Sudoku with Reinforcement Learning

Akshay Krishnan | Dipak Sairamesh
M.S. in Data Science | M.S in Robotics
Northeastern University, Boston, MA
krishnan.aks@northeastern.edu | sairamesh.d@northeastern.edu

Abstract

We investigate the underlying strategies of sudoku through the efficiency of deep reinforcement learning algorithms, specifically through Monte Carlo Tree Search, different CNN architectures in Deep Q-Learning (DQN) and off-policy Actor-Critic (AC) for creation and solution of puzzles. We also investigate the effects of a replay buffer type on performance and propose a distinctive replay buffer that seems to work the best. The presented models were trained on a dataset comprising of 3 million 9x9 puzzles and solutions. The presentation of data during the learning and problem-solving process is described.

The study's findings showed that models based on Q-Learning are more complex for tasks in constrained environments. The training process for these models must be supported by higher hardware requirements. The Monte-Carlo tree search approach was the most effective. It achieves superior results than other Deep Learning techniques even with less iterations. The difficulty of training the model poses a serious disadvantage, and the technology requirements are disproportionate for this sort of study. The environment complexity and behavior of a subset of constraint games utilizing reward structures are explored in this research.

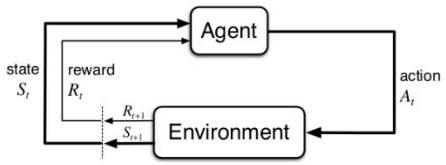
Introduction

Together with supervised and unsupervised learning, reinforcement learning (RL) is a branch of machine learning where agents interact and perform actions with their environment to maximize rewards. RL algorithms can learn from noisy and delayed rewards. With the help of RL, an agent may teach itself to execute actions with only a straightforward reward signal and gradually increase the benefits it receives. Sudoku is an excellent tool for practice since the decision-making needed to solve constraint games is comparable to the decision-making in control systems encountered in real-world scenarios. Sudoku comprises of large state and action spaces, the need for guesswork, undetermined states, and limitations over many cells. The goal is to complete a partial 9x9 grid with digits in each column, row, and each of the nine 3x3 sub-grids, which each include numbers from 1 to 9. Since there is just one solution to a Sudoku puzzle and only one set of actions that may be used, it is challenging to solve. We demonstrate that policies obtained through reinforcement learning can learn optimal control strategies. We also demonstrate the effectiveness of utilizing a memory/experience buffer in DQN and delve into the performance using actor-critic methodology.

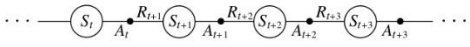
Background

We outline the techniques that will be applied to our game. An agent discovers the best course of action by exploring and monitoring the results of those actions. If the agent cannot immediately receive feedback on whether the actions were good or harmful, the environment provides sparse rewards. The network cannot be readily understood using policy gradients. So, we focused on Q-learning strategies. We employ Deep Q-Network representations of the Q-table as function approximation (DQN).

The Agent-Environment Interface



Agent and environment interact at discrete time steps: $t = 0, 1, 2, 3, \ldots$
Agent observes state at step t: $S_t \in \mathcal{S}$
produces action at step t: $A_t \in \mathcal{A}(S_t)$
gets resulting reward: $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$
and resulting next state: $S_{t+1} \in \mathcal{S}^+$



Markov decision process (MDP) is utilized in order to simulate decision-making in stochastic contexts. The collection of states and actions denoted by S and A respectively is finite. R is the immediate reward attained

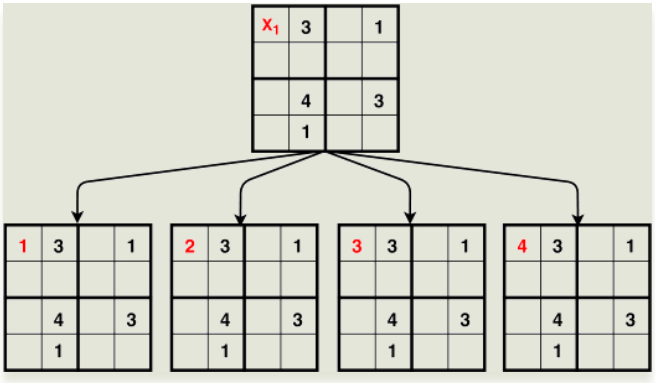
following a change from state S to state S' by performing action A. $\gamma \varepsilon [0, 1]$ is the discount factor between future and present rewards. P is the probability that action A in state S at time t will result in state S' at time t+1. When the agent observes an immediate reward, it alters its state and acts based on the transition probability.

Q-learning is an off-policy reinforcement learning algorithm that seeks to find the best action to take given the current state. It is termed off-policy since the q-learning function learns from actions that are outside the present policy, such as doing random actions. Q-learning aims to learn a policy that maximizes total reward. There is no model of transition probabilities available, thus the best policy is established by trial and error. It learns the best policy value irrespective of the agent's behavior. We do not keep any explicit rules for the discovery process, merely a value function. The policy is implicit in this case and may be deduced straight from the value function by selecting the action with the highest value (epsilon-greedy action selection).

Monte Carlo Tree Search uses Monte Carlo simulation to gather value estimations in order to direct the search tree towards highly rewarding paths. In other words, MCTS prioritizes more promising nodes to avoid having to brute force all possibilities, which is impractical. Its application extends beyond games, and MCTS can theoretically be applied to any domain that can be described in terms of (state, action) pairs and simulation used to forecast outcomes. The basic MCTS method is straightforward: a search tree is constructed node by node based on the results of simulated playouts. The procedure is divided into the following phases:

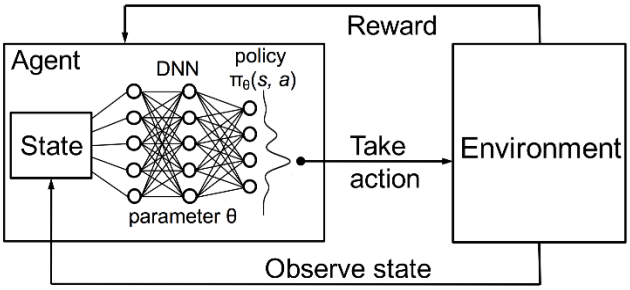
Selection: Starting at root node R, recursively select optimal child nodes until a leaf node L is reached
Expansion: If L is a not a terminal node (does not end the game) then create one or more child nodes and select one C
Simulation: Run a simulated playout from C until a result is achieved
Backpropagation: Update the current move sequence with the simulation result

An estimated value based on simulation results and the number of times it has been visited must both be present in each node. By selecting the node that maximizes a certain quantity, node selection during tree descent is accomplished. An Upper Confidence Bounds (UCB) is often implemented, which balances the utilization of known rewards with the investigation of comparatively unexplored nodes. MCTS estimates are typically unreliable at the beginning of a search but converge to more reliable estimates given enough time and to perfect estimates given infinite time. Reward estimates are based on random simulations, so nodes must be visited several times before these estimates become reliable.



Monte Carlo Tree Search rollout (4x4 grid)

DQN is used in continuous policy spaces, and the action space can be continuous or discrete. Moreover, since DQN is based on Q-learning, it is model-free and updates its policy after episodic tasks. The amount of memory and compute needed by Q will increase if the combinations of states and actions are too vast. To fix this, we change to a deep Q network (DQN) to approximate Q. (s, a). The deep Q-learning learning algorithm is used. Instead of recalling the answers, we generalize the Q-value function approximation.



Deep Q Network

Experience Buffer are only utilized as a main DQN agent learns from each experience without further processing it. However, because experiences are so closely connected to one another, the agent will suffer if it does not encounter enough variety. The training process must thus be sped up by creating random encounters and storing them in a buffer.

Related Work

Double Q-Learning entails updating each other with the use of two different Q-value estimators. We can estimate the unbiased Q-values of the actions chosen using the opposing estimator using these independent estimators. As a result, we can prevent maximizing bias by separating our updates from inaccurate estimations. In Clipped Double Q-learning, a form of Double Q-Learning, we also have two independent estimates of the true Q value. When one of our two Q networks produces a higher Q estimate than the other, we lower that estimate to the lowest to prevent overestimation and use that value to determine the update targets. Double Q-learning is frequently used in state-of-the-art Q-learning variants and Actor Critic methods.

Prioritized Experience Replay: DQN samples transitions from the replay buffer uniformly. However, we should pay less attention to samples that are already close to the target. Therefore, we can select transitions from the buffer based on the error value (pick transitions with higher error more frequently) or rank them according to the error value and select them by rank (pick the one with higher rank more often).

Noisy Nets: DQN uses ϵ-greedy to select actions. Alternatively, NoisyNet replaces it by adding parametric noise to the linear layer to help in exploration. In NoisyNet, ε-greedy is replaced by a greedy method to select actions from the Q-value function. But for the fully connected layers of the Q approximator, we can add trainable parameterized noise to explore actions. Adding noise to a deep network is often better than adding noise to an action like that in the ϵ-greedy method.

Problem Description

The study used three different types of environments (Sudoku boards), which were classified by difficulty. The practical part of the research includes constructing an environment using Python programming language and the implementation of various architectures and methods for Deep Learning using the PyTorch framework. This was performed iteratively, starting with agents' study in straightforward environments and increasing complexity if agents provided satisfying results. The goal of the project is to teach agents to answer Sudoku puzzles as accurately and quickly as possible, just like a person. Agents based on the evaluation system accurately identified how to solve the problem and learnt to think instead of mindlessly filling the cells with information thanks to training with reinforcement.

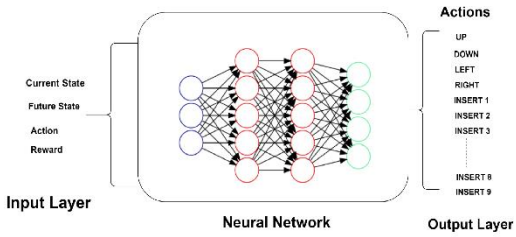
Monte Carlo Tree Search for Sudoku

Monte Carlo Tree Search has been generally used to solve puzzles and play games such as Kakuro, Tic-Tac-Toe, 2048 and so on. We can use the MCTS algorithm and solve a puzzle such as Sudoku as well. The Sudoku grid (9x9 in our case) would be the environment and all the cells of the sudoku grid would comprise the state space. The different nodes of the tree would comprise of all the possible values that could go inside a particular cell and the tree keeps expanding until all the cells of the sudoku are filled. The best action taken would depend on the MCTS. An action in this case, would be the insertion of a number between 1 and 9. The reward from backpropagation would be the ratio between the number of unfilled cells to the total number of cells in the sudoku grid.

Algorithm 4.9 Monte Carlo tree search

```
function SelectAction(s, d)
    loop
        Simulate(s, d, π_0)
    return arg max_a Q(s, a)
function Simulate(s, d, π_0)
    if d = 0
        return 0
    if s ∉ T
        for a ∈ A(s)
            (N(s, a), Q(s, a)) ← (N_0(s, a), Q_0(s, a))
        T = T ∪ {s}
        return Rollout(s, d, π_0)
    a ← arg max_a Q(s, a) + c √(log N(s) / N(s, a))
    (s', r) ~ G(s, a)
    q ← r + γ Simulate(s', d − 1, π_0)
    N(s, a) ← N(s, a) + 1
    Q(s, a) ← Q(s, a) + (q − Q(s, a)) / N(s, a)
    return q
```

DQN for Sudoku



source network. The target network's weights are updated after a certain number of frames. To train the network, we use an experience replay, which includes sampling a batch of previous encounters. The input layer of the model consists of the following:

- Current State – S

- Action taken – A
- Reward for acting – R
- Next State – S'

The middle comprises our neural network architecture for the DQN agent which is described below:

Input Shape	Layer	Activation	Output Shape
9,9,9	Convolution(Kernel size=(9,1),filters=9)	Relu	1,9,9
1,9,9	Convolution(Kernel size=(1,9),filters=9)	Relu	1,1,9
1,1,9	Convolution(Kernel size=(9,1),filters=9)	Relu	1,1,9
1,1,9	Reshape	None	9
9	Fully Connected(Batch Normalized)	Relu	81

ADAM optimizer is used which is an alternative optimization strategy to the conventional stochastic gradient descent method for iteratively updating network weights based on training data. When the learning rate is modest, SGD performs as well to conventional gradient descent. Using estimations of the first and second moments of the gradients, Adam's optimization technique calculates specific adaptive learning rates for various parameters.

RELU is another non-linear activation function that has become more prominent in the deep learning space. Rectified Linear Unit is referred to as ReLU. The ReLU function's primary benefit over other activation functions is that it does not simultaneously fire all the neurons. ReLU's use aids in preventing the exponential development of the compute needed to run the neural network helping compensate against exploding and vanishing gradients.

Batch Normalization is an additional network layer put between one hidden layer and the following hidden layer. Prior to sending them on as the input of the next hidden layer, it has the task of normalizing the outputs from the previous hidden layer. It is done along mini batches instead of the full data set to speed up training and use higher learning rates, thus making learning easier. The distribution of the data is also maintained via batch normalization. One of the key features we include in our model is batch normalization. It acts as a regularizer, normalizing the inputs throughout the backpropagation process, and can be used to most models to improve convergence.

The output layer comprises of the actions required to solve the Sudoku puzzle. The main components include:

- Actions – [UP, DOWN, LEFT, RIGHT]
- Inserting Numbers – [1, 2, 3, 4, 5, 6, 7, 8, 9]

Experience/Memory Buffer uses 500 data points that were saved initially to decide what to do in each of the states. Following those data points, the DQN Model, which includes the Target and Evaluated values, is used to make accurate predictions for further environments. The data points trained were of the form: [State, Action, Reward, Next State]

Actor Critic is used as an improvement from the **DQN with Experience Buffer** solution. The only difference here is that we have a q table for the actor one for the critic as well. Now based on the actor Q value we select the best possible actions and the critic Q table helps to choose the best possible Actor Q values and rewards.

Experiments

Based on different difficulty levels of the Sudoku, we tried solving the puzzles using the MCTS approach. We see that as the difficulty level of the Sudoku increases, the time and the number of iterations taken by the MCTS algorithm also increases as shown in Fig 1.

Difficulty Level	Average Time Taken(seconds)	Average No of Iterations
Easy	25.5	160
Medium	150.3	580
Hard	470.6	1300

Fig 1: Number of Iterations and time taken to solve the Sudoku using MCTS across different difficulty levels

We also tried changing the depth of the tree and the maximum number of iterations to see how well the algorithm would perform over time. As we see in Fig 2, a Max depth of 12 seems to be doing slightly better than the one which has a Max Depth of 4. Based on the experiments performed, we see that the algorithm gives best results for a Max Depth value between 8 and 12.

Varying the maximum iterations of the tree also plays a huge role in solving the puzzles. Finding the right number of iteration value is key, as a higher number would increase the computation and a lower number would result in the episode to not complete. As shown in Fig 3, we see that as the maximum iterations increases, the number of completed episodes also increase. Based on the experiments conducted,

we found out that a value between 800 to 1000 is the ideal value for the max iteration of the MCTS algorithm.

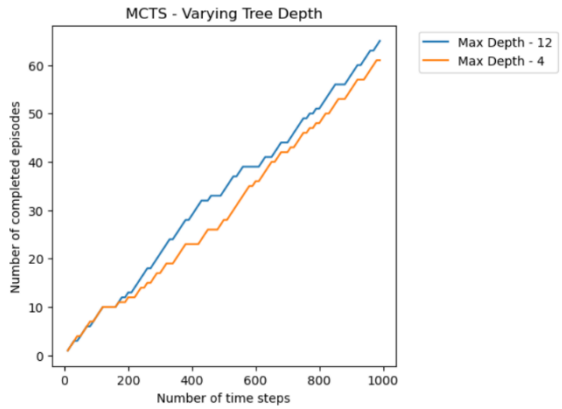


Fig 2: Performance of MCTS after varying the Max Depth of the Tree

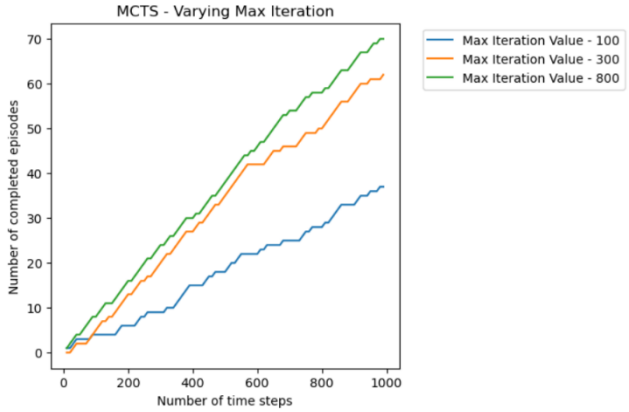


Fig 3: Performance of MCTS after varying the Max Iterations in the Tree

We also compared the MCTS algorithm with the DQN, DQN with experience buffer and the Actor Critic algorithm. The experiment was conducted on Sudoku puzzles that are of moderate difficulty. From Fig 4, we see that the MCTS algorithm performs the best out of all the other algorithms. The Actor Critic policy performs slightly better than the DQN with experience buffer and the DQN algorithm performs the worst.

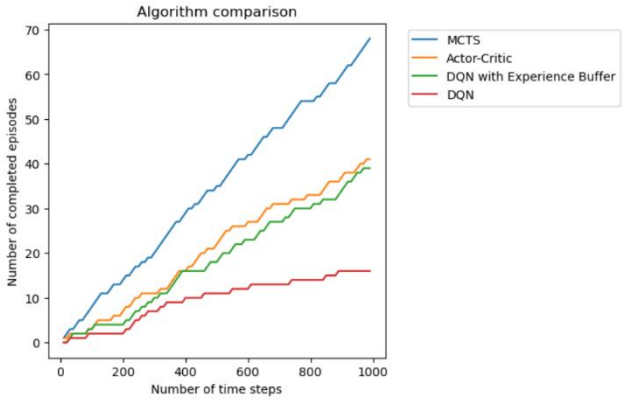


Fig 4: Model Performance

Results & Conclusions

In this paper, we see that Monte Carlo Tree Search is the best approach for solving puzzles such as Sudoku as compared to other techniques such as DQN, DQN with Experience buffer and Policy Gradient Techniques (Actor Critic). We also see how the depth of the tree and the maximum number of iterations within the Monte Carlo Tree Search affect the performance of the model. The MCTS can solve even the difficult sudoku puzzles whereas DQN and the Actor Critic methods found it difficult to solve even the easy puzzles. Although the DQN would perform better if a much more complex neural network is used, the MCTS would be the ideal Reinforcement Learning solution to solve a Sudoku puzzle.

References

i. Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G. et. al. (2016). Mastering the game of Go with deep neural networks and tree search. Nature, 529 (7587), 484–489. doi: http://doi.org/10.1038/nature16961

ii. Nguyen, N. D., Nguyen, T., Nahavandi, S. (2019). Multi-agent behavioral control system using deep reinforcement learning. Neurocomputing,359,.doi:http://doi.org/10.1016/j.neucom.2019.05.062

iii. Reinforcement Learning: An Introduction, Richard S. Sutton and Andrew G. Barto, Second Edition, MIT Press, Cambridge, MA, 2018, 15 Nov. 2020

iv. Gubin, S. (n.d.). A Sudoku Solver. iaeng. Retrieved October 26, 2022, from htttp://www.iaeng.org/publication/WCECS2009/WCECS2009_pp223-228.pdf

v. Mehta. https://arxiv.org/ftp/arxiv/papers/2102/2102.06019.pdf. Arxiv, arxiv.org/ftp/arxiv/papers/2102/2102.06019.pdf.

vi. Using constraint programming to solve Sudoku puzzles. IEEE Xplore. (n.d.). from https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4682365

Github Repository - https://github.com/akrishnan96/Sudoku-Solver