DR 101

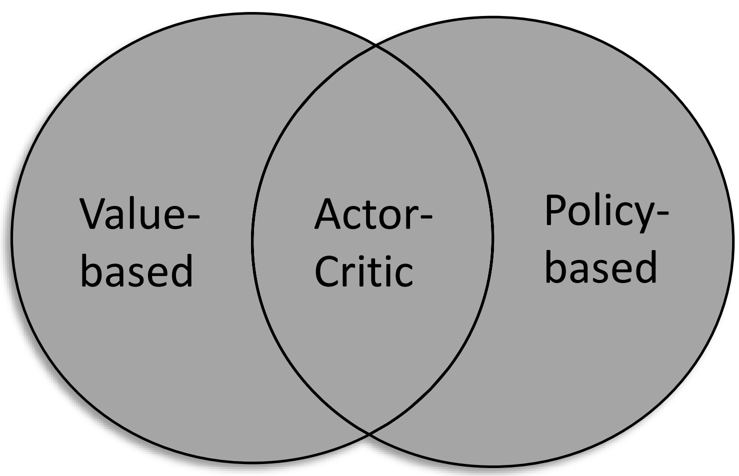
Policy Gradient, Model-based RL

2019-11-16

김영삼

Value-based vs policy-based methods

- Previously, all the RL methods are based on state/action value based methods
- Policy gradient methods directly learn a parameterized policy from an experience
- A value function may still be used, but not for action selection
- Actor-Critic methods learn both value function and policy



Example: Americano or Latte?

- Value-based
  - Americano if $q(S,$ 'Americano'$)$ > $q(S,$ 'Latte'$)$
  - Latte otherwise
- Policy-based
  - Americano if $\pi($'Americano'$|\, S)$ > $\pi($'Latte'$|\, S)$
  - Latte otherwise

VS

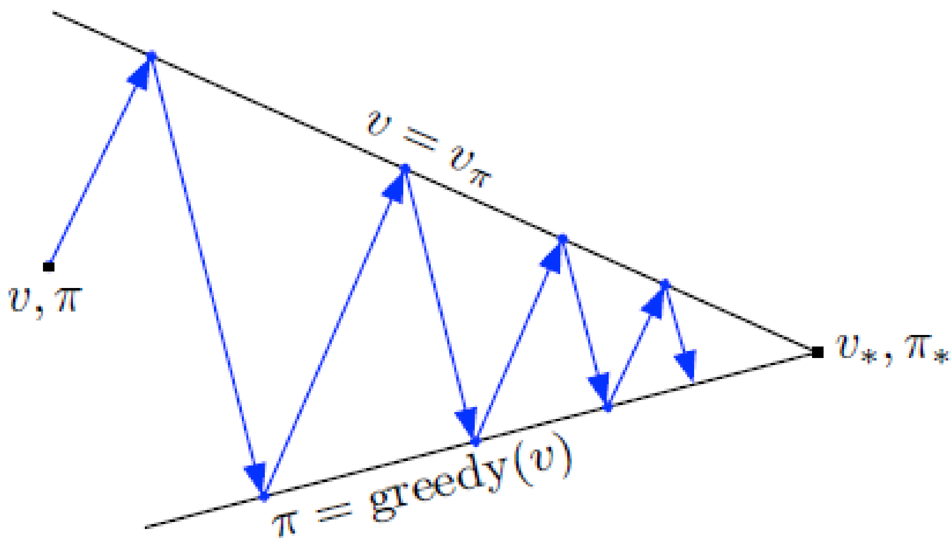
Policy-based approach

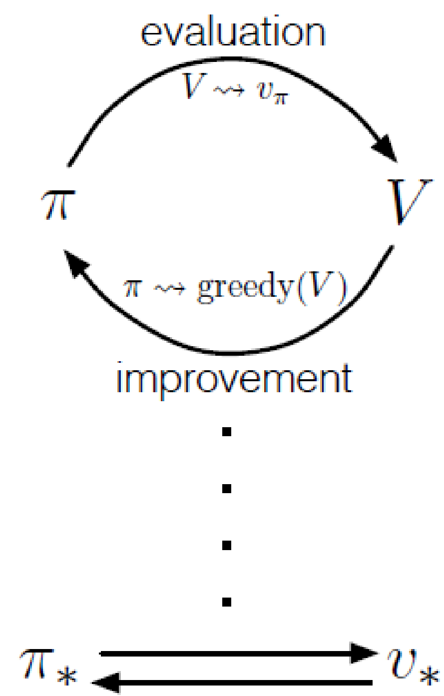
- Policy gradient methods directly parametrize the policy π
- Thus, we write
$$\pi(a|s,\theta) = \Pr\{A_t = a|S_t = s, \theta_t = \theta\}$$
- The policy provides a probability that action a is taken given state s with parameter θ at time t
- $\theta \in \mathbb{R}^d$ denotes the policy's parameter vector

Pros and Cons of Policy-based RL

- Pros
  - Useful when dealing with continuous action spaces
  - Policy will be deterministic with soft-max function
  - Choice of policy parametrization
- Cons
  - Will converge to a local optimum
  - Typically require more experiences
  - Learning might be with high variance

GPI for continuous action space





Policy evaluation → Estimate value function
Policy improvement → Generate better policy (possible?)

- **In policy-based approach, parameters of π are optimized in direction to more greedy returns (problem solved!)**

Example: Rock-Paper-Scissors

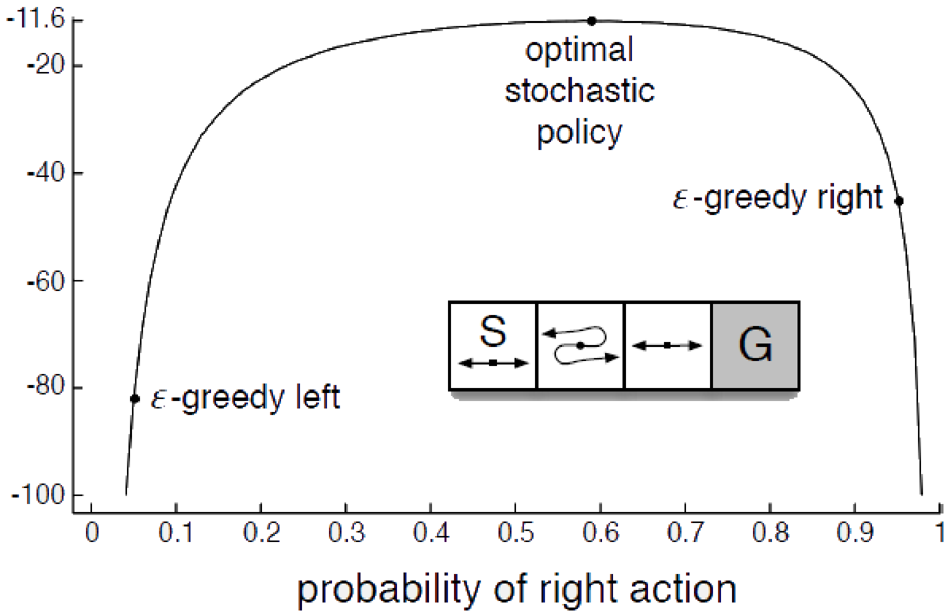
- Policy gradient method is convenient for stochastic policy learning
- The optimal policy for rock-paper-scissors is to draw a hand with the probability of 1/3

		
0.33	0.33	0.33

Example: Short-corridor

- Small corridor gridworld with switched actions
- Reward is -1 per step
- State is only one with left/right actions

$$J(\boldsymbol{\theta}) = v_{\pi_{\boldsymbol{\theta}}}(\mathsf{S})$$



Policy gradient ascent

- Objective function
  - $J(\theta) = \sum_s d^{\pi_\theta}(s) \sum_a \pi_\theta(a|s) R_s^a$
- Optimization
  - $\max_\theta J(\theta)$
  - We want $J(\theta)$ to be maximized
- Thus, we use stochastic gradient ascent
  - $\Delta\theta = \alpha \nabla_\theta J(\theta)$ ← Policy gradient
  - $\nabla_\theta J(\theta) = \left[\frac{\partial J(\theta)}{\partial \theta_1}, \frac{\partial J(\theta)}{\partial \theta_2}, \dots, \frac{\partial J(\theta)}{\partial \theta_d}\right]^T$
  - α is a learning-rate

Optimization

- We want to find a policy θ that create a trajectory
  - $\tau = (s_1, a_1, s_2, a_2, \ldots, s_H, a_H)$
- An identity:
  - $\mathrm{f(x)}\nabla_\theta \log \mathrm{f(x)} = \mathrm{f(x)}\frac{\nabla_\theta \mathrm{f(x)}}{\mathrm{f(x)}} = \nabla_\theta \mathrm{f(x)}$
- Thus, it goes with policy function as:
  - $\pi_\theta(\tau)\nabla_\theta \log \pi_\theta(\tau) = \nabla_\theta \pi_\theta(\tau)$
- Policy gradient:
  - $J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta(\tau)}[r(\tau)]$
  - $\nabla_\theta J(\theta) = \int \pi_\theta(\tau)\nabla_\theta \log \pi_\theta(\tau) r(\tau)\, d\tau$

    $= \mathbb{E}_{\tau \sim \pi_\theta(\tau)}[\nabla_\theta \log \pi_\theta(\tau) r(\tau)]$

Policy objective functions

- General episodic case
  - $J(\theta) = v_{\pi_\theta}(S_0)$
  - $v_{\pi\theta}$ is the true value function for π_θ
  - Every episode starts in state S_0
- Continuing case
  - $J(\theta) = \sum_s d^{\pi_\theta}(s) V^{\pi_\theta}(s)$
  - In this case, we use average value
- When using average reward per time-step
  - $J(\theta) = \sum_s d^{\pi_\theta}(s) \sum_a \pi_\theta(a|s) R_s^a$
- $d^{\pi_\theta}(s)$ is stationary distribution of Markov chain

Policy Gradient Theorem

- This theorem provides an analytic expression for the gradient
- For any differentiable policy $\pi_\theta(s, a)$, the policy gradient is as below:

$$\begin{aligned}\nabla J(\theta) &\propto \sum_s d(s) \sum_a q_{\pi_\theta}(s, a) \nabla \pi_\theta(a|s) \\ &= \mathbb{E}_\pi\left[\sum_a q_{\pi_\theta}(s, a) \nabla \pi_\theta(a|s)\right]\end{aligned}$$

- See the boxes for the detailed proofs (Sutton & Barto, 2018, p. 325; p. 334)

Intepretation of policy gradient

- $\nabla J(\theta) \propto \mathbb{E}_{\pi}\left[\sum_a q_{\pi_\theta}(s,a) \nabla \pi_\theta(a|s)\right]$
- It measures a likelihood of the observed experience
- That is, it measures how likely the trajectory is under the current policy
- Thus, maximizing the likelihood multiplied with the rewards means **increasing the likelihood of a successful policy**

Score function

- Assume the policy π_θ is differentiable
- We need to get $\nabla\pi(a|s,\theta)$
- Since following identity holds:

$$\begin{aligned}\nabla\pi_\theta(s,a) &= \pi_\theta(s,a)\frac{\nabla\pi_\theta(s,a)}{\pi_\theta(s,a)} \\ &= \pi_\theta(s,a)\nabla\ln\pi_\theta(s,a)\end{aligned}$$

- Thus, the score function is $\nabla\ln\pi_\theta(s,a)$

Soft-max policy

- Soft-max policy is often used:

$$\pi_\theta(s,a) = \frac{e^{\mathrm{f}(s,a,\theta)}}{\sum_b e^{\mathrm{f}(s,b,\theta)}}$$

- And if action values are from linear combination of features such as $\mathrm{f}(\mathrm{s},\mathrm{a},\theta) = \phi(s,a)^T\theta$,

$$\pi_\theta(s,a) \propto e^{\phi(s,a)^T\theta}$$

- The score function is

$$\nabla \ln \pi_\theta(s,a) = \phi(s,a) - \mathbb{E}_{\pi_\theta}[\phi(s,\cdot)]$$
$$= \phi(s,a) - \sum_b \pi_\theta(s,b)\phi(s,b)$$

Gaussian policy

- Gaussian policy is often used for continuous action spaces
- Mean is a linear combination of state features $\mu(s) = \phi(s)^T \theta$
- Variance may be fixed α^2 or can also parametrized
- Policy is Gaussian, a$\sim N(\mu(s), \alpha^2)$
- The score function is

$$\nabla \ln \pi_\theta(s, a) = \frac{\big(a - \mu(s)\big)\phi(s)}{\alpha^2}$$

From D. Silver's Lecture 7 (p. 18)

Measuring distance of polices

- Policy gradient methods make a small change of a policy parameter, improving the policy
- But, how we can measure the closeness between the current policy and the updated policy?
- In statistics, such distances can be approximated by its second order Taylor expansion:
  - $D_{KL}(\rho_\theta, \rho_{\theta+\Delta\theta}) \approx \Delta\theta^T F_\theta \Delta\theta$
  - F_θ is called Fisher Information Matrix
  - $F_\theta = \mathbb{E}_{\rho(x|\theta)}[\nabla \log \rho(x|\theta) \nabla \log \rho(x|\theta)^T]$

Natural policy gradient

- We restrict the change of the policy as $D_{KL}(\rho_\theta, \rho_{\theta+\Delta\theta}) \approx \Delta\theta^T F_\theta \Delta\theta = \epsilon$
- If the change is very small, the update is most similar to the true gradient:
  - $\arg\max_{\Delta\theta} \Delta\theta^T \nabla_\theta J$ such that $\Delta\theta^T F_\theta \Delta\theta = \epsilon$
  - Thus, $\Delta\theta \propto F_\theta^{-1} \nabla_\theta J$
- It finds ascent direction that is closest to vanilla gradient, when changing policy by a small, fixed amount
  - $\nabla_\theta^{nat} \pi_\theta(s, a) = F_\theta^{-1} \nabla_\theta \pi_\theta(s, a)$

Most common PG methods

- REINFORCE:

$$\theta \leftarrow \theta + \alpha G_t \nabla \ln \pi(a|s,\theta)$$

- Actor-Critic:

$$\theta \leftarrow \theta + \alpha \mathrm{A}_{\pi_\theta}(\mathrm{s},\mathrm{a}) \nabla \ln \pi(a|s,\theta)$$

- Natural Actor-Critic:

$$F_\theta^{-1} \nabla_\theta J(\theta) = w$$

REINFORCE: MC policy gradient

- REward Increment = Nonnegative Factor x Offset Reinforcement x Characteristic Eligibility (Williams, 1992)
- By the Policy Gradient theorem, we could update the weight vector:
  - $\theta_{t+1} = \theta_t + \alpha \sum_a \hat{q}\,(S_t, a, \mathbf{w}) \nabla \pi(a|S_t, \theta)$
- REINFORCE algorithm uses G_t as an unbiased sample of $Q_{\pi_\theta}(s, a)$
  - $\theta_{t+1} = \theta_t + \alpha G_t \nabla \ln \pi(a|S_t, \theta)$

Sutton & Barto (2018, p. 328)

REINFORCE: Monte-Carlo Policy-Gradient Control (episodic) for π_*

Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$
Algorithm parameter: step size $\alpha > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
    Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot, \boldsymbol{\theta})$
    Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
        $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$ (G_t)
        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \gamma^t G \nabla \ln \pi(A_t|S_t, \boldsymbol{\theta})$

Performance of REINFORCE

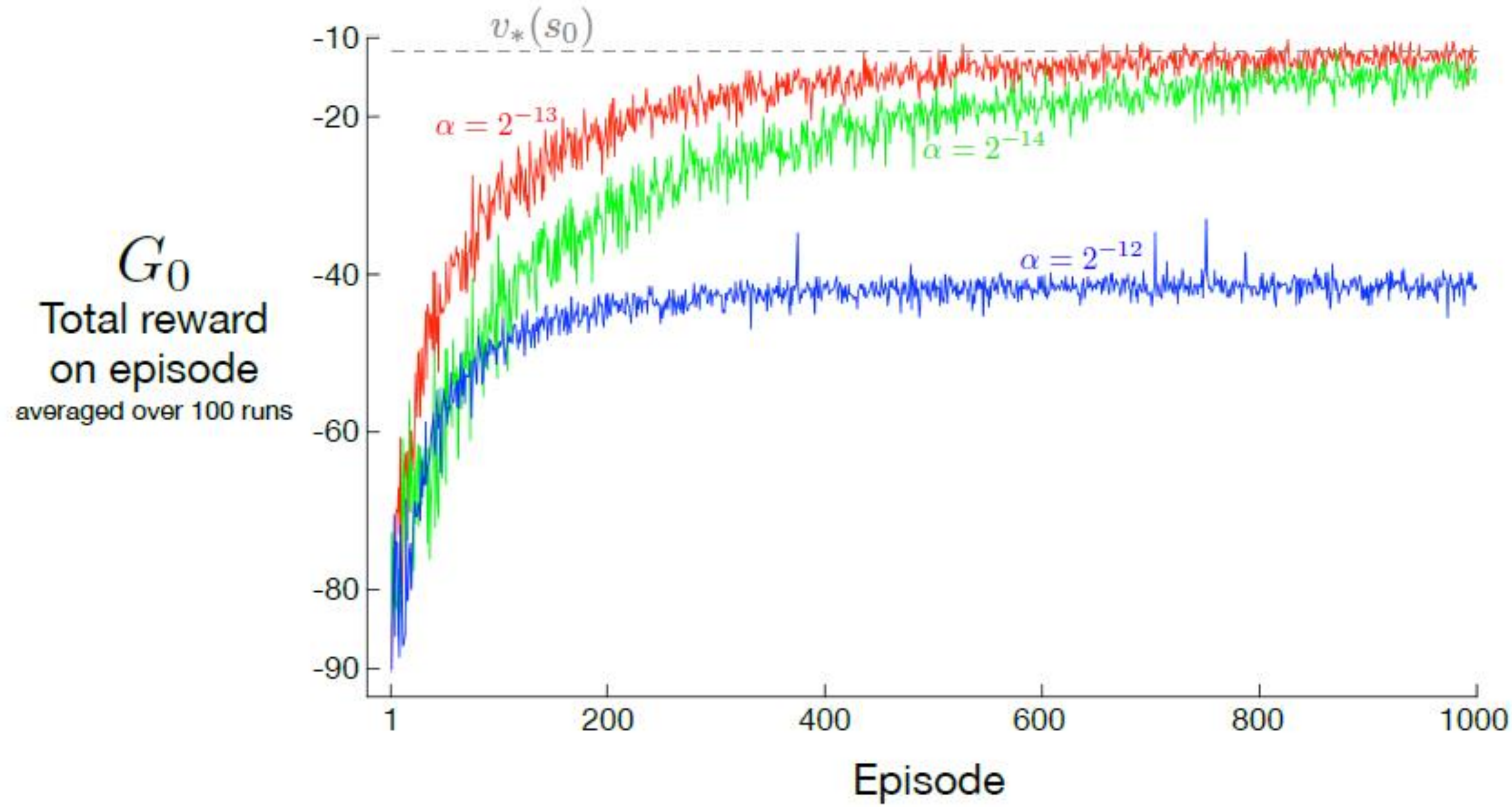


Figure 13.1: REINFORCE on the short-corridor gridworld example

Convergence of REINFORCE

- As a stochastic gradient method, REINFORCE has good theoretical convergence properties
- This assures an improvement in expected performance for sufficiently small and decreasing learning rate
- However, as a MC method, REINFORCE may be of high variance and thus produce slow learning

Three differences with Q-learning

- No epsilon-greedy strategy for exploration
- No replay buffer is used (PG methods are on-policy learning)
- No target network is needed

CODE: 02_cartpole_reinforce.py (Lapan, 2018: chapter 9)

```
GAMMA = 0.99
LEARNING_RATE = 0.01
EPISODES_TO_TRAIN = 4

class PGN(nn.Module):
  def __init__(self, input_size, n_actions):
    super(PGN, self).__init__()
    self.net = nn.Sequential(
    nn.Linear(input_size, 128),
    nn.ReLU(),
    nn.Linear(128, n_actions)
    )

  def forward(self, x):
    return self.net(x)
```

CODE: 02_cartpole_reinforce.py (Lapan, 2018: chapter 9)

```
if __name__ == "__main__":
  env = gym.make("CartPole-v0")
  writer = SummaryWriter(comment="-cartpole-reinforce")
  net = PGN(env.observation_space.shape[0],
        env.action_space.n)
  print(net)
  agent = ptan.agent.PolicyAgent(net,
          preprocessor=ptan.agent.float32_preprocessor,
          apply_softmax=True)
  exp_source =
          ptan.experience.ExperienceSourceFirstLast(env,
          agent, gamma=GAMMA)
  optimizer = optim.Adam(net.parameters(),
              lr=LEARNING_RATE)
```

CODE: 02_cartpole_reinforce.py (Lapan, 2018: chapter 9)

```
def calc_qvals(rewards):
    res = []
    sum_r = 0.0
    for r in reversed(rewards):
        sum_r *= GAMMA
        sum_r += r
        res.append(sum_r)
    return list(reversed(res))
                              ...
                              ...
    for step_idx, exp in enumerate(exp_source):
        batch_states.append(exp.state)
        batch_actions.append(int(exp.action))
        cur_rewards.append(exp.reward)

        if exp.last_state is None:
            batch_qvals.extend(calc_qvals(cur_rewards))
            cur_rewards.clear()
            batch_episodes += 1
```

CODE: 02_cartpole_reinforce.py (Lapan, 2018: chapter 9)

```
optimizer.zero_grad()
states_v = torch.FloatTensor(batch_states)
batch_actions_t = torch.LongTensor(batch_actions)
batch_qvals_v = torch.FloatTensor(batch_qvals)
logits_v = net(states_v)
log_prob_v = F.log_softmax(logits_v, dim=1)
log_prob_actions_v = batch_qvals_v *
    log_prob_v[range(len(batch_states)), batch_actions_t]
loss_v = -log_prob_actions_v.mean()
loss_v.backward()
optimizer.step()
```

$$\mathrm{L} = \mathbb{E}[\,-q(s,a)\ln\pi(a|s)]$$

Results of DQN and REINFORCE

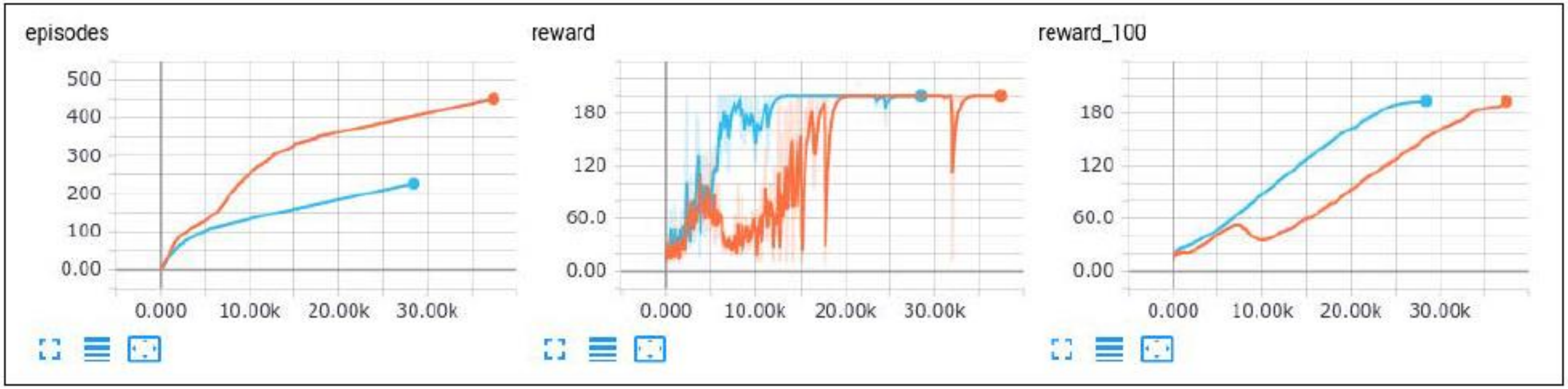


Figure 2: Convergence of DQN (orange) and REINFORCE (blue line)

REINFORCE with Baseline

- The Policy Gradient theorem can be generalized with an arbitrary baseline b(s):

$$\nabla J(\theta) \propto \sum_{s} d(s) \sum_{a} \big(q_\pi(s,a) - b(s)\big)\, \nabla \pi_\theta(s,a)$$

- Because the baseline could be uniformly zero, this update is a strict generalization of REINFORCE
- Natural choice for b(s) is an estimate of the state value, $\hat{v}(S_t, \mathbf{w})$, where $\mathbf{w} \in \mathbb{R}^d$

The possible choices of the baseline

1. Some constant value, which normally is the mean of the discounted rewards
2. The moving average of the discounted rewards
3. Value of the state *V(s)*

Advantage function

- Advantage function is defined by rewriting the policy gradient with the baseline function

$$A_{\pi_\theta}(s,a) = q_{\pi_\theta}(s,a) - v_{\pi_\theta}(s)$$

- Thus, the objective function is as below:

$$\nabla J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla \ln \pi_\theta(s,a) A_{\pi_\theta}(s,a)\right]$$

Sutton & Barto (2018, p. 330)

REINFORCE with Baseline (episodic), for estimating $\pi_{\boldsymbol{\theta}} \approx \pi_*$

Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$
Input: a differentiable state-value function parameterization $\hat{v}(s,\mathbf{w})$
Algorithm parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^{d}$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot, \boldsymbol{\theta})$
  Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$ (G_t)
    $\delta \leftarrow G - \hat{v}(S_t,\mathbf{w})$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \nabla \hat{v}(S_t,\mathbf{w})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} \gamma^t \delta \nabla \ln \pi(A_t|S_t, \boldsymbol{\theta})$

- Rules of thumb for $\alpha^{\mathbf{w}} = 0.1/\mathbb{E}[\|\nabla \hat{v}(S_t, \mathbf{w})\|_{\mu}^2]$

Performance of REINFORCE-baseline

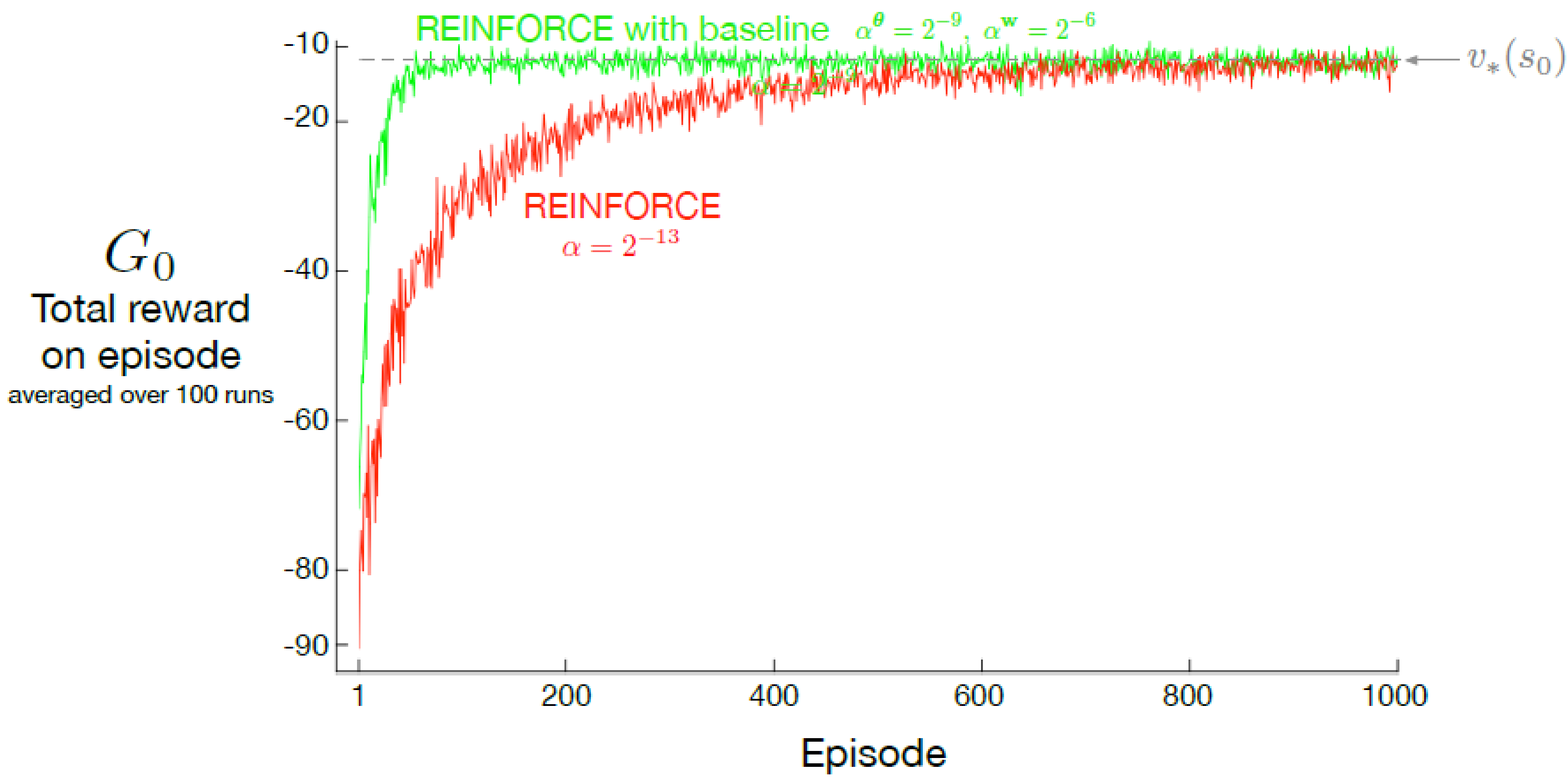


Figure 13.2: Adding REINFORCE-baseline on the short-corridor gridworld example

Convergence of REINFORCE-baseline

- Previous REINFORCE suffers high variance from the highly environment dependent rewards
- If the length of an episode is very long, the variance will be huge and learning will be slow
- Adding baseline to the learning target will reduce the variance, making the learning faster
- Note that this does not change its expectation

$$\sum_a b(s)\nabla\pi(a|s,\boldsymbol{\theta}) \;=\; b(s)\nabla\sum_a \pi(a|s,\boldsymbol{\theta}) \;=\; b(s)\nabla 1 \;=\; 0.$$

CODE: 03_cartpole_reinforce_baseline.py (Lapan, 2018: chapter 9)

```
def calc_qvals(rewards):
    res = []
    sum_r = 0.0
    for r in reversed(rewards):
        sum_r *= GAMMA
        sum_r += r
        res.append(sum_r)
    res = list(reversed(res))
    mean_q = np.mean(res)
    return [q - mean_q for q in res]
```

Actor-Critic methods

- REINFORCE-baseline still has high variance
- Also, like all MC methods, it is inconvenient for online problems
- Actor-Critic methods use a critic to estimate the action value function
  - Actor: updates policy parameters θ as suggested by critic
  - Critic: updates action value function parameters $\mathbf{w}$

Bias in Actor-Critic Algorithms

- Approximating the policy gradient introduces bias
- A biased policy gradient may not find the right solution
- Luckily, if we choose value function approximation carefully
- Then we can avoid introducing any bias
- i.e. We can still follow the exact policy gradient

From D. Silver’s Lecture 7 (p. 26)

Actor-Critic with advantage function

- Critic part can use the advantage function
- This will reduce variance of policy gradient
  - By estimating both $V_{\pi_\theta}(s)$ and $Q_{\pi_\theta}(s,a)$
- Using two function approximators and two parameter vectors

$$V_v(s) = V_{\pi_\theta}(s)$$

$$Q_w(s,a) = Q_{\pi_\theta}(s,a)$$

$$A(s,a) = Q_w(s,a) - V_v(s)$$

- In practice, we can use an approximate TD error

$$\delta_v = r + \gamma V_v(s') - V_v(s)$$

One-step Actor-Critic methods

- One-step methods avoid eligibility traces
- One-step AC methods replace the full return of REINFORCE with the one-step return

$$
\begin{aligned}
\theta_{t+1} &= \theta_t + \alpha\big(G_{t:t+1} - \hat{v}(S_t, \mathbf{w})\big)\frac{\nabla\pi(A_t|S_t,\theta_t)}{\pi(A_t|S_t,\theta_t)} \\
&= \theta_t + \alpha\underbrace{\boxed{\big(R_t + \gamma\hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w})\big)}}_{\text{TD-error, }\delta}\frac{\nabla\pi(A_t|S_t,\theta_t)}{\pi(A_t|S_t,\theta_t)} \\
&= \theta_t + \alpha\delta_t\frac{\nabla\pi(A_t|S_t,\theta_t)}{\pi(A_t|S_t,\theta_t)}
\end{aligned}
$$

Sutton & Barto (2018, p. 332)

One-step Actor–Critic (episodic), for estimating $\pi_{\boldsymbol{\theta}} \approx \pi_*$

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Input: a differentiable state-value function parameterization $\hat{v}(s,\mathbf{w})$
Parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^{d}$ (e.g., to $\mathbf{0}$)
Loop forever (for each episode):
  Initialize S (first state of episode)
  $I \leftarrow 1$
  Loop while S is not terminal (for each time step):
    $A \sim \pi(\cdot|S,\boldsymbol{\theta})$
    Take action A, observe S', R
    $\delta \leftarrow R + \gamma \hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})$ (if S' is terminal, then $\hat{v}(S',\mathbf{w}) \doteq 0$)
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \nabla \hat{v}(S,\mathbf{w})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} I \delta \nabla \ln \pi(A|S,\boldsymbol{\theta})$
    $I \leftarrow \gamma I$
    $S \leftarrow S'$

Policy Gradient with Eligibility Traces

- Like backward-view TD(λ), we can use eligibility traces
- By equivalence with TD(λ), substituting $\phi(s) = \nabla \ln \pi_\theta(s,a)$

$$
\begin{aligned}
\delta &= R_{t+1} + \gamma V_v(S_{t+1}) - V_v(S_t) \\
e_{t+1} &= \lambda e_t + \nabla \ln \pi_\theta(s,a) \\
\nabla\theta &= \alpha \delta e_t
\end{aligned}
$$

From D. Silver’s Lecture 7 (p. 34)

Sutton & Barto (2018, p. 332)

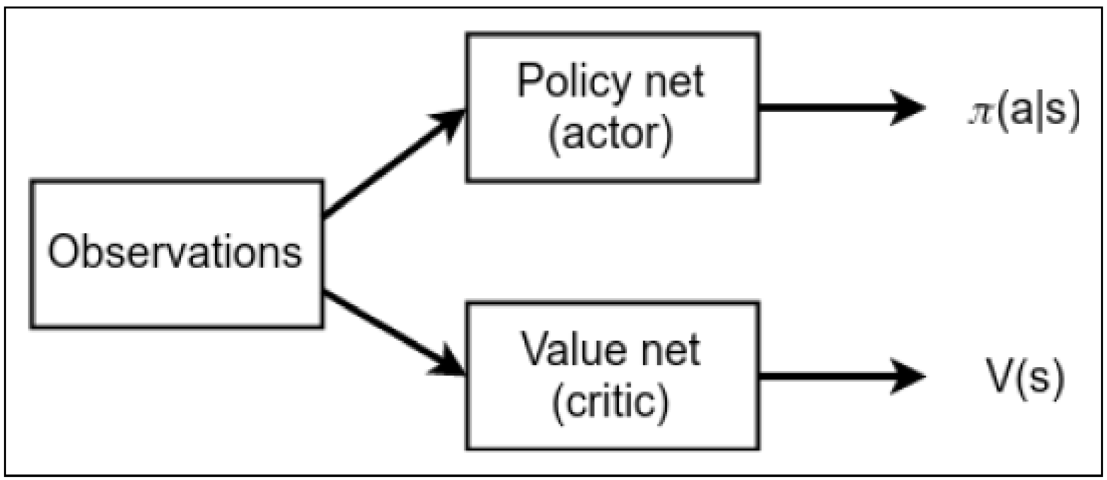
Actor–Critic with Eligibility Traces (episodic), for estimating $\pi_{\boldsymbol{\theta}} \approx \pi_*$

Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$
Input: a differentiable state-value function parameterization $\hat{v}(s,\mathbf{w})$
Parameters: trace-decay rates $\lambda^{\boldsymbol{\theta}} \in [0, 1]$, $\lambda^{\mathbf{w}} \in [0, 1]$; step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^{d}$ (e.g., to $\mathbf{0}$)
Loop forever (for each episode):
    Initialize S (first state of episode)
    $\mathbf{z}^{\boldsymbol{\theta}} \leftarrow \mathbf{0}$ (d'-component eligibility trace vector)
    $\mathbf{z}^{\mathbf{w}} \leftarrow \mathbf{0}$ (d-component eligibility trace vector)
    $I \leftarrow 1$
    Loop while S is not terminal (for each time step):
        $A \sim \pi(\cdot|S, \boldsymbol{\theta})$
        Take action A, observe S', R
        $\delta \leftarrow R + \gamma \hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})$ (if S' is terminal, then $\hat{v}(S',\mathbf{w}) \doteq 0$)
        $\mathbf{z}^{\mathbf{w}} \leftarrow \gamma \lambda^{\mathbf{w}} \mathbf{z}^{\mathbf{w}} + \nabla \hat{v}(S,\mathbf{w})$
        $\mathbf{z}^{\boldsymbol{\theta}} \leftarrow \gamma \lambda^{\boldsymbol{\theta}} \mathbf{z}^{\boldsymbol{\theta}} + I \nabla \ln \pi(A|S, \boldsymbol{\theta})$
        $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \mathbf{z}^{\mathbf{w}}$
        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} \delta \mathbf{z}^{\boldsymbol{\theta}}$
        $I \leftarrow \gamma I$
        $S \leftarrow S'$

A2C (Advantage Actor-Critic)

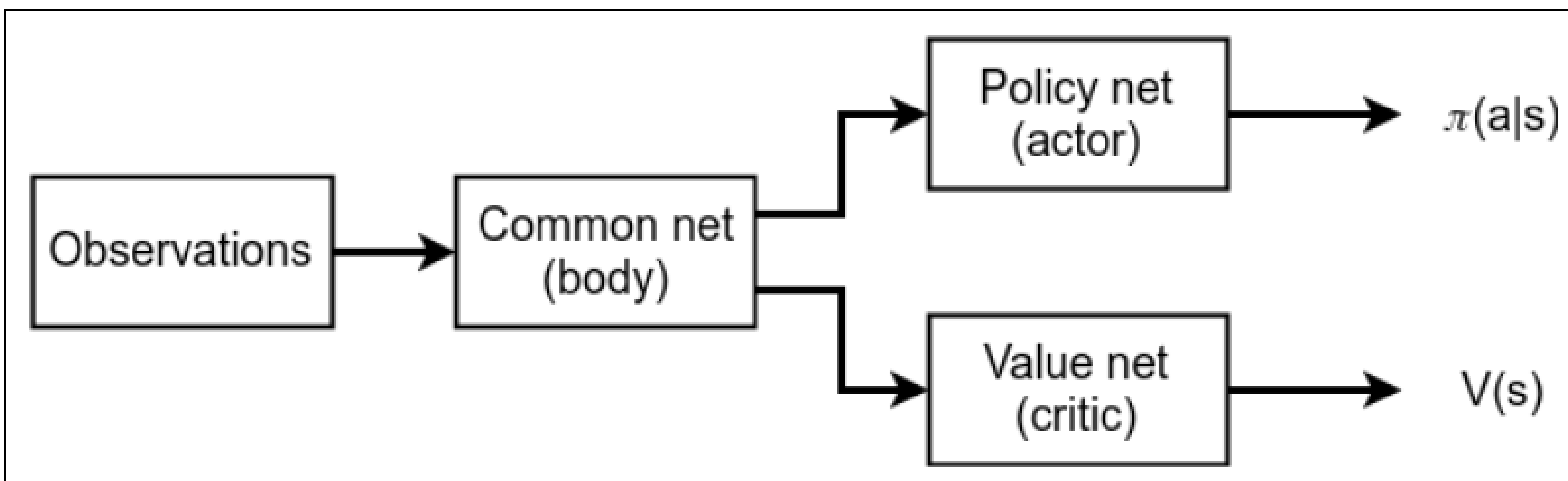
- In this method, advantage function is used as the baseline
- The problem is that we don’t know the value of the v(s), but instead we know q(s, a)
- To solve this, we use another neural network V(s) to approximate v(s) for every observation
- Now, we use policy network for the actor and another network for the critic

Basic architecture of A2C



Common architecture of A2C

- In practice, policy and value nets partially overlap
- Thus, policy and value nets are implemented as different heads of the same network
- Entropy bonus is added to loss function to improve exploration



Algorithm of A2C

1. Initialize network parameters θ with random values
2. Play N steps in the environment using the current policy π_θ, saving state st, action at, reward rt
3. R = 0 if the end of the episode is reached or $V_\theta(s_t)$
4. For $i = t-1 \dots t_{start}$ (note that steps are processed backwards):
   - $R \leftarrow r_i + \gamma R$
   - Accumulate the PG $\partial\theta_\pi \leftarrow \partial\theta_\pi + \nabla_\theta \log \pi_\theta(a_i|s_i)(R - V_\theta(s_i))$
   - Accumulate the value gradients $\partial\theta_v \leftarrow \partial\theta_v + \frac{\partial(R-V_\theta(s_i))^2}{\partial\theta_v}$
5. Update network parameters using the accumulated gradients, moving in the direction of PG $\partial\theta_\pi$ and in the opposite direction of the value gradients $\partial\theta_v$
6. Repeat from step 2 until convergence is reached

CODE: 02_pong_a2c.py (Lapan, 2018: chapter 10)

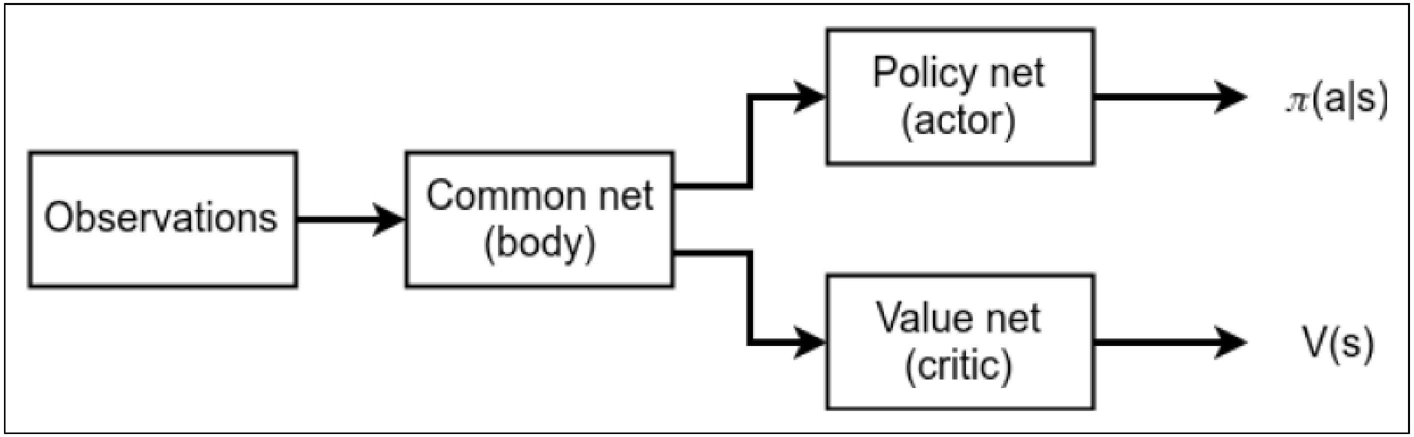
```
class AtariA2C(nn.Module):
    def __init__(self, input_shape, n_actions):
        super(AtariA2C, self).__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(input_shape[0], 32, kernel_size=8, stride=4),
            nn.ReLU(),
            nn.Conv2d(32, 64, kernel_size=4, stride=2),
            nn.ReLU(),
            nn.Conv2d(64, 64, kernel_size=3, stride=1),
            nn.ReLU()
        )
        conv_out_size = self._get_conv_out(input_shape)
        self.policy = nn.Sequential(
            nn.Linear(conv_out_size, 512),
            nn.ReLU(),
            nn.Linear(512, n_actions)
        )
        self.value = nn.Sequential(
            nn.Linear(conv_out_size, 512),
            nn.ReLU(),
            nn.Linear(512, 1)
        )
```

CODE: 02_pong_a2c.py (Lapan, 2018: chapter 10)

```
class AtariA2C(nn.Module):
    ...

    def _get_conv_out(self, shape):
        o = self.conv(torch.zeros(1, *shape))
        return int(np.prod(o.size()))

    def forward(self, x):
        fx = x.float() / 256
        conv_out = self.conv(fx).view(fx.size()[0], -1)
        return self.policy(conv_out), self.value(conv_out)
```

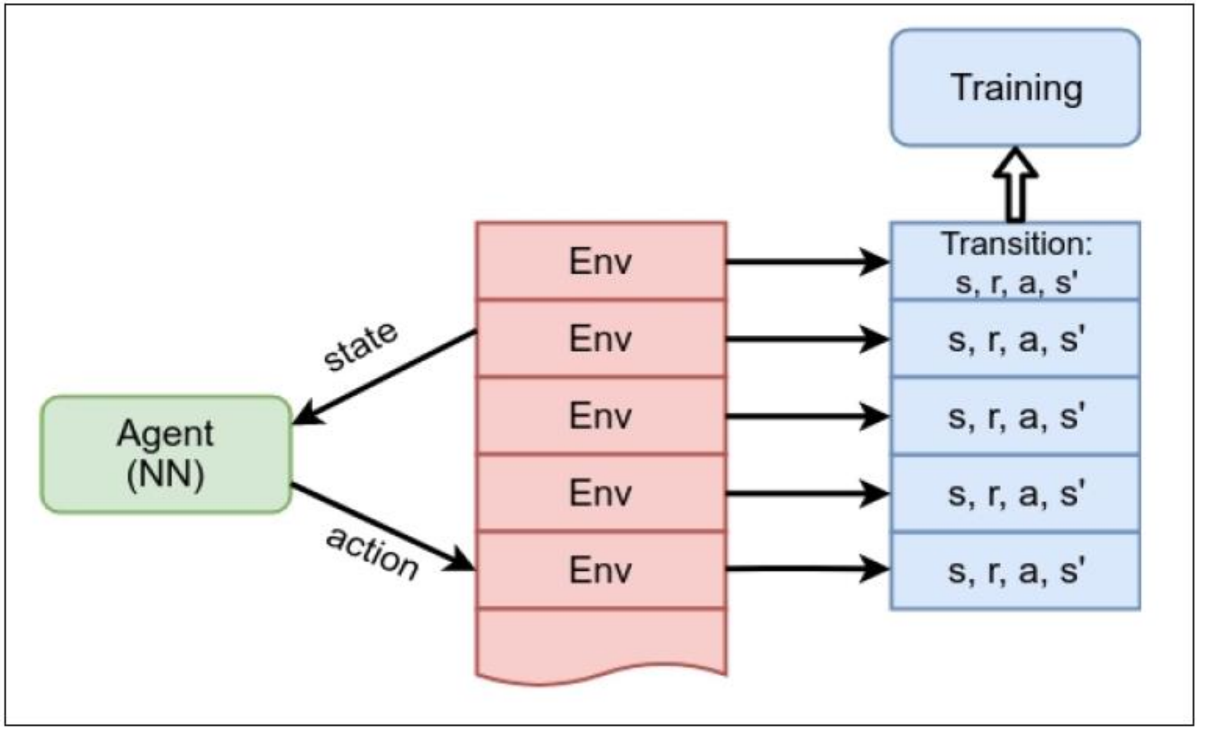


CODE: 02_pong_a2c.py (Lapan, 2018: chapter 10)

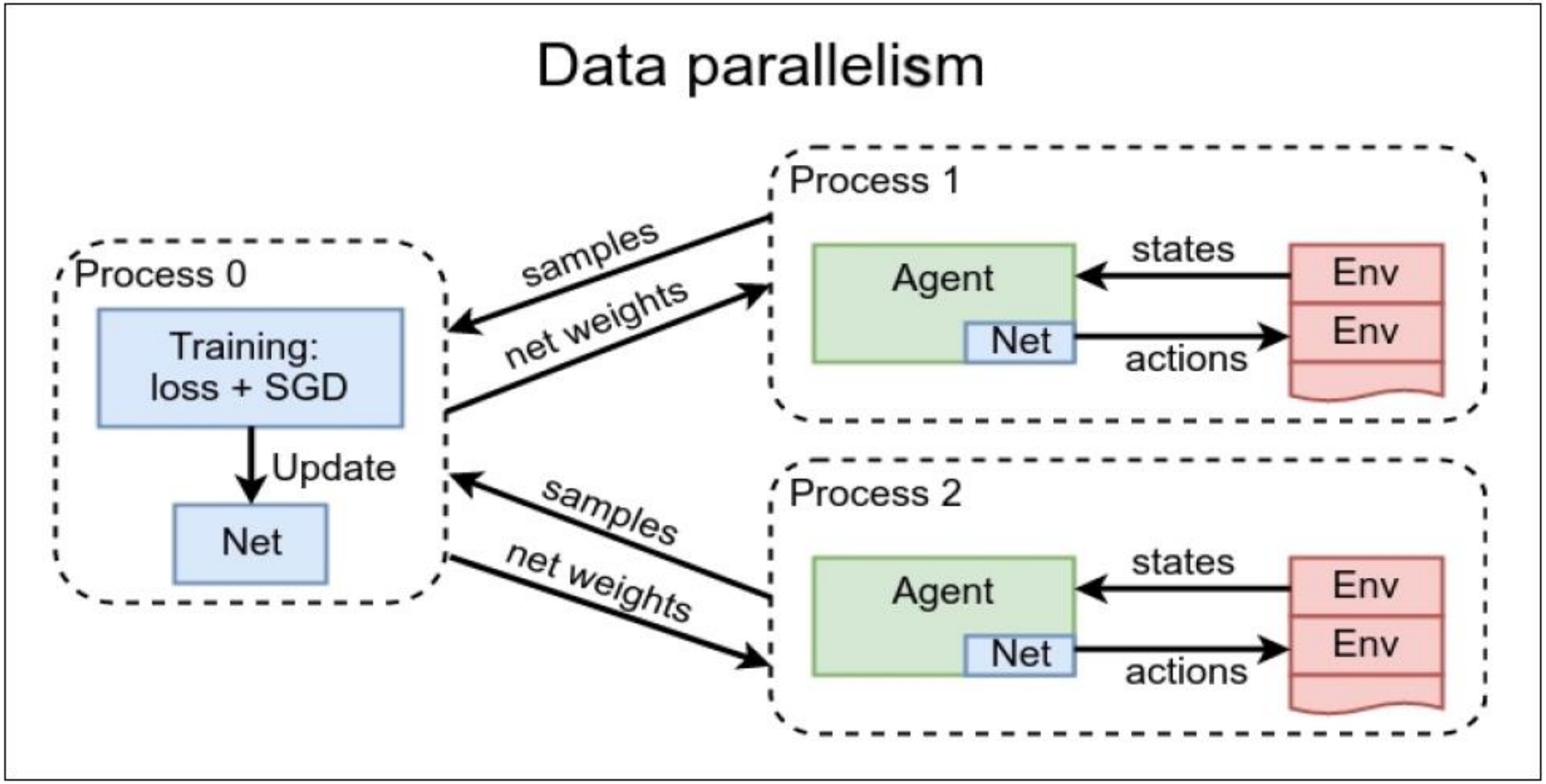
```
logits_v, value_v = net(states_v)
loss_value_v = F.mse_loss(value_v.squeeze(-1), vals_ref_v)
log_prob_v = F.log_softmax(logits_v, dim=1)
adv_v = vals_ref_v - value_v.detach()
log_prob_actions_v = adv_v * log_prob_v[range(BATCH_SIZE),
                      actions_t]
loss_policy_v = -log_prob_actions_v.mean()
prob_v = F.softmax(logits_v, dim=1)
entropy_loss_v = ENTROPY_BETA * (prob_v *
                 log_prob_v).sum(dim=1).mean()
# calculate policy gradients only
loss_policy_v.backward(retain_graph=True)
grads = np.concatenate([p.grad.data.cpu().numpy().flatten()
         for p in net.parameters()
         if p.grad is not None])
# apply entropy and value gradients
loss_v = entropy_loss_v + loss_value_v
loss_v.backward()
nn_utils.clip_grad_norm_(net.parameters(), CLIP_GRAD)
optimizer.step()
# get full loss
loss_v += loss_policy_v
```

Agent training using multiple envs

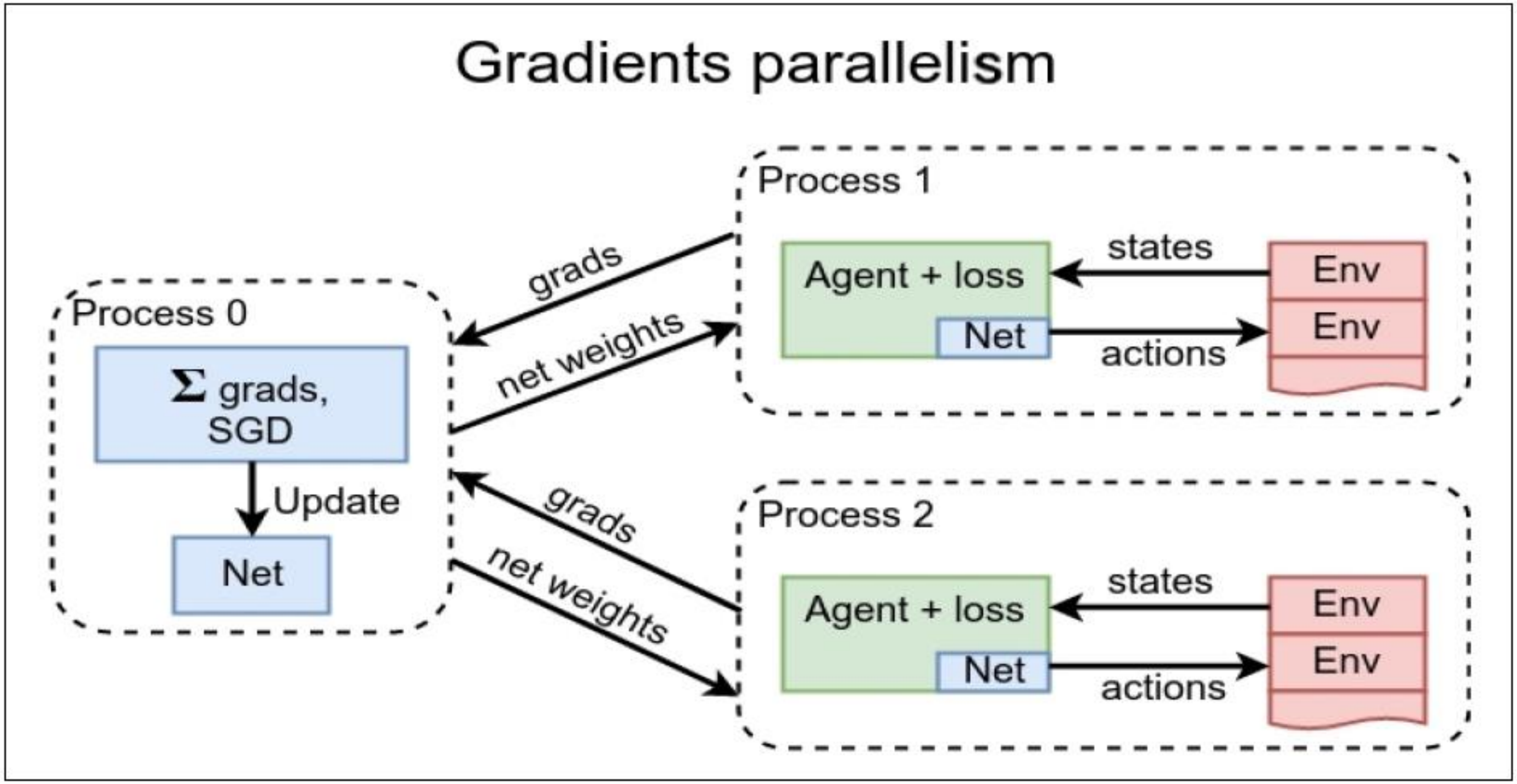
- Communicating with several parallel environments in A2C is done in synchronous way



Two approaches for AC parrelization



Two approaches for AC parrelization



A3C (Asychronous Advantage AC)

- A3C model (Mnih, 2016) suggests data paralleisim
- It has one main process and several children processes
- Each child process communicates with environment and gathers experience for training
- Instead of experience replay, A3C executes multiple agents in parallel
- This parallelism also decorrelates the agents’ data into a more stationary process

Recent works on parallelism of DRL

- A3C (Mnih et al, 2016)
- Impala (Espeholt, 2018)
- APE-X (Horgan, 2018)
- R2D2 (Kapturowski, 2019)

Model-based RL

Model-based Reinforcement Learning

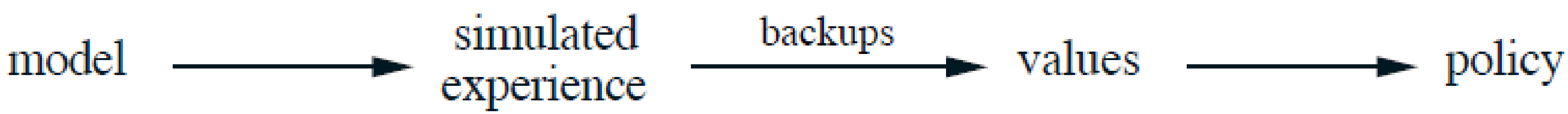
- Policy gradient methods learn policy directly from experience
- Value-based methods learn value function directly from experience
- Model-based methods learn model from experience → **model learning**
- Model-based methods use **planning** to learn value function or policy
- Thus, model-based methods combine learning and planning

Model Learning

- Model → anything that an agent can use to predict about environment
- Distribution models
  - A description of all possibilities and probabilities
  - e.g. probability mass function of a sum of 10 dice rolls
- Sample models
  - Just one of the possibilities and probabilities
  - e.g. an individual sum according to the probability distribution
- Model is used to produce **simulated experience**

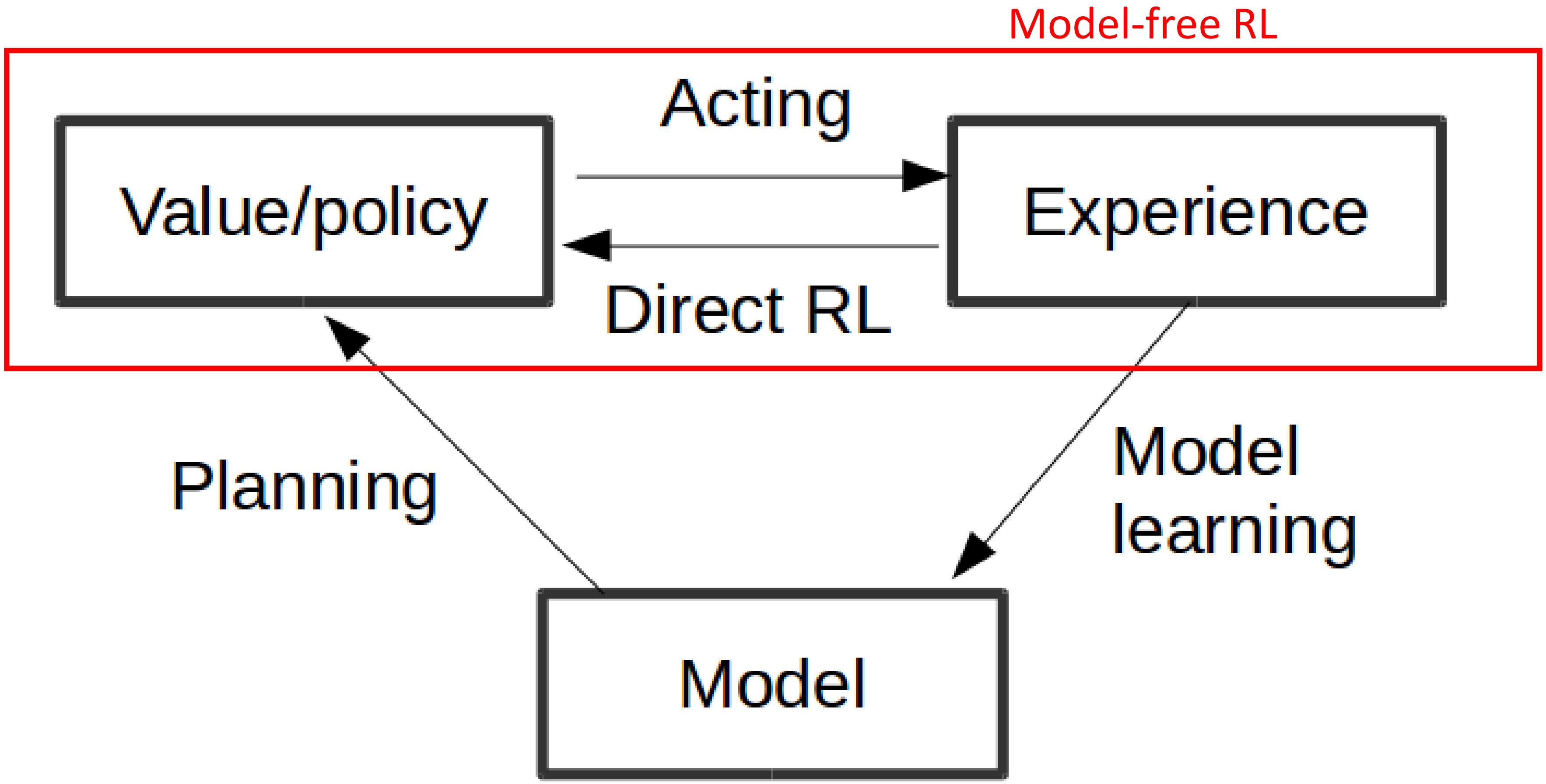
Planning

- Takes a model as input
- Returns or improves a policy for interacting with the modeled environment
- State-space planning
  - A search through the state space for an optimal policy

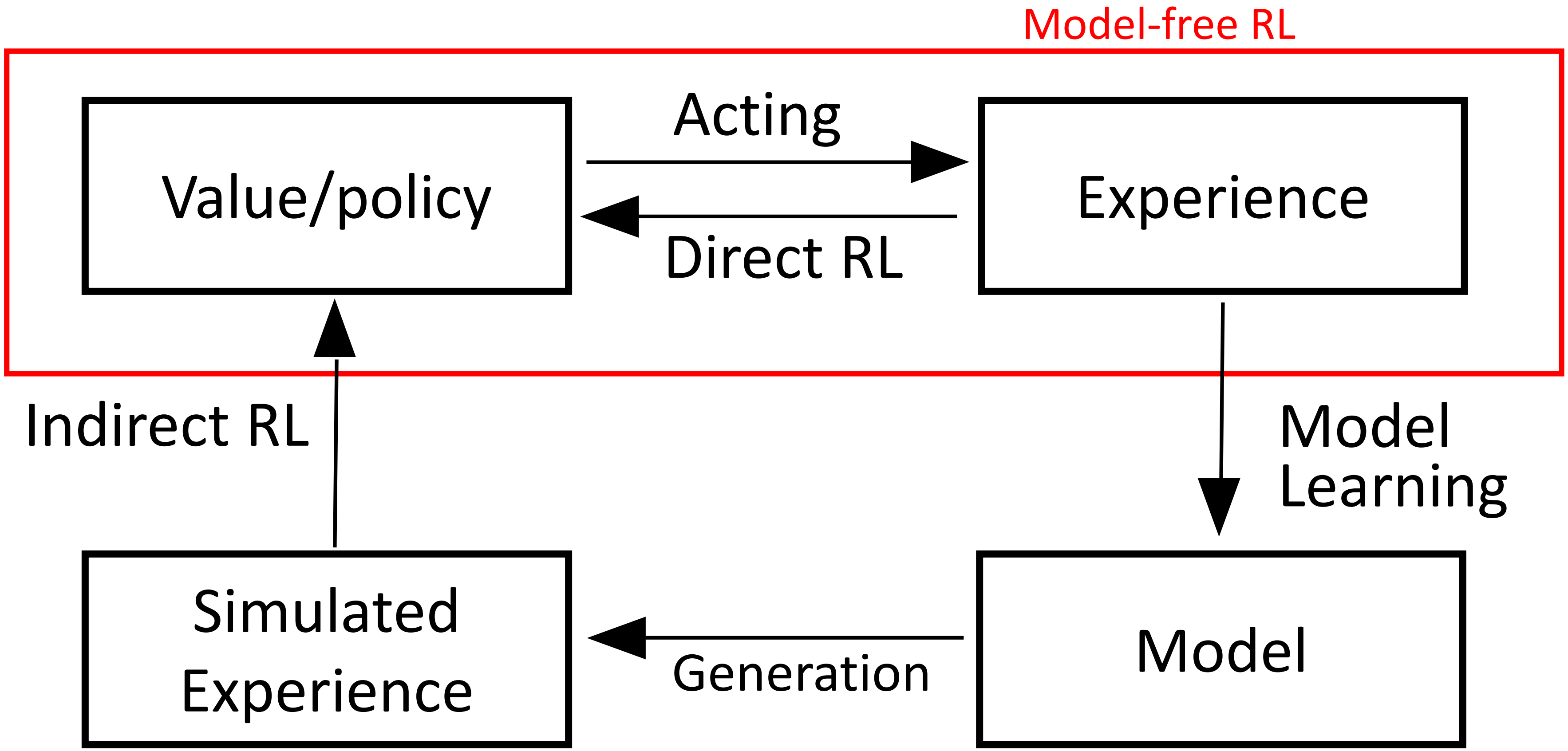


- Plan-space planning
  - A search through the space of plans

Abstract Schema of Model-based RL



Abstract Schema of Model-based RL



Properties of learning and planning

- Common properties
  - Both methods use value functions
  - Both methods learn value functions by backing-up updates
- Different properties
  - Planning uses simulated experience generated by a model
  - Learning methods use real experience generated by the environment

Definition of model

- A model is an MDP parameterized by η
- Thus, a model, represented by $M = (P_\eta, R_\eta)$ generates state transitions and rewards:

$$S_{t+1} \sim P_\eta(S_{t+1} | S_t, A_t)$$

$$R_{t+1} = R_\eta(R_{t+1} | S_t, A_t)$$

- Recall that complete knowledge of a model induces Dynamic Programming methods
- Typically, model learning is a supervised learning

Random-sample one-step tabular Q-planning

- Algorithm:

Loop forever:

1. Select a state, $s \in S$, and an action $a \in A(s)$, at random
2. Send s, a to a sample model, and obtain
   a sample next reward, r, and a sample next state, s'
3. Apply one-step tabular Q-learning to $s, \mathrm{a}, \mathrm{r}, \mathrm{s}'$:
   $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{\mathrm{a}} Q(s',a) - Q(s,a)]$

- A simple example of a planning method

Pros and Cons of Model-based RL

- Pros
  - Indirect RL methods make efficient use of a limited amount of experience
  - Thus, it achieves a better policy with fewer environmental interactions
- Cons
  - Inaccurate models will add another source of error
  - Learning can be complicated
  - Learning can be affected by biases in the design of the model

A view of model-based approaches

- An advice to model-free enthusiasts:

  "It is okay to tease out all the information that the data can provide, **but let's ask how far this will get us**"

  (Judea Pearl, 2018)

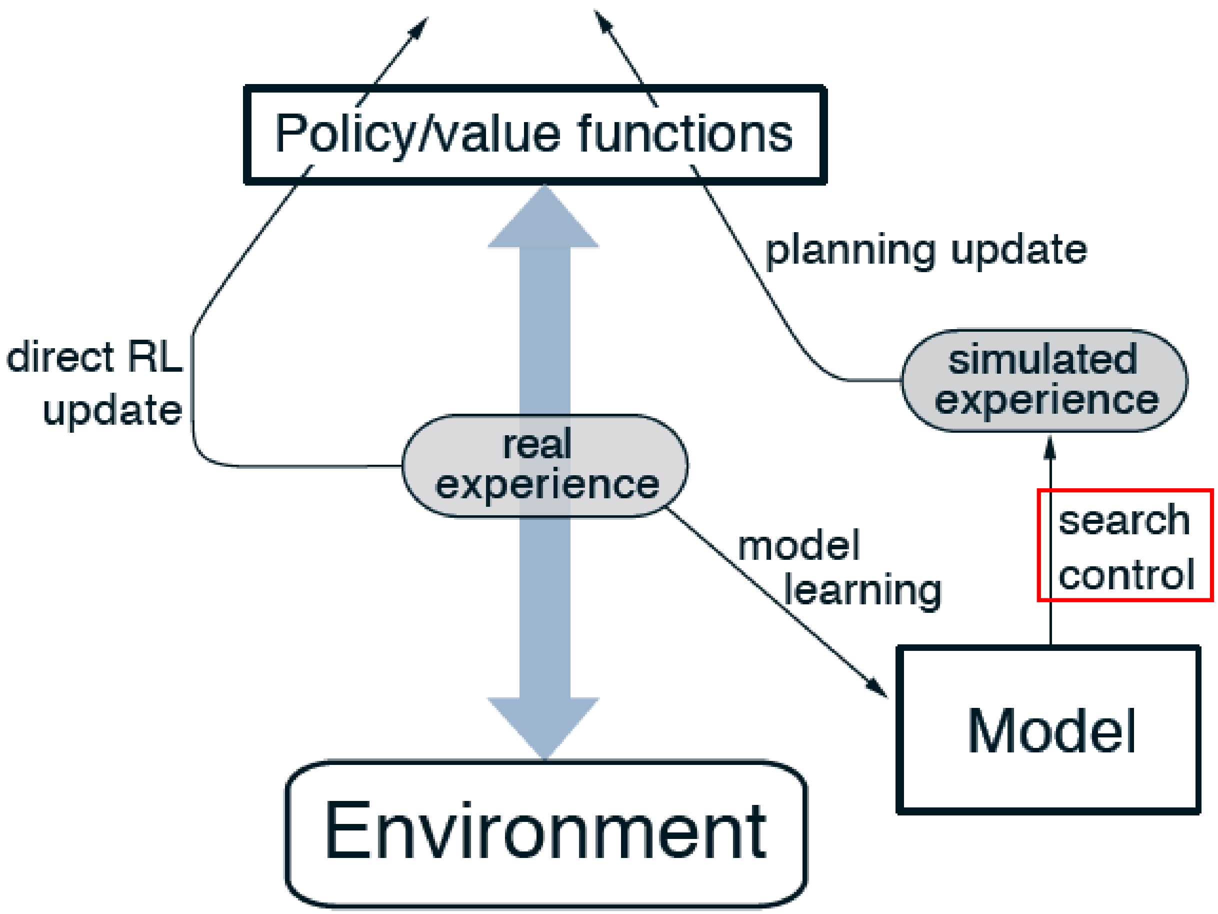
Dyna models

- The methods integrate planning, acting and learning
- They learn a model from real experience
- They learn and plan value function or policy from **real and simulated experience**
- Dyna-Q (Sutton, 1991)
  - A simple architecture, integrating planning and learning
  - An online planning agent
  - Use one-step tabular Q-learning for planning and learning

Search control

- Dyna-Q methods need to learn from real experience and gives rise to simulated experience
- Dyna-Q methods use **search control** for planning
- Definition of Search Control
  - A process that selects the starting states and actions for the simulated experiences

The general Dyna architecture



Algorithm

Tabular Dyna-Q

Initialize $Q(s, a)$ and $Model(s, a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$

Loop forever:

(a) $S \leftarrow$ current (nonterminal) state

(b) $A \leftarrow \varepsilon$-greedy(S, Q)

(c) Take action A; observe resultant reward, R, and state, S'

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

(e) $Model(S, A) \leftarrow R, S'$ (assuming deterministic environment)

(f) Loop repeat n times:

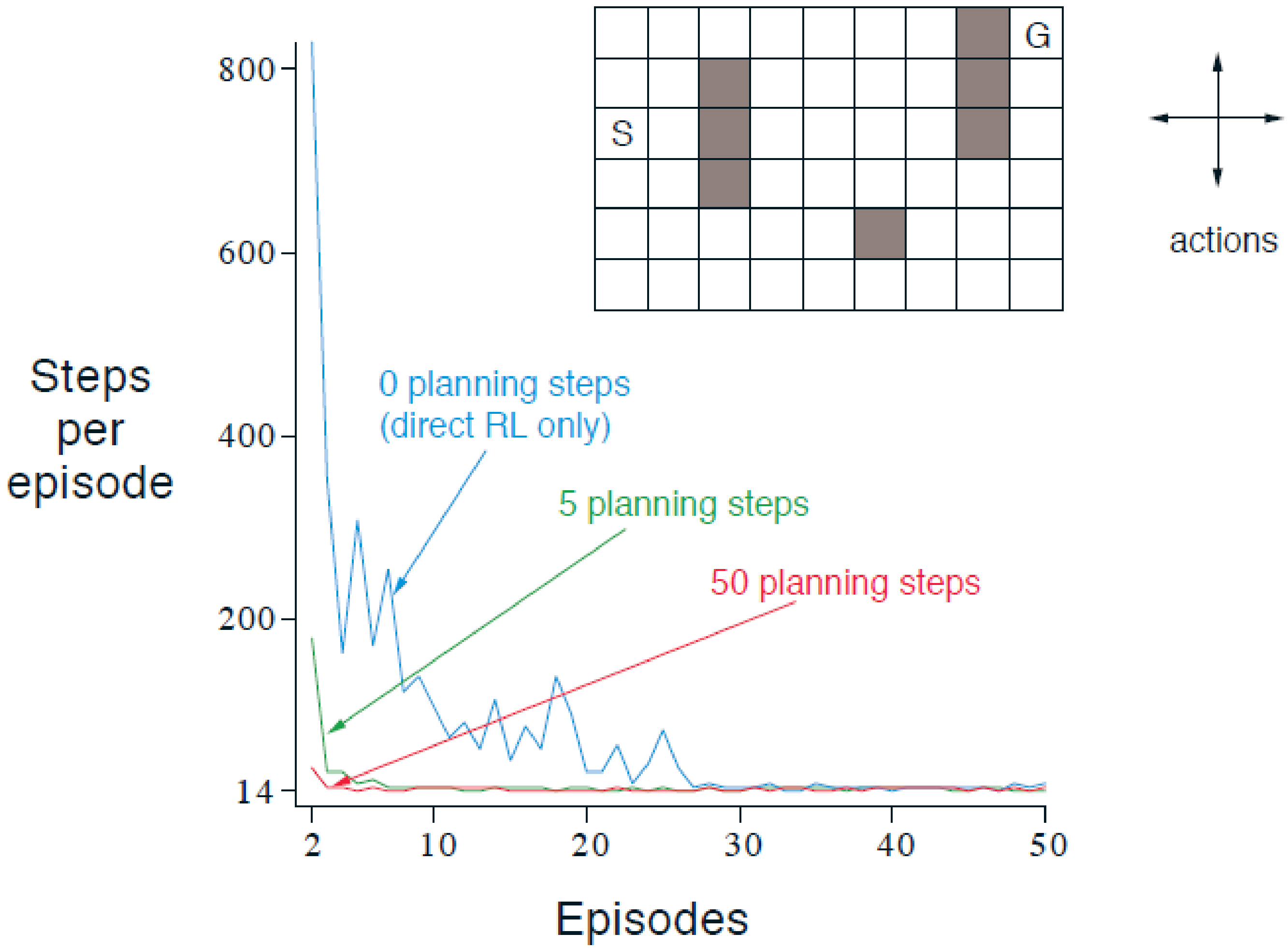
$S \leftarrow$ random previously observed state

$A \leftarrow$ random action previously taken in S

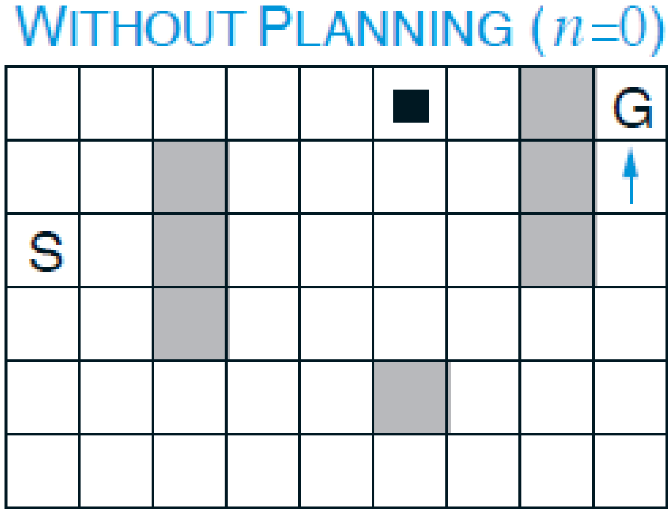
$R, S' \leftarrow Model(S, A)$

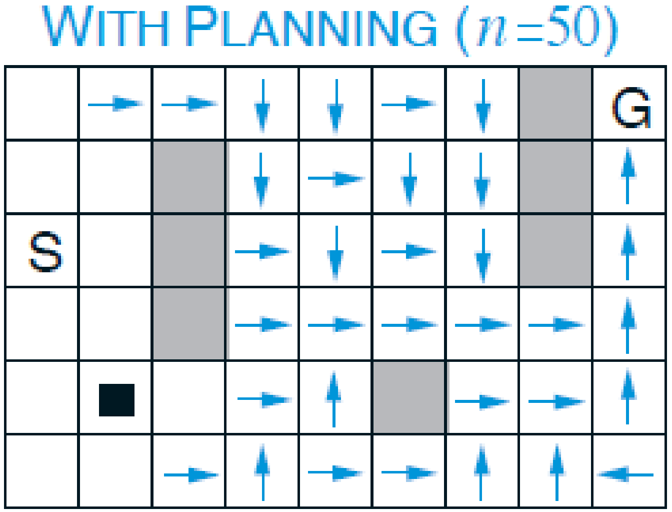
$Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$

Dyna Maze Problem

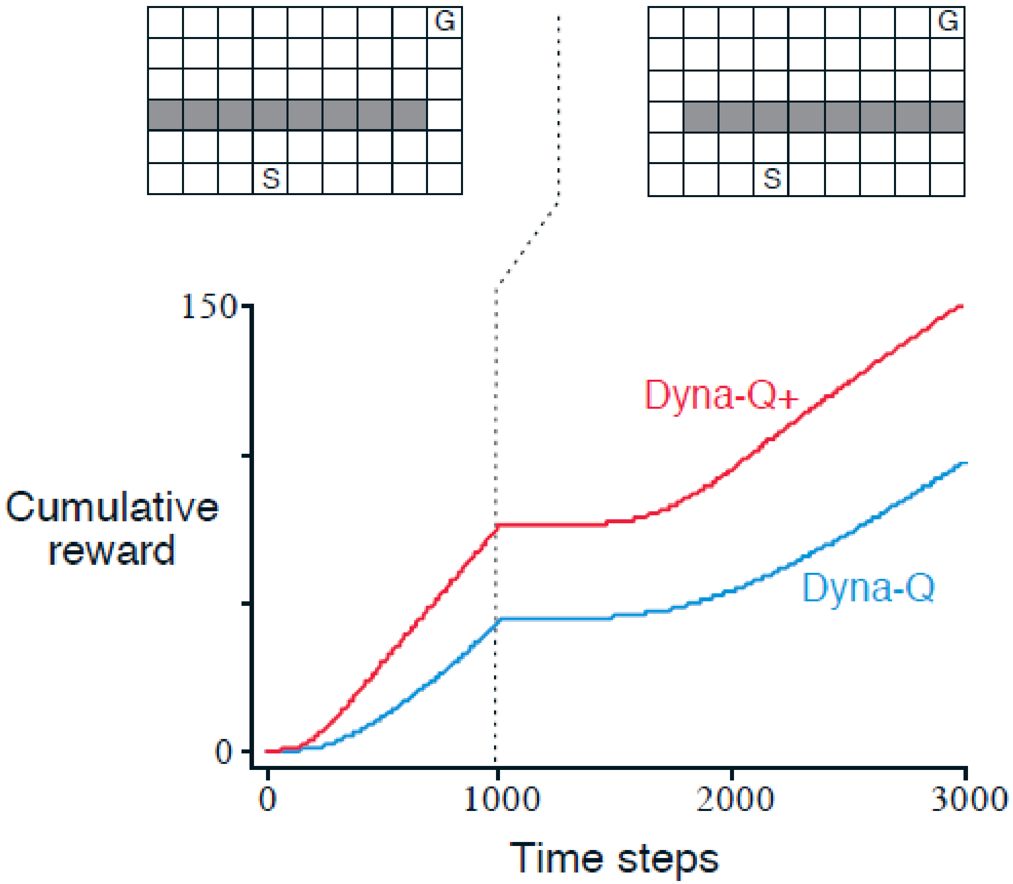


Effects of Planning at 2nd episode





Blocking Maze Problem

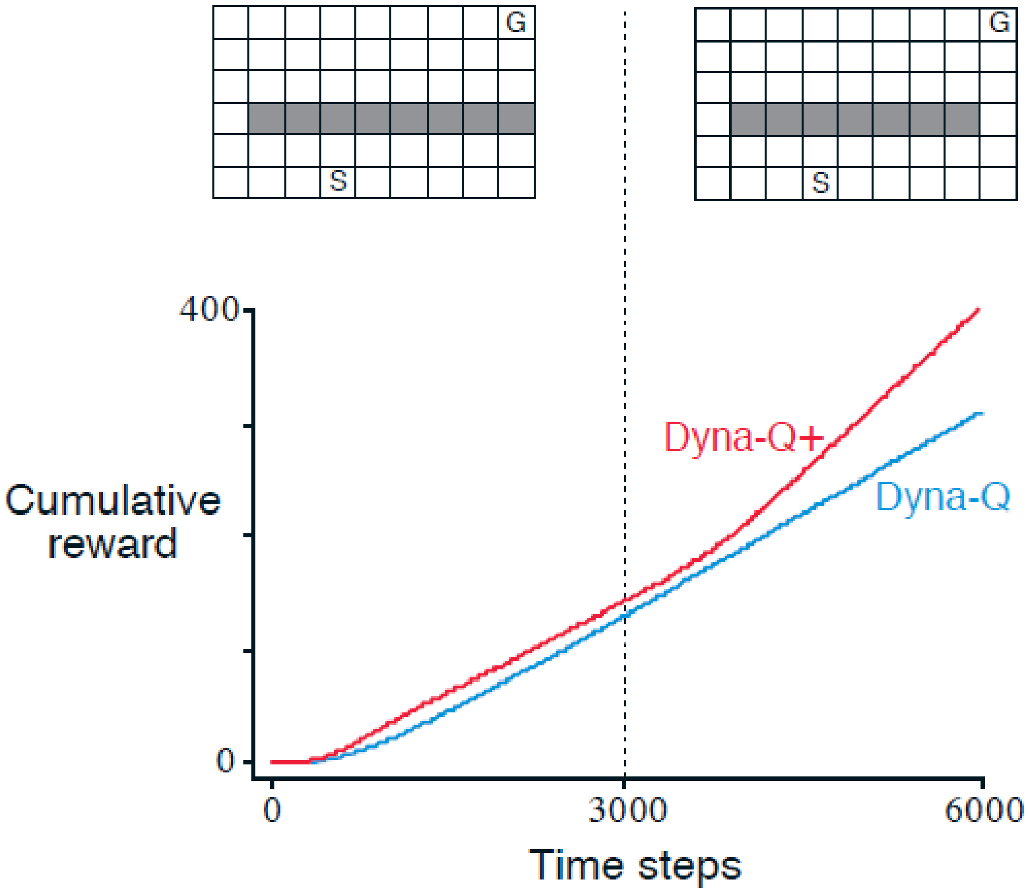


Dyna-Q+ agent uses a bonus reward system for long-untried actions:

$$r + k\sqrt{\tau}$$

where k is a some small value and τ is number of elapsed steps since the trail of the state transition

Shortcut Maze Problem



Prioritized Sweeping

- In Dyna agents, simulated transitions are selected uniformly at random from experiences
- But, a uniform selection is usually not the best
- Planning can be much more efficient if simulated transitions are prioritized

Backward focusing

- One can work backward from arbitrary states that have changed in value
- Values of some states may have changed a lot, whereas other may have changed little
- It is natural to prioritize the updates according to a measure of the urgency → prioritized sweeping
- A search control que is maintained for the purpose

Algorithm of Prioritized Sweeping

Prioritized sweeping for a deterministic environment

Initialize $Q(s,a)$, $Model(s,a)$, for all s,a, and $PQueue$ to empty

Loop forever:

(a) $S \leftarrow$ current (nonterminal) state

(b) $A \leftarrow policy(S,Q)$

(c) Take action A; observe resultant reward, R, and state, S'

(d) $Model(S,A) \leftarrow R, S'$

(e) $P \leftarrow |R + \gamma \max_a Q(S',a) - Q(S,A)|$.

(f) if $P > \theta$, then insert S,A into $PQueue$ with priority P

(g) Loop repeat n times, while $PQueue$ is not empty:

$S, A \leftarrow first(PQueue)$

$R, S' \leftarrow Model(S,A)$

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

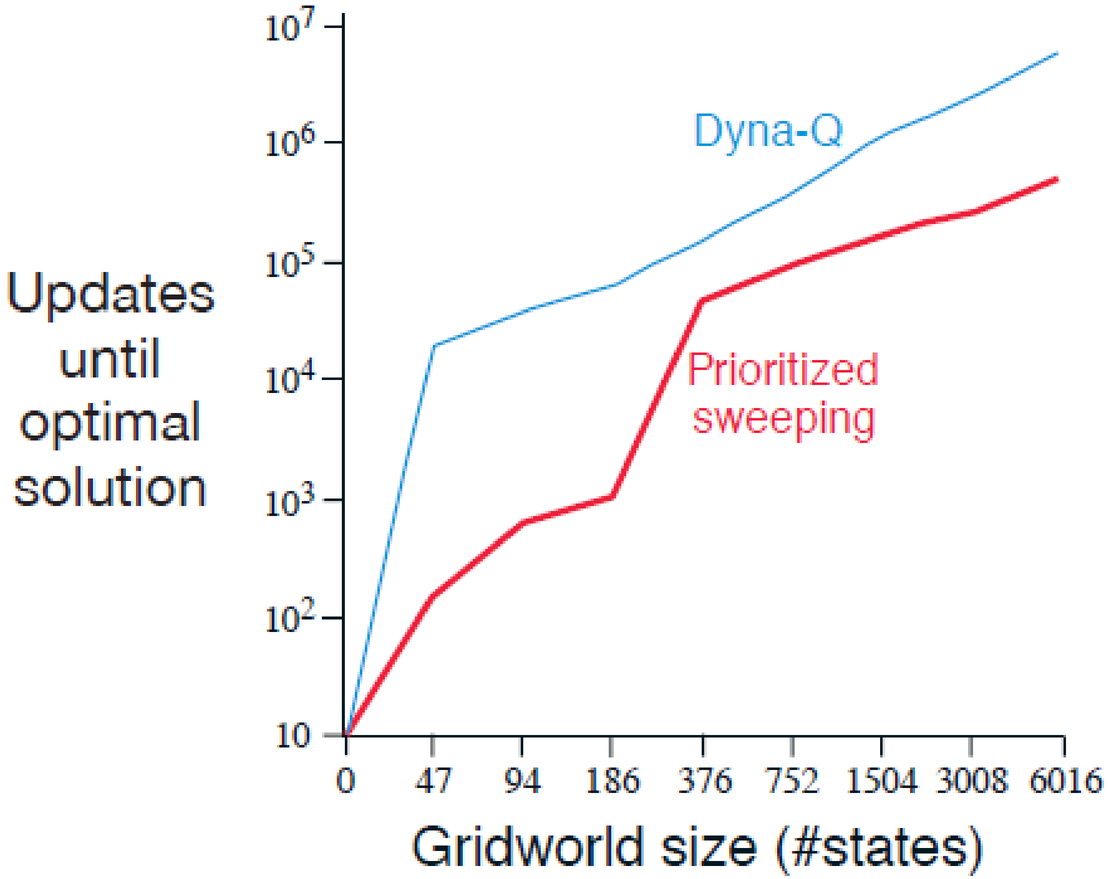
Loop for all $\bar{S}, \bar{A}$ predicted to lead to S:

$\bar{R} \leftarrow$ predicted reward for $\bar{S}, \bar{A}, S$

$P \leftarrow |\bar{R} + \gamma \max_a Q(S,a) - Q(\bar{S},\bar{A})|$.

if $P > \theta$ then insert $\bar{S}, \bar{A}$ into $PQueue$ with priority P

Prioritized Sweeping on Mazes



Prioritized sweeping increases the learning speed by a factor of 5 to 10

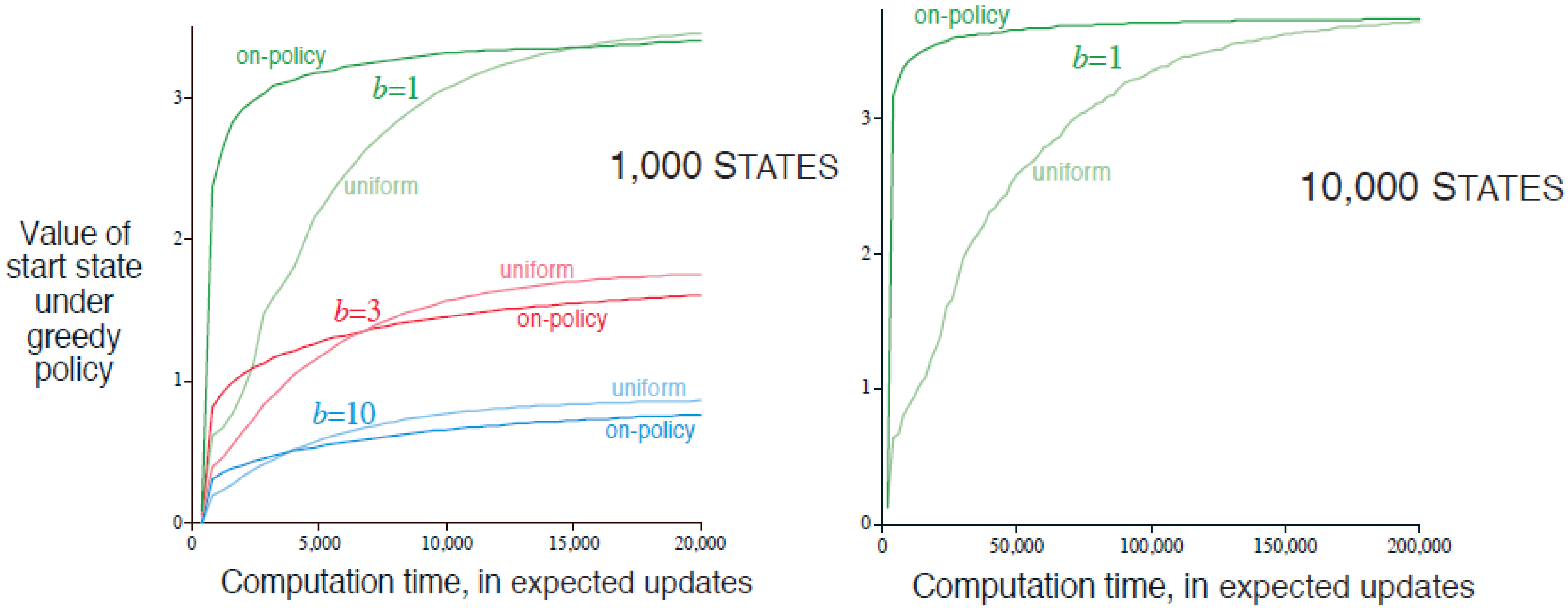
Trajectory Sampling

- Exhaustive sweeps are often used, but it is inefficient
- Basically, it is sample-based planning
- Its updates are drawn to the distribution observed when following the current policy
- Sample state transitions and rewards are given by the model
- Sample actions are given by the current policy

Pros and Cons of trajectory sampling

- Pros
  - At least, it is better than uniform distribution
  - It reduces huge search space into much smaller ones
- Cons
  - It causes the same old parts of the space to be repeatedly updated

Relative efficiency of updates

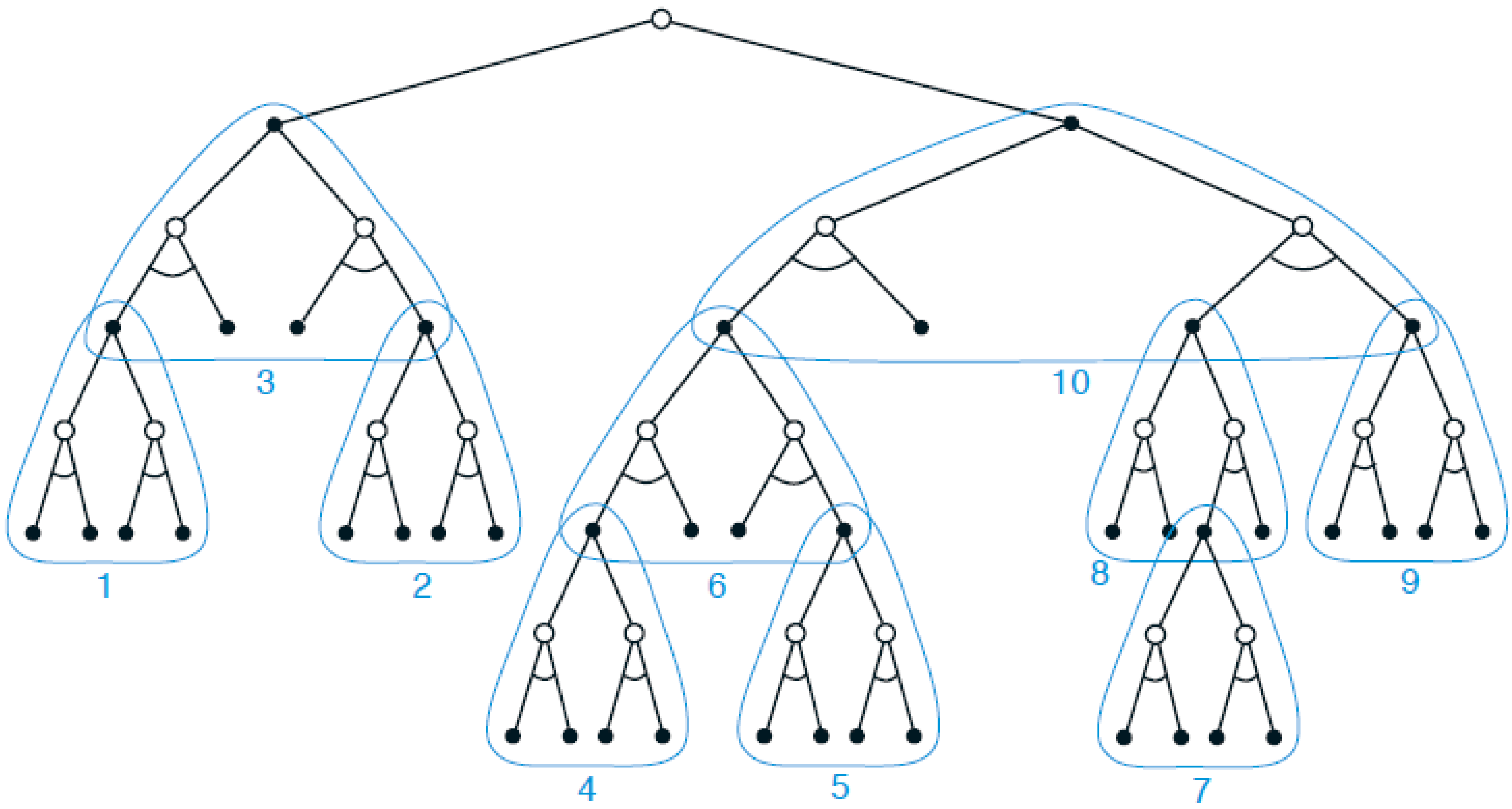


- These results are not conclusive, but show sampling by the on-policy distribution is very helpful for large problems

Decision time planning

- Planning can look much deeper than one-step-ahead
- Planning evaluates action choices leading to many different predicted state and reward trajectories
- Here, planning focuses on a particular state
- Heuristic search is an instance of decision-time planning

Heuristic search with one-step updates



Rollout Algorithm

- Rollout algorithms are decision-time planning algorithms based on Monte-Carlo control
- They estimate action values for a given policy by averaging the returns of many simulated trajectories

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

- And select an action with largest value

Advantages of Rollout Algorithm

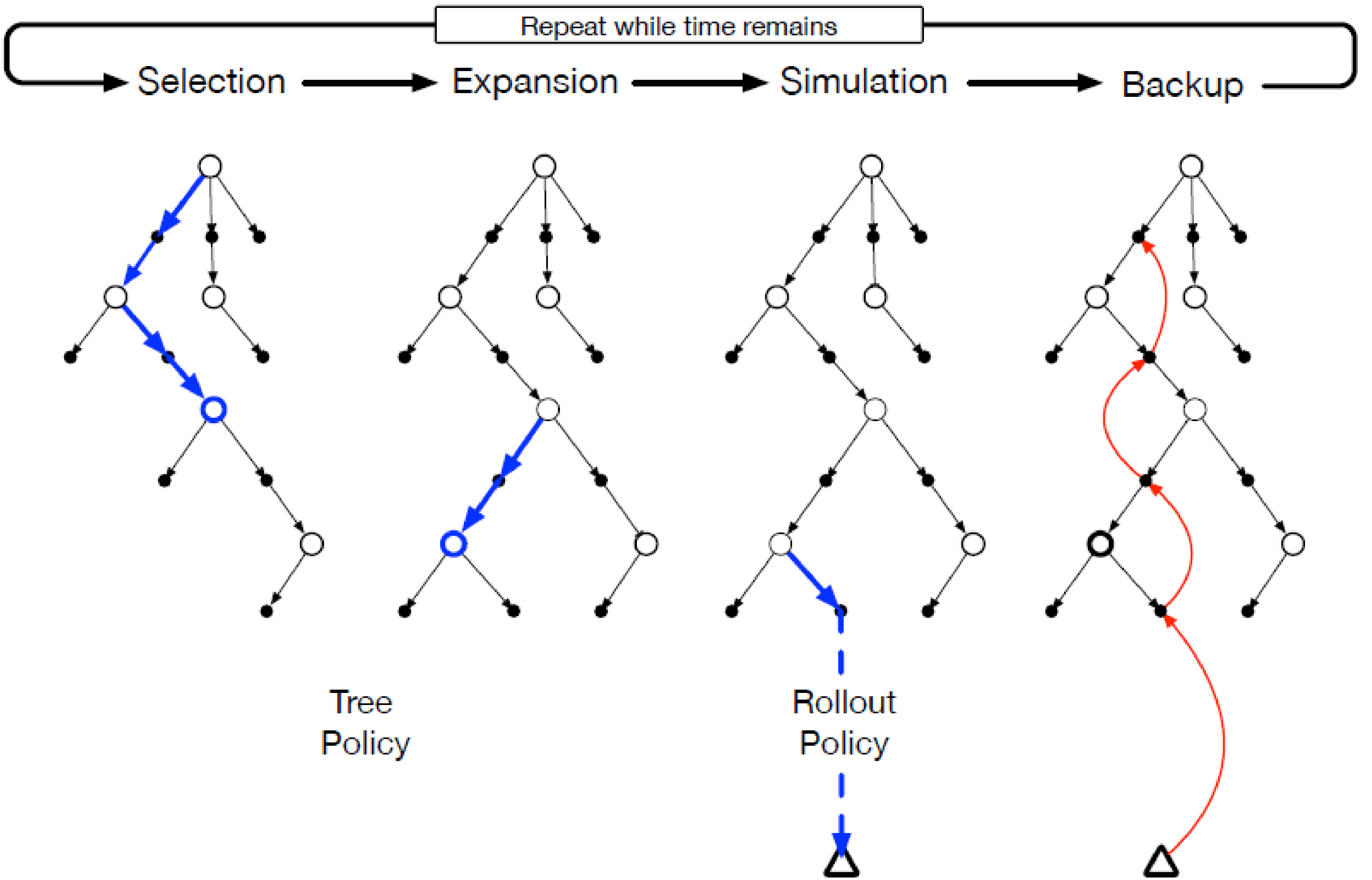
- They avoid the exhaustive sweeps
- They rely on sample models, avoiding distribution models
- Thus, they use sample updates
- Also, they take benefits from policy improvement property by acting greedily w.r.t the MC estimates of actions

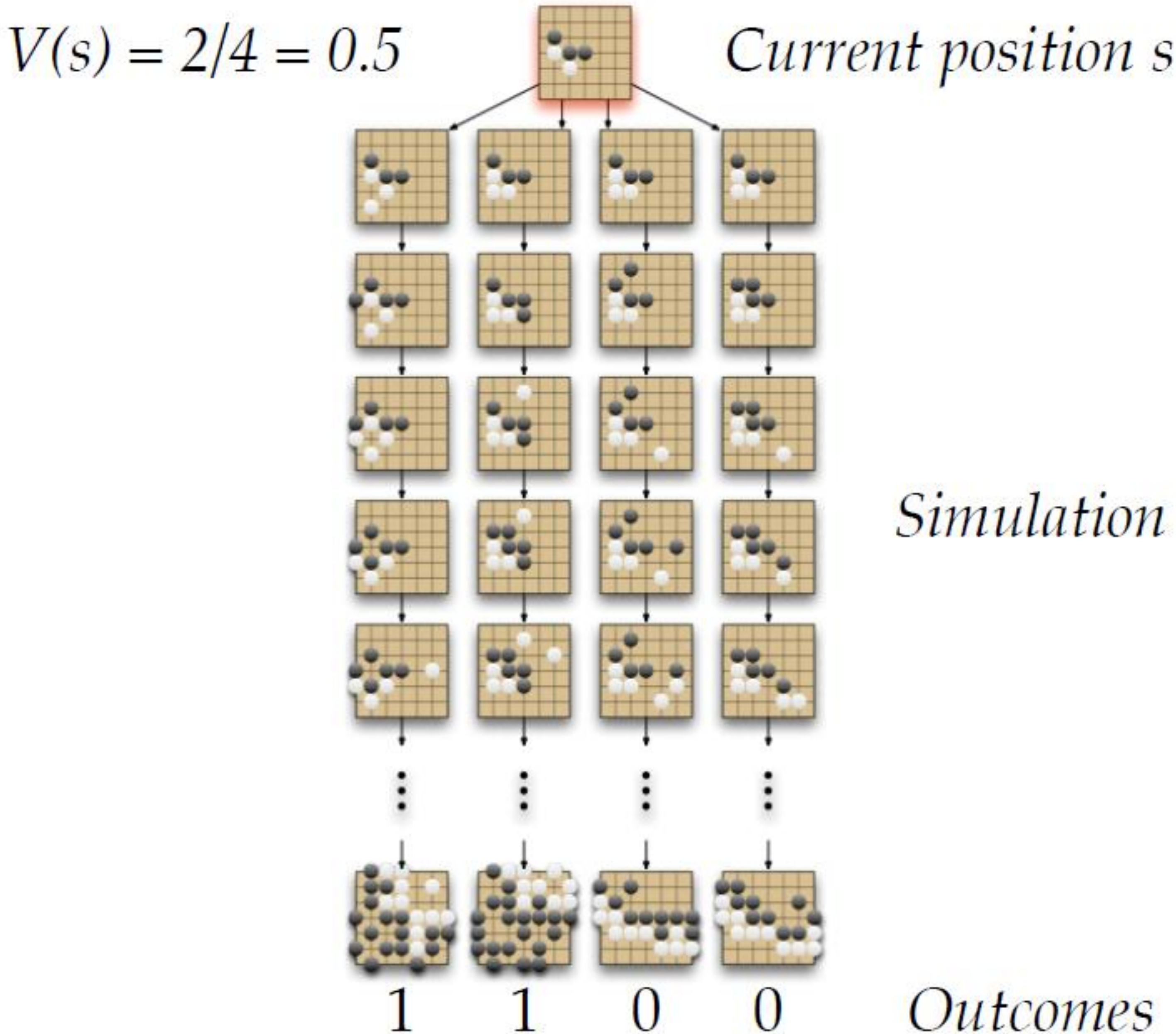
Monte-Carlo Tree Search

- MCTS is a successful example of decision-time planning
- MCTS is also a rollout algorithm
- MCTS is famous for the improvement in game Go (upto 6 dan in 2015)

Algorithm of MCTS

1. Selection
   - Starting at the root node
2. Expansion
   - The tree is expanded from the selected leaf node
3. Simulation
   - Simulation of a complete episode is run with actions by the rollout policy
4. Backup
   - The return by the simulation is backed up to update


Repeat while time remains
Selection
Expansion
Simulation
Backup
Tree
Policy
Rollout
Policy

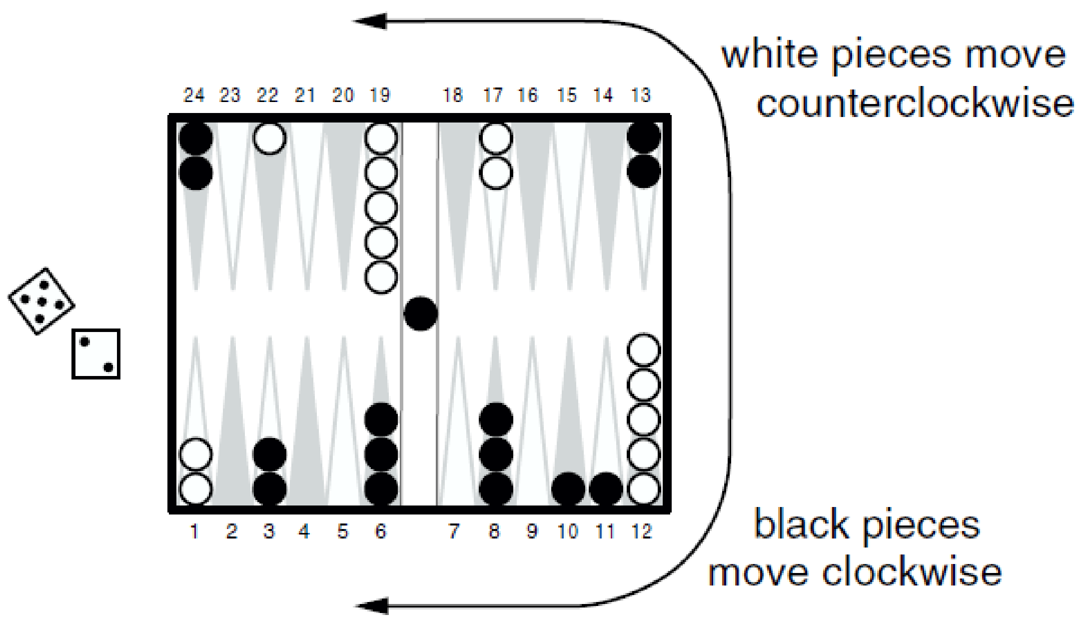


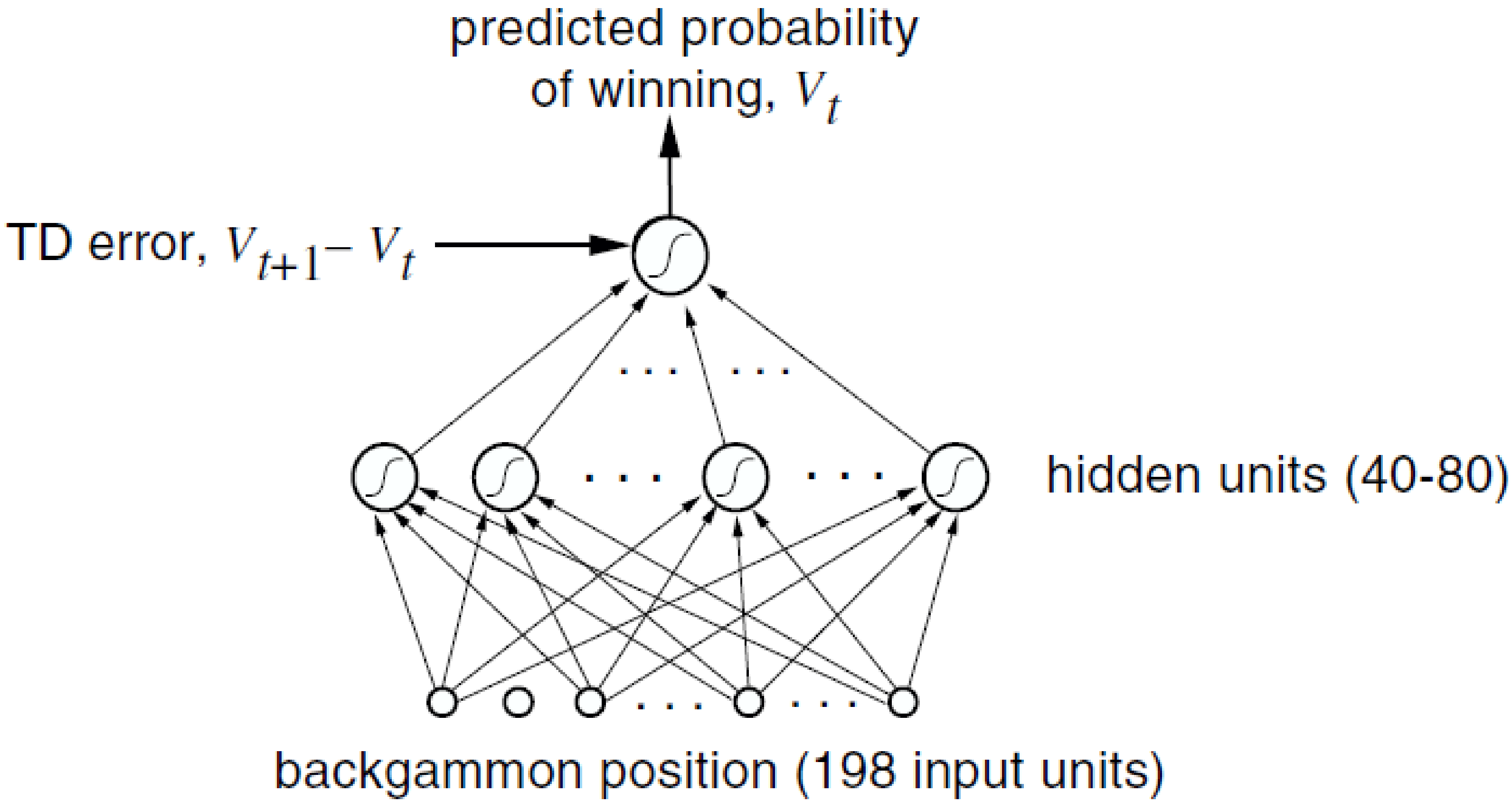
From D. Silver’s Lecture 8 (p. 41)

Case study: AlphaGo

TD-Gammon (Tesauro, 1992)

- RL learner of the game of backgammon
- It used TD(λ) with neural networks.
- Origin of DQN and self-play learning





- In TD-Gammon, $V_t(s)$ estimates the probability of winning starting from state s.

Results of TD-Gammon

Table 11.1 Summary of TD-Gammon results.

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

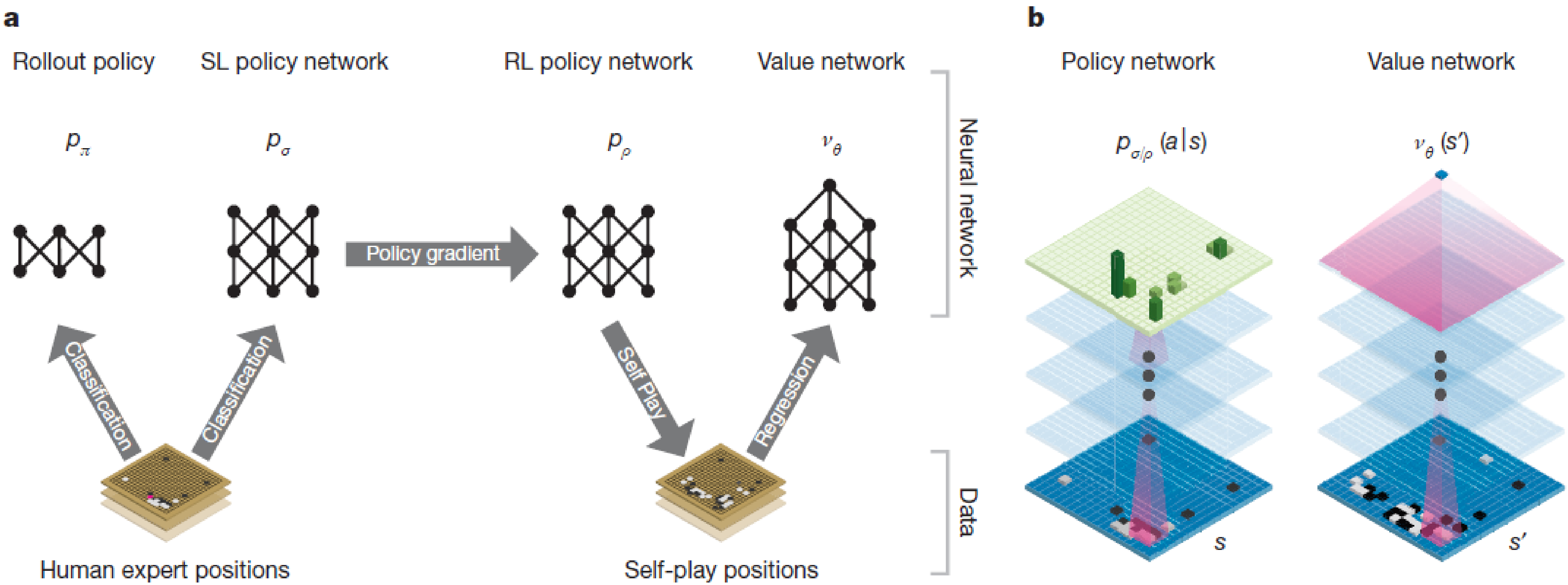
AlphaGo versions

- AlphaGo Fan (2015)
  - Beats European Go champion Fan Hui
- AlphaGo Lee (2016)
  - Beats World champion Sedol Lee for 4-1
- AlphaGo Master (2017)
  - Beats 60 pro Go players (including KeJie) in online battles
- AlphaGo Zero (2017.10)
  - Beats AlphaGo Master (89-11)
  - Only learn from self-play

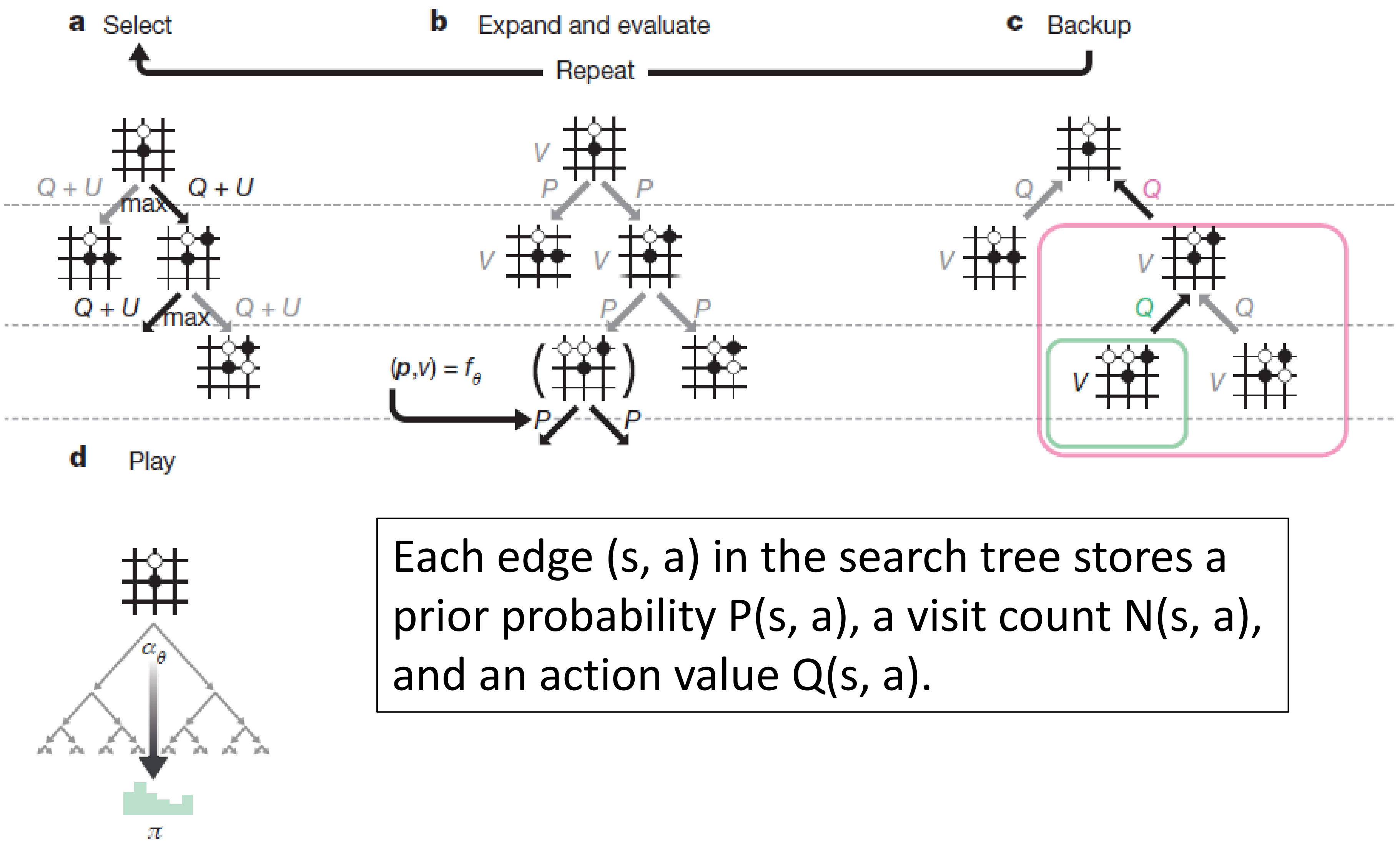
AlphaGo Fan: Basic Architecture

- Monte-Carlo Tree Search
- Value network
  - Used to evaluate current positions
- Policy network
  - A network generates a distribution for sampling
  - Three learning methods
    - Rollout policy learning (fast, but inaccurate local learner)
    - Supervised learning (from real human data)
    - Reinforcement learning (self-play)

Learning Pipeline



MCTS uses the network 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).

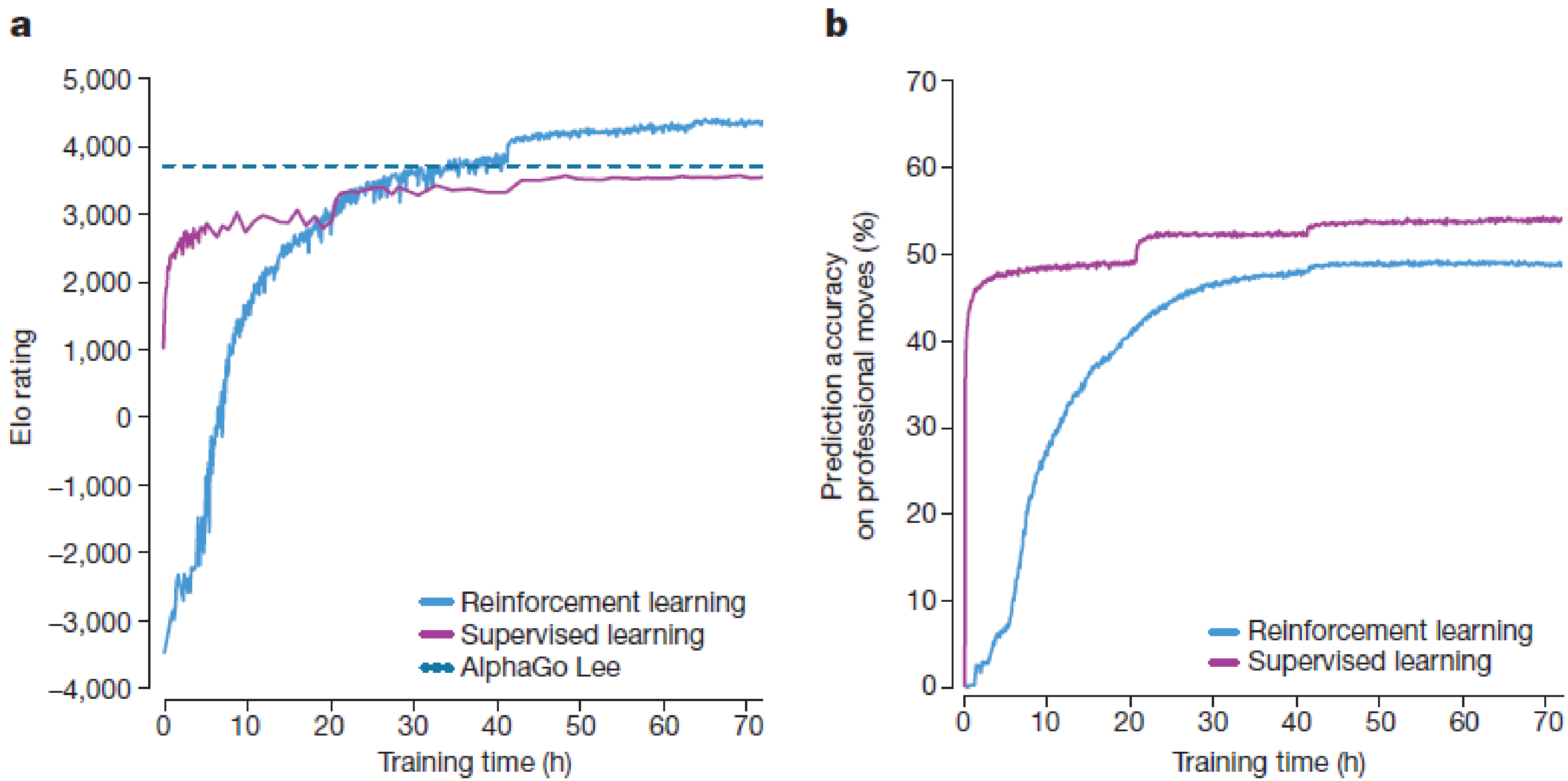
AlphaGo Zero

- Only self-play is used for learning
- Used pure image data for its input
- Only one network is used
  - Policy network and value network are integrated
- Rollout network is abandoned

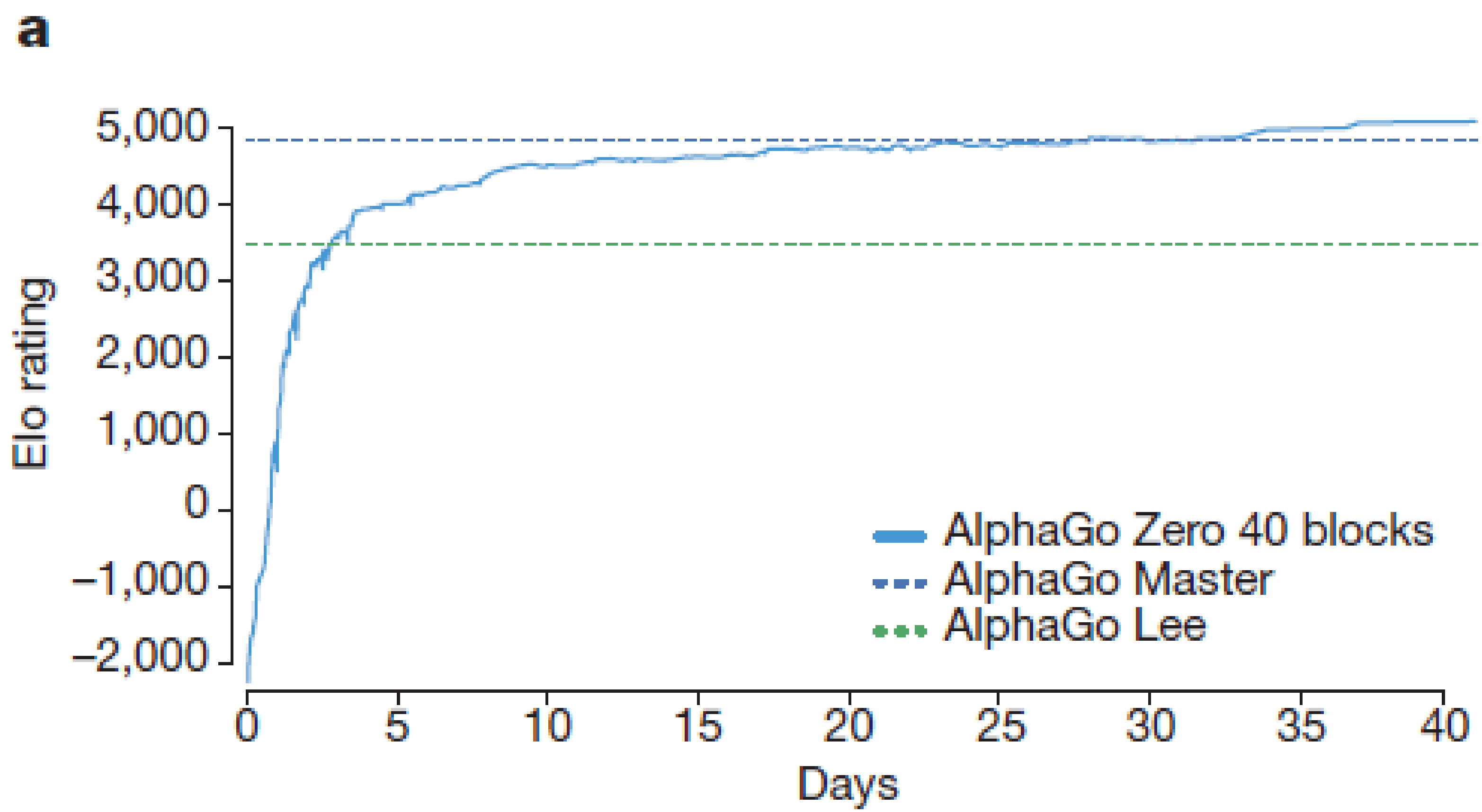
AlphaGo Zero training

- 4.9 million games of self-play are generated.
- 1,600 simulations for each MCTS (∼0.4s per move)
- Parameters are updated from 700,000 mini-batches of 2,048 positions.
- The neural network contained 20 residual blocks.
- AlphaGo Zero used a single machine with 4 TPUs (compared with AlphaGo Lee with 48 TPUs).
- Supervised learning with KGS data is constructed to be compared.

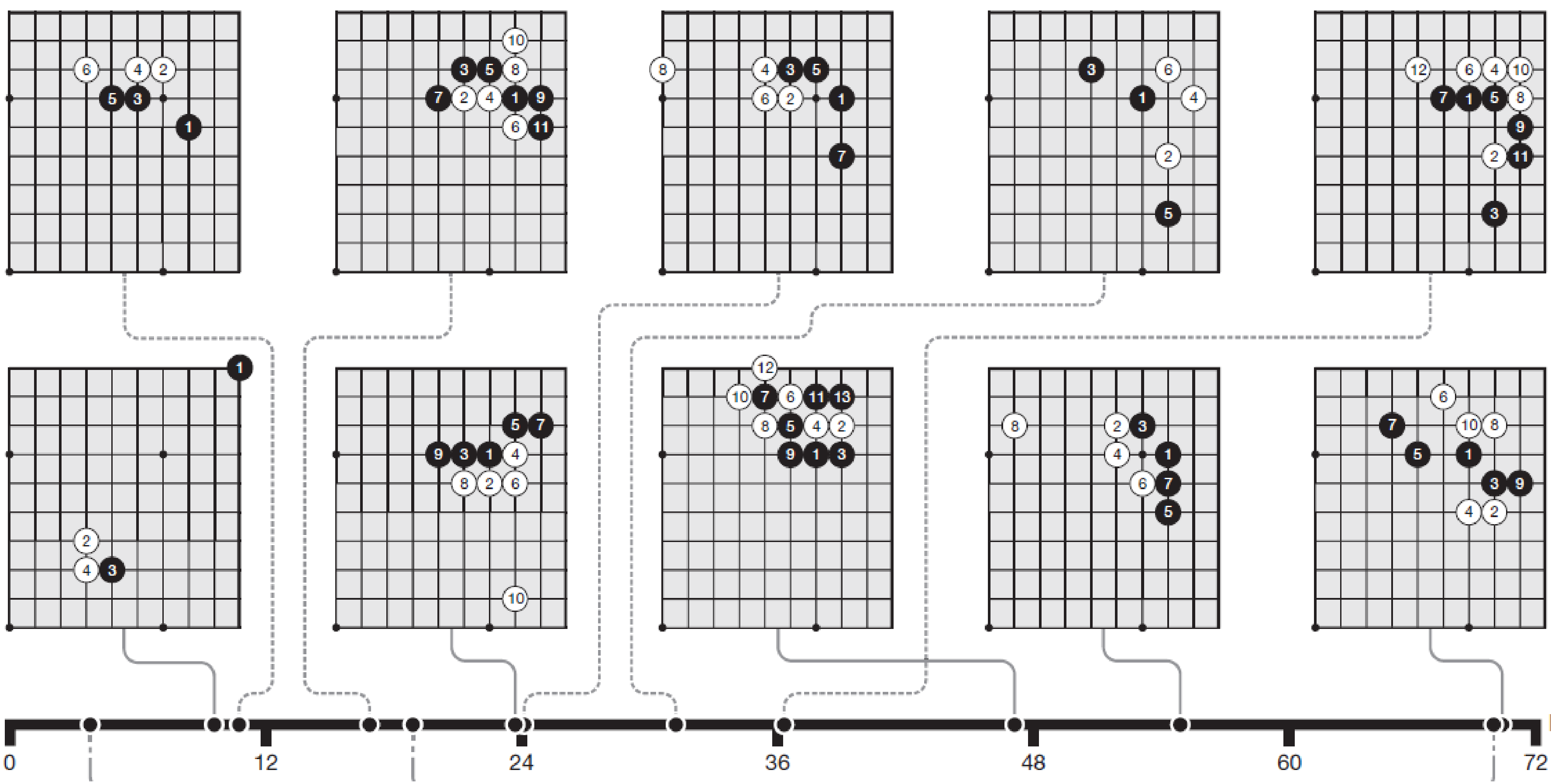
Supervised VS RL



Performance of AlphaGo Zero



Knowledge learned by AlphaGo



What surprises us in AlphaGo Zero

- It outperforms best human player, once thought a decade would take
- No supervised learning is used
- Thus, it does not require domain-specific knowledge for the given problem
- It discovers unknown facts on the game of Go

A deeper look at experience replay

- Experience replay stores past experience as a model
- Replay can be seen as model-based RL
- There is an equivalence between model-based policy evaluation and model-free method with replay (Seijen & Sutton, 2015)
- Experience replay can be seen as stochastic planning method (Pan et al., 2018)

Summary for Model-based RL

- Existing model-based RL methods are mostly tubular methods
- Recent studies using model-based RL shows various methods of manipulating replay mechanism
- The model can be actively used to imitate complex brain modules in animals
  - e.g. curriculum learning or prioritized experience replay methods
- AlphaGo is the most successful case for model-based RL method

Advanced RL Methods

- DQN methods:
  - Double DQN
  - Noisy Networks
  - Prioritized Experience Replay
  - Dueling DQN
  - Categorical DQN
  - DQN-Rainbow
- PG methods:
  - DPG
  - DDPG
  - TRPO
  - PPO
  - TD3

Algorithms

- High-throughput architectures
  - Distributed Prioritized Experience Replay (Ape-X)
  - Importance Weighted Actor-Learner Architecture (IMPALA)
  - Asynchronous Proximal Policy Optimization (APPO)
- Gradient-based
  - Advantage Actor-Critic (A2C, A3C)
  - Deep Deterministic Policy Gradients (DDPG, TD3)
  - Deep Q Networks (DQN, Rainbow, Parametric DQN)
  - Policy Gradients
  - Proximal Policy Optimization (PPO)
- Derivative-free
  - Augmented Random Search (ARS)
  - Evolution Strategies
- Multi-agent specific
  - QMIX Monotonic Value Factorisation (QMIX, VDN, IQN)
- Offline
  - Advantage Re-Weighted Imitation Learning (MARWIL)

References

Moore, A. W., & Atkeson, C. G. (1993). Prioritized sweeping: Reinforcement learning with less data and less time. Machine Learning, 13(1), 103–130..

Pan, Y., Zaheer, M., White, A., Patterson, A., & White, M. (2018). Organizing Experience: A Deeper Look at Replay Mechanisms for Sample-based Planning in Continuous State Domains.

Seijen, H., & Sutton, R. (2015). A deeper look at planning as learning from replay. In *International conference on machine learning* (pp. 2314–2322).

Schaul, T., Quan, J., Antonoglou, I., & Silver, D. (2015). Prioritized experience replay. ArXiv Preprint ArXiv:1511.05952.

Sutton, R. S. (1991). Integrated modeling and control based on reinforcement learning and dynamic programming. In Advances in neural information processing systems (pp. 471–478).

Sutton, R. S., Szepesvári, C., Geramifard, A., & Bowling, M. H. (2008). Dyna-Style Planning with Linear Function Approximation and Prioritized Sweeping.