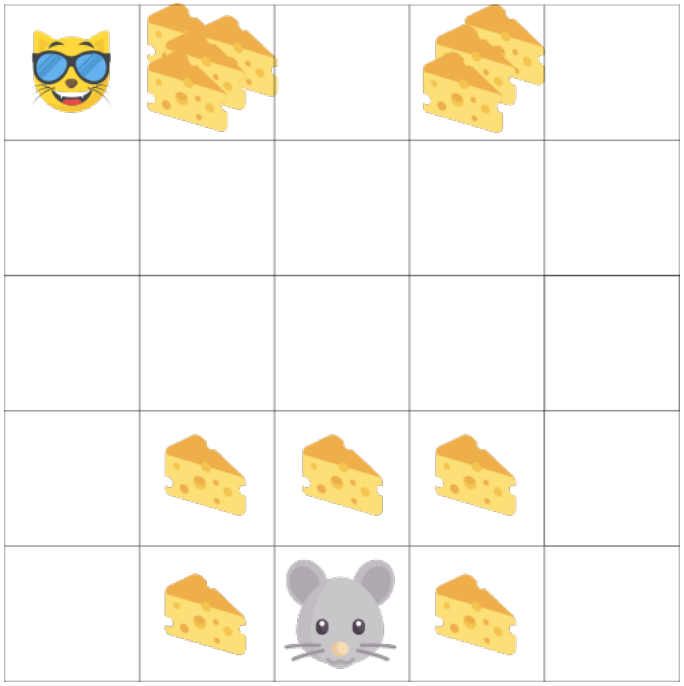
Reinforcement learning

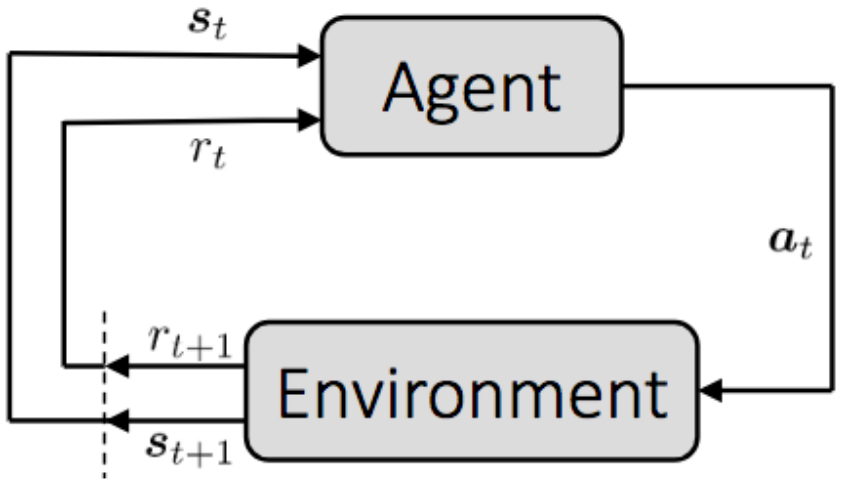
F. Noé[1]

Deep Learning Classes, FU Berlin 2018

Reinforcement Learning



Reinforcement Learning: Basic Terms

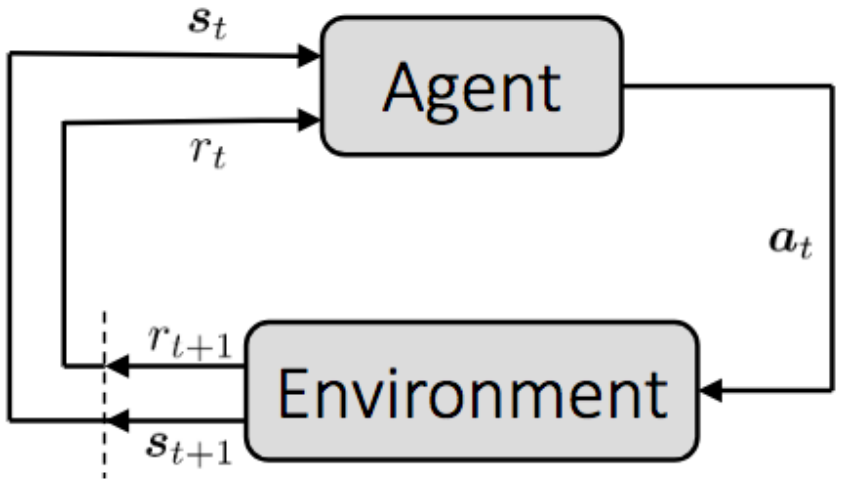


Basic model:

- **Agent** interacts with **Environment**, *e.g. a chess player interacts with the game board (states, rules) and the opponent*.
- **State** $\mathbf{s}_t \in \mathscr{S}$: The current state of the environment visible to the agent, *e.g. positions of all figures on the board*.
- **Action** $\mathbf{a}_t \in \mathscr{A}$: The current action taken by the agent, *e.g. move knight from B1 to C3*.
- **Reward** $r_t \in \mathbb{R}$: Immediate reward for the step $\mathbf{s}_t \xrightarrow{\mathbf{a}_t} \mathbf{s}_{t+1}$, e.g. *improvement of the board situation*.

Aim: Take actions $\mathbf{a}_t$ so as to maximize **long-term reward**.

Reinforcement Learning: Basic Terms

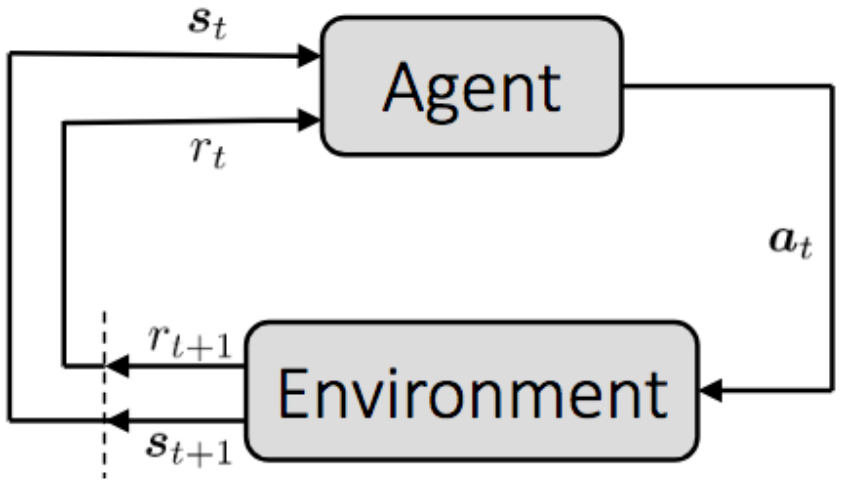


Basic model:

- **Agent** interacts with **Environment**, *e.g. a chess player interacts with the game board (states, rules) and the opponent*.
- **State** $\mathbf{s}_t \in \mathscr{S}$: The current state of the environment visible to the agent, *e.g. positions of all figures on the board*.
- **Action** $\mathbf{a}_t \in \mathscr{A}$: The current action taken by the agent, *e.g. move knight from B1 to C3*.
- **Reward** $r_t \in \mathbb{R}$: Immediate reward for the step $\mathbf{s}_t \xrightarrow{\mathbf{a}_t} \mathbf{s}_{t+1}$, e.g. *improvement of the board situation*.

Aim: Take actions $\mathbf{a}_t$ so as to maximize **long-term reward**.

Reinforcement Learning: Basic Terms

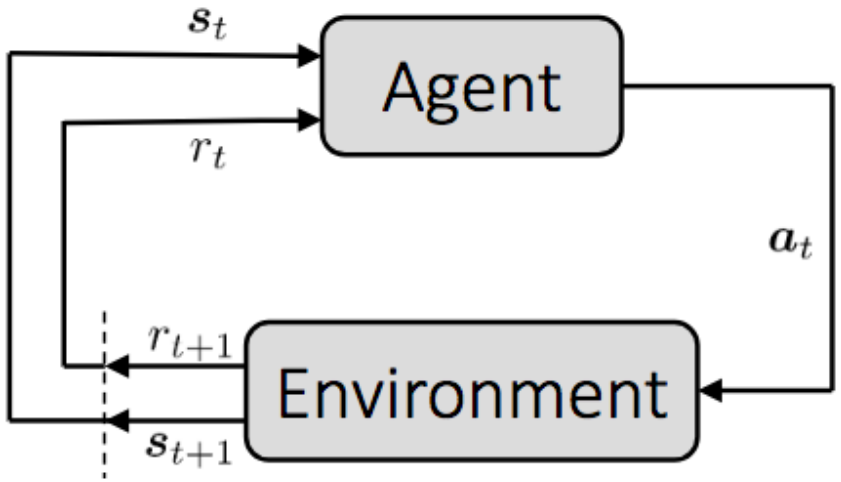


Basic model:

- **Agent** interacts with **Environment**, *e.g. a chess player interacts with the game board (states, rules) and the opponent*.
- **State** $\mathbf{s}_t \in \mathscr{S}$: The current state of the environment visible to the agent, *e.g. positions of all figures on the board*.
- **Action** $\mathbf{a}_t \in \mathscr{A}$: The current action taken by the agent, *e.g. move knight from B1 to C3*.
- **Reward** $r_t \in \mathbb{R}$: Immediate reward for the step $\mathbf{s}_t \xrightarrow{\mathbf{a}_t} \mathbf{s}_{t+1}$, e.g. *improvement of the board situation*.

Aim: Take actions $\mathbf{a}_t$ so as to maximize **long-term reward**.

Reinforcement Learning: Basic Terms

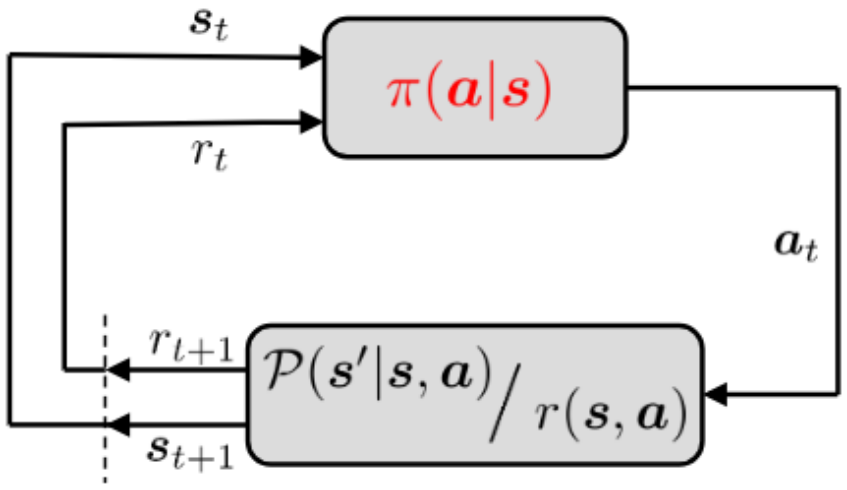


Basic model:

- **Agent** interacts with **Environment**, *e.g. a chess player interacts with the game board (states, rules) and the opponent*.
- **State** $\mathbf{s}_t \in \mathscr{S}$: The current state of the environment visible to the agent, *e.g. positions of all figures on the board*.
- **Action** $\mathbf{a}_t \in \mathscr{A}$: The current action taken by the agent, *e.g. move knight from B1 to C3*.
- **Reward** $r_t \in \mathbb{R}$: Immediate reward for the step $\mathbf{s}_t \xrightarrow{\mathbf{a}_t} \mathbf{s}_{t+1}$, e.g. *improvement of the board situation*.

Aim: Take actions $\mathbf{a}_t$ so as to maximize **long-term reward**.

Markov Decision Process (MDP)



Known **Inputs**:

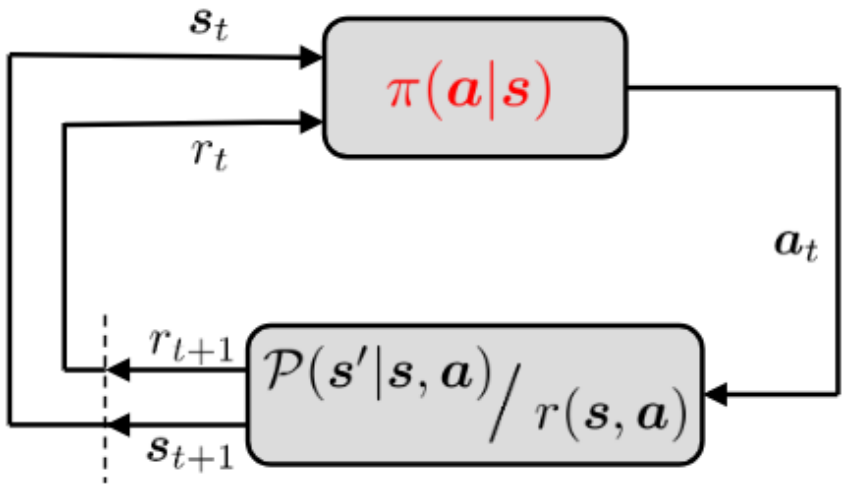
- Finite set of **discrete states** s_t and **actions** a_t.
- **Initial probabilities**: $\mu(s_0)$. **Transition dynamics** $P_{a_t}(s_{t+1} \mid s_t)$.
- **Reward** $r_t = R_{a_t}(s_t, s_{t+1})$ is paid after applying a_t in state s_t, leading to a transition to s_{t+1}.

Optimization problem: $\max_\pi J_\pi$ with:

- **Policy vector** $\pi(a \mid s)$: probabilities of choosing actions $a \sim \pi(a \mid s)$.
- **Long-time reward** using finite time horizon or infinite time horizon with discount factor γ:

$$J_\pi^\infty = \mathbb{E}_{\mu_0,P,\pi}\left[\sum_{t=0}^{\infty} \gamma^t r_t\right] \qquad J_\pi^{\mathrm{T}} = \mathbb{E}_{\mu_0,P,\pi}\left[\sum_{t=0}^{T} r_t\right]$$

Markov Decision Process (MDP)



Known **Inputs**:

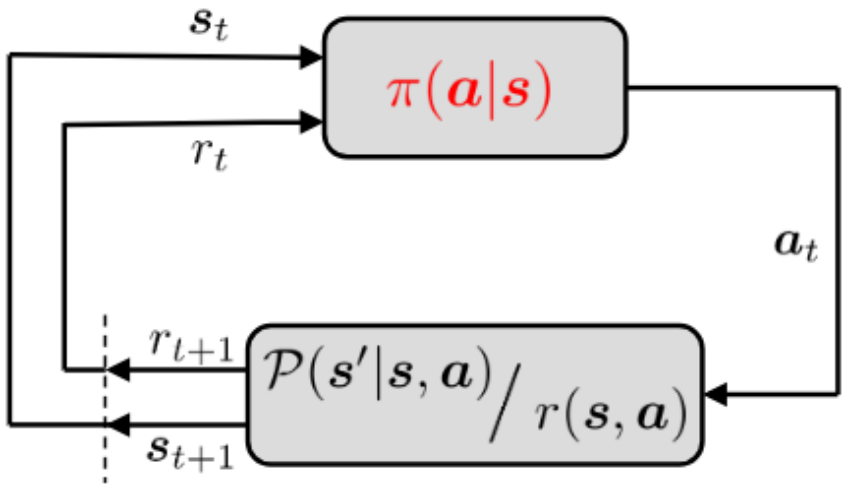
- Finite set of **discrete states** s_t and **actions** a_t.
- **Initial probabilities**: $\mu(s_0)$. **Transition dynamics** $P_{a_t}(s_{t+1} \mid s_t)$.
- **Reward** $r_t = R_{a_t}(s_t, s_{t+1})$ is paid after applying a_t in state s_t, leading to a transition to s_{t+1}.

Optimization problem: $\max_\pi J_\pi$ with:

- **Policy vector** $\pi(a \mid s)$: probabilities of choosing actions $a \sim \pi(a \mid s)$.
- **Long-time reward** using finite time horizon or infinite time horizon with discount factor γ:

$$J_\pi^\infty = \mathbb{E}_{\mu_0,P,\pi}\left[\sum_{t=0}^{\infty} \gamma^t r_t\right] \qquad J_\pi^{\mathrm{T}} = \mathbb{E}_{\mu_0,P,\pi}\left[\sum_{t=0}^{T} r_t\right]$$

Markov Decision Process (MDP)



Known **Inputs**:

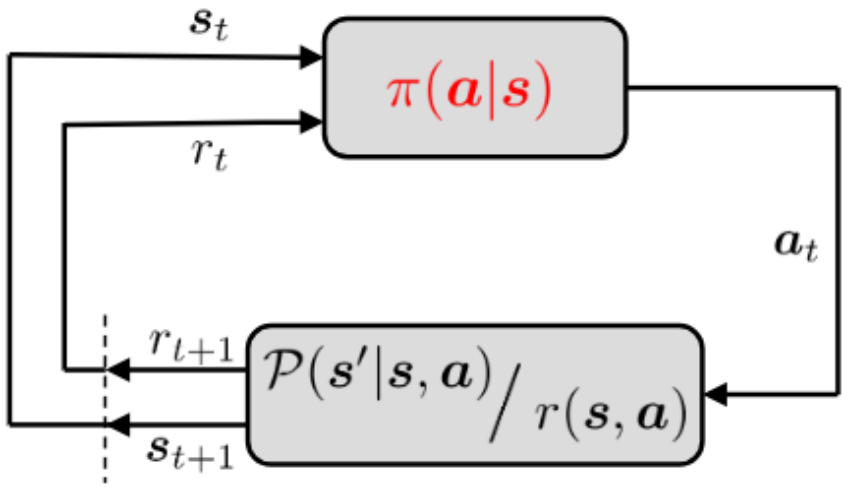
- Finite set of **discrete states** s_t and **actions** a_t.
- **Initial probabilities**: $\mu(s_0)$. **Transition dynamics** $P_{a_t}(s_{t+1} \mid s_t)$.
- **Reward** $r_t = R_{a_t}(s_t, s_{t+1})$ is paid after applying a_t in state s_t, leading to a transition to s_{t+1}.

Optimization problem: $\max_\pi J_\pi$ with:

- **Policy vector** $\pi(a \mid s)$: probabilities of choosing actions $a \sim \pi(a \mid s)$.
- **Long-time reward** using finite time horizon or infinite time horizon with discount factor γ:

$$J_\pi^\infty = \mathbb{E}_{\mu_0,P,\pi}\left[\sum_{t=0}^{\infty} \gamma^t r_t\right] \qquad J_\pi^{\mathrm{T}} = \mathbb{E}_{\mu_0,P,\pi}\left[\sum_{t=0}^{T} r_t\right]$$

Markov Decision Process (MDP)



Known **Inputs**:

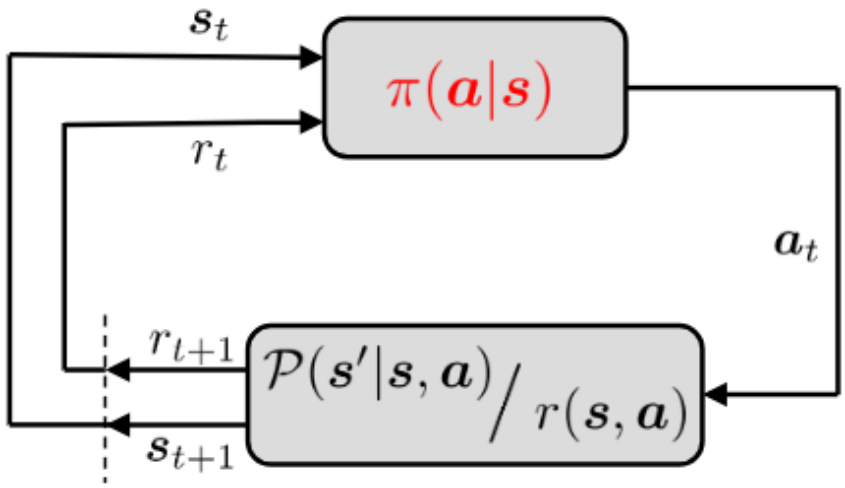
- Finite set of **discrete states** s_t and **actions** a_t.
- **Initial probabilities**: $\mu(s_0)$. **Transition dynamics** $P_{a_t}(s_{t+1} \mid s_t)$.
- **Reward** $r_t = R_{a_t}(s_t, s_{t+1})$ is paid after applying a_t in state s_t, leading to a transition to s_{t+1}.

Optimization problem: $\max_\pi J_\pi$ with:

- **Policy vector** $\pi(a \mid s)$: probabilities of choosing actions $a \sim \pi(a \mid s)$.
- **Long-time reward** using finite time horizon or infinite time horizon with discount factor γ:

$$J_\pi^\infty = \mathbb{E}_{\mu_0,P,\pi}\left[\sum_{t=0}^{\infty} \gamma^t r_t\right] \qquad J_\pi^T = \mathbb{E}_{\mu_0,P,\pi}\left[\sum_{t=0}^{T} r_t\right]$$

Markov Decision Process (MDP)



Known **Inputs**:

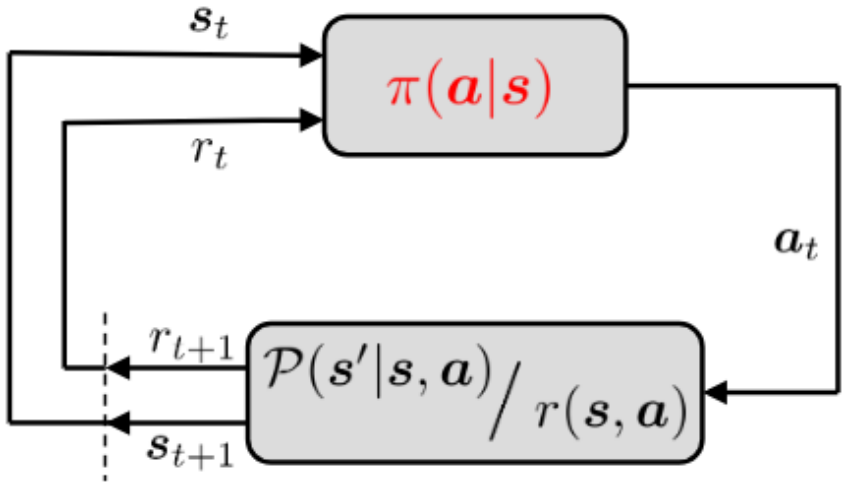
- Finite set of **discrete states** s_t and **actions** a_t.
- **Initial probabilities**: $\mu(s_0)$. **Transition dynamics** $P_{a_t}(s_{t+1} \mid s_t)$.
- **Reward** $r_t = R_{a_t}(s_t, s_{t+1})$ is paid after applying a_t in state s_t, leading to a transition to s_{t+1}.

Optimization problem: $\max_\pi J_\pi$ with:

- **Policy vector** $\pi(a \mid s)$: probabilities of choosing actions $a \sim \pi(a \mid s)$.
- **Long-time reward** using finite time horizon or infinite time horizon with discount factor γ:

$$J_\pi^\infty = \mathbb{E}_{\mu_0,P,\pi}\left[\sum_{t=0}^{\infty} \gamma^t r_t\right] \qquad J_\pi^{\mathrm{T}} = \mathbb{E}_{\mu_0,P,\pi}\left[\sum_{t=0}^{T} r_t\right]$$

Markov Decision Process (MDP)



Solution with **dynamic programming**: Optimal solution is a policy that assigns a best action to every state:

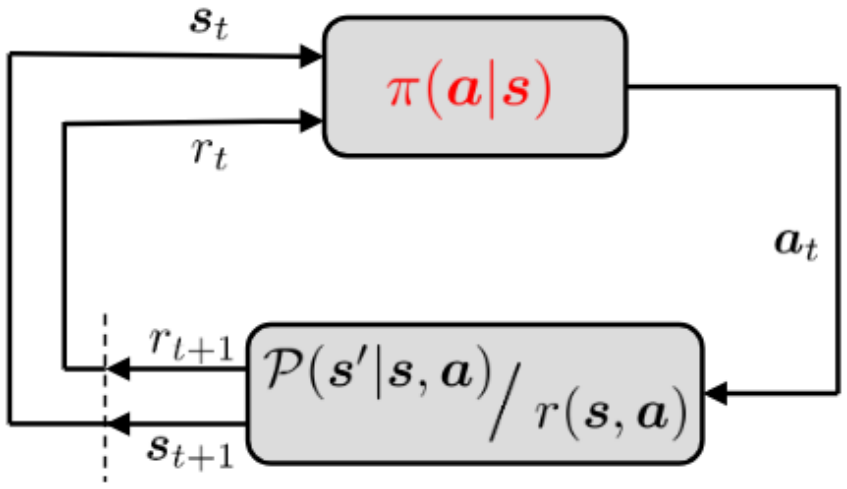
$$s \rightarrow \hat{a}(s)$$

Solution is found by iterating two equations that update the best action $\hat{a}(s)$ and the estimated value of a state s, $V(s)$ (Bellmann 1957):

$$\hat{a}(s) \leftarrow \arg\max_a \left\{ \sum_{s'} P_a(s' \mid s) \left(R_a(s,s') + \gamma V(s') \right) \right\}$$

$$V(s) \leftarrow \sum_{s'} P_{\hat{a}(s)}(s' \mid s) \left(R_{\hat{a}(s)}(s,s') + \gamma V(s') \right)$$

Markov Decision Process (MDP)



Solution with **dynamic programming**: Optimal solution is a policy that assigns a best action to every state:

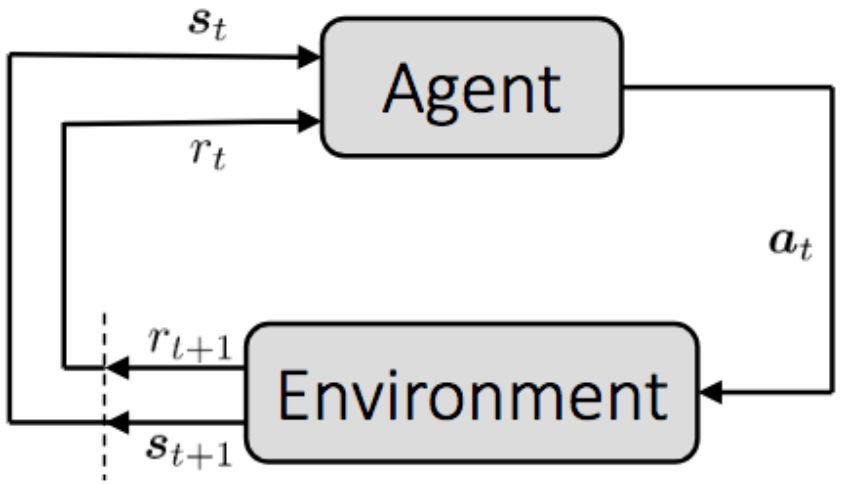
$$s \rightarrow \hat{a}(s)$$

Solution is found by iterating two equations that update the best action $\hat{a}(s)$ and the estimated value of a state s, $V(s)$ (Bellmann 1957):

$$\hat{a}(s) \leftarrow \arg\max_a \left\{ \sum_{s'} P_a(s' \mid s) \left(R_a(s,s') + \gamma V(s') \right) \right\}$$

$$V(s) \leftarrow \sum_{s'} P_{\hat{a}(s)}(s' \mid s) \left(R_{\hat{a}(s)}(s,s') + \gamma V(s') \right)$$

Reinforcement learning

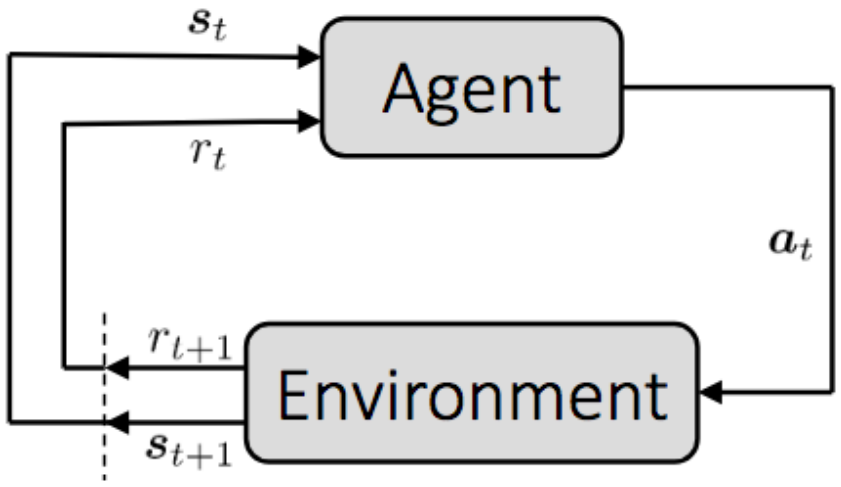


- **Reinforcement learning**: Probabilities or rewards are unknown.
- **Value-based RL / Q learning**: Define state-action quality function:

$$Q(\mathbf{s},\mathbf{a}) = \sum_{s'} P_{\mathbf{a}}(\mathbf{s}' \mid \mathbf{s})(R_{\mathbf{a}}(\mathbf{s},\mathbf{s}') + \gamma V(\mathbf{s}'))$$

  and then derives a policy selection probability vector $\pi(\mathbf{a} \mid \mathbf{s})$, e.g.:
  - For competitive performance: $\pi^* = \arg\max_{\mathbf{a}} Q(\mathbf{s},\mathbf{a})$
  - For exploring the search tree: $\pi(\mathbf{a} \mid \mathbf{s}) = \frac{e^{Q(\mathbf{s},\mathbf{a})}}{\sum_{\mathbf{a}'} e^{Q(\mathbf{s},\mathbf{a}')}}$
- **Policy-search RL**: Parametric policy $\pi(\mathbf{a} \mid \mathbf{s}; \theta)$. Optimize θ.
- Many modern RL techniques contain aspects of both, e.g. AlphaGo.

Reinforcement learning

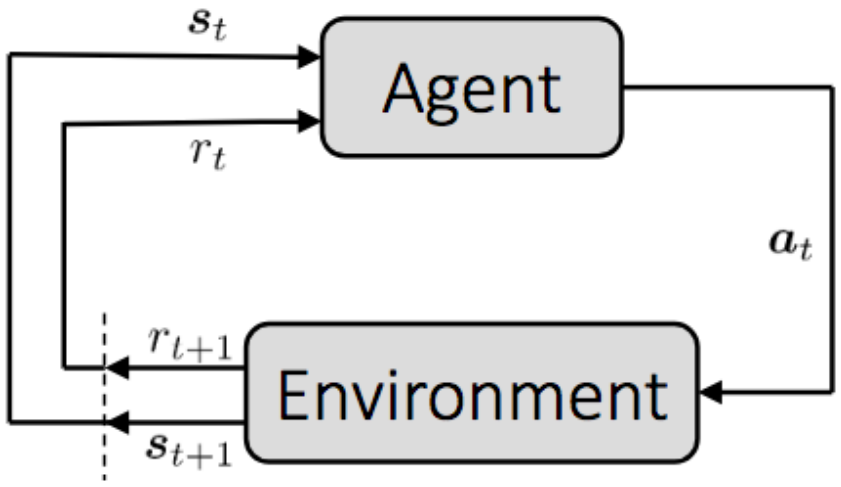


- **Reinforcement learning**: Probabilities or rewards are unknown.
- **Value-based RL** / **Q learning**: Define state-action quality function:

$$Q(\mathbf{s},\mathbf{a}) = \sum_{s'} P_{\mathbf{a}}(\mathbf{s}' \mid \mathbf{s})(R_{\mathbf{a}}(\mathbf{s},\mathbf{s}') + \gamma V(\mathbf{s}'))$$

  and then derives a policy selection probability vector $\pi(\mathbf{a} \mid \mathbf{s})$, e.g.:
  - For competitive performance: $\pi^* = \arg\max_{\mathbf{a}} Q(\mathbf{s},\mathbf{a})$
  - For exploring the search tree: $\pi(\mathbf{a} \mid \mathbf{s}) = \frac{e^{Q(\mathbf{s},\mathbf{a})}}{\sum_{\mathbf{a}'} e^{Q(\mathbf{s},\mathbf{a}')}}$
- **Policy-search RL**: Parametric policy $\pi(\mathbf{a} \mid \mathbf{s}; \theta)$. Optimize θ.
- Many modern RL techniques contain aspects of both, e.g. AlphaGo.

Reinforcement learning

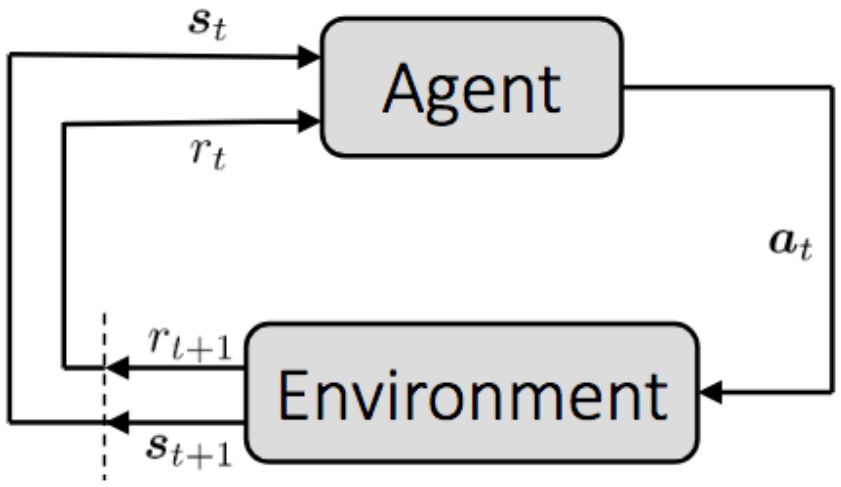


- **Reinforcement learning**: Probabilities or rewards are unknown.
- **Value-based RL** / **Q learning**: Define state-action quality function:

$$Q(\mathbf{s},\mathbf{a}) = \sum_{s'} P_{\mathbf{a}}(\mathbf{s}' \mid \mathbf{s})(R_{\mathbf{a}}(\mathbf{s},\mathbf{s}') + \gamma V(\mathbf{s}'))$$

  and then derives a policy selection probability vector $\pi(\mathbf{a} \mid \mathbf{s})$, e.g.:
  - For competitive performance: $\pi^* = \arg\max_{\mathbf{a}} Q(\mathbf{s},\mathbf{a})$
  - For exploring the search tree: $\pi(\mathbf{a} \mid \mathbf{s}) = \frac{e^{Q(\mathbf{s},\mathbf{a})}}{\sum_{\mathbf{a}'} e^{Q(\mathbf{s},\mathbf{a}')}}$
- **Policy-search RL**: Parametric policy $\pi(\mathbf{a} \mid \mathbf{s};\theta)$. Optimize θ.
- Many modern RL techniques contain aspects of both, e.g. AlphaGo.

Reinforcement learning

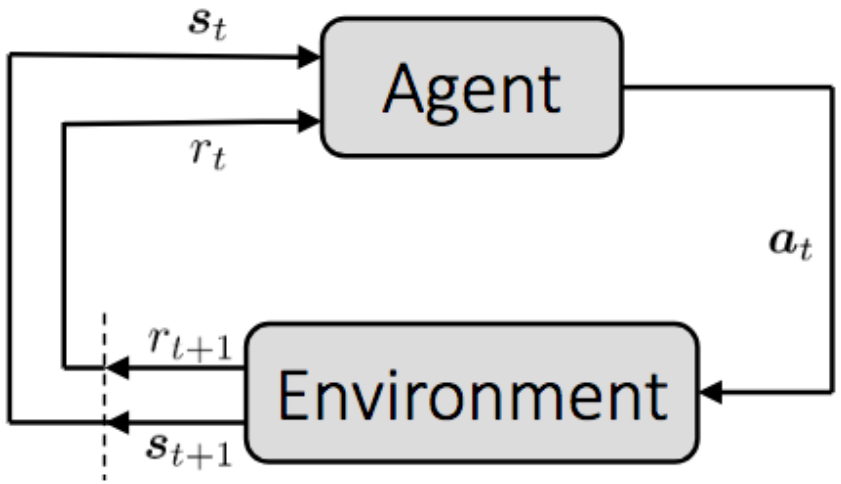


- **Reinforcement learning**: Probabilities or rewards are unknown.
- **Value-based RL / Q learning**: Define state-action quality function:

$$Q(\mathbf{s},\mathbf{a}) = \sum_{s'} P_{\mathbf{a}}(\mathbf{s}' \mid \mathbf{s})(R_{\mathbf{a}}(\mathbf{s},\mathbf{s}') + \gamma V(\mathbf{s}'))$$

  and then derives a policy selection probability vector $\pi(\mathbf{a} \mid \mathbf{s})$, e.g.:
  - For competitive performance: $\pi^* = \arg\max_{\mathbf{a}} Q(\mathbf{s},\mathbf{a})$
  - For exploring the search tree: $\pi(\mathbf{a} \mid \mathbf{s}) = \frac{e^{Q(\mathbf{s},\mathbf{a})}}{\sum_{\mathbf{a}'} e^{Q(\mathbf{s},\mathbf{a}')}}$
- **Policy-search RL**: Parametric policy $\pi(\mathbf{a} \mid \mathbf{s};\theta)$. Optimize θ.
- Many modern RL techniques contain aspects of both, e.g. AlphaGo.

Reinforcement learning

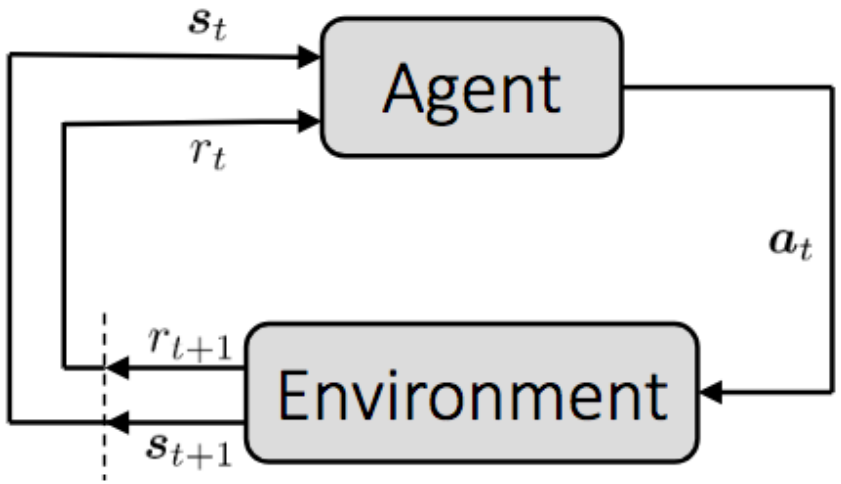


- **Reinforcement learning**: Probabilities or rewards are unknown.
- **Value-based RL / Q learning**: Define state-action quality function:

$$Q(\mathbf{s},\mathbf{a}) = \sum_{s'} P_{\mathbf{a}}(\mathbf{s}' \mid \mathbf{s})(R_{\mathbf{a}}(\mathbf{s},\mathbf{s}') + \gamma V(\mathbf{s}'))$$

  and then derives a policy selection probability vector $\pi(\mathbf{a} \mid \mathbf{s})$, e.g.:
  - For competitive performance: $\pi^* = \arg\max_{\mathbf{a}} Q(\mathbf{s},\mathbf{a})$
  - For exploring the search tree: $\pi(\mathbf{a} \mid \mathbf{s}) = \frac{e^{Q(\mathbf{s},\mathbf{a})}}{\sum_{\mathbf{a}'} e^{Q(\mathbf{s},\mathbf{a}')}}$
- **Policy-search RL**: Parametric policy $\pi(\mathbf{a} \mid \mathbf{s}; \theta)$. Optimize θ.
- Many modern RL techniques contain aspects of both, e.g. AlphaGo.

Reinforcement learning

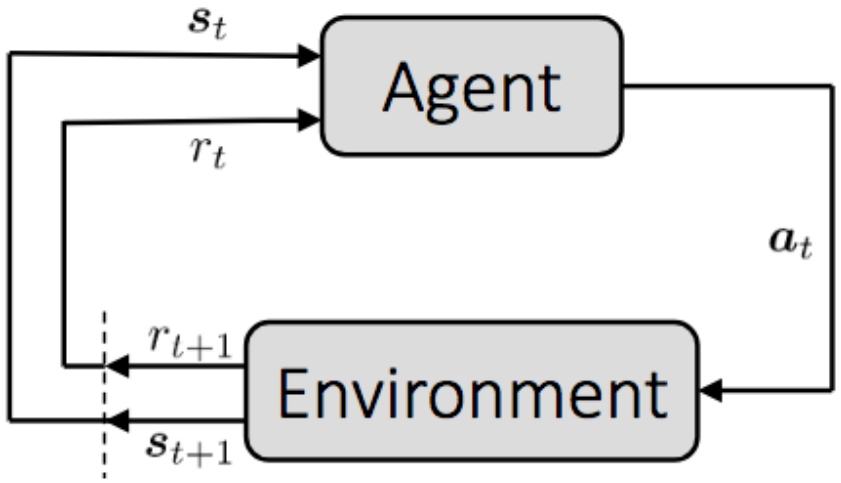


- **Reinforcement learning**: Probabilities or rewards are unknown.
- **Value-based RL / Q learning**: Define state-action quality function:

$$Q(\mathbf{s},\mathbf{a}) = \sum_{s'} P_{\mathbf{a}}(\mathbf{s}' \mid \mathbf{s})(R_{\mathbf{a}}(\mathbf{s},\mathbf{s}') + \gamma V(\mathbf{s}'))$$

  and then derives a policy selection probability vector $\pi(\mathbf{a} \mid \mathbf{s})$, e.g.:
  - For competitive performance: $\pi^* = \arg\max_{\mathbf{a}} Q(\mathbf{s},\mathbf{a})$
  - For exploring the search tree: $\pi(\mathbf{a} \mid \mathbf{s}) = \frac{e^{Q(\mathbf{s},\mathbf{a})}}{\sum_{\mathbf{a}'} e^{Q(\mathbf{s},\mathbf{a}')}}$
- **Policy-search RL**: Parametric policy $\pi(\mathbf{a} \mid \mathbf{s}; \theta)$. Optimize θ.
- Many modern RL techniques contain aspects of both, e.g. AlphaGo.

Reinforcement learning

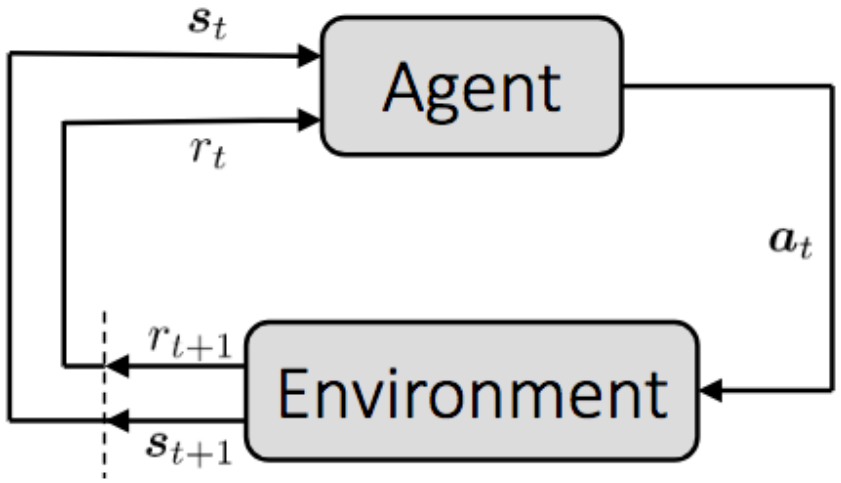


- **Probabilities** P or **rewards** r are **unknown** $\rightarrow$ How do we compute:

$$J_\pi^\infty = \mathbb{E}_{\mu_0,P,\pi}\left[\sum_{t=0}^{\infty}\gamma^t r_t\right] \qquad Q(\mathbf{s},\mathbf{a}) = \sum_{s'} P_{\mathbf{a}}(\mathbf{s},\mathbf{s}')(R_{\mathbf{a}}(\mathbf{s},\mathbf{s}') + \gamma V(\mathbf{s}'))$$

- **Monte-Carlo search**: Sample J_π^∞ or Q by generating many transition pairs $\mathbf{s}_t \overset{\mathbf{a}_t}{\rightarrow} \mathbf{s}_{t+\tau}$.
- **Playouts**: Start in an initial random state and simulate moves until T steps have been made or a terminal state has been reached.
- **Difficulties**: Playouts are expensive and search tree expands with branching factor n (Number of actions) in every step. $\rightarrow$ How do we sample **relevant** playouts likely to lead to winning strategies?

Reinforcement learning

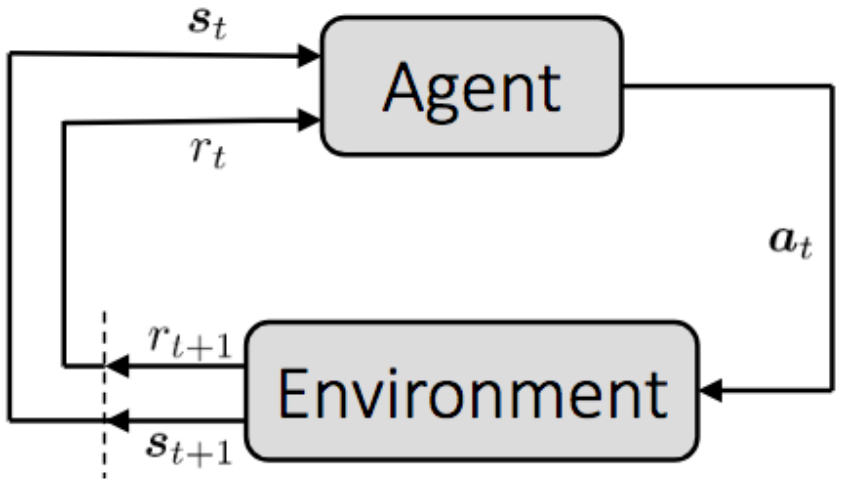


- **Probabilities** P or **rewards** r are **unknown** $\rightarrow$ How do we compute:

$$J_\pi^\infty = \mathbb{E}_{\mu_0,P,\pi}\left[\sum_{t=0}^{\infty}\gamma^t r_t\right] \qquad Q(\mathbf{s},\mathbf{a}) = \sum_{s'} P_{\mathbf{a}}(\mathbf{s},\mathbf{s}')(R_{\mathbf{a}}(\mathbf{s},\mathbf{s}') + \gamma V(\mathbf{s}'))$$

- **Monte-Carlo search**: Sample J_π^∞ or Q by generating many transition pairs $\mathbf{s}_t \xrightarrow{\mathbf{a}_t} \mathbf{s}_{t+\tau}$.
- **Playouts**: Start in an initial random state and simulate moves until T steps have been made or a terminal state has been reached.
- **Difficulties**: Playouts are expensive and search tree expands with branching factor n (Number of actions) in every step. $\rightarrow$ How do we sample **relevant** playouts likely to lead to winning strategies?

Reinforcement learning

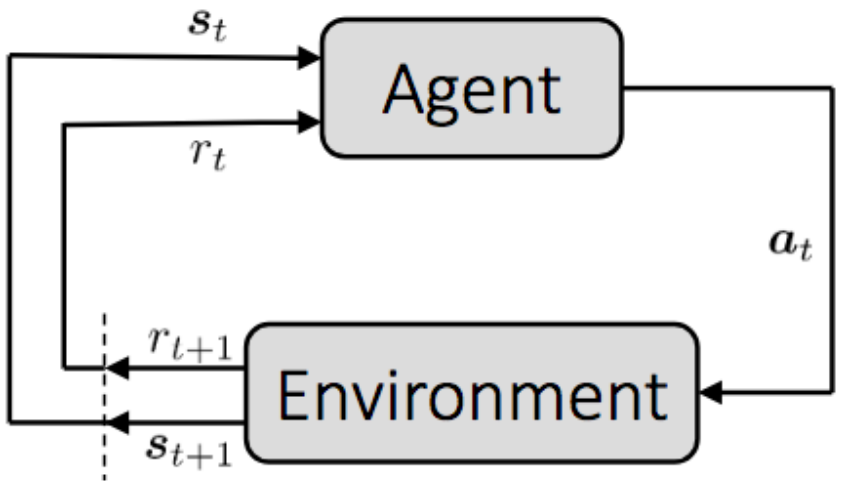


- **Probabilities** P or **rewards** r are **unknown** $\rightarrow$ How do we compute:

$$J_\pi^\infty = \mathbb{E}_{\mu_0,P,\pi}\left[\sum_{t=0}^{\infty}\gamma^t r_t\right] \qquad Q(\mathbf{s},\mathbf{a}) = \sum_{s'} P_{\mathbf{a}}(\mathbf{s},\mathbf{s}')(R_{\mathbf{a}}(\mathbf{s},\mathbf{s}') + \gamma V(\mathbf{s}'))$$

- **Monte-Carlo search**: Sample J_π^∞ or Q by generating many transition pairs $\mathbf{s}_t \xrightarrow{\mathbf{a}_t} \mathbf{s}_{t+\tau}$.
- **Playouts**: Start in an initial random state and simulate moves until T steps have been made or a terminal state has been reached.
- **Difficulties**: Playouts are expensive and search tree expands with branching factor n (Number of actions) in every step. $\rightarrow$ How do we sample **relevant** playouts likely to lead to winning strategies?

Reinforcement learning



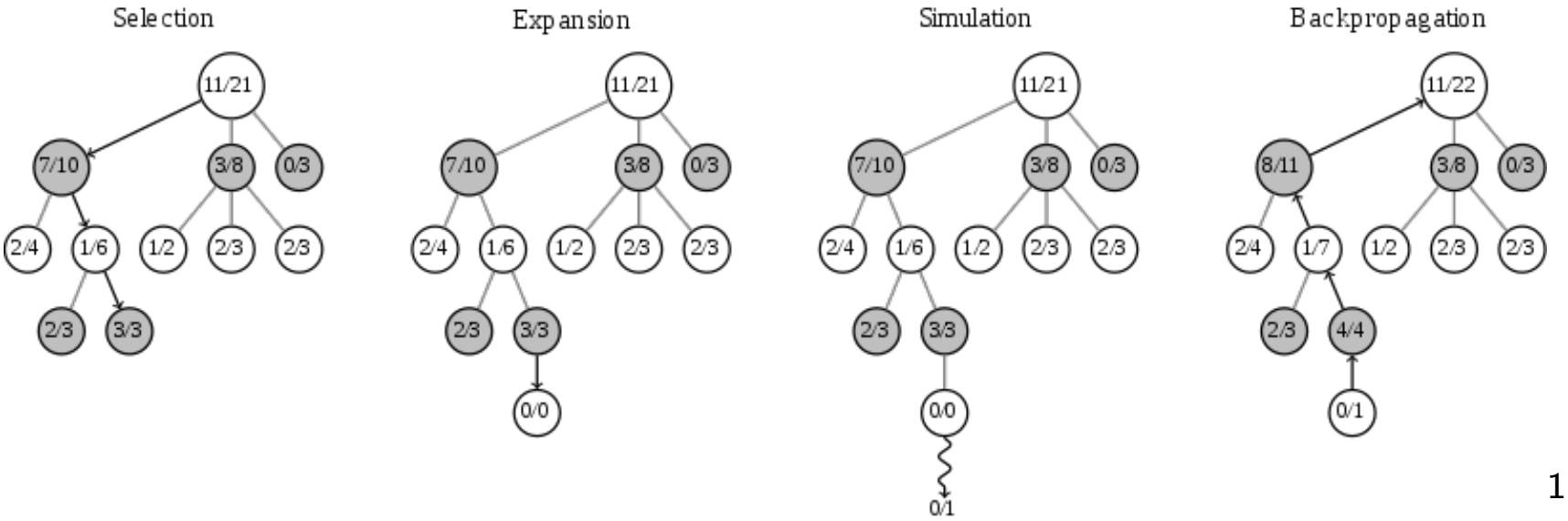
- **Probabilities** P or **rewards** r are **unknown** $\rightarrow$ How do we compute:

$$J_\pi^\infty = \mathbb{E}_{\mu_0,P,\pi}\left[\sum_{t=0}^{\infty}\gamma^t r_t\right] \qquad Q(\mathbf{s},\mathbf{a}) = \sum_{s'} P_{\mathbf{a}}(\mathbf{s},\mathbf{s}')(R_{\mathbf{a}}(\mathbf{s},\mathbf{s}') + \gamma V(\mathbf{s}'))$$

- **Monte-Carlo search**: Sample J_π^∞ or Q by generating many transition pairs $\mathbf{s}_t \xrightarrow{\mathbf{a}_t} \mathbf{s}_{t+\tau}$.
- **Playouts**: Start in an initial random state and simulate moves until T steps have been made or a terminal state has been reached.
- **Difficulties**: Playouts are expensive and search tree expands with branching factor n (Number of actions) in every step. $\rightarrow$ How do we sample **relevant** playouts likely to lead to winning strategies?

Monte Carlo Tree Search (MCTS)

Purpose: Faster/Better Playouts



[1]

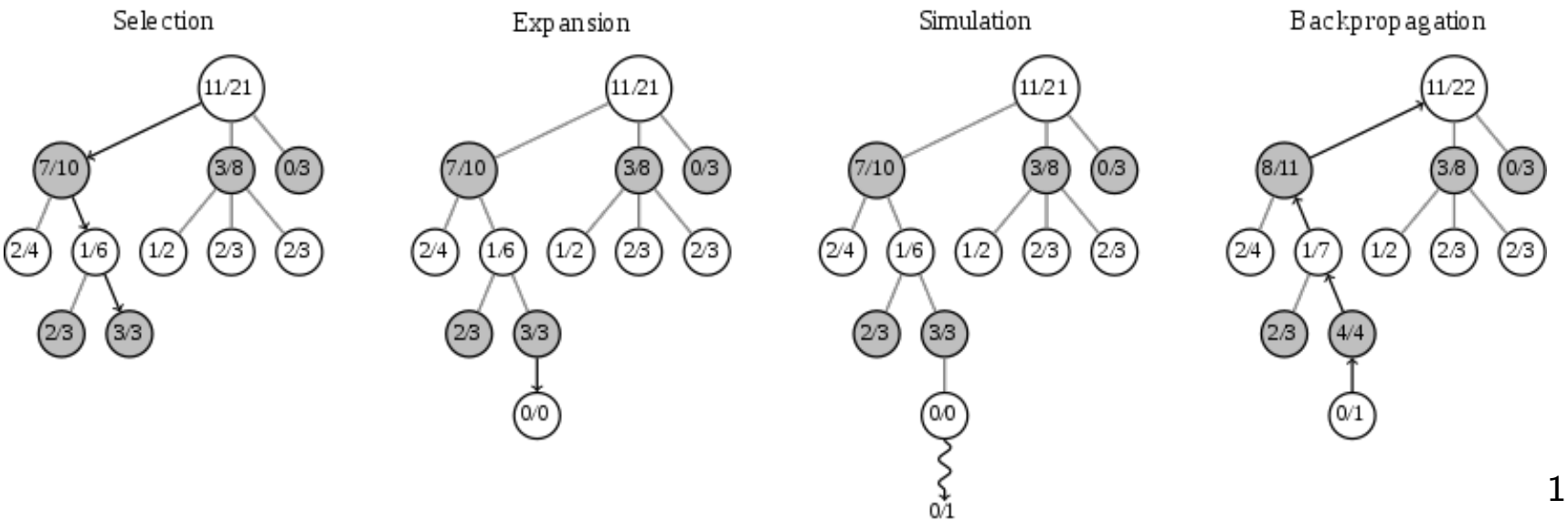
Number of games won / number of games played by black or white player.

- **Aim**: analyse most promising moves of a game by expanding the search tree based on random sampling of the search space.
- **Playouts**: in each playout, the game is played to the end or to the stopping node by selecting moves at random.
- **Weight nodes** in game tree with final game result of each playout $\rightarrow$ better nodes are more likely to be chosen in future playouts.

[1] From https://en.wikipedia.org/wiki/Monte_Carlo_tree_search

Monte Carlo Tree Search (MCTS)

Purpose: Faster/Better Playouts



[1]

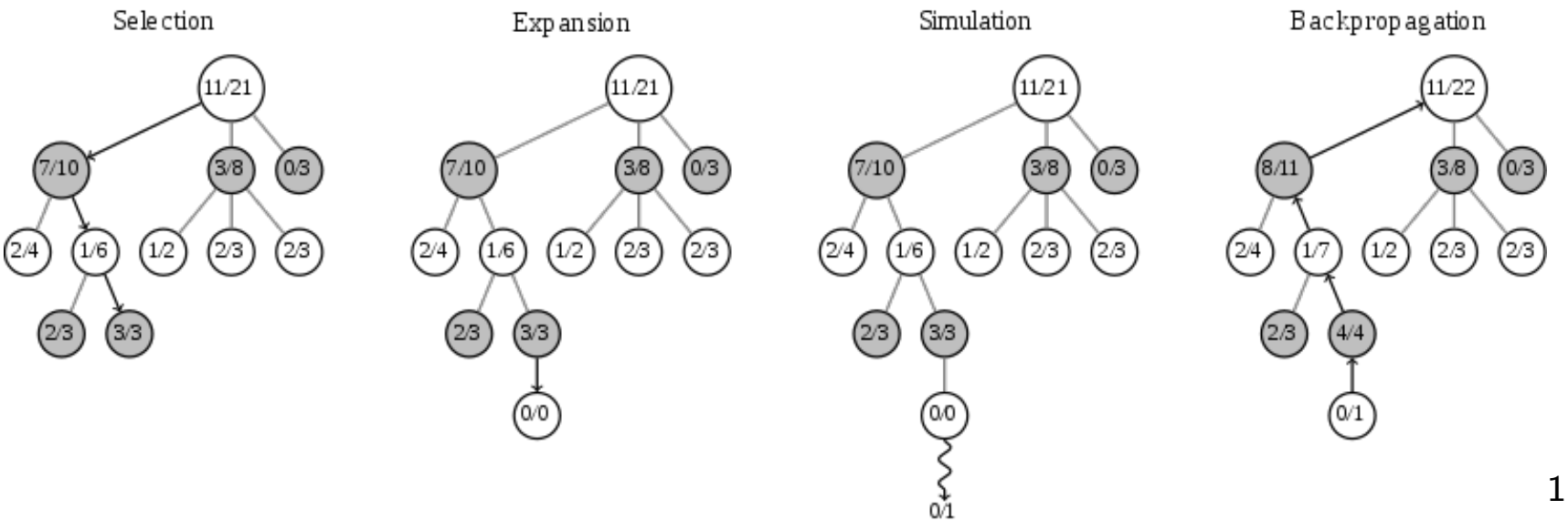
Number of games won / number of games played by black or white player.

- **Aim**: analyse most promising moves of a game by expanding the search tree based on random sampling of the search space.
- **Playouts**: in each playout, the game is played to the end or to the stopping node by selecting moves at random.
- **Weight nodes** in game tree with final game result of each playout → better nodes are more likely to be chosen in future playouts.

[1] From https://en.wikipedia.org/wiki/Monte_Carlo_tree_search

Monte Carlo Tree Search (MCTS)

Purpose: Faster/Better Playouts



[1]

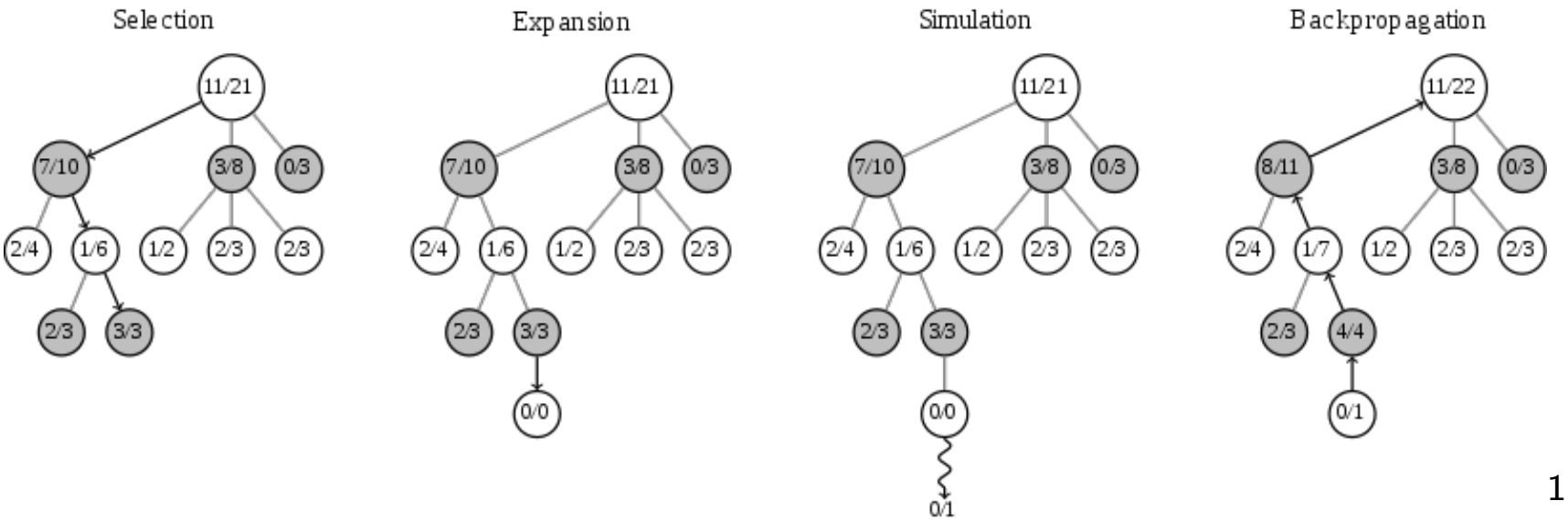
Number of games won / number of games played by black or white player.

- **Aim**: analyse most promising moves of a game by expanding the search tree based on random sampling of the search space.
- **Playouts**: in each playout, the game is played to the end or to the stopping node by selecting moves at random.
- **Weight nodes** in game tree with final game result of each playout $\rightarrow$ better nodes are more likely to be chosen in future playouts.

[1] From https://en.wikipedia.org/wiki/Monte_Carlo_tree_search

Monte Carlo Tree Search (MCTS)

Purpose: To avoid overfitting as a function of training time



[1]

Number of games won / number of games played by black or white player.

Steps of each MCTS round:

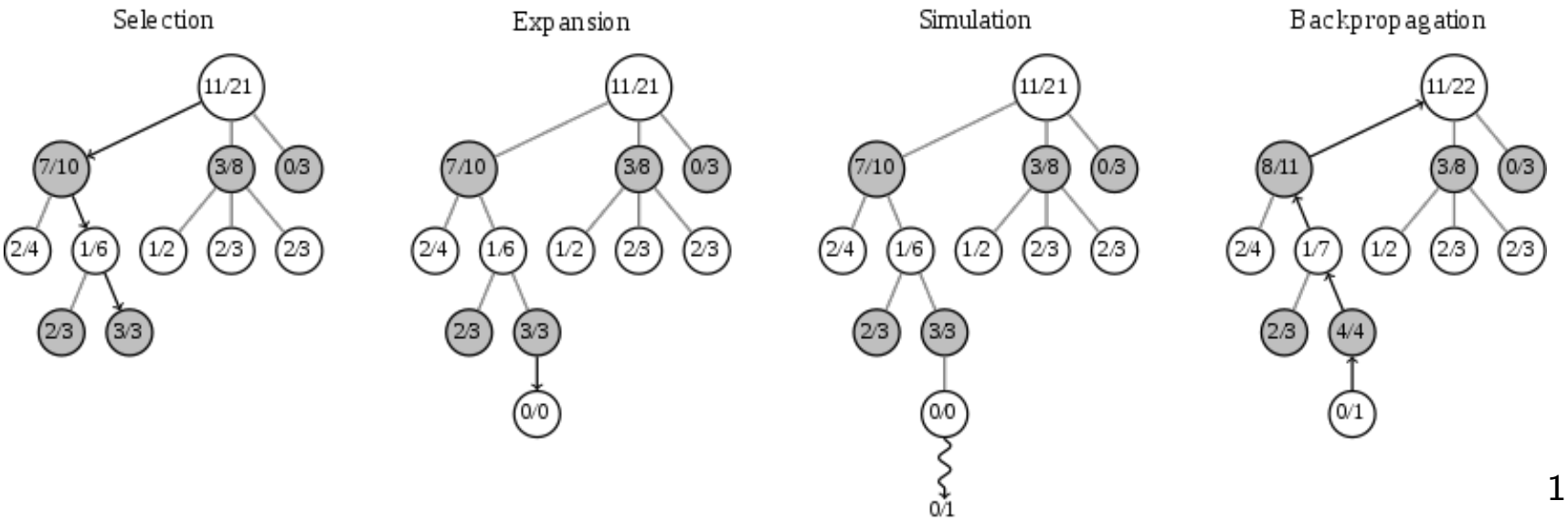
1. **Selection**: start from root R and select successive child nodes down to a leaf node L.
2. **Expansion**: unless L ends the game with a win/loss for either player, create child nodes and choose node C from one of them.
3. **Simulation**: play a random playout from node C.
4. **Backpropagation**: use the result of the playout to update information in the nodes on the path from C to R.

~~After that we make the move that~~ has the most simulation made.

[1] From https://en.wikipedia.org/wiki/Monte_Carlo_tree_search

Monte Carlo Tree Search (MCTS)

Purpose: To avoid overfitting as a function of training time



[1]

Number of games won / number of games played by black or white player.

Steps of each MCTS round:

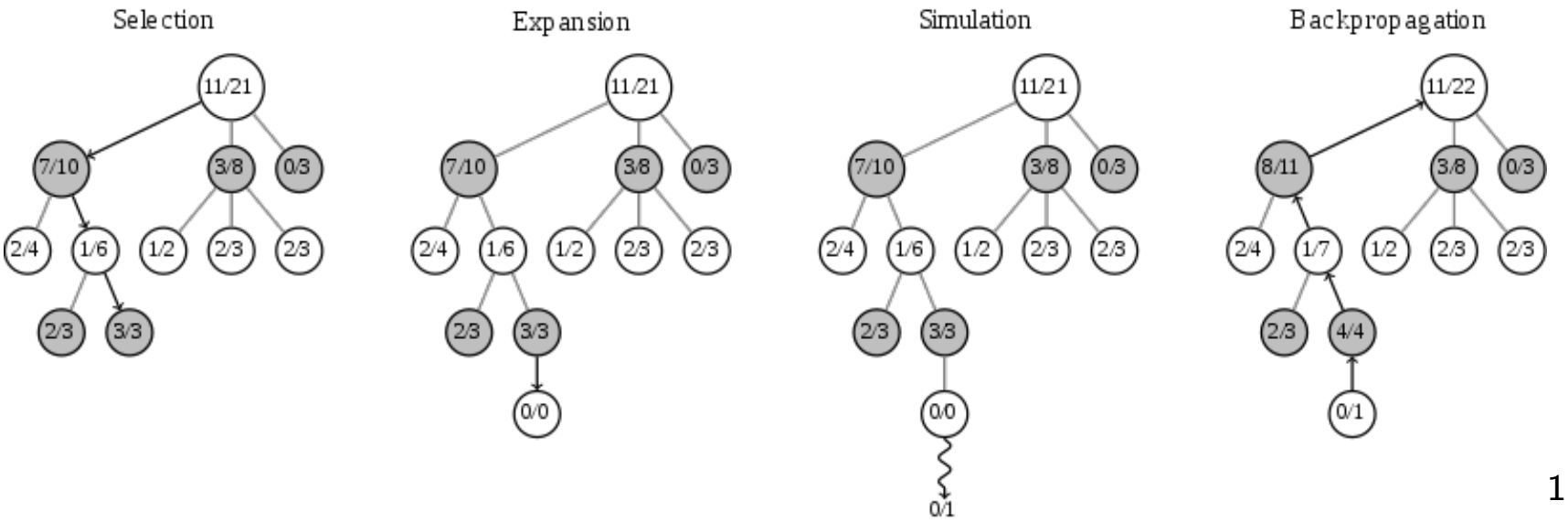
1. **Selection**: start from root R and select successive child nodes down to a leaf node L.
2. **Expansion**: unless L ends the game with a win/loss for either player, create child nodes and choose node C from one of them.
3. **Simulation**: play a random playout from node C.
4. **Backpropagation**: use the result of the playout to update information in the nodes on the path from C to R.

~~After that we make the move that~~ has the most simulation made.

[1] From https://en.wikipedia.org/wiki/Monte_Carlo_tree_search

Monte Carlo Tree Search (MCTS)

Purpose: To avoid overfitting as a function of training time



[1]

Number of games won / number of games played by black or white player.

Steps of each MCTS round:

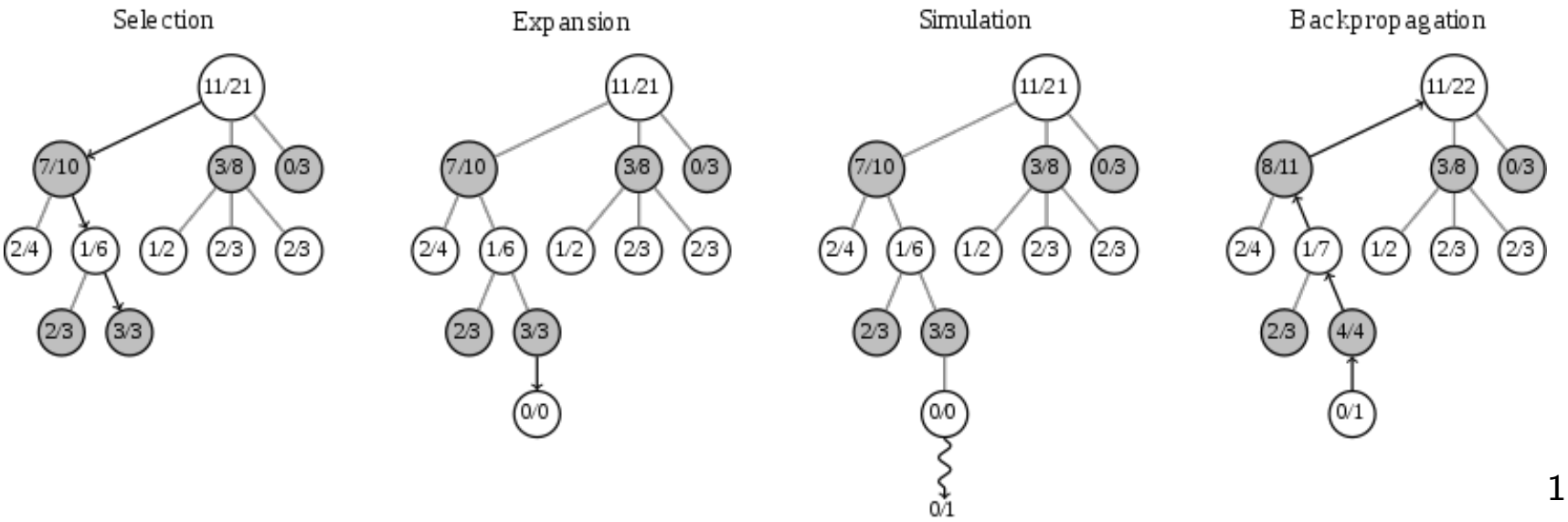
1. **Selection**: start from root R and select successive child nodes down to a leaf node L.
2. **Expansion**: unless L ends the game with a win/loss for either player, create child nodes and choose node C from one of them.
3. **Simulation**: play a random playout from node C.
4. **Backpropagation**: use the result of the playout to update information in the nodes on the path from C to R.

~~After that we make the move that~~ has the most simulation made.

[1] From https://en.wikipedia.org/wiki/Monte_Carlo_tree_search

Monte Carlo Tree Search (MCTS)

Purpose: To avoid overfitting as a function of training time



[1]

Number of games won / number of games played by black or white player.

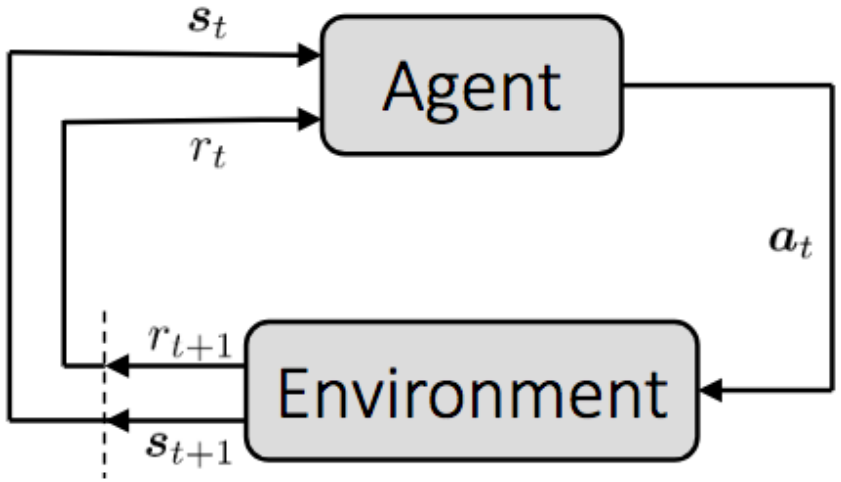
Steps of each MCTS round:

1. **Selection**: start from root R and select successive child nodes down to a leaf node L.
2. **Expansion**: unless L ends the game with a win/loss for either player, create child nodes and choose node C from one of them.
3. **Simulation**: play a random playout from node C.
4. **Backpropagation**: use the result of the playout to update information in the nodes on the path from C to R.

~~After that we make the move that~~ has the most simulation made.

[1] From https://en.wikipedia.org/wiki/Monte_Carlo_tree_search

Deep Reinforcement Learning



- **Policy network** Π: maps state to prior policy

$$s \to \pi = \Pi(s)$$

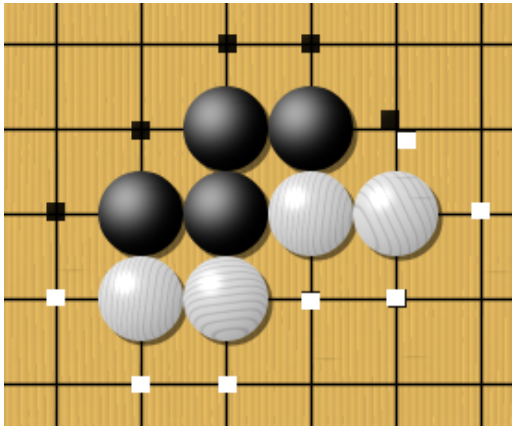
- **Value network** V: maps state to value

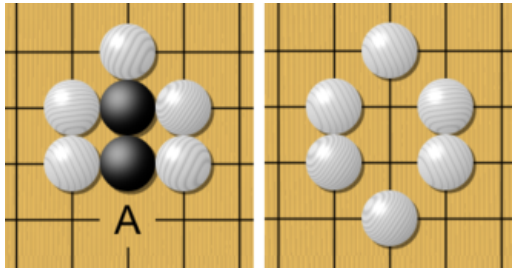
$$s \to v = V(s)$$

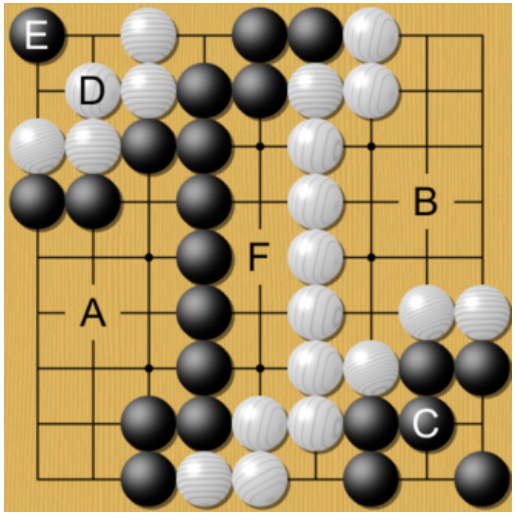
- **Iterate**:
  1. Generate **playouts**, i.e. apply Π and MCTS samples until game is won or a number of actions have been taken.
  2. Given the observed state-action sequences $s_t \xrightarrow{a_t} s_{t+1}$, **train** Π and V.

Alpha Go and Sons

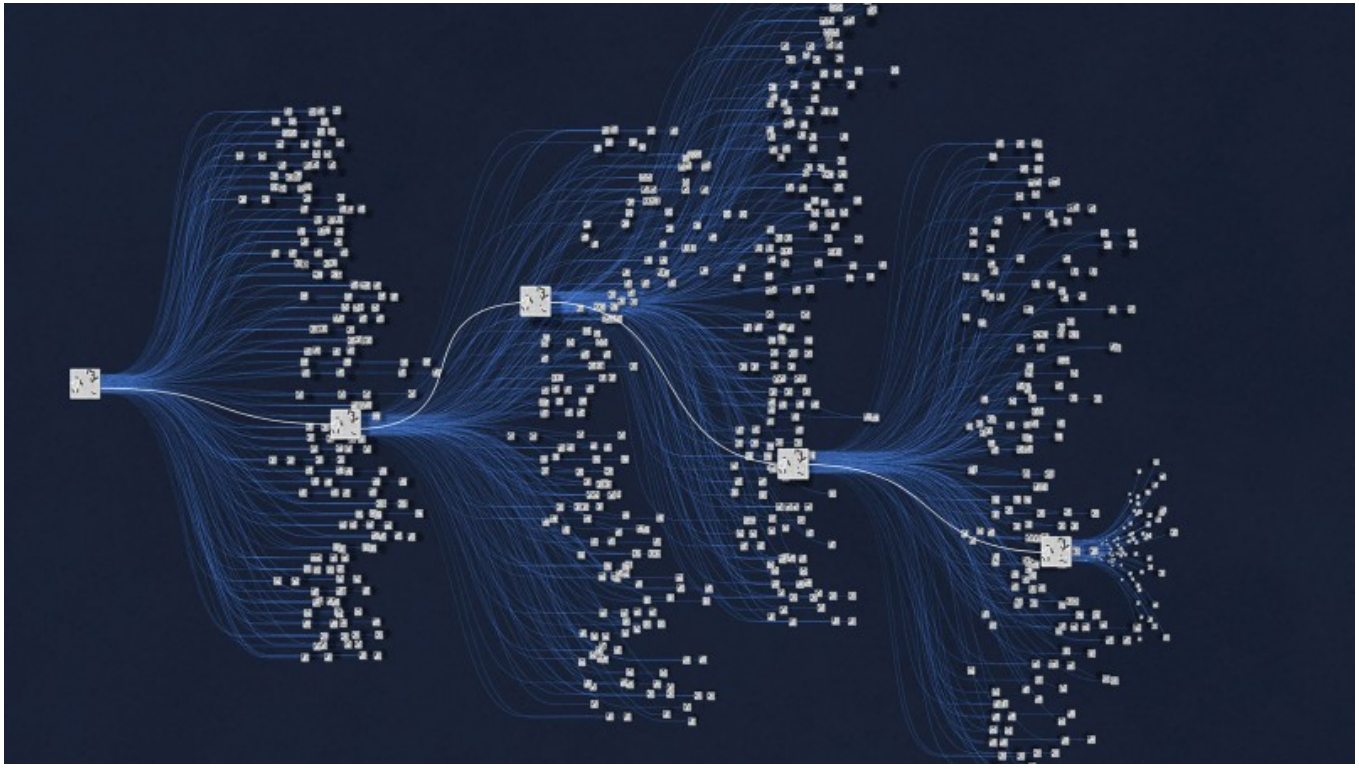
Go







Alpha Go and Sons



Alpha Go Zero (Silver et al., Nature 2017)

Self-play reinforcement learning

- **Self Play**: Create a training set
  - Best current player plays $25,000$ games against itself
  - Each move is made based on MCTS that is informed by a trainable policy/value network.
  - At each move, store the MCTS search probabilities.
  - For each game, store the winner $(+1, -1)$.
- **Train Network**: Optimize Π, V network weights
  - Sample mini-batch of $2,048$ positions from the last $500,000$ games.
  - Retrain Π network: minimize cross-entropy with π from MCTS
  - Retrain V network: minimize mean square error to actual winners.
- **Evaluate Network**: Test if we have a new champion
  - Play 400 games competitively between latest network and current best network.
  - Both Players use MCTS and their respective networks to select moves
  - Latest network is declared best player if it wins 55% of the games.

Alpha Go Zero (Silver et al., Nature 2017)

Self-play reinforcement learning

- **Self Play**: Create a training set
  - Best current player plays $25,000$ games against itself
  - Each move is made based on MCTS that is informed by a trainable policy/value network.
  - At each move, store the MCTS search probabilities.
  - For each game, store the winner $(+1, -1)$.
- **Train Network**: Optimize Π, V network weights
  - Sample mini-batch of $2,048$ positions from the last $500,000$ games.
  - Retrain Π network: minimize cross-entropy with π from MCTS
  - Retrain V network: minimize mean square error to actual winners.
- **Evaluate Network**: Test if we have a new champion
  - Play 400 games competitively between latest network and current best network.
  - Both Players use MCTS and their respective networks to select moves
  - Latest network is declared best player if it wins 55% of the games.

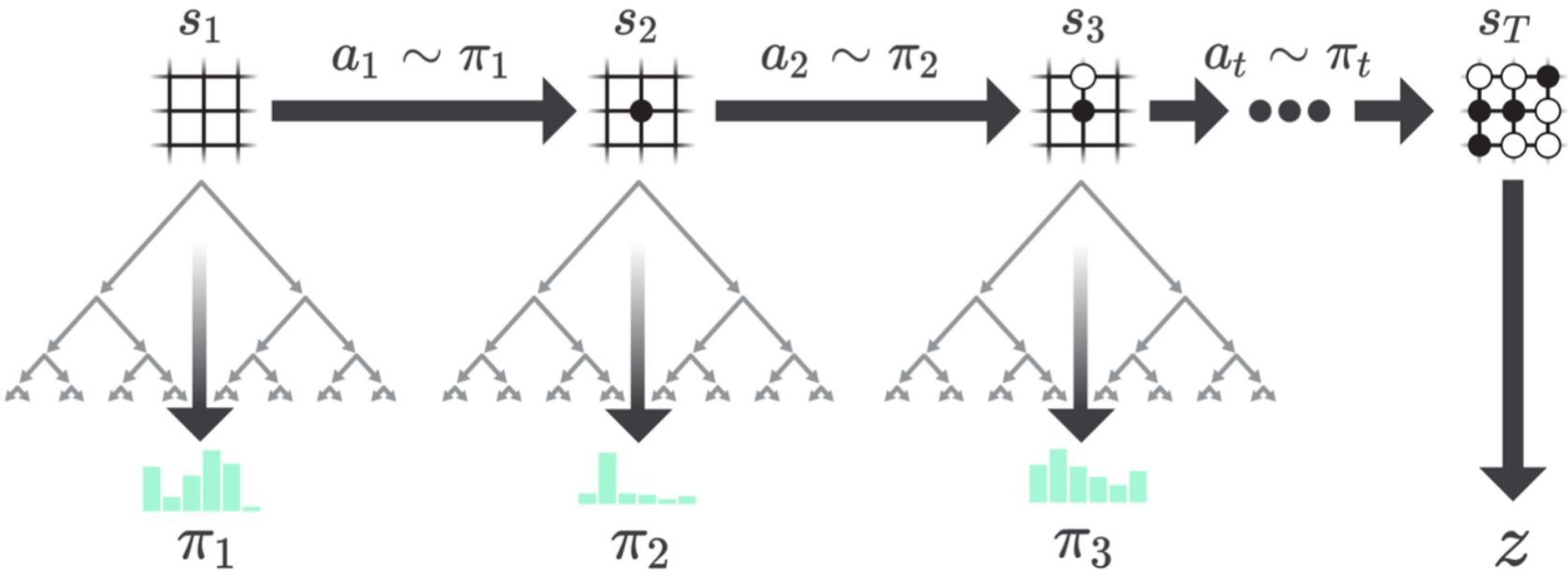
Alpha Go Zero (Silver et al., Nature 2017)

Self-play reinforcement learning

- **Self Play**: Create a training set
  - Best current player plays $25,000$ games against itself
  - Each move is made based on MCTS that is informed by a trainable policy/value network.
  - At each move, store the MCTS search probabilities.
  - For each game, store the winner $(+1, -1)$.
- **Train Network**: Optimize Π, V network weights
  - Sample mini-batch of $2,048$ positions from the last $500,000$ games.
  - Retrain Π network: minimize cross-entropy with π from MCTS
  - Retrain V network: minimize mean square error to actual winners.
- **Evaluate Network**: Test if we have a new champion
  - Play 400 games competitively between latest network and current best network.
  - Both Players use MCTS and their respective networks to select moves
  - Latest network is declared best player if it wins 55% of the games.

Alpha Go Zero (Silver et al., Nature 2017)

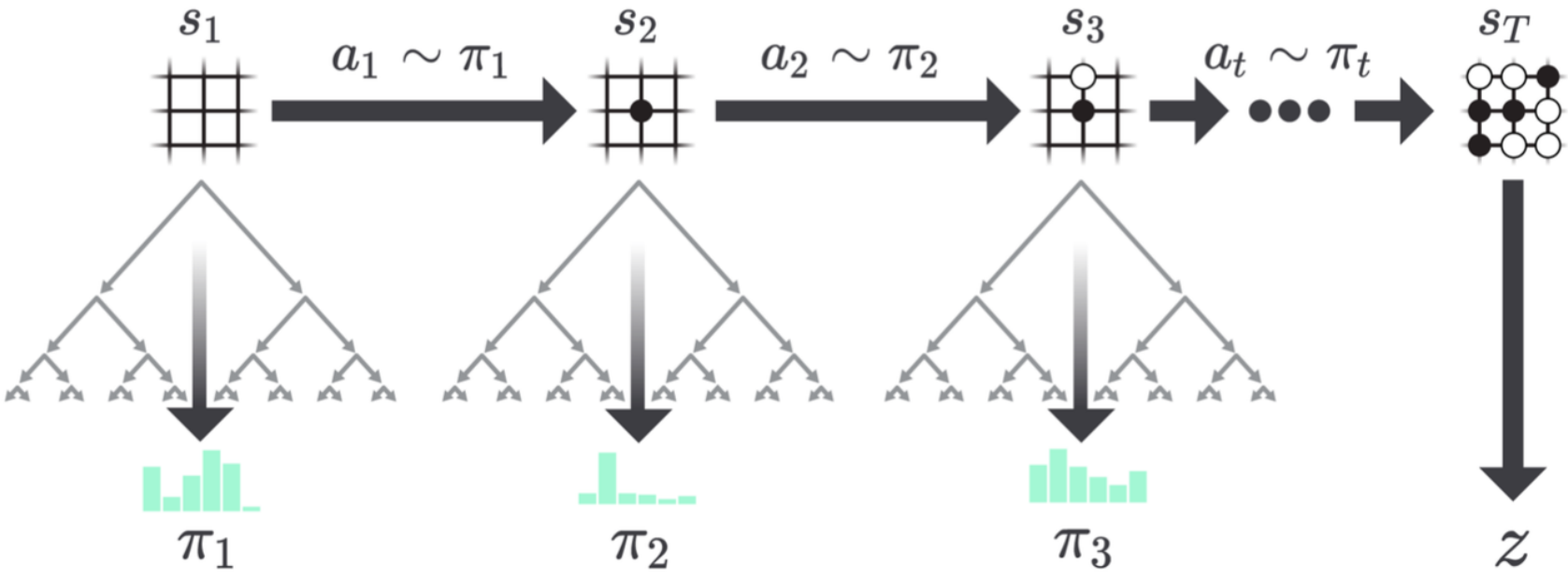
Self-play reinforcement learning



- Program plays games against itself – state sequences: $s_1, \ldots, s_T$.
- In each position s_t, execute Monte-Carlo tree search (MCTS) using the current policy-value network Π, V and compute search probabilities π_t.
- Select move $a_t \sim \pi_t$.
- Store game winner z defined by terminal position s_T.

Alpha Go Zero (Silver et al., Nature 2017)

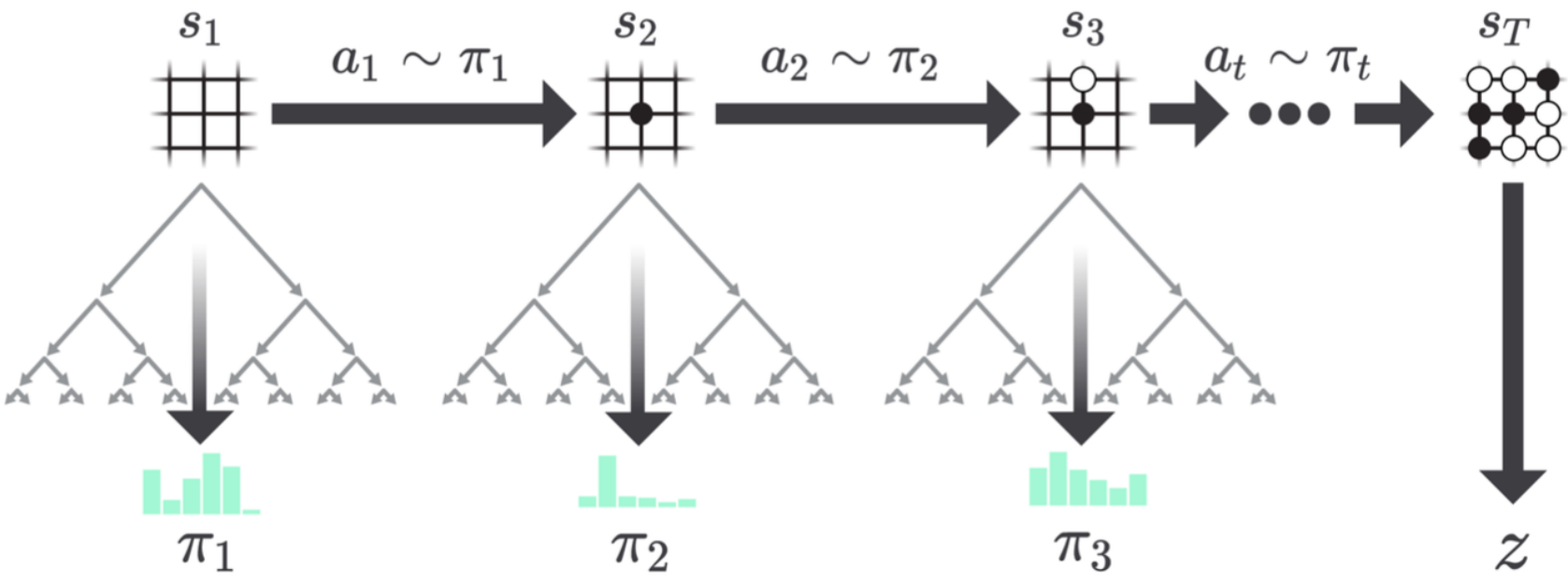
Self-play reinforcement learning



- Program plays games against itself – state sequences: $s_1, ..., s_T$.
- In each position s_t, execute Monte-Carlo tree search (MCTS) using the current policy-value network Π, V and compute search probabilities π_t.
- Select move $a_t \sim \pi_t$.
- Store game winner z defined by terminal position s_T.

Alpha Go Zero (Silver et al., Nature 2017)

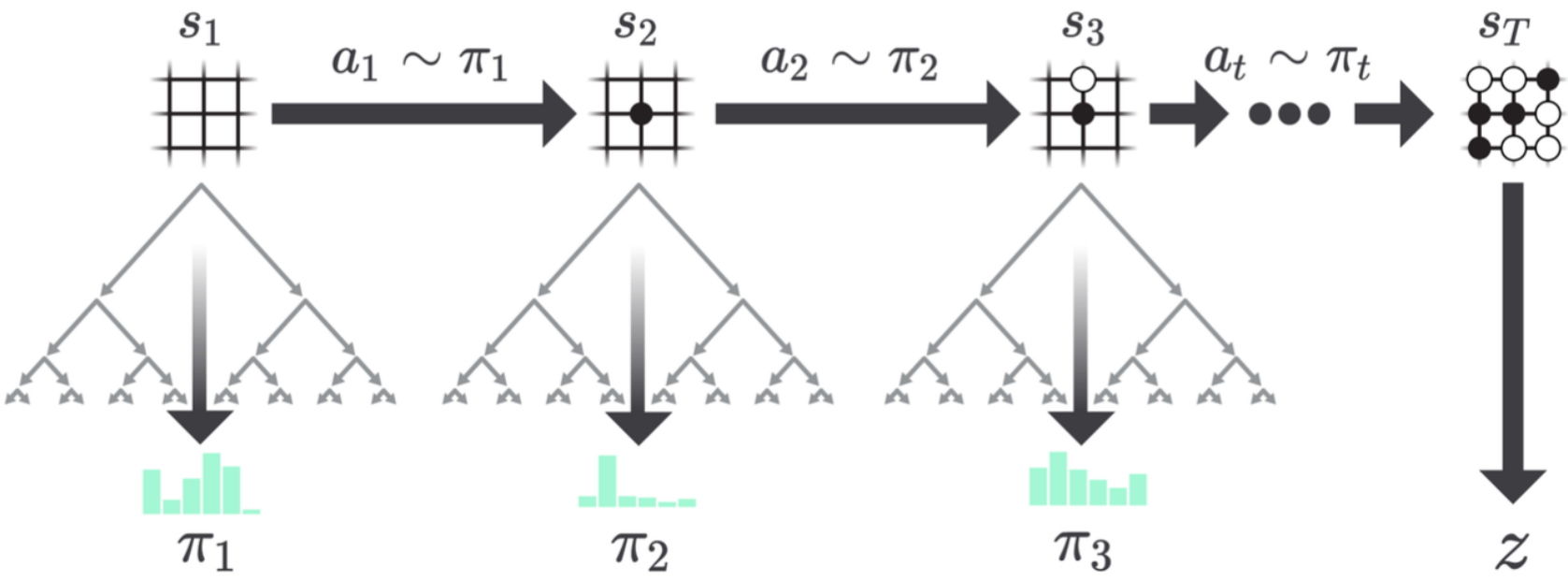
Self-play reinforcement learning



- Program plays games against itself – state sequences: $s_1, ..., s_T$.
- In each position s_t, execute Monte-Carlo tree search (MCTS) using the current policy-value network Π, V and compute search probabilities π_t.
- Select move $a_t \sim \pi_t$.
- Store game winner z defined by terminal position s_T.

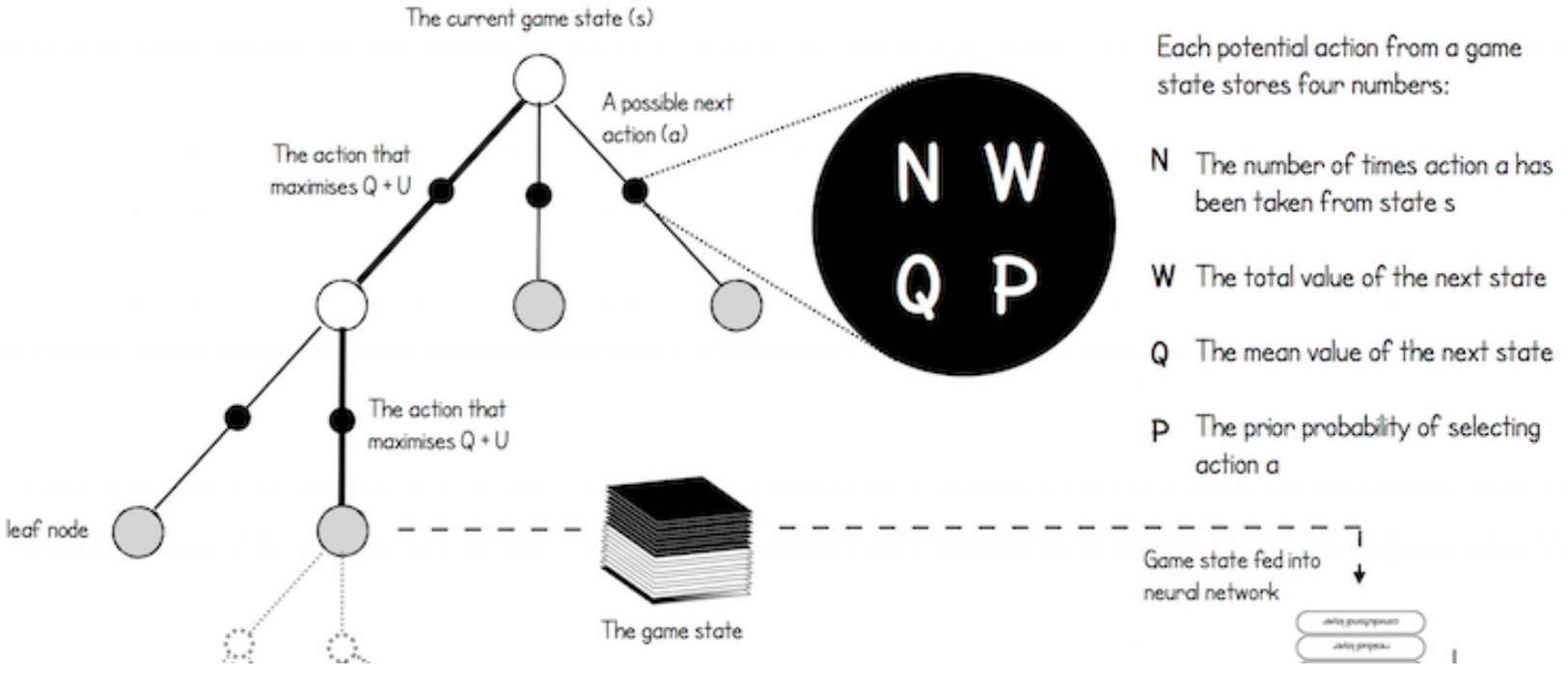
Alpha Go Zero (Silver et al., Nature 2017)

Self-play reinforcement learning

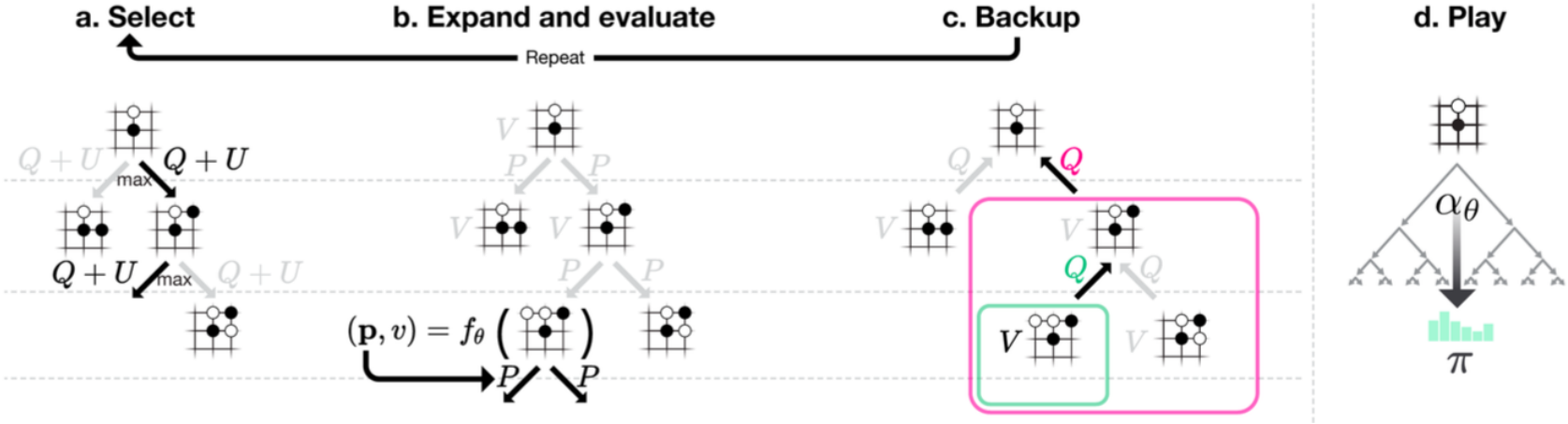


- Program plays games against itself – state sequences: $s_1, \ldots, s_T$.
- In each position s_t, execute Monte-Carlo tree search (MCTS) using the current policy-value network Π, V and compute search probabilities π_t.
- Select move $a_t \sim \pi_t$.
- Store game winner z defined by terminal position s_T.

Alpha Go Zero (Silver et al., Nature 2017)
MCTS



N Number of times selected, P prior selection probability, W/Q total/mean next value



Starting at current game state $\mathbf{s}_0 = \mathbf{s}$, $i = 0$, run 1,600 times:

- While $\mathbf{s}_i$ not leaf: $\mathbf{s}_i \xrightarrow{a} \mathbf{s}_{i+1}$ with action a that maximizes $Q + U(P, N)$
  U dominates early in simulation (exploration), Q dominates later (exploitation)
- Leaf node $\mathbf{s}_i$: Predict value $v = V(\mathbf{s}_i)$ and policy $\Pi(\mathbf{s}_i)$. Set priors of next nodes:

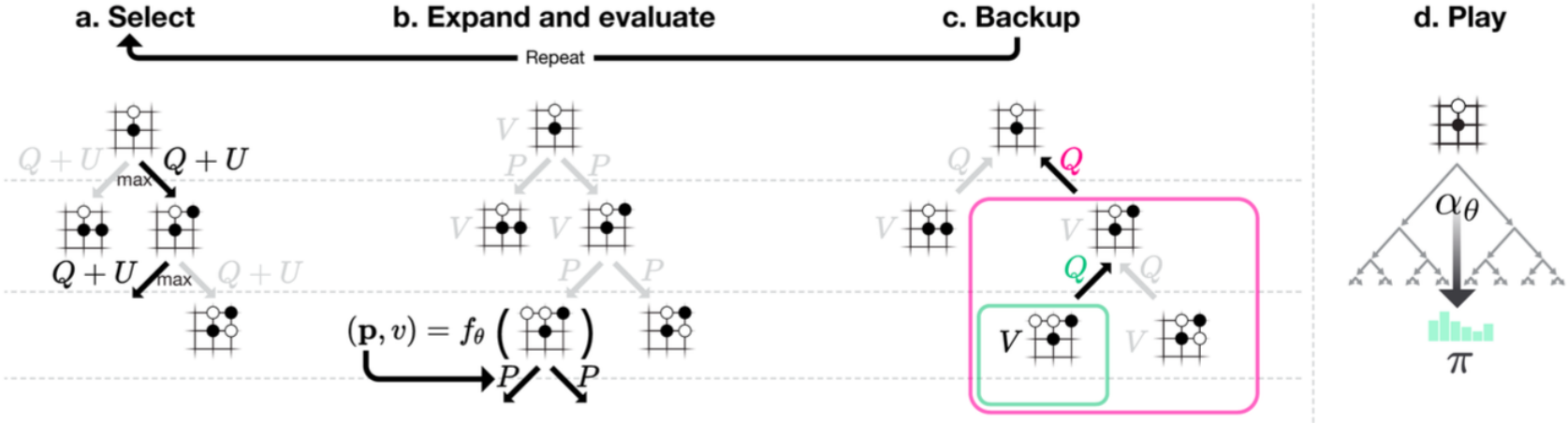
$$P(\mathbf{s}_i, a) = \Pi_a(\mathbf{s}_i)$$

- Backup edges: For each edge $(\mathbf{s}_i, a)$ traversed

$$N(\mathbf{s}_i, a) += 1 \qquad W(\mathbf{s}_i, a) += v \qquad Q = W/N$$

Alpha Go Zero
MCTS

N Number of times selected, P prior selection probability, W/Q total/mean next value



Starting at current game state $\mathbf{s}_0 = \mathbf{s}$, $i = 0$, run 1,600 times:

- While $\mathbf{s}_i$ not leaf: $\mathbf{s}_i \xrightarrow{a} \mathbf{s}_{i+1}$ with action a that maximizes $Q + U(P, N)$
  U dominates early in simulation (exploration), Q dominates later (exploitation)
- Leaf node $\mathbf{s}_i$: Predict value $v = V(\mathbf{s}_i)$ and policy $\Pi(\mathbf{s}_i)$. Set priors of next nodes:

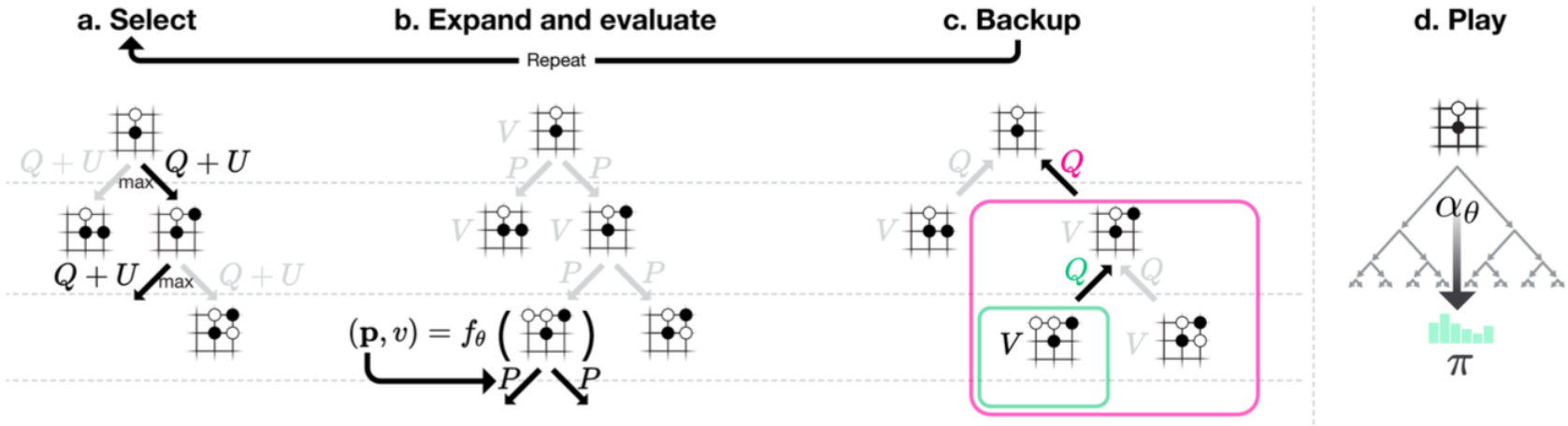
$$P(\mathbf{s}_i, a) = \Pi_a(\mathbf{s}_i)$$

- Backup edges: For each edge $(\mathbf{s}_i, a)$ traversed

$$N(\mathbf{s}_i, a) += 1 \qquad W(\mathbf{s}_i, a) += v \qquad Q = W/N$$

Alpha Go Zero
MCTS

N Number of times selected, P prior selection probability, W/Q total/mean next value



Starting at current game state $\mathbf{s}_0 = \mathbf{s}$, $i = 0$, run 1,600 times:

- While $\mathbf{s}_i$ not leaf: $\mathbf{s}_i \xrightarrow{a} \mathbf{s}_{i+1}$ with action a that maximizes $Q + U(P, N)$
  U dominates early in simulation (exploration), Q dominates later (exploitation)
- Leaf node $\mathbf{s}_i$: Predict value $v = V(\mathbf{s}_i)$ and policy $\Pi(\mathbf{s}_i)$. Set priors of next nodes:

$$P(\mathbf{s}_i, a) = \Pi_a(\mathbf{s}_i)$$

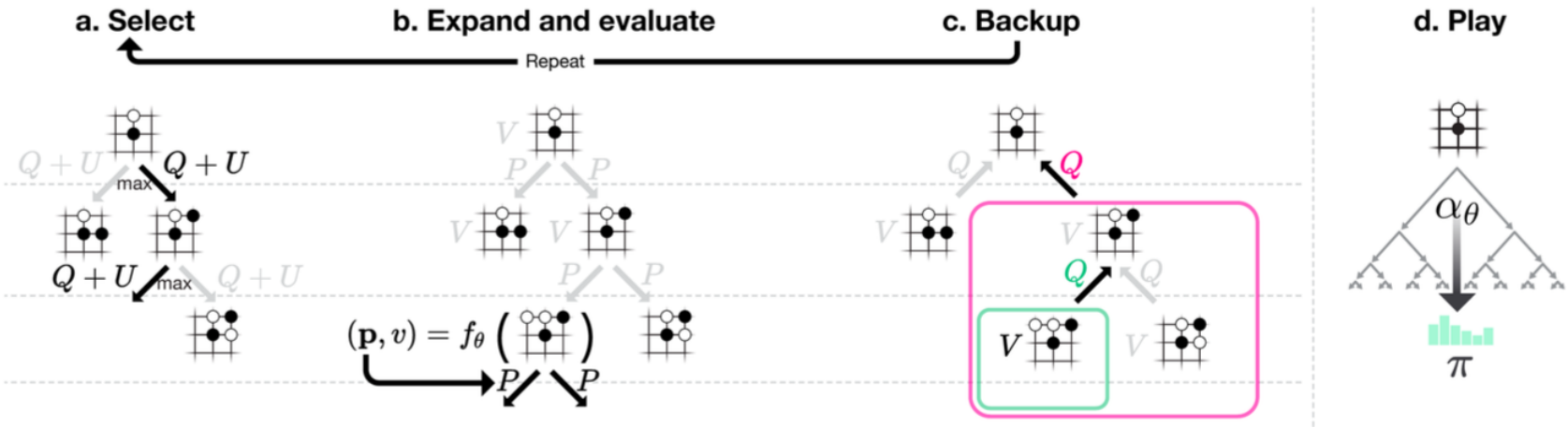
- Backup edges: For each edge $(\mathbf{s}_i, a)$ traversed

$$N(\mathbf{s}_i, a) += 1 \qquad W(\mathbf{s}_i, a) += v \qquad Q = W/N$$

Alpha Go Zero (Silver et al., Nature 2017)

Move selection

N Number of times selected, P prior selection probability, W/Q total/mean next value

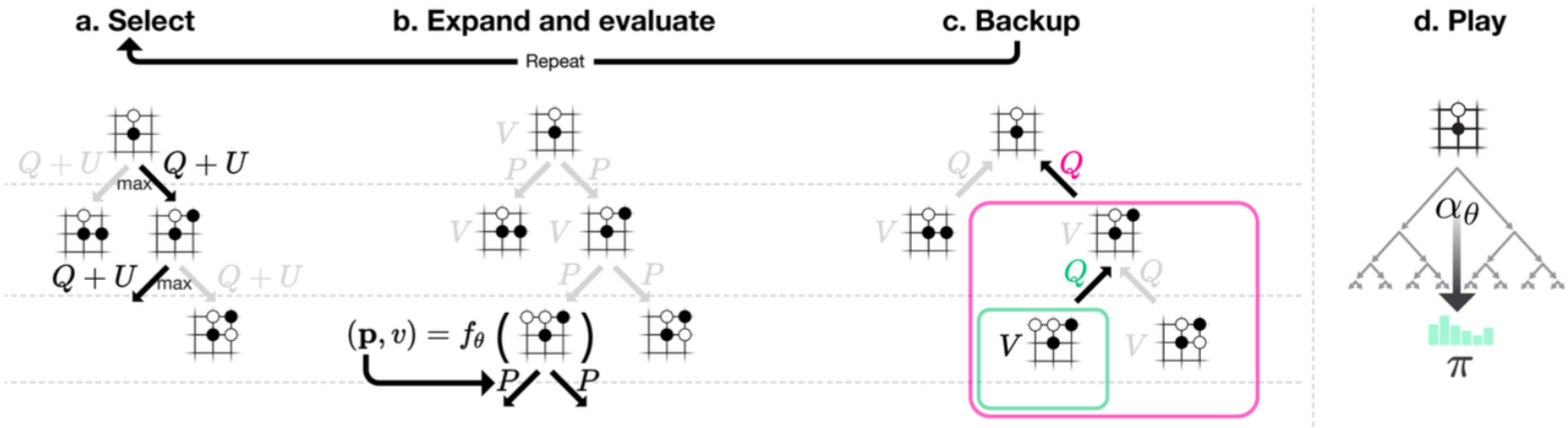


- After 1,600 simulations, choose move:
  - During self play (exploration): $\pi \sim N^{1/\tau}$, temperature parameter τ
  - During competitive play: $\max N$.
- Subtree with chosen move is kept, remaining tree is discarded.

Alpha Go Zero (Silver et al., Nature 2017)

Move selection

N Number of times selected, P prior selection probability, W/Q total/mean next value

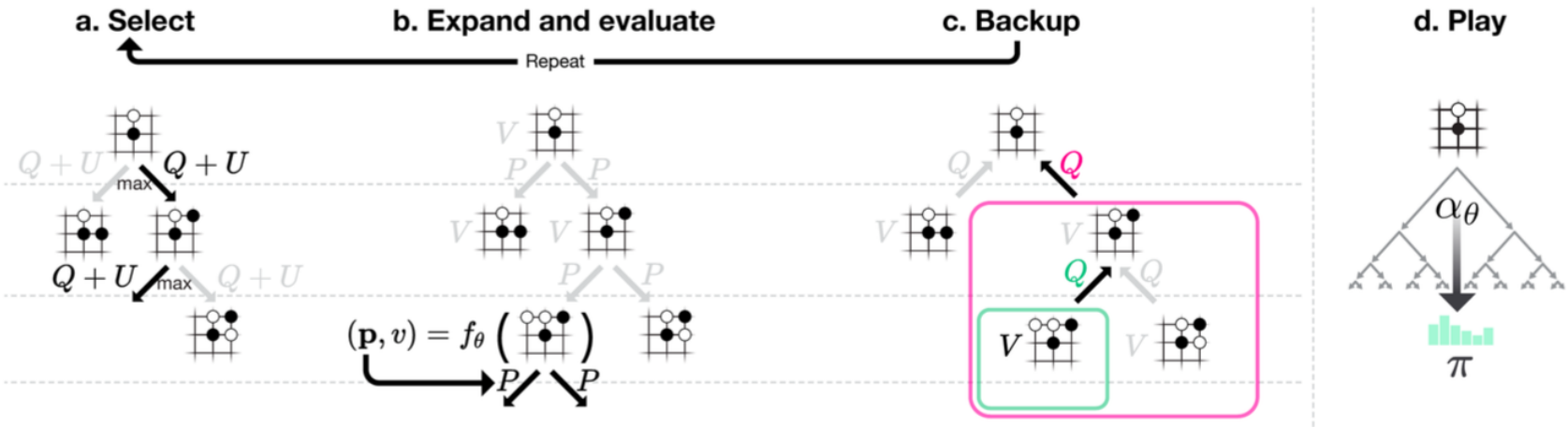


- After 1,600 simulations, choose move:
  - During self play (exploration): $\pi \sim N^{1/\tau}$, temperature parameter τ
  - During competitive play: $\max N$.
- Subtree with chosen move is kept, remaining tree is discarded.

Alpha Go Zero (Silver et al., Nature 2017)

Move selection

N Number of times selected, P prior selection probability, W/Q total/mean next value

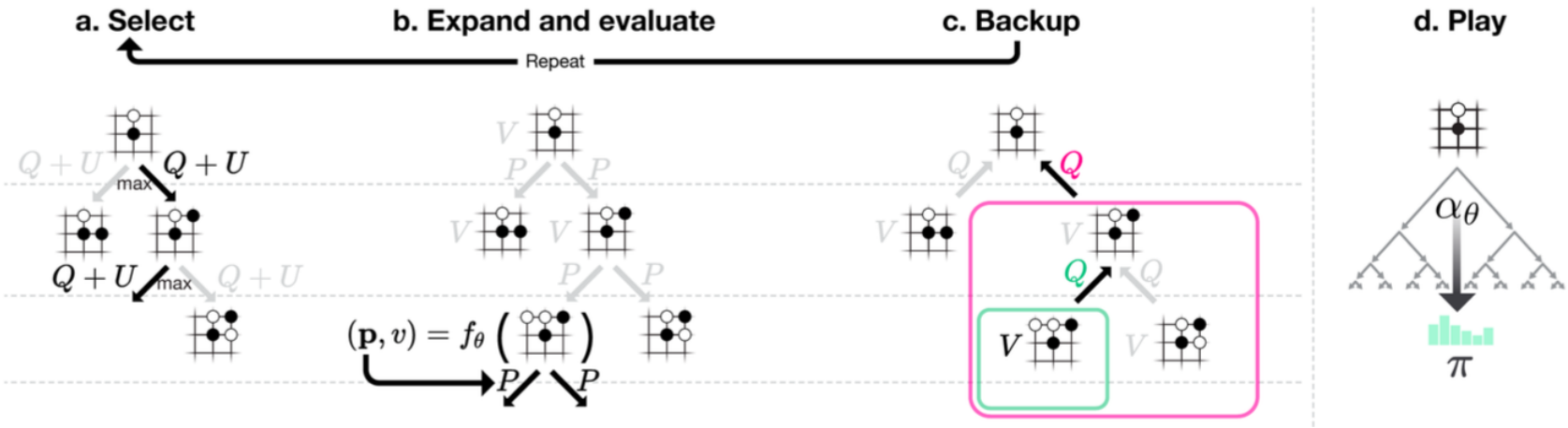


- After 1,600 simulations, choose move:
  - During self play (exploration): $\pi \sim N^{1/\tau}$, temperature parameter τ
  - During competitive play: $\max N$.
- Subtree with chosen move is kept, remaining tree is discarded.

Alpha Go Zero (Silver et al., Nature 2017)

Move selection

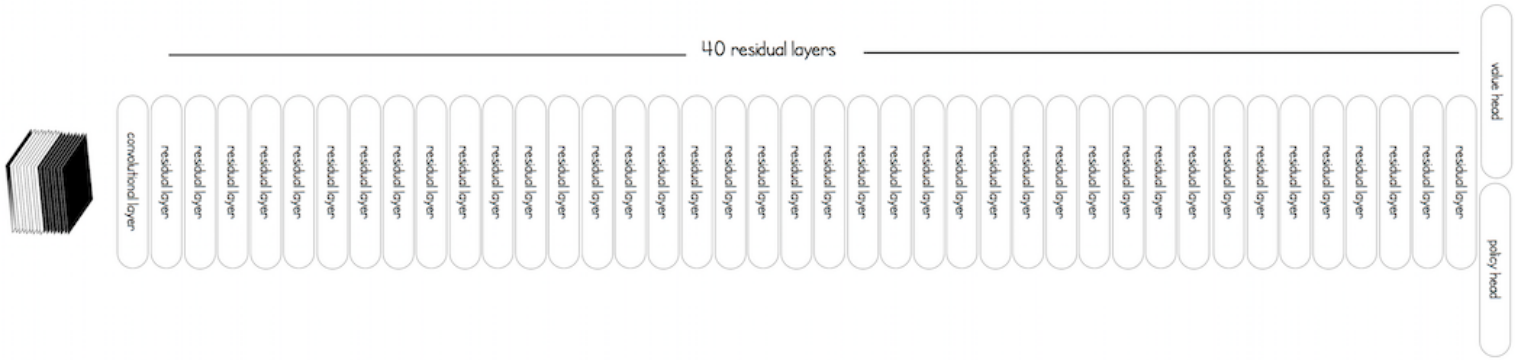
N Number of times selected, P prior selection probability, W/Q total/mean next value



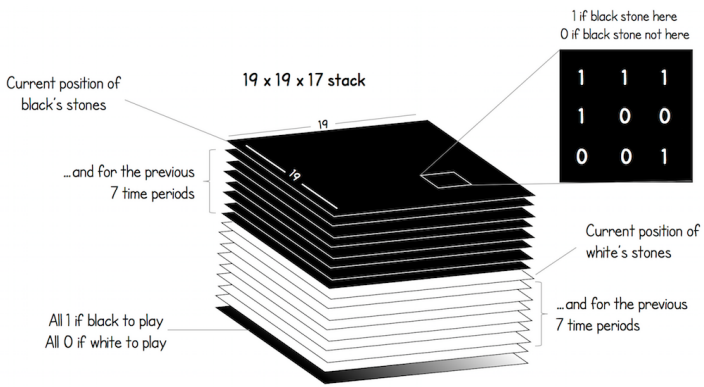
- After 1,600 simulations, choose move:
  - During self play (exploration): $\pi \sim N^{1/\tau}$, temperature parameter τ
  - During competitive play: $\max N$.
- Subtree with chosen move is kept, remaining tree is discarded.

Alpha Go Zero (Silver et al., Nature 2017)

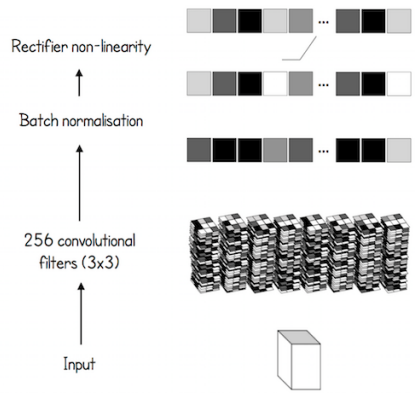
Neural network architecture



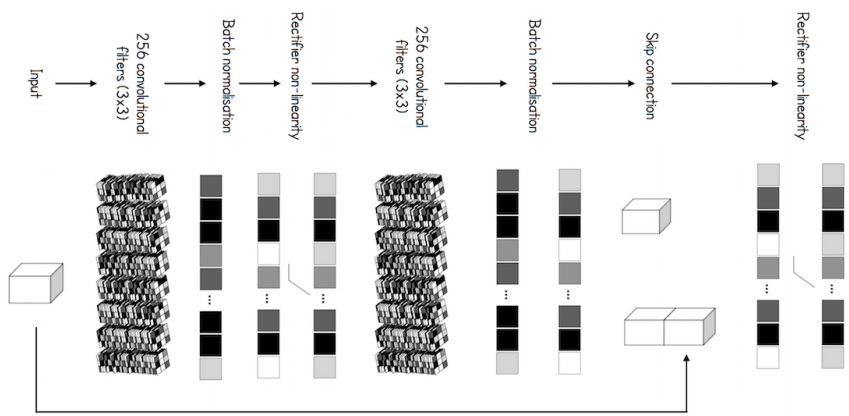
Game state:



Conv Layer:

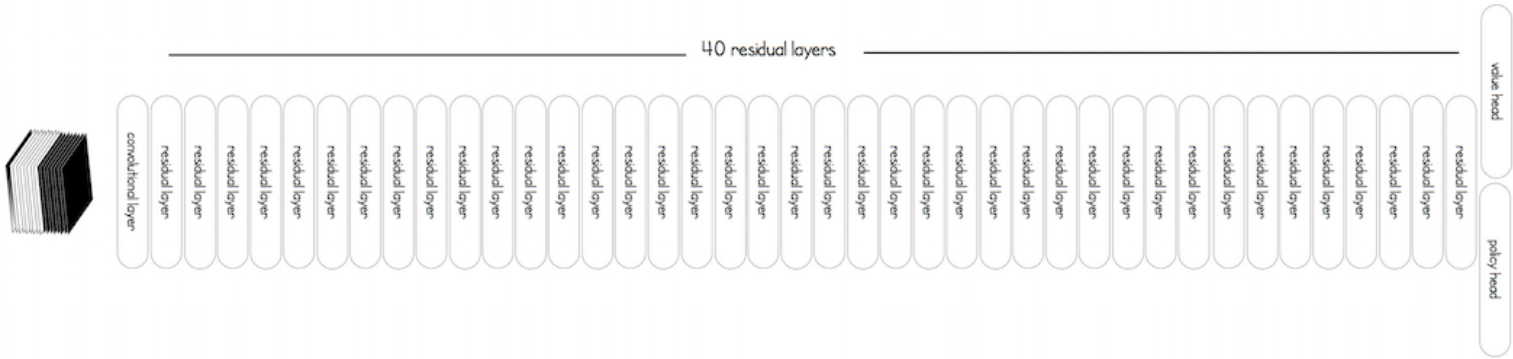


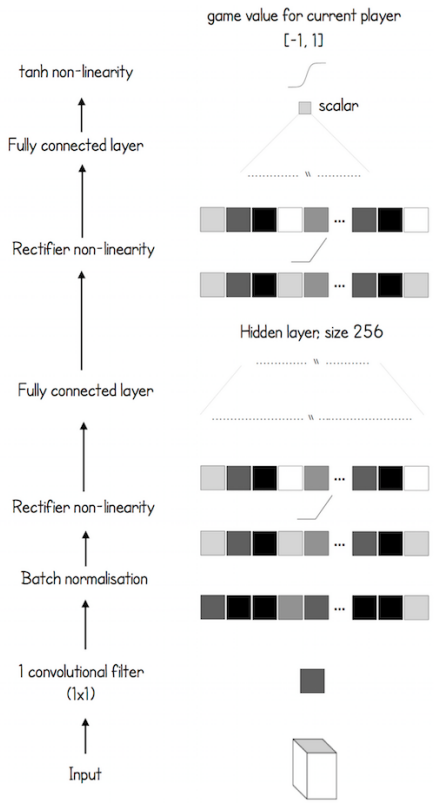
Residual Layer:



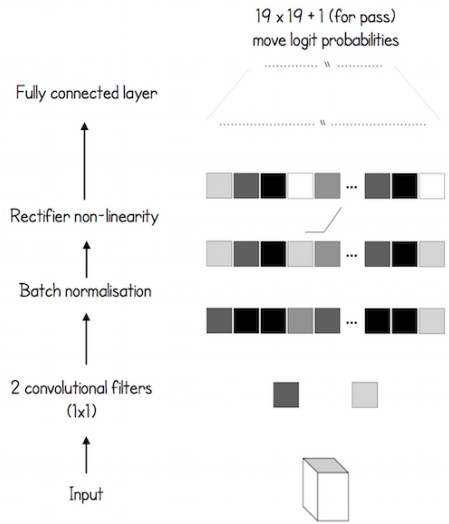
Alpha Go Zero (Silver et al., Nature 2017)

Neural network architecture





Value head



Policy head

Alpha Go Zero (Silver et al., Nature 2017)

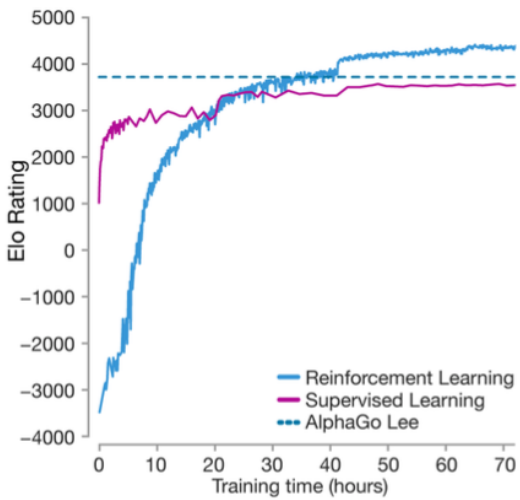
Neural network training in AlphaGo Zero

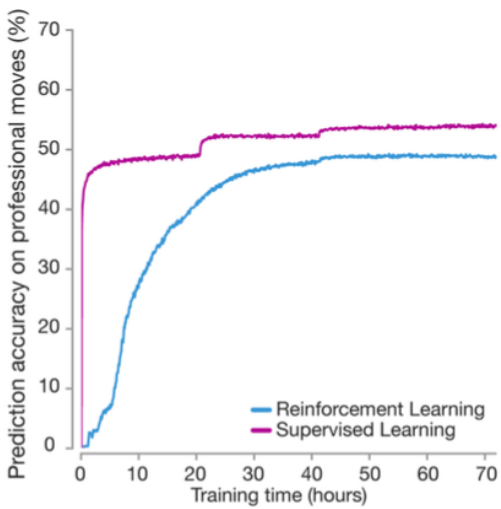
Elo rating E_A, E_B for zero sum games (A wins $\rightarrow$ B loses):

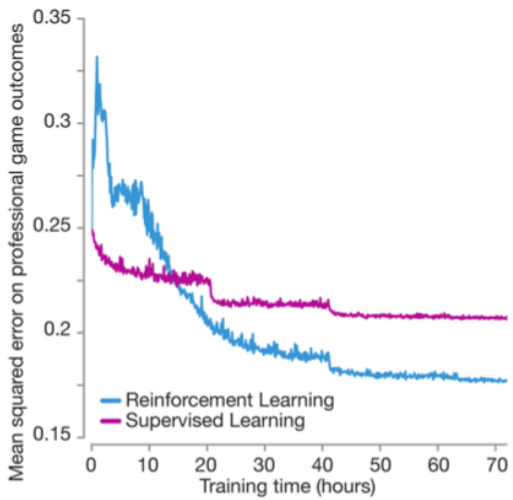
$$p(A\,\text{wins}) = \frac{1}{1+10^{-(E_A-E_B)/400}}$$

$$E_A - E_B = -400\log_{10}(1/p(A\,\text{wins}) - 1)$$

$p(A\,\text{wins})$	0.01	0.1	0.5	0.9	0.99
$E_A - E_B$	-798	-381	0	381	798







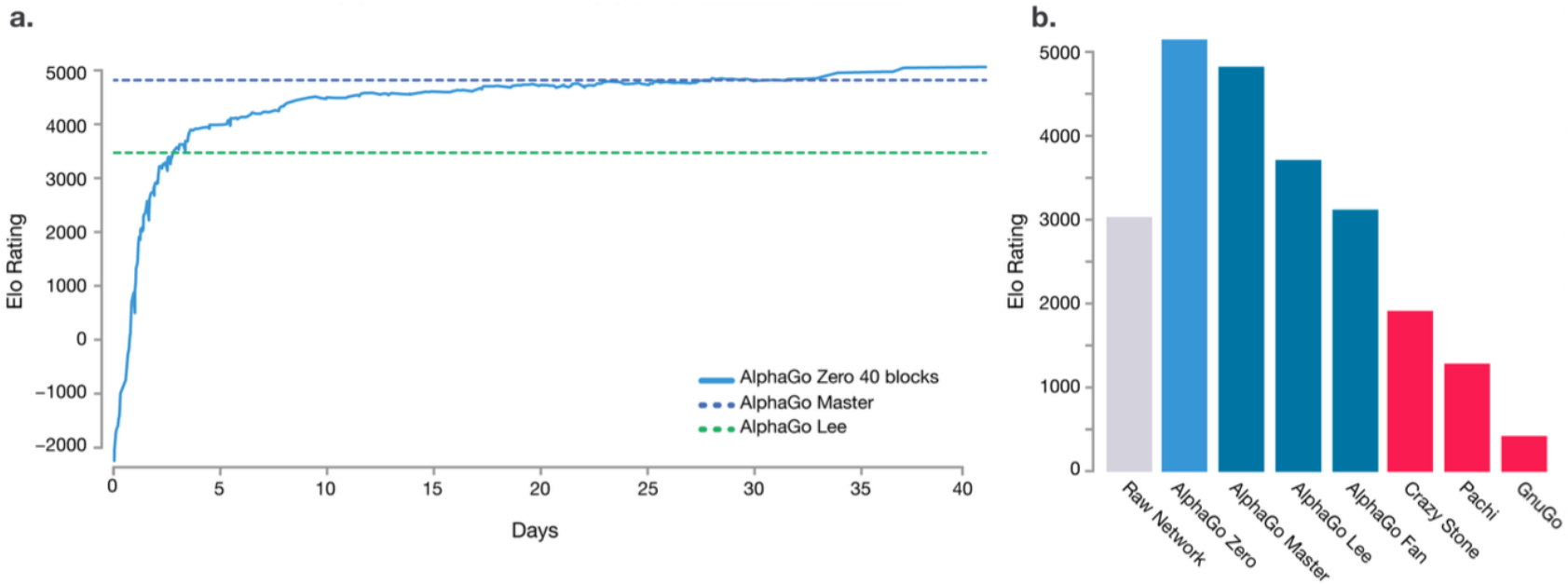
Left: Performance of self-play reinforcement learning.

Middle: Prediction accuracy on human professional moves.

Right: Mean-squared error (MSE) on human professional game outcomes.

Alpha Go Zero (Silver et al., Nature 2017)

Neural network training in AlphaGo Zero

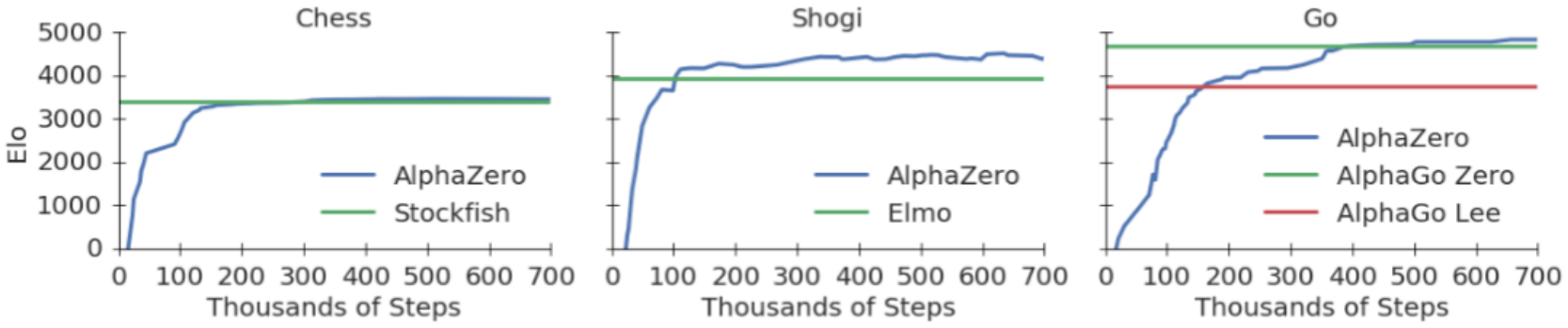


Performance of AlphaGo Zero.
a Learning curve for AlphaGo Zero using larger 40 block residual network over 40 days.
b Final performance of AlphaGo Zero. AlphaGo Zero was trained for 40 days using a 40 residual block neural network. The plot shows the results of a tournament between: AlphaGo Zero, AlphaGo Master (defeated top human professionals 60-0 in online games), AlphaGo Lee (defeated Lee Sedol), AlphaGo Fan (defeated Fan Hui), as well as previous Go programs Crazy Stone, Pachi and GnuGo.

Alpha Zero (Silver et al., 2017)

Main changes to Alpha Go Zero:

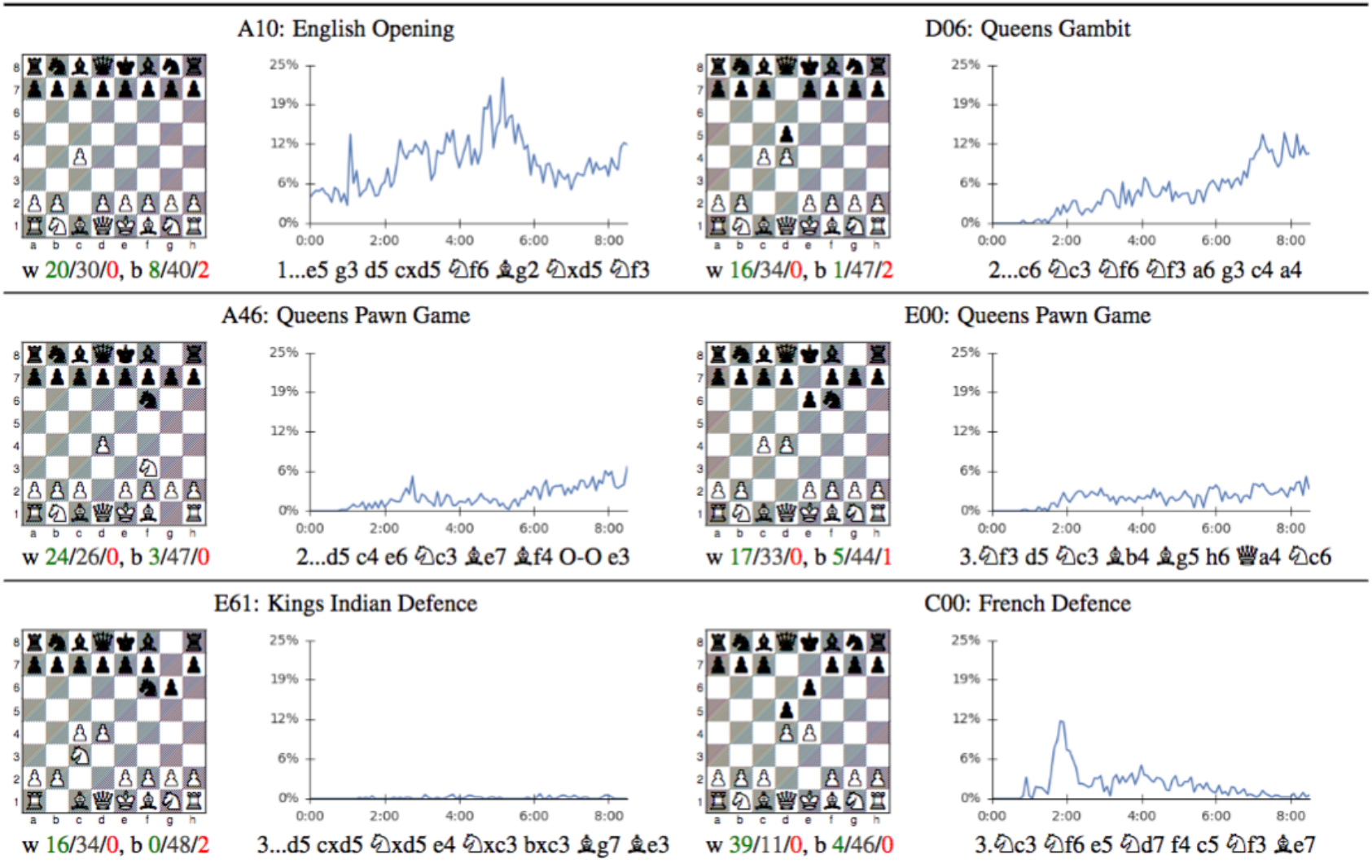
- Maximize expected number of wins instead of number of wins
- Keeps only one network (no "champion" who needs to win 55% of the games)
- Plays three games (Chess, Shogi, Go) with the same hyperparameters.



Game	White	Black	Win	Draw	Loss
Chess	*AlphaZero*	*Stockfish*	25	25	0
	Stockfish	*AlphaZero*	3	47	0
Shogi	*AlphaZero*	*Elmo*	43	2	5
	Elmo	*AlphaZero*	47	0	3
Go	*AlphaZero*	*AG0 3-day*	31	–	19
	AG0 3-day	*AlphaZero*	29	–	21

Versions	Playing hardware[22]	Elo rating	Matches
AlphaGo Fan	176 GPUs,[2] distributed	3,144[1]	5:0 against Fan Hui
AlphaGo Lee	48 TPUs,[2] distributed	3,739[1]	4:1 against Lee Sedol
AlphaGo Master	4 TPUs,[2] single machine	4,858[1]	60:0 against professional players; Future of Go Summit
AlphaGo Zero (40 days)	4 TPUs,[2] single machine	5,185[1]	100:0 against AlphaGo Lee 89:11 against AlphaGo Master
AlphaZero (34 hours)	4 TPUs, single machine[6]	4,000 (est.)[6]	60:40 against a 3-day AlphaGo Zero

Alpha Zero (Silver et al., 2017)



Videos of representative games on YouTube