



# Reinforcement Learning

## Applications and Case Studies

**U Kang**  
**Seoul National University**



# In This Lecture

- Applications of RL
- Solving real-world problems



# Outline

- ➔  **TD-Gammon**
- Human-level Video Game Play
- Mastering the Game of Go
- Conclusion

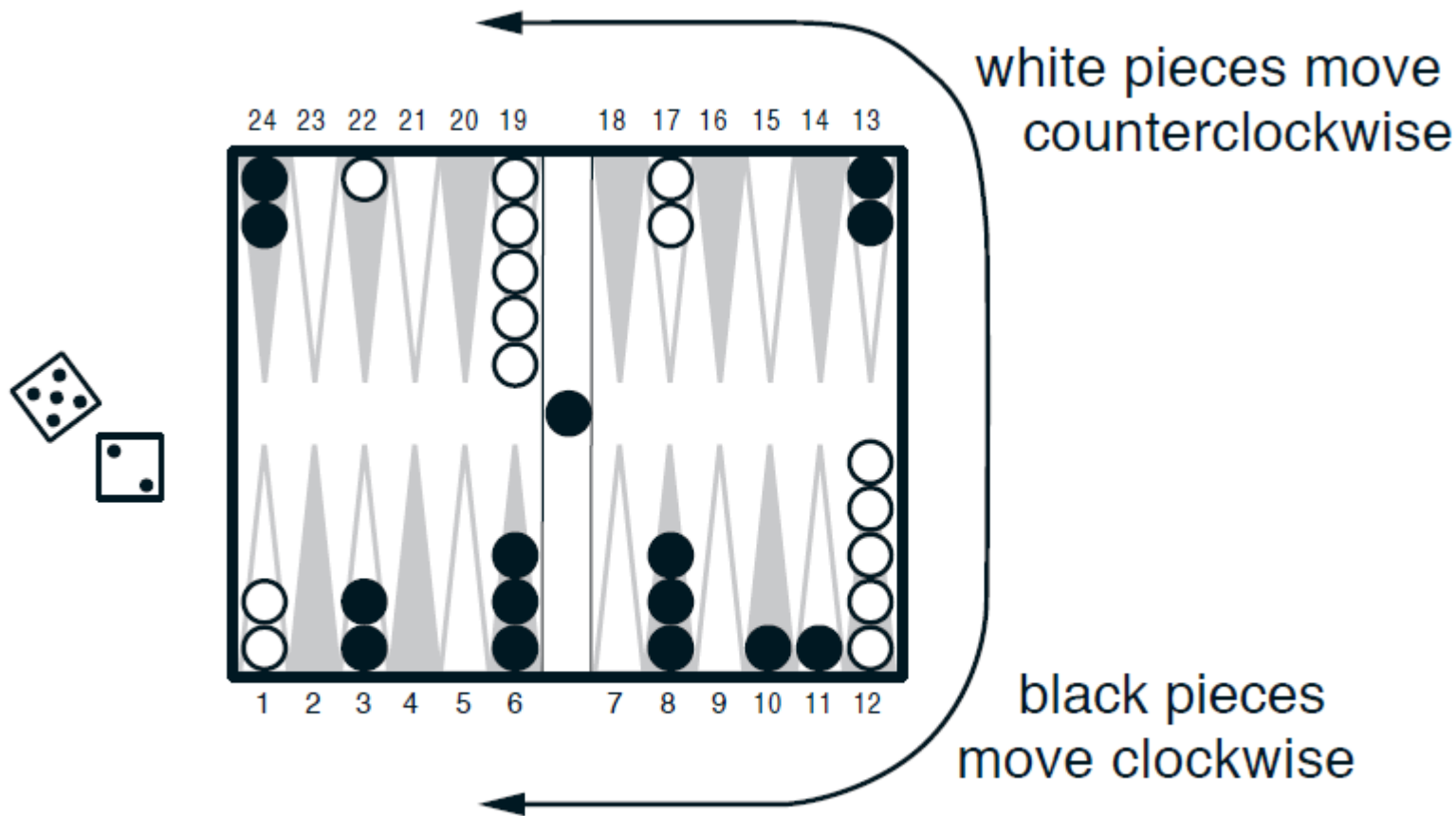


# TD-Gammon

- TD-Gammon: one of the most impressive applications of RL, by Gerald Tesauro
- TD-Gammon required little backgammon knowledge, yet learned to play extremely well, near the level of the world's strongest grandmasters
- The learning algorithm was a straightforward combination of the TD( $\lambda$ ) algorithm and nonlinear FA using a multilayer artificial neural network (ANN) trained by backpropagating TD errors



# Backgammon



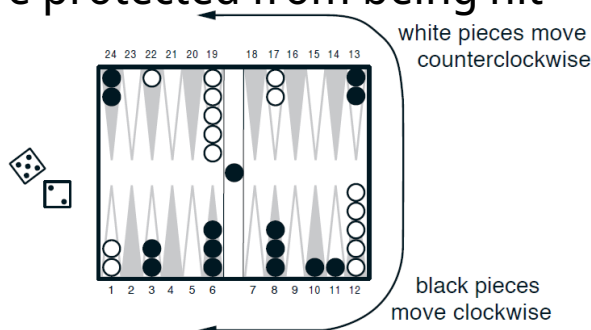
A backgammon position

Sutton and Barto,  
Reinforcement  
Learning, 2018



# Backgammon

- 15 white and 15 black pieces on a board of 24 locations, called points
- If a white player just rolled the dice and obtained a 5 and a 2, he can move one of his pieces 5 steps and one (possibly the same piece) 2 steps. E.g., he could move two pieces from the 12 point, one to the 17 point, and one to the 14 point
- White's objective is to advance all of his pieces into the last quadrant (points 19–24) and then off the board
- Hitting: if it were black's move, he could use the dice roll of 2 to move a piece from the 24 point to the 22 point, "hitting" the white piece there. Pieces that have been hit are placed on the "bar" in the middle of the board, from whence they reenter the race from the start
- However, if there are  $\geq 2$  pieces on a point, then the opponent cannot move to that point; the pieces are protected from being hit



A backgammon position



# TD-Gammon

- With 30 pieces and 24 possible locations (26, counting the bar and off-the-board) it should be clear that the number of possible backgammon positions is enormous
- The number of moves possible from each position is also large: for a typical dice roll there might be 20 different ways of playing
- The game tree has an effective branching factor of about 400 (due to the opponent)
- This is far too large to use the conventional heuristic search methods that have proved so effective in games like chess and checkers



# TD-Gammon

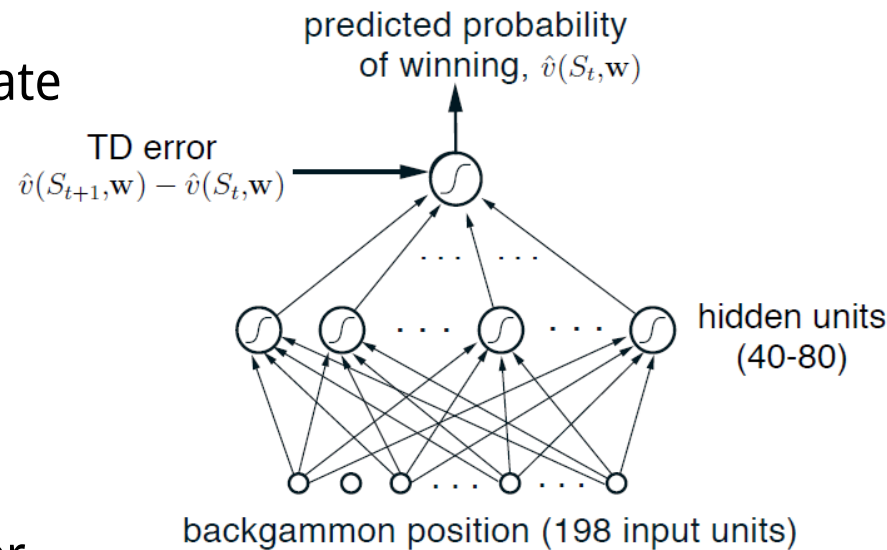
- On the other hand, the game is a good match to the capabilities of TD learning methods
- Although the game is highly stochastic, a complete description of the game's state is available at all times
- The game evolves over a sequence of moves and positions until finally ending in a win for one player or the other, ending the game
- The outcome can be interpreted as a final reward to be predicted
- The number of states is so large that a lookup table cannot be used, and the opponent is a source of uncertainty





# TD-Gammon

- TD-Gammon used a nonlinear form of TD( $\lambda$ )
- The estimated value  $\hat{v}(s, w)$  of any state (board position)  $s$  was meant to estimate the probability of winning starting from state  $s$
- Rewards: 0 for all time steps except those on which the game is won (1)
- Value function: a standard multilayer ANN with a layer of input units, a layer of hidden units, and a final output unit
- The input to the network was a representation of a backgammon position, and the output was an estimate of the value of that position



$$h(j) = \sigma \left( \sum_i w_{ij} x_i \right) = \frac{1}{1 + e^{-\sum_i w_{ij} x_i}}$$



# TD-Gammon

- Why 198 input units to the network?
  - For each point, four units indicated the number of white pieces on the point. If there were no white pieces, then all four units took on the value zero
  - If there was one piece, then the first unit took on the value 1. This encoded the elementary concept of a “blot,” i.e., a piece that can be hit by the opponent
  - If there were two or more pieces, then the second unit was set to 1. This encoded the basic concept of a “made point” on which the opponent cannot land
  - If there were exactly three pieces on the point, then the third unit was set to 1. This encoded the basic concept of a “single spare,” i.e., an extra piece in addition to the two pieces that made the point
  - Finally, if there are  $n > 3$  pieces, the fourth unit was set to  $(n-3)/2$  which encodes a linear representation of “multiple spares” at the given point



# TD-Gammon

- Why 198 input units to the network?
  - With 4 units for white and 4 for black at each of the 24 points, that made a total of 192 units
  - Two additional units encoded the number of white and black pieces on the bar (each took the value  $n/2$ , where  $n$  is the number of pieces on the bar)
  - Two more units encoded the number of black and white pieces already successfully removed from the board (these took the value  $n/15$ , where  $n$  is the number of pieces already removed)
  - Finally, two units indicated in a binary fashion whether it was white's or black's turn to move
  - Basically, Tesauro tried to represent the position in a straightforward way, while keeping the number of units relatively small
  - He provided one unit for each conceptually distinct possibility that seemed likely to be relevant, and he scaled them to roughly the same range, in this case between 0 and 1



# TD-Gammon

- TD-Gammon used the semi-gradient form of the TD( $\lambda$ ) algorithm, with the gradients computed by the error backpropagation

$$w_{t+1} \doteq w_t + \alpha [R_{t+1} + \gamma \hat{v}(S_{t+1}, w_t) - \hat{v}(S_t, w_t)] z_t$$

- where  $w_t$  is the vector of all modifiable parameters, and  $z_t$  is a vector of eligibility traces, one for each component of  $w_t$ , updated by

$$z_t \doteq \gamma \lambda z_{t-1} + \nabla \hat{v}(S_t, w_t)$$

- with  $z_0 = 0$
- For the backgammon application, in which  $\gamma = 1$  and the reward is always zero except upon winning, the TD error portion of the learning rule is usually just  $\hat{v}(S_{t+1}, w) - \hat{v}(S_t, w)$



# Reminder: TD( $\lambda$ )

## Semi-gradient TD( $\lambda$ ) for estimating $\hat{v} \approx v_\pi$

Input: the policy  $\pi$  to be evaluated

Input: a differentiable function  $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \rightarrow \mathbb{R}$  such that  $\hat{v}(\text{terminal}, \cdot) = 0$

Algorithm parameters: step size  $\alpha > 0$ , trace decay rate  $\lambda \in [0, 1]$

Initialize value-function weights  $\mathbf{w}$  arbitrarily (e.g.,  $\mathbf{w} = \mathbf{0}$ )

Loop for each episode:

  Initialize  $S$

$\mathbf{z} \leftarrow \mathbf{0}$

(a  $d$ -dimensional vector)

  Loop for each step of episode:

    | Choose  $A \sim \pi(\cdot|S)$

    | Take action  $A$ , observe  $R, S'$

    |  $\mathbf{z} \leftarrow \gamma\lambda\mathbf{z} + \nabla\hat{v}(S, \mathbf{w})$

    |  $\delta \leftarrow R + \gamma\hat{v}(S', \mathbf{w}) - \hat{v}(S, \mathbf{w})$

    |  $\mathbf{w} \leftarrow \mathbf{w} + \alpha\delta\mathbf{z}$

    |  $S \leftarrow S'$

  until  $S'$  is terminal

Sutton and Barto,  
Reinforcement  
Learning, 2018



# TD-Gammon

- To apply the learning rule we need a source of backgammon games
- Tesauro obtained an unending sequence of games by playing his learning backgammon player against itself
- To choose its moves, TD-Gammon considered each of the 20 or so ways it could play its dice roll and the corresponding resulting positions
- The resulting positions are afterstates
- The network was consulted to estimate each of their values
- The move was then selected that would lead to the position with the highest estimated value
- With TD-Gammon making the moves for both sides, it was possible to easily generate large numbers of backgammon games.
- Each game was treated as an episode, with the sequence of positions acting as the states,  $S_0, S_1, S_2, \dots$
- The nonlinear TD rule is used fully incrementally, after each individual move



# TD-Gammon

- The weights of the network were set initially to small random values; the initial evaluations were thus entirely arbitrary
- Because the moves were selected on the basis of these evaluations, the initial moves were inevitably poor, and the initial games often lasted hundreds or thousands of moves before one side or the other won, almost by accident
- After a few dozen games however, performance improved rapidly



# TD-Gammon

- After playing about 300,000 games against itself, TD-Gammon 0.0 as described above learned to play approximately as well as the best previous backgammon computer programs
- This was a striking result because all the previous high-performance computer programs had used extensive backgammon knowledge
- E.g., the best program at that time was Neurogammon (by Tesauro as well) that used an ANN but not TD learning
  - Neurogammon's network was trained on a large training corpus of exemplary moves provided by backgammon experts, and, in addition, started with a set of features specially crafted for backgammon
- TD-Gammon 0.0, on the other hand, was constructed with essentially zero backgammon knowledge
- That it was able to do as well as Neurogammon and all other approaches is striking testimony to the potential of self-play learning methods





# TD-Gammon

- The success of TD-Gammon 0.0 with zero expert backgammon knowledge suggested an obvious modification: add the specialized backgammon features but keep the self-play TD learning method
- This produced TD-Gammon 1.0 which was much better than previous programs and found serious competition only among human experts
- TD-Gammon 2.0 (40 hidden units) and TD-Gammon 2.1 (80 hidden units), were augmented with a selective two-ply search procedure
- To select moves, these programs looked ahead not just to the positions that would immediately result, but also to the opponent's possible dice rolls and moves
- Assuming the opponent always took the move that appeared immediately best for him, the expected value of each candidate move was computed and the best was selected



# TD-Gammon

- To save computer time, the second ply of search was conducted only for candidate moves that were ranked highly after the first ply, about four or five moves on average
- Two-ply search affected only the moves selected; the learning process proceeded exactly as before
- The final versions of the program, TD-Gammon 3.0 and 3.1, used 160 hidden units and a selective three-ply search
- TD-Gammon illustrates the combination of learned value functions and decision-time search (heuristic search)



# TD-Gammon

- During the 1990s, Tesauro was able to play his programs in a significant number of games against world-class human players
- TD-Gammon 3.0 appeared to play at close to, or possibly better than, the playing strength of the best human players in the world
- TD-Gammon 3.1 had a “lopsided advantage” in piece-movement decisions, and a “slight edge” in doubling decisions, over top humans

Program	Hidden Units	Training Games	Opponents	Results
TD-Gammon 0.0	40	300,000	other programs	tied for best
TD-Gammon 1.0	80	300,000	Robertie, Magriel, ...	-13 pts / 51 games
TD-Gammon 2.0	40	800,000	various Grandmasters	-7 pts / 38 games
TD-Gammon 2.1	80	1,500,000	Robertie	-1 pt / 40 games
TD-Gammon 3.0	80	1,500,000	Kazaros	+6 pts / 20 games



# TD-Gammon

- TD-Gammon had a significant impact on the way the best human players play the game
- E.g., it learned to play certain opening positions differently than was the convention among the best human players
- Based on TD-Gammon's success and further analysis, the best human players now play these positions as TD-Gammon does



# Outline

TD-Gammon

  **Human-level Video Game Play**

Mastering the Game of Go

Conclusion



# Human-level Video Game Play

- One of the greatest challenges in applying RL to real-world problems is deciding how to represent and store value functions and/or policies
- Unless the state set is finite and small enough to allow exhaustive representation by a lookup table—as in many of our illustrative examples—one must use a parameterized FA scheme
- Whether linear or nonlinear, FA relies on features that have to be readily accessible to the learning system and able to convey the information necessary for skilled performance
- Most successful applications of RL owe much to sets of features carefully handcrafted based on human knowledge and intuition about the specific problem to be tackled



# Human-level Video Game Play

- DeepMind developed an impressive demonstration that a deep multi-layer ANN can automate the feature design process
- Multi-layer ANNs have been used for FA in RL; striking results have been obtained by coupling RL with backpropagation (e.g., TD-Gammon and Watson)
- However, in these examples, the most impressive demonstrations required the network's input to be represented in terms of specialized features handcrafted for the given problem
- Mnih et al. (DeepMind) developed a RL agent called deep Q-network (DQN) that combined Q-learning with a deep convolutional ANN
- Mnih et al. used DQN to show how a RL agent can achieve a high level of performance on any of a collection of different problems without having to use different problem-specific feature sets



# Human-level Video Game Play

- Mnih et al. let DQN learn to play 49 different Atari 2600 video games by interacting with a game emulator
- DQN learned a different policy for each of the 49 games, but it used the same raw input, network architecture, and parameter values (e.g., step size, discount rate, exploration parameters, etc.) for all the games
- DQN achieved levels of play at or beyond human level on a large fraction of these games
- The games varied widely in other respects; their actions had different effects, they had different state-transition dynamics, and they needed different policies for learning high scores
- The deep convolutional ANN learned to transform the raw input common to all the games into features specialized for representing the action values required for playing at the high level DQN achieved for most of the games





# Human-level Video Game Play

- The Atari 2600 is a home video game console that was sold in various versions by Atari Inc. from 1977 to 1992
- Atari 2600 games have been attractive as testbeds for developing and evaluating RL methods
- Bellemare, Naddaf, Veness, and Bowling (2012) developed the publicly available Arcade Learning Environment (ALE) to encourage and simplify using Atari 2600 games to study learning and planning algorithms





# Atari 2600 Games



<https://arxiv.org/pdf/1312.5602.pdf>



# Human-level Video Game Play

- Mnih et al. used DQN for the learning
- DQN is similar to TD-Gammon in using a multi-layer ANN as the FA method for a semi-gradient form of a TD algorithm, with the gradients computed by the backpropagation
- However, instead of using  $TD(\lambda)$  as TD-Gammon did, DQN used the semi-gradient form of Q-learning
- TD-Gammon estimated the values of afterstates, which were easily obtained from the rules for making backgammon moves
- To use the same algorithm for the Atari games would have required generating the next states for each possible action



# Human-level Video Game Play

- This could have been done by using the game emulator to run single-step simulations for all the possible actions (which ALE makes possible), or a model of each game's state-transition function could have been learned and used to predict next states
- While these methods might have produced results comparable to DQN's, they would have been more complicated to implement and would have significantly increased the time needed for learning
- Another motivation for using Q-learning was that DQN used the *experience replay* method which requires an off-policy algorithm
- Being model-free and off-policy made Q-learning a natural choice



# Human-level Video Game Play

- Performance of DQN
  - Mnih et al. compared the scores of DQN with competitors in 46 games
  - The best system from the literature used linear FA with features designed using some knowledge about Atari 2600 games
  - DQN learned on each game by interacting with the game emulator for 50 million frames, which corresponds to about 38 days of experience with the game
  - To evaluate DQN's skill level after learning, its score was averaged over 30 sessions on each game, each lasting up to 5 minutes and beginning with a random initial game state
  - The professional human tester played using the same emulator
  - DQN learned to play better than the best previous RL systems on 40 of the 46 games, and played better than the human player on 22 of the games



# Human-level Video Game Play

- The breakthrough is that *the very same learning system* achieved these levels of play on widely varying games without relying on any game-specific modifications
- A human playing any of these 49 Atari games sees 210 x 160 pixel image frames with 128 colors
- To reduce memory and computation, Mnih et al. preprocessed each frame to produce an 84 x 84 array of luminance values
- Because the full states of many of the Atari games are not completely observable from the image frames, they “stacked” the four most recent frames so that the inputs to the network had dimension 84 x 84 x 4
- This did not eliminate partial observability for all of the games, but it was helpful in making many of them more Markovian
- These preprocessing steps were exactly the same for all 46 games; no game-specific prior knowledge was involved

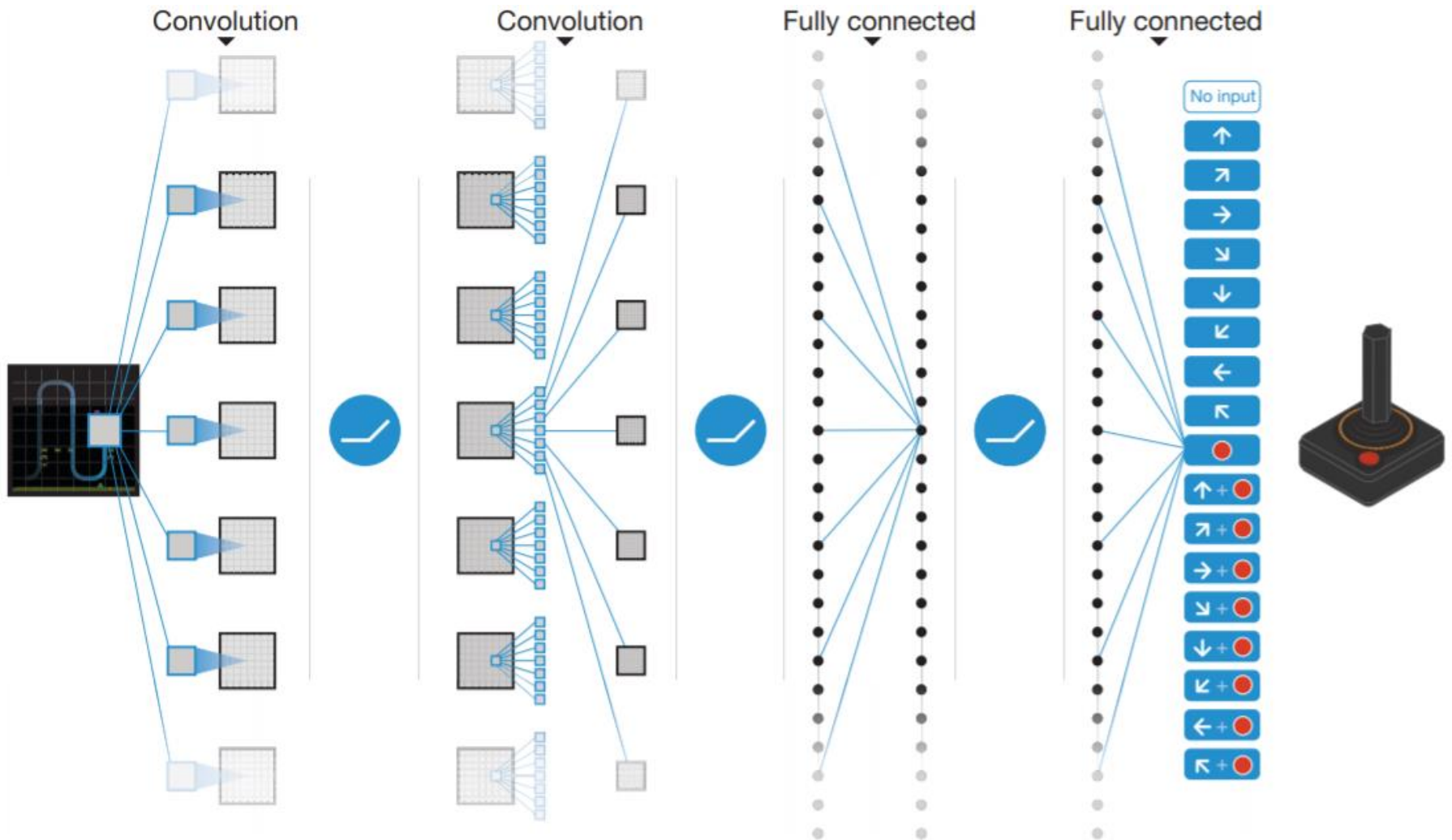


# Human-level Video Game Play

- The basic architecture of DQN is similar to the deep convolutional ANN
- DQN has three hidden convolutional layers, followed by one fully connected hidden layer, followed by the output layer
- The three successive hidden convolutional layers of DQN produce  $32 \times 20 \times 20$  feature maps,  $64 \times 9 \times 9$  feature maps, and  $64 \times 7 \times 7$  feature maps
- The activation function of the units of each feature map is a rectifier nonlinearity ( $\max(0, x)$ )
- The 3,136 ( $64 \times 7 \times 7$ ) units in this third convolutional layer all connect to each of 512 units in the fully connected hidden layer, which then each connect to all 18 units in the output layer, one for each possible action in an Atari game



# Human-level Video Game Play







# Human-level Video Game Play

- The activation levels of DQN's output units were the estimated optimal action values of the corresponding state–action pairs, for the state represented by the network's input
- The assignment of output units to a game's actions varied from game to game, and because the number of valid actions varied between 4 and 18 for the games, not all output units had functional roles in all of the games
- It helps to think of the network as if it were 18 separate networks, one for estimating the optimal action value of each possible action
- In reality, these networks shared their initial layers, but the output units learned to use the features extracted by these layers in different ways



# Human-level Video Game Play

- DQN's reward signal indicated how a game's score changed from one time step to the next: +1 whenever it increased, -1 whenever it decreased, and 0 otherwise
- This standardized the reward signal across the games and made a single step-size parameter work well for all the games
- DQN used an  $\epsilon$ -greedy policy, with  $\epsilon$  decreasing linearly over the first million frames and remaining at a low value for the rest of the learning session



# Q-learning: Off-policy TD Control

Q-learning (off-policy TD control) for estimating  $\pi \approx \pi_*$

Algorithm parameters: step size  $\alpha \in (0, 1]$ , small  $\varepsilon > 0$

Initialize  $Q(s, a)$ , for all  $s \in \mathcal{S}^+$ ,  $a \in \mathcal{A}(s)$ , arbitrarily except that  $Q(\text{terminal}, \cdot) = 0$

Loop for each episode:

  Initialize  $S$

  Loop for each step of episode:

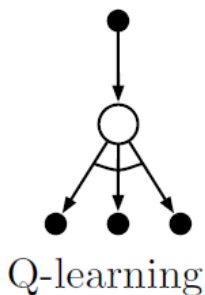
    Choose  $A$  from  $S$  using policy derived from  $Q$  (e.g.,  $\varepsilon$ -greedy)

    Take action  $A$ , observe  $R, S'$

$Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$

$S \leftarrow S'$

  until  $S$  is terminal





# Human-level Video Game Play

- After DQN selected an action, the action was executed by the game emulator, which returned a reward and the next video frame
- The frame was preprocessed and added to the four-frame stack that became the next input to the network
- DQN used the semi-gradient form of Q-learning to update the weights:

$$w_{t+1} = w_t + \alpha \left[ R_{t+1} + \gamma \max_a \hat{q}(S_{t+1}, a, w_t) - \hat{q}(S_t, A_t, w_t) \right] \nabla \hat{q}(S_t, A_t, w_t)$$

- The gradient was computed by backpropagation
- Mnih et al. used a *mini-batch* method that updated weights only after accumulating gradient information over a small batch of images (here after 32 images)
- This yielded smoother sample gradients compared to the usual procedure that updates weights after each action



# Human-level Video Game Play

---

**Algorithm 1** Deep Q-learning with Experience Replay

---

Initialize replay memory  $\mathcal{D}$  to capacity  $N$

Initialize action-value function  $Q$  with random weights

**for** episode = 1,  $M$  **do**

    Initialize sequence  $s_1 = \{x_1\}$  and preprocessed sequenced  $\phi_1 = \phi(s_1)$

**for**  $t = 1, T$  **do**

        With probability  $\epsilon$  select a random action  $a_t$

        otherwise select  $a_t = \max_a Q^*(\phi(s_t), a; \theta)$

        Execute action  $a_t$  in emulator and observe reward  $r_t$  and image  $x_{t+1}$

        Set  $s_{t+1} = s_t, a_t, x_{t+1}$  and preprocess  $\phi_{t+1} = \phi(s_{t+1})$

        Store transition  $(\phi_t, a_t, r_t, \phi_{t+1})$  in  $\mathcal{D}$

        Sample random minibatch of transitions  $(\phi_j, a_j, r_j, \phi_{j+1})$  from  $\mathcal{D}$

        Set  $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$

        Perform a gradient descent step on  $(y_j - Q(\phi_j, a_j; \theta))^2$

**end for**

**end for**

---



# Human-level Video Game Play

- Mnih et al. modified the basic Q-learning procedure in three ways
- 1) Experience replay
  - Store the agent's experience at each time step in a replay memory that is accessed to perform the weight updates
  - After the game emulator executed action  $A_t$  in a state represented by the image stack  $S_t$ , and returned reward  $R_{t+1}$  and image stack  $S_{t+1}$ , it added the tuple  $(S_t, A_t, R_{t+1}, S_{t+1})$  to the replay memory
  - This memory accumulated experiences over many plays of the same game
  - At each time step multiple Q-learning updates (a mini-batch) were performed based on experiences sampled uniformly at random from the replay memory



# Human-level Video Game Play

- 1) Experience replay
  - Q-learning with experience replay provided several advantages over the usual form of Q-learning
  - The ability to use each stored experience for many updates allowed DQN to learn more efficiently from its experiences
  - Experience replay reduced the variance of the updates because successive updates were not correlated with one another as they would be with standard Q-learning



# Human-level Video Game Play

- 2) Fixed target (for stable learning)
  - As in other methods that bootstrap, the target for a Q-learning update depends on the current action-value function estimate
  - Its dependence on  $w_t$  complicates the process compared to the simpler supervised-learning situation in which the targets do not depend on the parameters being updated
  - Solution: whenever a certain number,  $C$ , of updates had been done to the weights  $w$  of the action-value network, they inserted the network's current weights into another network and held these duplicate weights fixed for the next  $C$  updates of  $w$
  - The outputs of this duplicate network over the next  $C$  updates of  $w$  were used as the Q-learning targets
  - Letting  $\tilde{q}$  denote the output of this duplicate network, the update rule was:

$$w_{t+1} = w_t + \alpha \left[ R_{t+1} + \gamma \max_a \tilde{q}(S_{t+1}, a, w_t) - \hat{q}(S_t, A_t, w_t) \right] \nabla \hat{q}(S_t, A_t, w_t)$$





# Human-level Video Game Play

- 3) Error clipping
  - Goal: to improve stability
  - Clipped the error term

$$R_{t+1} + \gamma \max_a \tilde{q}(S_{t+1}, a, w_t) - \hat{q}(S_t, A_t, w_t)$$

- so that it remained in the interval  $[-1, 1]$
- Reminder: the parameter is updated by

$$w_{t+1} = w_t + \alpha \left[ R_{t+1} + \gamma \max_a \hat{q}(S_{t+1}, a, w_t) - \hat{q}(S_t, A_t, w_t) \right] \nabla \hat{q}(S_t, A_t, w_t)$$



# Human-level Video Game Play

- Creating artificial agents that excel over a diverse collection of challenging tasks has been an enduring goal of AI
- The promise of machine learning as a means for achieving this has been frustrated by the need to craft problem-specific representations
- DeepMind's DQN stands as a major step forward by demonstrating that a single agent can learn problem-specific features enabling it to acquire human-competitive skills over a range of tasks



# Human-level Video Game Play


- However, DQN is not a complete solution to the problem of task-independent learning
  - All the games were played by observing video images and using CNN
  - In addition, DQN's performance on some of the Atari 2600 games fell short of human skill levels on these games (e.g., Montezuma's Revenge)
  - <https://www.youtube.com/watch?v=TU-h8zLM2jA>



- Learning control skills through extensive practice, like DQN learned how to play the Atari games, is just one of the types of learning humans routinely accomplish
- Despite these limitations, DQN advanced the state-of-the-art in machine learning by combining RL and deep learning



# Outline

- TD-Gammon
- Human-level Video Game Play
-   **Mastering the Game of Go**
- Conclusion



# Mastering the Game of Go

- The ancient Chinese game of Go has challenged AI researchers for many decades
- Methods that achieve human-level skill in other games have not been successful in producing strong Go programs
- DeepMind developed the program AlphaGo that broke this barrier by combining deep ANNs, supervised learning, MCTS, and RL
- In 2016, AlphaGo had been shown to be decisively stronger than other current Go programs, and it had defeated the European Go champion Fan Hui 5 games to 0
- Shortly thereafter, a similar version of AlphaGo won stunning victories over the 18-time world champion Lee Sedol, winning 4 out of a 5 games in a challenge match, making worldwide headline news
- AI researchers thought that it would be many more years, perhaps decades, before a program reached this level of play



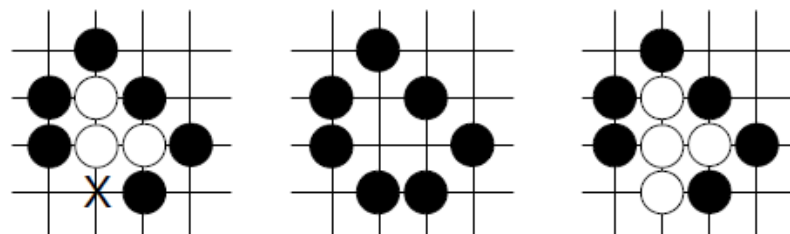
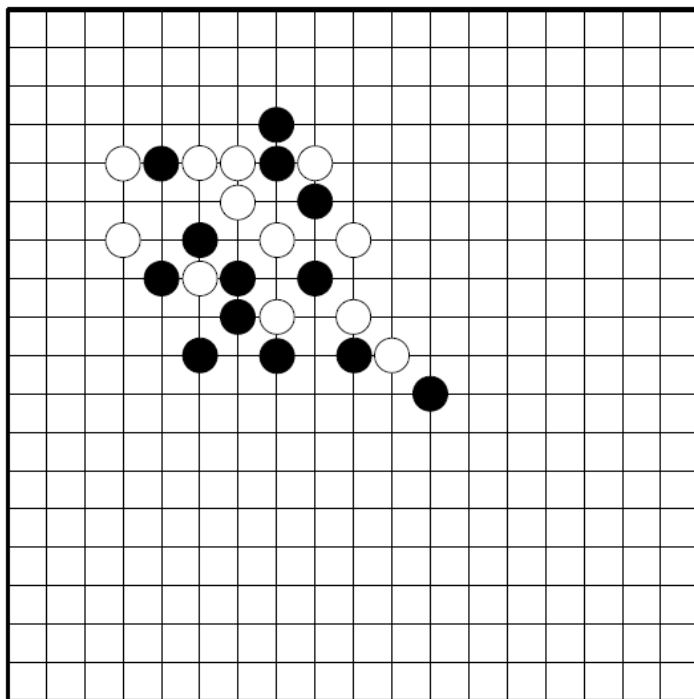
# Mastering the Game of Go

- We discuss AlphaGo (2016) and a successor program called AlphaGo Zero (2017)
- In addition to RL, AlphaGo relied on supervised learning from a large database of expert human moves; however, AlphaGo Zero used only RL and no human data or guidance beyond the basic rules of the game (hence the Zero in its name)
- In many ways, both AlphaGo and AlphaGo Zero are descendants of Tesauro's TD-Gammon
- All these programs included RL over simulated games of self-play
- AlphaGo and AlphaGo Zero also built upon the progress made by DeepMind on playing Atari games with the program DQN that used deep convolutional ANNs to approximate optimal value functions



# Mastering the Game of Go

- Go game



Sutton and Barto,  
Reinforcement  
Learning, 2018



# Mastering the Game of Go

- Difficulties of AI for Go
  - Methods that produce strong play for other games, such as chess, have not worked as well for Go
  - The search space for Go is significantly larger than that of chess because Go has a larger number of legal moves per position than chess (~ 250 versus ~ 35) and Go games tend to involve more moves than chess games (~ 150 versus ~ 80); but the size of the search space is not the major factor that makes Go so difficult
  - Exhaustive search is infeasible for both chess and Go, and Go on smaller boards (9 x 9) has proven to be exceedingly difficult as well
  - Experts agree that the major stumbling block to creating stronger-than-amateur Go programs is the difficulty of defining an adequate position evaluation function
  - A good evaluation function allows search to be truncated at a feasible depth by providing relatively easy-to-compute predictions of what deeper search would likely yield



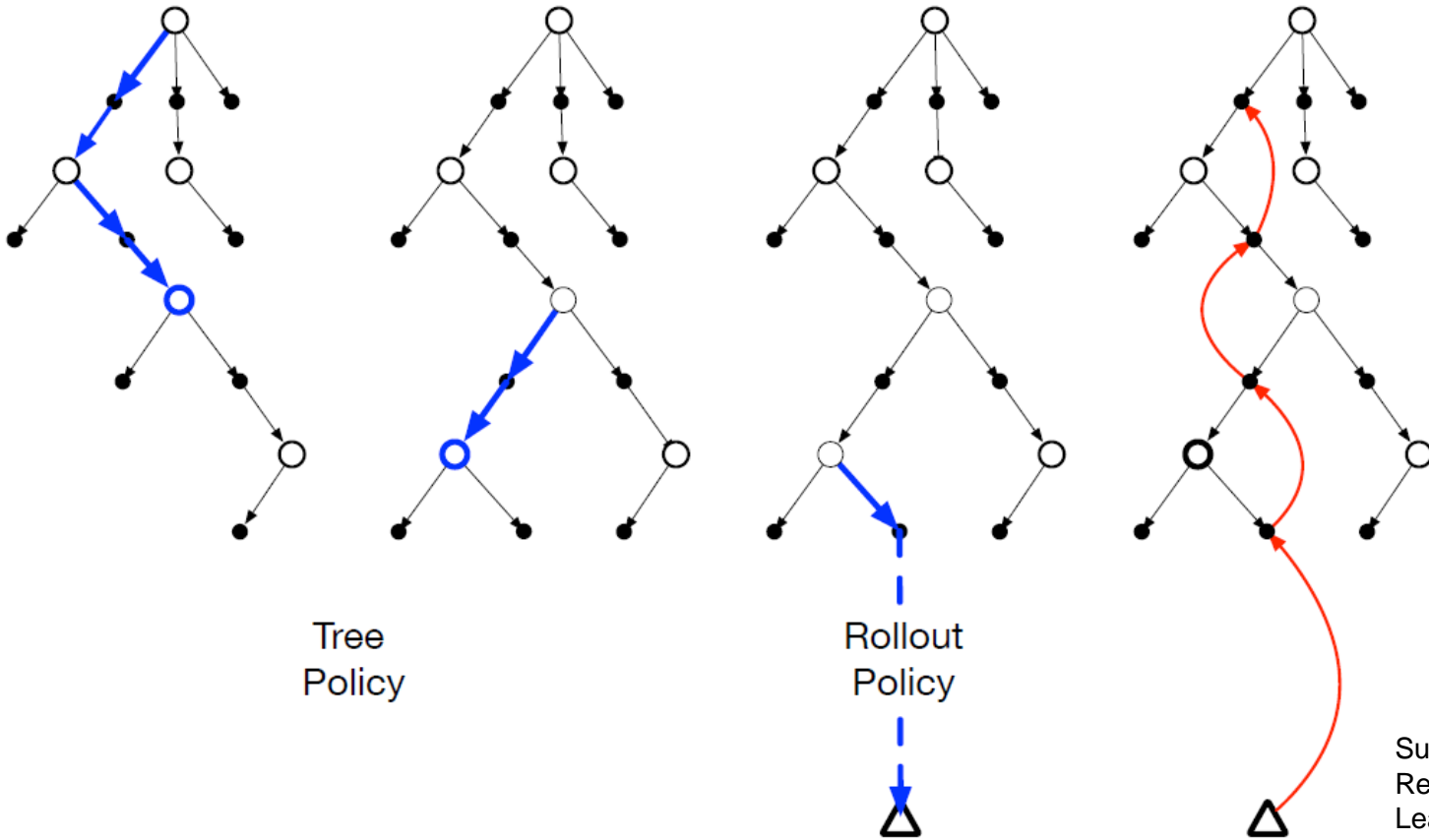
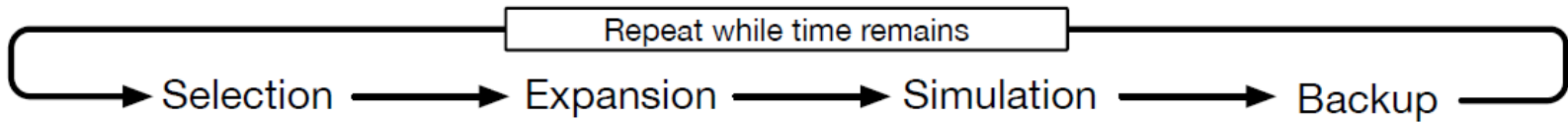


# AlphaGo

- Key features of AlphaGo
  - It selected moves by APV-MCTS (Asynchronous Policy and Value MCTS), a novel version of MCTS that was guided by both a policy and a value function learned by RL with FA provided by deep convolutional ANNs
  - Instead of RL starting from random network weights, it started from weights that were the result of previous supervised learning from a large collection of human expert moves



# Monte Carlo Tree Search



Tree  
Policy

Rollout  
Policy

Sutton and Barto,  
Reinforcement  
Learning, 2018



# AlphaGo

- AlphaGo is based on MCTS
- Steps of MCTS
  - Selection: uses the action value and prior probabilities

$$a_t = \operatorname{argmax}_a (Q(s_t, a) + u(s_t, a))$$

- Prior probabilities (or upper confidence bound):

$$u(s, a) \propto \frac{P(s, a)}{1 + N(s, a)}$$

← output from SL policy network  $p_\sigma$   
← encourage exploration

- Update of action value:

$$N(s, a) = \sum_{i=1}^n 1(s, a, i)$$
$$Q(s, a) = \frac{1}{N(s, a)} \sum_{i=1}^n 1(s, a, i) V(s_L^i)$$

L: leaf node  
 $z_L$ : 1 (win) or -1 (lose)  
 $v_\theta(s_L)$ : value network  
 $\lambda$ : mixing weight (best=0.5)

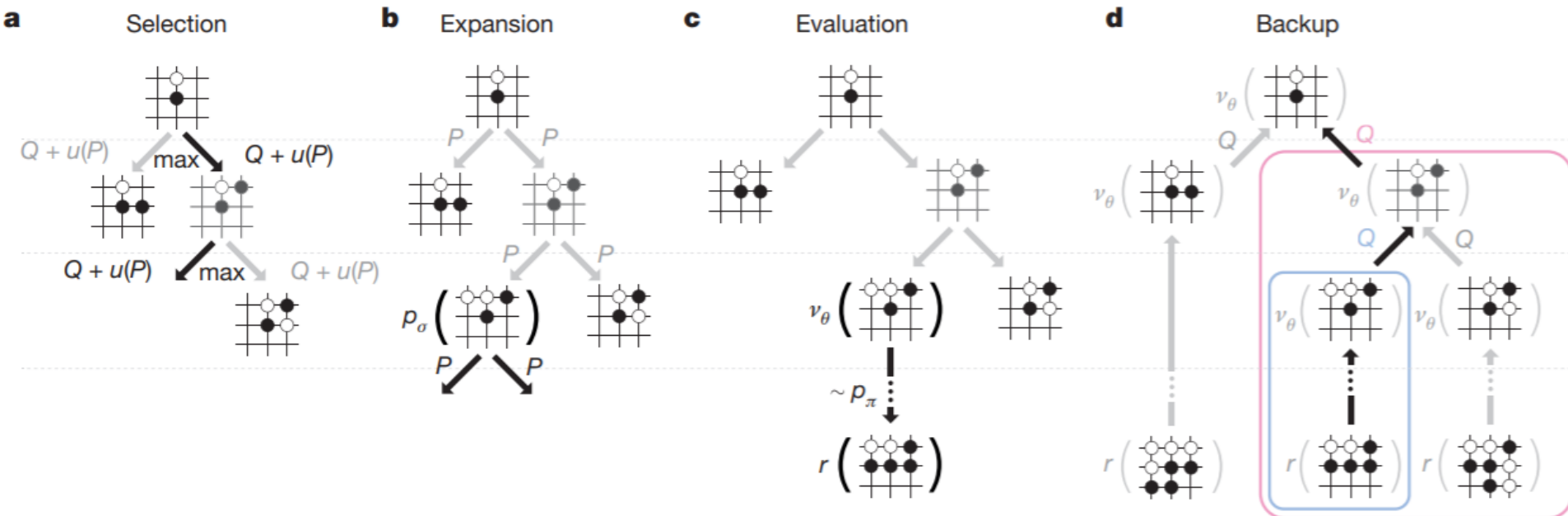
$$V(s_L) = (1 - \lambda)v_\theta(s_L) + \lambda z_L$$

← from rollout policy



# AlphaGo

## ■ MCTS in AlphaGo

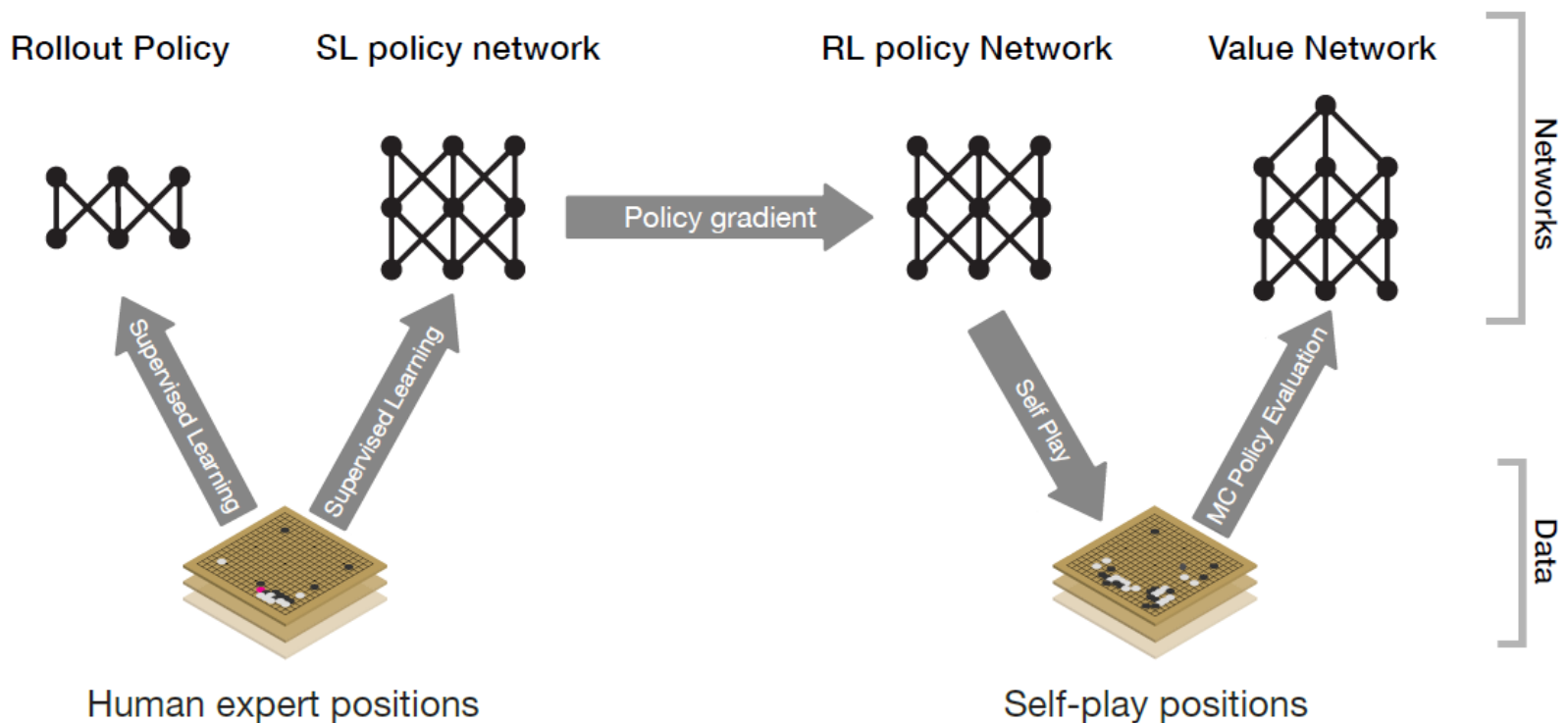




# AlphaGo

- Key components of APV-MCTS (Asynchronous Policy and Value MCTS)
  - SL policy network  $p_{\sigma}(a|s)$
  - Rollout policy network  $p_{\pi}(a|s)$
  - Value network  $v_{\theta}(s)$
  - RL policy network  $p_{\rho}(a|s)$

<https://www.nature.com/articles/nature16961>



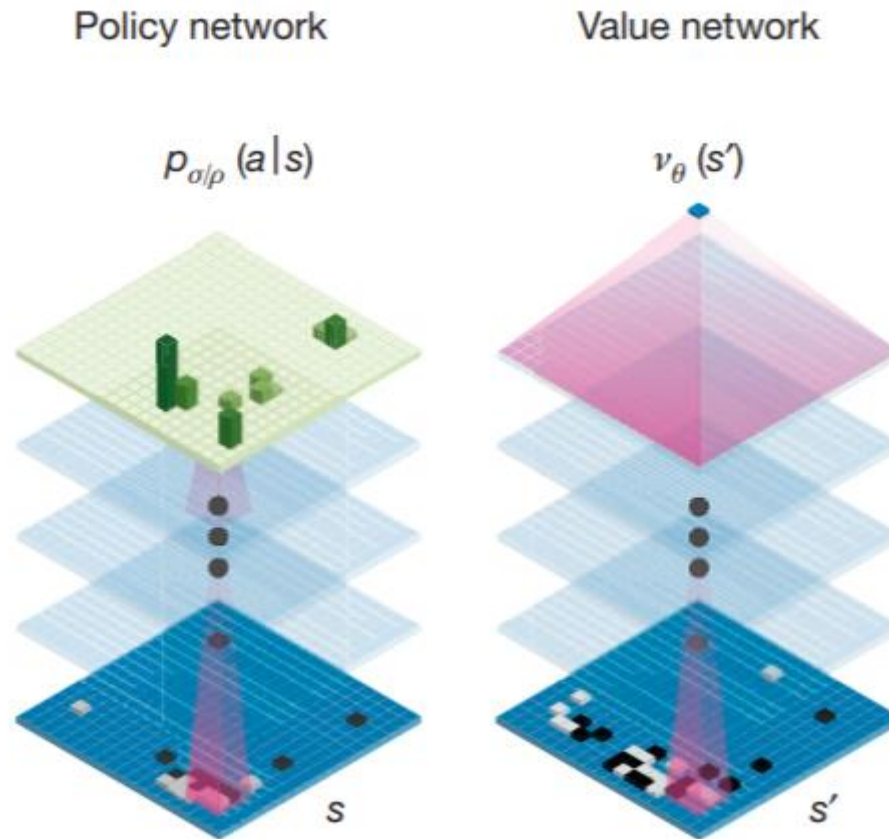


# AlphaGo

- SL policy network  $p_{\sigma}(a|s)$ 
  - Can be thought of as a classification model, which receives a state as an input, and outputs action probabilities
  - Uses 30 million training data of human moves from KGS Go Server
  - 13-layer CNN
  - The networks' input was a 19 x 19 x 48 image stack in which each point on the Go board was represented by the values of 48 binary or integer-valued features (in a sense, similar to CNN for DQN in Atari Games)
  - For each point, one feature indicated if the point was occupied by one of AlphaGo's stones, one of its opponent's stones, or was unoccupied, thus providing the "raw" representation of the board configuration
  - Other features were based on the rules of Go, such as the number of adjacent points that were empty, the number of opponent stones that would be captured by placing a stone there, the number of turns since a stone was placed there, and other features that the design team considered to be important



# AlphaGo



<https://www.nature.com/articles/nature16961>



# AlphaGo

- Rollout policy network  $p_{\pi}(a|s)$ 
  - Can be thought of as a classification model, which receives a state as an input, and outputs action probabilities
  - A simple linear network for fast simulation





# AlphaGo

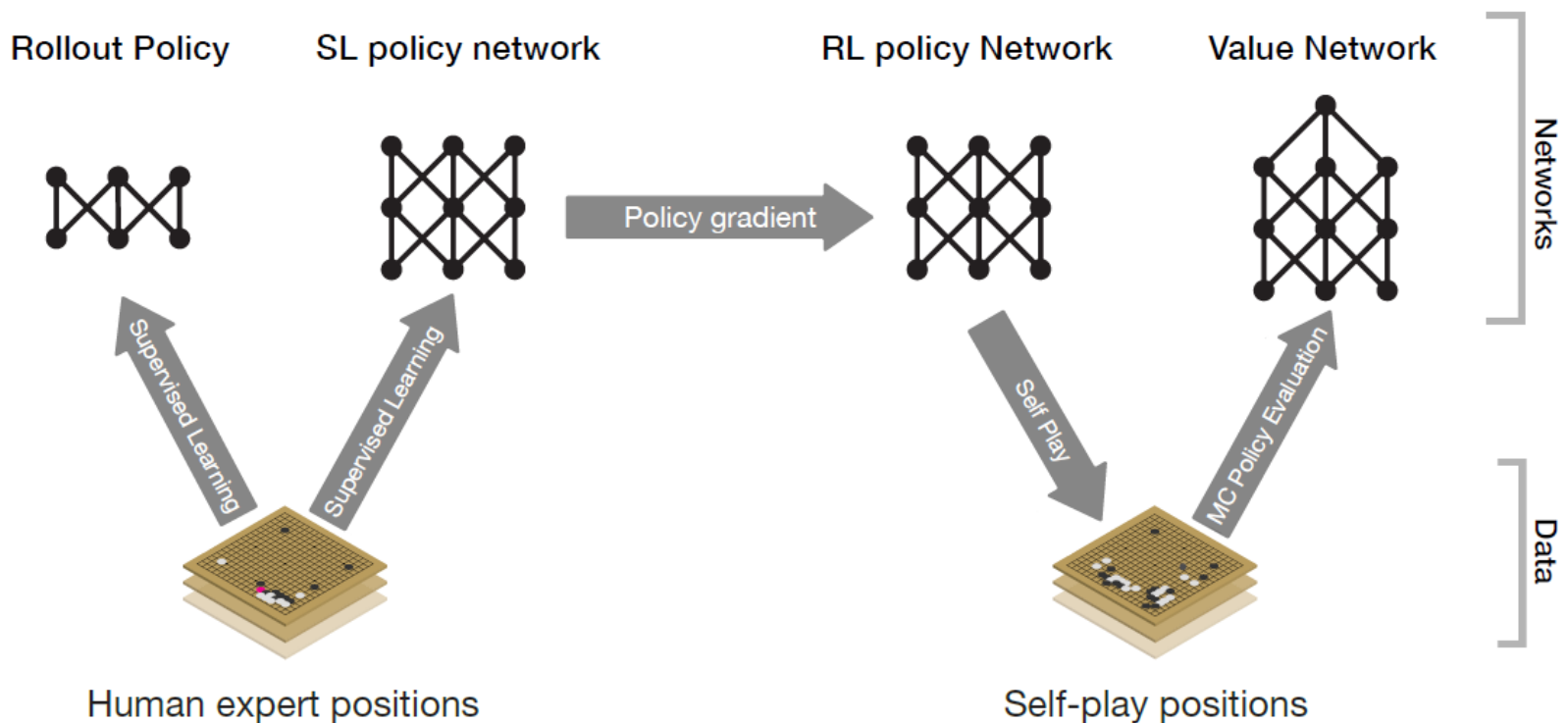
- Value network  $v_{\theta}(s)$ 
  - The value network had the same structure as SL policy network except that it had a single output unit that gave estimated values of game positions instead of the SL policy network's probability distributions over legal actions
  - They divided the process of training the value network into two stages
  - In the first stage, they learned RL policy network  $p_{\rho}(a|s)$  using policy gradient method (REINFORCE) via self-play games
    - $p_{\rho}(a|s)$  has the same structure as the SL policy network. It was initialized with the final weights of the SL policy network that were learned via supervised learning
  - In the second stage, the value network  $v_{\theta}(s)$  was trained with MC policy evaluation on data obtained from a large number of simulated self-play games with moves selected by the RL policy network  $p_{\rho}(a|s)$



# AlphaGo

- Key components of APV-MCTS (Asynchronous Policy and Value MCTS)
  - SL policy network  $p_{\sigma}(a|s)$
  - Rollout policy network  $p_{\pi}(a|s)$
  - Value network  $v_{\theta}(s)$
  - RL policy network  $p_{\rho}(a|s)$

<https://www.nature.com/articles/nature16961>





# AlphaGo

- Why the SL policy was used instead of the better RL policy to select actions in the expansion phase of APV-MCTS ?
  - These policies took the same amount of time to compute because they used the same network architecture
  - The team actually found that AlphaGo played better against human opponents when APV-MCTS used as the SL policy instead of the RL policy
  - They conjectured that the reason for this was that the latter was tuned to respond to optimal moves rather than to the broader set of moves characteristic of human play
  - Interestingly, the situation was reversed for the value function used by APV-MCTS
  - They found that when APV-MCTS used the value function derived from the RL policy, it performed better than if it used the value function derived from the SL policy



# AlphaGo Zero

- Building upon the experience with AlphaGo, DeepMind developed AlphaGo Zero
- In contrast to AlphaGo, this program used no human data or guidance beyond the basic rules of the game (hence the Zero in its name)
- It learned exclusively from self-play RL, with input giving just “raw” descriptions of the placements of stones on the Go board
- AlphaGo Zero implemented a form of policy iteration, interleaving policy evaluation with policy improvement
- AlphaGo Zero used MCTS to select moves throughout self-play RL, whereas AlphaGo used MCTS for live play after—but not during—learning
- Other differences besides not using any human data or human-crafted features are that AlphaGo Zero used only one deep convolutional ANN and used a simpler version of MCTS



# AlphaGo Zero

- A deep neural network  $f_\theta$  with parameter  $\theta$  takes as an input the raw board representation  $s$  of the position and its history, and outputs both move probabilities and a value,  $(\mathbf{p}, v) = f_\theta(s)$
- This neural network combines the policy network and value network into a single architecture
- The neural network consists of many residual blocks of convolutional layers with batch normalization and rectifier nonlinearities



# AlphaGo Zero

- The neural network in AlphaGo Zero is trained from games of self-play by a novel RL algorithm
- In each position  $s$ , an MCTS search is executed, guided by the neural network  $f_\theta$
- The MCTS search outputs probabilities  $\pi$  of playing each move. These search probabilities usually select much stronger moves than the raw move probabilities  $p$  of the neural network  $f_\theta(s)$ ; MCTS may therefore be viewed as a powerful policy improvement operator
- Self-play with search (using the improved MCTS-based policy to select each move, then using the game winner  $z$  as a sample of the value) may be viewed as a powerful policy evaluation operator



# AlphaGo Zero

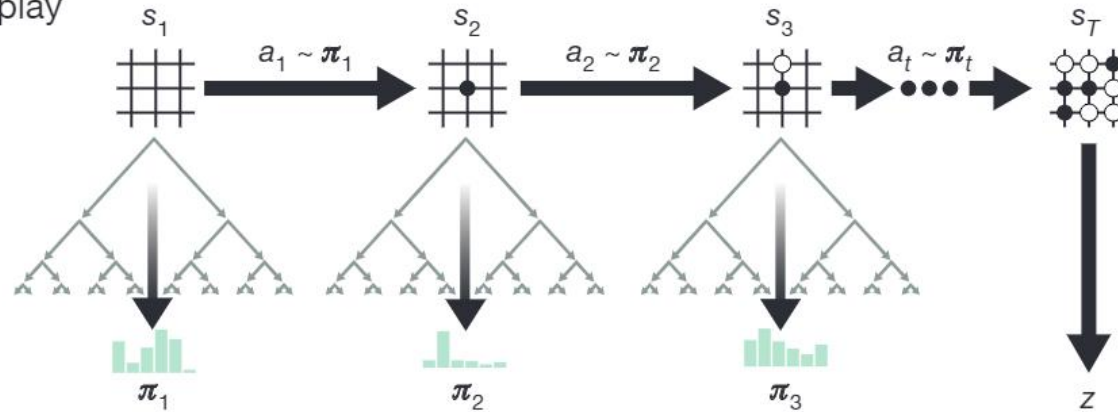
- The main idea is to use these search operators repeatedly in a policy iteration procedure
- The neural network's parameters are updated to make the move probabilities and value  $(\mathbf{p}, v) = f_{\theta}(s)$  more closely match the improved search probabilities and self-play winner  $(\pi, z)$
- These new parameters are used in the next iteration of self-play to make the search even stronger



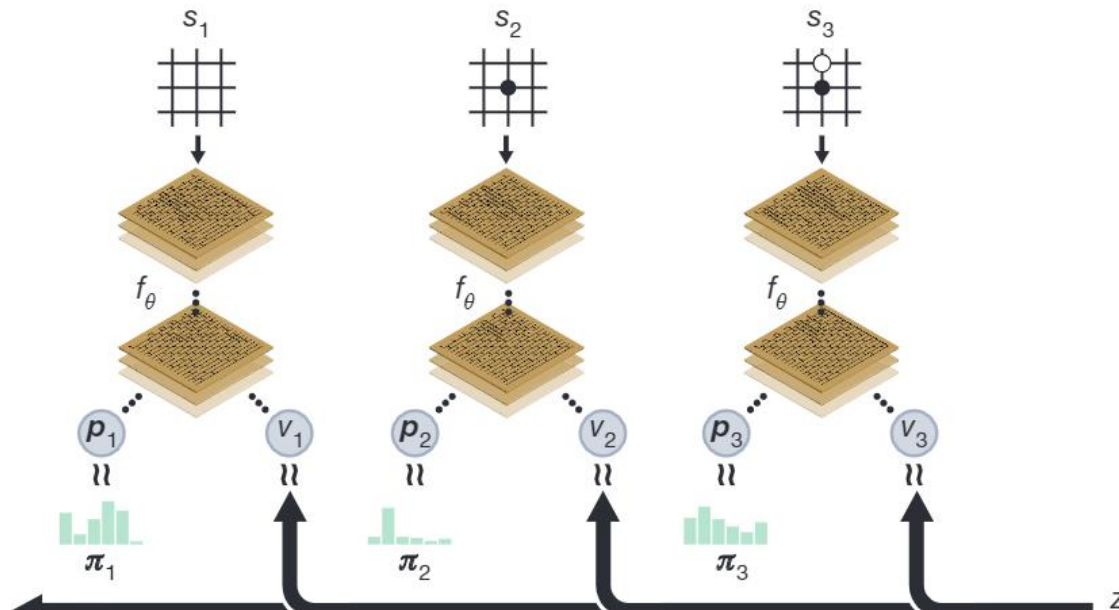
# AlphaGo Zero

<https://www.nature.com/articles/nature24270>

## a Self-play



## b Neural network training





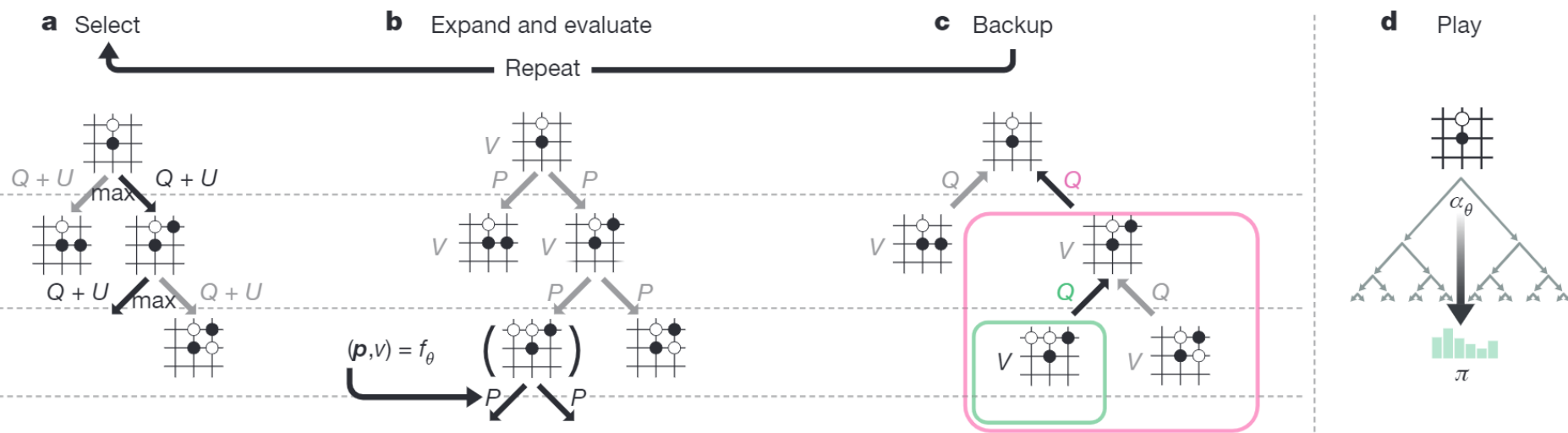


# AlphaGo Zero

## ■ MCTS of AlphaGo Zero

- The MCTS uses the neural network  $f_\theta$  to guide its simulations
- Each edge  $(s, a)$  in the search tree stores a prior probability  $P(s, a)$ , a visit count  $N(s, a)$ , and an action value  $Q(s, a)$
- Each simulation starts from the root state and iteratively selects moves that maximize an upper confidence bound  $Q(s, a) + U(s, a)$ , where  $U(s, a) \propto P(s, a) / (1 + N(s, a))$ , until a leaf node  $s'$  is encountered

<https://www.nature.com/articles/nature24270>



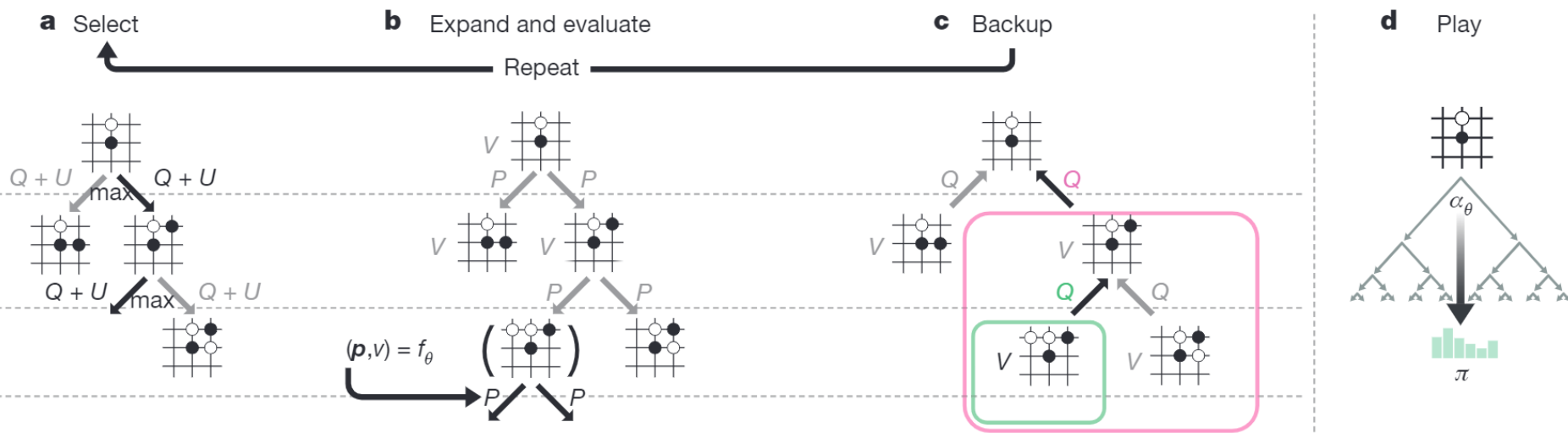


# AlphaGo Zero

## ■ MCTS of AlphaGo Zero

- This leaf position is expanded and evaluated only once by the network to generate both prior probabilities and evaluation,  $(P(s', \cdot), V(s')) = f_\theta(s)$
- Each edge  $(s, a)$  traversed in the simulation is updated to increment its visit count  $N(s, a)$ , and to update its action value to the mean evaluation over these simulations,  $Q(s, a) = [\sum_{s'|s,a \rightarrow s'} V(s')]/N(s, a)$  where  $s, a \rightarrow s'$  indicates that a simulation eventually reached  $s'$  after taking move  $a$  from position  $s$

<https://www.nature.com/articles/nature24270>



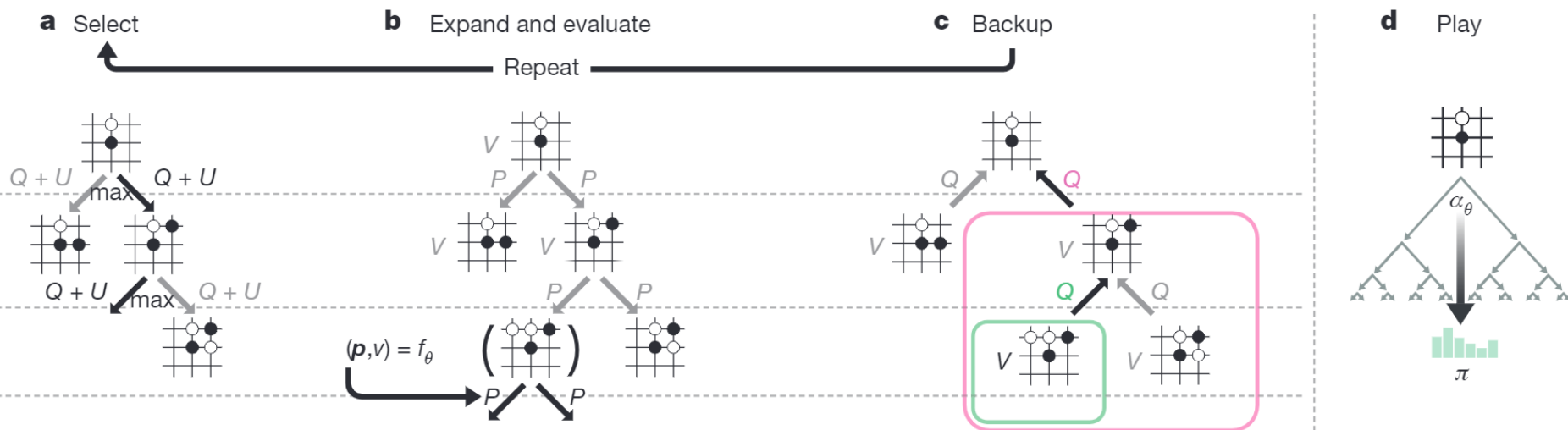


# AlphaGo Zero

## ■ MCTS of AlphaGo Zero

- MCTS may be viewed as a self-play algorithm that, given neural network parameters  $\theta$  and a root position  $s$ , computes a vector of search probabilities recommending moves to play,  $\pi = \alpha_\theta(s)$ , proportional to the exponentiated visit count for each move,  $\pi_a \propto N(s, a)^{1/\tau}$ , where  $\tau$  is a temperature parameter.

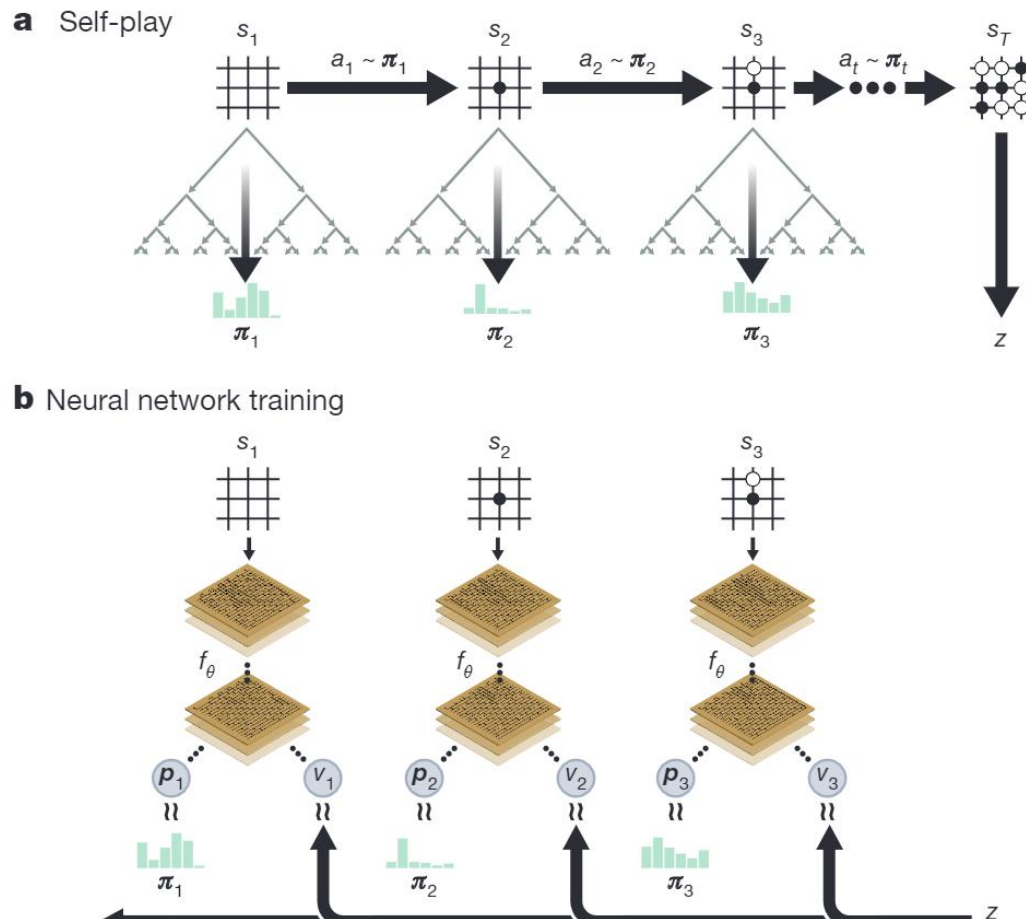
<https://www.nature.com/articles/nature24270>





# AlphaGo Zero

- Training of AlphaGo Zero's ANN
  - The ANN is trained by a self-play RL that uses MCTS to play each move





# AlphaGo Zero

- Training of AlphaGo Zero's ANN
  - First, the neural network is initialized to random weights  $\theta_0$
  - At each subsequent iteration  $i \geq 1$ , games of self-play are generated
  - At each timestep  $t$ , an MCTS search  $\boldsymbol{\pi}_t = \alpha_{\theta_{i-1}}(s_t)$  is executed using the previous iteration of neural network  $f_{\theta_{i-1}}$  and a move is played by sampling the search probabilities  $\boldsymbol{\pi}_t$
  - A game terminates at step  $T$  when both players pass, when the search value drops below a resignation threshold or when the game exceeds a maximum length; the game is then scored to give a final reward of  $r_T \in \{-1, +1\}$
  - The data for each timestep  $t$  is stored as  $(s_t, \boldsymbol{\pi}_t, z_t)$ , where  $z_t = \pm r_T$  is the game winner from the perspective of the current player at step  $t$



# AlphaGo Zero

- Training of AlphaGo Zero's ANN
  - In parallel, new network parameters  $\theta_i$  are trained from data  $(s, \pi, z)$  sampled uniformly among all timesteps of the last iteration(s) of self-play
  - The neural network  $(\mathbf{p}, v) = f_{\theta_i}(s)$  is adjusted to minimize the error between the predicted value  $v$  and the self-play winner  $z$ , and to maximize the similarity of the neural network move probabilities  $\mathbf{p}$  to the search probabilities  $\pi$
  - Specifically, the parameters  $\theta$  are adjusted by gradient descent on a loss function  $l$  that sums over the mean-squared error and cross-entropy losses, respectively

$$(p, v) = f_{\theta}(s) \text{ and } l = (z - v)^2 - \pi^{\top} \log p + c \|\theta\|^2$$



# AlphaGo Zero

- Features of AlphaGo Zero's ANN
  - The network took as input a 19 x 19 x 17 image stack consisting of 17 binary feature planes
  - The first 8 feature planes were raw representations of the positions of the current player's stones in the current and seven past board configurations: a feature value was 1 if a player's stone was on the corresponding point, and was 0 otherwise
  - The next 8 feature planes similarly coded the positions of the opponent's stones
  - A final input feature plane had a constant value indicating the color of the current play: 1 for black; 0 for white



# AlphaGo Zero

- AlphaGo Zero's performance
  - DeepMind team trained AlphaGo Zero over 4.9 million games of self-play, which took about 3 days; each move of each game was selected by running MCTS for 1,600 iterations, taking approximately 0.4 second per move
  - Network weights were updated over 700,000 batches each consisting of 2,048 board configurations
  - They then ran tournaments with the trained AlphaGo Zero playing against the version of AlphaGo that defeated Fan Hui by 5 games to 0, and against the version that defeated Lee Sedol by 4 games to 1
  - The Elo ratings of AlphaGo Zero, the version of AlphaGo that played against Fan Hui, and the version that played against Lee Sedol were respectively 4,308, 3,144, and 3,739
  - In a match of 100 games between AlphaGo Zero, and the exact version of AlphaGo that defeated Lee Sedol held under the same conditions that were used in that match, AlphaGo Zero defeated AlphaGo in all 100 games





# AlphaGo Zero

- AlphaGo Zero's performance
  - The DeepMind team also compared AlphaGo Zero with a program using an ANN with the same architecture but trained by supervised learning to predict human moves in a data set containing nearly 30 million positions from 160,000 games
  - They found that the supervised-learning player initially played better than AlphaGo Zero, and was better at predicting human expert moves, but played less well after AlphaGo Zero was trained for a day
  - This suggested that AlphaGo Zero had discovered a strategy for playing that was different from how humans play
  - In fact, AlphaGo Zero discovered, and came to prefer, some novel variations of classical move sequences




# AlphaGo Zero

- AlphaGo Zero soundly demonstrated that superhuman performance can be achieved by pure RL, augmented by a simple version of MCTS, and deep ANNs with very minimal knowledge of the domain and no reliance on human data or guidance



# Outline

- TD-Gammon
- Human-level Video Game Play
- Mastering the Game of Go
-   **Conclusion**



# Conclusion

- RL has been used for many interesting applications
  - TD-Gammon
  - Watson's daily-double wagering
  - Optimizing memory control
  - Human-level video game play
  - Go
  - Personalized web services
  - Thermal soaring
- There are many interesting opportunities for novel applications by RL



# Questions?