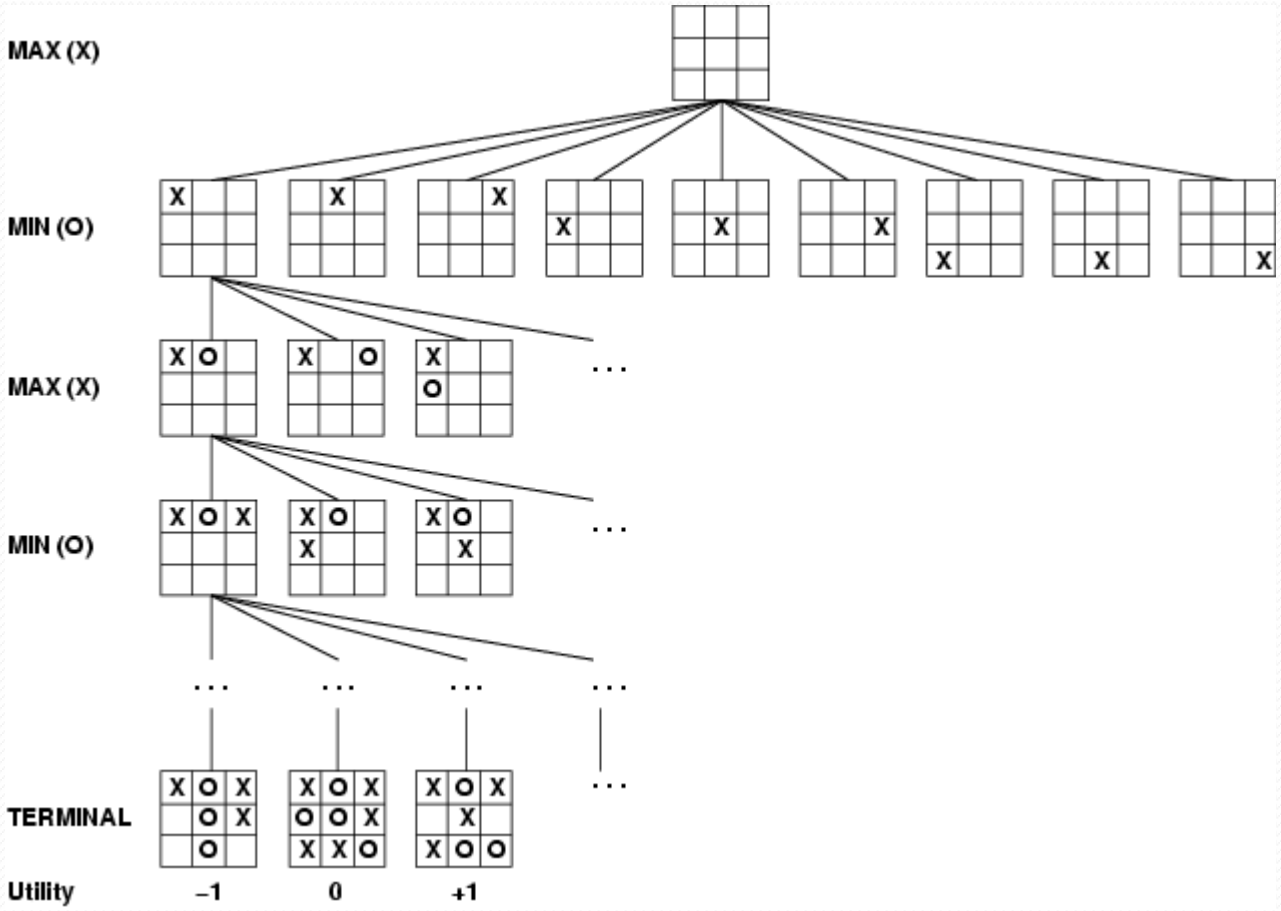
Applied Deep Learning

Game tree (2-player, deterministic, turns)



How do we search this tree to find the optimal move?

Challenge

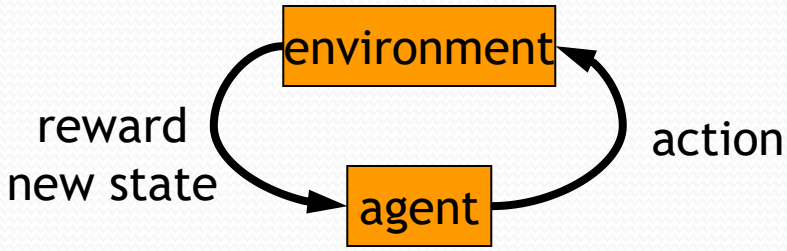
What if we cannot search all possibilities due to the size of the space?

Learn while explore!

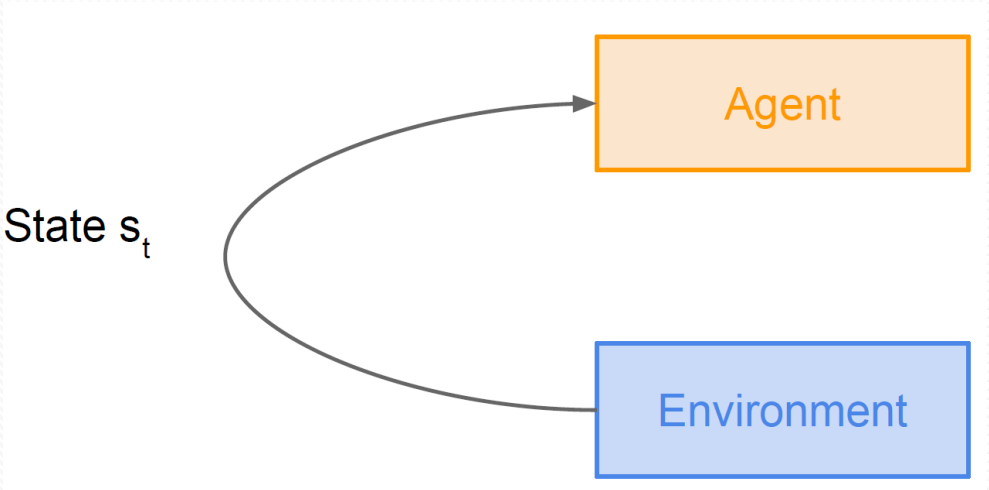
Deep Reinforcement Learning

Supervised, Unsupervised, Reinforcement

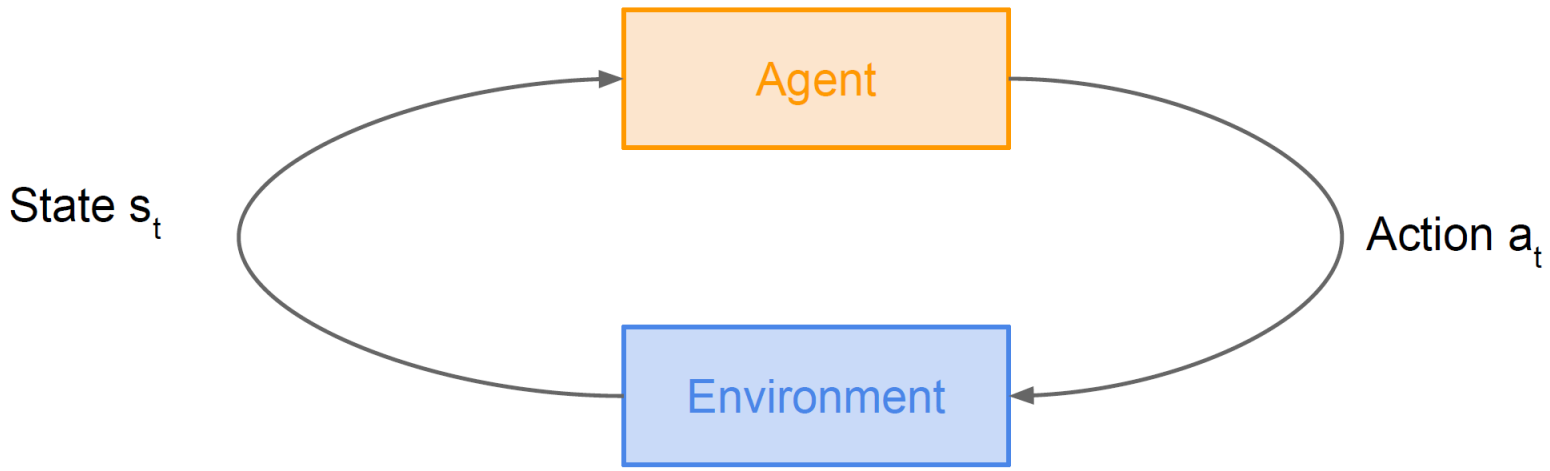
- Supervised learning
  - classification, regression
- Unsupervised learning
  - for example, autoencoder
- Reinforcement learning
  - more general than supervised/unsupervised learning
  - learn from interaction w/ environment to achieve a goal



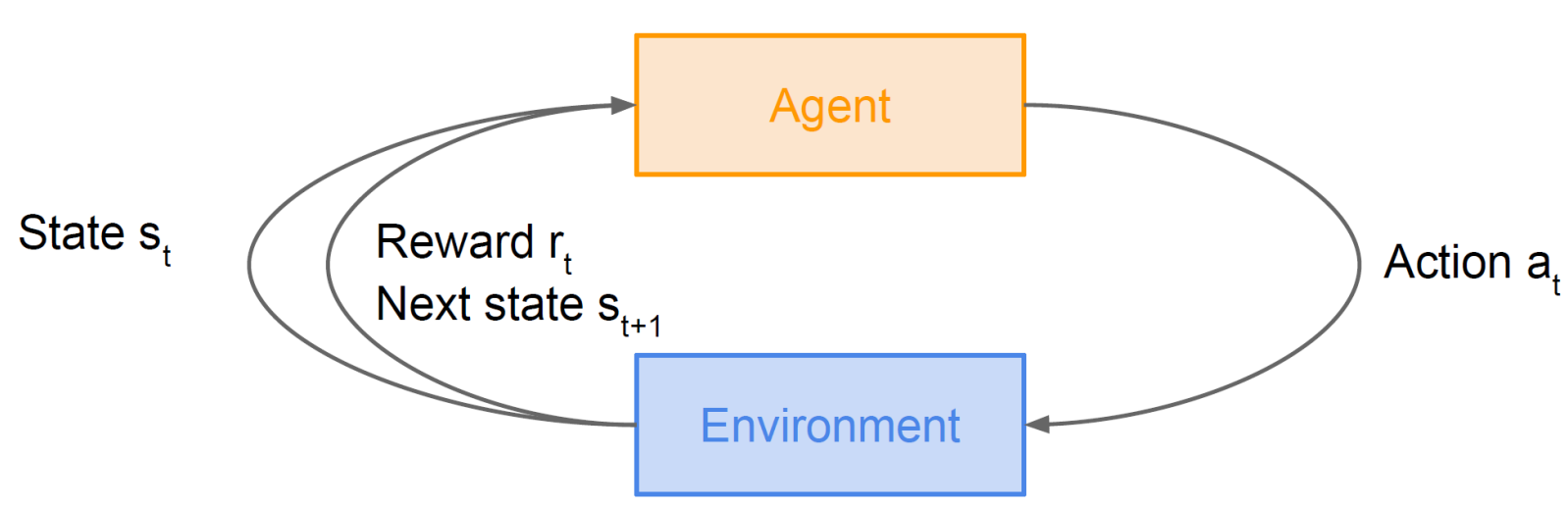
Reinforcement Learning



Reinforcement Learning (2)

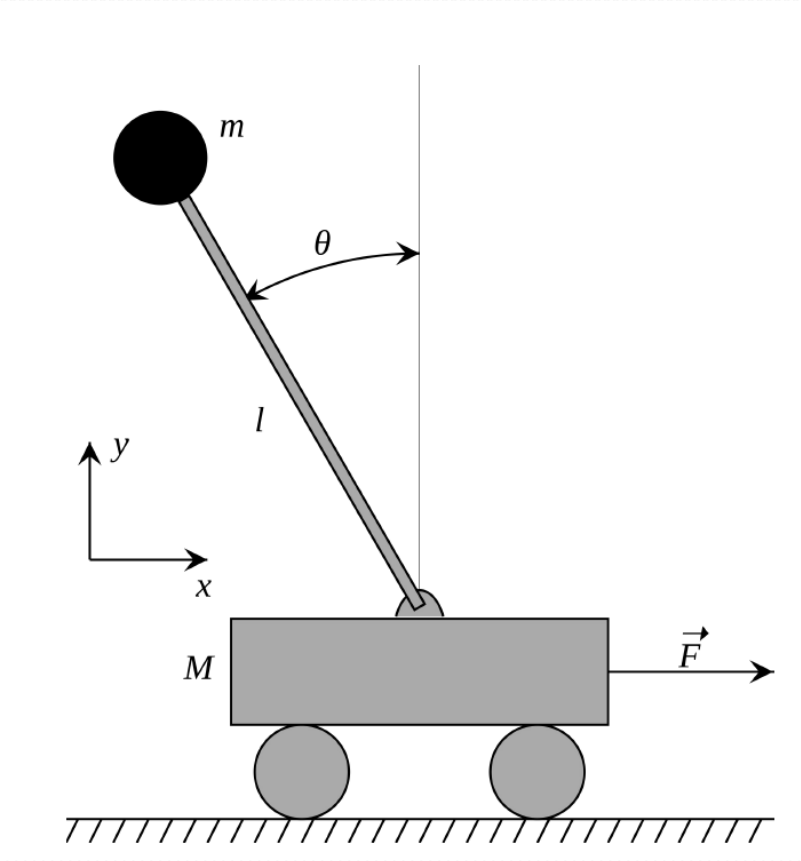


Reinforcement Learning (3)



Cart-Pole Problem

- Objective: Balance a pole on top of a movable cart
- State: angle, angular speed, position, horizontal velocity
- Action: horizontal force applied on the cart
- Reward: 1 at each time step if the pole is upright

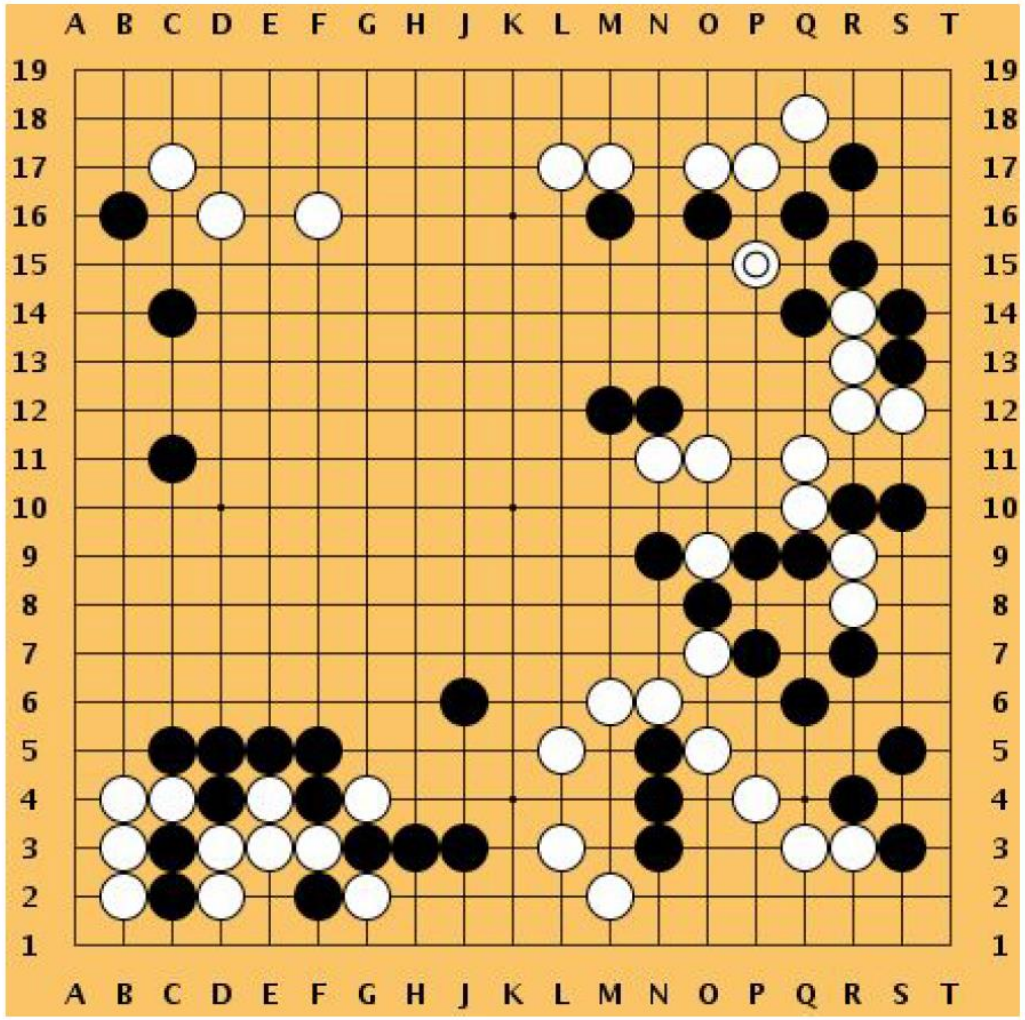


Atari Games

- Objective: Complete the game with the highest score
- State: Raw pixel inputs of the game state
- Action: Game controls e.g. Left, Right, Up, Down
- Reward: Score increase/decrease at each time step

Go

- Objective: Win the game!
- State: Position of all pieces
- Action: Where to put the next piece down
- Reward: 1 if win at the end of the game, 0 otherwise



Math Formulation

Defined by: $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathbb{P}, \gamma)$

- $\mathcal{S}$: set of possible states
- $\mathcal{A}$: set of possible actions
- $\mathcal{R}$: distribution of reward given (state, action) pair
- $\mathbb{P}$: transition probability i.e. distribution over next state given (state, action) pair
- γ : discount factor

Reinforcement Learning Objective

At time step t=0, environment samples initial state $s_0 \sim p(s_0)$
Then, for t=0 until done:

- Agent selects action a_t
- Environment samples reward $r_t \sim R(\cdot \mid s_t, a_t)$
- Environment samples next state $s_{t+1} \sim P(\cdot \mid s_t, a_t)$
- Agent receives reward r_t and next state s_{t+1}

Reinforcement Learning Objective (2)

At time step t=0, environment samples initial state $s_0 \sim p(s_0)$
Then, for t=0 until done:

- Agent selects action a_t
- Environment samples reward $r_t \sim R(. | s_t, a_t)$
- Environment samples next state $s_{t+1} \sim P(. | s_t, a_t)$
- Agent receives reward r_t and next state s_{t+1}

A policy π is a function from a state (s) to an action (a) that specifies what action to take in that state

- Clearly, a policy needs to specify an action for all possible states the agent may encounter.

Objective of RL: find the optimal policy π^* that maximizes cumulative discounted reward: $r(s_0) + \gamma * r(s_1) + \gamma^2 * r(s_2) + \ldots$

Computing cumulative rewards

- additive rewards
  - $V(s_0, s_1, \ldots) = r(s_0) + r(s_1) + r(s_2) + \ldots$
  - infinite value for continuing tasks

- discounted rewards
  - $V(s_0, s_1, \ldots) = r(s_0) + \gamma * r(s_1) + \gamma^2 * r(s_2) + \ldots$
  - value bounded if rewards bounded

Value functions

- Following a policy produces sample trajectories (or paths) $s_0, a_0, r_0, s_1, a_1, r_1, \ldots$

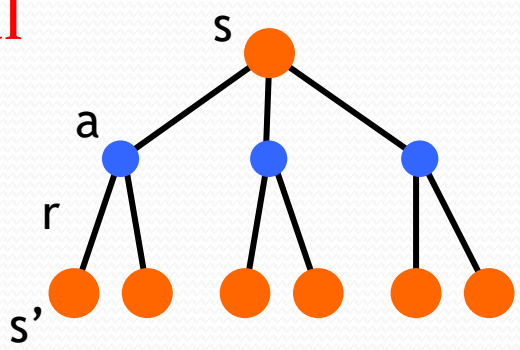
- (state) value function: $V^\pi(s)$ $\quad V^\pi(s) = \mathbb{E}\left[\sum_{t\geq 0} \gamma^t r_t | s_0 = s, \pi\right]$
  - Measures how good a state is
  - Value function of a state s is the expected return when starting in *s* and following π

- Action value (Q) function: $Q^\pi(s,a)$ $\quad Q^\pi(s,a) = \mathbb{E}\left[\sum_{t\geq 0} \gamma^t r_t | s_0 = s, a_0 = a, \pi\right]$
  - Measures how good a pair of state and action is
  - Q function is the expected return when starting in *s*, performing *a*, and following π

Value functions (2)

- Value function: $V^{\pi}(s)$
  - expected return when starting in *s* and following π
- Action value function: $Q^{\pi}(s,a)$
  - expected return when starting in *s*, performing *a*, and following π

- the value functions are useful for finding the optimal policy



- Recursive relationship between value functions

$$V^{\pi}(s) = \sum_{a} \pi(s,a) \sum_{s'} P^{a}_{ss'} \left[r^{a}_{ss'} + \gamma V^{\pi}(s') \right] = \sum_{a} \pi(s,a) Q^{\pi}(s,a)$$

Optimal value function

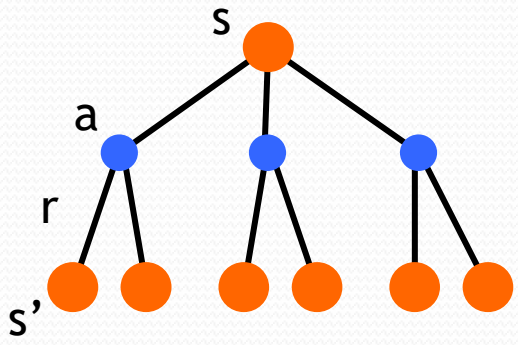
- Optimal value function is the maximum expected cumulative reward achievable from a given state.

$$V^*(s) = \max_\pi V^\pi(s)$$

  - The value function following an optimal policy

- Bellman optimality equation of value function

$$V^*(s) = \max_a \sum_{s'} P^a_{ss'} \left[r^a_{ss'} + \gamma V^*(s') \right]$$



  - If all information is known, we can use the Bellman equation to solve for V*(s)
  - We can follow the equation to derive optimal policy

Solving the Bellman Eq.

- Finding an optimal policy by solving the Bellman Optimality Equation requires the following:
  - accurate knowledge of environment dynamics;
  - we have enough space and time to do the computation;
- How much space and time do we need?
  - polynomial in number of states (via dynamic programming methods),
  - BUT, number of states is often huge
- We usually have to settle for approximations.
- Many RL methods can be understood as approximately solving the Bellman Optimality Equation.

Monte Carlo methods

- Don't need full knowledge of environment
  - just experience, or
  - simulated experience

- Two main components
  - policy evaluation: compute V^{π} from π
  - policy improvement: improve π based on V^{π}

  - start with an arbitrary policy
  - repeat evaluation/improvement until convergence

Policy Improvement

- Suppose we have computed V^{π} for a deterministic policy π.
- For a given state s, can we find a better action $a \neq \pi(s)$?
- It is better to switch to action a for state s if and only if

$$Q^{\pi}(s,a) > V^{\pi}(s)$$

- Do the following for all states to get a new policy π' that is **greedy** with respect to V^{π} :

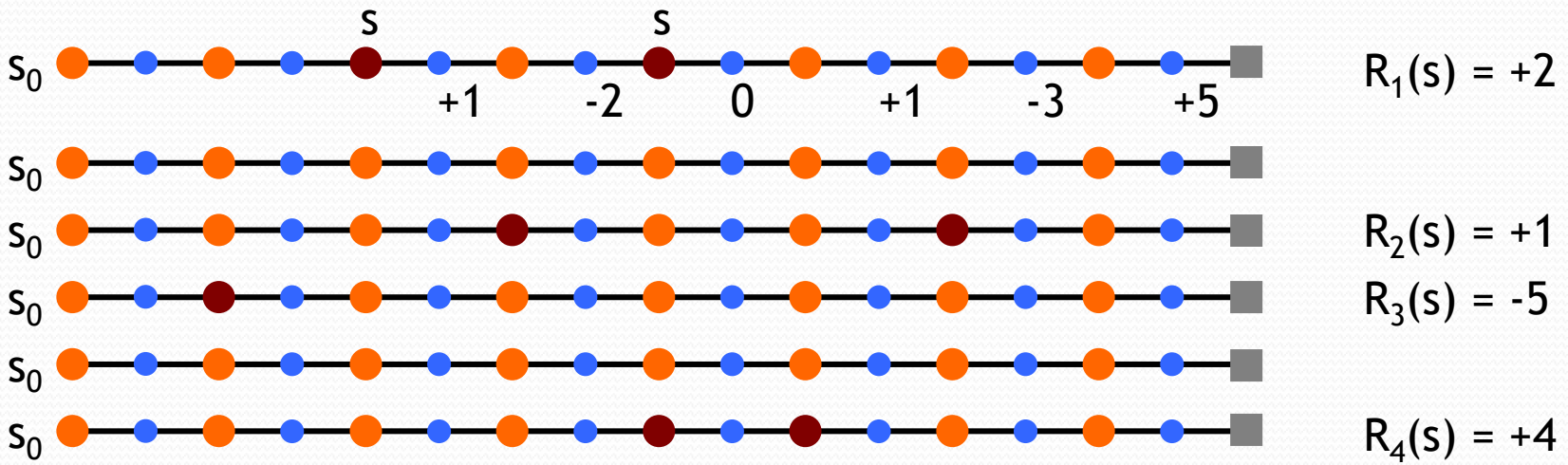
$$\pi'(s) = \text{argmax}_a \, Q^{\pi}(s,a)$$

$$= \arg\max_a \sum P^a_{ss'} \left[R^a_{ss'} + \gamma V^{\pi}(s') \right]$$

- Then $V\pi' \geq V\pi$

Monte Carlo policy evaluation

- want to estimate $V^{\pi}(s)$
  = expected return starting from s and following π
  - estimate as average of observed returns in state s
- first-visit MC
  - average returns following the first visit to state s



$V^{\pi}(s) \approx (2 + 1 - 5 + 4)/4 = 0.5$

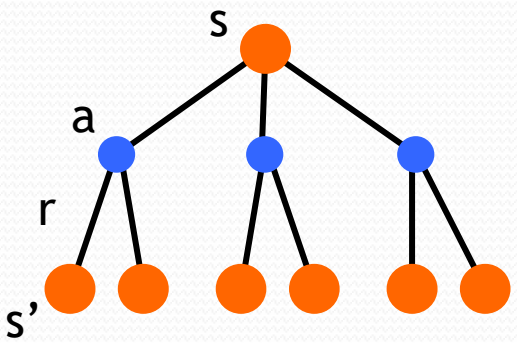
Optimal action value function

- The optimal Q-value function Q* is the maximum expected cumulative reward achievable from a given (state, action) pair:

$$Q^*(s,a) = \max_{\pi} \mathbb{E}\left[\sum_{t \geq 0} \gamma^t r_t | s_0 = s, a_0 = a, \pi\right]$$

- Bellman optimality equation of Q function

$$Q^*(s,a) = \mathbb{E}_{s' \sim \mathcal{E}}\left[r + \gamma \max_{a'} Q^*(s',a') | s,a\right]$$



- Q*(s,a) makes it simpler to find the optimal policy:

$$\pi^*(s) = \arg\max_{a} Q^*(s,a)$$

Q-Learning

$$\hat{Q}(s,a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s',a') \qquad s' = s_{t+1}$$

- This equation continually estimates Q at state s consistent with an estimate of Q at state s', one step in the future.
- Note that s' is closer to goal, and hence more "reliable", but still an estimate itself.
- We do an update after each state-action pair. I.e., we are learning online.
- We are learning useful things about explored state-action pairs. These are typically most useful because they are likely to be encountered again.
- Under suitable conditions, these updates can actually be proved to converge to the optimal Q.

Q-Learning (2)

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)\right]$$

Initialize $Q(s, a)$ arbitrarily
Repeat (for each episode):
    Initialize s
    Repeat (for each step of episode):
        Choose a from s using policy derived from Q (e.g., ε-greedy)
        Take action a, observe r, s'
        $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$
        $s \leftarrow s'$;
    until s is terminal

Function approximation

- How to represent state when there are millions of them
  - cannot use a table to list all states
- If state cannot be listed, what about value function and action value function?

- Represent V or Q as a parameterized function $V_t(s) = \vec{\theta}_t^T \vec{\phi}_s = \sum_{i=1}^{n} \theta_t(i)\phi_s(i)$
  - Neural network
  - Represent Q function using Deep NN -> deep Q-learning

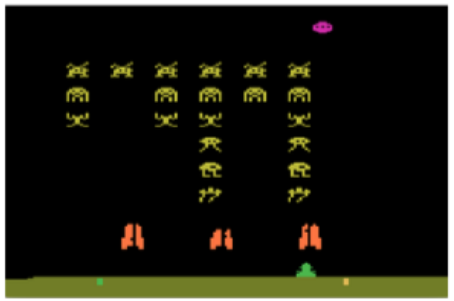
- Update parameters instead of entries in a table
  - better generalization
    - fewer parameters and updates affect "similar" states as well
  - How to make sure generalize in the right direction?

Deep Q-Networks (DQN, 2013)

- Introduced deep reinforcement learning
- It is common to use a function approximator $Q(s, a; \theta)$ to approximate the action-value function in Q-learning
- Deep Q-Networks is Q-learning with a deep neural network function approximator called the Q-network
- Discrete and finite set of actions A
- Uses epsilon-greedy policy to select actions

Learn to Playing Atari Games

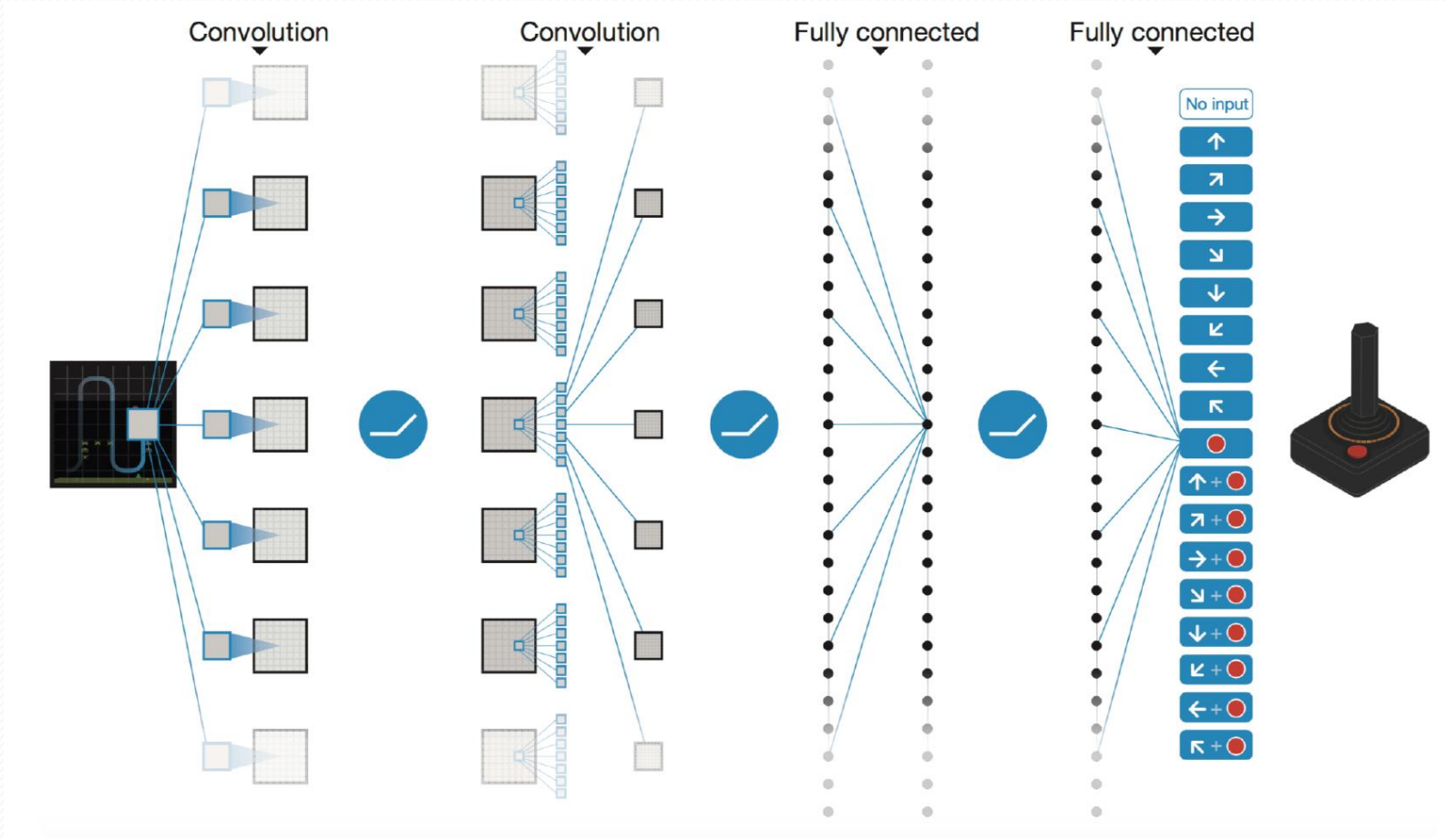





Q-Networks

- Core idea: We want the neural network to learn a non-linear hierarchy of features or feature representation that gives accurate Q-value estimates
- The neural network has a separate output unit for each possible action, which gives the Q-value estimate for that action given the input state
- The neural network is trained using mini-batch stochastic gradient updates and experience replay

Deep Q Network



Experience Replay

- The state is a sequence of actions and observations $s_t = x_1, a_1, x_2, \ldots, a_{t-1}, x_t$
- Store the agent's experiences at each time step $e_t = (s_t, a_t, r_t, s_{t+1})$ in a dataset $D = e_1, \ldots, e_n$ pooled over many episodes into a replay memory
- In practice, only store the last N experience tuples in the replay memory and sample uniformly from D when performing updates

State representation

- It is difficult to give the neural network a sequence of arbitrary length as input
- Use fixed length representation of sequence/history (why history?)
- Example: The last 4 image frames in the sequence of Breakout gameplay

Q-Network Training

- Sample random mini-batch of experience tuples uniformly at random from D
- Similar to Q-learning update rule but:
  - Use mini-batch stochastic gradient updates
  - The gradient of the loss function for a given iteration with respect to the parameter θ_i is the difference between the target value and the actual value multiplied by the gradient of the Q function approximator $Q(s, a; \theta)$ with respect to that specific parameter
- Use the gradient of the loss function to update the Q function approximator

Loss Function Gradient Derivation

network. We refer to a neural network function approximator with weights θ as a Q-network. A Q-network can be trained by minimising a sequence of loss functions $L_i(\theta_i)$ that changes at each iteration i,

$$L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot)}\left[(y_i - Q(s,a;\theta_i))^2\right], \tag{2}$$

where $y_i = \mathbb{E}_{s'\sim\mathcal{E}}\left[r + \gamma \max_{a'} Q(s',a';\theta_{i-1})|s,a\right]$ is the target for iteration i and $\rho(s,a)$ is a probability distribution over sequences s and actions a that we refer to as the *behaviour distribution*. The parameters from the previous iteration θ_{i-1} are held fixed when optimising the loss function $L_i(\theta_i)$. Note that the targets depend on the network weights; this is in contrast with the targets used for supervised learning, which are fixed before learning begins. Differentiating the loss function with respect to the weights we arrive at the following gradient,

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot);s'\sim\mathcal{E}}\left[\left(r + \gamma \max_{a'} Q(s',a';\theta_{i-1}) - Q(s,a;\theta_i)\right)\nabla_{\theta_i} Q(s,a;\theta_i)\right]. \tag{3}$$

Rather than computing the full expectations in the above gradient, it is often computationally expedient to optimise the loss function by stochastic gradient descent. If the weights are updated after every time-step, and the expectations are replaced by single samples from the behaviour distribution ρ and the emulator $\mathcal{E}$ respectively, then we arrive at the familiar *Q-learning* algorithm [26].

DQN Algorithm

Algorithm 1: deep Q-learning with experience replay.

Initialize replay memory D to capacity N

Initialize action-value function Q with random weights θ

Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$

For episode $= 1, M$ **do**

Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$

For $t = 1$,T **do**

With probability ε select a random action a_t

otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$

Execute action a_t in emulator and observe reward r_t and image x_{t+1}

Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$

Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D

Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D

Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$

Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ

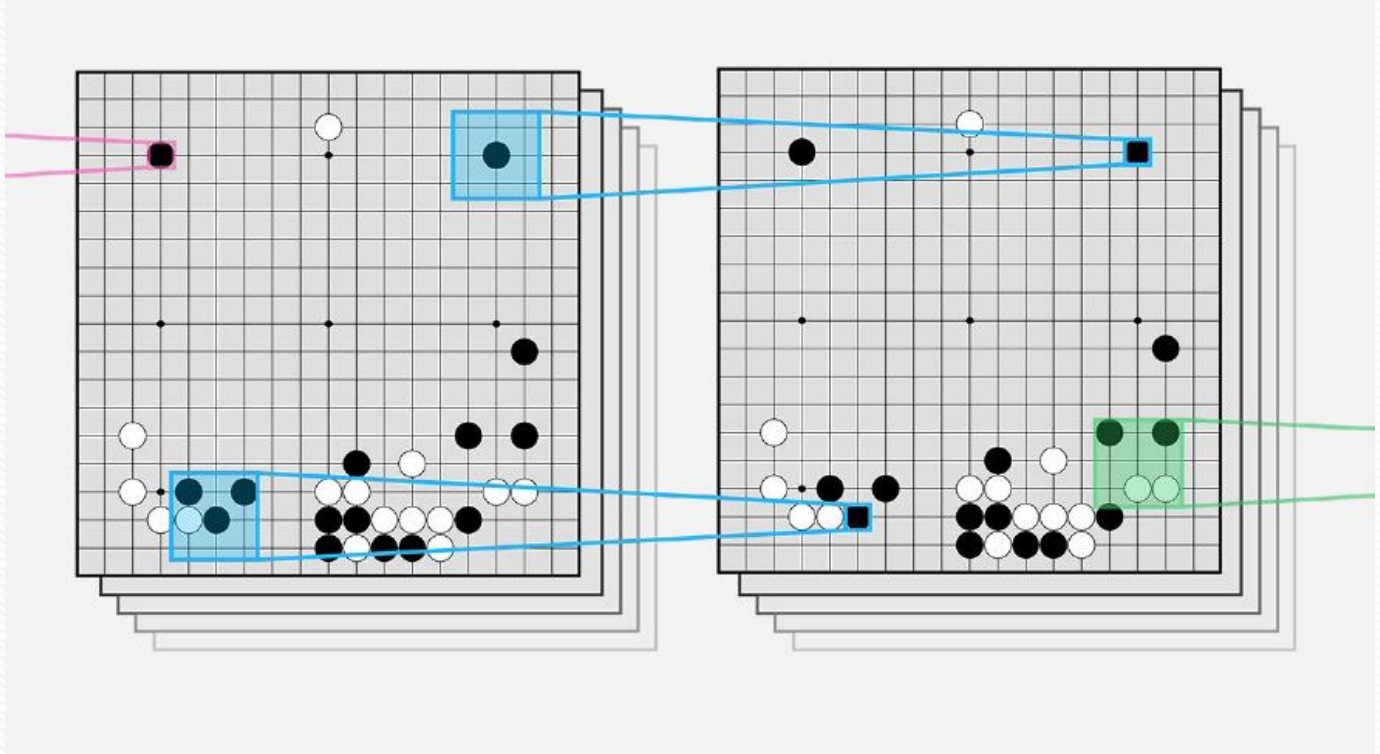
Every C steps reset $\hat{Q} = Q$

End For

End For

Convolutional Neural Network for Board Configuration

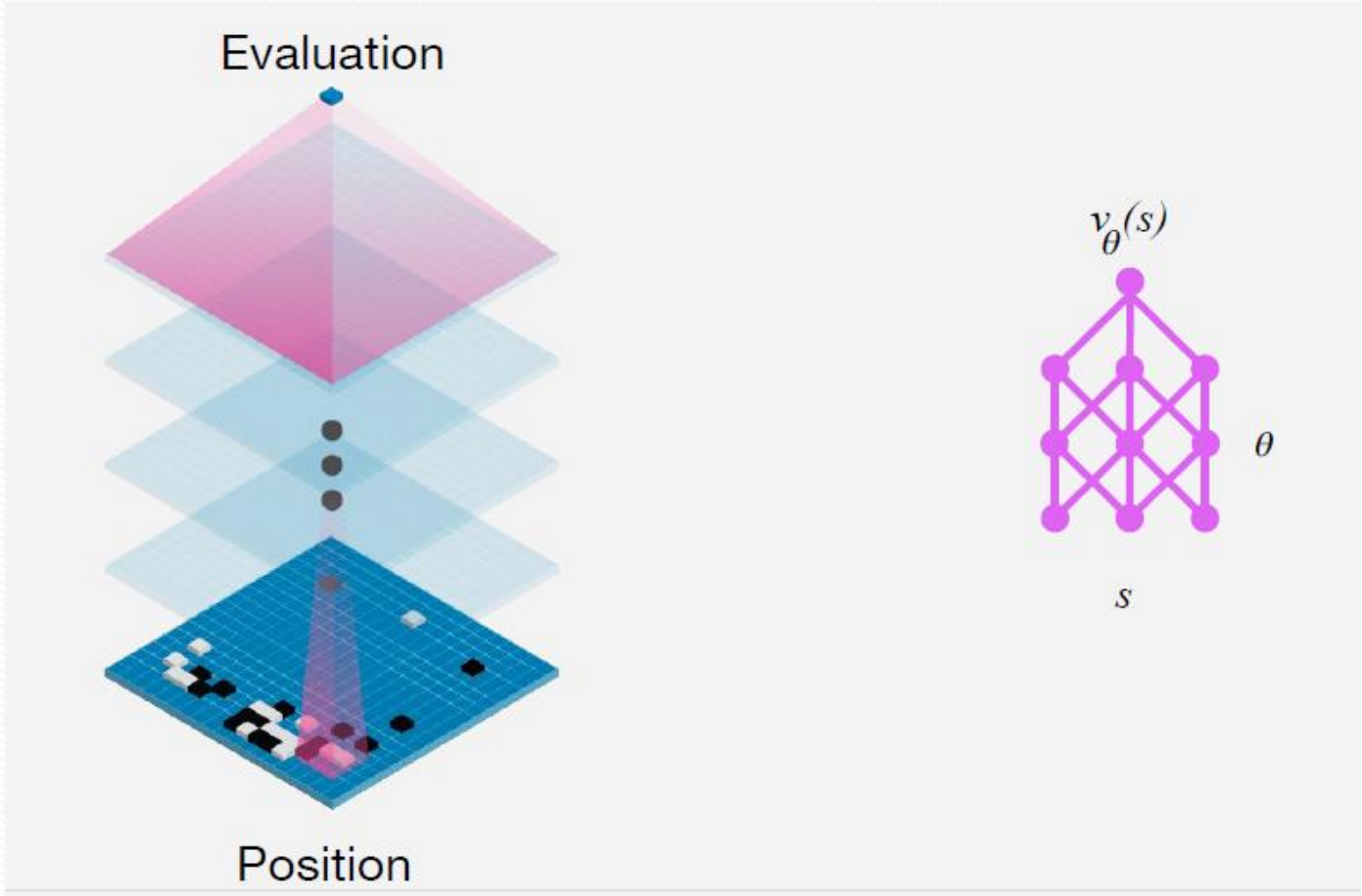
Treat board as an image: use (residual) convolutional neural network



Pictures from DeepMind

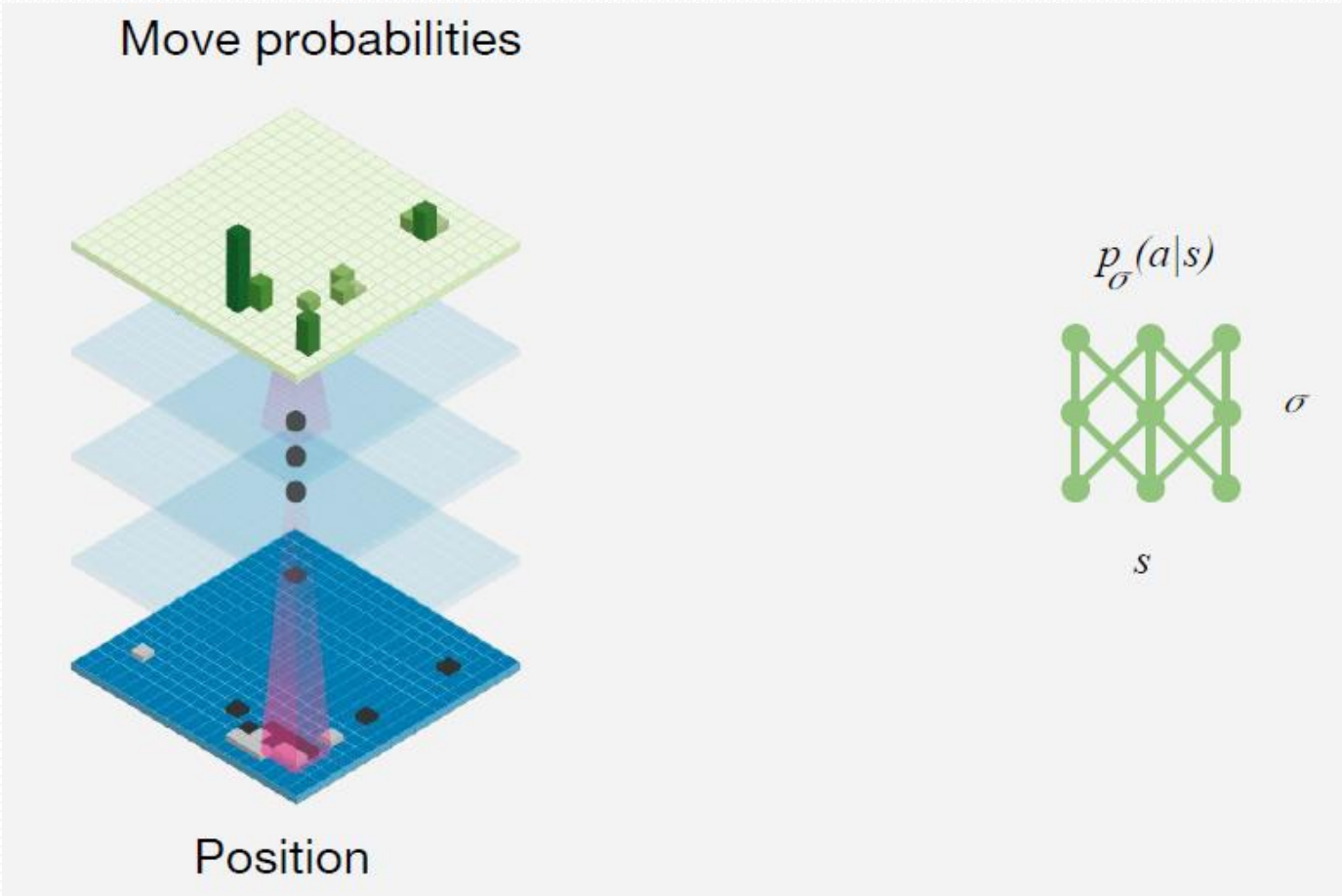
Value network: predict state value

- Value network: a DNN to predict (estimate/approximate) the state value V(s)



Policy network: decide next move given current state

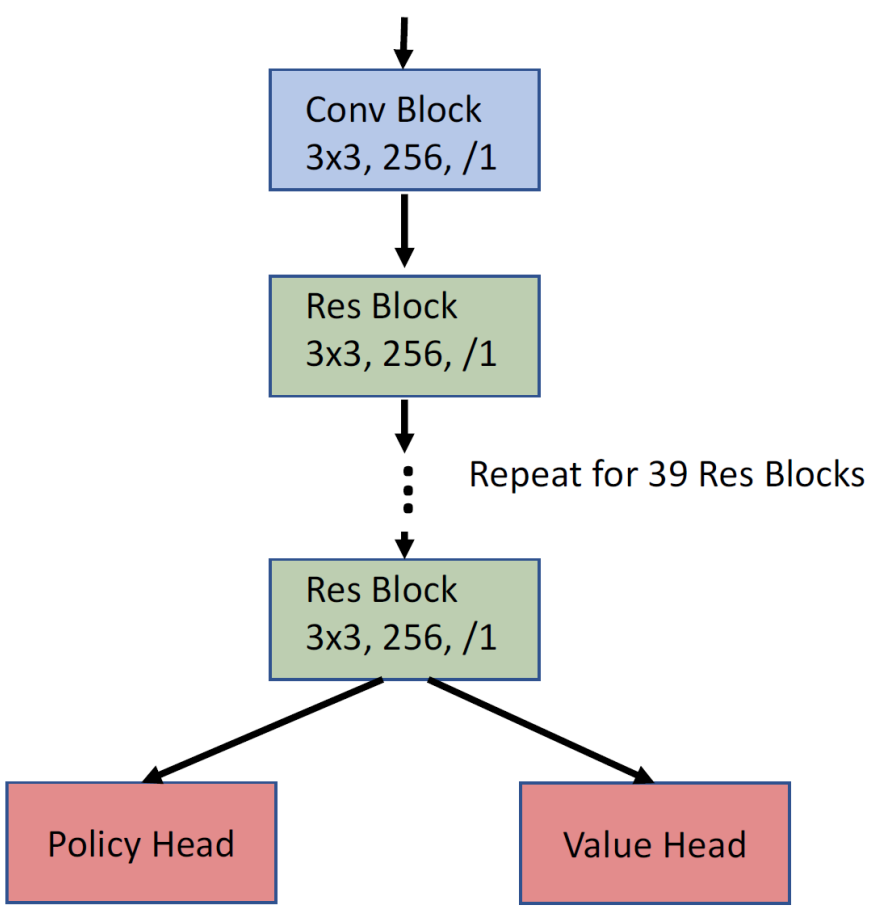
- Policy network: a DNN outputs probability over all possible next moves.



AlphaZero Network

Policy branch: softmax layer with 19*19 neurons

Value branch: a single neuron with tanh activation



AlphaZero Training

- Repeat:
  - Reset example buffer
  - Play Game (N times)
  - Update NN parameters θ using the game-play example buffer

- *Play Game*:
  - Repeat Until Win or Lose:
    - From current state *s*, perform Monte Carlo Tree Search (MCTS)
    - Estimate move probabilities π by MCTS
    - Record $(s, \pi, \cdot)$ as an example
    - Randomly draw next move from π
  - Let *z* be the game outcome (+1 or -1)
  - Collect the set of examples from this game $\{(s, \pi, z)\}$ and add them to the example buffer

- *Update*:
  - Sample from example buffer
  - Train DNN on the sample to update parameters θ