DeepSudoku

Felix Hammer, Jonas Bieber, Benjamin Fricke

September 2023

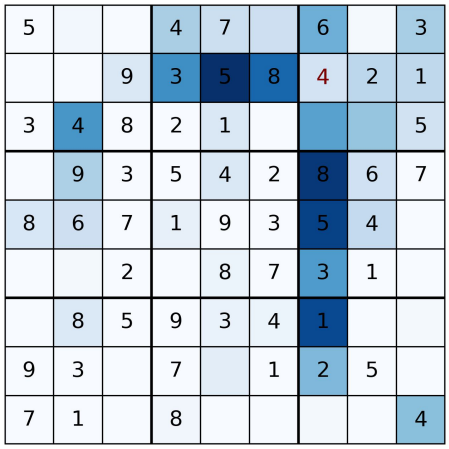

Figure 1: Visualizing the Choice of Our Best Agent

The red 4 is the number the model filled out. Marked in blue are the positive integrated gradient attentions. [18] The darker the blue, the higher the attention.

1 Introduction and Motivation

Deep Learning (DL) has seen remarkable advancements in recent years. However, its potential hasn't been fully explored in addressing certain emotional challenges people face in their daily lives. One such challenge is the feeling of intellectual defeat many experience when attempting a Sudoku puzzle, only to realize they've made a mistake earlier on. Even though Sudoku is largely an enjoyable pastime, it can sometimes lead to intense frustrations. With the strides made in Artificial Intelligence (AI), we now have an opportunity to create tools to mitigate this issue. Many Sudoku platforms already offer a "Hint" feature, designed to alleviate player frustration early on. These features predominantly

rely on established AI methodologies such as propositional satisfiability (SAT) or constraint-satisfaction solving (CSP), as cited by [9]. Such techniques provide accurate and efficient solutions, clearly demonstrating they are well-suited for the task.

Yet, a question lingers: Can the newer AI techniques grounded in Deep Learning also tackle Sudoku puzzles? The aim of this paper is to assess whether DL methodologies are viable for Sudoku. Specifically, we examined the potential of using Deep Reinforcement Learning to solve Sudoku puzzles. Our efforts culminated in a success rate of 78.40% for Sudoku puzzles with 27 empty cells, achieved using an ensemble technique with a deep reinforcement learning agent. In this paper, we outline our approach and share the insights we gathered during the process.

2 The game of Sudoku

There are many different variants of Sudoku, but we will confine the scope of this paper to discussing what is commonly referred to as *classical Sudoku*. We define a generalized classical n-Sudoku (or simply: n-Sudoku) to be a $n^2 \times n^2$ grid that is split in n^2 $n \times n$ subgrids, and whose cells are either left blank or filled with numbers between 1 and n^2. Rows of subgrids are called *bands*, and columns of subgrids are called *stacks*. The goal of such a Sudoku is to *solve* (or *complete*) it, which means to find a way to fill all the blank cells such that each row, each column and each $n \times n$ subgrid contains each of the numbers exactly once. A classical Sudoku (or just: Sudoku) is a generalized classical Sudoku with $n = 3$. A Sudoku that has only one possible completion is called *valid* or *proper* [5]. In this paper, we are only interested in valid Sudokus.
From a mathematical standpoint, the set of all possible complete Sudokus can be partioned into disjoint equivalence classes under arbitrary compositions of the following transformations:

- Reordering of the bands (stacks)
- Reordering of rows (columns) within a given band (stack)
- Transposing the grid
- Permuting the numbers

It is easy to show that the ability to transform one Sudoku into another Sudoku via some sequence of these transformations does in fact constitute a valid equivalence relation. The transformations themselves form a group (the so called sudoku-group) and the equivalence classes are called orbits of the set of complete Sudokus under said transformations. Using Burnside's lemma for counting objects up to an equivalence relation and some additional steps, Russell and Jarvis were able to calculate that there are exactly 5472730538 of these equivalence

classes [15] [5].

To fulfill our research objectives, we needed an efficient method for generating a variety of unsolved Sudoku puzzles. Our choice was a backtracking algorithm, which selects permissible numbers for each cell in a top-to-bottom, left-to-right sequence, all the while maintaining adherence to Sudoku's constraints.

Regarding the removal of numbers - essentially creating blank spaces in a puzzle - we adopted Brotsky's two-pass technique as described in [2]. During the first pass one only removes cells that are carefully constrained in by their neighborhood cells, namely those in the same row, column, and block. Puzzles crafted in this manner can be deciphered without resorting to guessing. For instance, if x signifies the number of blank or empty cells in the starting configuration, they can be analytically solved with precisely x inferences. During the second pass, the removal centers on cells that aren't immediately constrained by their surroundings. This necessitates the employment of more intricate strategies, such as "If I place a '2' here, I can't position a '3' there." To ensure that the puzzle retains a unique solution, Sudoku solvers are leveraged to compute the possible solutions post digit removal. If a removal results in a puzzle with multiple solutions, that action is reversed and a different cell is assessed for potential removal.

Classifying the difficulty level of a Sudoku puzzle can be approached in various ways. Some may gauge it by the number of empty cells, the guesses required to solve it, or even the puzzle's "density" as detailed in [2]. Our paper's primary aim is to tackle "Easy" Sudoku puzzles using Deep Reinforcement Learning. For our purposes, an "Easy" puzzle, as categorized by [2], has up to 27 vacant cells, constituting 44% of the total 81 cells, all of which are using the first pass approach.

3 Related Work

Taking a survey of the relevant literature, we found that there were already some attempts at solving Sudokus using DL-methods. In the following, we will give a short overview over the previous research that we were able to access and which we deemed relevant.

Supervised Learning Papers

- Park used a supervised approach and achieved a .86 accuracy overall when solving Sudokus of different difficulties between "Easy" and "Evil" that were gathered from `http://1sudoku.com` in [11]. He used a CNN with 10 convolutional layers and kernel size 3. Unfortunately, weights were never published. It is important to mention that Parks use of *accuracy* quantifies the percentage of correctly predicted tiles and not the percentage of complectly correctly predicted games, i.e. the *winrate*.

- Akin-David and Mantey give a more thorough overview over different possible architectures in [1]. They tested both LSTM and CNN architectures and found a 15 layer CNN with kernel size 3 to work best, with a test accuracy of again .86 on boards with ∼48 blanks.

- Yue and Lee investigate a Sudoku solver that is based on an energy driven quantum neuron architecture called Q'tron in [21]. To our understanding, Q'tron models are similar to Hopfield Networks in the sense that they can be seen as minimizing a certain Lyapunov energy function. They explain how the Sudoku game can be rephrased as such an energy-function. They state that their Q'tron solver is able to generate and solve Sudokus, but unfortunately, the provided links are broken, and it was therefore not feasible for us to verify their results.
  A similar, but seemingly less sophisticated approach, using simple Hopfield Networks, can be found in [7]. In this case, the performance of the network was not satisfying and had to be augmented by linear programming techniques.

- Palm et al. effectively solved Sudoku using Recurrent Relational Neural Networks as published in [10], achieving a 96.6% win rate on Sudoku puzzles with just 17 clues. This is the mathematically proven minimum number of clues required to produce unique solutions [5]. Their architecture follows the graph neural network paradigm by setting up tiles as nodes and connecting all tiles in a neighborhood with edges. By allowing the nodes to "communicate" over multiple steps, their network predicts complete Sudoku puzzles.

Deep Reinforcement Learning Papers

- Pujari examined the solvability of Sudoku puzzles using Deep Q-learning in [13]. Of special interest is the way in which he formulated the Sudoku game as an MDP. Unfortunately, the author did not provide any results, and we believed that his choice for the NN architecture (2 hidden layers with 20 neurons each) was not promising enough to justify the effort of training the model to produce the results ourselves.

- Mehta tried to solve Sudokus using deep Q-learning in [6], but achieved only very modest results (Sudokus did not seem to be the main focus of this paper). He achieved a winrate of 7% for Sudokus with 3 blanks, and a winrate of 1.2% for Sudokus with more than 6 blanks, by using only one hidden fully connected layer of size 810.

- The research project whose goals are most congruent with the goals we set (and which we were also able to find and access) is the work by Poloziuk and Yaremenko [12]. They apply DQN, DDQN, PPO and TD to Sudoku and compare the results. They report that they were not able to train an agent that consistently scored rewards greater than 0, ie. they were

not able to solve Sudoku using any of these methods. They ultimately resorted to solving it with Monte Carlo Tree Search, which worked fine and consistently. Unfortunately, they did not publish their architectures or weights, and it is therefore difficult to make any productive comparisons to our work.

To summarize, while Palm et. al [10] successfully solved the most difficult possible Sudoku's using supervised learning, until now there hasn't yet been any tangible success in solving even the simplest Sudoku puzzles using Deep Reinforcement Learning. Therefore we decided to investigate this further.

4 Supervised Experiments

With our primary objective for this study being to solve Sudokus using Deep Reinforcement Learning (DRL), we were faced with a myriad of decisions. These included selecting the appropriate DRL algorithm, understanding the nuances of the environment dynamics, and pinpointing the optimal blend of Sudoku representations and network architectures.

To streamline our approach, we commenced by tackling the aforementioned choices within a supervised learning framework. We leaned towards single-shot supervised learning, believing it to be both stable and well-defined. This method, having shown promising and swift results in prior research such as [11], offered us a stable foundation. Our goal was that by experimenting with the supervised setting first, we could transition to DRL more effectively and achieve faster successes.

In our initial explorations, we evaluated the efficacy of multiple compact neural network designs, ensuring they were suitably sized for standard DRL applications. Through this exercise, we conceptualized a unique last-layer activation function tailored specifically for Sudoku. This venture not only yielded this new tool but also deepened our insights into the training of supervised models for Sudoku tasks.

4.1 The SudokuDoubleSoftmaxLayer

In the context of our Supervised Learning experiments, we compiled a dataset encompassing 10,800,000 quiz and solution pairs. This was achieved by generating 400,000 pairs for each range of empty cells spanning from 1 to 27. The dataset was then divided into training (80%), validation (10%), and testing (10%) subsets.

Our objective for the Supervised learning tasks was to train networks to deduce a complete Sudoku solution in one step from a given Sudoku puzzle. For each of

the 81 cells (arranged in a 9 x 9 grid), our goal was for the network to predict 9 logits corresponding to the digits 1 through 9. These logits are then transformed into a probability distribution through the application of the softmax activation function for each cell. The loss function is then the CategoricalCrossentropy between the predicted probability distribution and the actual Sudoku solution, a methodology mirrored from [11].

While marrying this approach with Convolutional Neural Networks can yield satisfactory results [11], and even superior performance when integrating Recurrent Relation Networks [10], there exists an evident limitation: The process merely normalizes the probabilities of the digits within each individual cell, overlooking the distribution/normalization of identical digits within a neighborhood. To address this limitation, we conceived a unique output activation layer termed the *SudokuDoubleSoftmaxLayer*.

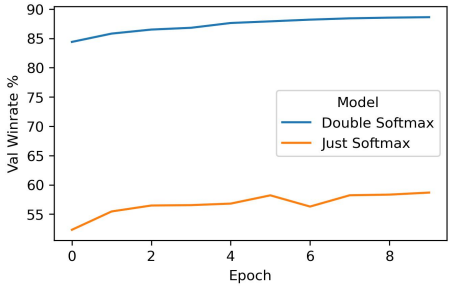

Figure 2: Impact of SudokuDoubleSoftmaxLayer
Using the OnlyConv Model

As the name intimates, this layer functions by consecutively applying two Softmax operations. The primary "Softmax" is more a generalized form of log softmax wherein for given predicted output logits p_l, we derive normalized logits p_n as:

$$p_n = p_l - logsumexp(N(p_l))$$

Here, the neighborhood function N yields the set of logits corresponding to similar digits in the neighborhood. Subsequently, we execute a softmax over the 9 digits of each cell, analogous to the method by [11].We implemented this method using various broadcast operations and some clever tricks with reshaping and transposing. For a closer look at how we did it, check out our code here.

The notable performance enhancement achieved through the introduction of the *SudokuDoubleSoftmaxLayer* is evident in Figure 2. Consequently, we incorporated this layer across all our supervised models.

4.2 Architecture Search

In our quest to find suitable architectures for subsequent use in the Deep Reinforcement Learning (DRL) scenario, we evaluated the following three architectures:

1. *MLP* - A compact MLP comprised of 3 fully connected layers.
2. *ConvwithFCHead* - It integrates two convolutional layers with kernel sizes of 3 and 9, which is then followed by flattening and two subsequent fully connected layers.
3. *OnlyConv* - This model has three convolutional layers with kernel sizes of 3, 9, and 9.

Additionally, we introduced *BigOnlyConv* as a benchmark to demonstrate the potential performance achievable through a larger model. It is built with 7 convolutional layers and incorporates skip connections. Sudoku cells' digits were inputted as one-hot encoded values, using 0 to signify an empty cell. For the *MLP*, we employed flattened vectors of size $81 \times 10 = 810$, whereas, for the other models, we used $9 \times 9 \times 10$ tensors. Each model was trained over 10 epochs using our dataset, followed by evaluation on the test dataset.

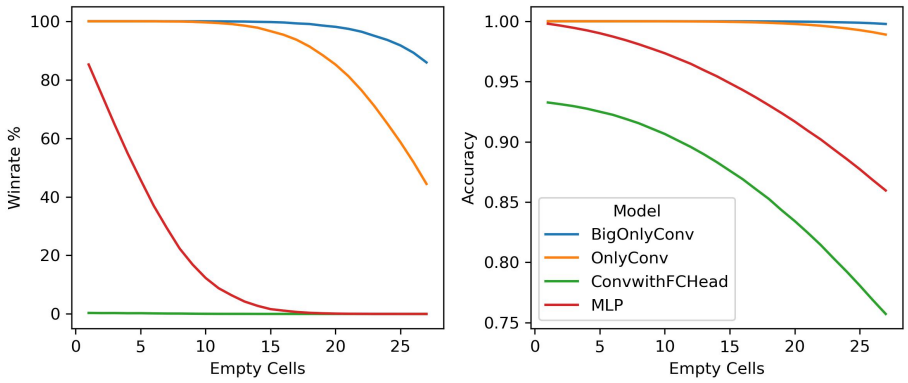

Figure 3: Test performance of different architectures.

The results in Figure 3 depict the performance dynamics of our models averaged over the number of empty cells in the test dataset. Observing these outcomes, we arrived at several pivotal conclusions:

Firstly, the *OnlyConv* model substantially outperforms the other two. Its positionally invariant design likely contributes to this since Sudoku rules inherently demonstrate some degree of positional invariance. However, when focusing on the left corner compared to the top right, positional invariance isn't perfect, especially when analyzed as an image through a convolutional kernel. Moreover,

relying solely on convolution is more weight-efficient, given the large matrix required for the last layer ($inputsize \cdot 9^3$).

Secondly, as the number of empty cells increases, there is a smooth and consistent decline in the performance across all models and metrics. This trend lends credence to our hypothesis that just as for humans, solving puzzles with more empty cells is an intrinsically more challenging task for neural networks.

Thirdly, *BigOnlyConv* not only showcases superior overall performance compared to *OnlyConv*, but also demonstrates a less pronounced decline in performance as the number of empty cells increases. This observation suggests that scaling up the model's complexity is a viable strategy. Such an approach, when further optimized, could potentially lead to even more impressive performance in solving Sudoku puzzles without the use of recurrent graph neural networks.

4.3 Overfitting on Specific Sudoku Orbits

During our early supervised learning experiments, we stumbled upon an intriguing observation. Initially, we relied on the code provided by [2] for generating Sudoku puzzles. Instead of utilizing the backtracking algorithm to produce fresh, random Sudoku puzzles, this generator began with sampling one of three predefined Sudoku grids, then applied a series of legal random transformations to them. When generating 1,350,000 quiz-solution pairs, this led to 1,350,000 unique quizzes but only 671,756 unique solutions, with the most frequent solution appearing 10 times.

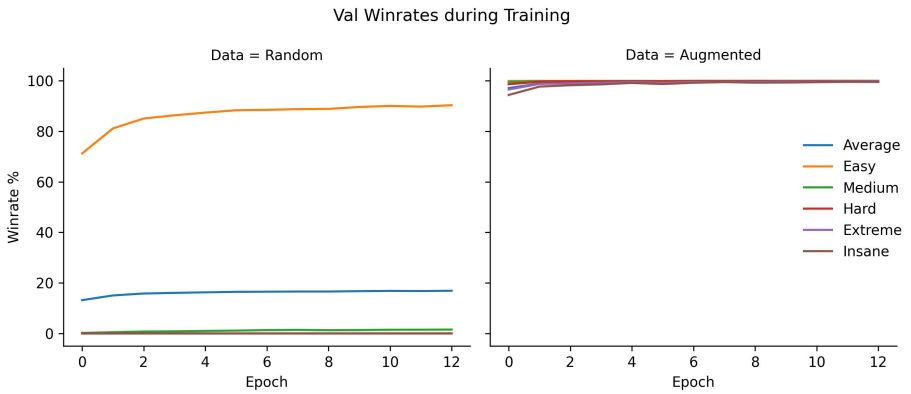

Figure 4: Overfitting on Specific Orbits

In segregating our dataset into training, validation, and testing subsets, we ensured there were no overlapping solutions between these groups. To our astonishment, our models swiftly attained stellar performance rates—even when faced with puzzles with a lot of empty cells. Such exceptional scores made

us skeptical, so we subjected our models to randomly generated Sudokus. As anticipated, the models faltered significantly. Our intuition was that the high performance on the main dataset was attributable to all our samples stemming from a maximum of three unique Sudoku orbits.

To investigate this hypothesis, we curated a comparative dataset using randomly generated Sudoku solutions for every puzzle creation instance. Apart from this, all other configurations, including the deletion of cells and model training, mirrored those of the original dataset. As illustrated in Figure 4, the performance disparity is stark. For data crafted by tweaking the three distinct Sudoku grids, the model resolves almost every puzzle difficulty, boasting win rates above 95%. However, for data sourced from random Sudoku starting points, the model could only reliably solve the "Easy" challenges. This led us to conclude that neural networks can indeed overfit on to specific Sudoku orbits.

5 Solving Sudoku Using Deep Reinforcement Learning

In our research endeavor, we aimed to resolve "easy" Sudoku puzzles through Deep Reinforcement Learning, striving to emulate the strategies a human might deploy. To do so, we employed a Markov Decision Process (MDP) formulation tailored for Sudoku-solving tasks. It is imperative to recognize that nuances in MDP formulations can profoundly affect the outcome of reinforcement algorithms. Consequently, our work encompasses a comparative analysis of different formulation strategies.

The common underlying principle our approaches lies in interpreting a Sudoku game as an k-step sequence. Herein, each step, synonymous with filling in a solitary blank cell, triggers a defined reward or penalty, represented by R_t. Within this context, populating a cell translates to taking an "action", and the game's configuration at any given step t defines the "state". Mathematically, the cardinality of the action space ($\mathcal{A}$) equals $n^2 \cdot n^2 \cdot n^2$, as there exist $n^2 \times n^2$ cells, each of which can accommodate any of the n^2 numbers. Conversely, the state space ($\mathcal{S}$) possesses a cardinality of $n^2 \cdot n^2 \cdot (n^2 + 1)$, given the $n^2 \times n^2$ cells can either hold one of the n^2 numbers or remain unoccupied. While other Sudoku MDP formulations, particularly concerning the action space, do exist, as demonstrated by [13] and [12], we believe our approach closely mirrors human-solving methods.

After formulating the MDP, we faced key decisions: choosing the appropriate learning algorithm and refining both the environment dynamics and agent architectures for optimal performance.

In the sections that follow, we detail our decision-making process and the resulting performance metrics. We begin by discussing our adoption of Masked PPO [3, 17]. Next, we delve into the effects of varying environment dynamics and agent architectures. Given that we adjusted both concurrently, direct comparisons presented challenges. To address this, we evaluated multiple environment dynamics using our best model, *SeperateOnlyConv*, and analyzed the performance of various agent architectures in our top-performing environments.

5.1 Masked Proximal Policy Optimization

After formulating the MDP and establishing our research objective, the next challenge was selecting the appropriate deep RL algorithm. Our initial experiments involved Q-learning [20, 8], but achieving satisfactory results proved elusive. Furthermore, generating an entirely new Sudoku grid to produce a fresh state is computationally intensive. Given these considerations, we transitioned to the generally more sample-efficient PPO [17].

Proximal Policy Optimization Clip (PPO-Clip, or simply PPO for our study) builds on the policy gradient theorem [19]. Its main goal is to mitigate the issues of excessively large policy updates during consecutive gradient computations without interim resampling. Essentially, PPO ensures gradient updates stay within a designated "Trust-Region" [16], limiting the magnitude of policy changes. This trust region is concretely implemented via a surrogate objective using a clip function:

$$L^{\text{CLIP}}(\theta) = \hat{\mathbb{E}}_t\Big[\ \min\Big(\frac{\pi_\theta(s_t|a_t)}{\pi_{\theta_{\text{old}}}(s_t|a_t)}\hat{A}_t, \text{clip}\Big(\frac{\pi_\theta(s_t|a_t)}{\pi_{\theta_{\text{old}}}(s_t|a_t)}, 1-\epsilon, 1+\epsilon\Big)\hat{A}_t\Big)\ \Big]$$

Intuitively, the ratio $\frac{\pi_\theta(s_t|a_t)}{\pi_{\theta_{\text{old}}}(s_t|a_t)}$ describes the "distance" between the current policy (which potentially already underwent multiple gradient steps) and the old policy (ie. the policy that was actually for sampling and calculation of $\hat{A}_t$), and we are clipping said distance in a "Trust Region" between $1-\epsilon$ and $1+\epsilon$. This creates the simple effect that during any series of gradient steps, the updated policy will never diverge too much from the original policy and thus effectively prevents the destructively large updates. The policy gradient $\hat{g}$ is of course then easily attained by differentiating $L^{\text{CLIP}}(\theta)$ w.r.t. θ, as discussed above. [17]

Proximal Policy Optimization (PPO) is often augmented with various refinements in practical implementations, differing from its basic form. For an extensive overview of these modifications, the blogpost "The 37 Implementation Details of Proximal Policy Optimization" by Huang et al. [4] is highly recommended. We based our work on the PPO-script provided by the same author, available at here. In line with good scientific practice, our code builds upon theirs.

A significant modification we adopted is *invalid action masking*. This involves setting the logits for invalid actions to a very large negative value (commonly -1e8) before creating a categorical distribution. Such a method significantly accelerates learning in vast action spaces with numerous invalid actions. For an in-depth discussion, see [3]. In our context, an invalid action attempts to populate a non-empty cell. However, relying exclusively on action masks may render an agent ineffective without them. Although our preliminary experiments with gradual mask removal during training showed promise, we ultimately retained the mask throughout, valuing its simplicity. Moreover, we believe that avoiding populated cells aligns closely with human behavior during Sudoku-solving.

Considering the probability of randomly selecting a correct action (P_r), the use of invalid action masking significantly influences this likelihood. Without the masking, the probability is determined as:

$$P_r = \frac{N_{empty}}{81} \times \frac{1}{9} + \frac{81 - N_{empty}}{81} \times 0 = \frac{N_{empty}}{729}$$

where N_{empty} denotes the number of empty cells. On the other hand, with the application of invalid action masking, this likelihood simplifies to:

$$P_r = \frac{1}{9}$$

By utilizing this approach, we were optimistic about the learning efficiency, even within our MDP's expansive action space of 729 options, given the relatively elevated probability of selecting a correct action.

5.2 Environment Dynamics

Although the broader structure of the Sudoku MDP was outlined, nuances in the environment dynamics were yet to be determined. There are multiple methodologies to refine a given MDP to better suit RL algorithms, notably via reward shaping and altered termination conditions. We studied the effects of diverse rewards and game termination conditions, resulting in four distinct environment settings as summarized in the table below.

Environments				
Name	*Sparse*	*Win Reward*	*Dense*	*No Stopping*
R_{t+1} for correct action	0	1	1	3
R_{t+1} for mistake	-1	0.5	0.5	-0.1
R_{t+1} for completion	1	100	0	0
Instant termination on mistake	Yes	Yes	Yes	No
N allowed mistakes	1	1	1	10

Here, N allowed mistakes denotes permissible mistakes per board state. Each environment was implemented using the *gymnasium* library and is available for

public use. For the *No Stopping* environment, both the "reward for mistake on empty cell" and the "allowed mistakes per board state" parameters were varied over our experiments.

To gauge reinforcement learning's efficiency in Sudoku-solving environments, we measured the win rate over 100 games, using greedy sampling to consistently choose the most probable action. We opted for the *Dense* environment as our benchmark for two reasons: its immediate termination upon an error, which mirrors real-world Sudoku scenarios, and its easily interpretable rewards. Consequently, all environment evaluations were standardized against the performance in the *Dense* setting.

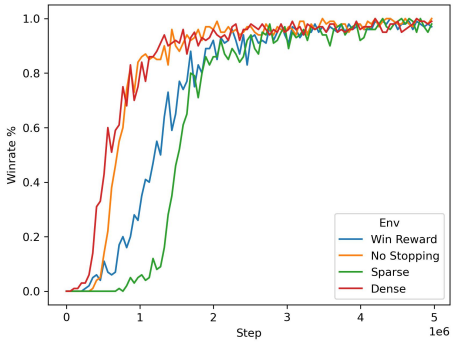

Figure 5: Performance Across Environments

Performance trend of the SeperateOnlyConv Model on Sudokus with 5 vacant cells.

Figure 5 showcases the performance comparison across different environments, given otherwise identical conditions. These results are for solving Sudokus that start with 5 unfilled cells. Notably, while all environments tend to achieve near-perfect win rates, the *No Stopping* and *Dense* environments promote accelerated learning compared to *Sparse* and *Win Reward*. In the *Win Reward* setting, we noticed considerable losses for the critic model, decelerating the training process. It's noteworthy that weaker architectural designs in our preliminary experiments displayed a more pronounced impact due to the choice of environment.

Based on our findings, the final environment choices for training our model were narrowed down to *Dense* and *No Stopping*. To ascertain which one would yield superior performance, we utilized our best architecture to train for 10 million steps, specifically targeting Sudoku puzzles that started with 27 empty cells. The outcomes from this extensive experiment can be examined in Figure 6. Several salient observations emerged:

Firstly, the *No Stopping* environment demanded less wall-clock time. This is likely attributed to the diminished need for Sudoku puzzle generation, a notably computation-intensive task.
Secondly, although the *Dense* environment achieved a comparable or slightly higher *Generalized Return*, its learning curve appeared more stable. Intriguingly, as training progressed, the *No Stopping* environment exhibited an increase in its *winrate* - a pattern conspicuously absent in the *Dense* environment.
Thirdly, by observing the typical "time of terminations" or steps leading to fail-

ures, we discerned that the *Dense* environment exhibited a pronounced peak at a specific juncture. This suggests that agents trained in this environment might either be solving simpler paths and then encountering dead-ends, or they might not be frequently reaching the more challenging latter stages of the game.

In light of these observations, and particularly due to the promising *winrate* uptrend in the *No Stopping* environment, our decision tilted in favor of the *No Winning* environment for the final model iteration.

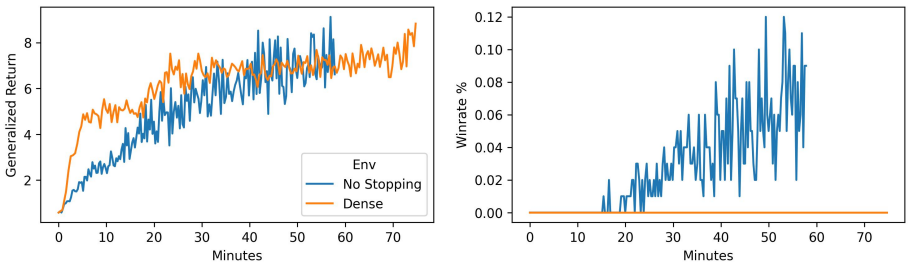

Figure 6: *No winning* vs *Dense* for 10M steps
Trained using the SeperateOnlyConv Model.

5.3 Architecture Search

To identify a robust agent architecture, we revisited our architecture search from the supervised setting. We concentrated on three main models: the *MLP*, *OnlyConv*, and *ConvwithFCHead*. Given that PPO employs an agent-critic method, modifications to these models were necessary.

For the *MLP* model, we straightforwardly incorporated an independent MLP for the critic network. With the *OnlyConv* and *ConvwithFCHead* models, we adhered to the methodology detailed in [4]. In this setup, the initial convolution layers are shared between both the actor and the critic. Atop these shared convolution layers, we introduced two fully connected layers dedicated to the critic network. To gauge the impact of this layer sharing on performance, we fashioned a variant of the *OnlyConv* model, which employed a wholly distinct critic MLP. This variation yielded two configurations: the *SharedOnlyConv* and the *SeperateOnlyConv*. Additionally, since we believed that the *OnlyConv* model's complete positional invariance was a big reason for its good results in the supervised setting, we wanted to try another model with natural positional invariance. So, we added a small *Transformer* agent. For this *Transformer*, we added one-hot encoded position indicators to the Sudoku inputs. These indicators show which row, column, and block a cell is in. This means the *Transformer* was working with inputs shaped as $81 \times (10 + 9 + 9 + 9) = 81 \times 37$.

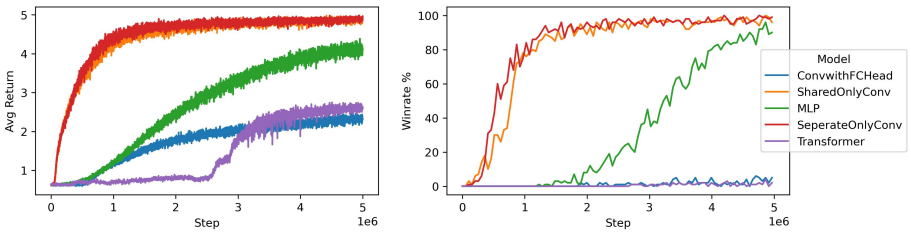

Figure 7: Influence of Agent Architecture on Learning

To evaluate the different architectures, we chose a scenario where the task was to solve sudokus with 5 empty cells. Each architecture was trained for 3M steps using the *Dense* environment. We routinely assessed each model's win rate over 100 games by sampling greedily. As depicted in Figure 7, the *OnlyConv* models clearly outperformed the others. The *MLP* came next, while both the *ConvwithFCHead* and *Transformer* didn't manage to win any games. Interestingly, these findings mirror the results we got in the supervised setting (see Figure 3). Moreover, the performances of *SharedOnlyConv* and *SeperateOnlyConv* were very similar, suggesting that sharing weights between the actor and critic models doesn't have a major impact.

However, when training in the *No Stopping* environment, we noticed a significant performance gap between the *SharedOnlyConv* and *SeperateOnlyConv* models. As illustrated in Figure 8, the *SeperateOnlyConv* model reached the optimal reward of 15 much quicker than the *SharedOnlyConv*. A closer look also revealed that both models had their highest value losses when their rewards ranged between 0 and 5. Moreover, the *SharedOnlyConv* model experienced high value losses for a longer period.

Our theory for these observations is that the *No Stopping* environment is tougher for the critic function than the *Dense* environment, especially since the agent can make multiple errors per board state and accumulate negative rewards. At

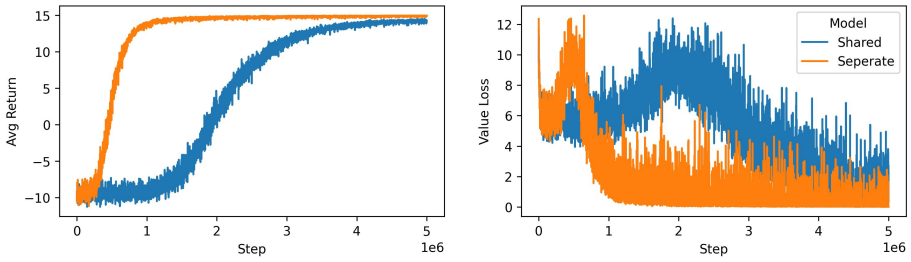

Figure 8: Effect of sharing Network Weights in the *No Stopping* Environment
For Sudoku puzzles with 5 empty cells and trained for 3M steps.

first, the agent frequently makes wrong moves, causing the critic network to anticipate negative rewards. As the agent gradually improves but still makes some mistakes, there's a mix of high and low rewards, complicating the value prediction task. This challenge seems to cause the critic and network losses to conflict, leading to slower training progress.

5.4 Training our Best Agent

Our primary aim was to solve "easy" Sudokus - those with 27 empty cells - using reinforcement learning. From our prior experiments, we decided to train the *SeperateOnlyConv* model in the *No Stopping* environment. The training progress can be seen in Figure 9. The model trained for around 107M environment steps, taking a total of approximately 11 hours. Regarding hyperparameters, we set the maximum allowed mistakes for any board state to 20 before ending the game. We started with a -1 reward for incorrect moves and, after 20M steps, adjusted this to -3, aiming to encourage the agent to minimize frequency of mistakes. The learning rate was gradually reduced with warm restarts, which explains the visible fluctuations in the graph.

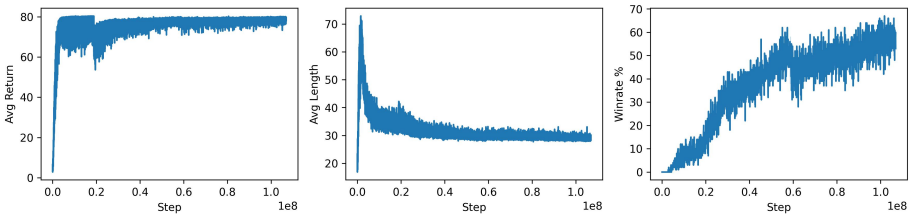

Figure 9: Training Progress of our Best Agent

Figure 10 shows the relationship between our final agent's win rate and the initial number of empty cells in a Sudoku. Notably, we evaluated our agent across 10,000 games with actions chosen greedily. As the number of empty cells increases, the win rate declines. The agent achieves nearly 100% success for puzzles with up to about 10 empty cells, but this drops to 57.34% for puzzles with 27 empty cells.

To enhance our final agent's performance, we devised an ensemble technique for Sudoku solving. Starting with an original Sudoku board state, we conduct n_{per} legal random permutations, such as swapping rows and columns within blocks or performing board rotations. Following this, we gather the actor's predictions for the permuted board state and map these predictions back to the original state. This process is repeated s_e times (representing the ensemble size). Afterward, we can either employ a majority voting mechanism over these predictions or calculate the average prediction, followed by a greedy action selection.

Illustrated in Figure 10 is the win rate of both ensemble approaches over Sudoku

puzzles with increasing numbers of empty cells. For both ensemble strategies, we set $s_e = 5$ and $n_{\text{per}} = 10$, and conducted evaluations on 1,000 games for each empty cell count. Notably, both ensemble methods augmented the win rate, achieving 70% for puzzles with 27 empty cells, with averaging method demonstrated marginally superior results.

We also explored the impact of varying s_e and n_{per} on the win rate. We evaluated a range of configurations for Sudokus with 27 empty cells over 1,000 games. As shown in Table 2, enlarging s_e or n_{per} individually enhanced the win rate. Employing our ensemble technique, we successfully elevated our agent's win rate for "easy" Sudokus from 57.34% to 78.40%.

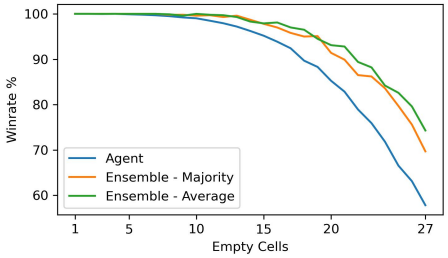

Figure 10: Performance of Best Agent

Mode	s_e	n_{per}	Win Rate (%)
Agent Only	1	1	57.34
Average	10	10	74.80
Average	10	20	75.40
Average	10	50	77.20
Average	20	10	76.60
Average	50	10	**78.40**

Table 2: Performance Across Different Ensemble Setups

6 Discussion

Incorporating the `SudokuDoubleSoftmax` into a supervised setting led to significant performance enhancements. Our implementation is particularly efficient, introducing minimal computational overhead. Viewing its positioning as the last layer, the `SudokuDoubleSoftmax` can be interpreted more as a refinement of the `CategoricalCrossEntropy` loss function rather than a standalone component. This suggests its potential for generalization across various supervised Sudoku-solving methodologies. An intriguing direction for future research could be its integration with state-of-the-art techniques, such as recurrent relation networks [10]. Such a meld might not only expedite convergence but also elevate the overall performance threshold.

Our research exposed the propensity of neural networks to overfit to distinct Sudoku orbits. This implies that even if a model is trained and evaluated on unique examples sourced solely from a limited number of orbits, there's a heightened risk of severe overfitting without any immediate indicators. Consequently, it's imperative to diversify the dataset across numerous orbits to ensure robust and generalizable performance.

Our tests with different architectures in the supervised setting produced similar results as in the deep reinforcement learning experiments. This pattern helped us move forward more quickly, since we could then focus on the other important factors. This link between supervised and reinforcement learning results might be useful for future research in this area.

We've demonstrated that solving Sudokus with reinforcement learning is feasible. However, our success so far has been limited to "easy" Sudoku puzzles, and we've masked already filled cells. The difficulty of a puzzle generally increases with the number of empty cells, a trend observed in both supervised and reinforcement learning. An attributing factor to this trend might be our current Sudoku MDP formulation. In our setup, only one cell can be filled at each decision step. Consequently, as the number of empty cells increases, the required sequence of correct decisions becomes lengthier, leading to an exponential decrease in the likelihood of completing the puzzle correctly. One potential adaptation could be to reformulate the MDP to allow multiple actions simultaneously. Additionally, introducing a confidence score for each cell, similar to the approach in [14], could provide further refinements.

Our experiments highlighted the significance of selecting the appropriate environment dynamics and architectures for achieving remarkable success. Differences in these choices led to significant improvements in learning speeds. In some cases, without the optimal architecture or environment, success might not have been feasible, even with extended training durations. Specifically, we identified that the positionally invariant *OnlyConv* models outperformed other architectures in both supervised and deep RL settings. This likely resonates with the nearly positionally invariant nature of Sudoku puzzles when viewed as a graph. In such a context, these parameter-efficient approaches excel. Thus, for further improvements in performance, seeking such architectures might be beneficial. The most adept incorporation of Sudoku's inherent structure, as measured by supervised performance, was demonstrated by [10] with their recurrent relation networks. Perhaps, integrating such recurrent graph networks with RL could lead to enhanced results. Additionally, general scaling of our approach might be a promising avenue for exploration.

References

[1] Charles Akin-David and Richard Mantey. Solving sudoku with neural networks. https://cs230.stanford.edu/files_winter_2018/projects/6939771.pdf, 2018.

[2] Daniel Brotsky. sudoku-generator. https://github.com/brotskydotcom/sudoku-generator, 2019.

[3] Shengyi Huang and Santiago Ontañón. A closer look at invalid action masking in policy gradient algorithms. https://arxiv.org/abs/2006.14171, 2020.

[4] Shengyi et al. Huang. The 37 implementation details of proximal policy optimization. https://iclr-blog-track.github.io/2022/03/25/ppo-implementation-details/, 2022.

[5] Gary McGuire, Bastian Tugemann, and Gilles Civario. There is no 16-clue sudoku: Solving the sudoku minimum number of clues problem via hitting set enumeration. https://arxiv.org/pdf/1201.0749.pdf, 2013.

[6] Anav Mehta. Reinforcement learning for constraint satisfaction game agents. https://arxiv.org/pdf/2102.06019.pdf, 2021.

[7] Valeri Mladenov, Panagiotis Karampelas, Christos Pavlatos, and E. Zirintsis. Solving sudoku puzzles by using hopfield neural networks. https://dl.acm.org/doi/10.5555/2001305.2001330, 2011.

[8] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[9] Peter Norvig. Solving every sudoku puzzle. https://norvig.com/sudoku.html, Retrieved on 6.9.23.

[10] Rasmus Berg Palm, Ulrich Paquet, and Ole Winther. Recurrent relational networks. https://arxiv.org/pdf/1711.08028.pdf, 2018.

[11] Kyubyong Park. Can convolutional neural networks crack sudoku puzzles? https://github.com/Kyubyong/sudoku, 2018.

[12] K. Poloziuk and V. Yaremenko. Neural networks and monte-carlo method usage in multi-agent systems for sudoku problem solving. https://ssrn.com/abstract=3763090, 2020.

[13] Aditya Pujari. Solving sudoku using deep q-learning. https://github.com/adityapujari98/Solving-Sudoku-using-Deep-Q-learning, 2019.

[14] Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. *CoRR*, abs/1506.02640, 2015.

[15] Ed Russel and Frazer Jarvis. There are 5472730538 essentially different sudoku grids ... and the sudoku symmetry group. `http://www.afjarvis.org.uk/sudoku/sudgroup.html`, Retrieved on 6.9.23.

[16] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization. *CoRR*, abs/1502.05477, 2015.

[17] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. `https://arxiv.org/pdf/1707.06347.pdf`, 2017.

[18] Mukund Sundararajan, Ankur Taly, and Qiqi Yan. Axiomatic attribution for deep networks. *CoRR*, abs/1703.01365, 2017.

[19] Richard S Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In S. Solla, T. Leen, and K. Müller, editors, *Advances in Neural Information Processing Systems*, volume 12. MIT Press, 1999.

[20] Christopher Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3):279–292, 1992.

[21] Tai-Wen Yue and Zou-Chung Lee. Sudoku solver by q'tron neural networks. `https://link.springer.com/book/10.1007/11816157`, 2006.