Reinforcement Learning

Deep Q-Networks, Policy Gradients, and Actor-Critic Algorithms

N. Rich Nguyen, PhD

SYS 6016

Review: Q-Learning Value Iteration

$$\mathbf{Q}_{t+1}(s_t, a_t) = \mathbf{Q}_t(s_t, a_t) + \alpha\left(\mathbf{R}_{t+1} + \gamma \max_{a} \mathbf{Q}_t(s_{t+1}, a) - \mathbf{Q}_t(s_t, a_t)\right)$$

new Q-value

old Q-value

learning rate

reward

discount factor

future Q-value upon selecting the best action

	A1	A2	A3	A4
S1	+1	+2	-1	0
S2	+2	0	+1	-2
S3	-1	+1	0	-2
S4	-2	0	+1	+1

```
initialize Q[num_states,num_actions] arbitrarily
observe initial state s
repeat
      select and carry out an action a
      observe reward r and new state s'
      Q[s,a] = Q[s,a] + α(r + γ max_a' Q[s',a'] - Q[s,a])
      s = s'
until terminated
```

Q-Learning: Representation Matters

Unfortunately, value iteration is **impractical**

- Limited states/actions
- Cannot generalize to unobserved states

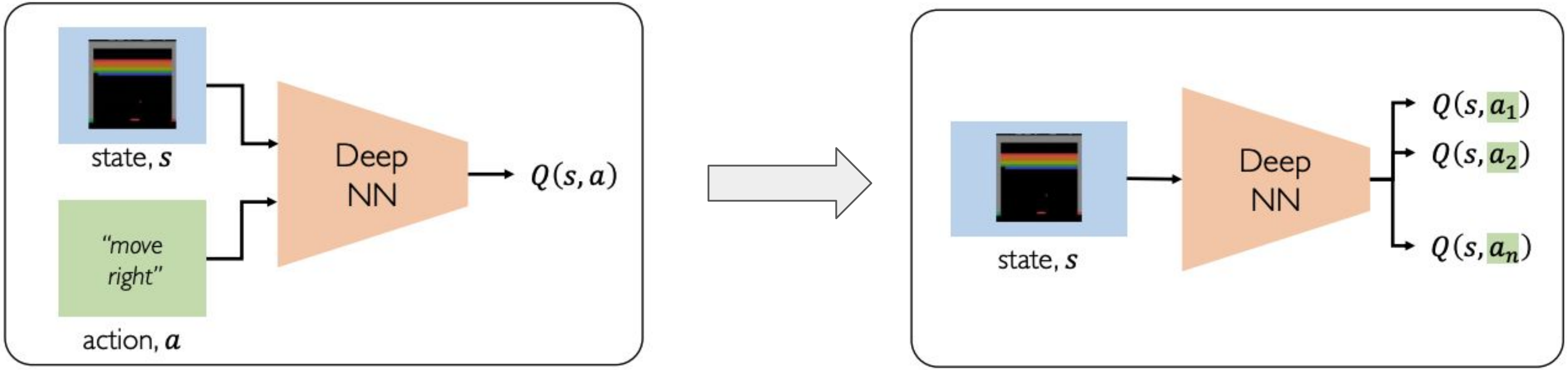
Think about the **Breakout** Arcade game

State: screen pixels

- Image size: 84 x 84 (resized)
- Consecutive 4 images
- Grayscale with 256 gray levels
- → **$256^{84x84x4}$** rows in the Q-table! ($256^{28,224} = 10^{69,970} >> 10^{82}$ atoms in the universe)

Deep RL = RL + Neural Networks

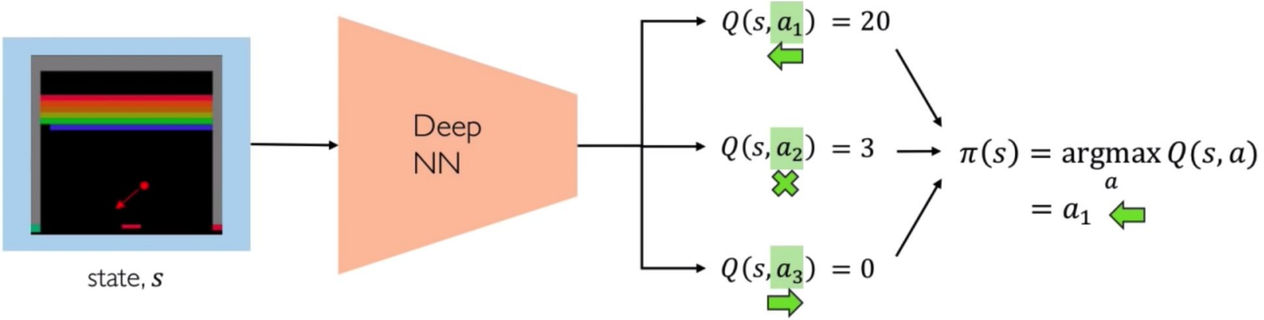
Use a deep neural network to approximate Q-function → Deep Q-Network (DQN):



Deep Q-Networks

Deep Q-Networks (DQN)

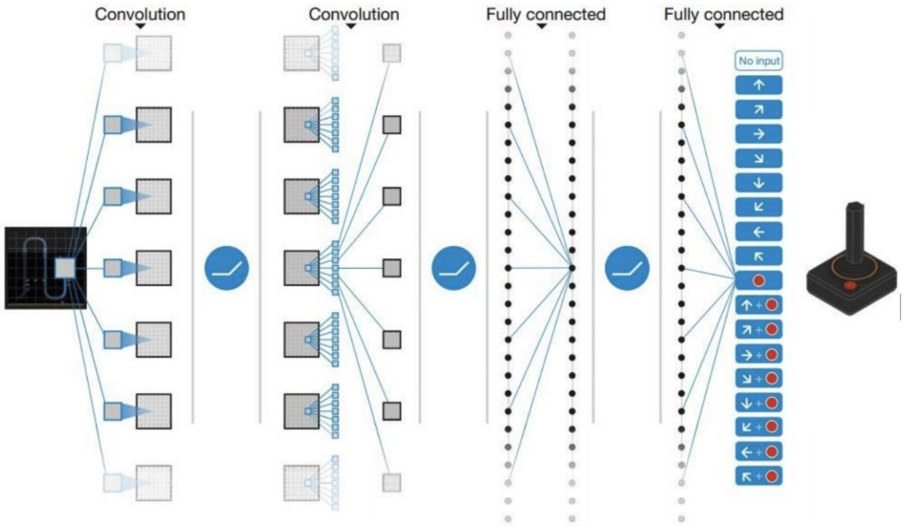
DQN: Approximate Q-function and use it to infer the optimal policy, $\pi(s)$



Loss function: $\mathbf{L} = \mathbb{E}\Big[\big|\big| \underbrace{\big(r + \gamma \max_{a'} \mathbf{Q}(s', a')\big)}_{\textbf{target}} - \underbrace{\mathbf{Q}(s, a)}_{\textbf{predicted}} \big|\big|^2 \Big]$

- **DQN:** same network for both Q functions
- **Double DQN:** separate network for each Q function to help reduce bias introduced by the inaccuracies of Q network at the beginning of training

A Deep Q-Network Architecture



Layer	Input	Filter size	Stride	Num filters	Activation	Output
conv1	84x84x4	8x8	4	32	ReLU	20x20x32
conv2	20x20x32	4x4	2	64	ReLU	9x9x64
conv3	9x9x64	3x3	1	64	ReLU	7x7x64
fc4	7x7x64			512	ReLU	512
fc5	512			18	Linear	18

Mnih et al. “Playing atari with deep reinforcement learning.” 2013.

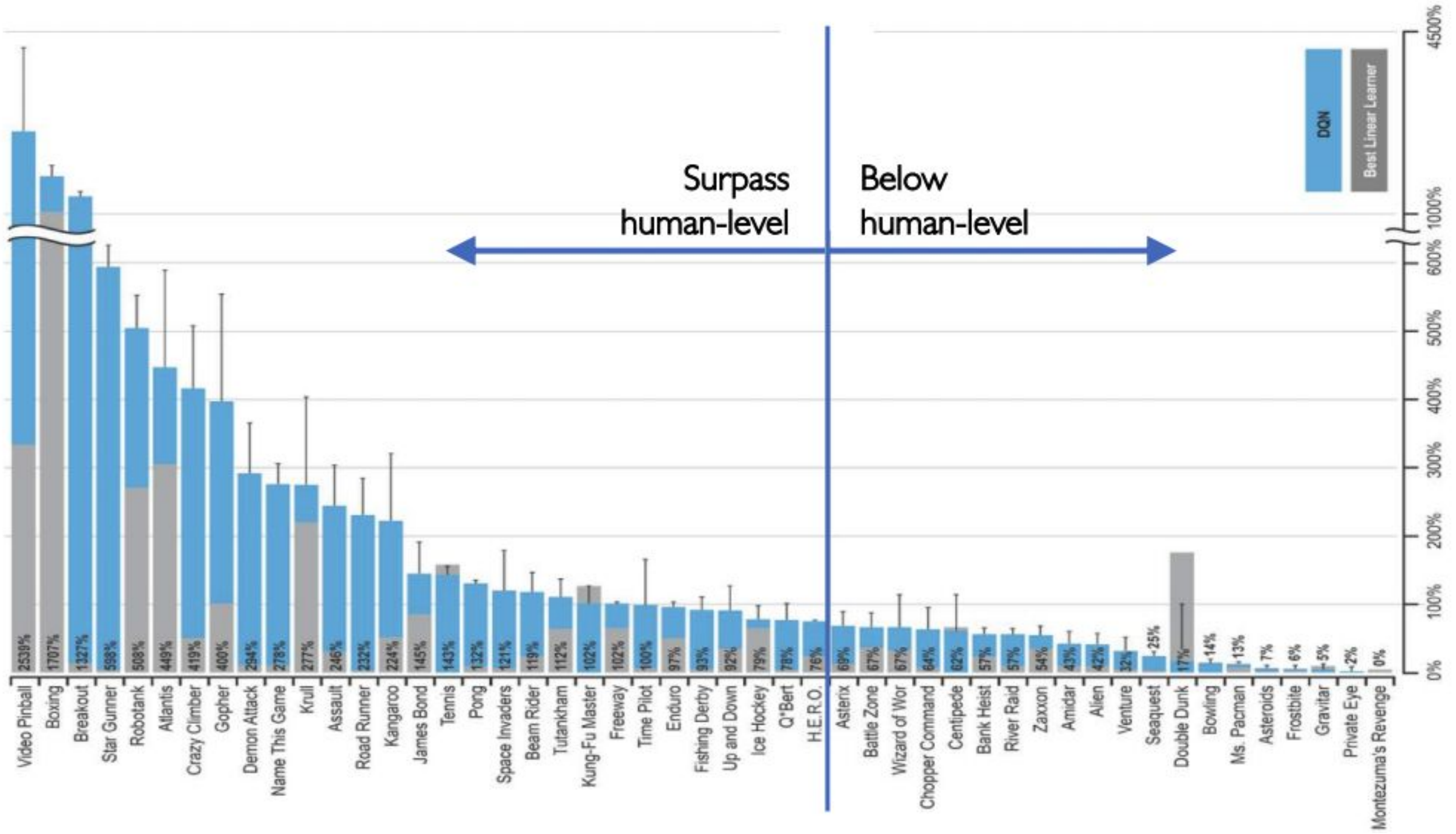
Experience Replay Tricks

$$\mathbf{L} = \mathbb{E}\Big[\big|\big|\underbrace{\big(r + \gamma \max_{a'} \mathbf{Q}(s', a')\big)}_{\textbf{target}} - \underbrace{\mathbf{Q}(s, a)}_{\textbf{predicted}}\big|\big|^2\Big]$$

- **<u>Problem</u>**: current Q-network parameters determines how the training progress → can lead to **bad feedback loop**
- **<u>Solution</u>**:
  - Stores experience (actions, state transitions, and rewards) and create **mini-batches** from them for the training process
  - Continually update a **replay memory table** of transitions as game experience are played
  - Update Q-network on random mini-batch of transitions from the replay memory, instead of consecutive samples

Demo video

DQN on Atari


Surpass human-level
Below human-level
DQN
Best Linear Learner

Downsides of Q-learning

- **Complex**: A problem with Q-learning is that the Q-function can be very complicated with high dimensional states
- **Discrete**: Because the action space is discrete and small, it cannot handle continuous space
- **Deterministic**: Policy is deterministically computed from Q-function, so it cannot learn stochastic policies

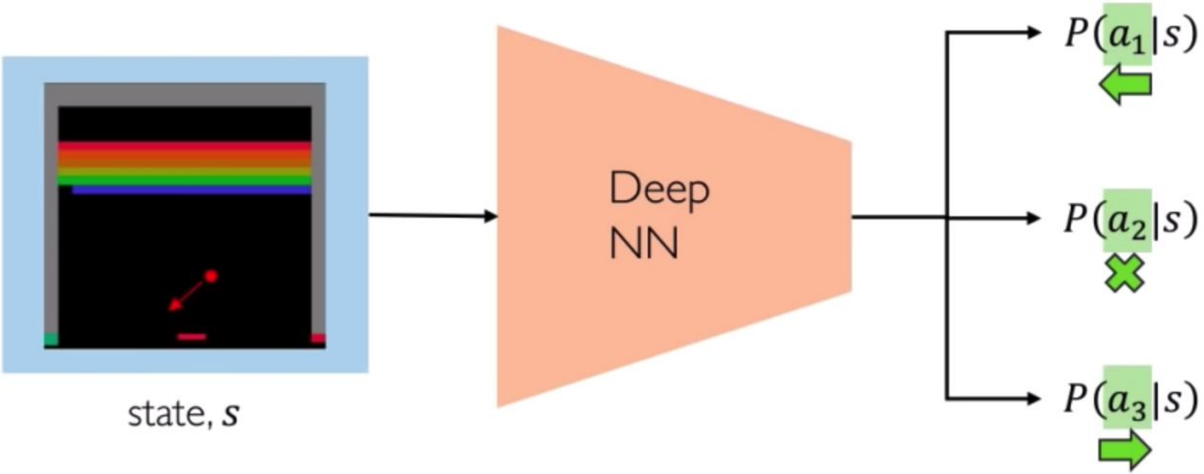
Can we optimize a policy **directly** by finding the best one from a set of policies?

Policy Learning / Policy Gradients

Policy Gradients (PG): Key Idea

DQN: Approximating Q and inferring the optimal policy,

Policy Gradients: directly optimize the policy!

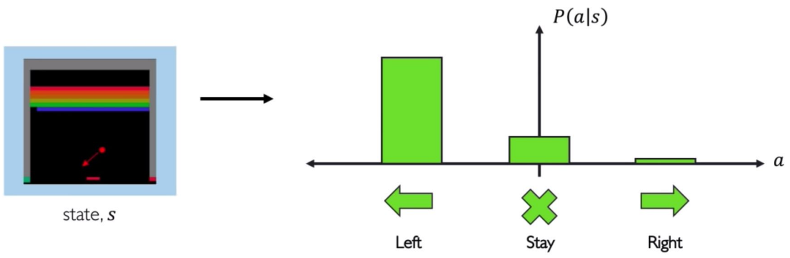


$$\sum_{a_i \in \mathbf{A}} P(a_i|s) = 1$$

Discrete vs Continuous Action Spaces

Discrete:

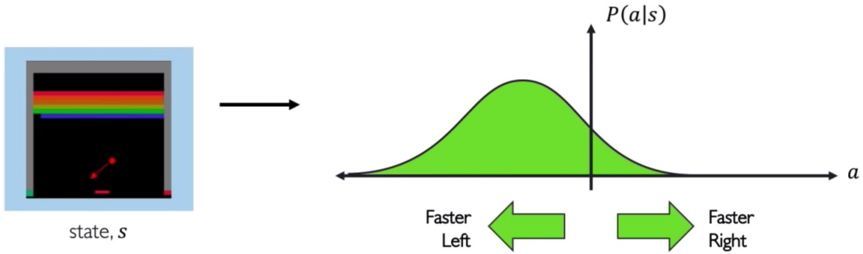
Which direction should I move?



Continuous:

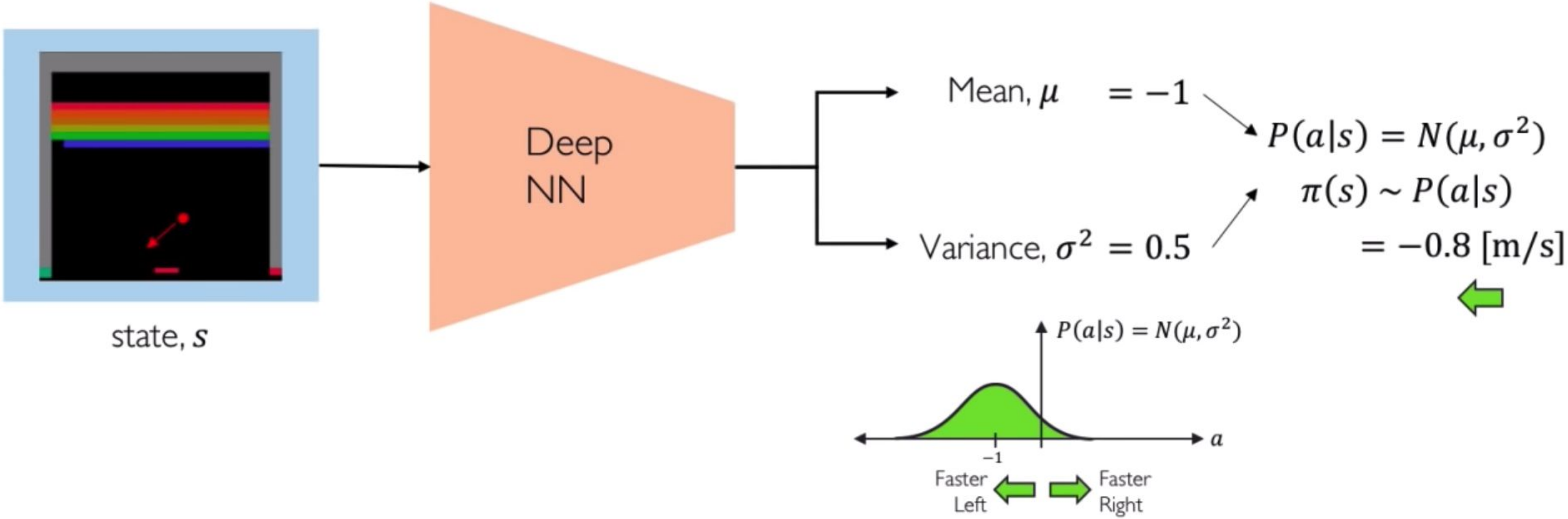
How fast should I move?

- 0.2 m/s



Policy Gradients: Continuous action space

Policy Gradients: enables modeling of **continuous** action space



Policy Gradients (PG): Formulation

Formally, let's define a class of policies: $\Pi = \{\pi_\theta, \theta \in \mathbb{R}^m\}$

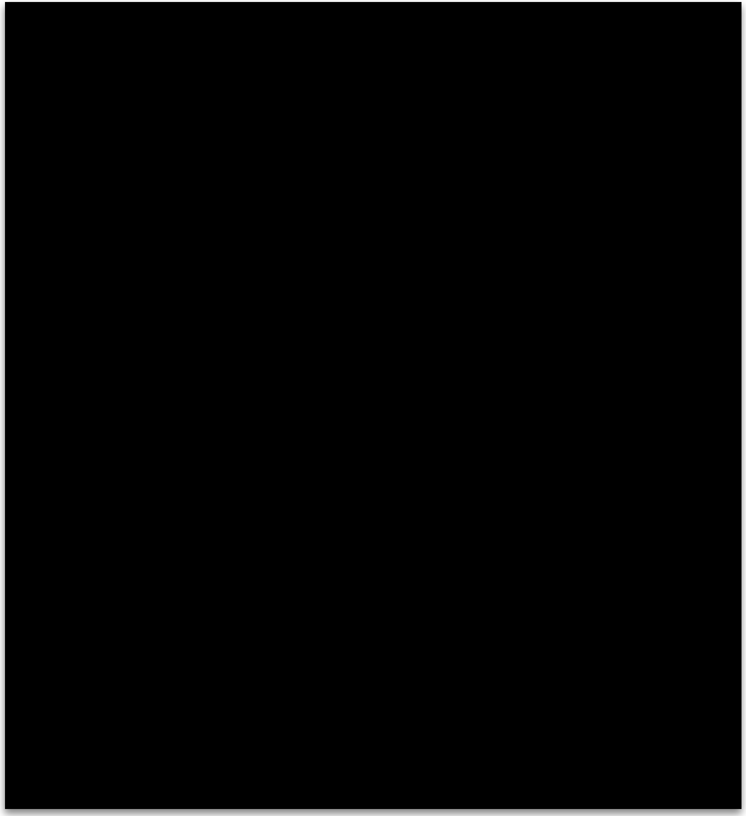
For each policy, define its value as the expected **discounted future rewards**:

$$J(\theta) = \mathbb{E}\Big[\sum_{t \geq 0} \gamma^t r_t | \pi_\theta\Big]$$

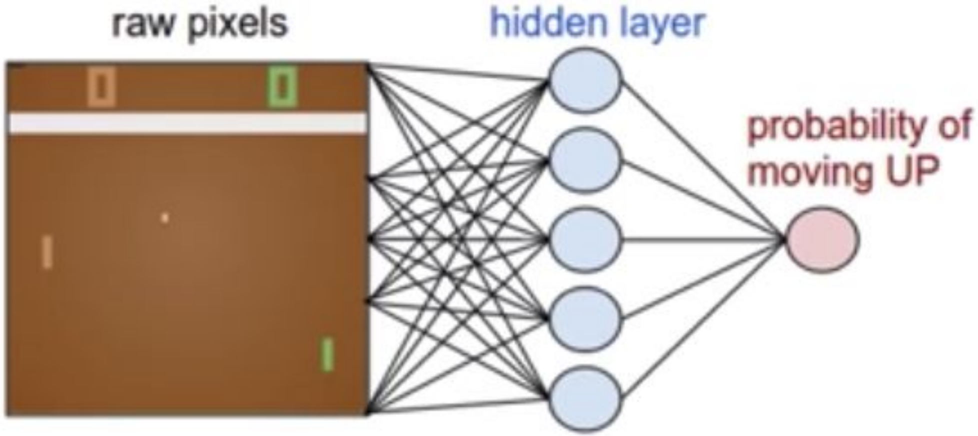
We want to find the optimal policy: $\pi_\theta^* = \arg\max_\theta J(\theta)$

How to do this? → **Gradient Descent/Ascent** $\nabla_\theta J(\theta)$ on policy parameter θ → gives raise to the name: Policy Gradient!

Policy Gradients -- on Pong*



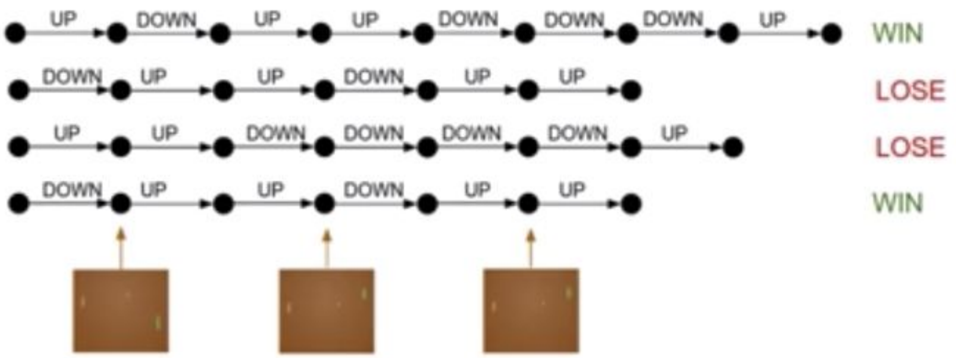
- 80 x 80 image (difference image)
- 2 actions: up or down
- 200,000 Pong games
- This is a step towards **general purpose AI** !



*Karpathy et al. “Deep Reinforcement Learning: Pong from Pixels.” 2016.

Policy Gradients: Training

Series of actions:



1. Run (in simulation) a policy π_θ for a while
2. Record all states, actions, rewards into episodes
3. **Increase ▲** probability of actions that lead to **high rewards**
4. **Decrease ▼** probability of actions that lead to **low/no rewards**

log-likelihood of action

discounted reward

$$J_\theta = -\boxed{\log P_\theta(a_t|s_t)}\boxed{R_t}$$

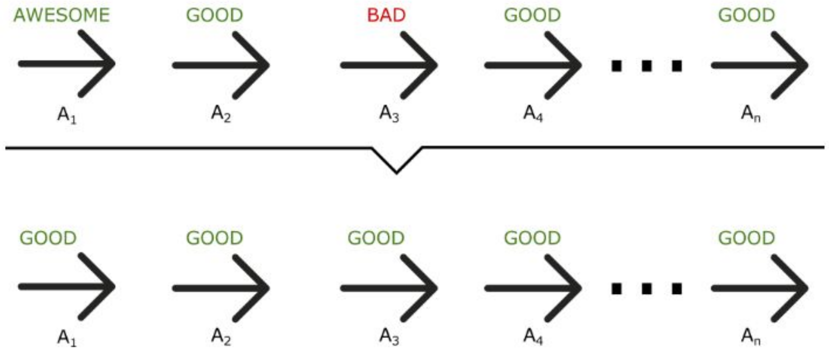
$$\theta = \theta - \alpha \nabla J_\theta$$

$$\theta = \theta + \alpha \boxed{\nabla \log P_\theta(a_t|s_t) R_t}$$

policy gradient

function REINFORCE
  Initialize θ
  for $episode \sim \pi_\theta$
    $\{s_i, a_i, r_i\}_{i=1}^{T-1} \leftarrow$ $episode$
    for t = 1 to T-1
      $\nabla \leftarrow \nabla_\theta \log P_\theta(a_t|s_t)\, R_t$
      $\theta \leftarrow \theta + \alpha\nabla$
  return θ

Some issues with Policy Gradients



We have to wait until the **end** of a trajectory to calculate the reward.

If the reward was high, then all actions were thought as good, even if some were **really bad!**

As a consequence, we need to have a **LOT of samples** to have an optimal policy. This also means **slow** learning and *long time to converge*

PG in comparison to DQN

Pros:

+ **Complexity:** If Q function is too complex to learn, DQN may fail while PG will still learn a good policy
+ **Speed**: Faster convergence
+ **Stochastic**: PG is capable of learning stochastic policies while DQN cannot
+ **Continuous actions:** It's easier to model PG on continuous space

Cons:

- **Data**: Sample inefficient (needs more data than DQN)
- **Stability**: Less stable during training process
- **Credit Assignment:** Poor assignment to (state, action) pairs for ***delayed*** rewards

What if we can do update at each step?!

Hybrid Method: Actor-Critic

Introducing Actor-Critic Algorithm

Using two neural networks:

1. **An Actor** that controls which action to take (**Policy-based Network**)
2. **A Critic** that measures how good the taken action (**Value-based DQN**)

The Actor Critic Design

- The actor decides which action to take, and the critic tells the actor how good its action was and how it should adjust
- Alleviates the task of the critic as it only has to learn the values of (state, action) pairs generated by the policy
- Can also incorporate some Q-Learning **tricks** (e.g. experience replay)

- **Advantage function**: define how much an action was better than expected

$$A^{\pi}(s,a) = \underbrace{\boxed{r(s,a) + \gamma V_{\phi}^{\pi}(s')}}_{\text{reward by the actor's action}} - \underbrace{\boxed{V_{\phi}^{\pi}(s)}}_{\text{expected reward (by the critic)}}$$

- Using this, our gradient estimator becomes:

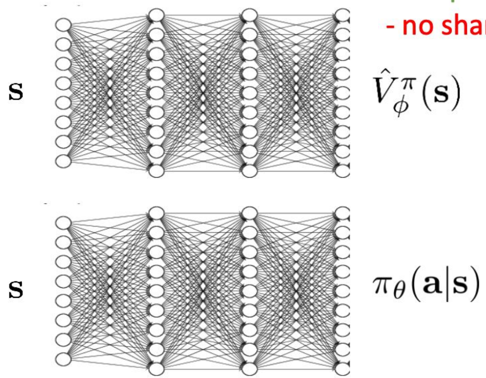
$$\nabla_{\theta} J_{\theta} \approx \nabla_{\theta} \log \pi_{\theta}(a|s) A^{\pi}(s,a)$$

Advantage Actor Critic (A2C) Algorithm

repeat

1. Take action $a \sim \pi(a|s)$, get (s, a, s', r)
2. Update $\hat{V}^{\pi}_{\phi}$ using target $r + \gamma \hat{V}^{\pi}_{\phi}(s')$
3. Evaluate advantage $\hat{A}^{\pi}(s,a) = r(s,a) + \gamma \hat{V}^{\pi}_{\phi}(s') - \hat{V}^{\pi}_{\phi}(s)$
4. Compute gradients $\nabla_\theta J_\theta \approx \nabla_\theta \log \pi_\theta(a|s) \hat{A}^{\pi}(s,a)$
5. Update weights $\theta \leftarrow \theta + \alpha \nabla_\theta J_\theta$

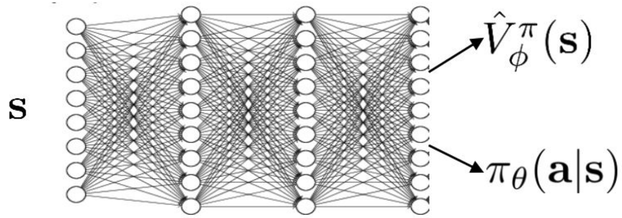
two network design



+ simple & stable

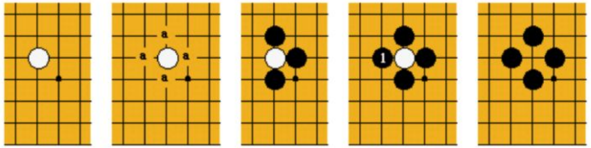
- no shared features between actor & critic

shared network design

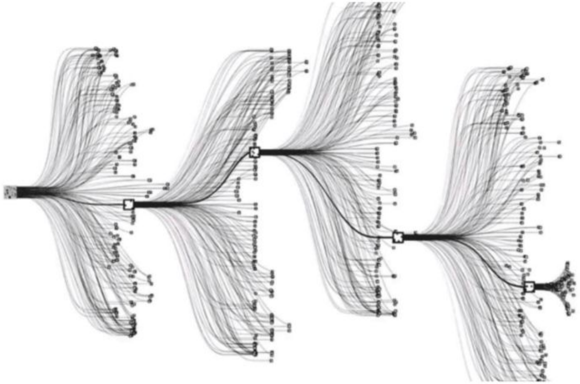


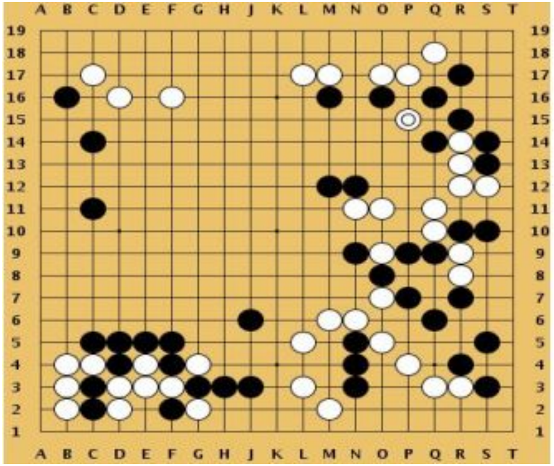
Application: The Game of Go

Goal: Get more board territory than your opponent



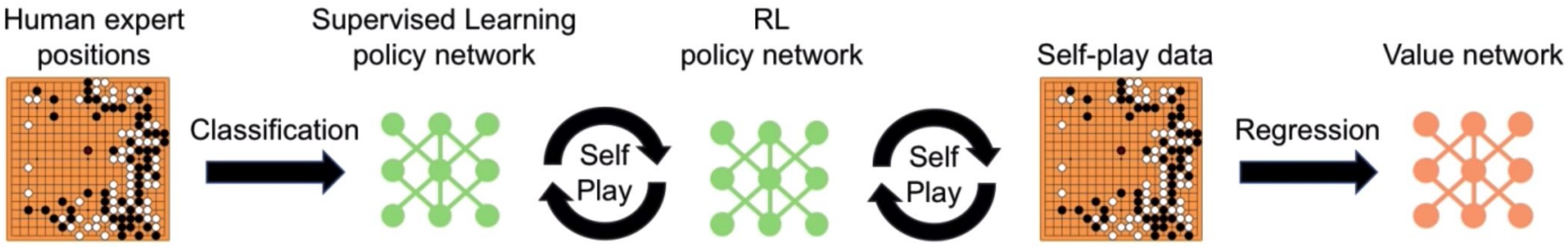
Dia. 3 | Dia. 4 liberties | Dia. 5 atari | Dia. 6 capture | Dia. 7 result





Game size	Board size N	3^N	Percent legal	legal game positions (A094777)[11]
1×1	1	3	33%	1
2×2	4	81	70%	57
3×3	9	19,683	64%	12,675
4×4	16	43,046,721	56%	24,318,165
5×5	25	8.47×10^{11}	49%	4.1×10^{11}
9×9	81	4.4×10^{38}	23.4%	1.039×10^{38}
13×13	169	4.3×10^{80}	8.66%	3.72497923×10^{79}
19×19	361	1.74×10^{172}	1.196%	$2.08168199382\times10^{170}$

AlphaGo by DeepMind (2016): Pipeline

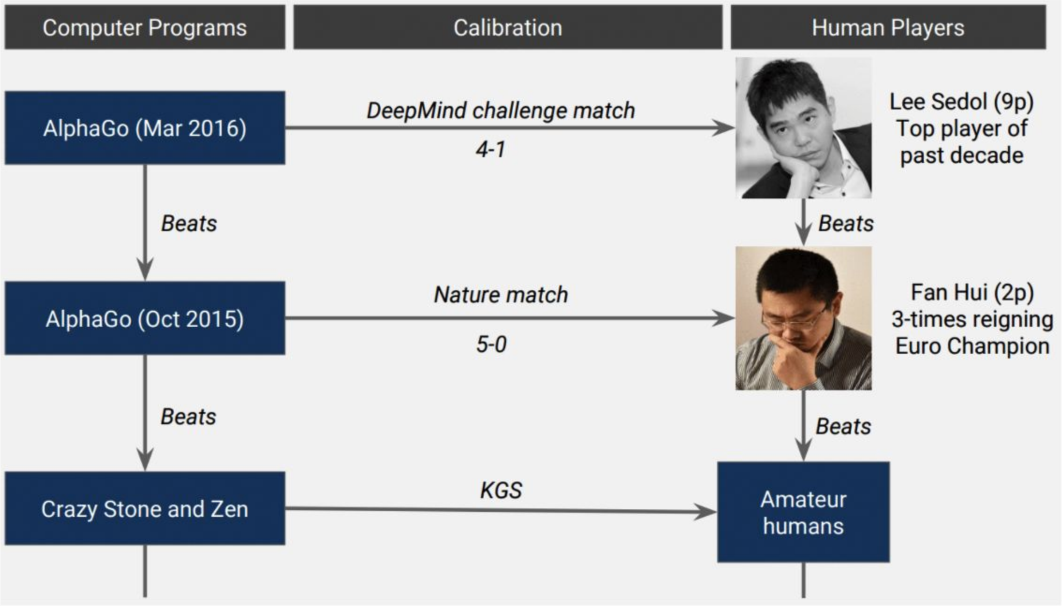


Phase 1) **Initial training**: human data from professional Go games

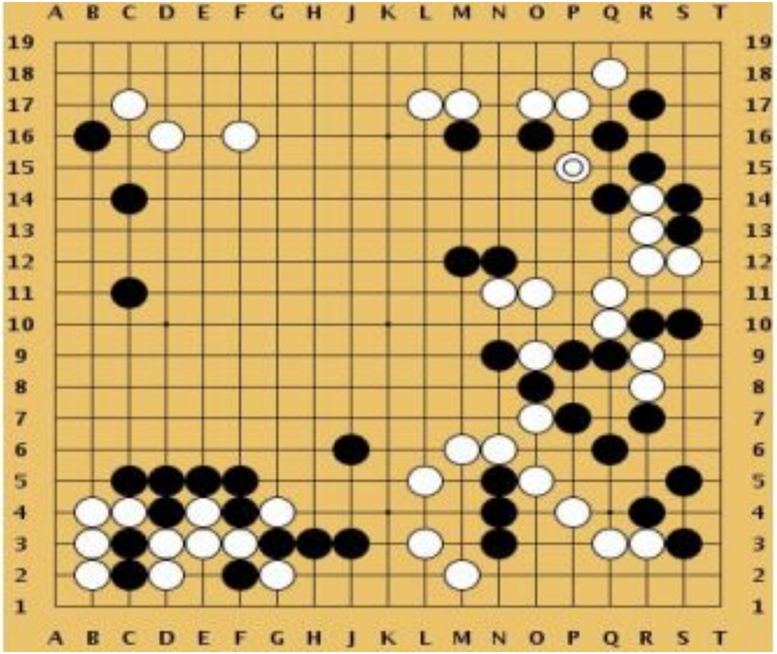
Phase 2) **Self-play** (with actor-critic framework) and train using policy gradients with +1/-1 reward for win/lost.

Phase 3) **Combine policy and value networks** in a Monte Carlo Tree Search algorithm to predict how good a current state and select actions by look-ahead search

AlphaGo: Journey to beat the best Go players



AlphaGo and variants



AlphaGo [Nature 2016]:

- Required many engineering tricks
- Bootstrapped from human play
- Beat 18-time world champion Lee Sedol

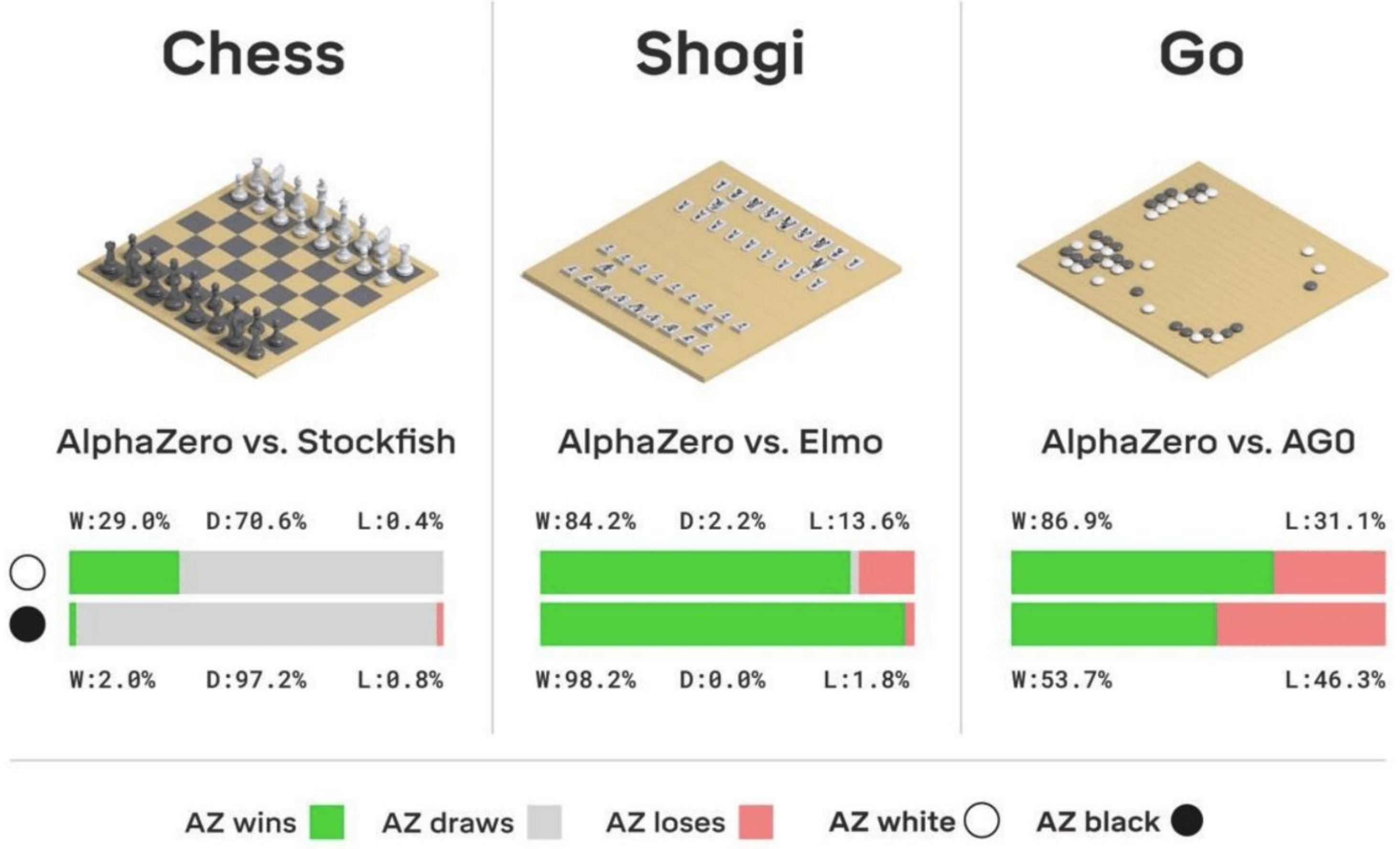
AlphaGo Zero [Nature 2017]:

- Simplified and elegant version of AlphaGo
- No longer need data from *human gameplays*
- Beat (at the time) #1 world ranked Ke Jie

Alpha Zero: [Science 2018]

- Generalized to beat world champion programs on *chess* and *shogi* as well

Alpha Zero in action



Summary:

- ✓ Value-based Deep Q-Networks to approximate Q function and infer action
- ✓ Policy-based Policy Gradients to learn the policy directly
- ✓ In the Actor-Critic algorithm, the actor decides which action to take and the critic tells the actor how good its action was and how it should adjust
- ✓ Applications in AlphaGo and AlphaZero

Next: **Implementation of RL using TF-Agents Library**

Acknowledgement

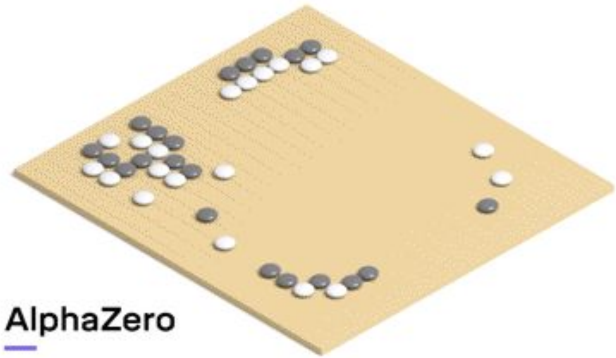
Slides contain materials and figures reproduced from Alex Amini (MIT) and Serena Yeung (Stanford) for educational purposes only.

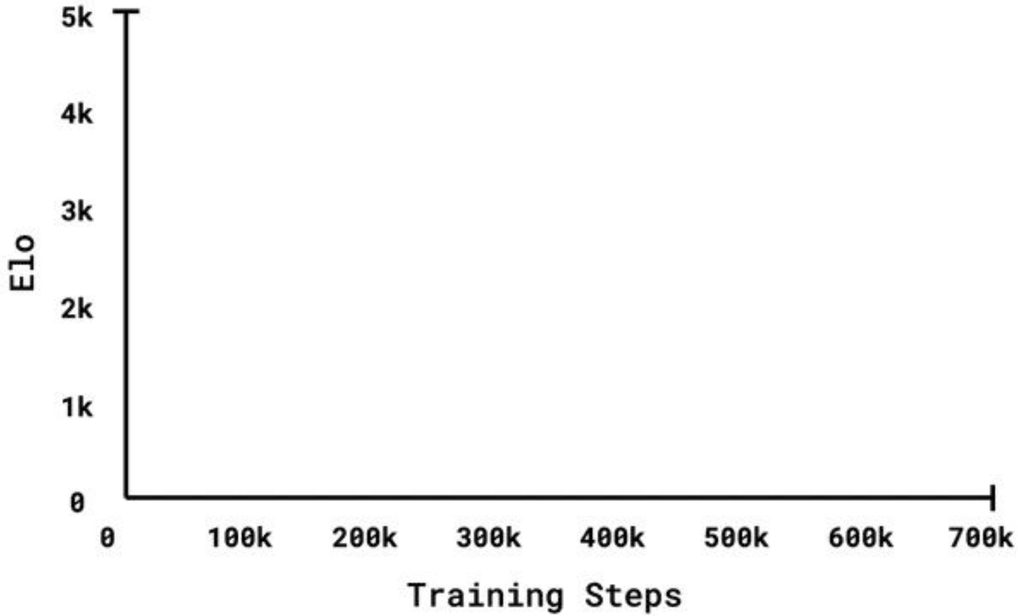
Bonus Content

*From DeepMind blog

Alpha Zero in action

Elo rating is a method to calculate relative skill level of player in a zero-sum based game such as chess. Each training step represents 4,096 board positions





Environment and Actions

Fully Observable (Chess) vs. **Partially Observable** (Poker)

Single Agent (Atari) vs. **Multi Agent** (DeepTraffic)

Deterministic (Cart Pole) vs. **Stochastic** (Life)

Static (Chess) vs. **Dynamic** (Doom)

Discrete (Chess) vs. **Continuous** (Cart Pole)

Policy Gradients (PG)

Mathematically, we can rewrite J() in terms of action trajectory:

$$J(\theta) = \mathbb{E}_{\tau \sim p(\tau;\theta)} \left[r(\tau) \right]$$

Where **r(τ)** is the reward of a trajectory: $\tau = (s_0, a_0, r_0, s_1, a_1, r_2, \ldots)$

Gradient Estimator (*skipping the derivation...*):

$$\nabla_\theta J(\theta) \approx \sum_{t \geq 0} r(\tau) \nabla_\theta \log \pi_\theta (a_t | s_t)$$

Interpretation:

- If **r(τ)** is high, push up the probabilities of the seen actions
- If **r(τ)** is low, push down the probabilities of the seen actions

PG Application: AlphaGo

Mix of supervised learning and reinforcement learning

How to beat the Go world champion:

- Featurize the board (stone color, move legality, bias,…)
- Initialize policy network with supervised training from professional go games, then continue training using policy gradient (**play against itself** from random previous iterations, +1 / -1 reward for winning / losing)
- Also learn value network (the critic)
- Finally, combine policy and value networks in a Monte Carlo Tree Search algorithm to select actions by look-ahead search