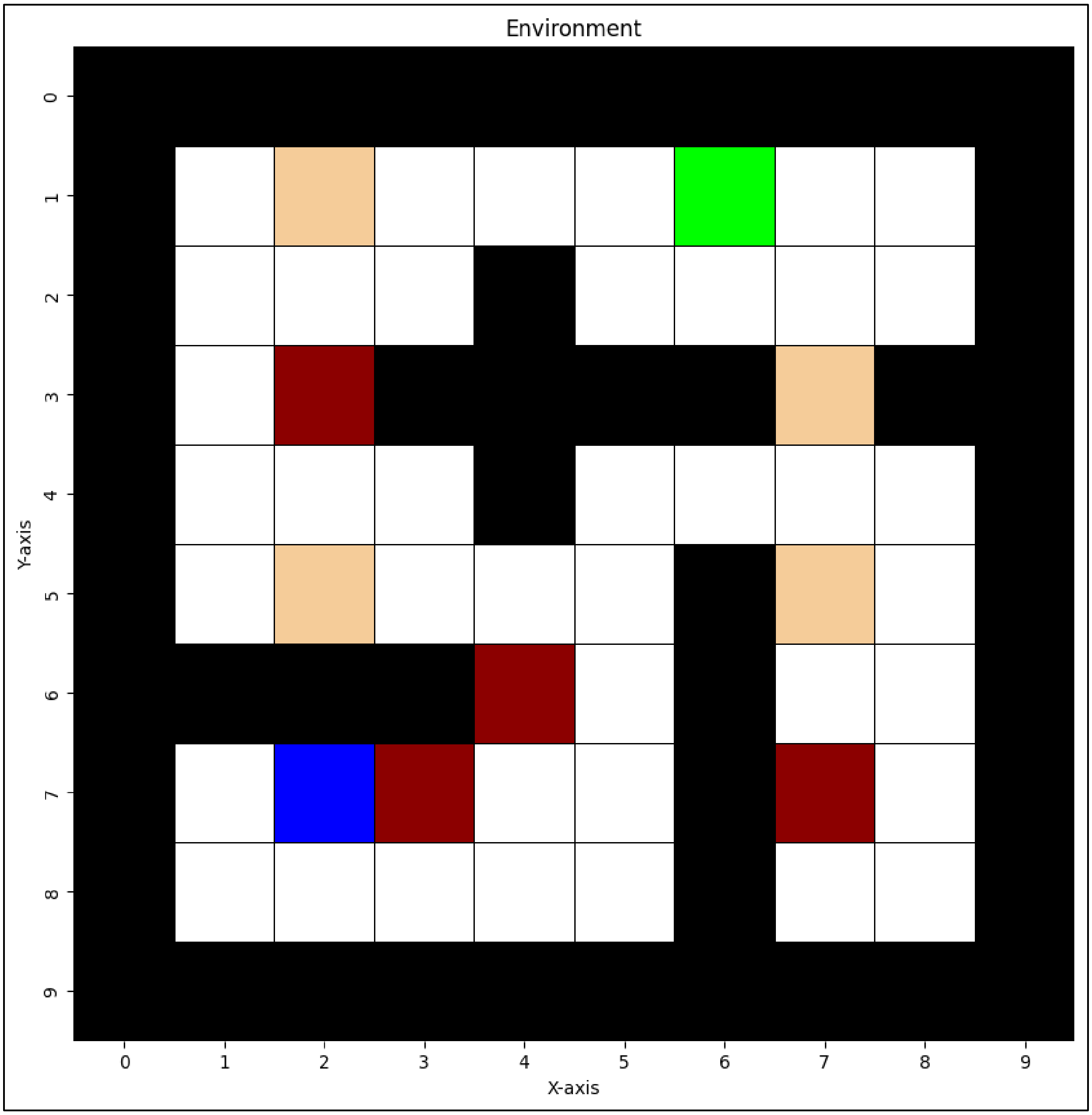
Project 4

Name: Ameya Padwad
NUID: 002284038

Implementation

The environment states are represented like this:



The following is the implementation of the DQN training process:

```
agent = DQNAgent(Env, lr, gamma, tau, buffer_size, batch_size
    rewards, losses = [], []

    for ep_no in range(num_episodes):
        state = get_random_state(Env)
        total_reward, total_loss = 0, 0
        step = 0

        while step < max_steps or state != GOAL_STATE:
            step += 1
            epsilon_value = calculate_epsilon(epsilon, step)
            action_taken = agent.select_action(state, epsilon_value)
            probabilities = get_probabilities(action, p)
      next_i, next_j, wall_hit = get_next_state(Env, state, 
action_taken)
            next_state = (next_i, next_j)
            reward = get_reward(Env[next_state], wall_hit)
            if next_state == GOAL_STATE:
                done = 1
            else:
                done = 0

            # Store transition
            agent.replay_buffer.push(state, action, reward, 
     next_state, done)

            # Train agent
            if step % steps_update == 0:
                loss = agent.train()
                if loss:
                    total_loss += loss
            state = next_state
            total_reward += reward

        rewards.append(total_reward)
        losses.append(total_loss)

    optimal_policy = agent.get_policy()
    q_values = agent.get_q_values()
    avg_rewards = calculate_moving_avg(rewards)
    avg_losses = calculate_moving_avg(losses)

    return optimal_policy, q_values, avg_rewards, avg_losses
```

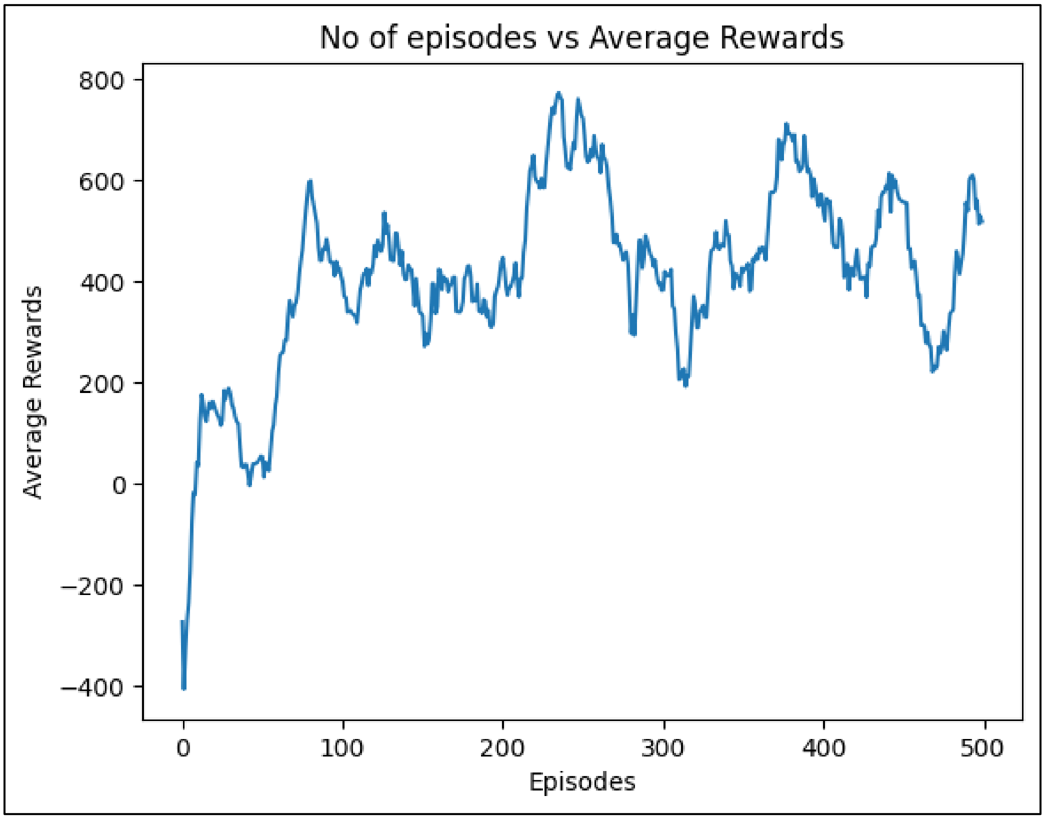
Epsilon was implemented using this function, and calculated in every step of each episode:

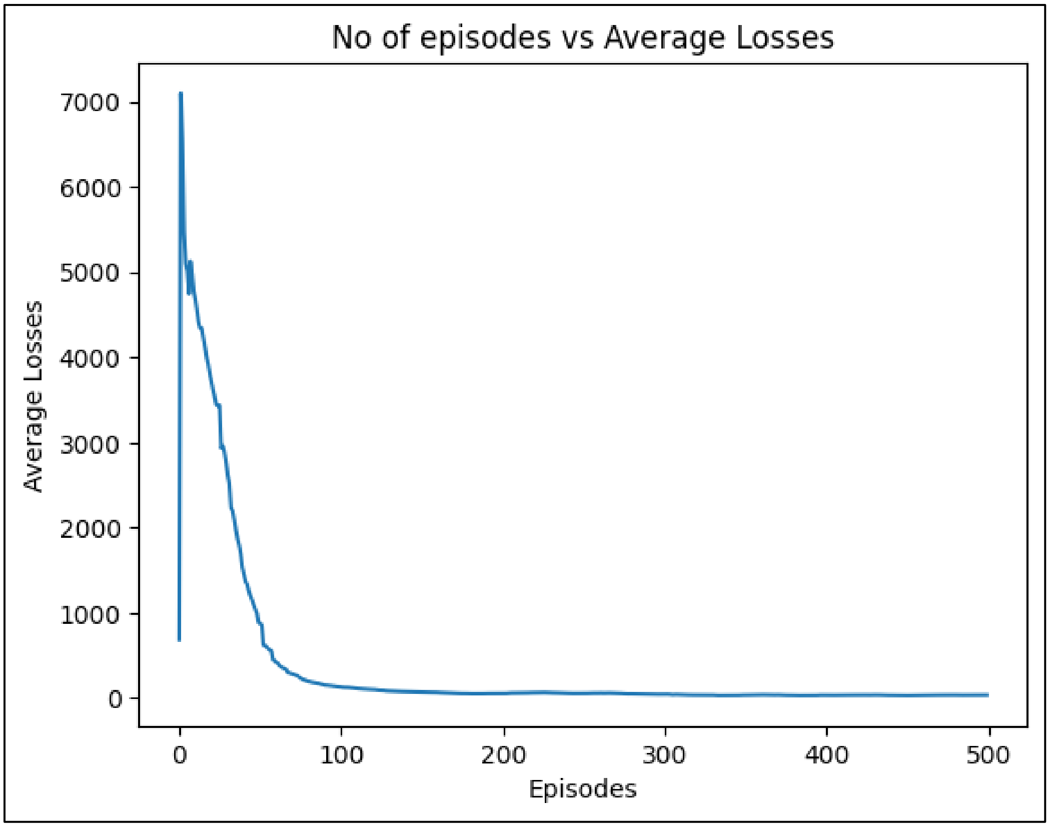
```
def calculate_epsilon(epsilon, ep_no):
    return max(0.1, epsilon ** ep_no)
```

DQN

1. **Parameters and hyperparameters:**
    a. The size of the replay memory: $|D|$ = 5000
    b. The maximum length of episode: T_{epi} = 50
    c. The number of steps to update the Q-network: N_{QU} = 1
    d. The discount factor: γ = 0.99
    e. The size of the minibatch: N_{batch} = 64
    f. The learning rate: α = 0.001
    g. The soft update hyperparameter: η = 0.01
    h. The number of episodes: N_{epi} = 500
    i. Epsilon: ε = 0.99

2. **Avg Reward and Avg Loss plots:**

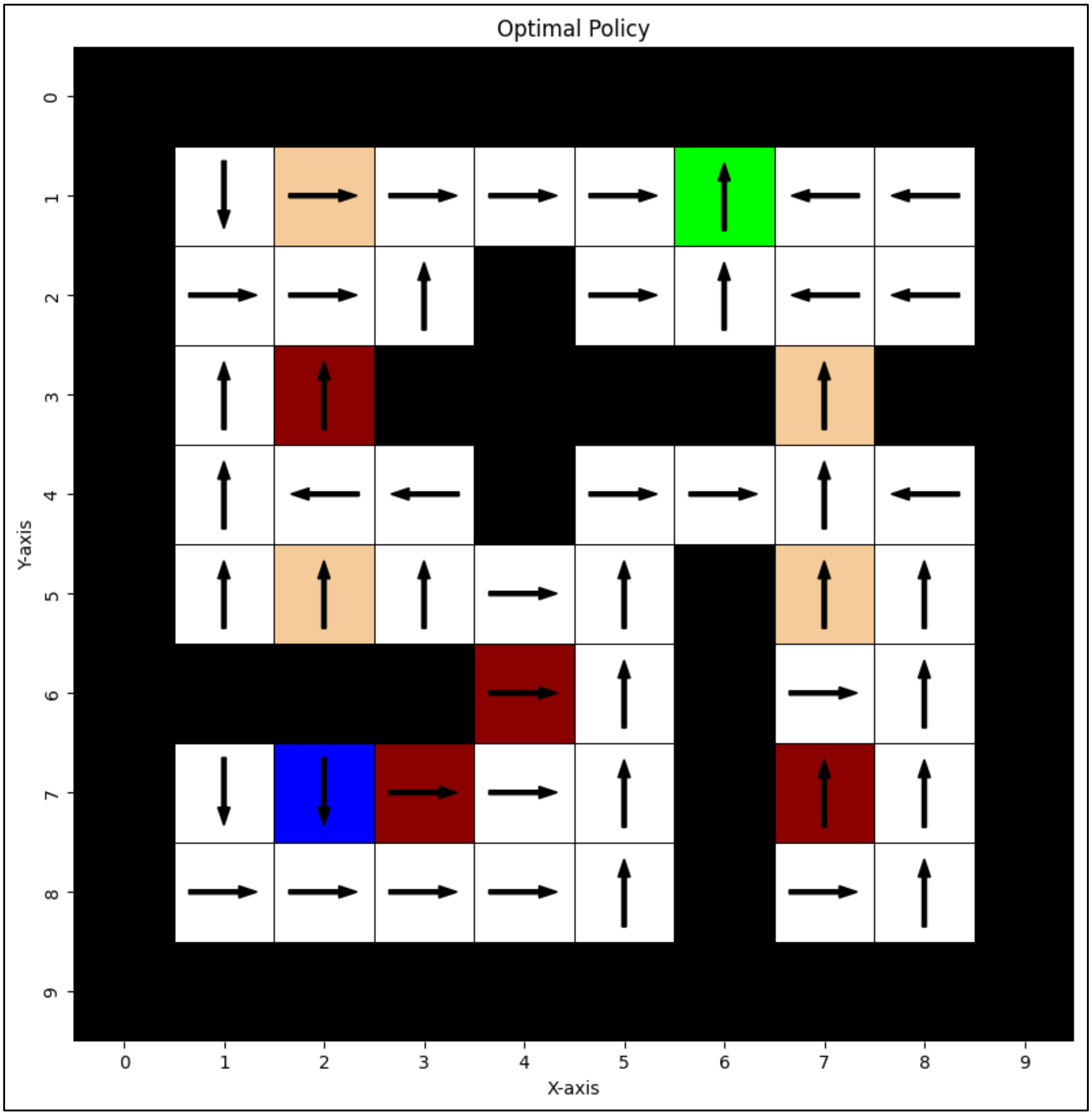




The first plot shows the variation of average rewards. Initially, in the first few episodes (approximately between 0 to 50), the average rewards are highly negative as the agent is exploring the environment mostly through random actions due to the high value of epsilon. As the training progresses, the agent starts learning better policies from its past experiences stored in the replay buffer, and hence, the average rewards start increasing steadily. The reward curve continues to rise until around episode 200, after which it stabilizes within a range, with some fluctuations. These fluctuations in the reward can be attributed to the stochasticity of the environment and the continued exploration enforced by the epsilon-greedy policy.

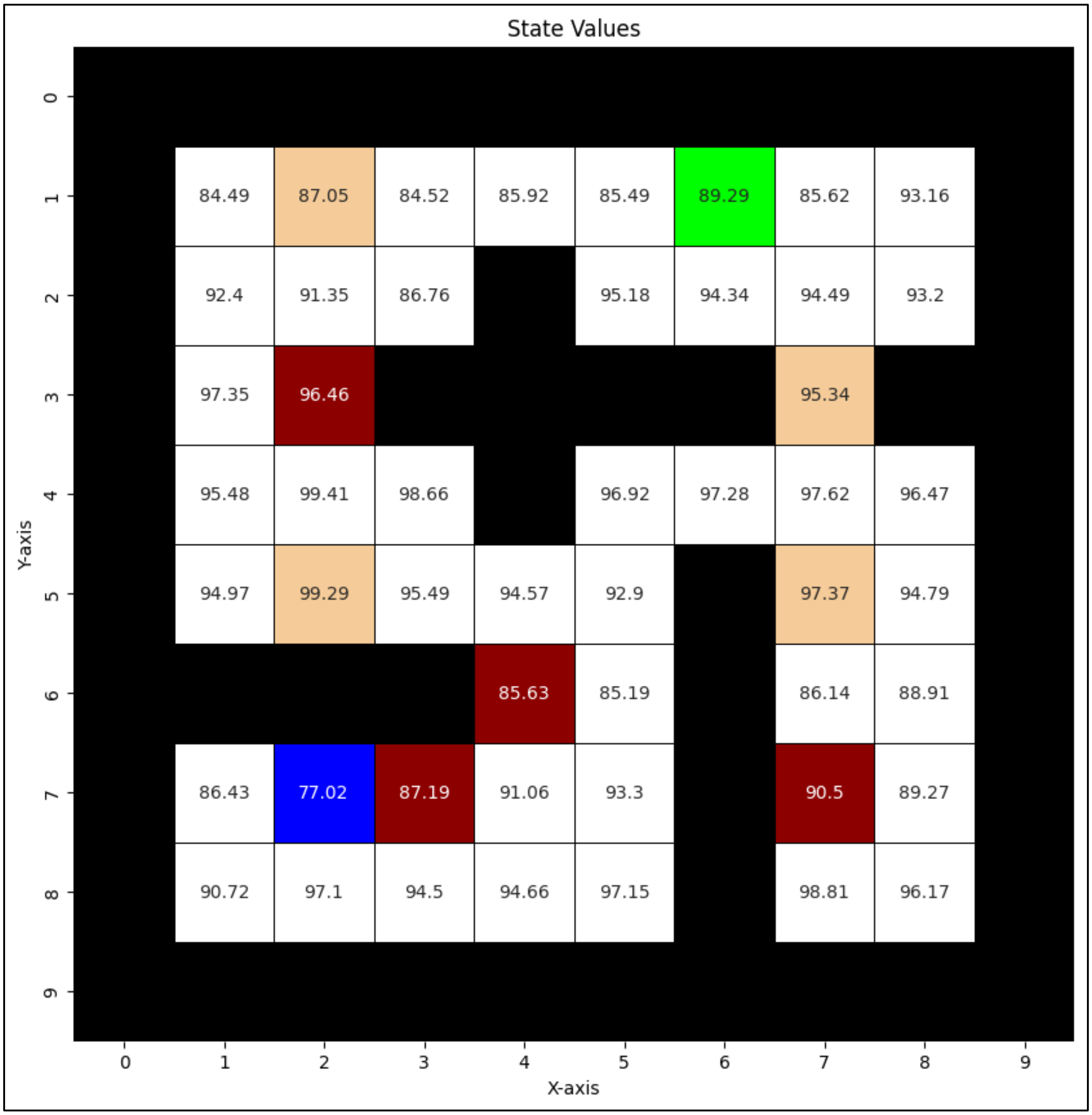
The second plot shows the change in the average loss per episode during training. In the initial episodes, the average loss is extremely high because, at the start, the Q-network is randomly initialized, and its predictions for the Q-values are highly inaccurate. As the training proceeds and the agent gathers more meaningful experiences in the replay buffer, the loss starts to decrease rapidly, stabilizing to near zero after episode 100.

3. **Obtained optimal policy:**



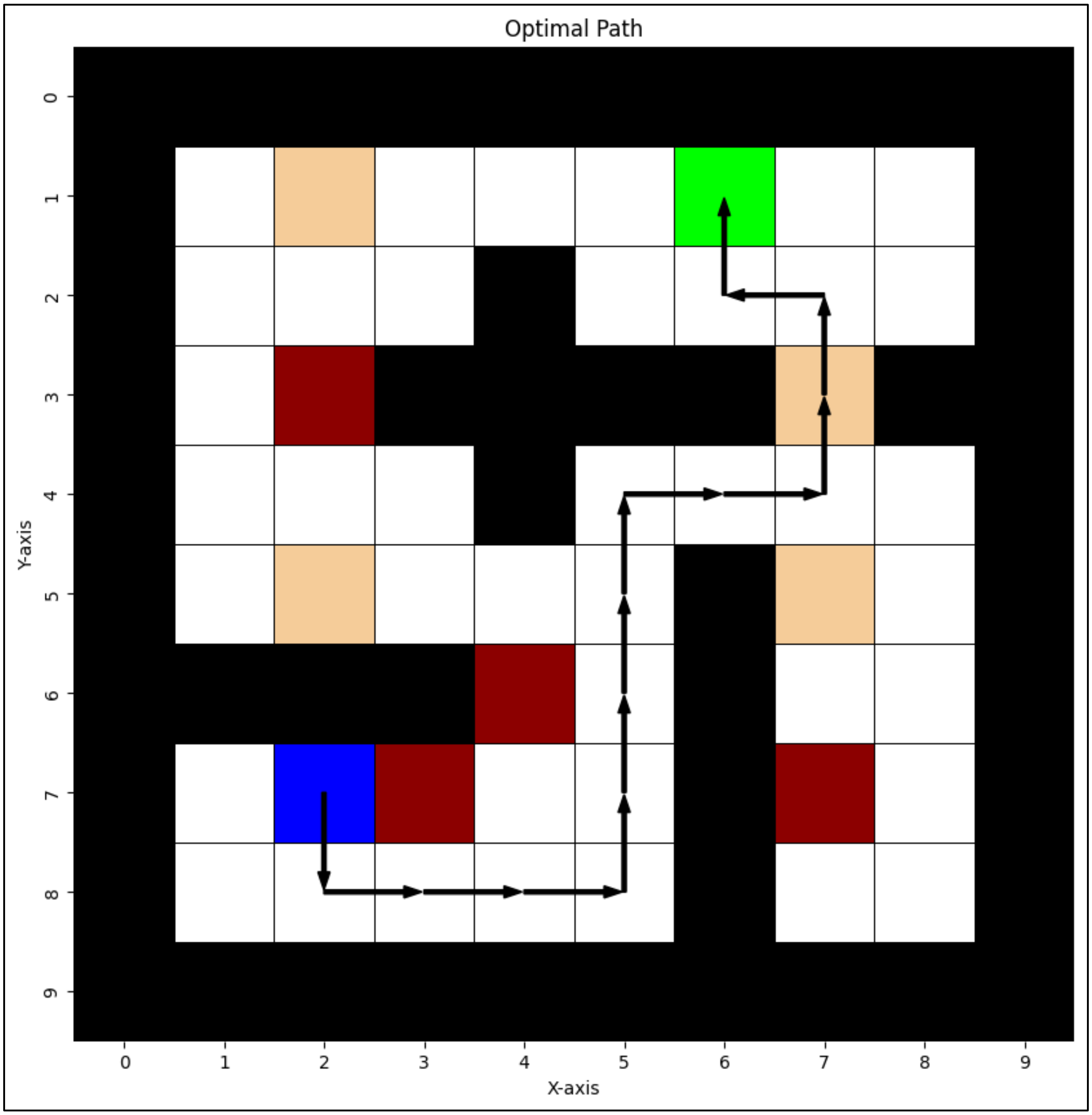
The obtained policy seems to be intuitively optimal, since it avoids as many penalty cells as possible while trying to maximize the returns to reach to the goal.

4. **Obtain optimal state values:**



As can be seen from the plot above, the state values are higher in regions that are closer to the goal state and the values gradually decrease as we move away from the goal. This happens because the agent would require fewer steps to reach to the goal from a state near the goal. The lowest values are observed around the red cells and certain dead-end regions, which is because those states either impose a high penalty or make it difficult to reach the goal efficiently. Therefore, the range of the state values (from around 77 to 99) is consistent with expectations for this environment and reward structure.

5. Obtained optimal path:



Double DQN

While implementing Double DQN, only the calculation of the target values was changed to:

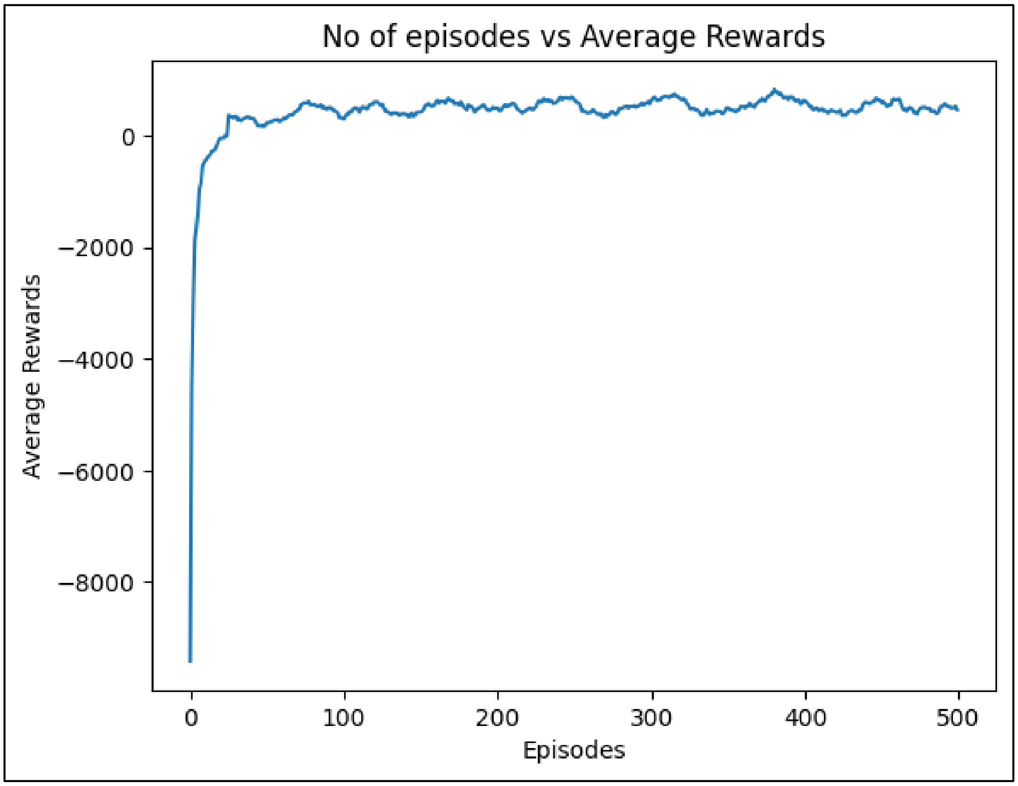
```
      target_zi ← ri + γ * Qw-(s'i, argmax Qw(s’i, a’)) * (1 - donei)
```

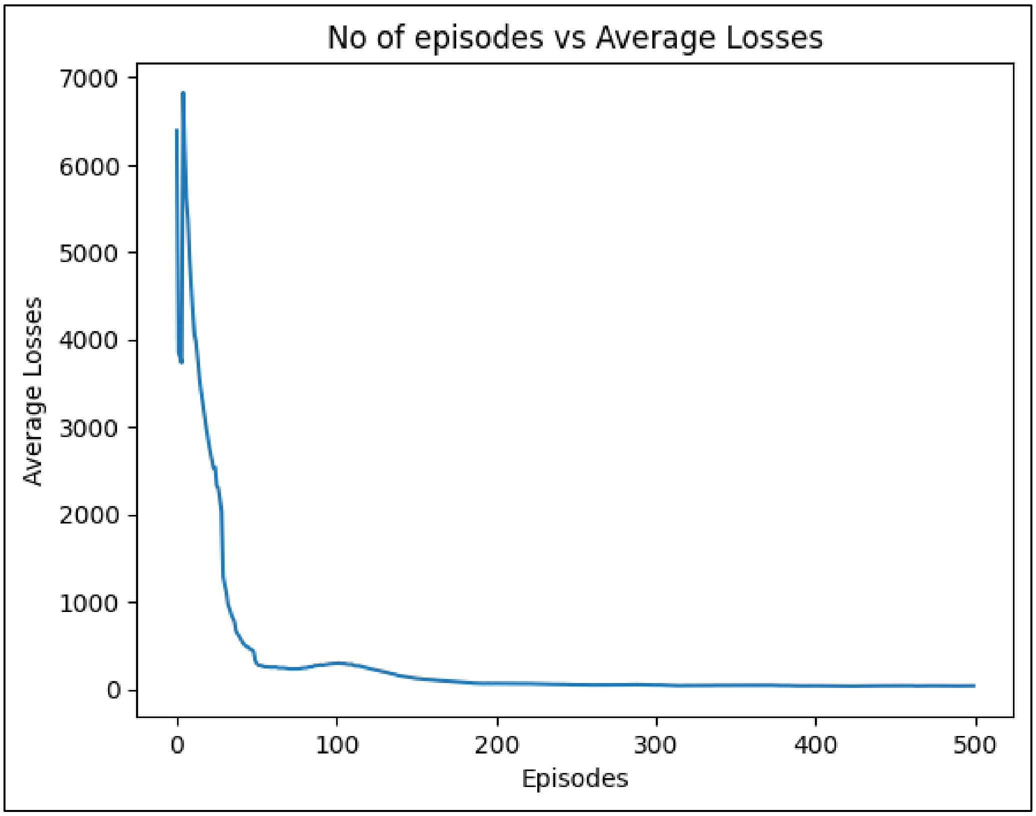
A simple boolean value was used in the implementation of the DQN agent to change it to a double DQN:

```
class DQNAgent:

      def init(self, Env, lr, gamma, tau, buffer_size, batch_size,
double_DQN = False, dueling_DQN = False):

      self.states = torch.tensor(get_all_states(Env),
dtype=torch.float32)

      self.double_DQN = double_DQN

      if dueling_DQN:
         self.q_network = DuelingQNetwork()
         self.target_network = DuelingQNetwork()
      else:
         self.q_network = QNetwork()
         self.target_network = QNetwork()

      self.target_network.load_state_dict(self.q_network.state_dict())

      self.optimizer = optim.Adam(self.q_network.parameters(), lr=lr)

      self.gamma = gamma
      self.tau = tau

      self.replay_buffer = ReplayBuffer(buffer_size)
      self.batch_size = batch_size
```

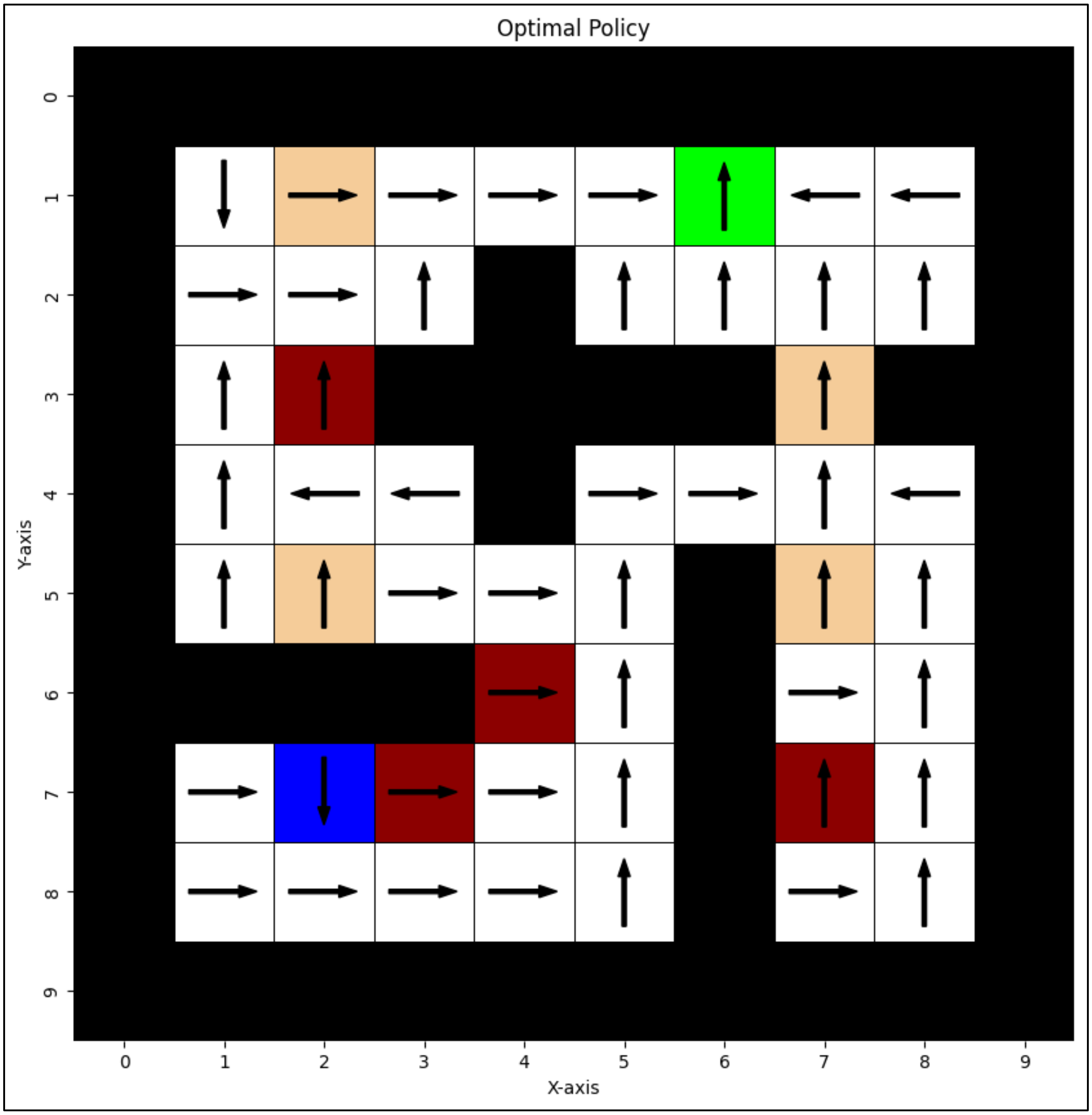
The same parameters were used as DQN.

1. **Avg Reward and Avg Loss plots:**

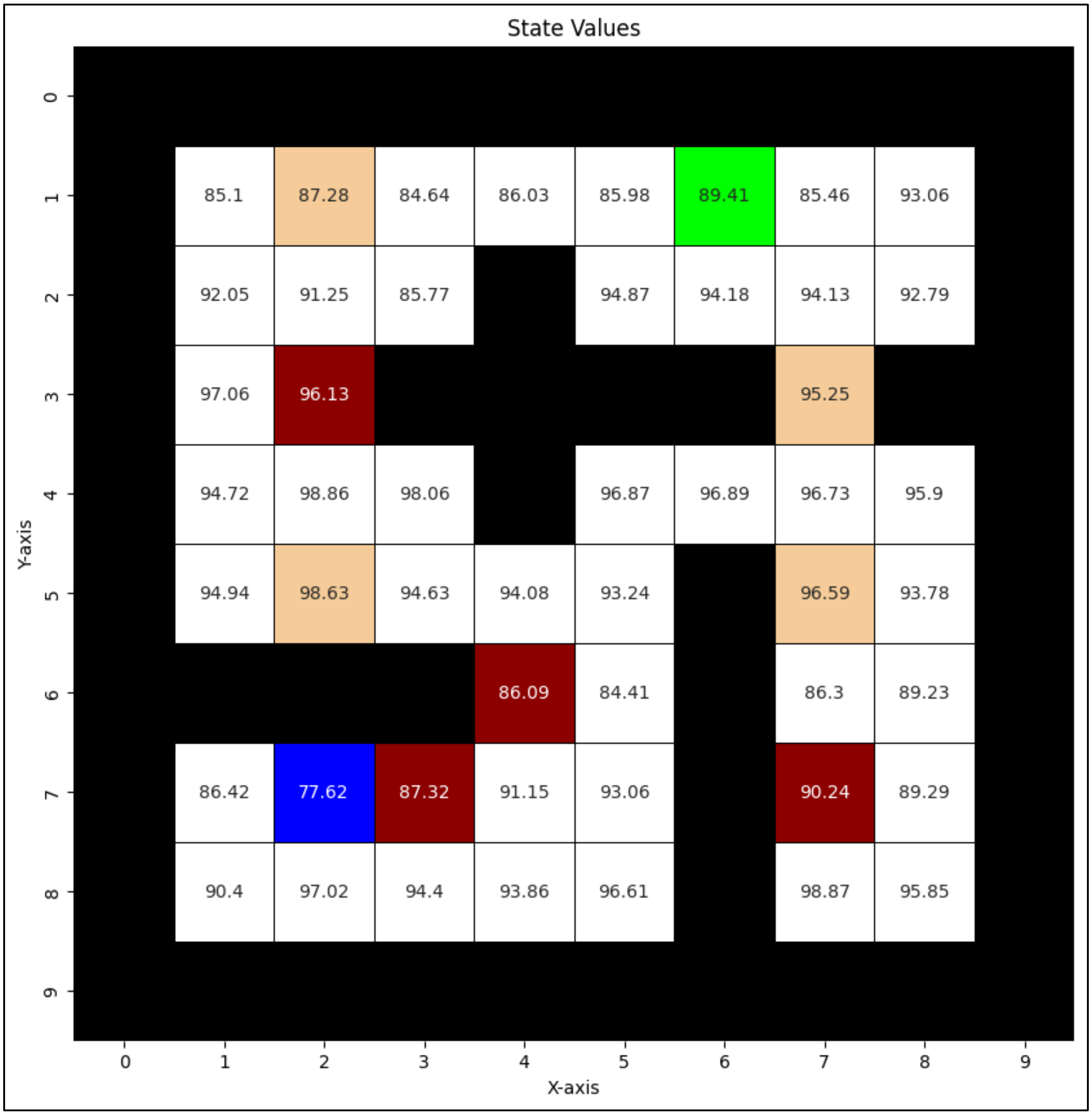




2. **Obtained optimal policy:**

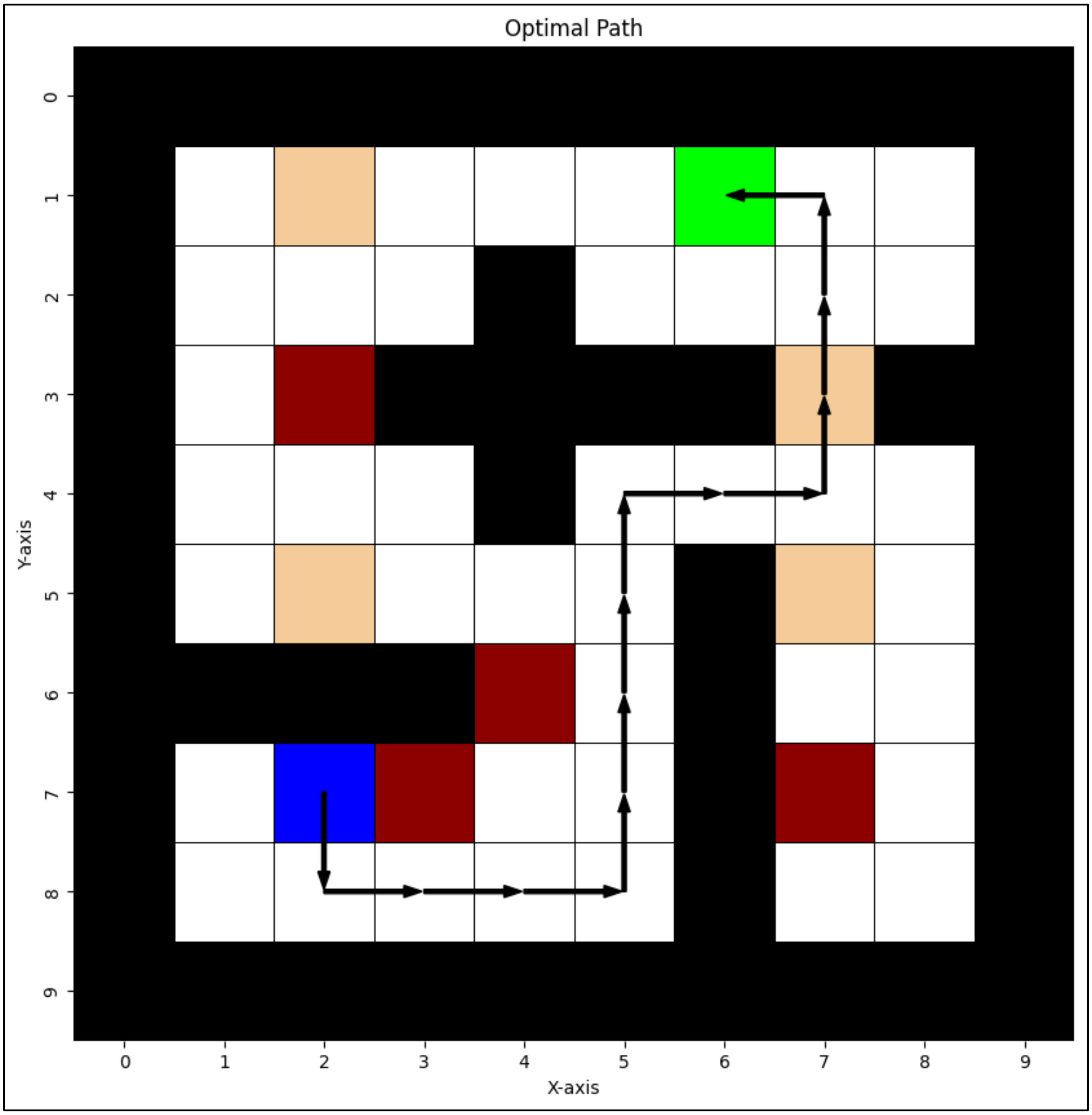


3. **Obtain optimal state values:**



	0	1	2	3	4	5	6	7	8	9
0										
1		85.1	87.28	84.64	86.03	85.98	89.41	85.46	93.06	
2		92.05	91.25	85.77		94.87	94.18	94.13	92.79	
3		97.06	96.13					95.25		
4		94.72	98.86	98.06		96.87	96.89	96.73	95.9	
5		94.94	98.63	94.63	94.08	93.24		96.59	93.78	
6					86.09	84.41		86.3	89.23	
7		86.42	77.62	87.32	91.15	93.06		90.24	89.29	
8		90.4	97.02	94.4	93.86	96.61		98.87	95.85	
9										

4. **Obtained optimal path:**



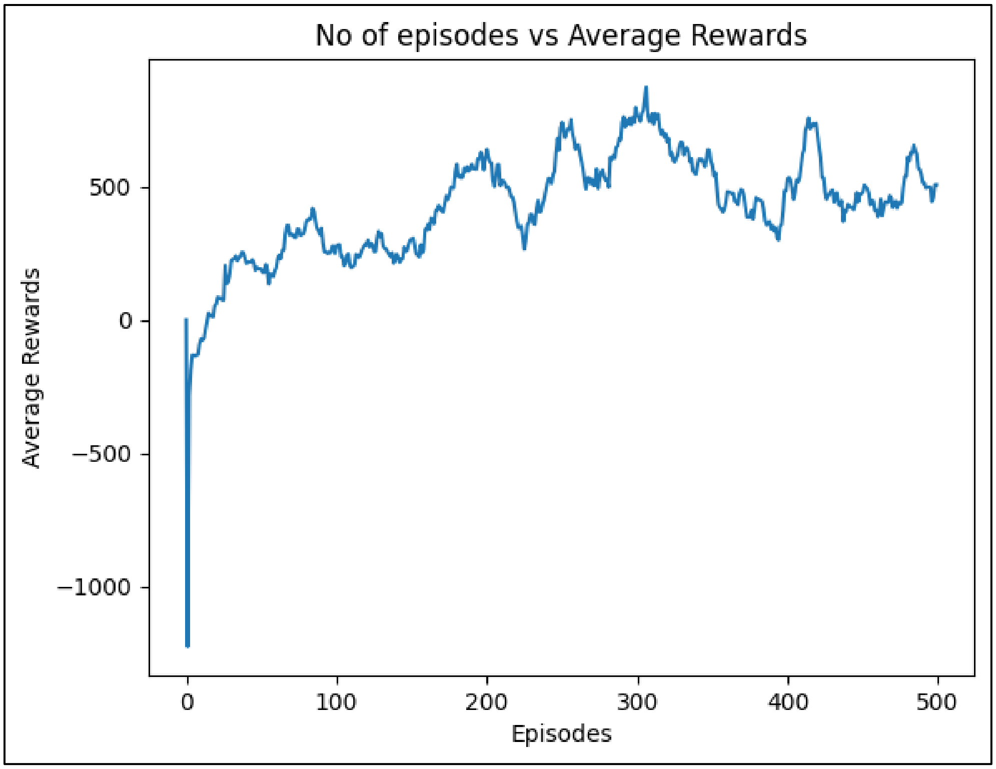
Dueling DQN

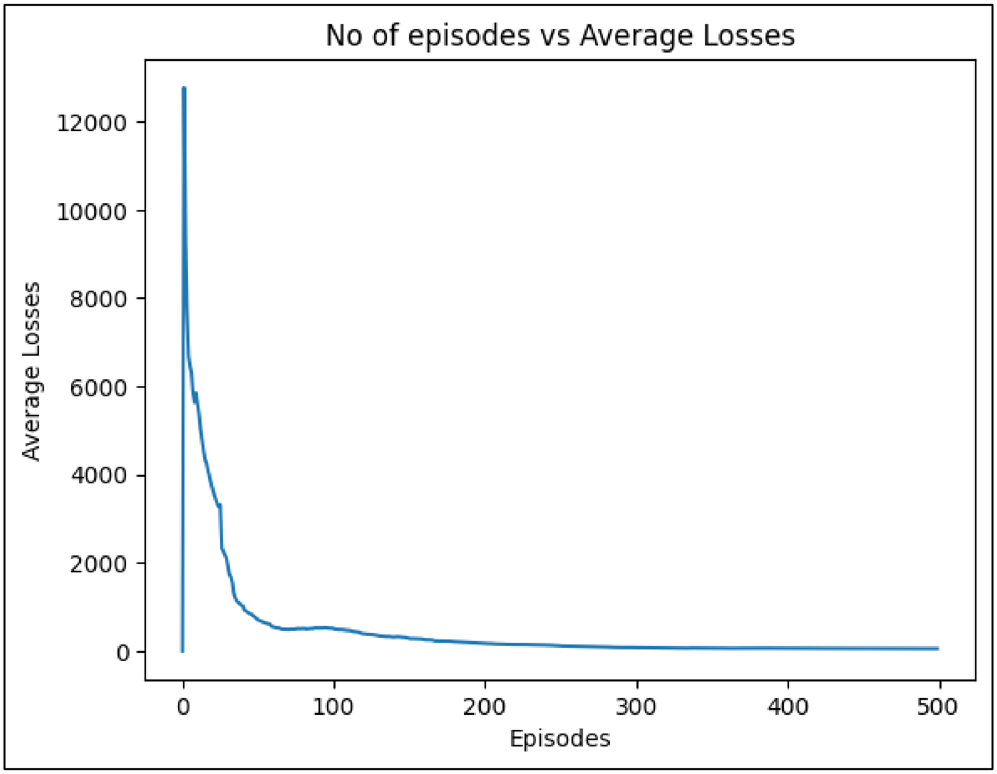
In the implementation of dueling DQN, the only difference was using a dueling Q network instead of the conventional Q network.

```
class DuelingQNetwork(nn.Module):

     def init(self): super(DuelingQNetwork, self).init()

           # Feature extractor
           self.features = nn.Sequential(
                nn.Linear(2, 128),
                nn.ReLU(),
                nn.Linear(128, 128),
                nn.ReLU()
                )

           # Value stream (output = 1)
           self.value_stream = nn.Linear(128, 1)

           # Advantage stream (output = 4)
           self.advantage_stream = nn.Linear(128, 4)


     def forward(self, x):
         features = self.features(x)

         value = self.value_stream(features)
         advantage = self.advantage_stream(features)

         # Combine V(s) and A(s, a) to get Q(s, a)
         q_values = value + (advantage - advantage.mean(dim=1, 
keepdim=True))

         return q_values
```

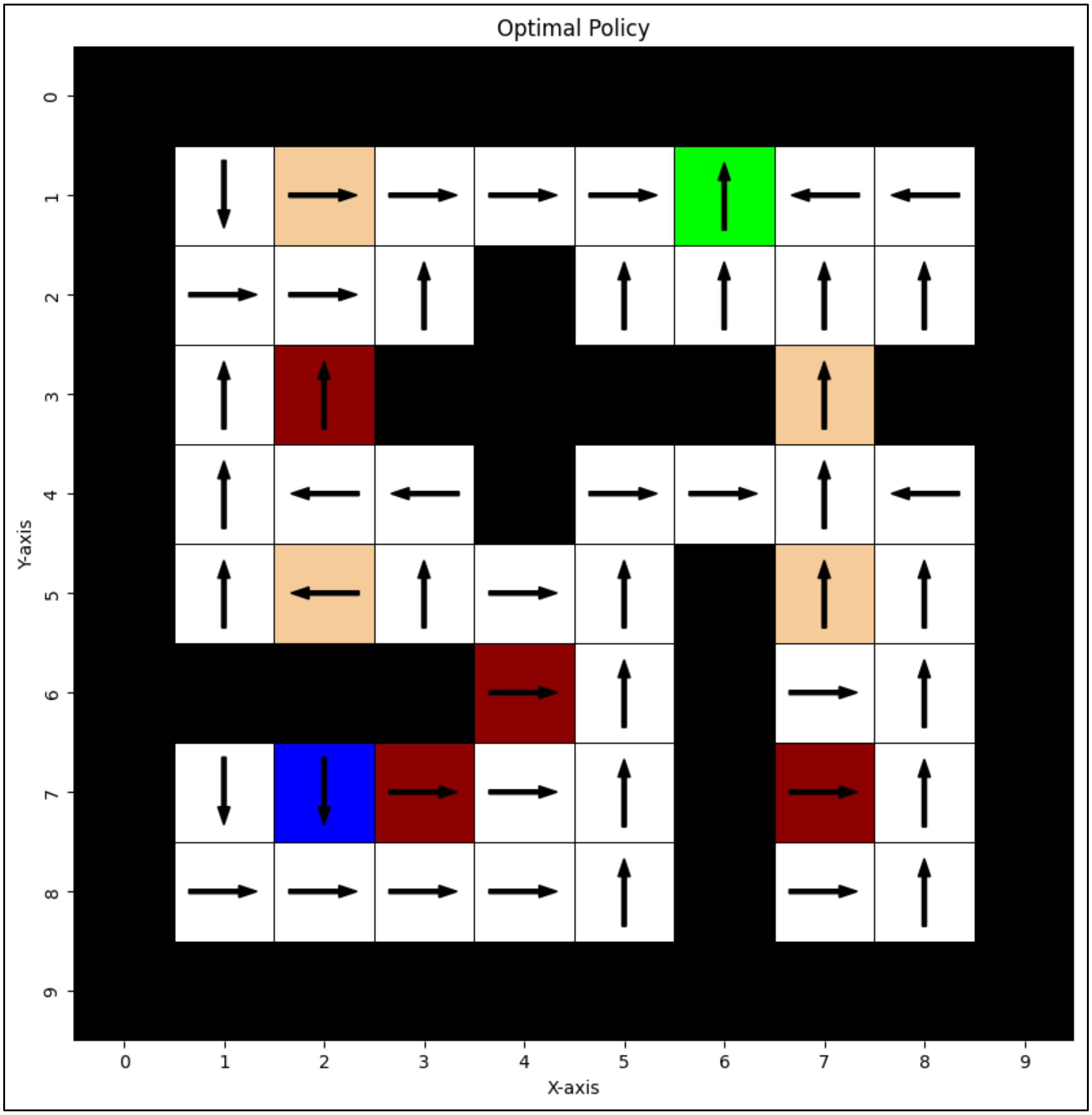
Everything else including the parameters and the training process were same as DQN.

1. **Avg Reward and Avg Loss plots:**

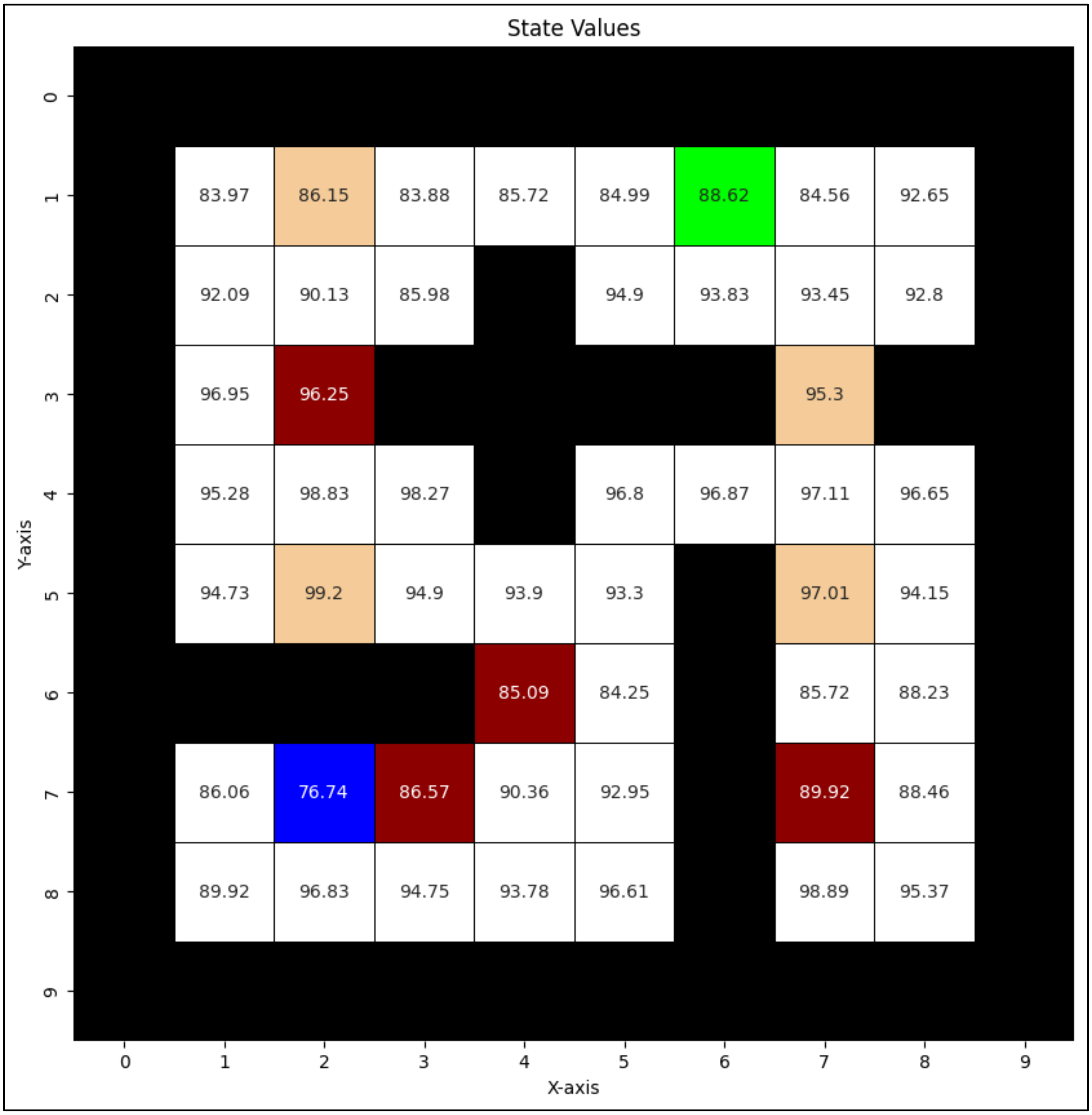




2. **Obtained optimal policy:**

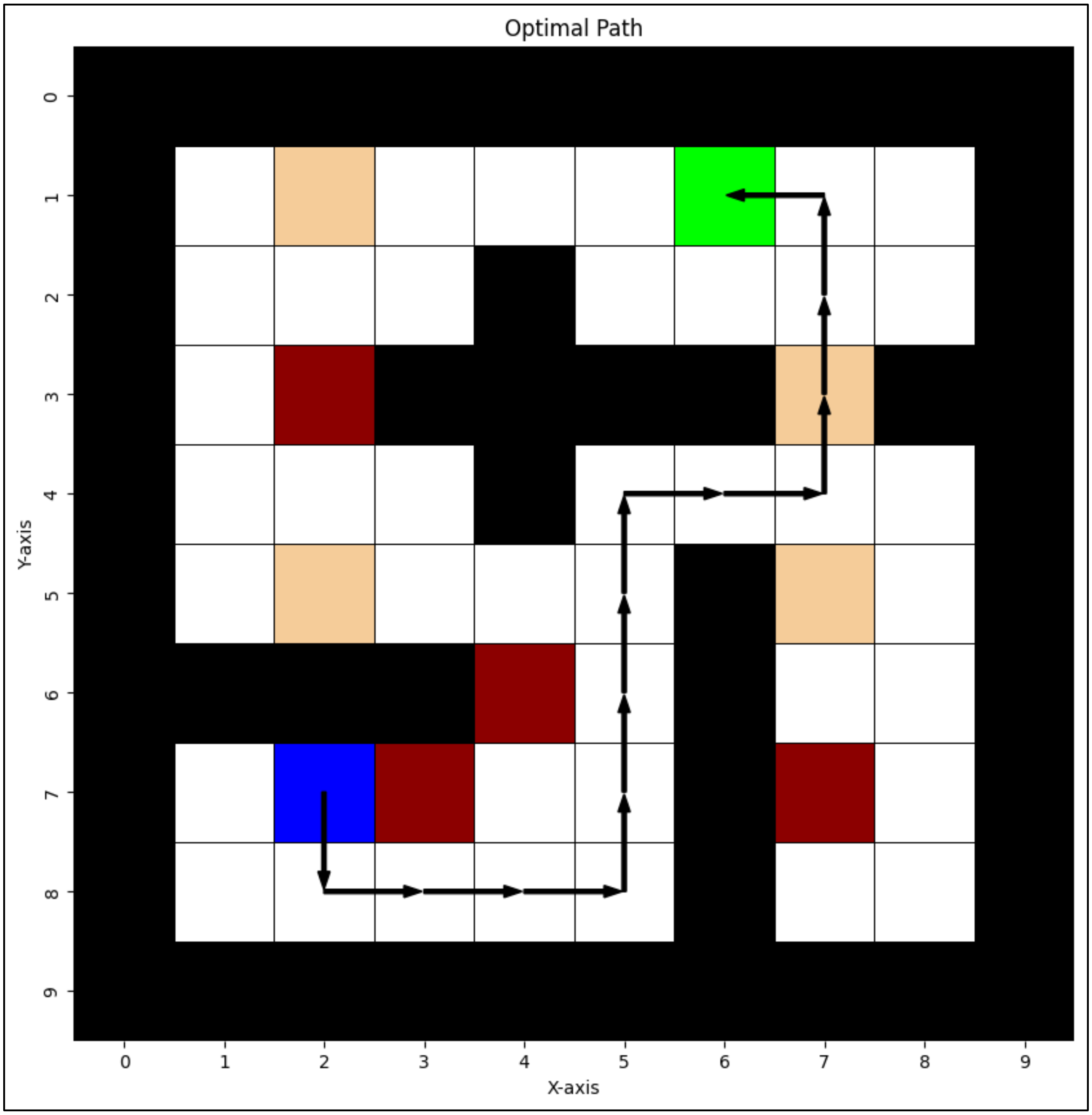


3. Obtained optimal state values:



	0	1	2	3	4	5	6	7	8	9
0										
1		83.97	86.15	83.88	85.72	84.99	88.62	84.56	92.65	
2		92.09	90.13	85.98		94.9	93.83	93.45	92.8	
3		96.95	96.25					95.3		
4		95.28	98.83	98.27		96.8	96.87	97.11	96.65	
5		94.73	99.2	94.9	93.9	93.3		97.01	94.15	
6					85.09	84.25		85.72	88.23	
7		86.06	76.74	86.57	90.36	92.95		89.92	88.46	
8		89.92	96.83	94.75	93.78	96.61		98.89	95.37	
9										

4. **Obtained optimal path:**



Comparison

In DQN, the average reward curve shows a clear upward trend in the early episodes, indicating that the agent was able to learn a reasonable policy, and then converges to a range of rewards. However, even after the convergence, the reward fluctuates quite a bit across episodes. This is a known issue in standard DQN due to overestimation bias in Q-values. The loss function also shows rapid convergence, dropping from a high value to near-zero by around episode 100.

The Double DQN implementation improves upon this by decoupling the action selection and action evaluation steps. The reward curve shows a more stable learning process, with less dramatic fluctuations in average rewards across episodes as compared to DQN. While the initial performance was slightly worse, it quickly recovered and maintained consistently high rewards. The average loss converges smoothly to zero, similar to DQN.

In the case of Dueling DQN, the benefits of the architectural change are evident. The average reward plot shows the most stable and consistent rise, with fewer abrupt dips. While the reward curve does not rise as steeply early on as in DQN, it catches up quickly and maintains reliable performance throughout training. The loss plot, while starting from a much higher peak than the other two, shows a steady and smooth decline over time. This behavior aligns with the more expressive power of the dueling architecture, which initially needs time to learn both value and advantage components but eventually produces a robust estimate of Q-values.