

PLANNING-BASED RL

ALPHAZERO

ALPHAGO



ALPHAGO

Critics Consensus

No consensus yet.



100%

TOMATOMETER

Total Count: 9



100%

AUDIENCE SCORE

User Ratings: 115

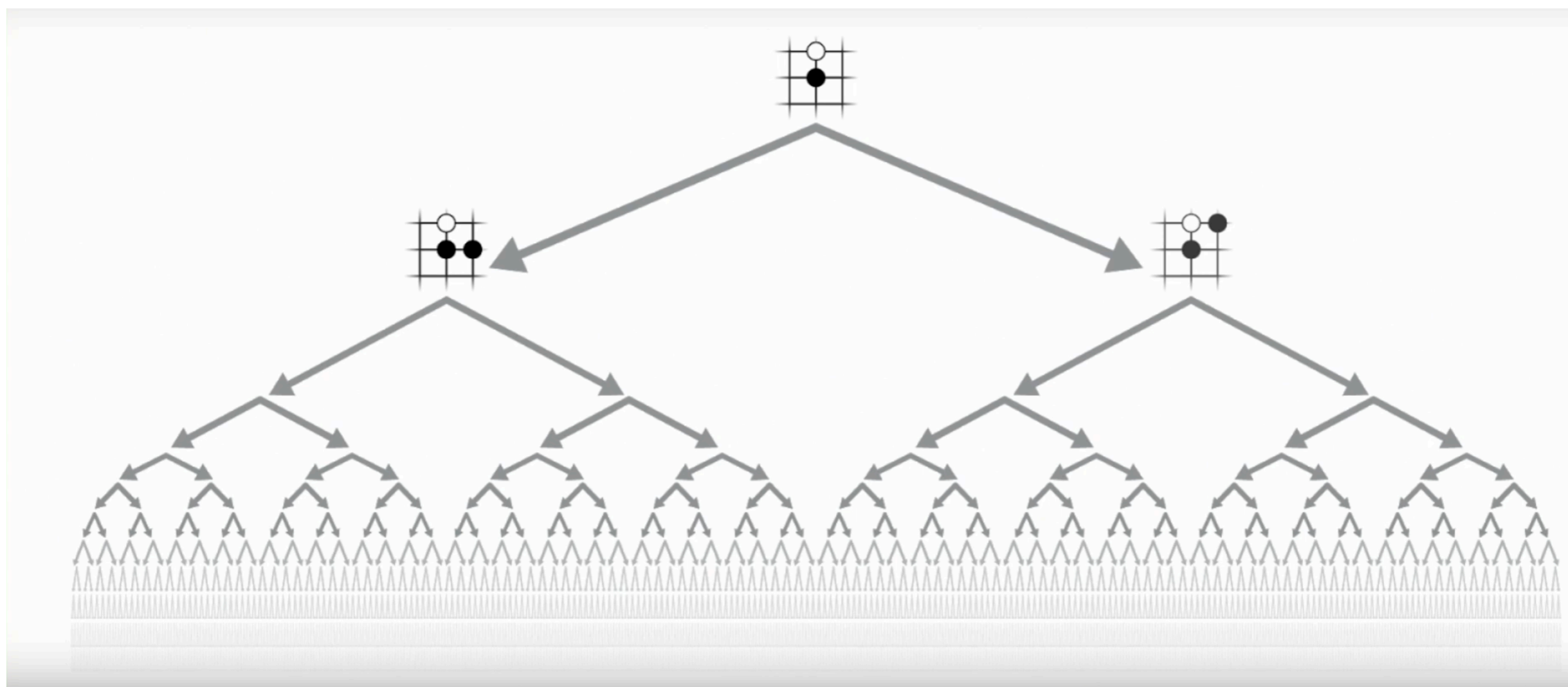
GO



- ▶ $\sim 10^{170}$ board positions
 - ▶ $\sim 10^{82}$ atoms in (observable) universe
- ▶ Might **never** be possible to brute-force solve Go
- ▶ The only 'reward' signal: win/loss at end of game!

GO

Naive Solution: Exhaustive Search



TIMELINE: ALPHA-FAMILY

- ▶ **2016:** AlphaGo
 - ▶ Only plays Go. Beats world champion
- ▶ **2017:** AlphaGo Zero
 - ▶ Removed need to train on human games first
- ▶ **2018:** AlphaZero
 - ▶ Generalized to work on Go, Chess, Shogi, etc.
 - ▶ Beat world computer champion, Stockfish
 - ▶ Stockfish has vast amounts of domain-specific engineering (14,000 LOC)
- ▶ **2019:** MuZero
 - ▶ Learns rules of games
 - ▶ Works for single-agent reinforcement learning (eg Atari)

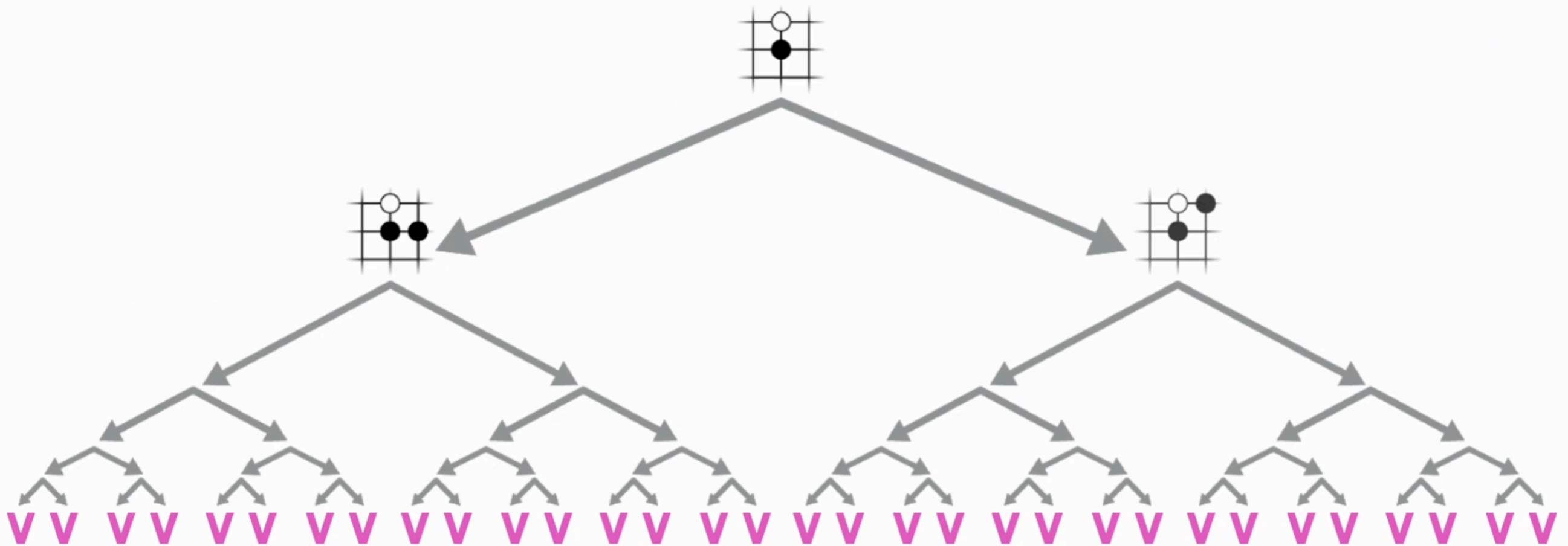
IDEA 1: REDUCE SEARCH SPACE WITH NEURAL NET

$$(\mathbf{p}, v) = f_{\theta}(s)$$

- ▶ s : state of the game (eg board position)
- ▶ \mathbf{p} : a probability distribution over possible actions
- ▶ v : a scalar value. The expected outcome of game in state s
- ▶ θ : learnable neural net parameters

REDUCE SEARCH DEPTH

Reduce **search-depth** with value v



MAKING A MOVE: MCTS

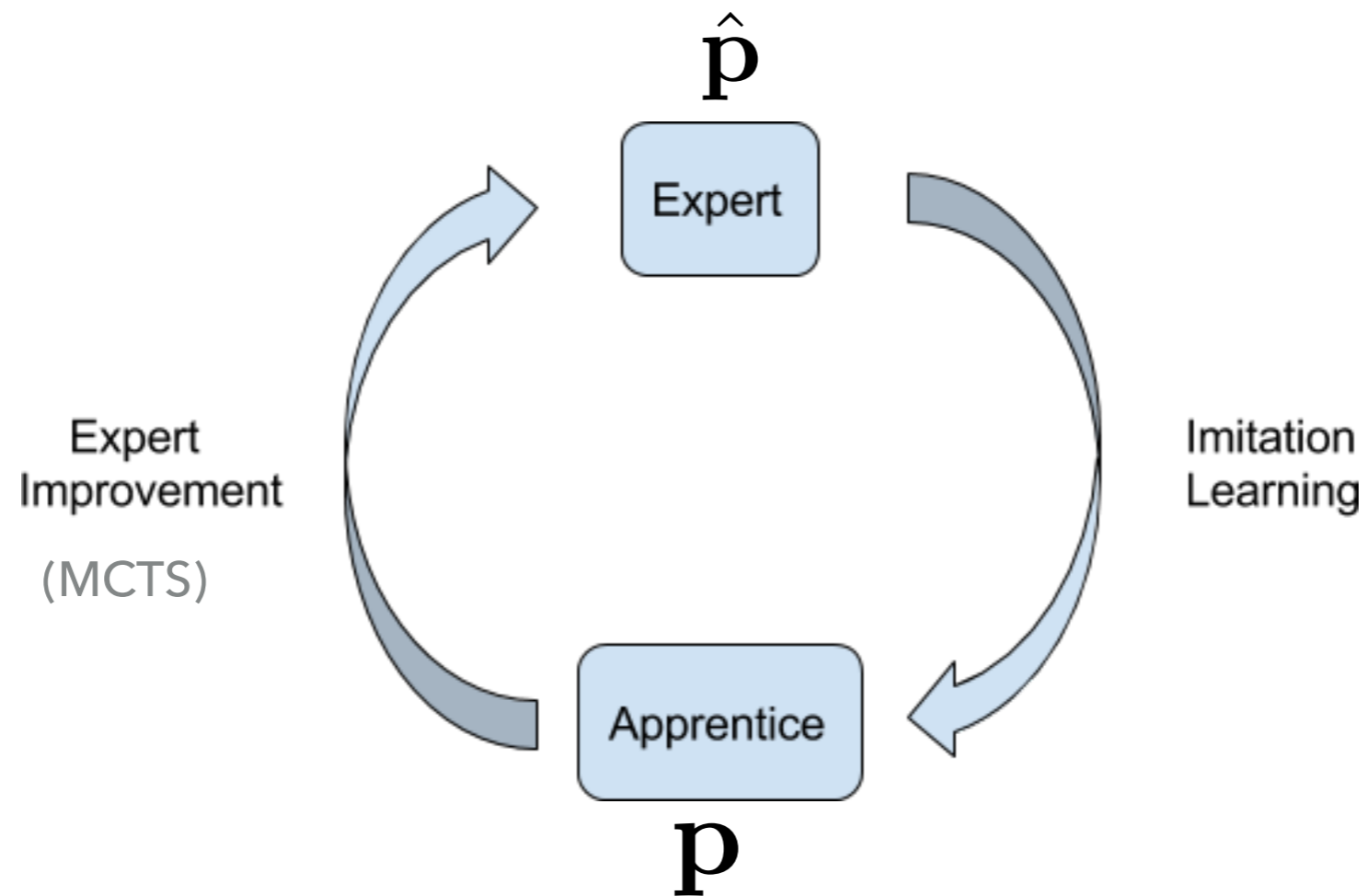
- ▶ Before making a real move:
 - ▶ **Search**: try the most promising moves *in its mind*, and see which leads to the highest value!
 - ▶ A position that looks good at first glance might lead to checkmate in 5 moves
 - ▶ Obviously needs to know the **rules/environment model**
 - ▶ Also add some noise to search: explore vs exploit
 - ▶ Known as **Monte-Carlo Tree Search** (MCTS)
 - ▶ Use the results of this search to create **new, better** policy $\hat{\mathbf{p}}$
 - ▶ MCTS can be seen as a *policy improvement operator*

IDEA 2: USE MCTS POLICY FOR TRAINING OBJECTIVE

- ▶ **Reward Sparsity:** win/loss/draw at end of game
 - ▶ Can take hundreds of moves to get there!
- ▶ **Solution:** train the neural net so that \mathbf{p} is the same as $\hat{\mathbf{p}}$
 - ▶ Minimize cross-entropy between two distributions

$$\mathcal{L}_{\text{alphazero}} = (z - v)^2 - \hat{\mathbf{p}}^T \log(\mathbf{p})$$

SAME IDEA: EXPERT ITERATION



Taken from: Thinking Fast and Slow with Deep Learning and Tree Search, T. Anthony et al, 2017

IDEA 3: SELF-PLAY

- ▶ **Difference with normal RL**
 - ▶ Don't have an agent to play against!
 - ▶ How to evaluate strength?

“Modern learning algorithms are outstanding test-takers: once a problem is packaged into a suitable objective, deep (reinforcement) learning algorithms often find a good solution. However, in many multi-agent domains, the question of what test to take, or what objective to optimize, is not clear...Learning in games is often conservatively formulated as training agents that tie or beat, on average, a fixed set of opponents. However, the dual task, that of *generating useful opponents to train and evaluate against, is under-studied*. It is not enough to beat the agents you know; it is also important to generate better opponents, which exhibit behaviours that you don't know.” ~ Balduzzi et al. (2019)

IDEA 3: SELF-PLAY

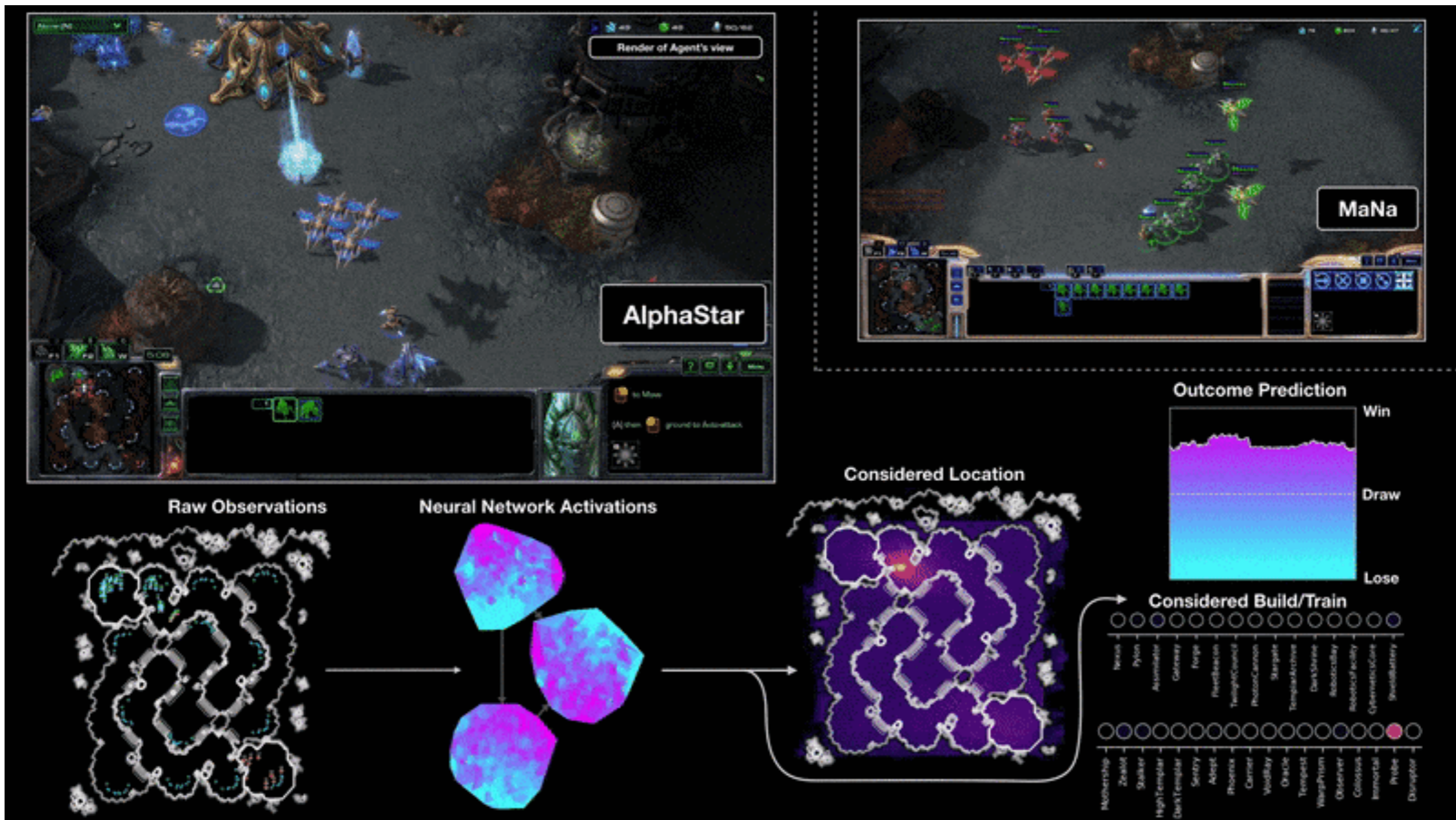
- ▶ Two types of game:
 - ▶ **Transitive**: Chess, Go, etc
 - ▶ **Intransitive** (or **Cyclic**): Rock-Paper-Scissors, StarCraft
- ▶ Transitive games have a special property: if agent **A** beats agent **B** and **B** beats **C**, then **A** beats **C**
 - ▶ Basis of ELO rating system in Chess

IDEA 3: SELF-PLAY

▶ Self-Play Algorithm

- ▶ Start with neural net agent f_1
- ▶ Play f_1 against itself to generate training data
- ▶ Train f_1 on this data using $\mathcal{L}_{\text{alphazero}} = (z - v)^2 - \hat{\mathbf{p}}^T \log(\mathbf{p})$ to produce f_2
- ▶ **Repeat**
- ▶ **Key:** f_n will beat f_1, f_2, \dots due to **transitivity!**

ALPHASTAR FOR INTRANSITIVE GAMES: NEEDS HUMAN GAMES!

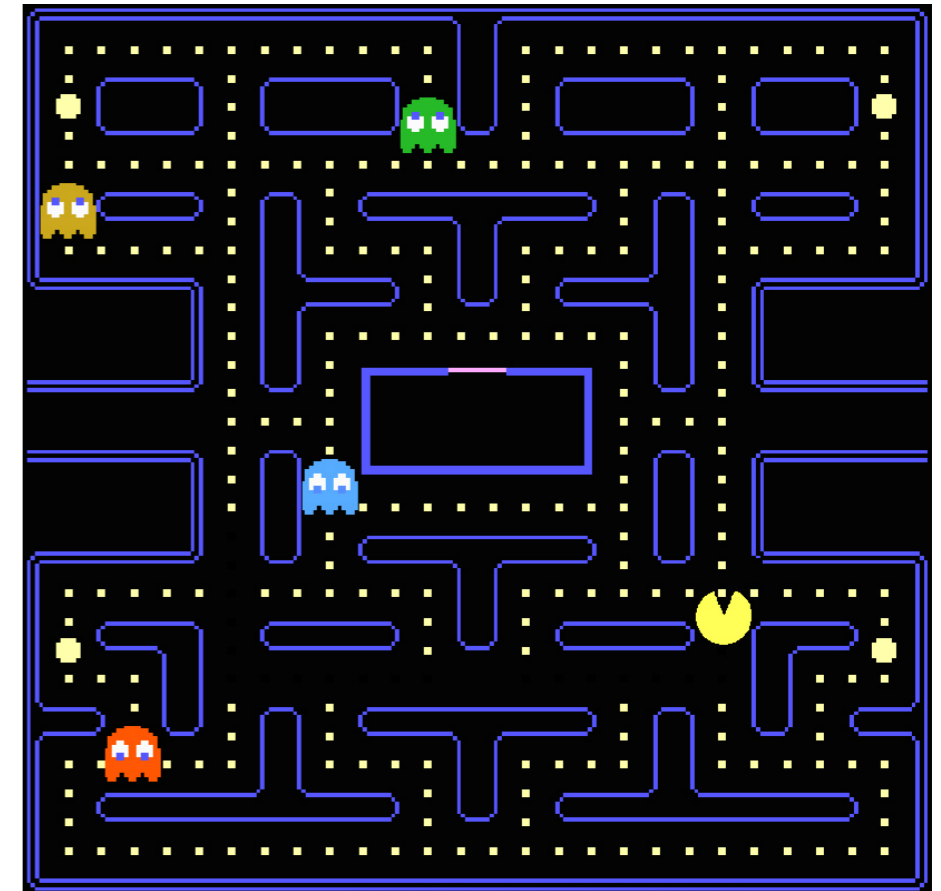
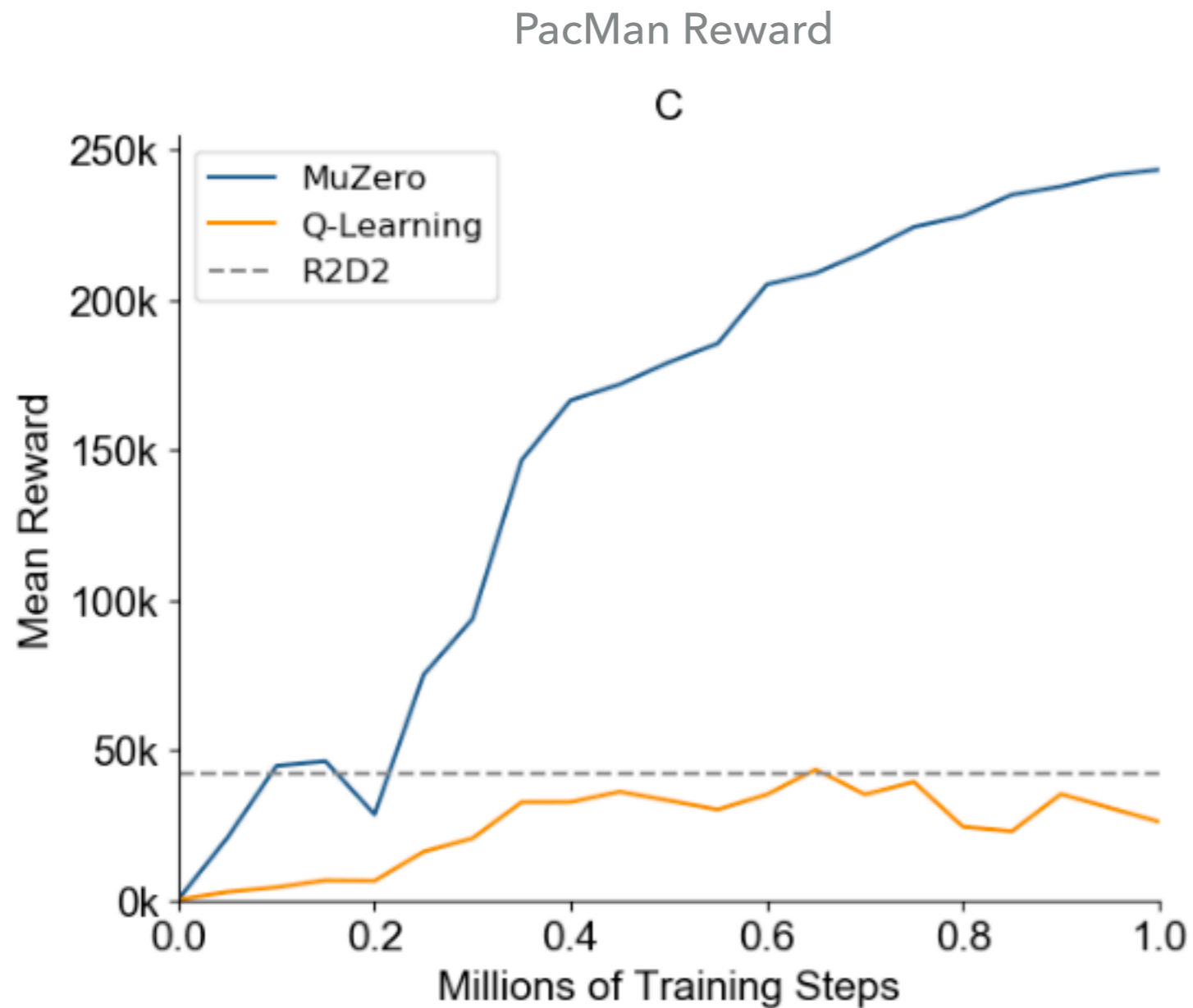


- <https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/>

IDEA 4: A LEARNED MODEL + MCTS = POWER

- ▶ For things like Atari, or almost any real-world RL problem, we don't know the rules/model:
 - ▶ if I have a state s_1 and take action a , what will state s_2 be?
- ▶ **MuZero**: use RNN to predict next state given previous states + previous actions!
 - ▶ Then use for planning with MCTS like in AlphaZero

IDEA 4: A LEARNED MODEL + MCTS = POWER

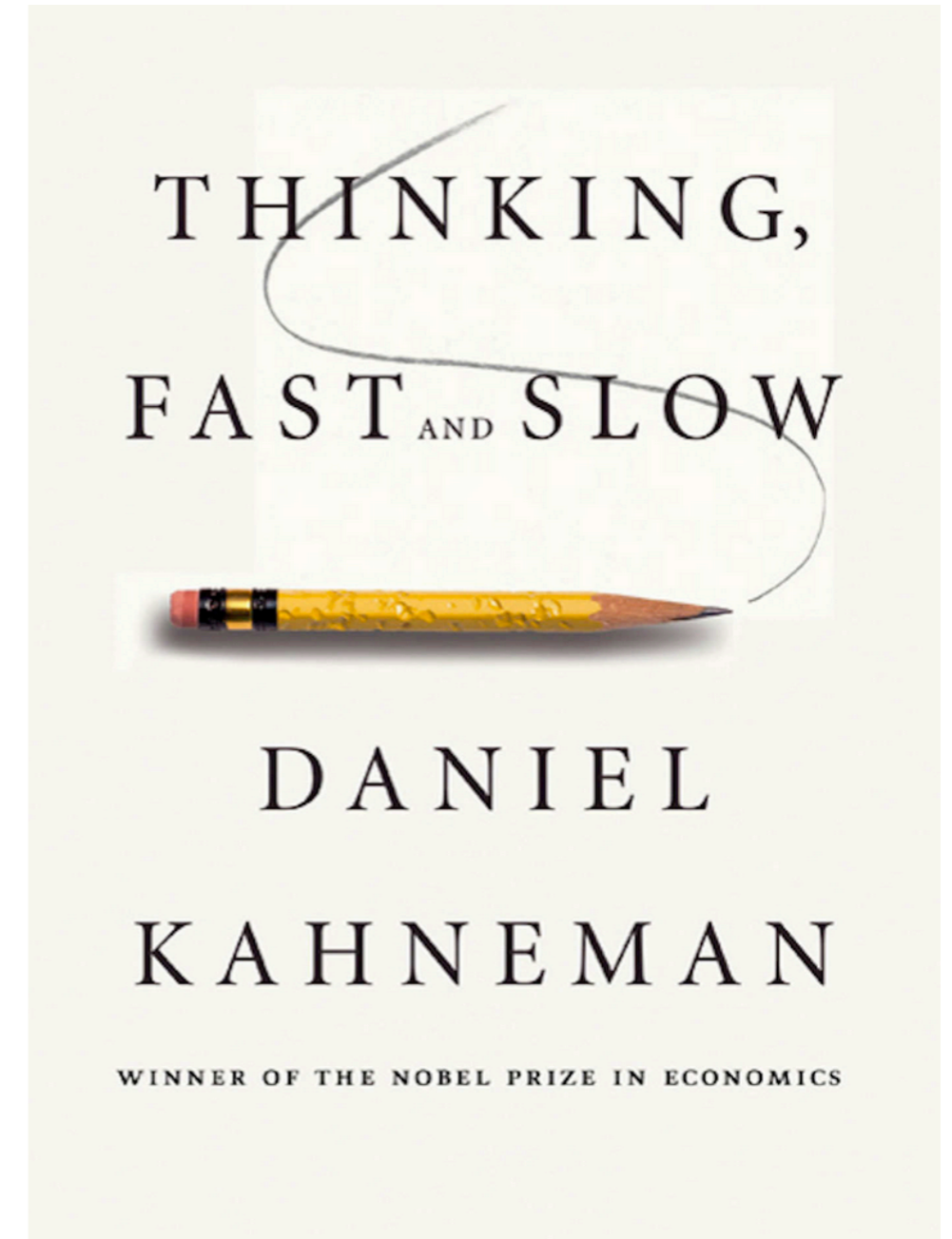


ALPHA ZERO

- ▶ **Idea 1:** Reduce search-space with policy-value neural net
 - ▶ Needs rules/model of environment. Can be learnt (MuZero)
- ▶ **Idea 2:** Use the MCTS policy as the training objective
- ▶ **Idea 3:** Use self-play to generate training data
- ▶ **Idea 4:** MCTS with a learned model of the world worlds well

COMPARISON TO HUMANS: DUAL PROCESS THEORY

“**System 1** operates automatically and quickly, with little or no effort and no sense of voluntary control. **System 2** allocates attention to the effortful mental activities that demand it, including complex computations. The operations of **System 2** are often associated with the subjective experience of agency, choice, and concentration.”



COMPARISON TO HUMANS: DUAL PROCESS THEORY

Dual process theory of thought



 Behaviour Design

System 1

Fast / Automatic

Emotional

- Impulses / Drives
- Habits
- Beliefs

Neural Net



@BehaviourDesign

System 2

Slow / Effortful

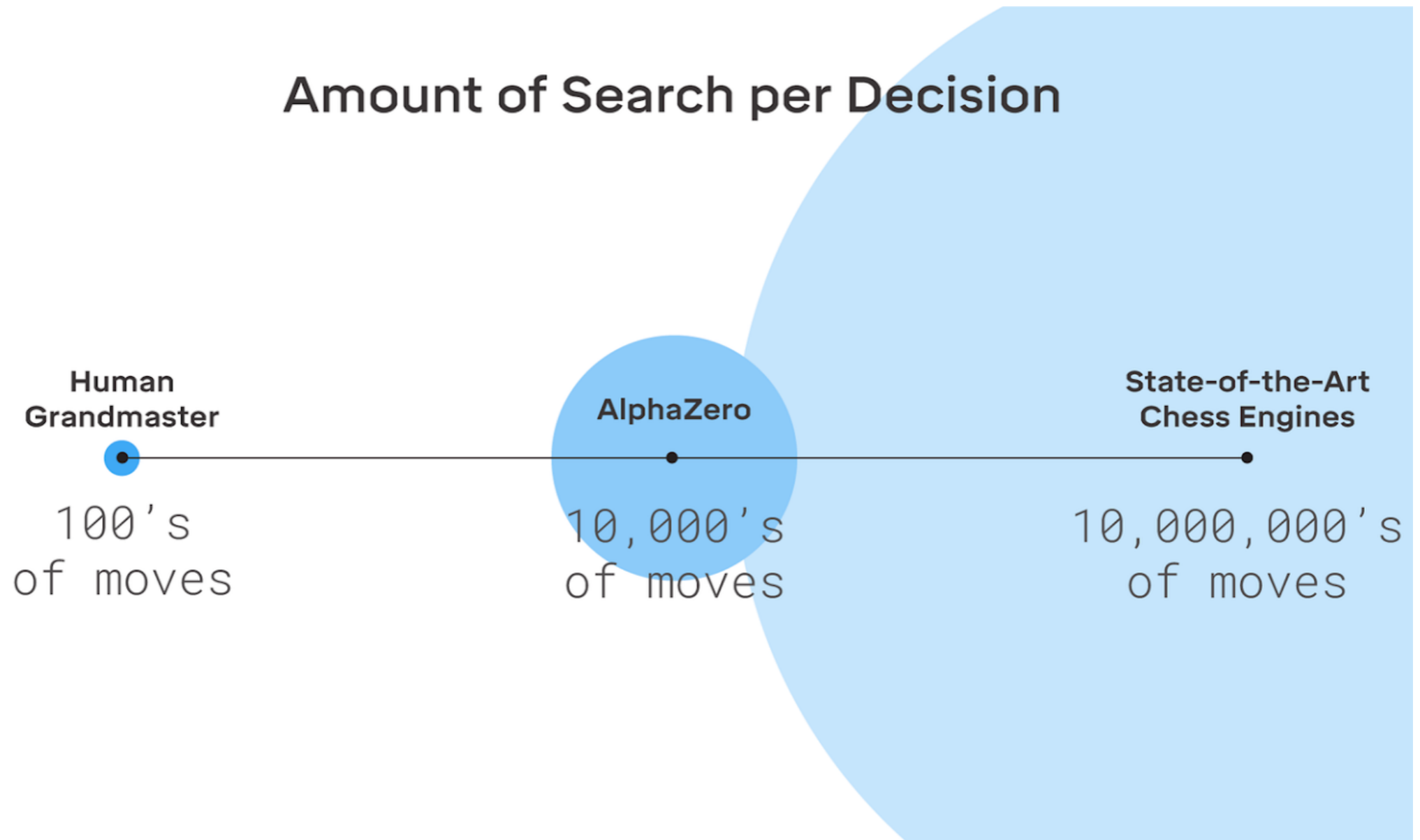
Logical

- Reflection
- Planning
- Problem solving

Neural Net used in MCTS

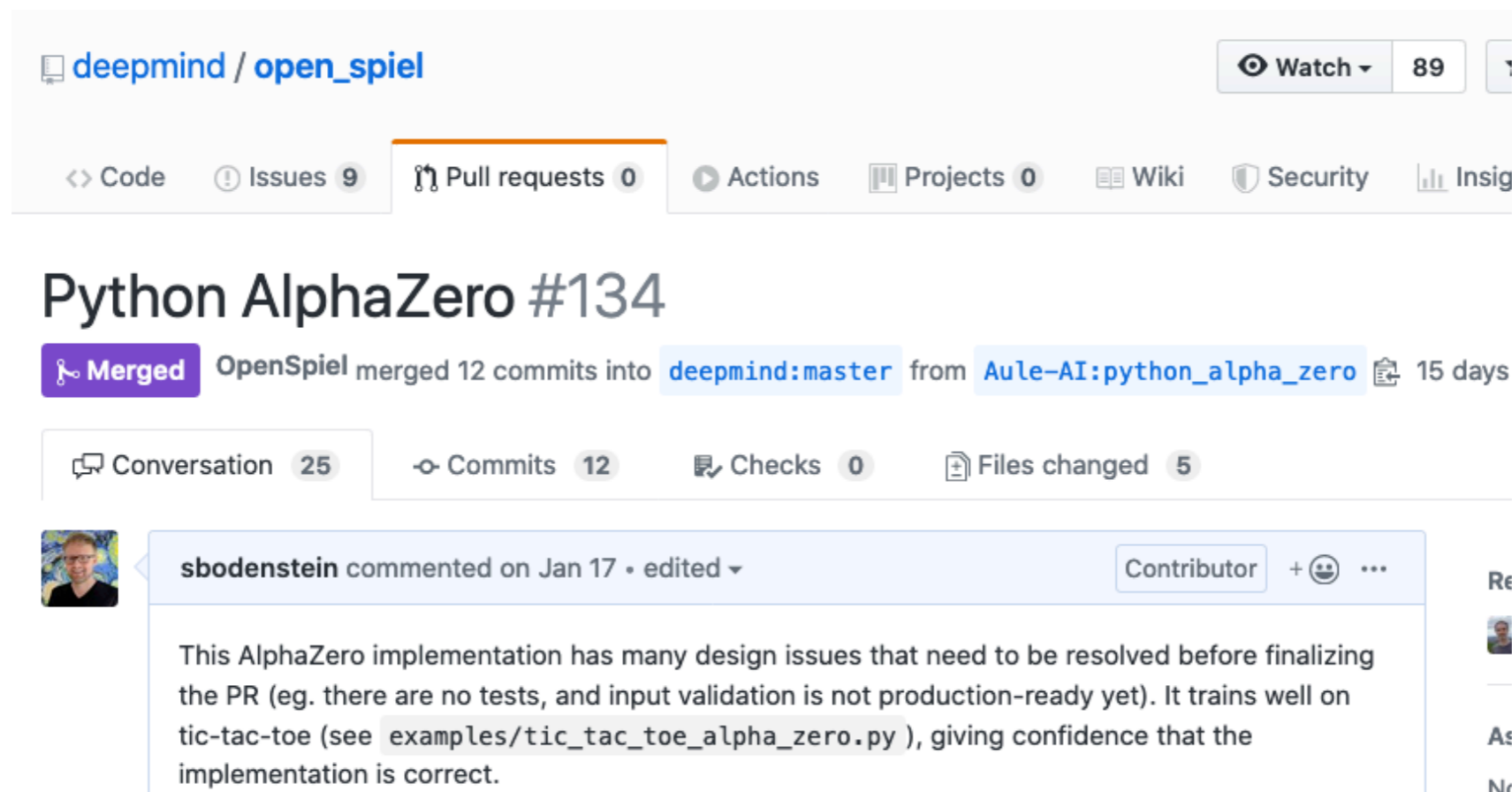
Taken from David Barber *Learning From Scratch by Thinking Fast and Slow with Deep Learning and Tree Search*

COMPARISON TO HUMANS



LESSONS FROM IMPLEMENTING

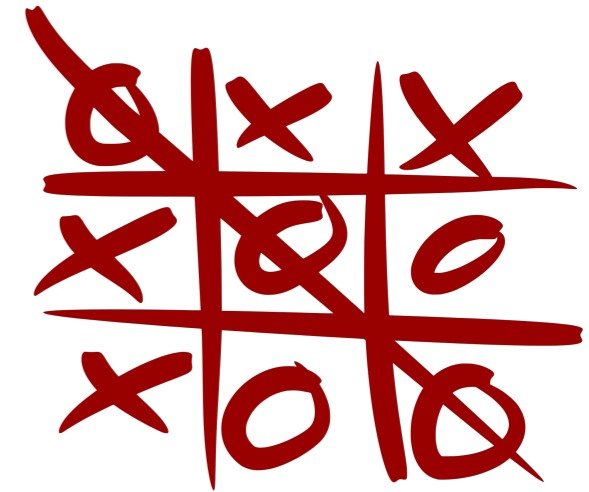
DeepMind OpenSpiel (https://github.com/deepmind/open_spiel): “OpenSpiel is a collection of environments and algorithms for research in general reinforcement learning and search/planning in games.”



The screenshot shows a GitHub pull request interface for the repository `deepmind / open_spiel`. The repository has 89 watchers. The pull request is titled "Python AlphaZero #134" and is marked as "Merged". It shows that "OpenSpiel" merged 12 commits into the `deepmind:master` branch from the `Aule-AI:python_alpha_zero` branch, 15 days ago. The pull request has 25 conversations, 12 commits, 0 checks, and 5 files changed. A comment from user `sbodenstein`, dated Jan 17 and edited, states: "This AlphaZero implementation has many design issues that need to be resolved before finalizing the PR (eg. there are no tests, and input validation is not production-ready yet). It trains well on tic-tac-toe (see `examples/tic_tac_toe_alpha_zero.py`), giving confidence that the implementation is correct."

LESSONS FROM IMPLEMENTING

- ▶ A simple tic-tac-toe example
 - ▶ Takes around ~2 mins to train on laptop



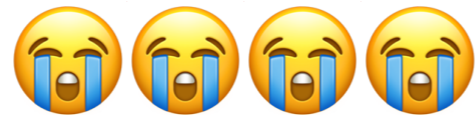
```
python open_spiel/python/examples/tic_tac_toe_alpha_zero.py
```

```
I0225 12:29:52.672022 4576402880 tic_tac_toe_alpha_zero.py:155] ----- Starting round 0 out of 25 -----
I0225 12:29:52.672169 4576402880 tic_tac_toe_alpha_zero.py:160] Playing 50 games against the minimax player.
I0225 12:30:10.688238 4576402880 tic_tac_toe_alpha_zero.py:164] Result against Minimax player: 49 losses and 1 draws.
I0225 12:30:10.688338 4576402880 tic_tac_toe_alpha_zero.py:166] Running 100 games of self play
I0225 12:30:24.127839 4576402880 tic_tac_toe_alpha_zero.py:169] Training the net for 10 epochs.
Epoch 0 mean losses. Total: 3.17, Policy: 2.26, Value: 0.892, L2: 0.0134
Epoch 1 mean losses. Total: 2.89, Policy: 2.12, Value: 0.757, L2: 0.0132
Epoch 2 mean losses. Total: 2.78, Policy: 2.07, Value: 0.704, L2: 0.0132
Epoch 3 mean losses. Total: 2.69, Policy: 2.03, Value: 0.638, L2: 0.0133
Epoch 4 mean losses. Total: 2.65, Policy: 2.02, Value: 0.623, L2: 0.0135
Epoch 5 mean losses. Total: 2.58, Policy: 1.98, Value: 0.587, L2: 0.0137
Epoch 6 mean losses. Total: 2.52, Policy: 1.93, Value: 0.574, L2: 0.014
Epoch 7 mean losses. Total: 2.52, Policy: 1.93, Value: 0.578, L2: 0.0144
Epoch 8 mean losses. Total: 2.41, Policy: 1.87, Value: 0.527, L2: 0.0147
Epoch 9 mean losses. Total: 2.43, Policy: 1.84, Value: 0.571, L2: 0.0152
I0225 12:30:24.672239 4576402880 tic_tac_toe_alpha_zero.py:155] ----- Starting round 1 out of 25 -----
I0225 12:30:24.672328 4576402880 tic_tac_toe_alpha_zero.py:166] Running 100 games of self play
I0225 12:30:38.603941 4576402880 tic_tac_toe_alpha_zero.py:169] Training the net for 10 epochs.
Epoch 0 mean losses. Total: 2.49, Policy: 1.87, Value: 0.607, L2: 0.0157
```

```
....|
```

LESSONS: HYPERPARAMETERS

"The hyperparameters of AlphaGo Zero were tuned by Bayesian optimization." ~ *AlphaZero paper*



- ▶ Luckily, only two game specific hyper parameters
 - ▶ Found reasonable values for tic-tac-toe with some trial and error

LESSONS: OPENSOURCE

- ▶ Code reviews by real experts
 - ▶ You cannot buy the sort of feedback you can get from DeepMind engineers/researchers!
 - ▶ Its easy to fool yourself that you understand something better than you do
- ▶ Fortuitous Connections
- ▶ Looks good for employers
 - ▶ “Research and software engineer experience demonstrated via an internship, **contributions to open source**, work experience, or coding competitions.” ~ from recent Cape Town AI company job posting
- ▶ **Benefits others!**

OPENSPIEL CALL FOR CONTRIBUTIONS

See: https://github.com/deepmind/open_spiel/blob/master/docs/contributing.md

should be possible in a future update.

- **Grid Worlds.** There are currently four grid world games in OpenSpiel: Markov soccer, the coin game, cooperative box-pushing, and laser tag. There could be more, especially ones that have been commonly used in multiagent RL. Also, the current grid worlds can be improved (they all are fully-observable).
- **Heuristic Payoff Tables and Empirical Game-Theoretic Analysis.** Methods found in [Analyzing Complex Strategic Interactions in Multi-Agent Systems](#), [Methods for Empirical Game-Theoretic Analysis](#), [An evolutionary game-theoretic analysis of poker strategies](#), [Ref4](#).
- **Monte Carlo Tree Search Solver.** General enhancement to Monte Carlo tree search, backpropagate proven wins and losses as far up as possible. See [Winands et al. '08](#).
- **Minimax-Q and other classic MARL algorithms.** Minimax-Q is a classic multiagent reinforcement learning algorithm ([Markov games as a framework for multi-agent reinforcement learning](#)). Other classic algorithms, such as [Correlated Q-learning](#), [NashQ](#), and Friend-or-Foe Q-learning ([Friend-or-foe q-learning in general-sum games](#)) would be welcome as well.
- **Nash Averaging.** An evaluation tool first described in [Re-evaluating Evaluation](#).
- **Negotiation Games.** A game similar to the negotiation game presented in [Ref1](#), [Ref2](#). Also, Colored Trails ([Modeling how Humans Reason about Others with Partial Information](#), [Metastrategies in the coloredtrails game](#)).
- **Opponent Modeling / Shaping Algorithms.** For example, [DRON](#), [LOLA](#), and [Stable Opponent Shaping](#).
- **PyTorch.** While we officially support Tensorflow, the API is agnostic to the library that is used for learning. We would like to have some examples and support for PyTorch as well in the future.
- **Repeated Games.** There is currently no explicit support for repeated games. Supporting repeated games as one sequential game could be useful for application of RL algorithms. This could take the form of another game

BACKPROP IN THE BRAIN

THANKS FOR LISTENING!