Recap: Actor-Critic Algorithm

Algorithm 1 Actor-Critic Algorithm

Input: Initial DNN parameters $\theta^{(0)}$, randomized policy $\tilde{\mu}(\theta^{(0)})$.

Input: Initial DNN parameters $\phi^{(0)}$, cost-go-go approximate function $\tilde{J}(\cdot, \phi^{(0)})$.

1: **for** $t = 0, ..., T$ **do** (obtaining new samples)

2: Collect S sample trajectories $z^s = (i_0^s, u_0^s, ..., i_M^s)$ using the policy $\tilde{\mu}(\theta^{(t)})$

3: Perform the **critic step**:

$$\phi_{m+1}^{(t)} = \phi_m^{(t)} - \gamma_p^{(t)} \sum_{l=1}^{L} \nabla_\phi \tilde{J}(i_0^l, \phi)\big(\tilde{J}(i_0^l, \phi) - \beta_p^l\big), \quad \forall m \in \{1, ..., P\}$$

$$\phi^{(t+1)} \leftarrow \phi_P^{(t)}$$

4: Evaluate the advantage $\tilde{A}(i_k^s, u_k^s)$ for every sample pair (i_k^s, u_k^s).

5: Compute the policy gradient:

$$\nabla_\theta\big(\mathbb{E}_{p(z|\theta^{(t)})}\big[F(z)\big]\big) \approx \frac{1}{S}\sum_{s=1}^{S}\sum_{k=0}^{M-1} \nabla_\theta\big(\ln(p(u_k^s|i_k^s, \theta^{(t)}))\Big(\tilde{A}(i_k^s, u_k^s)\Big)\Big]$$

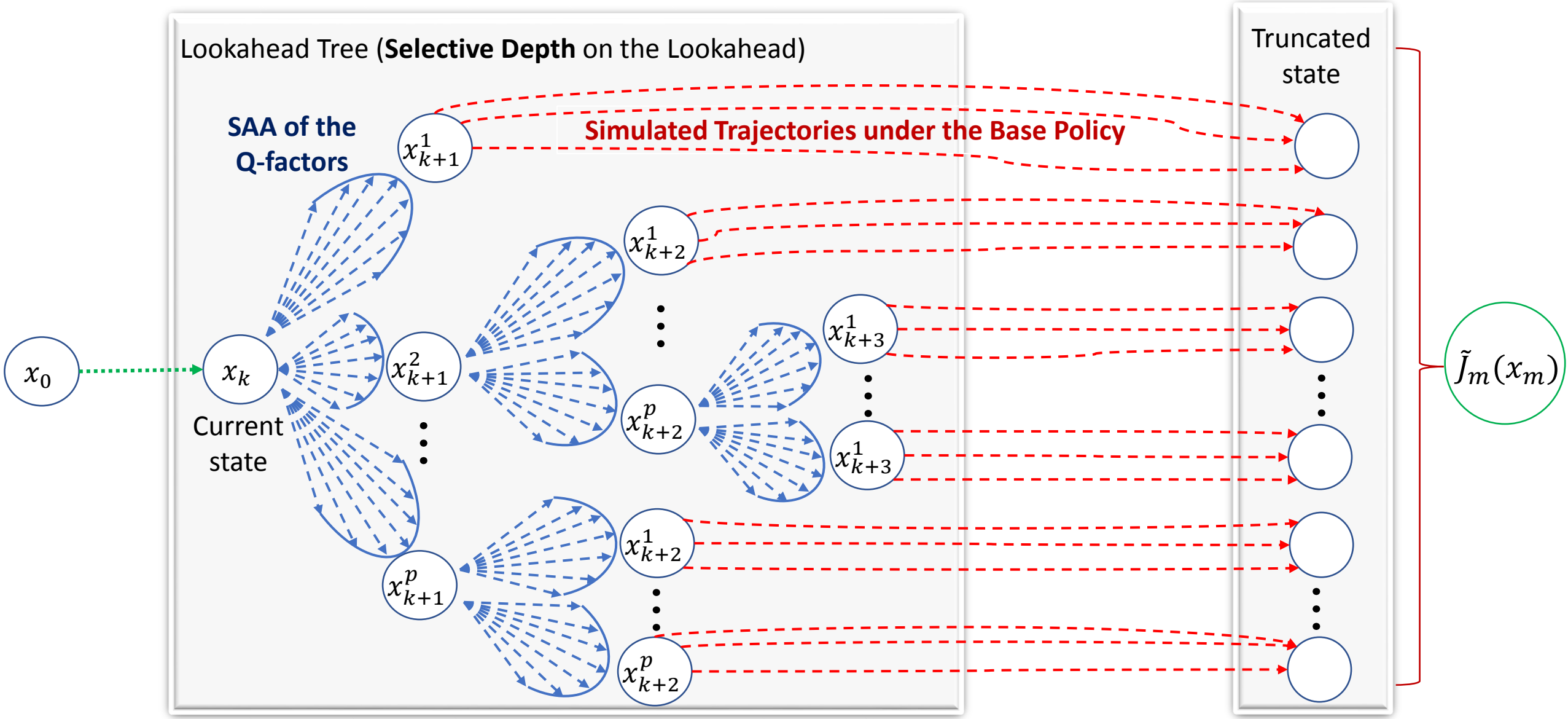
6: Perform the **actor-step** (gradient-step):

$$\theta^{(t+1)} = \theta^{(t)} - \gamma^{(t)} \nabla_\theta\big(\mathbb{E}_{p(z|\theta^{(t)})}\big[F(z)\big]\big)$$

7: **end for**

Output: The last DNN configurations $\theta^{(T)}$ and $\phi^{(T)}$. A suboptimal policy $\tilde{\mu}(\theta^{(T)})$. An approximate cost-to-go function $\tilde{J}(\cdot, \phi^{(T)})$

Monte-Carlo Tree Search (MCTS)



AlphaGo (Silver et al, 2016)

- The AlphaGo AI software uses handcrafted features, based on experience.
- The state x_k is the board position at the $k'th$ turn, which is extremely complex to represent. Therefore, the following features are used to represent the state:

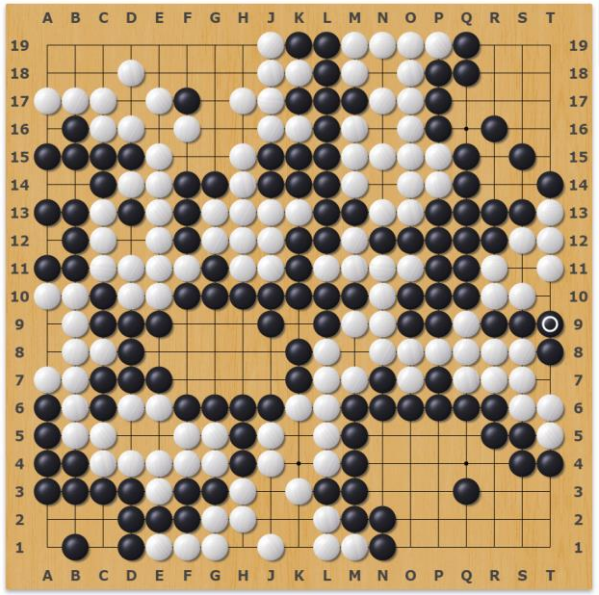
Extended Data Table 2 | Input features for neural networks

Feature	# of planes	Description
Stone colour	3	Player stone / opponent stone / empty
Ones	1	A constant plane filled with 1
Turns since	8	How many turns since a move was played
Liberties	8	Number of liberties (empty adjacent points)
Capture size	8	How many opponent stones would be captured
Self-atari size	8	How many of own stones would be captured
Liberties after move	8	Number of liberties after this move is played
Ladder capture	1	Whether a move at this point is a successful ladder capture
Ladder escape	1	Whether a move at this point is a successful ladder escape
Sensibleness	1	Whether a move is legal and does not fill its own eyes
Zeros	1	A constant plane filled with 0
Player color	1	Whether current player is black

Feature planes used by the policy network (all but last feature) and value network (all features).

Features are like binary ("one-hot") encodings of the board picture (a 19x19 image)

Binary Encoding with feature planes



19x19 image of the board

Feature Encoding

Feature: “Position Color”:

1. Black
2. White
3. Empty

19x19 binary matrices
(“feature planes”)

$$\begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Black

$$\begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

White

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

Empty

Policy Network: supervised-RL

- The first architecture tries to approximate a policy. As we saw before, it can be written as:

$$\tilde{\mu}(s,\sigma) = a, \text{ w.p. } \quad p_\sigma(a|s)$$

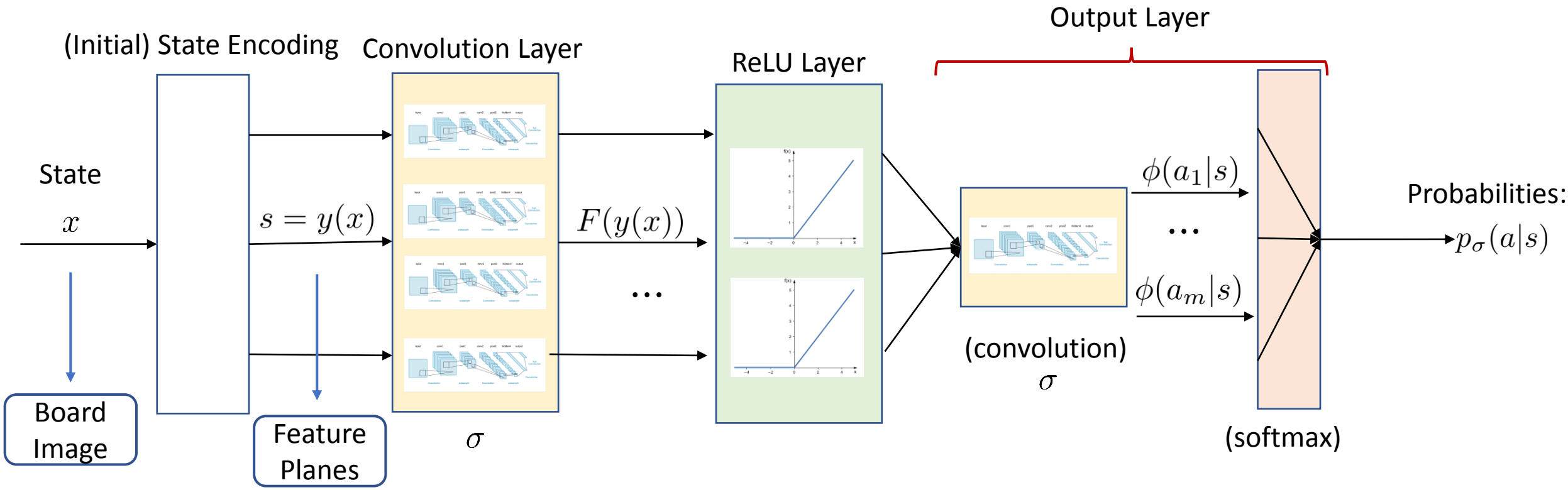
- This policy is randomized, i.e: it selects a random action a given state s with probability $p_\sigma(a|s)$. In particular the probabilities are given by:

$$p_\sigma(a|s) = \frac{e^{\beta\phi(a|s)}}{\sum_{a' \in A(s)} e^{\beta\phi(a'|s)}}$$

- Where β is a hyper-parameter ("soft-max temperature"). AlphaGo uses $\beta = 0.67$

Policy Network: Architecture overview

- The DNN can be summarized as follows, with hidden layer:



- AlphaGo uses 13 units like this (Convolution + ReLU) in the policy approximation $\tilde{\mu}(s, \sigma)$

Policy Network: Supervised Learning

- The policy approximation $\tilde{\mu}(s, \sigma)$ is trained in a "actor-only" fashion, where instead of policy gradient it relies on Supervised Learning. We will called it the *SL-policy*.

- It uses a data set of board states s and actions a, obtained from a database of "expert players"
    - KGS Go Server: available online.
    - In this case, players ranked from 6-9 *dan*.

- Then it solves a Maximum-Likelihood Estimation (MLE) problem:

$$\max_{\sigma} \left\{ F_{\mathrm{MLE}}(\sigma) \right\} = \max_{\sigma} \left\{ \prod_{k=1}^{K} p_{\sigma}(a^k | s^k) \right\}$$

- Where K is the total number of sample pairs (s^k, a^k)

Policy Network: Supervised Learning

- We can, of course, take the log:

$$\max_{\sigma} \{ f_{\mathrm{MLE}}(\sigma) \} = \max_{\sigma} \{ \sum_{k=1}^{K} \ln(p_{\sigma}(a^k|s^k)) \}$$

- Then it solves this problem via SGD, by taking the log of the objective and using sample batch B:

$$\nabla_{\sigma} f_{\mathrm{MLE}}(\sigma) \approx \frac{1}{B} \sum_{k=1}^{B} \nabla_{\sigma} \big(\ln(p_{\sigma}(a^k|s^k)) \qquad \sigma^{(t)} = \sigma^{(t)} + \alpha^{(t)} \hat{\nabla}_{\sigma} f_{\mathrm{MLE}}(\sigma)$$

- This type of training is often called **Imitation Learning,** because the policy is being trained to "imitate" as closely as possible the expert moves given any provided board state.

Policy Network: Supervised Learning

- A few statistics:

- The data set has 29.4 million state-action pairs $\left(s^k, a^k\right)$ collected from 160K games
  - The training set contains 28.4 million pairs
  - The validation set contains 1 million pairs
  - Pass moves were excluded
  - Data-set was augmented using symmetry. (recall the Tic-Tac-Toe example!)

- The SGD batch B size is 16 (very small).

- Step-size $\alpha^{(t)}$ was initialized to 0.003 and halved every 80 million training steps.

- Training was done using 50 GPU's, 340 million gradient-steps and took 3 weeks to finish!

Policy Network: Supervised Learning

- The training results were given in the table:

Extended Data Table 3 | Supervised learning results for the policy network

Architecture			Evaluation				
Filters	Symmetries	Features	Test accuracy %	Train accuracy %	Raw net wins %	*AlphaGo* wins %	Forward time (ms)
128	1	48	54.6	57.0	36	53	2.8
192	1	48	55.4	58.0	50	50	4.8
256	1	48	55.9	59.1	67	55	7.1
256	2	48	56.5	59.8	67	38	13.9
256	4	48	56.9	60.2	69	14	27.6
256	8	48	57.0	60.4	69	5	55.3
192	1	4	47.6	51.4	25	15	4.8
192	1	12	54.7	57.1	30	34	4.8
192	1	20	54.7	57.2	38	40	4.8
192	8	4	49.2	53.2	24	2	36.8
192	8	12	55.7	58.3	32	3	36.8
192	8	20	55.8	58.4	42	3	36.8

The policy network architecture consists of 128, 192 or 256 filters in convolutional layers; an explicit symmetry ensemble over 2, 4 or 8 symmetries; using only the first 4, 12 or 20 input feature planes listed in Extended Data Table 1. The results consist of the test and train accuracy on the KGS data set; and the percentage of games won by given policy network against AlphaGo's policy network (highlighted row 2): using the policy networks to select moves directly (raw wins); or using AlphaGo's search to select moves (AlphaGo wins); and finally the computation time for a single evaluation of the policy network.

"Size" of the Architectures

"best" trade-off between accuracy and speed

Highest accuracy, but slower computation

Table taken from Silver et al, 2016.

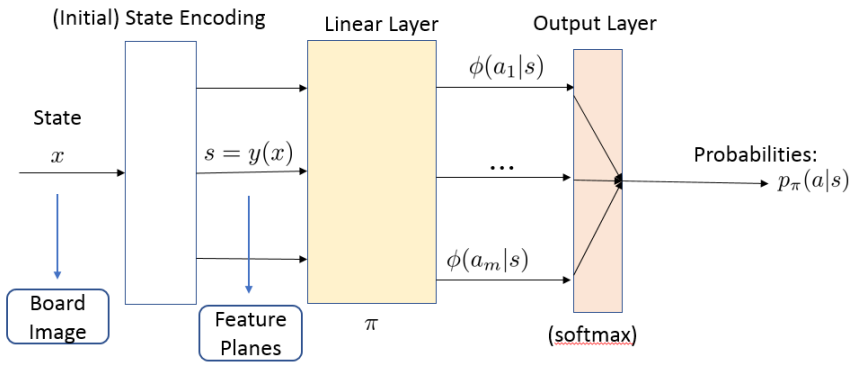
Policy Network: another one

- Alpha-GO, in fact, trains another policy network:

$$\tilde{\mu}(s,\pi) = a, \text{ w.p. } \quad p_\pi(a|s)$$

- It is trained similarly, via SGD for MLE. However the Network architecture is much simpler:

Simpler architecture



Smaller set of features

Extended Data Table 4 | Input features for rollout and tree policy

Feature	# of patterns	Description
Response	1	Whether move matches one or more response pattern features
Save atari	1	Move saves stone(s) from capture
Neighbour	8	Move is 8-connected to previous move
Nakade	8192	Move matches a *nakade* pattern at captured stone
Response pattern	32207	Move matches 12-point diamond pattern near previous move
Non-response pattern	69338	Move matches 3 × 3 pattern around move
Self-atari	1	Move allows stones to be captured
Last move distance	34	Manhattan distance to previous two moves
Non-response pattern	32207	Move matches 12-point diamond pattern centred around move

Features used by the rollout policy (first set) and tree policy (first and second set). Patterns are based on stone colour (black/white/empty) and liberties (1, 2, ≥3) at each intersection of the pattern.

Policy Network: another one

- This policy $\tilde{\mu}(s, \pi)$ is called the *Rollout Policy*
  - this name should ring some bells!
  - In our definitions, it is in fact the base policy, of a potential Rollout Algorithm

- As a comparison, the differences between $\tilde{\mu}(s, \sigma)$ and $\tilde{\mu}(s, \pi)$ are as follows:

- $\tilde{\mu}(s, \sigma)$ achieved an accuracy of 55.7% in the test set, but takes roughly $3ms$ to compute the action.

- $\tilde{\mu}(s, \pi)$ achieved an accuracy of 24.2% in the test set, but takes just $2\mu s$ to compute the action.

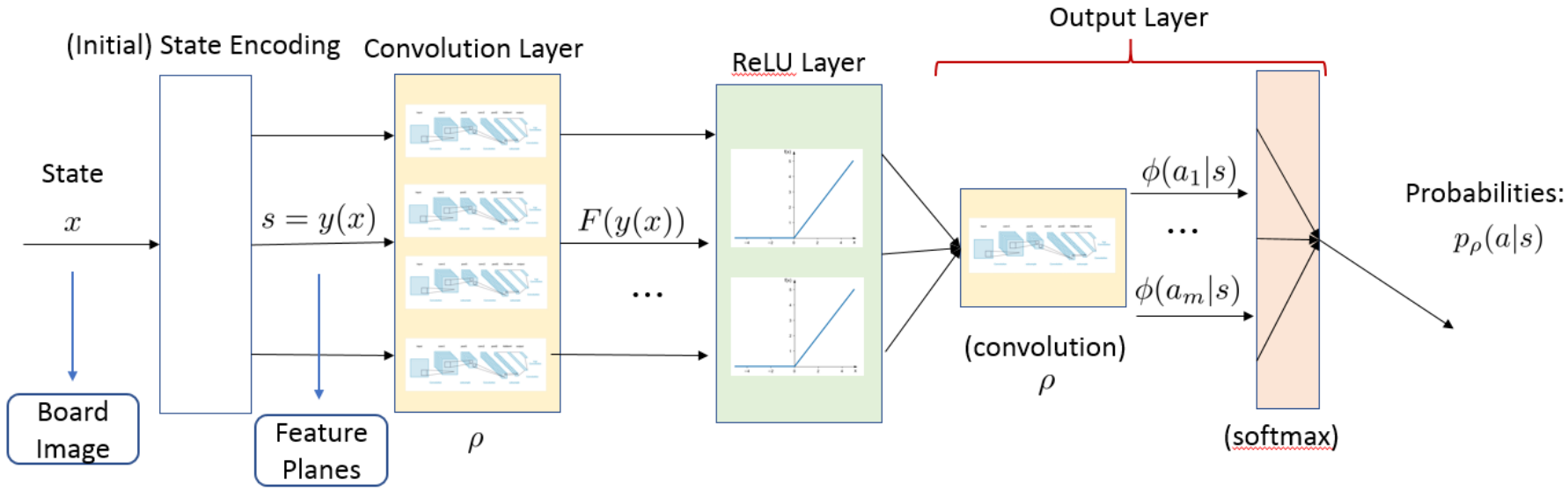
- The purpose of $\tilde{\mu}(s, \pi)$ will be revealed soon.

Policy Network: Policy Gradient

- Now we will train yet another policy! The policy $\tilde{\mu}(s, \rho)$ is called the *RL-policy.*

- Now this policy will be trained using the Policy Gradient.

- This policy share the same architecture as the SL-policy:



Policy Network: Policy Gradient

- Let σ^* be the final configuration of the SL-policy training.

- We initialize the RL-policy with the SL-policy: $\rho^{(0)} \coloneqq \sigma^*$.

- Now let's recall our abstract expression for the Policy Gradient (without any alterations):

$$\nabla_\theta \left(\mathbb{E}_{p(z|\theta)} \left[F(z) \right] \right) \approx \frac{1}{S} \sum_{s=1}^{S} \left(\sum_{k=0}^{M-1} \nabla_\theta \left(\ln(p(u_k | i_k^s, \theta)) \right) \right) \left(\sum_{k=0}^{M-1} \alpha^k g(i_k^s, u_k^s) + \alpha^M \hat{J}_M(i_m^s) \right)$$

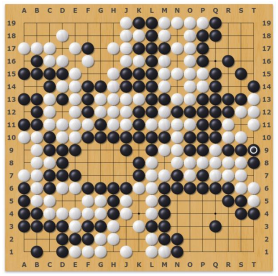
Maximum-Likelihood

Costs "weights"

“Costs” in AlphaGo

- Like we saw, in the timid-play/bold-play example, in AlphaGo, the “cost-to-go” is defined as the probability of winning the Go game given that the board position is x.

- For example we can collect pairs of board positions and the eventual outcome of the game associated with that position:



x^1

Feature encoding: → $y(x^1)$ Game outcome: →

“cost-to-go”

$J^1 = 1$ (a win)

x^2

Feature encoding: → $y(x^2)$ Game outcome: →

“cost-to-go”

$J^2 = 0$ (a loss)

Policy Network: Policy Gradient

- So, all stage costs are zero. And we do not use discount factor ($\alpha = 1$).
  - Note that, winning on turn 100 or on turn 50, does not really matter. All it matter is that we win.

- The key question is: How do we generate the samples?

- The answer is: **Self-play**. Let $\rho^{(0)}, \dots \rho^{(t-1)}$ be all the previous "versions" of the RL-policy.

  - **(1):** We let the current RL-policy $\rho^{(t)}$ play against a randomly selected past-version of itself. So against a random $\rho^{(t')}, t' \in \{0, \dots t-1\}$.

  - **(2):** By the word "play" we mean: we simulate the game until a result is achieved and we record it: +1 for a victory of $\rho^{(t)}$; -1 for a defeat.

Policy Network: Policy Gradient

- But we need to address the fact that this is a game:
  - In every turn, there are two moves, or “steps” in the game environment.
  - The “opponent” is not really nature, because they are using policies.

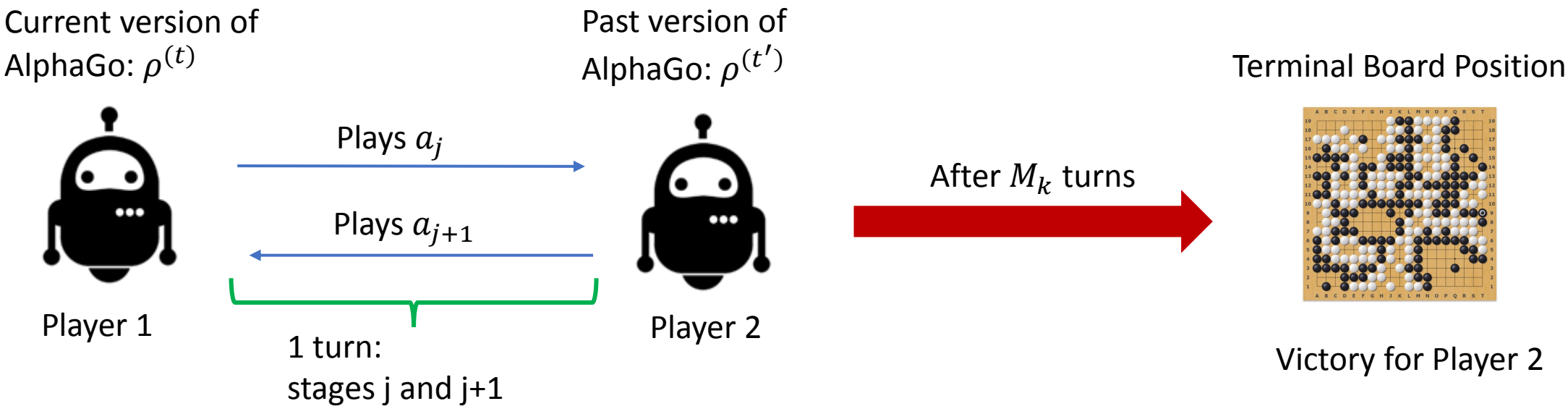
- The idea is to record the “costs” from the perspective of every player for every turn.

- In this interpretation, each pair of stages $(0,1), (2,3), (4,5)$ comprise turns 1, 2, and 3 and so forth.

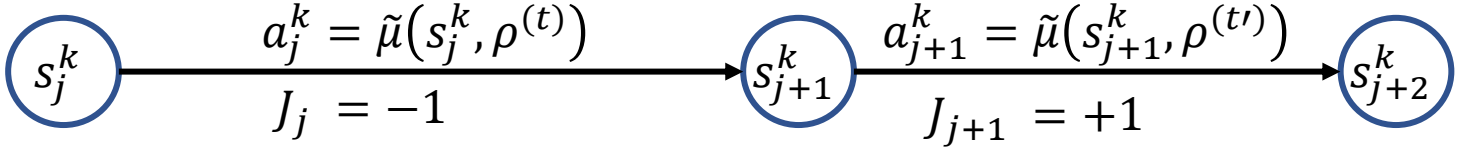
- This is better explained with a picture.

Policy Network: Policy Gradient

- Let's say we are playing a game k and we are on the beginning of Players 1 turn, on some stage j:



- Then for this turn we would record the following tuples (s, a, J) :



Policy Network: Policy Gradient

- The Policy Gradient, then becomes this:

$$\nabla_\rho\big(\mathbb{E}_{p(z|\rho)}\big[F(z)\big]\big) \approx \frac{1}{K}\sum_{k=1}^{K}\sum_{j=0}^{M_k-1}\nabla_\rho\big(\ln(p_\rho(a_j^k|s_j^k))\big)z_t^j$$

- Where K is the total number of simulated games. M_k is the total number of stages on each game k .

- z_t^j is reward variable computed from the perspective of the active player on stage j. so:

$$z_t^j = \begin{cases} +1, & \text{if the active player on stage j won the game} \\ -1, & \text{if the active player on stage j lost the game} \end{cases}$$

Policy Network: Policy Gradient

- A key question to ask is why are using “past-versions” instead of the current version.

- One may infer that using the current version as the opponent allows to keep *on-policy*, that is we preserve the probability structure of the game (since both players follow the same current policy).

- The decision to use past policies, thus making the Policy Gradient **off-policy**, is to enhance exploration and to remove the “moving-target” effect from self-play.
  - Recall our usage of *target networks* in the DQN, it is the same principle

- We have not analyze off-police methods yet, but it is good to keep that in mind.

Policy Network: Policy Gradient

- A few statistics of the Policy Gradient training:

- The batch K of games is of size 128 for 10000 iterations, using 50 GPU's to parallelize gameplay for one day.

- The opponent network is drawn from a "pool" of past versions, where every 500 the current configuration is added to the pool.

- After training the architecture ρ^* won 85% against a high-ranking GO-playing engine.

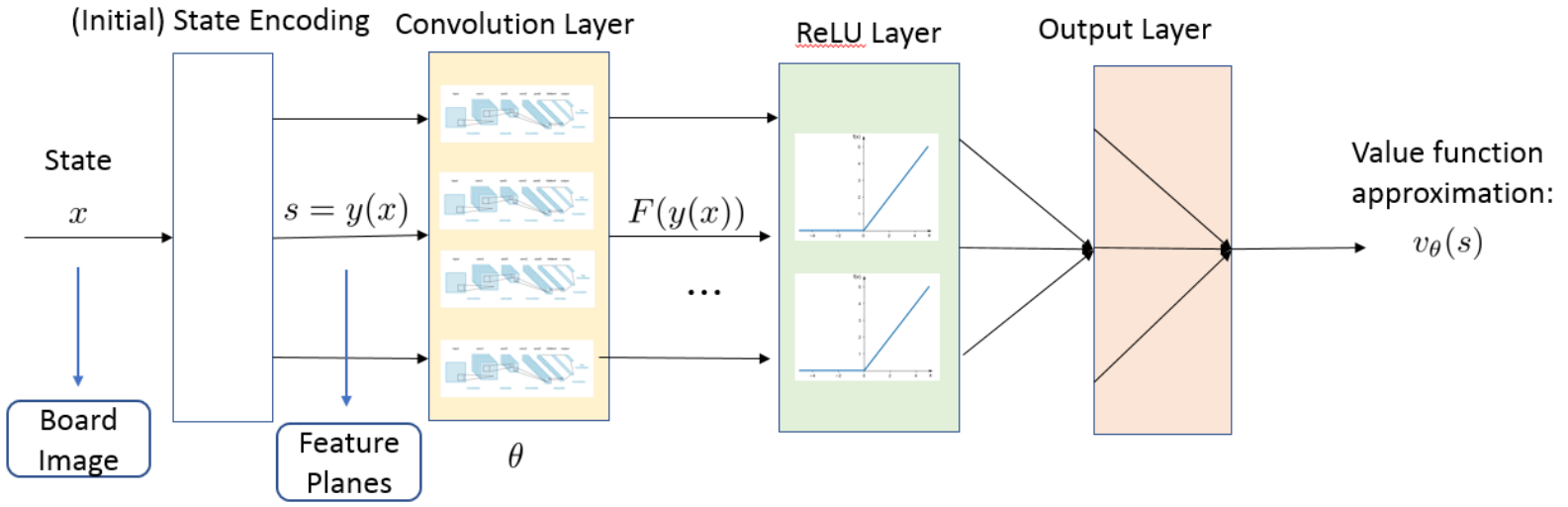
- In comparison, other previous supervised-learning based methods only achieve 11% win rate.

Value Network: Adding the Critic

- Now let's add the critic. The goal is to train another architecture, that approximates the Value Function (cost-to-go) of the game:

$$\tilde{J}_\rho(s,\theta) = v_\theta(s) \approx J^*(s)$$

- As before, the architecture θ is similar to SL-policy, but instead of mapping states to actions, it maps states to rewards (that is to probabilities of winning the game):



Value Network: Adding the Critic

- In this case, our labels are z_j^k for each state s_j^k, and the critic is reduced to the familiar Regression problem:

$$\theta^{(t)} = \arg\min_{\theta} \left\{ \sum_{k=1}^{K} \sum_{j=1}^{M_k} \left(v_\theta(s_j^k) - z_t^j \right)^2 \right\}$$

- There are few issues with the critic step:
  - We need the samples to be on-policy to correctly evaluate the probability of winning.
  - There is a very high degree of correlation between our samples.
  - Recall the goal of the critic is to approximate the cost-to-go <u>associated</u> with a policy:

$$\tilde{J}_\rho(s, \theta) = v_\theta(s) \approx J^*(s)$$

Value Network: Training the Critic

- To overcome these issues AlphaGo does the following:

- Generates a *complete* set of fresh samples from current self-play:
  - That is for the critic-step, the current best RL-policy ρ^* plays against itself (so against ρ^*)

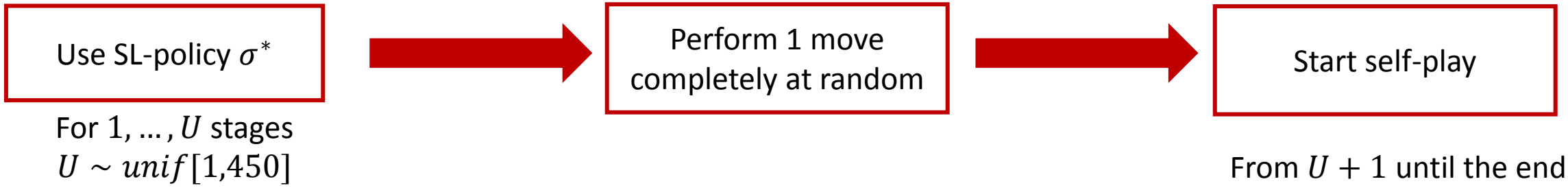
- The dataset contains 30million pairs of (s, z) where each pair comes from a different game.

- The initial board position is randomized and the game is played until conclusion. Only one pair (s, z) from the game is stored. Before the process is repeated.

- The regression problem used batches of 32 samples for 50million iterations, using 50 GPU's, for one week.

Value Network: Improving initial condition

- As we saw before, the Critic-step suffers heavily from the exploration-exploitation trade-off.
- The idea is to “diversify” the board states that can be achieved.
- To that end, the initial board position is designed in three steps:



- The intuition is that we use a “more-random” policy the imitates the expert plays before transitioning into self-play mode.

Combining Actor and Critic

- Note that, in general, it is possible to use the trained critic, to add the state-dependent base-line to the policy gradient:

$$\nabla_\rho \big(\mathbb{E}_{p(z|\rho)} \big[F(z) \big] \big) \approx \frac{1}{K} \sum_{k=1}^{K} \sum_{j=0}^{M_k - 1} \nabla_\rho \big(\ln(p_\rho(a_j^k | s_j^k)) \big) (z_t^j - v_{\theta^{(t)}}(s_j^k))$$

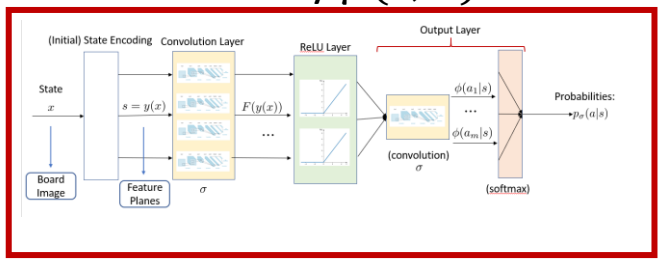
- And we could train both RL-policy ρ and the critic θ in the fashion we studied before, effectly implementing the Approximate PI algorithm for the game of Go.

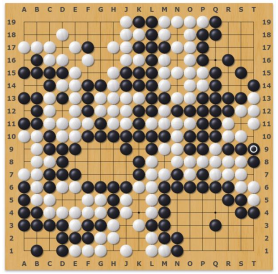
- And this actually achieves a very high level of play already, this totally on self-play.

- What happens when it plays against real-life opponents?

General Overview of the Training Process

SL-Policy $\tilde{\mu}(s, \sigma)$

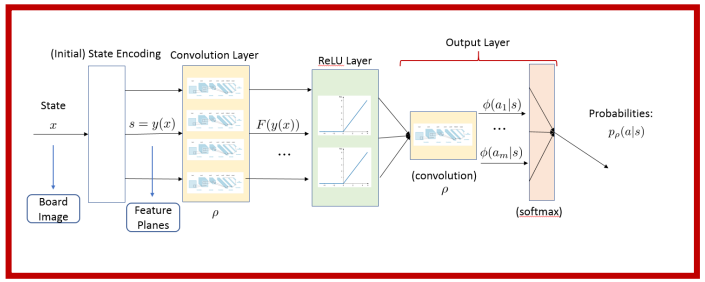




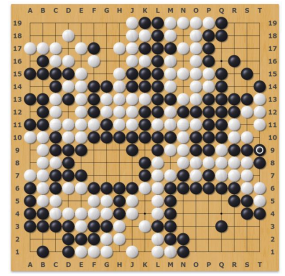
Supervised-Learning

Policy-Gradient

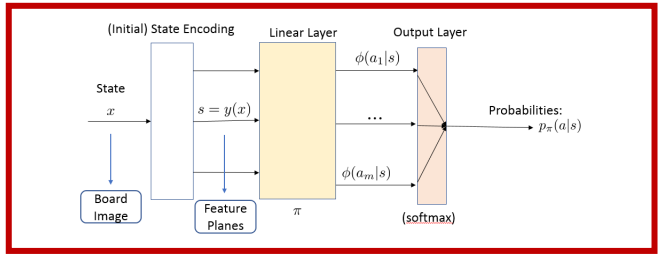
RL-Policy $\tilde{\mu}(s, \rho)$



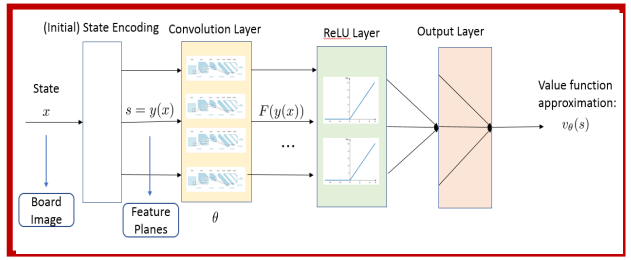
Self-play



Regression



Rollout-Policy $\tilde{\mu}(s, \pi)$

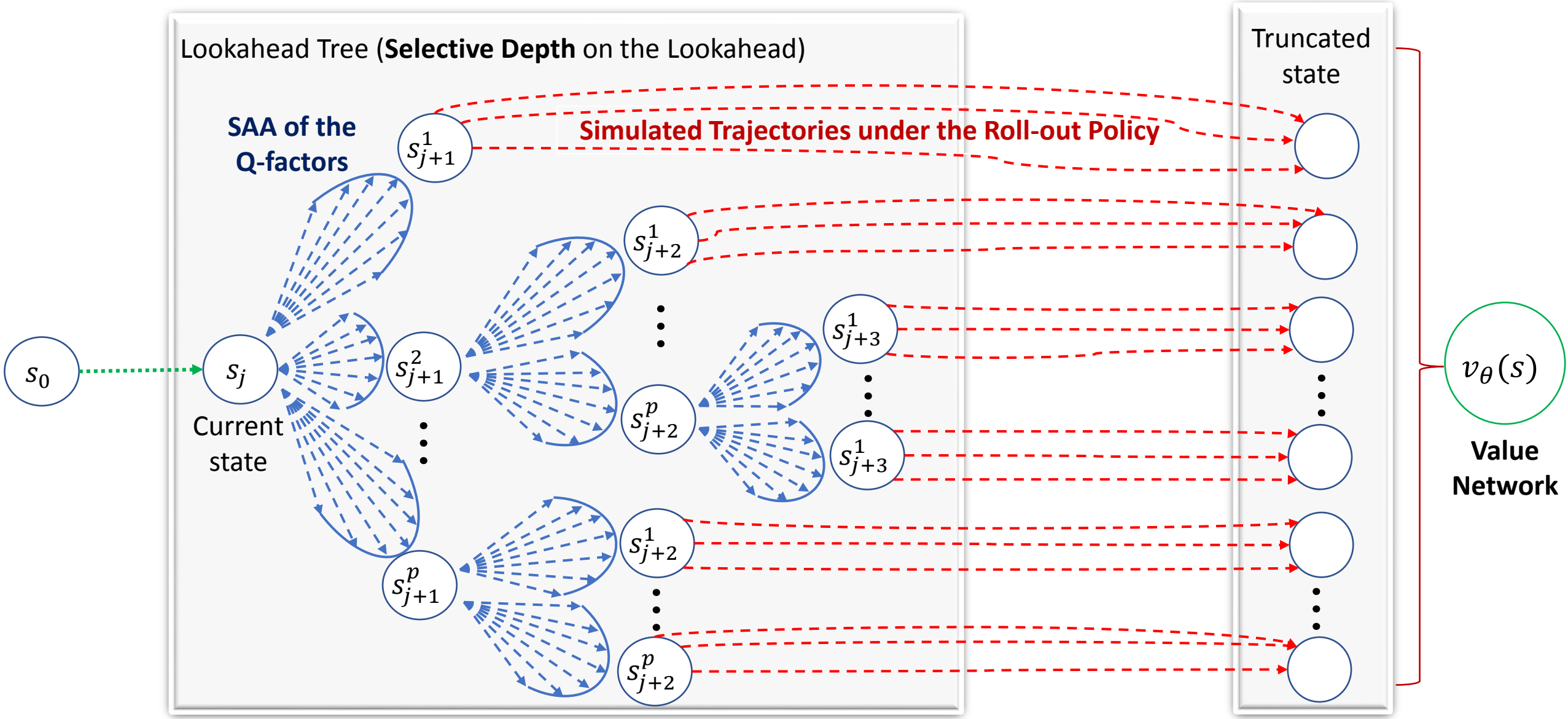


Value Function approximation: $\tilde{J}(s, \theta)$

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.
- Next time, we will see how to incorporate the trained architectures into the Monte-Carlo Tree Search framework

MCTS in AlphaGo



Value Network: Improving initial condition

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

$$v(s_l) = (1-\lambda)v_\theta(s_l) + \lambda z_l$$

$$N(s,a) = \sum_{i=1}^{n} \mathbf{1}(s,a,i)$$

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