RUNI: Reinforcement Learning 3640

Final Project RL - Sokoban

Fall Semester 2022-23 TASHPAG

Submitted by: Yarden Fogel ID: 011996279 (yarden.fogel@post.runi.ac.il)
Partner: Sharon Koubi ID: 301315040 (sharon.koubi@post.runi.ac.il)

Sokoban Puzzle: Academic Paper

1 Introduction

For this project we were tasked with solving the Japanese puzzle game Sokoban, in which the player has to move boxes to their target locations, by applying deep reinforcement learning (Deep RL) techniques. The project consisted of two distinct challenges - and fixed board environment, and a dynamic randomly-generated board. In the fixed board environment, the board is initialized with the same seed each time, and the box and target positions remain constant. In the randomly-generated board exercise, the locations of the box and target change on every reset, and the agent needs to learn to adapt to every possible environment setup.

Sokoban (meaning 'warehouse keeper') is a 2D grid-based puzzle video game in which the player pushes crates or boxes around in a warehouse, trying to get them to storage or "target" locations. The game has a set of discrete actions and the state transitions resulting from those actions are deterministic. The puzzle is considered solved when all boxes are on target storage locations. The Sokoban environment we worked with is a 7x7 board that contains empty floor spaces, walls, boxes, box targets and the player. The standard Sokoban setup is push-only but in our case we played with the push-pull variation, in which the player can also pull the box. The action space as initially set consists of 13 possible moves, including to push, pull, or move in any direction (up, down, left, right), as well as action 0 which is to stay put.

Sokoban is known to be a challenging game for traditional RL agents as there are many possibilities for being stuck in dead-ends and other endless loops, and avoiding these traps thus requires a degree of long-term planning that is not readily available for standard RL solutions. The amount of possible state permutations also leads to an enormous state space. In fact, the classic push-only Sokoban is considered to be a PSPACE-complete decision problem and an NP-Hard problem. (Sokoban Wiki Page)

Throughout our work on this project, we explored countless interesting approaches to solving this puzzle, with most of the intellectual horsepower devoted to solving exercise 2 and the randomly-generated board. While we generally employed a Deep Q-Learning Network (DQN) model to solve both exercises, that would grossly understate and oversimplify, as we experimented with an enormous amount of parameters, algorithms, constructs and heuristics, and that's where all the learning happened, both for the agent, and for us. One final note in this lengthy intro is that as the reader will notice, we decided to make our report light on pictures and heavier on words, as we explored and researched so many interesting concepts and had way more to say than could fit in the allotted maximum pages. Our linked project notebook is full of informative graphics, videos, and clear explanations throughout, and we wanted to supplement that with as much transparency into our thought process as possible in this report.

1.1 Related Works and Sources of Inspiration

In contrast to our work on the MidTerm maze project, the sheer quantity of references and interesting related research we studied and learned from, and their impact to our overall thought process and strategies was astounding. Below we highlight our sources of education and inspiration for this project, grouped into a few thematic categories.

Categories of References and Educational Material:

- Model selection: Q-Learning, DQN, DDQN, Dueling-DQN, PPO, A2C, etc.
- Guides on DQN architectures and implementation for Keras and PyTorch, visualization techniques, etc.
- Planning ahead, target network, exploration vs. exploitation
- Transfer learning, curriculum learning
- Reward shaping, heuristics, and other novel approaches

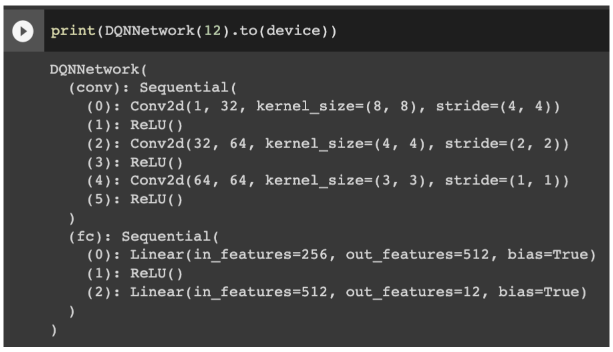
In the link that follows, we have included the nearly **one hundred** references that we combed through and benefited from as part of our overall learning process and Sokoban-solving journey. In addition, we of course leaned heavily on the algorithms, models, and in our view most importantly, the intuitions that we were presented with and taught during the course lectures by Dr. Moshe Butman. Links for resources and references used throughout the project

2 Overview of Our Solutions and Approaches

2.1 Our General Implementation Setup

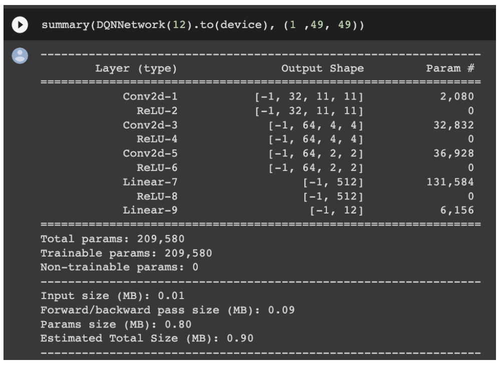
2.1.1 DQN Architecture:

Since we used the DQN framework as the foundation for all of our solutions and experiments, albeit with tons of variations and tweaks, we start by describing our underlying NN architecture, which at its core did not change throughout. While it was our intention to experiment with many more network configurations, we will file that under ideas for further future work. Our primary architecture was based on the configuration used in the paper by Yang, et al (Potential-based Reward Shaping in Sokoban), as depicted in the figure below:



```
print(DQNNetwork(12).to(device))
```

```
DQNNetwork(
  (conv): Sequential(
    (0): Conv2d(1, 32, kernel_size=(8, 8), stride=(4, 4))
    (1): ReLU()
    (2): Conv2d(32, 64, kernel_size=(4, 4), stride=(2, 2))
    (3): ReLU()
    (4): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1))
    (5): ReLU()
  )
  (fc): Sequential(
    (0): Linear(in_features=256, out_features=512, bias=True)
    (1): ReLU()
    (2): Linear(in_features=512, out_features=12, bias=True)
  )
)
```



```
summary(DQNNetwork(12).to(device), (1 ,49, 49))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 32, 11, 11]           2,080
              ReLU-2           [-1, 32, 11, 11]               0
            Conv2d-3             [-1, 64, 4, 4]          32,832
              ReLU-4             [-1, 64, 4, 4]               0
            Conv2d-5             [-1, 64, 2, 2]          36,928
              ReLU-6             [-1, 64, 2, 2]               0
            Linear-7                  [-1, 512]         131,584
              ReLU-8                  [-1, 512]               0
            Linear-9                   [-1, 12]           6,156
================================================================
Total params: 209,580
Trainable params: 209,580
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.01
Forward/backward pass size (MB): 0.09
Params size (MB): 0.80
Estimated Total Size (MB): 0.90
----------------------------------------------------------------
```

Figure 1: Backbone DQN Neural Net Architecture

2.1.2 Training Configurations

All of our training and experiments were run using a cfg variable for configuration, where we could use the default agent configuration, or input different values for any or all parameters and arguments. We then call our DQNTrainer class using the cfg variable instead of populating that instantiation with many arguments, with the intent being clarity of code and clearly deciphering what is behind each particular training run.

After doing a run of 100 episodes (or until early stopping), we save the best path so that it can be used to build on in subsequent runs, and also generate the video for the run's best model path. We then also output the agent's performance midway through training (whatever the midpoint is for that run), and have a mechanism to generate the same output data for intervals of 10, 30, and 50% through training.

2.1.3 Convergence and Early Stopping

Our implementation used an early stopping mechanism to improve efficiency in cases where the learning improvement flat-lined, and in an attempt to prevent overfitting. By default, we set the minimum early-stop trigger at 30 episodes, a delta (or change in rewards) of 0, and patience of 20 episodes, meaning the agent would continue to run until the plateauing lasted for 20 episodes. We thought this through carefully, as we did not want to miss potential continued gains in learning after short "coffee breaks" by the agent. All of the above defaults can be configured and changed. Also as a small sidenote, we added a train_safe method so that if a training run is taking too long and the user terminates the execution, all is not lost, and whatever was generated until that point is still accessible, which was a nice feature when at times the agent would run for many hours!

Convergence:
We wanted to spend a minute discussing what it means for the agent to "converge" in the context of Sokoban, or "success" for that matter, as there seemed to be little consensus on this issue among our classmates, with the primary options being some combination of rewards, losses, and solving the puzzle, and which thresholds constitute convergence. Importantly, defining convergence can be a trade-off of properly characterizing the agent's performance vs. computational efficiency, along with considerations of practicality. It was our view that basing convergence on solving the puzzle or reaching an absolute global rewards maximum was both impractical and unreliable, though we did keep track of the agent's "success rate" in solving the game as one of our outputs, along with number of steps and average steps. As the goal of an RL agent is to maximize cumulative future rewards, and after visually observing the agent's learning, we decided that using rewards to define convergence (and early stopping) was most appropriate. Also, since the loss measures the agent's ability to accurately predict future rewards, a declining and low loss does not always coincide with improving rewards, as we observed in several of our experiments. We will expand on the subject of rewards in the sections that follow.

2.1.4 Visualizations

We researched a fascinating variety visualization techniques for RL agent performance, and could spend months on that alone, we ultimately kept it simple and primarily used two basic graphs for each run, charting rewards and losses against episodes, and detail each below.

Rewards: The first chart on the left at the end of each run measures rewards on the Y-axis and episodes on the X-axis. The blue line is the reward for each episode, which is quite spiky and volatile, and that seemed to coincide somewhat with the updating of the target model. The

green line plots the average rewards per episode, so the goal is to have this line increasing and sloping upward. The orange line also smooths out the rewards, taking the average reward for the 10 most recent episodes. The dotted line signifies when early-stopping was triggered, and the red dot singles out the best path for that entire training run.
Losses: The second chart (on the right) plots the losses by episode, where we show the loss for each episode, average losses for all episodes, and the average loss of the 10 most recent episodes.
Best model: At the end of each exercise we run the best model over 100 episodes and plot the standard deviation of rewards and show the agent's success rate. In Ex1, with a fixed board that doesn't change, the std is of course 0 and the success rate is unsurprisingly 100%.

2.1.5 Platform use and computational resources footnote

We wanted to briefly touch on the issue of resources and runtime. We ran into this issue in the MidTerm assignment, only this time it became much more acute. We started and ended the project using Google Colab (Pro), and ran through several subscriptions of Colab Pro that used up all the compute units, but along the way attempted to run locally and leverage the GPU power of the most advanced Apple M2 chips. We replicated some experiments we found online and found a 7-8x speedup in Apple M2 GPU runtime (or 80-90% reduction), but couldn't consistently work across our machines in a simple-enough way for the Sokoban solver.

2.2 Important Solution Components and Features

2.2.1 Note on DQN vs. PPO

We wanted to briefly touch on our choice of model for these exercises, as we thoroughly researched every possible approach available to us. Proximal Policy Optimization (PPO) had some appeal as it can be faster to converge, its trust region optimization can prevent overly large swings in policy updates and stabilize learning, and its incorporation of an entropy term in the loss function helps encourage exploration, which can be particularly useful in solving complex environments. Since Sokoban is a discrete action space, we found DQN to be more suitable, as it is simpler to implement and interpret. Since DQN learns the action-value function that maps state-action pairs to expected rewards, it is clearer to interpret the learning and understand how the agent is making decisions. Lastly, we wanted to implement the purported DDQN improvement of predicting the next action using the base Q-network and then mapping that action's reward **value** from the target Q-network, but ultimately we left this as an area for future work and improvements.

2.2.2 Loss Function, Optimizer, and Random Agent

We tested both the mean squared error (MSE) loss and the Huber loss (more specifically the Smooth-L1 loss), but we primarily used the Huber loss in our training and experiments as it is less sensitive to outliers than MSE (which can happen when the agent encounters previously unseen states or makes mistakes during training) and in some cases can prevent exploding gradients. Lastly, by learning a smoother function with the Huber loss, it can help to combat overfitting, making the model more generalizable. Since the Huber loss is a hybrid loss function, we believe it struck a good balance between stability and accuracy.

We evaluated stochastic gradient descent (SGD) and Adam optimizers, and primarily used Adam since it combines the elements of SGD with adaptive momentum, which based on our experience in ML/DL should lead to more accurate and stable learning. Lastly, at the outset of each exercise, we ran a random agent that chose actions randomly from the available action space, in order to have a benchmark that we could compare our agent's performance against.

2.2.3 Rewards Shaping

One of the main challenges of Sokoban for traditional RL agents is the sparse rewards, in that the positive reward comes only from the distant outcome of the game being solved, and the agent lacks guidance towards that goal, which makes planning ahead even more difficult. Adding reward shaping adds additional rewards to guide the agent towards the goal along the way, and leverages prior knowledge to accelerate learning and convergence. Reward shaping strategies can be sparse, extremely dense (where every action has a meaningful reward consequence), or something in between, and can be positive or negative (punishments). Various combinations can also be used, along with human-guided heuristic solutions, so the possibilities are endless. Below we briefly describe the four additional reward shaping strategies we employed, and later in the report we will come back to other ideas for further experimentation, including intrinsic motivation, curriculum learning, and environment-specific heuristics.

Our reward shaping strategies included four variations, which we deployed in various combinations during our experiments, and are described below.
RS Strategy 1 - Punishment for the no movement 0 operation: This one is pretty self-explanatory, in that there was a -1 penalty when the agent chose action 0.
RS Strategy 2 - Punishment for player not moving: Here we penalized the player when it failed to execute the specified action in its given state. The logic behind this was to ensure the agent moved as instructed, to complete the game in the minimum number of steps possible, and to address getting stuck in one place, which we encountered several times.
RS Strategy 3 - Punishment for push/pull action without moving the box: This one too is pretty straightforward in that we didn't want the player to "waste" push/pull actions when it didn't end up using those push/pulls to move the box.
RS Strategy 4 - Punishment for box-target distance: Finally, and in line with the aforementioned paper and basic intuition, we added an adaptive punishment that changes as a function of the box-target distance, in order to create more incentive for the player to always be moving the box closer and closer to its target location.

3 Experimental Results - Exercise 1 Fixed Board

3.1 Ex1 Fixed Board - DQN

As mentioned above, we first ran a random agent in Ex1 to establish a benchmark, and based on 100 episodes, the random agent had average reward of -50, 500 average steps, and a 0% success rate. We really expended the vast majority of our effort and creative power on Ex2, so our analysis of Ex1 is short and to the point. We also saved all our hyperparameter tuning and interesting experiments for Ex2, as we will show below. Our agent was able to solve the fixed board in 9 steps with an average reward of $+10.1$.

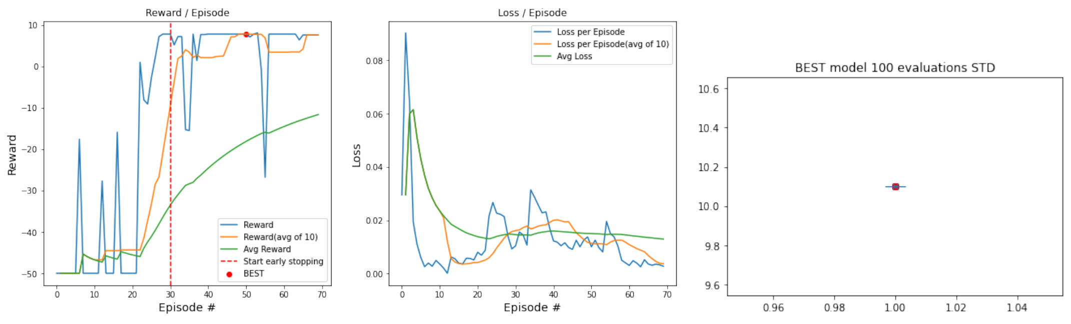


Figure 2: Ex1 Fixed Board - Rewards and Losses by Episode, and Best Model Reward STD

For the sake of robustness and overkill, we added additional testing with various random seeds to ensure that our agent wasn't just a "seed(2) expert", but could apply its skills and learning prowess to any fixed environment.

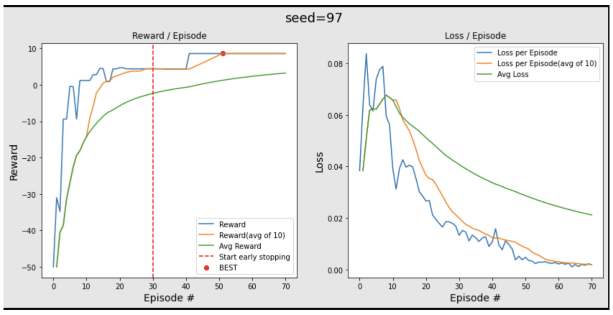


Figure 3: Ex1 Fixed Board - Robustness Check on Different Seed Inits - Example Seed = 97

3.2 Exercise 1 - Fixed Board: Conclusions and Insights

To reiterate, our insights here will be limited, but we wanted to highlight two additional thoughts and takeaways. The first is that we are well aware that a plain-vanilla Q-Learning agent would suffice to solve the fixed board environment, and would do so **MUCH** faster than DQN. While it would be interesting to compare in the future and expand on the advantages of each approach, we chose to let standard Q-Learning rest in peace in the MidTerm project. Our second point is related, in that the agent's excellent performance, and the ability to solve with Q-Learning, emphasize the significance of the size of the state space on the RL agent's ability to plan ahead and learn, because in the fixed board, the state space is very small and finite, so with enough training, it is no match for any flavor of Q-Learning or DQN, but as we'll soon see below, that was all about to change...

4 Experimental Results - Exercise 2 Random Board

We again started with the random agent to establish a benchmark. For better or worse, by wild chance, the random agent actually performed decently by some measures, and had an average reward of -31.72, about 361 average steps, and a surprisingly-high success rate of 40% across 100 evaluations. But this random chance luck is clearly on display in the enormous standard deviation of the outcomes for the random agent, as seen in the graphic below:

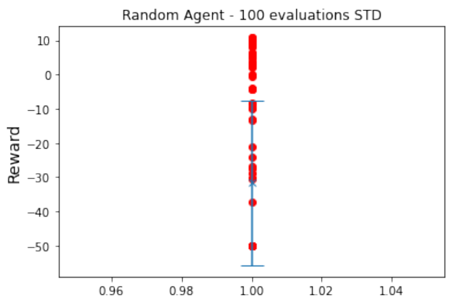


Figure 4: Ex2 Random Board - Random Agent Strikes (Fool's) Gold

4.1 Ex2 Hyperparameter Optimizations

4.1.1 Gamma γ Discount Factor and Learning Rate

One important caveat to our hyperparameter tuning experiments is that we ran them all *before* introducing reward shaping and other heuristics to improve performance and deal with specific challenges encountered. We first tested several values of the discount factor $\gamma = [0.1, 0.3, 0.5, 0.7, 0.9]$, which determines the importance of future vs. immediate rewards, and in the base construct of Sokoban this is a critical factor given the extreme sparsity of the positive reward (+10), residing only at the terminal state of solving the puzzle, compared to the immediate rewards of -0.1 for every step taken. As seen in the charts below, $\gamma = 0.9$ gave us the best convergence and the highest success rate, which made perfect intuitive sense, as it encourages the agent to consider a wide range of future rewards and leads to more consistent estimates of the values of state-action pairs. This also explains the wider standard deviations observed with $\gamma = 0.9$ as can be seen on the right-most chart below.

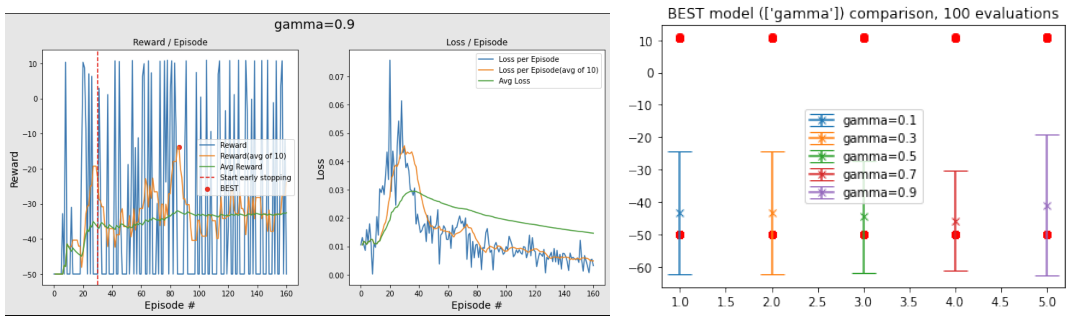


Figure 5: Ex2 Random Board - Gamma γ Tuning

We briefly just mention here that we also tested various learning rates, or the size of the steps taken during gradient descent and learning, as this is usually an influential hyperparameter choice. In our case though, we really didn't observe any noteworthy differences that popped out and singled out any single LR value as superior. The average rewards ranged from -41 to -46, average steps were all in the 400's, and the success rates were underwhelming, ranging from 6 to 15%. We were somewhat surprised, however, at the fact that the results improved as the LR got smaller and smaller, which was counter to our intuition.

4.1.2 Replay Memory Buffer, Batch Size, Target Network Weight Updates

Memory Buffer Size: In general using experience replay helps to break the successive correlation among samples and allows for more effective learning. The memory buffer size determines how many experiences are stored in the replay buffer, and thus a larger buffer size, at least in theory, can lead to better performance by allowing the agent to sample a broader variety of situations and improve the robustness of learning. Our research suggested that there may be diminishing returns to this logic, especially as older experiences are forgotten and overwritten with new ones filling the buffer. We tested memory buffer sizes of $1000, 10000, 30000, and 50000$. Our experiments indeed revealed the diminishing return of larger and larger buffers, in that the performance improved from 1000 to 10000, with average rewards, steps, and success rate of -39.7, 415, and 17%, respectively, dropped slightly at 30000, and then fell off precipitously at 50000, with the same metrics coming in at -45.1, 460, and 8%.

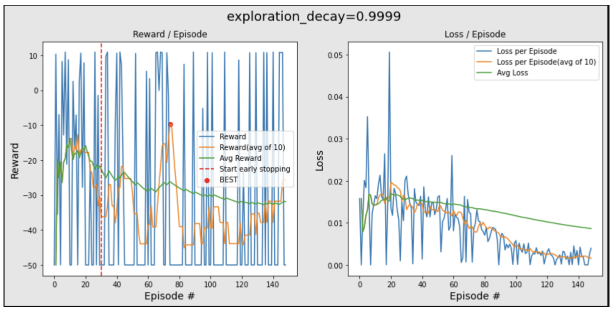
Batch Size: We tested batch sizes of $[1, 16, 32, 256]$, and there wasn't anything too exciting here, and batch size of 32 performed the best, but the differences were insignificant. What *was* slightly interesting here is the behavior of each batch size throughout the experiment, as they each arrived at their averages in wildly different ways, with some having better average results but huge swings along the way, which can be seen in the visualizations provided in the notebook.

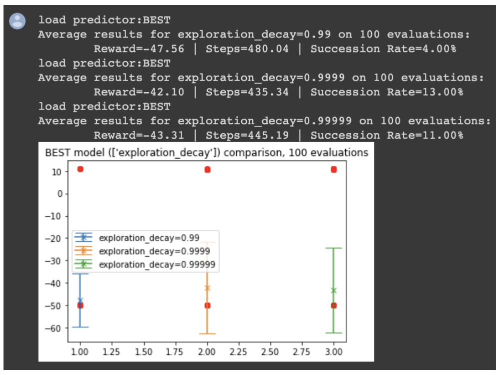
Target Network Updates: While very frequent updates can lead to faster learning as the agent gets more recent information, it generally leads to instability in the learning process, and we in fact observed this, as we pointed out previously how the target network updates coincides with the wild gyrations in rewards by individual episode. We tested values of $100, 500, 2000, and 10000$ for the amount of steps between copying weights to the target network, and our intuition described above indeed played out as expected, in that a middle ground that strikes a balance between getting updated information and stable learning results in the best outcome. The results were best using update intervals of 500 steps, with average rewards, steps, and success rate of -40.9, 425, and 15%, respectively, and the results for intervals of 2000 steps did only marginally worse.

4.1.3 ϵ Exploration Parameters

Max/starting *epsilon* or exploration rate: We will focus our discussion of exploration primarily on the *epsilon* decay factor below, but will briefly touch on starting max ϵ and minimum ending ϵ as well. We tested starting values of $\epsilon = [0.3, 0.5, 0.9]$ and the important takeaway is that a very low starting value results in insufficient exploration at the start of learning and poor performance across the board. In other words, it's important to explore! **Min/ending ϵ:** We then tested a few minimum ending values (after decay) of $\epsilon = [0.02, 0.3, 0.7]$, and while the differences are again not enormous, $\epsilon = 0.02$ performed the best with average rewards, steps, and success rate of -40.3, 420 and 16% which rounds out the view on exploration vs. exploitation in that while it's critical to include plenty of exploration early in training (higher starting max ϵ), there also comes a time where the fun and games of freely exploring must come to and end, and the agent is better served to focus on exploitation and have its "eyes on the prize" to an increasing degree as learning progresses.

ϵ Decay Factor: At every experience replay step, we multiply the exploration rate by this decay factor, which controls the pace of the decline in exploration towards more exploitation, and this turned out to be one of the most important parameters in our experiments. Here we tested values of ϵ decay factors of $[0.99, 0.9999, 0.99999]$, and the results were quite interesting with the middle value of 0.9999 outperforming the other two, and 0.99 by a very wide margin, with average rewards, steps, and success rate of -42.1, 435 and 13%, respectively. This suggests that with a decay factor of 0.99 the exploration was cut short too quickly and degraded performance, while at the other extreme of 0.99999 it kept the agent in exploration for more time than it needed, and this could be clearly seen in the reward and loss graphs for each as well, and we provide the key data and graphics below:





```
load predictor:BEST
Average results for exploration_decay=0.99 on 100 evaluations:
        Reward=-47.56 | Steps=480.04 | Succession Rate=4.00%
load predictor:BEST
Average results for exploration_decay=0.9999 on 100 evaluations:
        Reward=-42.10 | Steps=435.34 | Succession Rate=13.00%
load predictor:BEST
Average results for exploration_decay=0.99999 on 100 evaluations:
        Reward=-43.31 | Steps=445.19 | Succession Rate=11.00%
```

BEST model (['exploration_decay']) comparison, 100 evaluations
exploration_decay=0.99
exploration_decay=0.9999
exploration_decay=0.99999

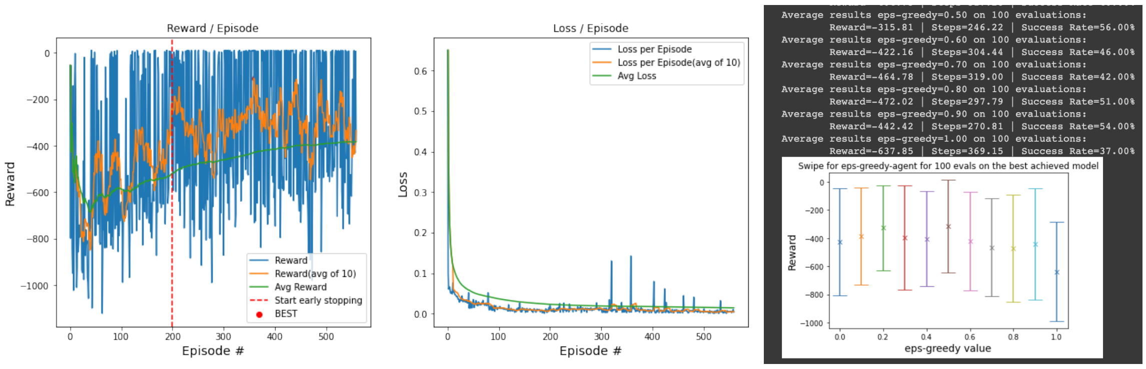
Figure 6: Ex2 Random Board - Exploration ϵ Decay Factor Tuning

4.2 Ex2 Continued Optimization

After tuning all those hyperparameters, we ran a model with each of the best-performing parameter values: $\gamma = 0.9$, lr$= 0.00025$, buffer size $= 10000$, batch size $= 32$, target-update interval $= 500$ steps, max-$\epsilon = 0.9$, min-$\epsilon = 0.02$, and ϵ-decay $= 0.9999$. But much to our dismay, the agent's performance fell flat on its face, with metrics that would be better left unmentioned in this report. We identified the issue as being caused by a deterministic network output, meaning that given a particular state, the network would always output the same suggested best action. So if the agent got stuck or sent back to the same state, the same action would be given by the network, and the "Loop of Death" was born. This is a form of policy mismatch and distribution shift, which is commonly encountered in RL problems, and we had to address it to improve our agent's performance.

This inspired us to really start utilizing reward shaping. While we discussed the key aspects of our reward shaping strategies above, we'll just point out here that reward shaping markedly improved our results, with success rates ranging from as low as 20% up to 36%, but we observed instances of overfitting when exploration was eliminated entirely, and what was more troubling is that while our agent now more effectively avoided getting stuck in the same place, it still wasn't able to avoid getting stuck in loops, no matter how small or large. To combat this, we ran our reward shaping modified agent to use an **epsilon-greedy approach** to encourage more exploration. While the maximum ϵ value of 1 resulted in higher success rates, our goal was to solve the game in the minimum number of steps and maximize cumulative rewards, so we used $\epsilon = 0.9$ in combination with the best algorithm combination, and this resulted in by far the most promising results with a whopping 54% success rate and an average of only 299 steps.

Finally, we wanted to have some long-running experiments, as all the previously trained models were stopped prematurely due to limited compute resources. To address this, we took the best parameters we had obtained thus far, combined with reward shaping, and trained the algorithm for an extended number of epochs, with maximum episodes of 250 and effectively without early stopping, as the trigger and patience variables were set. We used this opportunity of more training to increase the exploration phase. We then compared various ϵ-greedy values for our best models, and at the end of training with ϵ-greedy value of 0.5, we were able to achieve **our best result, with a success rate of 56% on only 246 average steps for solving 100 different random Sokoban boards.** We considered this result to be superior to one run that achieved a higher success rate of 58%, as that was done with an average of 284 steps.



```
Average results eps-greedy=0.50 on 100 evaluations:
        Reward=-315.81 | Steps=246.22 | Success Rate=56.00%
Average results eps-greedy=0.60 on 100 evaluations:
        Reward=-422.16 | Steps=304.44 | Success Rate=46.00%
Average results eps-greedy=0.70 on 100 evaluations:
        Reward=-464.78 | Steps=319.00 | Success Rate=42.00%
Average results eps-greedy=0.80 on 100 evaluations:
        Reward=-472.02 | Steps=297.79 | Success Rate=51.00%
Average results eps-greedy=0.90 on 100 evaluations:
        Reward=-442.42 | Steps=270.81 | Success Rate=54.00%
Average results eps-greedy=1.00 on 100 evaluations:
        Reward=-637.85 | Steps=369.15 | Success Rate=37.00%
```

Figure 7: Ex2 - Long-Run + RS + ϵ-greedy: 56% Success on 246 Avg Steps!

4.3 Exercise 2 - Random Board: Discussion and Overall Insights

While we've covered most of our challenges and learnings in the preceding sections, one recurring theme that we'd like to highlight is that throughout our work, we observed that every situation comes with certain trade-offs. Accuracy on one side, stability and speed on the other. Keeping it simple and not over-complicating vs. the potential benefits of further tweaks, of which there are infinite options to choose from, planning ahead longer-term vs. surviving in the immediate future, and so on. And lastly, we once again come back to the classic RL trade-off of exploration vs. exploitation, which was apparent at every critical juncture in this project. On the one hand, experimentation and tinkering and exploring are necessary in order to properly learn and uncover possible novel solutions, and it goes against our intuition to not be greedy and short-sighted. On the other, there comes a time when to get down to business and focus more crudely on the bottom line and exploit. That really stuck out and left a mark.

5 Ideas for Further Experimentation and Improvements

There are countless other avenues we would have wanted to explore further, and surely plenty of opportunities to improve our agent's performance. We break some of those down in two primary buckets below:

Network Considerations: We would have loved to spend more time trying other variations of DQN, PPO and A2C, and as mentioned the DDQN argmax-from-network approach. We could also experiment with many characteristics of the underlying neural net, including architecture, number of layers, fully-connected vs. convolutional layers, different kernels and activations, pooling, drop-out, regularization and batch normalization, etc.

Sokoban Action Space and Reward Shaping: One other idea that we contemplated was simplifying the action space to remove the 0 operation, and also the four "move" operations, since moving up is the same as push up when there is no box there to push. We're not certain of how "kosher" this would be, and which actions to eliminate, but it would be interesting experimentation. Regarding reward shaping and learning, we'd be interested to try curriculum learning, gradually increasing the difficulty as the agent learns, but weren't overly encouraged by what we read on the topic. Some kind of intrinsic motivation reward shaping giving the agent incentive to innovate and make breakthroughs would also be of interest, and here specifically and for reward shaping more generally it would be interesting to evaluate different scales of rewards and punishments, like having some outsized one-off rewards for the agent discovering something novel and thinking "outside the box", or having some kind of compounding multiplicative dynamic in reward calculations. Lastly, there is plenty more to explore in terms of heuristics that reflect human intuition in dealing with particular problem cases, though that can also create new unforeseen issues, but is interesting nonetheless.

6 Code Links

Google Colab Sokoban-Solver Notebook: Fogel-Koubi.RL-Final.Colab

[0]Compiled with LaTeX.