AI Assignment 2 Report - Antonio Luque Molina

Task 1 – CartPole – V1

1. Why do we need a policy net and target net?

In Deep Q-Networks (DQN), we use both a policy net and a target net to stabilize learning.

The policy network is used to evaluate the current policy, deciding on the action to take given the current state.

The target network, on the other hand, is used to provide a stable target for the policy network's updates.

In the CartPole-v1 environment, where the agent aims to balance a pole on a cart by applying forces left or right, the distinction between the policy net and target net helps in the following ways:

- The policy net continuously learns and adapts to the task by experiencing the environment (trying to balance the pole), while the target net provides a stable baseline to compare against when updating Q-values.
- Since the environment is relatively simple, the advantage of separating the policy net from the target net might not be as pronounced as in more complex environments. However, it still helps prevent the feedback loop where the Q-values might otherwise chase a moving target, leading to poor convergence.

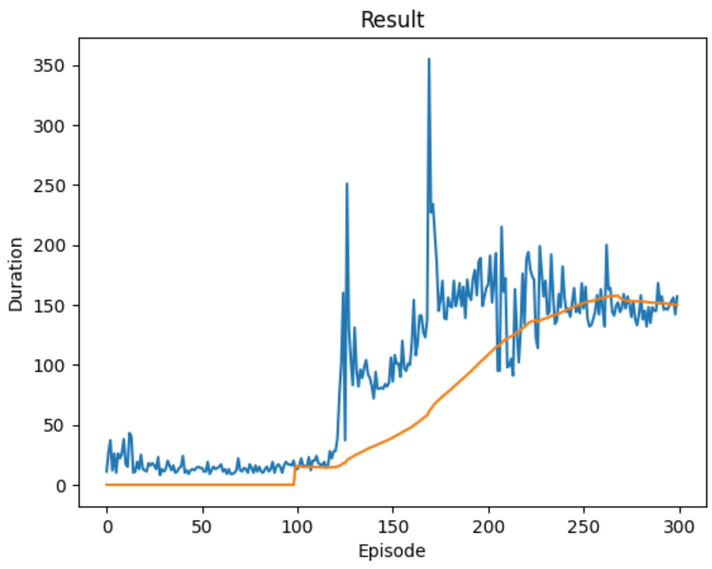
2. When do you think a model converges?

A model in reinforcement learning is considered to have converged when the learning process results in a stable policy, and further training does not significantly change the policy or the expected return from it.

For the CartPole-v1 environment, convergence might be indicated by a Consistent Performance, Stable Q-Values and an Optimal Policy.

3. Observe the effects of changing the exploration vs exploitation. (Show results in plots or print statements)

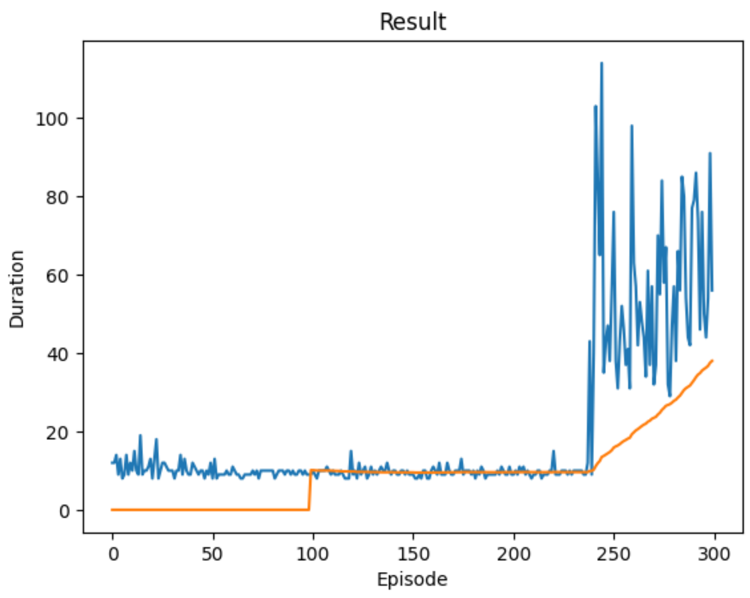
```
EPS_START = 0.9
EPS_END = 0.05
EPS_DECAY = 1000
```



In this graph, the duration of episodes increases over time, showing that the agent is learning, with occasional peaks indicating successful episodes. However, the exploration is still quite high even after many episodes, which can be both good for finding new strategies and bad if it prevents the agent from exploiting the best strategy it has found.

Now, We try a more conservative approach to exploration, changing the EPS_START to 0.5 and We will try a slower decay, allowing more exploration time.

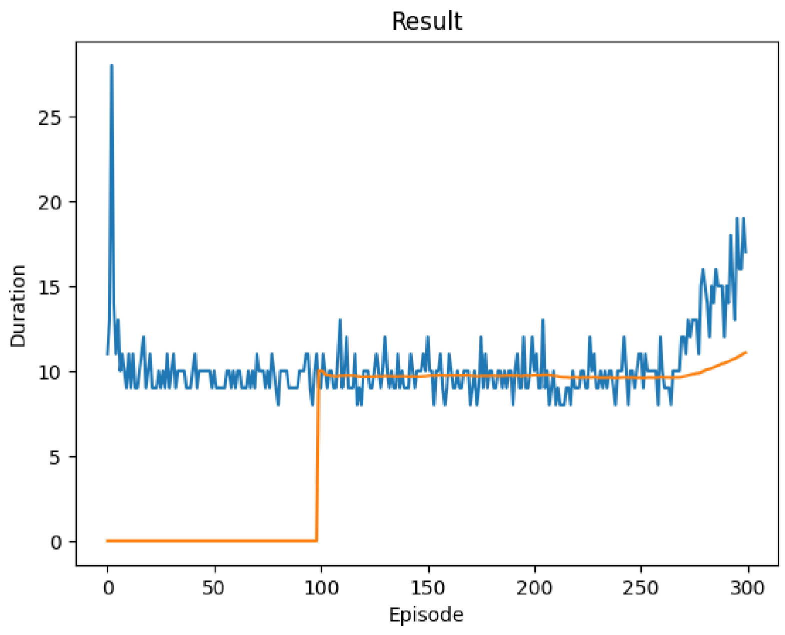
```
EPS_START = 0.5
EPS_END = 0.05
EPS_DECAY = 200
```



This graph shows a more aggressive approach to reducing exploration, as the starting epsilon is lower and it decays faster. The learning seems to progress faster initially, but then it hits a plateau. This could indicate that the agent doesn't explore enough towards the end and may get stuck on suboptimal policies.

After that We are going to try a fast Decay that allow us to quickly move from exploring to exploiting the learned values. (EPS_START = 0.9). Faster Decay, less time spent exploring (EPS_DECAY)

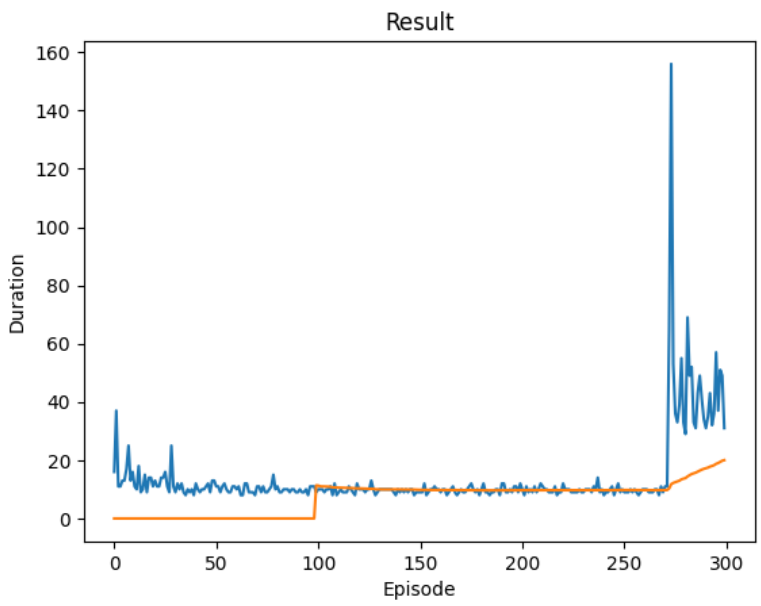
```
EPS_START = 0.9
EPS_END = 0.05
EPS_DECAY = 100
```



This graph shows that the exploration is reduced very quickly, as shown by the low EPS_DECAY value. The agent does not get much opportunity to explore different strategies, which might be why the duration does not improve much over time. It could be getting stuck in local optima due to insufficient exploration.

So now We are going to try a Slow Decay of Exploration: Longer time exploring before settling into exploitation. Slower decay, more time spent exploring(EPS_DECAY)

```
EPS_START = 0.9
EPS_END = 0.05
EPS_DECAY = 300
```



This setting offers a middle ground between the first and third scenarios. There's a gradual improvement in duration, but it's less consistent than in the first scenario. The peaks in duration are higher, suggesting the agent occasionally discovers very good strategies but may not consistently exploit them.

These graphs indicate that the balance between exploration and exploitation significantly affects the learning performance of the agent. Too much exploration can prevent the agent from exploiting known good strategies, while too little can prevent the agent from discovering them in the first place. The optimal policy seems to be one that allows for a gradual reduction in exploration, giving the agent enough time to exploit its knowledge while still looking for better solutions.

4. Vary the number of layers in the model and plot episode vs duration plots. Maybe good to keep an eye on how long!

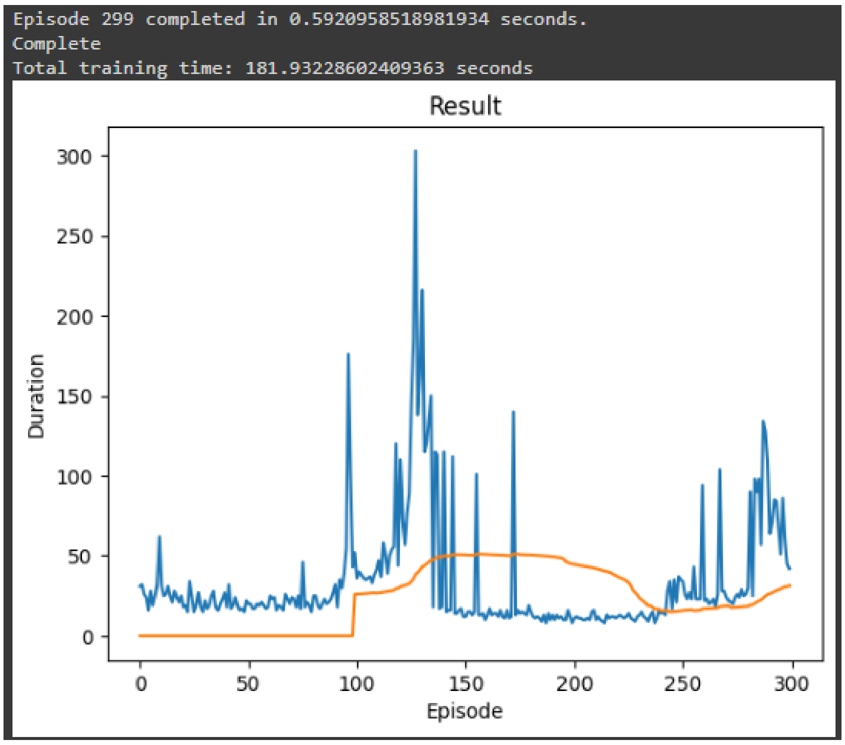
In order to add 1 more layer (4 Layers) I changed in the code the following:

```
class DQN_4Layer(nn.Module):


    def __init__(self, n_observations, n_actions):
        super(DQN_4Layer, self).__init__()
        self.layer1 = nn.Linear(n_observations, 128)
        self.layer2 = nn.Linear(128, 128)
        self.layer3 = nn.Linear(128, 128)
        self.layer4 = nn.Linear(128, n_actions)


    def forward(self, x):
        x = F.relu(self.layer1(x))
        x = F.relu(self.layer2(x))
        x = F.relu(self.layer3(x))
policy_net = DQN_4Layer(n_observations, n_actions).to(device)
target_net = DQN_4Layer(n_observations, n_actions).to(device)
```

and I also added time measurement, It can be seen in the google colab notebook. It returns the following:



```
Episode 299 completed in 0.5920958518981934 seconds.
Complete
Total training time: 181.93228602409363 seconds
```

Adding an extra layer seems to have introduced more variability in the learning progress. There are significant peaks which suggest that at times the model performs very well, but there are also dips indicating inconsistency. This could be due to overfitting where the model is learning to perform exceptionally well in specific states but fails to generalize across the board.

In order to add 2 more layer (5 Layers) I changed in the code the following:

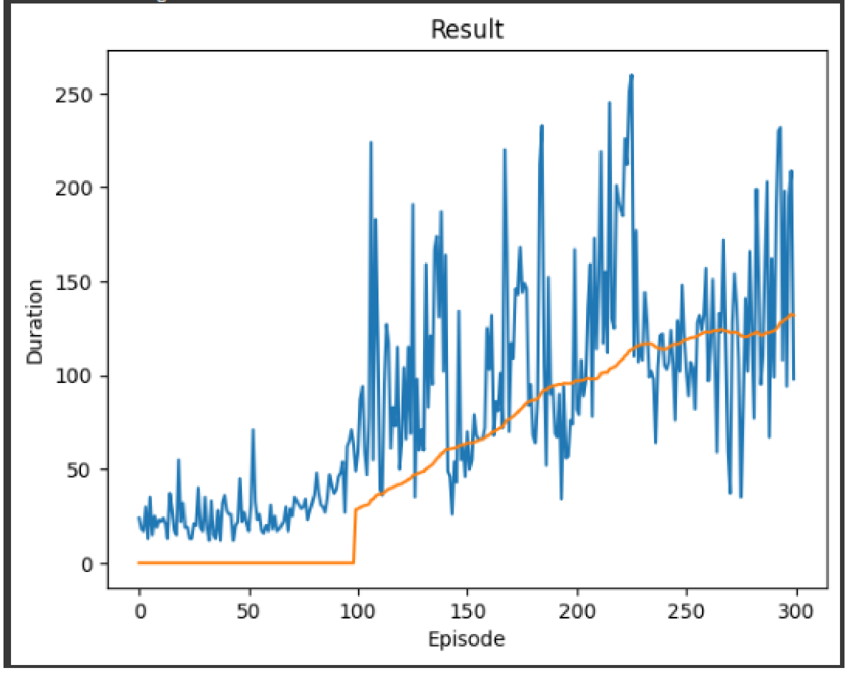
```
class DQN_5Layer(nn.Module):

    def __init__(self, n_observations, n_actions):
        super(DQN_5Layer, self).__init__()
        self.layer1 = nn.Linear(n_observations, 128)
        self.layer2 = nn.Linear(128, 128)
        self.layer3 = nn.Linear(128, 128)
        self.layer4 = nn.Linear(128, 128)
        self.layer5 = nn.Linear(128, n_actions)

    def forward(self, x):
        x = F.relu(self.layer1(x))
        x = F.relu(self.layer2(x))
        x = F.relu(self.layer3(x))
        x = F.relu(self.layer4(x))
        return self.layer5(x)
policy_net = DQN_5Layer(n_observations, n_actions).to(device)
target_net = DQN_5Layer(n_observations, n_actions).to(device)
```

and I also added time measurement, It can be seen in the google colab notebook. It returns the following:

```
Episode 299 completed in 1.6951797008514404 seconds.
Complete
Total training time: 341.521475315094 seconds
```

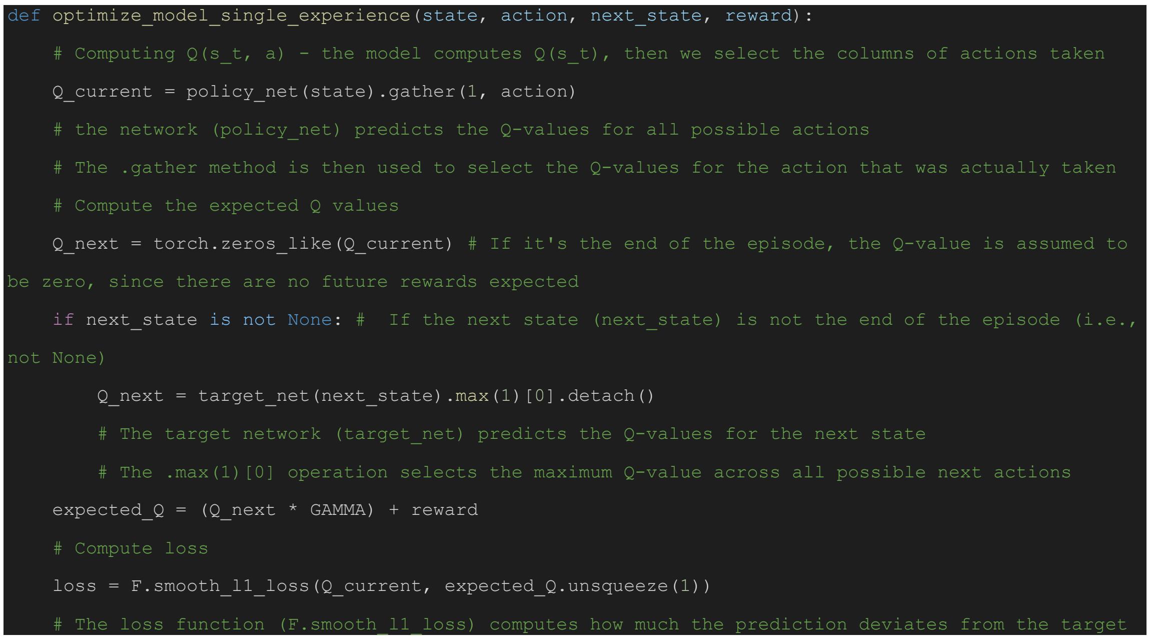


Adding 2 layers more shows even more variability and some of the highest peaks, suggesting moments of excellent performance. However, similar to the four-layer model, the performance is not consistent, and there are episodes of low performance as well. This could again be indicative of overfitting or complex strategy formation which only works in specific circumstances.

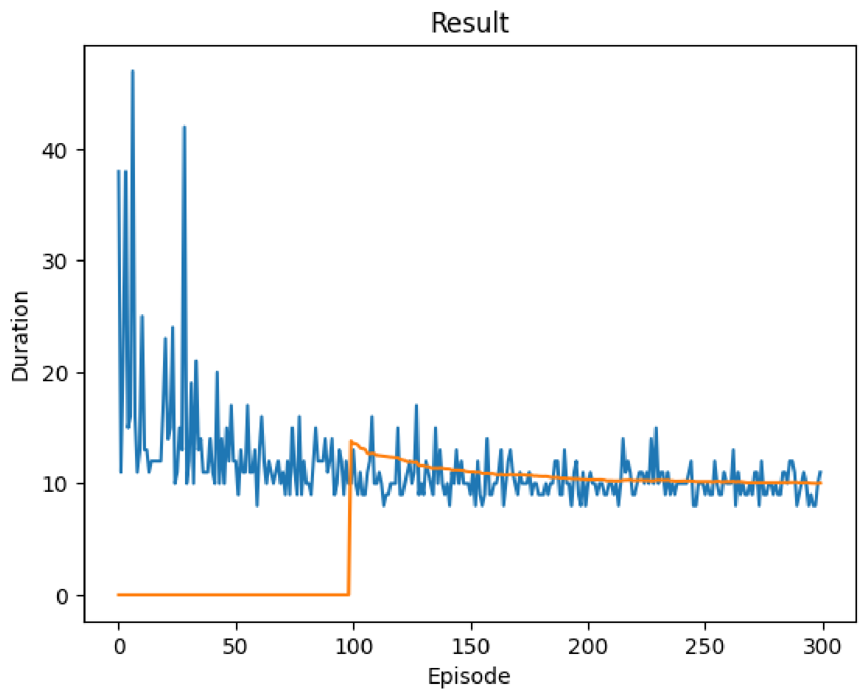
If we are looking for the highest peaks (potentially the best performance in some episodes), a more complex model might be chosen. If we are looking for consistency and steady learning, a simpler model might be preferable.

5. Implement without the replay buffer and observe performance.

For that I changed in the code the following:



```
def optimize_model_single_experience(state, action, next_state, reward):
    # Computing Q(s_t, a) - the model computes Q(s_t), then we select the columns of actions taken
    Q_current = policy_net(state).gather(1, action)
    # the network (policy_net) predicts the Q-values for all possible actions
    # The .gather method is then used to select the Q-values for the action that was actually taken
    # Compute the expected Q values
    Q_next = torch.zeros_like(Q_current) # If it's the end of the episode, the Q-value is assumed to
be zero, since there are no future rewards expected
    if next_state is not None: #  If the next state (next_state) is not the end of the episode (i.e.,
not None)
        Q_next = target_net(next_state).max(1)[0].detach()
        # The target network (target_net) predicts the Q-values for the next state
        # The .max(1)[0] operation selects the maximum Q-value across all possible next actions
    expected_Q = (Q_next * GAMMA) + reward
    # Compute loss
    loss = F.smooth_l1_loss(Q_current, expected_Q.unsqueeze(1))
    # The loss function (F.smooth_l1_loss) computes how much the prediction deviates from the target
    # Optimize the model
    optimizer.zero_grad() # .zero_grad() is called to clear any existing gradients
    loss.backward() # The loss is backpropagated through the network by calling .backward() on it
    for param in policy_net.parameters():
        param.grad.data.clamp_(-1, 1) # he clamp_ operation is used to prevent the gradients from
getting too large, which can help stabilize training
    optimizer.step() # After backpropagation, .step() is called on the optimizer to actually perform
the weight update
# Perform one step of the optimization (on the policy network) using the single experience
    optimize_model_single_experience(state, action, next_state, reward)
```
and I also added time measurement, It can be seen in the google colab notebook. It returns the following:



In this graph that shows the performance of the ChartPole implemented without the Replay Buffer we see:

- The duration per episode is generally low, indicating that the agent often fails to keep the pole balanced for long periods.
- The learning curve is relatively flat, with few significant increases in episode duration.
- There is significant variability in episode durations, but no sustained or consistent improvement over time.

So, It suggests that the use of a replay buffer contributes to better learning outcomes in this case. The agent not only achieves higher durations, indicative of better performance, but also shows a more stable learning progression over time. The replay buffer seems to contribute to both the stability and efficiency of the learning process.

Task 2 – CartPole2DEnv

1. Why do we need a policy net and target net?

In Deep Q-Networks (DQN), we use both a policy net and a target net to stabilize learning.

The policy network is used to evaluate the current policy, deciding on the action to take given the current state.

The target network, on the other hand, is used to provide a stable target for the policy network's updates.

For a 2D version of the CartPole environment (CartPole-2DEnv), which would naturally have a more complex state space and potentially additional actions, the separation between the policy net and target net becomes even more critical:

- The additional complexity means there's a greater risk of unstable learning due to the increased number of states and the intricacies of balancing the pole in a 2D plane.
- The target net acts as an anchor, providing stability in an environment where the agent must learn a policy that generalizes across a more diverse set of states and possibly actions (moving in a 2D plane).

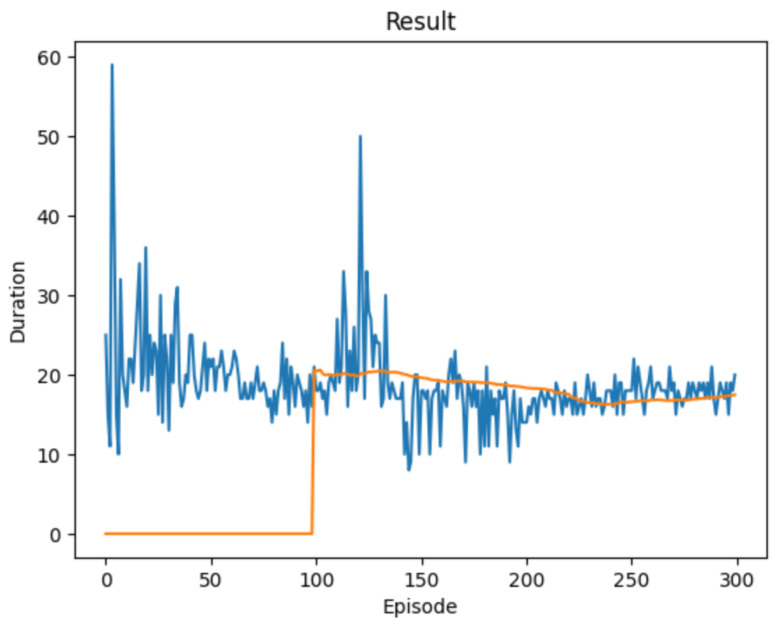
2. When do you think a model converges?

A model in reinforcement learning is considered to have converged when the learning process results in a stable policy, and further training does not significantly change the policy or the expected return from it.

Convergence in a CartPole-2Denv may be indicated by: Robust Performance, Diminished Returns from Training and Policy Stability.

3. Observe the effects of changing the exploration vs exploitation. (Show results in plots or print statements)

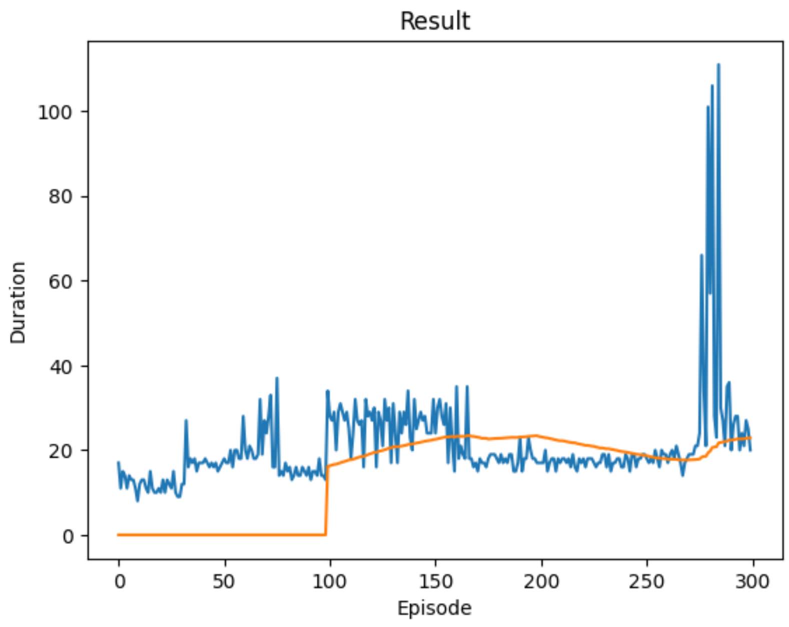
```
EPS_START = 0.9
EPS_END = 0.05
EPS_DECAY = 1000
```



This graph has a high starting exploration rate which decays slowly over a large number of episodes, indicating a gradual shift from exploration to exploitation. The results show some stability in the duration towards the end, suggesting the agent has learned a relatively stable policy over time.

Now, We try a more conservative approach to exploration, changing the EPS_START to 0.5 and We will try a slower decay, allowing more exploration time.

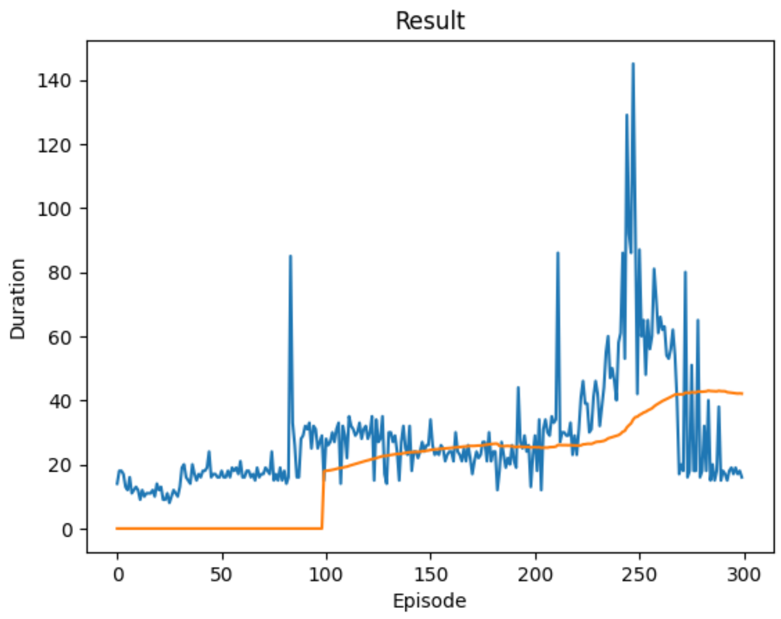
```
EPS_START = 0.5
EPS_END = 0.05
EPS_DECAY = 200
```



In this graph, with a lower starting exploration rate and a rapid decay to exploitation, the agent starts exploiting its knowledge sooner. This could lead to faster learning but might also risk getting stuck in suboptimal policies if the exploration is insufficient.

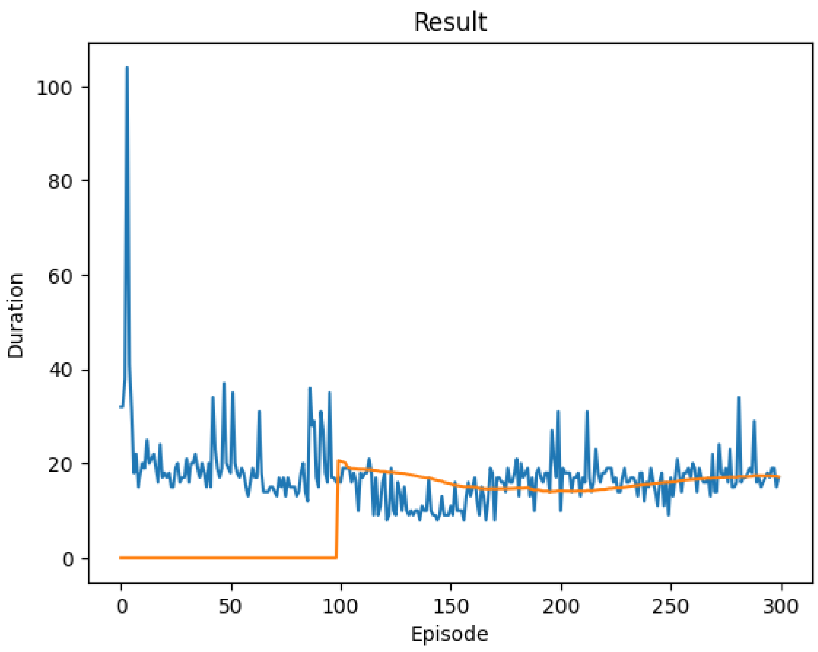
After that We are going to try a fast Decay that allow us to quickly move from exploring to exploiting the learned values. (EPS_START = 0.9). Faster Decay, less time spent exploring (EPS_DECAY)

```
EPS_START = 0.9
EPS_END = 0.05
EPS_DECAY = 100
```



This graph shows a high initial exploration like the first scenario, but with a much quicker decay. The results might show more erratic learning with possibly higher peaks (if the agent discovers better strategies) but less stability due to the rapid shift to exploitation.So now We are going to try a Slow Decay of Exploration: Longer time exploring before settling into exploitation. Slower decay, more time spent exploring(EPS_DECAY)

```
EPS_START = 0.9
EPS_END = 0.05
EPS_DECAY = 300
```

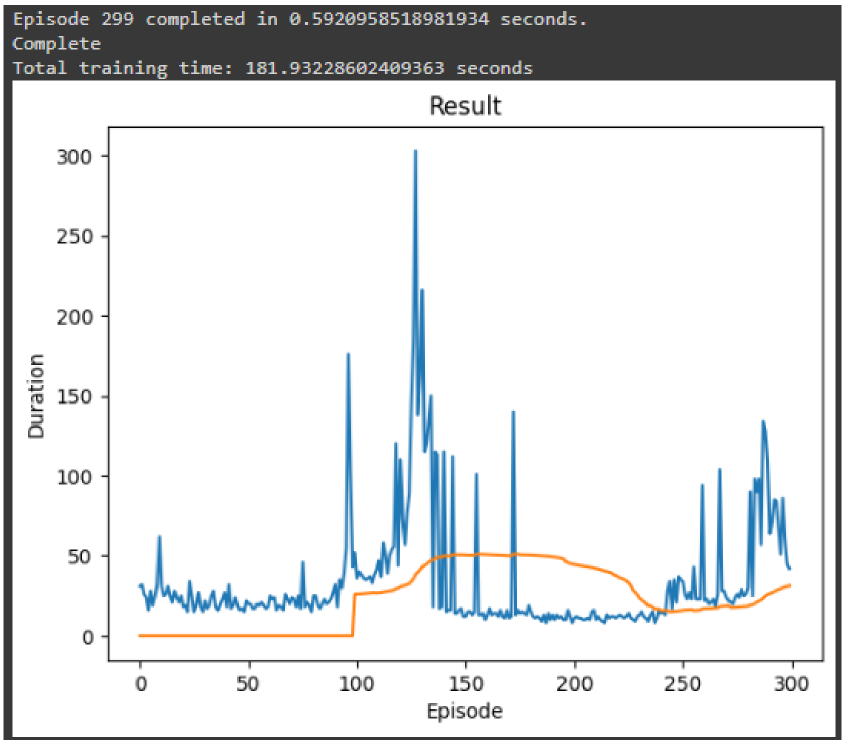


This graph offers a middle ground between the first and third scenarios, same as CartPoleV1. The learning curve may not reach as high as with a slower decay, but it can potentially offer a more consistent improvement over time.

In conclusion, a balance that allows sufficient exploration before exploitation seems to yield a smoother and steadily improving learning curve. Rapid decay rates can result in unstable learning, as the agent may not explore the environment enough to learn a robust policy. Conversely, very slow decay rates may cause the agent to explore excessively without adequately converging on an optimal policy.

4. Vary the number of layers in the model and plot episode vs duration plots. Maybe good to keep an eye on how long!

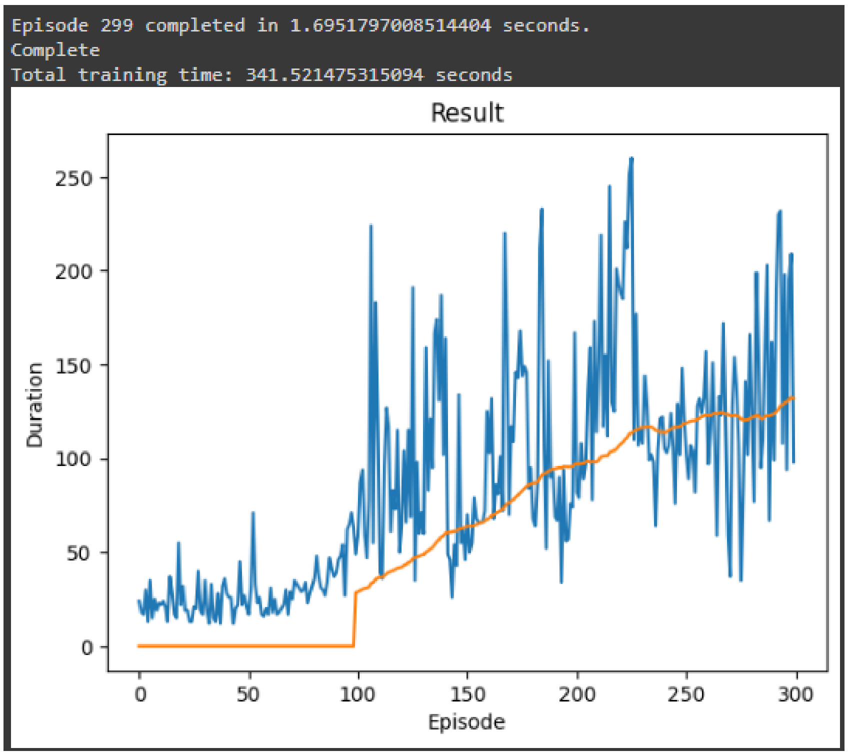
In order to add 1 more layer (4 Layers) I changed in the code the same as I changed in the first task. I also added time measurement, It can be seen in the google colab notebook. It returns the following:



```
Episode 299 completed in 0.5920958518981934 seconds.
Complete
Total training time: 181.93228602409363 seconds
```

This plot indicates more variability, suggesting that the added complexity may not have been beneficial.

In order to add 2 more layer (5 Layers) I changed in the code the same as I changed in the first task. I also added time measurement, It can be seen in the google colab notebook. It returns the following:



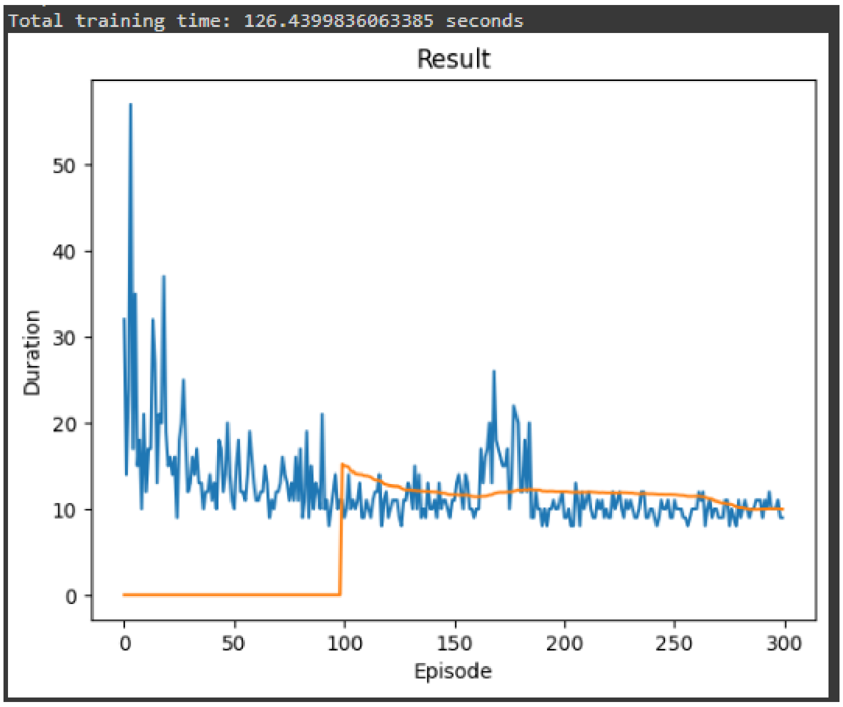
```
Episode 299 completed in 1.6951797008514404 seconds.
Complete
Total training time: 341.521475315094 seconds
```

Adding 2 layers more shows shows that additional complexity further increases the variability and possibly overfits, as it doesn't seem to stabilize even late in training.

In these scenarios, with more layers in the neural network, it appears that there's a tendency for the learning to improve as seen by an increase in the average duration the pole stays upright. However, with more layers, there's also more variability in performance. This could indicate that the network is learning a more complex representation of the environment but also might need more training episodes to stabilize due to increased parameters and potential for overfitting.

<u>**5. Implement without the replay buffer and observe performance.**</u>

For that I changed in the code the same as I changed in CartPoleV1. I also added time measurement, It can be seen in the google colab notebook. It returns the following:



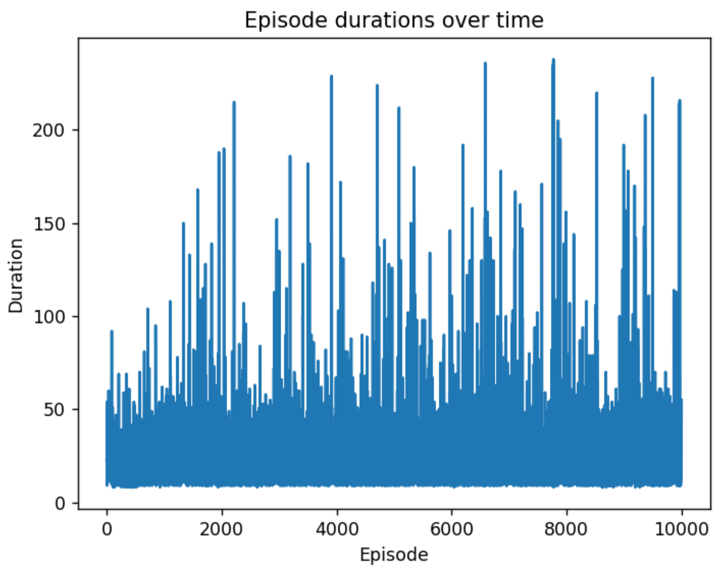
```
Total training time: 126.4399836063385 seconds
```

The inclusion of a replay buffer appears to stabilize the learning process, leading to a smoother improvement in balance duration over episodes, despite potentially increased computational cost. However, it does not necessarily lead to a superior final model performance after the same number of episodes, at least for this specific task and under the same settings for exploration, exploitation, and neural network layers.

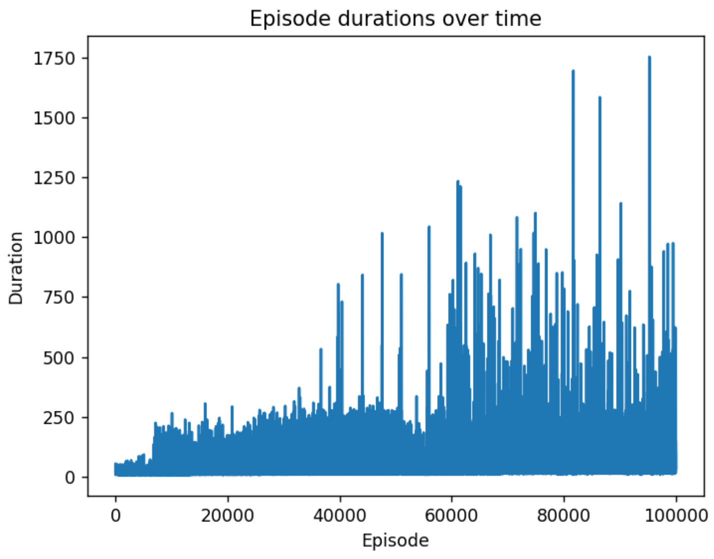
TASK 3 - Q-learning and SARSA for `CartPole-V1` Environment.

I implemented a traditional Q-Learning implementation, I explained it all using LateX on my google colab notebook that is attached with the name: "Antonio_Assignment2_AI_Notebook.ipynb". I also did it in visual studio code and I found out some differences:

In visual studio code I obtained these plots:

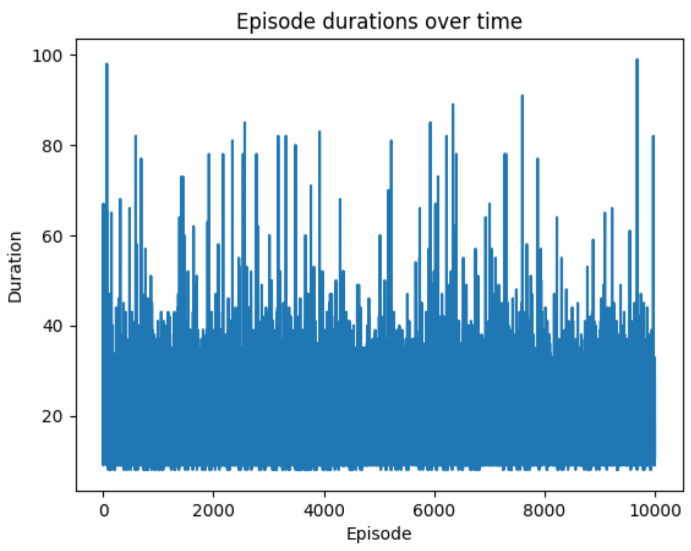


In the first one, runned out with 10000 episodes, We can see that the agent has learned to some extent (as indicated by the peaks), however, there's significant room for improvement in the learning process to achieve more consistent and stable performance.

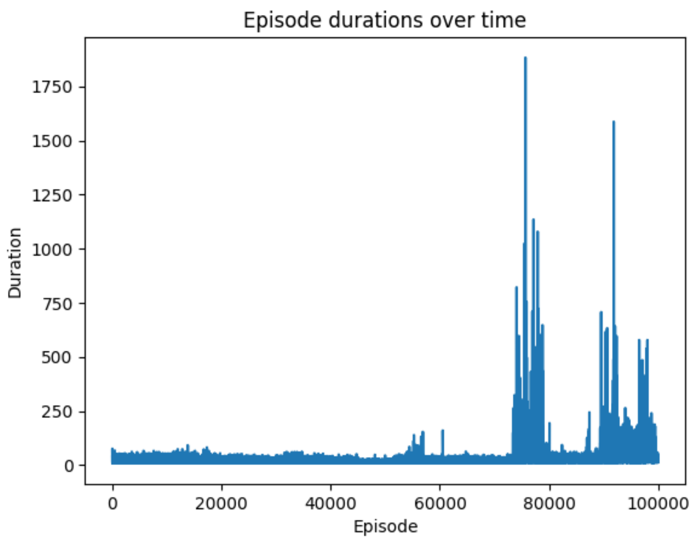


This other one, runned out with 100000 episodes, We can see that the agent is definitively learning as each time it gets better and better.

In the google colab notebook We can see these plots:



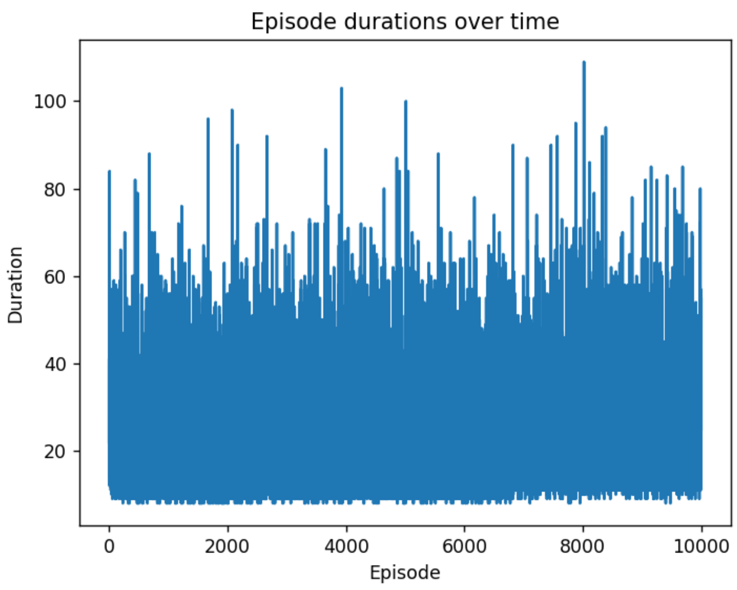
This one is quite similar to the one in visual studio code but with not so impressive peaks.



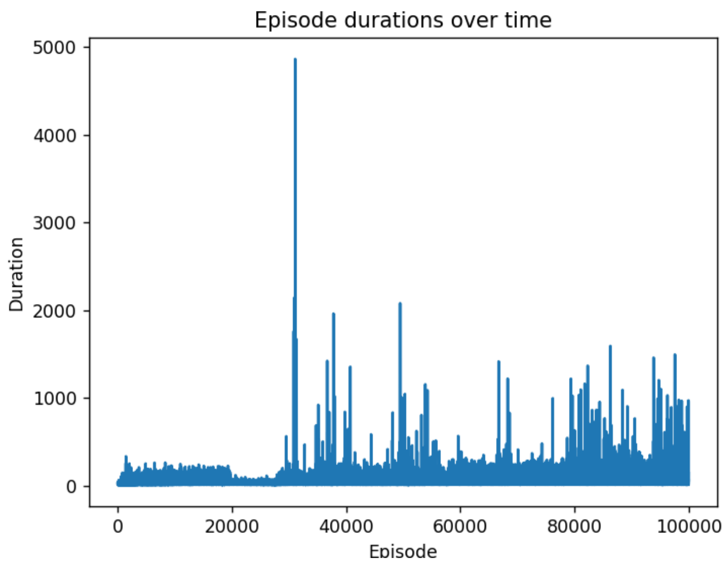
This one is not so similar to the one in visual studio but We can observe that the agent is learning as We saw better using visual studio.

I also did a SARSA implementation, I explained it all using LateX on my google colab notebook that is attached with the name: "Antonio_Assignment2_AI_Notebook.ipynb". I also did it in visual studio code and I found out some differences:

In visual studio code I obtained these plots:

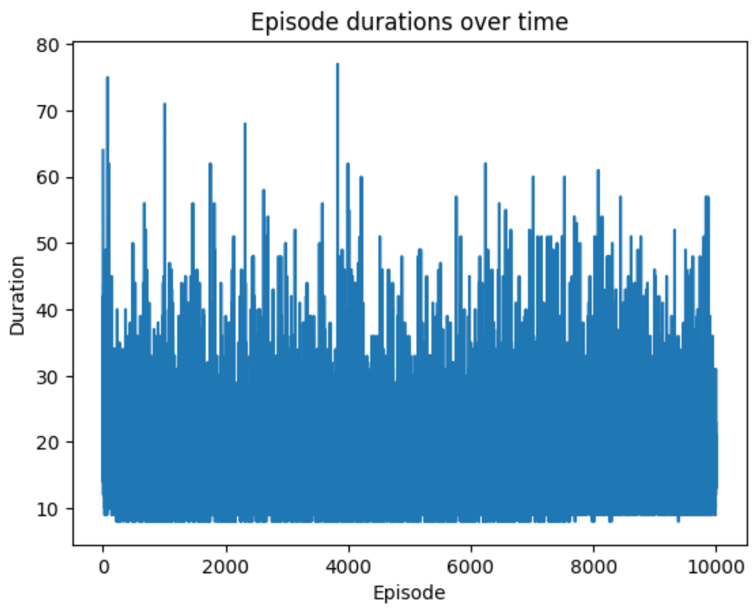


In the first one, runned out with 10000 episodes, We can see that the agent has learned to some extent (as indicated by the peaks), however, there's significant room for improvement in the learning process to achieve more consistent and stable performance as happened with the traditional Q-Learning implementation.

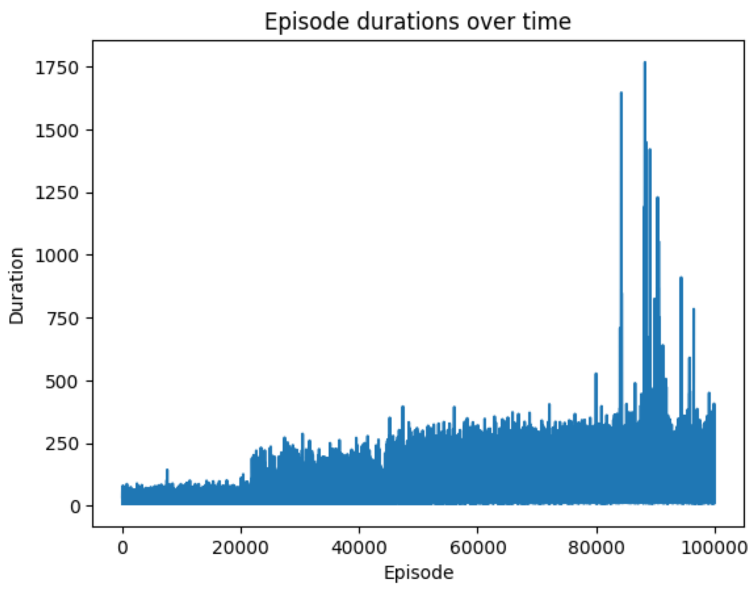


Here, the performance is quite varied and does not show a clear trend towards improvement. This suggests that while the agent learns good policies sporadically, it has not consistently applied an effective strategy across episodes.

In the google collab notebook We can see these plots:



This one is quite similar to the one that We saw in visual studio code.

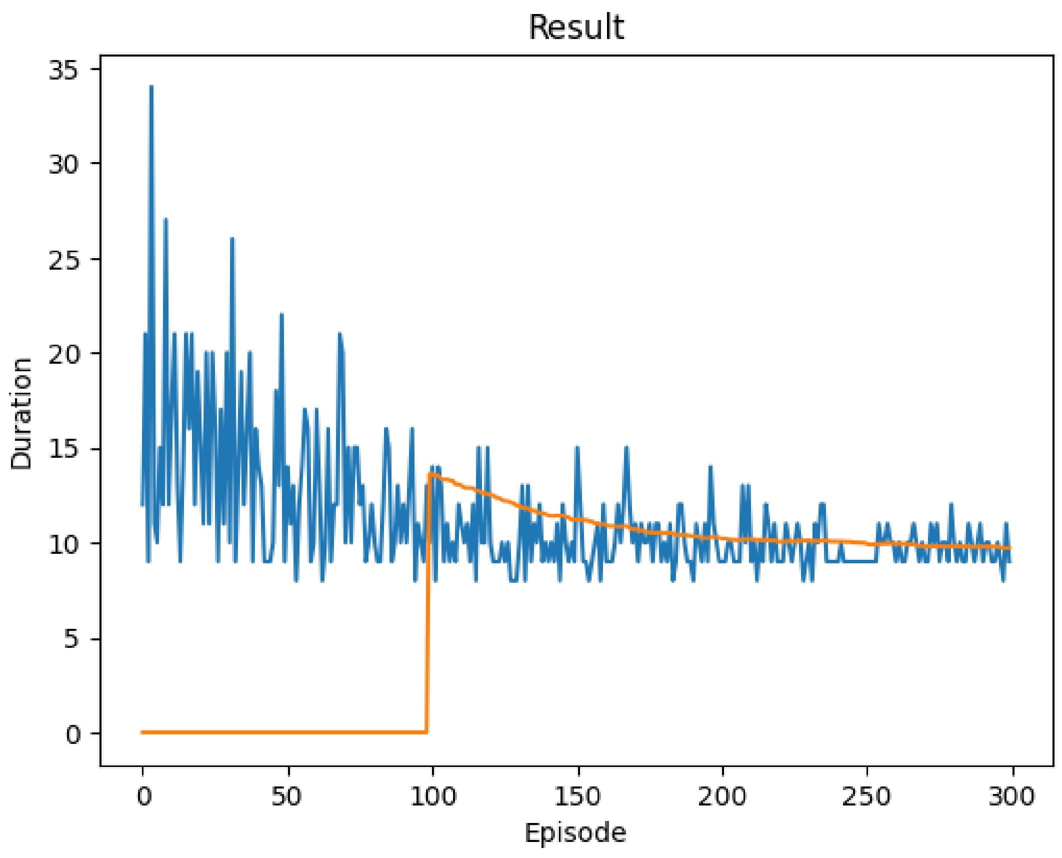


This plot suggests that while the agent does improve and has moments of high performance, it lacks consistency and has not stabilized on an optimal strateg. The performance is highly variable, and despite the training length, there's no clear indication that the agent has fully converged to a reliable policy

BONUS TASK – Implementation of Categorical DQN (C51) Algorithm.

It is all explained and well documented in the google colab notebook and in the files from vscode: Tak3_C51DQN.py.

It returned this plot:



We can see in this plot that the trend of the episode durations is decreasing, particularly noticeable after the initial 50 episodes. This suggest that the agent's policy is not improving over time. In fact, it appears to be deteriorating.

That's all! Thank you for reading!

Antonio.