Task 1

Why do we need a policy net and target net?

The policy net determines the action given in the current state, while the target net generates Q-values for updating the policy net.

The algorithms are looking for the optimal policy with the help of constant feeding of the target network.

When do you think a model converges?

In this case the model converges when the duration is fairly consistent even though the number for episodes changes.

Possible reasons for causing this:

- Proper balance between exploitation and exploration
  - Sufficient exploration makes sure the algorithm exploits optimal policies and converges towards them
- Appropriate learning rage
  - There are multiple strategies for this, such as:
    - fixed: which is manually tuned during training.
    - decaying: starting high and reducing the rate over time
    - adaptive: using algorithms that adjust learning rate based on training progress e.g Adam

Observe the effects of changing the exploration vs. exploitation.

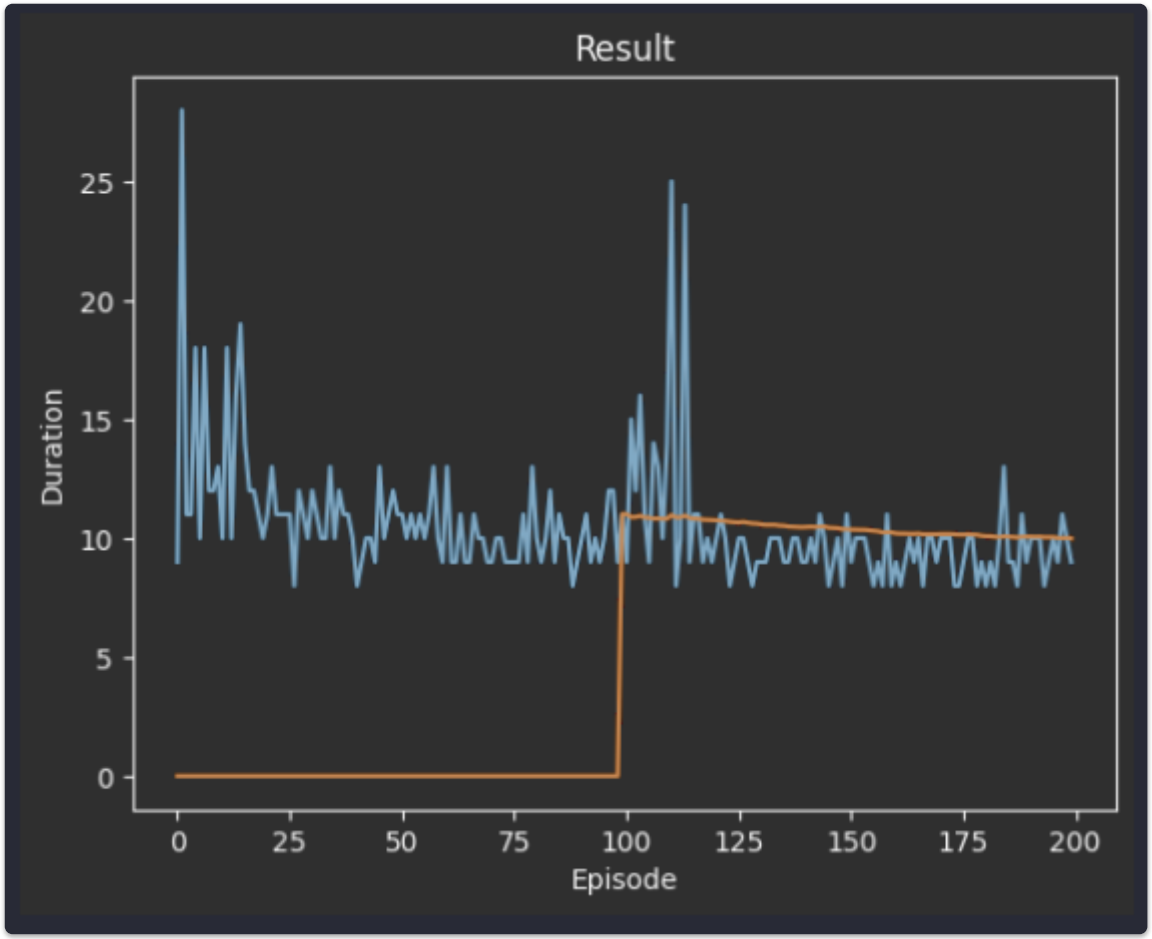
Explore and exploit are measured through the epsilon values.

Modifications:

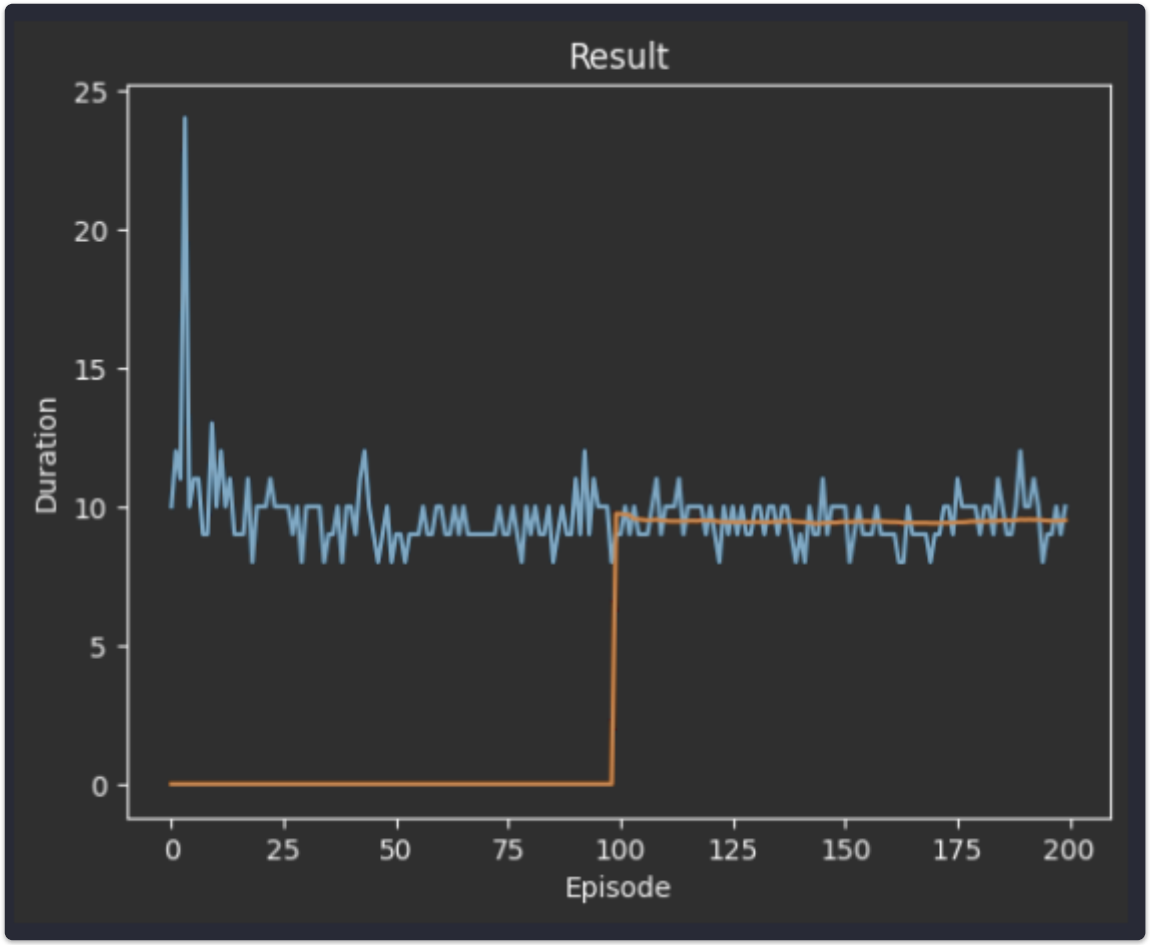
```
# Initial values:
EPS_START = 0.9
```

```
EPS_END = 0.05
EPS_DECAY = 1000

# Increasing exploration:
EPS_START = 0.9
EPS_END = 0.05
EPS_DECAY = 200

# More rapid exploitation:
EPS_START = 0.5
EPS_END = 0.01
EPS_DECAY = 100
```

Exploration:



Exploitation:



Vary the number of layers in the model and plot episode vs. duration plots.

Different layers where tested both greater and less then the original number provided.

```
class DQN(nn.Module):
    # layers endres her med n_nodes
    def __init__(self, n_observations, n_actions):
        n_nodes = 130  # 128, 125,
        super(DQN, self).__init__()
        self.layer1 = nn.Linear(n_observations, n_nodes)
```

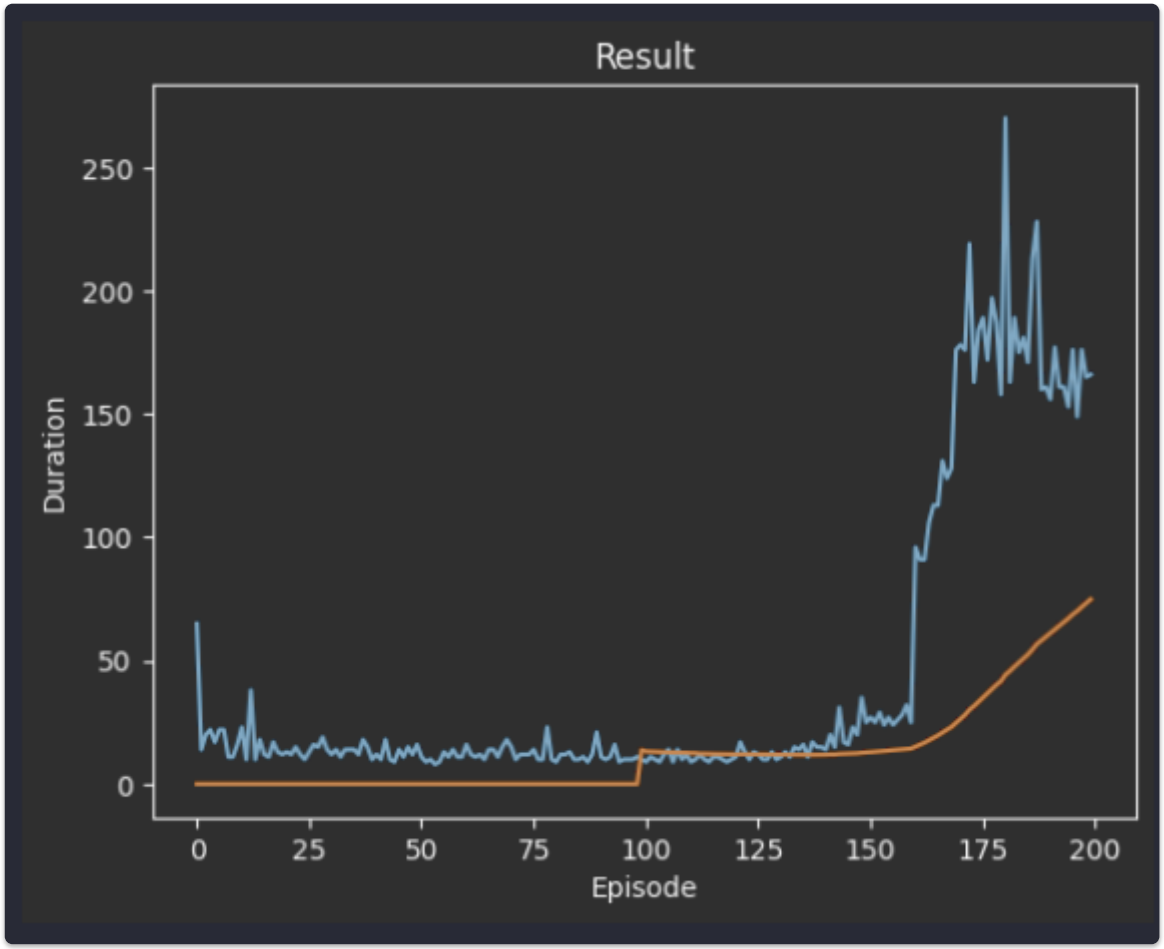
```
        self.layer2 = nn.Linear(n_nodes, n_nodes)
        self.layer3 = nn.Linear(n_nodes, n_actions)
```

Implement without the replay buffer and observe performance.

As we can see from bellow, removing the replay buffer significantly affects the performance. This is caused by:

- slower convergance to optimal policies
- increased variance in learning performance
- potential for overfitting to recent experiences, leading to less generalizable policies

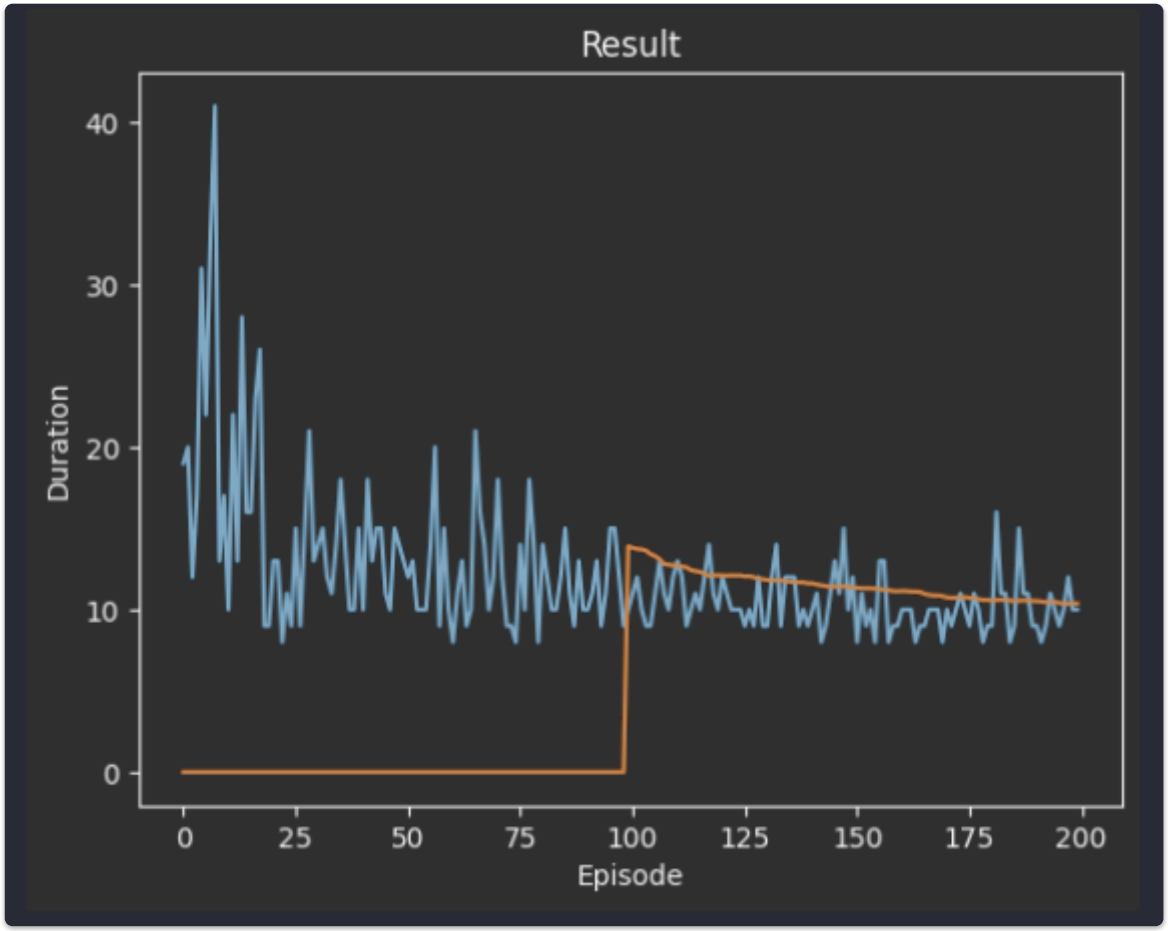
With memory:



Without memory:

Removing the following line from the code:

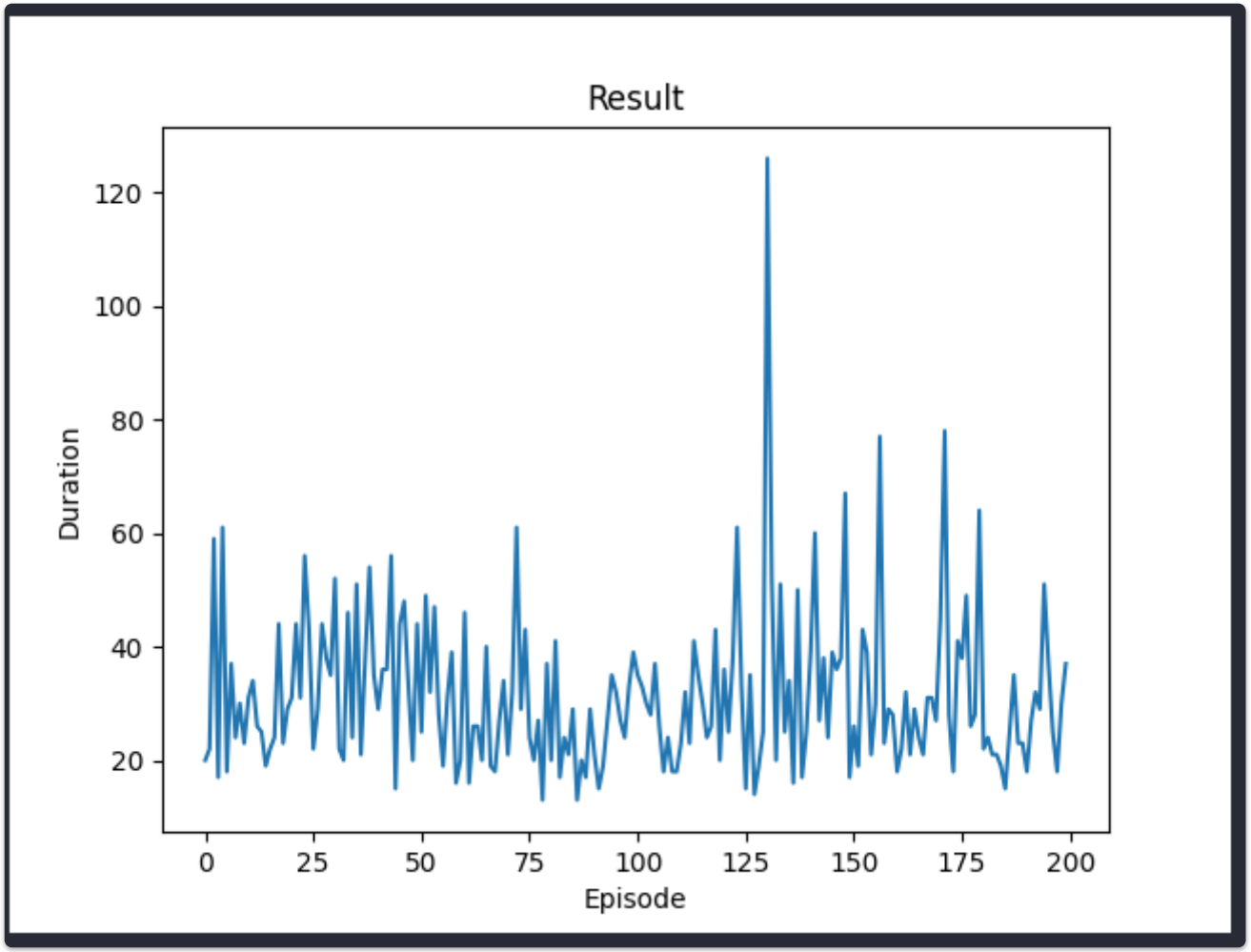
```
memory.push(state, action, next_state, reward)
```



Task 2

First of all we will look at the differences and similarities of the python code and the notebook. They are both tasked in solving the cart-pole balancing problem, with the help of a deep learning model called DQN. This model is implemented with the help of PyTorch. A major difference between the two is that the notebook only operates in a single dimension where as the python program runs in two.

Bellow is the performance of the algorithms with a replaybuffer, standard number of layers and a good balance between explore/exploit. We can se that the replaybuffer had a significantly better impact on the notebook than the python script.



Why do we need a policy net and target net?

```
target_model = DQN(env.observation_space.shape, env.action_space.n)
target_model.load_state_dict(model.state_dict())
target_model.eval()
```

This code helped providing a consistent target during the updating of the Q-value, which resulted in more stable and reliable learning outcomes.

When do you think a model converges?

Among other things this depends on the other subtasks linked to this assignment:

Explore/exlpoit:

- a higher focus on exploit will result in a model convergance, but maybe a bit premature and not necessarily with satisfactory results

Varying model layers:

- increasing the model layers will require more data and could cause overfitting. This will increase the time it takes for the model to converge

Without replay buffer:

- This would lead to learning directly from consecutive samples. It will result in faster convergance but the learning is not as stable due to the correlated nature og sequential observations

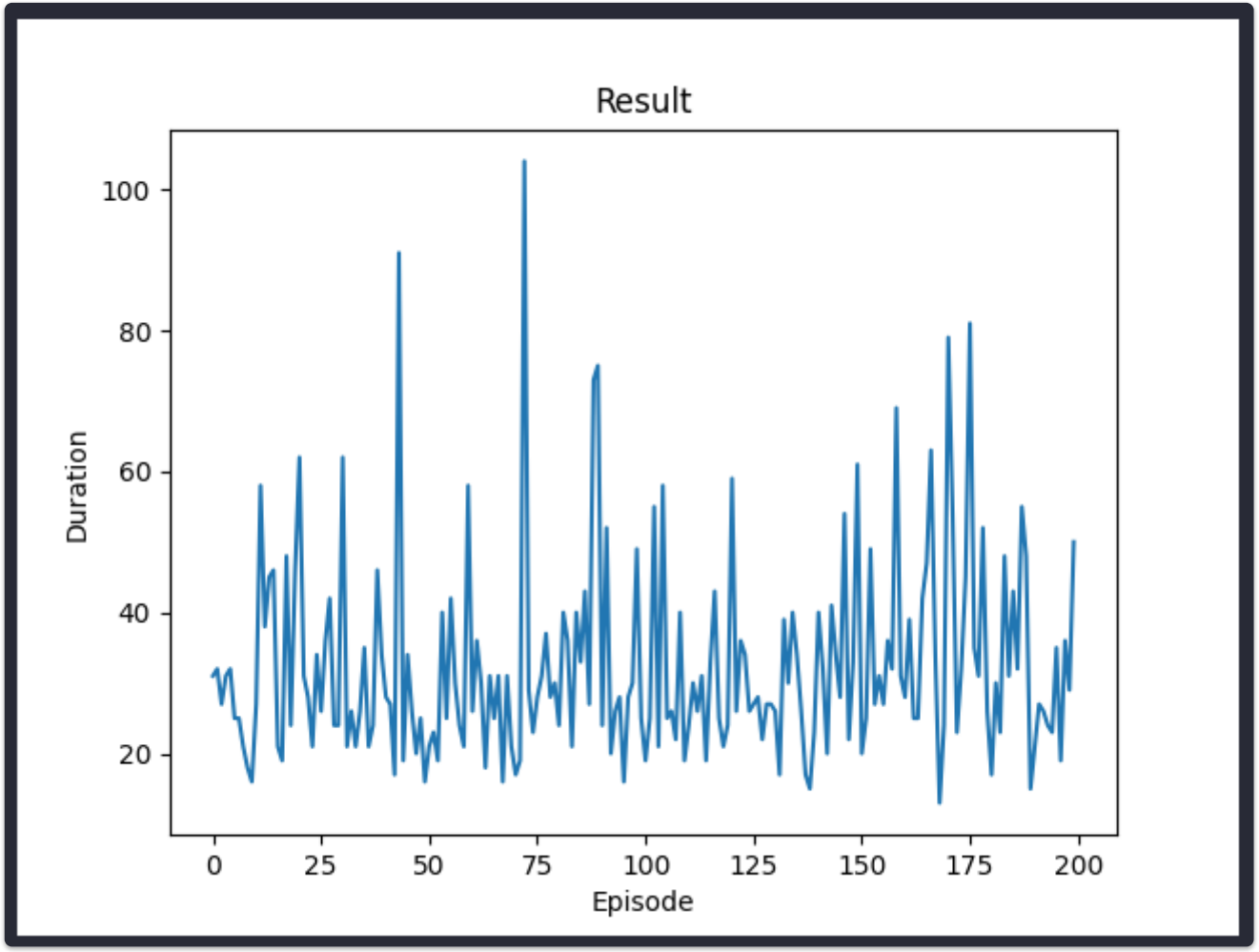
Observe the effects of changing the exploration vs. exploitation.

Similar as for the notebook, the epsilon values where modifies like this:

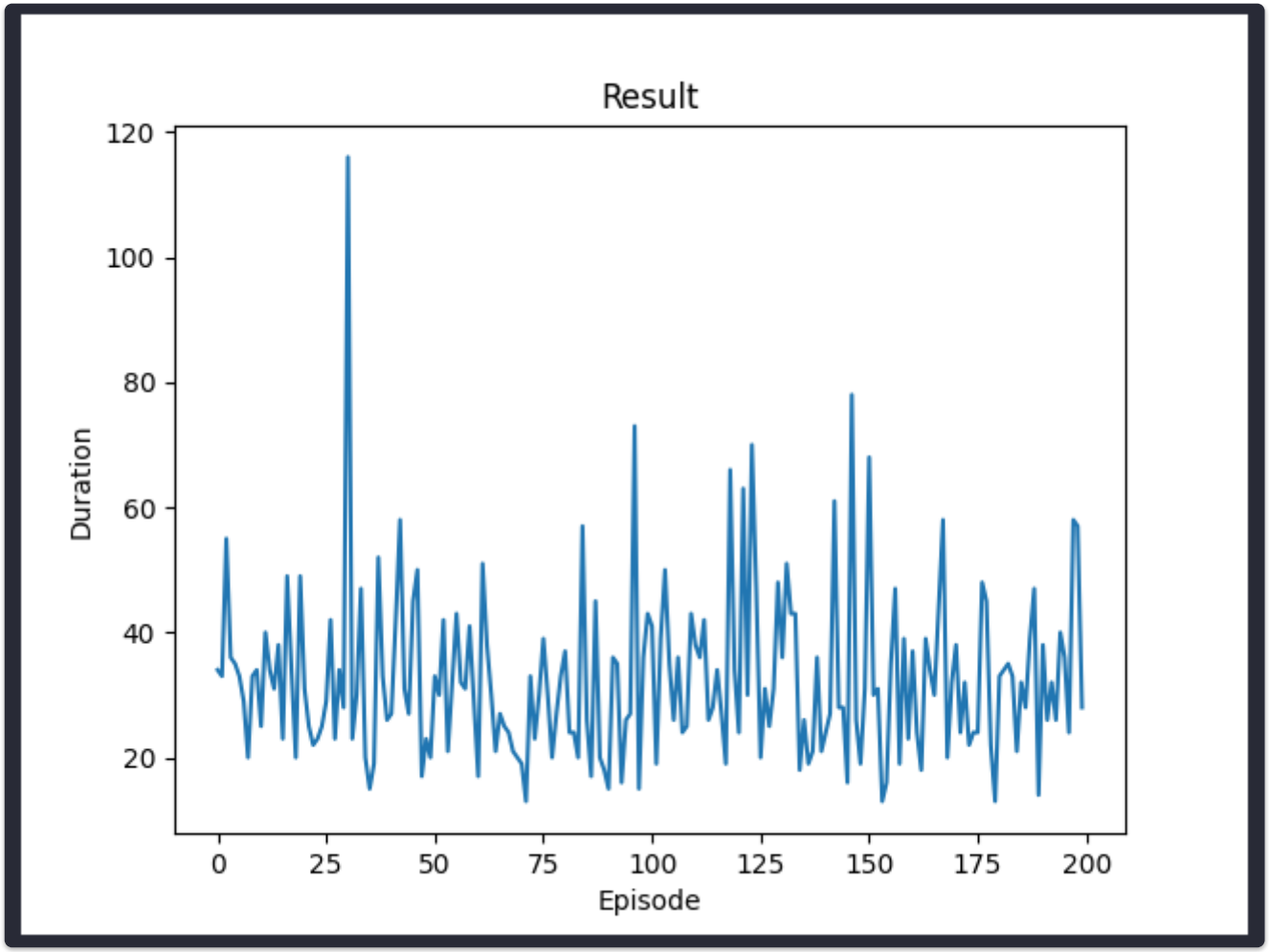
```
# Initial values:
epsilon_start = 0.9
epsilon_end = 0.05
epsilon_decay = 1000

# Increasing exploration:
epsilon_start = 0.9
epsilon_end = 0.05
epsilon_decay = 200

# More rapid exploitation:
```

```
epsilon_start = 0.5
epsilon_end = 0.01
epsilon_decay = 100
```

High exploration:



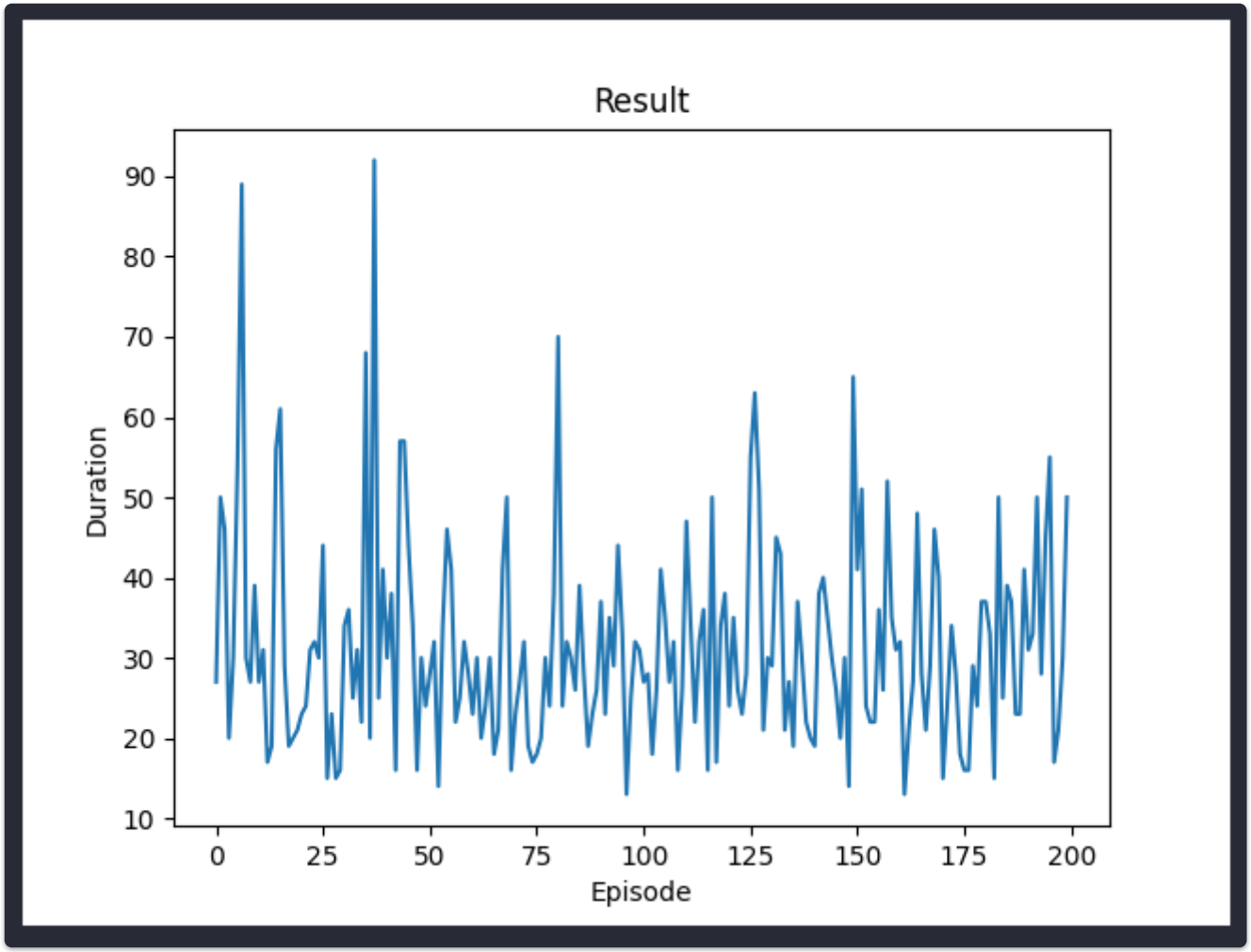
High exploitation:



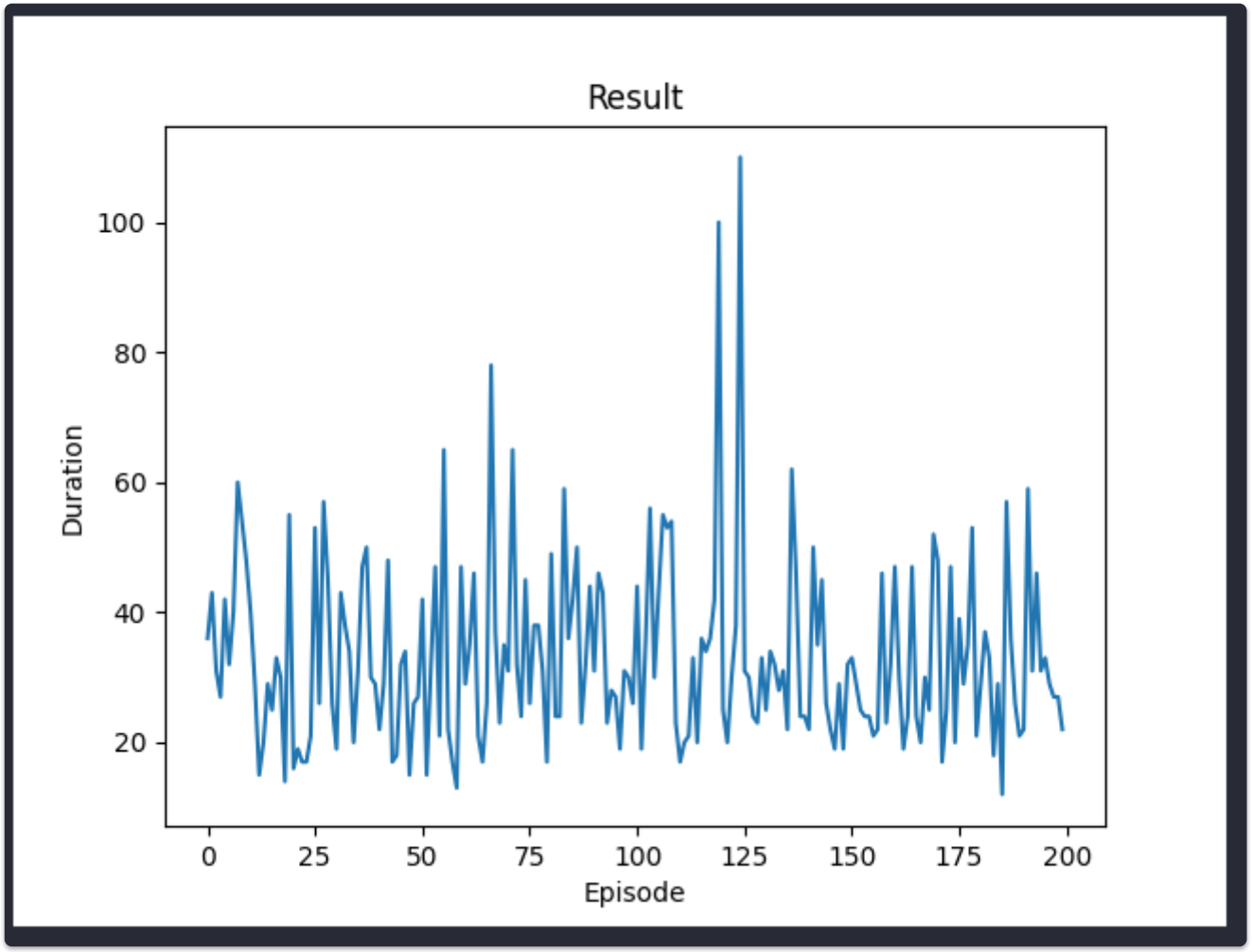
Vary the number of layers in the model and plot episode vs. duration plots.

The original number was 128, and can be seen from all the other plots.

layers: 125


Result
Duration
Episode
90
80
70
60
50
40
30
20
10
0
25
50
75
100
125
150
175
200

layers 130:



Implement without the replay buffer and observe performance.

The replaybuffer greatly improved the performance of the algorithm. For simplicity it was developed as a standalone class:

```
from collections import deque
import random
import numpy as np

class ReplayBuffer:
    def __init__(self, capacity):
```

```
        self.buffer = deque(maxlen=capacity)

    def add(self, state, action, reward, next_state, done):
        state = np.array(state, dtype=np.float32).flatten()
        next_state = np.array(next_state, dtype=np.float32).flatten()

        experience = (state, action, reward, next_state, done)
        self.buffer.append(experience)

    def sample(self, batch_size):
        return random.sample(self.buffer, batch_size)
```

Task 3

There were som minor issues implementing this on the mac m2 chips, but after some trial and error the issues were solved. The code is contained within a single file, since the difference between SARSA and Q-learning algorithm isn't that big implementation wise.

The update function where implemented seperatly and called based on which algortihm that was set for the current execution:

```
def sarsa_update(q, learning_rate, reward, next_q_value, discount_factor):
    return q + learning_rate * (reward + discount_factor * next_q_value - q)


def q_learning_update(q, state, action, reward, new_state, learning_rate, discount_factor):
    return q[state] + learning_rate * (reward + discount_factor * np.max(q[new_state]) - q[state])
```