# RL LE3 VT2026

## Model-Based and Actor-Critic RL

**Fredrik Heintz**

**Dept. of Computer Science**
**Linköping University**

**fredrik.heintz@liu.se**
**@FredrikHeintz**

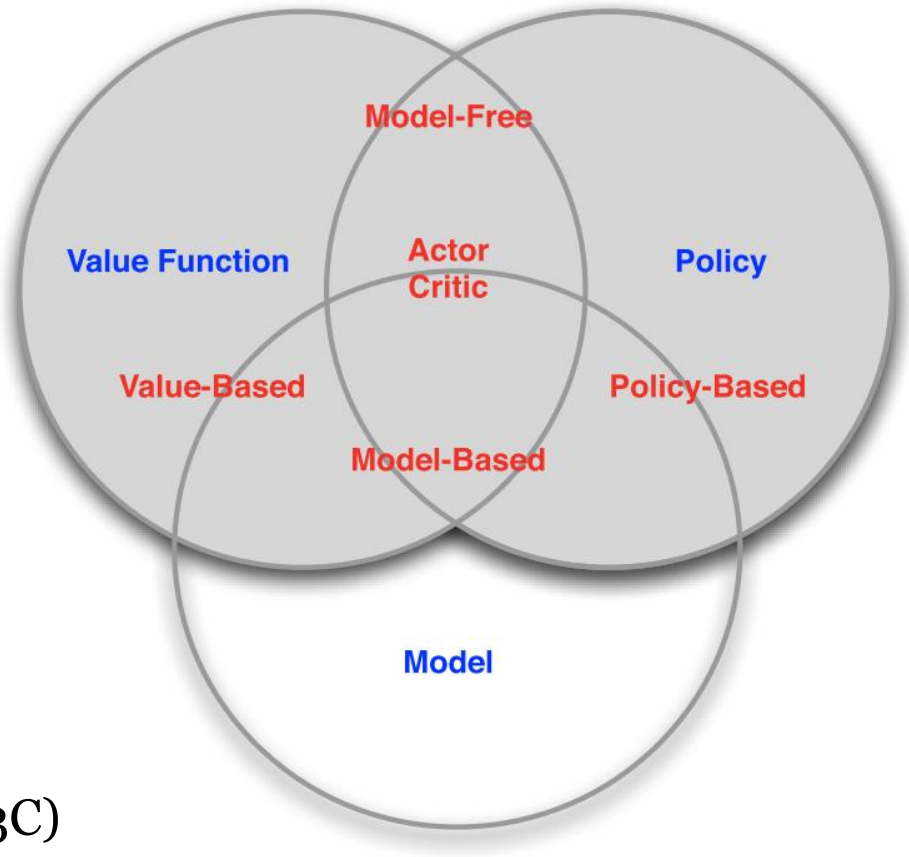
**Outline:**

- **Model-based vs Model-free RL**
- **Two-Player Games with Perfect Transitions**
- **Actor-Critic RL**

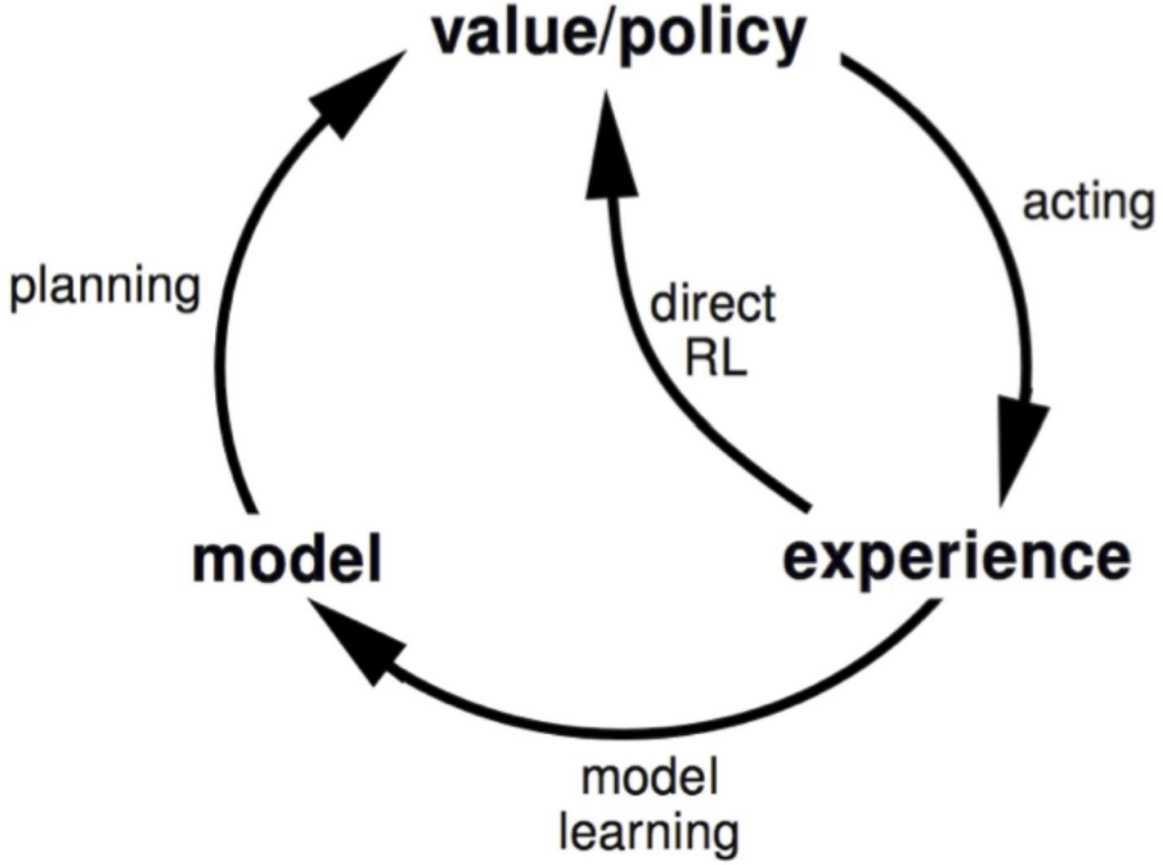
LINKÖPING UNIVERSITY

# Reinforcement Learning Approaches

- Value-Based:
  - Learn value function
  - Implicit policy (e.g. greedy selection)
  - Example: Deep Q Networks (DQN)
- Policy-Based:
  - No value function
  - Learn explicit (stochastic) policy
  - Example: Stochastic Policy Gradients
- Model-Based:
  - Learn transition model
  - Implicit policy
  - Example: Dreamer
- Actor-Critic:
  - Learn value function
  - Learn policy using value function
  - Example: Asynchronous Advantage Actor Critic (A3C)



# Model-Based vs Model-Free RL



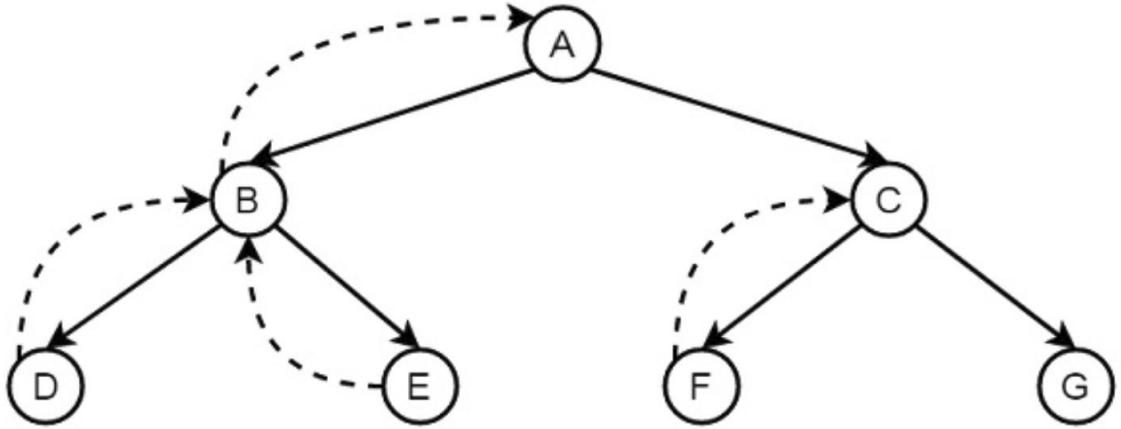

Learn *policy* direct or learn *transition* first and then policy?

# Learning Policies vs Learning Transitions

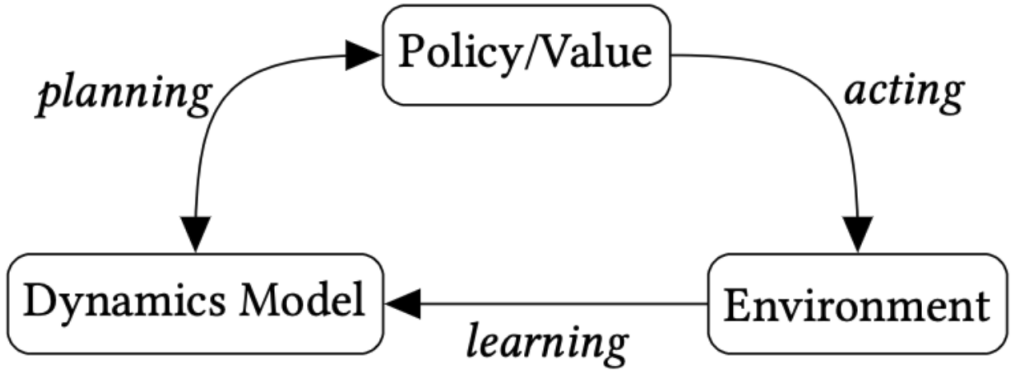
- $s \rightarrow a \rightarrow s' \rightarrow a' \rightarrow s'' \rightarrow a'' \rightarrow s''' \rightarrow a''' \rightarrow s''''$
- Learning a policy $s \rightarrow a$
  - Learning how to react in an environment
- Learning a transition $s \rightarrow a \rightarrow s'$
  - Learning how the environment reacts

# Learning vs Planning

- Learning
  - Agent changing state in the environment
  - Irreversible state change
  - Forward Path $s \rightarrow a \rightarrow s' \rightarrow a' \rightarrow s'' \rightarrow a'' \rightarrow s''' \rightarrow a''' \rightarrow s''''$
- Planning
  - Agent changing own local state
  - Reversible local state change
  - Backtracking Tree



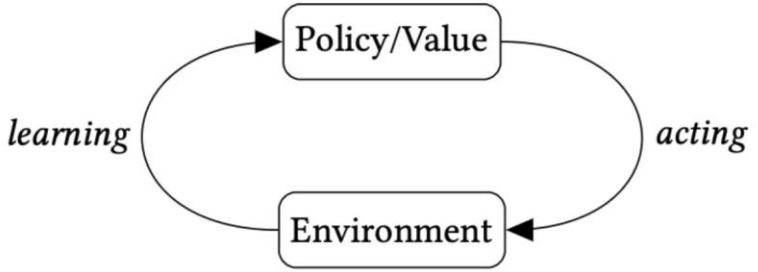
# Model-Based RL



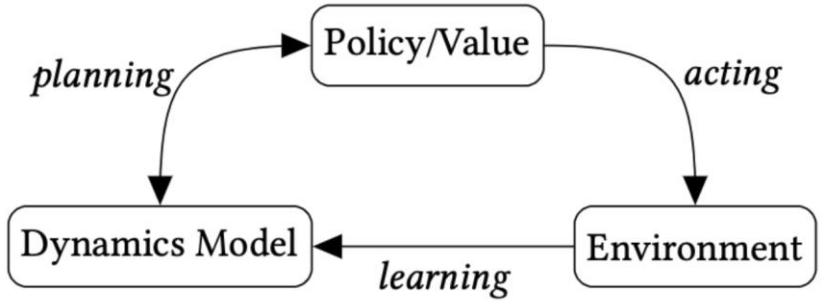

**repeat**
  Sample environment $E$ to generate data $D = (s, a, r', s')$
  Use $D$ to learn $M = T_a(s, s'), R_a(s, s')$ ▷ learning
  **for** $n = 1, \ldots, N$ **do**
    Use $M$ to update policy $\pi(s, a)$ ▷ planning
  **end for**
**until** $\pi$ converges

# Model-Based RL

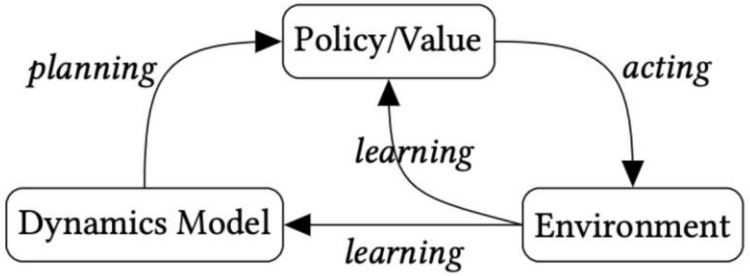
Learn policy directly




Learn model
and then plan actions




Use experience to
update both model and policy

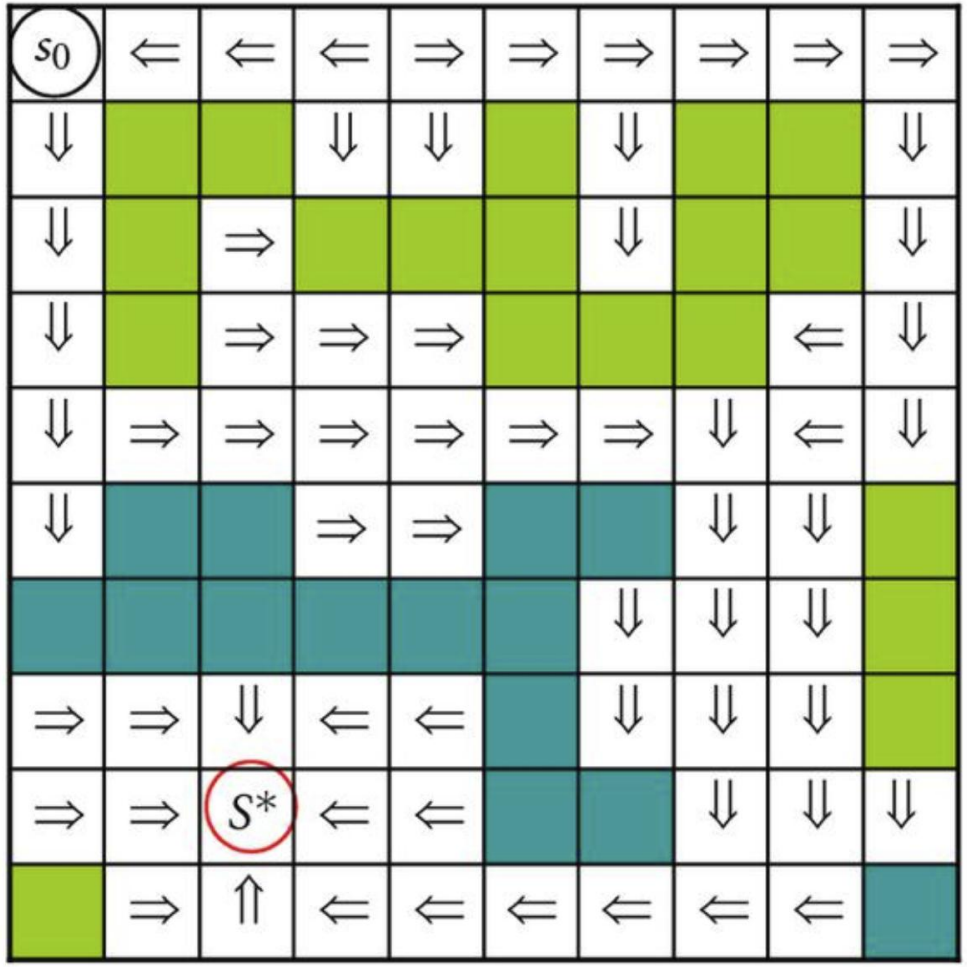


# Dyna [Sutton]

- Initialize Q-function
- Repeat
  - Initialize s; a←$\pi$(s); (s',r)←Env(s,a) :: **Learn**
  - Q(s,a) ← Q(s,a)+$\alpha$[r+$\gamma$max$_a$Q(s',$_a$)-Q(s,a)]
  - M(s,a) ← (s',r) :: **Model**
  - For n=1, …, N :
    - Select $\hat{s}$ and $\hat{a}$ randomly
    - (s',r) ← M($\hat{s}$,$\hat{a}$) :: **Plan for FREE!**
    - Q($\hat{s}$,$\hat{a}$) ← Q($\hat{s}$,$\hat{a}$)+$\alpha$[r+$\gamma$max$_a$Q(s',$_a$)-Q($\hat{s}$,$\hat{a}$)]
- Until Q converges
- return Q

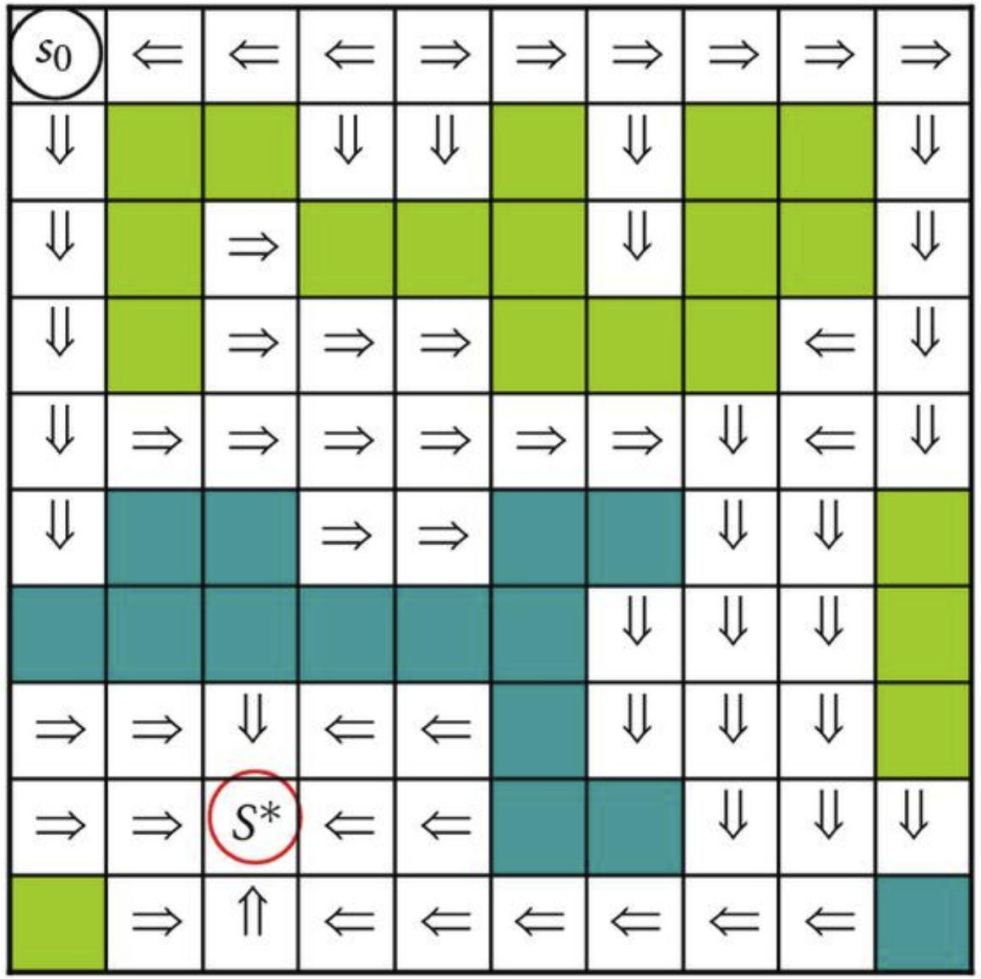
# Example Model-Free RL

- Initialize Q-function
- For All Episodes:
  - Initialize s
  - For All Time Steps in this Episode:
    - Select a $\epsilon$-greedy from Q(s)
    - Perform a in Environment giving s' and r
    - $Q(s,a) \leftarrow Q(s,a)+\alpha[r+\gamma \max_a Q(s',a)-Q(s,a)]$
    - $s \leftarrow s'$
- return Q



# Example Model-Based RL

- Initialize Q-function
- Repeat
  - Initialize s; a←$\pi$(s); (s',r)←Env(s,a) :: **Learn**
  - Q(s,a) ← Q(s,a)+$\alpha$[r+$\gamma$max$_a$Q(s',$_a$)-Q(s,a)]
  - M(s,a) ← (s',r) :: **Model**
  - For n=1, …, N :
    - Select $\hat{s}$ and $\hat{a}$ randomly
    - (s',r) ← M($\hat{s}$,$\hat{a}$) :: **Plan for FREE!**
    - Q($\hat{s}$,$\hat{a}$) ← Q($\hat{s}$,$\hat{a}$)+$\alpha$[r+$\gamma$max$_a$Q(s',$_a$)-Q($\hat{s}$,$\hat{a}$)]
- Until Q converges
- return Q



# Sample Complexity

- Model-based RL reduces sample complexity.
- As soon as Model has enough transition entries, the policy can be learned from the Model, for free.
- This free learning is called planning. It does not involve environment samples, hence, “free”.

# Why Planning?

- Internal to Agent
- Agent has state
- Agent can undo; allows to retrace your steps
- Agent can backtrack to try another action
- Planning is reversible learning
- Learning changes Environment state that the Agent cannot undo

# Trade-off Sample Complexity vs Model Quality

- Free planning concept only works if Model is perfect. When Environment is high dimensional, it will never be fully sampled, so Model will be imperfect.
- Learning T to reduce Env samples
- T is a function, a neural network
- With high dimensional problems, T is high capacity, so needs many samples to prevent overfitting
- Trade-off sample complexity vs quality of model

# Dealing with Imperfect Models

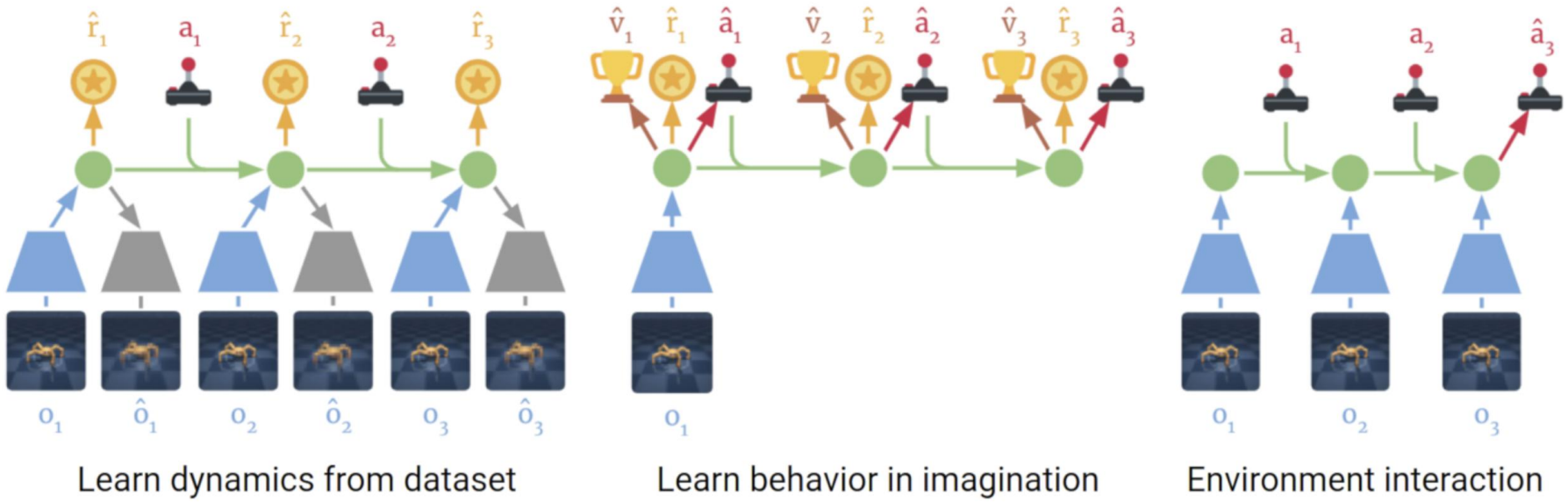
- Improving Model Learning
  - Modeling Uncertainty
  - Latent Models
- Improving Planning with Imperfect Model
  - Trajectory Rollouts and Model-Predictive Control
  - End-to-end learning and planning

# Modelling Uncertainty

- Gaussian Processes
  - PILCO, GPS, SVG
  - Works, but computationally expensive
- Ensembles
  - PETS
- Knowing uncertainty allows better planning
- Do planning sampling from distribution, plan with locally-linear search or with stochastic trajectory optimiser
- Does not scale to high dimensional problems

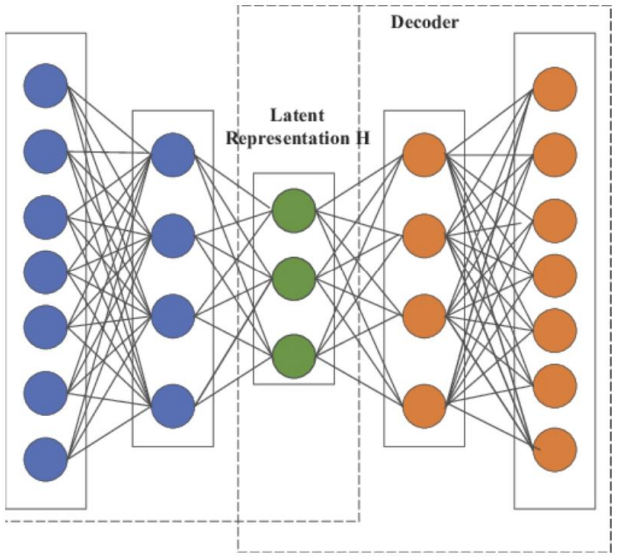
# Latent Models

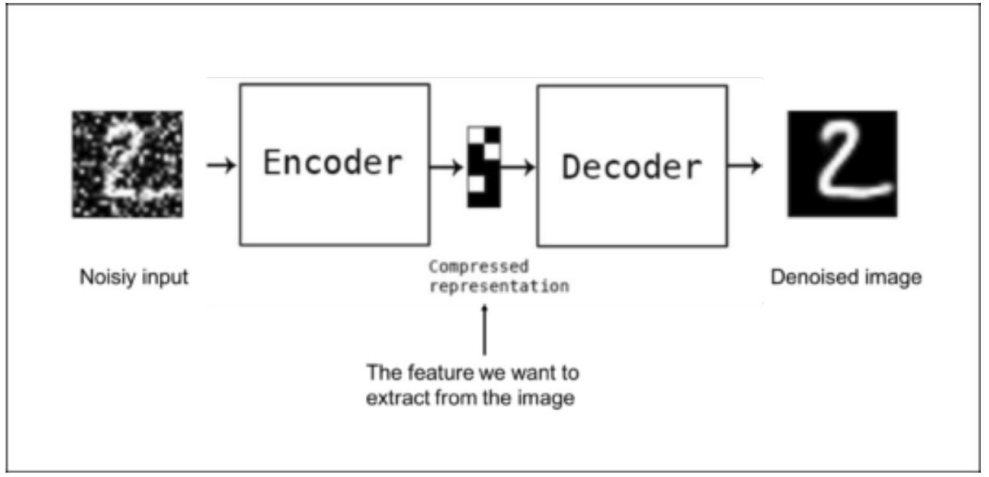
- Compress Observation Space into (smaller) Latent Space by modeling on value prediction
- Plan in small Latent Space [Dreamer]



# Latent Models

- Compression: Autoencoder
- Many networks, Complicated architecture, Good performance
- PlaNet, Dreamer, VPN



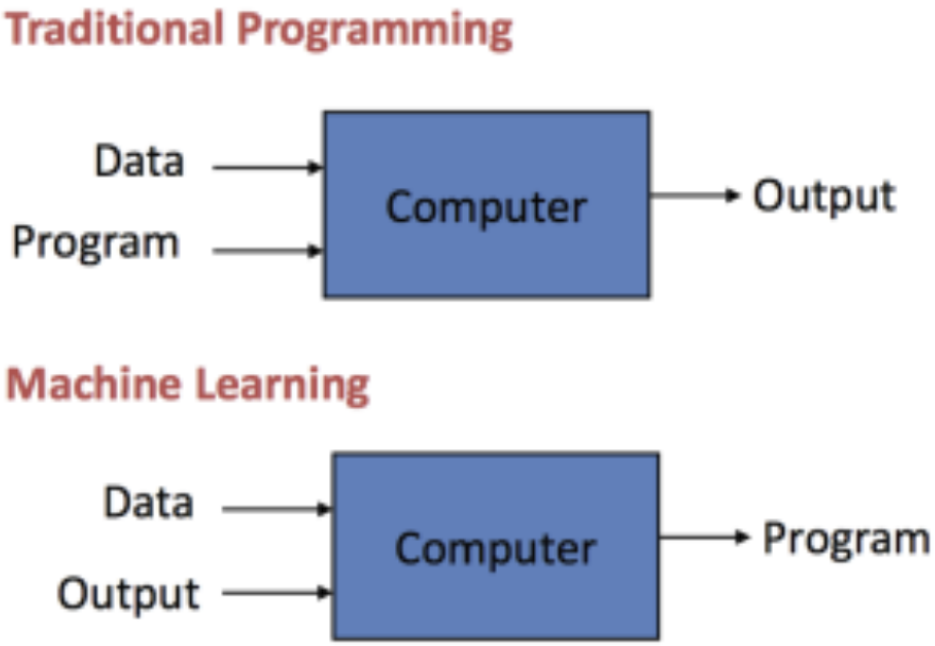



# Trajectory MPC

- Short trajectory rollouts
  - Reduce lookahead depth
  - Splits rollouts in near future (planned) and far-future (model free) -> MVE
- Model-predictive Control, Decision-time planning
  - Highly non-linear function are often locally linear
  - MPC: optimize model over limited time, and re-learn -> PETS

# End-to-End Learning/Planning

- Trend for programming by example
- Can a neural network do planning? (with backtrack?)
- Learn differentiable planning



# Value Iteration

Initialize V(s) to arbitrary values

Repeat until V(s) converge

For all states

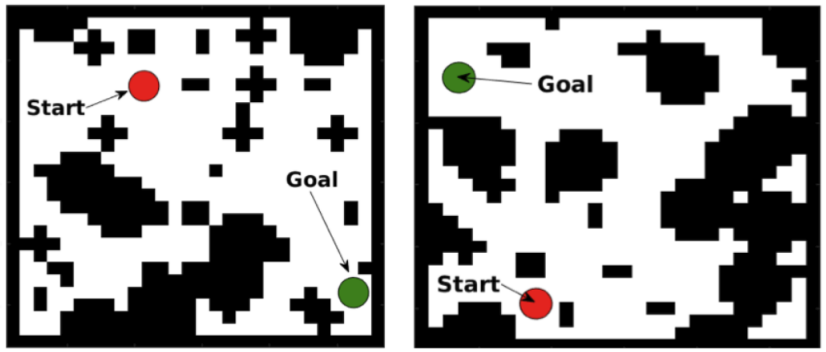
For all actions

$$Q(s,a) \leftarrow \sum_{s'} P^{a}_{ss'}(r(s,a) + \gamma V(s'))$$

$$V(s) \leftarrow \max_{a} Q(s,a)$$

# End-to-End Learning/Planning

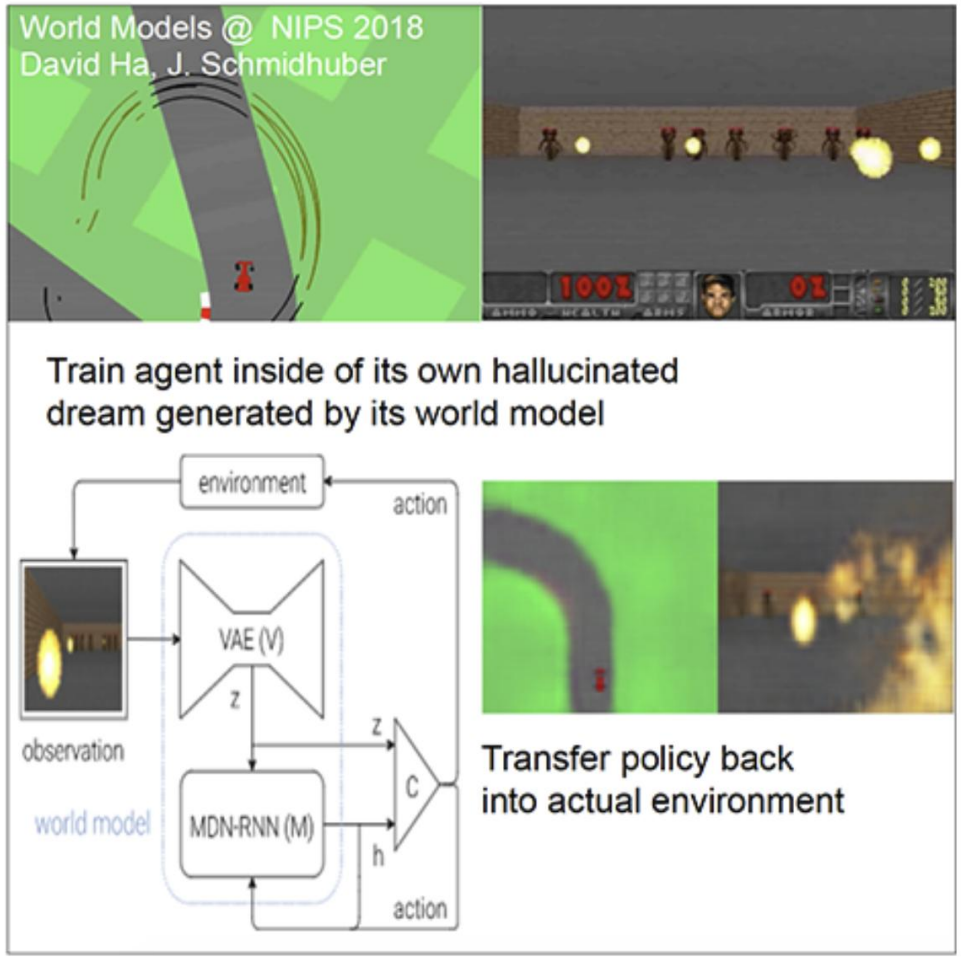
- Value Iteration Network
- Each layer one step of Value Iteration
- Learn $\sum_{s' \in S} T_a(s, s')(R_a(s, s') + \gamma V[s'])$
- Learn different Mazes




Our main observation is that each iteration of VI may be seen as passing the previous value function V and reward function R through a convolution layer and max-pooling layer. In this analogy, each channel in the convolution layer corresponds to the Q-function for a specific action, and convolution kernel weights correspond to the discounted transition probabilities.

# End-to-End Learning/Planning

- Later: RNN/LSTM for state: VProp
- Latent and End-to-End:
  - TreeQN, Predictron, MuZero, I2A, World Model
- Elaborate, Complex systems



# Learning and Planning

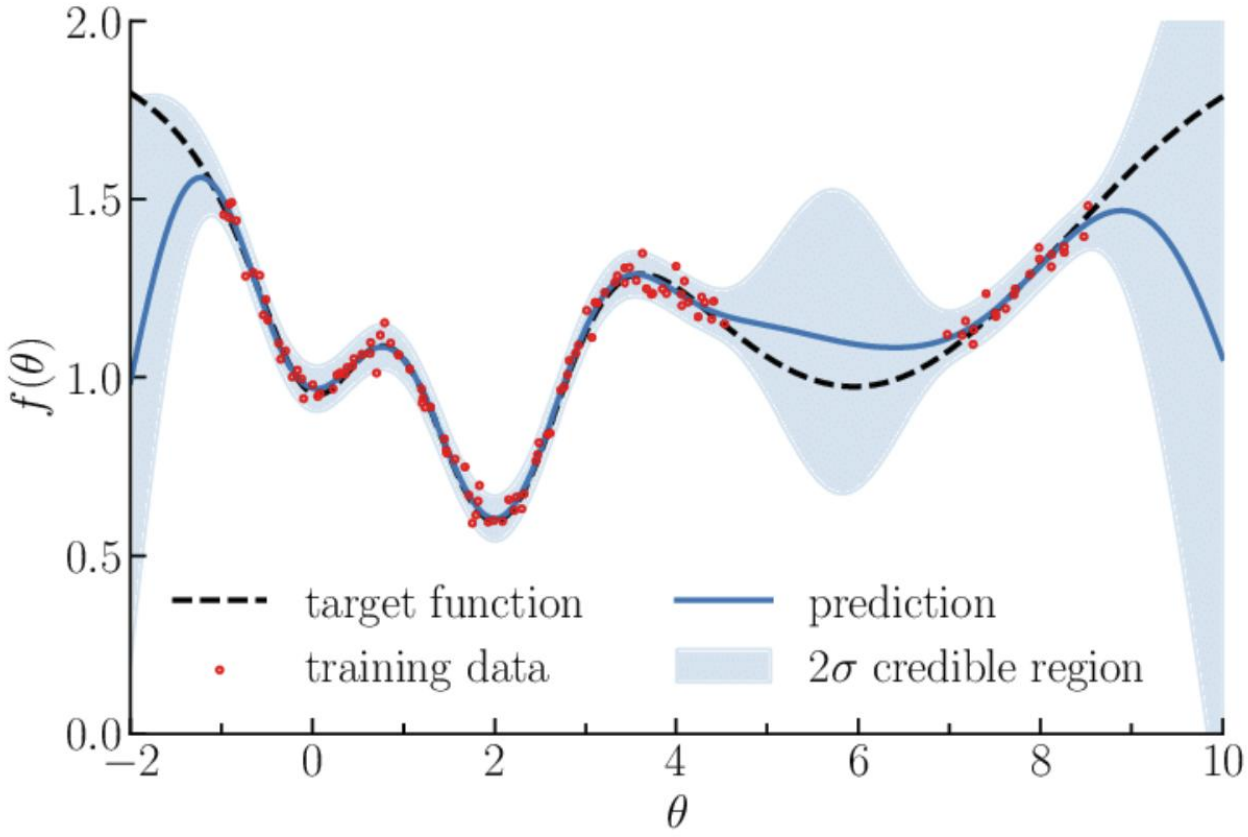
| | | LEARNING | |
|---|---|---|---|
| | | Uncertainty Ensembles CNN | Latent Models |
| PLANNING | Trajectory Model-predictive control | PILCO GPS SVG Local PETS MVE Meta | Dreamer Plan2Explore L3P VPN SimPLe Dreamer-v2 |
| | End-to-end | VIN VProp Network-Planning | TreeQN I2A Predictron World Model MuZero |

# Overview of Model-Based Approaches

| Name | Learning | Planning | Environment |
|---|---|---|---|
| PILCO | Uncertainty | Trajectory | Pendulum |
| iLQG | Uncertainty | MPC | Small |
| GPS | Uncertainty | Trajectory | Small |
| SVG | Uncertainty | Trajectory | Small |
| VIN | CNN | e2e | Mazes |
| VProp | CNN | e2e | Mazes |
| Planning | CNN/LSTM | e2e | Mazes |
| TreeQN | Latent | e2e | Mazes |
| I2A | Latent | e2e | Mazes |
| Predictron | Latent | e2e | Mazes |
| World Model | Latent | e2e | Car Racing |
| Local Model | Uncertainty | Trajectory | MuJoCo |
| Visual Foresight | Video Prediction | MPC | Manipulation |
| PETS | Ensemble | MPC | MuJoCo |
| MVE | Ensemble | Trajectory | MuJoCo |
| Meta Policy | Ensemble | Trajectory | MuJoCo |
| Policy Optim | Ensemble | Trajectory | MuJoCo |
| PlaNet | Latent | MPC | MuJoCo |
| Dreamer | Latent | Trajectory | MuJoCo |
| Plan2Explore | Latent | Trajectory | MuJoCo |
| $L^3P$ | Latent | Trajectory | MuJoCo |
| Video-prediction | Latent | Trajectory | Atari |
| VPN | Latent | Trajectory | Atari |
| SimPLe | Latent | Trajectory | Atari |
| Dreamer-v2 | Latent | Trajectory | Atari |
| MuZero | Latent | e2e/MCTS | Atari/Go |

# PILCO Uncertainty/Trajectory

- Gaussian processes
- Computationally expensive



# PETS Ensemble/MPC

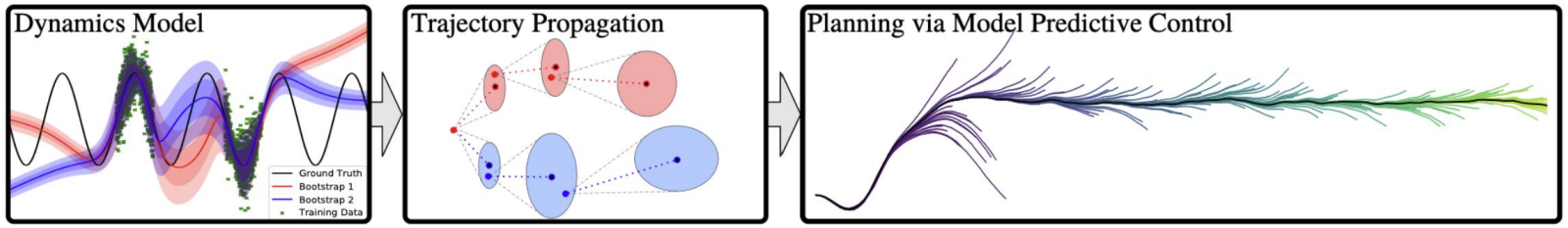



Figure 1: Our method (PE-TS): **Model**: Our probabilistic ensemble (PE) dynamics model is shown as an ensemble of two bootstraps (bootstrap disagreement far from data captures epistemic uncertainty: our subjective uncertainty due to a lack of data), each a probabilistic neural network that captures aleatoric uncertainty (inherent variance of the observed data). **Propagation**: Our trajectory sampling (TS) propagation technique uses our dynamics model to re-sample each particle (with associated bootstrap) according to its probabilistic prediction at each point in time, up until horizon $T$. **Planning**: At each time step, our MPC algorithm computes an optimal action sequence, applies the first action in the sequence, and repeats until the task-horizon.

(a) Cartpole (b) 7-dof Pusher

(c) 7-dof Reacher (d) Half-cheetah

# Value Prediction Network (Latent/Traj)

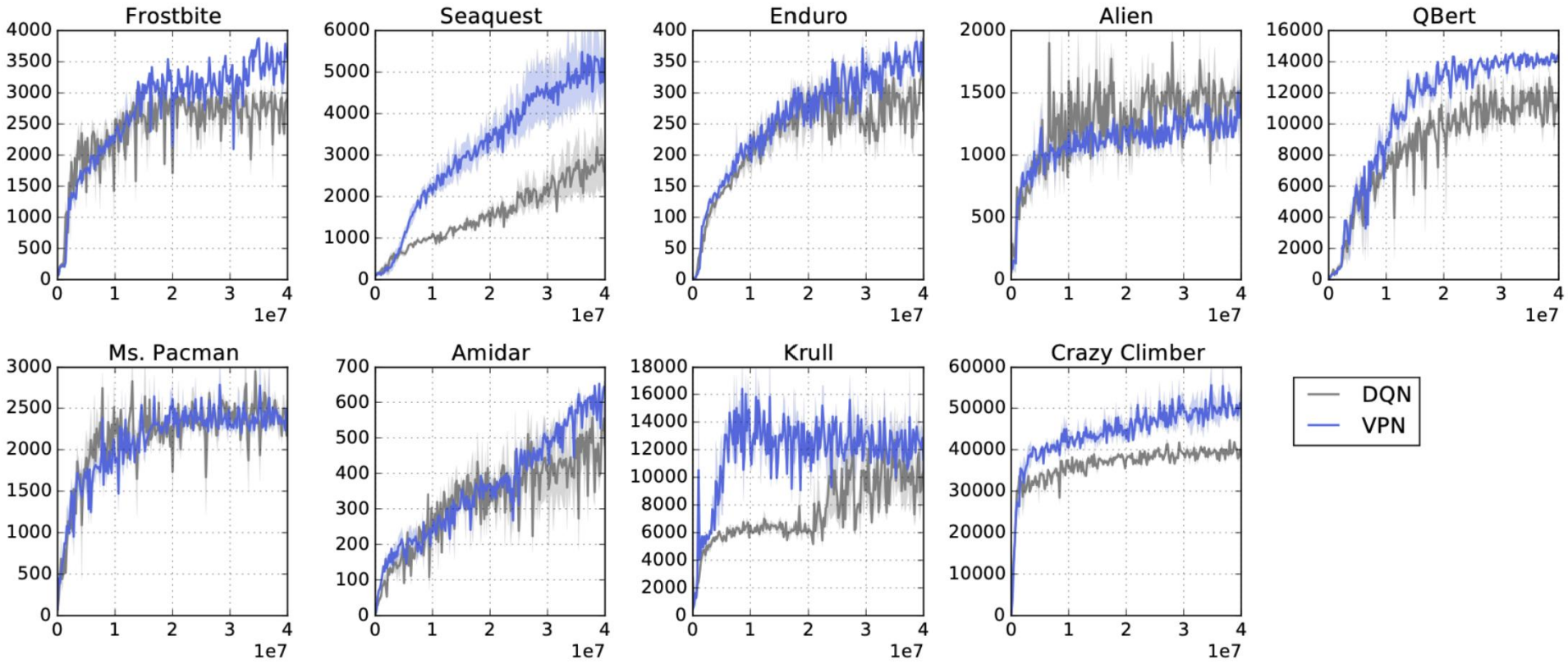
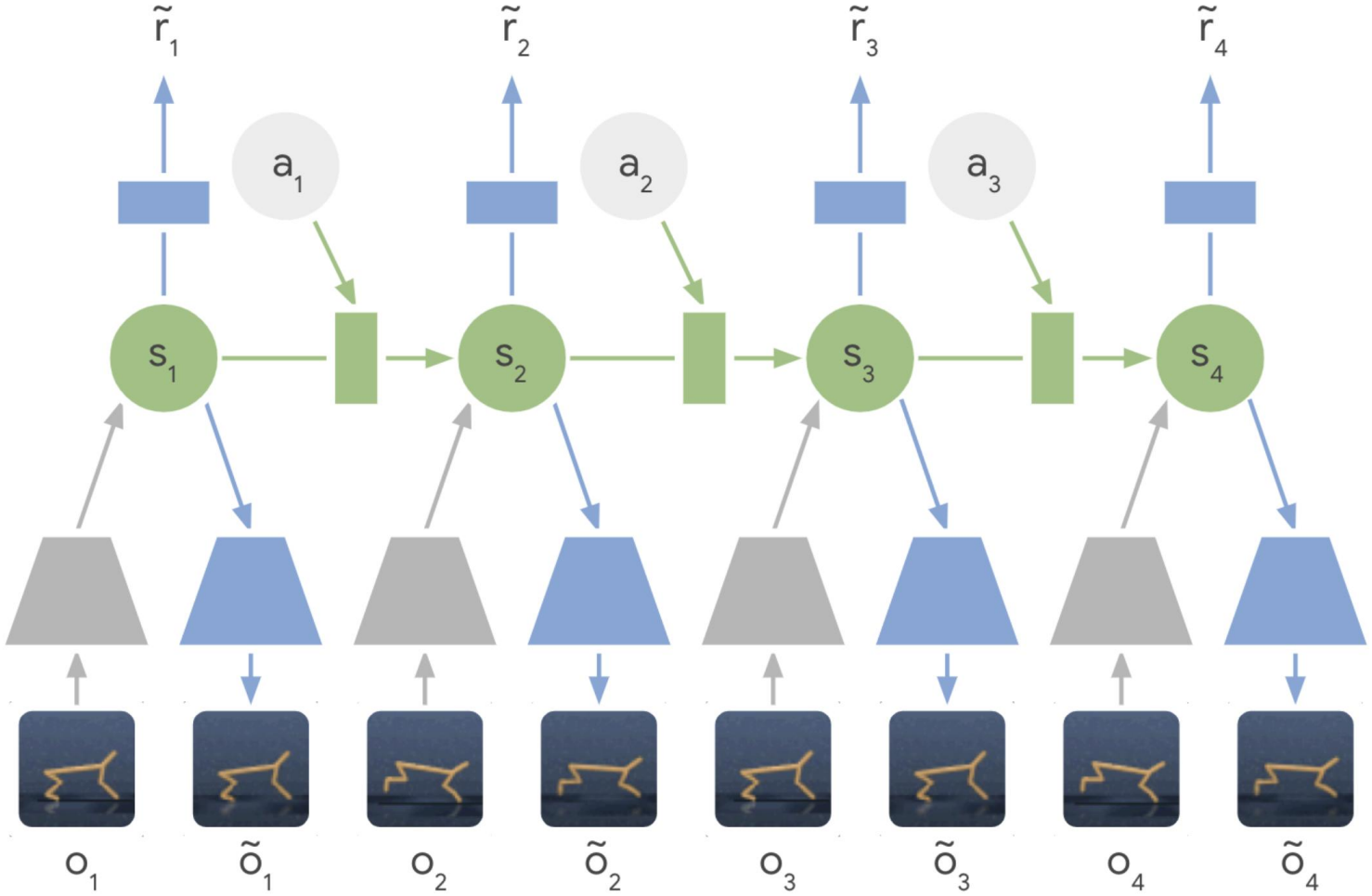



Figure 8: Learning curves on Atari games. X-axis and y-axis correspond to steps and average reward over 100 episodes respectively.

# Latent PlaNet



# Dreamer (Latent/Traj)


The three processes of the Dreamer agent. The world model is learned from past experience. From predictions of this model, the agent then learns a value network to predict future rewards and an actor network to select actions. The actor network is used to interact with the environment.

# I2A (Latent/E2E)

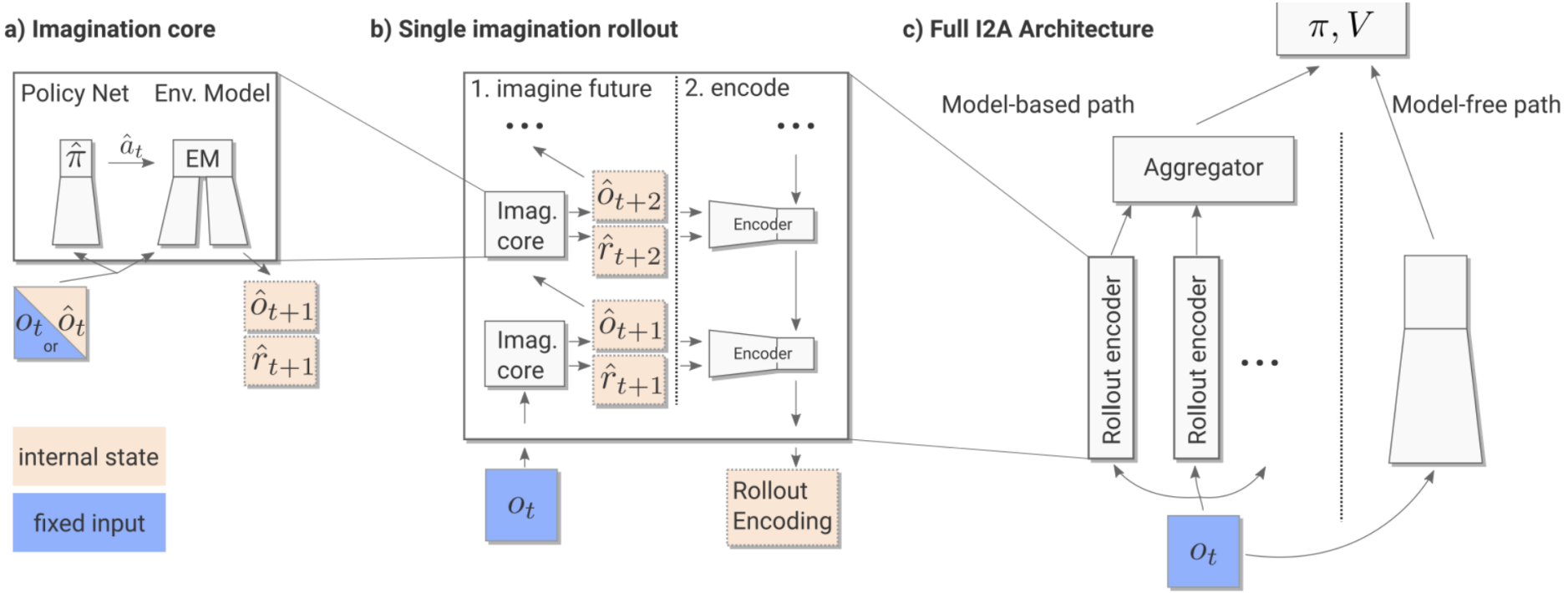
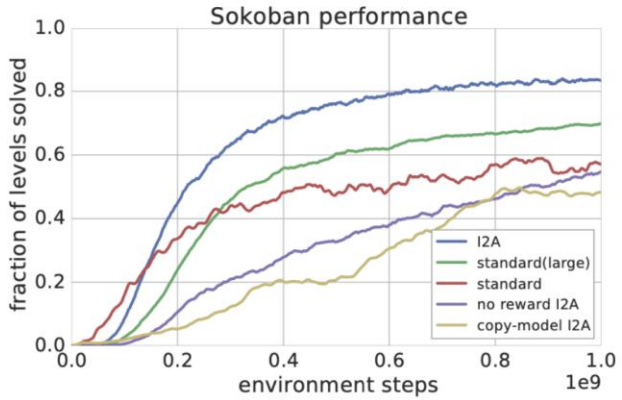



Figure 1: *I2A architecture*. $\hat{\cdot}$ notation indicates imagined quantities. *a)*: the imagination core (IC) predicts the next time step conditioned on an action sampled from the rollout policy $\hat{\pi}$. *b)*: the IC imagines trajectories of features $\hat{f} = (\hat{o}, \hat{r})$, encoded by the rollout encoder. *c)*: in the full I2A, aggregated rollout encodings and input from a model-free path determine the output policy $\pi$.

# I2A (Latent/E2E)

Figure 3: *Random examples of procedurally generated Sokoban levels.* The player (green sprite) needs to push all 4 boxes onto the red target squares to solve a level, while avoiding irreversible mistakes. Our agents receive sprite graphics (shown above) as observations.



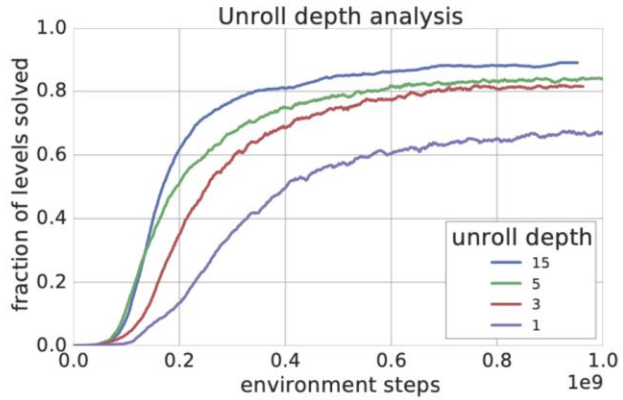
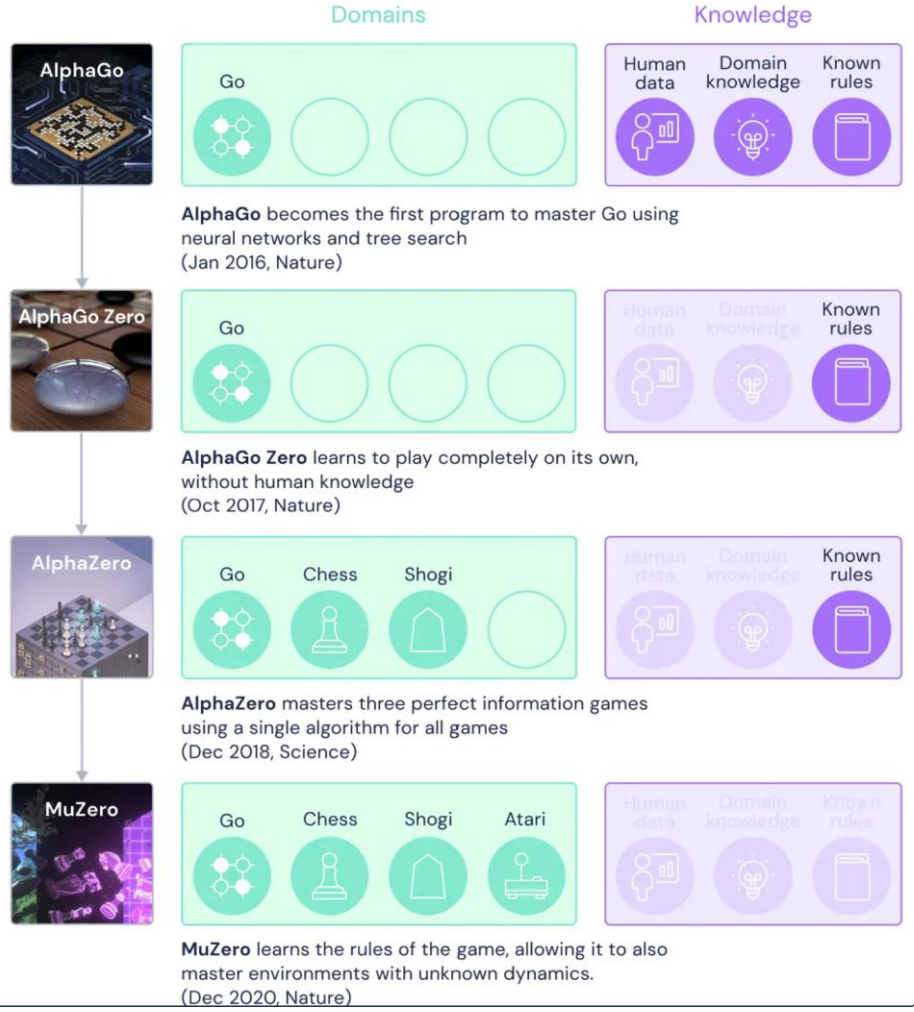
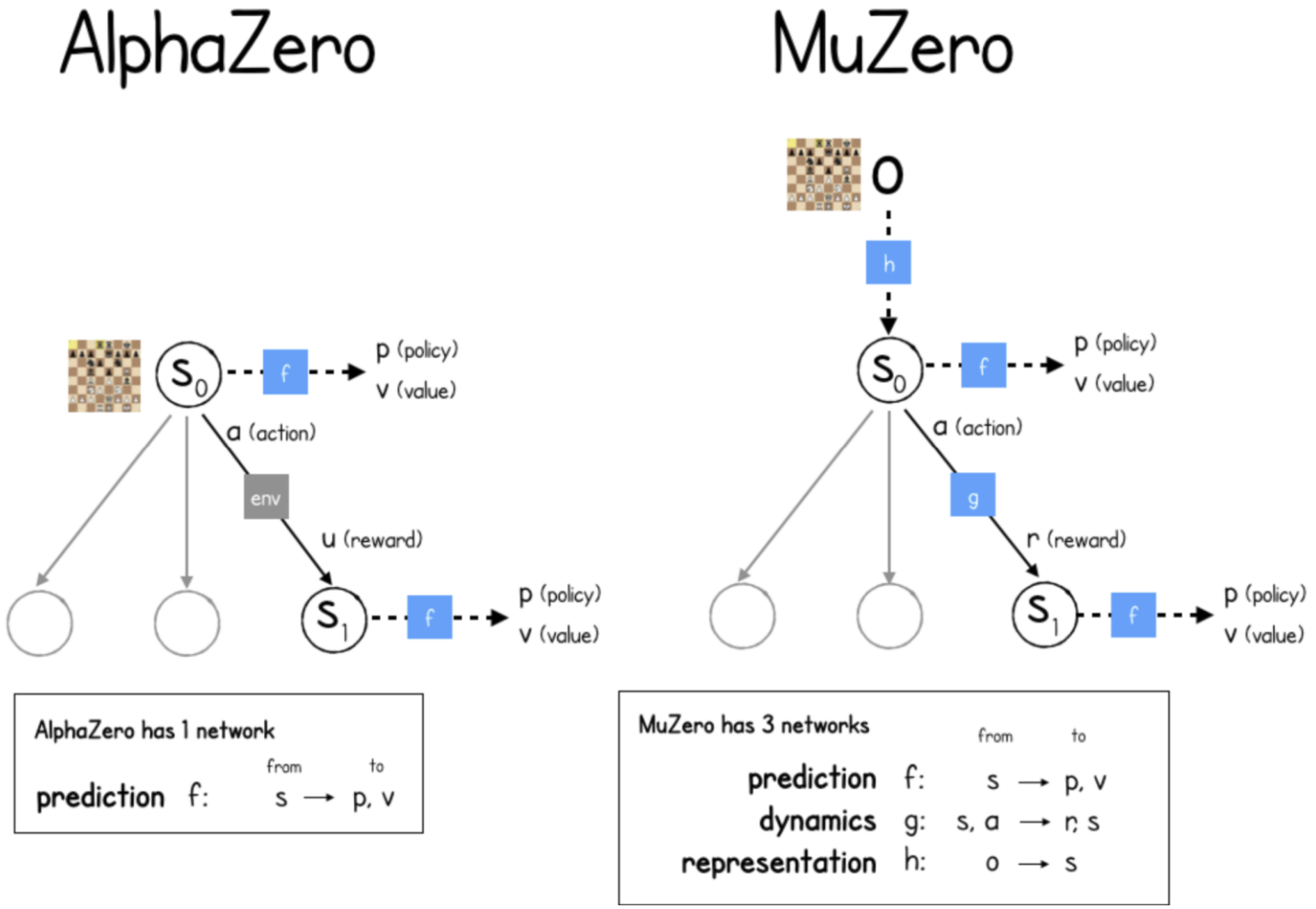




Figure 4: *Sokoban learning curves. Left:* training curves of I2A and baselines. Note that I2A use additional environment observations to pretrain the environment model, see main text for discussion. *Right:* I2A training curves for various values of imagination depth.

# MuZero (Latent/E2E)



# MuZero (Latent/E2E)



# MuZero (Latent/E2E)

representation $h_\theta(o_1, .., o_t) = s^0$

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

dynamics $g_\theta(s^{k-1}, a^k) = r^k, s^k$

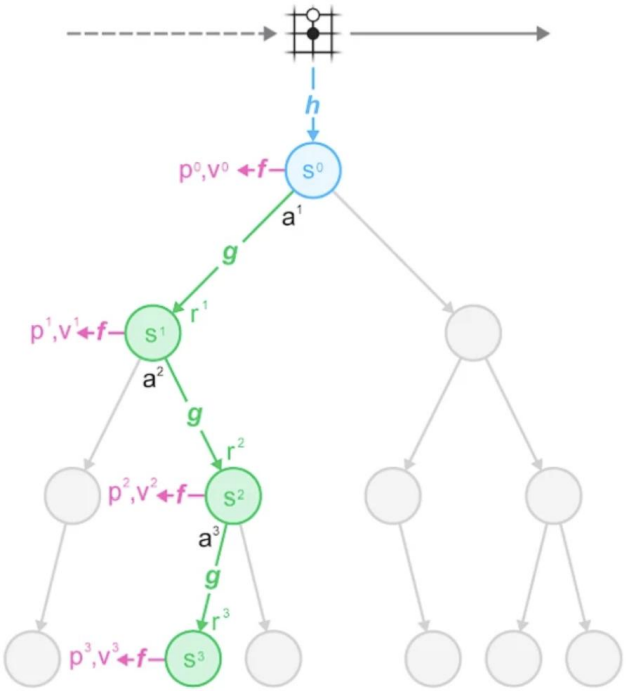
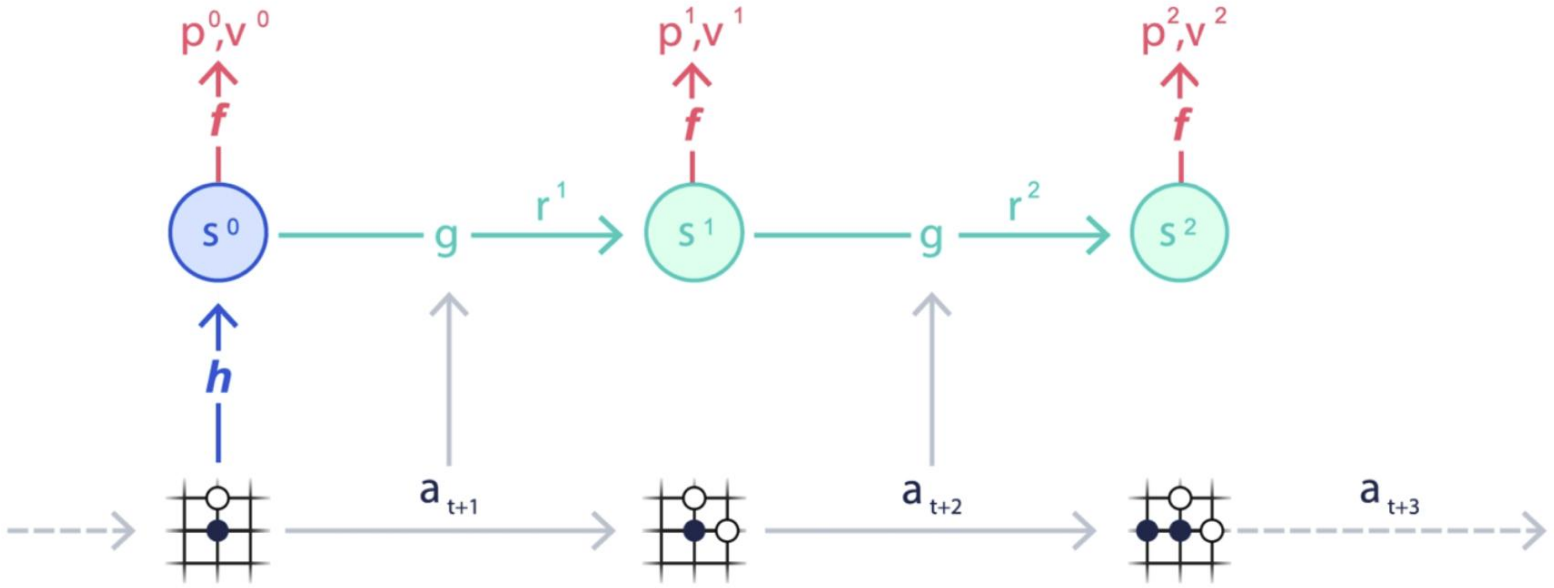



Illustration of how Monte Carlo Tree Search can be used to plan with the MuZero neural networks. Starting at the current position in the game (schematic Go board at the top of the animation), MuZero uses the representation function (h) to map from the observation to an embedding used by the neural network (s0). Using the dynamics function (g) and the prediction function (f), MuZero can then consider possible future sequences of actions (a), and choose the best action.

# MuZero (Latent/E2E)




During training, the model is unrolled alongside the collected experience, at each step predicting the previously saved information: the value function v predicts the sum of observed rewards (u), the policy estimate (p) predicts the previous search outcome (π), the reward estimate r predicts the last observed reward (u).

# Benchmark

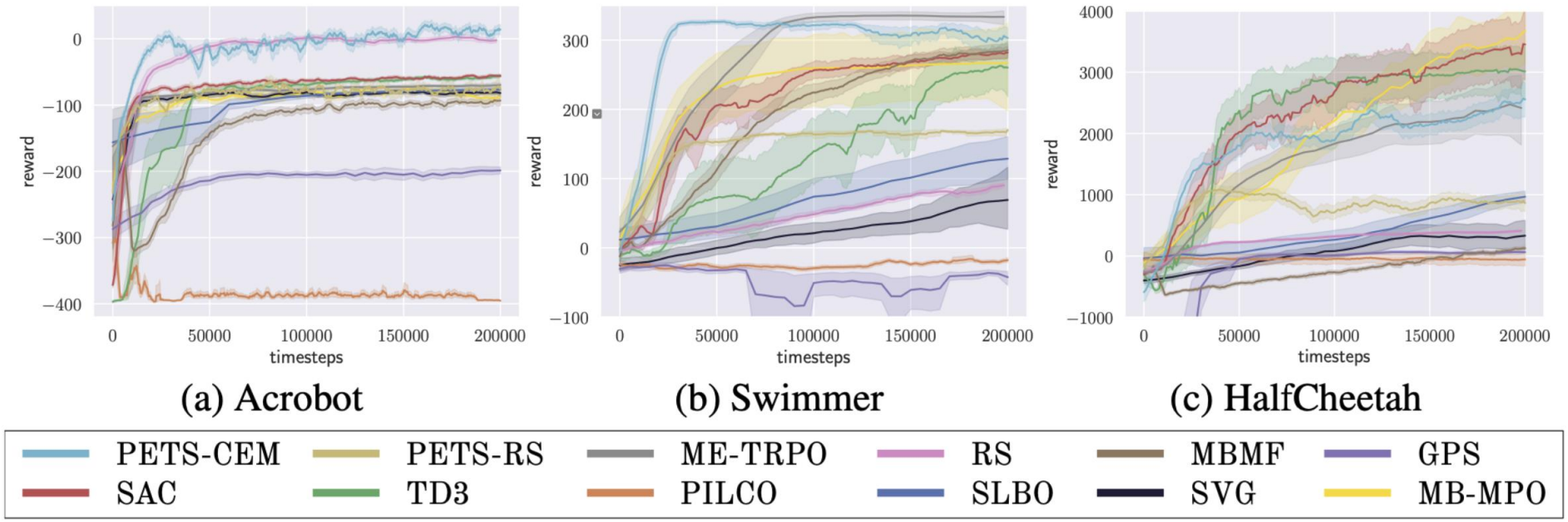



Figure 1: A subset of all 18 performance curve figures of the bench-marked algorithms. All the algorithms are run for 200k time-steps and with 4 random seeds. The remaining figures are in appendix C.

- SAC and TD3 are model-free baselines
- Model-based is sometimes better

# Conclusions

- What does the overview tell us?
- Accuracy of transition model: **Crucial**
- Sample complexity trade-off: **Success**
- Results sometimes better than model-free
- Still brittleness, sensitive to hyperparameters
- Still active field of research

# What Transition Model Perfectly Known?

- Previous chapter showed that accuracy of model is important.
- What if we have a perfect transition function?
- What if we can also use it to learn?
- Then World Champions get beaten:
  - Backgammon
  - Go
  - Chess
  - Shogi
- Today is about Best Case, when everything fits together and works

# Self-Play is Old

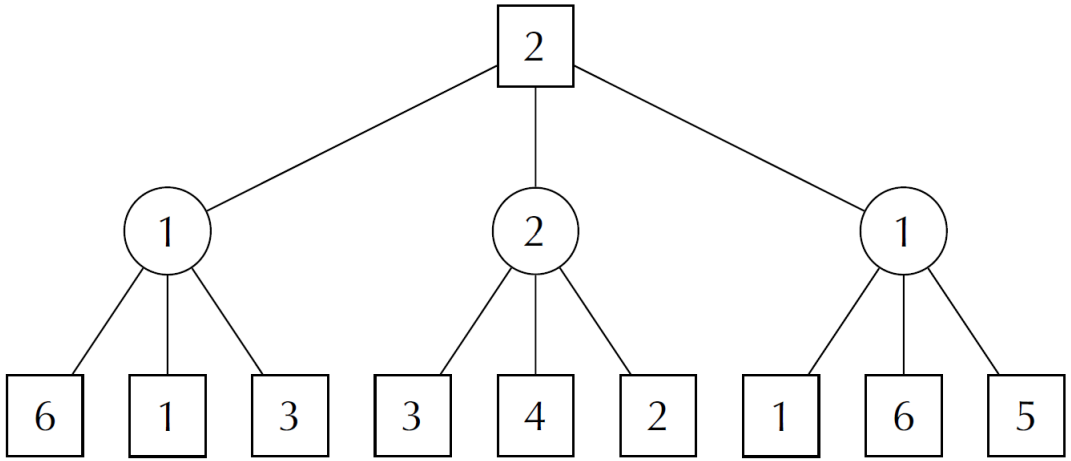
- Most two-agent board game-playing programs choose (versions of) themselves as opponent for simulation or learning.
- Minimax (1949) is self-play
- Samuel's checkers players (1950-1960) used self-play outcomes for modifying evaluation weights.
- TD-Gammon (1992) used self-play learning
- However, self-play is potentially unstable due to feedback and deadly triad
- It is overcome in AlphaGo in different ways

# AlphaZero – Three Levels of Self-Play

- Move-level self-play (minimax, MCTS)
- Example-level self-play (learning, Actor Critic)
- Game-level self-play (curriculum, self transcending player)

# Minimax

- Assume you play best move, and opponent has your knowledge
- Two-agent zero sum: my win is your loss
  - Max/min/max/min/max/min
  - Max: Square
  - Min: Circle
  - b=branching factor, d=search depth



# Minimax

```
INF = 99999

def eval(n):
    if n['type'] == 'LEAF':
        return n['value']
    else:
        error("Calling␣eval␣not␣on␣LEAF")

def minimax(n):
    if n['type'] == 'LEAF':
        return eval(n)
    elif n['type'] == 'MAX':
        g = -INF
        for c in n['children']:
            g = max(g, minimax(c))
    elif n['type'] == 'MIN':
        g = INF
        for c in n['children']:
            g = min(g, minimax(c))
    else:
        error("Wrong␣node␣type")
    return g

print("Minimax␣value:␣", minimax(root))
```

Listing 4.2: Minimax code

# State Space Complexity

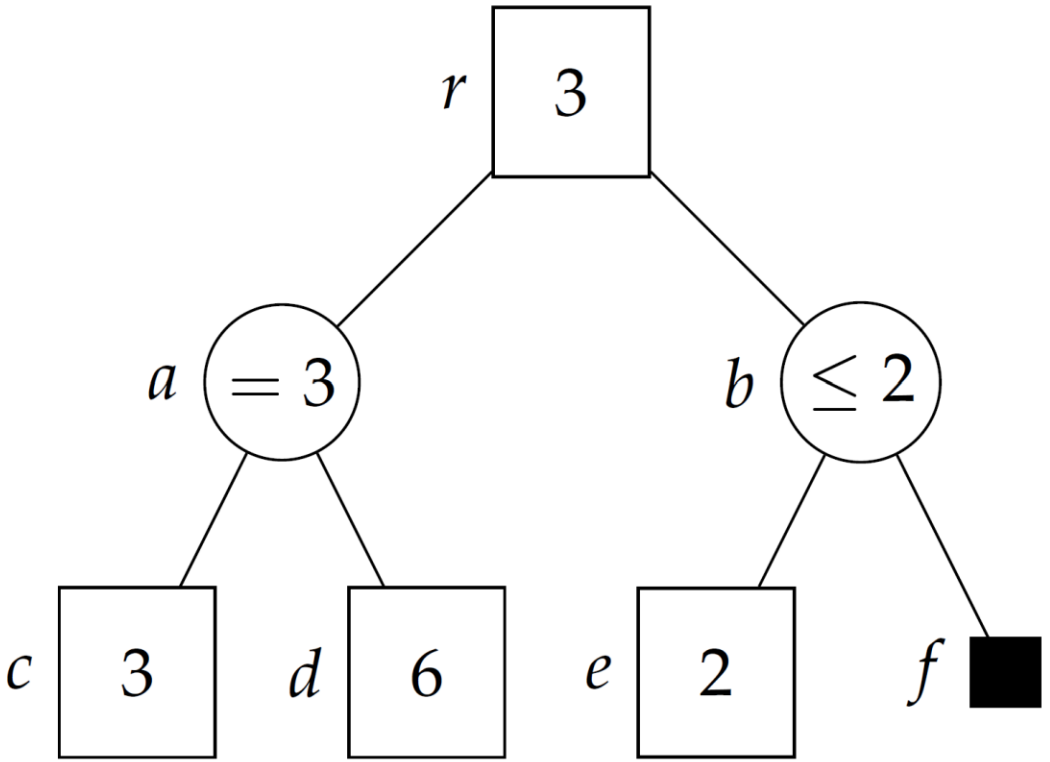
| Name | board | state space | zero sum | information | turn |
|---|---|---|---|---|---|
| Chess | $8 \times 8$ | $10^{47}$ | zero sum | perfect | turn |
| Checkers | $8 \times 8$ | $10^{18}$ | zero sum | perfect | turn |
| Othello | $8 \times 8$ | $10^{28}$ | zero sum | perfect | turn |
| Backgammon | 24 | $10^{20}$ | zero sum | chance | turn |
| Go | $19 \times 19$ | $10^{170}$ | zero sum | perfect | turn |
| Shogi | $9 \times 9$ | $10^{71}$ | zero sum | perfect | turn |
| Poker | card | $10^{161}$ | non-zero | imperfect | turn |
| StarCraft | real time strategy | $10^{1685}$ | non-zero | imperfect | simultaneous |

- 10^47... 1 ns/position -> 10^38 s -> 10^30 earth-year
- 10^21 times the age of the known universe/position

# Heuristics

- Material (pawns, bishops, knights, …)
- Mobility (# actions)
- Center control
- King Safety
- …

# Alpha-Beta Pruning




- A cutoff is an action (e) that is so strong for my opponent that I will not play b (because a is better) and hence we can stop searching b

# After Chess?

# Go!

# State Space Complexity

| Name | board | state space | zero sum | information | turn |
|---|---|---|---|---|---|
| Chess | $8 \times 8$ | $10^{47}$ | zero sum | perfect | turn |
| Checkers | $8 \times 8$ | $10^{18}$ | zero sum | perfect | turn |
| Othello | $8 \times 8$ | $10^{28}$ | zero sum | perfect | turn |
| Backgammon | 24 | $10^{20}$ | zero sum | chance | turn |
| Go | $19 \times 19$ | $10^{170}$ | zero sum | perfect | turn |
| Shogi | $9 \times 9$ | $10^{71}$ | zero sum | perfect | turn |
| Poker | card | $10^{161}$ | non-zero | imperfect | turn |
| StarCraft | real time strategy | $10^{1685}$ | non-zero | imperfect | simultaneous |

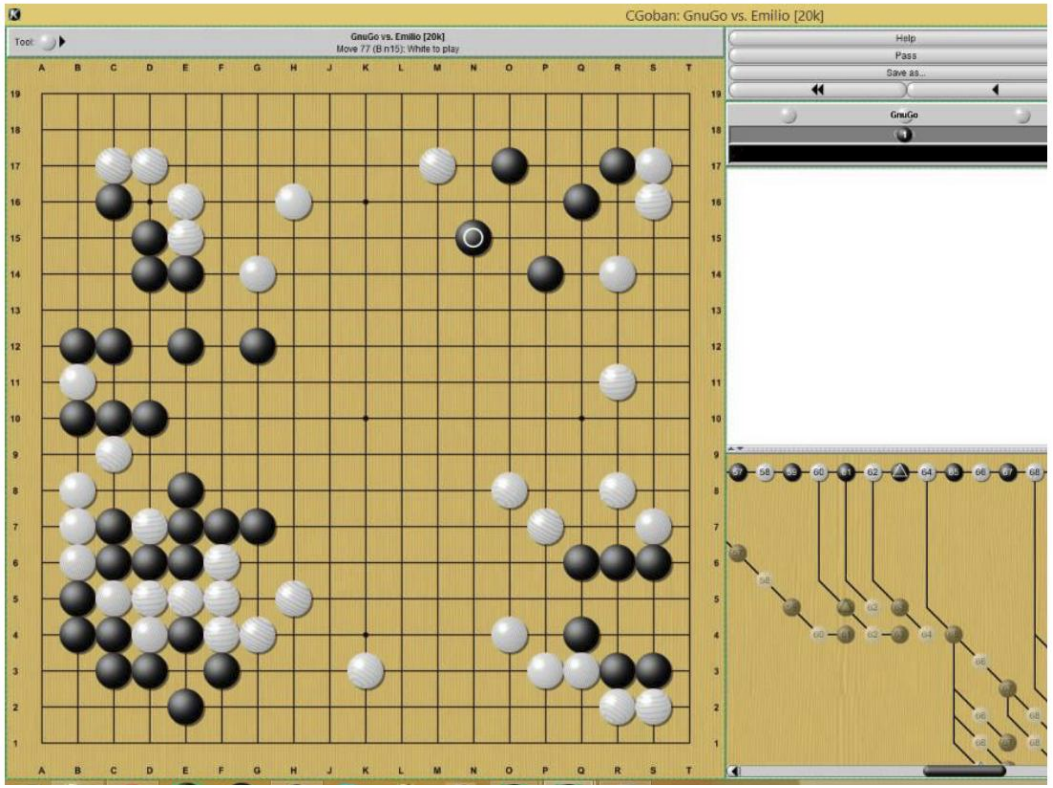
- Go even more complex than chess

# Heuristic Search

- Successful in games with tactical play, where efficient heuristics can be found
- Pieces move, and material is a good indicator of the score
- In Go, board is large, pieces do not move, material is typically balanced, and “influence” turned out to be difficult to program efficiently

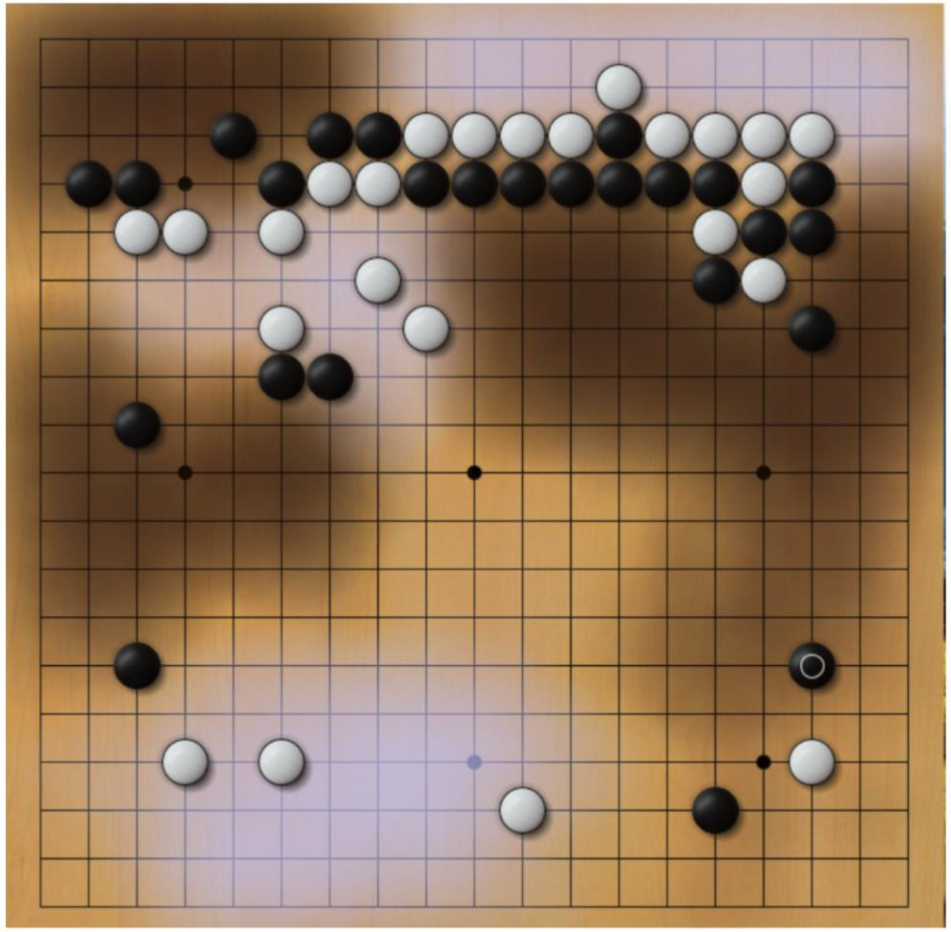
# Traditional Go Programs

- Minimax
- Forward Pruning:
  - Only try “sensible” actions such as connect, defend, territory jump
  - Like a knowledge-based expert system
- Influence calculation for scoring
- Weak amateur level (10 kyu)
- Years of no real progress
- Then: MCTS

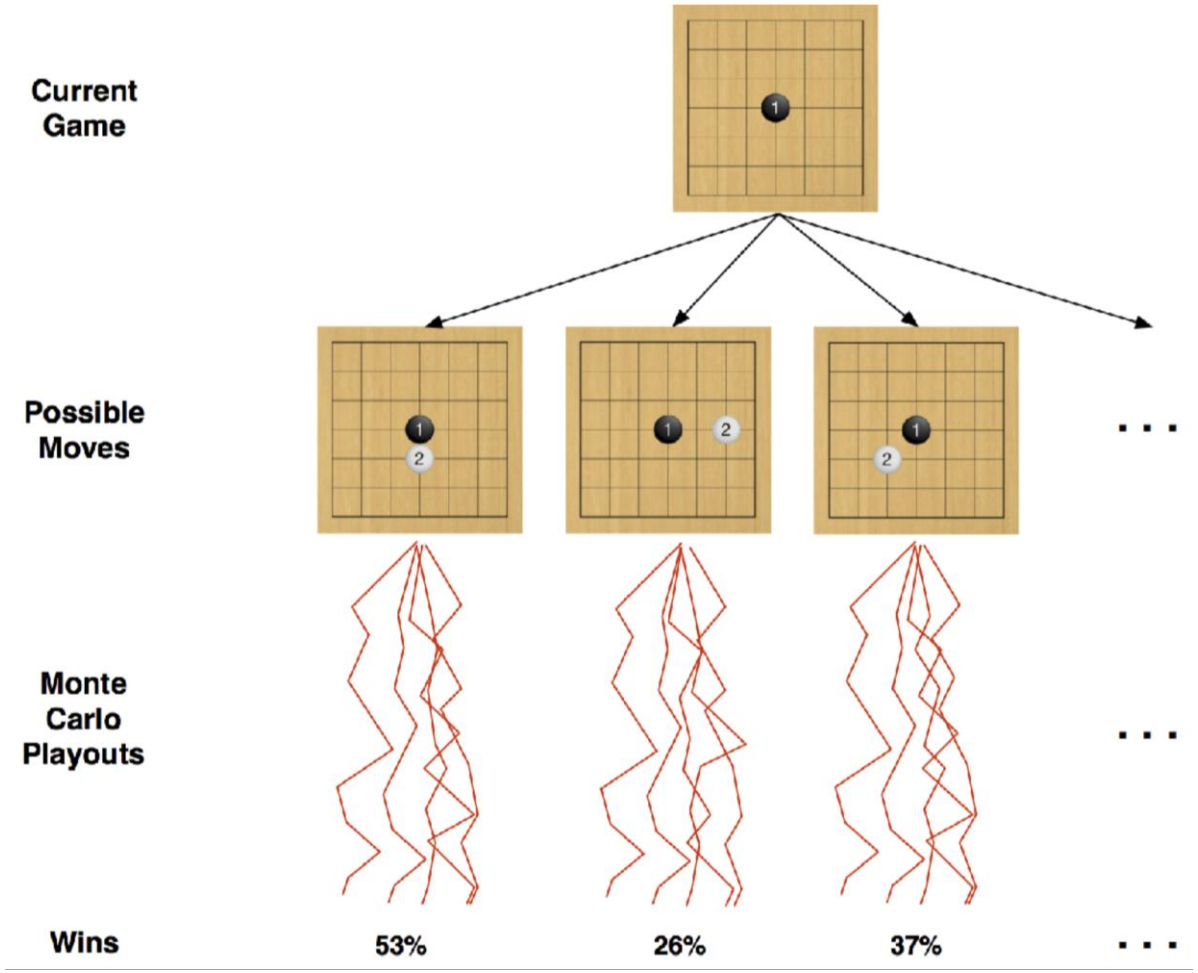


# MCTS – Adaptive Sampling

- No Efficient Heuristics for influence
- Rigid Search does not work well in large, flat state space



# Monte-Carlo Playouts



# Monte-Carlo Playouts

- Chess: b=10. Full width
- Go: b=200. Forward pruning
- Playout: Not search Tree b^d but search Path d

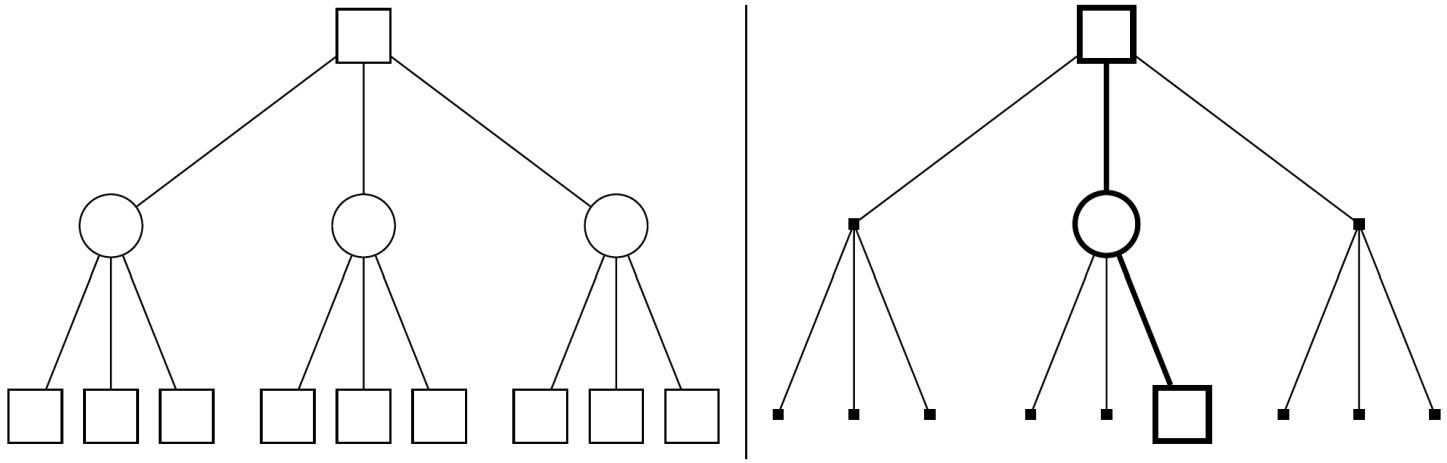


Figure 4.1: Searching a Tree vs a Path

# Monte-Carlo vs Minimax

- Minimax: Best of all actions
- Monte Carlo: Average of random playouts

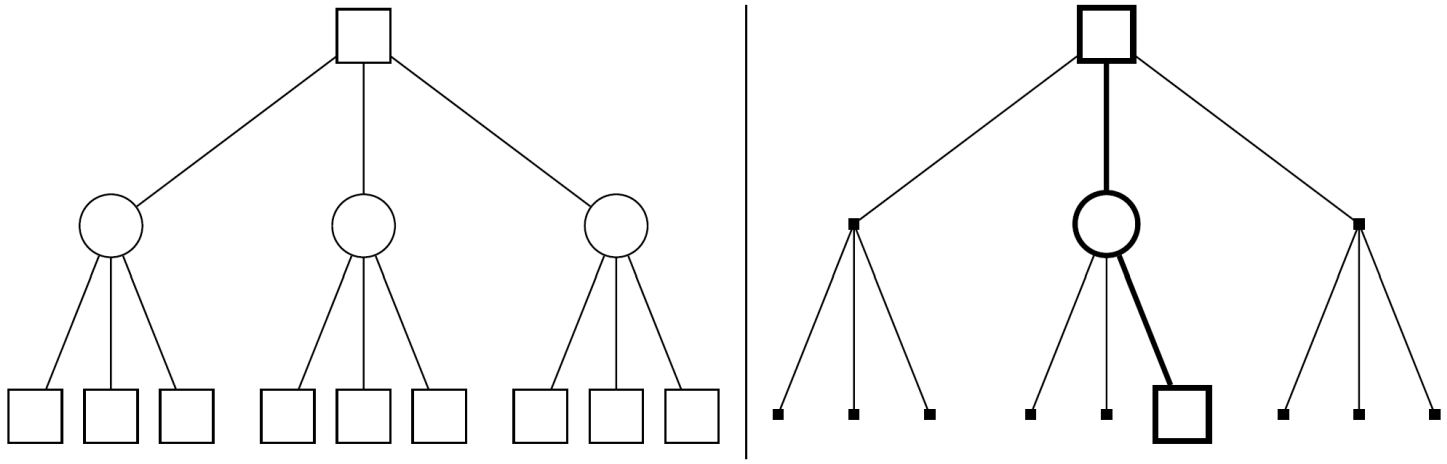


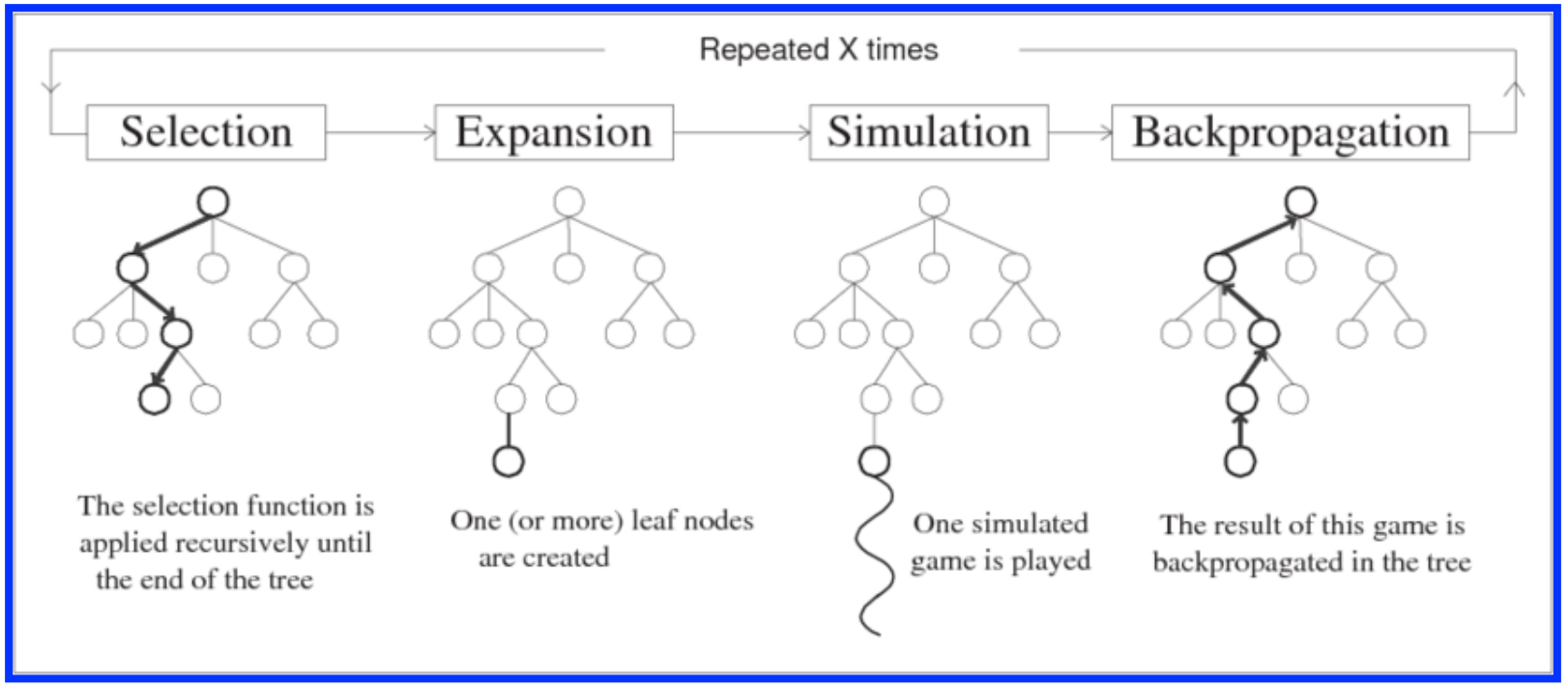
Figure 4.1: Searching a Tree vs a Path

# Monte-Carlo Tree Search

- Brügmann 1993 tried all playouts from the root (flat)
  Results were better than random, but not great -> MC
- Coulom 2006 (after work of others) tried it recursively, in a tree.
  This did give good results -> MCTS
- Kocsis & Szepesvari 2006 suggested the UCT selection rule to balance exploration and exploitation. Based on extensive work in multi-armed bandit theory.

# 4 Steps in MCTS

MCTS maintains a search tree and grows it on each iteration using the following steps:




Starting at the root of the search tree, choose a move using the selection policy, repeating the process until a leaf node is reached

Grow the search tree by generating a new child/children.

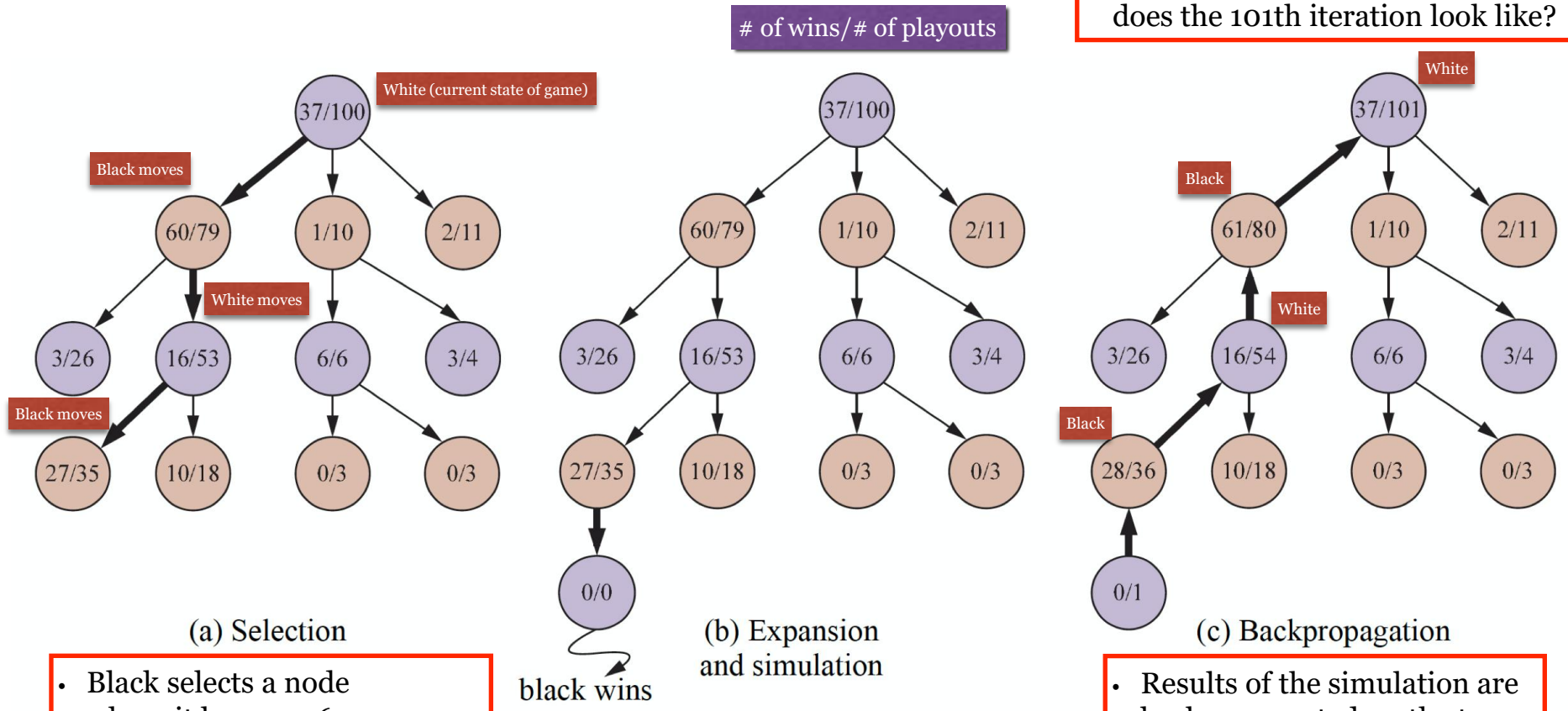
Perform a playout from a child using the playout policy. These moves are not recorded in the search tree

Use the simulation result to update the utilities of the nodes going back up to the root.

After X times: Choose the best move from start state

# One Iteration of MCTS

- White has previously moved.
- What should blacks move be (2nd level)?
- White has won 37 out of 100 playouts (37/100) done so far
- Suppose we will do 1000 iterations. What does the 101th iteration look like?




- Black selects a node where it has won 60 of 79 playouts (60/79)
- Uses UCT selection metric
- Selection continues to a leaf node where black has won 27 out of 35 playouts (27/35)

- Generate a new child node labeled 0/0
- Execute a playout
- Black wins this simulation

- Results of the simulation are back propagated up the tree branch.
- Black won, so black nodes are incremented in # of wins/# of playouts
- White loses, white nodes are incremented in number of playouts only.

# UCT: A Selection Policy

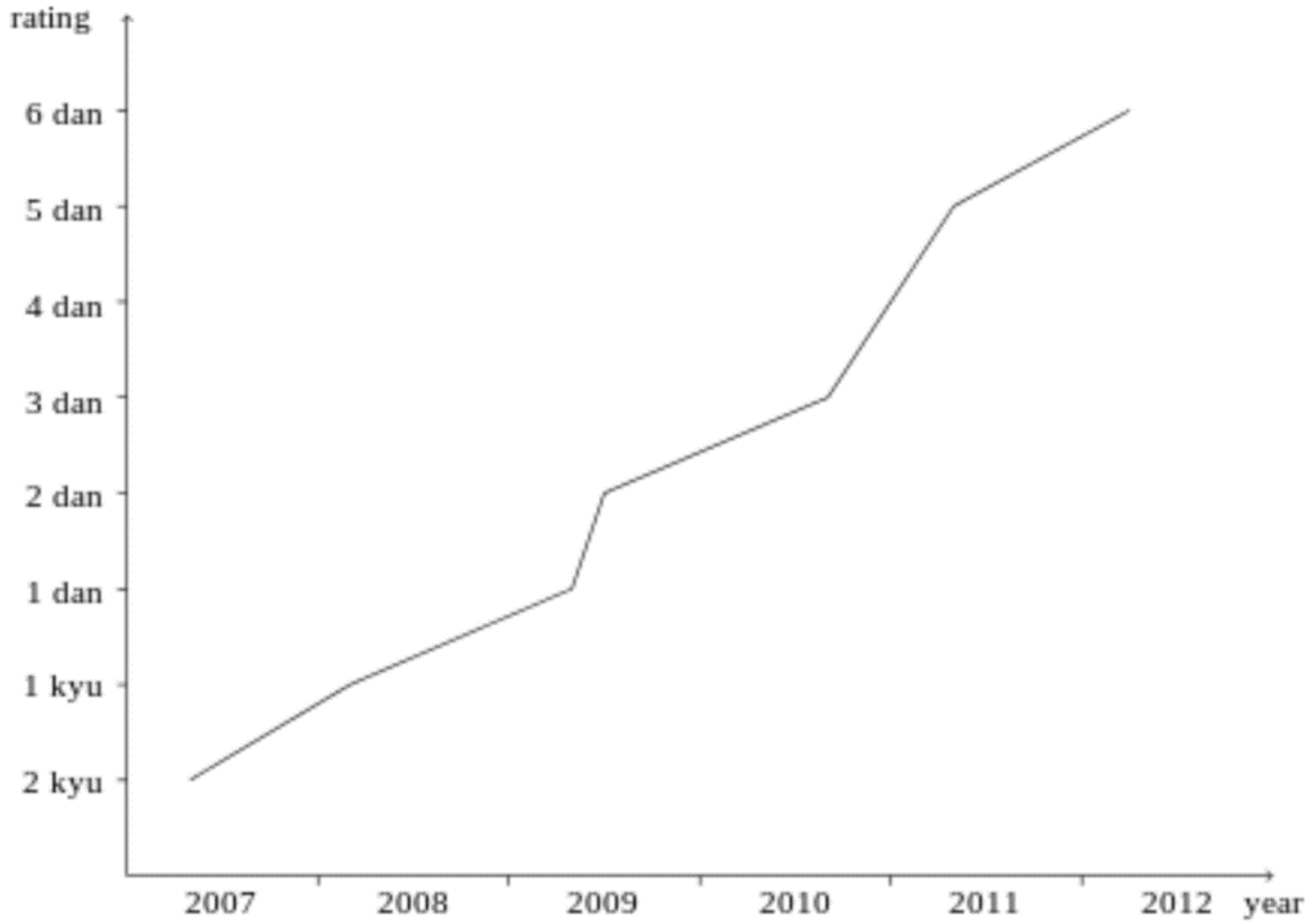
UCT: upper confidence bound applied to trees

Ranks each possible move based on an upper confidence bound formula called UCB1:

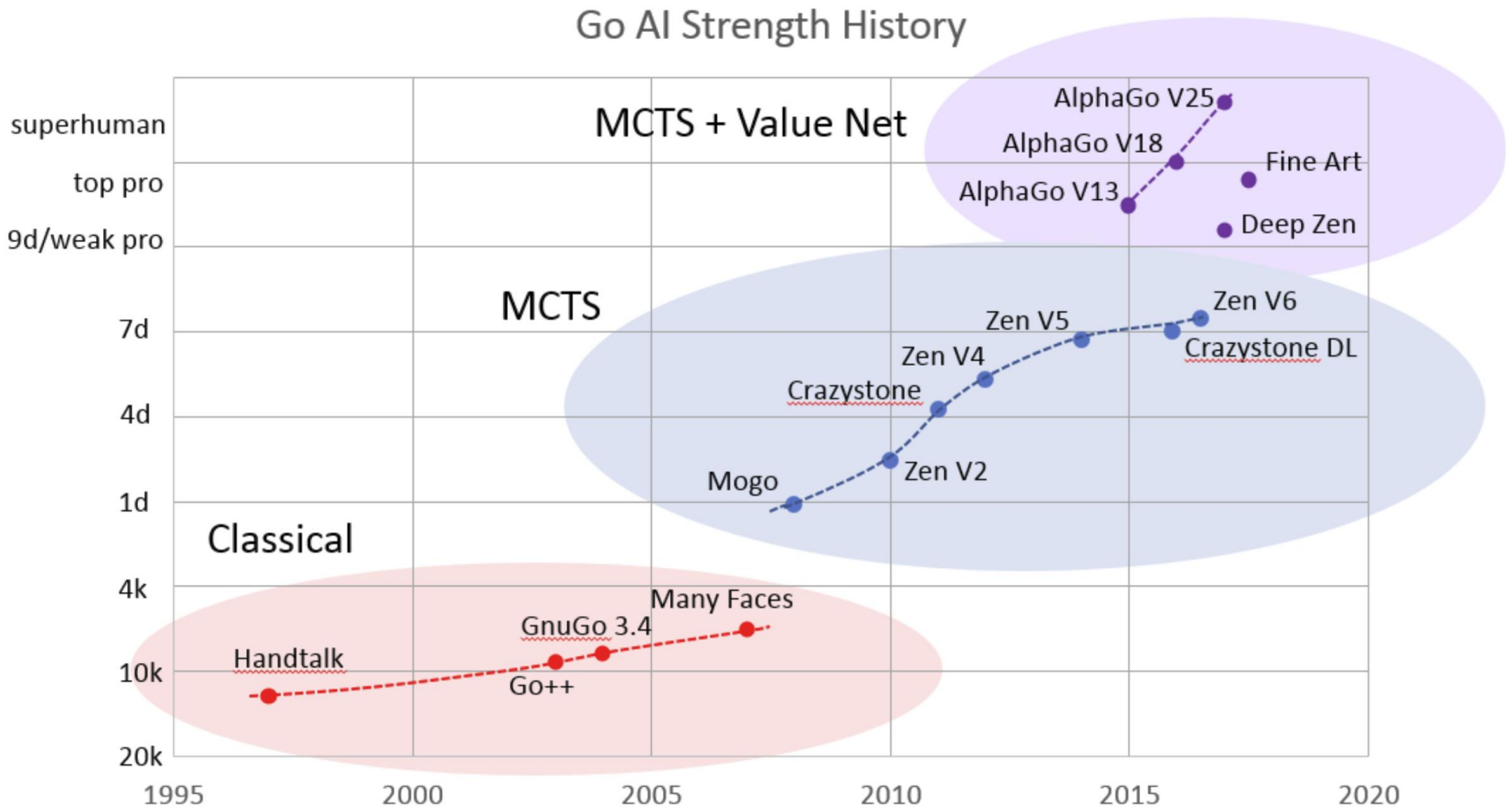
$$UCB1(n) = \frac{U(n)}{N(n)} + C \times \sqrt{\frac{lnN(Parent(n))}{N(n)}}$$

- $U(n)$: Total utility of all playouts that go through $n$
- $N(n)$: The number of playouts through node $n$
- $Parent(n)$: The parent node of node $n$
- $\frac{U(n)}{N(n)}$-term: is the exploitation term. The average utility of $n$. For example win percentage.
- $\sqrt{\ }$ - term : is the exploration term.
  - Numerator: $ln$ of the number of times we have explored the parent
    - If $n$ is selected some non-zero % of the time, the exploration term goes to zero as the counts increase, and eventually the playouts are given to the node with the highest average utility.
  - Denominator: count $N(n)$
    - The exploration term will be high for nodes only explored a few times
- $C$: Constant that balances exploitation and exploration.
  - Theoretically, $\sqrt{2}$ is best value for $C$, but this constant is often learned or chosen through trial and error.
  - $C = 1.4$ would choose the $60/79$ (more exploitation) node in the example during *Selection*, while $C = 1.5$ would choose the $2/11$ node (more exploration) during Selection.

# MCTS Impact on Go Programs Strength



# MCTS Go


Go AI Strength History
superhuman
top pro
9d/weak pro
7d
4d
1d
4k
10k
20k
1995
2000
2005
2010
2015
2020
MCTS + Value Net
AlphaGo V25
AlphaGo V18
AlphaGo V13
Fine Art
Deep Zen
MCTS
Zen V6
Zen V5
Zen V4
Crazystone DL
Crazystone
Zen V2
Mogo
Classical
Many Faces
GnuGo 3.4
Handtalk
Go++

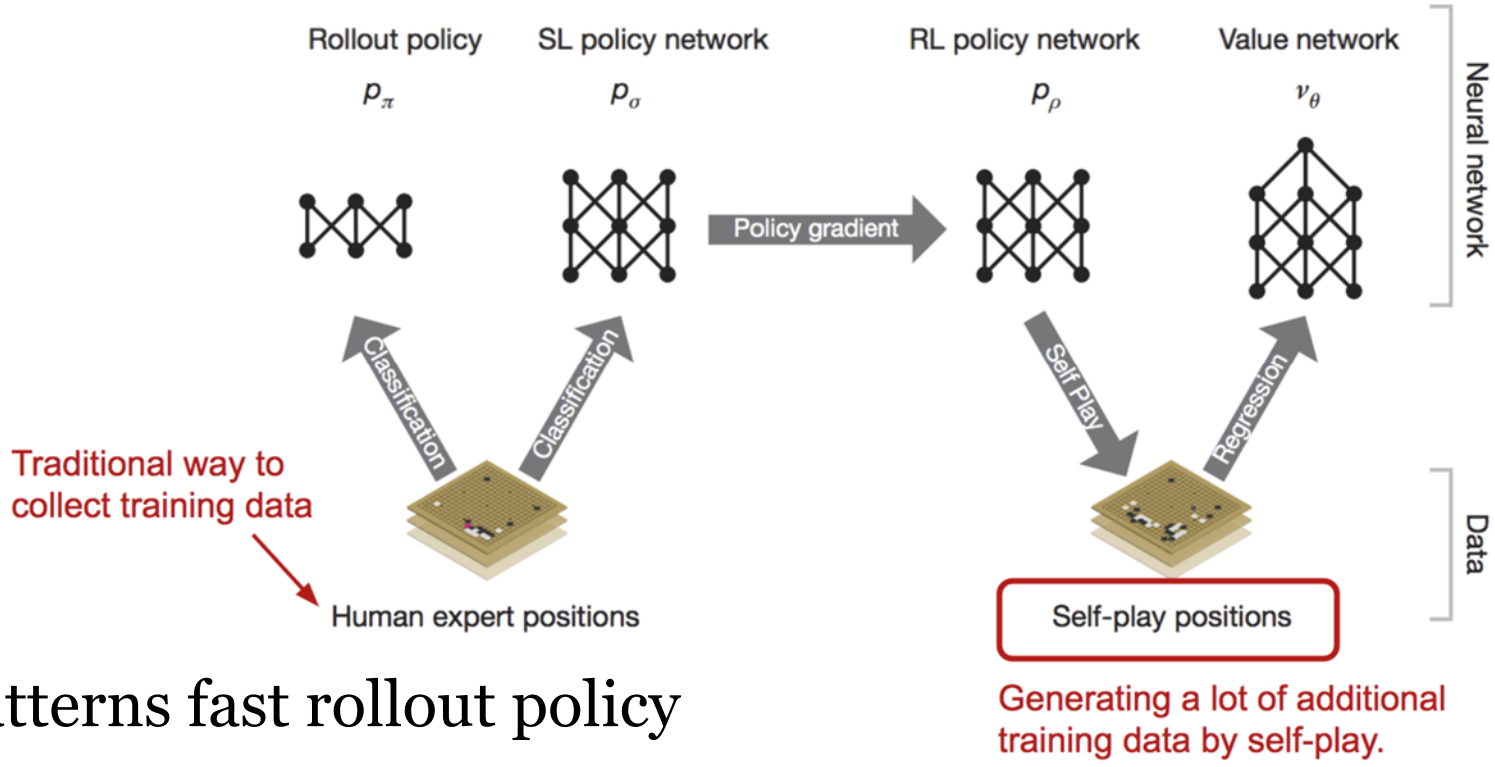
# AlphaGo 1, 2, 3

- AlphaGo: The Champion
- AlphaGo Zero: Tabula Rasa
  The Self-Learner
- AlphaZero: Three games: Chess, Shogi, Go
  The Generalist



# AlphaGo Structure

- 4 Nets
  - Fast rollout policy
  - Slow sl policy
  - Slow rl policy
  - Value net
- 3 Learning methods
  - Supervised small patterns fast rollout policy
  - Supervised database grandmaster games
  - Reinforcement from database de-correlated self-play games



# Policy, Playout, and Value

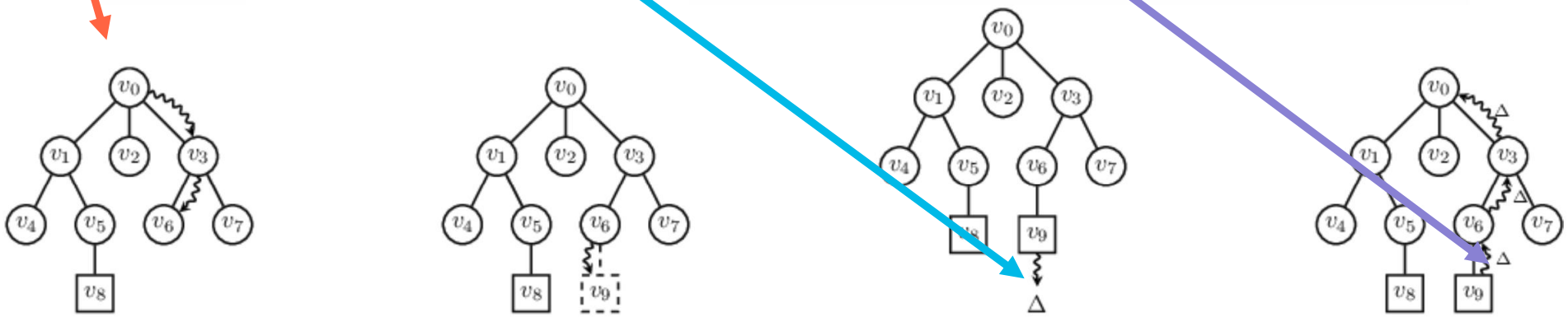



Figure 1: One iteration of MCTS.

# AlphaGo Overview

based on: Silver, D. et al. Nature Vol 529, 2016
copyright: Bob van den Hoek, 2016

Input Board Position
as 19 x 19 Image
48 Feature Planes

**Policy Network**
Expert Games
130 000 Games
30 M Positions
Supervised Learning
SL Policy
Position --> Next Move
Accuracy: 56%

**Fast Policy Network**
Expert Games
140 000 Patterns
130 000 Games
30 M Positions
Supervised Learning
Fast Policy
Pattern --> Next Move
Accuracy: 24%

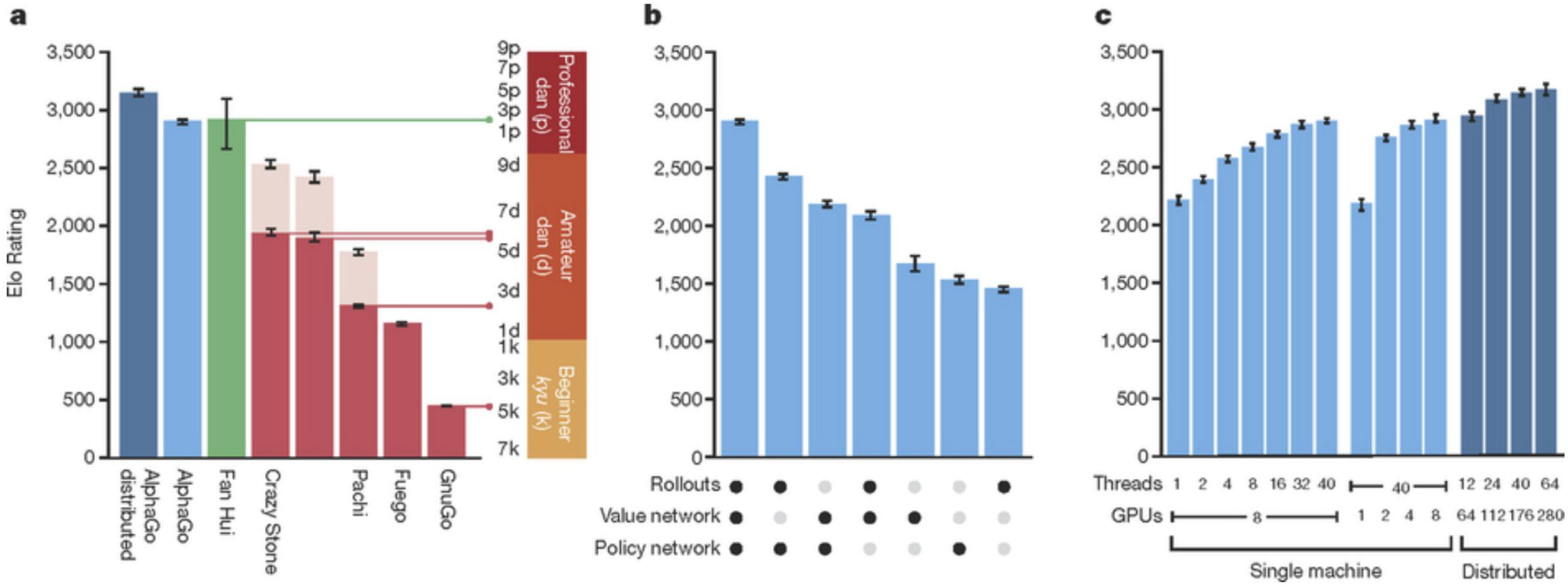
**Reinforcement Learning Policy Network**
Self-Play Games
1.3 M Games by
various versions
of RL Policy
Reinforcement Learning
RL Policy
Position --> Next Move
Wins 80% vs. SL Policy

**Value Network**
Self-Play Games
30 M Positions
by fixed version
of RL Policy
Reinforcement Learning
Position --> Win Probability
15 000 times faster than
MCTS Rollouts evaluations

**AlphaGo**

**Monte Carlo Tree Search**

# AlphaGo Performance



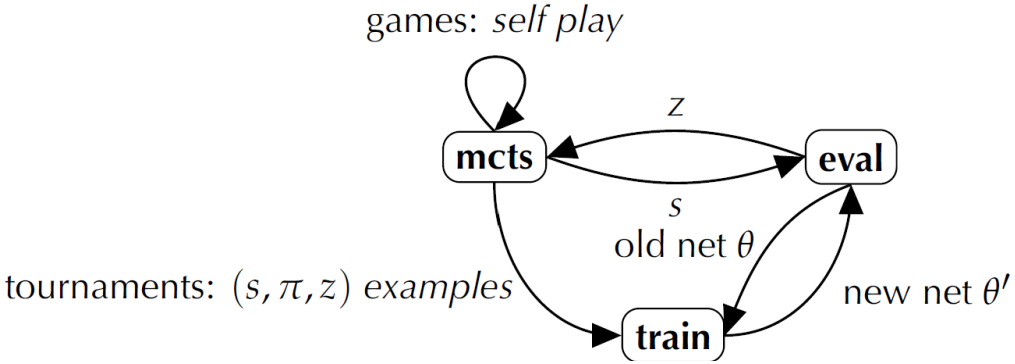

nature
THE INTERNATIONAL WEEKLY JOURNAL OF SCIENCE
At last — a computer program that can beat a champion Go player PAGE 484
ALL SYSTEMS GO
CONSERVATION
SONGBIRDS A LA CARTE
Illegal harvest of millions of Mediterranean birds
PAGE 402
RESEARCH ETHICS
SAFEGUARD TRANSPARENCY
Don't let openness backfire on individuals
POPULAR SCIENCE
WHEN GENES GOT 'SELFISH'
Dawkins's calling card 40 years on
PAGE 402
ARTICLE
Mastering the game of Go with deep neural networks and tree search
ARTICLE
Mastering the game of Go without human knowledge

# AlphaGo Zero

- Faster
  - Days, not weeks
- Better
  - Higher Elo
- Elegant
  - 1 network

# AlphaGo Zero Self-Play

- Generate a sequence of training examples through self-play

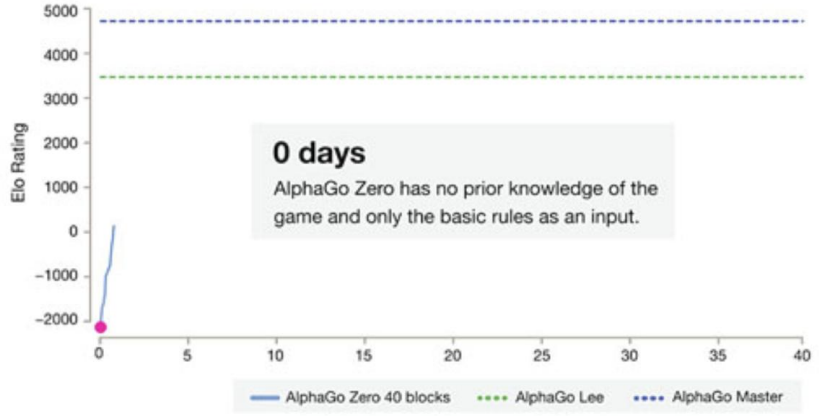



```
1  for it in range (1, max_iterations): # do a curric. of self-play tourn.
2      for game in range(1, max_games): # play a tourn. of games; then train
3          trim(triples) # if buffer full: replace old entries
4          while not game_over(): # generate the moves of one game
5              game_pairs += mcts(eval(net)) # move is a (state,action) pair
6          triples += add(games_pairs, game_outcome()) # add win/lose to buf
7      net = train(net, triples) # retrain with (state,action,outc) triples
```

# AlphaGo Zero Overview

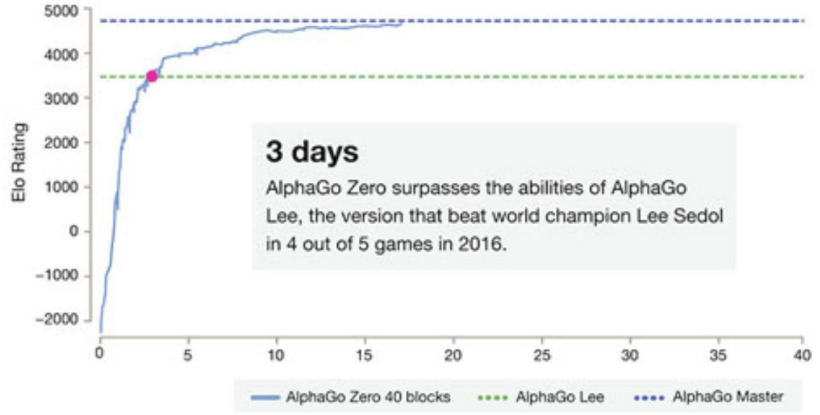
- Zero-knowledge
- One net (double-headed)
- One learning method: Self-Play
- Tabula Rasa:
  - Only the rules and input/output layers,
  - zero heuristics, zero grandmaster games
- Curriculum learning

# AlphaGo Zero Performance




**0 days**

AlphaGo Zero has no prior knowledge of the game and only the basic rules as an input.

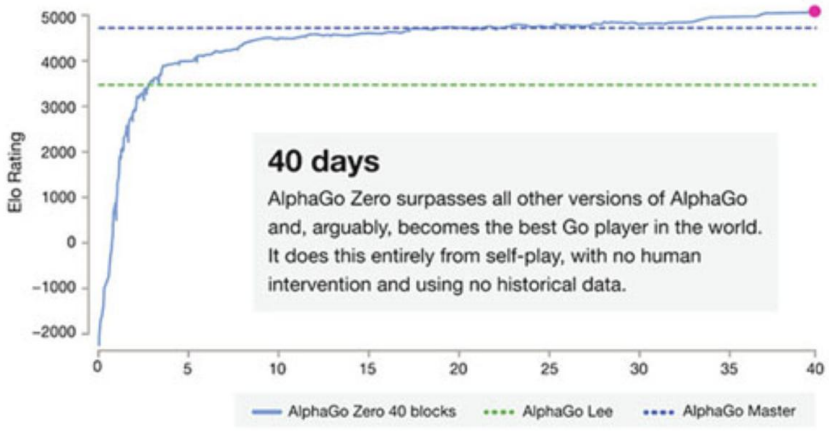



**3 days**

AlphaGo Zero surpasses the abilities of AlphaGo Lee, the version that beat world champion Lee Sedol in 4 out of 5 games in 2016.


**21 days**

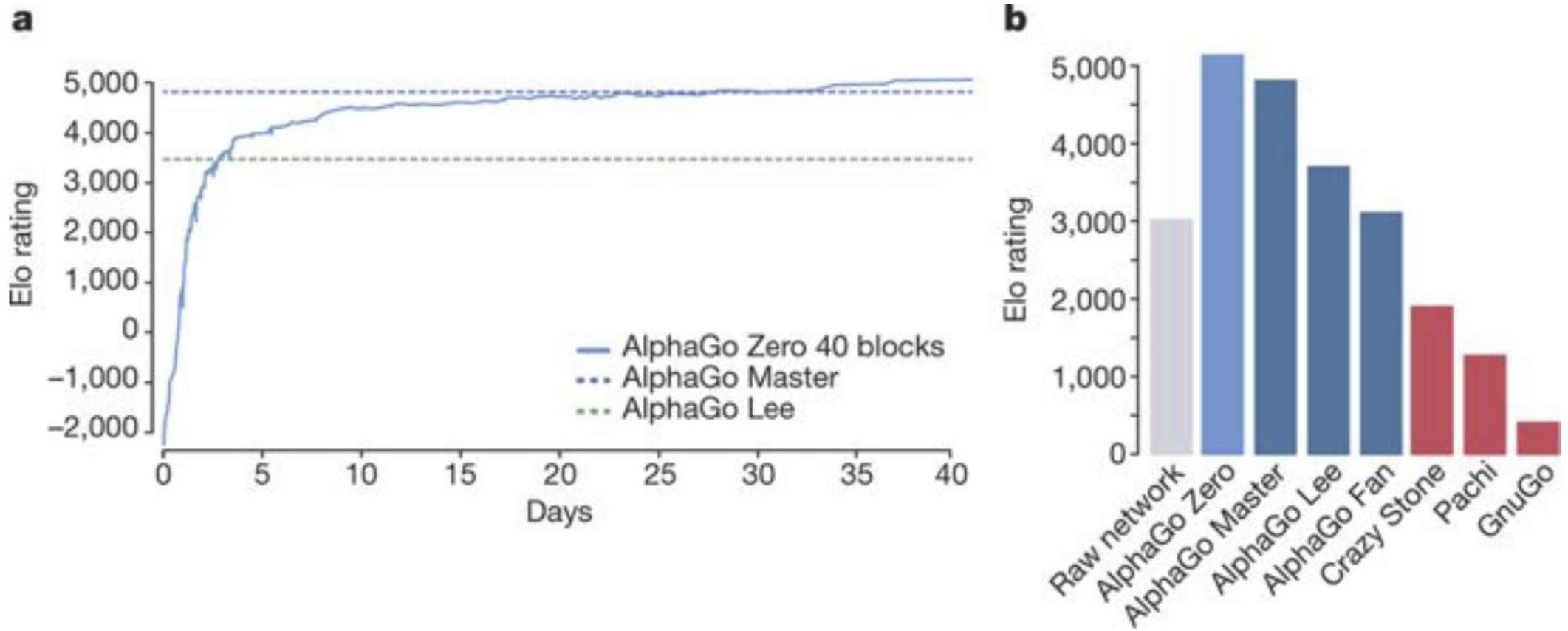
AlphaGo Zero reaches the level of AlphaGo Master, the version that defeated 60 top professionals online and world champion Ke Jie in 3 out of 3 games in 2017.



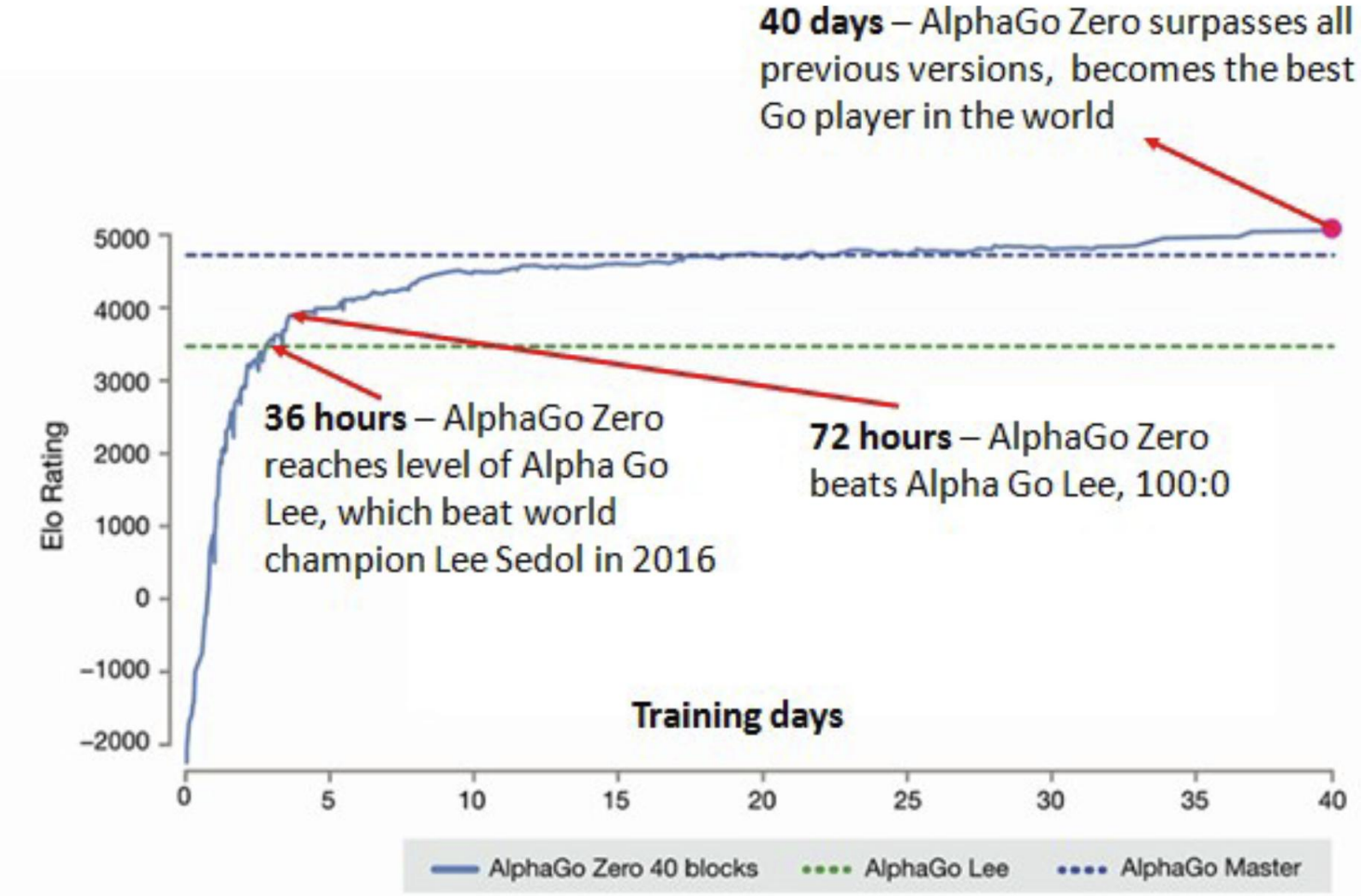

**40 days**

AlphaGo Zero surpasses all other versions of AlphaGo and, arguably, becomes the best Go player in the world. It does this entirely from self-play, with no human intervention and using no historical data.

# AlphaGo Zero Performance

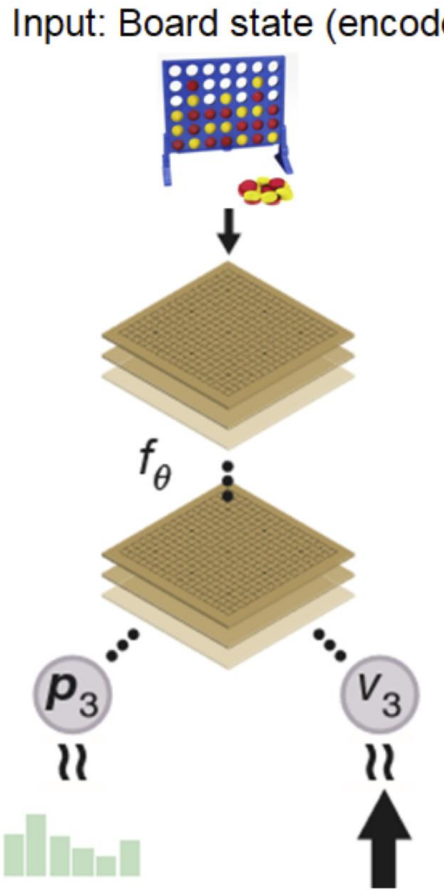


# AlphaGo Zero Performance



# AlphaGo Zero Structure

- 1 Net:
  - ResNet with policy head and value head
  - Combined loss-function
- 1 Learning: RL Self-Play
- Tabula Rasa




**One convolution block**

128 filters (3X3 kernel, stride 1) + Batch norm + relu

**19 Res blocks**

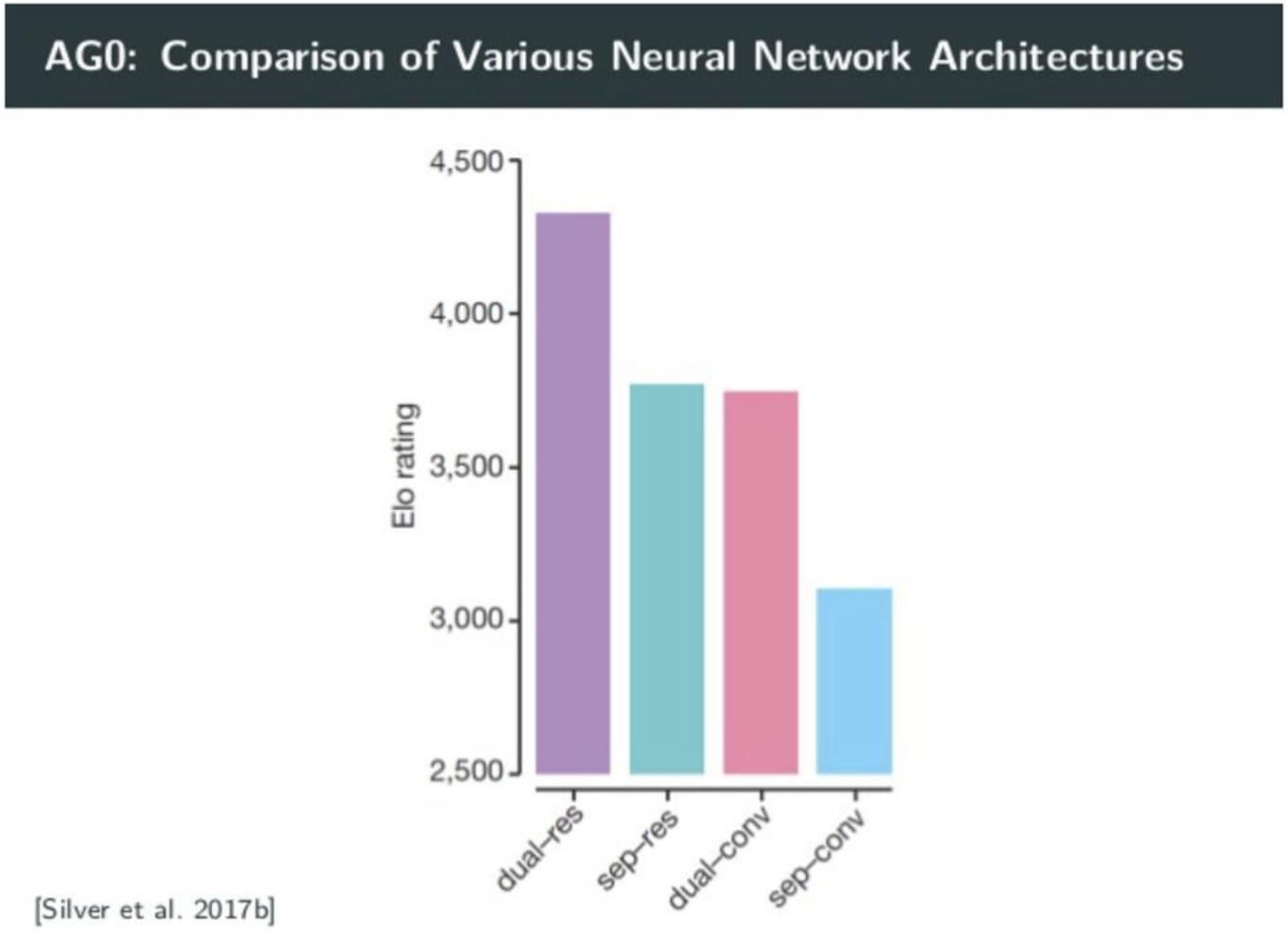
Each block has 128 filters (3X3 kernel, stride 1) + Batch norm + relu + 128 filters (3X3 kernel, stride 1) + Batch norm + residual connection + relu

**One Output Block**

Policy: convo of 32 filters (1X1 kernel, stride 1) + batch norm + relu + linear + softmax

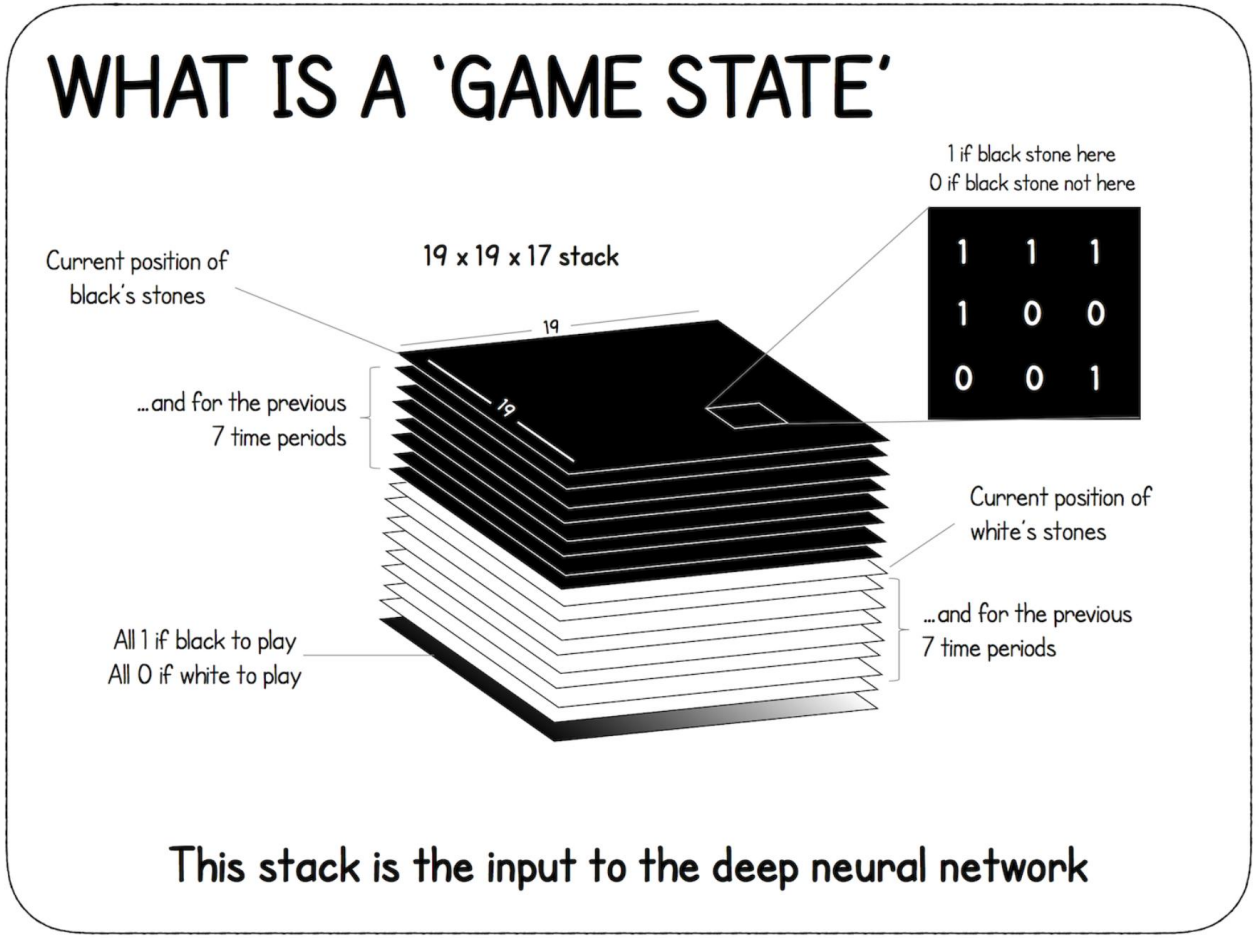
Value: convo of 3 filters (1X1 kernel, stride 1) + batch norm + relu + linear + relu + linear + tanh

# AlphaGo Zero Structure




[Silver et al. 2017b]

# AlphaGo Zero Game State



# ALPHAGO ZERO CHEAT SHEET

The training pipeline for AlphaGo Zero consists of three stages, executed in parallel

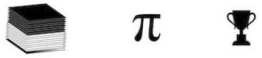
## SELF PLAY

Create a 'training set'

**The best current player plays 25,000 games against itself**

See MCTS section to understand how AlphaGo Zero selects each move

**At each move, the following information is stored**



**The game state** (see 'What is a Game State section')

**The search probabilities** (from the MCTS)

**The winner** (+1 if this player won, -1 if this player lost - added once the game has finished)

## RETRAIN NETWORK

Optimise the network weights

A TRAINING LOOP

**Sample a mini-batch of 2048 positions from the last 500,000 games**

**Retrain the current neural network on these positions**

- The game states are the input (see 'Deep Neural Network Architecture')

**Loss Function**

Compares predictions from the neural network with the search probabilities and actual winner

PREDICTIONS: p, v — Cross-entropy + Mean-squared error + Regularisation — ACTUAL: π, trophy

**After every 1,000 training loops, evaluate the network**

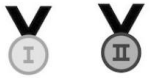
## EVALUATE NETWORK

Test to see if the new network is stronger

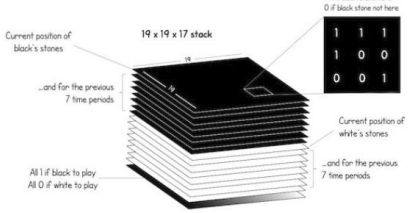
**Play 400 games between the latest neural network and the current best neural network**

Both players use MCTS to select their moves, with their respective neural networks to evaluate leaf nodes

**Latest player must win 55% of games to be declared the new best player**




## WHAT IS A 'GAME STATE'



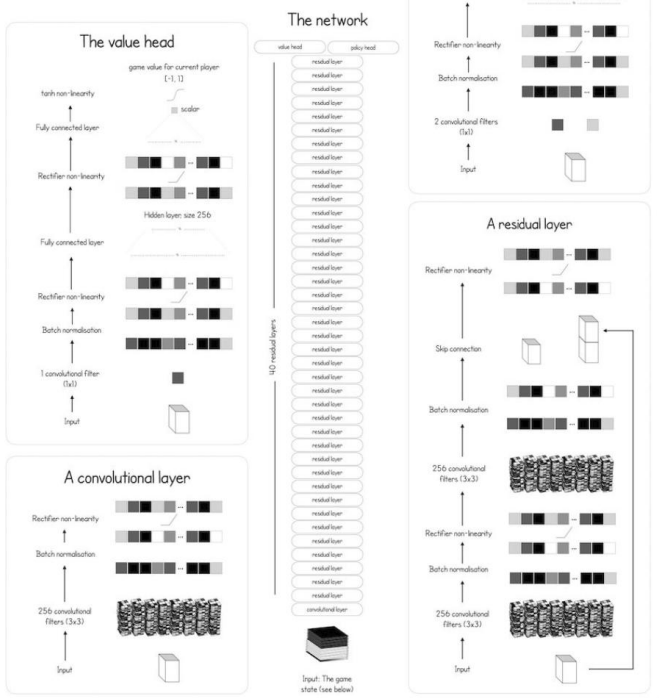

This stack is the input to the deep neural network

## THE DEEP NEURAL NETWORK ARCHITECTURE

How AlphaGo Zero assesses new positions

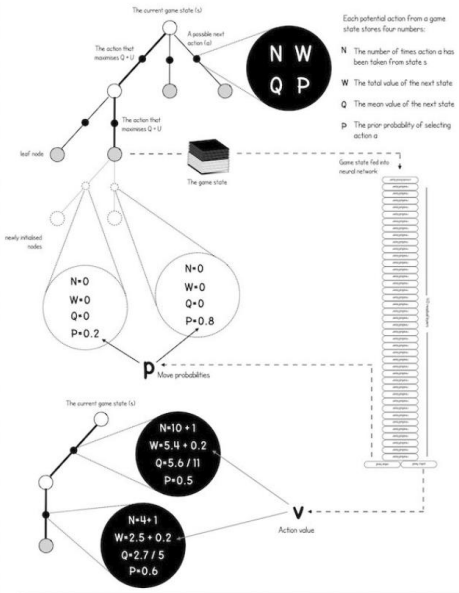
The network learns 'tabula rasa' (from a blank slate)

At no point is the network trained using human knowledge or expert moves




## MONTE CARLO TREE SEARCH (MCTS)

How AlphaGo Zero chooses its next move




Each potential action from a game state stores four numbers:

- **N** The number of times action a has been taken from state s
- **W** The total value of the next state
- **Q** The mean value of the next state
- **P** The prior probability of selecting action a

### First, run the following simulation 1,600 times...

Start at the root node of the tree (the current game state)

**1. Choose the action that maximises...**

$$Q + U$$

- Q: The mean value of the next state
- U: A function of P and N that increases if an action hasn't been explored much, relative to the other actions, or if the prior probability of the action is high

Early on in the simulation, U dominates (more exploration), but later, Q is more important (less exploration)

**2. Continue until a leaf node is reached**

The game state of the leaf node is passed into the neural network, which outputs predictions about two things:

- **p** Move probabilities
- **v** Value of the state (for the current player)

The move probabilities p are attached to the new feasible actions from the leaf node

**3. Backup previous edges**

Each edge that was traversed to get to the leaf node is updated as follows:

$N \rightarrow N + 1$

$W \rightarrow W + v$

$Q = W / N$

### ...then select a move

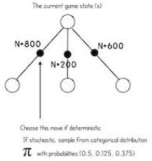
After 1,600 simulations, the move can either be chosen:

**Deterministically** (for competitive play)
Choose the action from the current state with greatest N

**Stochastically** (for exploratory play)
Choose the action from the current state from the distribution

$$\pi \sim N^{1/\tau}$$

where $\tau$ is a temperature parameter controlling exploration




### Other points

- The sub-tree from the chosen move is retained for calculating subsequent moves
- The rest of the tree is discarded

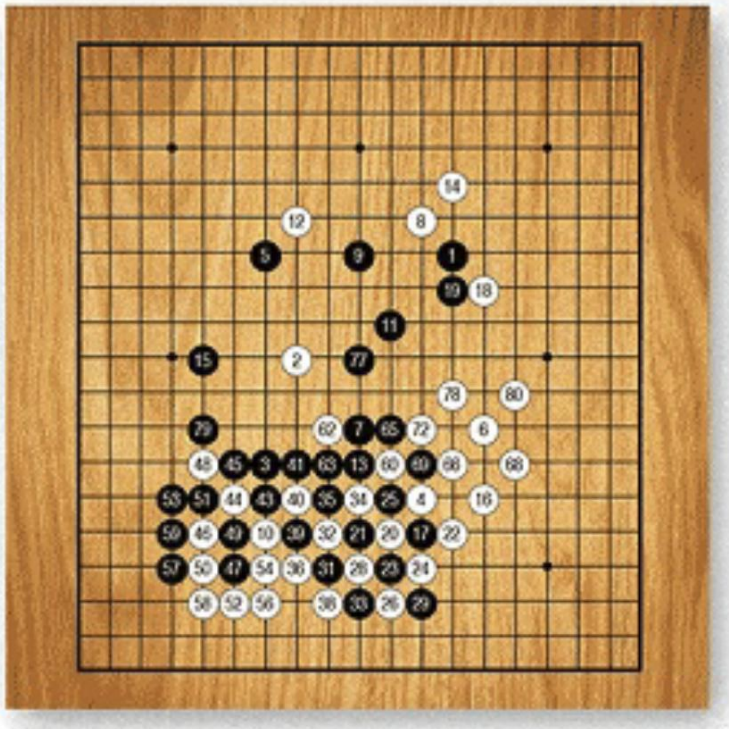
# AlphaGo Zero – Stability

- Stable
  - Extra Exploration
  - De-correlation
- How?
  - MCTS & Noise & Exploration & Replay Buffer & Many games
- AlphaGo Zero's nets are not optimized against themselves, but against MCTS-improved versions of themselves

# Curriculum Learning

- AlphaGo Zero learns better than AlphaGo
- AlphaGo Zero learns faster than AlphaGo. Why?
- Curriculum learning: Start with easy examples
- Many small steps are faster than one large step

# Learning to Play Go

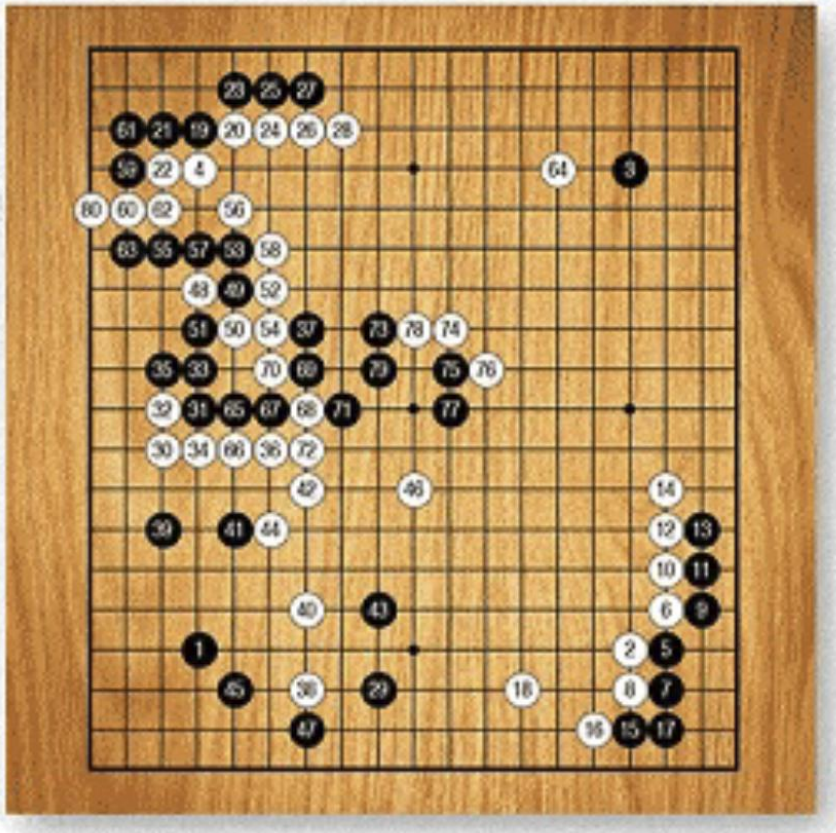


Captured Stones

## 3 hours

AlphaGo Zero plays like a human beginner, forgoing long term strategy to focus on greedily capturing as many stones as possible.

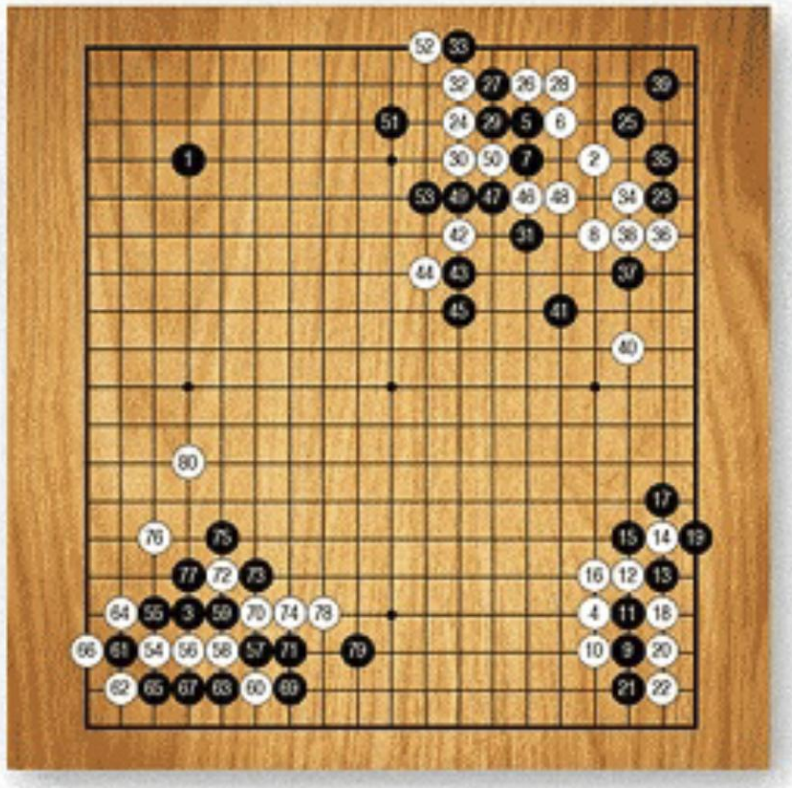
# Learning to Play Go



**19 hours**

AlphaGo Zero has learnt the fundamentals of more advanced Go strategies such as life-and-death, influence and territory.

# Learning to Play Go



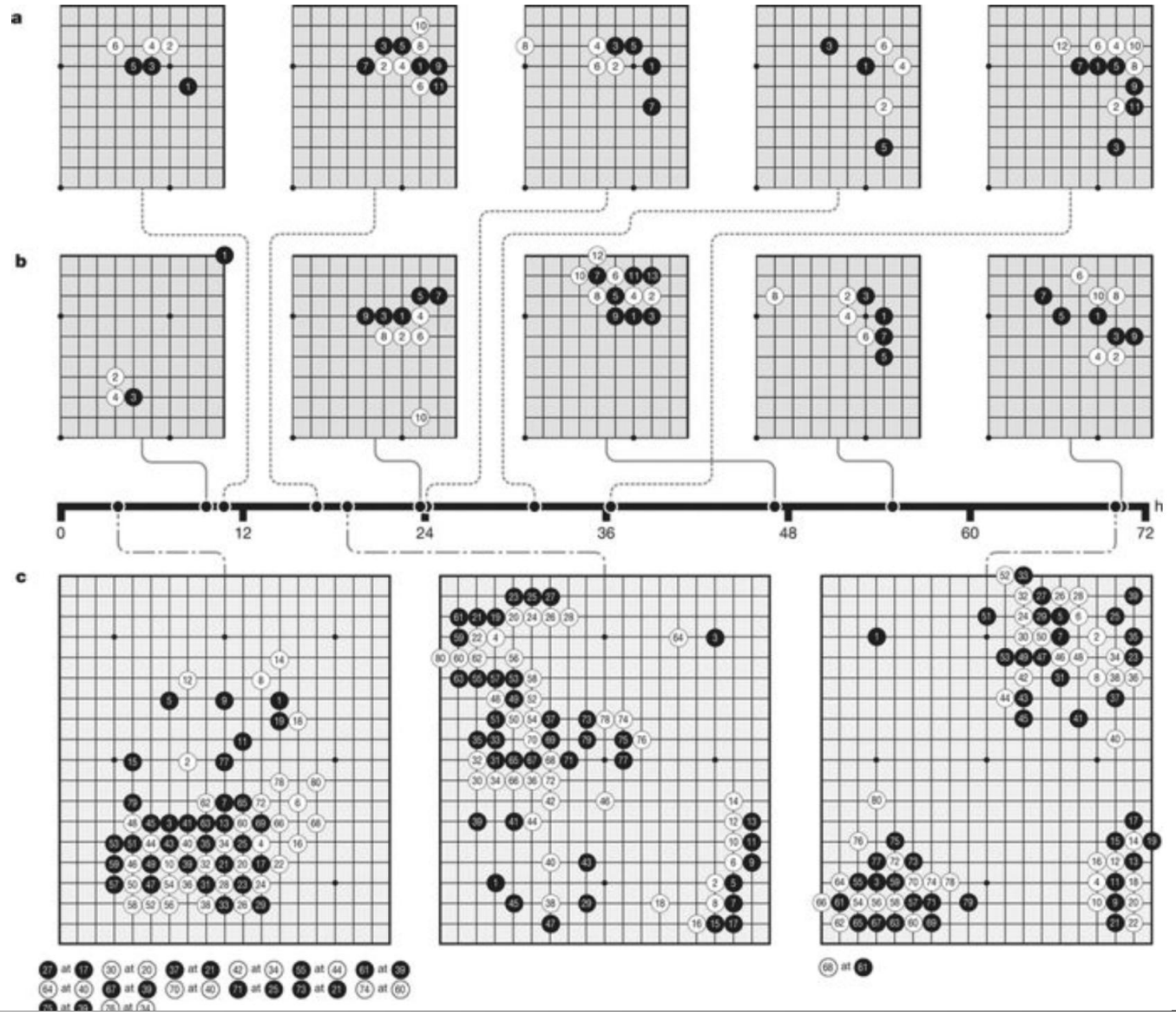
(68) at (61)

Captured Stones

## 70 hours

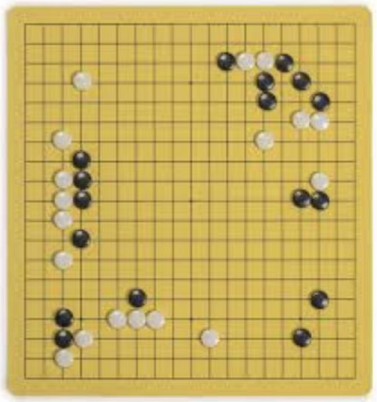
AlphaGo Zero plays at super-human level.
The game is disciplined and involves multiple challenges across the board.

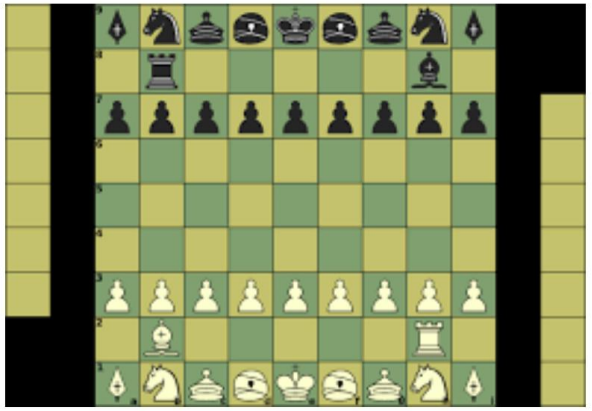
# Curriculum



# AlphaZero

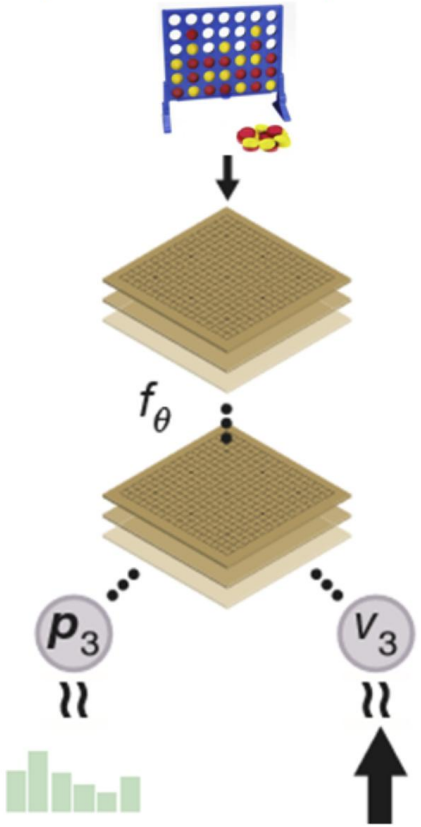
- Same net, same search, same tabula rasa self-play
- Different Input/Output layers
- Go, Chess, Shogi





# AlphaZero Structure

Input: Board state (encoded)




## One convolution block

128 filters (3X3 kernel, stride 1) + Batch norm + relu

## 19 Res blocks

Each block has 128 filters (3X3 kernel, stride 1) + Batch norm + relu + 128 filters (3X3 kernel, stride 1) + Batch norm + residual connection + relu

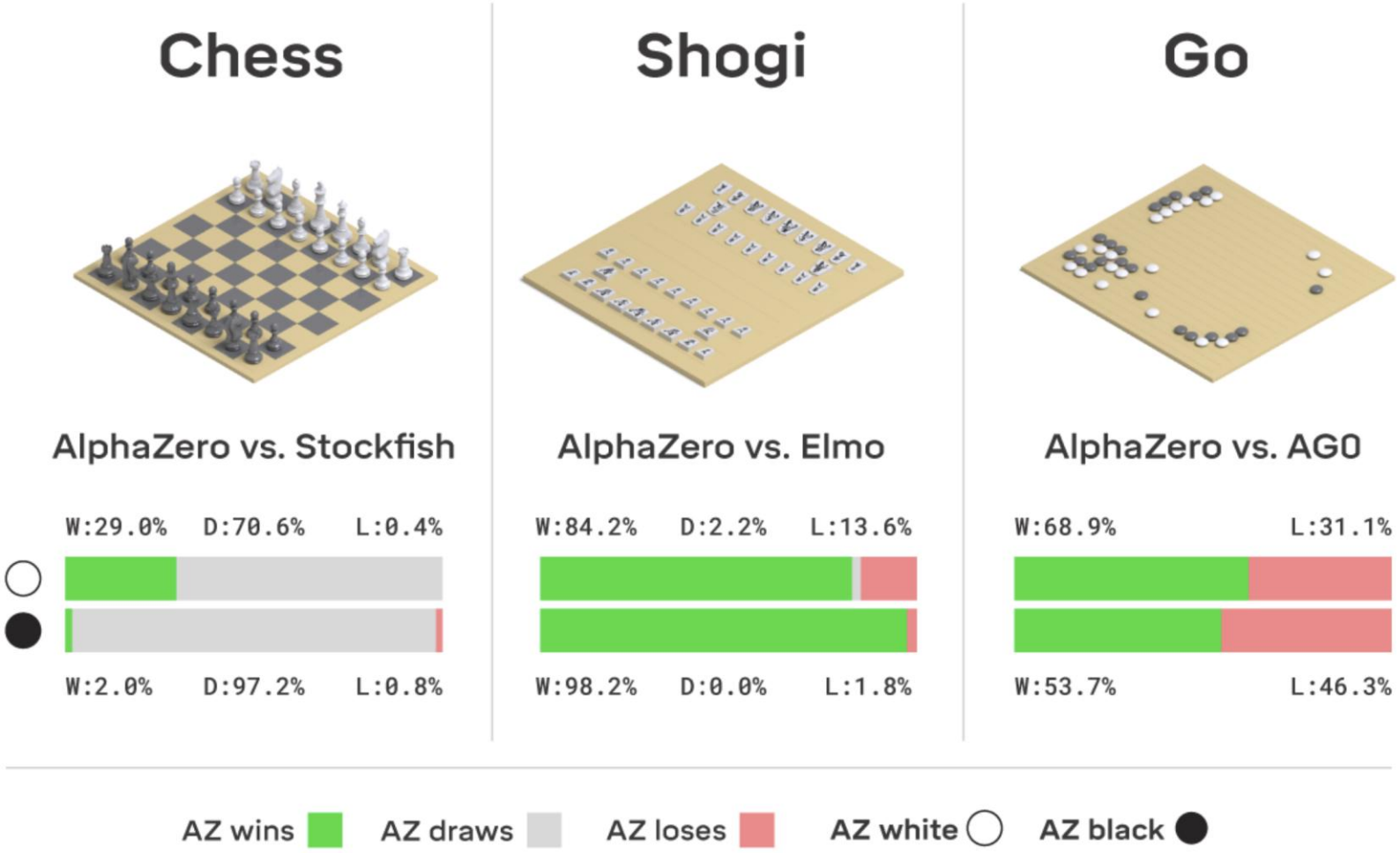
## One Output Block

Policy: convo of 32 filters (1X1 kernel, stride 1) + batch norm + relu + linear + softmax

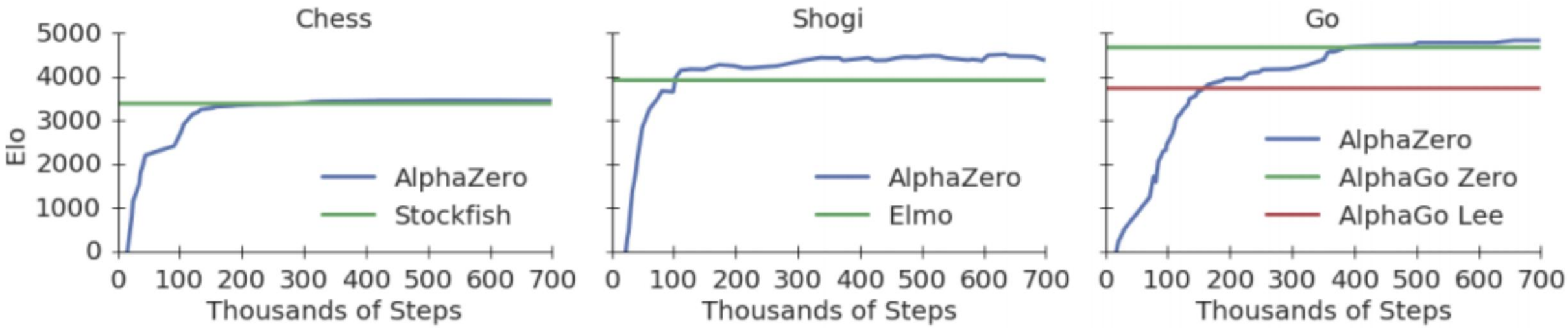
Value: convo of 3 filters (1X1 kernel, stride 1) + batch norm + relu + linear + relu + linear + tanh

Outputs: $P$ – Policy, $v$ - value

# AlphaZero Performance



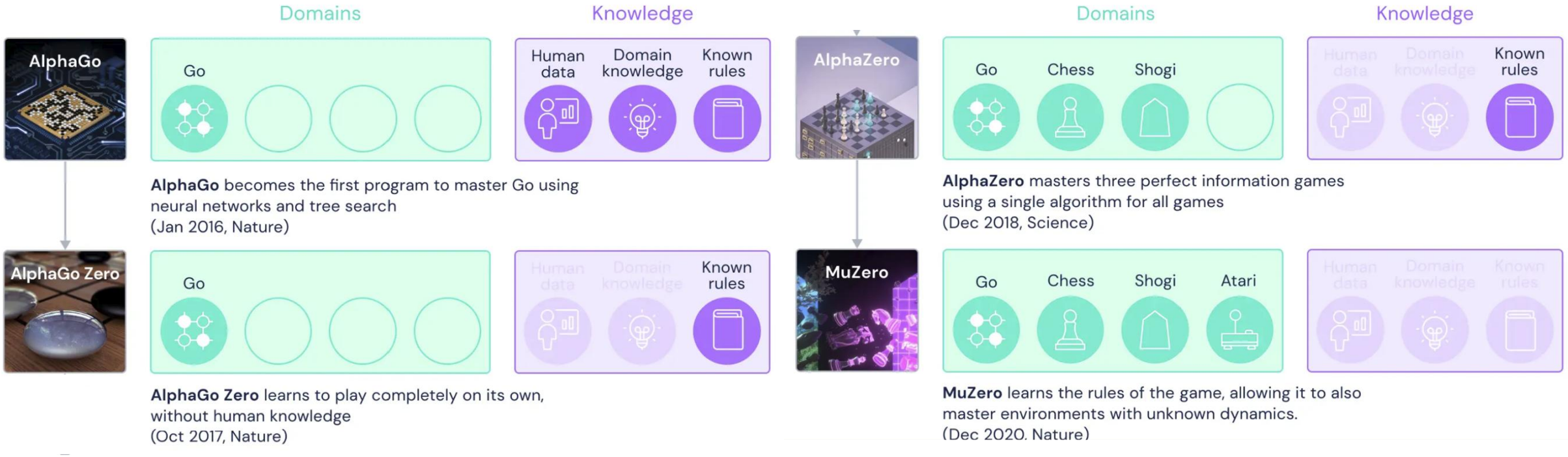
# AlphaZero Performance



# AlphaZero Conclusions

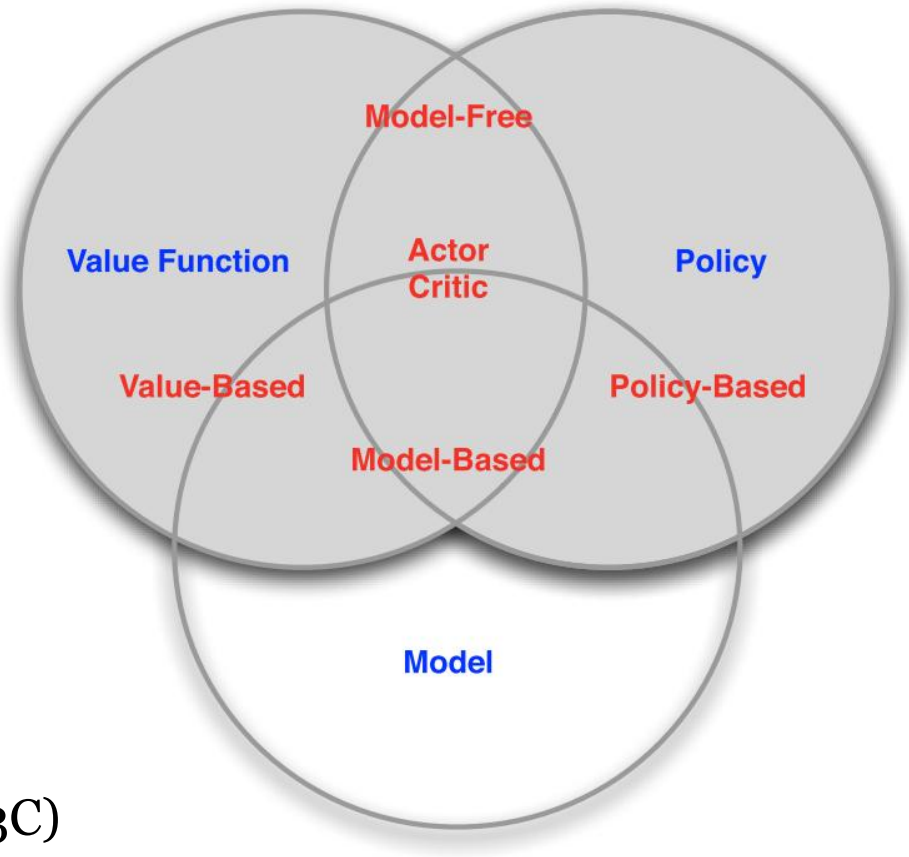
- First time learning, neural nets, and MCTS work in Chess
- Decades of heuristic planning research are surpassed
- Three Games share a general essence, since same architecture works (except I/O)
- Not same net. Net trained for Chess does not work for Shogi
- First architecture achieving very high performance in three games

# MuZero: Learning Dynamics for Planning (2020)

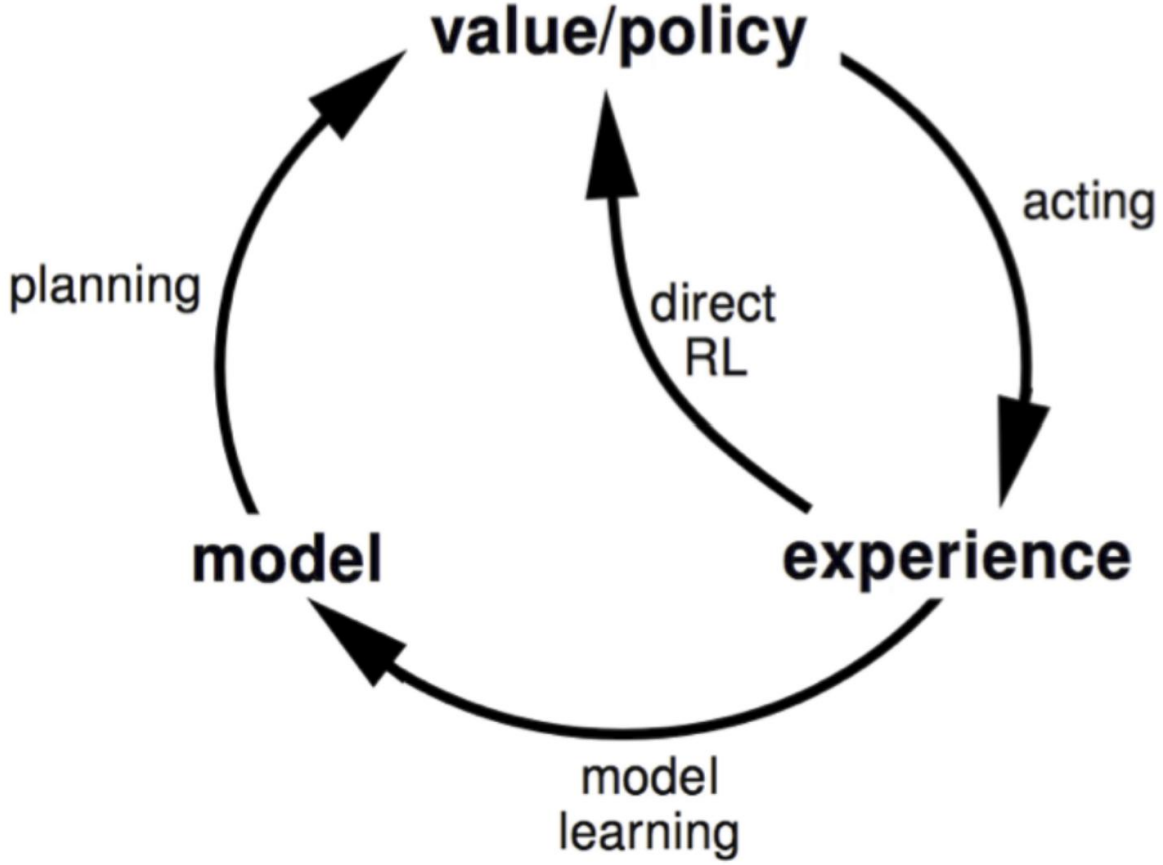


# Reinforcement Learning Approaches

- Value-Based:
  - Learn value function
  - Implicit policy (e.g. greedy selection)
  - Example: Deep Q Networks (DQN)
- Policy-Based:
  - No value function
  - Learn explicit (stochastic) policy
  - Example: Stochastic Policy Gradients
- Model-Based:
  - Learn transition model
  - Implicit policy
  - Example: Dreamer
- Actor-Critic:
  - Learn value function
  - Learn policy using value function
  - Example: Asynchronous Advantage Actor Critic (A3C)



# Model-Based vs Model-Free RL




Learn *policy* direct or learn *transition* first and then policy?

# Actor-Critic Deep RL

- *An Actor* that controls **how our agent behaves** (policy-based method).
- *A Critic* that measures **how good the action taken is** (value-based method).
- Two ideas to reduce variance
  - Temporal difference bootstrapping
  - Baseline subtraction

# Bootstrapping

- TD computes values step-wise (low variance)

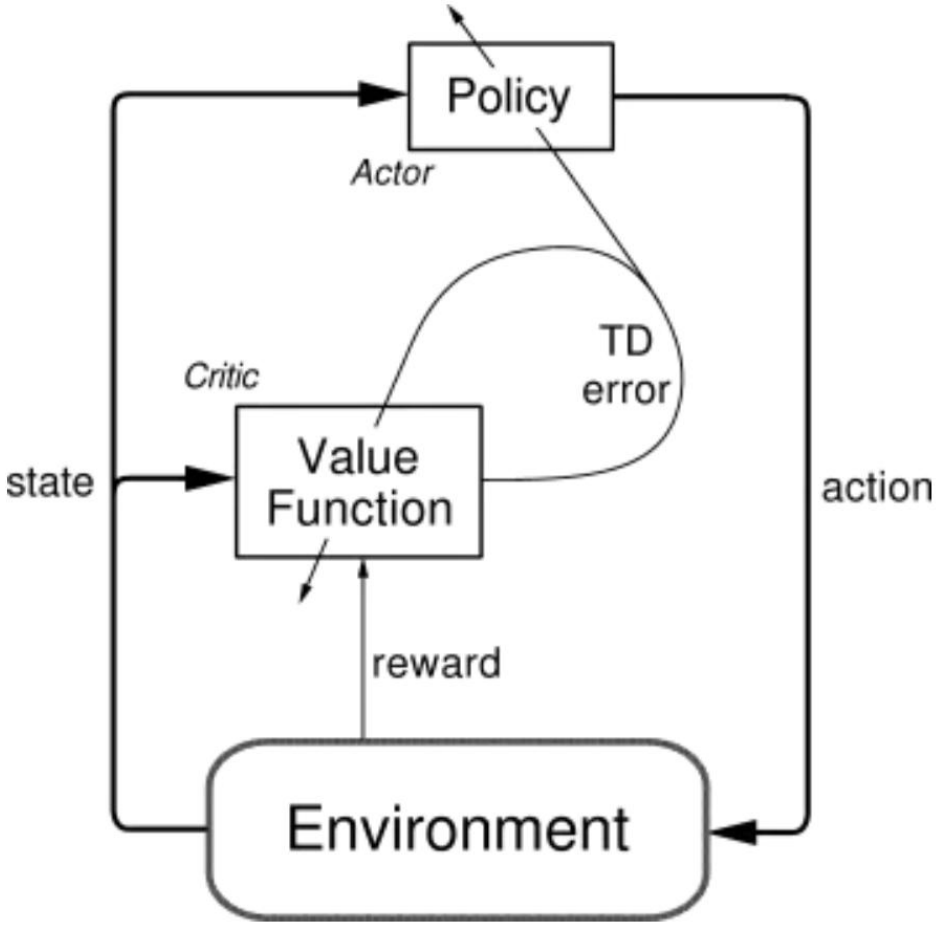
$$V(s) \leftarrow V(s) + \alpha[R' + \gamma V(s') - V(s)]$$

- We can also use n-step values
- TD bootstrapping reduces variance of Monte Carlo at the cost of more bias

# Baseline Subtraction

- Rewards are often skewed, such as, all positive, leading to high variance.
- Centering around zero would reduce variance
- When a baseline function is added to a function, the expectation does not change, and the variance is reduced
- The Value function is such a function for the Q function
  $A(s,a) = Q(s,a) - V(s)$
- *Advantage*, the difference between the actual game outcome and the expected outcome.
- The learning of the actor is based on policy gradient approach. In comparison, critics evaluate the action produced by the actor by computing the value function.

# Actor Critic RL



# Advantage Variants

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau_0 \sim p_\theta(\tau_0)}\left[\sum_{t=0}^{n} \Psi_t \nabla_\theta \log \pi_\theta(a_t|s_t)\right]$$

- Targets:

$$\Psi_t = \hat{Q}_{MC}(s_t, a_t) = \sum_{i=t}^{\infty} \gamma^i \cdot r_i \qquad \text{Monte Carlo target}$$

$$\Psi_t = \hat{Q}_n(s_t, a_t) \quad = \sum_{i=t}^{n-1} \gamma^i \cdot r_i + \gamma^n V_\theta(s_n) \qquad \text{bootstrap } (n\text{-step target})$$

$$\Psi_t = \hat{A}_{MC}(s_t, a_t) = \sum_{i=t}^{\infty} \gamma^i \cdot r_i - V_\theta(s_t) \qquad \text{baseline subtraction}$$

$$\Psi_t = \hat{A}_n(s_t, a_t) \quad = \sum_{i=t}^{n-1} \gamma^i \cdot r_i + \gamma^n V_\theta(s_n) - V_\theta(s_t) \qquad \text{baseline + bootstrap}$$

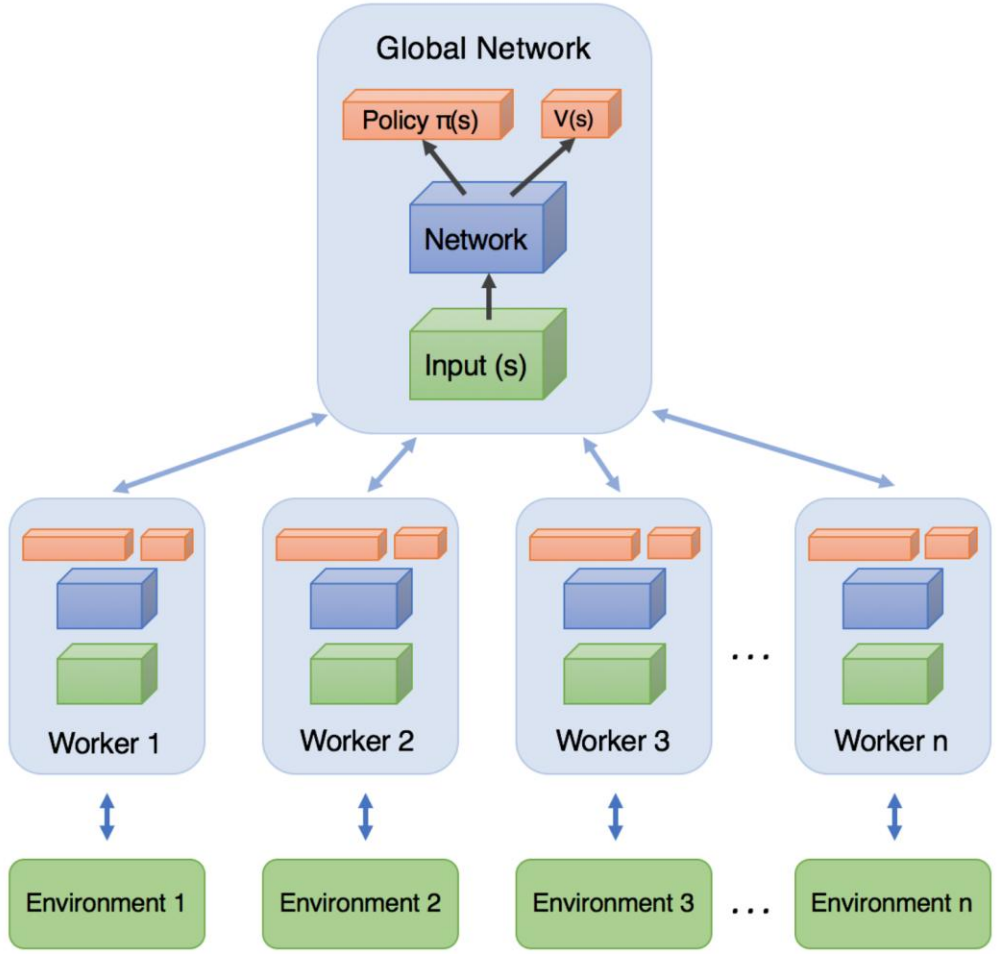
$$\Psi_t = Q_\phi(s_t, a_t) \qquad \text{Q-value approximation}$$

# A3C – Asynchronous Advantage Actor-Critic

- **Asynchronous**: The algorithm is an asynchronous algorithm where multiple worker agents are trained in parallel, each with their environment. This allows the algorithm to train faster as more workers are training in parallel and attain a more diverse training experience as each worker's experience is independent.
- **Advantage**: Advantage is a metric to judge how good its actions were and how they turned out. This allows the algorithm to focus on where the network's predictions were lacking. Intuitively, this will enable it to measure the advantage of taking action, following the policy $\pi$ at the given timestep.
- **Actor-Critic**: The Actor-Critic aspect of the algorithm uses an architecture that shares layers between the policy and value function.

# A3C – Asynchronous Advantage Actor-Critic

1. Fetch the global network parameters
2. Interact with the environment by following the local policy for *n* number of steps
3. Calculate value and policy loss
4. Get gradients from losses
5. Update the global network
6. Repeat




https://pylessons.com/A3C-reinforcement-learning

# Trust Region

- Vanilla REINFORCE is high variance (AC helps)
- Parameters $\theta$ are pushed wildly
- Normally, we would mitigate variance by reducing step size of function input ($\theta$)
- Trust Region is an approach using function output ($\pi$) to mitigate step size, using KL divergence (TRPO), or simple clipping (PPO)

# Proximal Policy Optimization (PPO)

- The idea is to improve the training stability of the policy by limiting the change at each training epoch: **we want to avoid having too large policy updates.**
- For two reasons:
  - We know empirically that smaller policy updates during training are **more likely to converge to an optimal solution.**
  - A too big step in a policy update can result in falling "off the cliff" (getting a bad policy) **and having a long time or even no possibility to recover.**
- **Therefore, PPO updates the policy conservatively**.
  - Measure how much the current policy changed compared to the former one using a ratio calculation between the current and former policy.
  - Clip this ratio in a range $[1-\epsilon,1+\epsilon][1-\epsilon,1+\epsilon]$, thus **removing the incentive for the current policy to go too far from the old one (hence the proximal policy term).**

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[ \min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t) \right]$$

https://huggingface.co/blog/deep-rl-ppo

# Proximal Policy Optimization (PPO)

| | $p_t(\theta) > 0$ | $A_t$ | Return Value of *min* | Objective is Clipped | Sign of Objective | Gradient |
|---|---|---|---|---|---|---|
| **1** | $p_t(\theta) \in [1-\epsilon, 1+\epsilon]$ | $+$ | $p_t(\theta)A_t$ | no | $+$ | ✓ |
| **2** | $p_t(\theta) \in [1-\epsilon, 1+\epsilon]$ | $-$ | $p_t(\theta)A_t$ | no | $-$ | ✓ |
| **3** | $p_t(\theta) < 1-\epsilon$ | $+$ | $p_t(\theta)A_t$ | no | $+$ | ✓ |
| **4** | $p_t(\theta) < 1-\epsilon$ | $-$ | $(1-\epsilon)A_t$ | yes | $-$ | **0** |
| **5** | $p_t(\theta) > 1+\epsilon$ | $+$ | $(1+\epsilon)A_t$ | yes | $+$ | **0** |
| **6** | $p_t(\theta) > 1+\epsilon$ | $-$ | $p_t(\theta)A_t$ | no | $-$ | ✓ |

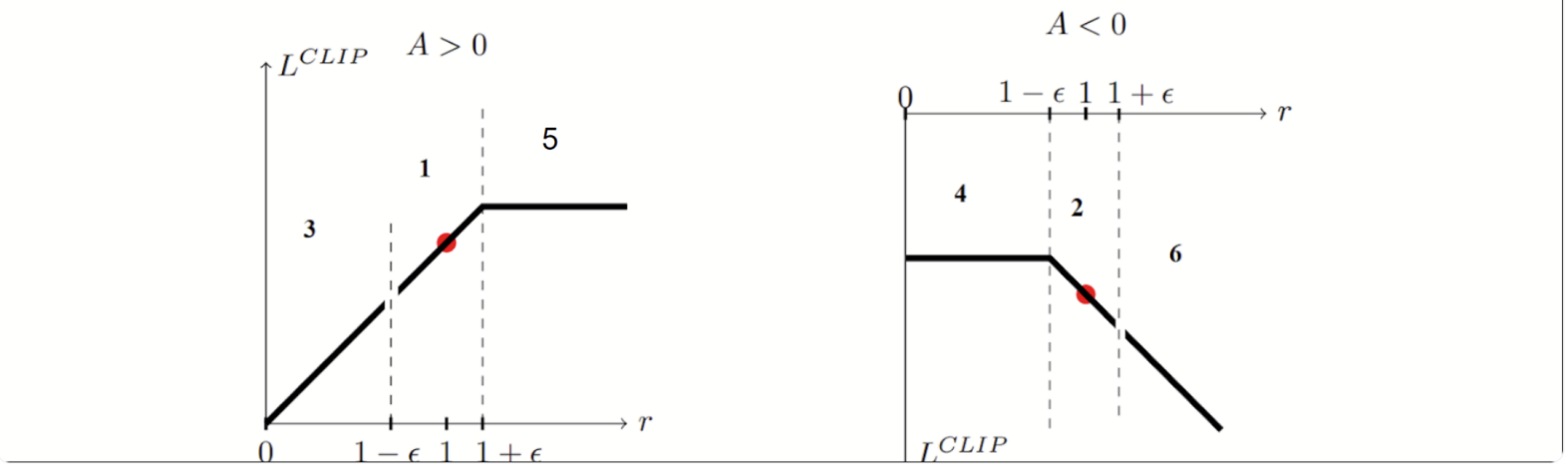



Table from "Towards Delivering a Coherent Self-Contained Explanation of Proximal Policy Optimization" by Daniel Bick

# Proximal Policy Optimization (PPO)

- Final PPO Actor-Critic Objective Function

$$L_t^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}_t\left[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t)\right]$$

Add an entropy bonus **to ensure sufficient exploitation.**

**c1 and c2 are coefficients.**

**Squared-error value loss:** $(V_\theta(s_t) - V_t^{targ})^2$

# Trust Region

- PPO just clips L to a small range

$$[1-\epsilon, 1+\epsilon] \cdot A_t$$

- TRPO compares old and new policy (output)

$$L(\theta) = \mathbb{E}_t \left[ \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} \cdot A_t \right]$$

- Limiting KL divergence (measure of distance of distributions)

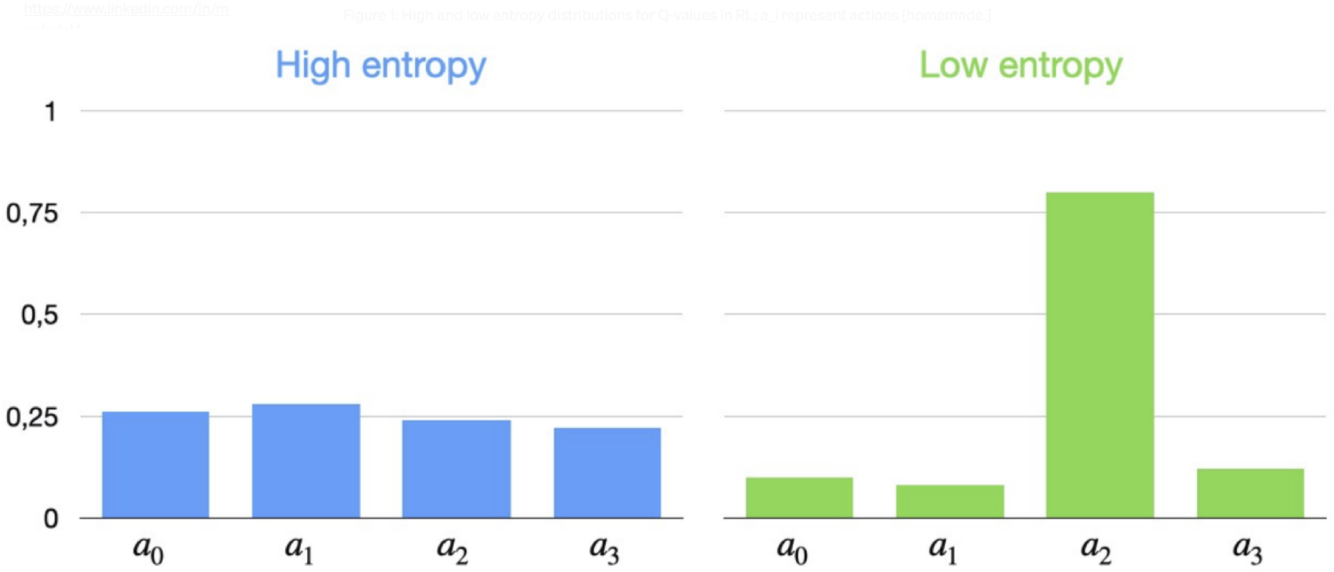
$$\mathbb{E}_t \left[ \text{KL}(\pi_{\theta_{\text{old}}}(\cdot|s_t), \pi_\theta(\cdot|s_t)) \right] \leq \delta$$

- A small divergence allows larger step size, a large divergence contracts

# Entropy / Exploration

- Too little exploration leads to local optima.
- Entropy is randomness. High entropy policies favor exploration
- Soft Actor Critic adds entropy H to the loss function

$$\theta_{t+1} = \theta_t + R \cdot \nabla_\theta \log \pi_\theta(a_t|s_t) + \eta \nabla_\theta H[\pi_\theta(a|s)]$$

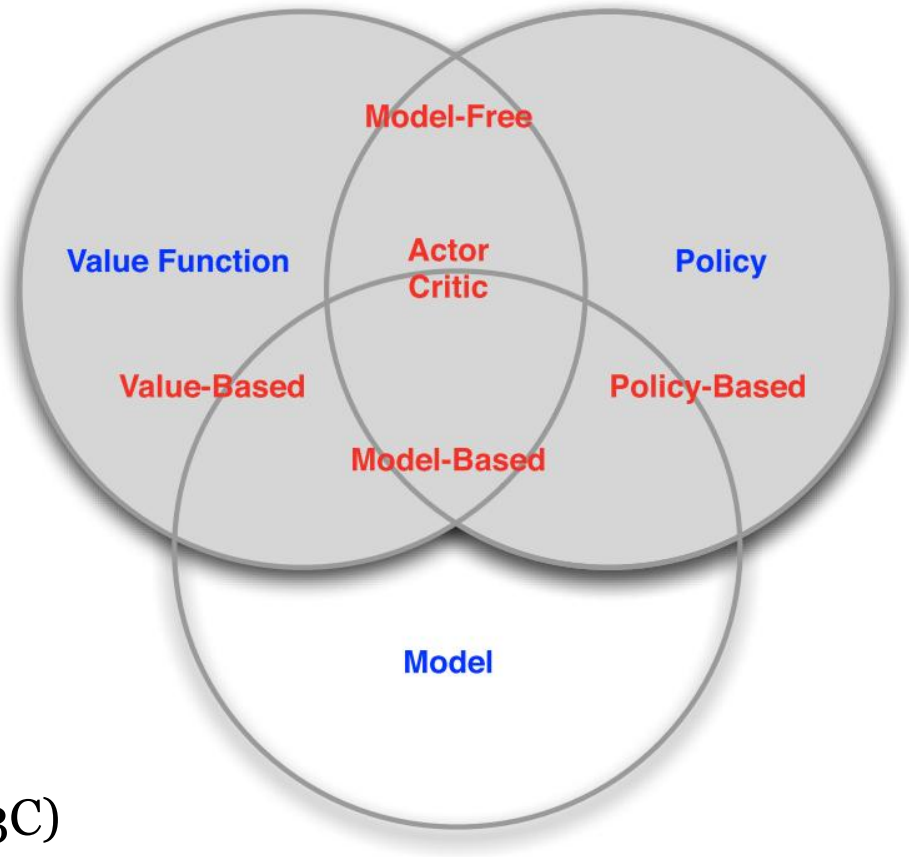


# Actor Critic Policy Methods

- (vanilla) REINFORCE - directly improving continuous policy, but high variance
- Actor Critic
  - TD Bootstrap
  - Advantage: A3C
- Trust Region: TRPO, PPO
- Entropy: SAC
- DQN-based: DDPG

# Reinforcement Learning Approaches

- Value-Based:
  - Learn value function
  - Implicit policy (e.g. greedy selection)
  - Example: Deep Q Networks (DQN)
- Policy-Based:
  - No value function
  - Learn explicit (stochastic) policy
  - Example: Stochastic Policy Gradients
- Model-Based:
  - Learn transition model
  - Implicit policy
  - Example: Dreamer
- Actor-Critic:
  - Learn value function
  - Learn policy using value function
  - Example: Asynchronous Advantage Actor Critic (A3C)



# RL LE3 VT2026:

## Model-based vs Model-free RL
## Two-Player Games with Perfect Transitions
## Actor-Critic RL

www.ida.liu.se/~TDDE78

LINKÖPING UNIVERSITY