Playing Sudoku with Reinforcement Learning

PA026: Artificial Intelligence Project

Karmazin Vasilii, učo 540500

Faculty of Informatics, Masaryk University Brno

June 6, 2024

1 Introduction

Reinforcement Learning (RL) shines at playing complex games by learning optimal strategies through trial and error. This project applies Reinforcement Learning to Sudoku, a challenging deductive puzzle that requires filling a 9x9 grid with numbers 1 to 9 so that each row, column, and 3x3 subgrid contains all digits exactly once.

Sudoku puzzles range from easy to hard; some can be solved by applying basic Sudoku rules, while others require complex strategies. Additionally, difficult puzzles are impossible to solve right away and require predicting several moves ahead.

The project's objective is to explore how well an RL-trained deep neural network can learn to solve Sudoku puzzles, demonstrating the potential of RL in handling deductive reasoning tasks.

The project code and running instructions are available at GitLab:
`https://gitlab.fi.muni.cz/xkarmaz/sudoku-rl`

2 Related Work

Numerous methods to solve Sudoku puzzles vary based on the puzzle's complexity. Simple brute-force approaches involve trying all possible combinations, which can be highly inefficient given the vast number of potential combinations. Specifically, the total number of possible Sudoku grids is approximately 6.67×10^{21}, making brute-force solutions impractical from algorithmic view and requiring more sophisticated approaches.

Professional human solvers typically use the following algorithm:

1. Scan the grid for known patterns.
2. Apply heuristic to find a pattern by filling digit or removing possible candidates.
3. Starting with the simplest ones repeat the process until no patterns are found.
4. If no patterns were found, the player tries to predict moves or look for implicit hints, such as grid symmetry or puzzle-specific patterns.

Modern systems for solving Sudoku puzzles, like SudokuSolver [1], operate on a similar algorithmic approach. SudokuSolver [1] incorporates 39 different patterns and heuristics for solving puzzles. This method is significantly faster than complete brute-force searches and can also evaluate the difficulty of a puzzle based on the heuristics used.

Sudoku puzzles can be framed as a Constraint Satisfaction Problem (CSP), utilizing algorithms from that domain, or as an optimization problem, leveraging machine learning techniques. Publicly available solvers such as Solving Sudoku with Neural Networks [2] use Convolutional Neural Networks (CNNs) and achieve good results. Projects like SudokuAI experiment with CNNs and Multilayer Perceptrons (MLPs), and other projects try Long Short-Term Memory Networks (LSTMs) or custom heuristics. However, there are few publicly available projects to solve Sudoku using reinforcement learning. The closest work to our approach is the recent project SudokuRL [3], whose status is currently unknown. The lack of available information on that topic encourages us to conduct RL experiments ourselves.

3 Reinforcement Learning

Reinforcement Learning involves an agent interacting with an environment to maximize cumulative rewards. The agent, observing the state (s) of the environment, takes actions (a) that influence the state and receive feedback in the form of rewards (r). The goal is to learn a policy, $\pi(a|s)$, that optimally balances exploration and exploitation to maximize the expected sum of rewards over time. Figure 1

3.1 Tabular Q-Learning

Tabular Q-Learning is a model-free RL algorithm that learns the value of state-action pairs. The value, known as the Q-value, represents the expected future rewards for taking a specific action in a given state, and following the optimal policy thereafter. The Q-value is updated iteratively using the following formula:

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

where:

- $Q(s_t, a_t)$ is the current Q-value for state s_t and action a_t.
- α is the learning rate (typically between 0 and 1).
- r_t is the reward received after taking action a_t in state s_t.
- γ is the discount factor, representing the importance of future rewards (typically between 0 and 1).
- $\max_a Q(s_{t+1}, a)$ is the maximum Q-value for the next state s_{t+1}.

The agent uses an ϵ-greedy policy for action selection, where ϵ is the probability of choosing a random action instead of the action with the highest Q-value, promoting exploration of the state space.

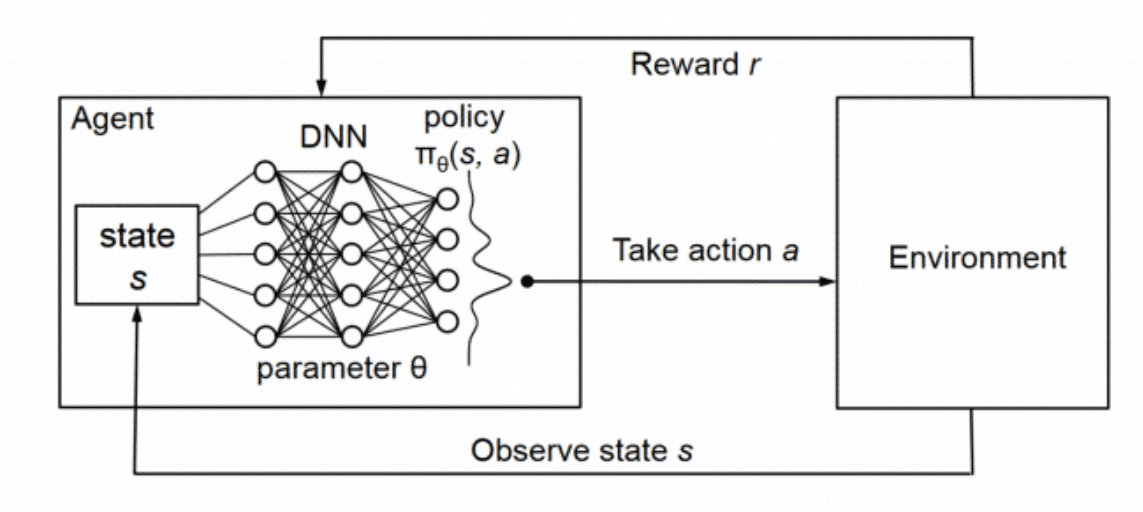


Figure 1: Deep Reinforcement Learning training process

3.2 Deep Q-Learning

Deep Q-Learning (DQN) extends tabular Q-learning by using a neural network to approximate the Q-value function, allowing it to handle large or continuous state spaces. The Q-network takes the state as input and outputs Q-values for all possible actions.

The Q-network is trained using a loss function that minimizes the difference between the predicted Q-values and the target Q-values. The target Q-value for a given state-action pair is given by:

$$y = r_t + \gamma \max_{a'} Q(s_{t+1}, a'; \theta^-)$$

where:

- y is the target Q-value.
- r_t is the reward received after taking action a_t in state s_t.
- γ is the discount factor.
- $\max_{a'} Q(s_{t+1}, a'; \theta^-)$ is the maximum Q-value for the next state s_{t+1} predicted by a target network with parameters θ^-.

The loss function for the Q-network is:

$$L(\theta) = E\left[(y - Q(s_t, a_t; \theta))^2\right]$$

where θ are the parameters of the Q-network.
To stabilize training, DQN employs two key techniques:

- **Experience Replay**: The agent stores its experiences (s_t, a_t, r_t, s_{t+1}) in a replay buffer and samples random mini-batches of experiences to break the correlation between consecutive updates.
- **Target Network**: A separate target network with parameters θ^- is used to generate target Q-values. The parameters of the target network are periodically updated with the parameters of the Q-network.

These methods help in reducing the variance of updates and in stabilizing the learning process.

4 Implementation details

4.1 Environment

A training environment for the RL agent was implemented using the *gymnasium* [4] and *pygame* [5] libraries. This environment enforces the rules of the Sudoku puzzle, validating the grid and providing rewards for each action taken. The observation space is a 9x9 grid of integers from 0 to 9, where 0 represents an empty cell. The action space is a 9x9x9 vector, which can be reshaped into a (row, column, digit) tuple.

The reward scheme implemented in the environment is as follows:

- $+1$ for each step.
- -5 for each incorrect step, e.g., if the cell is already occupied.
- -10 for each invalid step, e.g., if it creates an invalid Sudoku configuration.
- $+10$ for each solved row, column, or subgrid.
- $+100$ for solving the entire puzzle.

Any move leading to an incorrect puzzle configuration ends the episode. An alternative approach was tried where the game does not end but the move is reverted, allowing the agent to try again. However, this approach caused the agent to get stuck at a single point if the exploration rate was low, preventing progress until the move limit was reached.

An example environment is shown in the Figure 2.

4.2 Agent

The agent utilizes a Convolutional Neural Network with the following configuration. The input is a 9x9 matrix representing the Sudoku grid. This input is passed through two convolutional layers with 32 and 64 filters, respectively. The resulting matrix is then reshaped into a vector and processed by a fully connected layer, producing a 256-dimensional vector. This vector represents the extracted features from the game board. Next, two additional fully connected layers are used for action classification. The final output is a 729-dimensional vector, encoding the probability distribution of digits across the entire puzzle.

At each step, the agent processes the Sudoku grid and chooses an action. If the chosen action correspond to already occupied cell, then the next most probable action is selected. This action masking accelerates the training process, as the agent does not need to learn the additional rule of avoiding occupied cells. This approach was better for the convergence and training stability.

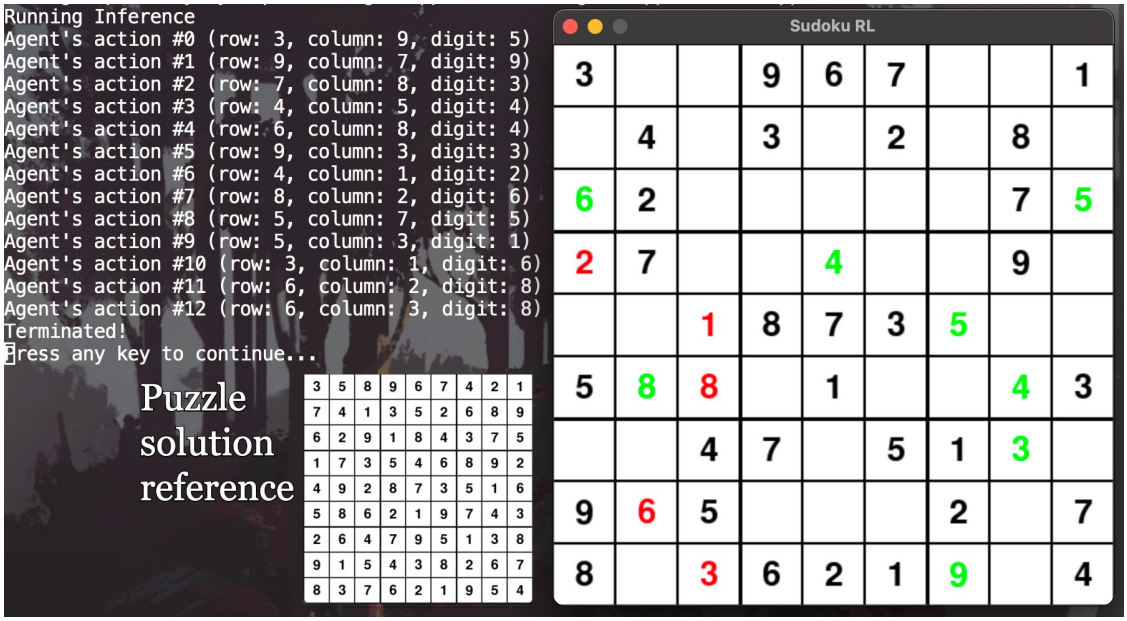


Figure 2: Training Environment: Green indicates correctly placed numbers, while red highlights incorrect ones based on the unique solution. The agent's first mistake was on move 5, and the terminal error was on move 12, violating the uniqueness rule for 4-th subgrid.

4.3 Dataset

Two datasets were used:

1) Kaggle 3m Dataset [6]

- General dataset contains: 2.9M training puzzles, 3k validation puzzles, and 3k test puzzles.
- The minimum number of clues in the dataset is 19, and the maximum is 31.
- Difficulty is calculated based on the average search depth from 10 solver attempts.
- 43% of puzzles have a zero rating, solvable by scanning.

2) Handcrafted Puzzles from sudokuwiki.org [1]

- Test dataset with handmade 59 puzzles.
- Each puzzle requires specific strategies to solve.
- Some puzzles are unsolvable even with extreme strategies.
- This dataset is valuable because the puzzles are manually curated, targeting specific patterns or interesting situations.

4.4 Training

The agent training process involves the following Deep Q-Learning algorithm:

1. **Environment Setup**: The Sudoku puzzle is loaded, and the state is represented as a 9x9 matrix.

2. **Experience Collection**: During each episode, the agent interacts with the environment by selecting actions based on the current state of the Sudoku grid. These actions, along with the resulting states and rewards, are stored in a *replay buffer*.

3. **Action Selection**: Actions are selected using an ϵ-greedy policy, balancing exploration and exploitation. The agent avoids selecting actions that correspond to already occupied cells by masking these actions.

4. **Model Optimization**: Periodically, a batch of experiences is sampled from the *replay buffer* to train the Q-network. The Q-network is updated to minimize the difference between predicted Q-values and target Q-values.

5. **Target Network**: A target network is used to stabilize training. This network is periodically updated with the weights from the Q-network, providing consistent targets for Q-value updates.

6. **Performance Evaluation**: The agent's performance is evaluated periodically using a separate validation set to ensure that the model generalizes well to unseen puzzles.

Initially, puzzles are loaded randomly, which may result in very challenging puzzles for the agent. To address this, Curriculum Learning was experimented with, starting with simple puzzles with only one digit missing and gradually increasing the difficulty.

The training sample example is shown in Table 1.

Type	Encoded board
Puzzle	`...81.....2........1.9..7...7..25.934.2.....5` `...975.....563.....4......68.`
Solution	`934817256728653419615942738176425893452398167389176542897564321563281974241739685`

Table 1: Training sample example. Note: In training '**.**' is preprocessed to 0.

5 Experiments and Results

The initial experiments demonstrated that the training code is correct, allowing the agent to memorize and solve a single puzzle. However, when scaling to the entire dataset, all metrics dropped to zero. The agent converged to a suboptimal solution, with metrics plateauing and the loss constantly increasing.

Next, we describe how we arrived at the final experimental setup.

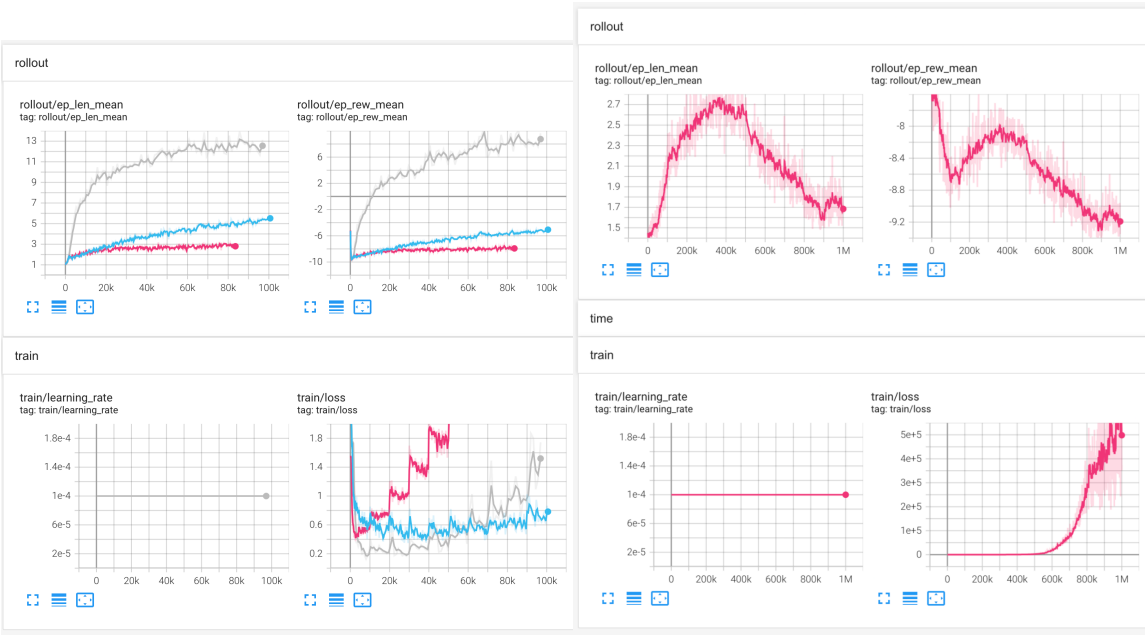
Rewards: Modifying the reward function significantly impacts the entire training process, sometimes making it impossible to compare results. We did not observe substantial changes in test metrics, so we fixed the reward function for all experiments.

Model Architecture: We increased the network size until no further improvements in metrics were observed. With a small number of parameters, the agent finds a dummy strategy and fails to learn effectively.

Curriculum Learning: We start training on - simplified puzzles, which initially fill the replay buffer with simplified puzzles and gradually increase difficulty.

Hyperparameters: Unlike standard hyperparameters, we increased the batch size to 256, the learning rate to 0.001, and the exploration rate to 0.05. The batch size is particularly beneficial as larger batch sizes smooth out the graphs and make optimization steps more accurate.

We trained the agent with this setup, achieving an accuracy of no more than 2%, which is better than a random guess but insufficient to fully solve the puzzle. Despite our efforts, the training graphs showed a similar trend — the agent learns a basic strategy and does not develop further (see Figure 3a). Even with prolonged training, the loss graphs continue to increase (see Figure 3b). We hypothesize that the agent does not generalize the rules but memorizes the simplest situations until the model capacity is exhausted.



(a) Convergence trend for different agents.

(b) Training for 1M steps on the final setup.

Figure 3: Agent convergence problem.

We also attempted to train the agent specifically on the test puzzles, hoping that the agent could learn a pattern or heuristic. The agent achieved an accuracy of 16.79%, but did not completely solve any puzzles. The results are provided in Table 2. It is evident from the results that the agent performs better on simple puzzles where sufficient hints are already given. Among advanced tactics, the agent seems to tried to learn 3D Medusa rule.

Our results somewhat correlate with "Reinforcement Learning For Constraint Satisfaction Game Agents" [7] paper.

Table 2: Evaluation results on test puzzles.
Correctness Rate - the ratio of correct moves made, where 1 indicates that all moves were correct and the puzzle was solved.
Difficulty - human based puzzle difficulty from SudokuSolver wiki [1].
Complexity Score - computer based puzzle difficulty score from SudokuSolver wiki [1]; higher numbers indicate the need for more complex heuristics.
Clues - number of pre-filled numbers in the puzzle.

Puzzle	Correctness Rate	Difficulty	Complexity Score	Clues
Hidden UR Type 1	0.6500	Easy	9	61
Simple Col. Rule 4	0.5833	Easy	8	61
3D Medusa Rule 6	0.5556	Hard	258	39
Y-Wing example	0.3500	Hard	163	44
3D Medusa Rule 4	0.3529	Hard	159	39
Hidden UR Type 2	0.3243	Hard	182	33
Aligned Pair Excl.	0.3243	Hard	277	36
3D Medusa Rule 3	0.3171	Hard	277	34
3D Medusa Rule 2	0.3000	Hard	249	39
3D Medusa Rule 1	0.2857	Hard	174	41
Swordfish	0.2667	Easy	37	51
Gentle	0.2545	Easy	18	26
Simple Col. Rule 2	0.2400	Easy	40	50
Unique Rect Type 2b	0.2391	Moderate	80	38
Simple Col. Rule 3	0.1600	Easy	37	49
X-Wing	0.1818	Tough	109	48
Riddle of Sho	0.1864	Extreme	229	40
Quad Forcing Chain	0.2000	Extreme	527	31
AIC - strong link	0.2000	Extreme	466	22
Unique Rect Type 2	0.1935	Moderate	71	28
Shining Mirror	0.1579	Easy	25	53
Almost Locked Set	0.1333	Easy	27	57
Sue-De-Coq	0.0444	Extreme	1256	22
Finned Swordfish	0.0435	Brute Force	-	21
Finned X-Wing	0.0476	Brute Force	-	21
Hidden UR Type 2b	0.0698	Hard	332	27
Unique Rect Type 4b	0.1250	Tough	808	17
SK Loop	0.1356	Tough	502	23
Unique Rect Type 4	0.1667	Tough	187	25
Escargot	0.1053	Extreme	522	35
XYZ-Wing	0.1064	Hard	207	41
Intersection Removal	0.1034	Easy	11	56
Naked Triples	0.0909	Easy	16	56
Diabolical	0.0926	Hard	277	34
Moderate	0.0566	Moderate	77	47
Hard 17 Clue	0.0781	Easy	75	17
Easy 17 Clue	0.0156	Easy	18	53
Easiest Sudoku	0.1633	Easy	11	58
Empty Rectangle	0.2292	Extreme	184	36
Grouped X-Cycle	0.2143	Extreme	428	36
X-Cycle (strong)	0.2258	Hard	301	28
X-Cycle (weak)	0.3929	Hard	258	39
XY-Chain	0.2174	Hard	387	35
AIC - weak link	0.1509	Moderate	80	35
AIC - off chain	0.1463	Hard	163	44
Dual Cell Forcing Ch.	0.1778	Easy	33	54
Triple Cell Forcing Ch.	0.0714	Tough	122	41
Triple CFC + ALS	0.1316	Hard	258	39
Triple Unit Forcing Ch.	0.1739	Easy	22	50
Death Blossom	0.1667	Easy	38	46
Exocet	0.0847	Brute Force	-	24
Easter Monster	0.1333	Extreme	374	39
Arto Inkala	0.0833	Brute Force	-	22

6 Conclusion

This project explored using RL to solve Sudoku puzzles. Sudoku's strict rules and need for precise deduction make it difficult for neural networks, which usually perform better in unpredictable or continuous environments. Traditional search algorithms can solve Sudoku efficiently, reducing the advantages of RL.

Our experiments showed that while a neural network agent can learn some rules, the lack of opponents and the game's deterministic nature limit RL's effectiveness. We tried Curriculum Learning, but this did not fully solve the problems. Additionally, RL has many hyperparameters, making it hard to configure and debug.

In conclusion, while Sudoku seems like an easy game, it is challenging for neural networks, especially in RL settings, which introduce additional complexities. It has been shown that neural networks that solve the entire puzzle in one step or use heuristic methods perform better than the iterative RL approach.

References

[1] "Sudoku solver by sudokuwiki.org," 2021. [Online]. Available: `https://www.sudokuwiki.org/sudoku.htm`.

[2] "Can convolutional neural networks crack sudoku puzzles?," 2017. [Online]. Available: `https://github.com/Kyubyong/sudoku`.

[3] "Sudoku game using reinforcement learning," 2024. [Online]. Available: `https://github.com/CristianCerasuolo-UniSa/SudokuRL`.

[4] M. Towers, J. K. Terry, A. Kwiatkowski, *et al.*, *Gymnasium*, Mar. 2023. DOI: `10.5281/zenodo.8127026`. [Online]. Available: `https://zenodo.org/record/8127025` (visited on 07/08/2023).

[5] P. Shinners, *Pygame*, `http://pygame.org/`, 2011.

[6] "3 million sudoku puzzles with ratings," 2020. [Online]. Available: `https://www.kaggle.com/datasets/radcliffe/3-million-sudoku-puzzles-with-ratings`.

[7] A. Mehta, "Reinforcement learning for constraint satisfaction game agents (15-puzzle, minesweeper, 2048, and sudoku)," *arXiv preprint arXiv: 2102.06019*, 2021.