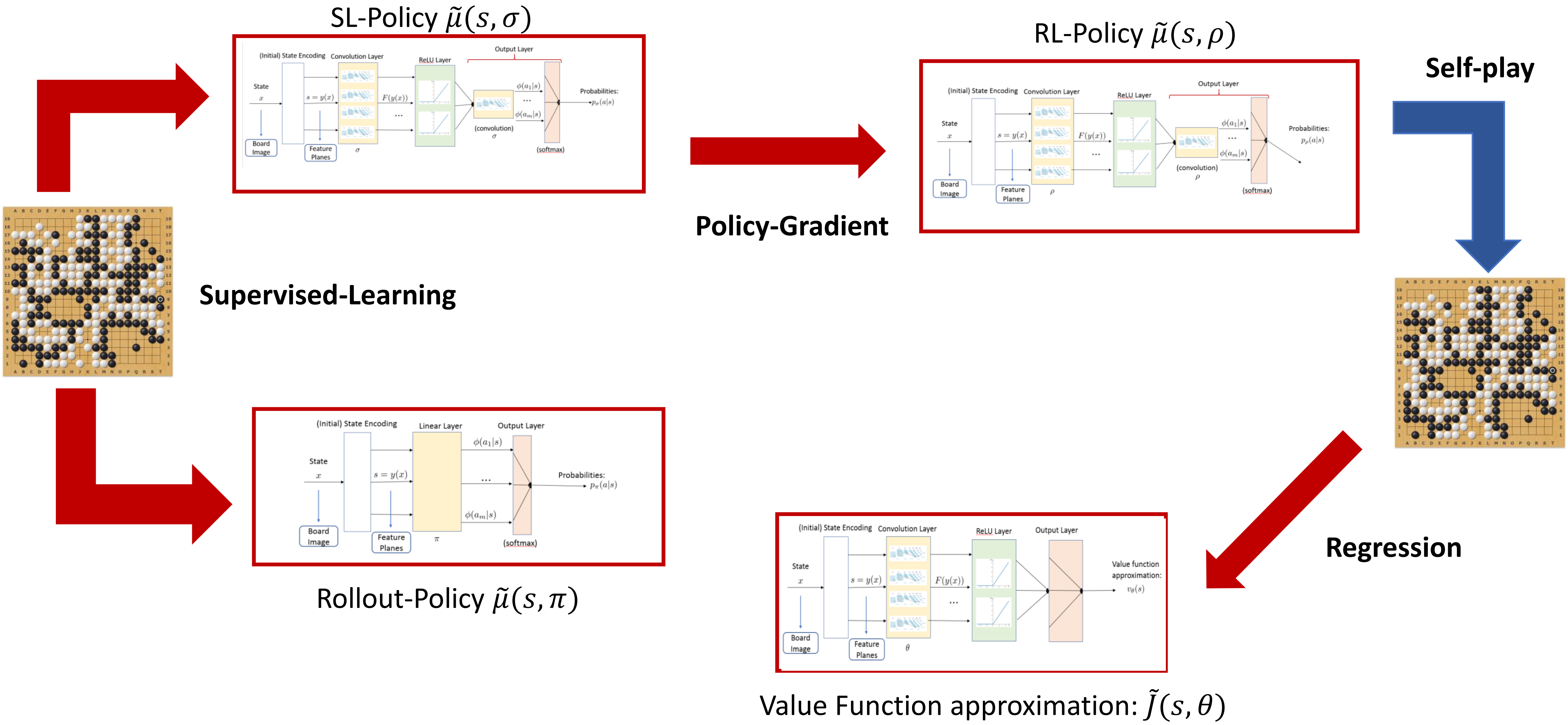
General Overview of the Training Process



Training Process

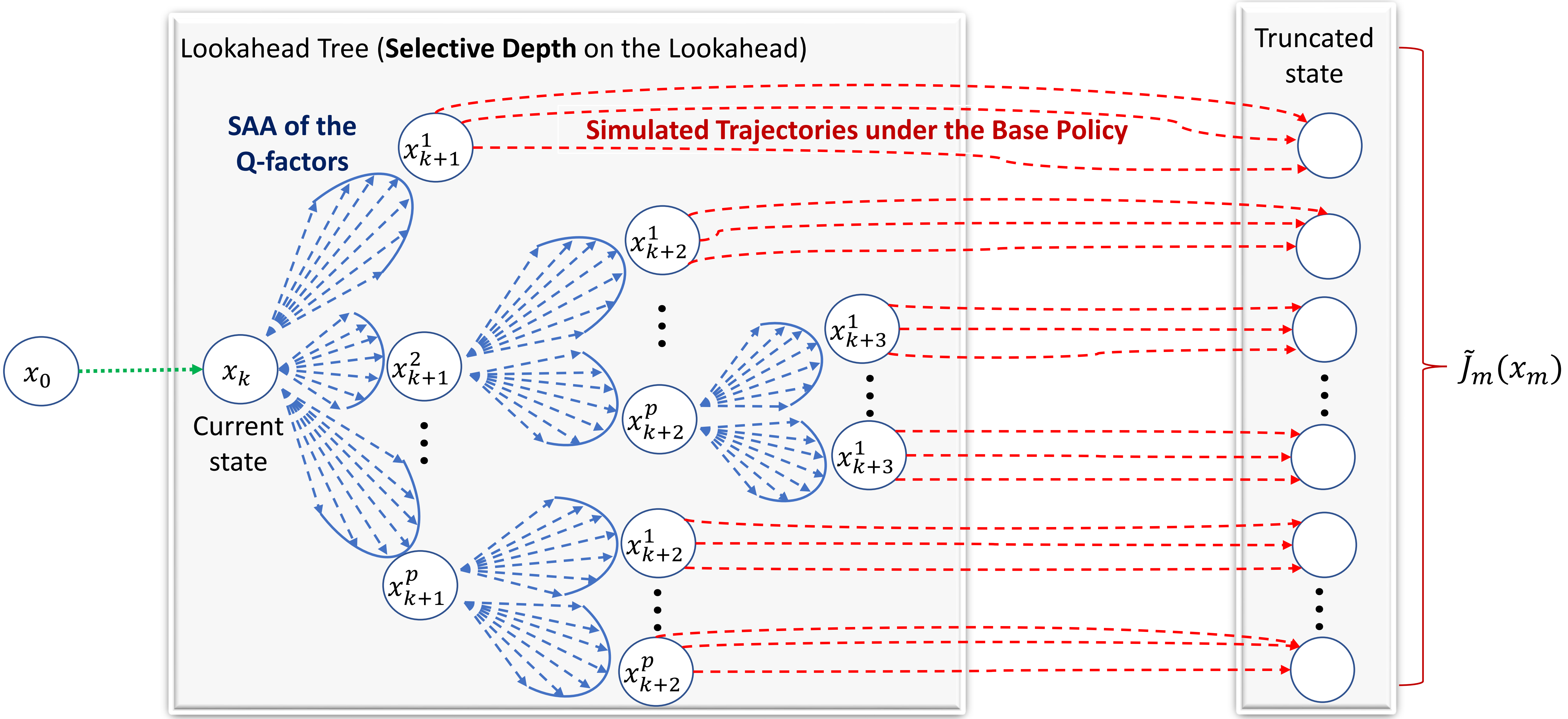
- The overall training process is costly and lengthy.

- Takes a lot of time and a lot of GPU’s.

- When we play online, that is, against an opponent in real-time, there is no more training:
  - We have to output a decision (in reasonable time) so the game can move on
  - Tournaments have strict time limits, so we have to operate in real-time
  - That is where Monte-Carlo Tree Search comes in to play

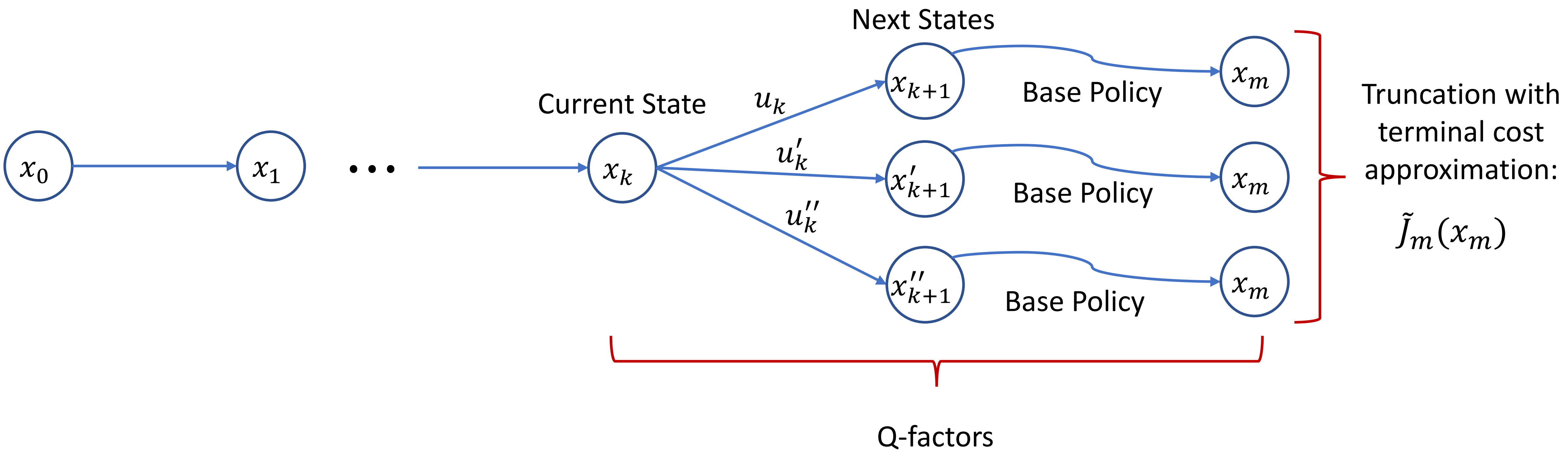
- In addition MCTS gives the policy a degree of adaptation as it is required to play against different opponents
  - Each opponent has a different strategy so the “system” evolves according to different probabilities
  - You cannot train in-between matches

Monte-Carlo Tree Search (MCTS)



Tree Search: Deterministic Case

- Let's quickly recap how the Monte-Carlo Tree Search in the determinist case first:



- So in the Deterministic case, we essentially perform the Rollout Algorithm, with some Base Policy.

Rollout Algorithm: Deterministic Case

- The **Rollout Policy** can be thus defines as:

$$\tilde{\mu}_k(x_k) \in \arg\min_{u_k \in U_k(x_k)} \{g_k(x_k, u_k) + \tilde{J}_{k+1}(f_k(x_k, u_k))\}, \forall k \in \{0, \dots, N-1\}$$

- This is 1-step look ahead minimization where the terminal-cost approximation is given by:

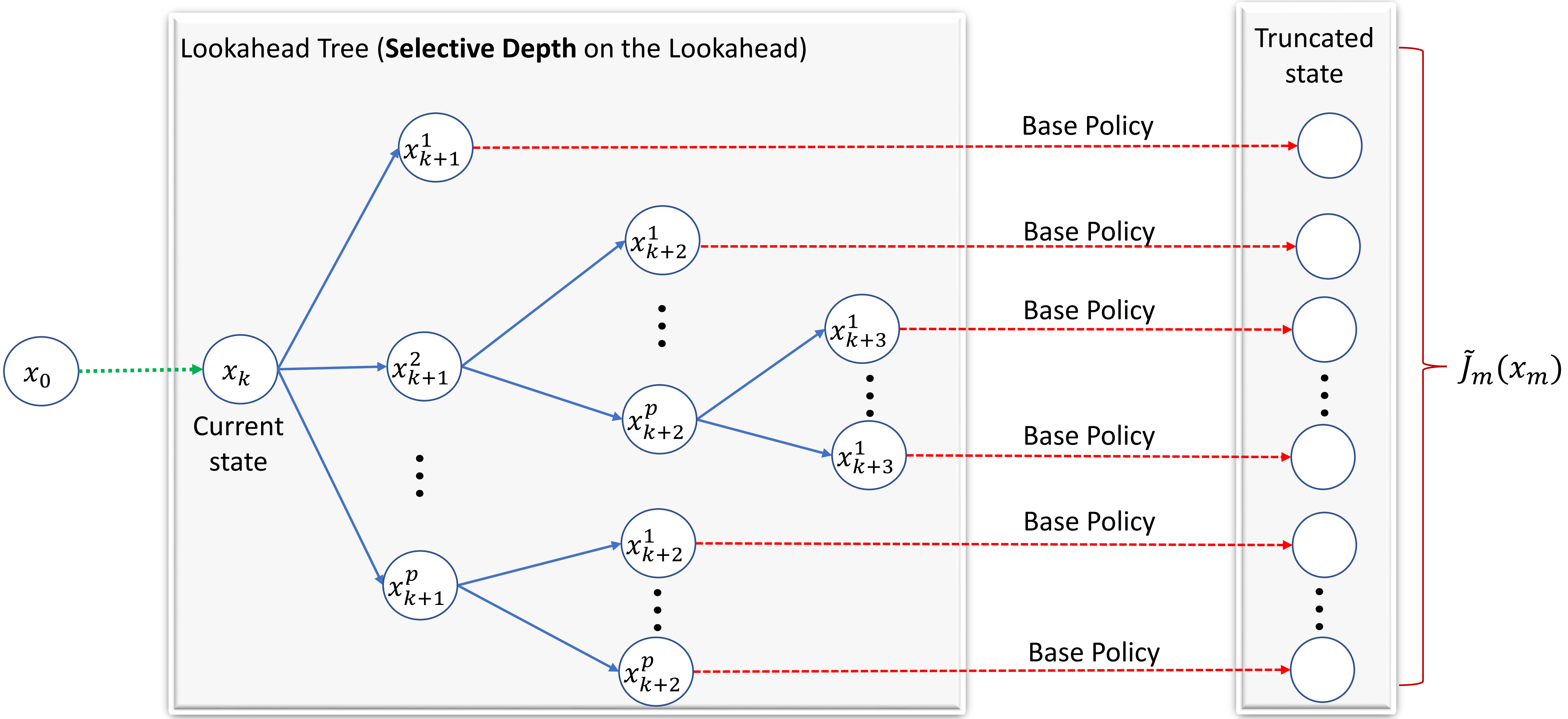
$$\tilde{J}_{k+1}(x_{k+1}) = g_N(x_N) + \sum_{i=k+1}^{M-1} g_i(x_i, \underbrace{\hat{\mu}_i(x_i)}_{\boxed{\text{Base Policy}}}) + \tilde{J}_M(x_m)$$

- Or in terms of the Q-factors:

$$\tilde{\mu}_k(x_k) = \arg\min_{u_k \in U_k(x_k)} \{\tilde{Q}_k(x_k, u_k)\}, \forall k \in \{0, \dots, N-1\}$$

$$\tilde{Q}_k(x_k, u_k) = g_k(x_k, u_k) + \tilde{J}_{k+1}(f_k(x_k, u_k)), \forall k \in \{0, \dots, N-1\}$$

Deterministic Tree Search



Rollout Algorithm: Stochastic Case

- In our case, the problem is stochastic (the policies are randomized!)

- Then the Q-factors become:

$$\tilde{Q}_k(x_k, u_k) = \mathbb{E}_{w_k}[g_k(x_k, u_k, w_k) + \tilde{J}_{k+1}(f_k(x_k, \hat{\mu}_k(x_k), w_k))]$$

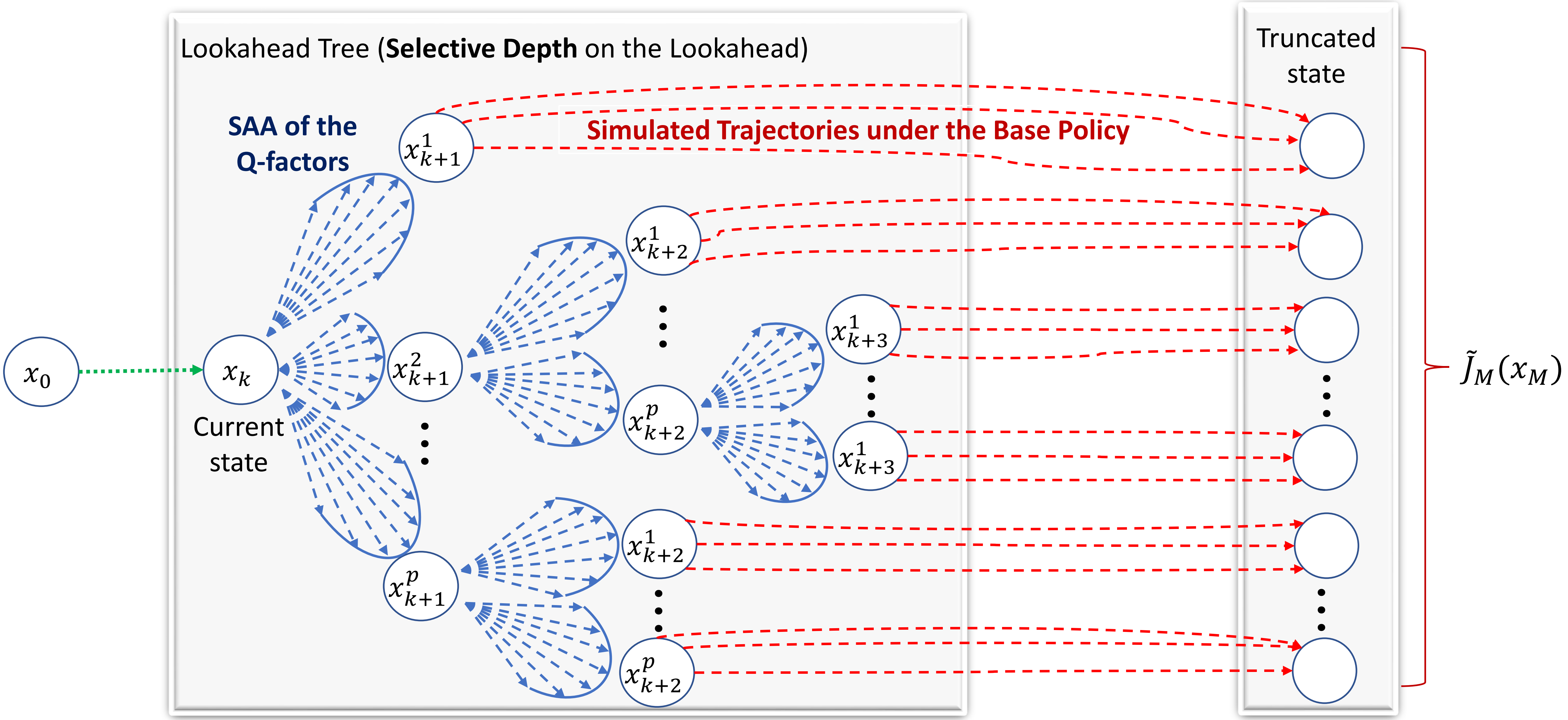
- We cannot compute such expectation in general so we resort to sampling and simulation:

$$\tilde{Q}_k(x_k, u_k) \approx \sum_{s=1}^{S} r_s \big(g_k(x_k, u_k, w_k^s) + \tilde{J}_{k+1}(f_k(x_k, \hat{\mu}_k(x_k), w_k^s))\big)$$

- And the Rollout Policy becomes:

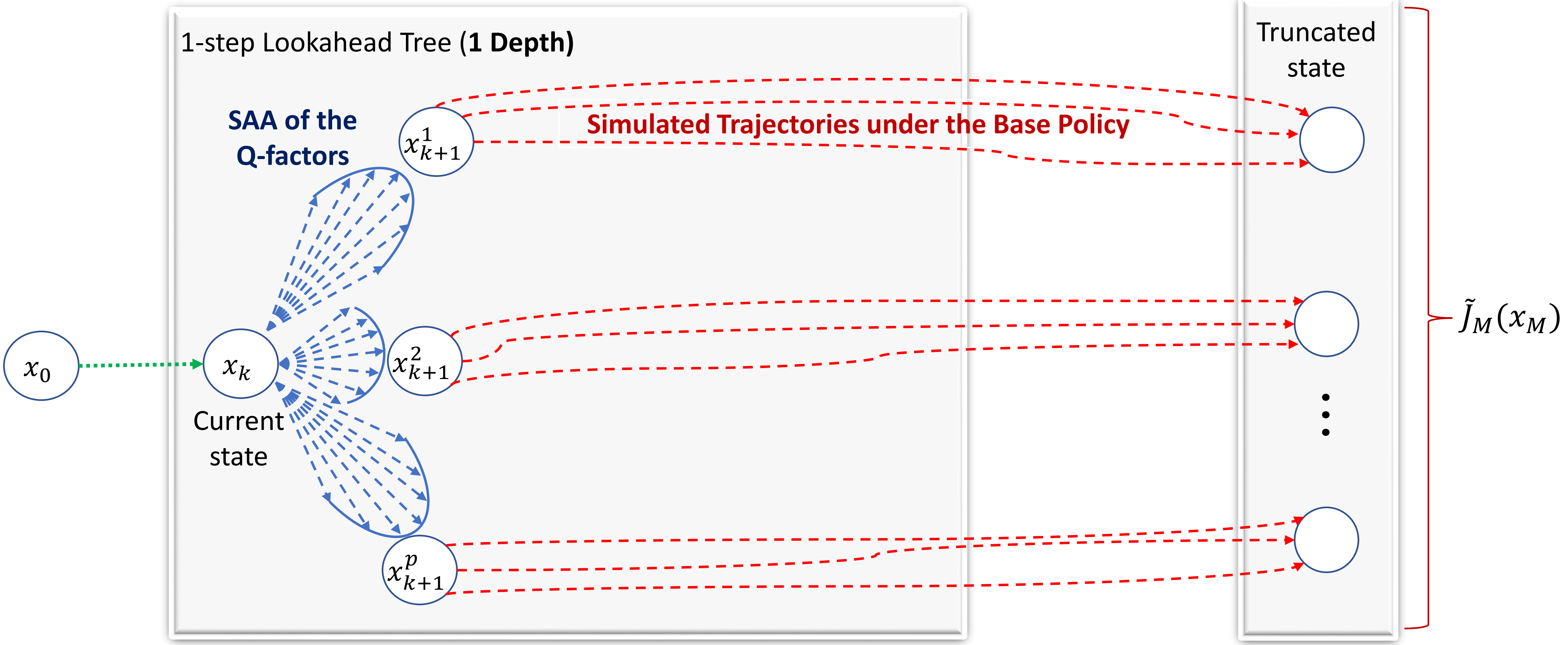
$$\tilde{\mu}_k(x_k) \in \min_{u_k \in U_k(x_k)} \Big\{ \sum_{s=1}^{S} r_s \big(g_k(x_k, u_k, w_k^s) + \tilde{J}_{k+1}(f_k(x_k, \hat{\mu}_k(x_k), w_k^s))\big) \Big\}$$

Monte-Carlo Tree Search (MCTS)



MCTS: Adaptive Sampling

- Let's focus first on 1-look ahead tree. That is a tree with "depth 1".



MCTS: Adaptive Sampling

- Let's now return to the Infinite-Horizon setting with MDP's. The goal is still to compute the Q-factors:

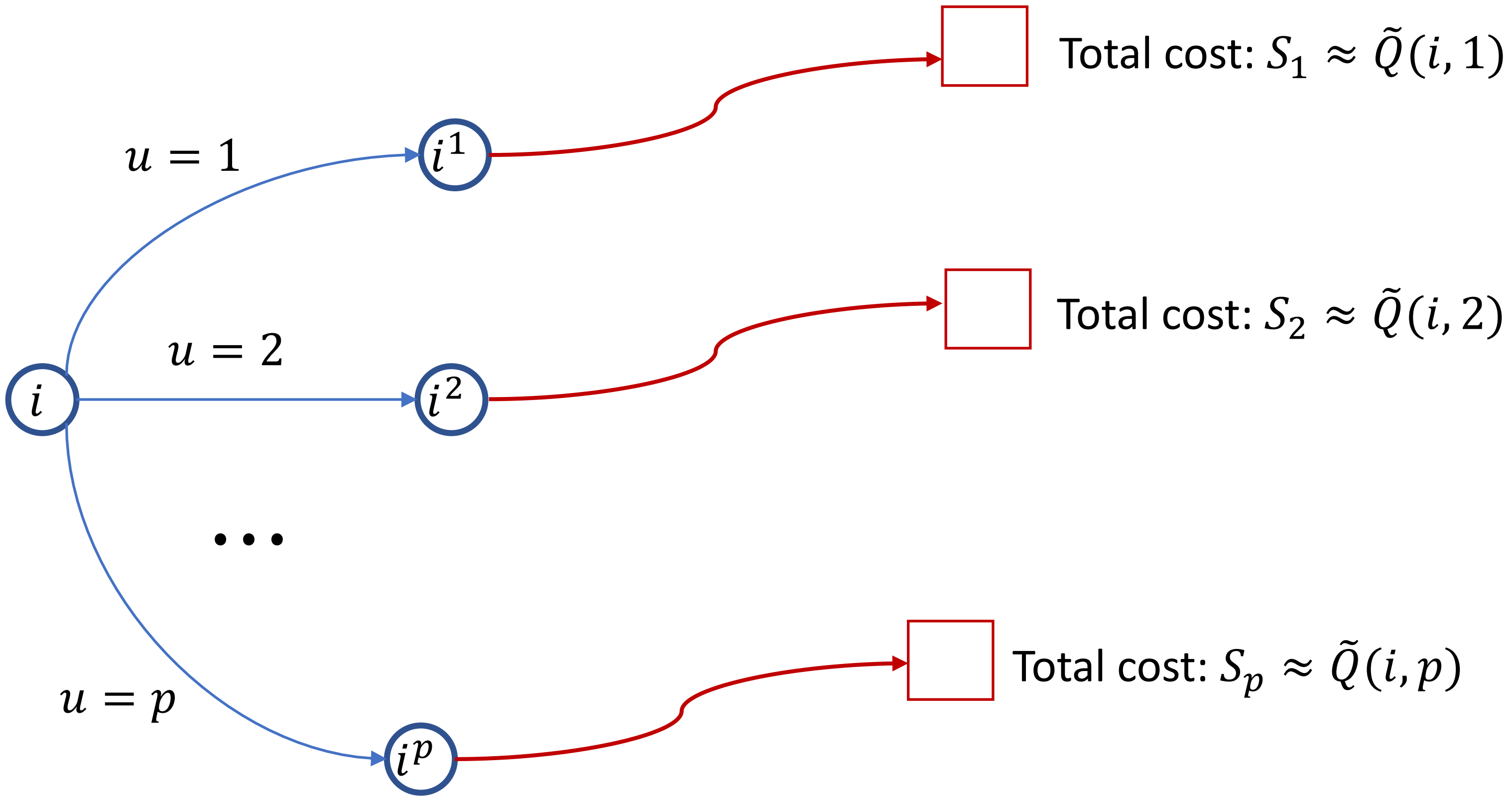
$$Q_{\tilde{\mu}(\pi)}(i_k, u_k) = g(i_k, u_k) + \mathbb{E}_{p(z|\pi)}\left[\sum_{j=k+1}^{\infty} \alpha^{j-k} g(i_j, u_j) \mid i_k^s, u_k^s\right]$$

- Remember: We need to perform the sampling online!
  - No more training.
  - We can perform simulation, but we cannot do as we did before, with huge-scale computations

- The idea is to use **Multi-Armed Bandits**!
  - (or in this case, it's called **Adaptive Sampling**)

MCTS: Adaptive Sampling

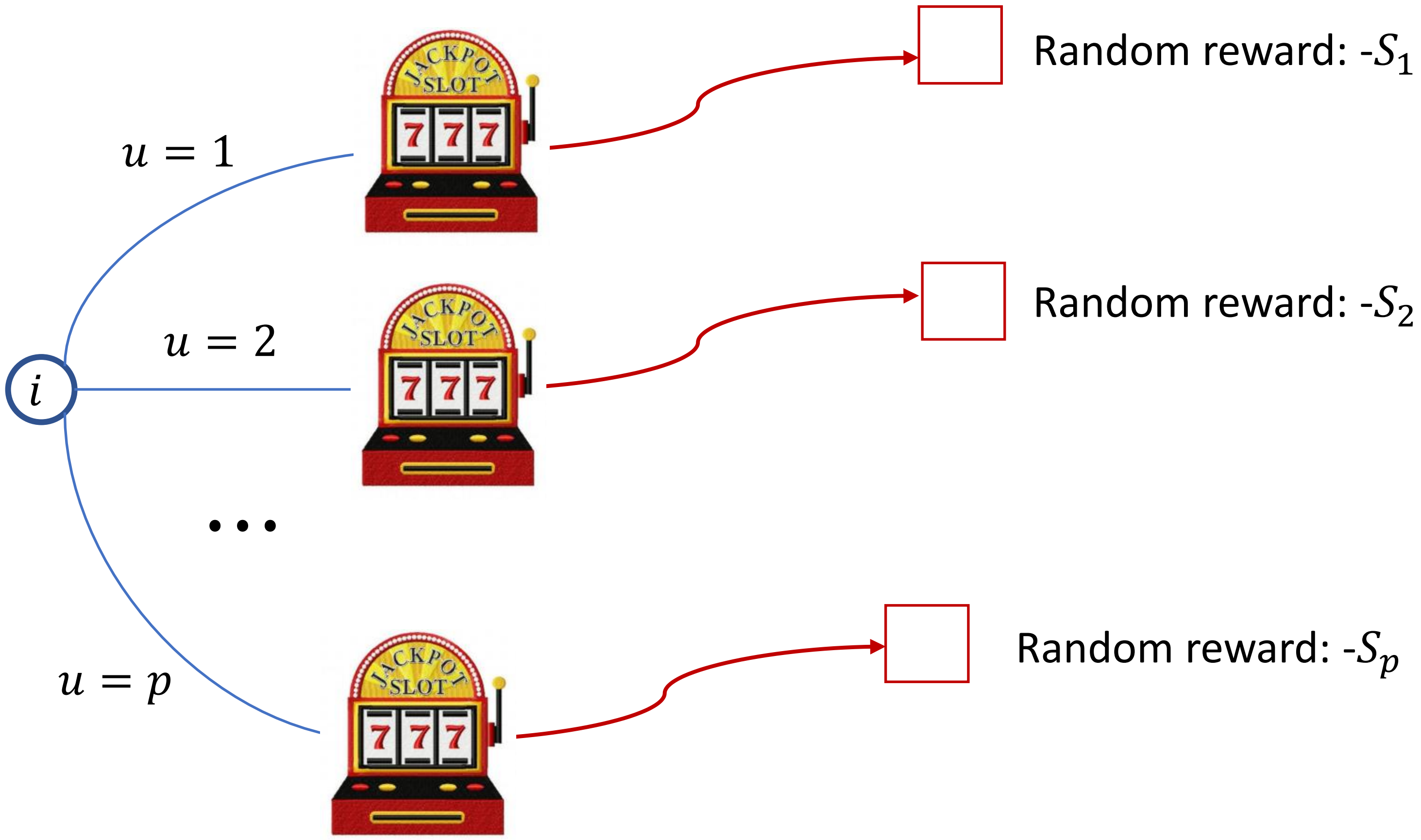
- Let's say at some state i, we have a total of p available controls u $\in (1, \dots, p)$.



- The key question is: Which control to select at each sampling time?

MCTS: Adaptive Sampling

- One way of looking at each is that suppose we are at state i, we have p slot-machines:



Multi-Armed Bandits

- The key question is: Which arm to pull to maximize reward?

MCTS: Adaptive Sampling

- Let's suppose we "pull" the arms T times. Let u^* be the best arm. Then we quantify our **regret** by:

$$R(T) = \sum_{t=1}^{T} S_{u^{(t)}} - T\mathbb{E}[S_{u^*}]$$

- Where $u^{(t)}$ is the "arm" pulled at sampling time $t \in \{1, \ldots, T\}$.

- Want to minimize regret, or rather, provide an arm-selection **policy** that minimizes the "growth" of our regret.

- It turns out that there is a policy (or rule) that achieves $\mathrm{R(T)} = O(\log(T))$, so the regret grows with the "log" of time. And this is optimal.
  - "Auer et al: Finite-time analysis of the multi-armed bandit problem"

MCTS: Adaptive Sampling

- This policy is called the UCB rule (Upper Confidence Bound rule):

$$u^{(t)} = \arg\max_{u} \left\{ \hat{m}_u^{(t)} + \sqrt{\frac{2\ln(t)}{N^{(t)}(u)}} \right\}$$

- Where:

$$\hat{m}_u^{(t)} = \frac{\sum_{j=1}^{t-1} \delta(u^{(j)} = u) S_{u^{(j)}}}{\sum_{j=1}^{t-1} \delta(u^{(j)} = u)} \qquad N_u^{(t)} = \sum_{j=1}^{t-1} \delta(u^{(j)} = u)$$

$$\delta(u^{(j)} = u) = \begin{cases} 1, \text{ if } u^{(j)} = u \\ 0, \text{ if } u^{(j)} \neq u \end{cases} \qquad \forall j \in \{1, ..., p\}$$

MCTS: Adaptive Sampling

- We can interpret this arm-selection rule as exploitation/exploration tradeoff:

$$u^{(t)} = \arg\max_u \left\{\hat{m}_u^{(t)} + \sqrt{\frac{2\ln(T)}{N^{(t)}(u)}}\right\}$$

$$u^{(t)} = \arg\max_u \{\underbrace{Q^{(t)}(u)}_{\text{Exploitation Index}} + \underbrace{R^{(t)}(u)}_{\text{Exploration Index}}\}$$

- So the rule can be tuned to the application by modifying these two indices.

MCTS: Adaptive Sampling

- Let's return to DP. If we were to apply the UCB rule, we need to do for the Q-factors:

$$u^{(t)} = \arg\min_u \{\underbrace{\tilde{Q}^{(t)}(i,u) - R^{(t)}(i,u)}_{\text{Indices are state dependent!}}\}$$

- From UCB, the exploration index should decrease with the number of "visits" to state-control par (i, u).

- So it should follow something like this:

$$R^{(t)}(i,u) \propto \frac{c(t)}{N^{(t)}(i,u)} \qquad N^{(t)}(i,u) = \sum_{j=1}^{t-1} \delta(u^{(j)} = u | i)$$

MCTS: Adaptive Sampling

- There are many different types of rules. We present two variants:

$$R^{(t)}(i,u) = 2c\sqrt{\frac{\ln(\sum_{u \in U(i)} N^{(t)}(i,u))}{N^{(t)}(i,u)}}$$

UCT Rule (extension of UCB)

$$R^{(t)}(i,u) = cP(i,u)\frac{\sqrt{\sum_{u \in U(i)} N^{(t)}(i,u)}}{N^{(t)}(i,u)+1}$$

Alpha-Go (Variation of PUCT)

- More in-depth reading on different indices:
  - “Kocsis Szepesvári, Bandit Based Monte-Carlo Planning, 2006”: UCT
  - “Rosin, Multi-armed bandits with episode context, 2011”: PUCT

MCTS: Adaptive Sampling

- The idea is given some total number of allotted "pulls" T, we keep selecting controls:

$$u^{(t)} = \arg\min_{u} \{\tilde{Q}^{(t)}(i,u) - R^{(t)}(i,u)\}, t \in \{1, \ldots, T\}$$

- And at the end we have our SAA of the Q-factors:

$$\tilde{Q}(i,u) \approx \frac{\sum_{j=1}^{t-1} \delta(u^{(j)} = u) S_{u^{(j)}}}{\sum_{j=1}^{t-1} \delta(u^{(j)} = u)}$$

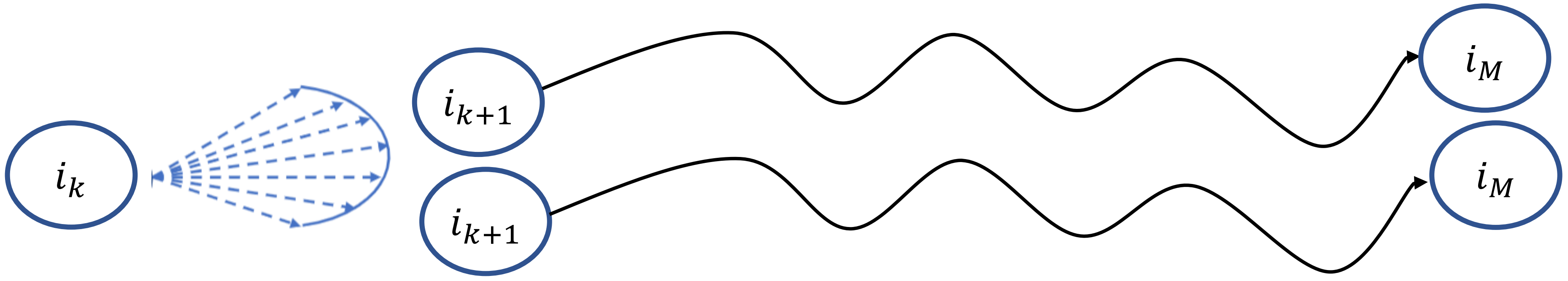
- Where for $(i_k, u_k) = (i, u)$:

$$S_{u^{(t)}} = g(i_k, u_k) + \sum_{j=k+1}^{M-1} \alpha^{j-k} g(i_j, u_j) + \alpha^{M-k} \hat{J}_M(i_M)$$

A single sample of the Q-factor associated with pair (i, u)

MCTS: Adaptive Sampling

- So for a single pair, we can describe it's approximate Q-factor with the following picture:



- And the SAA yields:

$$\tilde{Q}(i,u) \approx \frac{\sum_{j=1}^{t-1} \delta(u^{(j)} = u) S_{u^{(j)}}}{\sum_{j=1}^{t-1} \delta(u^{(j)} = u)}$$

MCTS on AlphaGo

- Now let's return to AlphaGo. The implementation of MCTS follow closely what we covered with a few modifications. Let's still focus on the case 1-step look (so a tree with "depth 1").

- Recall that:
  - s is the board state.
  - $a \in A(s)$ are the legal actions.
  - We are maximizing the probability of winning the game.
  - We have our trained DNN's $(\sigma^*, \rho^*, \pi^*, \theta^*)$ at our disposal.

MCTS on AlphaGo

- Suppose the game is at some state s, and we can perform a total of T simulation steps.
  - Or "T pulls" of the bandit

- Then at each "pull" t, Alpha-Go will select the action a according to:

$$a^{(t)} = \arg\max_a \left\{ \tilde{Q}^{(t)}(s,a) + R^{(t)}(s,a) \right\}$$

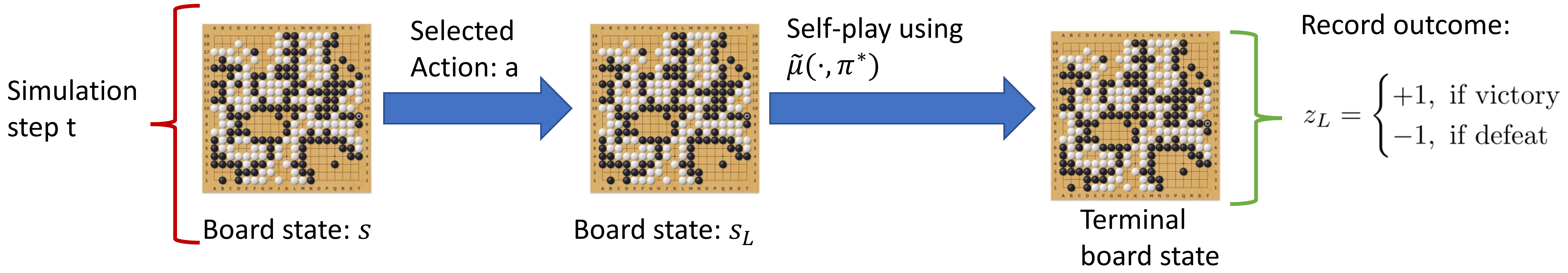
- Where

$$R^{(t)}(s,a) = cP(s,a) \frac{\sqrt{\sum_{a \in A(s)} N^{(t)}(s,a)}}{N^{(t)}(s,a) + 1}$$

- Now, Alpha-Go could call any of the trained policies $(\sigma^*, \rho^*, \pi^*)$ to act as the Base Policy.

MCTS on AlphaGo

- After applying action a the board evolves from s to s_L. ("L" stands for "leaf" of the tree)
- Then it uses the policy π^* to play the game until conclusion and records the outcome.
  - Recall π^* is the "inaccurate policy" but it's very fast to compute the actions.
- This can be represented as follows:



MCTS on AlphaGo

- Alpha-Go also evaluates the position s_L by using the Value Network $v_{\theta^*}(s_L)$
  - Recall: $v_{\theta^*}(\cdot)$ acts as the critic, it provides an approximation to the probability of winning.

- Then it computes the value function ("cost-to-go") from state s_L as convex combination of the critic valuation and the outcome of the base policy (the outcome of self-play using π^*:

$$V(s_L) = (1-\lambda)v_{\theta^*}(s_L) + \lambda z_L$$

- Note that $V(\cdot)$ is still a valid approximation for the Value Function, as it is "mixing" the critic output with a self-play coming from executing the base policy (in the Rollout Algorithm Fashion).

MCTS on AlphaGo

- Then it computes the approximate Q-factor:

$$\tilde{Q}^{(t)}(s,a) \approx \frac{\sum_{j=1}^{t-1} \delta(a^{(j)} = a|s) V(s_L^{(t)})}{\sum_{j=1}^{t-1} \delta(a^{(j)} = a|s)}$$

$$V(s_L) = (1-\lambda) v_{\theta^*}(s_L) + \lambda z_L$$

- Lastly we need to compute $P(s,a)$:
  - We do so by calling either σ^* or ρ^*(AlphaGo uses σ^*)

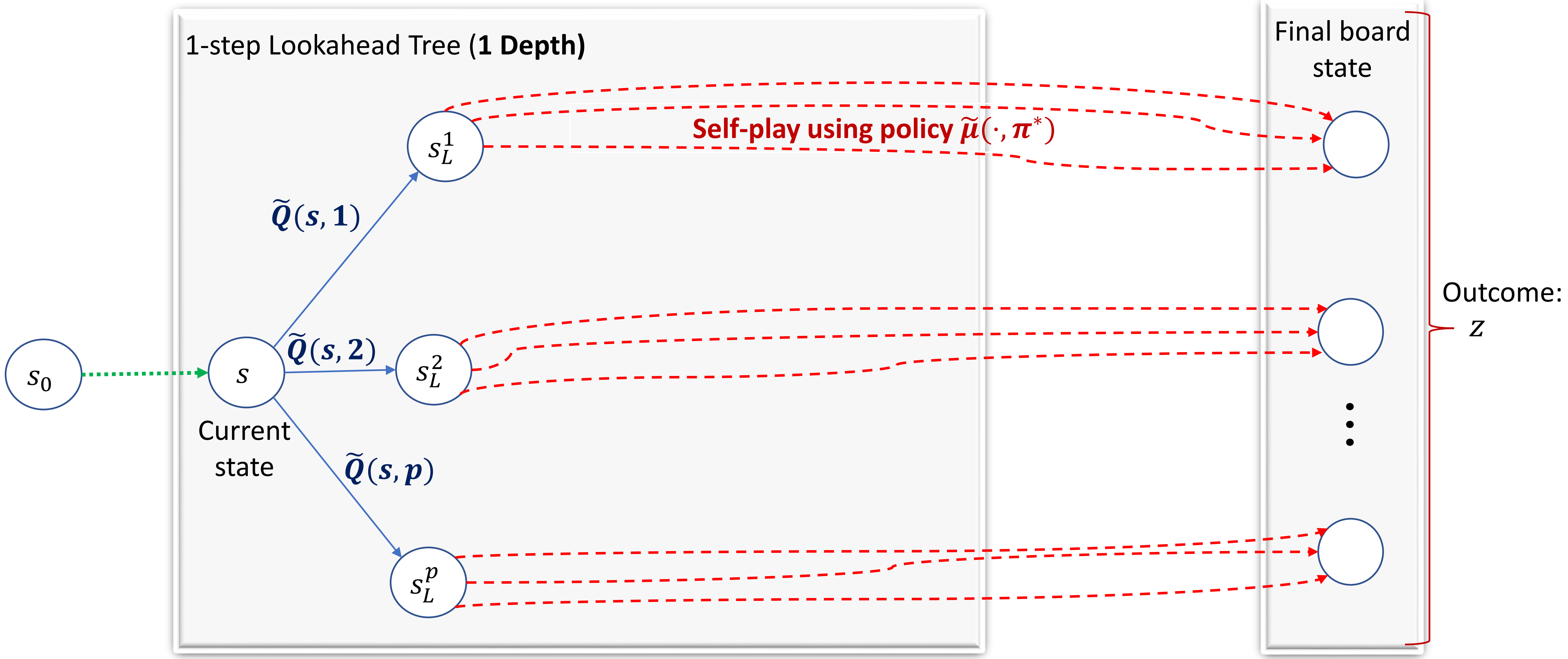
$$P(s,a) \approx p_{\sigma^*}(a|s)$$

- Recall:

$$\begin{cases} \tilde{\mu}(s,\sigma) = a, \text{ w.p.} \quad p_\sigma(a|s) \\ p_\sigma(a|s) = \dfrac{e^{\beta\phi(a|s)}}{\sum_{a' \in A(s)} e^{\beta\phi(a'|s)}} \end{cases}$$

MCTS on AlphaGo

- But this process works for a Tree "1-depth":



MCTS on AlphaGo

- Now we need to expand the tree. That is we need to implement a **selective-depth** lookahead tree.
- There are many ways to do this. Alpha-Go implements asynchronous updates and leverages parallel computing to process many leaves at the same time.
- Note that every leaf that sprouts of a node is actually a **variations** of the board position.
- So AlphaGo explores several variations of a given position while playing the game.
  - This is often called “lines of play”, as different variations lead to different tactics and so forth.
- But in essence, we will expand the Tree by computing a score for each variation.

MCTS on AlphaGo

- In the simplest case the scoring function is simply the counts $N^{(t)}(s,a)$.

- So if $N^{(t)}(s,a) > \tau$, where τ is some threshold. Then we add the resulting leaf s_L to the tree and we expand it by considering all possible moves out of s_L.

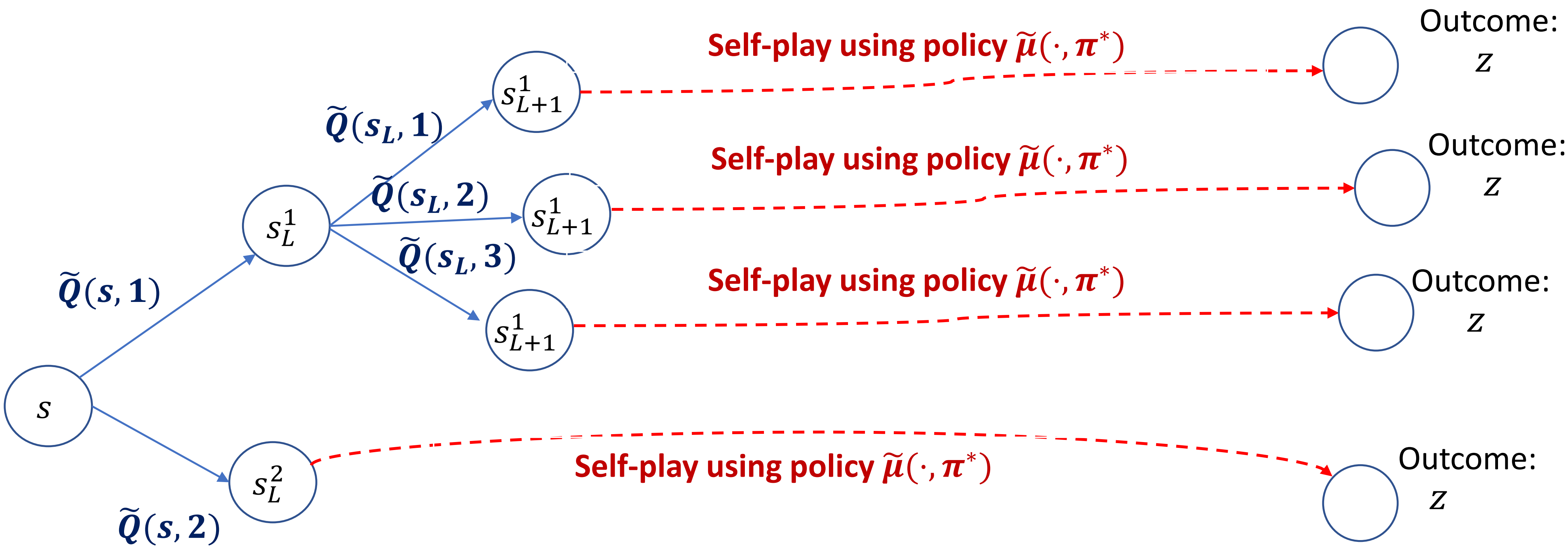
- The probabilities $P(s_l,a)$ are updates using $p_{\sigma^*}(s_L,a)$ and the new counts from s_L $N(s_L,a)$ are set to zero.

- Then on the next simulation step, the self-play will start from s_L if it is visited.

- The Q-factors are updated **backwards**, from leaves towards the root.

MCTS on AlphaGo

- This is best shown with a figure:



MCTS on AlphaGo

- Then for every edge (s, a) we store the following quantities:
  - $N(s, a)$: number of visits to that edge in the simulation
  - $\tilde{Q}(s, a)$: Q-factor value of the edge
  - $P(s, a)$: probability of visiting the edge
- At the end of T simulation steps, we have a selective-depth lookahead tree, where each edge on the Tree contain the above information.
- Now we can finally answers the question on which move do we make.

MCTS on AlphaGo

- Surprisingly, the **actual** policy used by AlphaGo to play the game is a **Rollout Policy** of the following form:

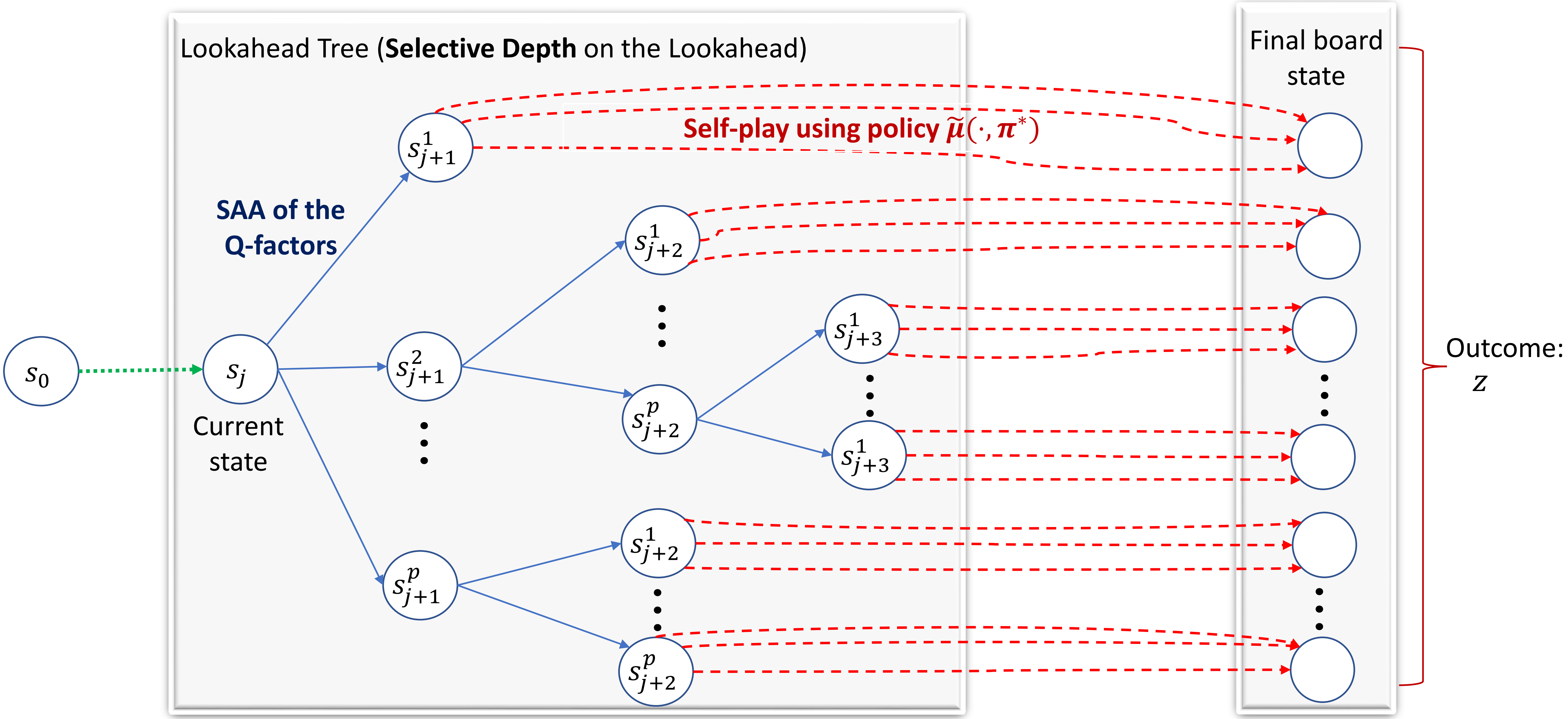
$$\mu(s) = \arg\max_a \left\{ N^{(T)}(s,a) \right\}$$

- So AlphaGo picks the move that was most used during the Tree Search.
    - It’s interesting that it is the most common move, instead of the move with highest chance of winning.

- For example, it could have used:

$$\mu(s) = \arg\max_a \left\{ \tilde{Q}^{(T)}(s,a) \right\}$$

MCTS in AlphaGo



AlphaGo Performance

- Some statistics of the complete AlphaGo AI software:
  - Using all trained DNN’s
  - Using MCTS

- During online play (tree search) AlphaGo used 8 GPU’s, 48 CPU’s. A distributed version of AlphaGo that exploits multiple machines, has 176 GPU’s and 1202 CPU’s.

- AlphaGo has a time-allocating strategy for it’s MCTS:
  - It allocates time to solve any divergence in the final move computation
  - It allocates time, prioritizing the mid-game
  - “Huang et al. Time management in Monte-Carlo tree search applied to the game of Go. 2010”.

- AlphaGo’s resigns the game if the computed probability of winning is less than 10%.

AlphaGo Performance

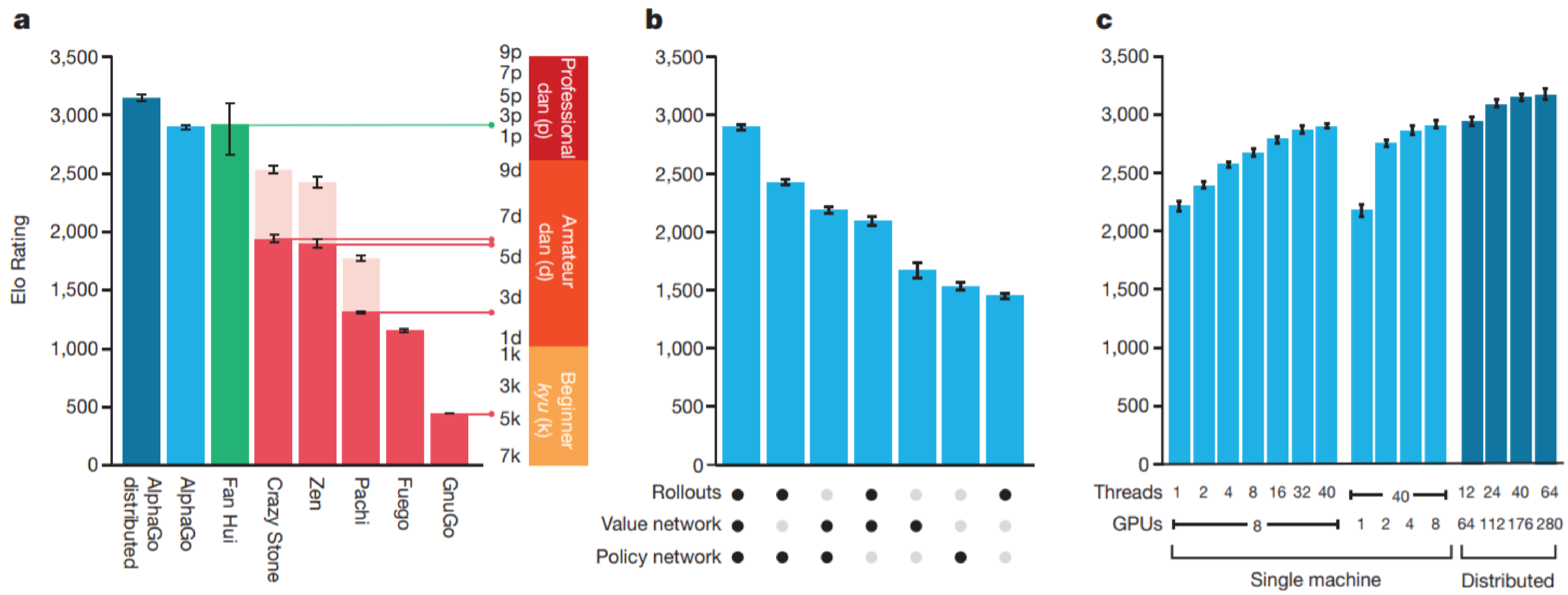


Figure taken from "Silver et al. Mastering the game of Go with deep neural networks and tree search
(Original paper for AlphaGo)