# Deep Learning

## Project 2

## Team Rocket

Members:

David Castro nº 60973
Guilherme Antunes nº 70231
José Morgado nº 59457
Ricardo Rodrigues nº 72054

# Task 1

# Problem Definition

The goal of this project is to train an agent to play the snake game using reinforcement learning. Given this, we feel it is important to define the exact scenario we are using for this purpose, as this will directly affect our approach and results:

- A board of 32x32 (30x30 board and 1 pixel border), we chose a power of 2 to make it practical for convolution and pooling; this size gives us plenty of space for exploration.
- No grass on the board
- A single apple, which is our agent’s goal
- A maximum of 500 steps per training episode. This may influence the late game playstyle of our agent, even though very frequently the episode ends before reaching step 500.

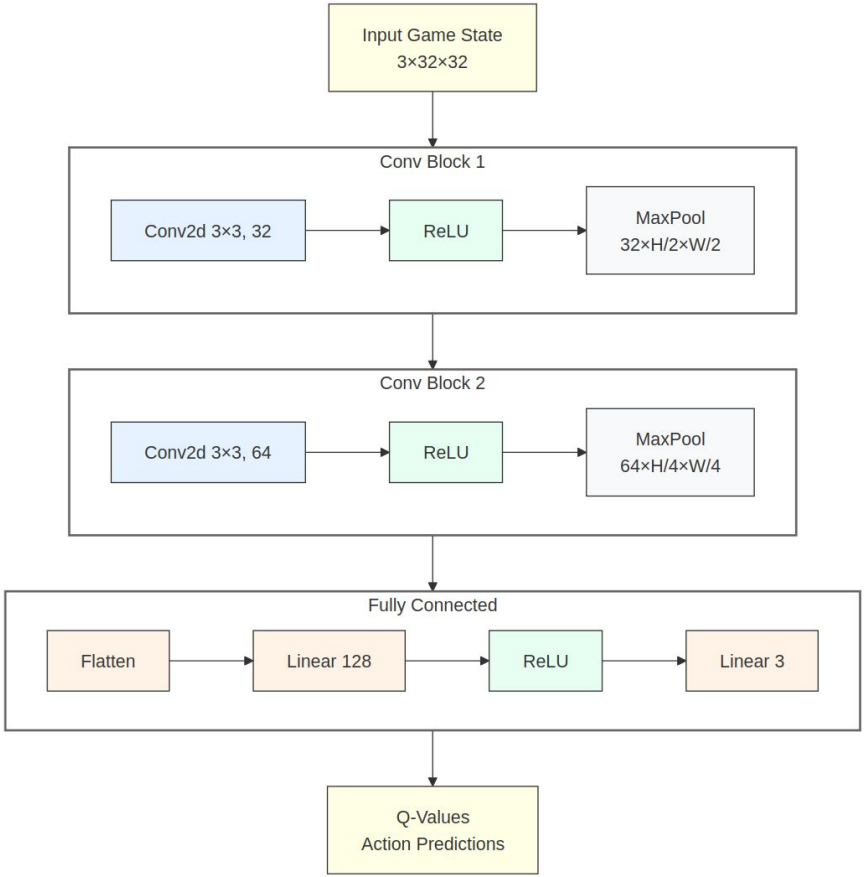
# 1.1 Model Architecture

Since our problem could be simplified to the processing of an image (the board state) to obtain a decision, a CNN architecture was the natural choice. The neural network approximates the Q-value function Q(s,a), which represents the estimated gain for taking a given action “a”, on a given state “s” (game frame).

The input is an image representing the game state, formatted as a 3D tensor (C,H,W), where C is the number of channels, H and W are the height and width of the board (with the border)

The model uses two convolutional layers to extract spatial features; the layers reduce the spatial dimensions and increase depth, enabling the network to learn relevant spatial data from the input frames. Each of these layers is connected to a ReLU, as it is the standard non-linear activation function, and finally to a max pooling layer to reduce spatial dimensions while still maintaining the most important features.

Finally, we employ a fully connected layer to piece the extracted information together and make an *informed* decision about our Q-values.



# 1.2 Q-learning Loss Function

Our loss function is defined as the mean squared error of the Bellman equation, given by:

$$Loss = (r + \gamma * max(Q(s', a')) - Q(s, a))^2$$

To decompose this into simpler terms, we have the difference between two components, the target and the prediction. The prediction is fairly simple, it is the taken action's predicted Q-value. The target represents the desired predicted value, hence why we subtract the two. This is done by calculating the highest Q-value for the following state, multiplying it by gamma, which is the weight we give to future rewards, plus the actual reward obtained by the action we took. Finally, to ensure a correct target value, we also multiply it by (1-done); this is done so that an action that leads to the game ending (hitting a wall or itself) has an expected Q-value of 0.

The python implementation is seen on the right:

```python
def compute_q_loss(pred_q, action, reward, next_state, done, model, gamma=0.99):
    with torch.no_grad():
        next_q = model(next_state)
        max_next_q = next_q.max(1)[0]
        target_q = reward + gamma * max_next_q * (1 - done)

    pred_q_action = pred_q.gather(1, action.unsqueeze(1)).squeeze(1)
    loss = F.mse_loss(pred_q_action, target_q)
    return loss
```

# 1.3 Training loop and evaluation

For our training we implemented a fairly simple and standard loop, iterating over the number of *episodes* we set, where each *episode* corresponds to a game. But first we need to set some variables and hyperparameters; we set ours as follows:

- **Discount factor (γ):** 0.99 – A higher gamma encourages our agent to stay alive and search for long-term rewards
- **Exploration (ε):** 0.005 – We defined a small ε to to enforce exploitation of the learned Q-Values, with rare exploratory actions to prevent local maxima entrapment
- **Learning rate:** 1e-4
- **Optimizer:** Adam
- **Number of episodes:** 1000 – This corresponds to the number of iterations of our outer loop.
- **Max game steps:** 500 – The maximum number of steps a game takes, finishing at this number of steps if it doesn't end before. A game corresponds to an episode.

# 1.3 Training loop and evaluation

Our actual training starts by initializing a SnakeGame, as defined in our previous Problem Definition slide. Every episode starts by reseting our environment (the SnakeGame). After that, we start playing the game, corresponding to the inner loop:

- We take a step if game is not over yet (done != 0) and number of steps < max number of steps
- Every step we select an action using one of the following methods:
  - Heuristic based moved
  - Epsilon based random move
  - Model calculated highest action Q-value
- We take that action in our environment to get the next state
- We use all this information to calculate the loss
- We apply backpropagation

# 1.3 Training loop and evaluation

We first ran a training session without using any heuristic player actions, relying entirely on the model's Q-value predictions.

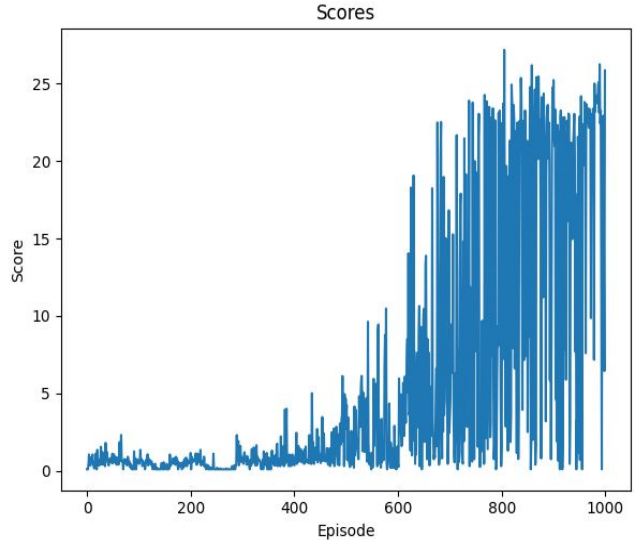
A fixed epsilon value of 0.005 was used throughout training, without any decay, to maintain a predominantly greedy policy while still allowing occasional random exploratory actions.

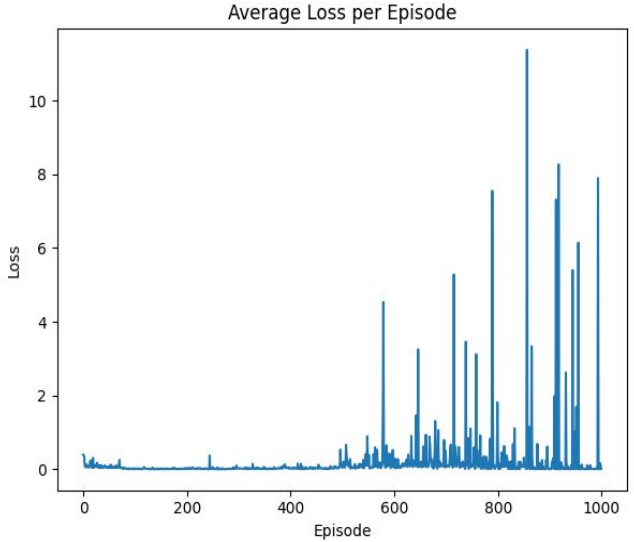
This small but nonzero probability ensures the agent continues to explore the state-action space, reducing the risk of getting trapped in local maxima of the Q-target estimates. Specifically, at each decision step, the agent selected a random action with probability $\varepsilon = 0.005$, otherwise, it chose the action corresponding to the maximal Q-value.

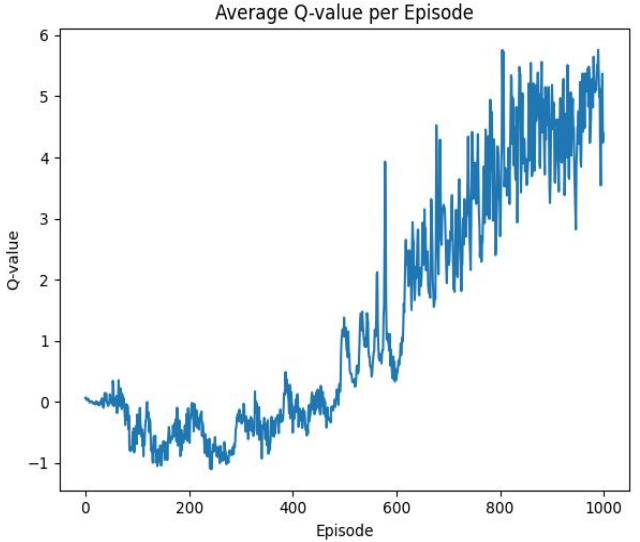
# 1.3 Training loop and evaluation

Over time, the agent's performance improved substantially. The average score increased from 0.64 in the first 50 episodes to 19.83 in the final 50 episodes, indicating that the agent learned to locate food and avoid self-collisions effectively, even with minimal exploration. The model took 548.59 seconds to train, on a NVIDIA GeForce GTX 1650 Mobile GPU.

This training behavior represents a proper DQN baseline: slow, gradual improvement from a poor initial policy toward a competent one, even without external guidance.

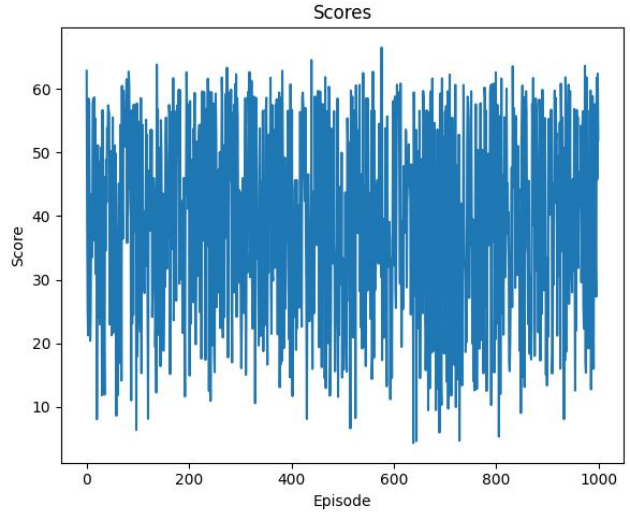


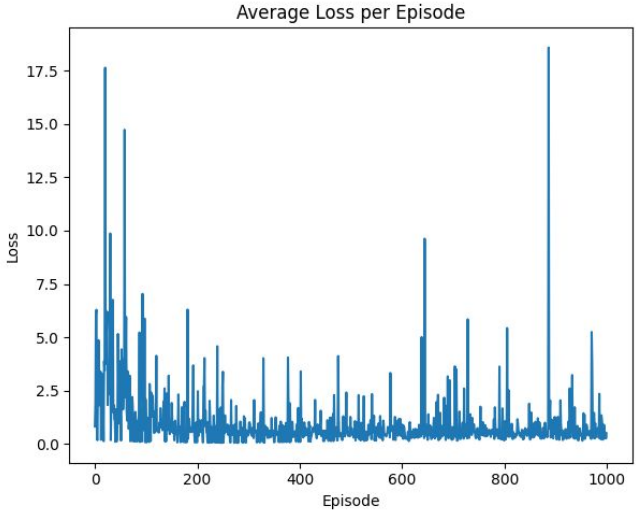



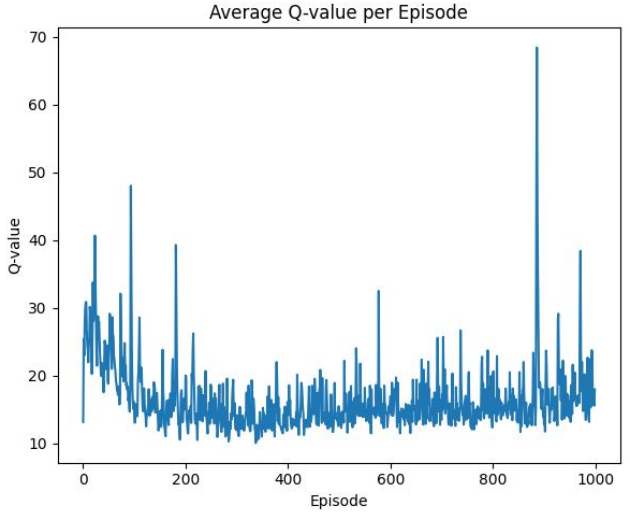



# 1.4 Heuristic Policy

In our previous examples we were selecting our actions, in one way or another, randomly. While we got some interesting results, by implementing an heuristic policy to teach our agent, we are able to learn much better and faster. We implemented an heuristic that uses the snake's Manhattan distance to the apple to select our next best move, while avoiding walls and itself. It calculates which action leads to the greatest decrease in distance to the apple and breaks ties in the order of straight > right > left. Bellow, we observe our results by executing a training run **selecting all our actions with the defined heuristic**, giving us a baseline for this heuristic. We reach an average score of just under 40(39.18) and converge on an average Q-value of 16.3885. The model took 565.52 seconds to train, on a NVIDIA GeForce GTX 1650 Mobile GPU.

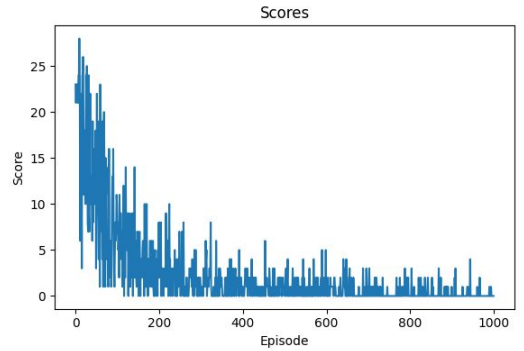


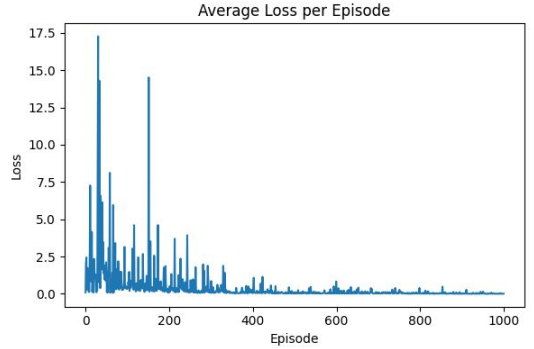



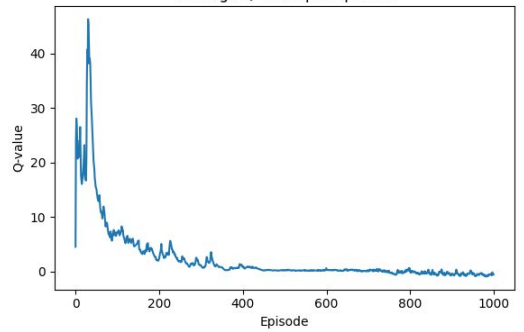



# 1.4 Heuristic Policy

After setting our baseline we ran another training run, but this time with a decaying epsilon, so that the number of heuristic moves we took gradually decreased. Starting at 1, with the value we set previously, by episode 1,000 epsilon had decayed to around 0.13. The most clear pattern we see in our results is just how poorly our model learns, with the scores and Q-value decreasing as our number of heuristic moves decreases. This is the perfect example of the problem of training without a target network, which causes the training of our model to be too unstable and converging on 0 for every metric.



# Task 2

# 2.1 Experience Replay

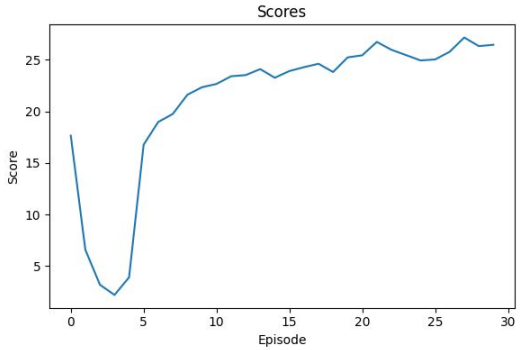
For this section we tested 2 different approaches, first we used the *ReplayBuffer* implementation provided by the professor and then we implemented our own, with Prioritized Experience Replay, where for both of them we start by running a few pure heuristic episodes to warm start the buffer. This technique is very useful for a faster initial learning, gathering the more informative experiences with a higher priority, and for dealing more effectively with edge cases, since they usually result in a higher loss and are therefore sampled more often. Given the exact challenge we are tackling, however, we theorized that this would not have a significant impact, given its simplicity and lack of rare occurrences. Further in this presentation we will present our results,  where our theory proves being correct. The implementation of this technique requires some additional nuances relative to the base *ReplayBuffer*. When adding to the buffer, we set the priority to the max, since we don't yet know how important that experience is. When sampling, we use the relative priorities to calculate the probability of picking a given experience. We also retrieve the indices of the experiences we samples, so that after training, we use the computed loss to update their priorities.

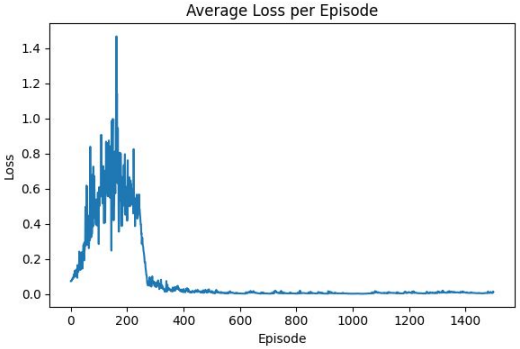
# 2.2 Target Network

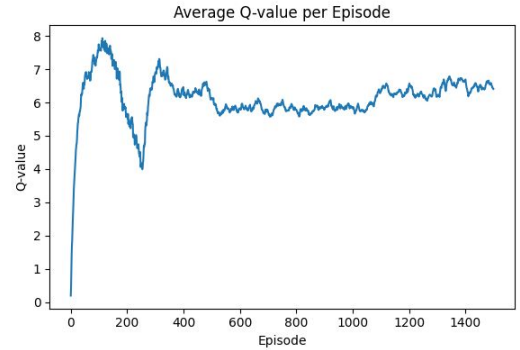
For implementing the target network we required fairly few modifications. We start by simply copying the weights of the model we initialize in the beginning to a separate network. Then, during training, we predict the Q-values with our model and with the target network and use these two to calculate the loss. The main model is updated every step while the target is updated every 100 steps, this value can be changed but it is what we went with. In order to update the target weights we decided to use the technique of simply copying the weights from the main network.

# 2.3 Integration and Performance Evaluation

When putting everything together the results are what we can see on the right, using the base *ReplayBuffer*. Our training is extremely more stable now, with not only the loss but also our average Q-value and scores converging. Also important to note is the exact values they are converging to: an average score of 25 and Q-value of 6. These are very close the the values we were obtaining by solely using the heuristic policy, hinting that our agent learned extremely well and gathered most of the important patterns to correctly play and win at the snake game. The model using the basic Replay Buffer took 16 minutes and 47 seconds to train, on a NVIDIA GeForce GTX 1650 Mobile GPU.

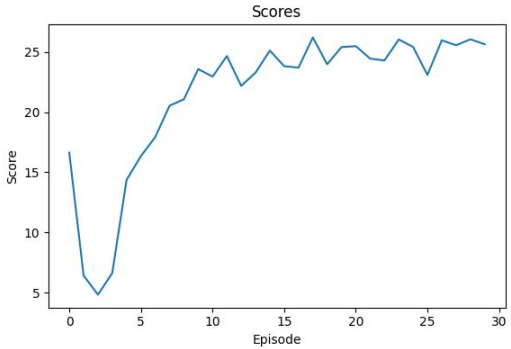


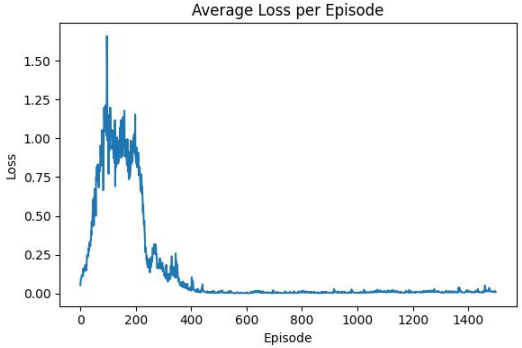



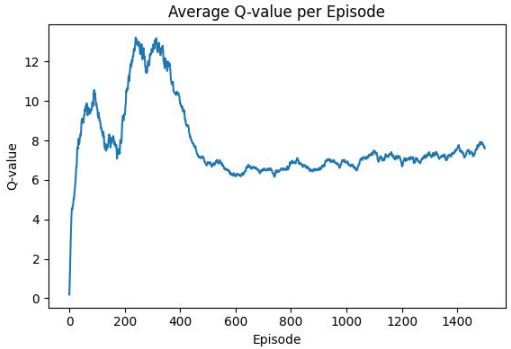



# 2.3 Integration and Performance Evaluation

When using the *priority replay buffer*, what we see is a higher average loss initially, but a convergence on the same values. This is the due to the reason we discussed earlier, where this method would give us better results in a scenario where more uncertainty is to be expected, which is not our case. The most notable effect is on the loss, we see it has a much higher average. This makes sense since the priority replay buffer samples experiences with higher loss more often. The model using the Priority Replay Buffer took 20 minutes and 55 seconds to train, on a NVIDIA GeForce GTX 1650 Mobile GPU.



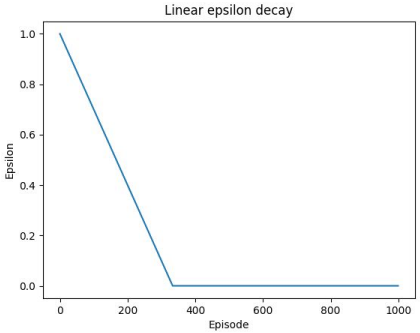



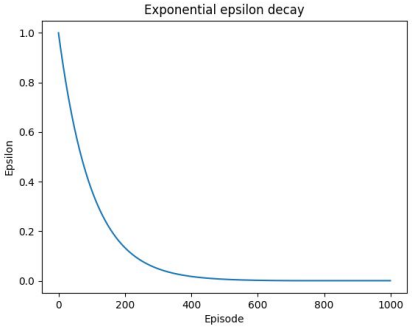



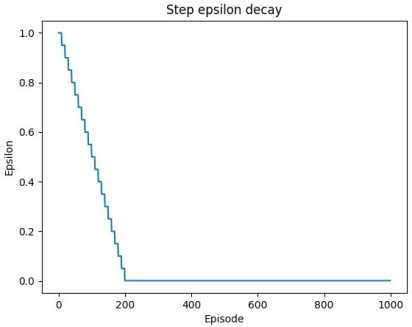
# Task 3

# 3.1 Epsilon Greedy

For this task we used every technique we had implemented earlier while still following the guidelines in the project description, namely the replay buffer and target network. The replay buffer we used was the base one given by the professor, with a warm start of 50 episodes using our heuristic. We used all three suggested decay patterns: linear, exponential and step. Their plotted values are seen on the right. For a fair comparison, we tuned their decay parameters so that they reach their minimum values at roughly the same episode. Epsilon's value starts at 1 and decays until a minimum of 0,001. Each of these experiments took roughly 12 minutes to run. On the following slides we show the results for each of these.

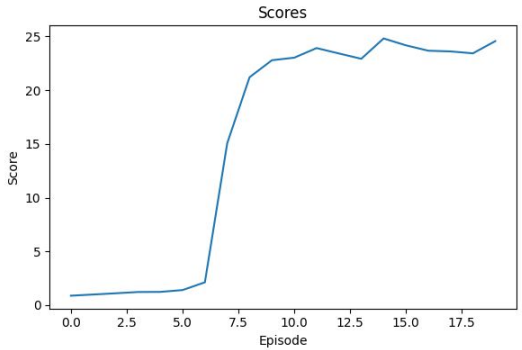


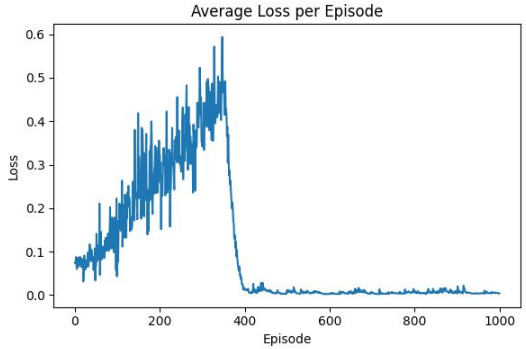



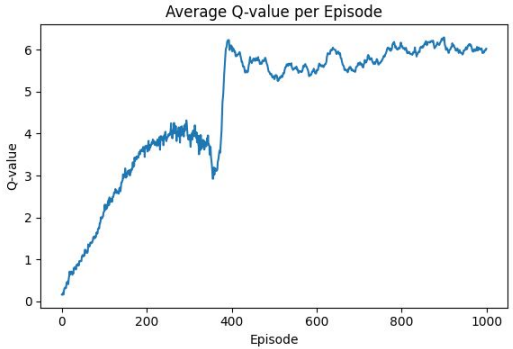



# 3.1 Epsilon Greedy – Linear

What we observe is a very clear distinction between the section where we use a meaningful epsilon versus the rest. As soon as it reaches its minimum value, the average scores and Q-value shoot up while the loss descends abruptly. As seen in previous sections, the simplicity of the setup we are working with is mostly likely the underlying cause of this. Since we have a good amount of time dedicated to exploration, our agent is able to experience a lot of different game scenarios and get a good underlying sense of how to play the game. When our epsilon reaches its minimum, the exploration phase ends and it starts being purely greedy, polishing its playstyle and strictly aiming for the highest score, without random actions pulling him back.

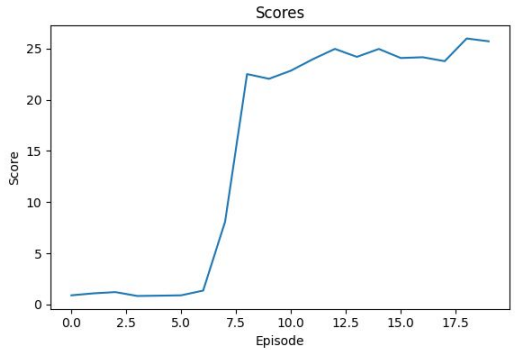


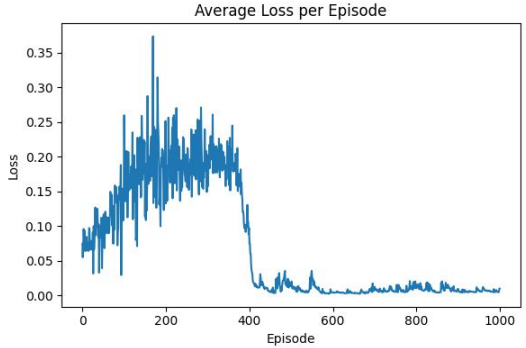



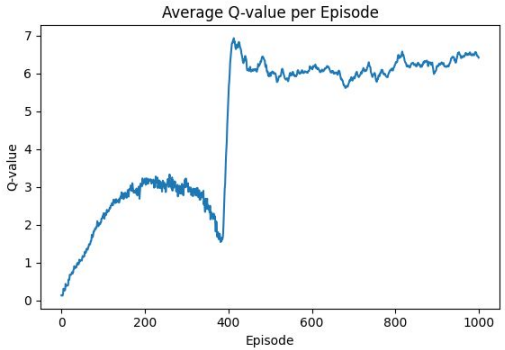



# 3.1 Epsilon Greedy – Exponential

The patterns we observe for this decay type is very similar to what we saw previously. A particularly interesting thing we also see is how the average loss and q-value, when comparing to the previous example, seem to follow a trend of the exponential line, while the previous lines follow a more straight, linear one.

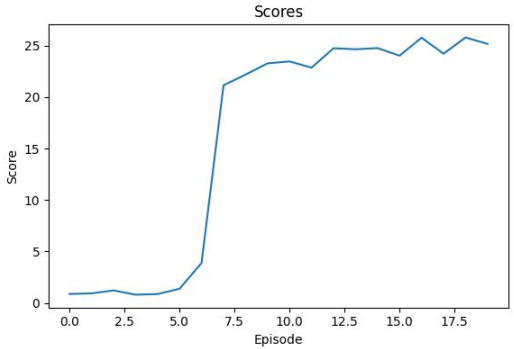


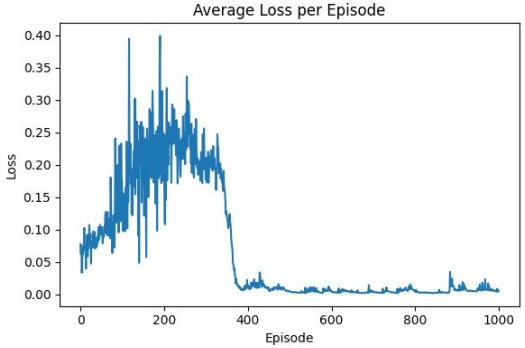



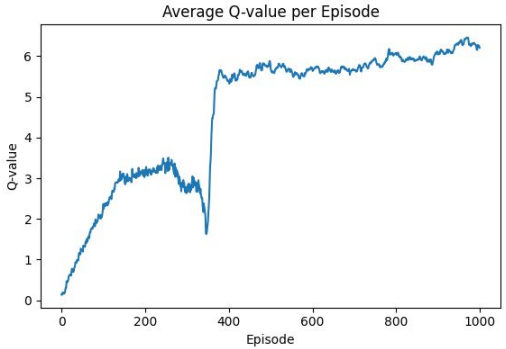



# 3.1 Epsilon Greedy – Step

With step decay what we see is very similar to our previous cases. This makes sense since, regardless of the decay pattern, what is actually happening is that we are using epsilons with fairly similar values. The results we get from this approach seem to be more noisy, possible due to the plateauing nature of a step-based approach. The plateaus are very likely to lead to a certain short-lived state of equilibrium between exploration and exploitation that cause the loss – and consequently the q-value, although not very clear from our graph – to shoot up and down, causing noisier looking plots.



# 3.1 Epsilon Greedy – Conclusion

After a lot of experimenting, the conclusion we reached was that none of these patterns produce different results by themselves, what is really important is the actual value epsilon is taking. The resulting average scores were pretty much the same for all of them, only seeing some slight changes in the loss and q-value graphs. The takeaway from this experiment is that the implementation itself is not important, the underlying pattern is. We get our best results by starting with a short exploration phase, followed by a long exploitation phase, so as long as the decay pattern reflects that, we are able to achieve a good score.

# 3.2 Boltzmann Exploration

This exploration method uses a Softmax approach paired with temperature scaling to decide what actions should be taken. The selection strategy consists of assigning a probability to each action based on their Q-values, the higher the value, the more likely the action is to be chosen. The temperature factor allows us to control the balance between exploration and greediness, with higher temperature there is more exploration, while with lower temperature the model becomes more greedy.

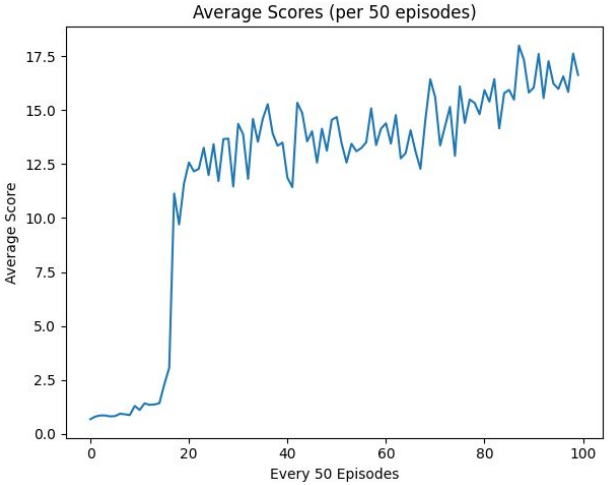
We will try three different temperatures to see how the model performs with more or less exploration.

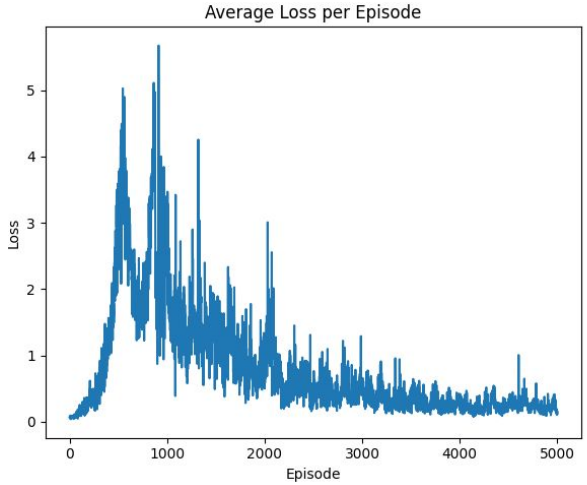
The following tests were all done on a CPU, so the training time will be considerably longer. The first temperature we tested, 2, took 1:21h to train, while for a temperature of 0.7 it took 1:53h to train. This leads us to believe that having more exploration lightens the training complexity.

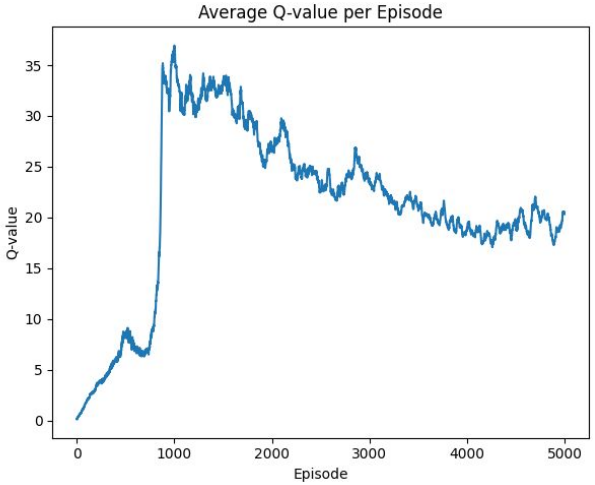
# 3.2 Boltzmann Exploration

These results were obtained with a temperature of 2, focusing more on random exploration. We can see that the loss performed well, and that the average scores saw a jump at 1000 episodes, after that it slowly improves.

The Q-values also go down, indicating that the model has learned a relatively good policy and is correcting its overconfidence.



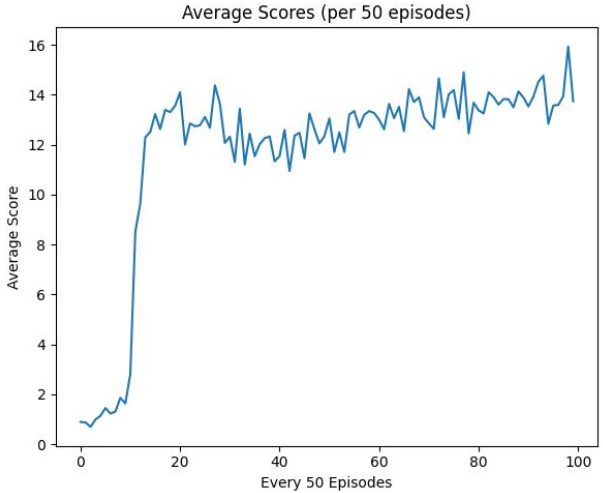



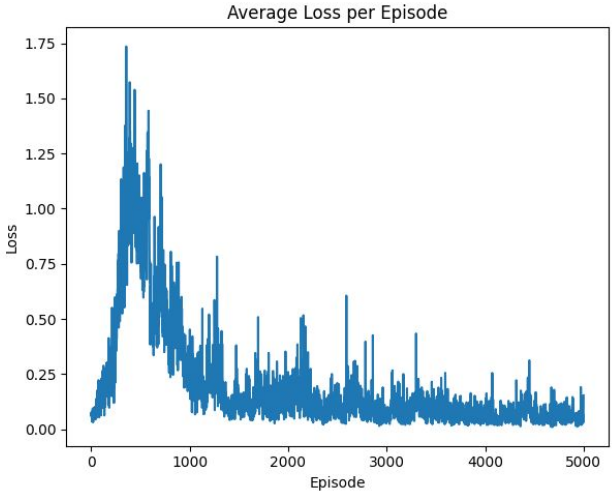



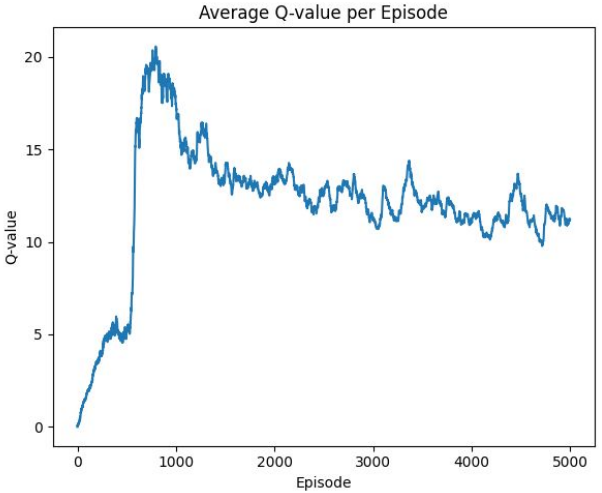
# 3.2 Boltzmann Exploration

These results were obtained with a temperature of 1, this provides a more balanced approach between exploration and exploitation.

The results were very similar to what we got with a more exploration focused approach in terms of stability, however the scores actually decreased slightly.



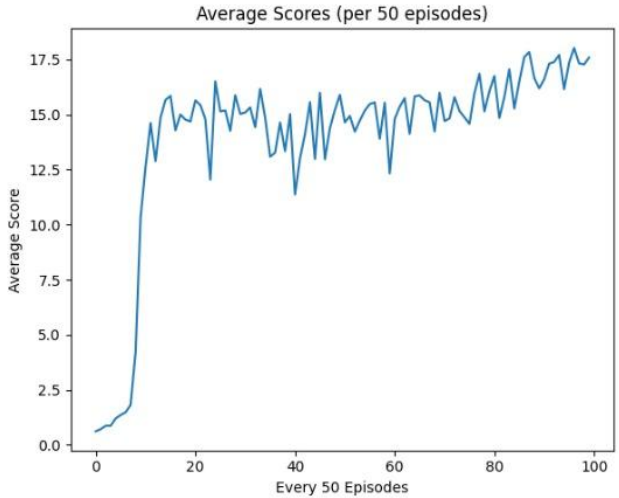



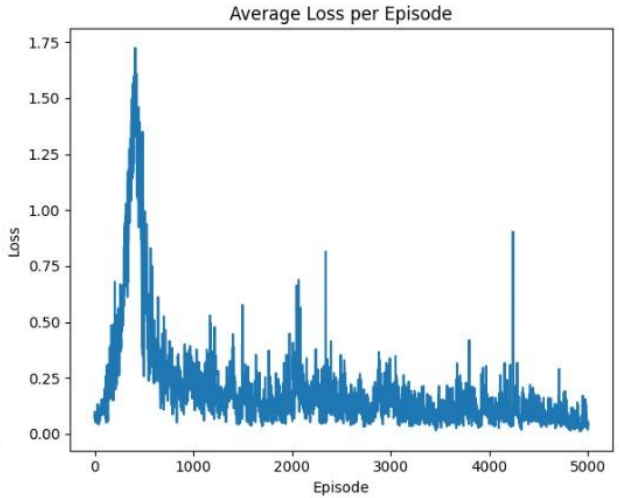



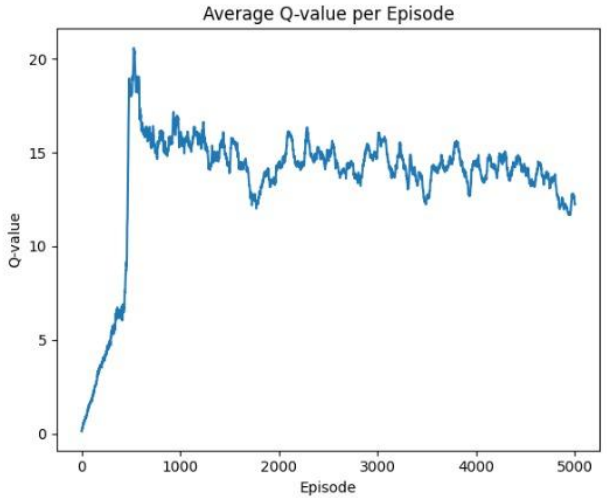
# 3.2 Boltzmann Exploration

These results were obtained with a temperature of 0.7, this leads to a more exploitation focused approach with some random exploration.

These results are more stable, after the first jump both the average scores and the average Q-values seem to become pretty stable. There is still a slight increase in score, and a slight decrease in Q-values.



# 3.3 Comparative Analysis

In this section, we compare the best-performing versions of the Epsilon-Greedy and Boltzmann exploration strategies by evaluating the learning efficiency, stability, exploration behavior, hyperparameter sensitivity, and final performance of the two models.

The Epsilon-Greedy model used an exponentially decaying epsilon schedule and was trained on an NVIDIA GeForce GTX 1650 Mobile GPU, taking 89 minutes and 55.3 seconds to complete training.

The Boltzmann model, using the temperature parameter set to 0.7, was trained on a CPU, requiring 1 hour and 53 minutes to finish.

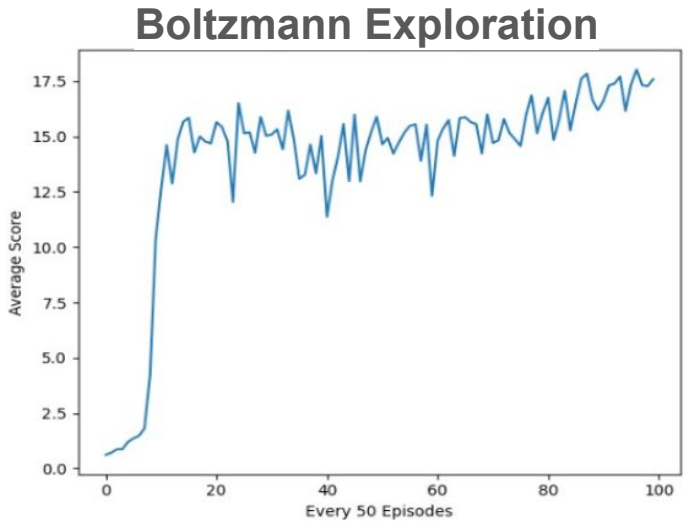
Both models were trained for exactly 5000 episodes, ensuring identical training duration and conditions for a fair and consistent comparison across all evaluated parameters.

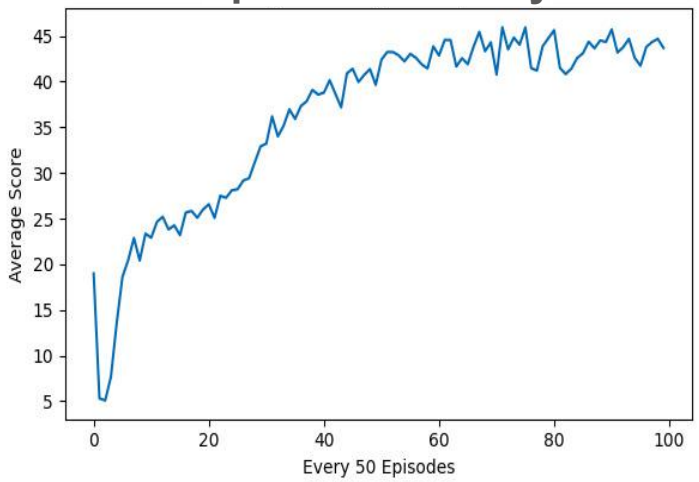
# 3.3 Learning efficiency and Stability

Epsilon-Greedy clearly outperforms Boltzmann exploration in terms of learning efficiency. The Epsilon agent shows a consistently smooth and steep learning curve, reaching key milestones much faster and more reliably than its Boltzmann counterpart. Epsilon reaches an average score of 20 within the first 500 episodes, a score that Boltzmann fails to achieve even after the full 5000 episodes of training.

At the 1000th episode, Epsilon averages a score of 25, whereas Boltzmann lags behind with an average score of only 15. This performance gap only increases with time. By the end of training, the Epsilon-Greedy version achieves an average score of approximately 45, while Boltzmann struggles to surpass an average of 18. Which shows not only faster learning but also superior overall capacity for performance.

Additionally, Boltzmann exploration exhibits unstable and erratic learning behavior, with pronounced drops in performance at various points, something that is largely absent in the Epsilon-Greedy approach. Epsilon's learning curve remains comparatively stable and steadily increasing, indicating more reliable convergence.

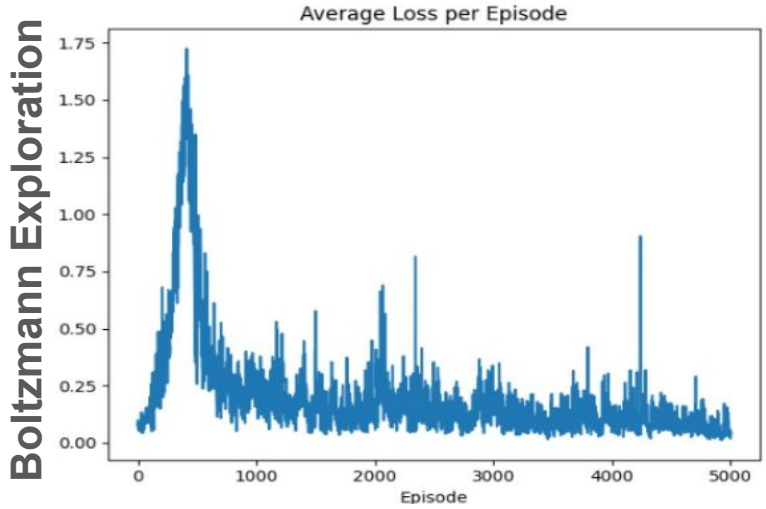


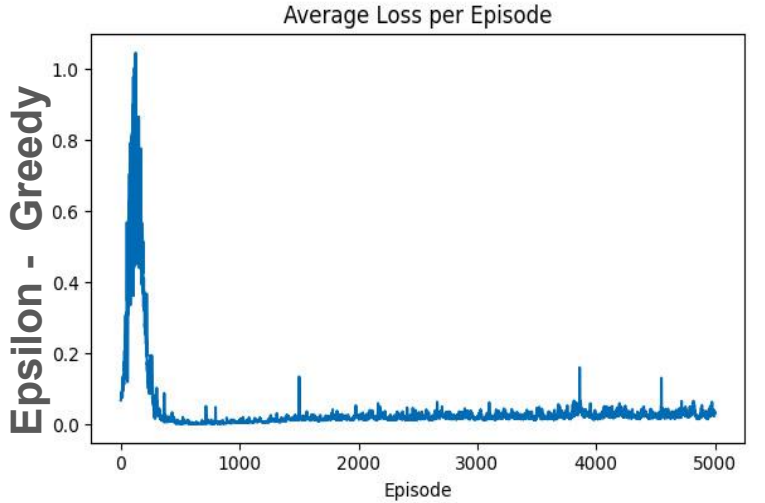



# 3.3 Learning efficiency and Stability

This difference in stability is further corroborated by the loss plots of both versions during training. The Boltzmann version shows frequent and severe spikes in the loss function, reflecting instability in the target network updates and inconsistent learning steps.

In contrast, the Epsilon-Greedy agent maintains lower and more stable loss values, suggesting more controlled and coherent updates to its value function. These loss spikes in Boltzmann reinforce the observed inconsistency in its learning curve and final performance, cementing the conclusion that, **in the context of our project**, Epsilon-Greedy provided not only faster but also more stable and robust learning.



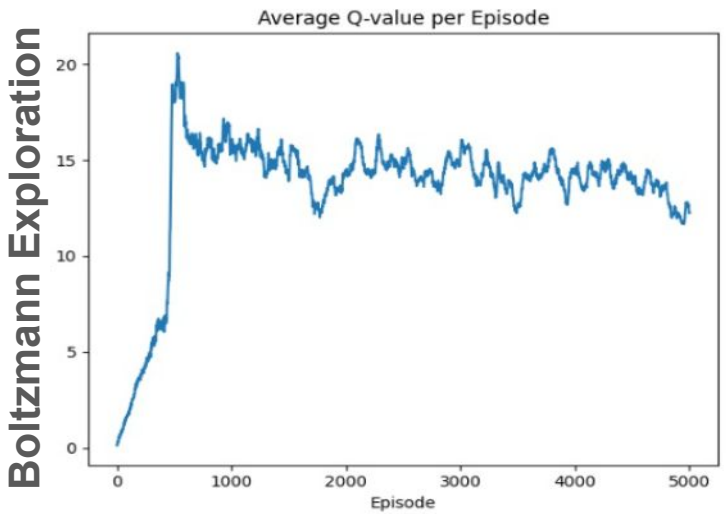



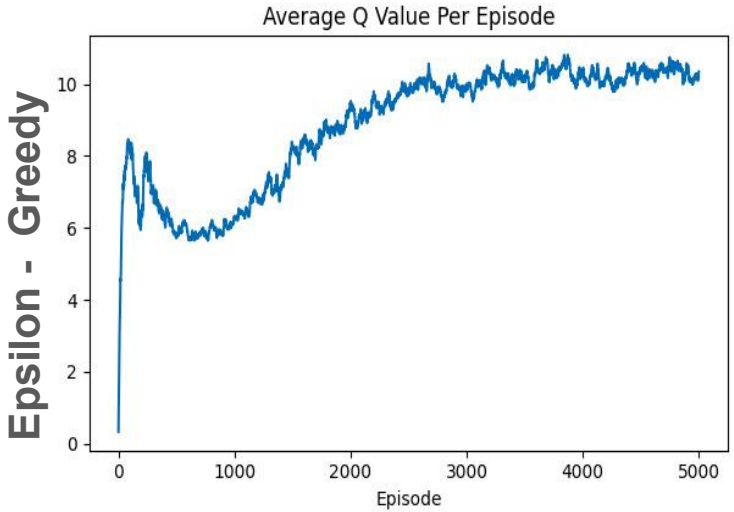
# 3.3 Final Performance

In terms of final performance, the Epsilon-Greedy version remains the superior strategy. By the end of training, it reaches an average score of around 45, more than double Boltzmann's best average of 18, as seen in the learning efficiency and stability slides of this section.

Interestingly, although Boltzmann shows a higher average Q-value throughout most of training (peaking early and averaging around 14 by the end, compared to Epsilon's 11), this does not translate into better performance. Which suggests an overestimation or miscalibration of Q-values in the Boltzmann approach, likely due to its temperature based action selection being bias towards suboptimal choices as the training progresses.

In contrast, Epsilon-Greedy's lower but more reliable Q-values seem better aligned with true state-action values, leading to more effective decisions and a better performance.



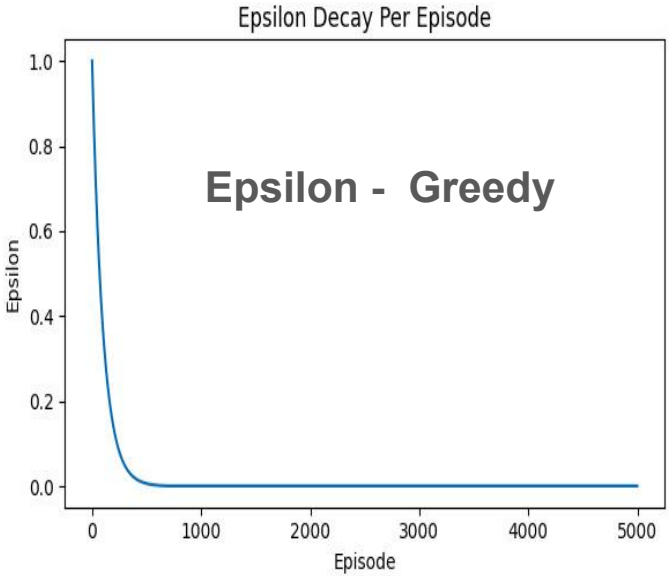



# 3.3 Exploration Behavior and Hyperparameter Sensitivity

In Epsilon-Greedy, a higher epsilon results in more heuristic (random) moves, while a lower one shifts the actions to rely on the network's own Q-Value estimates. We used an initial epsilon of 1.0 with aggressive exponential decay, quickly reducing exploration after the 50 heuristic warm-up games. By approximately 500 episodes, epsilon reached its final value of 0.001, allowing the agent to rely almost entirely on its learned Q-values.

The Epsilon-Greedy strategy showed high sensitivity to the epsilon decay rate. Insufficient decay slowed learning by causing the agent to rely too heavily on heuristic actions, while overly fast decay led to premature exploitation before the network was sufficiently trained.

For the Boltzmann exploration we clearly saw that there wasn't a great difference in scores for the temperatures we tested. However, lower temperatures, due to removing randomness from the training process, ended up leading to more stable Q-values, meaning that the policy the model learns isn't overestimating good results, like with higher temperatures.



# Insights Learned

In this project, we learned that a baseline DQN, even without external guidance, should show slow and gradual improvement, moving from a poor initial policy towards a competent one over time. This learning process can be significantly accelerated and stabilized with external help, such as replay buffers and heuristic players.

We also discovered that a well designed heuristic policy can greatly reduce the randomness of early training data, providing more meaningful and structured experiences that lead to a faster and more stable convergence in DQN training.

A critical insight we gained relates to the exploration vs exploitation trade-off. The choice of epsilon or temperature values, and the decay schedule, played a crucial role in our training, especially when heuristics or replay buffers were used. In the early stages of training, high exploration is important because the network lacks knowledge and confidence to make reliable decisions on its own. As the network learns and its internal representation of the environment improves, reducing exploration and increasing exploitation becomes essential to refine and maximize performance. Balancing this transition carefully is key to effective DQN training.

# What Went Wrong

- We spent a lot of time trying to figure out why priority buffer did not offer a meaningful increase in performance, due to our preconceived notion that it should be much better, even though in our case wasn’t.

# What Went Great

- We implemented a very reliable and effective heuristic player that not only guided the snake towards food efficiently but also avoided collisions, significantly improving the quality of early training data and accelerating the network’s initial learning phase
- We established a strong baseline model early, providing a clear reference for improvements
- We successfully implemented and experimented two types of replay buffers
- We gained a solid understanding of the exploration vs. exploitation balance and how to adjust it during DQN training