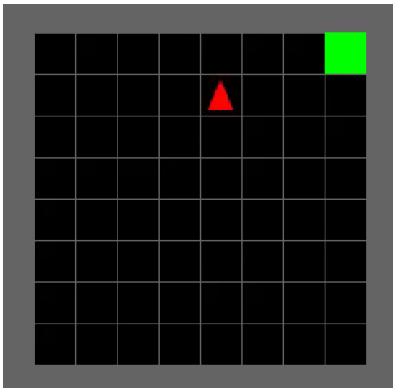
Asaf Mesilaty

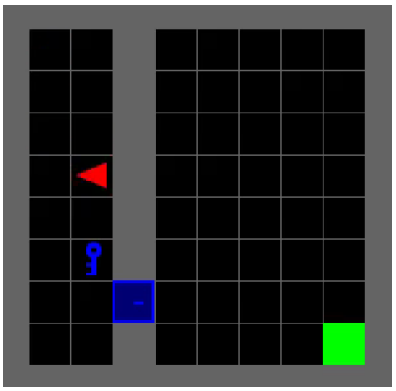
Reinforcement Learning- Final Project 2024

In this exercise we will solve two versions of the MiniGrid environment.

MiniGrid is a 2D environment with goal-oriented tasks. The agent is a red triangle with discrete action space.

The environments look like this:





In each of them the actions are as follows:

Empty MiniGrid: turn left, turn right, move forward.
Key MiniGrid: turn left, turn right, move forward, pickup, toggle.

Reward shaping:

As part of the exercise we must define a reward system, with the help of which the different algorithms will learn.

Environment documentation: https://minigrid.farama.org/environments/minigrid/

The exercise is divided into two parts:

Part A: tabular Q learning:

we will solve the environments using Q-Learning so that every state and action have a value saved in memory. The agent's course of action (Policy) is determined according to these values.

Part B: Deep learning:

we will solve the environments using 3 different methods that use deep neural networks (In my case: DQN, Actor Critic, PPO).

Asaf Mesilaty

Part A: tabular methods:

Goal- solve the problem as fast as you can (less episodes).

Environment analysis:

Markov decision process (MDP) is a discrete-time stochastic control process. It provides a mathematical framework for modelling decision making in situations where outcomes are partly random and partly under the control of a decision maker.

This environment is MDP since it is defined by states, actions, and rewards, where each action outcomes only one possibility.

Episodic is an environment where each state is independent of each other. The action on a state has nothing to do with the next state. This environment is episodic since each state can be solved regardless of what was the previous state and what action was taken.

The action space is discrete as mentioned above.

Lastly the environment is fully observable- for each state we know all the game data.

Q-Learning solution:

Q-learning is a model-free reinforcement learning algorithm to learn the value of an action in a particular state. In reinforcement learning, a model-free RL algorithm can be thought of as an "explicit" trial-and-error algorithm.

Tabular DQN algorithm:

Create Q table of size |S| x |A|

Initialize values in the table.

Until learning is stopped do:

- Choose an action based on current Q value.
- You get a reward r. You are now in state s'.
- Update Q(s, a): $Q(s,a) \leftarrow Q(s,a) + \eta\left(r + \gamma \cdot \max_{a'}(s',a') - Q(s,a)\right)$

Where η is the learning rate and γ is the discount factor.

I defined the state to be:

Empty MiniGrid: agent position, agent direction, goal position.
Key MiniGrid: agent position, agent direction, goal position, key position, door position, is carrying key, is door open.

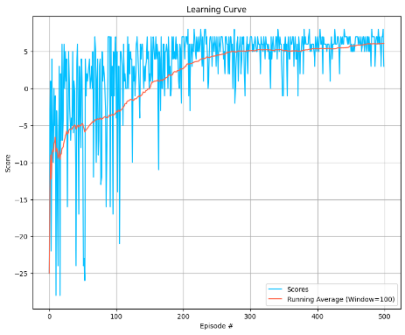
The Memory needed for the algorithm: $|Q-Table| = |state| * |number\ of\ actions|$.

The reward system gives penalties for redundant moves while encouraging crucial moves. You can see the exact system get reward function in the code.

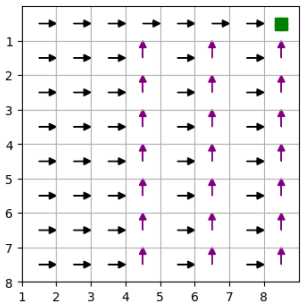
Asaf Mesilaty

Graphs:

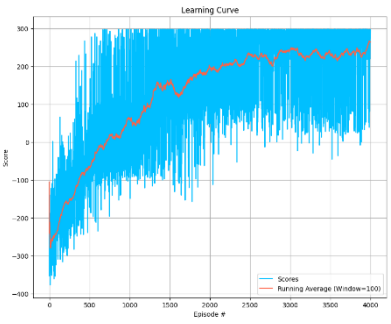
Empty environment:



Q-Table visualization:

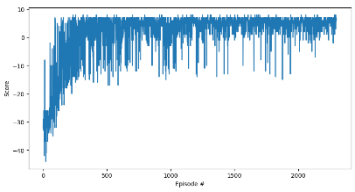


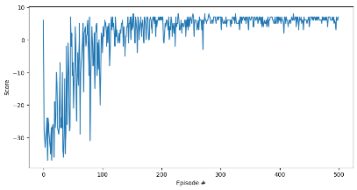
Key environment:

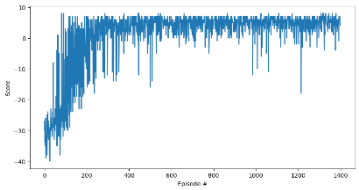


Hyperparameters:

Learning rate- have much impact on the convergence time, it should be around 0.5:







From left to right we the scores when the learning rate equals 0.1, 0.5, 0.7.

I defined convergence when the average of the last 100 scores reach the target reward goal. As you can see, for 0.1 it is about 2000 episodes, for 0.5 it is about 400 episodes, and for 0.7 it is about 1400 episodes.

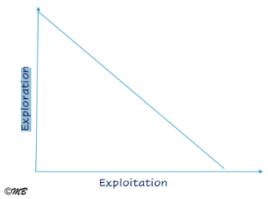
Discount factor (aka Gamma)- if I kept it in the range of (0.8-0.99) the performance did not change much. If I set low gamma, the algorithm did not got to the optimal reward.

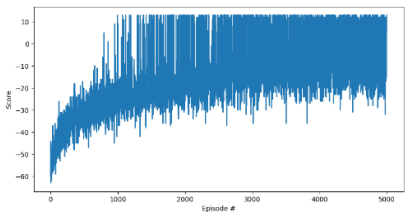
Epsilon- Setting the start epsilon to be between 0.4 to 0.6 yielded the best results since it was a good Exploration-Exploitation for the beginning.

Another exploration technique that was used is starting each time in a random position.

Of course, I used epsilon decay because as we learn our algorithm knows better.

We the I choose slow epsilon dacey the algorithem learnt very slowly, as shown in the graph:





Initialization- the action that should be taken most of the time is move forward so I checked if initializing the table with preference for the move forward action converges faster. It is not.

Asaf Mesilaty

Conclusion:

The initialization did not make much difference since the algorithm learnt fast that it should not make much turning actions.

Low learning rate made the change more subtle.

Low Gamma prevents the q table to update on better solutions.

Starting epsilon makes huge difference on the rest of the learning, make it large will make the model learn slower.

The min epsilon must be small enough for the model to get to the optimal score during training.

Different Consideration for each environment:

The random key environment is more complex hence we need a more complex reward system. I divided it into subgoals and each subgoal got a meaningful reward.

Since we use Tabular Q-Learning we can give large penalty for actions we do not want to happen at all, for example- redundant moves like: Run into a wall, Pickup nothing, Toggle nothing, Closing the door.

Another thing I used to make the learning smoother, was to choose randomly from the distribution defined by the SoftMax of the Q values, this approach for the key environment changed the number of episodes required for convergence drastically.

Number of episodes to solve the Empty environment: 600 episodes.

Number of episodes to solve the Key environment: 3500 episodes.

Tabular Q-Learning: Good Points

1. Simplicity and Interpretability: Tabular Q-Learning is straightforward to implement and understand.
2. Convergence Guarantees: Under proper conditions (all state-action pairs are visited infinitely often), it guarantees convergence to the optimal action-value function, providing a solid theoretical foundation.
3. Effectiveness in Discrete, Small Spaces: It works very well in environments with small and discrete state and action spaces where the Q-table can fully represent the environment.

Tabular Q-Learning: Bad Points

1. Scalability Issues: It becomes impractical in environments with large or continuous state spaces, requiring an enormous amount of memory and computation.
2. Lack of Generalization: Tabular methods do not generalize across states; they must learn the value of each state-action pair independently, missing out on potentially useful abstractions.

Asaf Mesilaty

Part B: Deep RL:

I solved the problem using 3 different Deep RL algorithms:

1. DQN extends Q-learning to large state spaces by using a deep neural network to approximate the Q-value function, stabilizing learning with techniques like experience replay and fixed target networks.

2. Actor-Critic methods maintain two models: an actor that updates the policy distribution in the direction suggested by the critic, which evaluates the action taken by the actor by computing value functions.

3. Actor-Critic with PPO: PPO improves upon policy gradient methods by using a clipped surrogate objective function, making it easier to train by taking multiple steps of gradient ascent on the same batch of data.

Let's discuss some advantages and disadvantages of our approaches relating to the MiniGrid mission.

Deep Q-Network (DQN):

Advantage: Stability and Efficiency: DQN introduces experience replay and fixed Q-targets, which stabilize the learning process by breaking the correlation between consecutive samples and reducing the variance in updates.

Disadvantage: Overestimation Bias: DQN can overestimate Q-values due to the max operator used in selecting actions. This issue was somewhat mitigated due to the use of zero rewards.

Actor-Critic Methods:

Advantage: Actor-critic algorithms offer enhanced efficiency compared to basic policy gradient methods by employing the critic's value function as a baseline, effectively reducing the variance of the policy gradient, and thus requiring fewer samples to achieve convergence.

Disadvantage: Complexity and Tuning: When using a shared network architecture with two heads for the actor and critic in Actor-Critic methods, the complexity increases as does the need for careful hyperparameter tuning, due to the intertwined nature of the policy and value function updates within a single network.

Proximal Policy Optimization (PPO):

Advantage: Sample Efficiency: PPO is more sample-efficient compared to traditional policy gradient methods due to its effective use of each sample through multiple epochs of minibatch updates.

Disadvantage: Complex Implementation: Implementing PPO can be more complex due to the need for managing multiple epochs and minibatches, as well as balancing the clipping mechanism.

Asaf Mesilaty

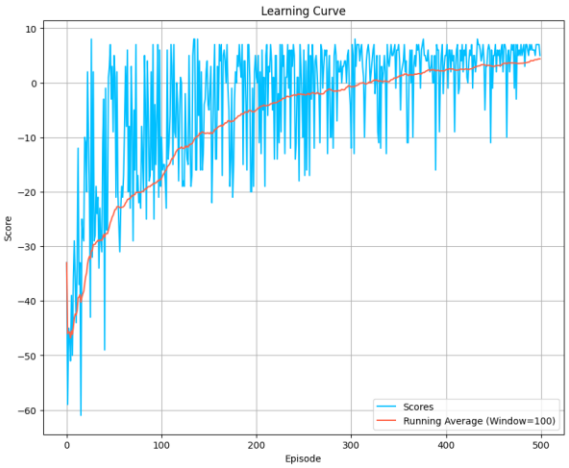
The DQN was shown to be the best algorithm for the task being both fast and accurate.

The model that worked best for the empty environment was the fully connected DQN. I tried CNN DQN as well and it yielded similar results, with one exception- It took about 4 times longer in computational time. The FC DQN was both fast and accurate.

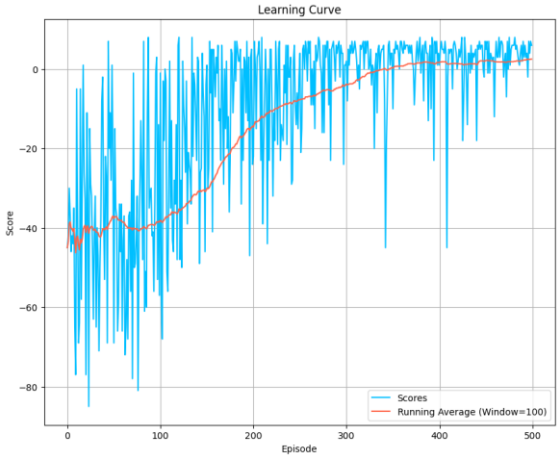
For that reason, I used it to solve the Key environment as well.

Learning curve of each of the algorithms used to solve the empty and key environment:

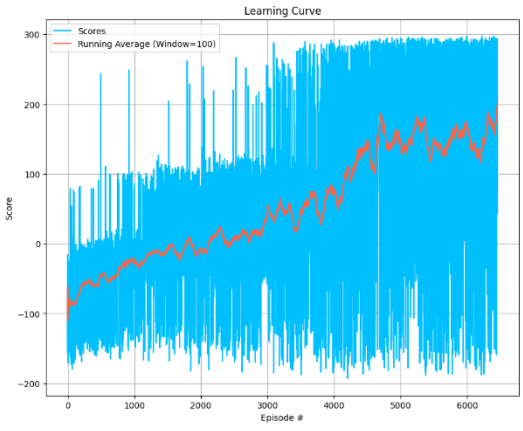
DQN Empty:



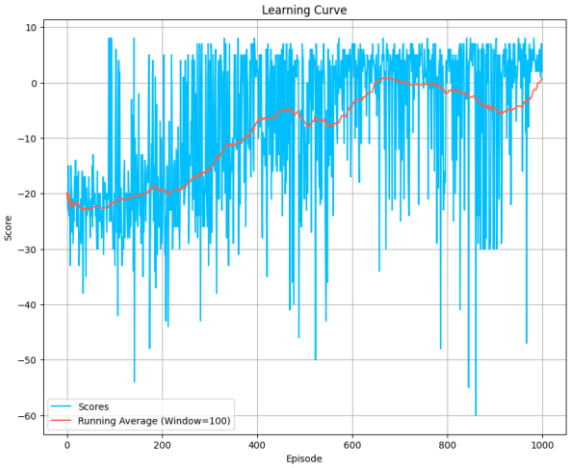
Actor Critic Empty:



DQN Key:



PPO Empty:



- for farther exploring of different hyperparameters you should check out the notebooks.

Asaf Mesilaty

Image preprocessing:

Image preprocessing, such as resizing the original image from (320, 320, 3) to a smaller scale, normalizing pixel values, and converting to grayscale, is crucial as it reduces the computational load, speeds up the learning process, and helps the neural network focus on relevant features instead of raw pixel values.

Since fully connected networks is the only network that worked for me I flatten the input. Since the network expects batches, I added another dimension.

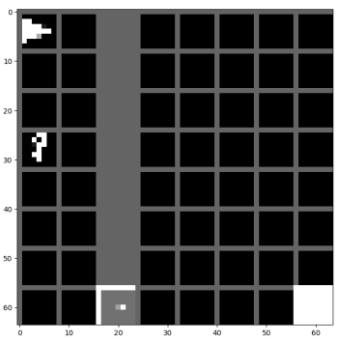
In the empty environment I also removed the lines to make the image less noisy.

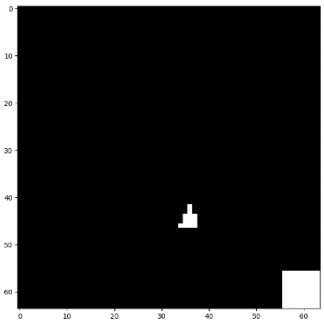
You remember from the begging how the two environments look.

Here is how it looks after preprocessing:

```python
def crop(state):
  return state[32:-32, 32:-32]

vrs_func = np.vectorize(lambda t: 1 if (t == 100) else t)
def remove_lines(state):
  return vrs_func(state)

def down_sample(state, n):
  # Down sample by factor n- down samlpe the 32x32 block into (32/n)x(32/n)
  return state[::n, ::n]

def gray_scale(state):
  # return np.average(obs, axis=2)
  return np.max(state, axis=2)
```

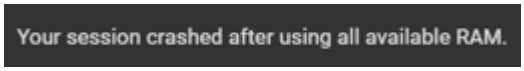
```python
def prepro(state):
  state = crop(state)
  state = down_sample(state, 4)
  state = gray_scale(state)
  state = state.flatten() / 255.0
  state = np.expand_dims(state, 0)
  return state
```





Those images size is (64, 64) making the new size 75 times smaller than the original size.

Another important thing to notice is that if you are using a replay buffer, without preprocessing your RAM memory might not be enough.



Asaf Mesilaty

Different considerations for the two environments

The empty environment was more forgiving since even if I prioritized move forward action- it was easy for the agent to get to the goal fast (an agent that always moves forward and turn to the same side when hitting a wall will always win in the empty environment).

The Key environment was way less forgiving and when I gave positive reward for the move forward action- the agent just made circles in the left side of the grid, without opening the door.

The main idea here was to understand that the agent should learn alone what actions should be prioritized alone. From that came the idea to give zero reward for moving forward- that way the agent can explore the state space without losing or earning any reward, which will not harm his learning.

Another thing that was crucial is the combination of penalty and exploration (choosing the right epsilons). I found that if the penalty was quite high (let's say -10) for redundant moves like toggle nothing, then by using the epsilon greedy method with high epsilon for an extended period, the agent avoids making the toggle action. The only option for the agent to choose toggle is when the action is random thanks to the epsilon greedy method. Even if it does manage to do so, the learning will be slower since there is more to compensate for.

The Key environment was the hardest to solve, and in the notebook, you can see the experiences of two different reward systems.

In conclusion, when comparing Tabular Q-Learning and Deep Q-Learning within the MiniGrid context, several distinctions are evident:

Reward System: Tabular Q-Learning allows precise adjustments for specific (state, action) pairs, enabling targeted incentives or penalties. In contrast, Deep Q-Learning's reward updates impact multiple states, necessitating subtler reward structures but offering a more generalizable approach.

Convergence and Scalability: Tabular Q-Learning, while quick to converge in simple scenarios, lacks scalability for complex environments like MiniGrid due to excessive memory requirements. Deep Q-Learning, though potentially slower to converge, handles larger state spaces efficiently without needing memory proportional to state-action pairs.

Handling Novel States: Tabular Q-Learning struggles with novel states, acting randomly when outside its direct experience. Deep Q-Learning, powered by neural networks, generalizes better, offering reasonable action predictions for unseen states based on learned patterns.

Overall, while Tabular Q-Learning has its merits in smaller, well-defined environments, Deep Q-Learning's robustness and scalability make it more suitable for the diverse and dynamic challenges presented by MiniGrid. This project's exploration and results underline the importance of choosing the right approach based on the specific requirements and complexities of the task at hand.

The end.