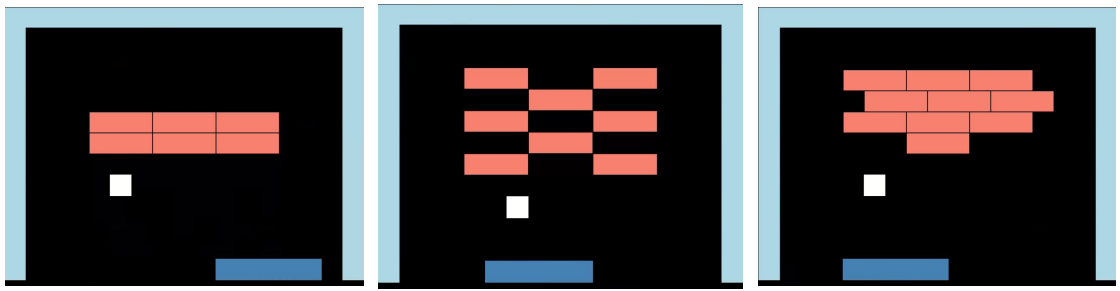
Reinforcement Learning

Fischer Hauke Edgar, Krofitsch Christoph, Miksa Maciej

If this is the PDF version, please open https://docs.google.com/presentation/d/1SAE31VG7jB05t5X5YeK59o1jNZiWk4rsBrvVC1mrzL4/edit?usp=sharing now to see the full videos! Thank you

Breakout Game implementation

- **Attention**: To run the application, please install Turtle/Tkinter for Python3 on your machine!
- Developed via different classes representing game elements:
  - paddle.py
  - bricks.py / brick.py
  - ball.py - contains main collision detections
  - main.py - currently starts the game in user-control mode. It can also be used to start the game in AI-control mode
- We chose a 15x11 game field
- 3 different brick layouts we used to train and evaluate AI:



Used Reinforcement Learning Approaches

1. Monte Carlo Control: First-Visit Exploring Starts
2. Monte Carlo Control: Every-Visit On-Policy (using ε-soft policies)
3. Monte Carlo Control: Every-Visit Off-Policy

- We defined one episode to be equal to game. The episode ends when the game was won, or lost.
- -1 reward for each step
- -250 reward for a lost game
- ES learned with 1M and 10M episodes; On/Off-Policy with 1M Episodes
- Initial ball velocity was random during learning; however videos show it one after another

Main RL class: TabularRL | Basic structure:

- Constructs a Tabular object to save the "**Q table**" (current state-action values) shared by all algorithm implementations

  Dimensions: HGrid x VGrid x 5 x 2 x HGrid-4 x 5 x 2^Nx3

  HGrid = horizontal # grid points, VGrid = vertical # grid points,

  5 = vx speed of ball, 2 = vy speed of ball, HGrid-4 = x-pos of paddle

  5 = vx speed of paddle, 2^N does brick exist y/n, 3 = action taken

- The two main methods serving as an **interface** to the game are the **set_state** and **get_state** methods. **get_state** retrieves the correct indices for the **Q table** corresponding to the current state of the game, with **set_state** you can determine the state of the game (only relevant for Exploring Starts method).

Monte Carlo ES

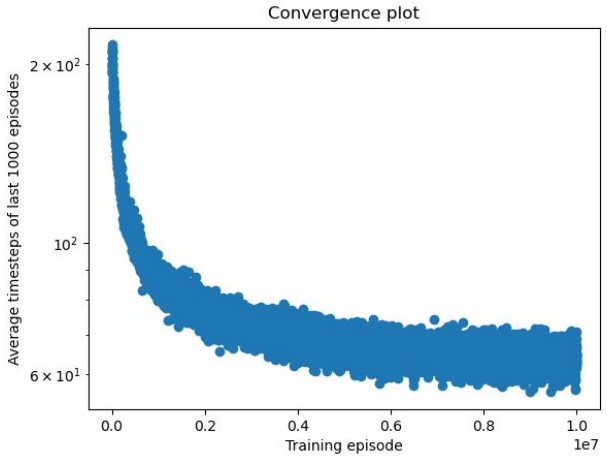
Monte Carlo ES (Exploring Starts)

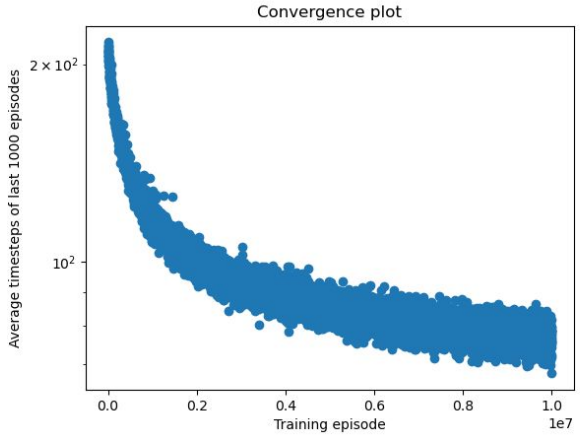
Initialize:
$\pi(s) \in \mathcal{A}(s)$ (arbitrarily), for all $s \in \mathcal{S}$
$Q(s,a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$
$Returns(s,a) \leftarrow$ empty list, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

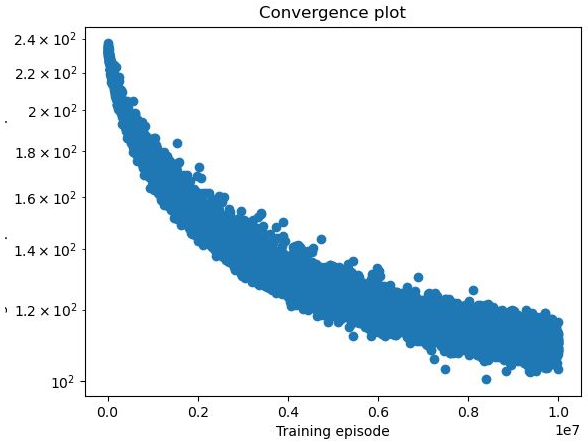
Loop forever (for each episode):
Choose $S_0 \in \mathcal{S}$, $A_0 \in \mathcal{A}(S_0)$ randomly such that all pairs have probability > 0
Generate an episode from S_0, A_0, following π: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
$G \leftarrow 0$
Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
$G \leftarrow \gamma G + R_{t+1}$
Unless the pair S_t, A_t appears in $S_0, A_0, S_1, A_1 \ldots, S_{t-1}, A_{t-1}$:
Append G to $Returns(S_t, A_t)$
$Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$

The first algorithm we used was **MC exploring starts**. To lower memory consumption no returns table was used, but rather a **counter table** that counts which state occurred how often. This number can be used to update average returns after each episode, without saving all previous returns.

Convergence:







Left: 6 Bricks, Middle: 8 Bricks, Right: 10 Bricks. Grid: 15 x 11

- A lost episodes counts as -250 return
- Episodes are capped at 10000 time steps
- Only on the smallest structure does the algorithm get close to its optimum
- It requires a huge number of episodes (millions) to reach decent performance

Monte Carlo On-Policy

- More realistic, as game is properly started from the beginning, and no artificial game states have to be created
- Using ε-soft policies with ε=0.02
- Learned initial wins really quick, however further improvements took too long
- Wasn't able to win all games after 1M learning episodes

Monte Carlo Off-Policy

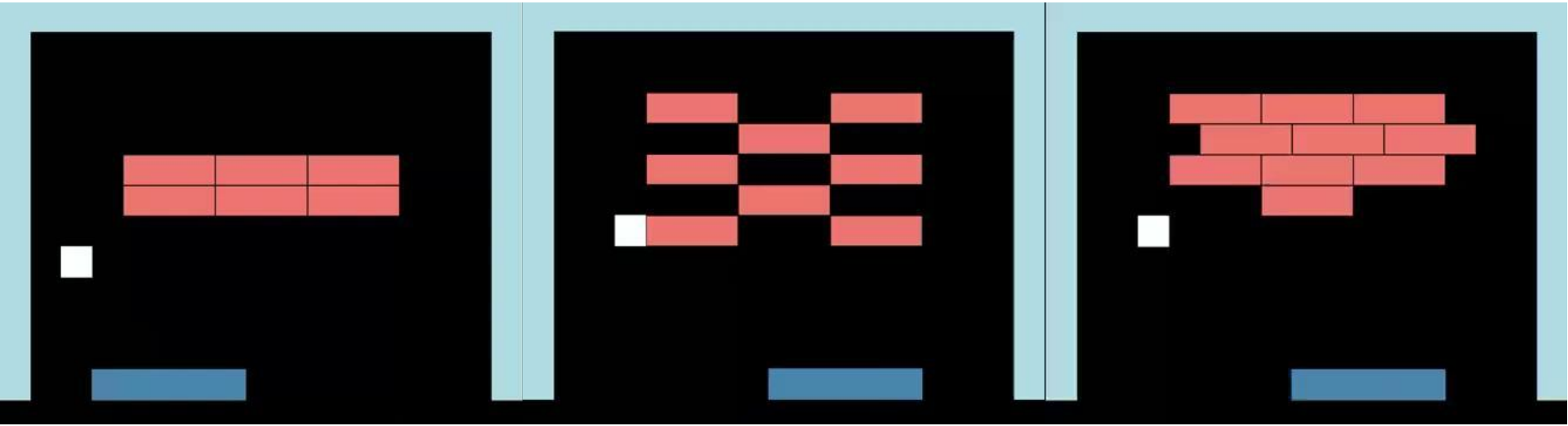
- Performed better than On-Policy
- Separation into target policy and behavior policy

- Using weighted importance sampling: $V(s) \doteq \frac{\sum_{t \in \mathcal{T}(s)} \rho_{t:T(t)-1} G_t}{\sum_{t \in \mathcal{T}(s)} \rho_{t:T(t)-1}},$

- in combination with a completely random behavior policy (P(action) = ⅓ for all actions, for all states)
- Managed to win all games and was overall the best performer among all methods

convergence plots (for On-Policy and Off-Policy) available upon request

Monte Carlo Off-Policy playing game after 1M episodes



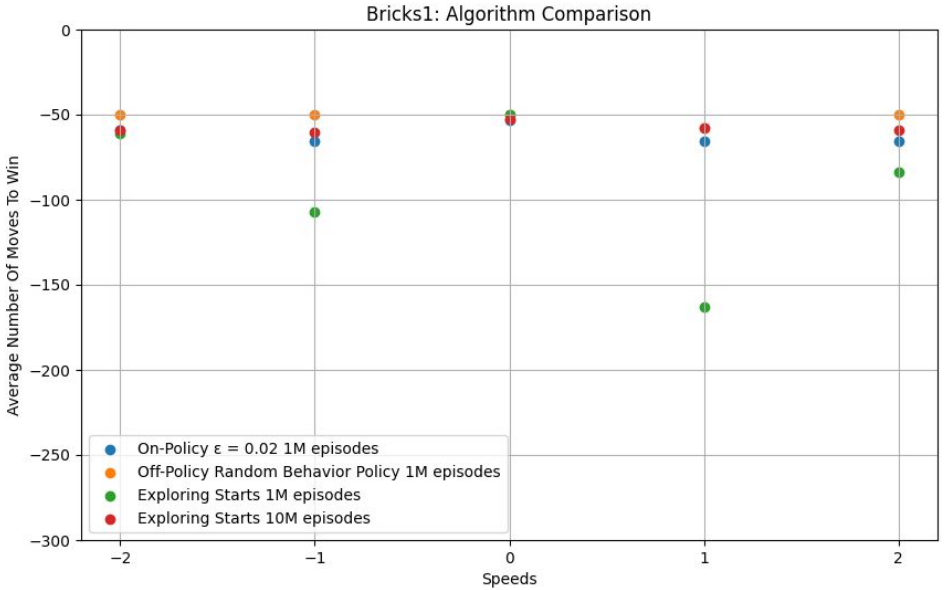
Brick layout 1

Brick layout 2

Brick layout 3

If you have the PDF version, use the link on the front page to access the videos online by clicking on them

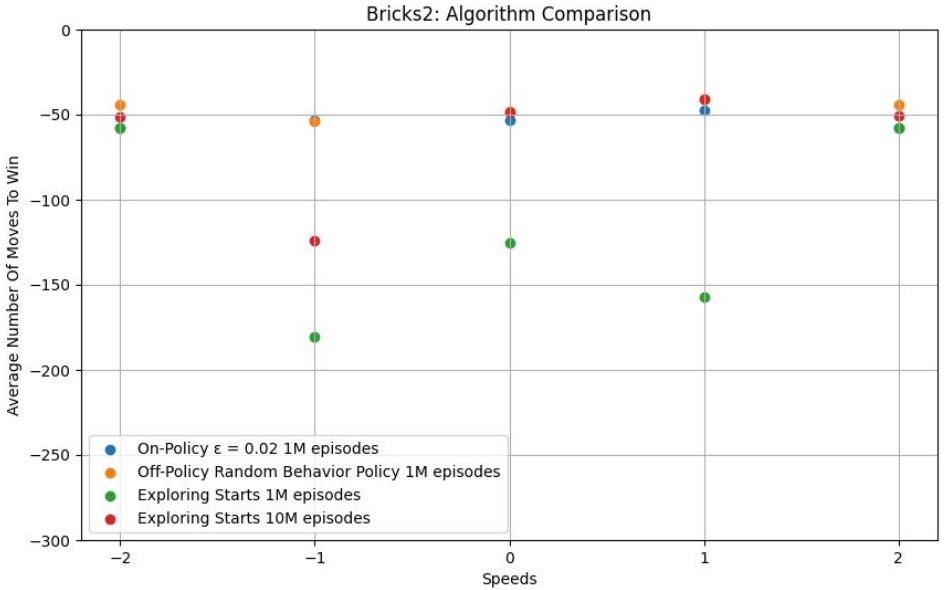
Algorithms Comparison



Bricks Layout 1:

On-Policy (1M episodes ε = 0.02): The average performance is good as the number of moves ranges from around -65 to -53, with no significant variation based on speed, showing consistency.
Off-Policy Random Behavior Policy (1M episodes): The average return is consistently around -50 for all speeds, making it the best performing algorithm in this layout across all speeds.
Exploring Starts (1M episodes): The average return varies from approximately -163 to -50, it is rather inconsistent across different speeds and the results are on average worse than other algorithms, we can clearly see that this variation of Exploring Starts algorithm struggled here, but in the end managed to win the game.
Exploring Starts (10M episodes): The average performance relatively good and consistent, ranging from around -60 to -52, with no significant variation based on speed.

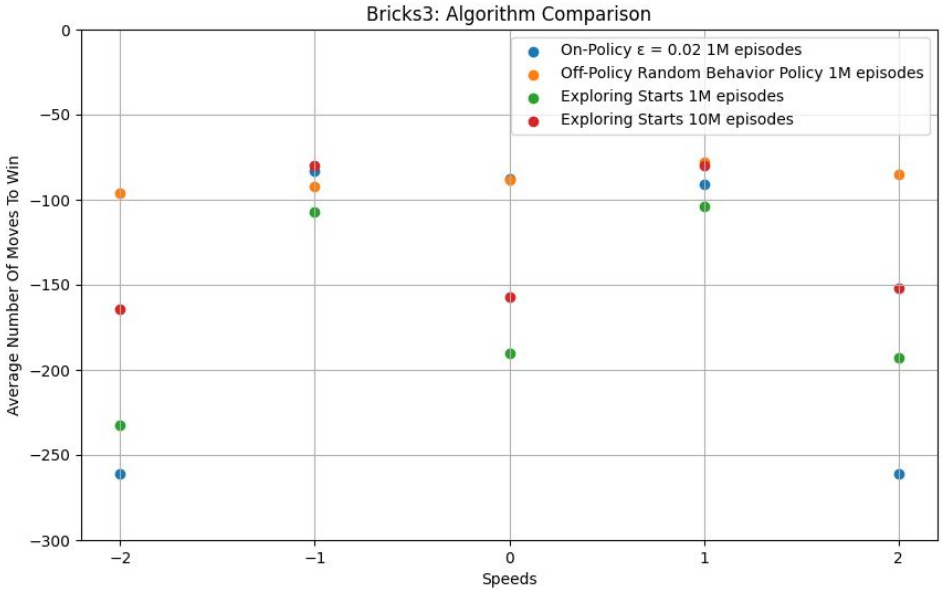
Algorithms Comparison



Bricks Layout 2:

On-Policy (1M episodes ε = 0.02): The average return ranges from approximately -58 to -47, with no clear trend based on speed.
Off-Policy Random Behavior Policy (1M episodes): The average return varies from around -54 to -41, with no significant variation based on speed.
Exploring Starts (1M episodes): The average return shows significant variation, ranging from approximately -180 to -57, with no clear trend based on speed.
Exploring Starts (10M episodes): The average return stays around -50, with exception for speed -1 where the return is a much worse of -125.
We can observe than Exploring Starts Algorithm struggled here in both variations in comparison to On-Policy and Off-Policy, with 10M episodes variation showing a bit better and more consistent performance.

Algorithms Comparison



Bricks Layout 3:

On-Policy (1M episodes ε = 0.02): Algorithm is highly influenced by speed, for speeds -2 and 2. The performance is pretty bad, as the algorithm does not even win the game with speed -2 and 2 and loses after on average 11 moves.
Off-Policy Random Behavior Policy (1M episodes): The average return ranges from around -96 to -78, with no significant variation based on speed. This algorithm performs relatively good and yield consistent results overall.
Exploring Starts(1M episodes): The average return varies from approximately -232 to -104. It is rather inconsistent and the performance is mediocre, especially with speeds -2 and 2.
Exploring Starts (10M episodes): The average return ranges from approximately -164 to -80, with no significant variation based on speed. It performs much better than its 1M episodes variation.