Report for ML project

Matini Gabriele, 1934803

July 17, 2024

1 Introduction: the environment

The Atari SpaceInvaders environment was used to train and test the RL agents. The action space is a discrete 6 moves actions space, with actions like fire, move left, move right, shoot left, shoot right and noop (no operation). The observation space is constituted by 128 bytes (integers from 0 to 255) that describe the state of the game through the RAM bytes. The environment has a frame skip in the game of 4 frames for every action by default. The reason frames are skipped, is twofold: we first have more learning efficiency, because we have less states to consider (less steps to do), and the changes that a certain action does to the environment are more clear and smoothed out to provide a stabler learning environment. Moreover, a human player would not be able to react at every frame with a different command, so we move closer to a normal reaction time. The setup is deterministic by itself, however some stochasticity is given by the "repeat" special action probability: there is a 0.25 probability that the previous action is repeated. This was introduced by the environment to avoid the agent simply memorizing an optimal sequence without paying attention to the environment observations. If we lose the environment sets a "truncated" boolean to true, and we will have to reset it. The game gives the agent a reward equal to how many points it makes, thus the only way to get a reward is to kill the space invaders. If the agent manages to kill all the enemies, a new batch of faster enemies will spawn. Thus the objective is solely to score as many points as possible. Notice that no negative reward is given for losing lives, of which we have up to 3.

2 Q-Table Agent

A highlight of the learning process of the Q-Table was the memory usage of the state: the environment returns a numpy array of 128 unsigned 8bit integers, for a total of 240 bytes. Thus, the state occupies a lot of space in the dictionary. Furthermore, converting the array to a tuple (hashable type) creates a massive overhead, going from 240bytes to 1064bytes, just to represent a 128bytes RAM state. This is why it was opted instead for a bytes array representation of the state occupying only 161 bytes. The way it was learned of these facts was by detecting a memory problem through Google Colab's memory leaks, and using in locale a python memory profiling library to discover that the observations where occupying all the RAM's space. Using sys.getsizeof(), the size of the single representations was then printed, finding out that the problem was using a tuple representation. In this way we are able to construct Q-Tables of 9 or 10 Gigabytes.

2.1 The learning process

First of all, the ϵ greedy policy was defined: this policy let the agent pick a random action for the state with probability of ϵ, and the action with the maximum reward associated in that state in the Q-Table with a probability of 1 - ϵ:

```
class EpsGreedyPolicy:
    ...
  def __call__(self, obs, eps, Q) -> int:
    greedy = random.random() > eps
    if greedy:
      if np.all(Q[obs] == Q[obs][0]):
        return np.random.choice(len(Q[obs]))
      else:
        if self.show:
          print("Chosen action ", np.argmax(Q[obs]))
        return np.argmax(Q[obs])
    else:
      return env.action_space.sample()
```

Notice how, if the state is new but the policy is greedy, all the actions associated to it will have reward 0. We decide to choose a random action in this case, to favour exploration in training and randomness (otherwise we would just pick the first action all the time). Next, we actually define the Q-Learning Agent and its training:

```
class QTable:

  def __init__(self, eps=1.0, showAction=False):
    self.Q = defaultdict(lambda: np.zeros(env.action_space.n))
    self.eps = eps
    self.policy = EpsGreedyPolicy(show=showAction)

    def qlearn(
      self,
      env,
      alpha0,
      gamma,
      max_steps
  ):
    rewards = []
    obs, info = env.reset()
    obs = bytes(obs)
    done = False
    tot_rew = 0
    for step in tqdm(range(max_steps)):

      if done:
        rewards.append(tot_rew)
        tot_rew = 0
        obs, info = env.reset()
        obs = bytes(obs)
        done = False
      self.eps = (max_steps - step) / max_steps
      action = self.policy(obs, self.eps, self.Q, env)
      results = env.step(action)

      if len(results) == 5:
        obs2, reward, terminated, truncated, info = results
      else:
        obs2, reward, terminated, info = results
        truncated = False

      obs2 = bytes(obs2)
      done = terminated or truncated
      max_next_q = np.max(self.Q[obs2])
      tot_rew += reward

      self.Q[obs][action] += alpha0 * (reward + gamma * max_next_q - self.Q[obs][action])
      self.Q[obs][action] = round(self.Q[obs][action], 5)

      obs = obs2
    return rewards
```

The Q-Table is a dictionary $< state : rewards\_array >$, where each index of the rewards array defines the correspondent action. Wanting to control how many total steps we take, it was decided to have the training last a fixed number of steps, and if the game ends sooner, we just reset the environment instead of restarting the episode. For each episode, ϵ descends in line with the number of steps, and at each iteration, the action taken with the state is updated in reward value according to the Q-Function update rule:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right] \tag{1}$$

Where we maximize the sum of future rewards, where γ is a weight telling how much we care about the next state rewards, between 1 and 0 (we use 0.99). α is the learning rate and we set that to 0.1, and r is the reward we got by doing our action. So this is going to be our non-deterministic update function. Every episode is defined from its start to the point at which the game ends.

The way we train the agent is by making it do 25 milion steps in total and plotting, for each episode, how the average collected reward in the last 100 episodes evolves through each episode from the 99th to the last. This is because the actual plotting of the reward through each episode is not stable at all, and thus not very informative. A plotting of the average can actually give us a better idea. We also try to change gamma and see how the performances are affected.

2.2 The testing

We test the agent with a rollout function as follows:

```python
def rollout(
      self,
      env: gym.Env,
      policy,
      gamma: float,
      n_episodes: int,
      max_steps: int,
      render=[],
      takeVideo=False
  ) -> float:
    sum_returns = 0.0
    obs, info = env.reset()
    discounting = 1
    for ep in tqdm(range(n_episodes)):
      done = False
      obs, info = env.reset()
      discounting = 1
      while not done:
        obs = tuple(obs)
        action = self.policy(obs, self.eps, self.Q)
        obs, reward, terminated, truncated, info= env.step(action)
        done = terminated or truncated
        sum_returns += reward * discounting
        if takeVideo:
          frame = env.render()
          render.append(frame)
        discounting *= gamma
    print("Info: ",info)
    return sum_returns / n_episodes
```

The rollout function supports the rendering of the environment through taking singular frames of every state in the game. The function supports by itself more games than one, in order to test N times the training approaches.

2.3 The results

The training seems, to work, since the average reward per episode does go up with each iteration, even though slowly. We end up with a 9Gbytes Q-Table that is able to eliminate around 10 or 15 enemies per run.

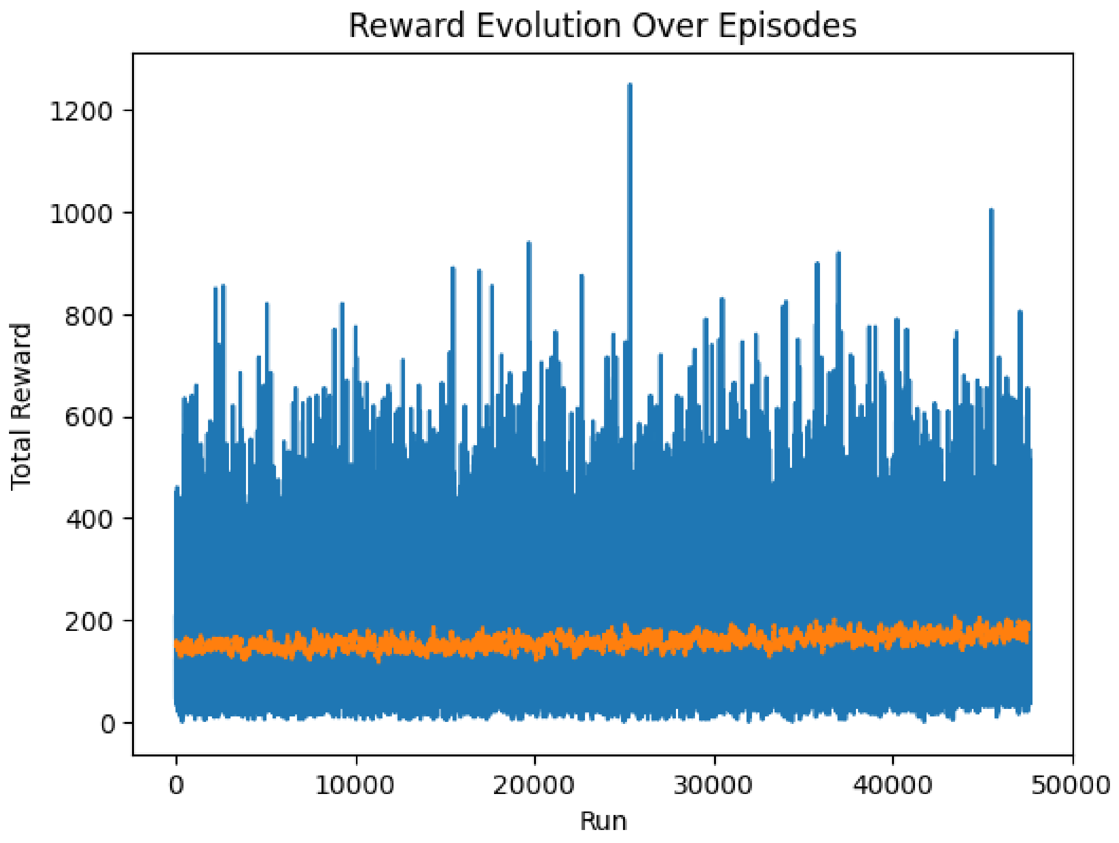

Figure 1: In orange: average reward evolution in the last 100 episodes, in blue we have the reward per episode, as one can see it's not that informative because of how unstable it is, that's why it was chosen to plot the average reward. The gamma used was 0.99, alpha 0.1.

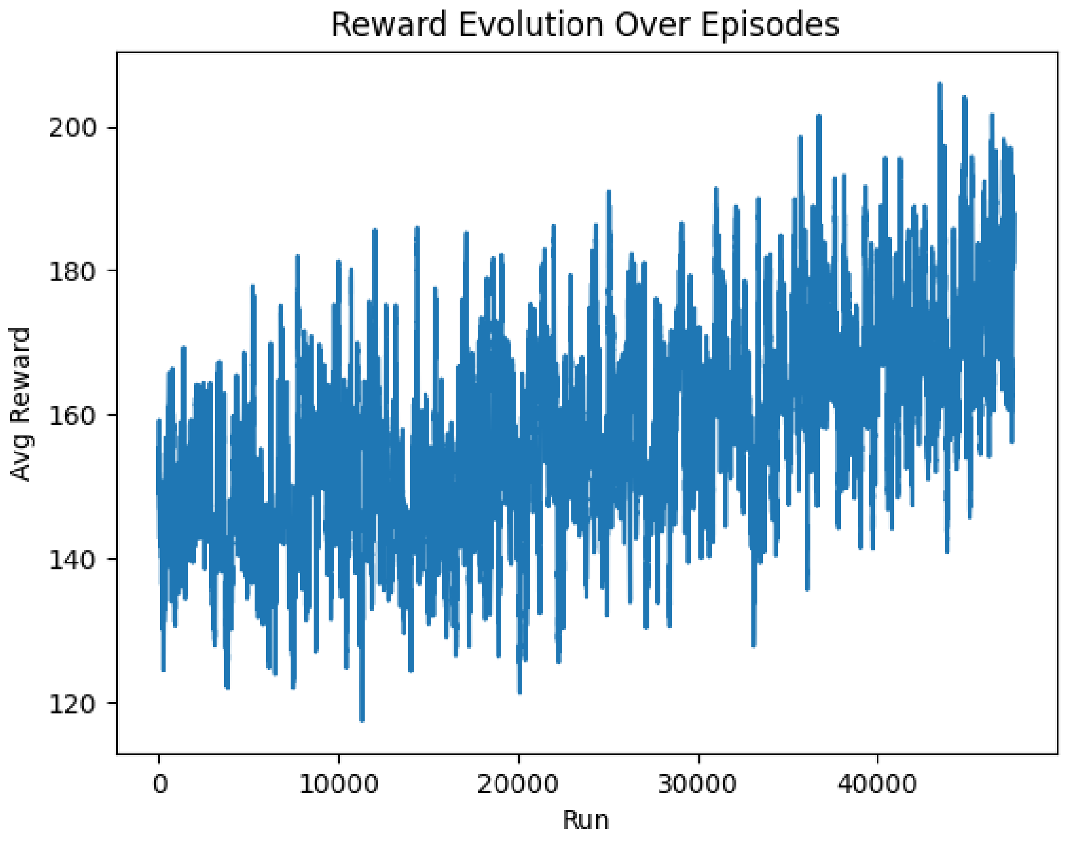

Figure 2: Here we have a zoomed-in plot of the average reward.

Finally, modifying gamma a bit (to 0.9 or 0.999) does not seem to have a noticeable impact on the performances.

3 Deep Q-Learning Agent

A more sophisticated approach is the use of a neural network that replaces the Q-Table with respect to learning states and actions' Q-values from the environment. The states will still be updated with the cited Q-Function updated rule, but instead of updating a dictionary, we update the weights of a neural network, by computing a loss gradient between the estimated Q-values and the actual target ones and using the back-propagation algorithm to update the weights. The Q-Table method becomes infeasible in environments with large or continuous state spaces due to the sheer size of the table required, and our state space is very big. In contrast, the neural network can generalize across similar states, making it possible to handle much larger and more complex environments efficiently.

3.1 DQN Training and the used NN architecture

We start with a 4 layers network (2 inner layers) of 32 nodes. The agent is trained at each single step, provided the replay buffer is large enough, and each episode lasts until the agent loses at the game. This time we don't have any problems with memory space, since the NN is a smaller solution to approximate the environment, with respect to the Q-Table. ReLu was used as the activation function, a batch of 32 elements was used as input, and Adam as the optimizer with a learning rate of 0.00025, and an MSE loss function. A 0.99 γ value was used, as well as a decay of $\epsilon = 0.999$, since the agent needs to train for 1000 episodes. ϵ must decay relatively slowly, as to expose the agent to as many states and situations as possible. The agent is trained in a twofold manner: first, only one main network is used to do everything and then a second NN is used to estimate the target Q-values for each batch. In the single-NN mode, the main NN is trained by sampling a batch of 32 experiences, computing the Q-values for these states, and then it also has to compute the target Q-values for the next states, meaning that it computes the maximum Q-values for the next states and then we use the Q-Function update rule that makes use of those maximum Q-values to update the target Q-values. We then calculate the loss between the initial estimated Q-values and the target Q-values and back-propagate the gradient results to update the weights of our deep Q-Network. In this way, we can use an NN as a shortened version of the Q-Table. This approach by itself suffers from instability, due to the fact that the main NN has to estimate the target Q-values too, and so whatever noise or error is propagated much more throughout the training. As we will see in the results section, this translates into a more or less consistent overestimation of some action with respect to others. A better approach is to leave the target Q-values estimation to another NN that in our case is the old (last episode) version of the main NN, so that whatever noise, and consequent overestimation may get smoothed out by the target NN, which gets updated once per episode, instead of once per step. As we will see, the target model will play a fundamental role in lowering the noise. To this purpose, it's also of paramount importance to expose the agent to as many different actions as possible, and that's another reason why ϵ was chosen to decay very slowly.

```python
class DQN(nn.Module):
    def __init__(self, input_shape, num_actions):
        super(DQN, self).__init__()
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(input_shape[0] * input_shape[1], 32)
        self.fc2 = nn.Linear(32, 32)
        self.fc3 = nn.Linear(32, 32)
        self.fc4 = nn.Linear(32, num_actions)

    def forward(self, x):
        x = self.flatten(x)
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = torch.relu(self.fc3(x))
        x = self.fc4(x)
        return x


class ReplayBuffer:
    def __init__(self, max_size):
        self.buffer = deque(maxlen=max_size)

    def add(self, experience):
        self.buffer.append(experience)

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

    def size(self):
        return len(self.buffer)


class DQNAgent:
    def __init__(self, state_shape, num_actions, buffer_size, batch_size,
    gamma, alpha, epsilon, epsilon_decay, min_epsilon, use_target_model=False):
        self.state_shape = state_shape
        self.num_actions = num_actions
        self.memory = ReplayBuffer(buffer_size)
        self.batch_size = batch_size
        self.gamma = gamma
        self.alpha = alpha
        self.use_target_model = use_target_model
        self.epsilon = epsilon
        self.epsilon_decay = epsilon_decay
        self.min_epsilon = min_epsilon
        self.model = DQN(state_shape, num_actions)
        self.target_model = DQN(state_shape, num_actions)
        self.update_target_model()
        self.optimizer = optim.Adam(self.model.parameters(), lr=0.00025)
        self.loss_fn = nn.MSELoss()

    def update_target_model(self):
        self.target_model.load_state_dict(self.model.state_dict())


    def act(self, state):
        if np.random.rand() < self.epsilon:
            return np.random.randint(self.num_actions)
        state = torch.FloatTensor(state).unsqueeze(0)
        with torch.no_grad():
            q_values = self.model(state)
        return torch.argmax(q_values).item()

    def remember(self, state, action, reward, next_state, done):
        self.memory.add((state, action, reward, next_state, done))

    def train(self):

        batch = self.memory.sample(self.batch_size)
        states, actions, rewards, next_states, dones = zip(*batch)

        states = torch.FloatTensor(states)
        next_states = torch.FloatTensor(next_states)
        rewards = torch.FloatTensor(rewards)
        dones = torch.FloatTensor(dones)

        q_values = self.model(states)
        q_values = q_values.gather(1, torch.LongTensor(actions).unsqueeze(1)).squeeze(1)

        if self.use_target_model:
          max_next_q_values = self.target_model(next_states).max(1)[0]
          target_q_values = q_values * (1 - self.alpha) + self.alpha * (rewards + self.gamma *
          max_next_q_values * (1 - dones))
        else:
          max_next_q_values = self.model(next_states).max(1)[0]
          target_q_values = q_values * (1 - self.alpha) + self.alpha * (rewards + self.gamma *
          max_next_q_values * (1 - dones))

        loss = self.loss_fn(q_values, target_q_values.detach())

        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()
        return loss.item()

def train_dqn(env, agent, episodes):
    rewards = []
    actions = []
    total_reward = 0
    avg_losses = []
    for episode in range(episodes):
        losses = []
        state, info = env.reset()
        state = np.reshape(state, [1, *state.shape])
        done = False
        while not done:
            action = agent.act(state)
            actions.append(action)
            next_state, reward, terminated, truncated, info = env.step(action)
            done = terminated or truncated
            next_state = np.reshape(next_state, [1, *next_state.shape])
            agent.remember(state, action, reward, next_state, done)
            state = next_state
            total_reward += reward
            if agent.memory.size() > agent.batch_size:
              loss = agent.train()
              losses.append(loss)

        avg_losses.append(np.mean(losses))
        if agent.use_target_model:
            agent.update_target_model()
        print(f"Episode: {episode + 1}/{episodes}, Reward: {total_reward}, Epsilon:
        {agent.epsilon} MemSize: {agent.memory.size()}")
        rewards.append(total_reward)
        total_reward = 0
        if agent.epsilon > agent.min_epsilon:
            agent.epsilon *= agent.epsilon_decay

    return (rewards, avg_losses, actions)
```

Other than the neural netowork, the main architecture is also composed by a replay buffer, which is as a deque data structure, to provide a mechanism to store and randomly sample up to 100000 experiences ($< state, action, reward, nextstate, done >$ tuples), used to train the NN. Additionally, a second neural network has been contemplated in the code, providing a second NN called "target neural network" or "target model", which can be activated or not during training, and which will be updated using a "hard" update technique, which consists of copying the weights of the main NN into the target NN at each new episode. Functions to save and load the agent have also been provided too. Finally, the training function is expected to run for N episodes and collect important data, such as the evolution of the loss and the reward, and how many times the actions were chosen during training, so as to detect any biases towards certain actions. In the end, the collected data will be the average loss per episode, the reward per episode and the average reward evolution calculated on the last 100 episodes (to have a better understanding if the reward is going up or not), and finally, for each action, how many times that action was chosen during training.

```
def test_dqn(env, agent, episodes, render):
    agent.epsilon = 0.0  # Disable exploration
    total_reward = 0
    for episode in range(episodes):
        state, info = env.reset()
        state = np.reshape(state, [1, *state.shape])
        done = False
        while not done:
            action = agent.act(state)
            next_state, reward, terminated, truncated, info = env.step(action)
            done = terminated or truncated
            next_state = np.reshape(next_state, [1, *next_state.shape])
            state = next_state
            total_reward += reward
            print("Chosen action is ", action)
            frame = env.render()
            render.append(frame)
            if done:
                print(f"Episode: {episode + 1}/{episodes}, Reward: {total_reward}")
                break
    return total_reward / episodes
```

The test function is designed to not only collect the reward, but also the frames to then be able to construct a video of the agent playing in the environment. It was chosen to have the function print the actions chosen by the agent in order to understand better its behavior. ϵ is also set to 0 in order to fully exploit what the agent has learned.

3.2 The results

The agent does always score some points, however from the videos and the test function prints, it's clear that both with and without the use of the target model, the agent seems to think that the ”right”, ”right-fire” and ”fire” actions will award it the most points in any state. In particular, with the use of the target model the agent will consistently overestimate ”right-fire” and ”fire”, while without target model the agent will use both ”right” and ”right-fire”, but fundamentally the agent is experiencing the same problem. This translates to the agent moving to the right side of the screen while shooting at the start, and ultimately keep shooting until they get hit by the enemies. The agent might start taking other actions only when some enemies have been defeated, but it will still overwhelmingly choose ”right-fire” and ”fire”. This stems from a documented problem of noise in some Atari environments, which makes the agent overestimate some actions at the expense of every other action. However, the agent does seem to learn something during training though, as documented by these graphs plotted from the data acquired during the training:

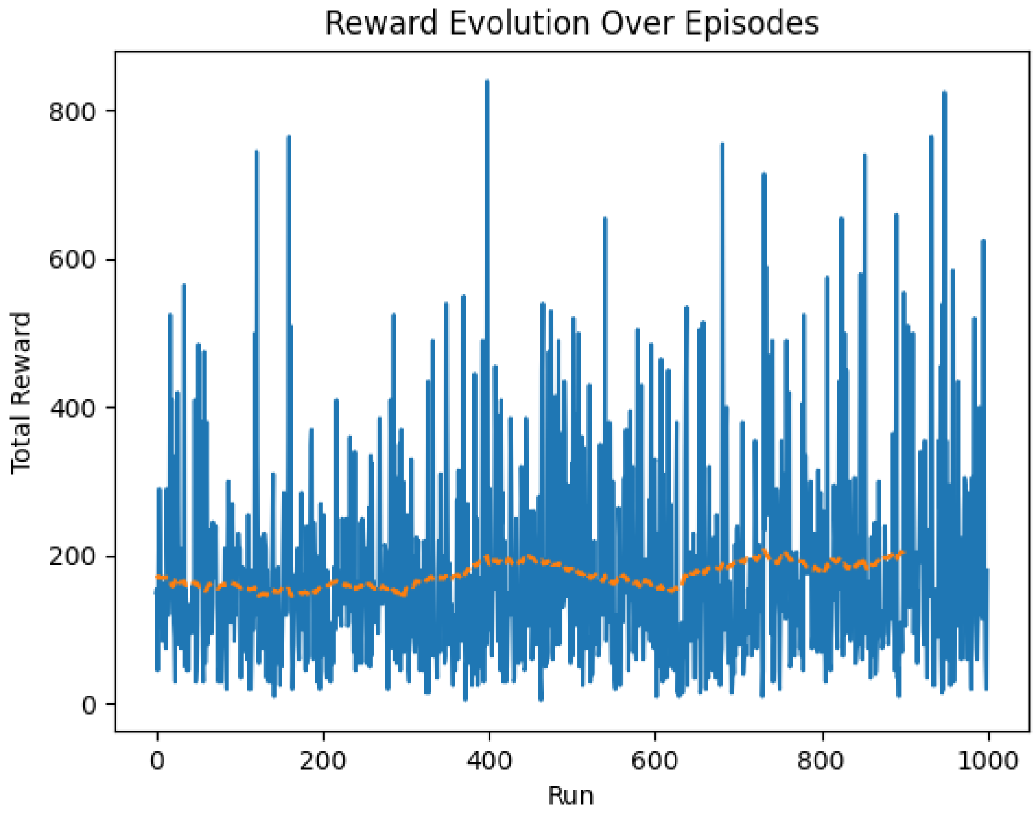

Figure 3: Single-NN DQN reward graph. The orange line represents the average reward of the last 100 runs compared to that run.

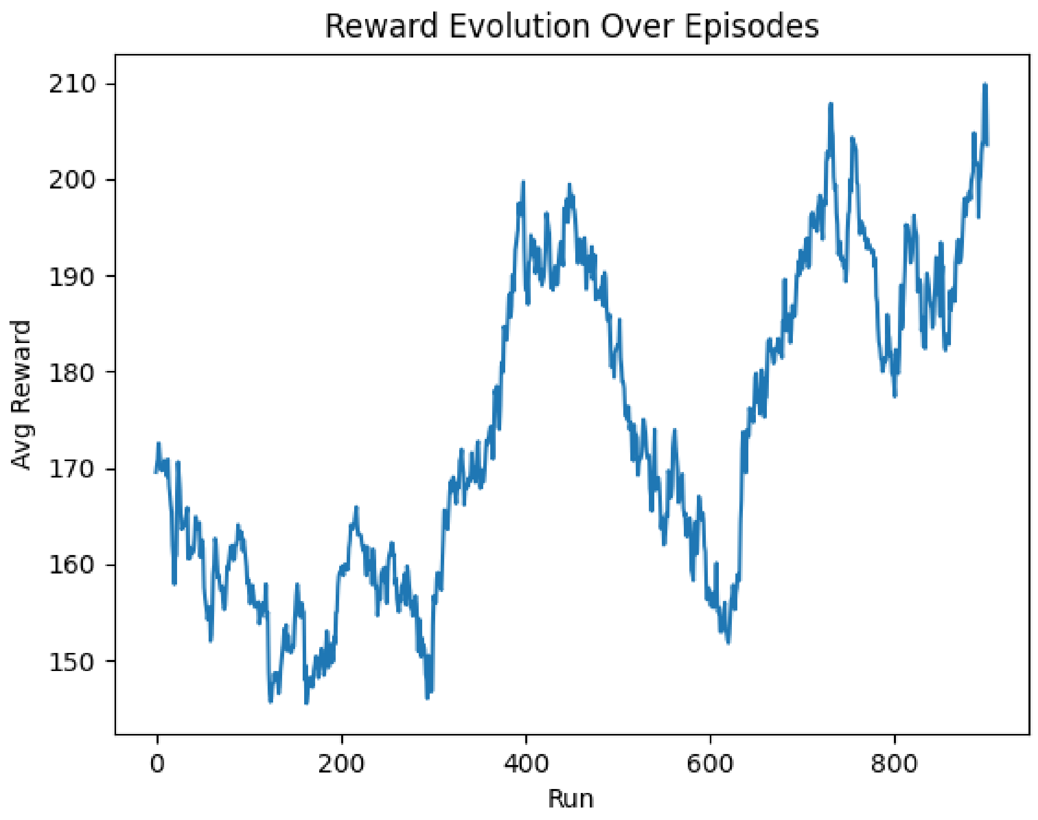

Figure 4: The average single-NN DQN reward plotted alone for a better understanding of its evolution. We can see the reward has converged at the end, but suffers from great ups and downs, hinting at an unstable learning.

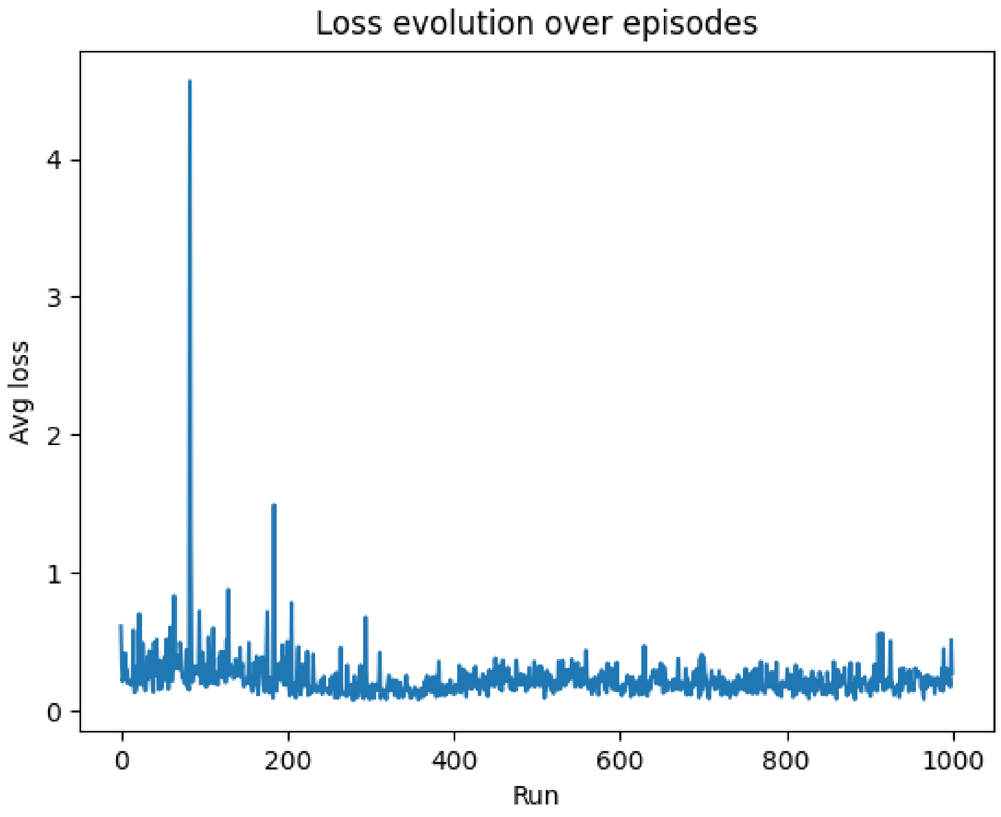

Figure 5: Avg loss evolution for single-NN DQN. A peak of more than 4 can be seen, which might signify instability.

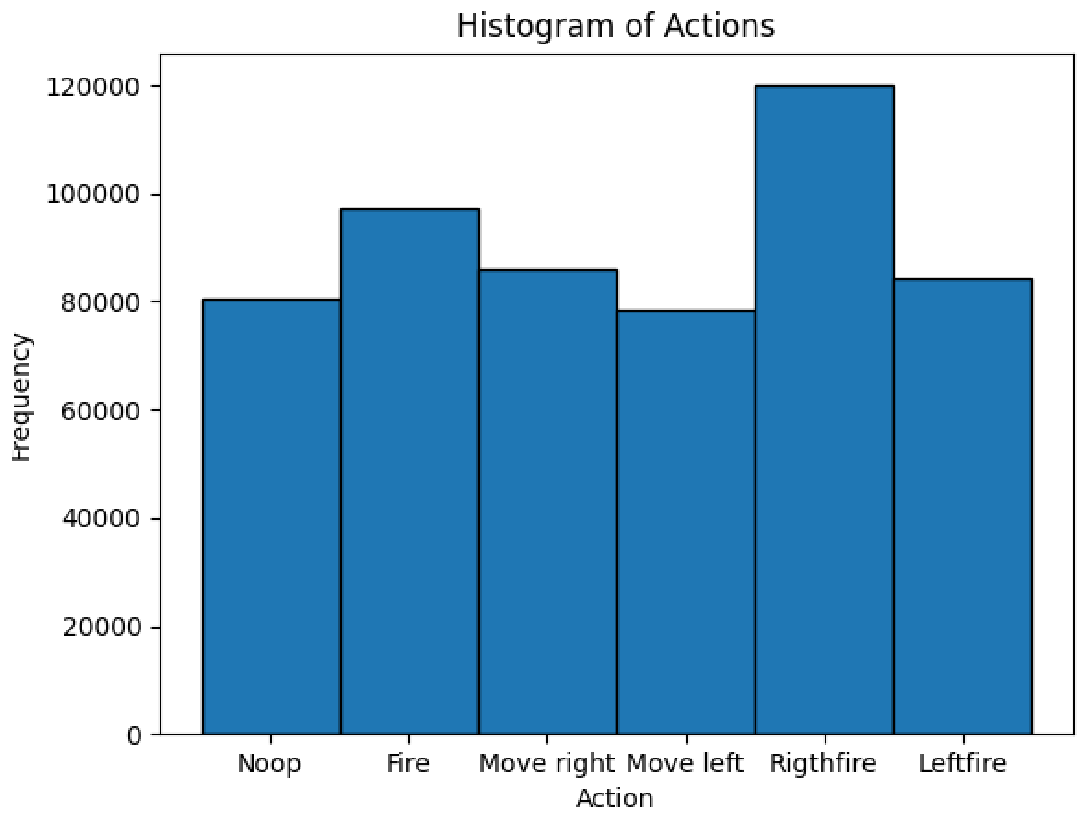

Figure 6: Histogram of actions to see if some action where used more than others: the ”right-fire” action was used significantly more often, hinting as some over-evaluation of this action during the exploit phase. Indeed the agent will take this action significantly more than others in the testing phases.

The plots of the same data for the Target-Model-DQN is available in the source code folder, and has not been published here since the graphs are very similar, and the overestimation problem is still very present, in fact, in the actions' histogram of the target-model DQN we can still see clearly that action "right-fire" is the most chosen action by the agent. However, as anticipated, the target model will now be used to solve the overestimation problem. We do also have one difference, that being the target-model DQN loss graph:

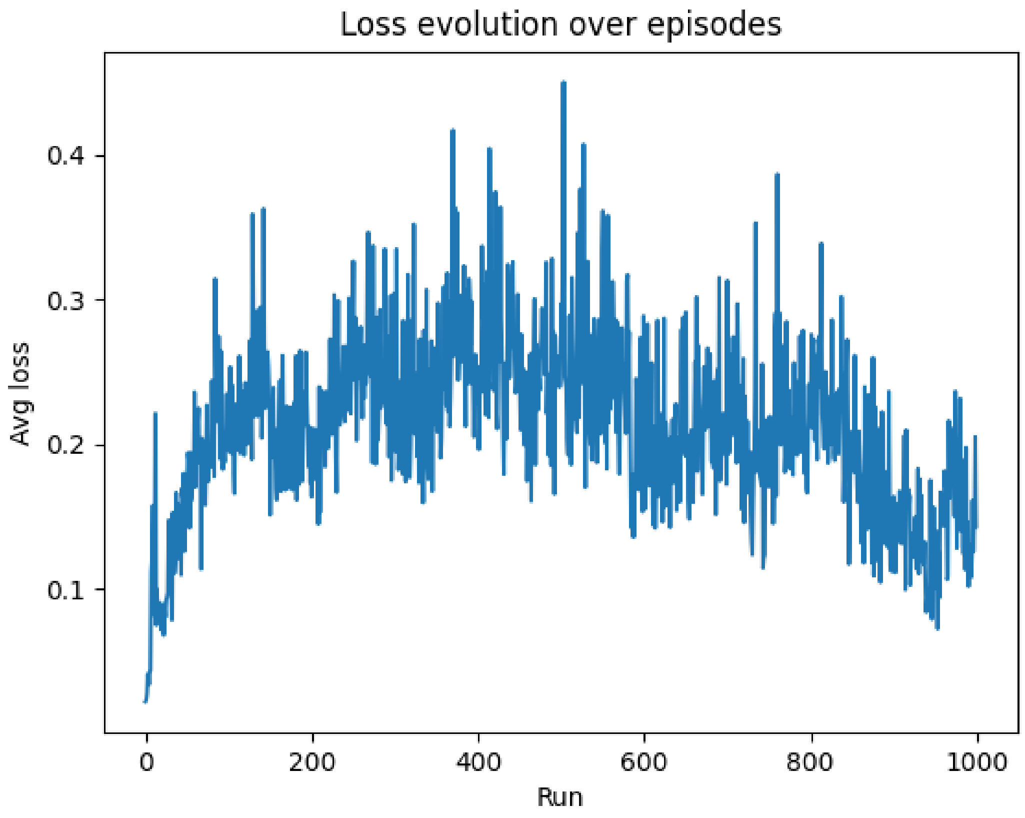

Figure 7: Average loss graph of the Target-Model DQN

As we can see, there are no peaks of loss that surpass 1, which, contrary to the single-NN DQN might be a sign of a stabler learning. More than one training was done, and, even if the number of times is not enough to confirm this trend, in each and every one of them the loss trend remained consistent.

4 Double DQN: addressing the problem of overestimation

In order to solve the overestimation of some actions, it is required to smooth out the noise even more. It turns out that in the equation for the target values, the max function is the problem: it consistently overestimates the values of some actions. In traditional DQN, the target values for training the Q-network are computed using the maximum Q-value of the next state. This is achieved by using the max function, which selects the action with the highest estimated Q-value for the next state. However, this approach tends to overestimate the Q-values of some actions. This overestimation bias occurs because the same Q-network is used both to select and evaluate the action, which can lead to an overemphasis on certain actions, particularly in noisy and stochastic environments. This means that noise gets amplified too and in our case this was a major cause of action overestimation, and it might take a long time for the agent to figure out better policies. Double DQN relies on decoupling the action selection from the action evaluation, so that these steps are performed by two different neural networks. In particular, the main NN will select the action and the target NN will evaluate it. This, in practical terms, means using the argmax function when the main NN chooses the action, and having the target network evaluate said action for the next states:

Traditional DQN Update Rule

$$y_t = (1-\alpha)Q(s_t, a_t) + \alpha(r_t + \gamma \max_{a'} Q(s_{t+1}, a'; \theta)) \tag{2}$$

Double DQN Update Rule

$$y_t = (1-\alpha)Q(s_t, a_t) + \alpha(r_t + \gamma Q(s_{t+1}, \arg\max_{a'} Q(s_{t+1}, a'; \theta); \theta^-)) \tag{3}$$

In the first equation, the target value y_t is computed normally, by utilizing the Q-value for state s_t and chosen action a_t, the reward r_t, and the max Q-value for the next state's (s_{t+1}) action to update our estimation. θ represents the parameters of the main NN. In the second equation, we input into the target model with parameters θ^- the output action from the main NN (that is why we use the *argmax* function) estimated for the next states, and thus use the target NN to estimate the Q-value for such action.

Notice: the Double DQN approach is not the same as just using a target model. In fact, while the target model would just compute the maximum Q-value for the next action using the traditional update rule, the Double DQN approach purposefully decouples the action selection from the action evaluation for the next state. In contrast, the max function implicitly makes the same NN perform these two steps, may it be the main NN or the target model when the target model is used.

```python
class DDQNAgent:

    ...same code of normal DQN...

    def train(self):

        batch = self.memory.sample(self.batch_size)
        states, actions, rewards, next_states, dones = zip(*batch)

        states = torch.FloatTensor(states)
        next_states = torch.FloatTensor(next_states)
        rewards = torch.FloatTensor(rewards)
        dones = torch.FloatTensor(dones)

        q_values = self.model(states)
        q_values = q_values.gather(1, torch.LongTensor(actions).unsqueeze(1)).squeeze(1)

        # Double DQN Implementation
        #Decoupling of action selection and action evaluation: the main network now will estimate
        #the best index and feed it to the target model for evaluation
        next_action_indices = self.model(next_states).max(1)[1] # Get the indices of the actions
        #with the max Q-value from the primary model, max(*)[1] gives index of max value
        max_next_q_values = self.target_model(next_states).gather(1, next_action_indices.unsqueeze(1)
        .squeeze(1)
        target_q_values = (1 - self.alpha ) * q_values + self.alpha * (rewards + self.gamma *
        max_next_q_values * (1 - dones))

        loss = self.loss_fn(q_values, target_q_values.detach())

        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()
        return loss.item()
```

The only real modification is in the training function, specifically in the target Q-values calculation.

4.1 The results

The results are surprising, as now the DoubleDQN agent actually chooses a plethora of different actions each time. We can see also the average reward converging more clearly to around 190 points, indicating a stabler learning technique. This is confirmed by the loss as well. The videos of the agent playing the game are available in the source code.

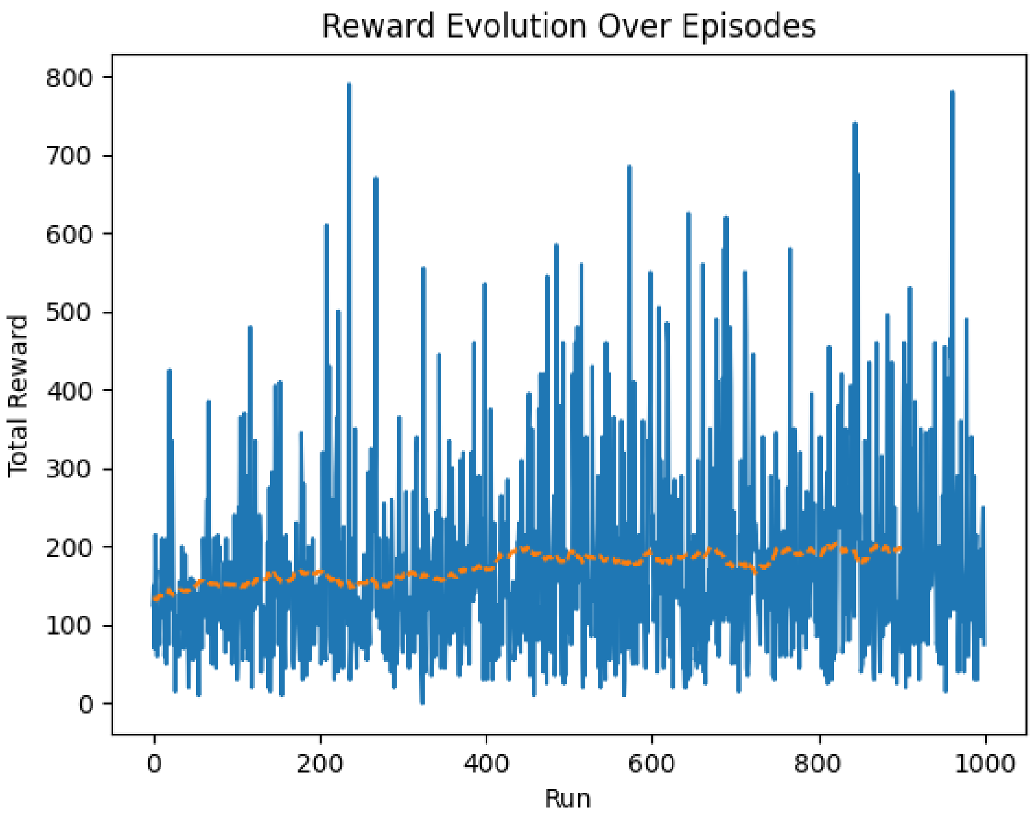

Figure 8: DoubleDQN agent reward evolution. We can see clearly more stability in the average reward evolution.

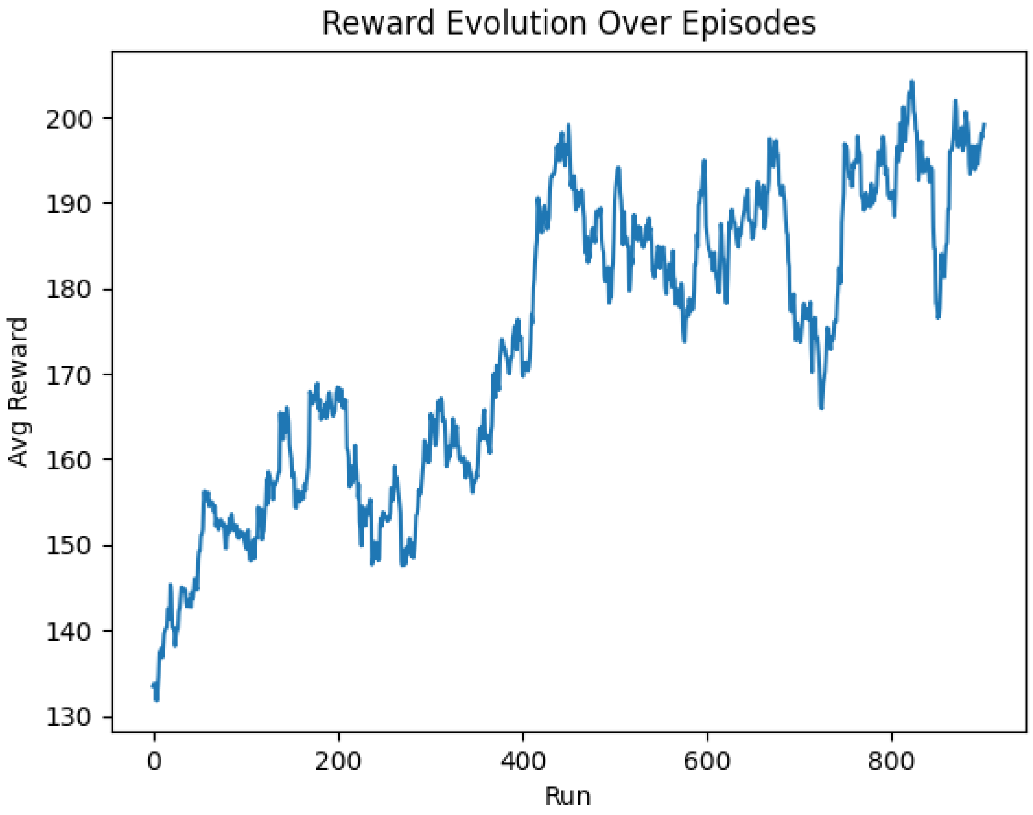

Figure 9: A zoom-in on the average reward evolution reveals more stable convergence with respect to the single-NN DQN. The convergence might also be faster, however a sample of many trainings would be required to confirm this as a trend.

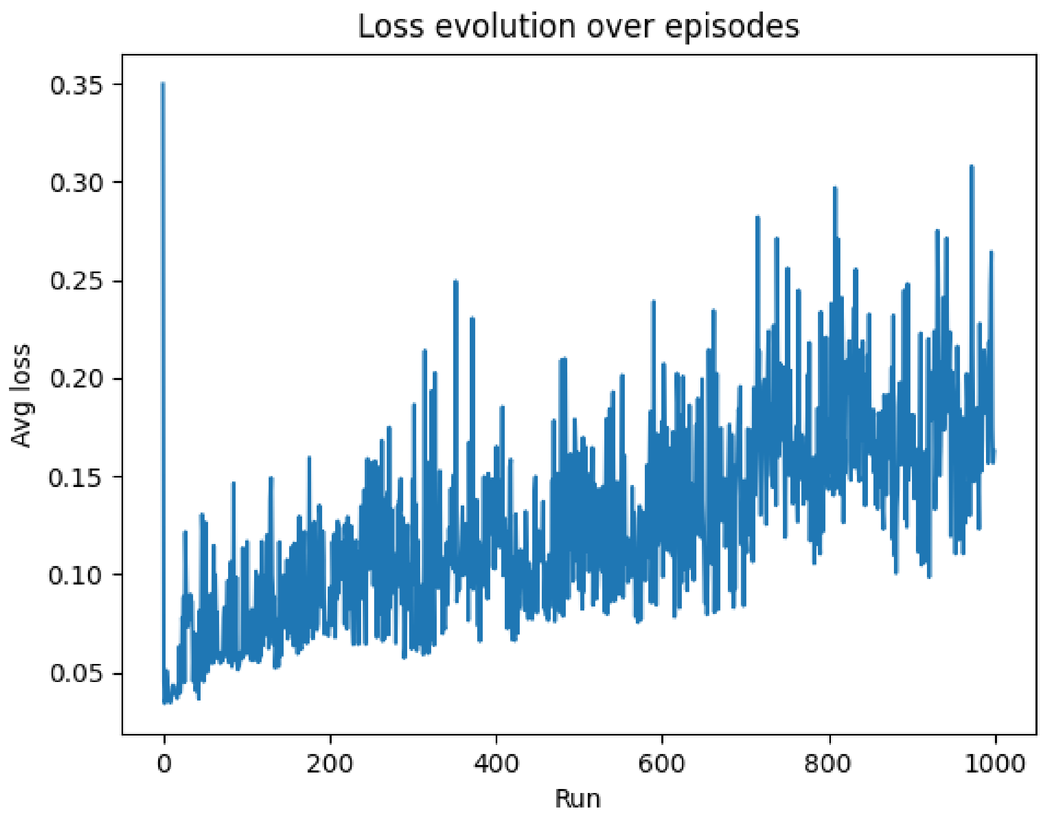

Figure 10: The loss is also stabler, as there are no peaks that surpass 1, in contrast to the single-NN DQN. Also, again the loss sits at around 0.20 at the end of the training, just like the Target-Model DQN did.

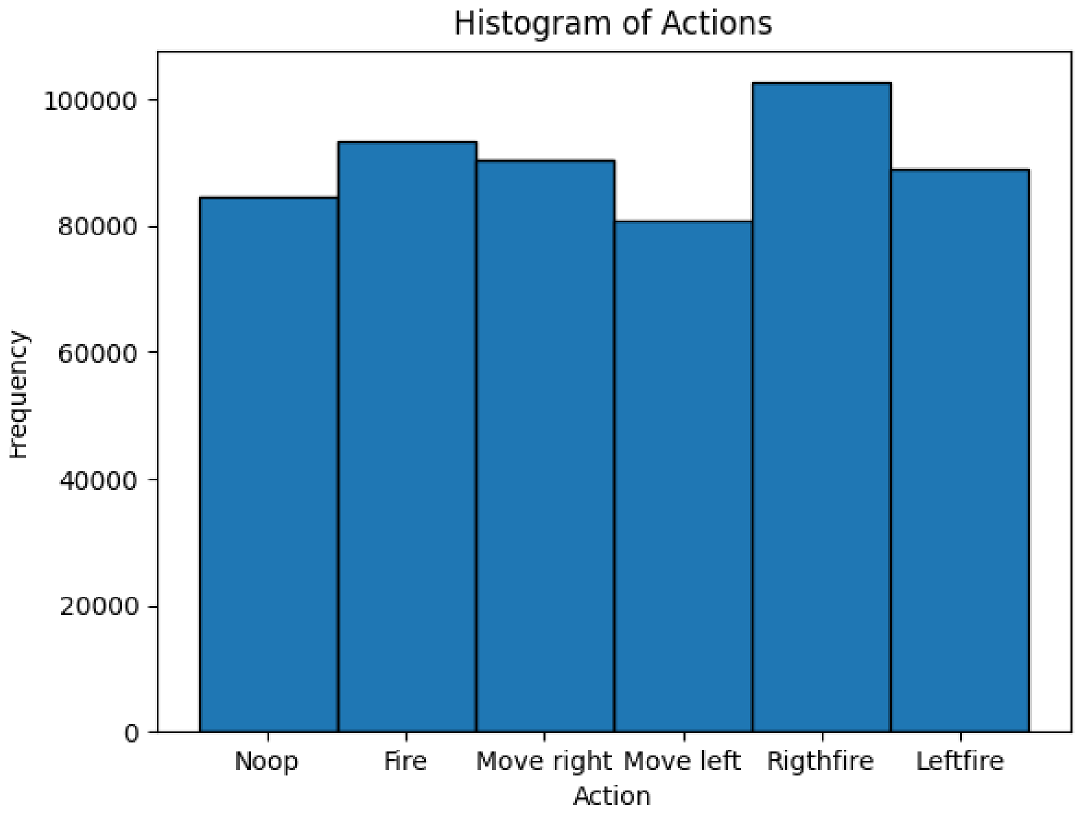

Figure 11: Even the histogram of actions shows a better trend with respect to the agent not choosing an action much more often than all the other actions during training.

4.2 Q-Table vs Traditional DQN vs Double DQN

It's clear from the videos that the Double DQN is a superior alternative, thanks to it solving the overestimation problem: the agent can now be seen actively pursuing the purple ship that awards more points than the other enemies, dodging enemy shots and exploiting the defenses to avoid being hit. In double DQN the agent will also sometimes try to remain on the left of the map, waiting for the leftmost column of enemies to be right on its line of fire so that it can eliminate that first column almost immediately, and at the same time making small movements to the right (under the shield) and back to the left when an enemy shot is approaching in order to dodge said shot. Another tactic adopted by the agent in both Traditional and Double DQN techniques, is that the agent will shoot a hole through its defenses and exploit that small gap to keep shooting the enemies whilst remaining in cover at the same time. Also, turning on the mode that reveals the agent's shots will reveal that the agent will start shooting more often if a column of enemies is directly in its line of fire, which is understandable since in that way it will take out many enemies in little time. In any case, the overall performance of the agent does not change much: the agent still scores about 200 points before losing the game, so this is not much different from when the agent overestimated the "right-fire" action. However, comparing the results with the Q-Table it's apparent that the DQN is a much better alternative, since whilst the Q-Table reaches around 170 as average reward with a lot more training than the DQN, the DQN is able to reach 190 or 200 as average reward in comparatively little time: the DQN employs circa 45 to 60 minutes of training vs the 6 hours of training needed by the Q-Table. Also, the Q-Table is expensive to train and store memory-wise, while the DQN is a much more agile approach.