## Benefits of Using Experience Replay in DQN and How Its Size Can Influence the Result

Experience replay allows an agent to learn from past experienced by storing the previous experiences in a replay buffer and then sampling experiences from the buffer randomly during training.

Advantages if using Experience Replay

- Decorrelate sequential experiences by randomly sampling experiences from the replay buffer. This is necessary because training directly on sequential experiences would result in high temporal correlation resulting in unstable learning.
- Reusing previous experiences allows the agent to maximize learning from each interaction with the environment.
- Training is more stable as the variance in updates is reduced.

Buffer Size Influence

- When the buffer size is small the agent can only store recent experiences meaning the diversity of stored experiences is small. This can potentially result in overfitting to recent experiences.
- When the buffer size is large there is a greater diversity of experiences this also ensures the agent dosent miss out on older experiences which were useful. However if the buffer size is too large, the agent will continue to learn from older and outdated experiences that don't correspond to the current policy and therefore result in a slower learning process.

**Introducing the target network**

- The update rule in Q learning depends on both the current and next q values which results in instability if both fluctuate during the training process. The target network provides a stable reference Q value as its updated after a defined period of time.
- Prevents oscillation and divergence problem.
- Overall more smother training as the target network updates less frequently.

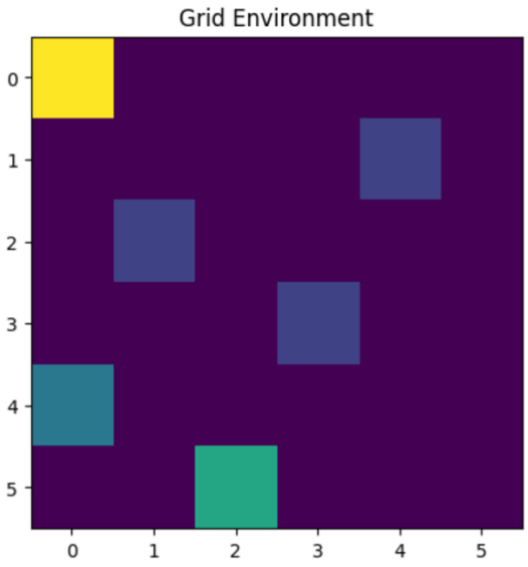
**Representing the Q function as qˆ(s, w)**

- Neural networks can compute complex functions, can handle high dimensional inputs and handle large state spaces.
- A parametrized Q function generalizes over different states.
- More efficient learning as neural networks can update multiple q values simultaneously because of shared weights.

## Describing Environments

**Warehouse Robot Scenario:**

A robot operates in a warehouse grid where its task is to pick up an item from specified locations and deliver them to designated drop-off points. The warehouse has shelves that act as obstacles, and the robot must navigate around them efficiently.




The above picture is the Environment that the agent, obstacles and drop off points start off with.

- Initial position of Agent is (0,0) in Yellow
- Positions of obstacles are (2,1), (1,4) and (3,3) in Blue
- Item is located at (4,0)
- Drop off location is at (5,2)

The reward is distributed using the following mechanism:

- Each step the robot takes will result in -1.
- Agent can take any position on the board except for the positions of the obstacles.
- If agent hits an obstacle the reward will be -20 however the agent will not move into the block of an obstacle. It will remain in the position it was in prior to hitting the obstacle.
- Item can be picked up and then dropped off at any location (except for locations the obstacles are locates in because agent cannot enter them).
- If item is dropped off at the drop off location +20.
- There is no penalty for hitting the edge of the grid.
- The agent can take 6 actions:
    - Up
    - Down
    - Left
    - Right
    - Pickup
    - Dropoff

**CartPole-v1**

- Environment: The agent controls a cart that can move left or right on a one-dimensional track with a pole balanced on it.
- Goal: The objective is to keep the pole upright for as long as possible by moving the cart to balance it.
- State Space: The state includes four continuous values: cart position, cart velocity, pole angle, and pole velocity.
- Actions: There are two discrete actions — move the cart left (0) or move the cart right (1).
- Rewards: A reward of +1 is given for each time step the pole remains balanced. The episode ends if the pole falls, or the cart moves off-track.
- Agent: The agent learns to control the cart to maintain balance.

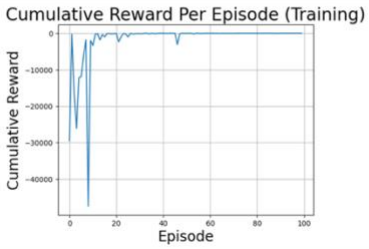
**CliffWalking-v0**

- Environment: A grid world with a "cliff" along the bottom edge, except for the starting and goal positions.
- Goal: The agent starts at one end of the grid and must reach the goal on the other end while avoiding the cliff.
- State Space: The grid cells represent discrete states (e.g., (x, y) coordinates).
- Actions: Four discrete actions are available — move up (0), down (1), right (2), and left (3).
- Rewards: The agent receives -1 for each move, -100 for falling off the cliff (which resets it to the start), and 0 upon reaching the goal.
- Agent: The agent learns to navigate the grid, balancing between reaching the goal quickly and avoiding the cliff.

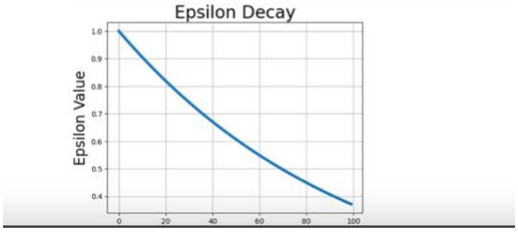
## Show and discuss your results after applying your DQN implementation on the three environments. Plots should include epsilon decay and the total reward per episode.

**Warehouse Robot Scenario:**

```
In [64]: # Reward Dynamics
         plt.figure()
         plt.plot(rewards_per_episode_deterministic)
         plt.xlabel('Episode', fontsize=20)
         plt.ylabel('Cumulative Reward', fontsize=20)
         plt.title('Cumulative Reward Per Episode (Training)', fontsize=24)
         plt.grid()
         plt.show()
```




```
In [65]: # Epsilon-decay graph
         plt.figure()
         plt.plot(epsilon_values_deterministic, linewidth=4)
         plt.xlabel('Episode', fontsize=20)
         plt.ylabel('Epsilon Value', fontsize=20)
         plt.title('Epsilon Decay', fontsize=24)
         plt.grid()
         plt.show()
```

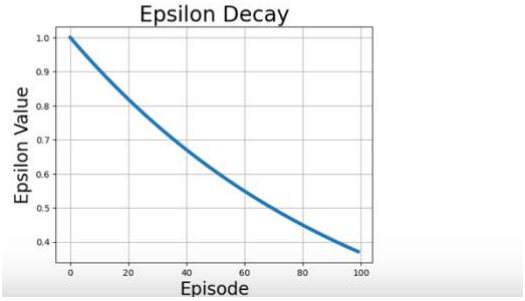


From the graph above we see that when the epsilon value is high, there is a lot of learning taking place with a high cumulative reward per episode, but as the epsilon value decays, the agent moves from the exploring the environment to exploiting it and has learnt the optimal policy which results in the agent achieving the best reward to complete an episode which is 13.

**CartPole Environment:**

```
plt.figure()
plt.plot(rewards_per_episode_cartpole)
plt.xlabel('Episode', fontsize=20)
plt.ylabel('Cumulative Reward', fontsize=20)
plt.title('Cumulative Reward Per Episode (Training)', fontsize=24)
plt.grid()
plt.show()
```


```
# Epsilon-decay graph
plt.figure()
plt.plot(epsilon_values_cartpole, linewidth=4)
plt.xlabel('Episode', fontsize=20)
plt.ylabel('Epsilon Value', fontsize=20)
plt.title('Epsilon Decay', fontsize=24)
plt.grid()
plt.show()
```

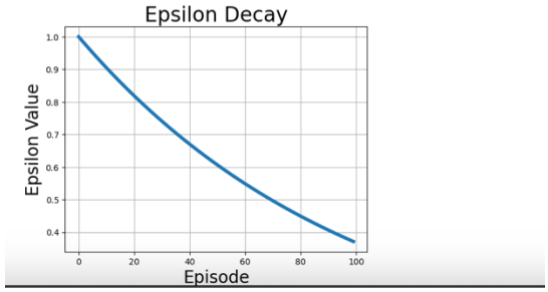



When looking at the cumulative result per episode graph, we see that early on some learning took place but as we progressed the learning was very unstable between episode 30 and 42 with a high cumulative reward and high variance in that reward per episode this may be due to the learning hyperparameters or the fact that a lot of exploration was taking place. As the episodes increased after 50, we see that the epsilon value drops to around 0.3 and lower resulting in more exploitation than exploration which in turn means the overall reward per episode got better and better.

**CliffWalking Environment:**

```
plt.figure()
plt.plot(rewards_per_episode_LunarLander)
plt.xlabel('Episode', fontsize=20)
plt.ylabel('Cumulative Reward', fontsize=20)
plt.title('Cumulative Reward Per Episode (Training)', fontsize=24)
plt.grid()
plt.show()
```

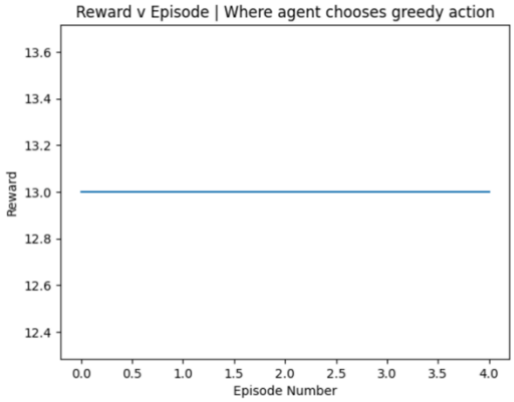

```
# Epsilon-decay graph
plt.figure()
plt.plot(epsilon_values_LunarLander, linewidth=4)
plt.xlabel('Episode', fontsize=20)
plt.ylabel('Epsilon Value', fontsize=20)
plt.title('Epsilon Decay', fontsize=24)
plt.grid()
plt.show()
```



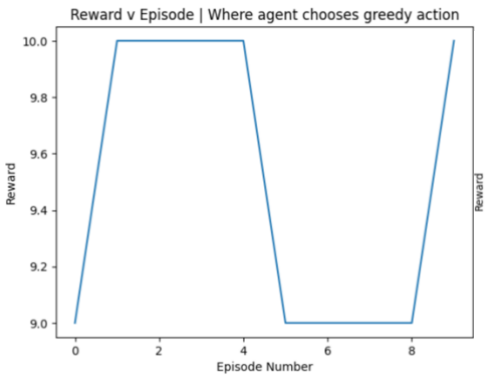

From the graph above we see that when the epsilon value is high, there is a lot of learning taking place with a high cumulative reward per episode, but as the epsilon value decays, the agent moves from the exploring the environment to exploiting it and has learnt the optimal policy which results in the agent achieving the best reward to complete an episode which is -13.

## Evaluation Results

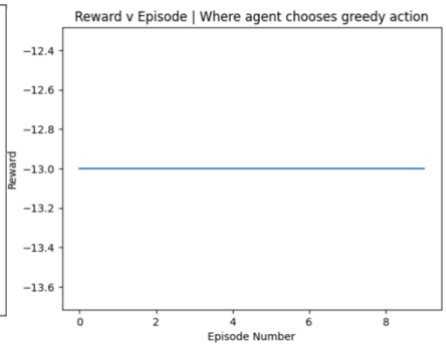
**Warehouse Robot Scenario**




**CartPole**




**CliffWalking**



We can see from the graphs above that the DeepQNetwork has learnt the optimal policy for the Warehouse and CliffWalking environments as we have the maximum reward per episode for each of the episode we have run the environment on when selecting the greedy policy. However, the agent has not learnt the optimal policy for the cartpole problem as the overall cumulative reward per episode is not constant indicating that more learning is required.

# Checkpoint 3 Content

**DDQN**

I choose to implement DDQN to the previous environments that I implemented DQN on. DDQN is an improved version of DQN as it reduces the effects of overestimation of Q Values in DQN.

In standard DQN the same network is used for selection and evaluation of actions while in DDQN there are 2 separate networks that work together to do action selection and evaluation.

**Two Networks (Online and Target)**: DDQN maintains two separate networks:

- **Online Network (Q-Network)**: Used to select actions.
- **Target Network**: Used to evaluate the Q-value of the selected action.

**Action Selection and Evaluation**:

- In each update, the Online Network chooses the action with the highest Q-value for the next state.
- The Target Network then evaluates the Q-value of this chosen action.
- The key difference is that the Online Network selects action, but the Target Network evaluates action.

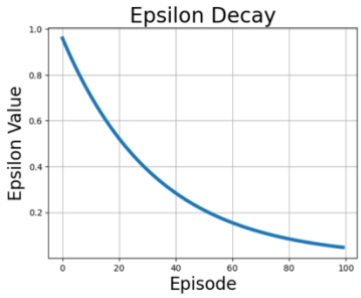
**Training the Online Network**:

- The Online Network is trained to minimize the difference between its predicted Q-value and the target Q-value provided by the Target Network.
- This difference, is used to update only the Online Network, while the Target Network is updated after a defined number of iterations.

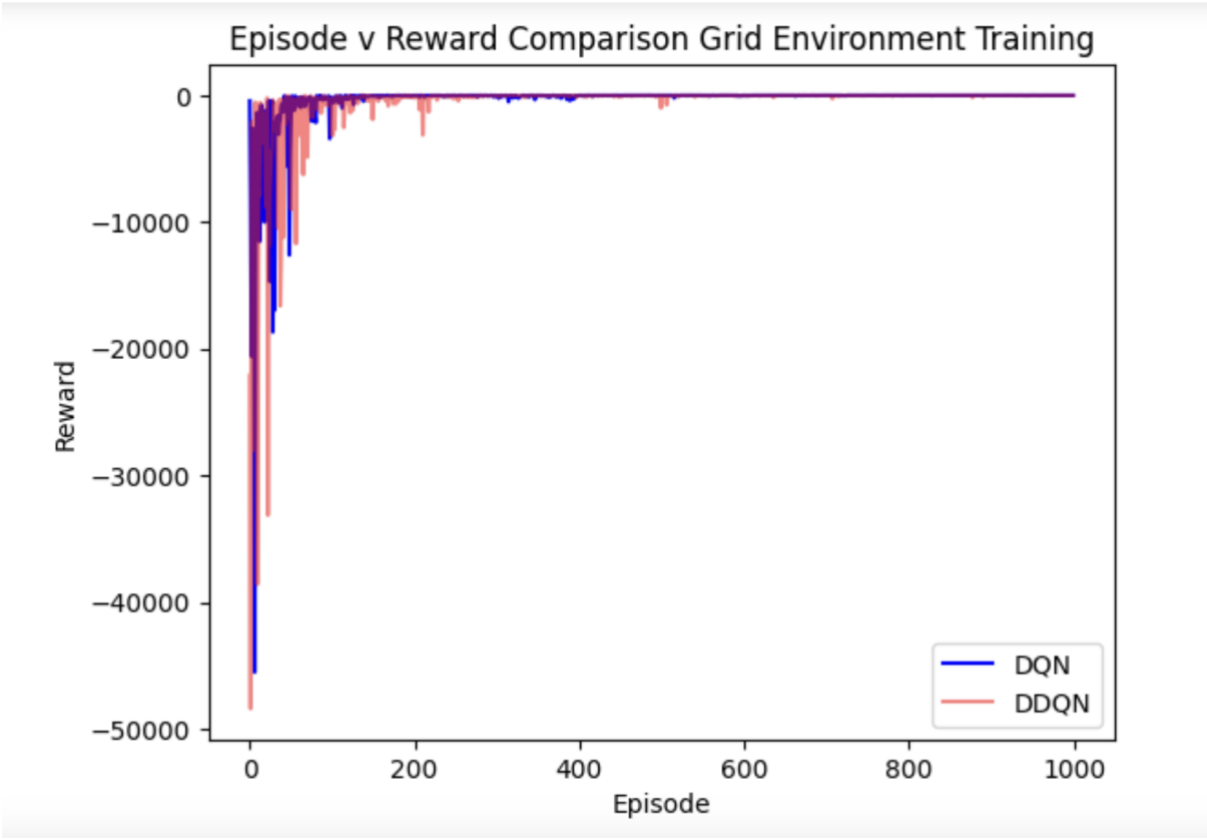
**Why DDQN is an Improvement**

- By separating action selection and evaluation, DDQN reduces the tendency to overestimate Q-values,
- This leads to more reliable and stable updates.
- Reduced overestimation often converges to optimal policies more efficiently.

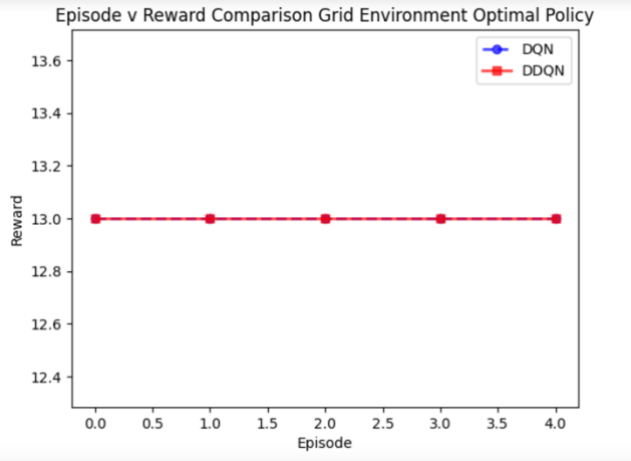
# Grid Environment:




The above graph shows the epsilon decay during the training process for both the DDQN and DQN algorithms.

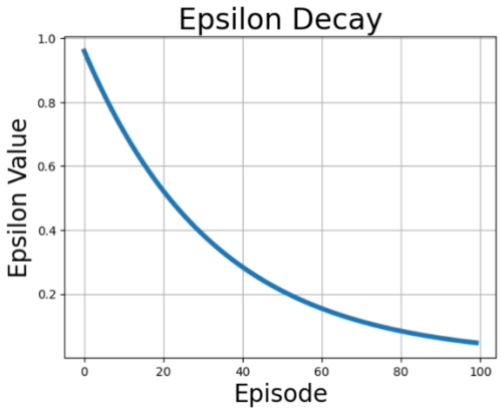



The graph above shows the training process of DQN and DDQN. Initially the rewards are very negative however as the episodes increase the agent starts exploiting the environment, the rewards tend to become higher. We see from the graph that the DQN algorithm converged to the most optimal policy before DDQN and that the DDQN algorithm tends to have more instability or spikes in the reward values after initial learning suggesting it may have more instability than DQN in this particular environment.

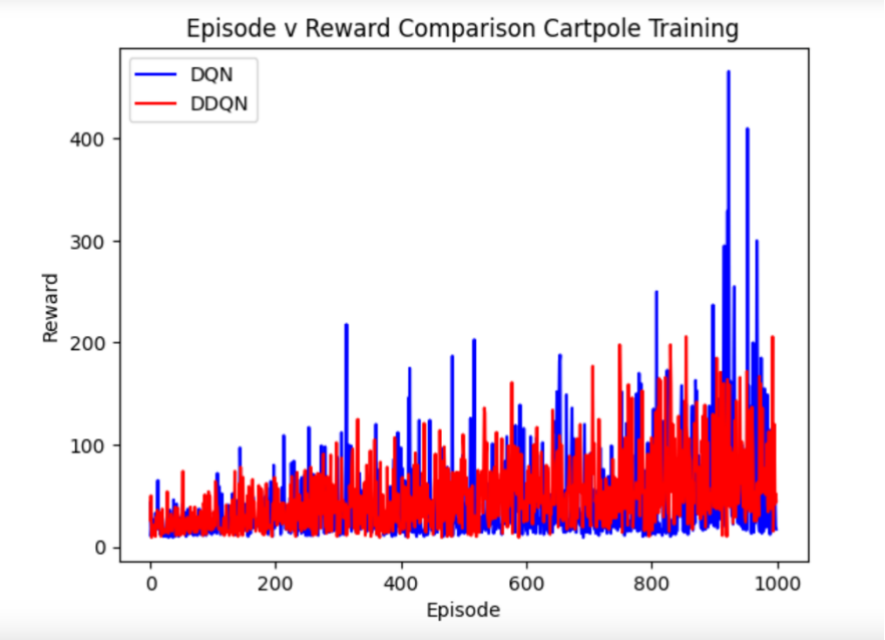



The above graph shows the reward of running both the algorithms on the optimal policy. We see that both the algorithms obtained the maximum possible reward proving the fact they learnt the optimal policy for this environment.

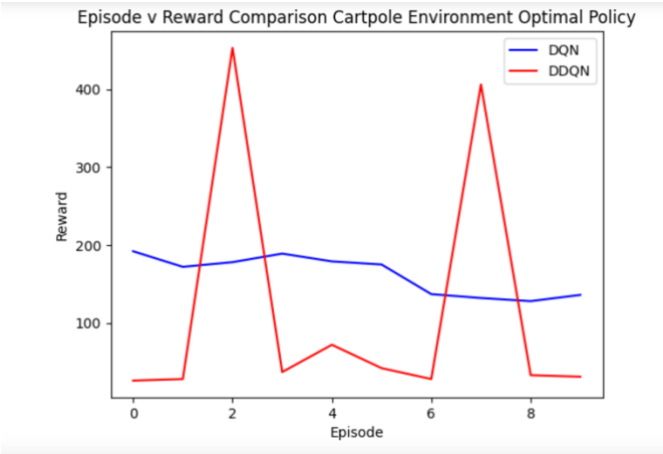
## Cartpole Environment




The above graph shows the epsilon decay during the training process for both the DDQN and DQN algorithms.

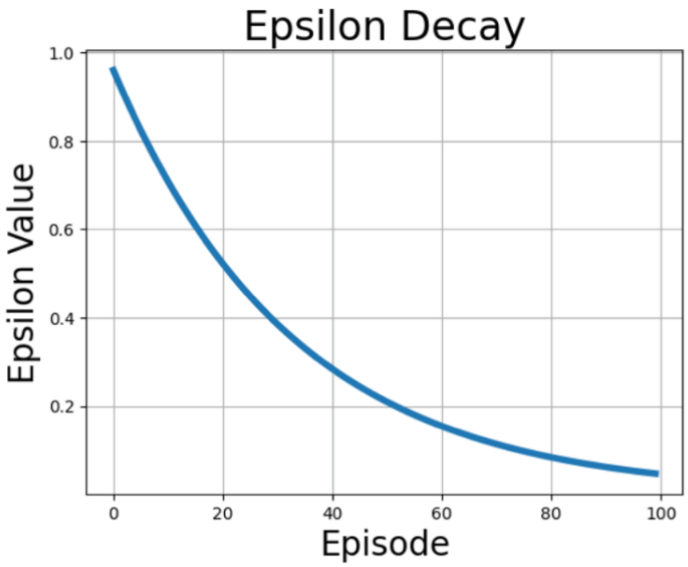



From the above graph we see that reward increases over time for both the algorithms when the algorithm starts exploiting the environment after learning during the initial phases. However, neither algorithm has fully stabilized or converged. DQN has higher spikes in rewards compared to DDQN which has comparatively low spikes suggesting more consistent but less optimal performance.

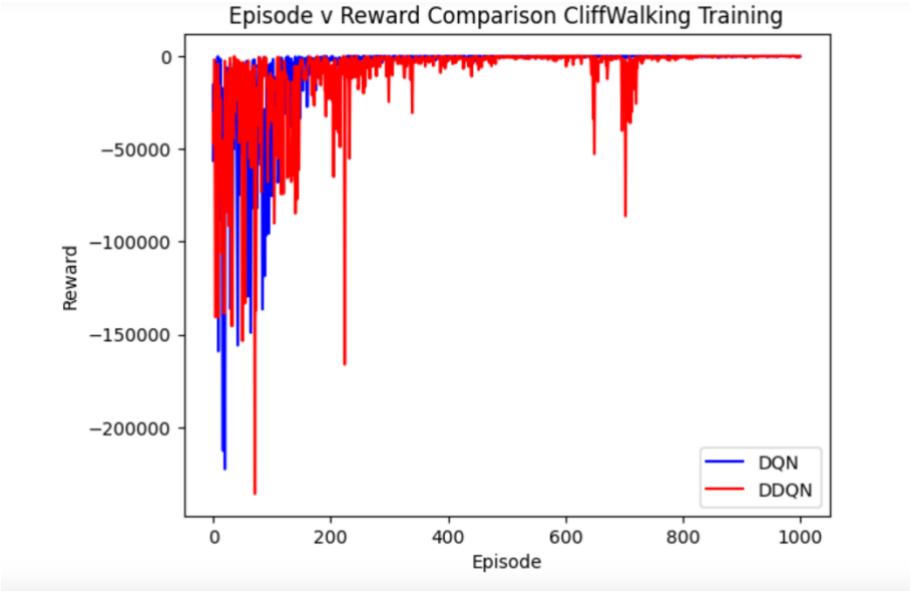



The analysis conducted in the training process aligns with our result of running the most optimal policy on the environment with DQN showing comparatively lower but stable rewards while the DDQN algorithm shows more variation with some episodes such as 2 and 7 having very high reward while other episodes such as 1,3,4,6,8 having much lower rewards. This shows that DDQN has learnt a better policy however is not able to apply it to every episode.

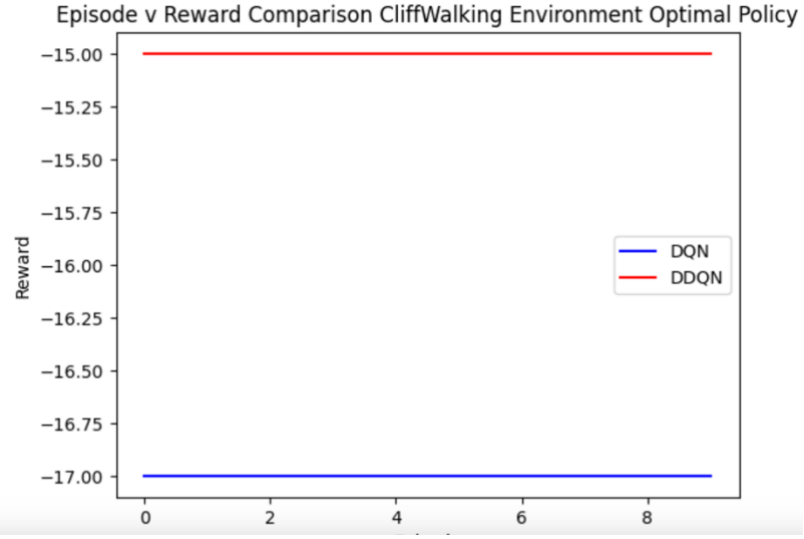
## Cliff Walking




The above graph shows the epsilon decay during the training process for both the DDQN and DQN algorithms.




From the above graph we see that as the episodes increase the algorithms learn a better policy since the agents start exploiting the environment after learning resulting in receiving a higher reward and converging closer to the optimal policy. Both algorithms show a rapid convergence towards higher rewards, DQN stabilizes faster with fewer deep drops in rewards after the initial learning process on the other hand DDQN shows more pronounced and frequent spikes in negative rewards suggesting it is struggling with stability. In this environment it seems like DQN is more robust.




The graph above shows the reward of running the optimal policy on the environment for both the algorithms. They both have stable rewards with DDQN having a higher reward implying it has learnt a better policy than DQN.

## Conclusion

DQN is Better Suited for Environments Requiring Robustness and Rapid Stabilization as DQN tends to converge faster and stabilize sooner as well as Low-Variance or Simpler Environments as DQN can efficiently learn without significant fluctuations for example Cliff Walking and Grid environments.

DDQN is Better Suited for Environments Prone to Overestimation Bias: as DDQN addresses the overestimation issue common in DQN by using a double Q-learning structure. Scenarios Favoring Stability Over Immediate Peak Rewards as DDQN provides a more cautious and stable learning curve. For example: Complex environments like Cartpole, where reward estimates can easily become exaggerated, and stability is preferred over high immediate rewards.

## References

https://campus.datacamp.com/courses/deep-reinforcement-learning-in-python/