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

\[ \pi^*(s) = \arg\max_a Q^*(s, a) \]

- 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 $w$

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

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 \( r + \gamma \max_{a'} Q(s', a', w) \) as a target

- Minimize MSE loss by stochastic gradient descent

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

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

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

- Minimize MSE loss by stochastic gradient descent

\[ l = \left( r + \gamma \max_a Q(s', a', w) - Q(s, a, w) \right)^2 \]
Q-Learning: Off-Policy TD Control

- One-step Q-learning:

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

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

Repeat (for each episode):
  Initialize \( S \)
  Repeat (for each step of episode):
    Choose \( A \) from \( S \) using policy derived from \( Q \) (e.g., \( \varepsilon \)-greedy)
    Take action \( A \), observe \( R, S' \)
    \[
    Q(S, A) \leftarrow Q(S, A) + \alpha \left[ R + \gamma \max_a Q(S', a) - Q(S, A) \right]
    \]
    \( S \leftarrow S' \);
  until \( S \) is terminal
Q-Learning with FA

- Minimize MSE loss by stochastic gradient descent

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

- 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

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

- 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

<table>
<thead>
<tr>
<th>$s_1, a_1, r_2, s_2$</th>
<th>$s, a, r, s'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_2, a_2, r_3, s_3$</td>
<td>$s_t, a_t, r_{t+1}, s_{t+1}$</td>
</tr>
<tr>
<td>$s_3, a_3, r_4, s_4$</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

- Sample experiences from data-set and apply update

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

- To deal with non-stationarity, target parameters $w^-$ are held fixed
Experience Replay

- Given experience consisting of \( \langle \text{state, value} \rangle \), or \( \langle \text{state, action/value} \rangle \) pairs

\[
D = \{ \langle s_1, v_1^\pi \rangle, \langle s_2, v_2^\pi \rangle, ..., \langle s_T, v_T^\pi \rangle \}
\]

- Repeat
  - Sample state, value from experience
    \[
    \langle s, v^\pi \rangle \sim D
    \]
  - Apply stochastic gradient descent update
    \[
    \Delta w = \alpha (v^\pi - \hat{v}(s, w)) \nabla_w \hat{v}(s, w)
    \]
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

\[
\mathcal{L}_i(w_i) = \mathbb{E}_{s,a,r,s' \sim D_i} \left[ \left( r + \gamma \max_{a'} Q(s', a'; w_i^-) - Q(s, a; w_i) \right)^2 \right]
\]

- 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/

Network architecture and hyperparameters fixed across all games

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. $Q(s,\text{argmax}_a(Q(s,a))$ is positive while $q(s,\text{argmax}_a(q(s,a)))=0$. 

![Graph](image-url)
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:
  \[
  Q_1(S_t, A_t) \leftarrow Q_1(S_t, A_t) + \\
  + \alpha \left( R_{t+1} + Q_2(S_{t+1}, \arg\max_a Q_1(S_{t+1}, a)) - Q_1(S_t, A_t) \right)
  \]
- Action selections are $\varepsilon$-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:

\[
Q_1(S_t, A_t) \leftarrow Q_1(S_t, A_t) + \alpha \left( R_{t+1} + Q_2(S_{t+1}, \arg\max_a Q_1(S_{t+1}, a)) - Q_1(S_t, A_t) \right)
\]

- Action selections are $\varepsilon$-greedy with respect to the sum of $Q_1$ and $Q_2$
Double Tabular Q-Learning

Initialize $Q_1(s, a)$ and $Q_2(s, a), \forall s \in S, a \in A(s)$, arbitrarily

Initialize $Q_1(terminal-state, \cdot) = Q_2(terminal-state, \cdot) = 0$

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., $\varepsilon$-greedy in $Q_1 + Q_2$)
    Take action $A$, observe $R, S'$
    With 0.5 probability:
      $Q_1(S, A) \leftarrow Q_1(S, A) + \alpha \left( R + \gamma Q_2(S', \text{argmax}_a Q_1(S', a)) - Q_1(S, A) \right)$
    else:
      $Q_2(S, A) \leftarrow Q_2(S, A) + \alpha \left( R + \gamma Q_1(S', \text{argmax}_a Q_2(S', a)) - Q_2(S, A) \right)$
    $S \leftarrow S'$;
  until $S$ is terminal
Double Deep Q-Learning

- Current Q-network $w$ is used to select actions
- Older Q-network $w^-\text{ is used to evaluate actions}$

Action evaluation: $w^-$

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

Action selection: $w$

van Hasselt, Guez, Silver, 2015
Prioritized Replay

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

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

- Stochastic Prioritization

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

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

Schaul, Quan, Antonoglou, Silver, ICLR 2016
Multistep Returns

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

- Multistep Q-learning update rule:
  \[
  I = \left( R_t^{(n)} + \gamma_t^{(n)} \max_{a'} Q(S_{t+n}, a', w) - Q(s, a, w) \right)^2
  \]

- Singlestep Q-learning update rule:
  \[
  I = \left( r + \gamma \max_a {Q(s', a', w)} - Q(s, a, w) \right)^2
  \]
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

![Graph showing the performance of different algorithms over millions of frames.](image)
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

# games > 20% human  
# games > 50% human  
# games > 100% human  
# games > 200% human

# games > 500% human

<table>
<thead>
<tr>
<th>Algorithm</th>
<th># games &gt; 20% human</th>
<th># games &gt; 50% human</th>
<th># games &gt; 100% human</th>
<th># games &gt; 200% human</th>
<th># games &gt; 500% human</th>
</tr>
</thead>
<tbody>
<tr>
<td>DQN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DDQN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prioritized DDQN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dueling DDQN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A3C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Noisy DQN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rainbow</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Algorithm</th>
<th># games &gt; 20% human</th>
<th># games &gt; 50% human</th>
<th># games &gt; 100% human</th>
<th># games &gt; 200% human</th>
<th># games &gt; 500% human</th>
</tr>
</thead>
<tbody>
<tr>
<td>no double</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>no priority</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>no dualing</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>no multi-step</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>no distribution</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>no noisy</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rainbow</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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

---

![Graph showing the performance of Rainbow and various other methods over millions of frames.](image-url)
Imagine we have access to the internal state of the Atari simulator. Would online planning (e.g., using MCTS), outperform the trained DQN policy?
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.
Then why do we 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.
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?*
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
Sample actions according to the following score:

- 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
Monte-Carlo Tree Search

Gradually grow the search tree:

1. **Iterate Tree-Walk**
   - **Building Blocks**
     - **Select next action**
     - **Add a node**
     - **Grow a leaf of the search tree**
     - **Select next action bis**
     - **Random phase, roll-out**
     - **Compute instant reward**
     - **Evaluate**
     - **Update information in visited nodes**
     - **Propagate**

2. **Returned solution**:

   - **Path visited most often**

---

**Basic MCTS pseudocode**

```python
def 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)
```

---

**Diagram**

- **Search Tree**
- **Explored Tree**
Basic MCTS pseudocode

```python
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)
```

Bandit-Based Phase

Search Tree

Explored Tree
Basic MCTS pseudocode

```python
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)
```
Monte-Carlo Tree Search

Gradually grow the search tree:

I Iterative Tree-Walk

I Building Blocks

I Select next action

Bandit phase

I Add a node

Grow a leaf of the search tree

I Select next action bis

Random phase, roll-out

I Compute instant reward

Evaluate

I Update information in visited nodes

Propagate

Returned solution:

I Path visited most often

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)
Monte-Carlo Tree Search

Kocsis Szepesvári, 06

Gradually grow the search tree:

I Iterative Tree-Walk

I Building Blocks

I Select next action

Bandit phase

I Add a node

Grow a leaf of the search tree

I Select next action bis

Random phase, rollout

I Compute instant reward

Evaluate

I Update information in visited nodes

Propagate

Returned solution:

I Path visited most often

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**

```python
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)
```

![Diagram of MCTS search tree with Bandit-Based Phase and Explored Tree](image-url)
Monte-Carlo Tree Search

Gradually grow the search tree:
- Iteration Tree-Walk
- Building Blocks
  - Select next action
  - Bandit phase
    - Add a node
    - Grow a leaf of the search tree
    - Select next action bis
  - Random phase, roll-out
    - Compute instant reward
    - Evaluate
    - Update information in visited nodes
    - Propagate

- Returned solution:
  - Path visited most often

Basic MCTS pseudocode

```python
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

```python
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)
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

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

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

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

<table>
<thead>
<tr>
<th>Agent</th>
<th>B.Rider</th>
<th>Breakout</th>
<th>Enduro</th>
<th>Pong</th>
<th>Q*bert</th>
<th>Seaquest</th>
<th>S.Invaders</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCT</td>
<td>7233</td>
<td>406</td>
<td>788</td>
<td>21</td>
<td>18850</td>
<td>3257</td>
<td>2354</td>
</tr>
</tbody>
</table>

Results
Results

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

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

Classification is doing much better than regression! indeed, we are training for exactly what we care about.
Interleaving is important to prevent mismatch between the training data and the data that the trained policy will see at test time.

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

<table>
<thead>
<tr>
<th>Agent</th>
<th>B.Rider</th>
<th>Breakout</th>
<th>Enduro</th>
<th>Pong</th>
<th>Q*bert</th>
<th>Seaquest</th>
<th>S.Invaders</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCT</td>
<td>7233</td>
<td>406</td>
<td>788</td>
<td>21</td>
<td>18850</td>
<td>3257</td>
<td>2354</td>
</tr>
</tbody>
</table>
Results

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

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

<table>
<thead>
<tr>
<th>Agent</th>
<th>B.Rider</th>
<th>Breakout</th>
<th>Enduro</th>
<th>Pong</th>
<th>Q*bert</th>
<th>Seaquest</th>
<th>S.Invaders</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCT</td>
<td>7233</td>
<td>406</td>
<td>788</td>
<td>21</td>
<td>18850</td>
<td>3257</td>
<td>2354</td>
</tr>
</tbody>
</table>

Results improve further if you allow MCTS planner to have more simulations and build more reliable Q estimates.
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.
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
Benigno Uria
Sriram Srinivasan
Adrià Puigdomènech
Oriol Vinyals
Demis Hassabis
Daan Wierstra
Charles Blundell
DeepMind, London UK

APRITZEL@GOOGLE.COM
BURIA@GOOGLE.COM
SRSRINIVASAN@GOOGLE.COM
ADRIAP@GOOGLE.COM
VINYALS@GOOGLE.COM
DEMISHASSABIS@GOOGLE.COM
WIERSTRA@GOOGLE.COM
CBLUNDELL@GOOGLE.COM
Nearest neighbors Lookup

\[ Q(s, a) = \sum_i w_i Q_i \]

\[ w_i = \frac{k(h, h_i)}{\sum_j k(h, h_j)} \]
If identical key $h$ present:

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

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

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**

- **D**: replay memory.
- **Mₐ**: 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 \( D \).
    end for
end for
```
Algorithm 1 Neural Episodic Control

\( D \): replay memory.
\( M_a \): a DND for each action \( a \).
\( N \): horizon for \( N \)-step \( Q \) estimate.

\textbf{for} each episode \textbf{do}

\hspace{1em} \textbf{for} \( t = 1, 2, \ldots, T \) \textbf{do}

\hspace{2em} Receive observation \( s_t \) from environment with embedding \( h \).

\hspace{2em} Estimate \( Q(s_t, a) \) for each action \( a \) via (1) from \( M_a \)

\hspace{2em} Take action \( a_t \), receive reward \( r_{t+1} \)

\hspace{2em} Append \( (h, Q^{(N)}(s_t, a_t)) \) to \( M_{a_t} \).

\hspace{2em} Append \( (s_t, a_t, Q^{(N)}(s_t, a_t)) \) to \( D \).

\hspace{2em} Train on a random minibatch from \( D \).

\textbf{end for}

\textbf{end for}

\[
Q(s_t, a) = \sum_i w_i Q_i
\]

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

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

\[
-\frac{1}{2} \nabla_w J(w) = (q_\pi(S, A) - \hat{q}(S, A, w)) \nabla_w \hat{q}(S, A, w)
\]

\[
\Delta w = \alpha (q_\pi(S, A) - \hat{q}(S, A, w)) \nabla_w \hat{q}(S, A, w)
\]