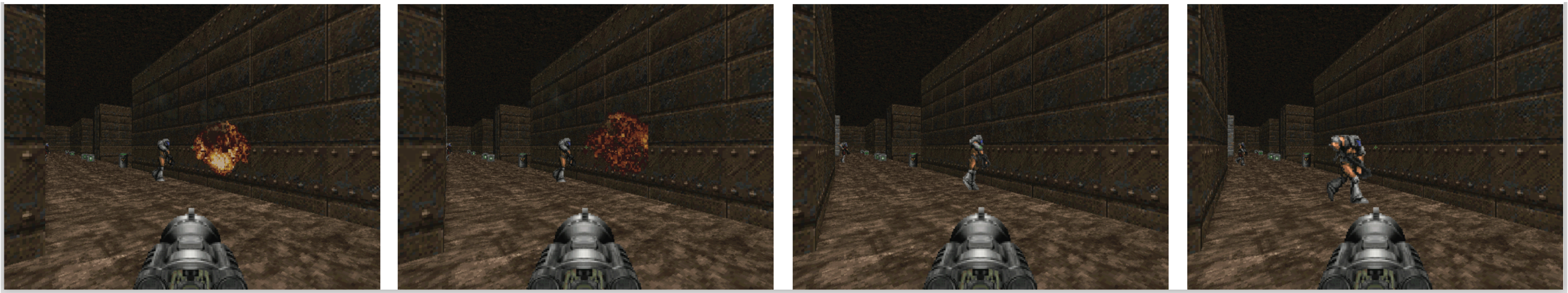

14
93%
1
0%
AMMO
HEALTH
FRAGS
ARMOR
0 / 200
0 / 50
14 / 50
0 / 300

Visualization of Value functions

Best 4 frames (agent is about to shoot the enemy)



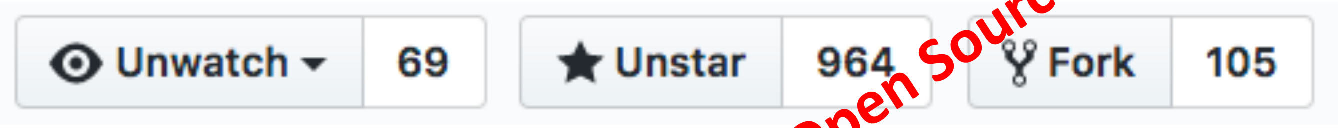
Worst 4 frames (agent missed the shoot and is out of ammo)



ELF: Extensive, Lightweight and Flexible Framework for Game Research

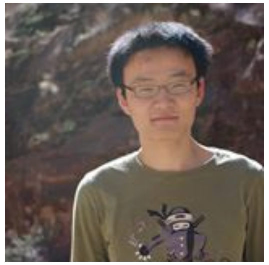
Yuandong Tian, Qucheng Gong, Wendy Shang, Yuxin Wu, Larry Zitnick (Submitted to NIPS 2017)

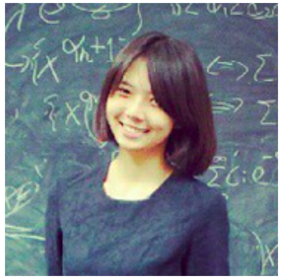
https://github.com/facebookresearch/ELF



Open Sourced!

- Extensive
  - Any games with C++ interfaces can be incorporated.
- Lightweight
  - Fast. Mini-RTS (40K FPS per core)
  - Minimal resource usage (1GPU+several CPUs)
- Flexible
  - Environment-Actor topology
  - Parametrized game environments.
  - Choice of different RL methods.


Qucheng Gong

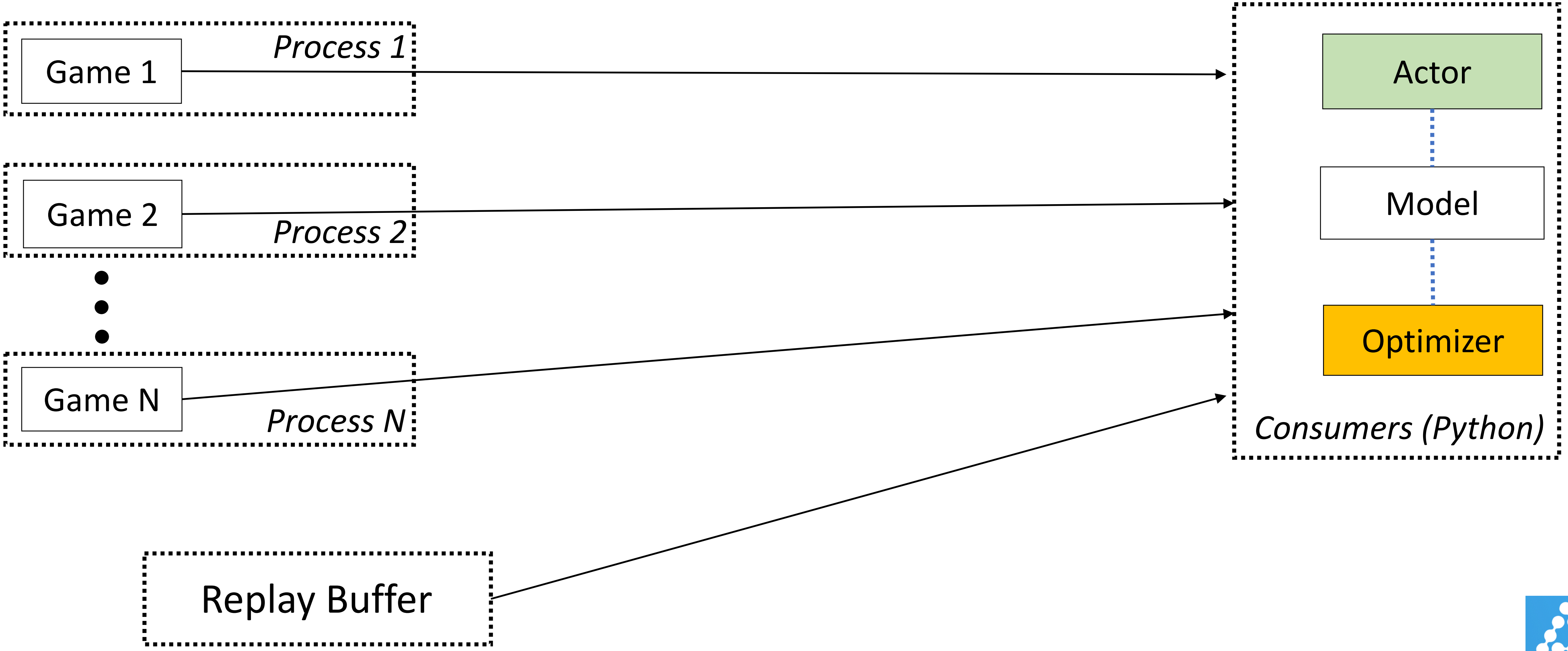

Wendy Shang


Yuxin Wu

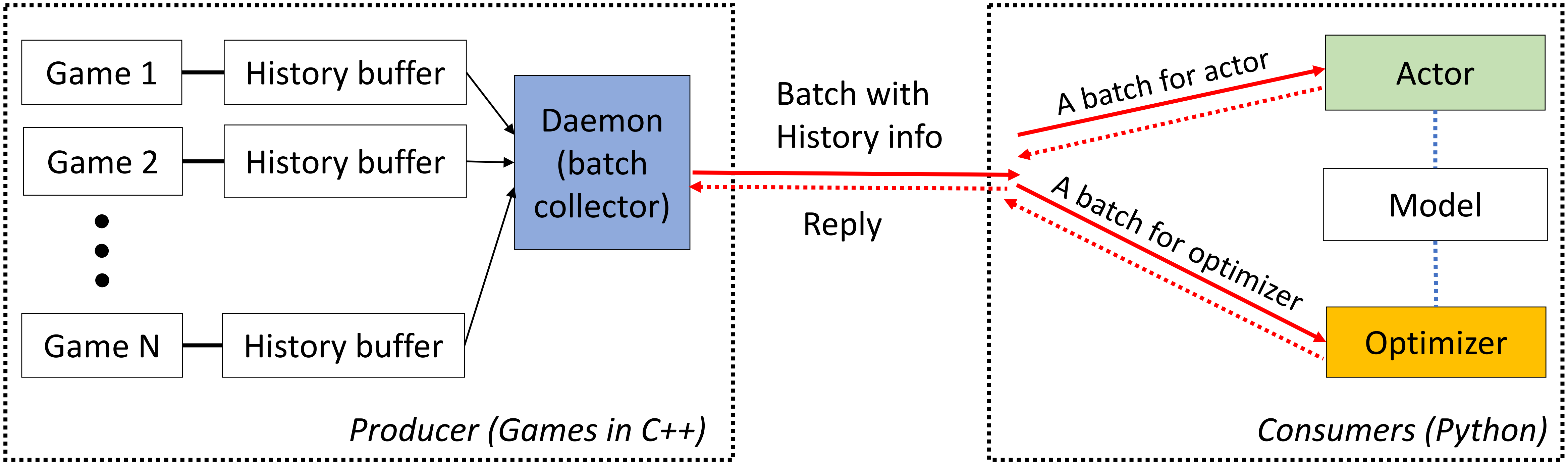

Larry Zitnick

Arxiv: https://arxiv.org/abs/1707.01067

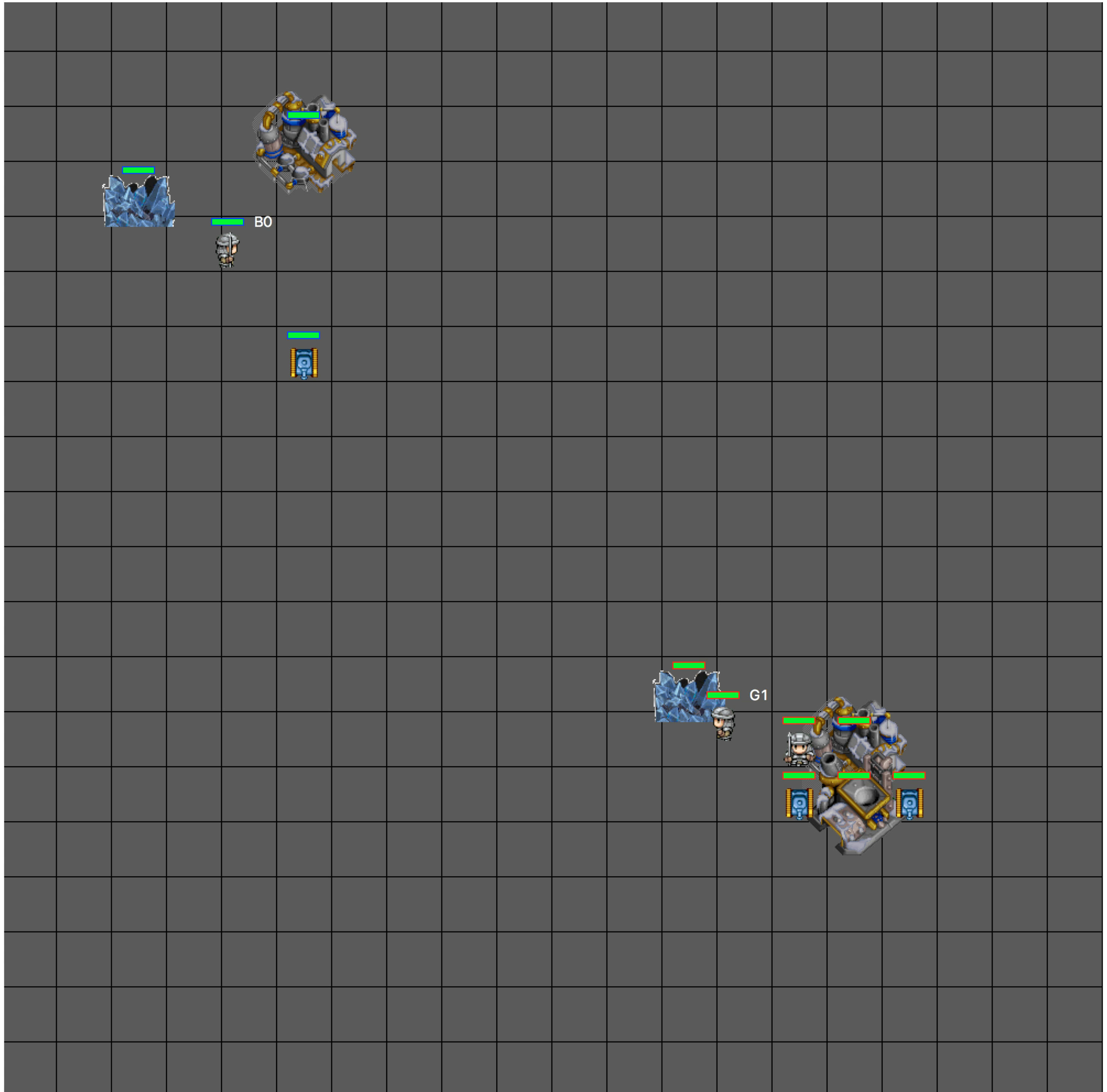
How RL system works



ELF design



Plug-and-play; no worry about the concurrency anymore.


B0
G1

Possible Usage

- Game Research
  - Board game (Chess, Go, etc)
  - Real-time Strategy Game
- Complicated RL algorithms.
- Discrete/Continuous control
  - Robotics
- Dialog and Q&A System

Initialization

```python
# Sample Usage
# We run 1024 games concurrently.
num_games = 1024

# Every time we wait for an arbitrary set of 256 games and return.
batchsize = 256

# The return states contain key 's', 'r' and 'terminal'
# and the reply contains key 'a', 'V' and 'pi', which is to be filled from the Python side.
# Their definitions are defined in the C++ wrapper of the game.
desc = dict(
    actor = dict(
        batchsize=args.batchsize,
        input=dict(T=1, keys=set(["s", "last_r", "last_terminal"])),
        reply=dict(T=1, keys=set(["pi", "V", "a"]))
    )
)

GameContext = InitializeGame(num_games, batchsize, desc)
```

Main Loop

```
# Start all game threads
GameContext.Start()

while True:
    # Wait until a batch of game states are returned.
    # Note that these game instances will be blocked.
    batch = GameContext.Wait()
   if batch.desc == "actor":
       # Apply a model to the game state. you can do forward/backward propagation here.
       output = model(batch)

       # Sample from the output to get the actions of this batch.
       reply['pi'][:] = output['pi']
       reply['a'][:] = SampleFromDistribution(output)
       reply['V'][:] = output["V"]

    # Resume games.
    GameContext.Steps()

# Stop all game threads.
GameContext.Stop()
```

Training

```
desc = dict(
    actor = dict(
        batchsize=args.batchsize,
        input=dict(T=1, keys=set(["s", "last_r", "last_terminal"])),
        reply=dict(T=1, keys=set(["pi", "v", "a"]))
    ),
    train = dict(
        batchsize=args.batchsize,
        input=dict(T=6, keys=set(["s", "last_r", "last_terminal", "a", "pi"])),
        reply=None
    )
)
while True:
    ...
    if batch["desc"] == "actor":
        # Act given the current states to move the game environment forward.
        # It could be an act for a game, for its internal MCTS search, etc.
    elif batch["desc"] == "train":
        # Train your model. All the previous actions of the games and
        # their probabilities can be made available.
    ...
```

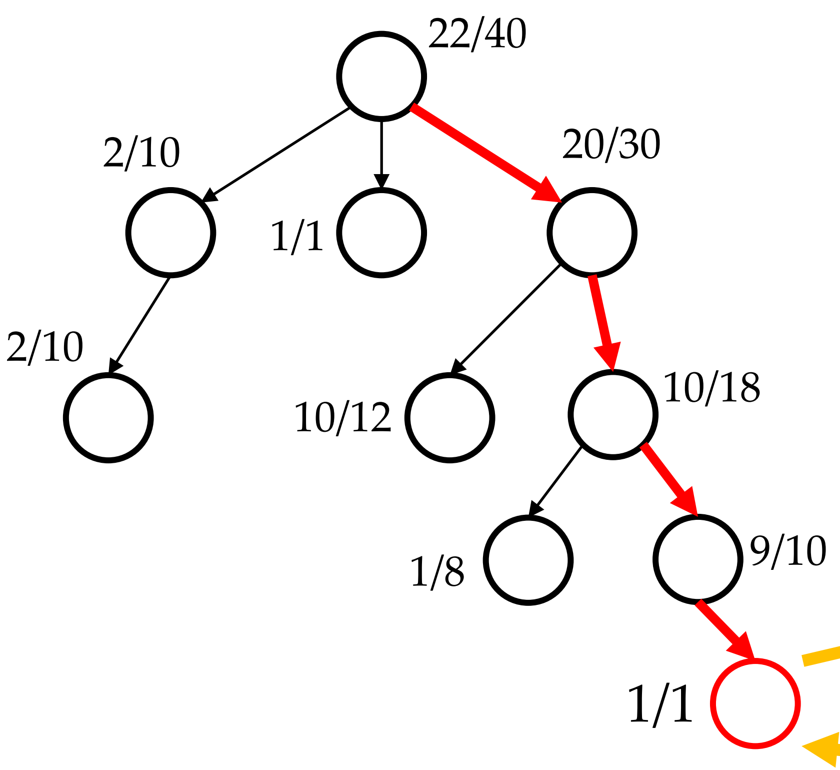
Self-Play

```
desc = dict(
    actor0 = dict(
        batchsize=args.batchsize,
        input=dict(T=1, keys=set(["s", "last_r", "last_terminal"])),
        reply=dict(T=1, keys=set(["pi", "v", "a"])),
        filter=dict(id=0)
    ),
    actor1 = dict(
        batchsize=args.batchsize,
        input=dict(T=1, keys=set(["s", "last_r", "last_terminal"])),
        reply=dict(T=1, keys=set(["pi", "v", "a"])),
        filter=dict(id=1)
    ),
    train = dict(
        batchsize=args.batchsize,
        input=dict(T=6, keys=set(["s", "last_r", "last_terminal", "a", "pi"])),
        reply=None,
        filter=dict(id=0)
    )
)
while True:
    ...
    if batch["desc"] == "actor0":
        # Act for player 0
    elif batch["desc"] == "actor1":
        # Act for player 1
    elif batch["desc"] == "train":
        # Train your model only for player 0.
    ...
```

Multi-Agent

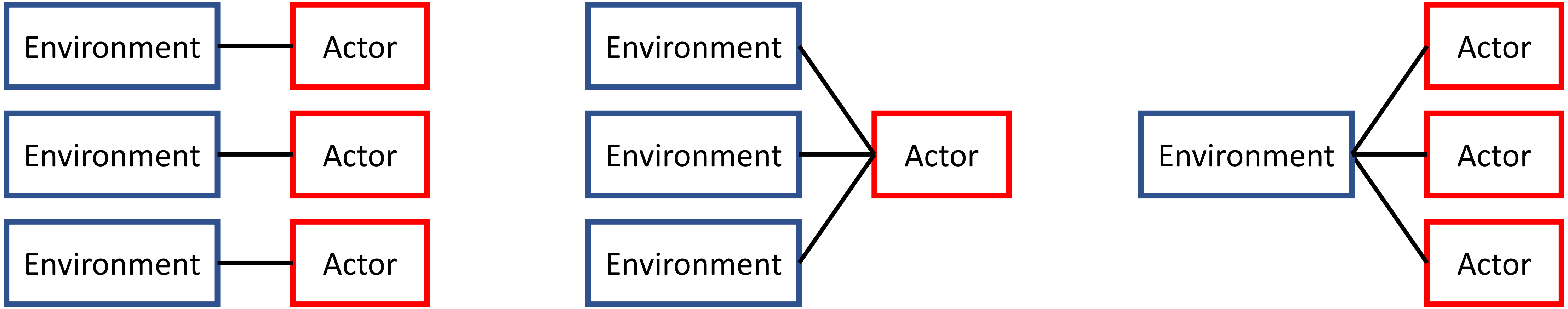
```
desc = { }
for i in range(num_agents):
    desc["actor%d" % i] = dict(
        batchsize=args.batchsize,
        input=dict(T=1, keys=set(["s", "last_r", "last_terminal"])),
        reply=dict(T=1, keys=set(["pi", "v", "a"])),
        filter=dict(id=i)
    )
while True:
    ...
    for i in range(num_agents):
        if batch["desc"] == "actor%d" % i:
            # Act for player i
    ...
```

Monte-Carlo Tree Search




```python
desc = dict(
    actor = dict(
        batchsize=args.batchsize,
        input=
          dict(T=1,
               keys=set([
                  "s", "last_r", "last_terminal"])),
        reply=dict(T=1, keys=set(["pi", "v", "a"])),
    )
)
while True:
    batch = GameContext.Wait()
    if batch["desc"] == "actor":
        # Act for player. During MCTS search, one
        # game instance could send multiple requests
        # for python side to respond.
    GameContext.Step()
```

Flexible Environment-Actor topology



(a) One-to-One
Vanilla A3C

(b) Many-to-One
BatchA3C, GA3C

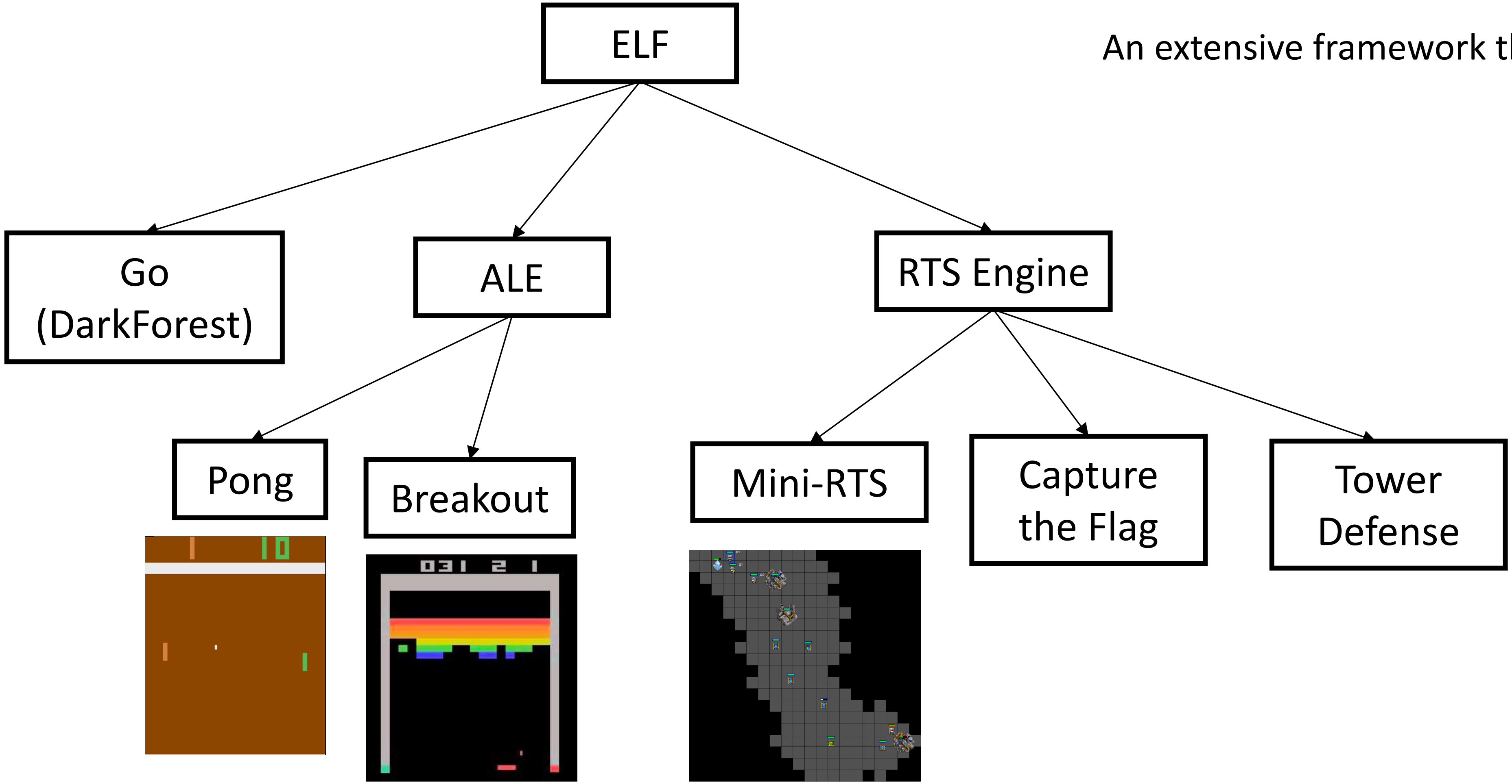
(c) One-to-Many
Self-Play,
Monte-Carlo Tree Search

RLPytorch

- A RL platform in PyTorch
- A3C in 30 lines.
- Interfacing with dict.

```
# A3C
def update(self, batch):
    ''' Actor critic model '''
    R = deepcopy(batch["V"][T - 1])
    batchsize = R.size(0)
    R.resize_(batchsize, 1)

    for t in range(T - 2, -1, -1):
        # Forward pass
        curr = self.model_interface.forward("model", batch.hist(t))

        # Compute the reward.
        R = R * self.args.discount + batch["r"][t]
        # If we see any terminal signal, do not backprop
        for i, terminal in enumerate(batch["terminal"][t]):
            if terminal: R[t][i] = curr["V"].data[i]

        # We need to set it beforehand.
        self.policy_gradient_weights = R - curr["V"].data

        # Compute policy gradient error:
        errs = self._compute_policy_entropy_err(curr["pi"], batch["a"][t])
        # Compute critic error
        value_err = self.value_loss(curr["V"], Variable(R))

        overall_err = value_err + errs["policy_err"]
        overall_err += errs["entropy_err"] * self.args.entropy_ratio
        overall_err.backward()
```

Architecture Hierarchy

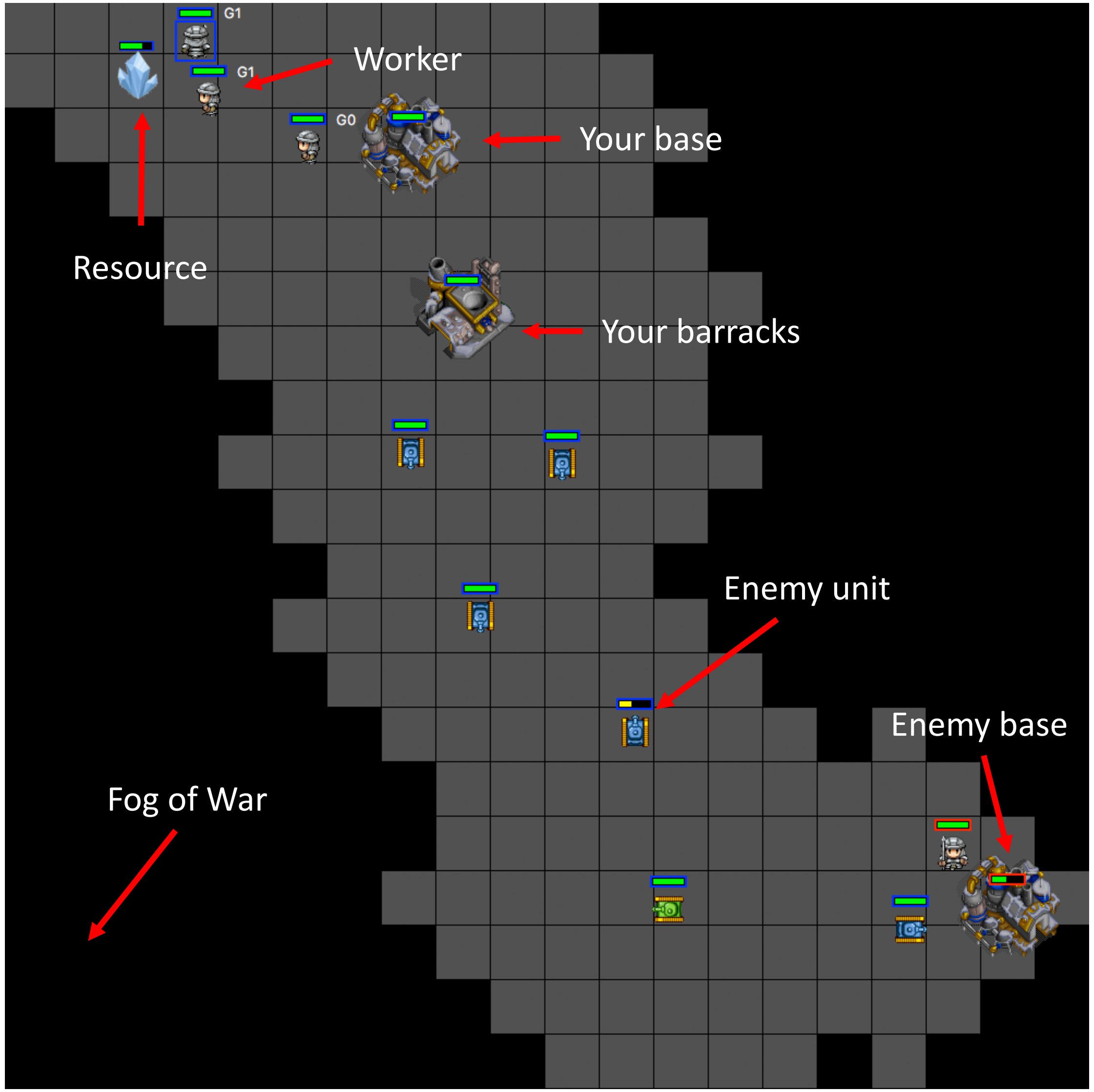


An extensive framework that can host many games.

Specific game engines.

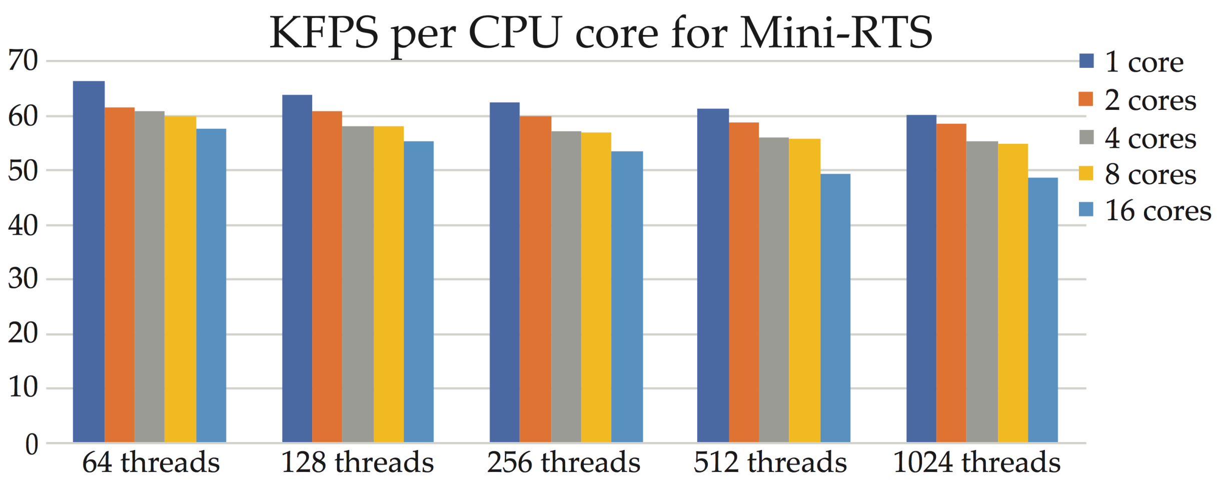
Environments

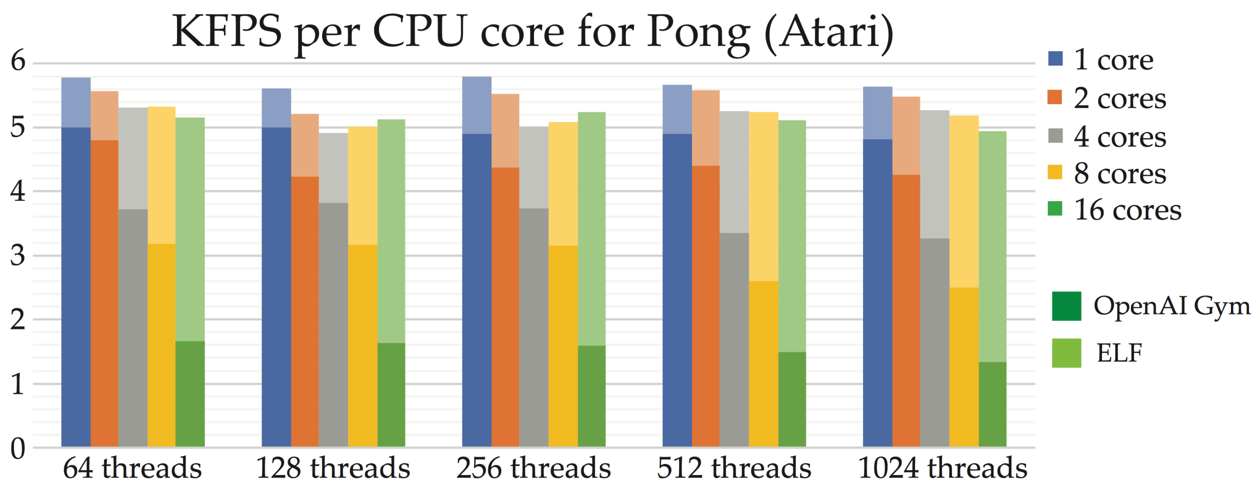
A miniature RTS engine



Game Name	Descriptions	Avg Game Length
Mini-RTS	Gather resource and build troops to destroy opponent's base.	1000-6000 ticks
Capture the Flag	Capture the flag and bring it to your own base	1000-4000 ticks
Tower Defense	Builds defensive towers to block enemy invasion.	1000-2000 ticks

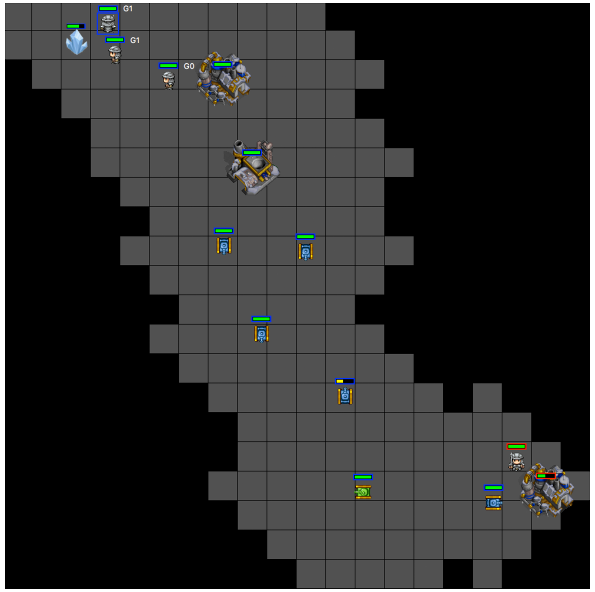
Simulation Speed



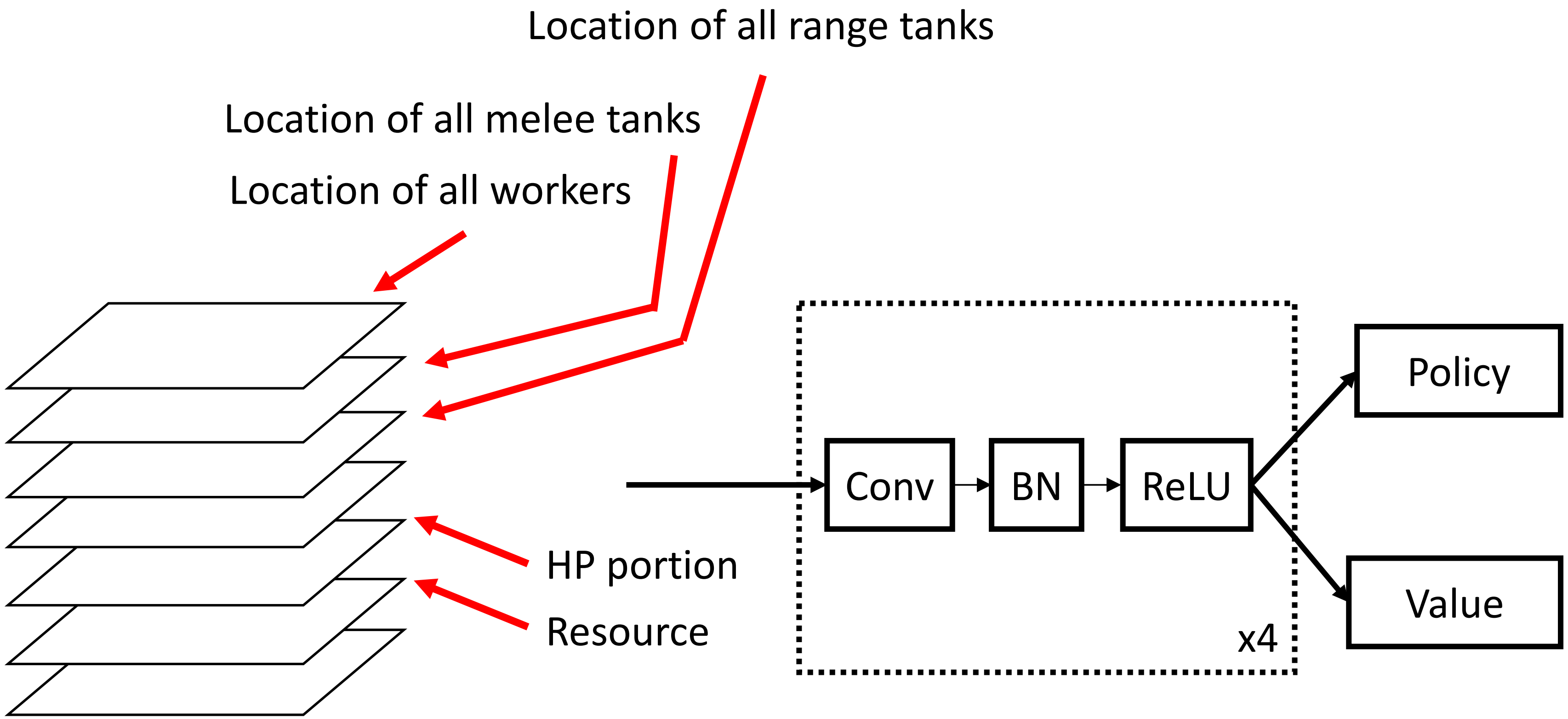


Platform	ALE	RLE	Universe	Malmo
FPS	6000	530	60	120
Platform	**DeepMind Lab**	**VizDoom**	**TorchCraft**	***Mini-RTS***
FPS	287(C) / 866(G) 6CPU + 1GPU	7,000	2,000 (Frameskip=50)	*40,000*

Training AI



Game visualization



Game internal data
(respecting fog of war)

Using Internal Game data and A3C.
Reward is only available once the game is over.

MiniRTS

Building that can build workers and collect resources.

Resource unit that contains 1000 minerals.

Building that can build melee attacker and range attacker.

Worker who can build barracks and gather resource.
Low speed in movement and low attack damage.

Tank with high HP, medium movement speed, short attack range, high attack damage.

Tank with low HP, high movement speed, long attack range and medium attack damage.

Training AI

9 discrete actions.

No.	Action name	Descriptions
1	IDLE	Do nothing
2	BUILD WORKER	If the base is idle, build a worker
3	BUILD BARRACK	Move a worker (gathering or idle) to an empty place and build a barrack.
4	BUILD MELEE ATTACKER	If we have an idle barrack, build an melee attacker.
5	BUILD RANGE ATTACKER	If we have an idle barrack, build a range attacker.
6	HIT AND RUN	If we have range attackers, move towards opponent base and attack. Take advantage of their long attack range and high movement speed to hit and run if enemy counter-attack.
7	ATTACK	All melee and range attackers attack the opponent's base.
8	ATTACK IN RANGE	All melee and range attackers attack enemies in sight.
9	ALL DEFEND	All troops attack enemy troops near the base and resource.

Win rate against rule-based AI

Frame skip (how often AI makes decisions)

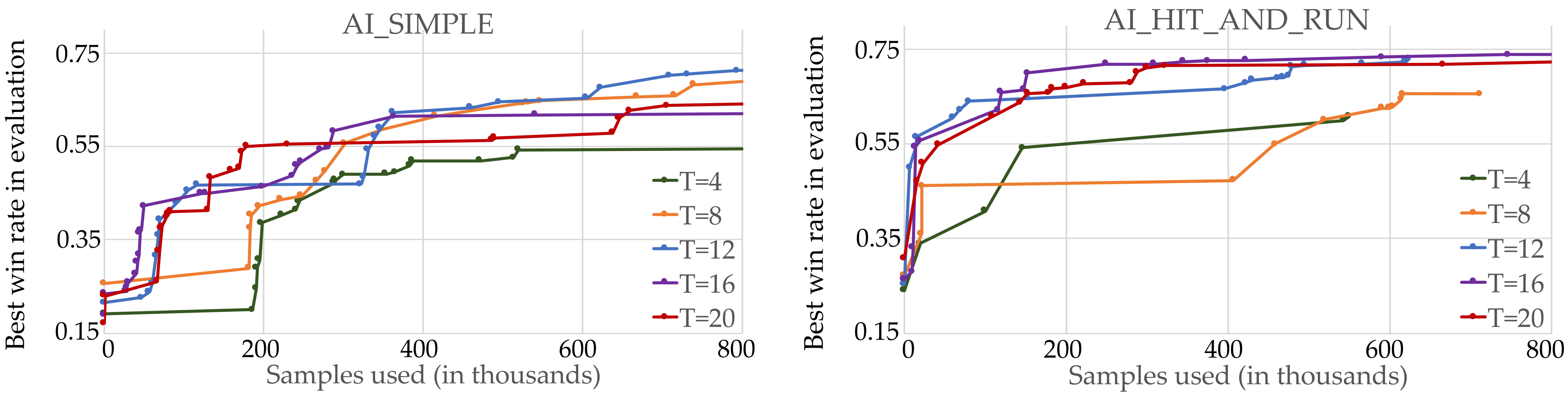
Frame skip	AI_SIMPLE	AI_HIT_AND_RUN
50	68.4(±4.3)	63.6(±7.9)
20	61.4(±5.8)	55.4(±4.7)
10	52.8(±2.4)	51.1(±5.0)

Network Architecture

Conv → BN → ReLU

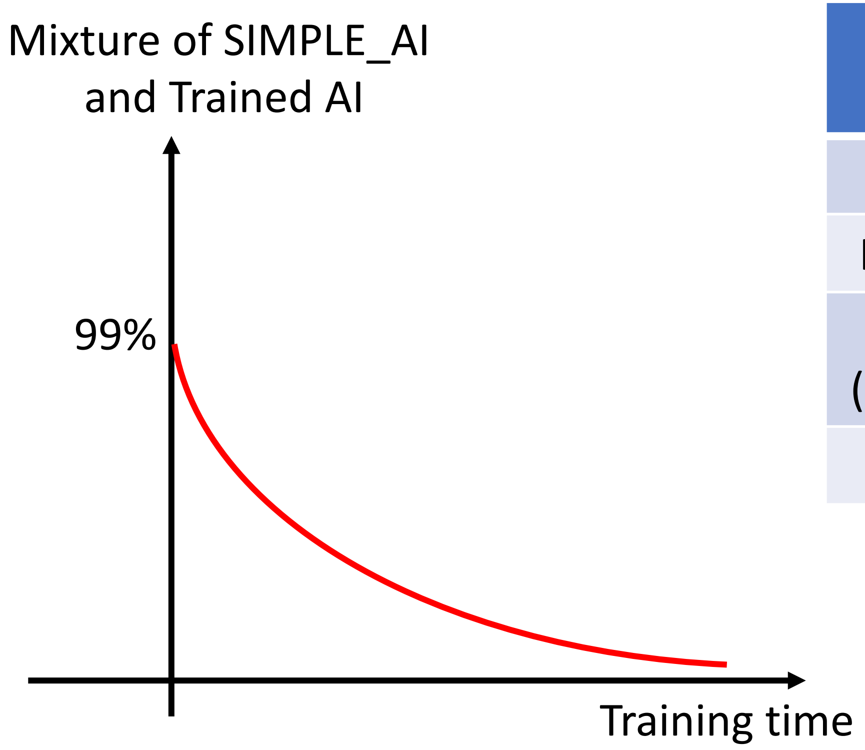
	SIMPLE (median)	SIMPLE (mean/std)	HIT_AND_RUN (median)	HIT_AND_RUN (mean/std)
ReLU	52.8	54.7(±4.2)	60.4	57.0(±6.8)
Leaky ReLU	59.8	61.0(±2.6)	60.2	60.3(±3.3)
ReLU + BN	61.0	64.4(±7.4)	55.6	57.5(±6.8)
Leaky ReLU + BN	72.2	68.4(±4.3)	65.5	63.6(±7.9)

Effect of T-steps



Large T is better.

Transfer Learning and Curriculum Training



	AI_SIMPLE	AI_HIT_AND_RUN	Combined (50%SIMPLE+50% H&R)
SIMPLE	**68.4 (±4.3)**	26.6(±7.6)	47.5(±5.1)
HIT_AND_RUN	34.6(±13.1)	**63.6 (±7.9)**	49.1(±10.5)
Combined (No curriculum)	49.4(±10.0)	46.0(±15.3)	47.7(±11.0)
Combined	51.8(±10.6)	54.7(±11.2)	**53.2(±8.5)**

Highest win rate against AI_SIMPLE: 80%

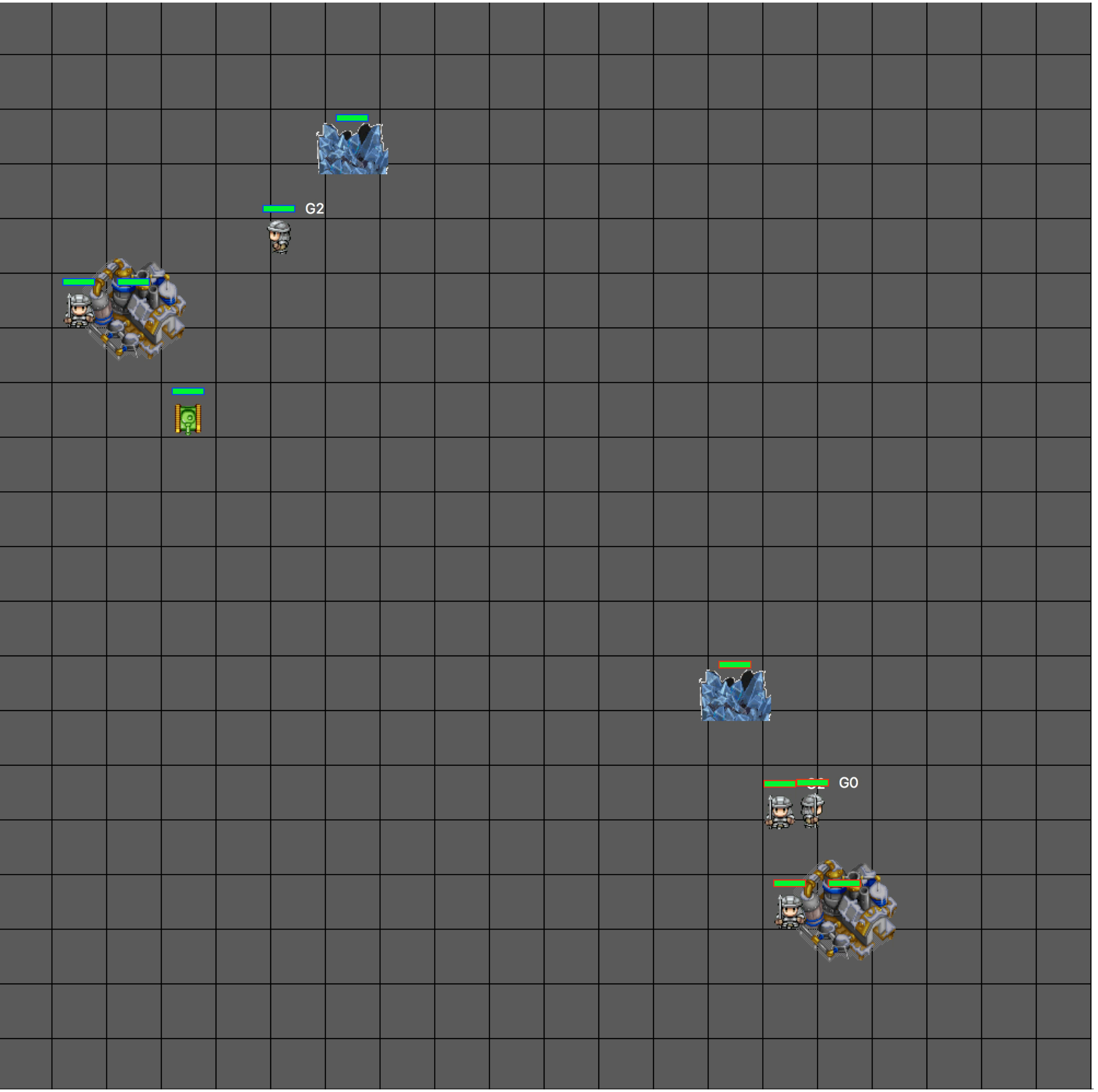
	AI_SIMPLE	AI_HIT_AND_RUN	CAPTURE_THE_FLAG
Without curriculum training	66.0 (±2.4)	54.4 (±15.9)	54.2 (±20.0)
With curriculum training	**68.4 (±4.3)**	**63.6 (±7.9)**	**59.9 (±7.4)**

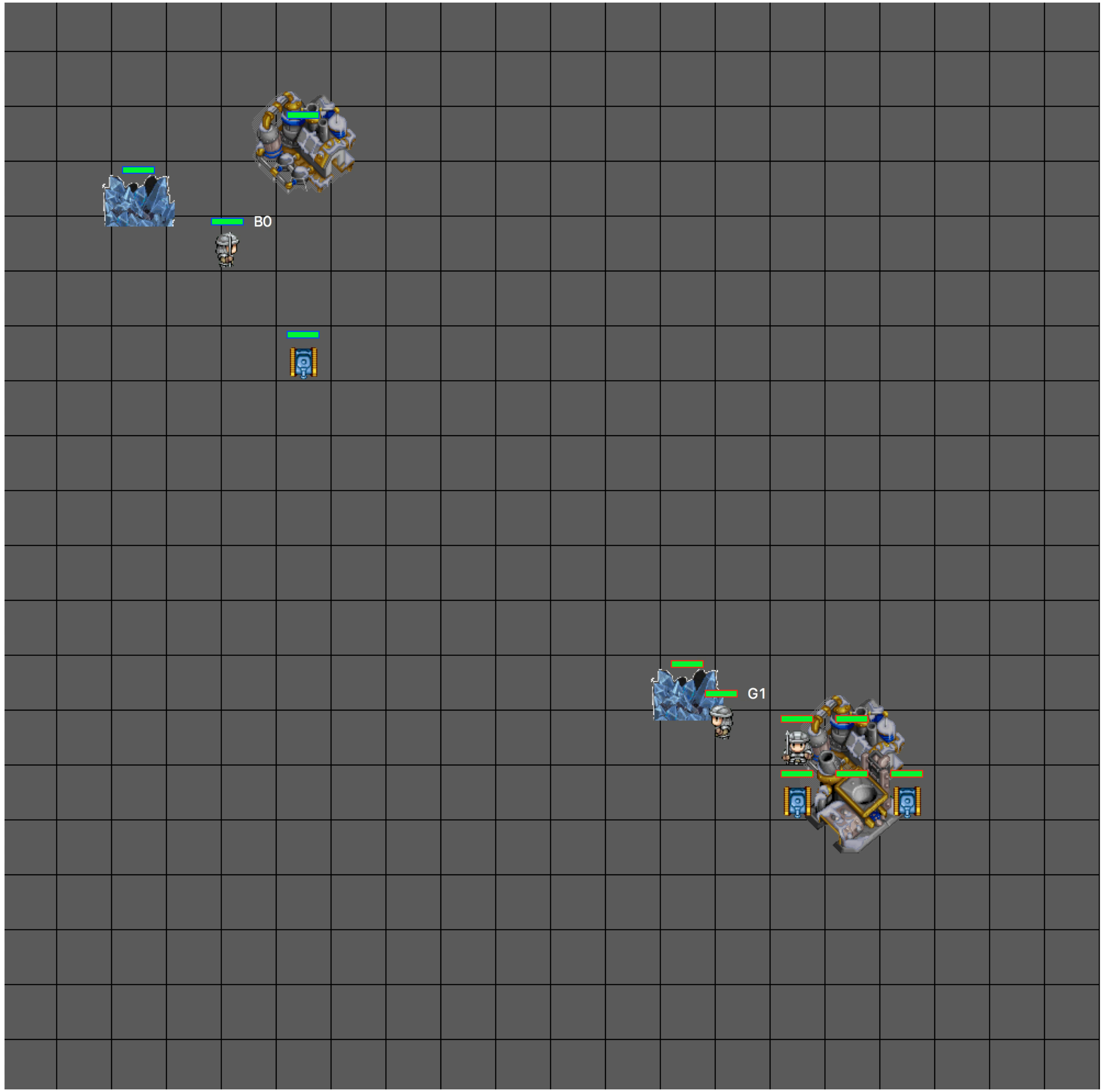
Monte Carlo Tree Search

	MiniRTS (AI_SIMPLE)	MiniRTS (Hit_and_Run)
Random	24.2 (±3.9)	25.9 (±0.6)
MCTS	73.2 (±0.6)	62.7 (±2.0)

MCTS evaluation is repeated on 1000 games, using 800 rollouts.

MCTS uses complete information and perfect dynamics


G2
G0


B0
G1

Future Work

- Richer game scenarios.
  - Multiple bases (Expand? Rush? Defending?)
  - More complicated units.
- More Realistic action space
  - Assign one action per unit
- Model-based Reinforcement Learning
  - MCTS with perfect information and perfect dynamics also achieves ~70% winrate
- Self-Play (Trained AI versus Trained AI)

Thanks!