Carnegie Mellon

School of Computer Science

Deep Reinforcement Learning and Control

Monte Carlo Tree Search

Fall 2021, CMU 10-703

Instructors:
Katerina Fragkiadaki
Russ Salakhutdinov

Part of today's lecture is inspired by the MCTS presentation of Bryce Wiedenbeck

Definitions

Learning: the acquisition of knowledge or skills through experience, study, or by being taught.

Planning: any computational process that uses a model to create or improve a policy

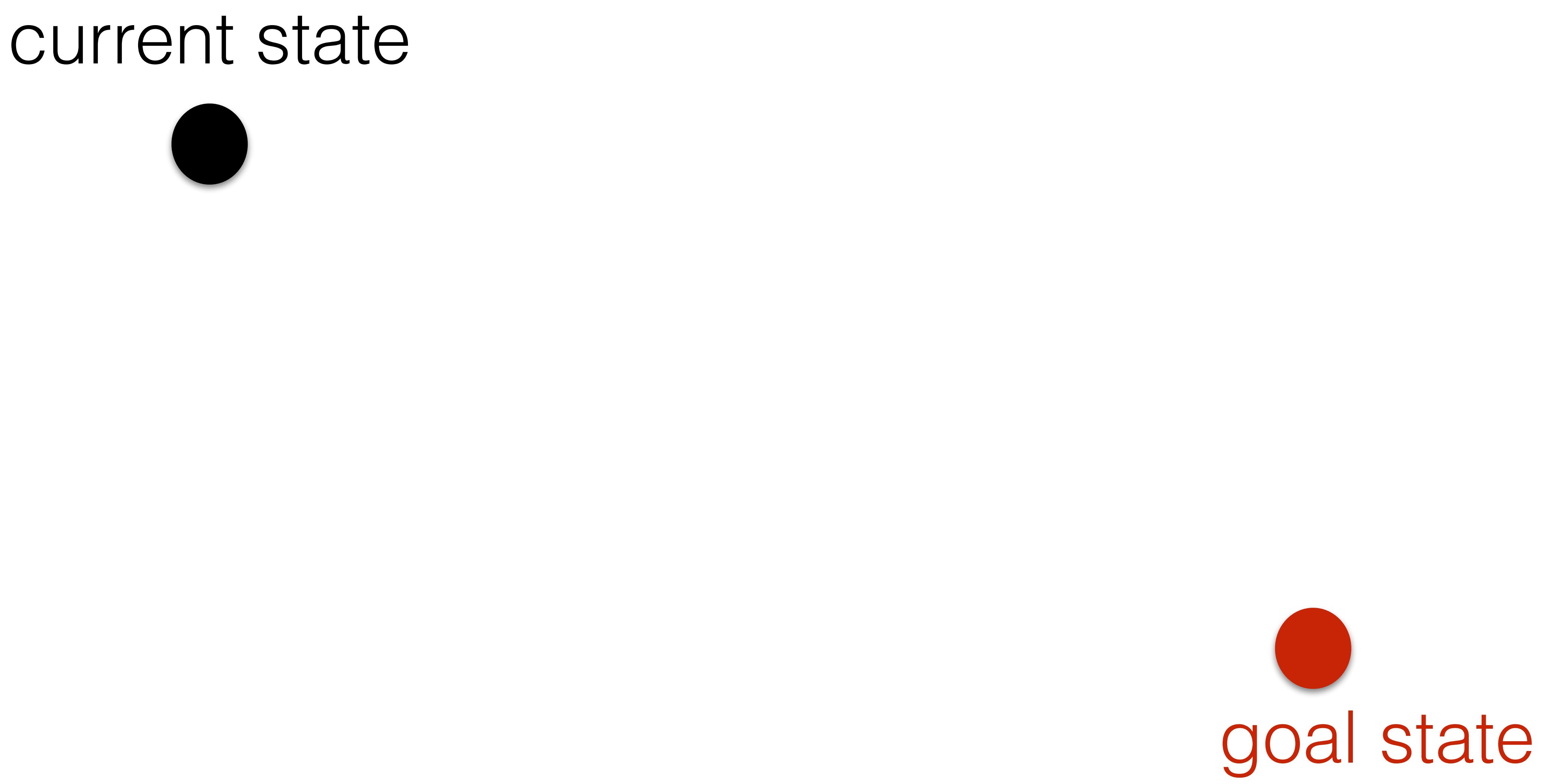
$$\begin{tikzcd} \text{Model} \arrow[r, "\text{Planning}"] & \text{Policy} \end{tikzcd}$$

What is Online Planning?

Unroll the model of the environment forward in time to select the right action sequences to achieve your goal.

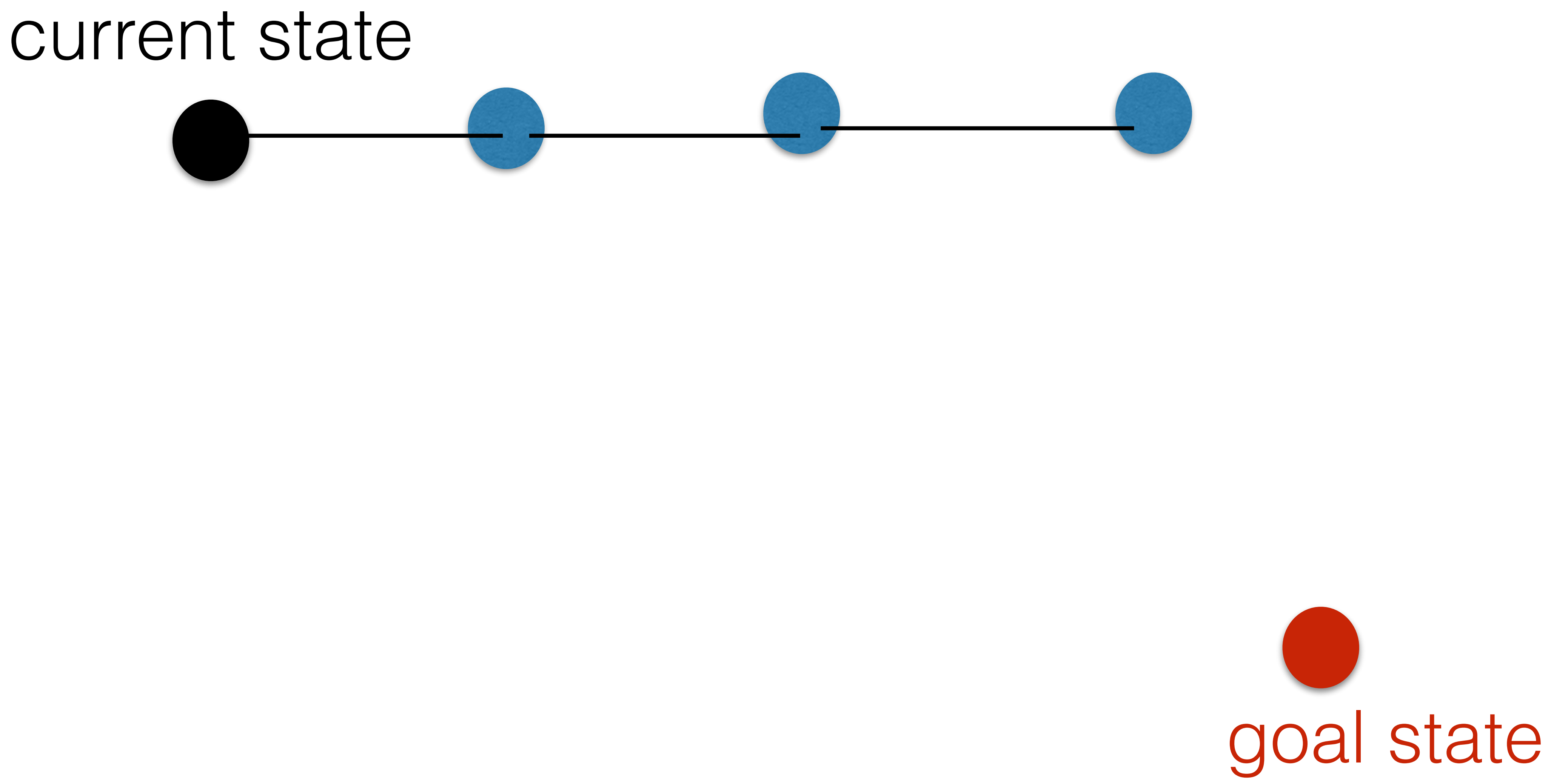
What is Online Planning?

Unroll the model of the environment forward in time to select the right action sequences to achieve your goal.



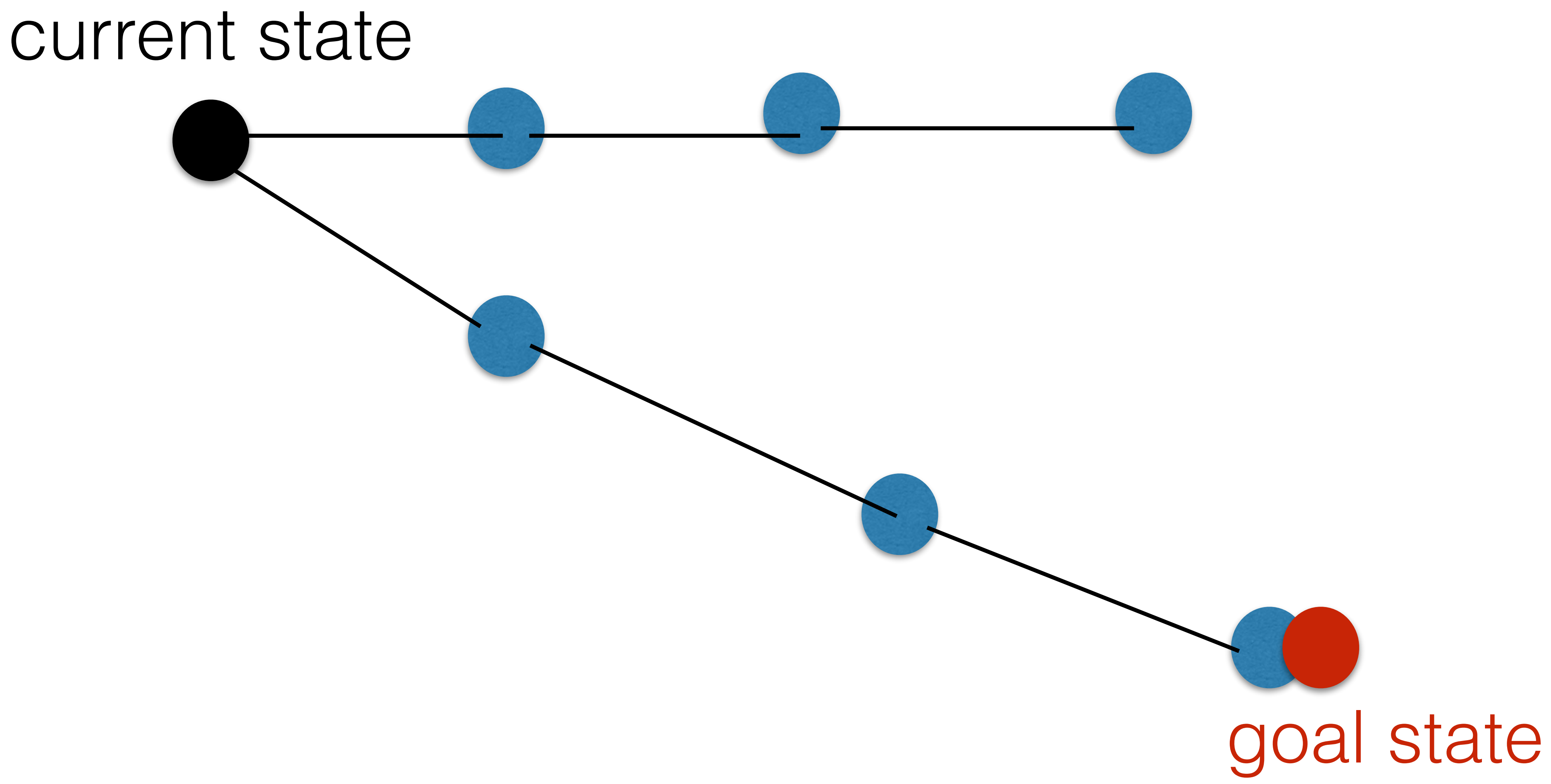
What is Online Planning?

Unroll the model of the environment forward in time to select the right action sequences to achieve your goal.



What is Online Planning?

Unroll the model of the environment forward in time to select the right action sequences to achieve your goal.



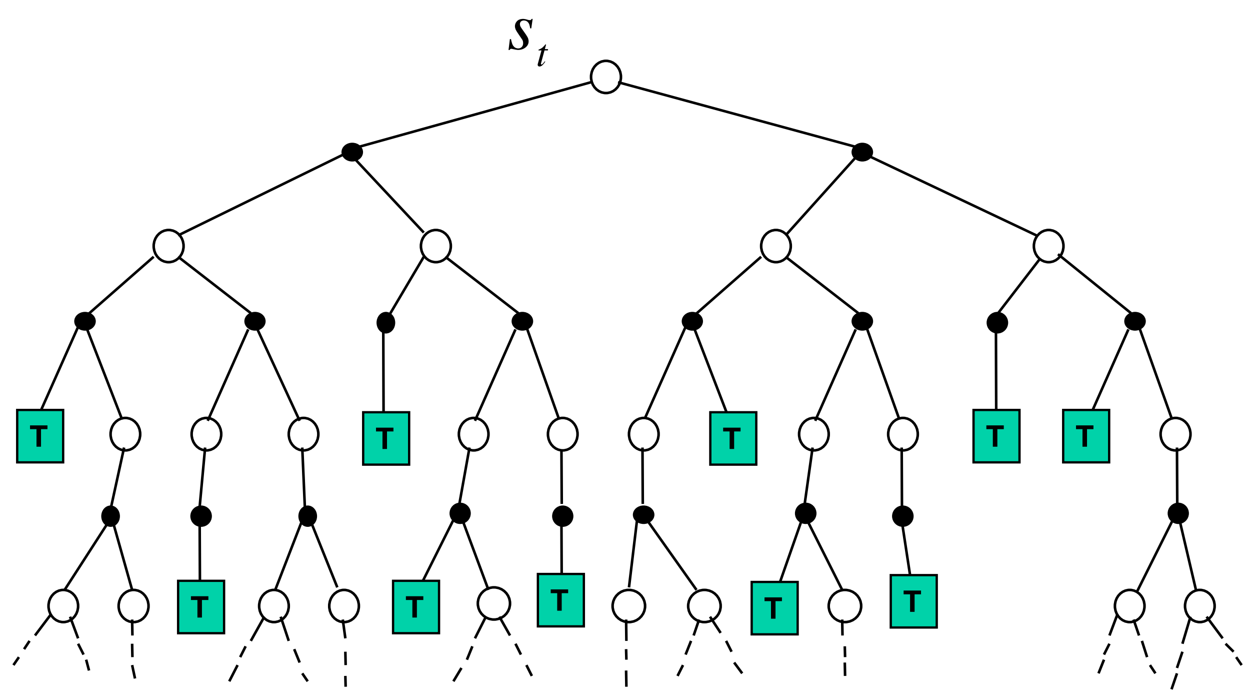
Why online planning?

Why don’t we *just* learn a value function directly for every state offline, so that we do not waste time online?

- Because the environment has many many states (consider Go 10^170, Chess 10^48, real world ….)
- Very hard to compute a good value function for each one of them, most of them you will never visit at training time.
- Thus, condition on the current state you are in, try to estimate the value function of the relevant part of the state space online.
- Focus your resources on sub-MDP starting from now, often dramatically easier than solving the whole MDP.

Online Planning with Search

1. Build the full search tree **with the current state of the agent** at the root
2. Select the next move to execute using heuristics
3. Execute it
4. GOTO 1



Curse of dimensionality

The sub-MDP rooted at the current state the agent is in may still be very large (too many states are reachable), despite much smaller than the original one.

Too many actions possible: large tree branching factor

Too many steps: large tree depth

I cannot exhaustively search the full tree

Curse of dimensionality

Consider hex on an NxN board.

branching factor ≤ N^2

$2N \leq$ depth $\leq N^2$

board size	max branching factor	min depth	tree size	depth of 10^{10} nodes
6x6	36	12	$>10^{17}$	7
8x8	64	16	$>10^{28}$	6
11x11	121	22	$>10^{44}$	5
19x19	361	38	$>10^{96}$	4

Goal of HEX: to make a connected line that links two antipodal points of the grid

How to handle the curse of dimensionality?

Solution 1 (we will visit this in a later lecture): Intelligent instead of exhaustive search

The depth of the search may be reduced by position evaluation: truncating the search tree at state s and replacing the subtree below s by an approximate value function $v(s) = v*(s)$that predicts the outcome from state s.

The breadth of the search may be reduced by sampling actions from a policy $p(a \mid s)$, that is, a probability distribution over plausible moves a in position s, instead of trying every action.

Value function

We can estimate values for states in two ways:

- Engineering them using human experts (DeepBlue)
- Learning them from self-play (TD-gammon)

Problems with human engineering:

- tiring
- non transferrable to other domains.

YET: that's how Kasparov was first beaten.

http://stanford.edu/~cpiech/cs221/apps/deepBlue.html

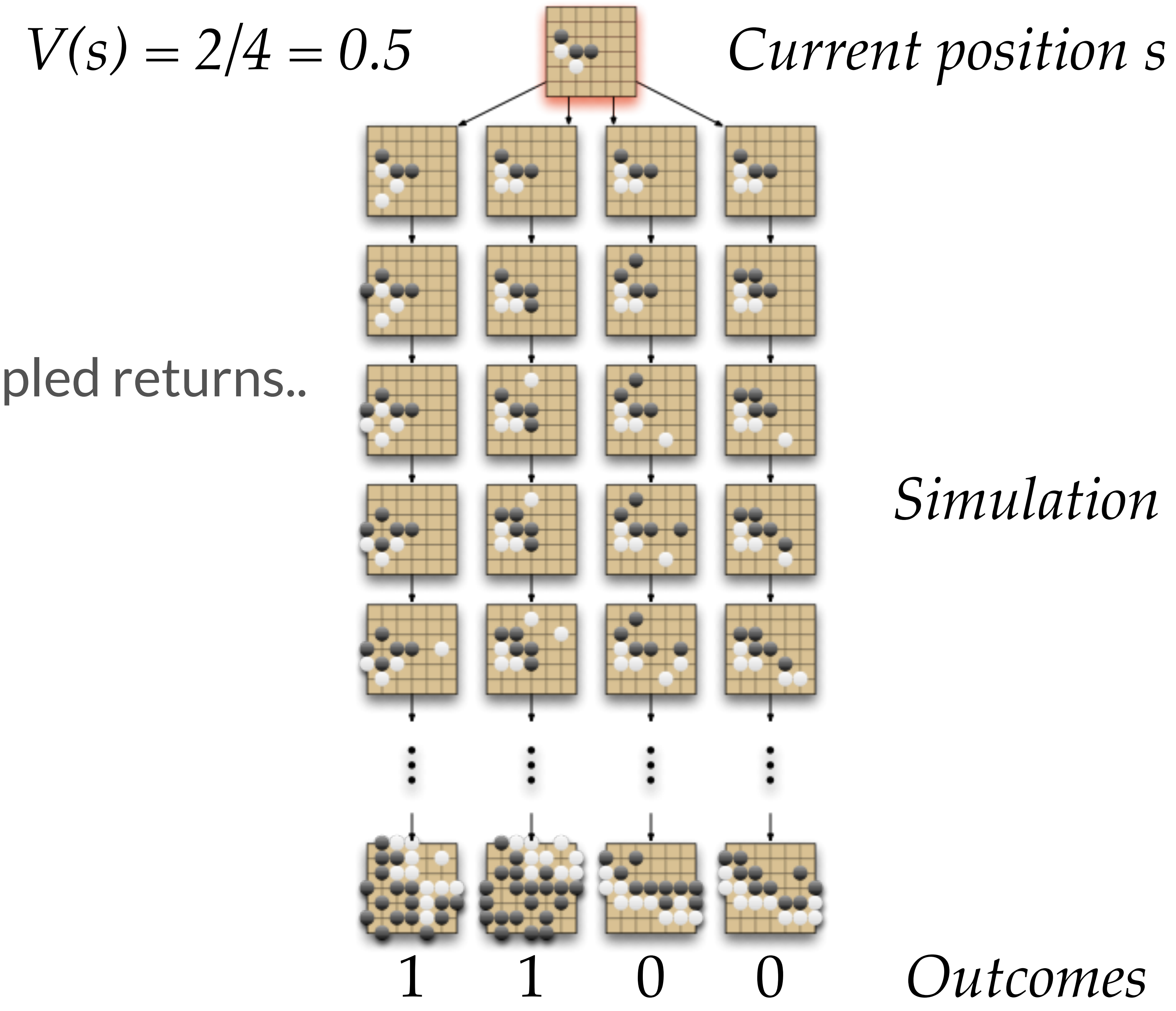
Solution 2 (today): Random instead of exhaustive searc

1. We will be sampling from our actions instead of exhaustively trying them all.

2. We will concurrently be learning from (mental) outcomes of our (mental) samples how to (mentally) sample better in the next round (trajectory).

Monte-Carlo search

Averaging sampled returns..



Monte-Carlo position evaluation

```
function MC_BoardEval(state):
    wins = 0
    losses = 0
    for i=1:NUM_SAMPLES
        next_state = state
        while non_terminal(next_state):
            next_state = random_legal_move(next_state)
        if next_state.winner == state.turn: wins++
        else: losses++ #needs slight modification if draws possible
    return (wins - losses) / (wins + losses)
```

What policy shall we use to draw our simulations?

The cheapest one is random..

Simplest Monte-Carlo Search

Given a deterministic transition function T, a root state s and a simulation policy π (potentially random)

Simulate K episodes from current (real) state:

$$\{s, a, R_1^k, S_1^k, A_1^k, R_2^k, S_2^k, A_2^k, \ldots, S_T^k\}_{k=1}^K \sim \mathrm{T}, \pi$$

Evaluate action value function of the root by mean return:

$$Q(s,a) = \frac{1}{K}\sum_{k=1}^{K} G_k \to q_\pi(s,a)$$

Select root action: $a = \mathrm{argmax}_{a \in \mathscr{A}} Q(s,a)$

Simplest Monte-Carlo Search

Given a deterministic transition function T, a root state s and a simulation policy π (potentially random)

For each action $a \in \mathscr{A}$

$$Q(s,a) = \text{MC-boardEval}(s'), \quad s' = \mathrm{T}(s,a)$$

Select root action: $a = \mathrm{argmax}_{a \in \mathscr{A}} Q(s,a)$

Can we do better?

- Could we be improving our simulation policy the more simulations we obtain?
- Yes we can! We can have two policies:
  - Internal to the tree: keep track of action values Q not only for the root but also for nodes internal to a tree we are expanding, and use to improve the simulation policy over time
  - External to the tree: we do not have Q estimates and thus we use a random policy

In MCTS, the simulation policy improves

- Can we think anything better than $\epsilon -$ greedy?

Upper Confidence Bound (UCB)

$$A_t = \text{argmax}_a \left[Q_t(a) + c\sqrt{\frac{\log t}{N_t(a)}} \right]$$

- t : parent node visits
- $N_t(a)$: times the action has been tried out

- Probability of choosing an action:
  - decreases with the number of visits (explore)
  - increases with a node's value (exploit)
- Always tries every option once.

- A better exploration-exploitation than $\epsilon -$ greedy

Monte-Carlo Tree Search

1. **Selection**
    - Used for nodes we have seen before
    - Pick actions according to UCB
2. **Expansion**
    - Used when we reach the frontier
    - Add one node per rollout
3. **Simulation**
    - Used beyond the search frontier
    - Don’t bother with UCB, just pick actions randomly
4. **Back-propagation**
    - 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*

Monte-Carlo Tree Search

```
function UCB_sample(node):
    weights = []
    for child of node:
        w = child.value
        w += C*sqrt(ln(node.visits) / child.visits)
        add w to weights
    distribution = normalize weights to sum to 1
    return child sampled according to distribution
```

Monte-Carlo Tree Search

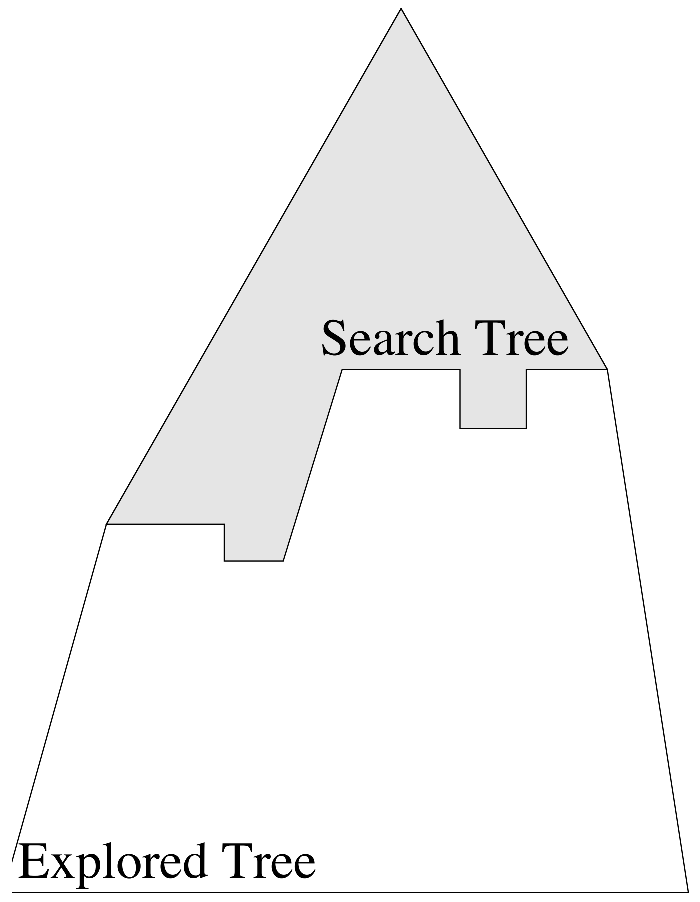
```
function MCTS_sample(node)
    if all children expanded:#selection
        next = UCB_sample(node)
        outcome = MCTS_sample(next)
    else:#expansion
        next = random unexpanded child
        create node for next, add to tree
        #simulation
        outcome = random_playout(next.state)
    #backpropagation
    update_value(node, outcome)
```

For every state within the search tree we bookkeep # of visits and # of wins

Monte-Carlo Tree Search (helper functions)

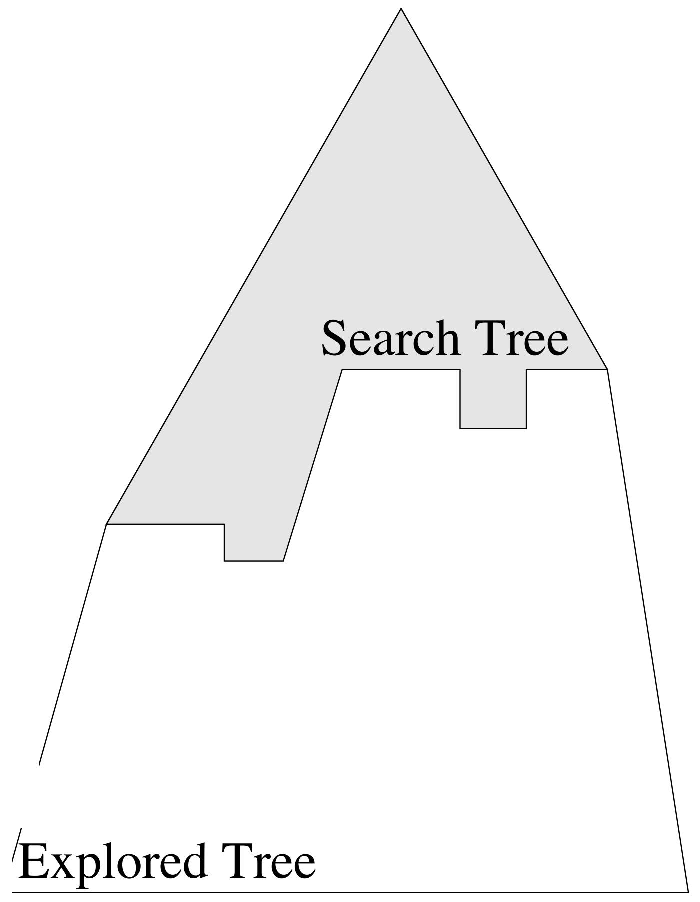
```
function random_playout(state):
   while state is not terminal:
       state = make a random move from state
   return outcome



function update_value(node, outcome):
   #combine the new outcome with the average value
   node.value *= node.visits
   node.visits++
   node.value += outcome
   node.value /= node.visits
```

```
function MCTS_sample(node)
    if all children expanded:#selection
        next = UCB_sample(node)
        outcome = MCTS_sample(next)
    else:#expansion
        next = random unexpanded child
        create node for next, add to tree
        #simulation
        outcome = random_playout(next.state)
    #backpropagation
    update_value(node, outcome)
```

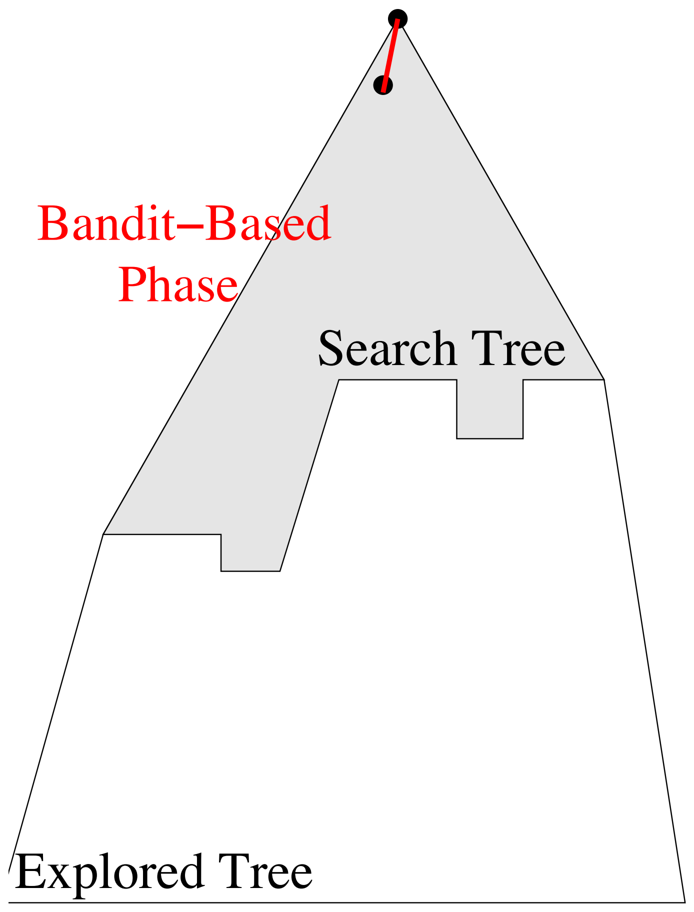


Search tree contains states whose children have been tried at least once

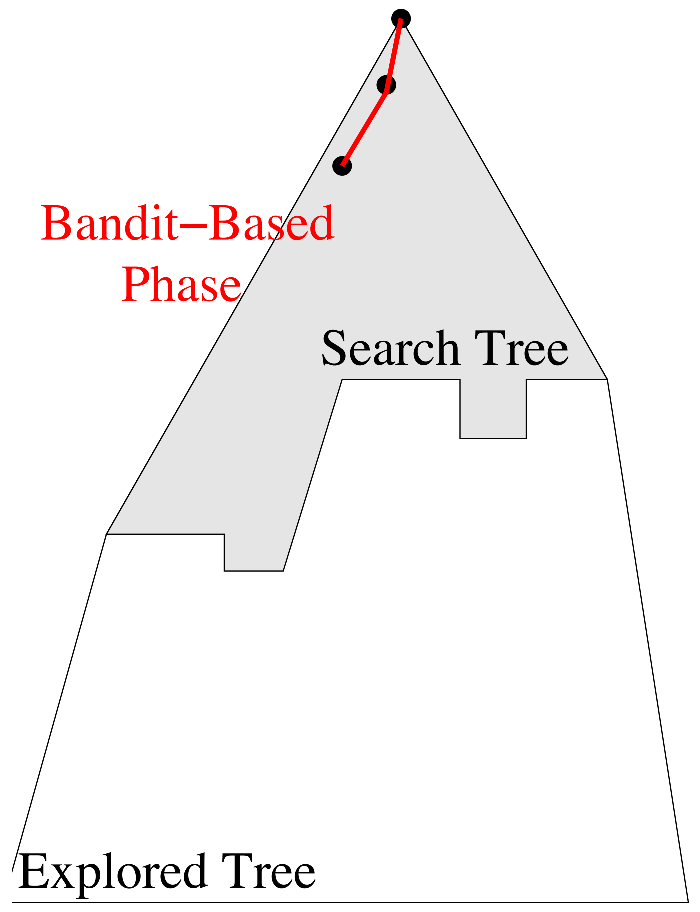
```
function MCTS_sample(node)
    if all children expanded:#selection
        next = UCB_sample(node)
        outcome = MCTS_sample(next)
    else:#expansion
        next = random unexpanded child
        create node for next, add to tree
        #simulation
        outcome = random_playout(next.state)
    #backpropagation
    update_value(node, outcome)
```



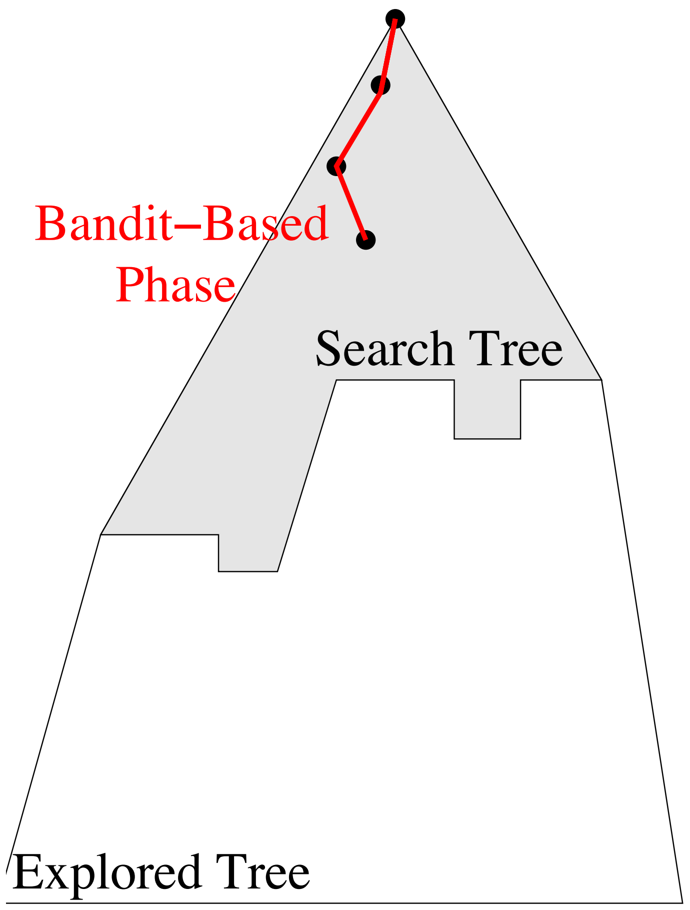
```
function MCTS_sample(node)
   if all children expanded:#selection
      next = UCB_sample(node)
      outcome = MCTS_sample(next)
   else:#expansion
      next = random unexpanded child
      create node for next, add to tree
      #simulation
      outcome = random_playout(next.state)
   #backpropagation
   update_value(node, outcome)
```



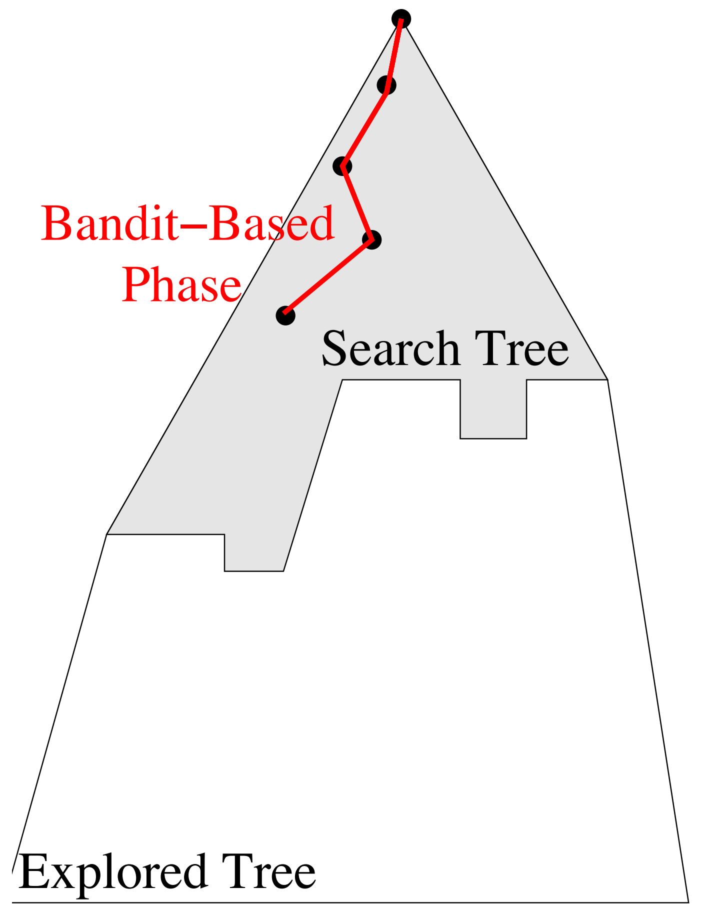
```
function MCTS_sample(node)
   if all children expanded:#selection
      next = UCB_sample(node)
      outcome = MCTS_sample(next)
   else:#expansion
      next = random unexpanded child
      create node for next, add to tree
      #simulation
      outcome = random_playout(next.state)
   #backpropagation
   update_value(node, outcome)
```



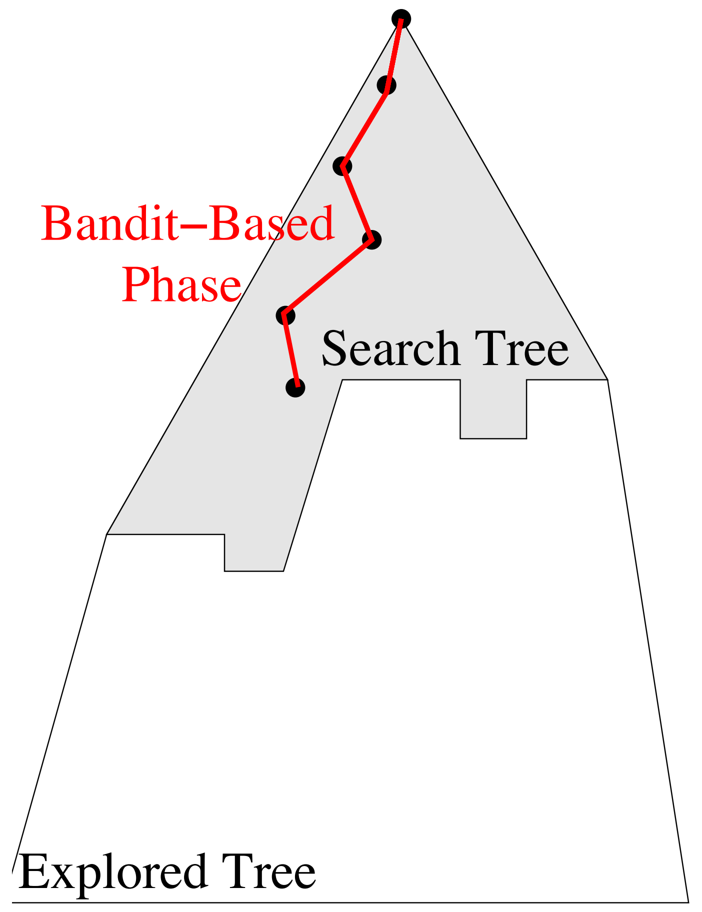
```
function MCTS_sample(node)
    if all children expanded:#selection
        next = UCB_sample(node)
        outcome = MCTS_sample(next)
    else:#expansion
        next = random unexpanded child
        create node for next, add to tree
        #simulation
        outcome = random_playout(next.state)
    #backpropagation
    update_value(node, outcome)
```



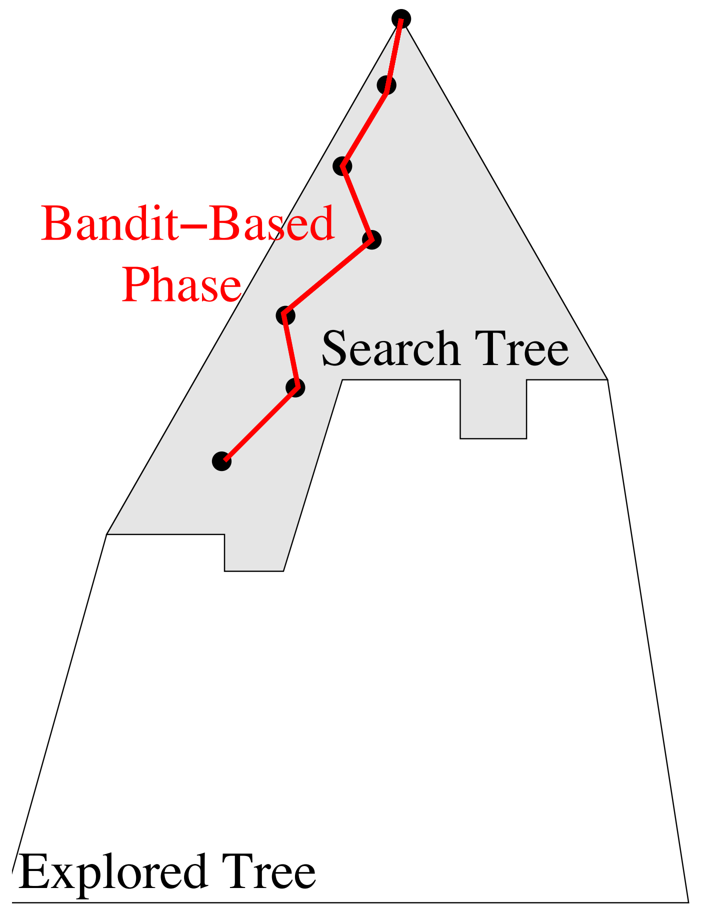
```
function MCTS_sample(node)
   if all children expanded:#selection
      next = UCB_sample(node)
      outcome = MCTS_sample(next)
   else:#expansion
      next = random unexpanded child
      create node for next, add to tree
      #simulation
      outcome = random_playout(next.state)
   #backpropagation
   update_value(node, outcome)
```



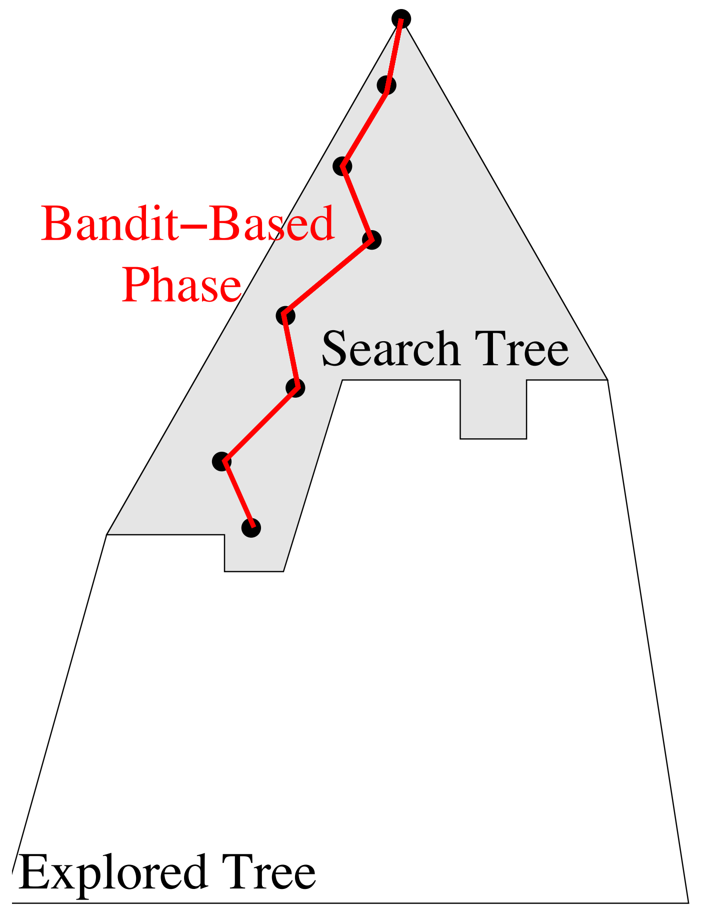
```
function MCTS_sample(node)
   if all children expanded:#selection
      next = UCB_sample(node)
      outcome = MCTS_sample(next)
   else:#expansion
      next = random unexpanded child
      create node for next, add to tree
      #simulation
      outcome = random_playout(next.state)
   #backpropagation
   update_value(node, outcome)
```



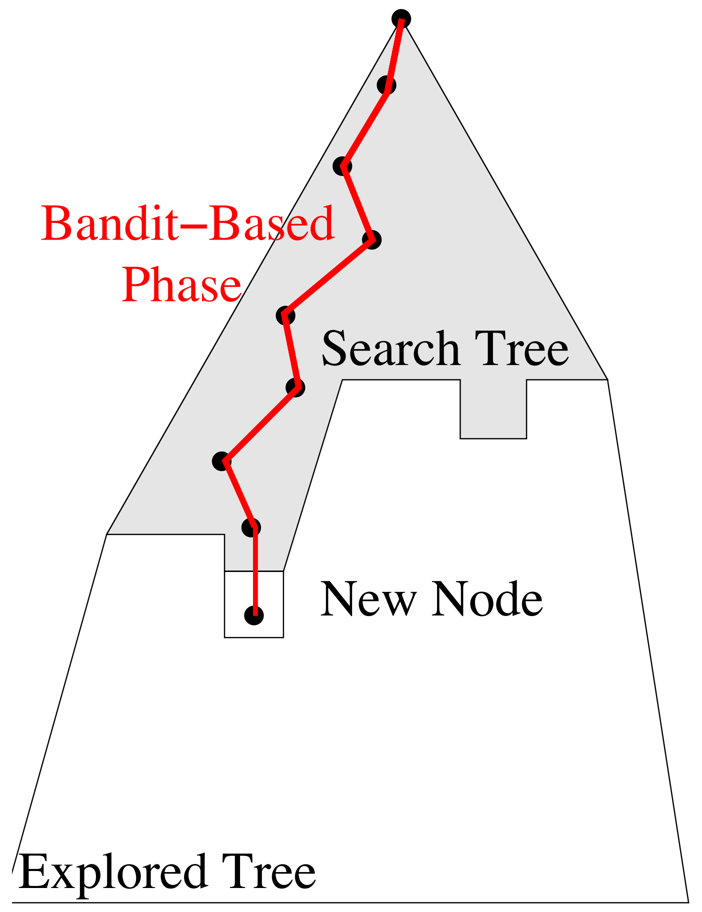
```
function MCTS_sample(node)
   if all children expanded:#selection
      next = UCB_sample(node)
      outcome = MCTS_sample(next)
   else:#expansion
      next = random unexpanded child
      create node for next, add to tree
      #simulation
      outcome = random_playout(next.state)
   #backpropagation
   update_value(node, outcome)
```



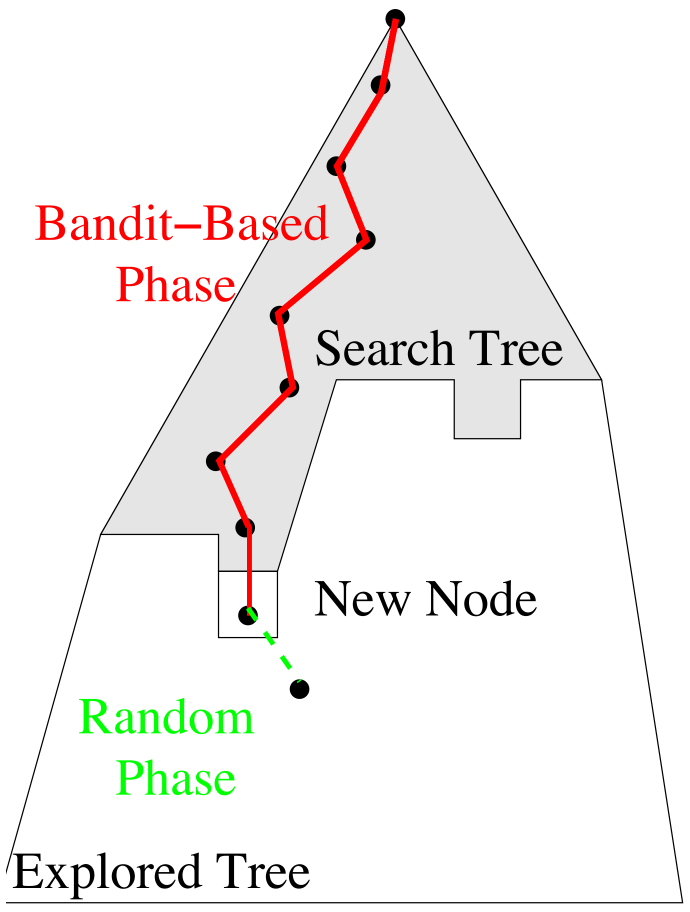
```
function MCTS_sample(node)
   if all children expanded:#selection
      next = UCB_sample(node)
      outcome = MCTS_sample(next)
   else:#expansion
      next = random unexpanded child
      create node for next, add to tree
      #simulation
      outcome = random_playout(next.state)
   #backpropagation
   update_value(node, outcome)
```



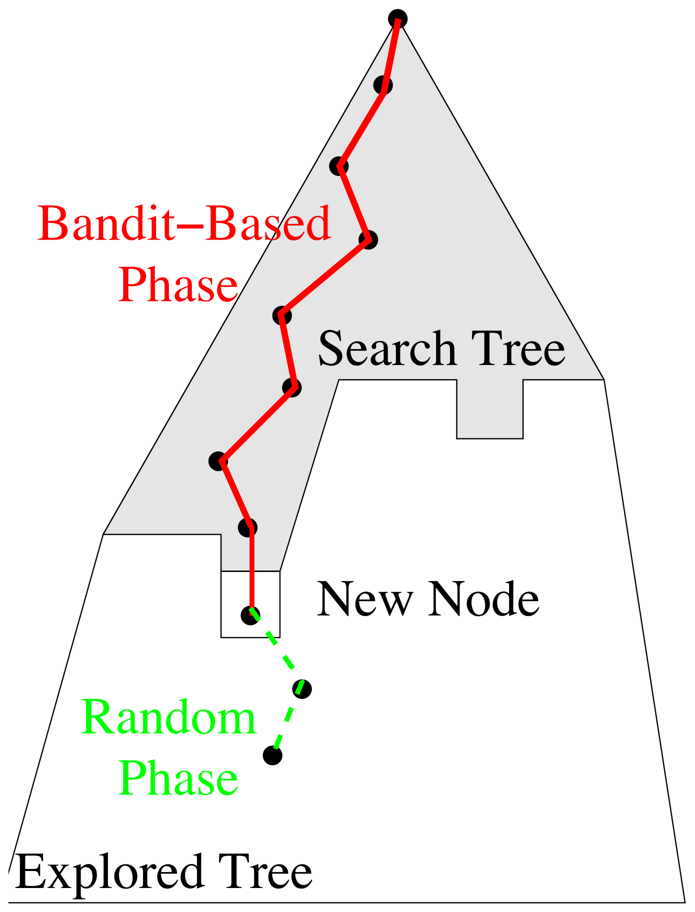
```
function MCTS_sample(node)
   if all children expanded:#selection
      next = UCB_sample(node)
      outcome = MCTS_sample(next)
   else:#expansion
      next = random unexpanded child
      create node for next, add to tree
      #simulation
      outcome = random_playout(next.state)
   #backpropagation
   update_value(node, outcome)
```



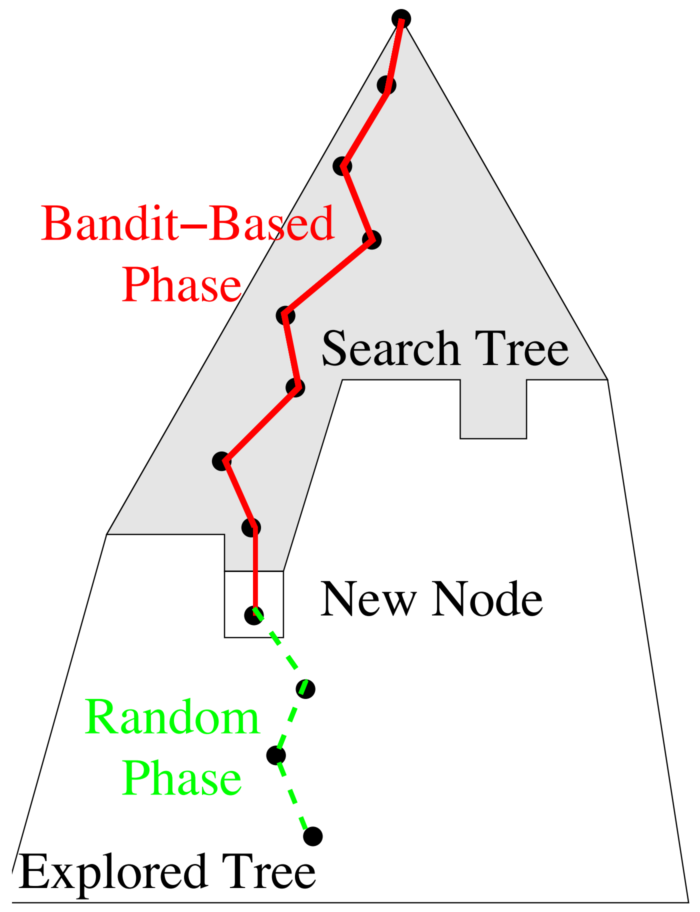
```
function MCTS_sample(node)
    if all children expanded:#selection
        next = UCB_sample(node)
        outcome = MCTS_sample(next)
    else:#expansion
        next = random unexpanded child
        create node for next, add to tree
        #simulation
        outcome = random_playout(next.state)
    #backpropagation
    update_value(node, outcome)
```



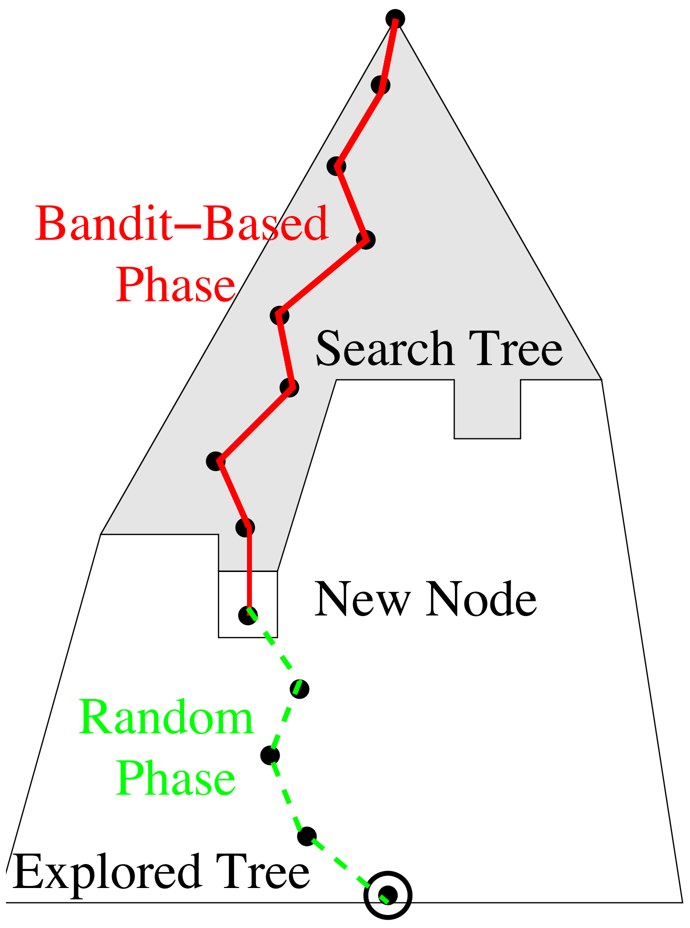
```
function MCTS_sample(node)
   if all children expanded:#selection
      next = UCB_sample(node)
      outcome = MCTS_sample(next)
   else:#expansion
      next = random unexpanded child
      create node for next, add to tree
      #simulation
      outcome = random_playout(next.state)
   #backpropagation
   update_value(node, outcome)
```



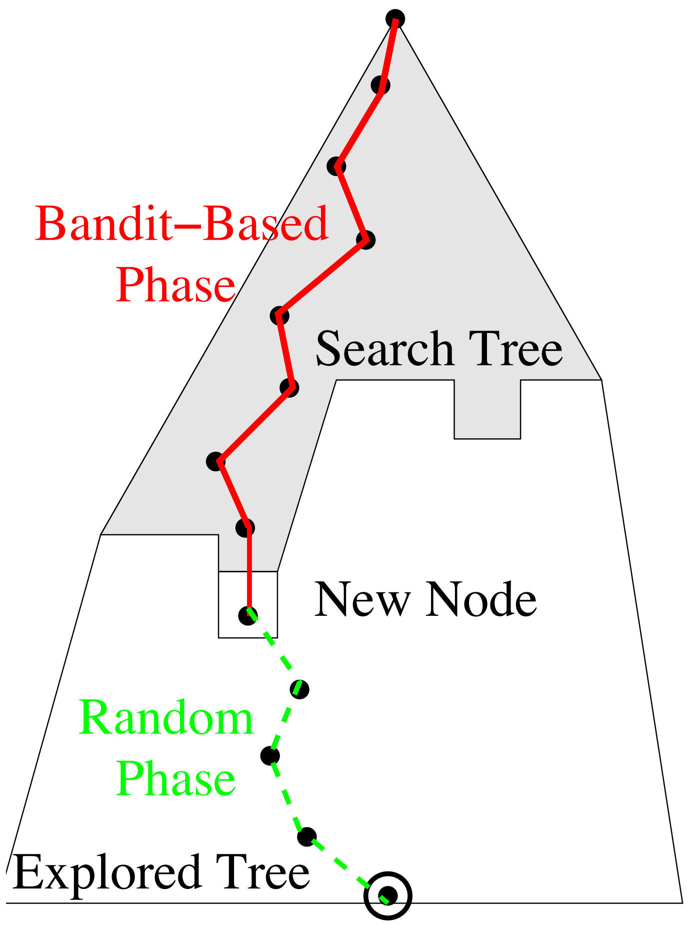
```
function MCTS_sample(node)
    if all children expanded:#selection
        next = UCB_sample(node)
        outcome = MCTS_sample(next)
    else:#expansion
        next = random unexpanded child
        create node for next, add to tree
        #simulation
        outcome = random_playout(next.state)
    #backpropagation
    update_value(node, outcome)
```



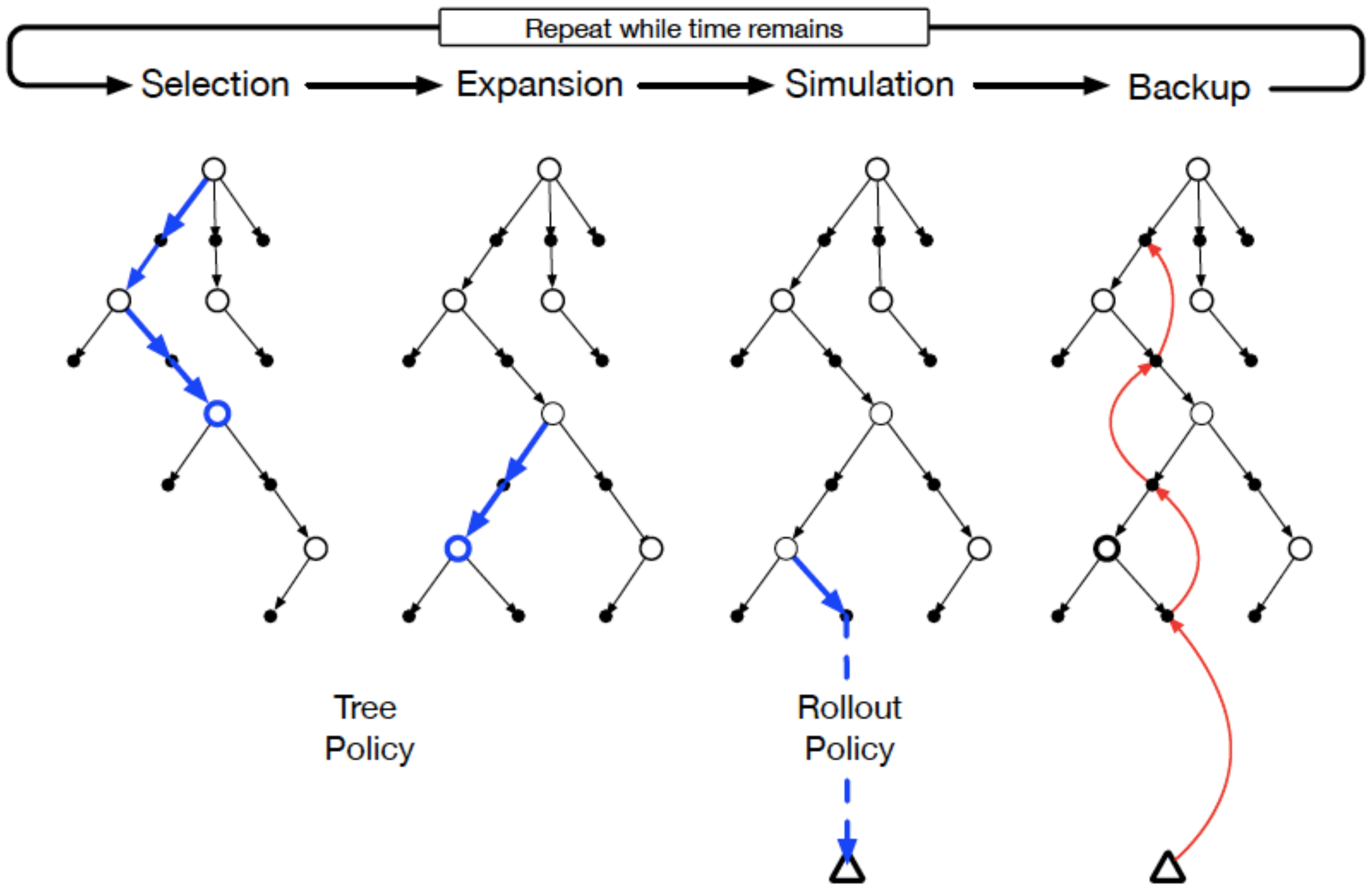
```
function MCTS_sample(node)
    if all children expanded:#selection
        next = UCB_sample(node)
        outcome = MCTS_sample(next)
    else:#expansion
        next = random unexpanded child
        create node for next, add to tree
        #simulation
        outcome = random_playout(next.state)
    #backpropagation
    update_value(node, outcome)
```



```
function MCTS_sample(node)
    if all children expanded:#selection
        next = UCB_sample(node)
        outcome = MCTS_sample(next)
    else:#expansion
        next = random unexpanded child
        create node for next, add to tree
        #simulation
        outcome = random_playout(next.state)
    #backpropagation
    update_value(node, outcome)
```

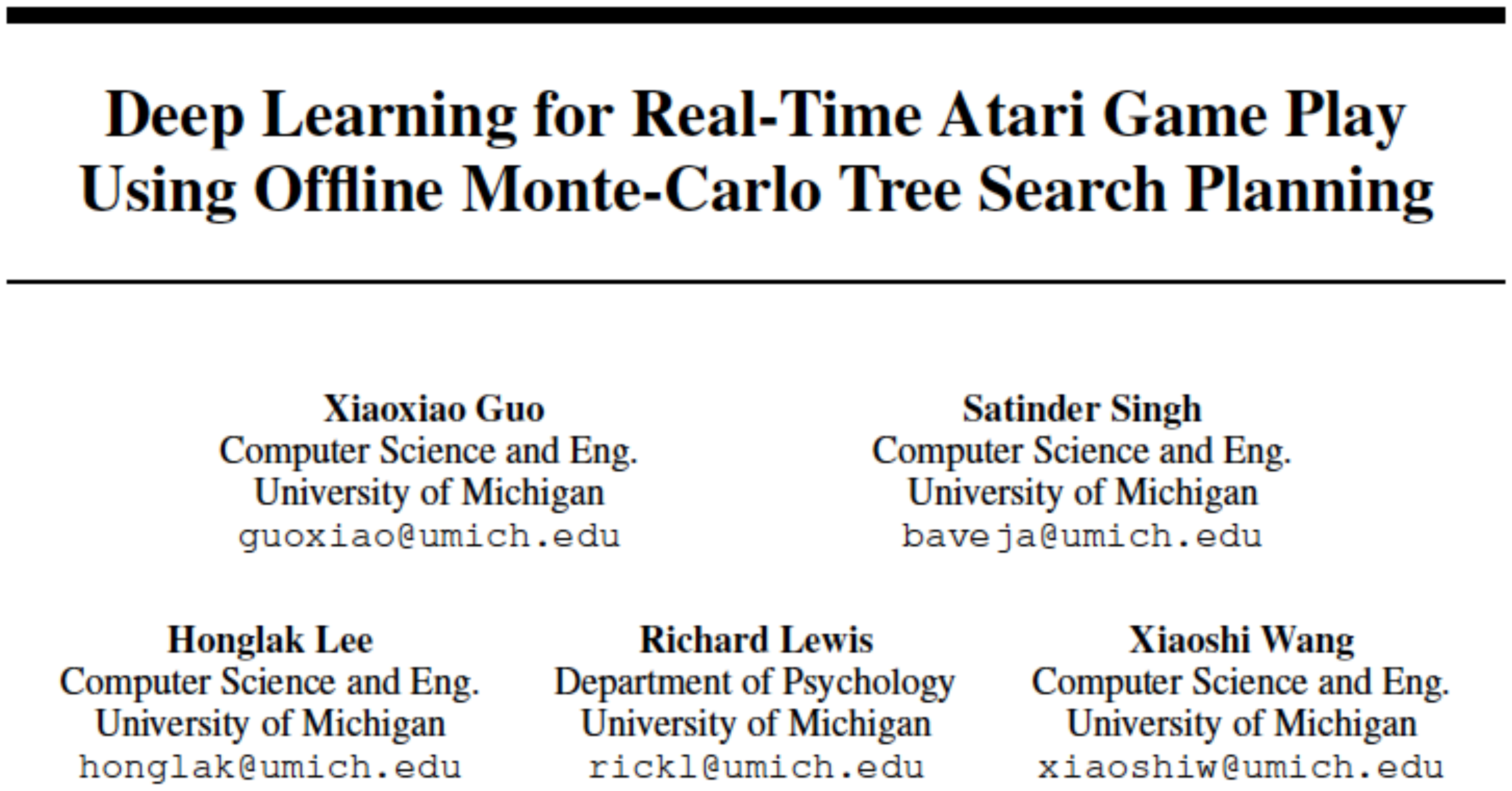


Monte-Carlo Tree Search



Monte-Carlo Tree Search planner

- Estimates action-state values $Q(s, a)$ by look-ahead planning.
- Questions:
  - Which one, MCTS or DQN, discovers better actions, that is, Q estimates that if we act greedily with respect to them, we achieve higher returns on expectation?
  - Why don't we simply use MCTS to select actions during playing of Atari games (no prior knowledge)?
  - How can we use the estimates discovered with MCTS but at the same time play fast at test time?



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

Idea: Use MCTS for Q value estimation and action selection at training time instead of the Q learning update rule.

At test time just use the reactive policy network, without any look-ahead planning. In other words, imitate the MCTS planner.

Learning to play from offline MCTS

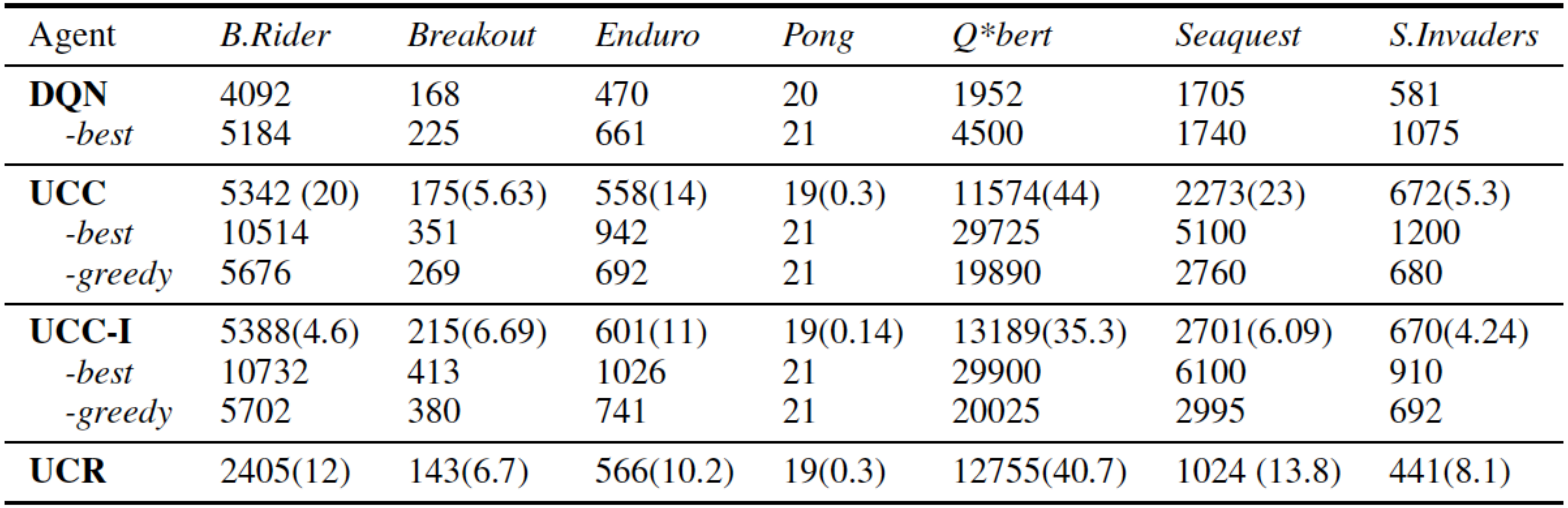
- The MCTS agent plays against itself and generates $(s, a, Q(s, a))$ tuples. Use this data to train:
    - **UCTtoRegression**: A regression network, that given 4 frames regresses to $Q(s, a, w)$ for all actions. Select actions using argmax Q.
    - **UCTtoClassification**: A classification network, that given 4 frames predicts the best action.
- Q: Could we use the learned policies to play the game?

Learning from offline MCTS

- 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:

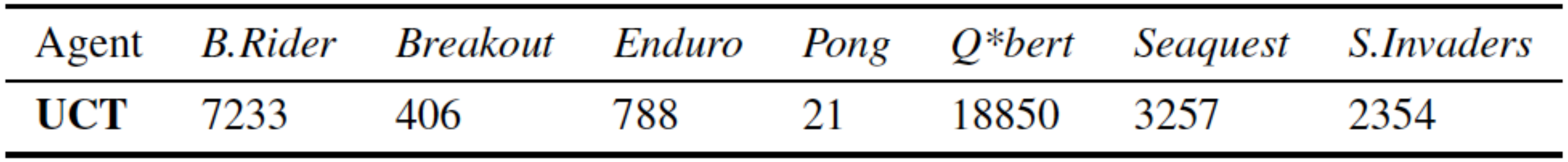
    1. Start from 200 runs with MCTS.
    2. Train the policy **UCTtoClassification**.
    3. Deploy the policy for 200 runs allowing 5% of the time a random action to be sampled.
    4. Use MCTS to decide best action for those states,
    5. GOTO 2
- At test time, just deploy the learnt policy.

Results



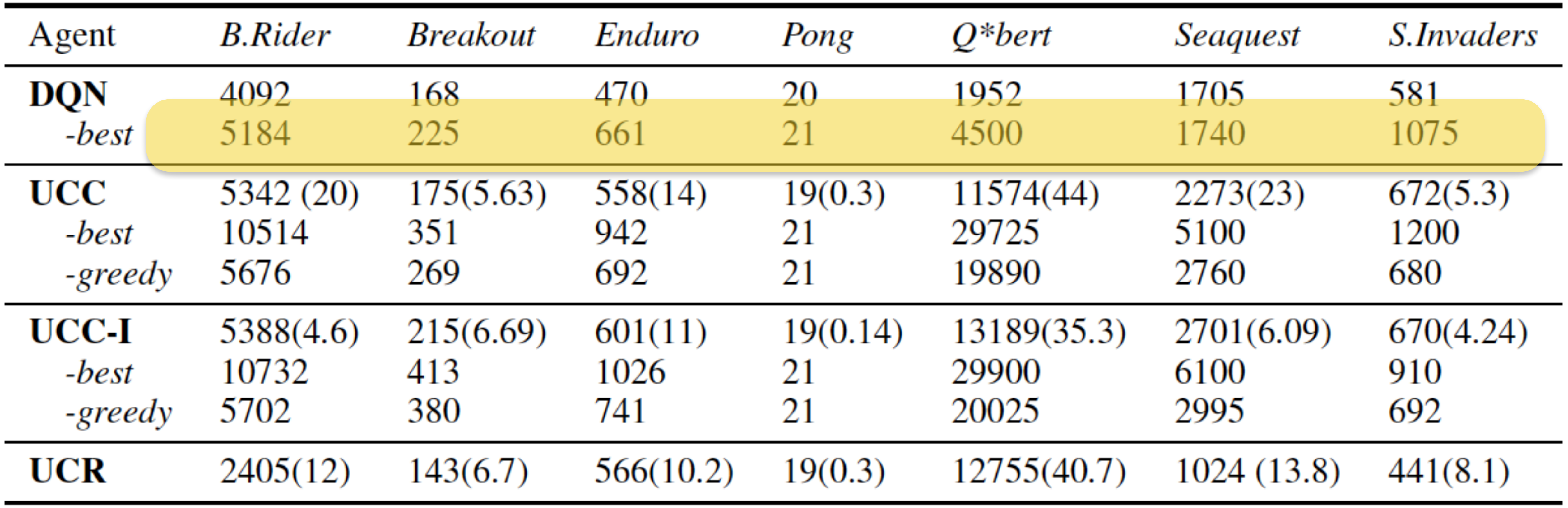
Agent	*B.Rider*	*Breakout*	*Enduro*	*Pong*	*Q*bert*	*Seaquest*	*S.Invaders*
DQN	4092	168	470	20	1952	1705	581
-best	5184	225	661	21	4500	1740	1075
UCC	5342 (20)	175(5.63)	558(14)	19(0.3)	11574(44)	2273(23)	672(5.3)
-best	10514	351	942	21	29725	5100	1200
-greedy	5676	269	692	21	19890	2760	680
UCC-I	5388(4.6)	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.



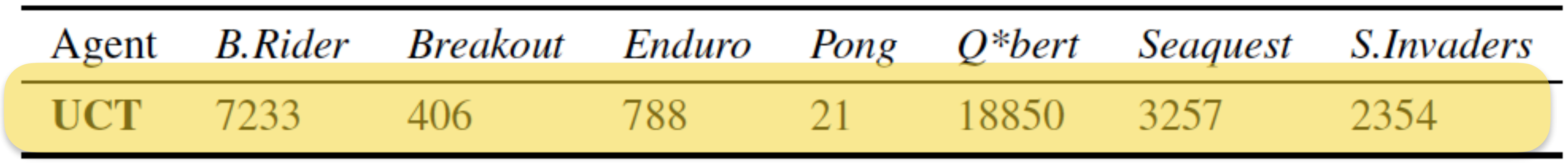
Agent	*B.Rider*	*Breakout*	*Enduro*	*Pong*	*Q*bert*	*Seaquest*	*S.Invaders*
UCT	7233	406	788	21	18850	3257	2354

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)

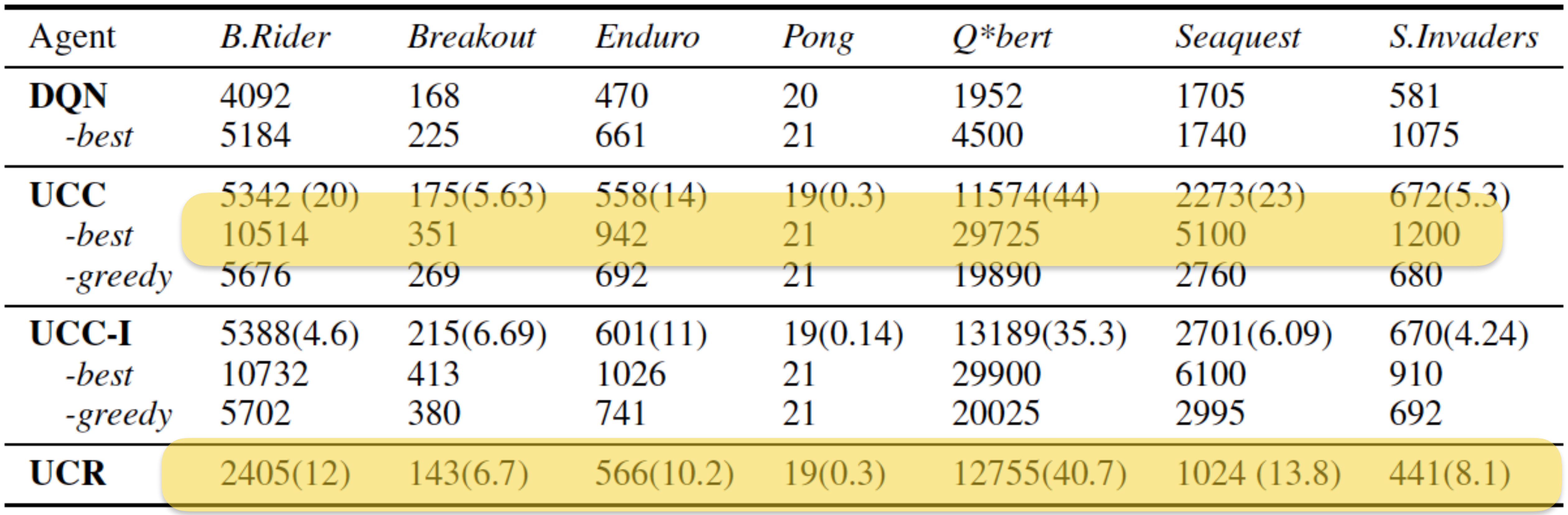
Table 2: Performance (game scores) of the off-line UCT game playing agent.



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

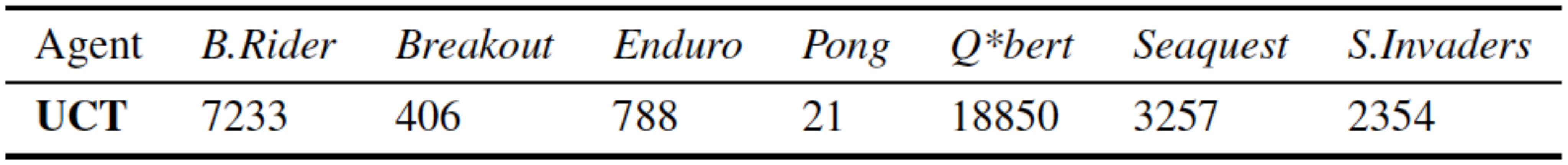
MCTS planning discovers better actions than deep Q learning. It takes though "a few days on a recent multicore computer to play for each game".

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)

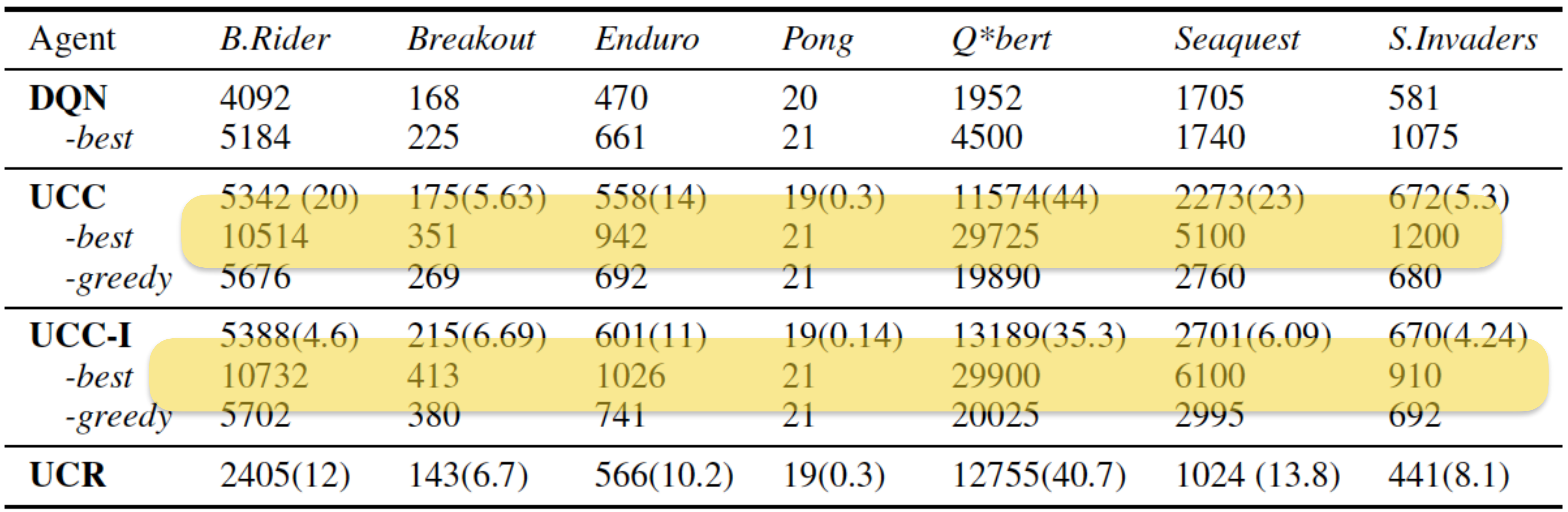
Table 2: Performance (game scores) of the off-line UCT game playing agent.



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

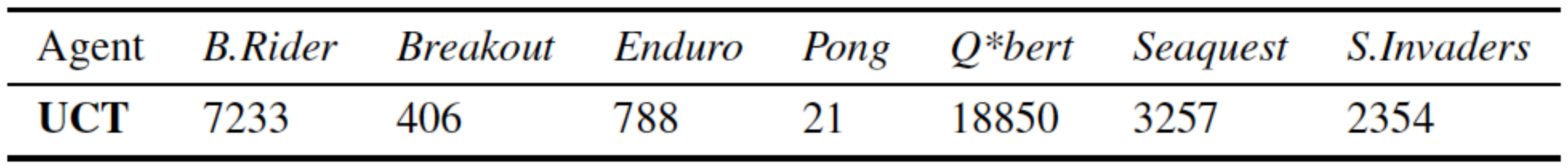
Classification is doing much better than regression! Indeed, we are training for exactly what we care about.

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)

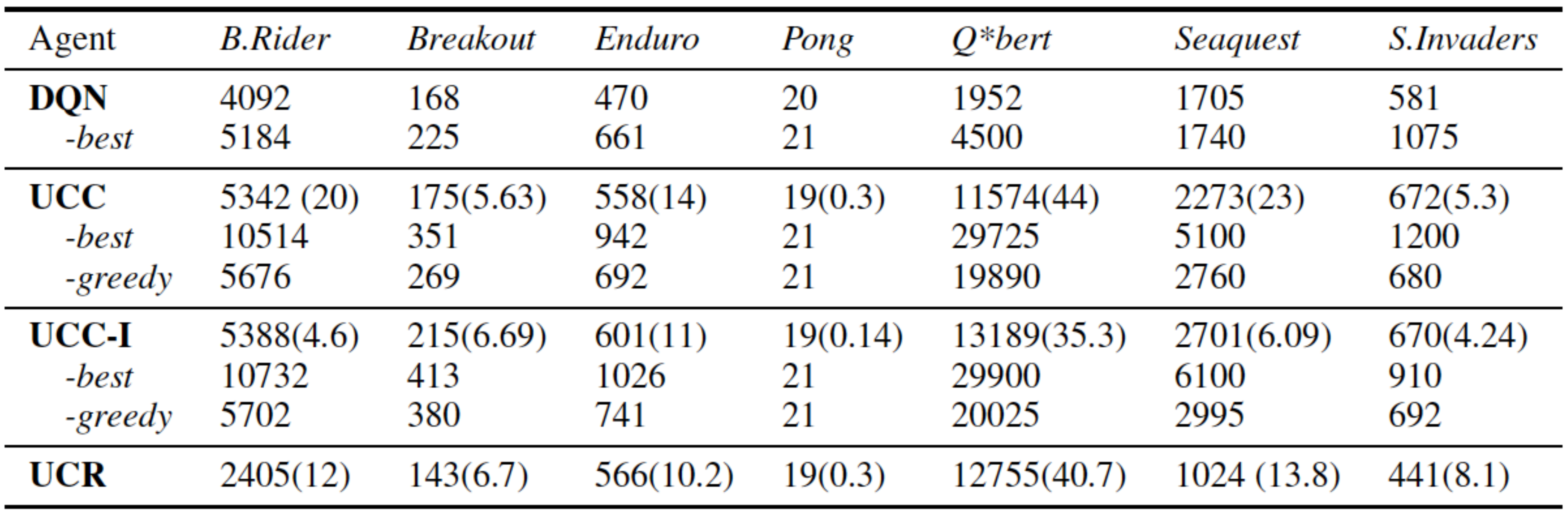
Table 2: Performance (game scores) of the off-line UCT game playing agent.



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

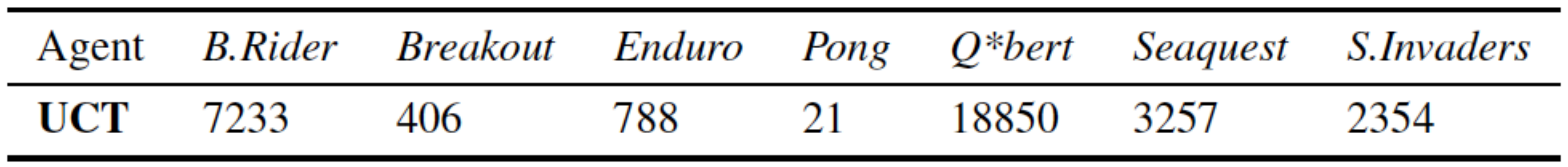
Interleaving is important to prevent mismatch between the training data and the data that the trained policy will see at test time.

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)

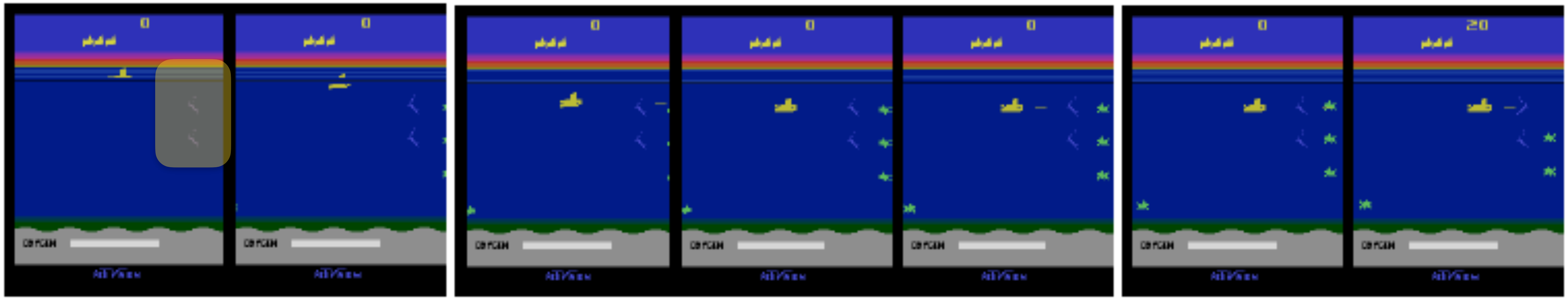
Table 2: Performance (game scores) of the off-line UCT game playing agent.



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

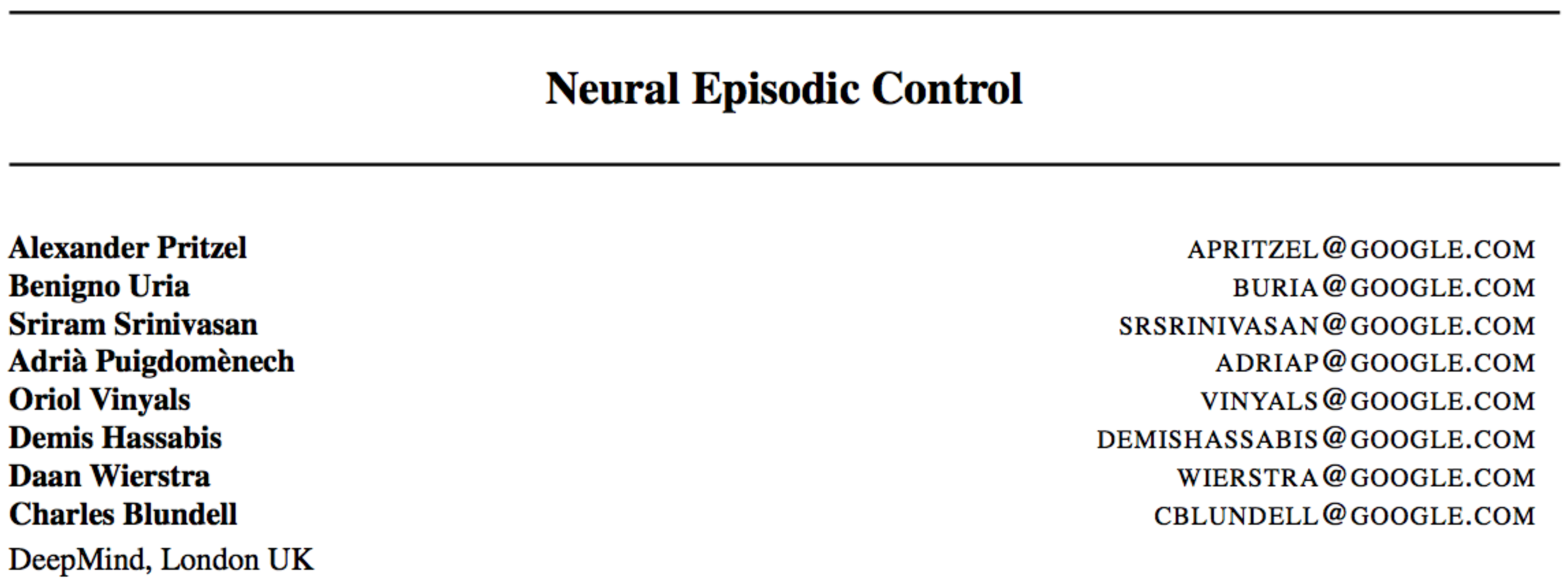
Results improve further if you allow MCTS planner to have more simulations and build more reliable Q estimates.

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.

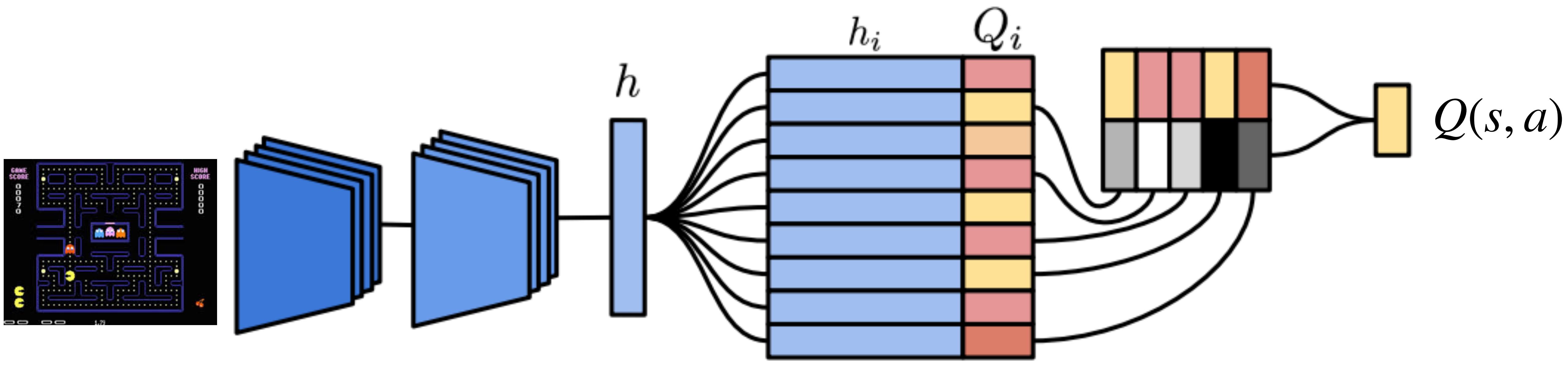


Neural Episodic Control

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

Interleaving of Q updates with real world experience collection: the experience we collect it takes long time till they are reflected to our Q values.

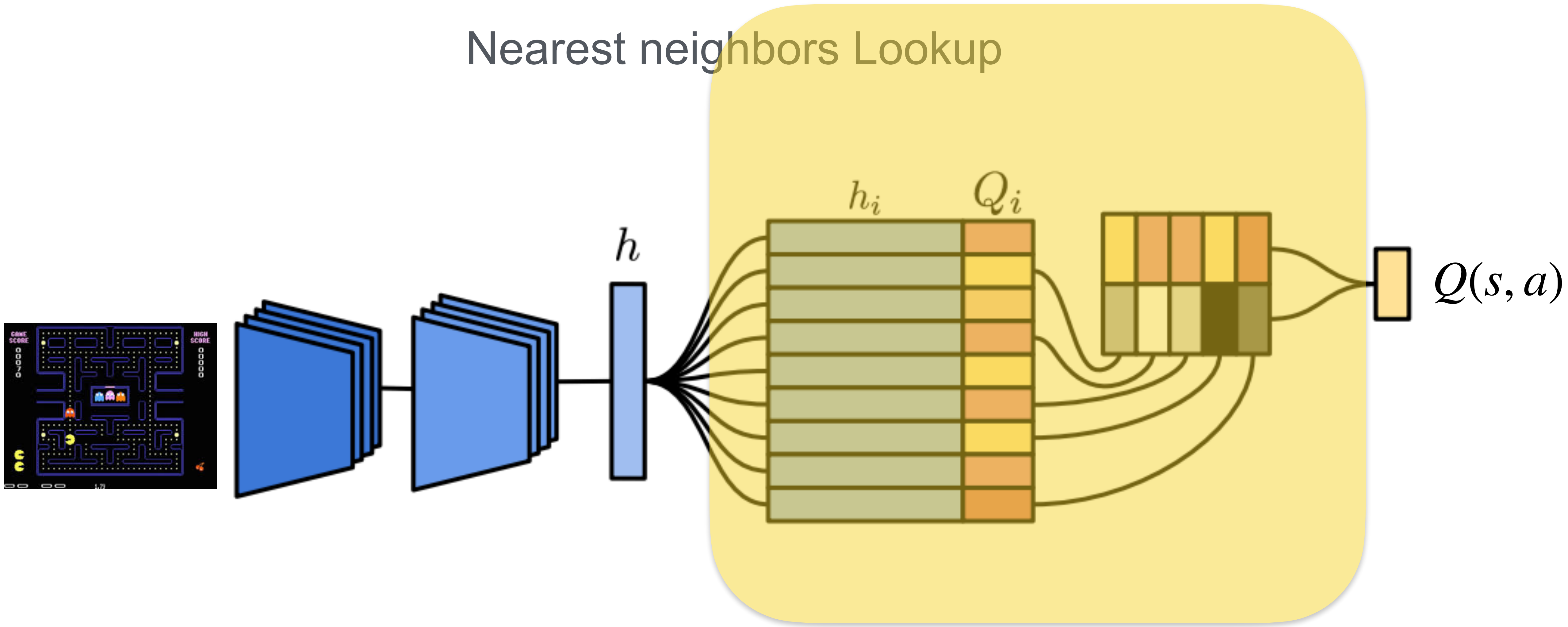
Image to Q value mapping through a nearest neighbor look-up



$$w_i = \frac{k(h, h_i)}{\sum_j k(h, h_j)}$$

$$Q(s, a) = \sum w_i Q_i$$

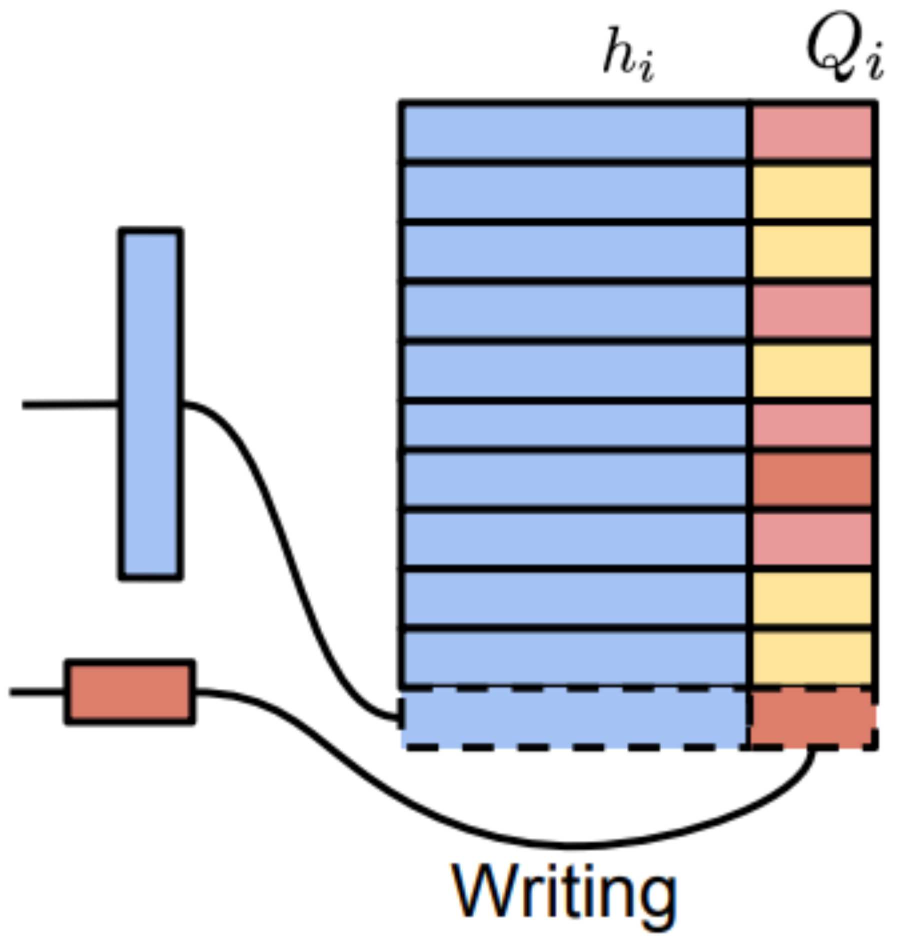
Image to Q value mapping through a nearest neighbor look-up



$$w_i = \frac{k(h, h_i)}{\sum_j k(h, h_j)}$$

$$Q(s,a) = \sum w_i Q_i$$

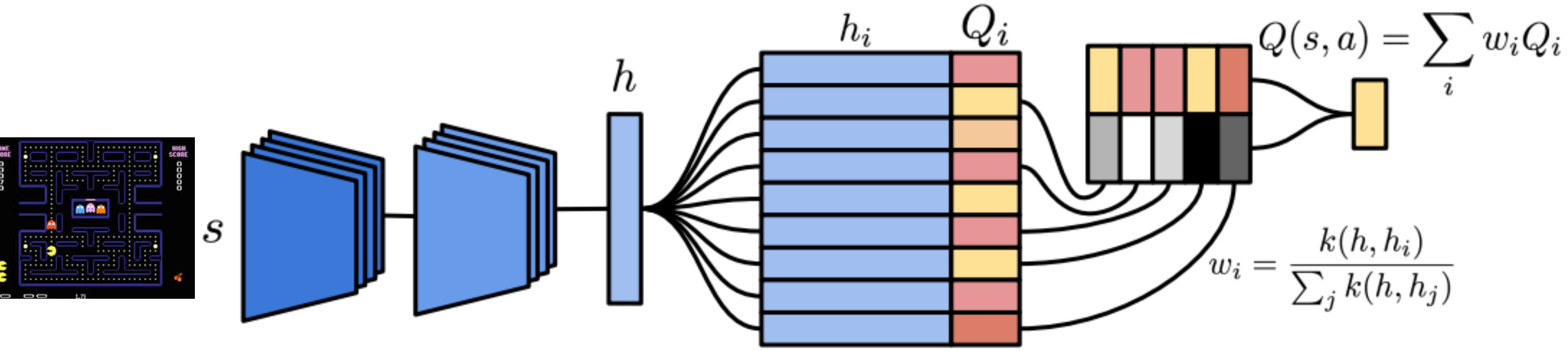
Writing in the memory



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

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


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

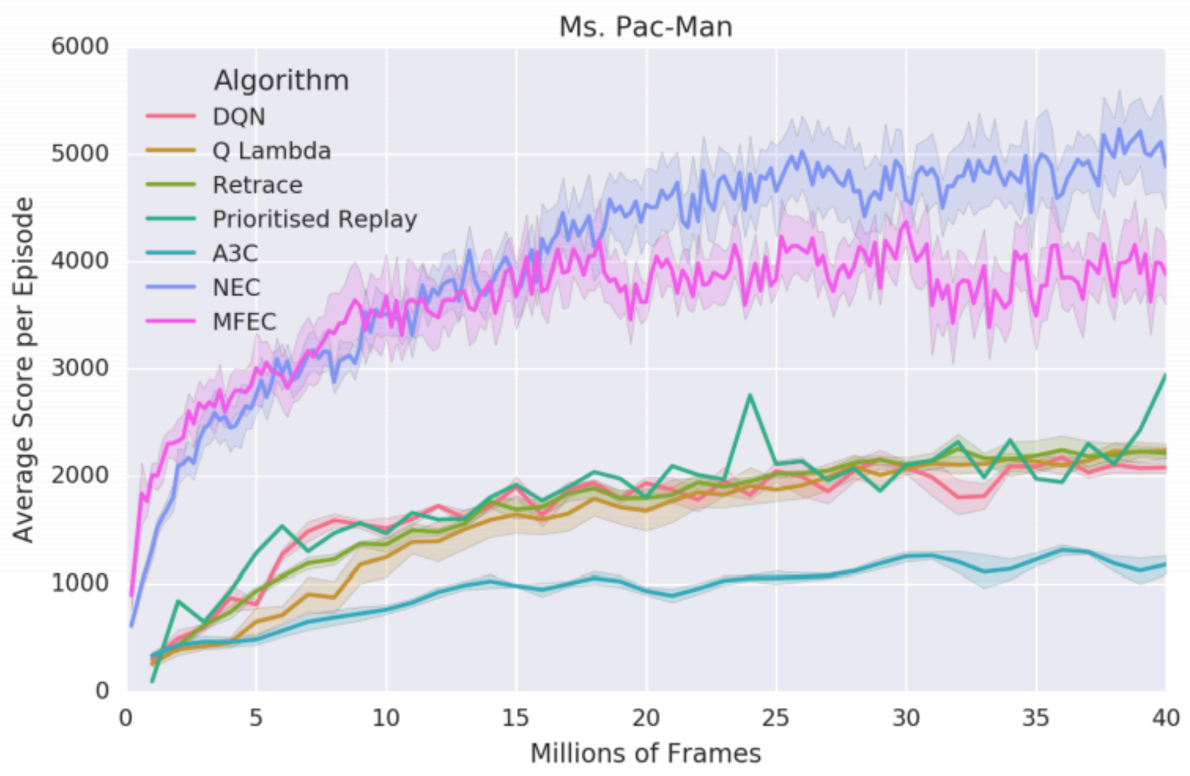
Algorithm 1 Neural Episodic Control

```
$\mathcal{D}$: 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**
```

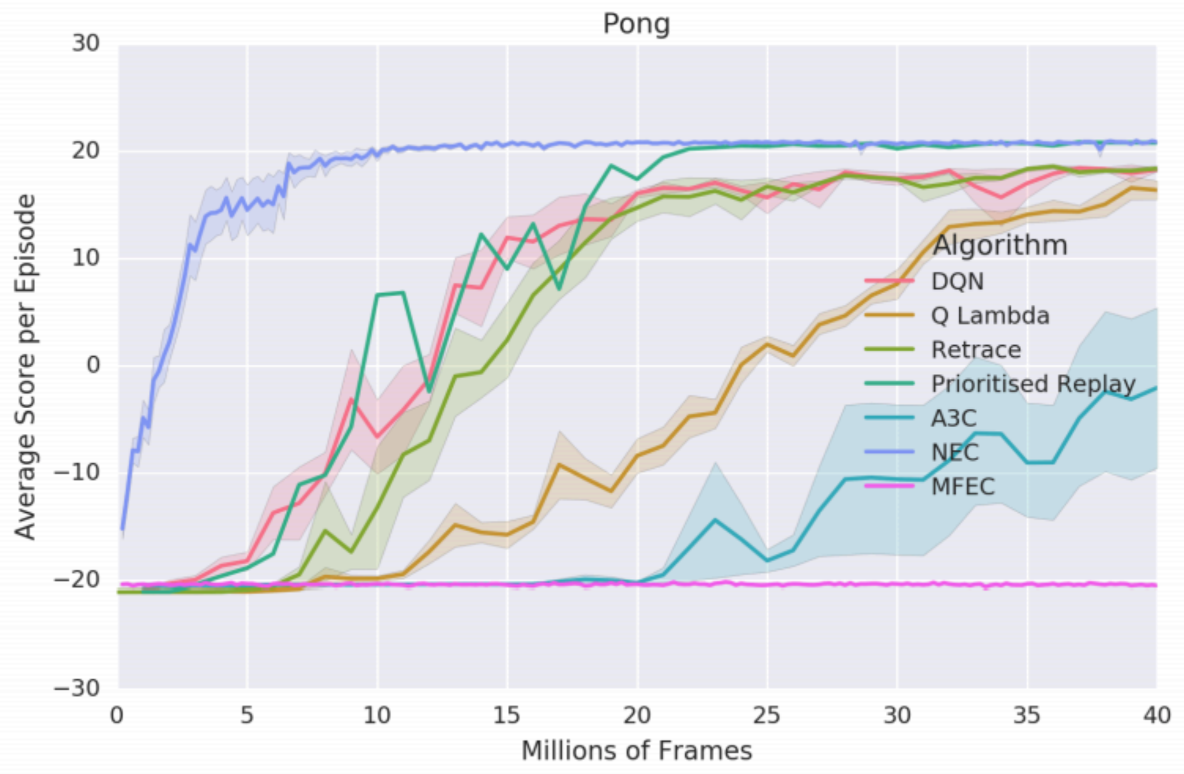
Frames	Nature DQN	Q*(λ)	Retrace(λ)	Prioritised Replay	A3C	NEC	MFEC
1M	-0.7%	-0.8%	-0.4%	-2.4%	0.4%	**16.7%**	12.8%
2M	0.0%	0.1%	0.2%	0.0%	0.9%	**27.8%**	16.7%
4M	2.4%	1.8%	3.3%	2.7%	1.9%	**36.0%**	26.6%
10M	15.7%	13.0%	17.3%	22.4%	3.6%	**54.6%**	45.4%
20M	26.8%	26.9%	30.4%	38.6%	7.9%	**72.0%**	55.9%
40M	52.7%	59.6%	60.5%	**89.0%**	18.4%	83.3%	61.9%

Table 1. Median across games of human-normalised scores for several algorithms at different points in training

Frames	Nature DQN	Q*(λ)	Retrace(λ)	Prioritised Replay	A3C	NEC	MFEC
1M	-10.5%	-11.7%	-10.5%	-14.4%	5.2%	**45.6%**	28.4%
2M	-5.8%	-7.5%	-5.4%	-5.4%	8.0%	**58.3%**	39.4%
4M	8.8%	6.2%	6.2%	10.2%	11.8%	**73.3%**	53.4%
10M	51.3%	46.3%	52.7%	71.5%	22.3%	**99.8%**	85.0%
20M	94.5%	135.4%	**273.7%**	165.2%	59.7%	121.5%	113.6%
40M	151.2%	**440.9%**	386.5%	332.3%	255.4%	144.8%	142.2%



Ms. Pac-Man
Algorithm
DQN
Q Lambda
Retrace
Prioritised Replay
A3C
NEC
MFEC
Average Score per Episode
Millions of Frames



Pong
Algorithm
DQN
Q Lambda
Retrace
Prioritised Replay
A3C
NEC
MFEC
Average Score per Episode
Millions of Frames