Carnegie Mellon
School of Computer Science

Deep Reinforcement Learning and Control

Deep Q Learning

CMU 10-403

Katerina Fragkiadaki

Used Materials

- **Disclaimer**: Much of the material and slides for this lecture were borrowed from Russ Salakhutdinov, Rich Sutton's class and David Silver's class on Reinforcement Learning.

Optimal Value Function

- An optimal value function is the maximum achievable value

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

- Once we have Q^*, the agent can act optimally

$$\pi^*(s) = \operatorname*{argmax}_{a} Q^*(s,a)$$

- Formally, optimal values decompose into a Bellman equation

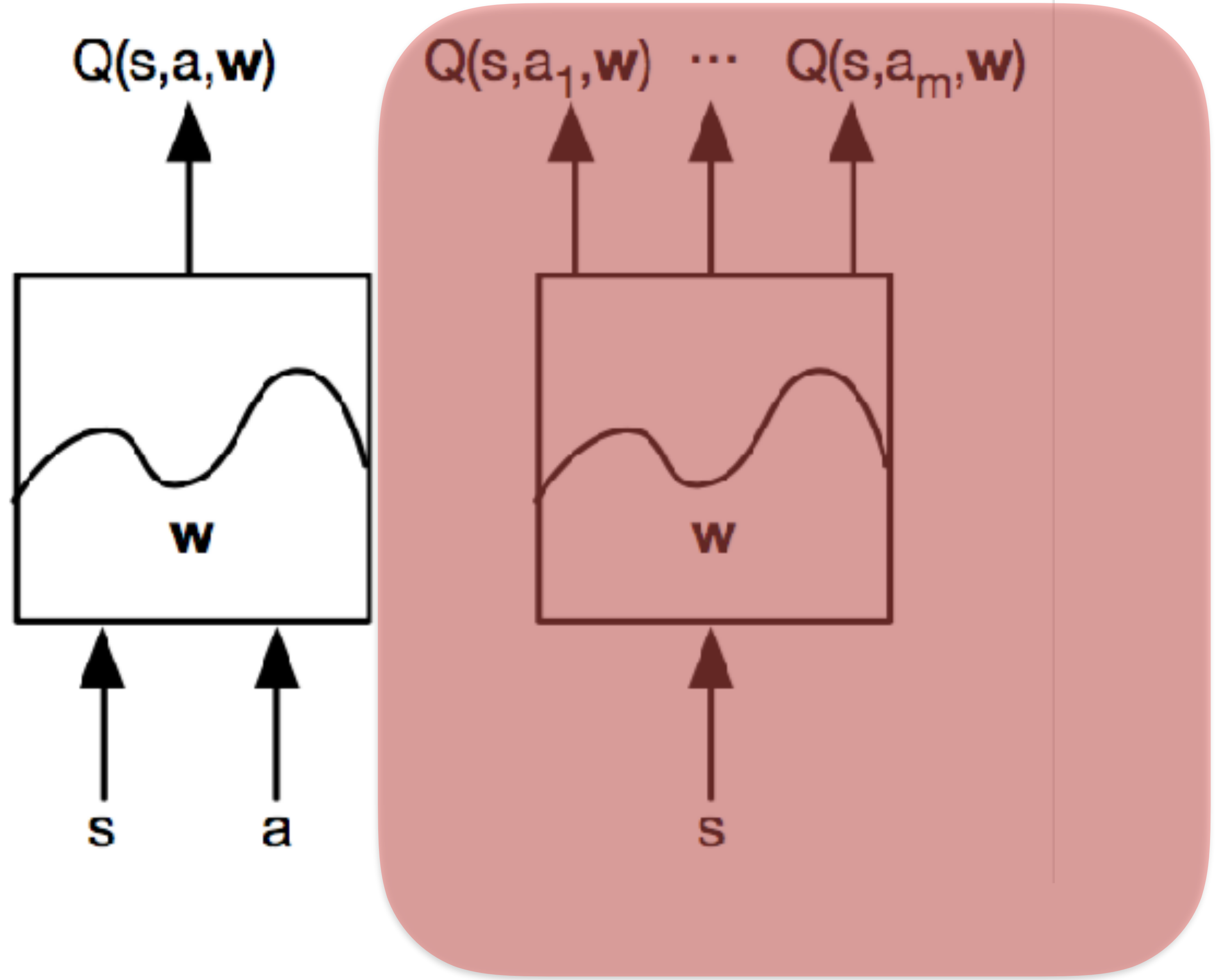
$$Q^*(s,a) = \mathbb{E}_{s'}\left[r + \gamma \max_{a'} Q^*(s',a') \mid s,a\right]$$

Deep Q-Networks (DQNs)

- Represent action-state value function by Q-network with weights w

$$Q(s, a, \mathbf{w}) \approx Q^*(s, a)$$

When would this be preferred?



Q-Learning with FA

- Optimal Q-values should obey Bellman equation

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

- Treat right-hand $r + \gamma \max_{a'} Q(s',a',\mathbf{w})$ as a target

- Minimize MSE loss by stochastic gradient descent

$$l = \left(r + \gamma \max_{a} Q(s',a',\mathbf{w}) - Q(s,a,\mathbf{w})\right)^2$$

- Remember VFA lecture: Minimize mean-squared error between the true action-value function $q_\pi(S,A)$ and the approximate Q function:

$$J(\mathbf{w}) = \mathbb{E}_\pi\left[(q_\pi(S,A) - \hat{q}(S,A,\mathbf{w}))^2\right]$$

Q-Learning with FA

- Minimize MSE loss by stochastic gradient descent

$$l = \left(r + \gamma \max_{a} Q(s', a', \mathbf{w}) - Q(s, a, \mathbf{w}) \right)^2$$

Q-Learning: Off-Policy TD Control

- One-step Q-learning:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\Big]$$

Initialize $Q(s, a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
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

Q-Learning with FA

- Minimize MSE loss by stochastic gradient descent

$$l = \left(r + \gamma \max_{a} Q(s', a', \mathbf{w}) - Q(s, a, \mathbf{w}) \right)^2$$

- Converges to Q^* using table lookup representation
- But diverges using neural networks due to:
  1. Correlations between samples
  2. Non-stationary targets

Q-Learning

- Minimize MSE loss by stochastic gradient descent

$$l = \left(r + \gamma \max_{a} Q(s', a', \mathbf{w}) - Q(s, a, \mathbf{w}) \right)^2$$

- Converges to Q^* using table lookup representation

- But diverges using neural networks due to:
  1. Correlations between samples
  2. Non-stationary targets

Solution to both problems in DQN:

Playing Atari with Deep Reinforcement Learning

Volodymyr Mnih Koray Kavukcuoglu David Silver Alex Graves Ioannis Antonoglou

Daan Wierstra Martin Riedmiller

DeepMind Technologies

DQN

- To remove correlations, build data-set from agent's own experience

s_1, a_1, r_2, s_2
s_2, a_2, r_3, s_3
s_3, a_3, r_4, s_4
...
$s_t, a_t, r_{t+1}, s_{t+1}$

$$\rightarrow \quad s, a, r, s'$$

- Sample experiences from data-set and apply update

$$l = \left(r + \gamma \max_{a'} Q(s', a', \mathbf{w}^-) - Q(s, a, \mathbf{w}) \right)^2$$

- To deal with non-stationarity, target parameters w− are held fixed

Experience Replay

- Given experience consisting of ⟨state, value⟩, or <state, action/value> pairs

$$\mathcal{D} = \{\langle s_1, v_1^\pi \rangle, \langle s_2, v_2^\pi \rangle, \ldots, \langle s_T, v_T^\pi \rangle\}$$

- Repeat
  - Sample state, value from experience

$$\langle s, v^\pi \rangle \sim \mathcal{D}$$

  - Apply stochastic gradient descent update

$$\Delta \mathbf{w} = \alpha(v^\pi - \hat{v}(s, \mathbf{w}))\nabla_{\mathbf{w}} \hat{v}(s, \mathbf{w})$$

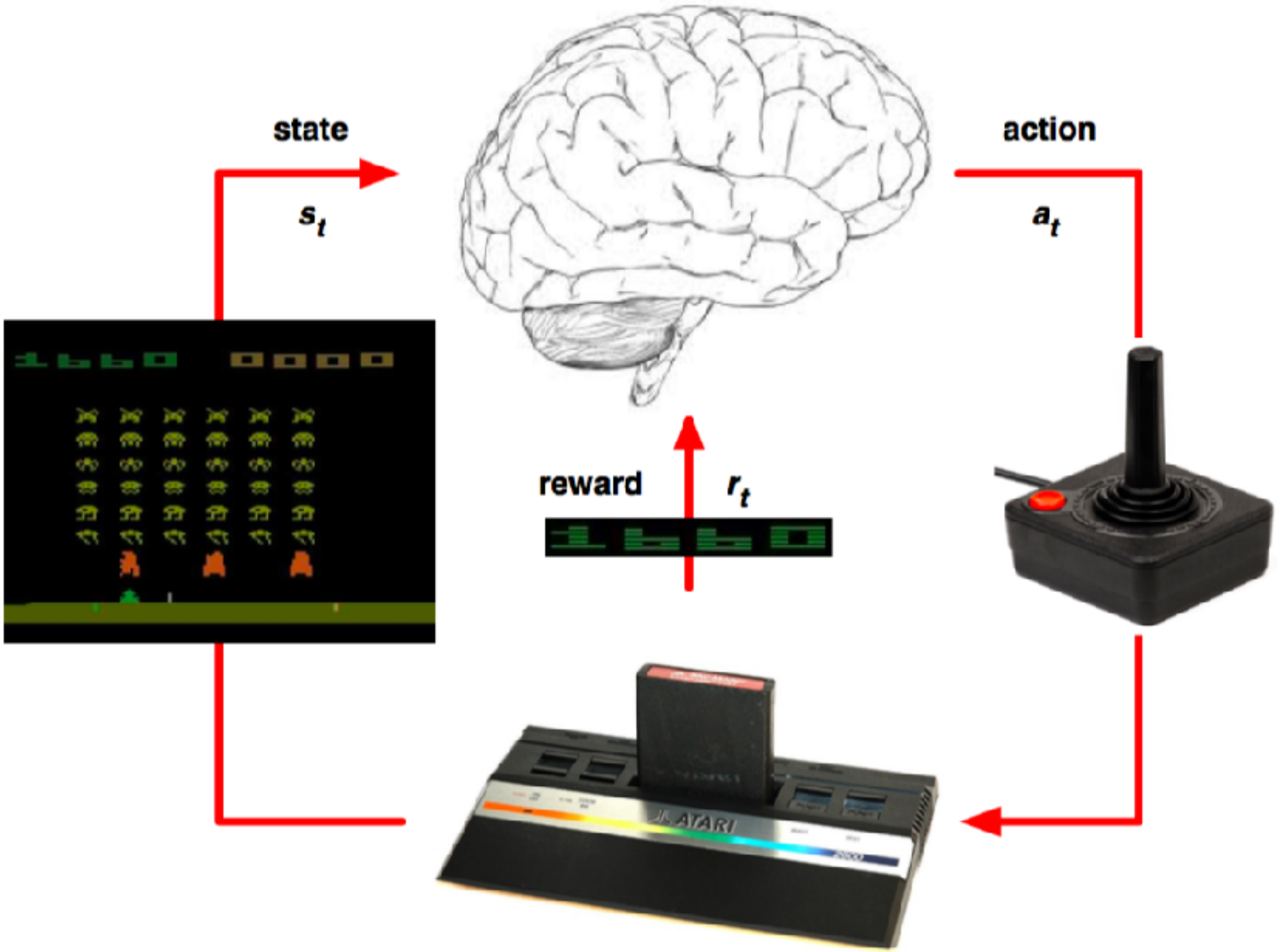
DQNs: Experience Replay

- DQN uses experience replay and fixed Q-targets
- Store transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in replay memory D
- Sample random mini-batch of transitions (s,a,r,s′) from D
- Compute Q-learning targets w.r.t. old, fixed parameters w−
- Optimize MSE between Q-network and Q-learning targets

$$\mathcal{L}_i(w_i) = \mathbb{E}_{s,a,r,s' \sim \mathcal{D}_i}\left[\left(\underbrace{r + \gamma \max_{a'} Q(s', a'; w_i^-)}_{\text{Q-learning target}} - \underbrace{Q(s, a; w_i)}_{\text{Q-network}}\right)^2\right]$$

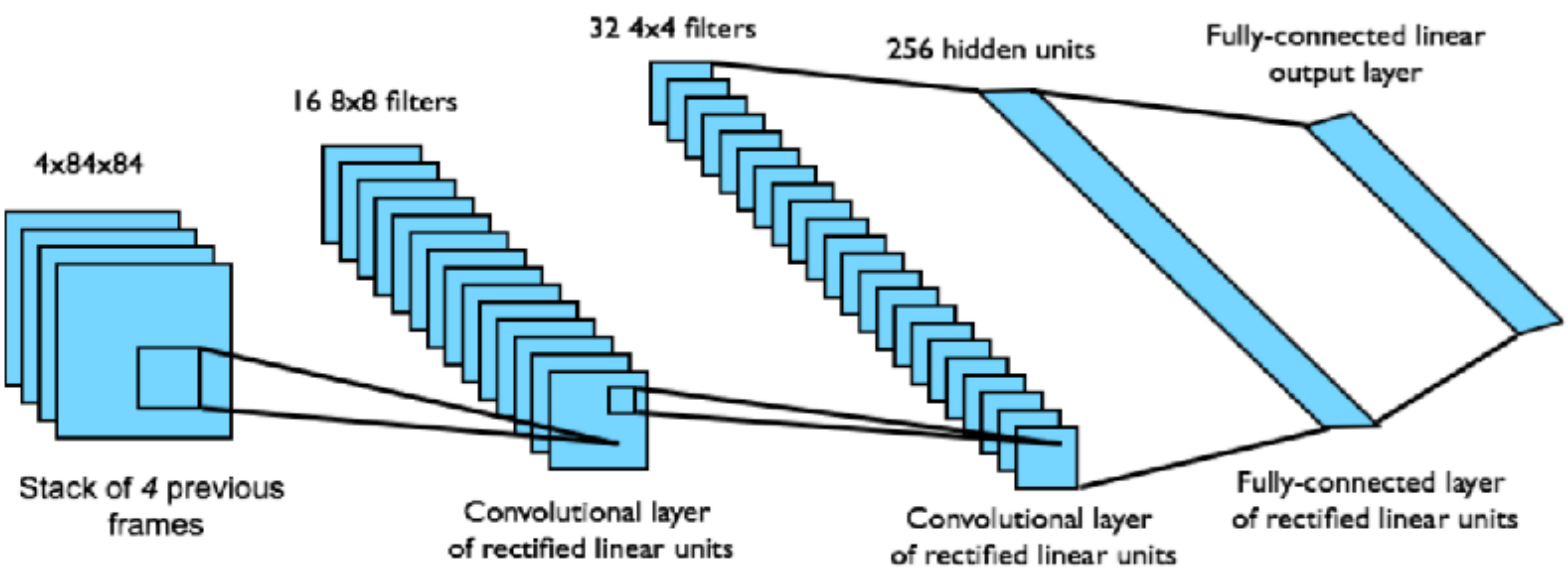
- Use stochastic gradient descent

DQNs in Atari



DQNs in Atari

- End-to-end learning of values Q(s,a) from pixels
- Input observation is stack of raw pixels from last 4 frames
- Output is Q(s,a) for 18 joystick/button positions
- Reward is change in score for that step

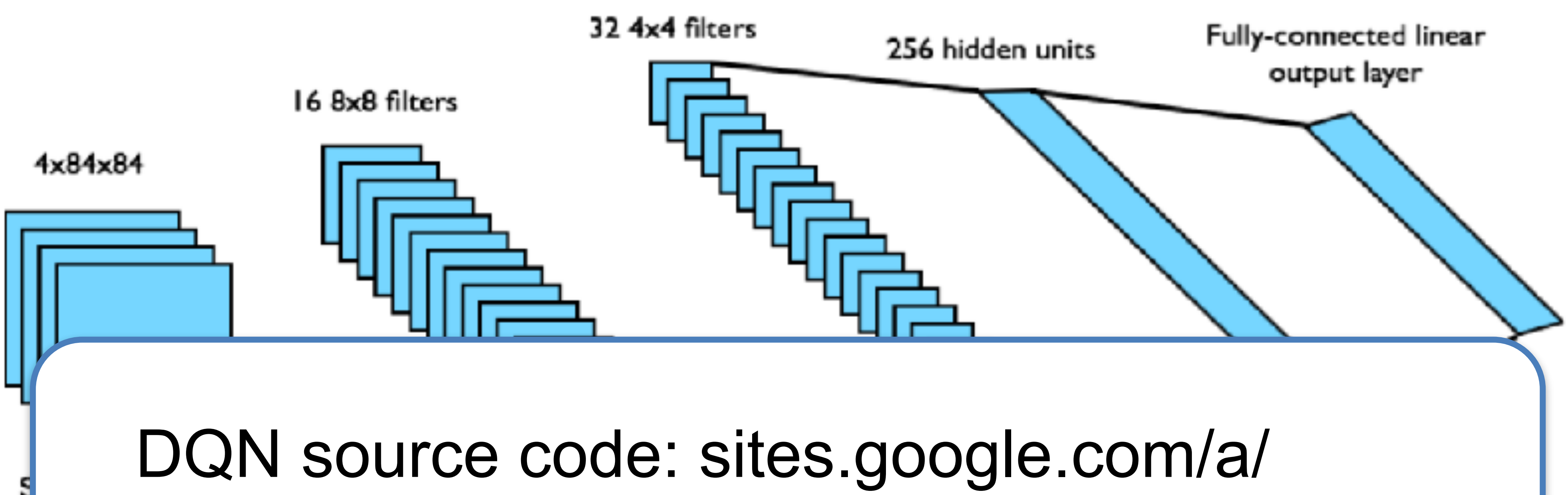


- Network architecture and hyperparameters fixed across all games

Mnih et.al., Nature, 2014

DQNs in Atari

- End-to-end learning of values Q(s,a) from pixels s
- Input observation is stack of raw pixels from last 4 frames
- Output is Q(s,a) for 18 joystick/button positions
- Reward is change in score for that step



DQN source code: sites.google.com/a/deepmind.com/dqn/

- Netw[illegible]s

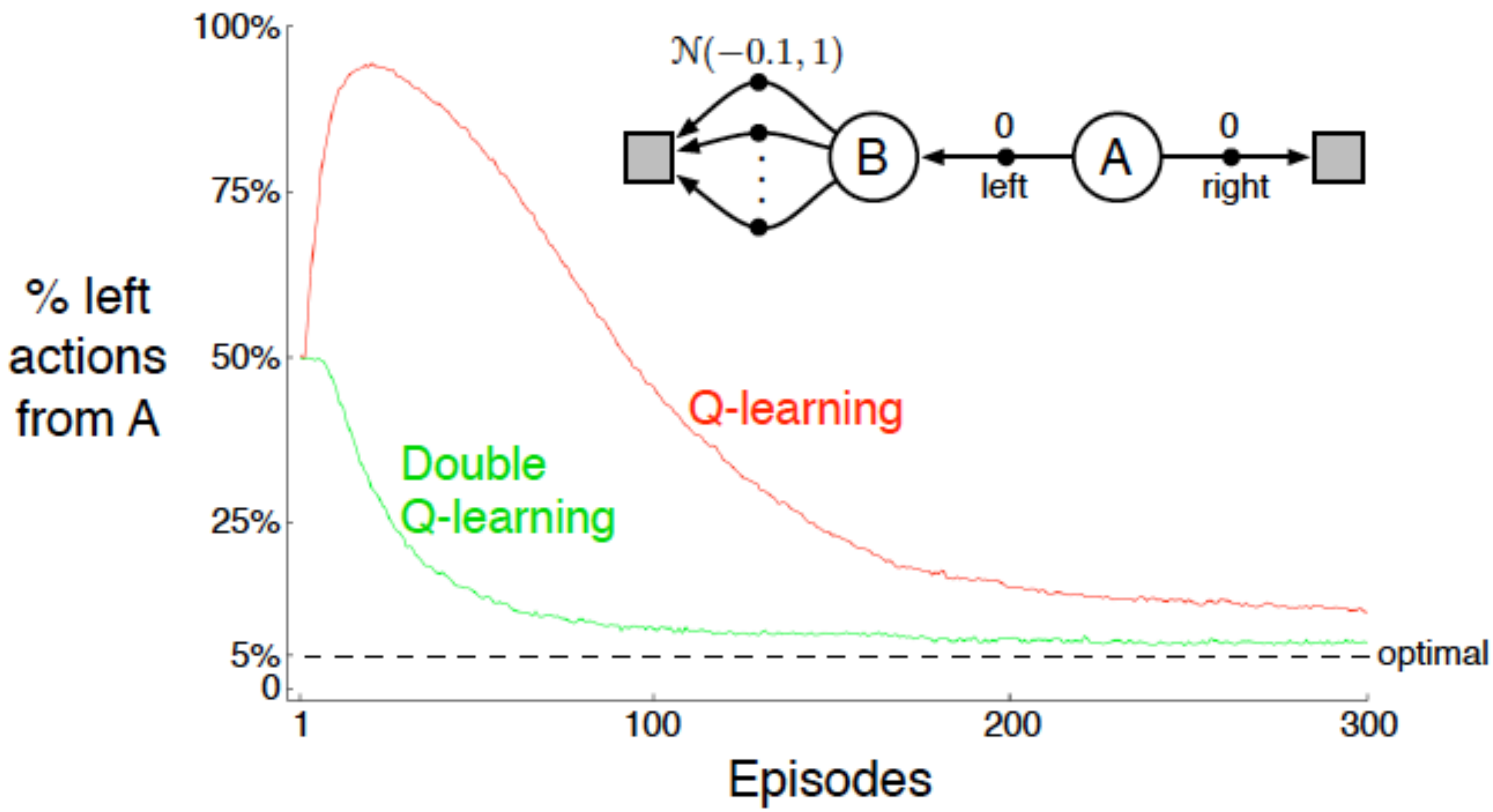
Mnih et.al., Nature, 2014

Extensions

- Double Q-learning for fighting maximization bias
- Prioritized experience replay
- Dueling Q networks
- Multistep returns
- Value distribution
- Stochastic nets for explorations instead of \epsilon-greedy

Maximization Bias

- We often need to maximize over our value estimates. The estimated maxima suffer from maximization bias
- Consider a state for which all ground-truth q(s,a)=0. Our estimates Q(s,a) are uncertain, some are positive and some negative. Q(s,argmax_a(Q(s,a)) is positive while q(s,argmax_a(q(s,a))=0.



Double Q-Learning

- Train 2 action-value functions, Q_1 and Q_2
- Do Q-learning on both, but
  - never on the same time steps (Q_1 and Q_2 are independent)
  - pick Q_1 or Q_2 at random to be updated on each step

- If updating Q_1, use Q_2 for the value of the next state:

$$Q_1(S_t, A_t) \leftarrow Q_1(S_t, A_t) + \\ + \alpha\Big(R_{t+1} + Q_2\big(S_{t+1}, \arg\max_a Q_1(S_{t+1}, a)\big) - Q_1(S_t, A_t)\Big)$$

- Action selections are ε-greedy with respect to the sum of Q_1 and Q_2

Double Q-Learning

- Train 2 action-value functions, Q_1 and Q_2
- Do Q-learning on both, but
  - never on the same time steps (Q_1 and Q_2 are independent)
  - pick Q_1 or Q_2 at random to be updated on each step

- If updating Q_1, use Q_2 for the value of the next state:

$$Q_1(S_t, A_t) \leftarrow Q_1(S_t, A_t) + \\ + \alpha\Big(R_{t+1} + Q_2\big(S_{t+1}, \arg\max_a Q_1(S_{t+1}, a)\big) - Q_1(S_t, A_t)\Big)$$

- Action selections are ε-greedy with respect to the sum of Q_1 and Q_2

Double Tabular Q-Learning

Initialize $Q_1(s,a)$ and $Q_2(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily
Initialize $Q_1(\textit{terminal-state}, \cdot) = Q_2(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
    Initialize S
    Repeat (for each step of episode):
        Choose A from S using policy derived from Q_1 and Q_2 (e.g., ε-greedy in $Q_1 + Q_2$)
        Take action A, observe R, S'
        With 0.5 probabilility:
$$Q_1(S,A) \leftarrow Q_1(S,A) + \alpha\Big(R + \gamma Q_2\big(S', \arg\max_a Q_1(S',a)\big) - Q_1(S,A)\Big)$$
        else:
$$Q_2(S,A) \leftarrow Q_2(S,A) + \alpha\Big(R + \gamma Q_1\big(S', \arg\max_a Q_2(S',a)\big) - Q_2(S,A)\Big)$$
        $S \leftarrow S'$;
    until S is terminal

Hado van Hasselt 2010

Double Deep Q-Learning

- Current Q-network w is used to select actions
- Older Q-network w− is used to evaluate actions

$$I = \Bigg(r + \gamma \overbrace{Q(s', \underbrace{\operatorname*{argmax}_{a'} Q(s', a', \mathbf{w})}_{\text{Action selection: w}}, \mathbf{w}^-)}^{\text{Action evaluation: w−}} - Q(s, a, \mathbf{w})\Bigg)^2$$

van Hasselt, Guez, Silver, 2015

Prioritized Replay

- Weight experience according to ``surprise" (or error)
- Store experience in priority queue according to DQN error

$$\underbrace{\left| r + \gamma \max_{a'} Q(s', a', \mathbf{w}^-) - Q(s, a, w) \right|}$$

- Stochastic Prioritization

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$$

p_i is proportional to DQN error

- α determines how much prioritization is used, with α = 0 corresponding to the uniform case.

Schaul, Quan, Antonoglou, Silver, ICLR 2016

Multistep Returns

- Truncated n-step return from a state s_t: $$R_t^{(n)} = \sum_{k=0}^{n-1} \gamma_t^{(k)} R_{t+k+1}$$

- Multistep Q-learning update rule:

$$I = \left(R_t^{(n)} + \gamma_t^{(n)} \max_{a'} Q(S_{t+n}, a', \mathbf{w}) - Q(s, a, \mathbf{w})\right)^2$$

- Singlestep Q-learning update rule:

$$I = \left(r + \gamma \max_{a} Q(s', a', \mathbf{w}) - Q(s, a, \mathbf{w})\right)^2$$

Rainbow: Combining Improvements in Deep Reinforcement Learning

Matteo Hessel DeepMind

Joseph Modayil DeepMind

Hado van Hasselt DeepMind

Tom Schaul DeepMind

Georg Ostrovski DeepMind

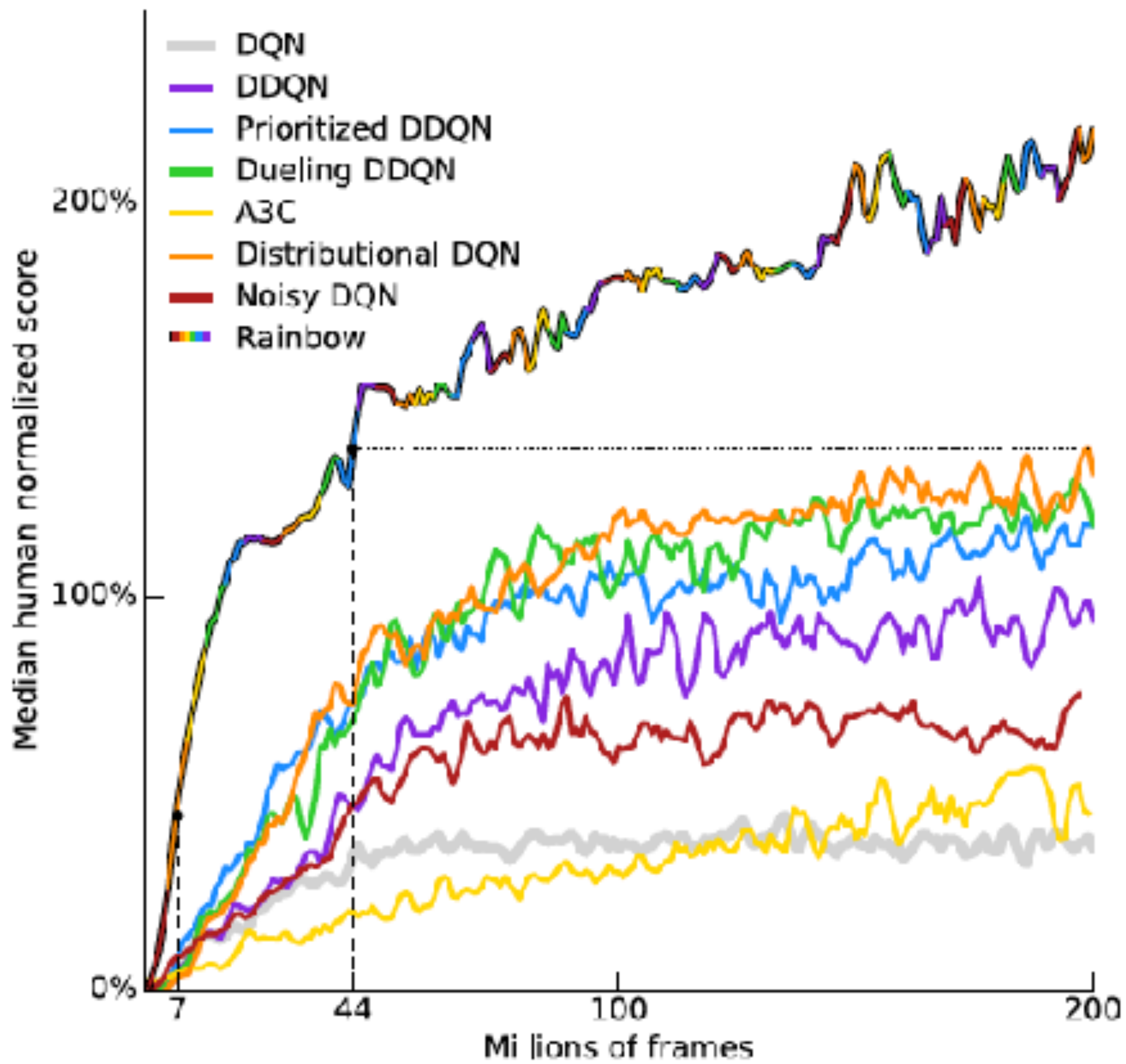
Will Dabney DeepMind

Dan Horgan DeepMind

Bilal Piot DeepMind

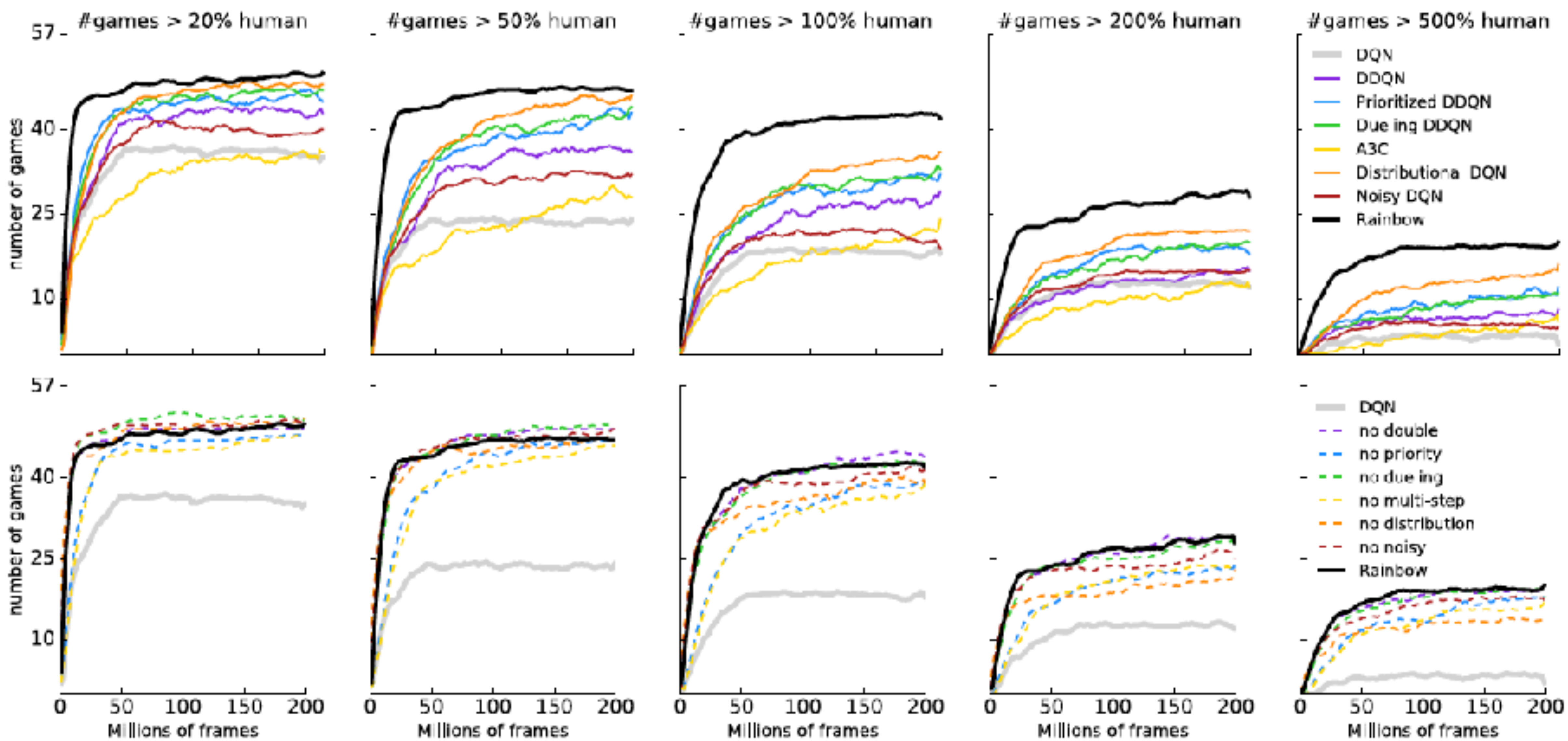
Mohammad Azar DeepMind

David Silver DeepMind



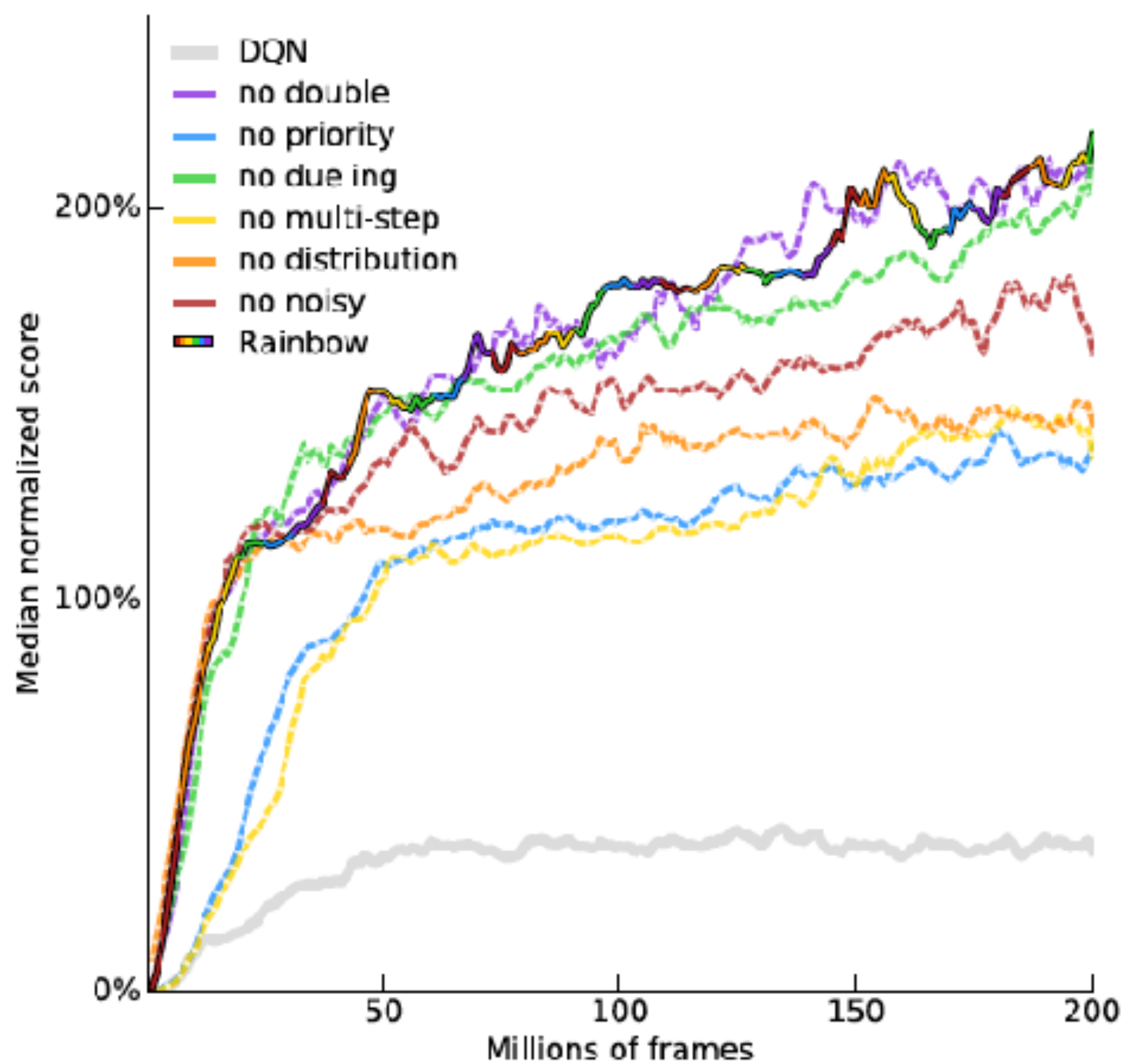
Rainbow: Combining Improvements in Deep Reinforcement Learning

Matteo Hessel DeepMind
Joseph Modayil DeepMind
Hado van Hasselt DeepMind
Tom Schaul DeepMind
Georg Ostrovski DeepMind
Will Dabney DeepMind
Dan Horgan DeepMind
Bilal Piot DeepMind
Mohammad Azar DeepMind
David Silver DeepMind



Rainbow: Combining Improvements in Deep Reinforcement Learning

Matteo Hessel DeepMind
Joseph Modayil DeepMind
Hado van Hasselt DeepMind
Tom Schaul DeepMind
Georg Ostrovski DeepMind
Will Dabney DeepMind
Dan Horgan DeepMind
Bilal Piot DeepMind
Mohammad Azar DeepMind
David Silver DeepMind



Question

- Imagine we have access to the internal state of the Atari simulator. Would online planning (e.g., using MCTS), outperform the trained DQN policy?

Question

- Imagine we have access to the internal state of the Atari simulator. Would online planning (e.g., using MCTS), outperform the trained DQN policy?
    - With enough resources, yes.
    - Resources = number of simulations (rollouts) and maximum allowed depth of those rollouts.
    - There is always an amount of resources when a vanilla MCTS (not assisted by any deep nets) will outperform the learned with RL policy.

Question

- Then why we do not use MCTS with online planning to play Atari instead of learning a policy?

Question

- Then why we do not use MCTS with online planning to play Atari instead of learning a policy?

  - Because using vanilla (not assisted by any deep nets) MCTS is very very slow, definitely very far away from real time game playing that humans are capable of.

Question

- If we used MCTS during training time to suggest actions using online planning, and we would try to mimic the output of the planner, would we do better than DQN that learns a policy without using any model while playing in real time?

Question

- If we used MCTS during training time to suggest actions using online planning, and we would try to mimic the output of the planner, would we do better than DQN that learns a policy without using any model while playing in real time?

    - That would be a very sensible approach!

Deep Learning for Real-Time Atari Game Play Using Offline Monte-Carlo Tree Search Planning

Xiaoxiao Guo
Computer Science and Eng.
University of Michigan
guoxiao@umich.edu

Satinder Singh
Computer Science and Eng.
University of Michigan
baveja@umich.edu

Honglak Lee
Computer Science and Eng.
University of Michigan
honglak@umich.edu

Richard Lewis
Department of Psychology
University of Michigan
rickl@umich.edu

Xiaoshi Wang
Computer Science and Eng.
University of Michigan
xiaoshiw@umich.edu

Offline MCTS to train online fast reactive policies

- **AlphaGo**: train policy and value networks at training time, combine them with MCTS at test time
- **AlphaGoZero**: train policy and value networks with MCTS in the training loop and at test time (same method used at train and test time)
- **Offline MCTS**: train policy and value networks with MCTS in the training loop, but at test time use the (reactive) policy network, without any lookahead planning.
  - Where does the benefit come from?

Revision: Monte-Carlo Tree Search

1. **Selection**
   - Used for nodes we have seen before
   - Pick according to UCB
2. **Expansion**
   - Used when we reach the frontier
   - Add one node per playout
3. **Simulation**
   - Used beyond the search frontier
   - Don't bother with UCB, just play randomly
4. **Backpropagation**
   - After reaching a terminal node
   - Update value and visits for states expanded in selection and expansion

Bandit based Monte-Carlo Planning, Kocsis and Szepesvari, 2006

Upper-Confidence Bound

Sample actions according to the following score:

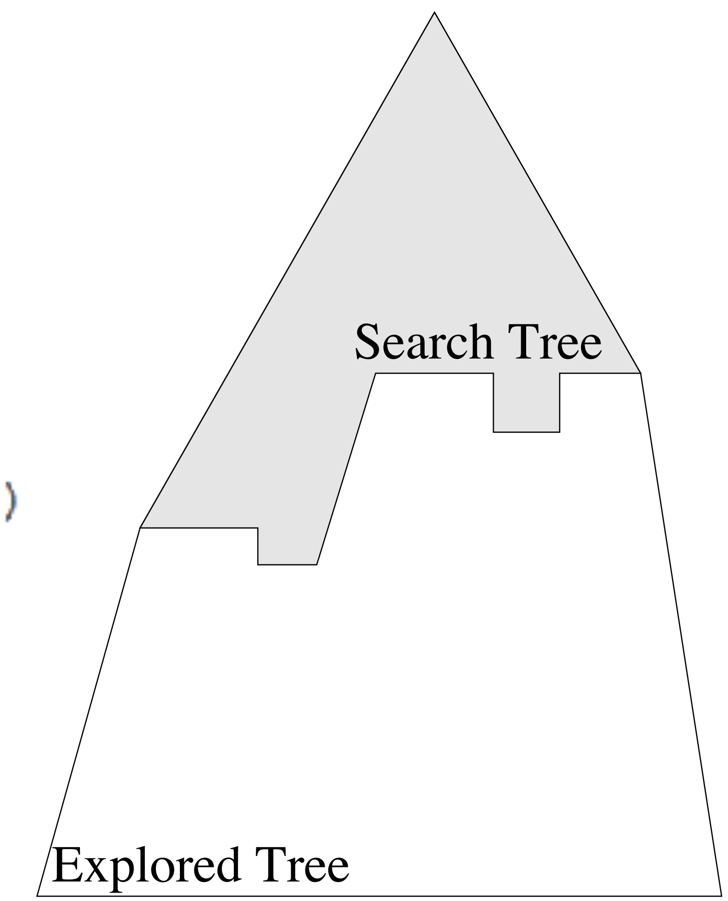
$$\underbrace{v_i}_{\text{value estimate}} + \underbrace{C}_{\text{tunable parameter}} \times \sqrt{\frac{\ln(\overbrace{N}^{\text{parent node visits}})}{\underbrace{n_i}_{\text{number of visits}}}}$$

- score is decreasing in the number of visits (explore)
- score is increasing in a node's value (exploit)
- always tries every option once

Finite-time Analysis of the Multiarmed Bandit Problem, Auer, Cesa-Bianchi, Fischer, 2002

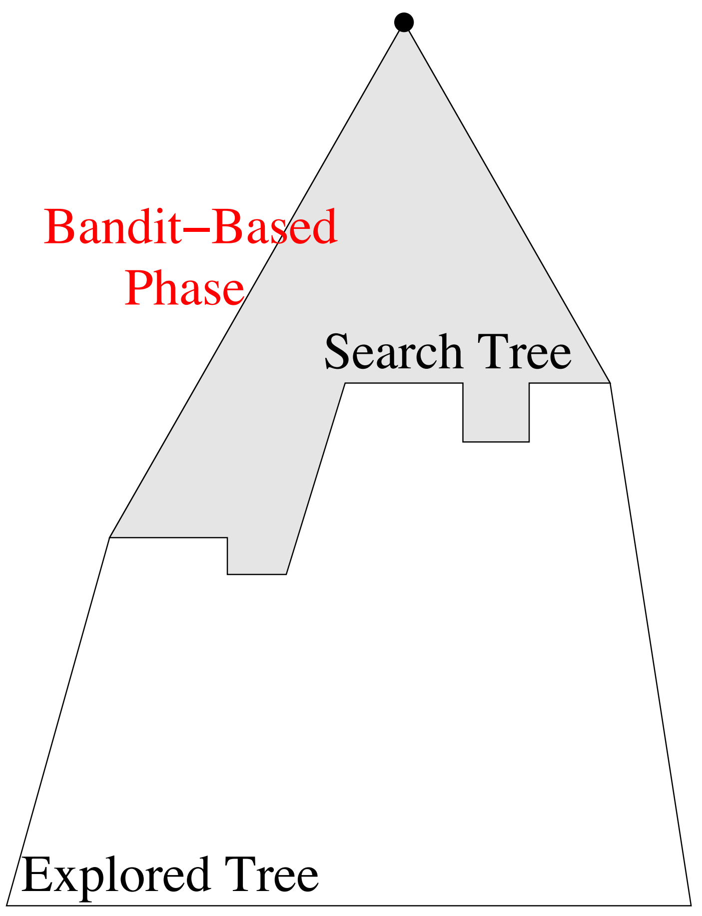
Basic MCTS pseudocode

```
function MCTS_sample(state)
    state.visits++
    if all children of state expanded:
        next_state = UCB_sample(state)
        winner = MCTS_sample(next_state)
    else:
        if some children of state expanded:
            next_state = expand(random unexpanded child)
        else:
            next_state = state
        winner = random_playout(next_state)
    update_value(state, winner)
```



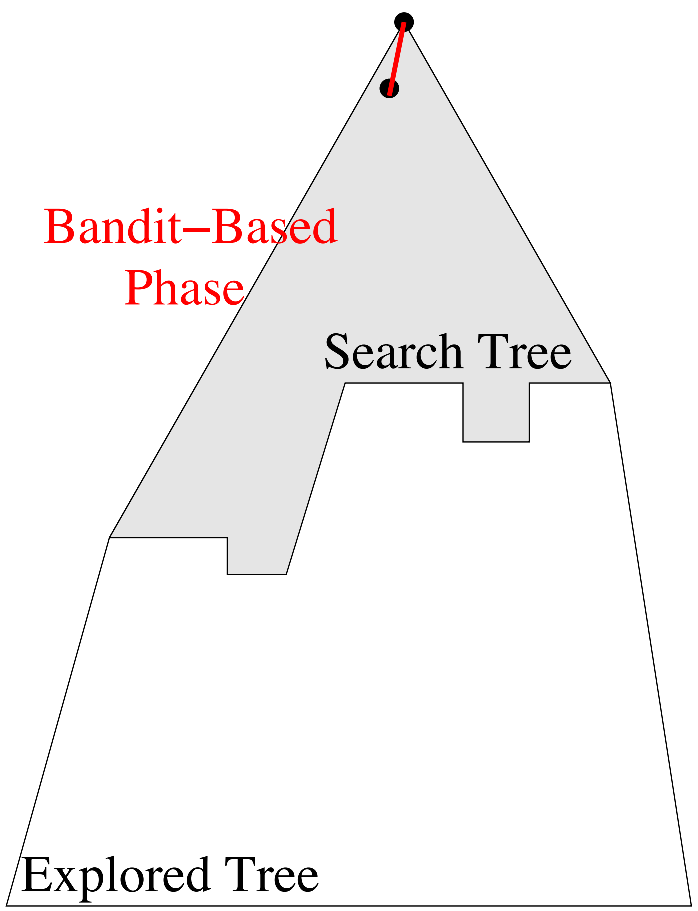
Basic MCTS pseudocode

```
function MCTS_sample(state)
    state.visits++
    if all children of state expanded:
        next_state = UCB_sample(state)
        winner = MCTS_sample(next_state)
    else:
        if some children of state expanded:
            next_state = expand(random unexpanded child)
        else:
            next_state = state
        winner = random_playout(next_state)
    update_value(state, winner)
```



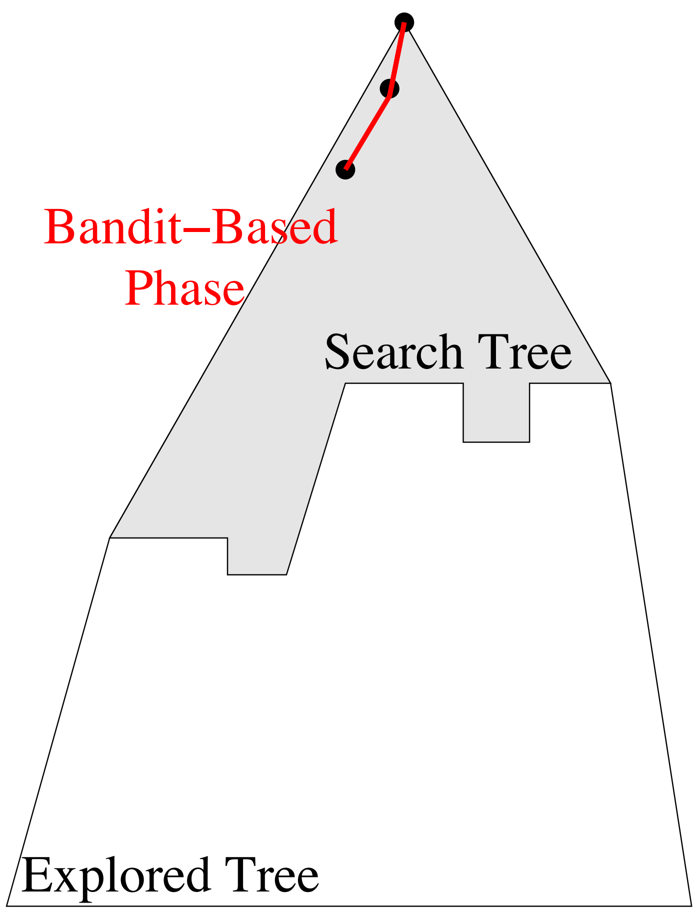
Basic MCTS pseudocode

```
function MCTS_sample(state)
    state.visits++
    if all children of state expanded:
        next_state = UCB_sample(state)
        winner = MCTS_sample(next_state)
    else:
        if some children of state expanded:
            next_state = expand(random unexpanded child)
        else:
            next_state = state
        winner = random_playout(next_state)
    update_value(state, winner)
```



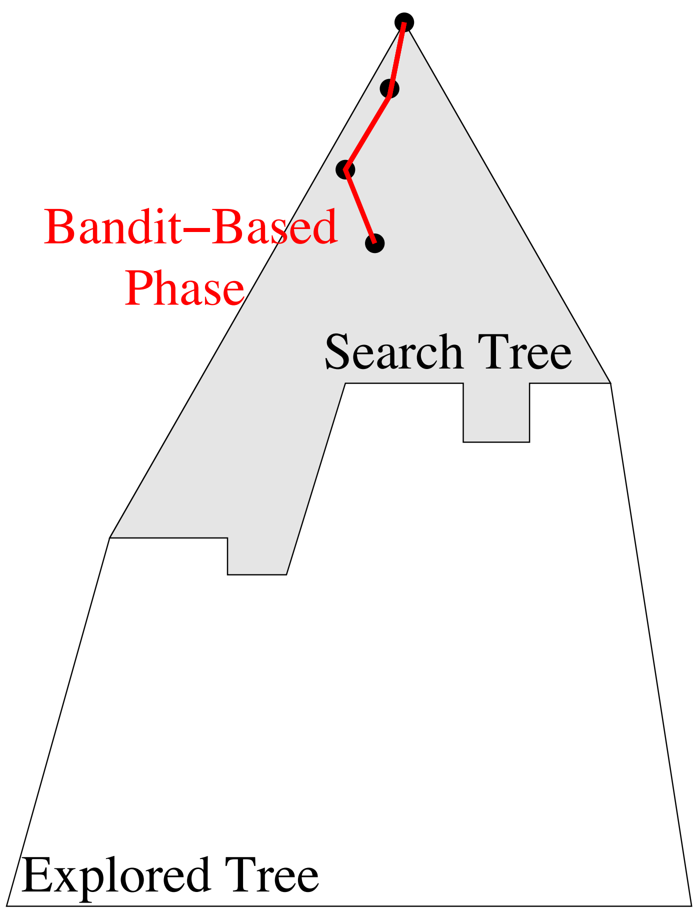
Basic MCTS pseudocode

```
function MCTS_sample(state)
    state.visits++
    if all children of state expanded:
        next_state = UCB_sample(state)
        winner = MCTS_sample(next_state)
    else:
        if some children of state expanded:
            next_state = expand(random unexpanded child)
        else:
            next_state = state
        winner = random_playout(next_state)
    update_value(state, winner)
```



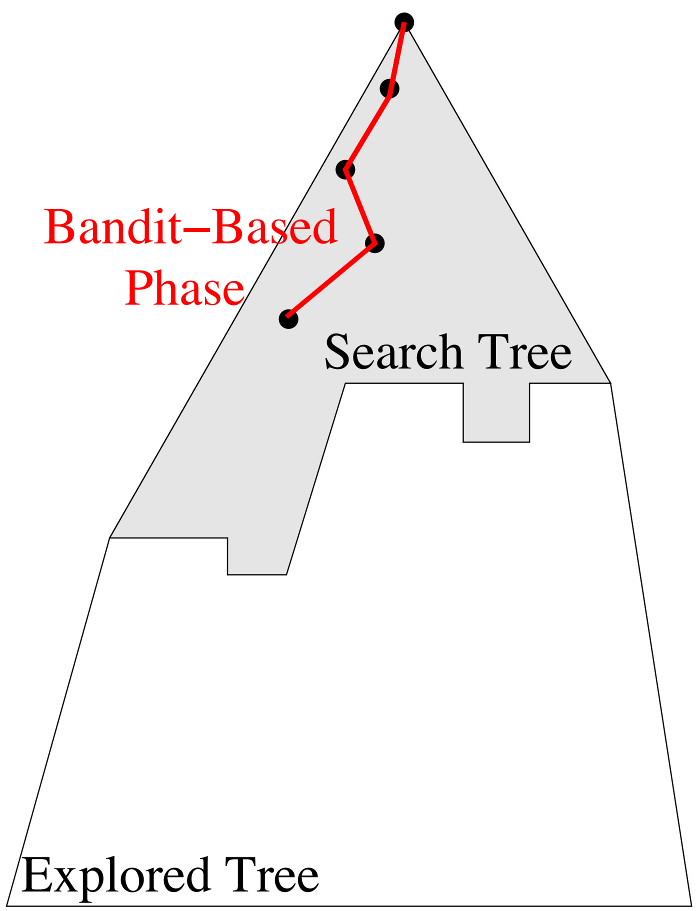
Basic MCTS pseudocode

```
function MCTS_sample(state)
    state.visits++
    if all children of state expanded:
        next_state = UCB_sample(state)
        winner = MCTS_sample(next_state)
    else:
        if some children of state expanded:
            next_state = expand(random unexpanded child)
        else:
            next_state = state
        winner = random_playout(next_state)
    update_value(state, winner)
```



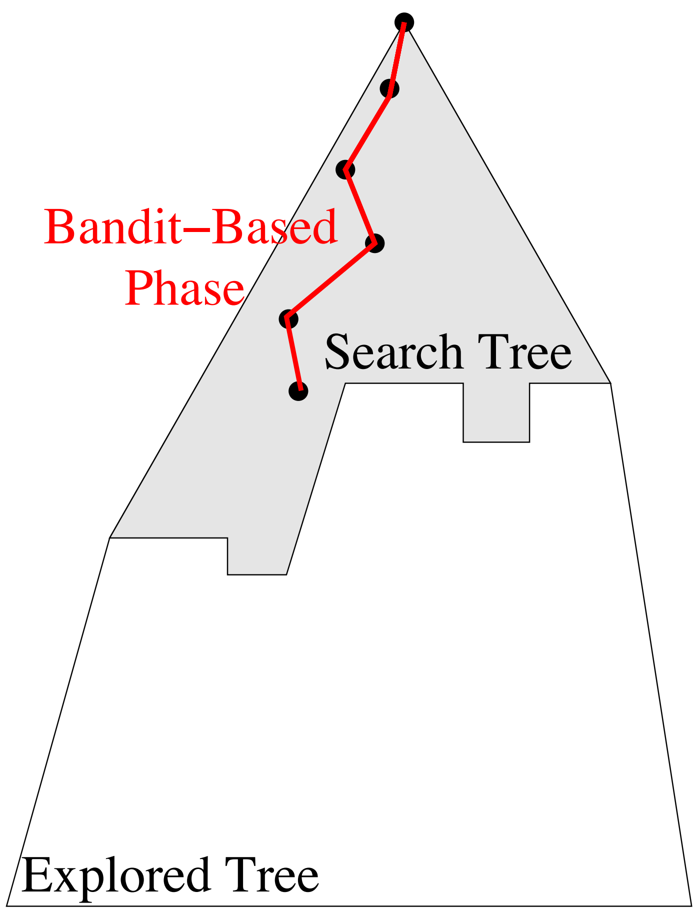
Basic MCTS pseudocode

```
function MCTS_sample(state)
    state.visits++
    if all children of state expanded:
        next_state = UCB_sample(state)
        winner = MCTS_sample(next_state)
    else:
        if some children of state expanded:
            next_state = expand(random unexpanded child)
        else:
            next_state = state
        winner = random_playout(next_state)
    update_value(state, winner)
```



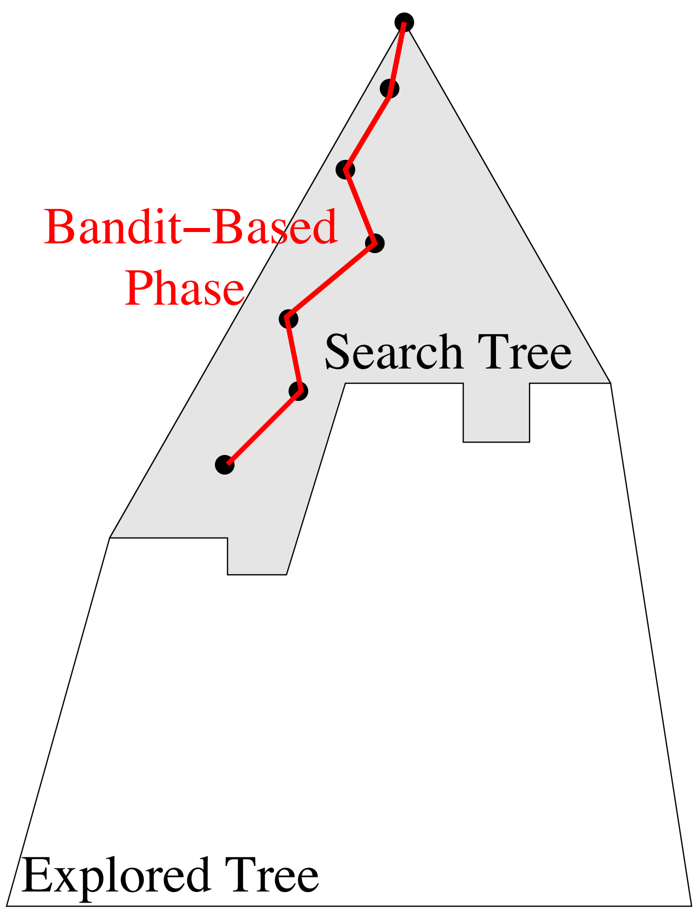
Basic MCTS pseudocode

```
function MCTS_sample(state)
    state.visits++
    if all children of state expanded:
        next_state = UCB_sample(state)
        winner = MCTS_sample(next_state)
    else:
        if some children of state expanded:
            next_state = expand(random unexpanded child)
        else:
            next_state = state
        winner = random_playout(next_state)
    update_value(state, winner)
```



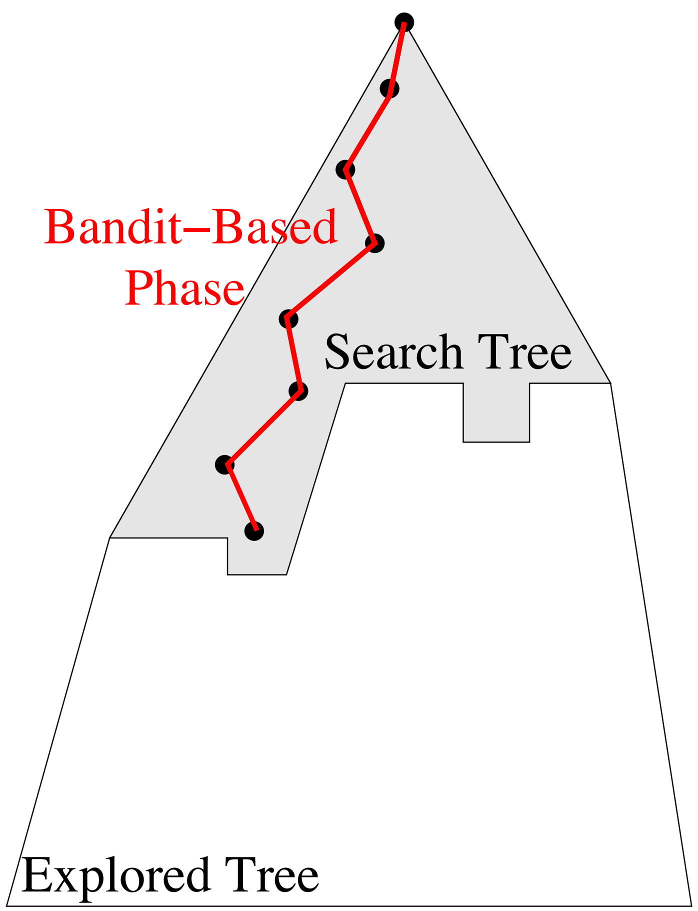
Basic MCTS pseudocode

```
function MCTS_sample(state)
    state.visits++
    if all children of state expanded:
        next_state = UCB_sample(state)
        winner = MCTS_sample(next_state)
    else:
        if some children of state expanded:
            next_state = expand(random unexpanded child)
        else:
            next_state = state
        winner = random_playout(next_state)
    update_value(state, winner)
```



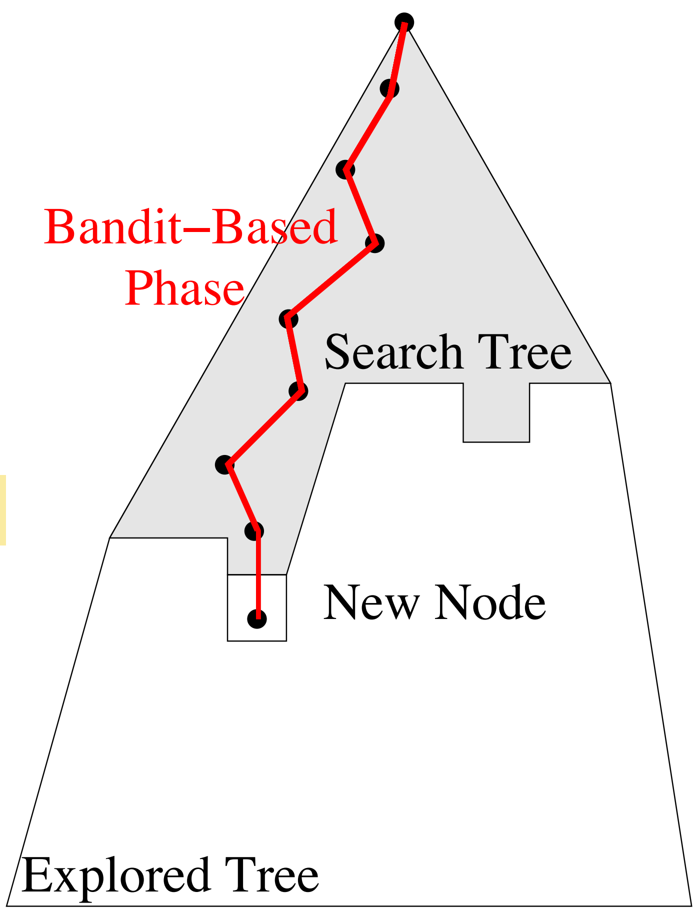
Basic MCTS pseudocode

```
function MCTS_sample(state)
    state.visits++
    if all children of state expanded:
        next_state = UCB_sample(state)
        winner = MCTS_sample(next_state)
    else:
        if some children of state expanded:
            next_state = expand(random unexpanded child)
        else:
            next_state = state
        winner = random_playout(next_state)
    update_value(state, winner)
```



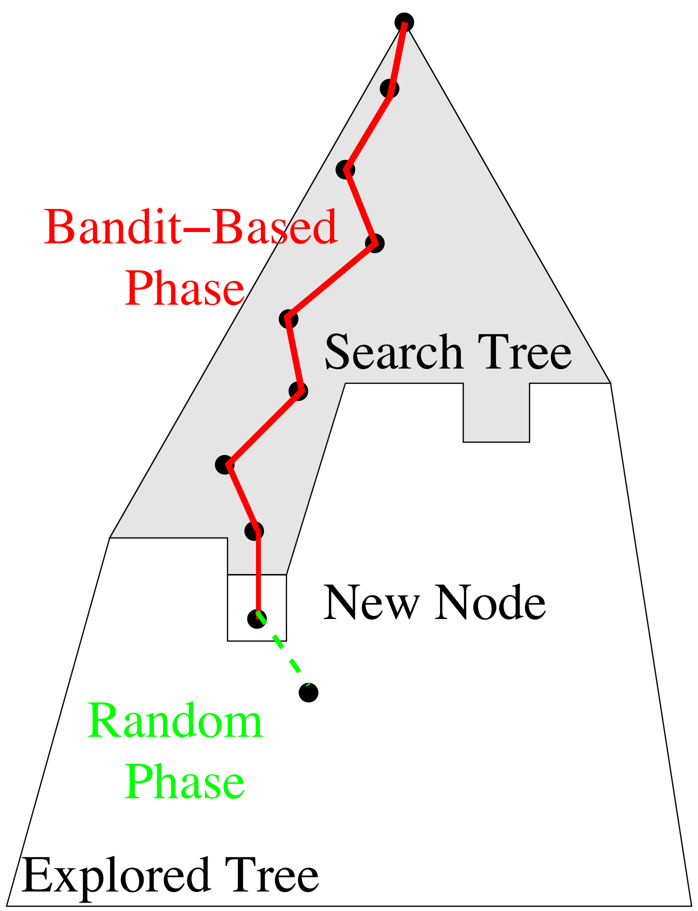
Basic MCTS pseudocode

```
function MCTS_sample(state)
    state.visits++
    if all children of state expanded:
        next_state = UCB_sample(state)
        winner = MCTS_sample(next_state)
    else:
        if some children of state expanded:
            next_state = expand(random unexpanded child)
        else:
            next_state = state
        winner = random_playout(next_state)
    update_value(state, winner)
```



Basic MCTS pseudocode

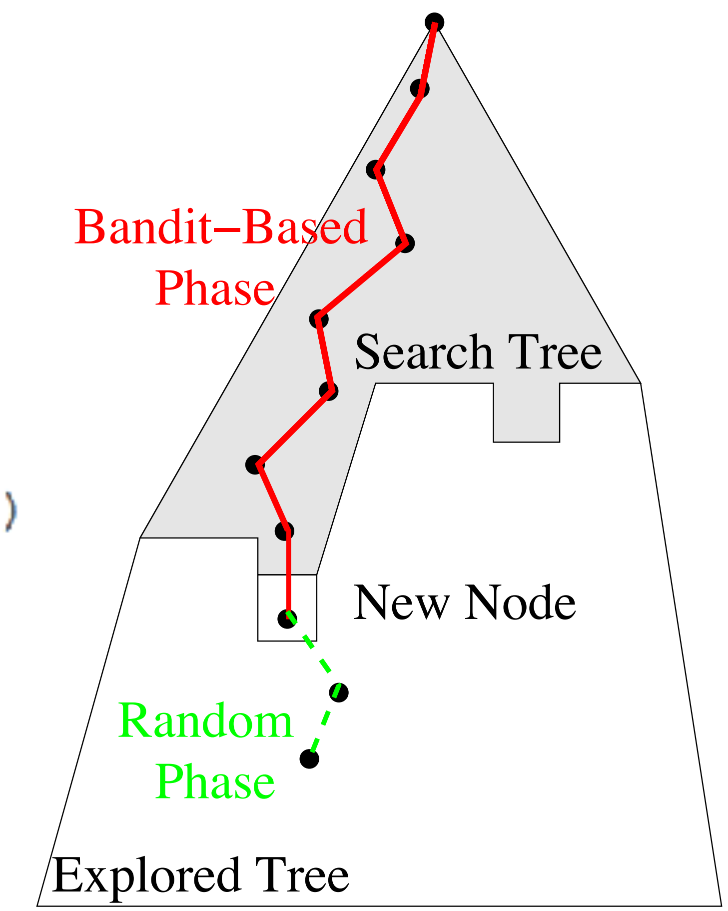
```
function MCTS_sample(state)
    state.visits++
    if all children of state expanded:
        next_state = UCB_sample(state)
        winner = MCTS_sample(next_state)
    else:
        if some children of state expanded:
            next_state = expand(random unexpanded child)
        else:
            next_state = state
        winner = random_playout(next_state)
    update_value(state, winner)
```




```
function random_playout(state):
    if is_terminal(state):
        return winner
    else: return random_playout(random_move(state))
```

Basic MCTS pseudocode

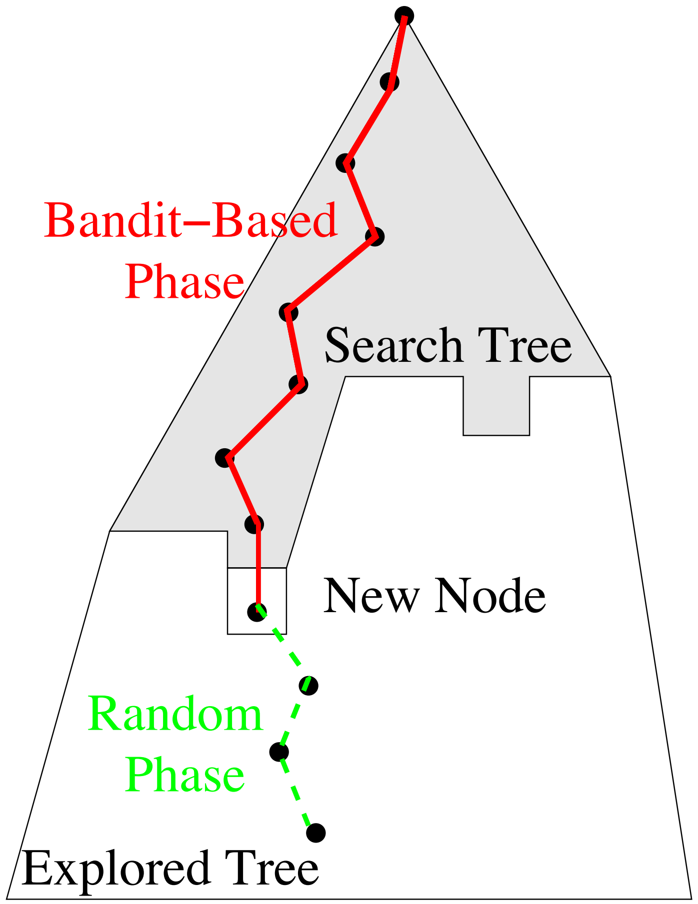
```
function MCTS_sample(state)
    state.visits++
    if all children of state expanded:
        next_state = UCB_sample(state)
        winner = MCTS_sample(next_state)
    else:
        if some children of state expanded:
            next_state = expand(random unexpanded child)
        else:
            next_state = state
        winner = random_playout(next_state)
    update_value(state, winner)
```




```
function random_playout(state):
    if is_terminal(state):
        return winner
    else: return random_playout(random_move(state))
```

Basic MCTS pseudocode

```
function MCTS_sample(state)
    state.visits++
    if all children of state expanded:
        next_state = UCB_sample(state)
        winner = MCTS_sample(next_state)
    else:
        if some children of state expanded:
            next_state = expand(random unexpanded child)
        else:
            next_state = state
        winner = random_playout(next_state)
    update_value(state, winner)
```

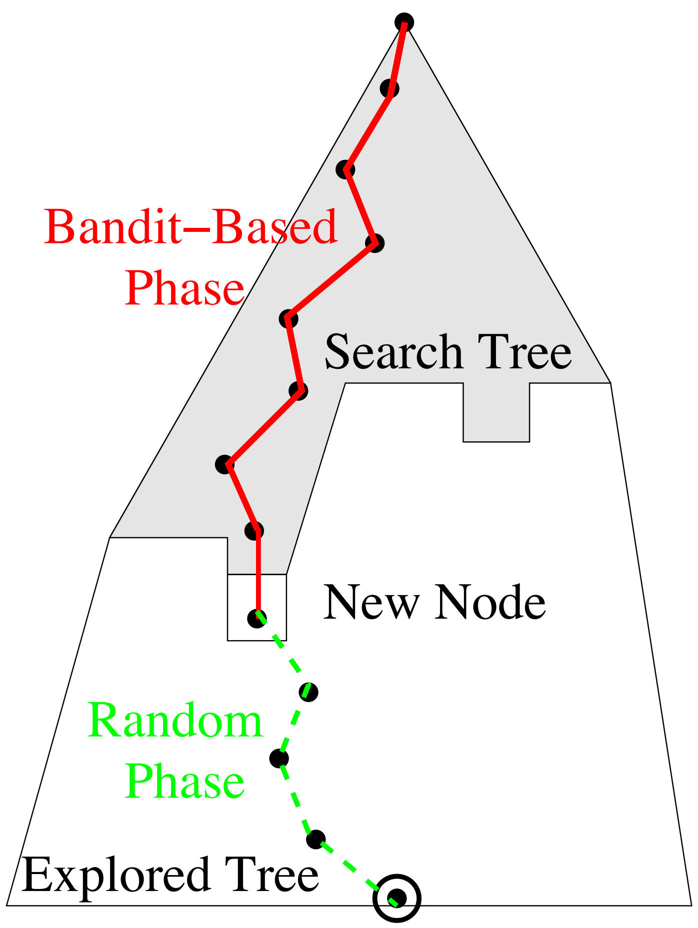



```
function random_playout(state):
    if is_terminal(state):
        return winner
    else: return random_playout(random_move(state))
```

Basic MCTS pseudocode

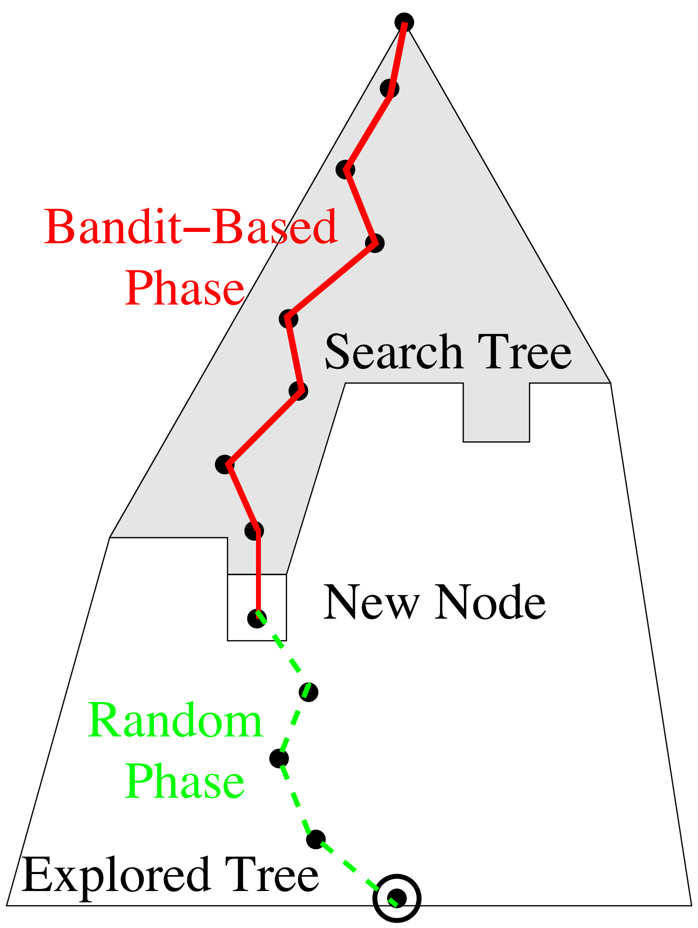
```
function MCTS_sample(state)
    state.visits++
    if all children of state expanded:
        next_state = UCB_sample(state)
        winner = MCTS_sample(next_state)
    else:
        if some children of state expanded:
            next_state = expand(random unexpanded child)
        else:
            next_state = state
        winner = random_playout(next_state)
    update_value(state, winner)
```

```
function random_playout(state):
    if is_terminal(state):
        return winner
    else: return random_playout(random_move(state))
```



Basic MCTS pseudocode

```
function MCTS_sample(state)
    state.visits++
    if all children of state expanded:
        next_state = UCB_sample(state)
        winner = MCTS_sample(next_state)
    else:
        if some children of state expanded:
            next_state = expand(random unexpanded child)
        else:
            next_state = state
        winner = random_playout(next_state)
    update_value(state, winner)
```



Learning from MCTS

- The MCTS agent plays against himself and generates (s, a, Q(s,a)) tuples. Use this data to train:
  - **UCTtoRegression:** A regression network, that given 4 frames regresses to Q(s,a,w) for all actions
  - **UCTtoClassification:** A classification network, that given 4 frames predicts the best action through multiclass classification
- The state distribution visited using actions of the MCTS planner will not match the state distribution obtained from the learned policy.
  - **UCTtoClassification-Interleaved:** Interleave UCTtoClassification with data collection: Start from 200 runs with MCTS as before, train UCTtoClassification, deploy it for 200 runs allowing 5% of the time a random action to be sampled, use MCTS to decide best action for those states, train UCTtoClassification and so on and so forth.

Results

Agent	*B.Rider*	*Breakout*	*Enduro*	*Pong*	*Q*bert*	*Seaquest*	*S.Invaders*
DQN	4092	168	470	20	1952	1705	581
-best	5184	225	661	21	4500	1740	1075
UCC	5342 (20)	175(5.63)	558(14)	19(0.3)	11574(44)	2273(23)	672(5.3)
-best	10514	351	942	21	29725	5100	1200
-greedy	5676	269	692	21	19890	2760	680
UCC-I	5388(4.6)	215(6.69)	601(11)	19(0.14)	13189(35.3)	2701(6.09)	670(4.24)
-best	10732	413	1026	21	29900	6100	910
-greedy	5702	380	741	21	20025	2995	692
UCR	2405(12)	143(6.7)	566(10.2)	19(0.3)	12755(40.7)	1024 (13.8)	441(8.1)

Table 2: Performance (game scores) of the off-line UCT game playing agent.

Agent	*B.Rider*	*Breakout*	*Enduro*	*Pong*	*Q*bert*	*Seaquest*	*S.Invaders*
UCT	7233	406	788	21	18850	3257	2354

Results

Agent	*B.Rider*	*Breakout*	*Enduro*	*Pong*	*Q*bert*	*Seaquest*	*S.Invaders*
DQN	4092	168	470	20	1952	1705	581
-best	5184	225	661	21	4500	1740	1075
UCC	5342 (20)	175(5.63)	558(14)	19(0.3)	11574(44)	2273(23)	672(5.3)
-best	10514	351	942	21	29725	5100	1200
-greedy	5676	269	692	21	19890	2760	680
UCC-I	5388(4.6)	215(6.69)	601(11)	19(0.14)	13189(35.3)	2701(6.09)	670(4.24)
-best	10732	413	1026	21	29900	6100	910
-greedy	5702	380	741	21	20025	2995	692
UCR	2405(12)	143(6.7)	566(10.2)	19(0.3)	12755(40.7)	1024 (13.8)	441(8.1)

Table 2: Performance (game scores) of the off-line UCT game playing agent.

Agent	*B.Rider*	*Breakout*	*Enduro*	*Pong*	*Q*bert*	*Seaquest*	*S.Invaders*
UCT	7233	406	788	21	18850	3257	2354

Online planning (without aided by any neural net!) outperforms DQN policy. It takes though ``a few days on a recent multicore computer to play for each game”.

Results

Agent	*B.Rider*	*Breakout*	*Enduro*	*Pong*	*Q*bert*	*Seaquest*	*S.Invaders*
DQN	4092	168	470	20	1952	1705	581
-best	5184	225	661	21	4500	1740	1075
UCC	5342 (20)	175(5.63)	558(14)	19(0.3)	11574(44)	2273(23)	672(5.3)
-best	10514	351	942	21	29725	5100	1200
-greedy	5676	269	692	21	19890	2760	680
UCC-I	5388(4.6)	215(6.69)	601(11)	19(0.14)	13189(35.3)	2701(6.09)	670(4.24)
-best	10732	413	1026	21	29900	6100	910
-greedy	5702	380	741	21	20025	2995	692
UCR	2405(12)	143(6.7)	566(10.2)	19(0.3)	12755(40.7)	1024 (13.8)	441(8.1)

Table 2: Performance (game scores) of the off-line UCT game playing agent.

Agent	*B.Rider*	*Breakout*	*Enduro*	*Pong*	*Q*bert*	*Seaquest*	*S.Invaders*
UCT	7233	406	788	21	18850	3257	2354

Classification is doing much better than regression! indeed, we are training for exactly what we care about.

Results

Agent	*B.Rider*	*Breakout*	*Enduro*	*Pong*	*Q*bert*	*Seaquest*	*S.Invaders*
DQN	4092	168	470	20	1952	1705	581
-best	5184	225	661	21	4500	1740	1075
UCC	5342 (20)	175(5.63)	558(14)	19(0.3)	11574(44)	2273(23)	672(5.3)
-best	10514	351	942	21	29725	5100	1200
-greedy	5676	269	692	21	19890	2760	680
UCC-I	5388(4.6)	215(6.69)	601(11)	19(0.14)	13189(35.3)	2701(6.09)	670(4.24)
-best	10732	413	1026	21	29900	6100	910
-greedy	5702	380	741	21	20025	2995	692
UCR	2405(12)	143(6.7)	566(10.2)	19(0.3)	12755(40.7)	1024 (13.8)	441(8.1)

Table 2: Performance (game scores) of the off-line UCT game playing agent.

Agent	*B.Rider*	*Breakout*	*Enduro*	*Pong*	*Q*bert*	*Seaquest*	*S.Invaders*
UCT	7233	406	788	21	18850	3257	2354

Interleaving is important to prevent mismatch between the training data and the data that the trained policy will see at test time.

Results

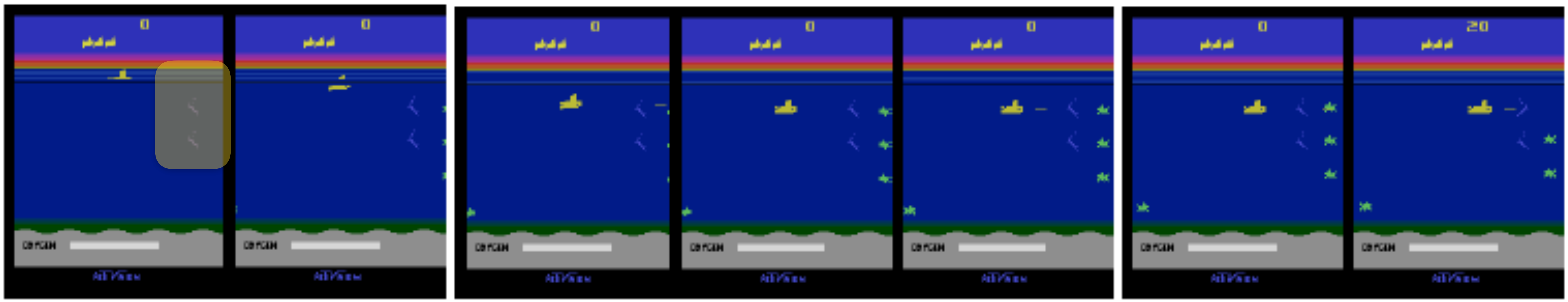
Agent	*B.Rider*	*Breakout*	*Enduro*	*Pong*	*Q*bert*	*Seaquest*	*S.Invaders*
DQN	4092	168	470	20	1952	1705	581
-best	5184	225	661	21	4500	1740	1075
UCC	5342 (20)	175(5.63)	558(14)	19(0.3)	11574(44)	2273(23)	672(5.3)
-best	10514	351	942	21	29725	5100	1200
-greedy	5676	269	692	21	19890	2760	680
UCC-I	5388(4.6)	215(6.69)	601(11)	19(0.14)	13189(35.3)	2701(6.09)	670(4.24)
-best	10732	413	1026	21	29900	6100	910
-greedy	5702	380	741	21	20025	2995	692
UCR	2405(12)	143(6.7)	566(10.2)	19(0.3)	12755(40.7)	1024 (13.8)	441(8.1)

Table 2: Performance (game scores) of the off-line UCT game playing agent.

Agent	*B.Rider*	*Breakout*	*Enduro*	*Pong*	*Q*bert*	*Seaquest*	*S.Invaders*
UCT	7233	406	788	21	18850	3257	2354

Results improve further if you allow MCTS planner to have more simulations and build more reliable Q estimates.

Problem



Step 69: FIRE Step 70: DOWN+FIRE Step 74:DOWN+FIRE Step 75:RIGHT+FIRE Step 76:RIGHT+FIRE Step 78: RIGHT+FIRE Step 79:DOWN+FIRE

We do not learn to save the divers. Saving 6 divers brings very high reward, but exceeds the depth of our MCTS planner, thus it is ignored.

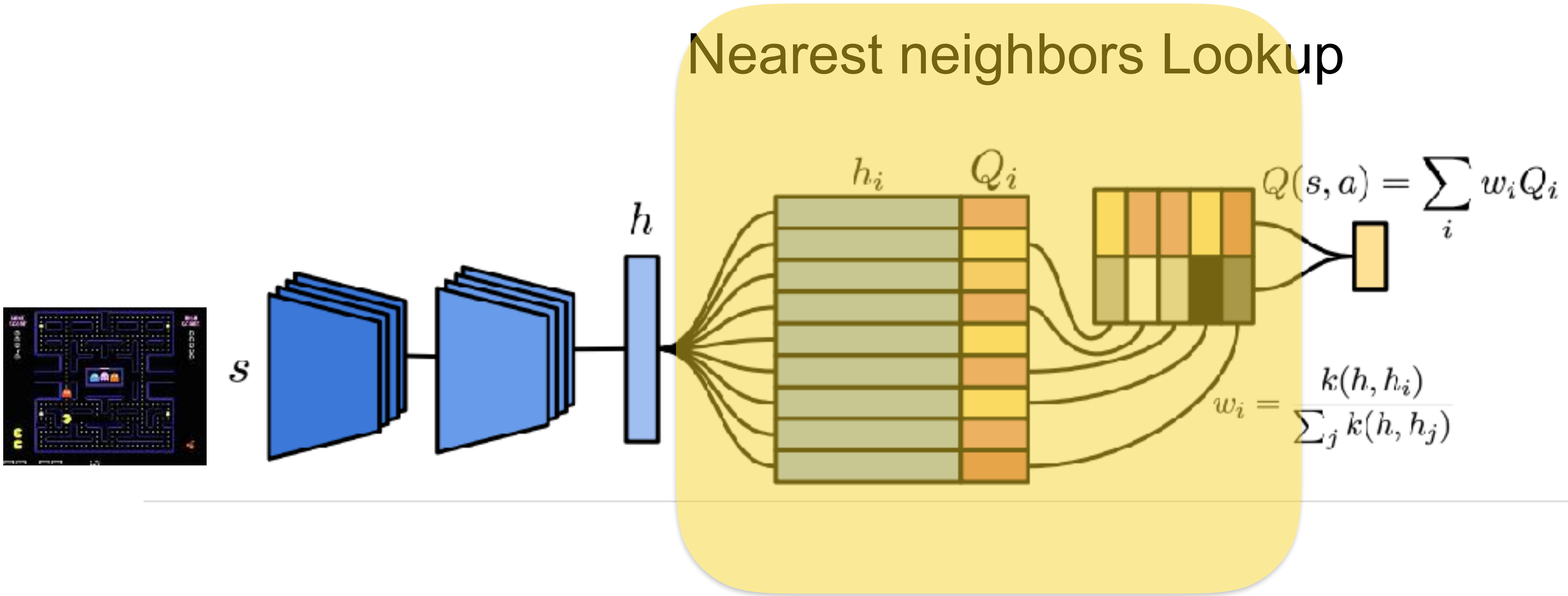
Question

- Why don't we always use MCTS (or some other planner) as supervision for reactive policy learning?

  - Because in many domains we do not have access to the dynamics.

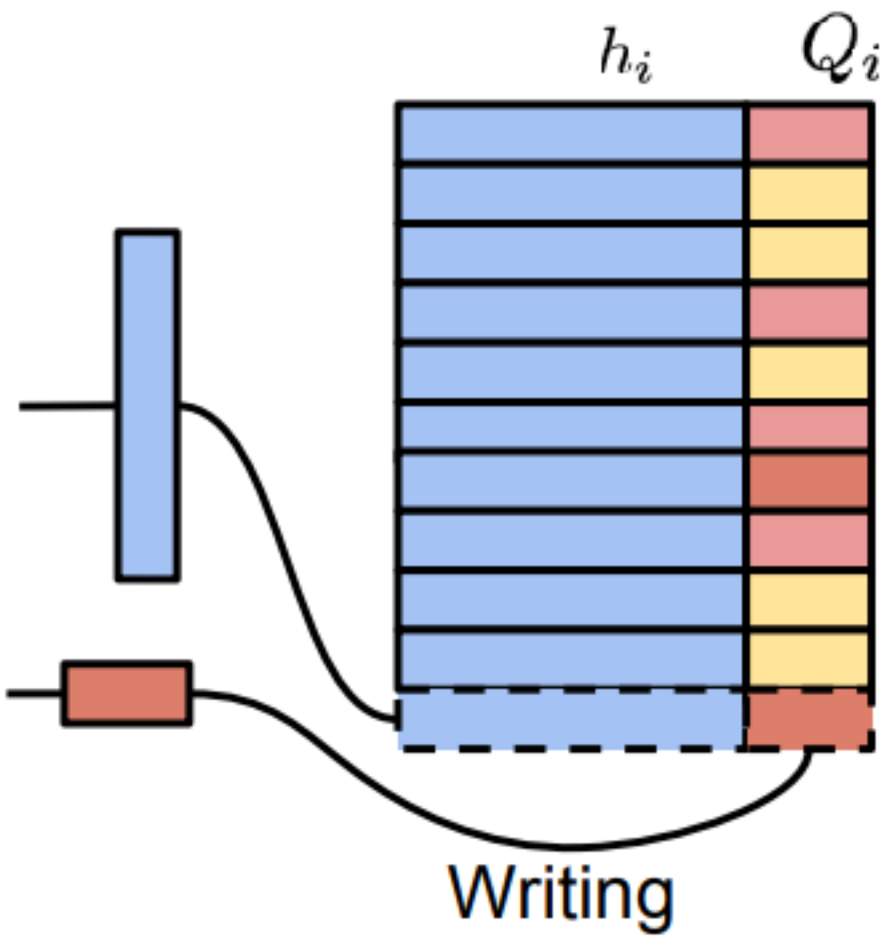
Neural Episodic Control

Alexander Pritzel APRITZEL@GOOGLE.COM
Benigno Uria BURIA@GOOGLE.COM
Sriram Srinivasan SRSRINIVASAN@GOOGLE.COM
Adrià Puigdomènech ADRIAP@GOOGLE.COM
Oriol Vinyals VINYALS@GOOGLE.COM
Demis Hassabis DEMISHASSABIS@GOOGLE.COM
Daan Wierstra WIERSTRA@GOOGLE.COM
Charles Blundell CBLUNDELL@GOOGLE.COM
DeepMind, London UK



Nearest neighbors Lookup
s
h
h_i
Q_i
$Q(s,a) = \sum_i w_i Q_i$
$w_i = \frac{k(h, h_i)}{\sum_j k(h, h_j)}$

Writing in the memory

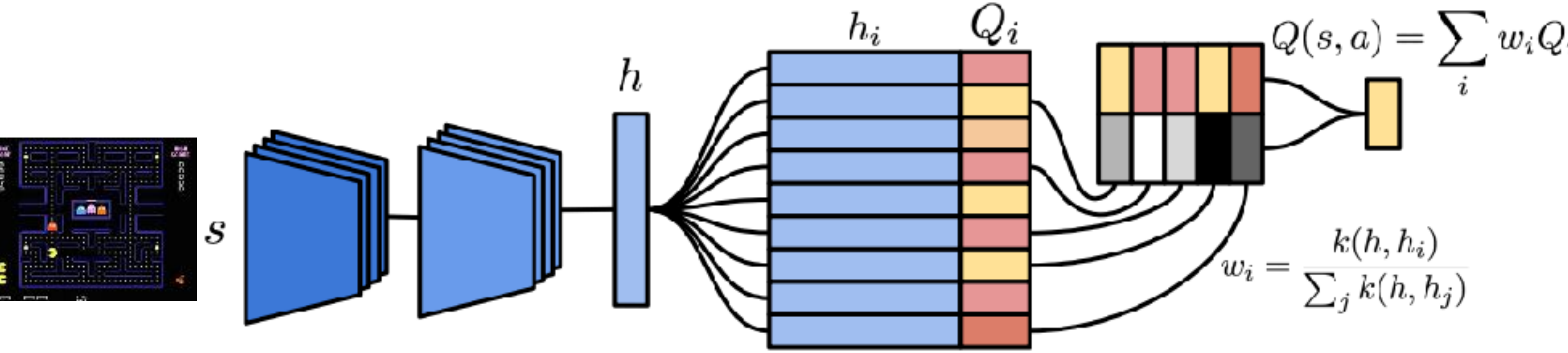


$$Q^{(N)}(s_t, a) = \sum_{j=0}^{N-1} \gamma^j r_{t+j} + \gamma^N \max_{a'} Q(s_{t+N}, a')$$

If identical key h present:

$$Q_i \leftarrow Q_i + \alpha(Q^{(N)}(s, a) - Q_i)$$

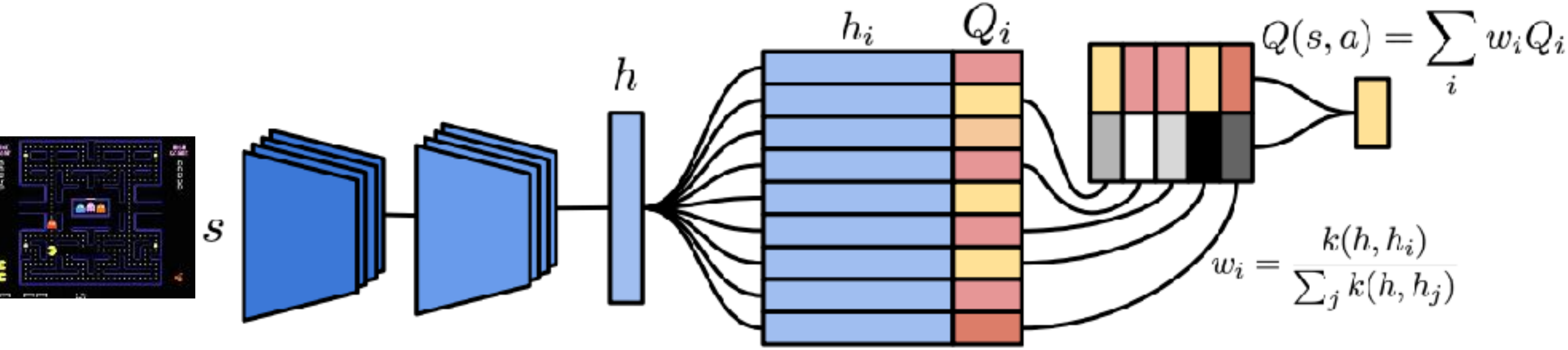
Else add row $(h, Q^N(s, a))$ to the memory



$$Q^{(N)}(s_t, a) = \sum_{j=0}^{N-1} \gamma^j r_{t+j} + \gamma^N \max_{a'} Q(s_{t+N}, a')$$

Algorithm 1 Neural Episodic Control

$\mathcal{D}$: replay memory.
M_a: a DND for each action a.
N: horizon for N-step Q estimate.
for each episode **do**
  for $t = 1, 2, \ldots, T$ **do**
    Receive observation s_t from environment with embedding h.
    Estimate $Q(s_t, a)$ for each action a via (1) from M_a
    $a_t \leftarrow \epsilon$-greedy policy based on $Q(s_t, a)$
    Take action a_t, receive reward r_{t+1}
    Append $(h, Q^{(N)}(s_t, a_t))$ to M_{a_t}.
    Append $(s_t, a_t, Q^{(N)}(s_t, a_t))$ to $\mathcal{D}$.
    Train on a random minibatch from $\mathcal{D}$.
  end for
end for



$$Q^{(N)}(s_t, a) = \sum_{j=0}^{N-1} \gamma^j r_{t+j} + \gamma^N \max_{a'} Q(s_{t+N}, a')$$

$$-\frac{1}{2}\nabla_{\mathbf{w}} J(\mathbf{w}) = (q_\pi(S,A) - \hat{q}(S,A,\mathbf{w}))\nabla_{\mathbf{w}}\hat{q}(S,A,\mathbf{w})$$

$$\Delta\mathbf{w} = \alpha(q_\pi(S,A) - \hat{q}(S,A,\mathbf{w}))\nabla_{\mathbf{w}}\hat{q}(S,A,\mathbf{w})$$

Algorithm 1 Neural Episodic Control

$\mathcal{D}$: replay memory.
M_a: a DND for each action a.
N: horizon for N-step Q estimate.
for each episode **do**
  for $t = 1, 2, \ldots, T$ **do**
    Receive observation s_t from environment with embedding h.
    Estimate $Q(s_t, a)$ for each action a via (1) from M_a
    $a_t \leftarrow \epsilon$-greedy policy based on $Q(s_t, a)$
    Take action a_t, receive reward r_{t+1}
    Append $(h, Q^{(N)}(s_t, a_t))$ to M_{a_t}.
    Append $(s_t, a_t, Q^{(N)}(s_t, a_t))$ to $\mathcal{D}$.
    Train on a random minibatch from $\mathcal{D}$.
  end for
end for