April 03, 2024

Model based RL
Neural MCTS

$\dot{x} = Ax + Bu$

$J = \int_0^\infty (x^T Q x + u^T R u)\, dt$

State, action

$x_{k+1} = A^* x_k + B^* u_k$ → transition

$J = \sum_0^\infty [x_k^T Q x_k + u_k^T R u_k]$

$cost(t_k)$

$G_t = \sum_{\tau=t}^{\infty} r_t \gamma^{\tau - t}$ $\quad s_t, a_t$

$|r_t| \le R_0$

$\max |G_t| = \frac{R_0}{1-\gamma}$

$P(S_{t+1} \mid S_t, a_t)$

transition prob.

Markov decision process (MDP) → $S_{t-1}, a_{t-1}, S_{t-2}, \dots S_0, a_0$ independent

(S, a, r, P, γ) → defines MDP

Inspiration from model predictive control (MPC)

control problem: $\min_{u_i, i=1,\dots\infty} J = \sum_{i=1}^{\infty} [w_x (r_i - x_i)^2 + w_u u_i^2]$

reference, state, input

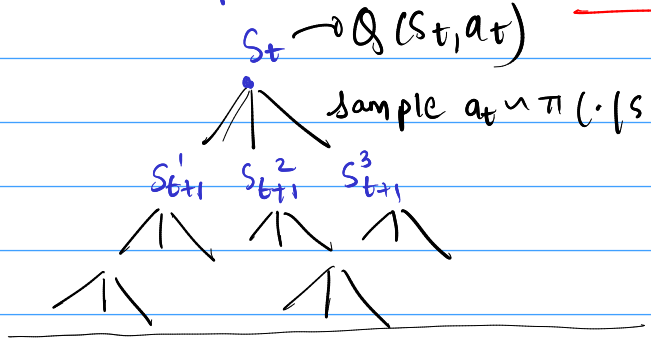
→ Too hard to solve

→ Note that dynamics $f(x, \dot{x}, \ddot{x}, \dots, u) = 0$ and constraints are assumed known

→ Solution: Solve for $J_n = \sum_1^n [\ ]$, where n is moderately large, compute $u_1, \dots, u_n$, and implement only a subset. Then solve the problem again ....

Known to be extremely competitive if → solvable, sufficient actuator power

How does this work in deep RL? → $P(s_{t+1} \mid s_t, a_t)$



→ Replace by $Q(s,a)$ at some future time step ⇝ Neural MCTS!

→ Need some idea about $P(s' \mid s, a)$ ⇝ Model!

→ Need to decide breadth, depth ... dynamically?

→ Note: Applicable to both deterministic & stochastic problems

How to think about the model?

① Episodic memory → deterministic only

② Bayesian estimation → sample efficient but compute inefficient

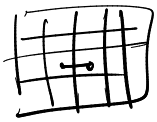
③ Function approximation → majority of MBRL

④ Simulation → if problem allows it

In model-based reinforcement learning, a model of the dynamics is used to make predictions, which is used for action selection. Let $\hat{f}_\theta(\mathbf{s}_t, \mathbf{a}_t)$ denote a learned discrete-time dynamics function, parameterized by θ, that takes the current state $\mathbf{s}_t$ and action $\mathbf{a}_t$ and outputs an estimate of the next state at time $t + \Delta t$. We can then choose actions by solving the following optimization problem:

$$(\mathbf{a}_t, \ldots, \mathbf{a}_{t+H-1}) = \arg \max_{\mathbf{a}_t, \ldots, \mathbf{a}_{t+H-1}} \sum_{t'=t}^{t+H-1} \gamma^{t'-t} r(\mathbf{s}_{t'}, \mathbf{a}_{t'}) \quad (1)$$

connect to MPC



Model-based deep RL [Nagabandi et. al.]

① Run random episodes for experience collection $\underbrace{p(s_{t+1}|s_t, a_t)}_{\text{train?}}$

② Train on dynamics: $\hat{f}_\theta(s_t, a_t) \approx s_{t+1} - s_t$

learning Δs helpful for { slow dynamics, small time steps }

③ Roll out trajectories with random actions, using $\hat{f}_\theta$

④ Pick the best trajectory and implement the first action

good thing → completely transferable to any task

bad thing → needs known reward function
→ not enough exploration for combinatorial problems

Improvements

$\hat{f}_\theta(s_t, a_t) \to \hat{s}_{t+1}$

① Use DAGGER for imitation learning
→ MPC for collecting experience
→ Train policy $\pi_\phi(a|s)$ using behaviour cloning

policy
→ rollout using π_ϕ
→ At every state s, get MPC action
→ compute action selection loss

MbMf

② Use $\pi_\phi(a|s)$ as initial policy for TRPO

Data preprocessing: We slice the trajectories $\{\tau\}$ into training data inputs $(\mathbf{s}_t, \mathbf{a}_t)$ and corresponding output labels $\mathbf{s}_{t+1} - \mathbf{s}_t$. We then subtract the mean of the data and divide by the standard deviation of the data to ensure the loss function weights the different parts of the state (e.g., positions and velocities) equally. We also add zero mean Gaussian noise to the training data (inputs and outputs) to increase model robustness. The training data is then stored in the dataset $\mathcal{D}$.

C. Model-Based Control

In order to use the learned model $\hat{f}_\theta(\mathbf{s}_t, \mathbf{a}_t)$, together with a reward function $r(\mathbf{s}_t, \mathbf{a}_t)$ that encodes some task, we formulate a model-based controller that is both computationally tractable and robust to inaccuracies in the learned dynamics model. Expanding on the discussion in Sec. III, we first optimize the sequence of actions $\mathbf{A}_t^{(H)} = (\mathbf{a}_t, \cdots, \mathbf{a}_{t+H-1})$ over a finite horizon H, using the learned dynamics model to predict future states:

reliable?

$$\mathbf{A}_t^{(H)} = \arg\max_{\mathbf{A}_t^{(H)}} \sum_{t'=t}^{t+H-1} r(\hat{\mathbf{s}}_{t'}, \mathbf{a}_{t'}) \quad :$$

$$\hat{\mathbf{s}}_t = \mathbf{s}_t, \hat{\mathbf{s}}_{t'+1} = \hat{\mathbf{s}}_{t'} + \hat{f}_\theta(\hat{\mathbf{s}}_{t'}, \mathbf{a}_{t'}). \qquad (4)$$

Newly generated data points can be added to buffer for training

A. Initializing the Model-Free Learner

We first gather example trajectories with the MPC controller detailed in Sec. IV-C, which uses the learned dynamics function $\hat{f}_\theta$ that was trained using our model-based reinforcement learning algorithm (Alg. 1). We collect the trajectories into a dataset $\mathcal{D}^*$, and we then train a neural network policy $\pi_\phi(\mathbf{a}|\mathbf{s})$ to match these "expert" trajectories in $\mathcal{D}^*$. We parameterize π_ϕ as a conditionally Gaussian policy $\pi_\phi(\mathbf{a}|\mathbf{s}) \sim \mathcal{N}(\mu_\phi(\mathbf{s}), \Sigma_{\pi_\phi})$, in which the mean is parameterized by a neural network $\mu_\phi(\mathbf{s})$, and the covariance Σ_{π_ϕ} is a fixed matrix. This policy's parameters are trained using the behavioral cloning objective

$$\min_\phi \frac{1}{2} \sum_{(\mathbf{s}_t, \mathbf{a}_t) \in \mathcal{D}^*} ||\mathbf{a}_t - \mu_\phi(\mathbf{s}_t)||_2^2, \tag{5}$$

which we optimize using stochastic gradient descent. To achieve desired performance and address the data distribution problem, we applied DAGGER [40]: This consisted of iterations of training the policy, performing on-policy rollouts, querying the "expert" MPC controller for "true" action labels for those visited states, and then retraining the policy.

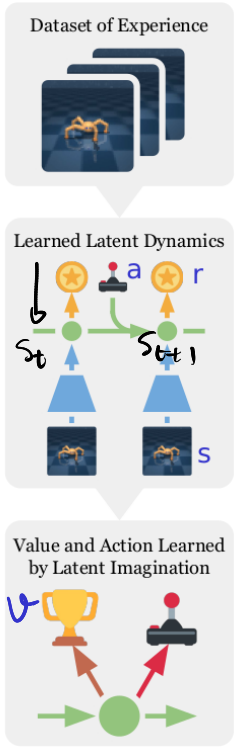
DAGGER:
1. Collect
2. Train
3. Rollout
4. Correct

B. Model-Free Reinforcement Learning

After initialization, we can use the policy π_ϕ, which was trained on data generated by our learned model-based controller, as an initial policy for a model-free reinforcement learning algorithm. Specifically, we use trust region policy optimization (TRPO) [3]; such policy gradient algorithms are a good choice for model-free fine-tuning since they do not require any critic or value function for initialization [41], though our method could also be combined with other model-free RL algorithms.

Dreamer [Hafner et. al.]

① Learn a world model



$\hat{f}_\theta(s_t, a_t) \rightarrow s_{t+1}$

- Dynamics learning ← from memory
- Behaviour learning ← in simulation "imagination"
- Environment interaction ← on real system, using pretrained policy

Difference from MBRL: ① RL used from the start

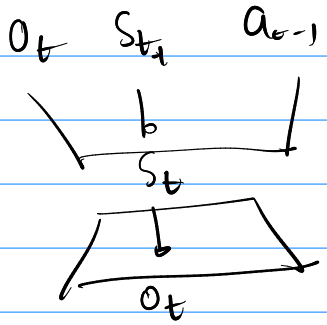
② Experience collected by RL policy

Important point

- Imagination involves full pixel reconstruction
- Need consistent reconstruction H steps into future

Real
Imagined
Imagined

Representation model:	$p(s_t \mid s_{t-1}, a_{t-1}, o_t)$	latest info (image)
Transition model:	$q(s_t \mid s_{t-1}, a_{t-1})$	prediction
Reward model:	$q(r_t \mid s_t)$.	



1. Collect
2. Train dynamics
3. Train RL policy without env interactions
4. Back to 1 with new policy

Difference from previous MBRL is the policy for experience collection (not pure random now)

Algorithm 1: Dreamer

Initialize dataset $\mathcal{D}$ with S random seed episodes.
Initialize neural network parameters θ, ϕ, ψ randomly.
while *not converged* **do**
  for *update step* $c = 1..C$ **do**
    `// Dynamics learning`
    Draw B data sequences $\{(a_t, o_t, r_t)\}_{t=k}^{k+L} \sim \mathcal{D}$.
    Compute model states $s_t \sim p_\theta(s_t \mid s_{t-1}, a_{t-1}, o_t)$.
    Update θ using representation learning.
    `// Behavior learning`
    Imagine trajectories $\{(s_\tau, a_\tau)\}_{\tau=t}^{t+H}$ from each s_t.
    Predict rewards $\mathrm{E}\big(q_\theta(r_\tau \mid s_\tau)\big)$ and values $v_\psi(s_\tau)$.
    Compute value estimates $\mathrm{V}_\lambda(s_\tau)$ via Equation 6.
    Update $\phi \leftarrow \phi + \alpha \nabla_\phi \sum_{\tau=t}^{t+H} \mathrm{V}_\lambda(s_\tau)$.
    Update $\psi \leftarrow \psi - \alpha \nabla_\psi \sum_{\tau=t}^{t+H} \frac{1}{2} \big\| v_\psi(s_\tau) - \mathrm{V}_\lambda(s_\tau) \big\|^2$.
  `// Environment interaction`
  $o_1 \leftarrow$ `env.reset()`
  for *time step* $t = 1..T$ **do**
    Compute $s_t \sim p_\theta(s_t \mid s_{t-1}, a_{t-1}, o_t)$ from history.
    Compute $a_t \sim q_\phi(a_t \mid s_t)$ with the action model.
    Add exploration noise to action.
    $r_t, o_{t+1} \leftarrow$ `env.step(`a_t`)`.
  Add experience to dataset $\mathcal{D} \leftarrow \mathcal{D} \cup \{(o_t, a_t, r_t)_{t=1}^{T}\}$.

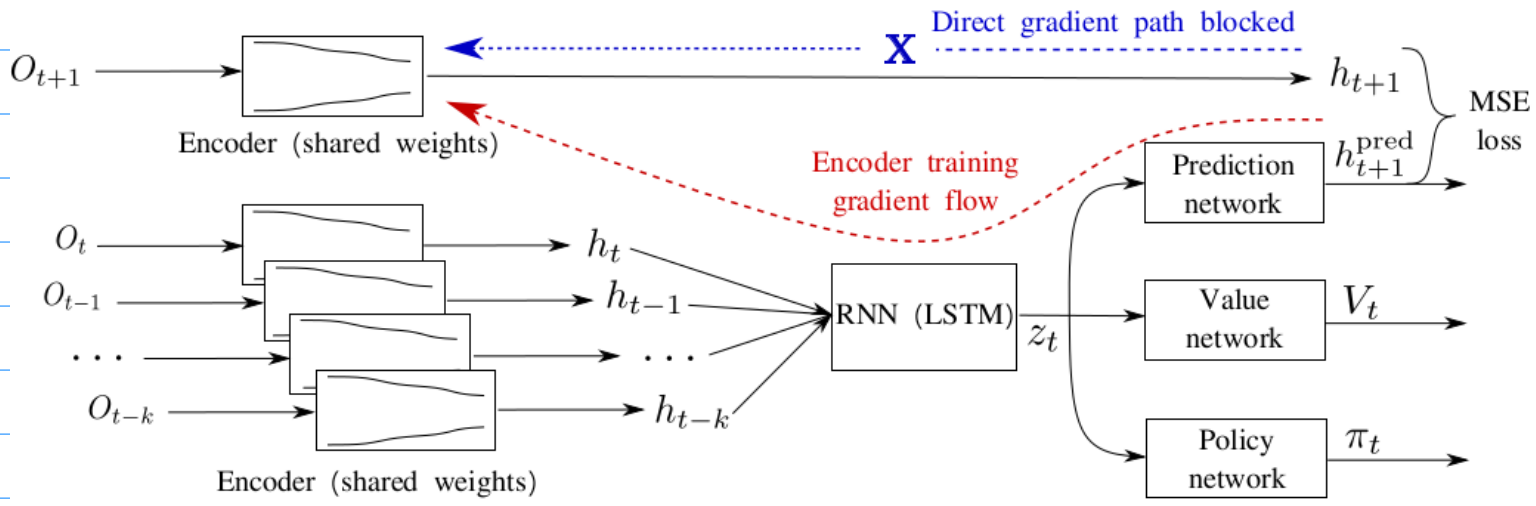
Model components

Representation	$p_\theta(s_t \mid s_{t-1}, a_{t-1}, o_t)$
Transition	$q_\theta(s_t \mid s_{t-1}, a_{t-1})$
Reward	$q_\theta(r_t \mid s_t)$
Action	$q_\phi(a_t \mid s_t)$
Value	$v_\psi(s_t)$

Hyper parameters

Seed episodes	S
Collect interval	C
Batch size	B
Sequence length	L
Imagination horizon	H
Learning rate	α

FoLaR: Foggy Latent Representations for Reinforcement Learning with Partial Observability



Alpha Go (and Alpha Go Zero)

→ Our first proper look at combinatorial problems

→ Importance of original paper: popularising
- self play
- neural MCTS

→ In combinatorial problems

breadth (b) + depth (d) ⇒ b^d complexity

- breadth (b): possible decisions at time t — use $\pi(a|s)$
- depth (d): time horizon to get results — use $v(s)$

Four stages of Alpha Go

(1) Supervised learning, policy p_σ from human games [57% accuracy] → essentially, predict human moves

(2) Fast (shallow) policy p_π for rapid sampling in rollouts [24% accuracy]

(3) Self-play trained policy p_ρ → starts with $p_\rho = p_\sigma$ for initialisation

(4) winner prediction value function v_θ

same thing except NN architecture

against various checkpoints

The game of Go has long been viewed as the most challenging of classic games for artificial intelligence owing to its enormous search space and the difficulty of evaluating board positions and moves. Here we introduce a new approach to computer Go that uses 'value networks' to evaluate board positions and 'policy networks' to select moves. These deep neural networks are trained by a novel combination of supervised learning from human expert games, and reinforcement learning from games of self-play. Without any lookahead search, the neural networks play Go at the level of state-of-the-art Monte Carlo tree search programs that simulate thousands of random games of self-play. We also introduce a new search algorithm that combines Monte Carlo simulation with value and policy networks. Using this search algorithm, our program AlphaGo achieved a 99.8% winning rate against other Go programs, and defeated the human European Go champion by 5 games to 0. This is the first time that a computer program has defeated a human professional player in the full-sized game of Go, a feat previously thought to be at least a decade away.

We train the neural networks using a pipeline consisting of several stages of machine learning (Fig. 1). We begin by training a supervised learning (SL) policy network p_σ directly from expert human moves. This provides fast, efficient learning updates with immediate feedback and high-quality gradients. Similar to prior work[13,15], we also train a fast policy p_π that can rapidly sample actions during rollouts. Next, we train a reinforcement learning (RL) policy network p_ρ that improves the SL policy network by optimizing the final outcome of games of self-play. This adjusts the policy towards the correct goal of winning games, rather than maximizing predictive accuracy. Finally, we train a value network v_θ that predicts the winner of games played by the RL policy network against itself. Our program AlphaGo efficiently combines the policy and value networks with MCTS.

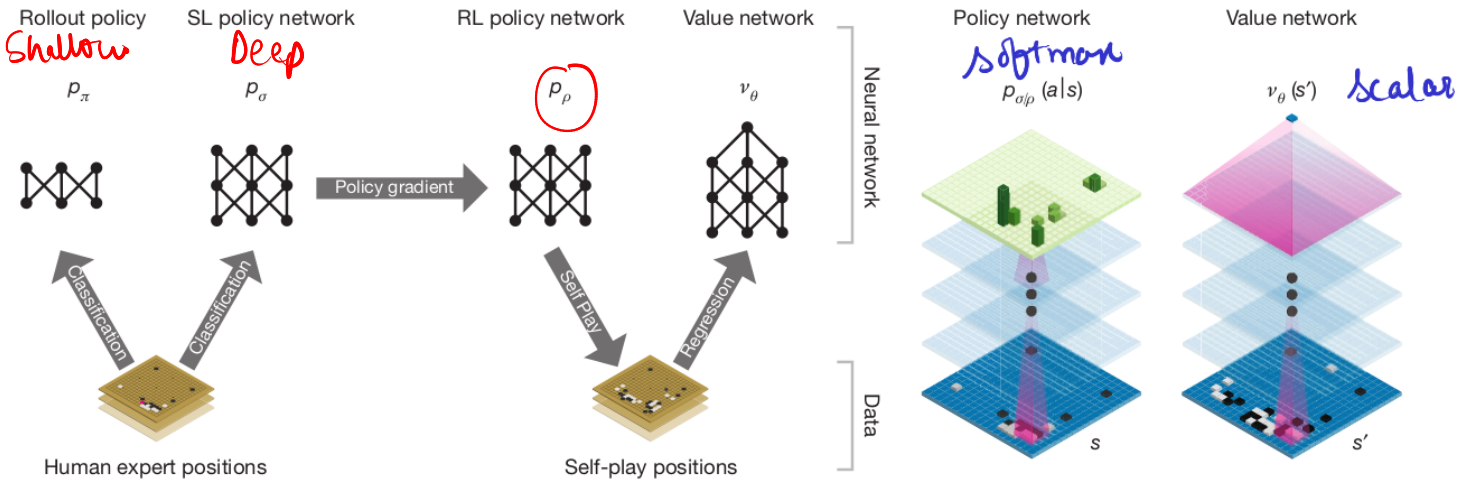


Figure 1 | Neural network training pipeline and architecture. a, A fast rollout policy p_π and supervised learning (SL) policy network p_σ are trained to predict human expert moves in a data set of positions. A reinforcement learning (RL) policy network p_ρ is initialized to the SL policy network, and is then improved by policy gradient learning to maximize the outcome (that is, winning more games) against previous versions of the policy network. A new data set is generated by playing games of self-play with the RL policy network. Finally, a value network v_θ is trained by regression to predict the expected outcome (that is, whether the current player wins) in positions from the self-play data set. **b**, Schematic representation of the neural network architecture used in AlphaGo. The policy network takes a representation of the board position s as its input, passes it through many convolutional layers with parameters σ (SL policy network) or ρ (RL policy network), and outputs a probability distribution $p_\sigma(a|s)$ or $p_\rho(a|s)$ over legal moves a, represented by a probability map over the board. The value network similarly uses many convolutional layers with parameters θ, but outputs a scalar value $v_\theta(s')$ that predicts the expected outcome in position s'.

probability of action / it visitcount

average win % of all trajectories passing through

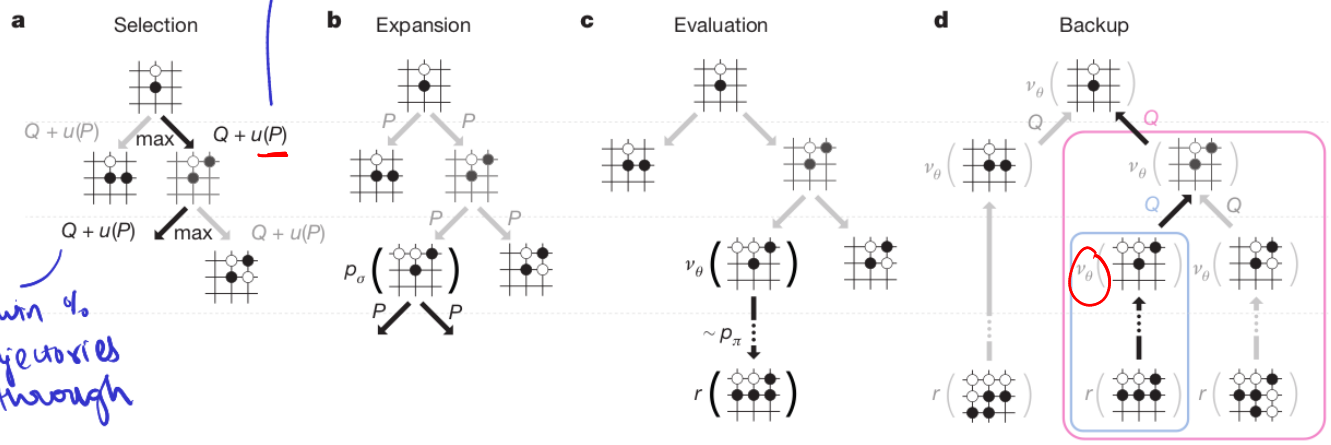


Figure 3 | Monte Carlo tree search in AlphaGo. a, Each simulation traverses the tree by selecting the edge with maximum action value Q, plus a bonus $u(P)$ that depends on a stored prior probability P for that edge. **b**, The leaf node may be expanded; the new node is processed once by the policy network p_σ and the output probabilities are stored as prior probabilities P for each action. **c**, At the end of a simulation, the leaf node is evaluated in two ways: using the value network v_θ; and by running a rollout to the end of the game with the fast rollout policy p_π, then computing the winner with function r. **d**, Action values Q are updated to track the mean value of all evaluations $r(\cdot)$ and $v_\theta(\cdot)$ in the subtree below that action.

During the match against Fan Hui, AlphaGo evaluated thousands of times fewer positions than Deep Blue did in its chess match against Kasparov[4]; compensating by selecting those positions more intelligently, using the policy network, and evaluating them more precisely, using the value network—an approach that is perhaps closer to how humans play. Furthermore, while Deep Blue relied on a handcrafted evaluation function, the neural networks of AlphaGo are trained directly from gameplay purely through general-purpose supervised and reinforcement learning methods.

Go is exemplary in many ways of the difficulties faced by artificial intelligence[33,34]: a challenging decision-making task, an intractable search space, and an optimal solution so complex it appears infeasible to directly approximate using a policy or value function. The previous major breakthrough in computer Go, the introduction of MCTS, led to corresponding advances in many other domains; for example, general game-playing, classical planning, partially observed planning, scheduling, and constraint satisfaction[35,36]. By combining tree search with policy and value networks, AlphaGo has finally reached a professional level in Go, providing hope that human-level performance can now be achieved in other seemingly intractable artificial intelligence domains.

Jan 2016

Alpha Go Zero: October 2017

from clean slate

A long-standing goal of artificial intelligence is an algorithm that learns, *tabula rasa*, superhuman proficiency in challenging domains. Recently, *AlphaGo* became the first program to defeat a world champion in the game of Go. The tree search in *AlphaGo* evaluated positions and selected moves using deep neural networks. These neural networks were trained by supervised learning from human expert moves, and by reinforcement learning from self-play. Here, we introduce an algorithm based solely on reinforcement learning, without human data, guidance, or domain knowledge beyond game rules. *AlphaGo* becomes its own teacher: a neural network is trained to predict *AlphaGo*'s own move selections and also the winner of *AlphaGo*'s games. This neural network improves the strength of tree search, resulting in higher quality move selection and stronger self-play in the next iteration. Starting *tabula rasa*, our new program *AlphaGo Zero* achieved superhuman performance, winning 100-0 against the previously published, champion-defeating *AlphaGo*.

self play only, and no hand-crafted features

Our program, *AlphaGo Zero*, differs from *AlphaGo Fan* and *AlphaGo Lee* [12] in several important aspects. First and foremost, it is trained solely by self-play reinforcement learning, starting from random play, without any supervision or use of human data. Second, it only uses the black and white stones from the board as input features. Third, it uses a single neural network, rather than separate policy and value networks. Finally, it uses a simpler tree search that relies upon this single neural network to evaluate positions and sample moves, without performing any Monte-Carlo rollouts. To achieve these results, we introduce a new reinforcement learning algorithm that incorporates lookahead search *inside* the training loop, resulting in rapid improvement and precise and stable learning. Further technical differences in the search algorithm, training procedure and network architecture are described in Methods.

→ Note this - there is a tree search but not as extensive as MCTS

AlphaGo Zero: $s_t, s_{t-1}, \ldots \rightarrow f_\theta \rightarrow \bar{p}(a|s)$, $v(s)$

$v(s)$: probability of winning from s

$\pi(a|s)$: proportional to $N^{1/\tau}$, where N = freq and τ = temperature

$\pi(a|s)$ derived from $\bar{p}(a|s)$ using MCTS guided by f_θ

Eventually, $\bar{p}$ is trained to approximate π,
and $v(s)$ is trained to predict self-play winner z

Important note: In MCTS, nodes are traversed using P(prior), Q(values), and N(visitation frequencies) without calling neural net until a leaf node is encountered

clearly only possible for discrete states

Loss function: $l = (z-v)^2 + \pi^T \log \bar{p} + c\|\theta\|^2$

$(z-v)^2$: value MSE

$\pi^T \log \bar{p}$: cross entropy

$c\|\theta\|^2$: L2 regulariser

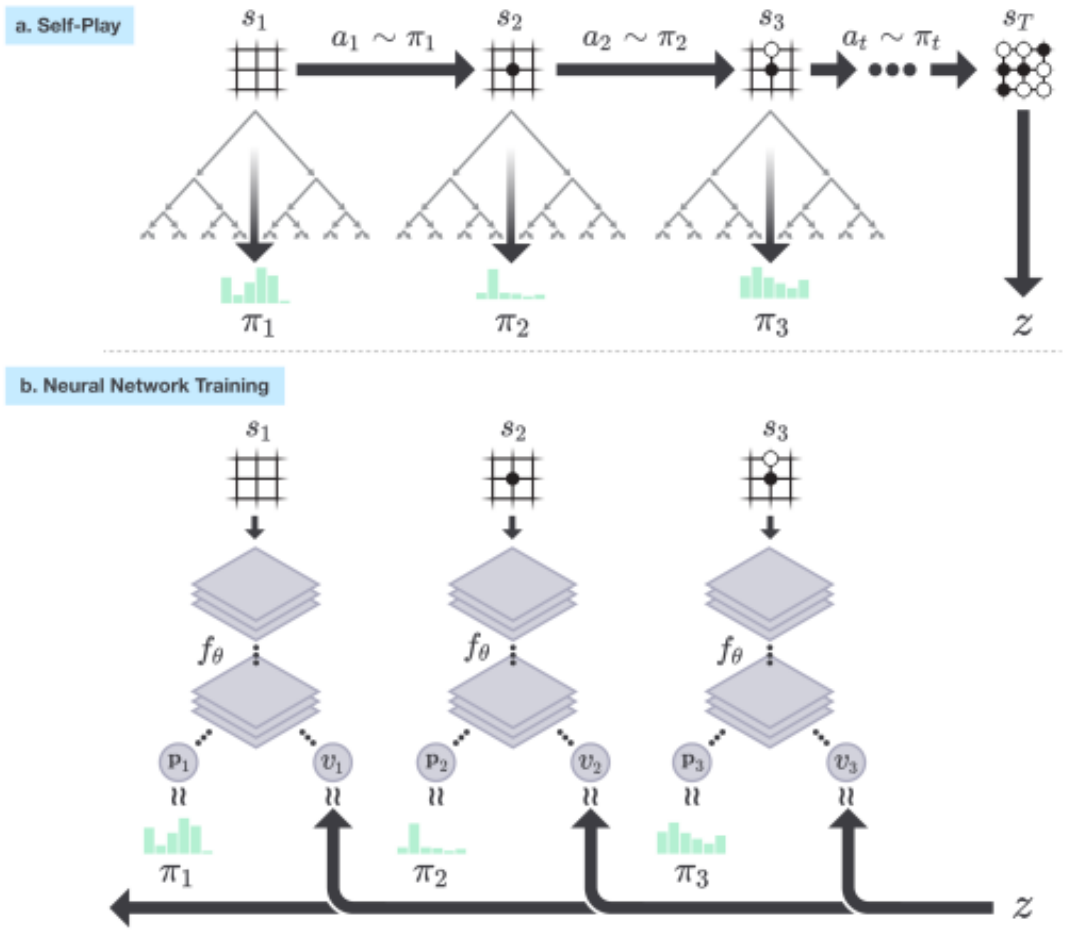


Figure 1: **Self-play reinforcement learning in *AlphaGo Zero*. a** The program plays a game $s_1, ..., s_T$ against itself. In each position s_t, a Monte-Carlo tree search (MCTS) α_θ is executed (see Figure 2) using the latest neural network f_θ. Moves are selected according to the search probabilities computed by the MCTS, $a_t \sim \boldsymbol{\pi}_t$. The terminal position s_T is scored according to the rules of the game to compute the game winner z. **b** Neural network training in *AlphaGo Zero*. The neural network takes the raw board position s_t as its input, passes it through many convolutional layers with parameters θ, and outputs both a vector $\mathbf{p_t}$, representing a probability distribution over moves, and a scalar value v_t, representing the probability of the current player winning in position s_t. The neural network parameters θ are updated so as to maximise the similarity of the policy vector $\mathbf{p_t}$ to the search probabilities $\boldsymbol{\pi_t}$, and to minimise the error between the predicted winner v_t and the game winner z (see Equation 1). The new parameters are used in the next iteration of self-play **a**.