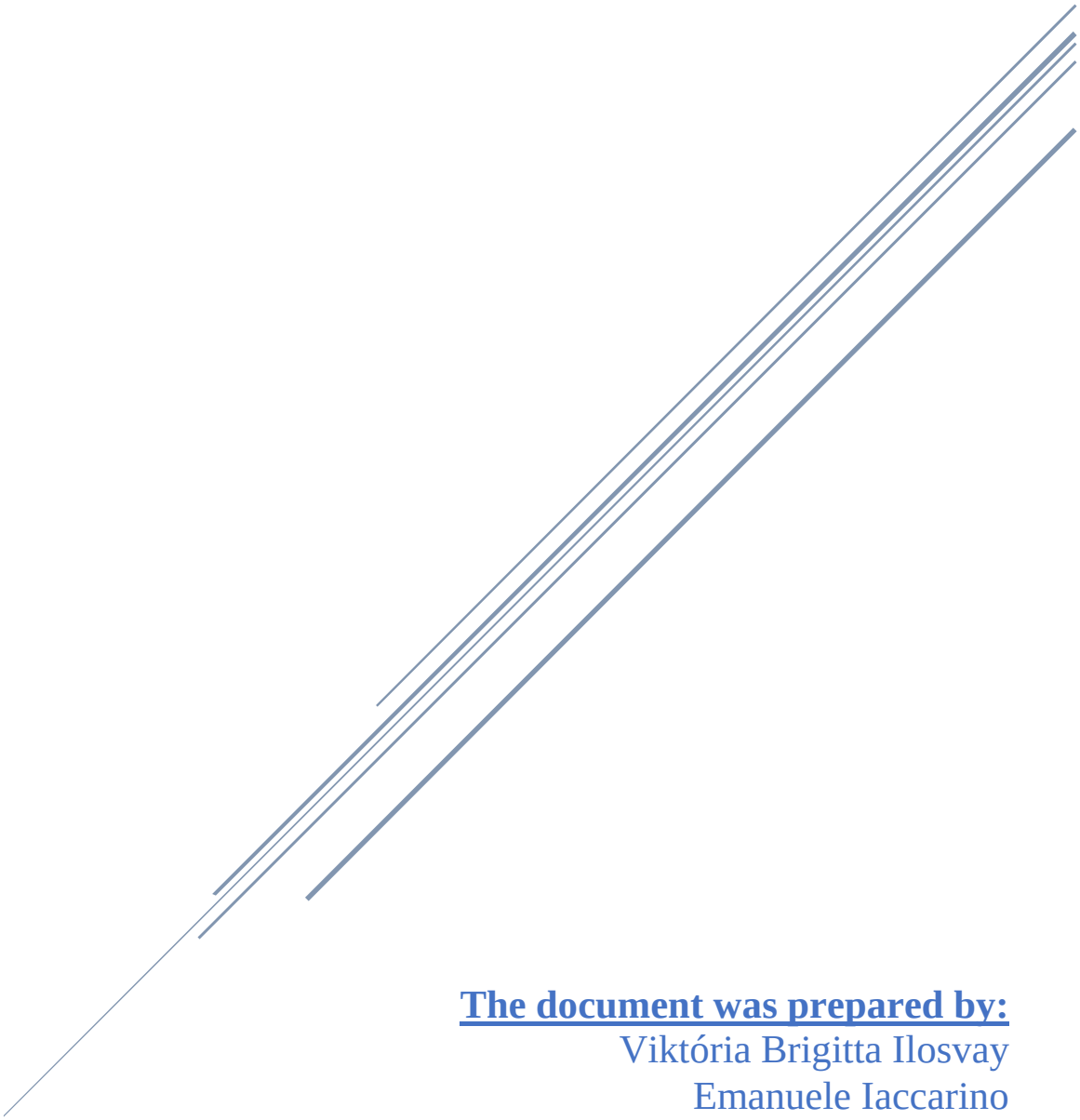
Activity 1 – RL

Distinctions among different algorithm parameters and algorithms



The document was prepared by:
Viktória Brigitta Ilosvay
Emanuele Iaccarino

Deusto
Universidad de Deusto

Table of contents

Short introduction to Reinforcement Learning 2
Deterministic vs Stochastic environments 2
Model-based vs Model-free methods 2
Markov Decision Process - MDP 3
Exploration vs exploitation 4
Type of the policies 4
Sparse vs Dense rewards 5
High vs Low learning rate 5
Frozen Lake environment 6
Training and Evaluation 7
Epsilon Function 8
Monte Carlo algorithm 9
Sarsa algorithm 10
Expected Sarsa algorithm 12
Q-learning algorithm 13
Double Q-learning algorithm 15
Some correlations and observations between the algorithms 16
The suboptimal performance of the SARSA algorithm in comparison to Q-Learning and Expected SARSA can be attributed to several factors: 16
Similarities of Expected Sarsa and Q-Learning 16
Hyperparameter Optimization of Q-Learning using Optuna 17
Hyperparameter Optimization 17
Methodology 17
Results 17
Optimization history plot 18
Contour plot 19
Slice plot 20
Hyperparameter importance 21
Conclusion 21
How does are model perform on a deterministic environment? (is_slippery=”False”) 22
Conclusion 24
Discussion 24
Additional resources used apart from the course material 25

Deusto
Universidad de Deusto

Short introduction to Reinforcement Learning

During Reinforcement Learning our goal is to teach our agent how to behave and make decisions in an environment based on its experience. At the end of the learning process the agent must achieve its goal as best as possible.

The main advantage of this machine learning paradigm over the other two (Supervised and Unsupervised Learning) is that it allows the agent to adapt easily to the changing environment, as it can update its knowledge of the environment based on its experience.

Using a policy allows us to map between states and actions. The policy is the essence of a reinforcement learning agent in the sense that it is sufficient to determine behavior by itself.

Deterministic vs Stochastic environments

In our world we can find both deterministic and stochastic environments. The main difference between these environments is whether the same thing happens after an action or not.

In a deterministic environment if the agent does the exact same action several times, then every time the exact same thing will happen. It will always end up in the same next state, so this type of environment is ignoring uncertainty. We have a deterministic environment for example in chess, in tic-tac-toe and in go.

Meanwhile when an agent does the exact same action several times in a stochastic environment then the agent moves to the next state based on probability distribution. In this case randomness is involved, so in the same state performing the same action won't always lead the agent to the same next state. Such environments are dice rolling, racing, card games or even self-driving cars.

Model-based vs Model-free methods

We call a model a representation of the environment. Based on having previous knowledge of the environment the agent can have a model or not.

When a model is available for the agent, then it can use **model-based methods**. In this case based on the known model the agent can predict the next state using the information of its current state and the performable actions, so the learning will be guided by the model- For example, teaching a robot in an environment and providing it with information about gravity.

When there is no available model, then the agent starts with a blank page and must learn via experiments (trial-and-error). This method is called **model-free method**. In this case the learning is not guided by the model, so the agent can be more "creative". The agent can adapt also to other environments, so it is a big advantage against the model-based methods. It is popular because we don't have to create a model. It's very easy to use the algorithm, so we can speed up the training. It is also possible to create a model of the environment for a neural network simulating the environment. Once this newly created model has been created, it can be used for learning.

Sometimes when we are starting with no information regarding the model, then with a model-free method the agent can create a model for itself and after that it can use this newly created model for its decisions.

Markov Decision Process - MDP

Markov Decision Process is the mathematical formulation of a decision process. With this process we create a decision-making model. With the Markov Property the agent can choose the future state depending on its current state. MDP has 4 main tuples: set of states, set of actions, probability of transmission and immediate reward after transmission. With MDP we want to maximize the final cumulative reward.

If the task is not based on episodes, it is called a continuous task. In this case the time is infinite, so the agent must optimize for it. At this point a **Discount Rate** ($0 \leq \gamma \leq 1$) is required. However even in episodic tasks we often use it to reduce the weight of future rewards the further away they are. This concept is since it is better to get something today than tomorrow. The discount rate represents the importance of future rewards. A **high discount rate** places significant importance on future rewards, while a **low discount rate** places less importance on future rewards.

Exploration vs exploitation

Many Reinforcement Learning methods are ε-based to promote at least initially the exploration. When we know nothing about the environment, we must make some explorations before exploitation. When we know more about the environment the focus will be on exploitation instead of the explorations.

Type of the policies

As we discussed earlier the environments can be also deterministic and stochastic ones. It is no different for policies. When a policy is a **deterministic** one, then when the agent is staying in the same state, the same action is always carried out. Besides when the policy is **stochastic**, then when the agent is staying in the same state, there is a distribution of probabilities about the action to be taken.

The **on-policy** means that during time the policy is being updated. It may happen that it does not promote sufficient exploration.

The **off-policy** stands for if a different policy (behavior policy) is used to explore than the one being updated (target policy).

When we want to always select the best action, then the **greedy/max policy** is required. Using this, the agent will end up with the best possible action based on its knowledge. However, if we always act greedily, there will be many state-action pairs that may never be visited. In the case of these states, we will have nothing to average, and these state values will be 0. If we want to carry out exploitation, then this policy is the best for it.

If we use the **random policy**, then the agent will select a random action. This is the best for exploration.

The **soft policy** is used when all actions have some probability of being selected.

We use the **ε-greedy policy** ($0 \leq \varepsilon \leq 1$) to balance exploration and exploitation in reinforcement learning. Using this policy, the agent chooses the best action with probability (1-ε). Any action (including the best one) will be chosen with probability ε (each action with probability ε / number of actions). In this case, if the value of the ε is 0, we will use the greedy/max policy and if the value of the ε is 1, we will use the random policy. So, giving a lower ε will result in less randomness, but the randomness is necessarily, because it solves the problem of exploration mentioned above under the greedy/max policy. Furthermore, it can be an on-policy, so in this case we can improve it by acting greedy, so by taking the action that has a maximum q-value for a state. Initial policy is usually some randomly initialized values so to make sure we explore more, and this can be ensured if we keep the value of ε around 1. In the later time we desire a more deterministic policy, this can be ensured by setting ε near to 0.

Sparse vs Dense rewards

The **spare reward** is given punctually. For example, only when the agent successfully reaches the final objective.

The **dense rewards** are given more frequently, for intermediate objectives. They act like a gradient.

High vs Low learning rate

The **learning rate** affects how quickly the agent update its knowledge. When the value is **high**, the agent updates its knowledge very frequently, however due to it, it might overshoot the optimal solution. If we choose a **low** value for the learning rate, then the learning will be more stable, however it can result that for finding the best solution the agent will need way more time.

Finding the right value can be very tricky. The optimal learning swiftly reaches the minimum point.

Deusto
Universidad de Deusto

Frozen Lake environment

For our experiences we are using an 8x8 Frozen Lake environment. In this environment 4 actions are possible: going LEFT (0), going DOWN (1), going RIGHT (2) and going UP (3).

This environment can be deterministic and stochastic based on the value of the is_slippery parameter. During our experiments we are trying out both. When it comes to a stochastic environment, then based on the properties of the environment the agent will move in the chosen direction ⅓ of the time, and in each of the perpendiculars another ⅓.

This environment is enriched with a dense reward structure. During the experience the agent can reach the goal, drop into a hole, and not reach the goal, but also not drop into a hole. It assigns a penalty of -1 if the agent falls into a hole and a reward of +1 if it successfully reaches the goal.

The environment, as mentioned above, consists of an 8x8 grid, and the agent's task is to navigate from a designated initial state to the target state while avoiding treacherous holes.

To rigorously evaluate the performance of the algorithms, we have carefully selected the following set of reinforcement learning algorithms: Monte Carlo, Sarsa, Expected Sarsa, Q-learning and Double Q-Learning. In the following, these algorithms and results will be discussed.

Training and Evaluation

The training and evaluation procedures are designed to ensure reliable performance comparisons. Here's an outline of our approach:

Custom Epsilon Selection: We utilize a custom epsilon, a critical parameter for balancing exploration and exploitation during training.

Training Duration: The training phase spans a specific number of steps, set at 3,000,000 steps, to ensure ample learning time.

Max Step Limit: To prevent infinite loops and facilitate consistent comparisons, we introduce a max step limit. This limit is calculated as the product of the number of actions, the state space, and a scaling factor of 2. For our 8x8 board, this results in a max step limit of 4 * 64 * 2 = 512.

TensorBoard Logging: We leverage the TensorBoard library to log essential training data and performance metrics, offering comprehensive insights into the learning process.

Evaluation with Max Policy: For evaluation, we employ a max policy action selection approach, which ensures that the best-performing actions are chosen.

Hyperparameter Tuning: To identify the optimal hyperparameters for each algorithm, we will initially employ fixed hyperparameters. After assessing performance, we will further fine-tune these parameters using the Optuna framework.

This approach guarantees that the algorithms are tested under the same conditions, providing a fair basis for performance evaluation.

Epsilon Function

In essence, our custom_epsilon function offers a dynamic and adaptable approach to exploration, allowing the agent to initially explore the environment extensively and then progressively shift its focus towards exploitation as it gains more experience.

Our custom epsilon function is defined as follows:

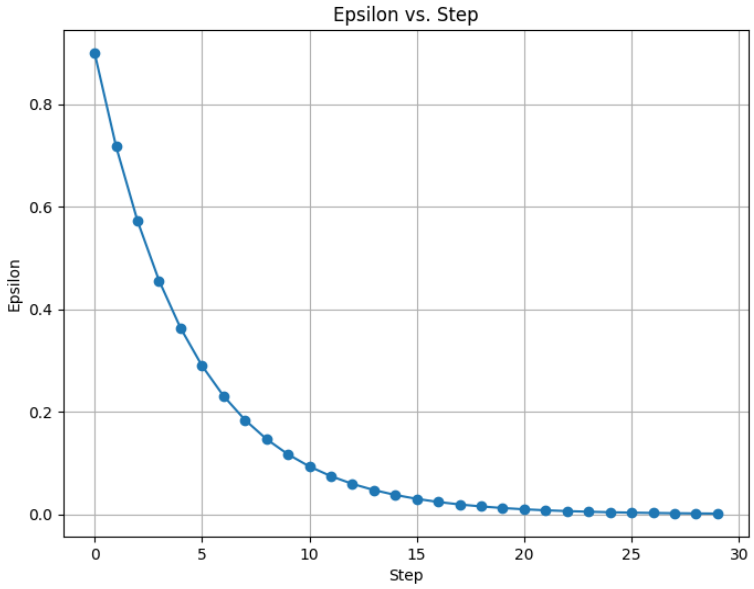
$\epsilon(t) = \epsilon\_initial * e^{(-decay\_factor * t)}$

Here's a breakdown of the key components. $\epsilon(t)$ represents the value of epsilon at a specific time step (t). We initiate the function with a relatively high exploration rate, denoted as ϵ_initial. Over time, this exploration rate gradually diminishes as training progresses.

The rate of decay is determined by the decay factor, calculated as:

decay factor = -ln(єinitial / єfinal) / total steps

The process of decay is exponential, ensuring a smooth transition from exploration to exploitation throughout the training process.



Monte Carlo algorithm

The Monte Carlo algorithm is a model-free algorithm, so we use it when there is no prior information regarding the environment and all information is essentially gathered from experience. With this reinforcement learning algorithm the agent learns directly from episodes, so through sequences of states, actions, and rewards. When an agent is in a state, it performs an action and receives a reward based on that action. After that, the agent moves to the next state. From this state the same will happen to the agent.

Monte Carlo learns by sampling the rewards from the environment and averaging the resulting rewards. In each episode, our agent acts and receives a reward. When each episode (experience) is completed, we take the average.

In a deterministic environment we can choose the Monte Carlo algorithm for learning, however it is generally not the best idea. As it is mentioned above, the agent always gets the same result for the same actions. The Monte Carlo algorithm requires running several episodes to estimate the values. In a deterministic environment, other algorithms may converge to the correct values faster. For example, dynamic programming or Q-Learning is more efficient at calculating exact values.

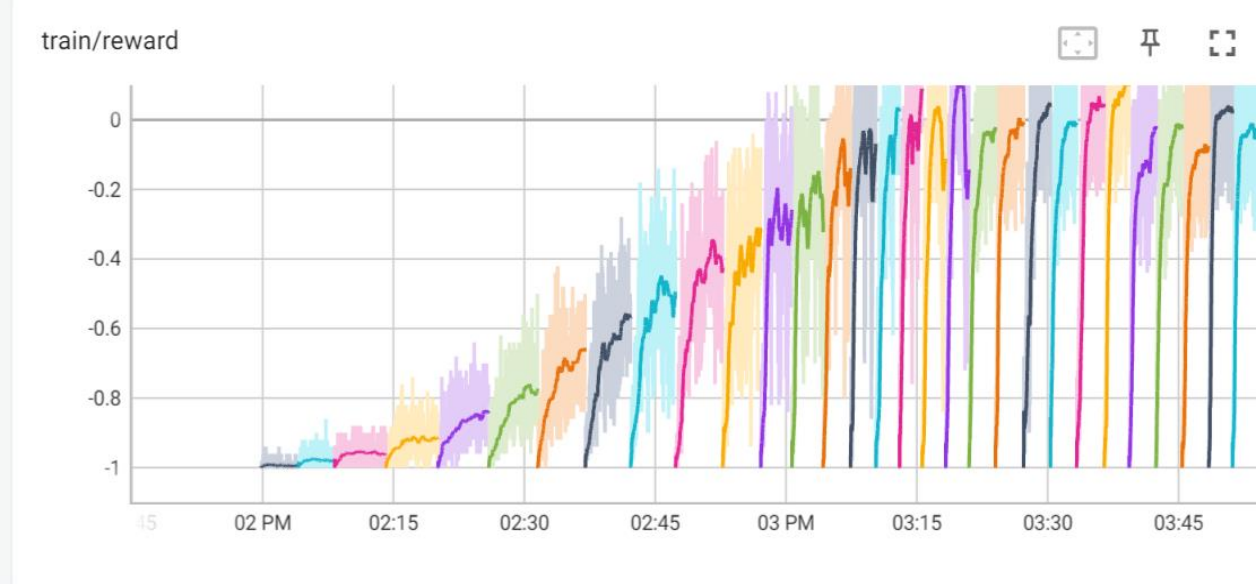
The Monte Carlo algorithm can also be used in a stochastic environment. In such an environment it is very useful that this algorithm learns efficiently from experience, so it can estimate the rewards very well. Since the Monte Carlo method is based on experience, several trials are needed to calculate reliable values, and each run may give different results due to random factors.

Sarsa algorithm

Sarsa stands for State, Action, Reward, State' and Action'. Sarsa is a temporal difference learning algorithm. The biggest advantage over the Monte Carlo algorithm is that if we want to update the estimate of state values we don't have to wait until the end of an episode. It is an on-policy temporal difference (TD) method for control, which means it updates the values of a state based on the estimation of the next state without having to wait for the real value. It uses the same policy for sampling its experiences and after for optimizing it.

Sarsa is an on-policy algorithm as it is updated based on the current choices of our policy, so the agent learns the value function and the policy based on the actions that are taken by itself.

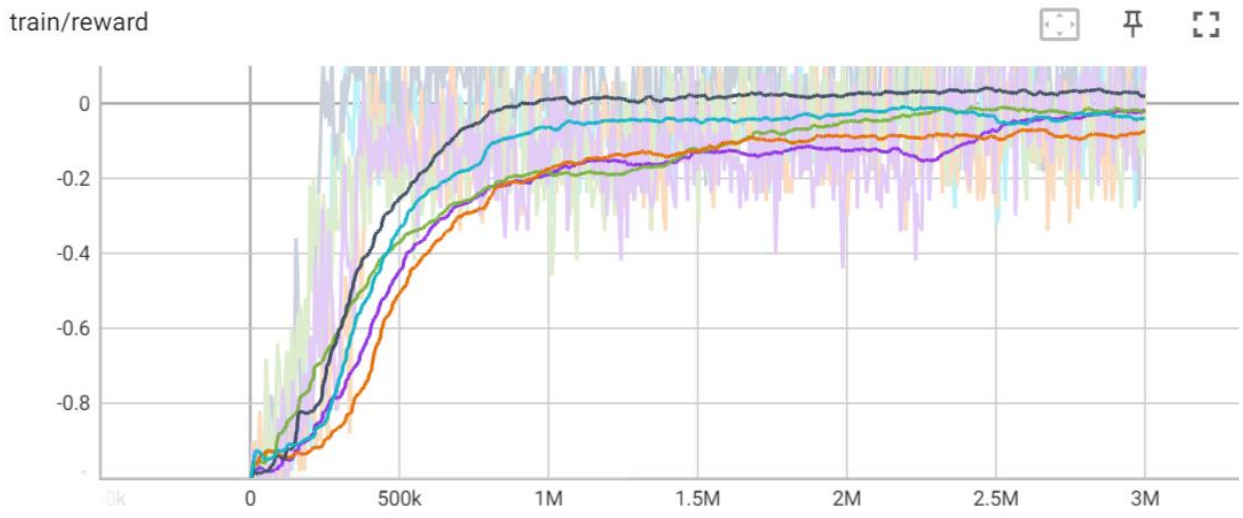
We conducted a comprehensive analysis of the Sarsa algorithm's performance over time. We can see the results of these experiments through TensorBoard and analyze how our custom epsilon influenced the agent's behavior. The results from these experiments are depicted in the TensorBoard log, allowing for a detailed examination of how the agent's behavior evolves during training.



As demonstrated in the performance analysis over time, our algorithm exhibits a notable transition from exploration to exploitation, resulting in improved results. The x-axis represents the training time, where the algorithm progresses from higher epsilon values to lower ones.

This transition signifies a critical aspect of reinforcement learning. At the outset of training, with higher epsilon values, the agent actively explores various actions, effectively searching the environment for optimal strategies. During this phase, there may be episodes of random or suboptimal actions, leading to initial fluctuations in performance.

However, as training continues, the epsilon value gradually decreases, guiding the algorithm toward a more exploitation-focused approach. The agent increasingly relies on its learned policies and tends to choose actions with higher expected rewards. Consequently, the results stabilize and tend to converge towards more desirable outcomes.



Directing our attention to the final iteration of our algorithm, characterized by a lower epsilon value, we observe a distinct pattern in the training process. After approximately one to two million steps, a noteworthy phenomenon unfolds as the average reward stabilizes. This stabilization signals a critical transition in the algorithm's behavior.

However, it is essential to acknowledge the substantial breadth of the confidence interval surrounding the average reward. This variability in performance is primarily attributed to the inherent nature of the environment in which the agent operates. The presence of environmental 'slipperiness' introduces unpredictability and randomness into the agent's interactions, resulting in a wider range of outcomes.

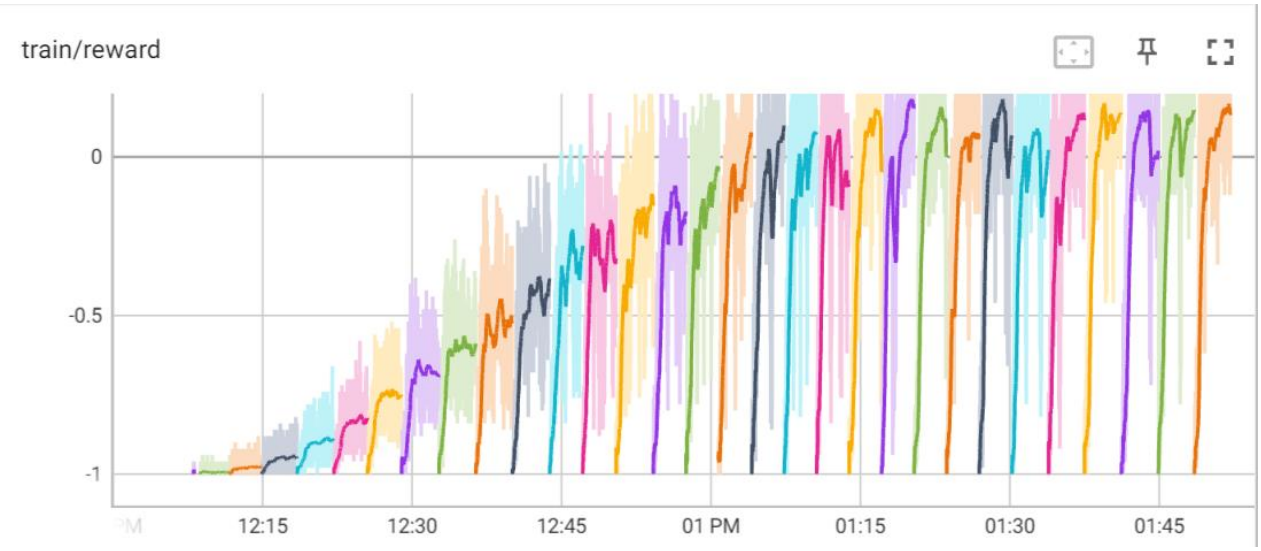
In conclusion, while our implementation of the Sarsa algorithm provides valuable insights and learning capabilities, it is important to note that, in the context of our environment, its overall performance may not be optimal. The observation of a relatively low average reward in the final stages of training suggests that alternative algorithms or approaches may warrant exploration to achieve more favorable results.

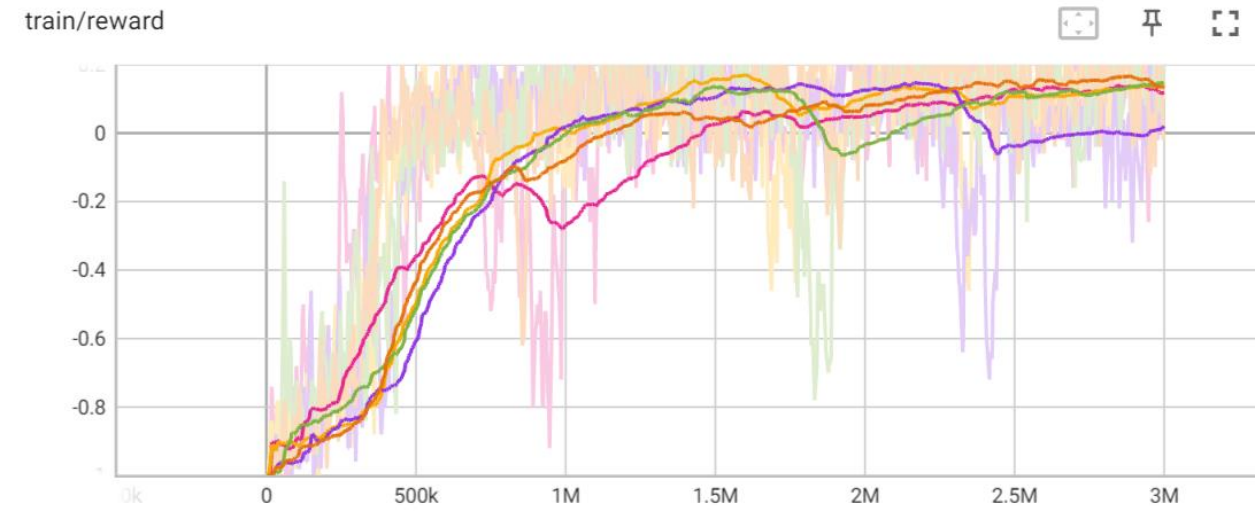
Expected Sarsa algorithm

Expected Sarsa is very similar to Sarsa, but instead of stochastically sampling the state-action values using our current policy, it calculates the expected value of all future state-action pairs, thus considering the probability of each action according to the current policy.

Both Sarsa and Expected Sarsa can be used as on-policy methods for verification, however the Expected Sarsa is computationally more complex, so way slower than Sarsa. Expected Sarsa also eliminates the variance introduced by Sarsa due to the stochastic selection of subsequent actions, however on the other hand it generally outperforms Sarsa when the same amount of experience is obtained, since it calculates the weighted sum of actions instead of noisy sampling. To sum up, Sarsa is computationally more efficient, while Expected Sarsa performs better with less experience.

In a similar manner to the Sarsa algorithm, we conducted a comprehensive analysis of the Expected Sarsa algorithm's performance over time. The results from these experiments are depicted in the TensorBoard log, allowing for a detailed examination of how the agent's behavior evolves during training.





Q-learning algorithm

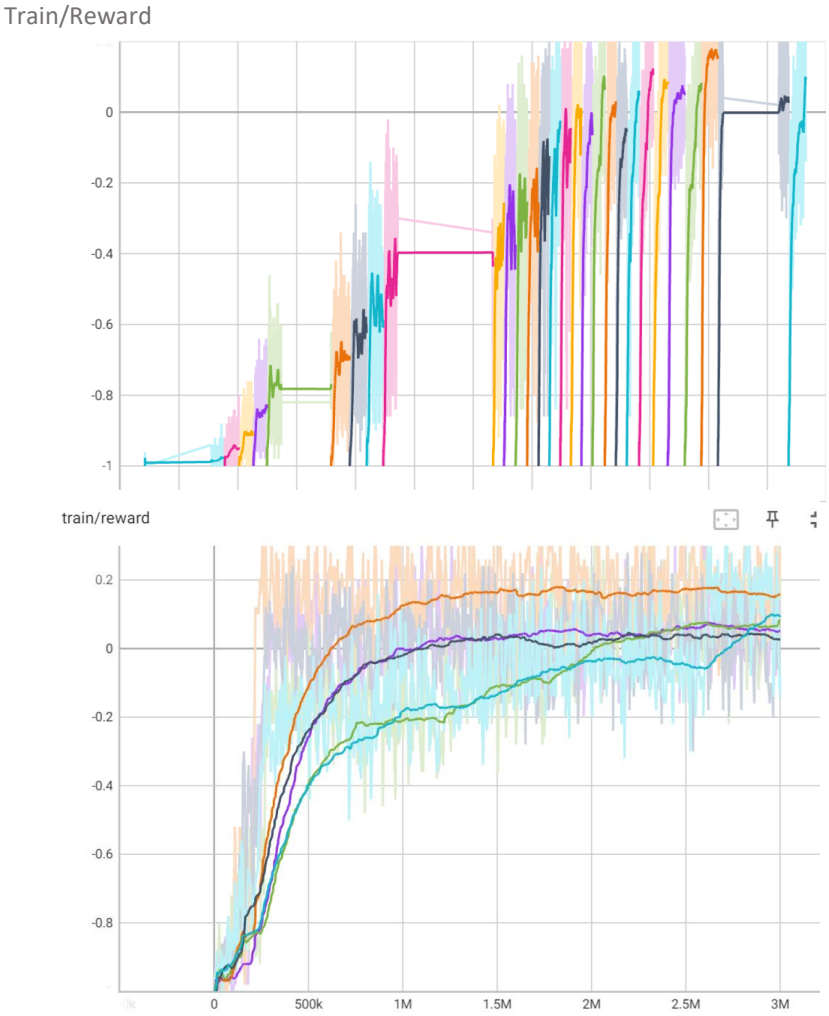
In this case, the learned state-action function Q directly approximates Q*, the optimal function, regardless of the policy that is followed.

Q-learning has a different update rule than SARSA. While Sarsa is an on-policy algorithm, Q-Learning is an off-policy one. This means we are updating differently in behavior from the policy

we use to explore the world. For action we usually use the ε-greedy policy and for evaluation we usually use greedy/max policy.

In an environment where we give a very high penalty for something, the Q-Learning algorithm takes more risk because it chooses the highest Q-value, so the agent won't be so bothered to get the penalty. Whereas if we use the previously mentioned Sarsa, which has learned that there is a high probability of stepping off the cliff and getting a high negative reward, it doesn't take that much risk.

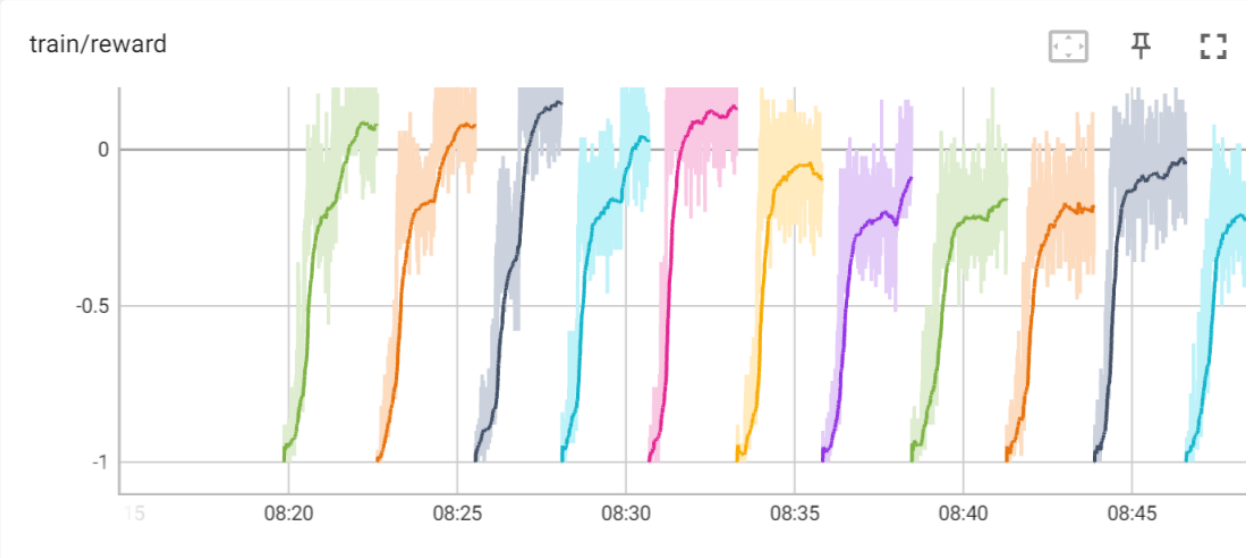
The code presented below is an implementation of the Q-Learning algorithm for reinforcement learning. We can see the results of these experiments through TensorBoard and analyze how our custom epsilon influenced the agent's behavior.



Double Q-learning algorithm

The Double Q-Learning algorithm is a further development of Q-Learning, which uses two different Q-tables for learning, thus preventing overestimation of Q-table values. 2 Q-tables are needed in this algorithm using one to select the action and the other to evaluate it.

Using the same pattern, we conducted a comprehensive analysis of the Double Q-Learning algorithm's performance over time. The results from these experiments are depicted in the TensorBoard log, allowing for a detailed examination of how the agent's behavior evolves during training.



The Frozen Lake environment is a relatively simple environment, so it is possible that Q-Learning can learn the optimal policy without the need for the additional complexity of Double Q-Learning.

Double Q-Learning can be more sensitive to the exploration-exploitation trade-off. It is more sensitive to the choice of hyperparameters.

Some correlations and observations between the algorithms

The suboptimal performance of the SARSA algorithm in comparison to Q-Learning and Expected SARSA can be attributed to several factors:

On-Policy Learning: SARSA is an on-policy reinforcement learning algorithm, which means it updates its Q-values based on the policy it follows during exploration. This can be a double-edged sword. On one hand, on-policy methods are safer as they directly evaluate the policy they intend to follow. However, it can also be a limitation because SARSA may get stuck following suboptimal policies.

If the initial policy or exploration strategy is not effective, SARSA may fail to escape these suboptimal paths, resulting in suboptimal performance.

Sample Variability: SARSA updates its Q-values based on the actual actions taken during exploration. This introduces more variability in the updates compared to Expected SARSA, which uses expected values. Higher variability can lead to slower convergence and potentially suboptimal results. SARSA's updates are influenced by the specific actions taken, making it more sensitive to the noise or randomness in the environment.

Similarities of Expected Sarsa and Q-Learning

Expected Sarsa algorithm reveals similar characteristics to Q-Learning, with a notable transition from exploration to exploitation during training. The observation of a relatively low average reward in the final stages of training suggests that the inherent environmental factors, such as slipperiness, contribute to the variability in performance. This, in turn, raises the possibility that alternative algorithms or approaches may yield more favorable results in the context of our environment. Further exploration and experimentation are warranted to refine the performance of reinforcement learning agents in such complex and dynamic settings.

Deusto
Universidad de Deusto

Hyperparameter Optimization of Q-Learning using Optuna

Hyperparameter Optimization

Hyperparameter optimization is the process of finding the best hyperparameter values for a machine learning algorithm. Optuna is a hyperparameter optimization library that uses Bayesian optimization to find the best hyperparameter values.

Methodology

We use Optuna to optimize the discount rate and learning rate of Q-Learning for the FrozenLake environment. We use a TPESampler as the sampler for Optuna. We run 50 trials of Optuna.

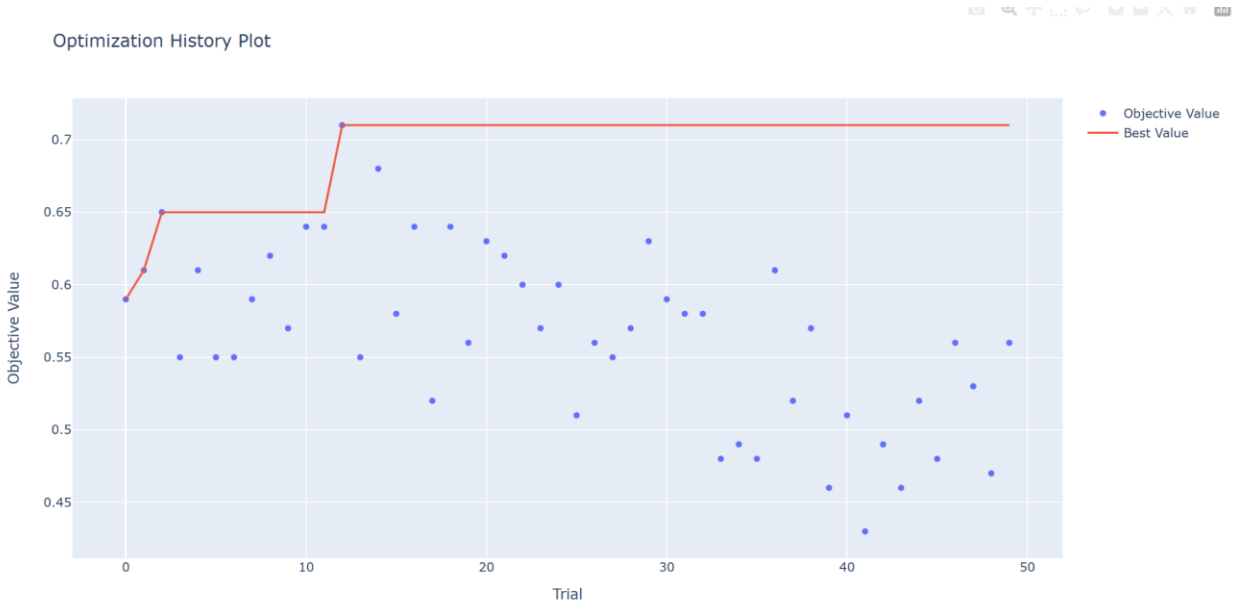
Results

The best trial achieved a reward of 0.71 on the FrozenLake environment. This is better than the performance of the default Q-Learning hyperparameters.

Best trial: Value: 0.71,
Params:{"discount_rate": 0.9961459723114788, "lr": 0.06698053340104898}

We'll use the plot provided by Optuna to have a full understanding of our results and how are model perform:

Optimization history plot



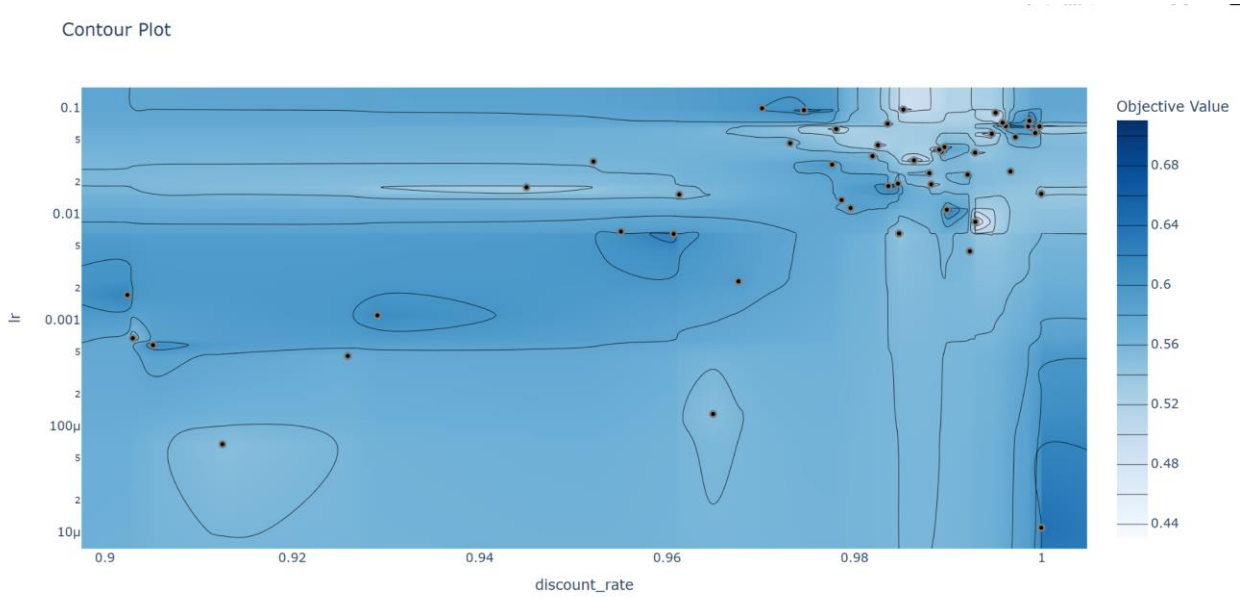
The optimization history plot shows how the objective function changes over the course of the optimization process.

This plot can be used to:

- Track the progress of the optimization process and identify the best hyperparameter values.
- Explain to the readers how the optimization process works and how it converges to the best hyperparameter values.

Here we have to remember that each of our trial provide different parameters for Learning Rate and Discount Factor, we used the most optimal one at the start and the worse performing one in the end. We just got unlucky, of course every time you run this code you will have different result.

Contour plot



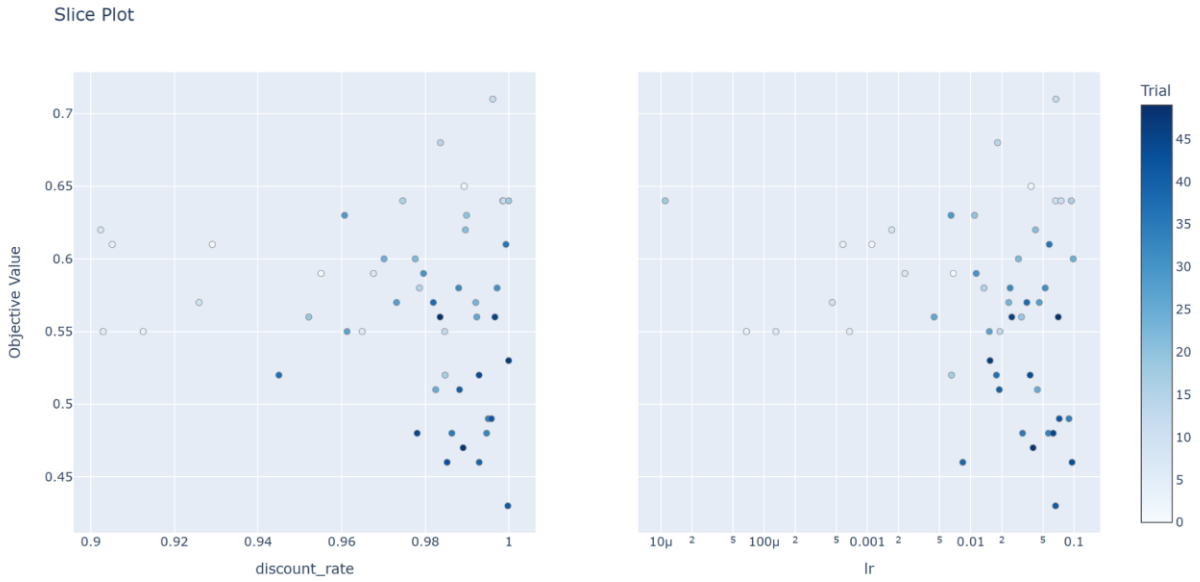
The contour plot shows the objective function as a surface over the hyperparameter space. It can be used to identify the region of hyperparameter space where the objective function is maximized.

This plot can be used to:

- Visualize the hyperparameter space and identify the region where the objective function is maximized.
- Motivate your choice of hyperparameter values for your final model, choosing values that are likely to lead to good performance.

Here the results are clearer, we find out the best combination of Discount Rate and Learning Rate through the Heat Map.

Slice plot

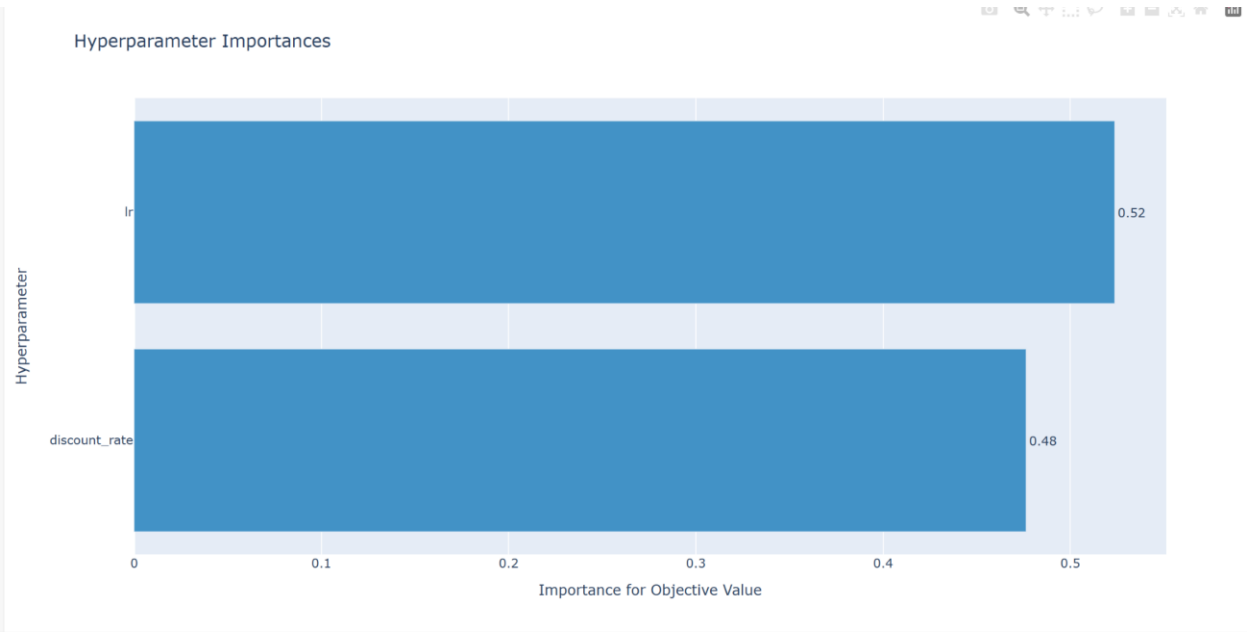


The slice plot shows how the objective function changes when two hyperparameters are varied, while keeping all other hyperparameters fixed. It can be used to identify interactions between hyperparameters.

This plot can be used to:

- Identify interactions between hyperparameters.
- Explain to the readers how the hyperparameters interact with each other to affect the performance of your model.
- Motivate your choice of hyperparameter values for your final model, taking into account interactions between hyperparameters.

Hyperparameter importance



The hyperparameter importance plot shows the relative importance of each hyperparameter in the optimization process. It is calculated by measuring how much the objective function changes when each hyperparameter is varied. The more important a hyperparameter is, the more the objective function will change when it is varied.

This plot can be used in your paper to:

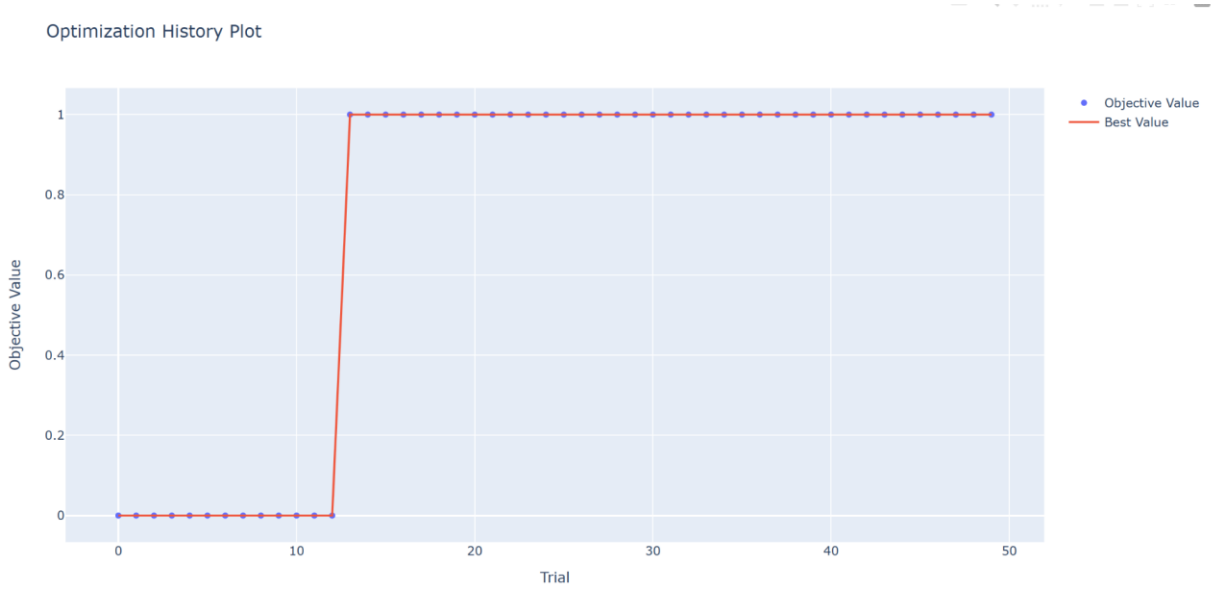
- Identify the hyperparameters that have the biggest impact on the performance of your model.
- Explain to the readers how the hyperparameters affect the performance of your model.
- Motivate the choice of hyperparameter values for your final model.

More or less both the hyperparameters have the same importance, Learning Rate is slightly more important.

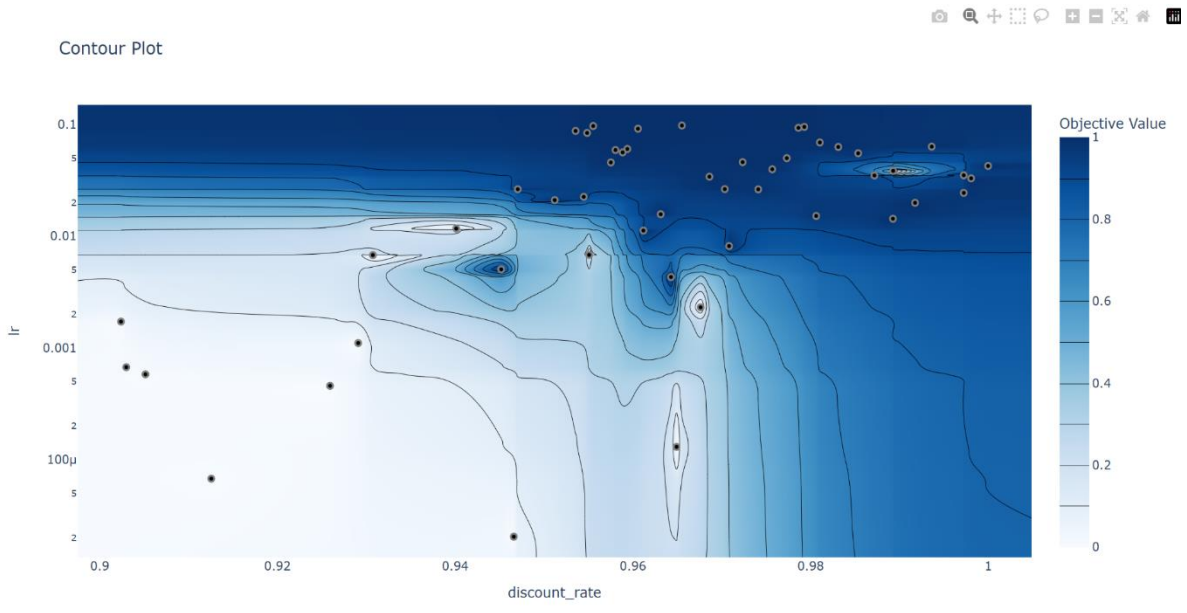
Conclusion

We have shown that Optuna can be used to optimize the hyperparameters of Q-Learning for the FrozenLake environment. We found that Optuna was able to find hyperparameters that led to improved performance on the FrozenLake environment.

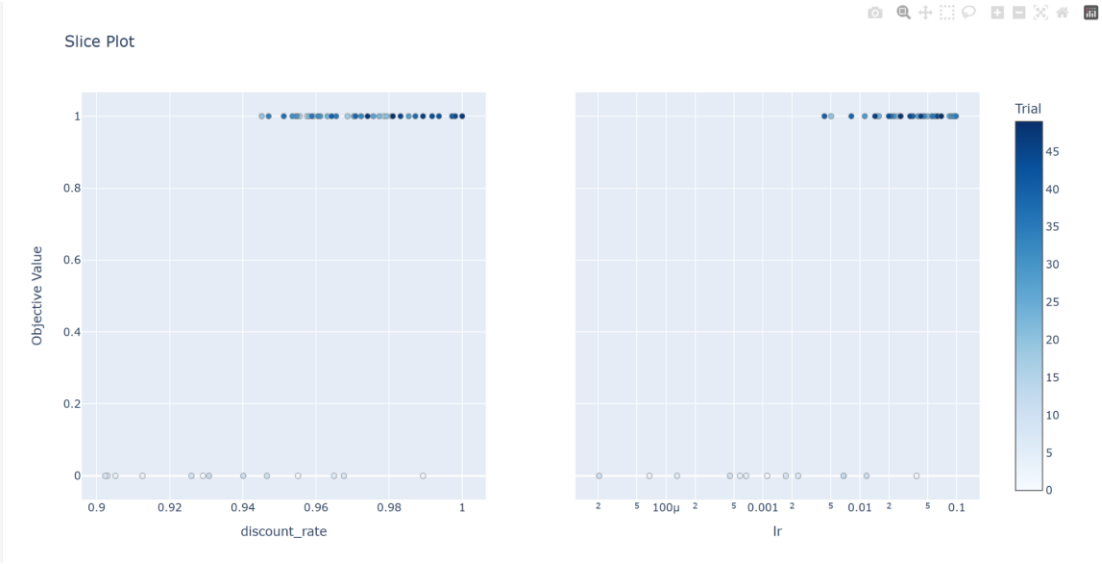
How does are model perform on a deterministic environment? (is_slippery=”False”)



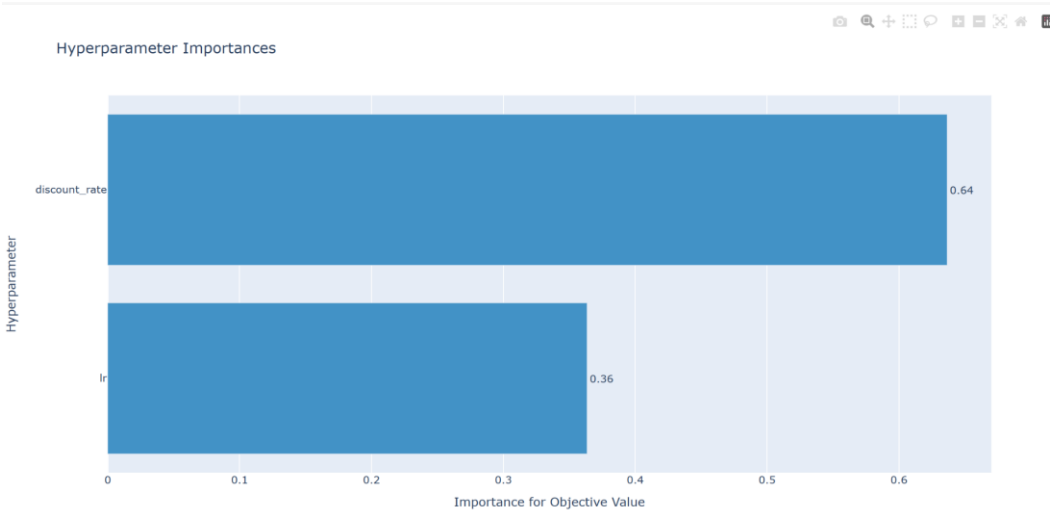
The optimization history plot shows how the average reward changed over the course of the optimization process. The plot shows that the optimization process converged to good hyperparameter values after approximately 12 trials.



The contour plot shows the average reward as a surface over the hyperparameter space. The region of hyperparameter space where the average reward is maximized is shown in dark blue. We had bad performance only on the firsts trials were the Discount_rate and Learning_rate was low.



The slice plot shows how the average reward changes when two hyperparameters are varied, while keeping all other hyperparameters fixed. The plot shows that having high values of discount rate and learning rate have a significant interaction on the performance of the Q-Learning algorithm.



The hyperparameter importance plot shows the relative importance of each hyperparameter in the optimization process. The discount rate is the most important hyperparameter, followed by the learning rate.

Conclusion

our findings revealed a clear trend of convergence to optimal hyperparameter values after an initial period of less favorable performance. The results from this deterministic environment emphasize the robustness and adaptability of the optimized hyperparameters and underscore the importance of understanding the context in which the model will be deployed.

In conclusion, this study not only demonstrates the potential of Optuna as a valuable asset for hyperparameter optimization in reinforcement learning but also highlights the importance of tuning hyperparameters to achieve optimal model performance.

Discussion

In this study, we delved into various reinforcement learning algorithms, each offering unique advantages and insights into the dynamic process of learning from experience. We've discussed the behaviors and characteristics of Monte Carlo, Sarsa, Expected Sarsa, Q-Learning, and Double Q-Learning in the context of the challenging FrozenLake environment.

Our exploration extends to the realm of hyperparameter optimization with Optuna. The results obtained with Optuna showcased its potential to fine-tune Q-Learning for FrozenLake, highlighting the significance of selecting optimal hyperparameters in reinforcement learning.

Our comprehensive analysis and experimentation contribute to the understanding of reinforcement learning algorithms and the crucial role of hyperparameter optimization. However, it's important to recognize the complexities of the environment and the potential for alternative algorithms to yield more favorable results.

In conclusion, our study not only demonstrates the potential of Optuna in hyperparameter optimization for reinforcement learning but also emphasizes the importance of adaptability and exploration in complex environments.

Additional resources used apart from the course material

Baeldung. (2023, March 24th). Retrieved from https://www.baeldung.com/cs/q-learning-vs-sarsa

Soons, J. (2021, October 21st). Retrieved from https://jochemsoons.medium.com/a-comparison-between-sarsa-and-expected-sarsa-66b931202c75

Towards AI. (2022, April 29th). Retrieved from https://towardsai.net/p/l/reinforcement-learning-monte-carlo-learning

Vidhya, A. (2022, July 22nd). Retrieved from https://www.analyticsvidhya.com/blog/2018/11/reinforcement-learning-introduction-monte-carlo-learning-openai-gym/

The Optuna Documentation. Retrieved from https://optuna.readthedocs.io/en/latest/

Plot in Optuna. Retrieved from https://medium.com/optuna/visualizing-hyperparameters-in-optuna-86c224bd255f

Epsilon Decay Function. Retrieved from https://medium.com/mlearning-ai/a-deep-dive-into-reinforcement-learning-q-learning-and-deep-q-learning-on-a-10x10-frozenlake-c76d56810a46