Advanced Prediction Models

Deep Learning, Graphical Models and Reinforcement Learning

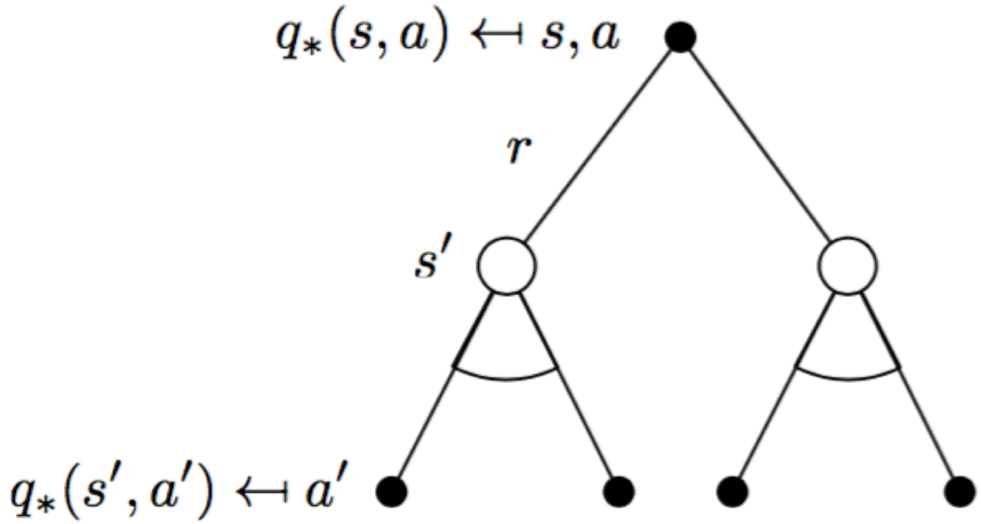
Today's Outline

- Value Function Approximation
- Deep Reinforcement Learning
  - DQN for Atari Games
  - AlphaGo for Go

Value Function Approximation

The Q Learning Algorithm

- If we know the model
  - Turn the Bellman Optimality Equation into an iterative update
  - This is called Value Iteration

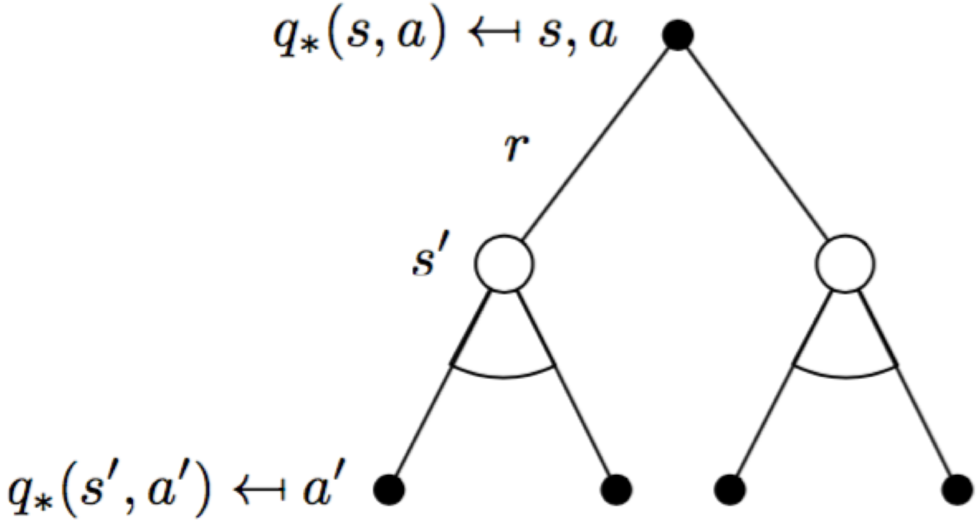


$$q_*(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \max_{a'} q_*(s',a')$$

[1]Reference: David Silver, 2015

The Q Learning Algorithm

- If we do not know the model
  - Do sampling to get an incremental iterative update
  - Choose next actions to ensure exploration

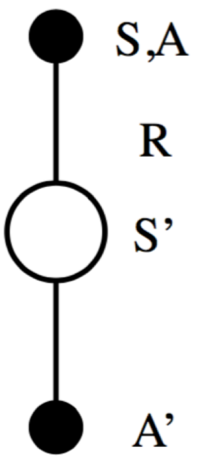


$$q_*(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \max_{a'} q_*(s',a')$$

[1]Reference: David Silver, 2015

The Q Learning Algorithm

- If we do not know the model
  - Do sampling to get an incremental iterative update
  - Choose next actions to ensure exploration

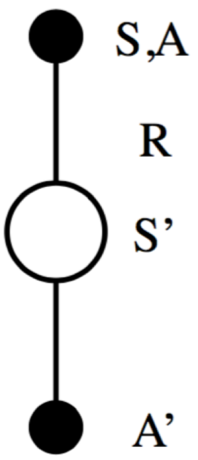


$$Q(S,A) \leftarrow Q(S,A) + \alpha \left(R + \gamma Q(S',A') - Q(S,A) \right)$$

[1]Reference: David Silver, 2015

The Q Learning Algorithm

- If we do not know the model
  - Do sampling to get an incremental iterative update
  - Choose next actions to ensure exploration

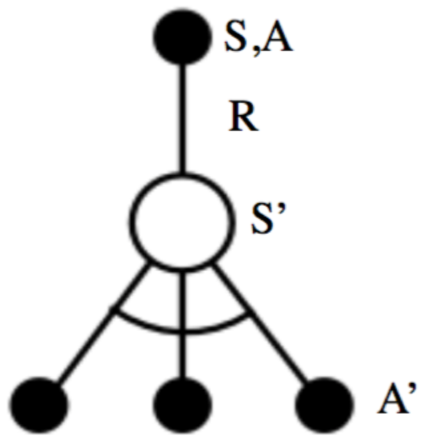


$$Q(S, A) \leftarrow Q(S, A) + \alpha \left(R + \gamma Q(S', A') - Q(S, A) \right)$$

[1]Reference: David Silver, 2015

The Q Learning Algorithm

- If we do not know the model
  - Do sampling to get an incremental iterative update
  - Choose next actions to ensure exploration



$$Q(S, A) \leftarrow Q(S, A) + \alpha \left(R + \gamma \boxed{\max_{a'} Q(S', a')} - Q(S, A) \right)$$

[1]Reference: David Silver, 2015

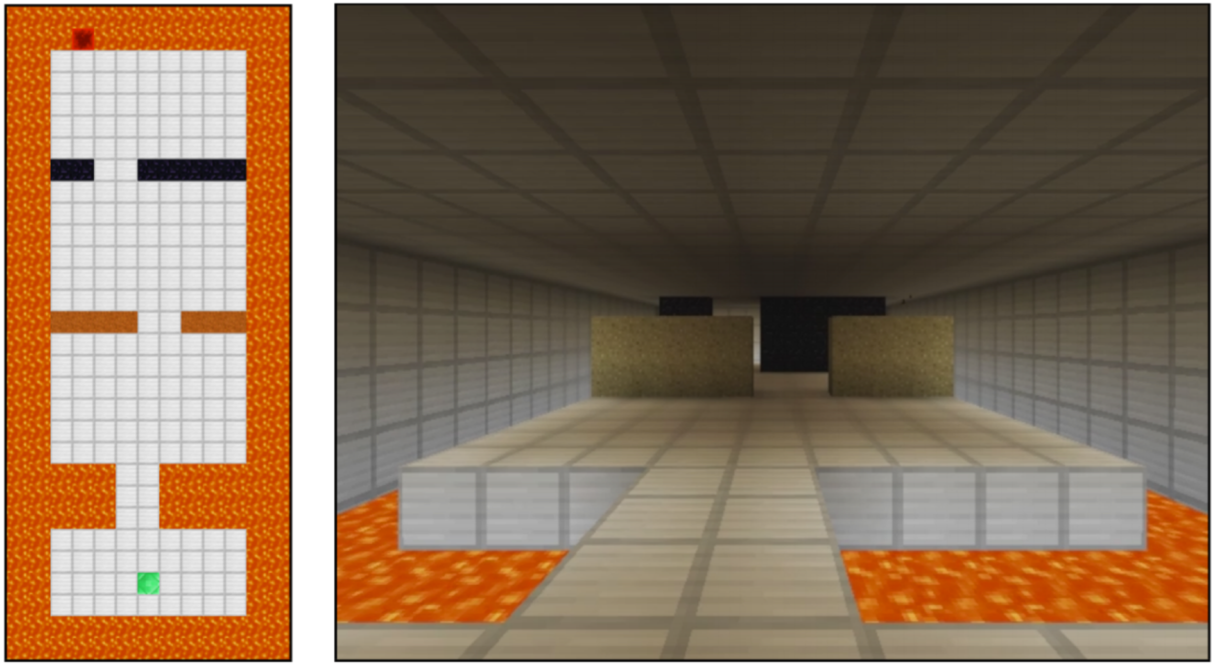
The Q Learning Algorithm

- Initialize Q, which is a table of size #states×#actions
- Start at state s_1
- For $t = 1,2,3, \ldots$.
  - Take A_t chosen uniformly at random with probability ϵ — Explore
  - Take $\mathrm{argmax}_{a \in A}\, Q(S_t, a)$ with probability $1 - \epsilon$ — Exploit
  - Update Q:
    - $Q(S_t, A_t) = Q(S_t, A_t) + \alpha_t \underbrace{\left(R_{t+1} + \gamma \max_{a \in A} Q(S_{t+1}, a) - Q(S_t, A_t)\right)}_{\text{Temporal difference error}}$

- Parameter ϵ is the exploration parameter
- Parameter α_t is the learning rate
- Under appropriate assumptions[1], $\lim_{t \to \infty} Q = Q^*$

[1] Reference: Christopher J. C. H. Watkins and Peter Dayan, 1992

Tabular Q Learning is Not Enough



Robotic agent navigating in real-world (left)

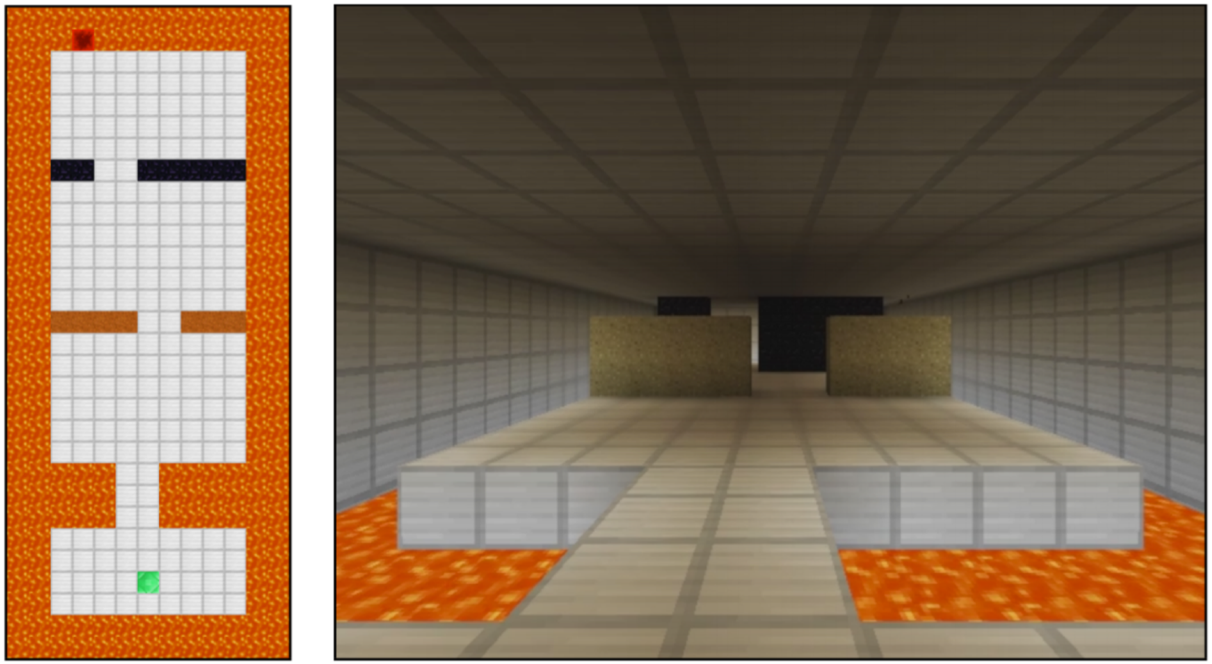
States: Position in a grid

Actions: Forward/Back/Left/Right

Reward: 1 on reaching target, -100 for dying

[1]Reference: Krishnamurthy et al. https://arxiv.org/abs/1602.02722

Tabular Q Learning is Not Enough



Robotic agent navigating in real-world (right)

States: Camera view in front of the robot

Actions: Forward/Back/Left/Right

Reward: 1 on reaching target, -100 for dying

[1]Reference: Krishnamurthy et al. https://arxiv.org/abs/1602.02722

Function Approximation Recipe

- Use a deep network or any other function class to to represent
  - the value function, and/or
  - the policy, and/or
  - the model

[1] Reference: David Silver, 2015

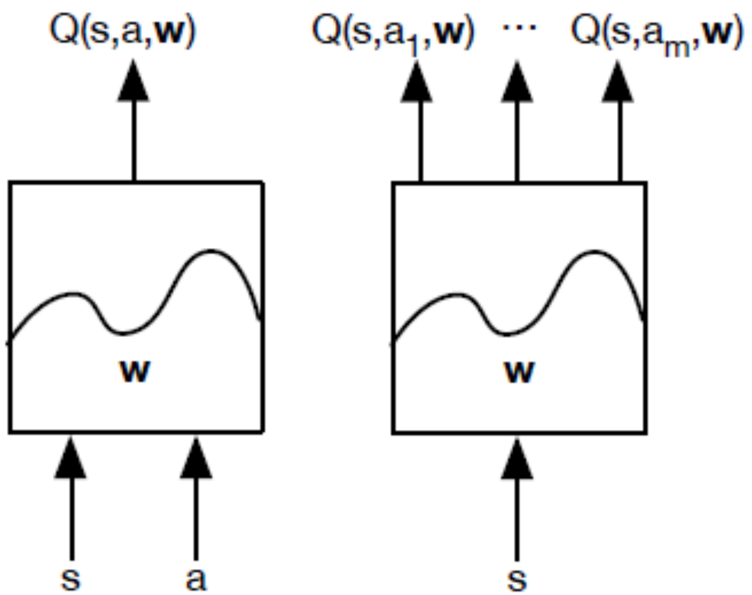
Function Approximation Recipe

- Use a deep network or any other function class to to represent
  - the value function, and/or
  - the policy, and/or
  - the model
- Optimize this network end to end
  - Example:
    - If the approximator is differentiable
    - Use stochastic gradient descent
- Do the optimization incrementally or in batch mode

[1]Reference: David Silver, 2015

Q Function Approximation

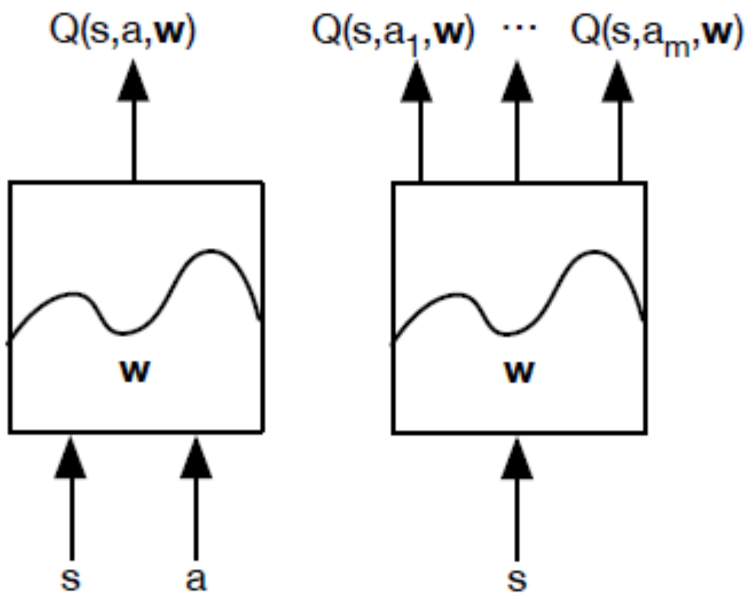
- Instead of storing #states×#action parameters in a table, we want to find more scalable ways to capture Q values
- Represent Q using a function approximator with weights w: $Q(s, a; w) \approx Q^*(s, a)$



[1]Figure: David Silver

Q Function Approximation

- Instead of storing #states×#action parameters in a table, we want to find more scalable ways to capture Q values
- Represent Q using a function approximator with weights w: $Q(s, a; w) \approx Q^*(s, a)$



- Linear
- Decision tree
- Neural network
- Basis functions
- Nearest neighbor

[1]Figure: David Silver

Q Function Approximation

- Approximate the action-value function

$$\hat{q}(S, A, \mathbf{w}) \approx q_\pi(S, A)$$

[1] Reference: David Silver, 2015

Q Function Approximation

- Approximate the action-value function

$$\hat{q}(S, A, \mathbf{w}) \approx q_\pi(S, A)$$

- Minimise mean-squared error between approximate action-value fn $\hat{q}(S, A, \mathbf{w})$ and true action-value fn $q_\pi(S, A)$

$$J(\mathbf{w}) = \mathbb{E}_\pi \left[\left(\boxed{q_\pi(S, A)} - \hat{q}(S, A, \mathbf{w}) \right)^2 \right]$$

[1]Reference: David Silver, 2015

Q Function Approximation

- Approximate the action-value function

$$\hat{q}(S, A, \mathbf{w}) \approx q_\pi(S, A)$$

- Minimise mean-squared error between approximate action-value fn $\hat{q}(S, A, \mathbf{w})$ and true action-value fn $q_\pi(S, A)$

$$J(\mathbf{w}) = \mathbb{E}_\pi \left[(q_\pi(S, A) - \hat{q}(S, A, \mathbf{w}))^2 \right]$$

- Use stochastic gradient descent to find a local minimum

$$-\frac{1}{2}\nabla_{\mathbf{w}} J(\mathbf{w}) = (q_\pi(S, A) - \hat{q}(S, A, \mathbf{w}))\nabla_{\mathbf{w}}\hat{q}(S, A, \mathbf{w})$$

$$\Delta\mathbf{w} = \alpha(q_\pi(S, A) - \hat{q}(S, A, \mathbf{w}))\nabla_{\mathbf{w}}\hat{q}(S, A, \mathbf{w})$$

[1] Reference: David Silver, 2015

Q Function Approximation: Example

- Represent state *and* action by a *feature vector*

$$\mathbf{x}(S, A) = \begin{pmatrix} \mathbf{x}_1(S, A) \\ \vdots \\ \mathbf{x}_n(S, A) \end{pmatrix}$$

[1]Reference: David Silver, 2015

Q Function Approximation: Example

- Represent state *and* action by a *feature vector*

$$\mathbf{x}(S, A) = \begin{pmatrix} \mathbf{x}_1(S, A) \\ \vdots \\ \mathbf{x}_n(S, A) \end{pmatrix}$$

- Represent action-value fn by linear combination of features

$$\hat{q}(S, A, \mathbf{w}) = \mathbf{x}(S, A)^\top \mathbf{w} = \sum_{j=1}^{n} \mathbf{x}_j(S, A)\mathbf{w}_j$$

[1]Reference: David Silver, 2015

Q Function Approximation: Example

- Represent state *and* action by a *feature vector*

$$\mathbf{x}(S,A) = \begin{pmatrix} \mathbf{x}_1(S,A) \\ \vdots \\ \mathbf{x}_n(S,A) \end{pmatrix}$$

- Represent action-value fn by linear combination of features

$$\hat{q}(S,A,\mathbf{w}) = \mathbf{x}(S,A)^\top \mathbf{w} = \sum_{j=1}^{n} \mathbf{x}_j(S,A)\mathbf{w}_j$$

- Stochastic gradient descent update

$$\nabla_{\mathbf{w}} \hat{q}(S,A,\mathbf{w}) = \mathbf{x}(S,A)$$

$$\Delta \mathbf{w} = \alpha(\boxed{q_\pi(S,A)} - \hat{q}(S,A,\mathbf{w}))\mathbf{x}(S,A)$$

[1]Reference: David Silver, 2015

Q Function Approximation: Another Perspective

- Recall the Q Learning update

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha_t(R_{t+1} + \gamma \max_{a \in A} Q(S_{t+1}, a) \; - Q(S_t, A_t))$$

[1]Reference: David Silver, 2015

Q Function Approximation: Another Perspective

- Recall the Q Learning update

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha_t(R_{t+1} + \gamma \max_{a \in A} Q(S_{t+1}, a) \; - Q(S_t, A_t))$$

- At optimality
  - $\mathrm{E}\left[R_{t+1} + \gamma \max_{a \in A} Q(S_{t+1}, a) \; - Q(S_t, A_t)\right] = 0$

[1]Reference: David Silver, 2015

Q Function Approximation: Another Perspective

- Recall the Q Learning update

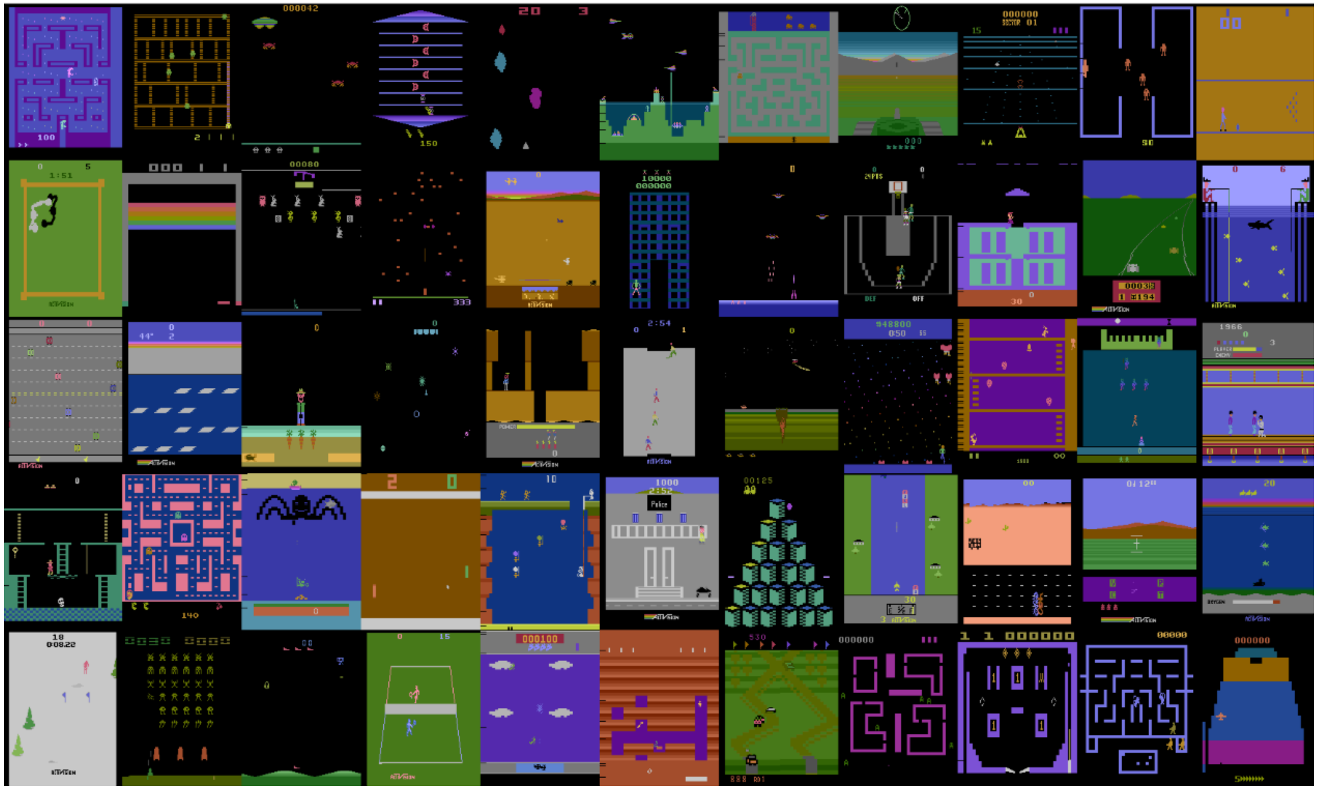
$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha_t (R_{t+1} + \gamma \max_{a \in A} Q(S_{t+1}, a) \; - Q(S_t, A_t))$$

- At optimality
  - $\mathrm{E}\left[R_{t+1} + \gamma \max_{a \in A} Q(S_{t+1}, a) \; - Q(S_t, A_t)\right] = 0$

- Intuitively, this tells us to minimize the empirical error between
  - $R_{t+1} + \gamma \max_{a \in A} Q(S_{t+1}, a, w)$ and $Q(S_t, A_t, w)$

[1]Reference: David Silver, 2015

Example: Function Approximation Success (2013)



[1]Figure: Defazio Graepel, Atari Learning Environment

Issues with Function Approximation

- This can potentially be a nonlinear optimization over w
  - Unless we use a linear approximator

- Can optimize incrementally or in batch
  - Which is better? (*we will answer this for DQN later*)

- Naïve optimization may diverge and oscillate! This is because
  - The data is not i.i.d.
  - Policy/Value may be too sensitive to action choice (max over actions may completely change future trajectory)

Questions?

Today's Outline

- Value Function Approximation
- Deep Reinforcement Learning
  - DQN for Atari Games
  - AlphaGo for Go

Deep Reinforcement Learning I: DQN

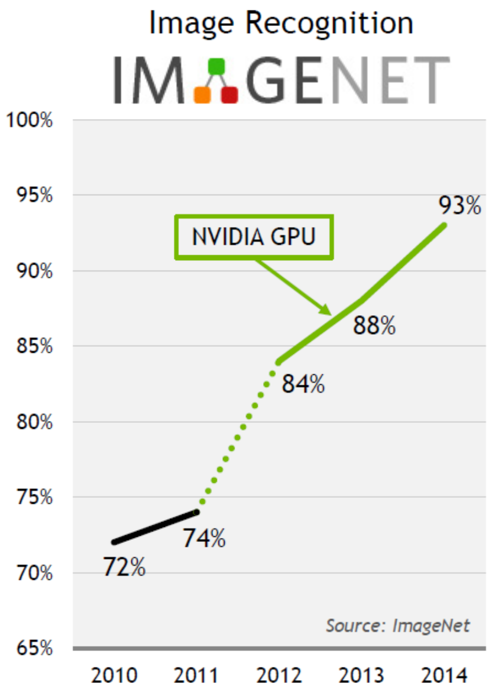
Deep Reinforcement Learning

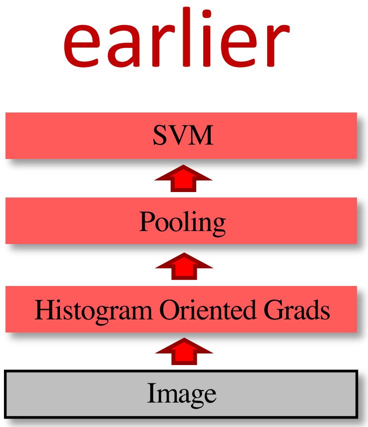
- Bringing in the success of deep perception/prediction architectures to function approximation

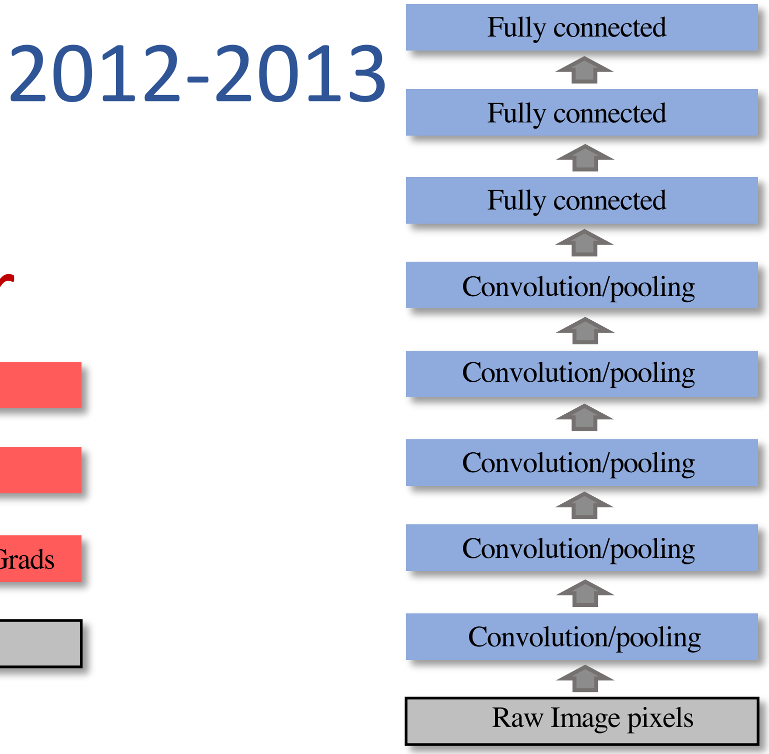
- We will look at two RL agents
  - DQN (2013)
  - AlphaGo (2016)

- Attempt to highlight some additional aspects that made these agents succeed so well in their respective domains

Why Deep Representations?



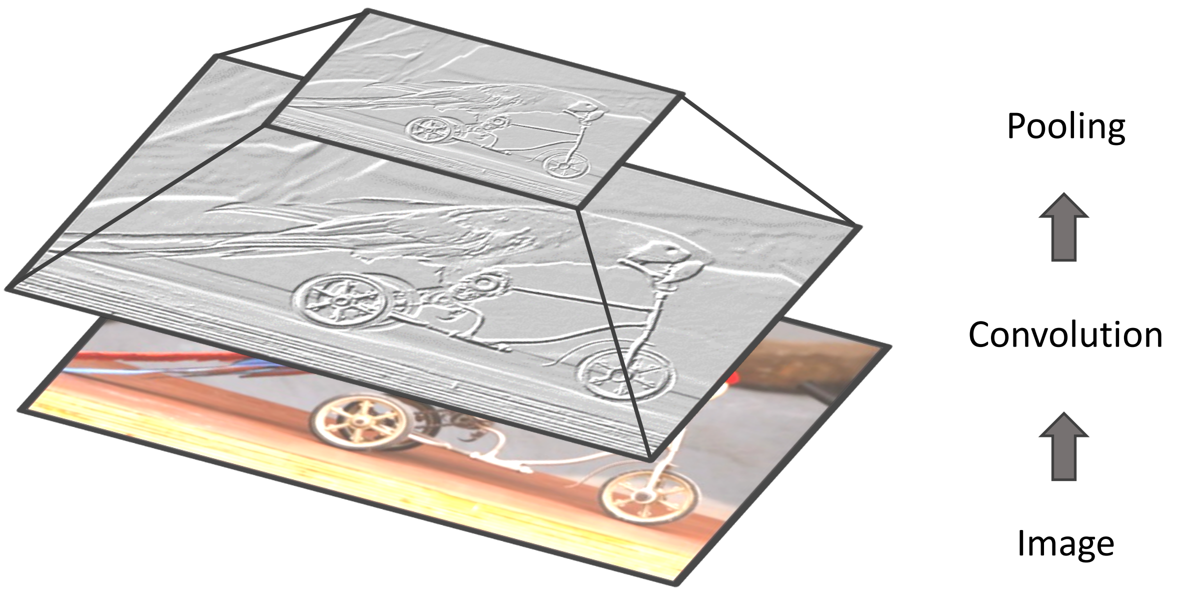




[1] Reference: Julie Bernauer/Ryan Olson, Li Deng

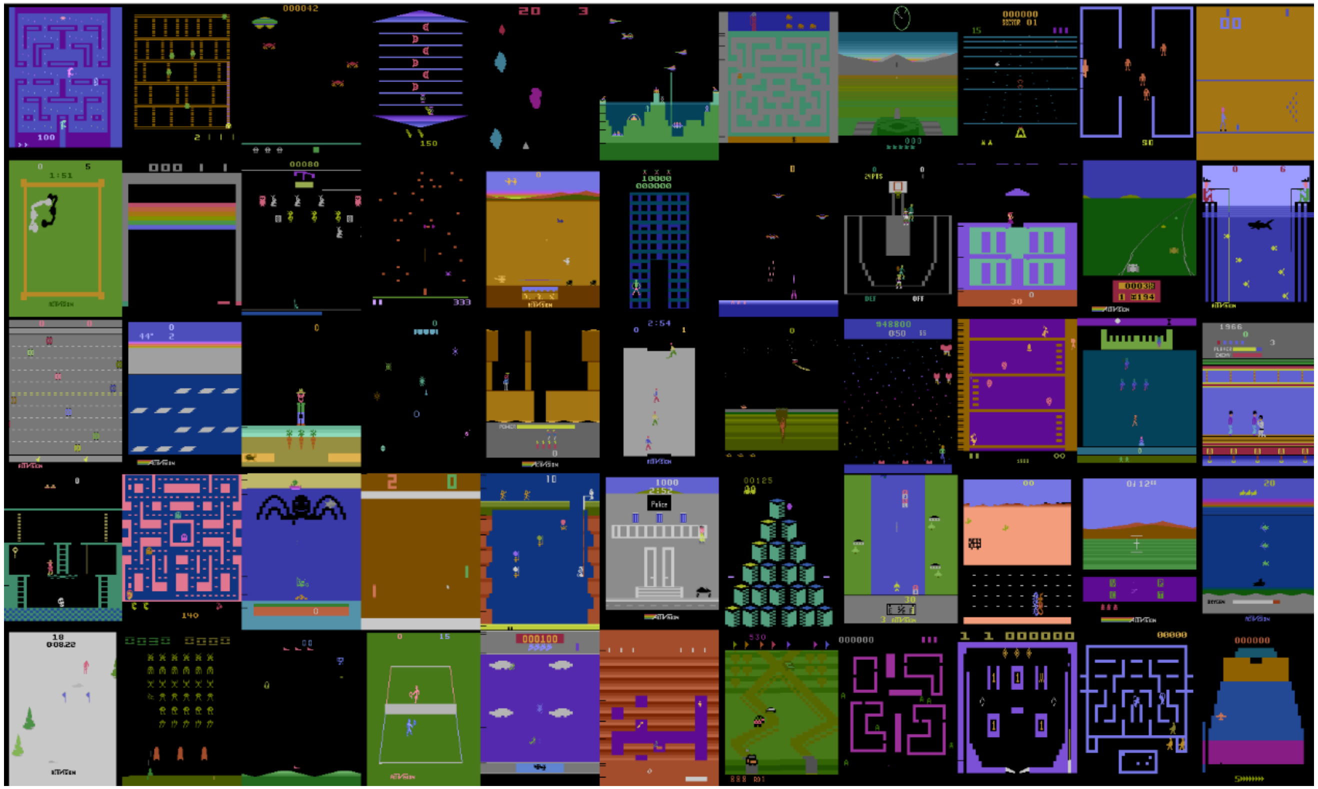
Why Deep Representations?

- CNN as the Function Approximator
- Captures two key properties
  - Local connections with weight sharing
  - Pooling for translation invariance



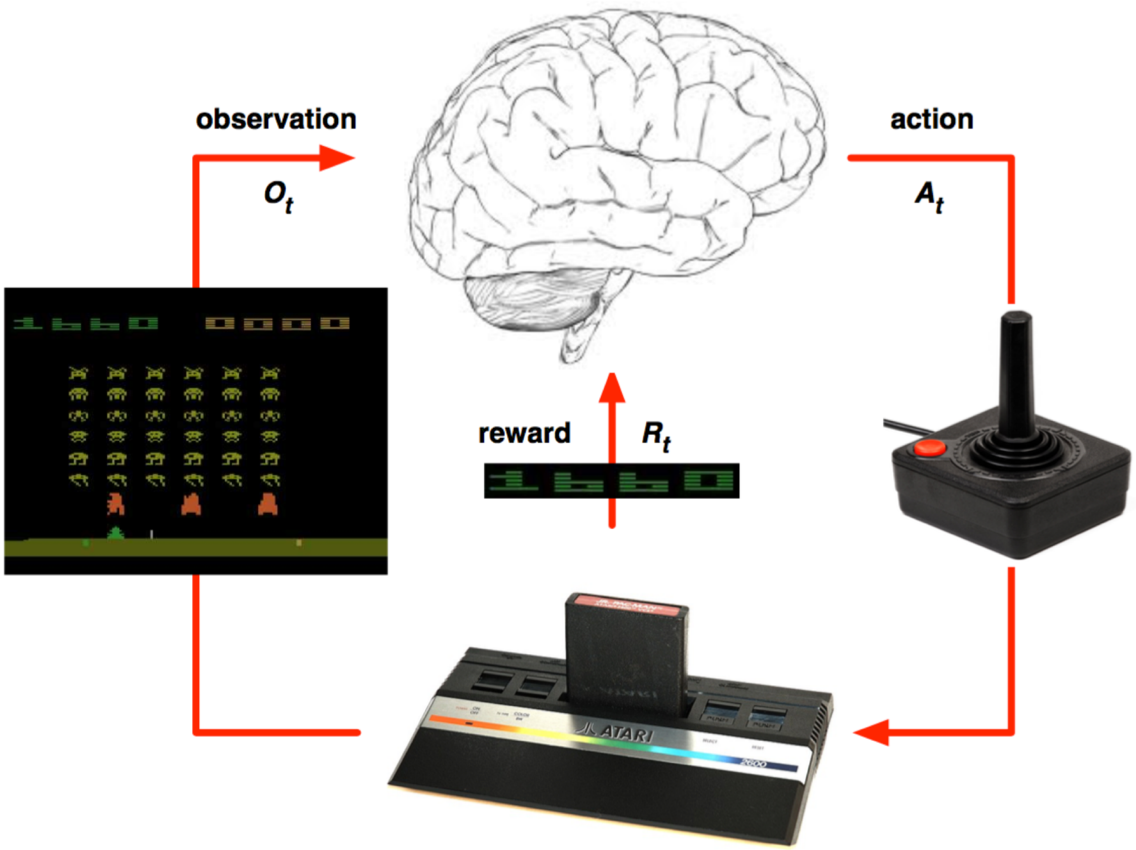
[1]Figure: Li Deng

DQN Plays Atari (2013)



[1]Figure: Defazio Graepel, Atari Learning Environment

DQN Architecture



- Rules of the game are unknown
- Learn directly from interactive game-play
- Pick actions on joystick, see pixels and scores

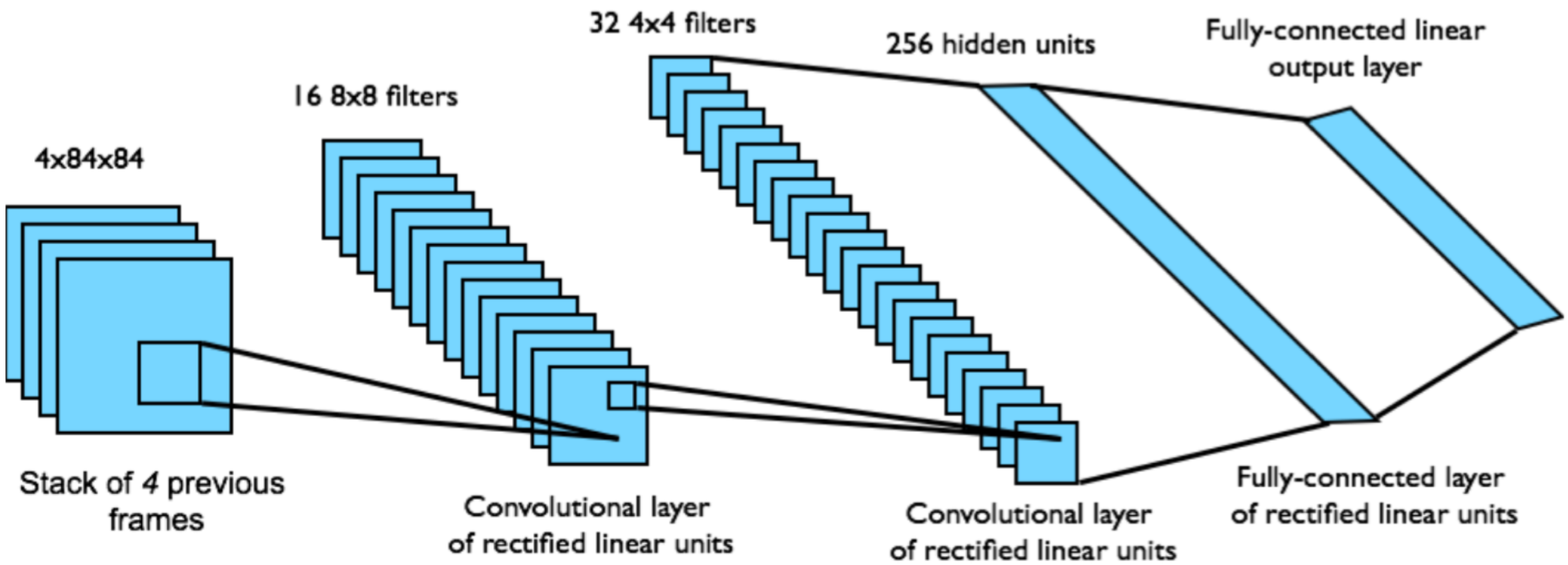
[1]Reference: David Silver, 2015

DQN Extends Function Approximation

- DQN does Q learning with function approximation
- Uses a CNN as the approximator
- Extension
  - Does batch optimization to update the weights
  - Freezes targets over several steps

DQN Extends Function Approximation

- End-to-end learning of values $Q(s, a)$ from pixels s
- Input state s is stack of raw pixels from last 4 frames
- Output is $Q(s, a)$ for 18 joystick/button positions
- Reward is change in score for that step



[1]Reference: David Silver, 2015

DQN Extends Function Approximation

DQN uses experience replay and fixed Q-targets

- Take action a_t according to ϵ-greedy policy

[1]Reference: David Silver, 2015

DQN Extends Function Approximation

DQN uses experience replay and fixed Q-targets

- Take action a_t according to ϵ-greedy policy
- Store transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in replay memory $\mathcal{D}$
- Sample random mini-batch of transitions (s, a, r, s') from $\mathcal{D}$

[1] Reference: David Silver, 2015

DQN Extends Function Approximation

DQN uses experience replay and fixed Q-targets

- Take action a_t according to ϵ-greedy policy
- Store transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in replay memory $\mathcal{D}$
- Sample random mini-batch of transitions (s, a, r, s') from $\mathcal{D}$
- Compute Q-learning targets w.r.t. old, fixed parameters w^-

[1]Reference: David Silver, 2015

DQN Extends Function Approximation

DQN uses experience replay and fixed Q-targets

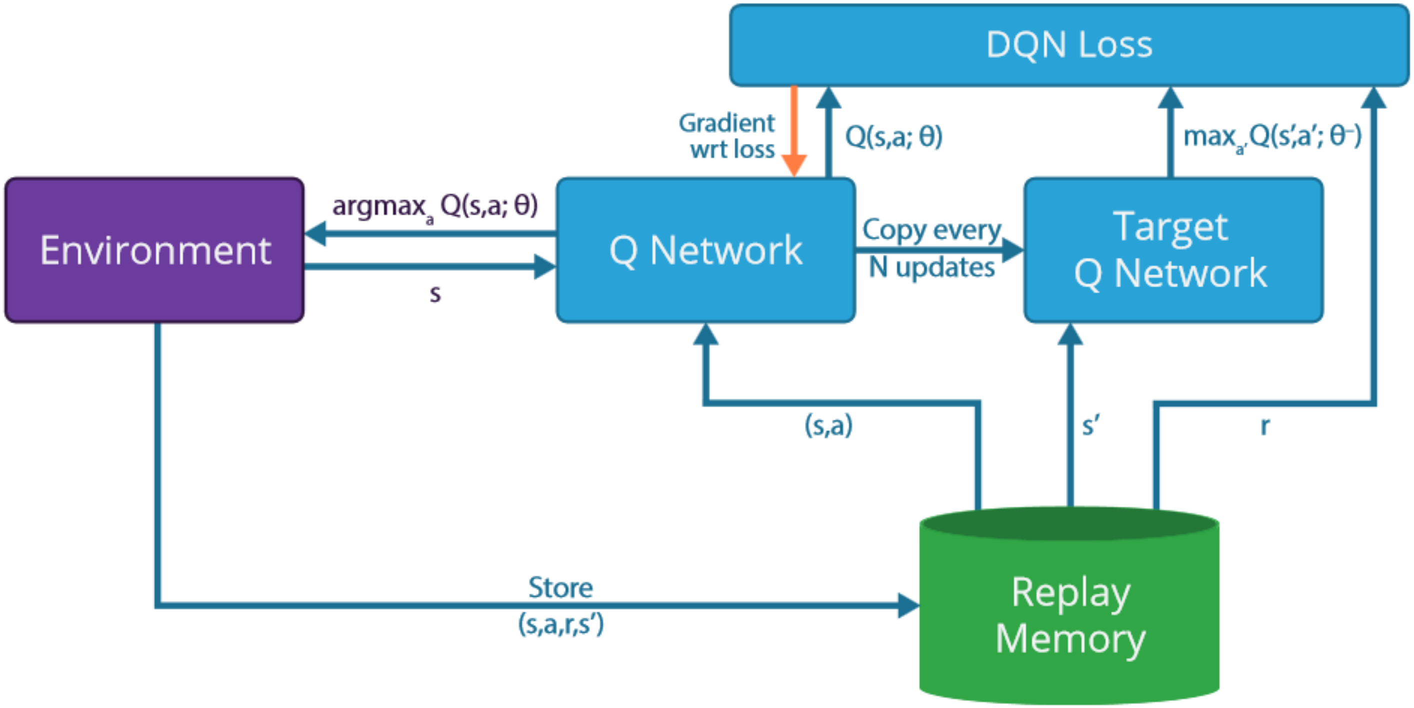
- Take action a_t according to ϵ-greedy policy
- Store transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in replay memory $\mathcal{D}$
- Sample random mini-batch of transitions (s, a, r, s') from $\mathcal{D}$
- Compute Q-learning targets w.r.t. old, fixed parameters w^-
- Optimise MSE between Q-network and Q-learning targets

$$\mathcal{L}_i(w_i) = \mathbb{E}_{s,a,r,s' \sim \mathcal{D}_i}\left[\left(r + \gamma \max_{a'} Q(s', a'; w_i^-) - Q(s, a; w_i)\right)^2\right]$$

- Using variant of stochastic gradient descent

[1]Reference: David Silver, 2015

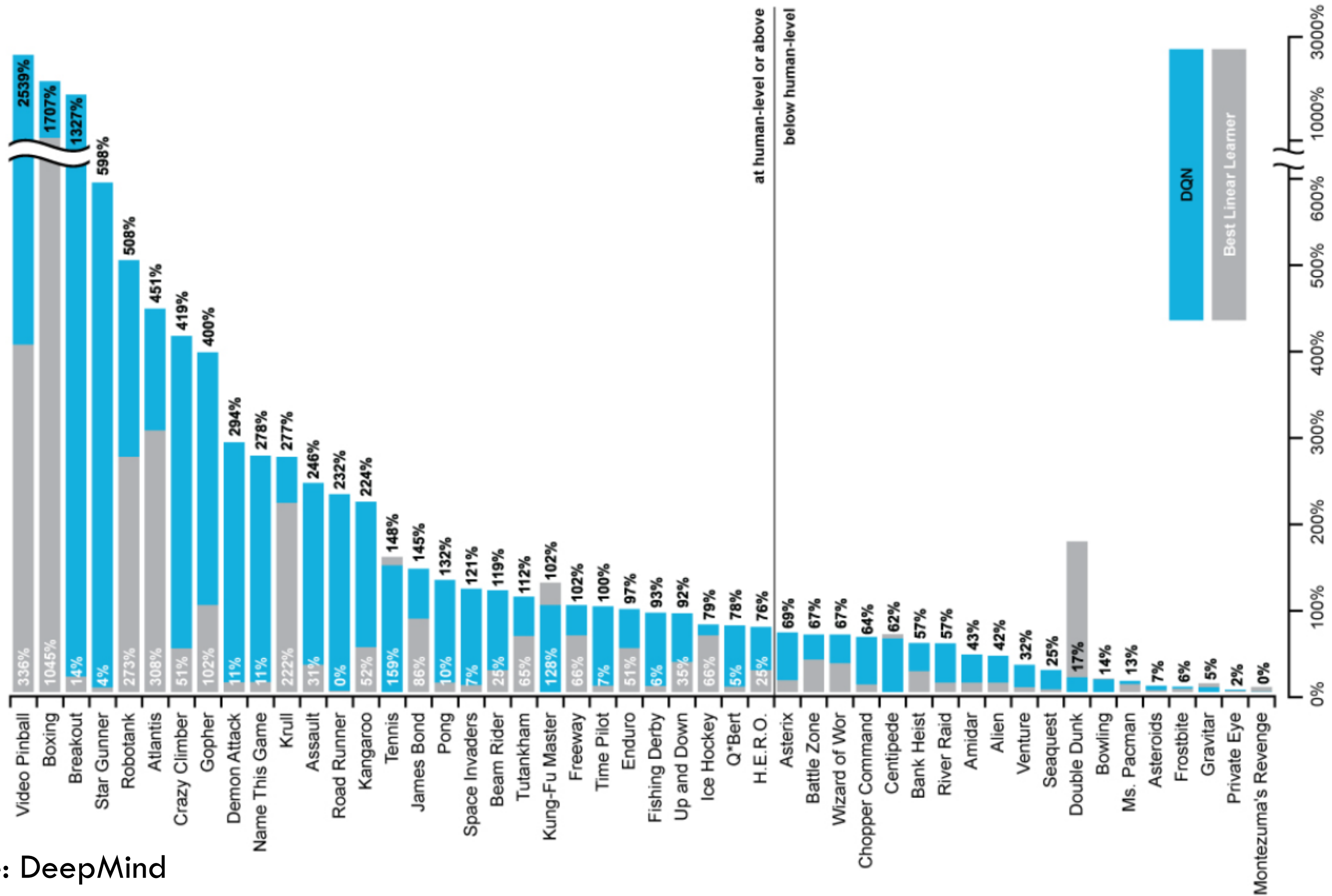
DQN Architecture



[1]Reference: David Silver, 2015

DQN Performance Results

- DQN does not know the rules of the game a-priori
- No feature engineering or hyper-parameter tuning for DQN across games



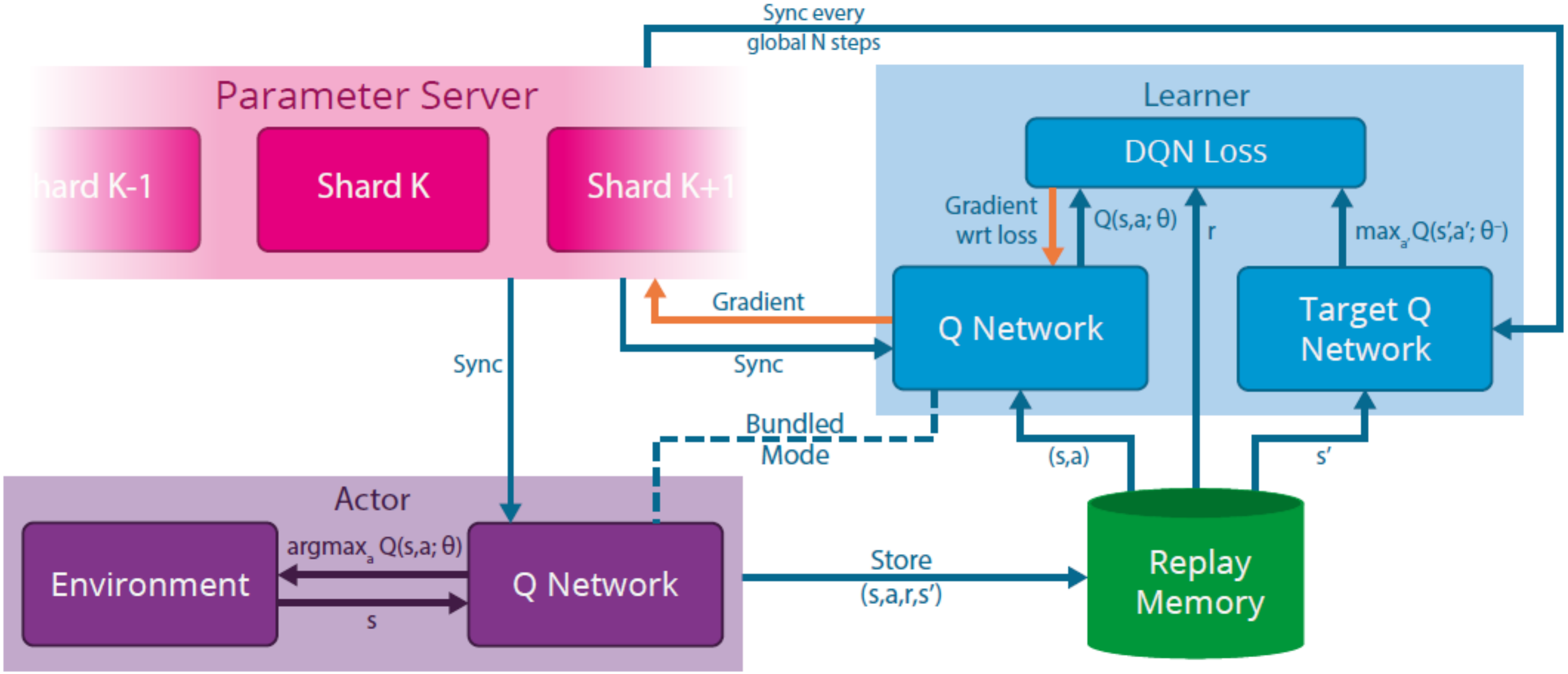
[1]Figure: DeepMind

Did the Extensions Help?

	Replay Fixed-Q	Replay Q-learning	No replay Fixed-Q	No replay Q-learning
Breakout	316.81	240.73	10.16	3.17
Enduro	1006.3	831.25	141.89	29.1
River Raid	7446.62	4102.81	2867.66	1453.02
Seaquest	2894.4	822.55	1003	275.81
Space Invaders	1088.94	826.33	373.22	301.99

[1] Reference: David Silver, 2015

Scalable Version: An Architecture by Google



- 100 actors, 100 learners, and 31 parameter holding machines.
- Reduce compute from 14 days to 6 hours
- This is a 30x speedup using 200x compute power

[1]Reference: Nair, et al. Massively parallel methods for deep reinforcement learning. arXiv:1507.04296

Questions?

Today's Outline

- Value Function Approximation
- Deep Reinforcement Learning
  - DQN for Atari Games
  - AlphaGo for Go

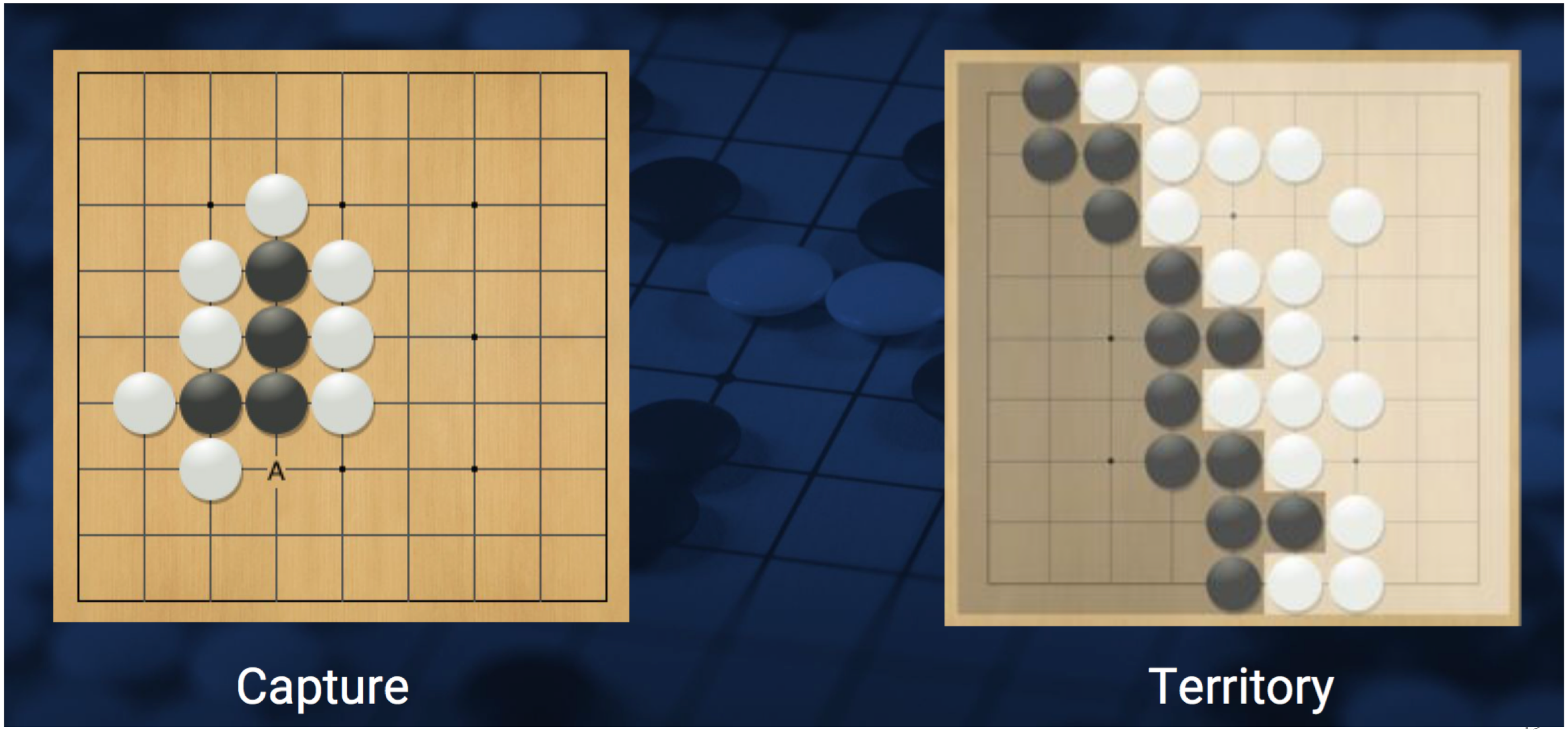
Deep Reinforcement Learning I: AlphaGo

AlphaGo Conquers Go (2016)

[1]Reference: DeepMind, March 2016

The Game of Go

- Go is 2500 years old. Has about 10^{270} states.
- Making it impossible for computers to evaluate who is winning



[1]Reference: DeepMind, IJCAI 2016

The Game of Go

- Go was one of the only classic board games before March 2016, where AI agents were not the best

Program	Level of Play	RL Program to Achieve Level
Checkers	Perfect	*Chinook*
Chess	International Master	*KnightCap / Meep*
Othello	Superhuman	*Logistello*
Backgammon	Superhuman	*TD-Gammon*
Scrabble	Superhuman	*Maven*
Go	Grandmaster	*MoGo*[1], *Crazy Stone*[2], *Zen*[3]
Poker[4]	Superhuman	*SmooCT*

[1] 9×9
[2] 9×9 and 19×19
[3] 19×19
[4] Heads-up Limit Texas Hold'em

[1] Reference: DeepMind, IJCAI 2016

The Forward Search Problem

- Recall the two sequential decision making problems
  - Reinforcement learning
  - Planning
- The forward search problem is a planning problem
  - That is, we know the model of the world
- Useful in the case when we cannot plan everything beforehand
- Focus on what action to take next

The Forward Search Problem for Go

- How good is a position s?
- Reward function (undiscounted):

$$R_t = 0 \text{ for all non-terminal steps } t < T$$

$$R_T = \begin{cases} 1 & \text{if Black wins} \\ 0 & \text{if White wins} \end{cases}$$

- Policy $\pi = \langle \pi_B, \pi_W \rangle$ selects moves for both players
- Value function (how good is position s):

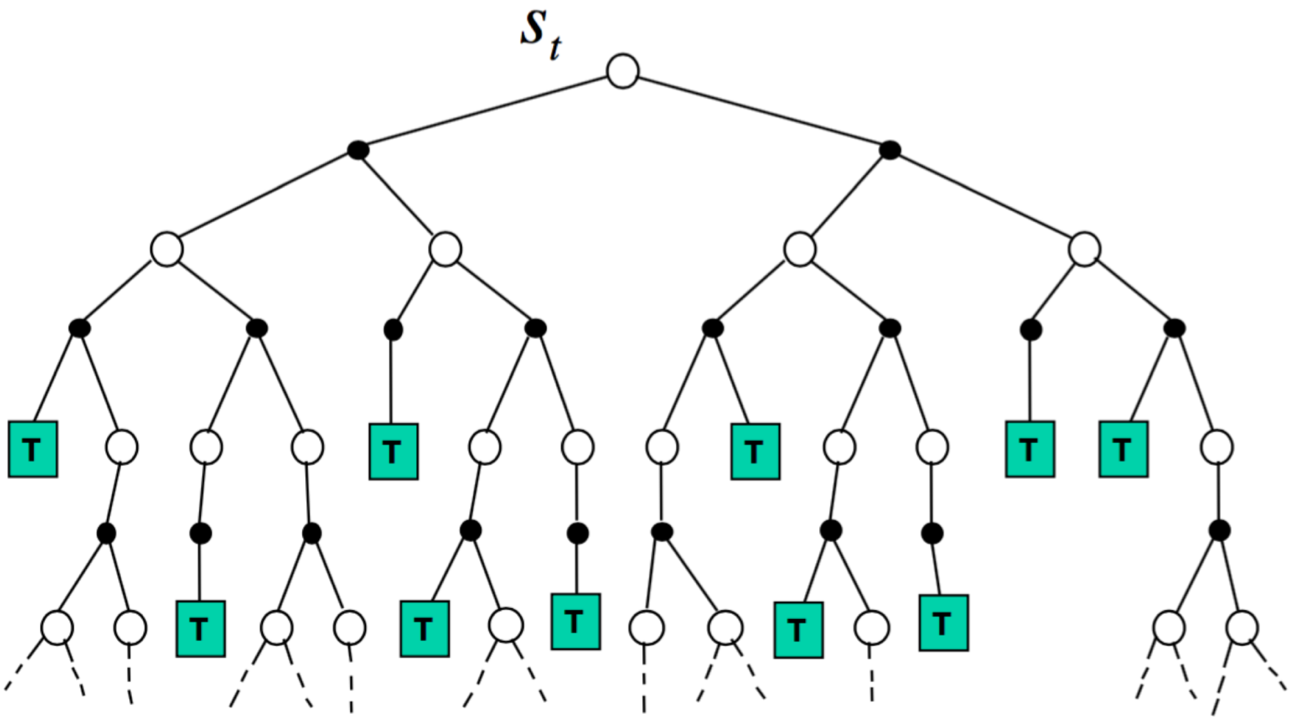
$$v_\pi(s) = \mathbb{E}_\pi \left[R_T \mid S = s \right] = \mathbb{P} \left[\text{Black wins} \mid S = s \right]$$

$$v_*(s) = \max_{\pi_B} \min_{\pi_W} v_\pi(s)$$

[1] Reference: David Silver, 2015

Forward Search Using Simulations

- Forward search algorithms select the best action by lookahead
- They build a search tree with the current state s_t at the root
- Using a model of the MDP to look ahead

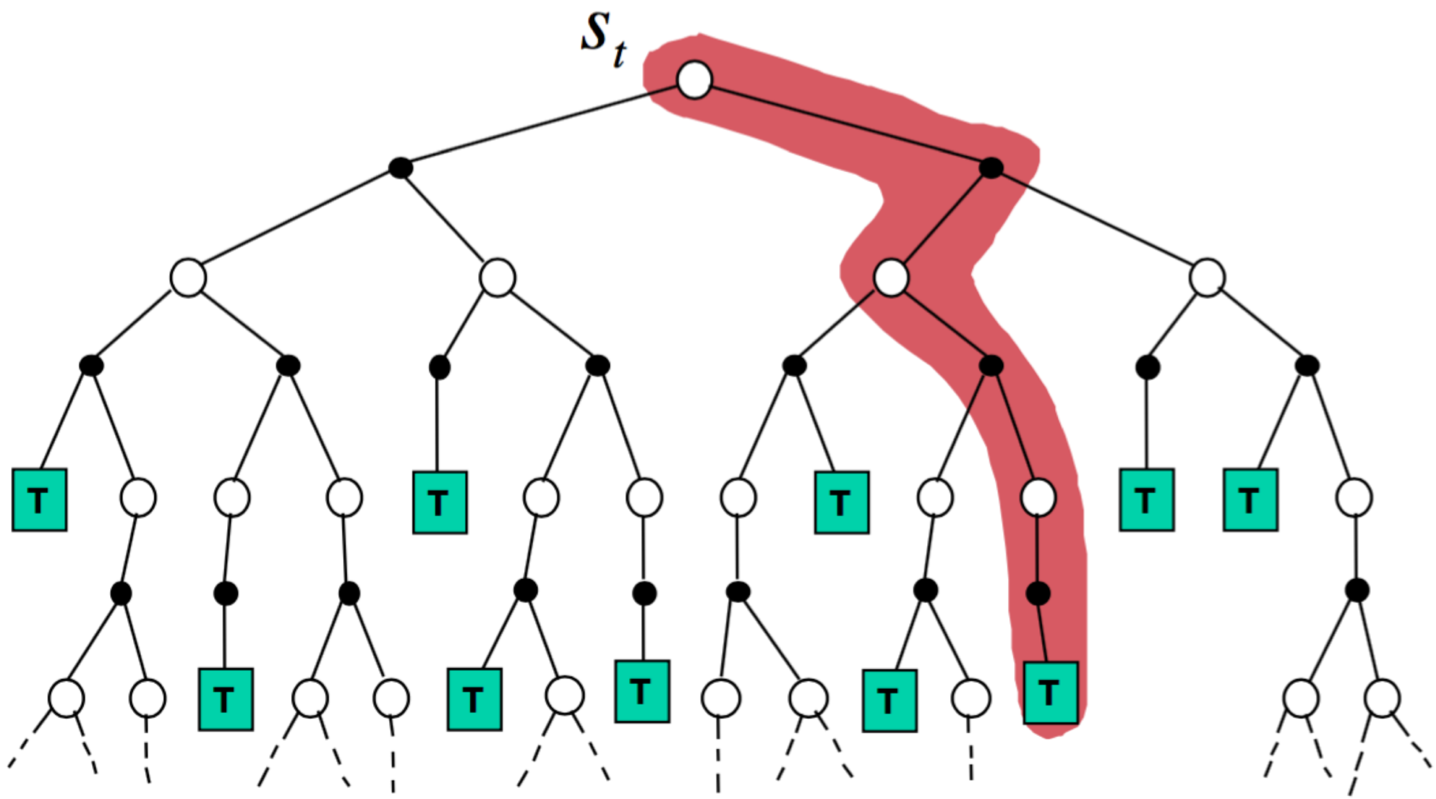


- No need to solve whole MDP, just sub-MDP starting from now

[1]Reference: David Silver, 2015

Forward Search Using Simulations

- Simulate episodes of experience from now with the model
- Apply model-free RL to simulated episodes



[1] Reference: David Silver, 2015

Forward Search Using Simulations

- Simulate episodes of experience from now with the model

$$\{s_t^k, A_t^k, R_{t+1}^k, ..., S_T^k\}_{k=1}^K \sim \mathcal{M}_\nu$$

- Apply model-free RL to simulated episodes

- We will look at two variants
  - Simple Monte Carlo Search
  - Monte Carlo Tree Search

[1]Reference: David Silver, 2015

Simple Monte Carlo Search

- Given a model $\mathcal{M}_\nu$ and a simulation policy π
- For each action $a \in \mathcal{A}$
  - Simulate K episodes from current (real) state s_t

$$\{s_t, a, R_{t+1}^k, S_{t+1}^k, A_{t+1}^k, ..., S_T^k\}_{k=1}^K \sim \mathcal{M}_\nu, \pi$$

  - Evaluate actions by mean return (Monte-Carlo evaluation)

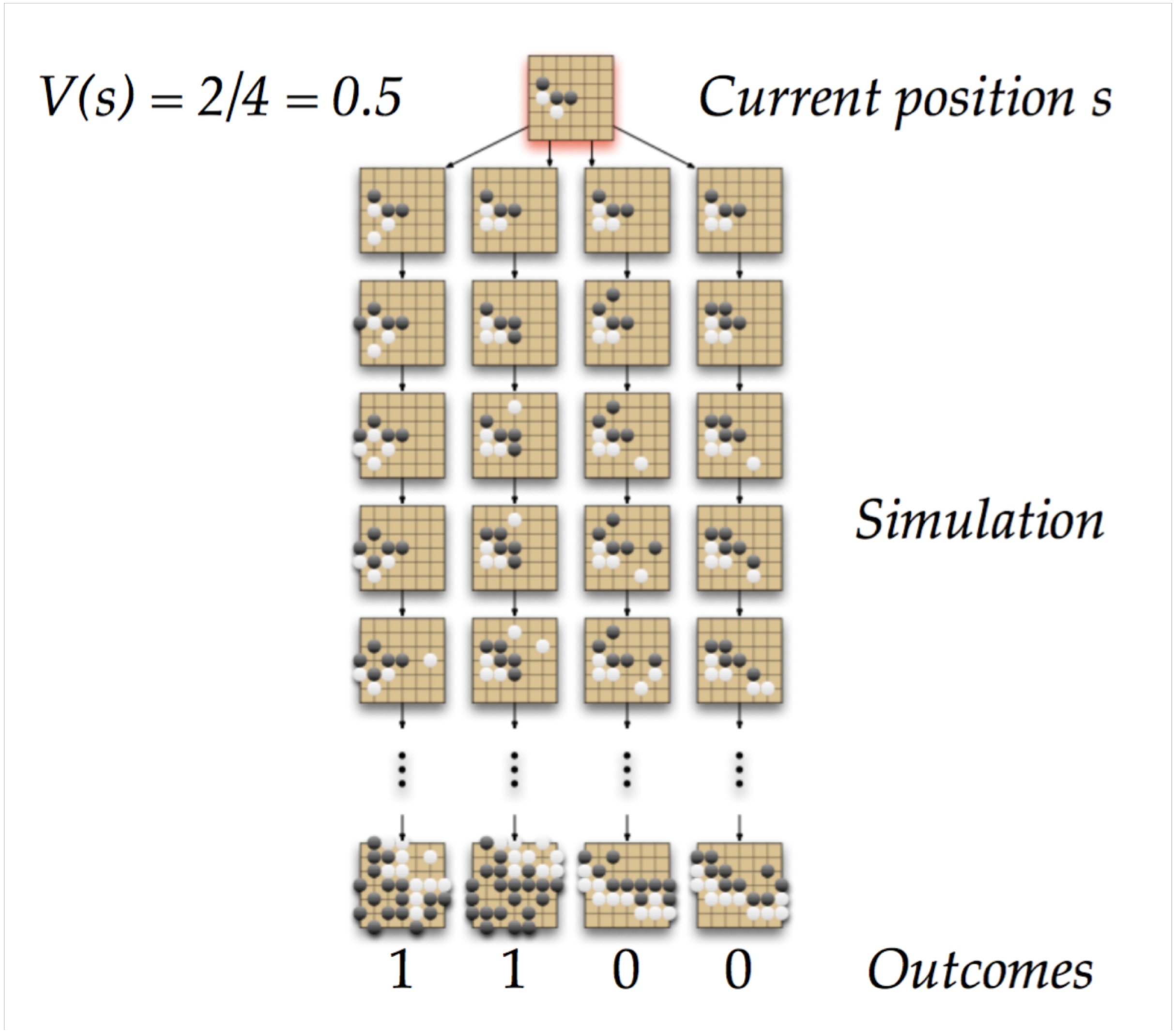
$$Q(s_t, a) = \frac{1}{K} \sum_{k=1}^{K} G_t \xrightarrow{P} q_\pi(s_t, a)$$

- Select current (real) action with maximum value

$$a_t = \underset{a \in \mathcal{A}}{\operatorname{argmax}}\ Q(s_t, a)$$

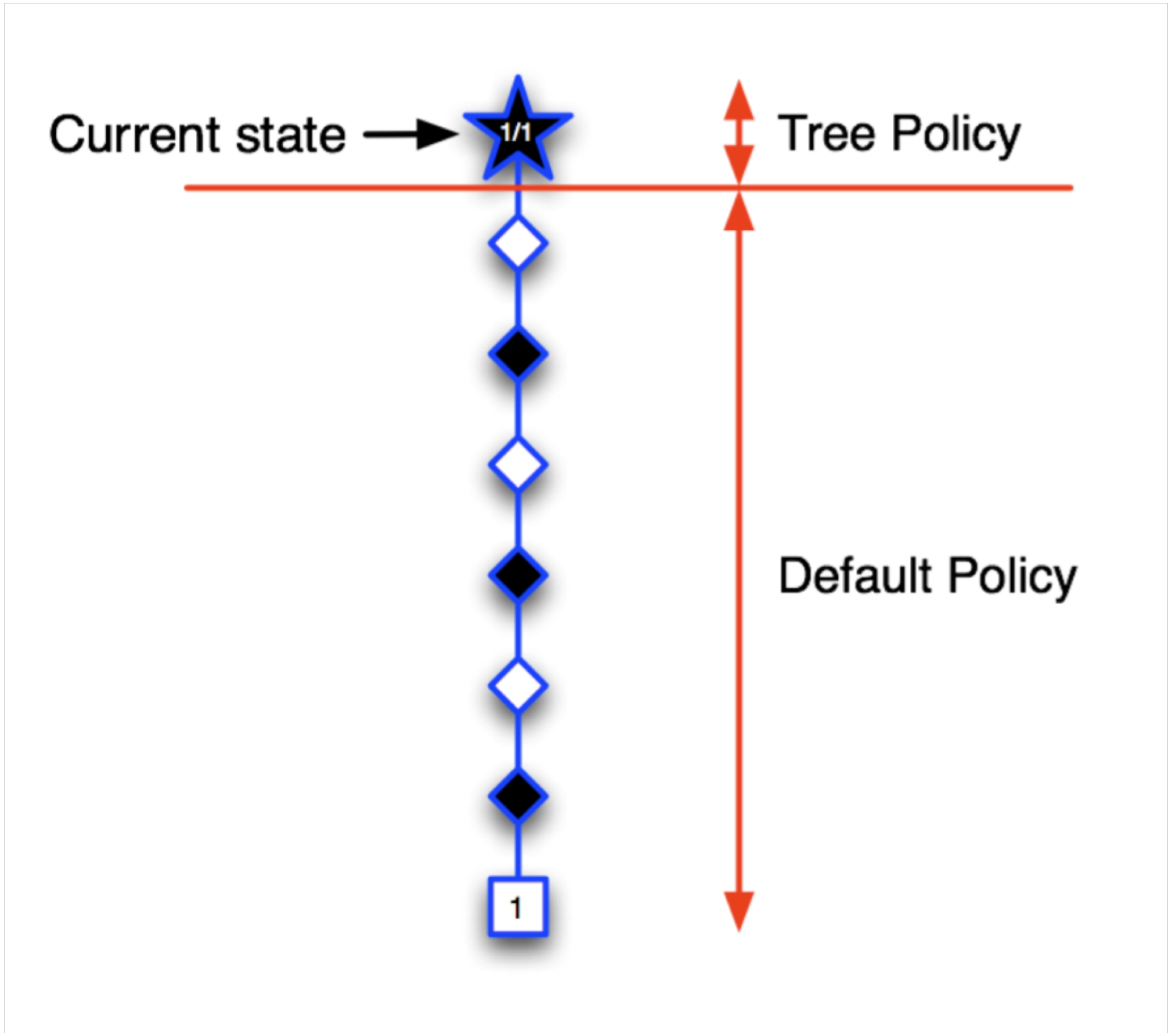
[1]Reference: David Silver, 2015

Simple Monte Carlo Search for Go



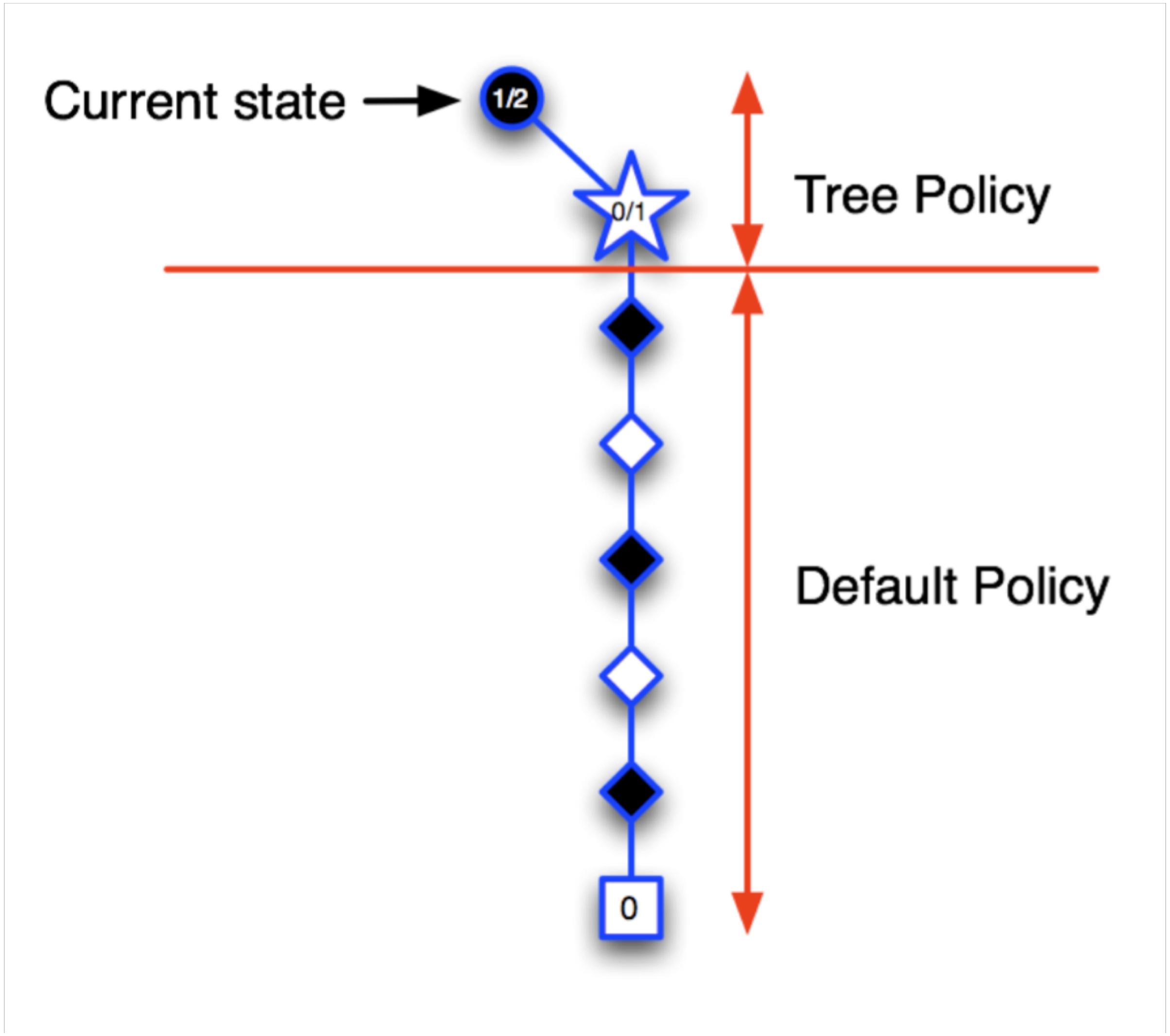
[1]Reference: David Silver, 2015

Monte Carlo Tree Search for Go



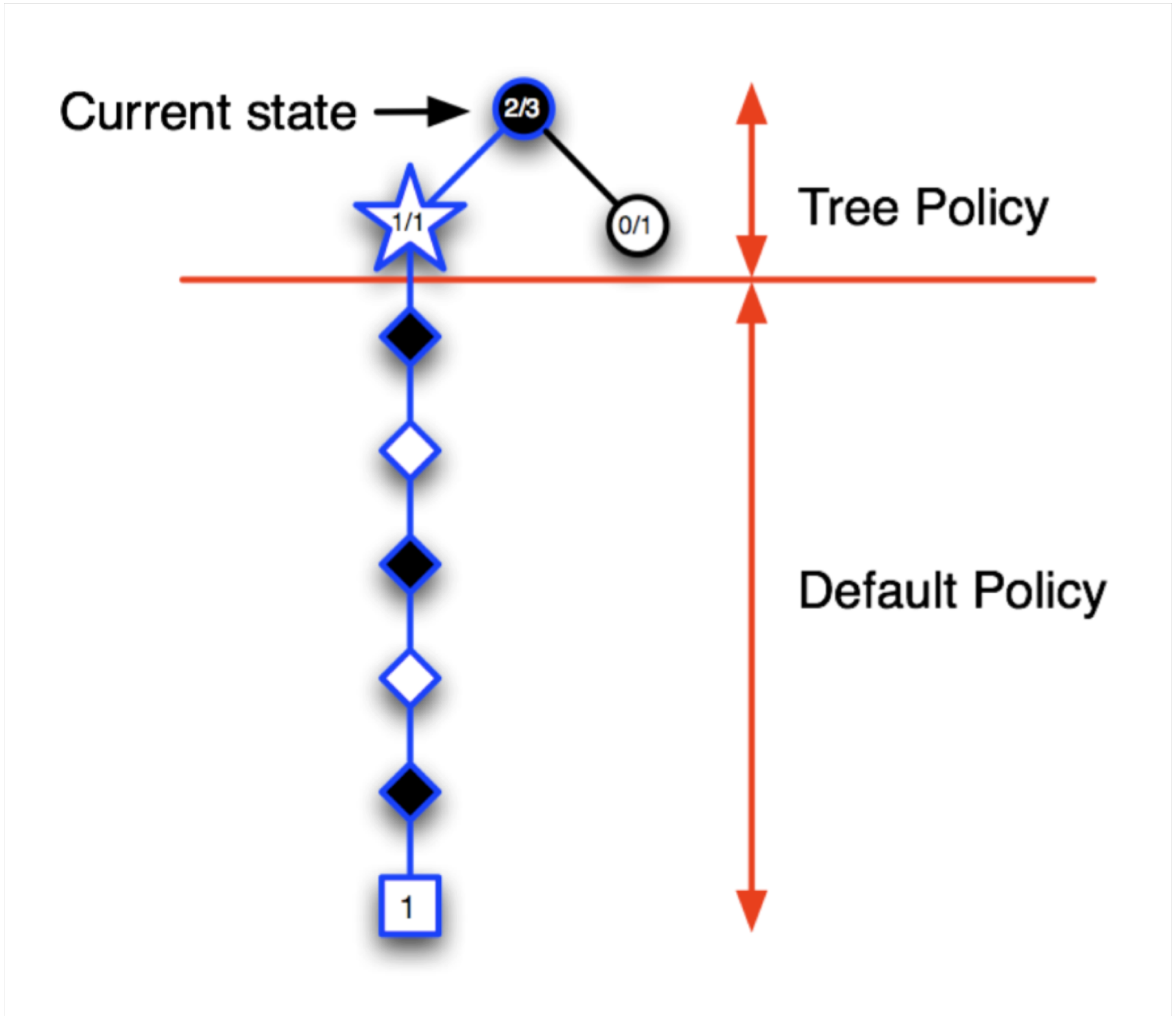
[1]Reference: David Silver, 2015

Monte Carlo Tree Search for Go



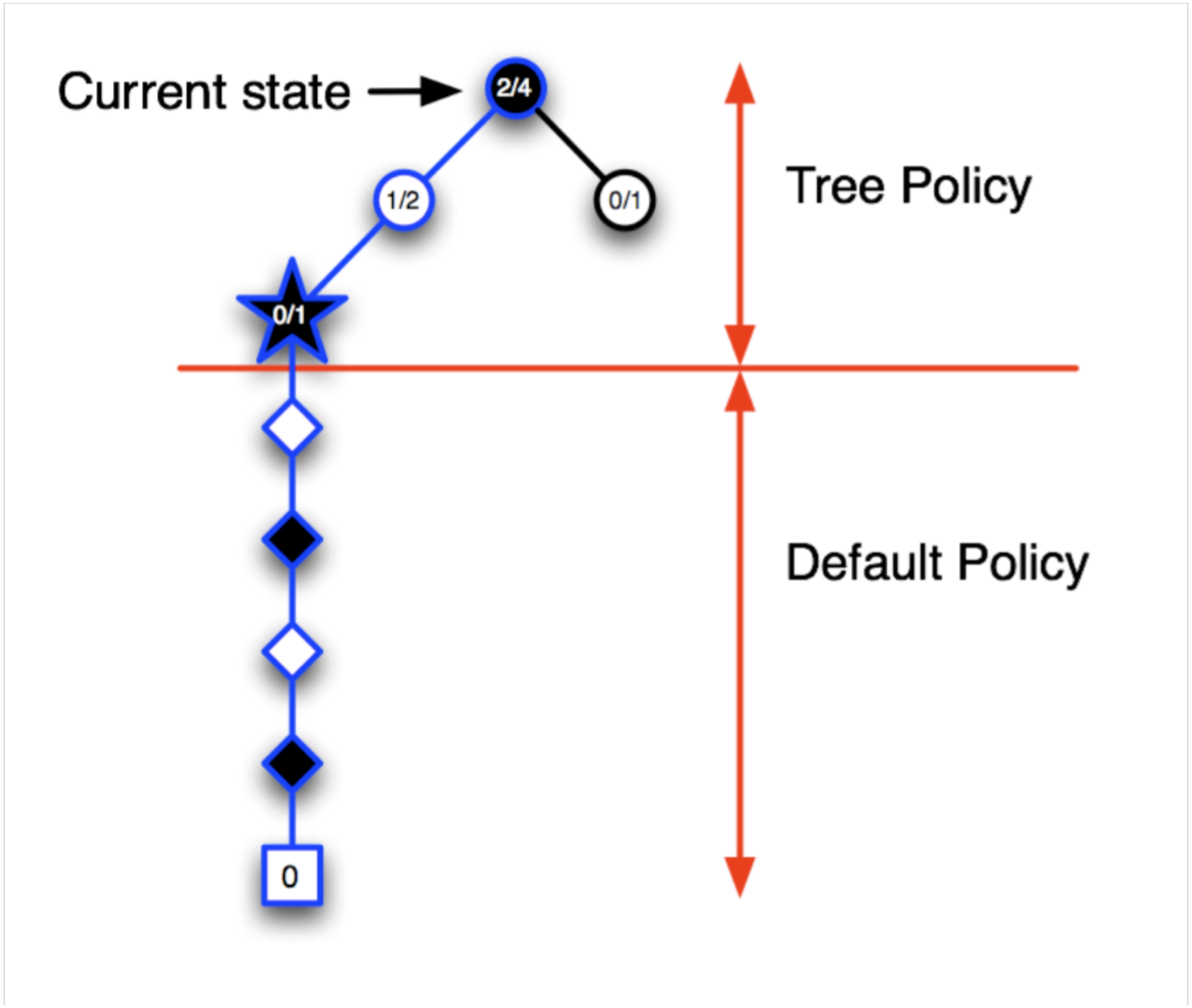
[1]Reference: David Silver, 2015

Monte Carlo Tree Search for Go



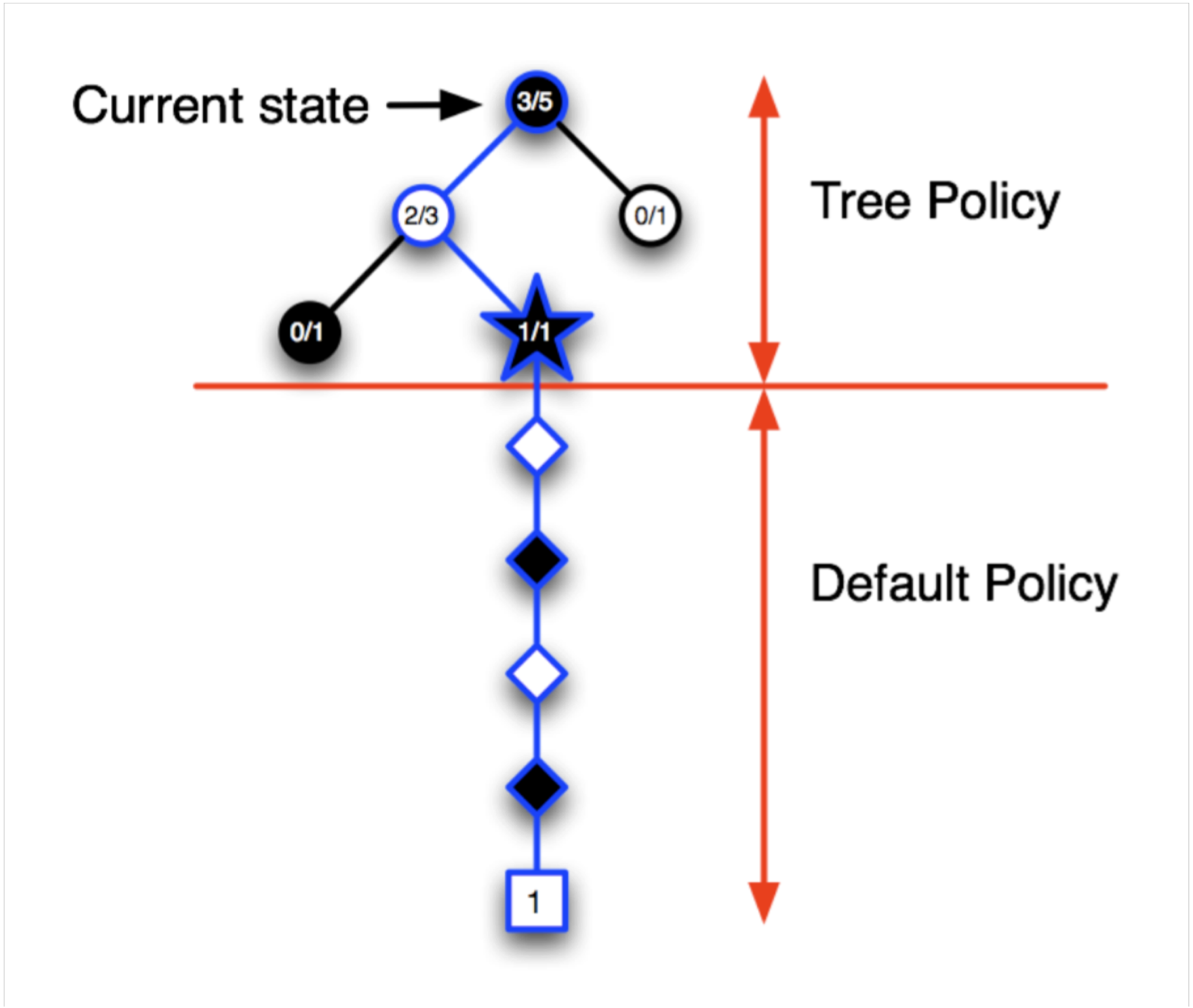
[1]Reference: David Silver, 2015

Monte Carlo Tree Search for Go



[1]Reference: David Silver, 2015

Monte Carlo Tree Search for Go



[1]Reference: David Silver, 2015

Monte Carlo Tree Search: Evaluation

- Given a model $\mathcal{M}_\nu$
- Simulate K episodes from current state s_t using current simulation policy π

$$\{s_t, A_t^k, R_{t+1}^k, S_{t+1}^k, ..., S_T^k\}_{k=1}^K \sim \mathcal{M}_\nu, \pi$$

- Build a search tree containing visited states and actions
- Evaluate states $Q(s, a)$ by mean return of episodes from s, a

$$Q(s, a) = \frac{1}{N(s, a)} \sum_{k=1}^{K} \sum_{u=t}^{T} \mathbf{1}(S_u, A_u = s, a) G_u \xrightarrow{P} q_\pi(s, a)$$

- After search is finished, select current (real) action with maximum value in search tree

$$a_t = \underset{a \in \mathcal{A}}{\text{argmax}}\ Q(s_t, a)$$

[1] Reference: David Silver, 2015

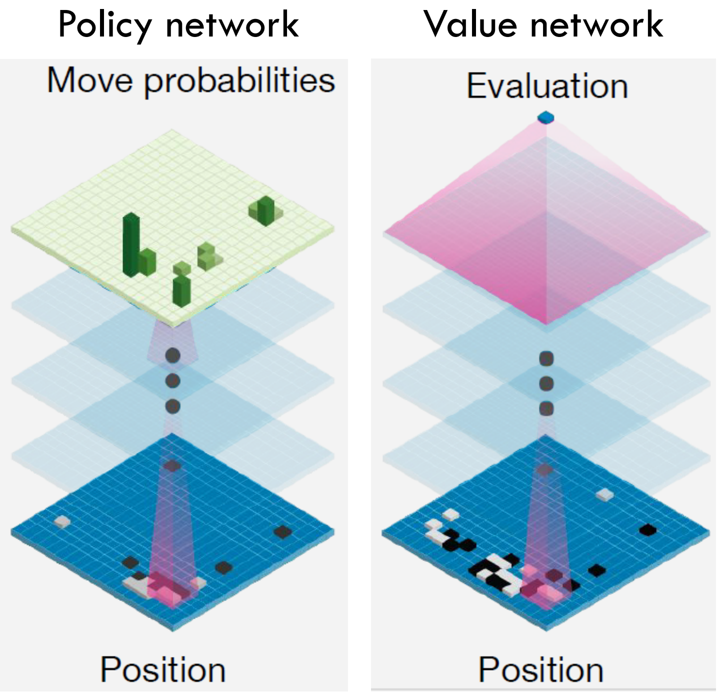
Monte Carlo Tree Search: Simulation

- In MCTS, the simulation policy π improves
- Each simulation consists of two phases (in-tree, out-of-tree)
  - Tree policy (improves): pick actions to maximise $Q(S, A)$
  - Default policy (fixed): pick actions randomly
- Repeat (each simulation)
  - Evaluate states $Q(S, A)$ by Monte-Carlo evaluation
  - Improve tree policy, e.g. by $\epsilon-$greedy(Q)
- Monte-Carlo control applied to simulated experience
- Converges on the optimal search tree, $Q(S, A) \rightarrow q_*(S, A)$

[1]Reference: David Silver, 2015

AlphaGo Extensions

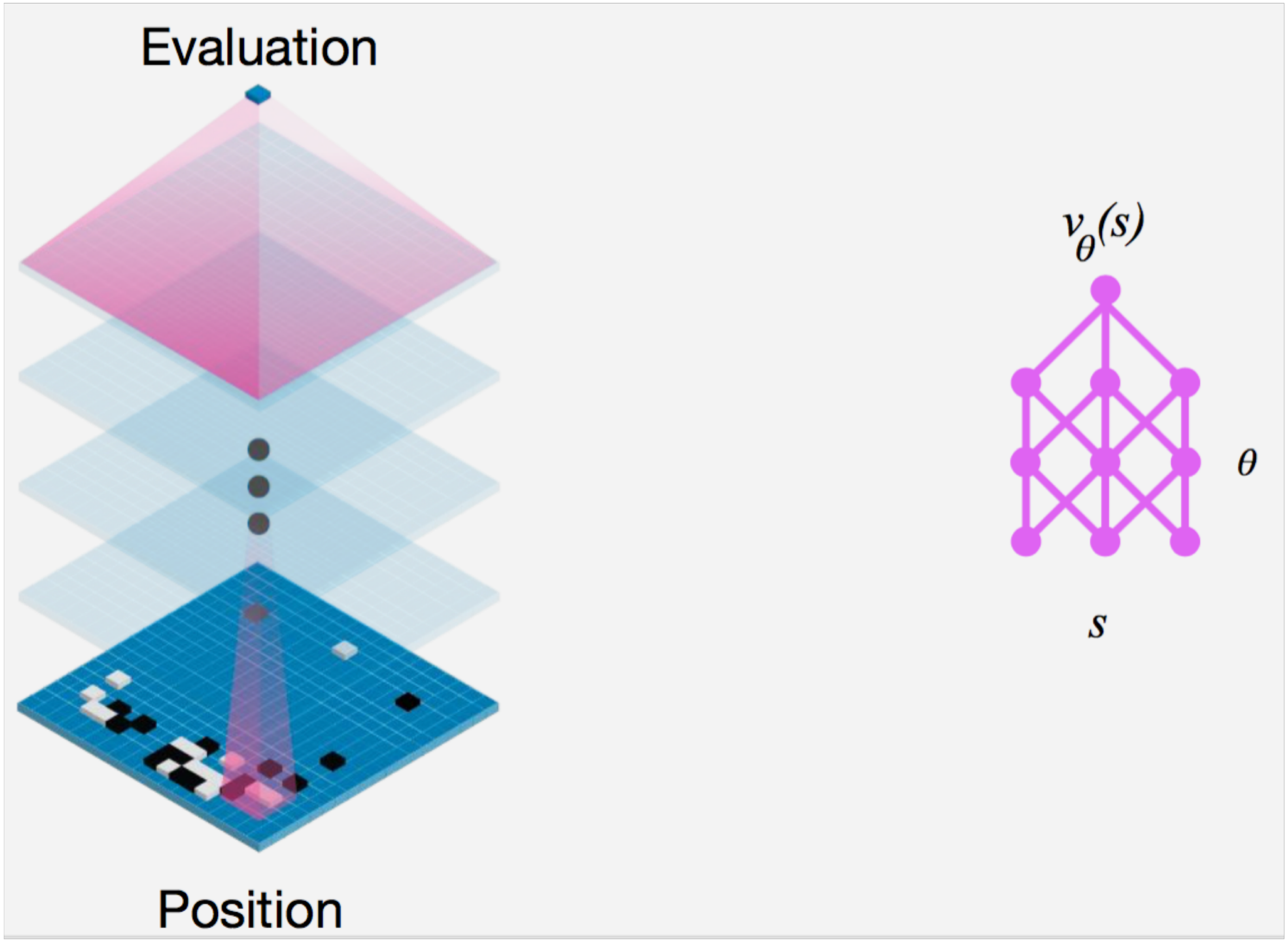
- Uses Monte Carlo tree search for action selection
- But uses a deep policy network and a deep value network to truncate the search tree



[1]Figures: DeepMind

AlphaGo Extensions

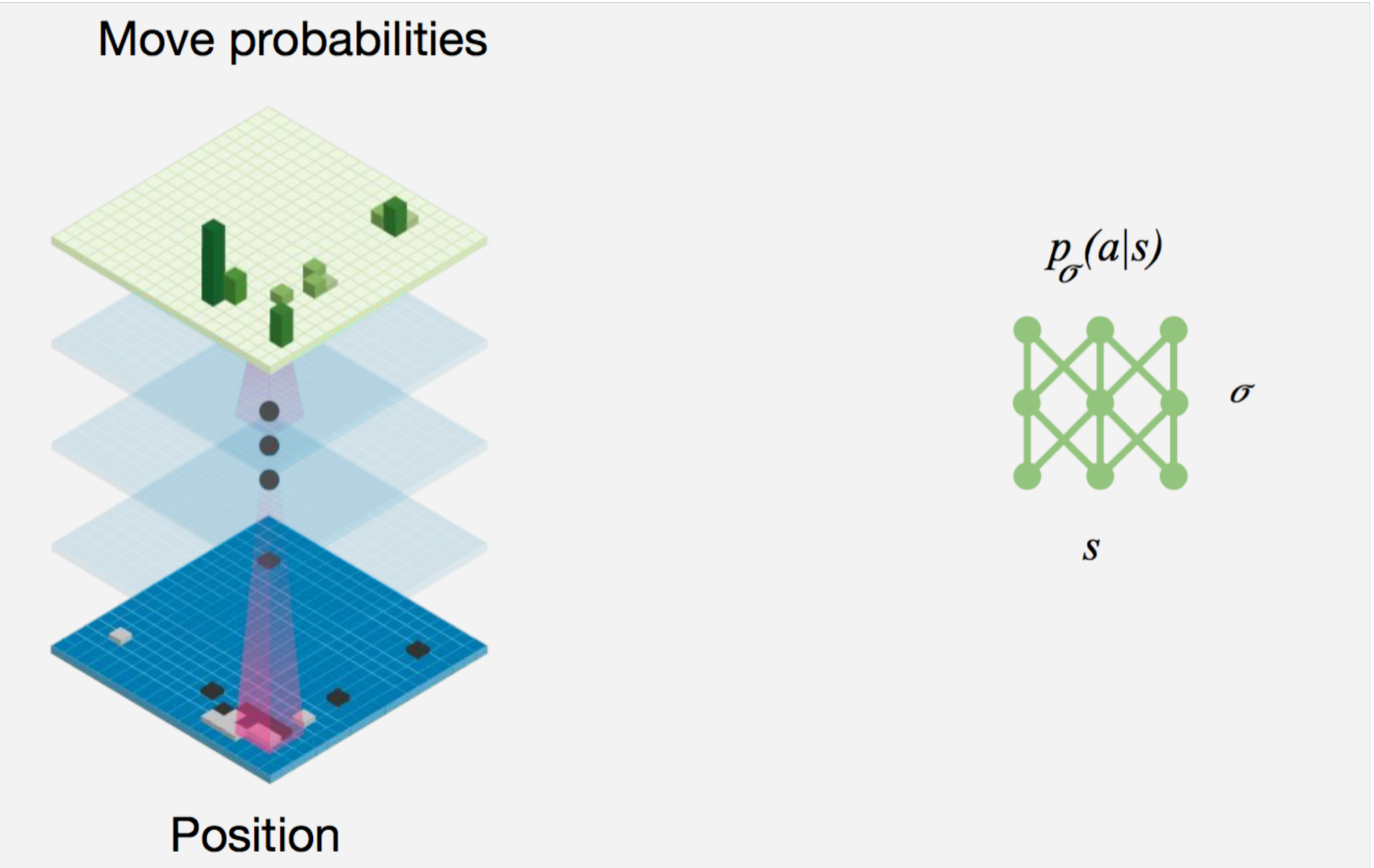
- Value Network



[1]Figures: DeepMind

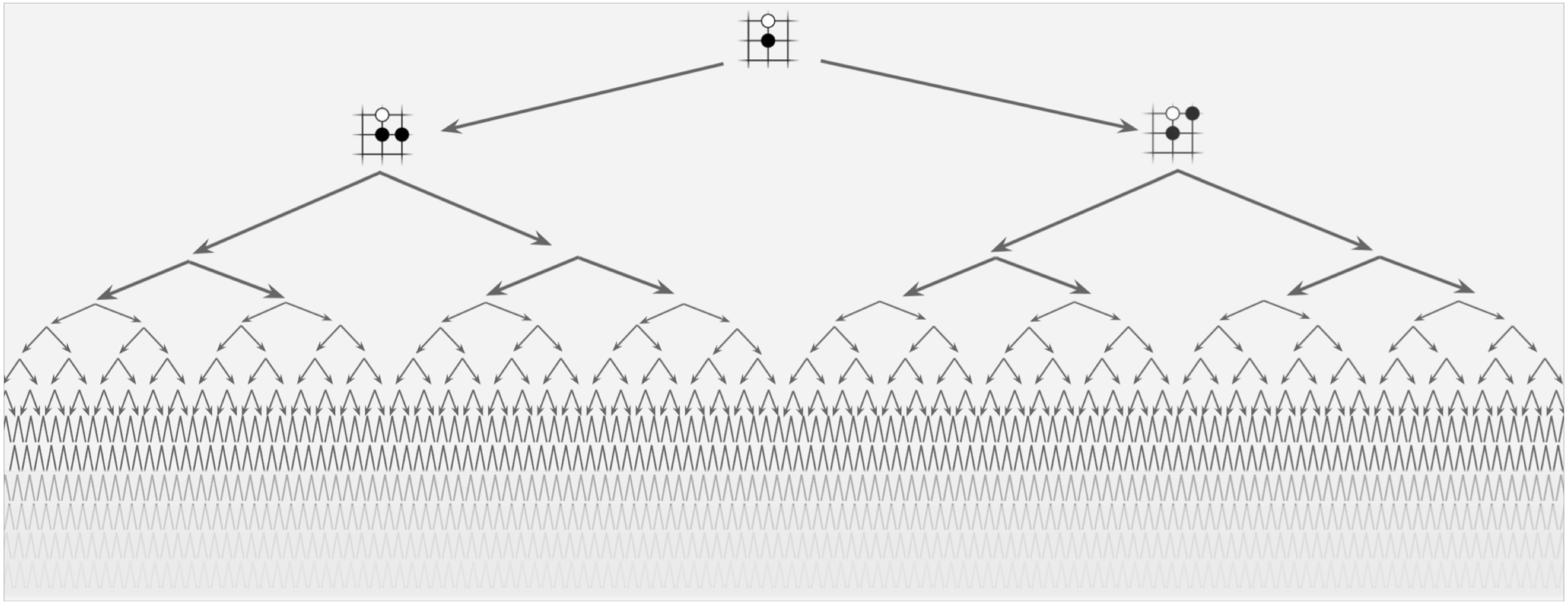
AlphaGo Extensions

- Policy Network



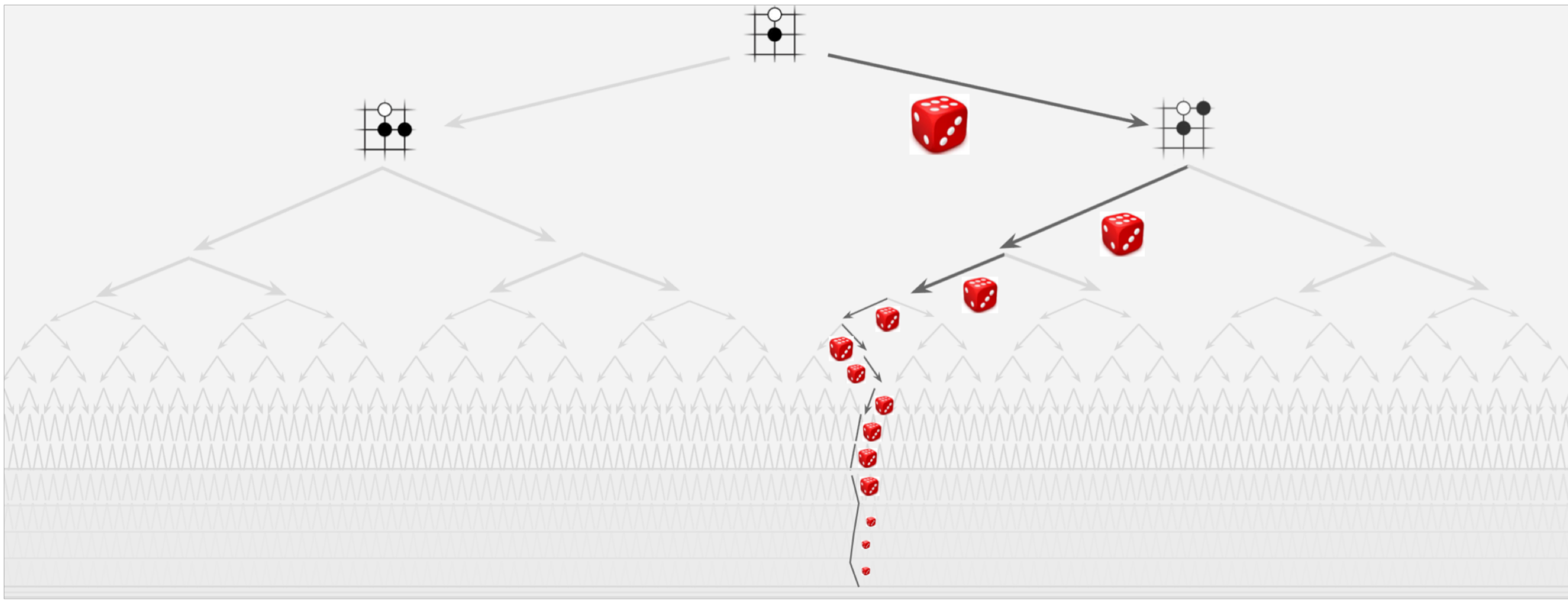
[1]Figures: DeepMind

AlphaGo Extensions



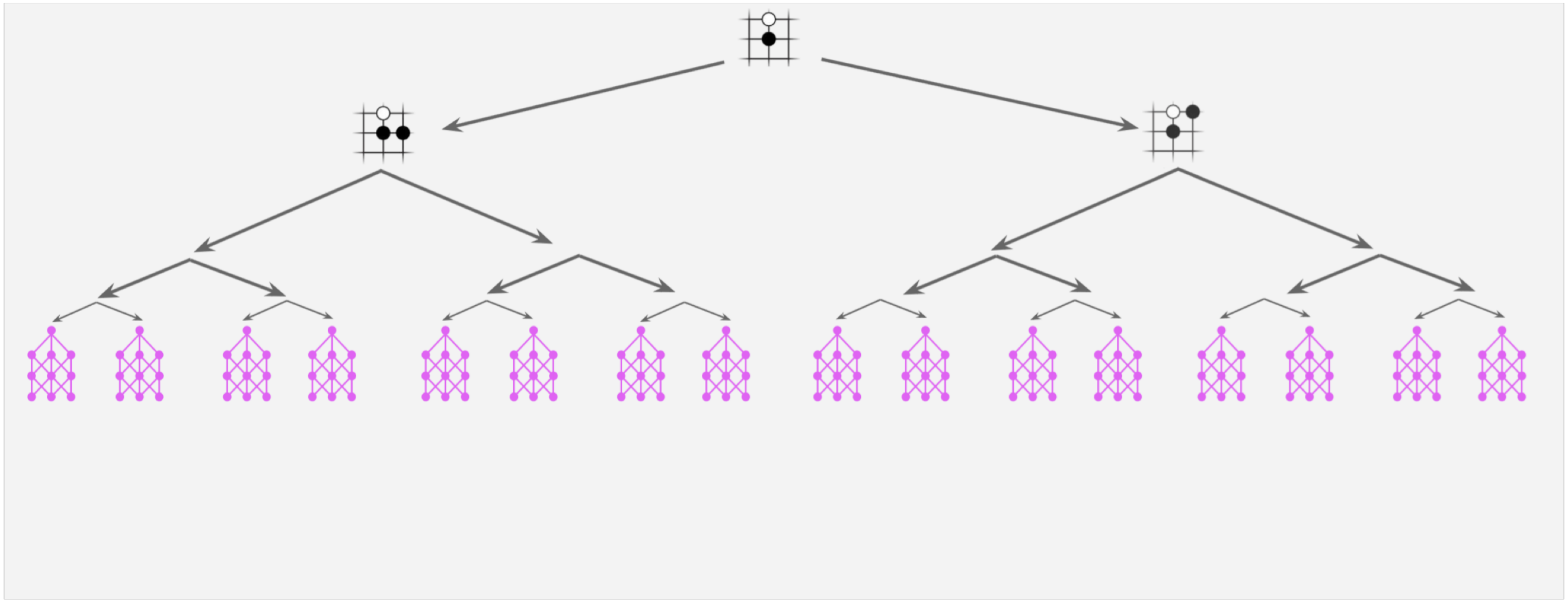
[1]Figures: DeepMind

AlphaGo Extensions



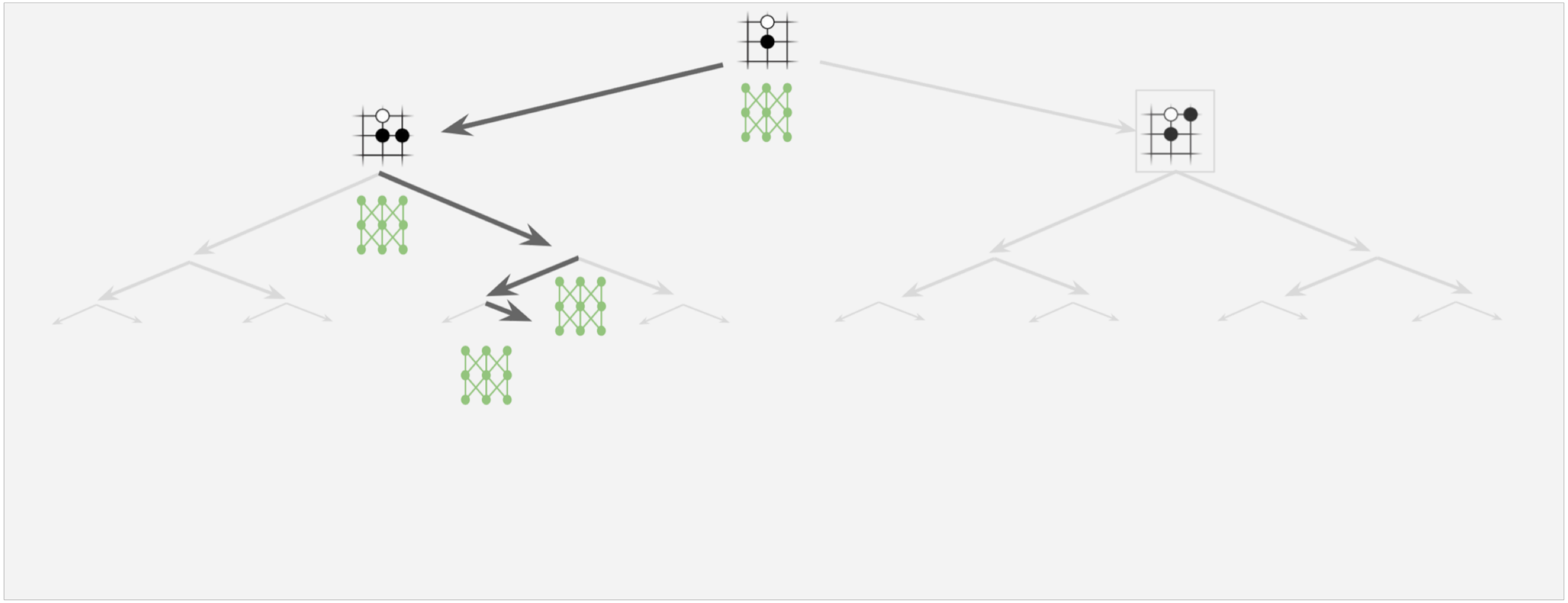
[1]Figures: DeepMind

AlphaGo Extensions



[1]Figures: DeepMind

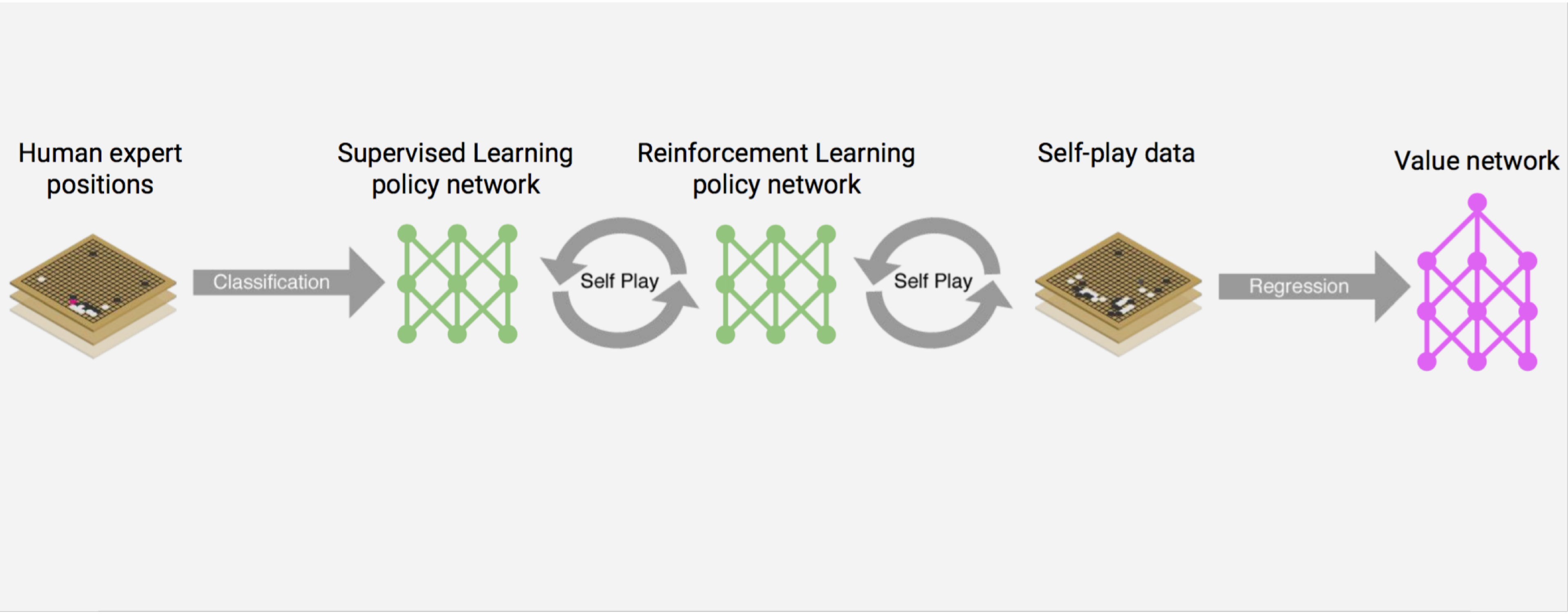
AlphaGo Extensions



[1]Figures: DeepMind

AlphaGo Extensions

- Training the two networks



[1]Figures: DeepMind

AlphaGo Extensions

- The initial policy network

Policy network: 12 layer convolutional neural network

Training data: 30M positions from human expert games (KGS 5+ dan)

Training algorithm: maximise likelihood by stochastic gradient descent

$$\Delta\sigma \propto \frac{\partial \log p_\sigma(a|s)}{\partial \sigma}$$

Training time: 4 weeks on 50 GPUs using Google Cloud

Results: 57% accuracy on held out test data (state-of-the art was 44%)

[1]Figures: DeepMind

AlphaGo Extensions

- The final policy network

Policy network: 12 layer convolutional neural network

Training data: games of self-play between policy network

Training algorithm: maximise wins z by policy gradient reinforcement learning

$$\Delta\sigma \propto \frac{\partial \log p_\sigma(a|s)}{\partial \sigma} z$$

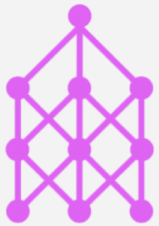
Training time: 1 week on 50 GPUs using Google Cloud

Results: 80% vs supervised learning. Raw network ~3 amateur dan.

[1]Figures: DeepMind

AlphaGo Extensions

- The value network



Value network: 12 layer convolutional neural network

Training data: 30 million games of self-play

Training algorithm: minimise MSE by stochastic gradient descent

$$\Delta\theta \propto \frac{\partial v_\theta(s)}{\partial \theta}(z - v_\theta(s))$$

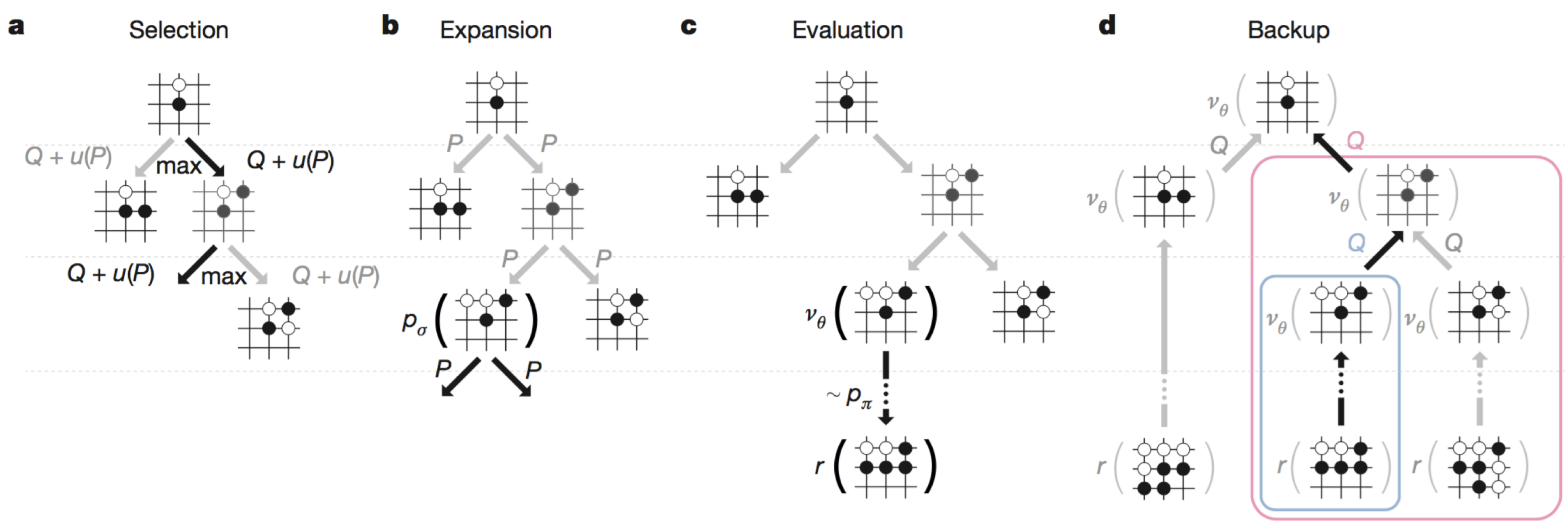
Training time: 1 week on 50 GPUs using Google Cloud

Results: First strong position evaluation function - previously thought impossible

[1]Figures: DeepMind

AlphaGo Extensions

- The MCTS procedure



[1]Figures: DeepMind

AlphaGo

[1]Figure: http://static1.uk.businessinsider.com/image/56e0373052bcd05b008b5217-810-602/

Questions?

Summary

- RL is a great framework to make agents intelligent
  - Specify goals and provide feedback
  - Traditional methods are not scalable

- Function approximation lets us manage scale (number of states)

- Complements deep learning (that solves the perception problem) allowing practical AI agents
  - DQN: Experience replay, freezed Q-targets
  - AlphaGo: Monte Carlo Tree Search with approximations

- Many challenges still remain
  - Inefficient exploration, partial observability etc.

[1]Reference: See https://www.youtube.com/watch?v=WiTnICjWFuw. Demo by Nvidia at CES 2017

Appendix

Sample Exam Questions

- What is the purpose of function approximation?
- Can state value function be function approximated? Is the data in the replay memory i.i.d.?
- What is a search tree? Why is it used?
- How are simulations used in a forward search? (i.e., in a simple Monte Carlo search)
- What are some practical issues with deploying an RL agent in real world?

Additional Resources

- An Introduction to Reinforcement Learning by Richard Sutton and Andrew Barto
  - http://incompleteideas.net/sutton/book/the-book.html
- Course on Reinforcement Learning by David Silver at UCL (includes video lectures)
  - http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html
- Research Papers
  - Deep RL collection: https://github.com/junhyukoh/deep-reinforcement-learning-papers
  - [MKSRVBGRFOPBSAKKWLH2015] Mnih et al. Human-level control through deep reinforcement learning. Nature, 518:529–533, 2015.
  - [SHMGSDSAPLDGNKSLLKGH2016] Silver et al. Mastering the game of Go with deep neural networks and tree search. Nature, 529: 484–489, 2016.

Recap of DQN Extensions

- Experience replay
  - Store transitions in replay memory D
  - Sample a subset from D
  - Optimize mean squared error between
    - $R_{t+1} + \gamma \max_{a \in A} Q(S_{t+1}, a, w)$ and $Q(S_t, A_t, w)$ on this data

- Fixed Q-targets
  - Fix parameter w in $R_{t+1} + \gamma \max_{a \in A} Q(S_{t+1}, a, w)$ for several steps

Cons of RL

- In general, Reinforcement Learning requires experiencing the environment many many times
- This is because it is a trial and error based approach

- May be impractical for many complex tasks
- Unless one has access to simulators where an RL agent can practice a billon+ times

RL Topics Not Covered

- Partial observability of states
- Monte Carlo methods
  - Example: ϵ-Greedy Policy Iteration with Monte Carlo estimation
- Temporal difference methods
  - Example: SARSA(λ)
- Policy function approximation
- Model based methods
- …