INF-2600: Artificial Intelligence, AI - Methods and applications

Assignment 2

Amund Strøm

UiT id: ast283@uit.no

April 2, 2024

1 Task 1: 1D environment

1.1 Why do we need a policy net and a target net?

Both the policy net and the target net are independent neural networks that serve different purposes in a Deep-Q-Network (DQN) algorithm.

The **policy net** is the main network and acts as the decision maker of the algorithm. It takes the information about the environment as input and gives the best action to take as output. While the **target net** acts as a steady anchor and its job is to provide stable and reliable guidance to the policy network.

During the training process, the policy network is continuously updated to improve its performance based on the experiences gained from interacting with the environment. Meanwhile, the target network is updated less frequently and remains fixed until the next update. This fixed target network provides stable target Q-values for training the policy network, helping to stabilize and improve the learning process.

The target Q-value is like predicting how good a move might be. It's based on the reward you get for making said move now, plus how good future moves could be if you keep making smart moves.

While the target network is indeed updated less frequently compared to the policy network, this deliberate pace helps maintain stability in the learning process. Rapid changes in strategy can sometimes lead to instability or divergence in learning. By updating less frequently, the target network provides a consistent and stable reference point for the policy network to learn from. This helps mitigate exploration and improves exploitation.

1.2 When do you think a model converges?

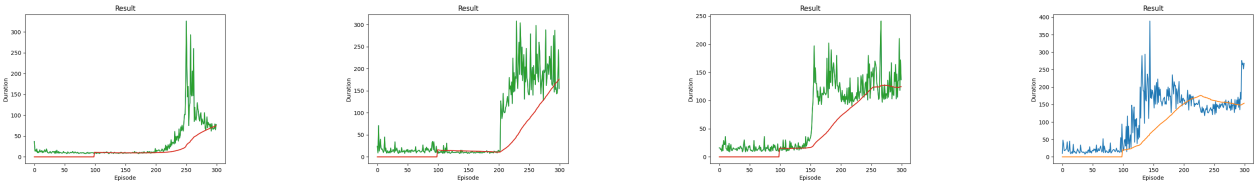
A model converges whenever it cannot be improved upon, meaning the performance of the model stabilizes. This indicates that the model has learned as much as it can from the available data.

1.3 Changing the exploration vs. exploitation

The balance between exploration and exploitation is dictated by an parameter commonly called epsilon. This parameter determines the probability of selecting a random action (exploration) versus selecting the action with the highest Q-value (exploitation)

During the training process, epsilon typically decays over time from an initial value towards a minimum value. This allows the agent to gradually shift its focus from exploration to exploitation as training progresses. Generally, when epsilon is high, the agent explores the environment more by taking random actions. As training progresses and epsilon decreases, the agent relies more on what it has learned to exploit its current knowledge and make decisions that maximize reward.

In the case of this implementation, a low epsilon decay value means a faster decay resulting in more exploitation. While a high epsilon decay value results in more exploration. As seen in the figures on the next page this will give different results and it seems that it is generally better to have the decay value somewhere in the middle.



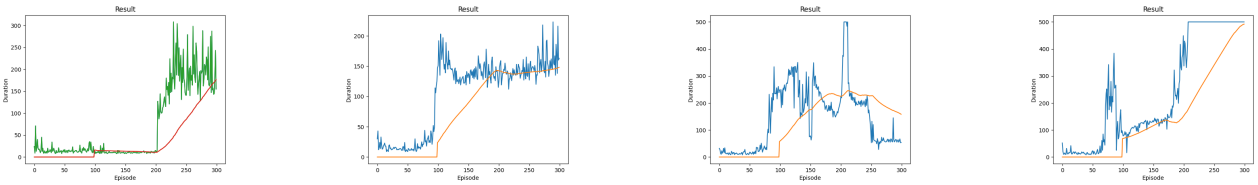
(a) Decay 250 (b) Decay 1000 (c) Decay 2500 (d) Decay 5000

Figure 1: Result of changing the epsilon decay

1.4 Varying the number of layers in the model

Changing the number of layers can have many different results such as increasing the complexity of the model, over-fitting on the available data, increase of training time, and difficulties to generalize data if they are not to complex.

Some of these results was observed during testing, especially the increase of training time. For comprising, 3 layers used about 230 seconds, while 6 layers used 1300 seconds. Over-fitting was also observed in the case of 5 layers, but interestingly this was not seen when testing 6 layers. My guess is that there are a lot of randomness in these sorts of cases and that results will change between each independent execution.

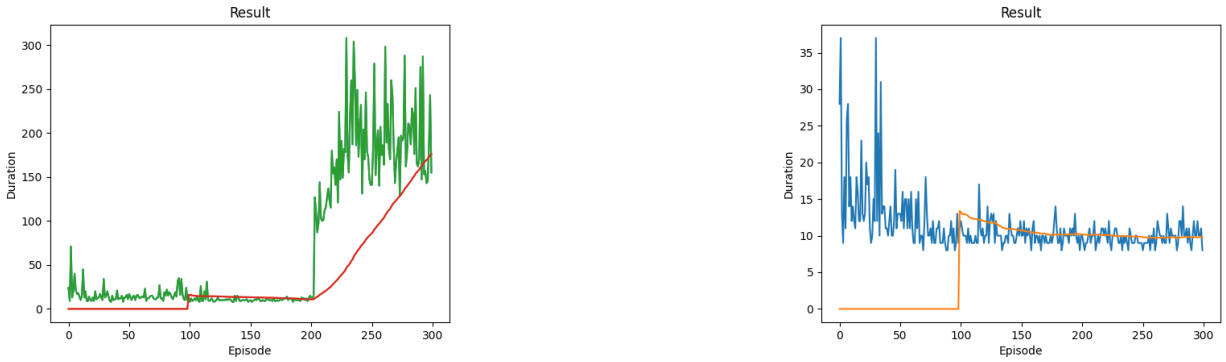


(a) 3 layers (b) 4 layers (c) 5 layers (d) 6 layers

Figure 2: Result of changing number of layers in the network

1.5 Implementation without the replay buffer

By removing the replay buffer we effectively removes the models ability to learn from past experiences, and the model would be limited to only learn from the most recent experience. This would of course destroy the performance of the model, which can be seen in the illustration.



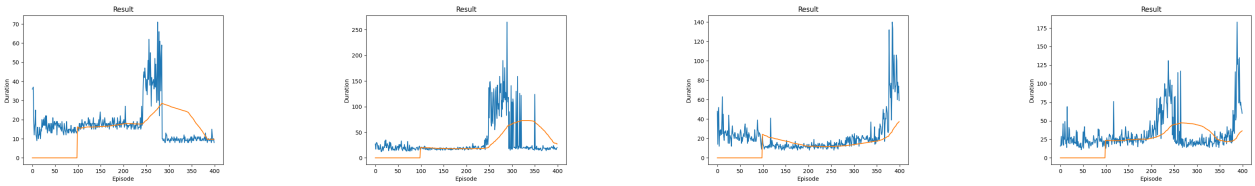
(a) With replay buffer (b) Without replay buffer

Figure 3: Comparison of implementation with and without replay buffer

2 Task 2: 2D environment

Much of what was explained in task 1 can be expected in task 2. However, we see some interesting changes in performance all over the graphs, and that is they generally preform worse over time. This is likely due to the environment being more complex and the model require more episode to start converging. Since I am running this on my laptop, I am not able to run the necessary amount of episodes required to make the model converge and the model therefore falls short.

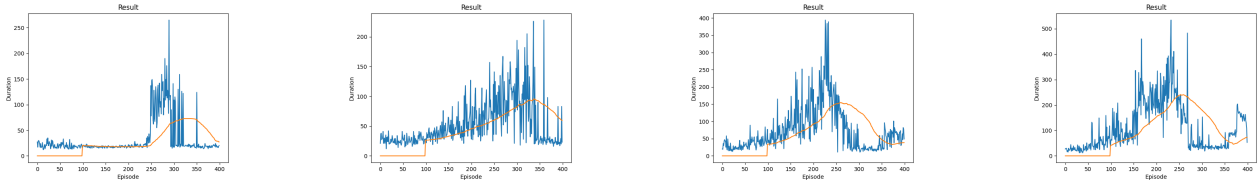
2.1 Changing the exploration vs. exploitation



(a) Decay 250 (b) Decay 1000 (c) Decay 2500 (d) Decay 5000

Figure 4: Result of changing the epsilon decay

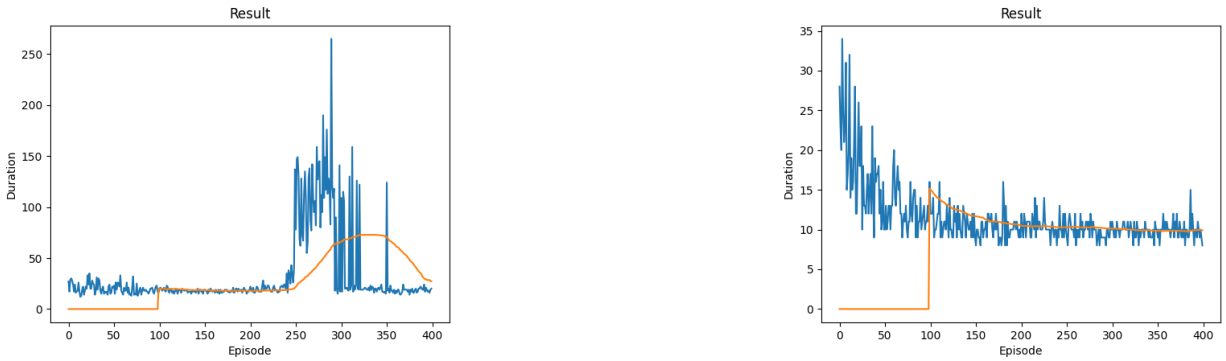
2.2 Varying the number of layers in the model



(a) 3 layers (b) 4 layers (c) 5 layers (d) 6 layers

Figure 5: Result of changing number of layers in the network

2.3 Implementation without the replay buffer



(a) With replay buffer (b) Without replay buffer

Figure 6: Comparison of implementation with and without replay buffer

3 Task 3: Q-learning and SARSA for the CartPole-V1 environment

Task 3 required us to implement Q-learning and SARSA to solve the CartPole-V1 environment. My implementation does not work, however I do have an idea what is missing and how to implement it.

Firstly, if I managed to solve either Q-learning or SARSA, it would be easy to solve the other one. This is because the logic of both are the same, they differ in how they update their Q-values.

The problem with the implementation is that Q-learning and SARSA requires a translation or a mapping of the observations in the environment to the index of the Q-table. Since the environment has 4 different states and they range from 0 to 1, the observation is not compatible to be used as an index in a table, and therefore require an translation that converts the observation into an index.

The reason we want all this is because the Q-table holds all the Q-value of every state-action pairs (state is the observation of the environment), which guides the agent's decision-making process by estimating the expected future reward of taking a specific action in a particular state.

References

[1] Kaustav kumar Chanda. Q-Learning in Python. In GeeksforGeeks. (21 Mars 2024), from `https://www.geeksforgeeks.org/q-learning-in-python/`