Reinforcement Learning
DQN for Pong – Project Report

Elise Chin, Sanjith Bonela

May 23, 2021

1 Deep Q-learning method

The deep Q-learning method was first introduced by Deepmind in 2013 [2] and then further improved and elaborated in 2015 [1]. The algorithm is an enhancement of the classic RL algorithm called Q-learning with neural networks used as a function approximator of the optimal state-action value function.

1.1 Q-Learning

Q-learning is an algorithm to find an estimate Q of the optimal state-action value function q_* from experience. The q-function of a policy π, $q_\pi(s, a)$, corresponds to the expected return or discounted sum of rewards of taking the action a at state s first, and then following the policy π. In other words, it tells us how good it is to take this specific action at that particular state. The optimal q-function is then defined as the maximum return that can be obtained starting from observation s, taking action a, and then following the optimal policy π_*. It obeys the Bellman optimality equation:

$$q_*(s, a) = \mathbb{E}[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a')|S_t = s, A_t = a]$$

where R_{t+1} is the immediate reward, S_{t+1} the next state after taking action a from state s at time t.

The Q-learning update is based on the Bellman optimality equation and is defined as:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t))$$

and it can be shown that this converges to the optimal action-value function $q_*(s, a)$ as $N(s, a) \to \infty$ if the step size α decreases towards 0 with a suitable rate.

1.2 Deep Q-Learning

In Q-learning, a memory table Q[s,a] is built to store all q-values for all possible combinations of states and actions. However, if the state or action space is too large, the memory and the computation requirement for Q-learning will be very high. To address this problem, one can train a function approximator, such as a neural network with paramaters $\mathbf{w}$, to estimate the q-values, i.e. $Q(s, a, \mathbf{w}) \approx q_*(s, a)$. This can be done by minimizing the mean squared error at each step t:

$$L(\mathbf{w}_t) = \mathbb{E}[(U_t - Q(S_t, A_t, \mathbf{w}_t))^2]$$

where $U_t = R_{t+1} + \gamma \max_a Q(S_{t+1}, a, \mathbf{w}_t)$ is the TD-target. The loss function can be minimized using Stochastic (semi-)gradient descent:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha[U_t - Q(S_t, A_t, \mathbf{w}_t)]\nabla Q(S_t, A_t, \mathbf{w}_t)$$

However, the training will be unstable if we apply the standard Q-learning update naively. As we can see in the formula, there are some correlation between successive updates, and the estimate Q and the target are not completely independent from each other.

To address that, two ideas were introduced in the original paper [1].

1. Experience replay: store the agent's experience at each time step t, $e_t = (S_t, A_t, R_t, S_{t+1})$, in a replay buffer and sample mini-batches from it to train the network, instead of only using the last transition to compute the loss and its gradient. This has two advantages: better stability since the transitions in a batch are uncorrelated, and better data efficiency since an experience can be used in many weight updates.
2. Target network: use a separate network to estimate the TD-target. The target network has the same architecture as the function approximator but with frozen weights for a number of steps. More precisely, every C steps (a hyperparameter), the parameters of the Q-network is copied to the target network, and the latter is used to compute the TD-target for C steps. This adds more stability to the training since the target function is fixed for a certain time.

2 DQN for CartPole-v0

Since DQN can be tricky to debug, we first implement it to solve a simple environment, the CartPole-v0.

2.1 Description of CartPole-v0

A pole is attached to a cart, which can move along a frictionless track. The pole starts upright and the goal is to prevent it from falling over by controlling the cart.[1]

- The observation from the environment is a 4D vector representing the position and velocity of the cart, and the angle and angular velocity of the pole.
- The agent can control the system by taking one of 2 actions: push the cart right or left.
- A reward of 1 is provided for every timestep that the pole remains upright.
- The episode ends when one of the following is true:
  - the pole tips over some angle limit
  - the cart moves outside of the world edges
  - 200 time steps pass

The goal of the agent is to find a policy as to maximize the expected discounted return $G = \sum_{t=0}^{T} \gamma^t R_{t+1}$ of each episode, where $\gamma \in (0, 1]$ is the discounted factor. If the agent has been trained well, an episode should end by satisfying the third condition. Hence, the final agent should have a mean return of +200.0 over several episodes, which corresponds to receiving a reward of 1 for 200 time steps.

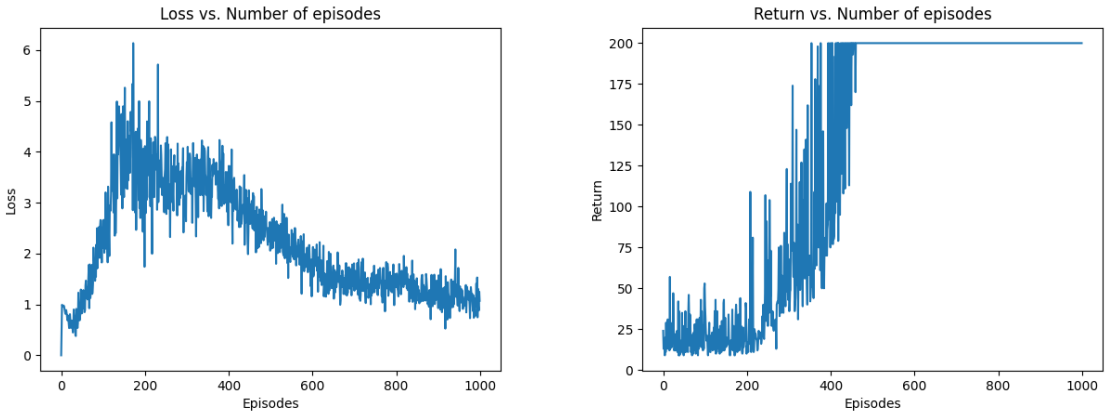
2.2 Results with default hyperparameters

We use the following hyperparameters to train the DQN for 1000 episodes.

[1] Description from https://www.tensorflow.org/agents/tutorials/0_intro_rl, https://gym.openai.com/envs/CartPole-v0/

Hyperparameter	Value	Description
Memory size	50000	Maximum size of the replay buffer.
Batch size	32	Size of mini-batches sampled from the replay buffer.
Target update frequency	100	The frequency (measured in time steps) with which the target network is updated.
Train frequency	1	The frequency (measured in time steps) with which the network is trained.
Gamma	0.95	Discounted factor gamma used in the Q-learning update.
Learning rate	0.0001	The learning rate used by Adam optimizer.
Epsilon start	1	Initial value of ϵ-greedy exploration.
Epsilon end	0.05	Final value of ϵ-greedy exploration.
Anneal length	10^4	Value to compute the number of steps to anneal epsilon for `eps_step = (eps_start - eps_end) / anneal_length`

Below, we plot episode loss and undiscounted return over the course of training. After increasing a bit, the loss slowly starts to decrease after 200 episodes, suggesting that the network is learning. The loss seems to continue to decrease if we ran for a greater number of episodes but it was not necessary to do so as we can see on the return plot that the agent has already reached a mean return of 200 after 400 episodes.



(a) (b)

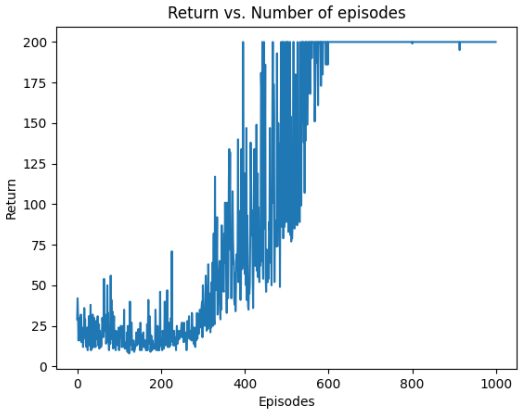
Figure 1: (a) The loss vs. the number of episodes. The network is learning well since the loss is decreasing. (b) The undiscounted return vs. the number of episodes. During the first episodes, the return is low since the agent is starting to learn, then it is increasing until it reached the maximum return which can be obtained in the CartPole-v0 environment.

2.3 Varying the target update frequency

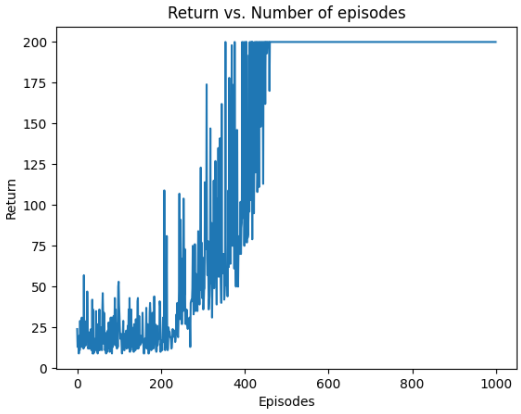
In the CartPole-v0 environment, the reward provided at each time step is 1, so the undiscounted return for an episode also corresponds to the number of steps of that episode. With a target update frequency of 100, we update the target network in the middle of an episode in average. What happens if we set this hyperparameter to a low or large value?

Below, we plot the undiscounted return over 1000 episodes, with different target update frequency (10, 100, 200, and 1000). Firstly, we can observe that the final agent always reach a return of 200 after 400 episodes regardless of the target update frequency. Secondly, the larger the target frequency is, the faster the agent learns. As we mentioned earlier in Section 2, using a separate network to compute the TD-target adds more stability to the learning. This observation can be seen in our plots where stability is reflected in the degree of fluctuation of the curves. When the target update frequency is small, the

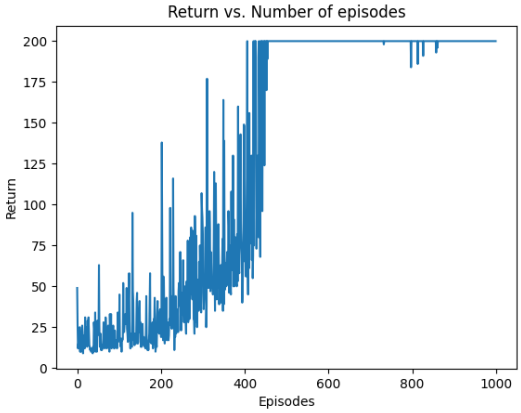
return's curve fluctuates a lot before 400 episodes. The training is more stable when the target update frequency is large because the target network is fixed for a longer period of time.



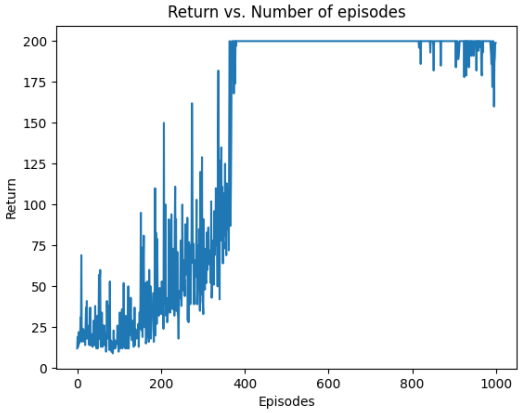
(a) target_update_frequency = 10



(b) target_update_frequency = 100



(c) target_update_frequency = 200

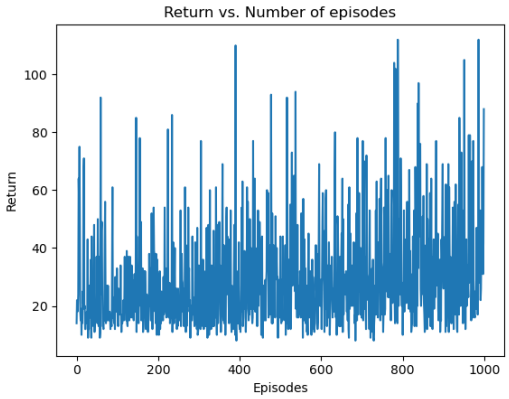


(d) target_update_frequency = 1000

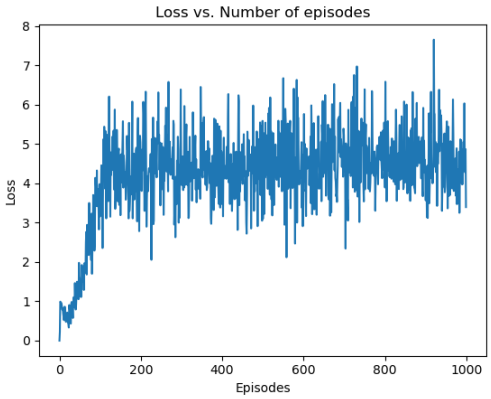
Figure 2: Undiscounted return over 1000 episodes, with different target update frequency.

2.4 Exploring different exploration parameters

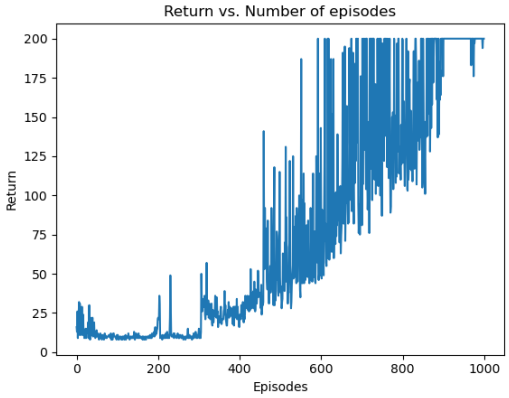
We had also explored the results with varying different hyperparameters. This exploration was concentrated around change of epsilon by varying the anneal length (which varies the decay in epsilon) in one case and exploration ($\epsilon = 1$, picks random action) and exploitation ($\epsilon = 0.05$, picks action based on max values of q) concepts in another case. The graphs are plotted below in figure [3].



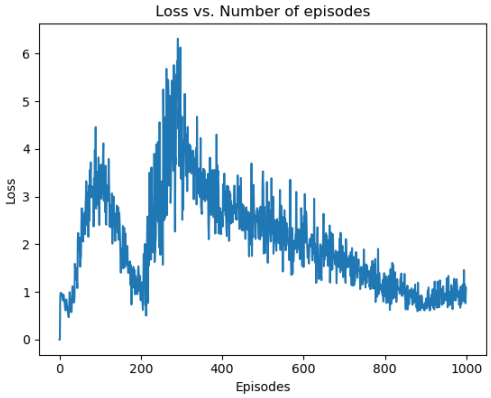
(a) Return vs. Number of episodes with anneal_length 1e-5



(b) Loss vs. Number of episodes with anneal_length 1e-5



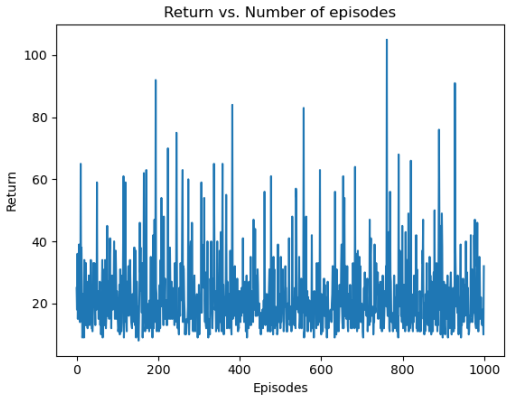
(c) Return vs. Number of episodes with anneal_length 1e-3



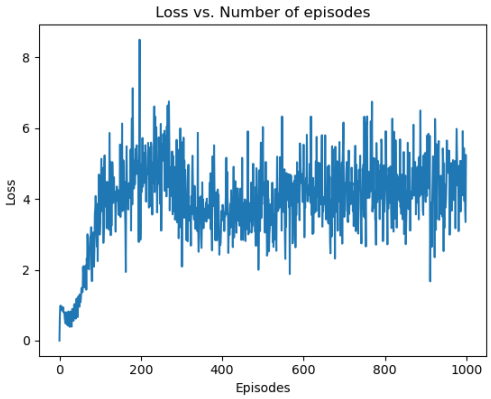
(d) Loss vs. Number of episodes with anneal_length 1e-3

Figure 3: Plots for losses and returns vs Number of episodes with varying anneal lengths

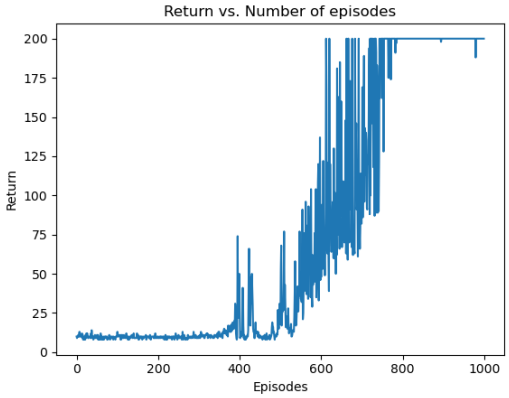
From the plots in figure [3], we can see for very low anneal length (1e-5) (fig 3a, 3b), the loss was almost constant after around 100 episodes and the mean return was not as high as expected for 1000 episodes which may require some more training. On the contrary, if the anneal length was high (1e-3), the loss was constantly decreasing after around 300 episodes and the loss-episode behaviour prior to 300 episodes may account in random-action pick policy (exploration) instead of picking action that maximizes q-values and the return obtained was 200 (maximum) at around 900 episodes.



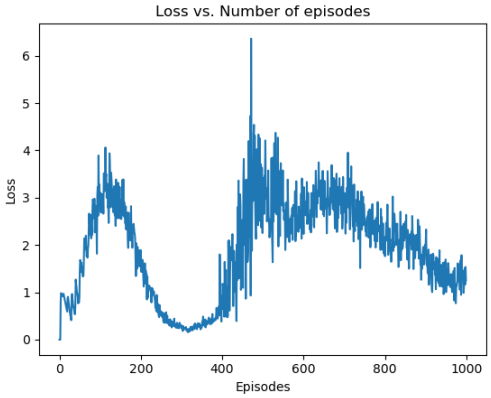
(a) Return vs Number of episodes with $\epsilon = 1$



(b) Loss vs Number of episodes with $\epsilon = 1$



(c) Return vs Number of episodes with $\epsilon = 0.05$



(d) Loss vs Number of episodes with $\epsilon = 0.05$

Figure 4: Plots for losses and returns vs Number of episodes for exploration and exploitation respectively

From the plots in the figures [4a, 4b], i.e, $\epsilon = 1$, the loss did not varied significantly and the returns fluctuated a lot because the action was always picked up randomly and this would account to this behaviour. On the contrary, in the figures [4c, 4d], when $\epsilon = 0.05$, after around 800 episodes the return was constant (200) and the loss was reducing after around 600 episodes.

3 DQN for Pong

3.1 Description of Pong-v0

Pong is 2D tennis table-themed arcade video game, manufactured by Atari and originally released in 1972. In Pong, one player scores if the ball passes by the other player. An episode is over when one of the players reaches 21 points. In the OpenAI Gym framework version of Pong[2], the agent is displayed on the right and the enemy on the left.

- The observation from the environment is an RGB-image of size (210, 160, 3) with 128 possible colors for each pixel.
- The agent has 6 possible actions (`['NOOP', 'FIRE', 'RIGHT', 'LEFT', 'RIGHTFIRE', 'LEFTFIRE']`) but 3 of them are redundant: actions 2/4 and 3/5 makes the paddle go down and up respectively, while action 0/1 do nothing.

[2] https://gym.openai.com/envs/Pong-v0/

- The agent receives a +1 reward if the ball went past the opponent, a -1 reward if it missed the ball, and 0 otherwise.

3.2 Implementation

The environment was Atari preprocessed, with a frame skip parameter 4 and the image was converted into grayscale image for further processing. Now each frame (observation) was converted into a pytorch tensor and proceeds into the further operations. An observation stack of size 4 was implemented where the latest observation was appended at the last index and the rest of the implementation proceeds just as in CartPole-v0 implementation.

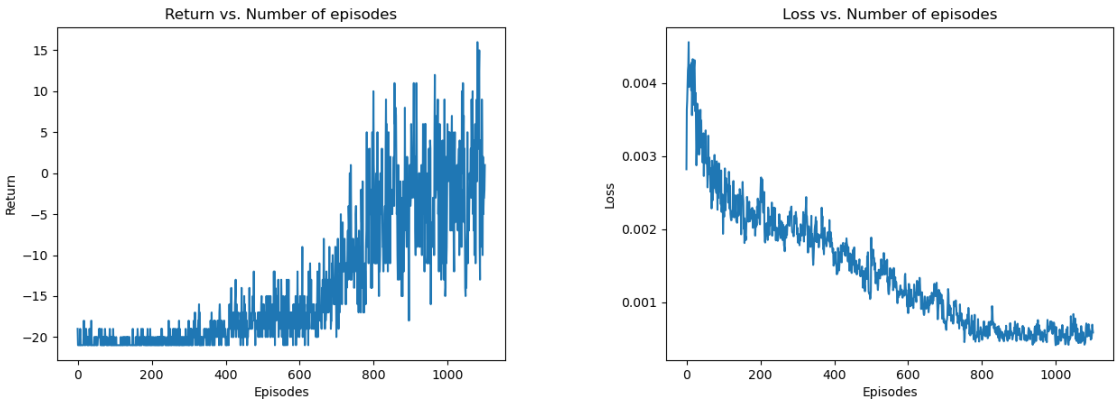
As mentioned above some of the actions were redundant and so the actions were mapped from 0 to 2 and 1 to 3.

3.3 Results and discussion

We aimed to train the model for 10000 episodes and used the default hyperparameters to train the DQN. The hyperparameters are described in the table below.

Hyperparameter	Value	Description
Screen size	84	Resizing the frame.
Grayscale Obs	True	Returns grayscale image.
Frame Skip	1	Frequency at which agent experiences game.
Noop Max	30	Maximum number of no-ops.
Scale Obs	True	Returns the observation in range [0, 1].
Memory size	10000	Maximum size of the replay buffer.
Observation stack	4	Size of observation stack.
Batch size	32	Size of mini-batches sampled from the replay buffer.
Target update frequency	1000	The frequency (measured in time steps) with which the target network is updated.
Train frequency	4	The frequency (measured in time steps) with which the network is trained.
Gamma	0.99	Discounted factor gamma used in the Q-learning update.
Learning rate	0.0001	The learning rate used by Adam optimizer.
Epsilon start	1	Initial value of ϵ-greedy exploration.
Epsilon end	0.01	Final value of ϵ-greedy exploration.
Anneal length	10^6	Value to compute the number of steps to anneal epsilon for `eps_step = (eps_start - eps_end) / anneal_length`
No. of actions	2	Number of actions the agent takes

The plots for training losses and returns during the course of training are below.



(a) Return vs Number of episodes

(b) Loss vs Number of episodes

Figure 5: Losses and returns of Pong agent over 1200 episodes

At the end of 1200 episodes, our agent was able to reach a best mean return of 15.6, which is a decent value. The loss is decreasing over the course of the training, going from 0.004 to less than 0.001, suggesting that our agent is learning. Looking at the return graph, we can see that the training is becoming more and more unstable after 600 episodes: the curve fluctuates a lot. This illustrates the fact that using a neural network in reinforcement learning adds unstability to the learning.

There were some few challenges we have faced while running this program:

- While testing the program, we used GPU provided in Google Colab but unfortunately GPU limit usage got exceeded forcing us to work on our local machines which took almost 8 hours to run 1000 episodes.
- We aimed to run for 10000 episodes but after certain number of episodes our systems became unresponsive and hence we ran for 1200 episodes and we attained a decent best mean return of 15.6 which had a significant improvement in return at around 300 episodes while training.

4 Appendix

```
"""
In this file, you may edit the hyperparameters used for different environments.

memory_size: Maximum size of the replay memory.
n_episodes: Number of episodes to train for.
batch_size: Batch size used for training DQN.
target_update_frequency: How often to update the target network.
train_frequency: How often to train the DQN.
gamma: Discount factor.
lr: Learning rate used for optimizer.
eps_start: Starting value for epsilon (linear annealing).
eps_end: Final value for epsilon (linear annealing).
anneal_length: How many steps to anneal epsilon for.
n_actions: The number of actions can easily be accessed with env.action_space.n, but we do
    some manual engineering to account for the fact that Pong has duplicate actions.
"""
```

```

# Hyperparameters for CartPole-v0
CartPole = {
    'memory_size': 50000,
    'n_episodes': 1000,
    'batch_size': 32,
    'target_update_frequency': 100,
    'train_frequency': 1,
    'gamma': 0.95,
    'lr': 1e-4,
    'eps_start': 1,
    'eps_end': 0.05,
    'anneal_length': 10**4,
    'n_actions': 2,
}

# Hyperparameters for Pong-v0
Pong = {
    'obs_stack_size': 4,
    'memory_size': 10000,
    'n_episodes': 10000,
    'batch_size': 32,
    'target_update_frequency': 1000,
    'train_frequency': 4,
    'gamma': 0.99,
    'lr': 1e-4,
    'eps_start': 1.0,
    'eps_end': 0.01,
    'anneal_length': 10**6,
    'n_actions': 2,
}
```

Listing 1: config.py

```
import random

import gym
import torch
import torch.nn as nn
import torch.nn.functional as F

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")


class ReplayMemory:
    def __init__(self, capacity):
        self.capacity = capacity
        self.memory = []
        self.position = 0

    def __len__(self):
        return len(self.memory)

    def push(self, obs, action, next_obs, reward):
        if len(self.memory) < self.capacity:
            self.memory.append(None)

        self.memory[self.position] = (obs, action, next_obs, reward)
        self.position = (self.position + 1) % self.capacity
```

```
26
27      def sample(self, batch_size):
28          """
29          Samples batch_size transitions from the replay memory and
    returns a tuple
30              (obs, action, next_obs, reward)
31          """
32          sample = random.sample(self.memory, batch_size)
33          return tuple(zip(*sample)) # ((obs1, obs2), (action1, action2),
    ...)
34
35
36  class DQN(nn.Module):
37      def __init__(self, env_name, env_config):
38          super(DQN, self).__init__()
39
40          self.env_name = env_name
41
42          # Save hyperparameters needed in the DQN class.
43          self.batch_size = env_config["batch_size"]
44          self.gamma = env_config["gamma"]
45          self.eps_start = env_config["eps_start"]
46          self.eps_end = env_config["eps_end"]
47          self.anneal_length = env_config["anneal_length"]
48          self.n_actions = env_config["n_actions"]
49
50          if self.env_name == 'CartPole-v0':
51              self.fc1 = nn.Linear(4, 256)
52              self.fc2 = nn.Linear(256, self.n_actions)
53          elif self.env_name == 'Pong-v0':
54              self.conv1 = nn.Conv2d(4, 32, kernel_size=8, stride=4,
    padding=0)
55              self.conv2 = nn.Conv2d(32, 64, kernel_size=4, stride=2,
    padding=0)
56              self.conv3 = nn.Conv2d(64, 64, kernel_size=3, stride=1,
    padding=0)
57              self.fc1 = nn.Linear(3136, 512)
58              self.fc2 = nn.Linear(512, self.n_actions)
59
60          self.relu = nn.ReLU()
61          self.flatten = nn.Flatten()
62
63      def forward(self, x):
64          """Runs the forward pass of the NN depending on architecture."""
65
66          if self.env_name == 'CartPole-v0':
67              x = self.relu(self.fc1(x))
68              x = self.fc2(x)
69          if self.env_name == 'Pong-v0':
70              x = self.relu(self.conv1(x))
71              x = self.relu(self.conv2(x))
72              x = self.relu(self.conv3(x))
73              x = self.flatten(x)
74              x = self.relu(self.fc1(x))
75              x = self.fc2(x)
76
77          return x
78
```

```
    def act(self, observation, epsilon, exploit=False):
        """Selects an action with an epsilon-greedy exploration strategy.

        Args:
            observation (tensor): observation tensor of shape [batch_size, state_dimension] (e.g. [32, 4])
            epsilon (int)
            exploit (bool): if True, acts greedily

        Returns:
            actions (tensor): tensor of actions according to the epsilon-greedy strategy, of shape [batch_size, 1]
        """

        batch_size = observation.shape[0]
        rand_value = random.random()

        if exploit or rand_value > epsilon:
            # Choose the action which gives the largest Q-values for each observation
            with torch.no_grad():
                output = self.forward(observation)
                actions = torch.argmax(output, dim=1)

                if self.env_name == "Pong-v0":
                    if actions.item() == 0:
                        actions = torch.tensor([2]) # right
                    else:
                        actions = torch.tensor([3]) # left
        else:
            # Choose a random action for each observation
            if self.env_name == "CartPole-v0":
                random_actions = [random.randrange(self.n_actions) for _ in range(batch_size)]
            if self.env_name == "Pong-v0":
                random_actions = [random.randrange(2, 4) for _ in range(batch_size)]
            actions = torch.tensor(random_actions)
        return actions.unsqueeze(1)

def optimize(dqn, target_dqn, memory, optimizer):
    """This function samples a batch from the replay buffer and optimizes the Q-network."""

    # If we don't have enough transitions stored yet, we don't train.
    if len(memory) < dqn.batch_size:
        return 0

    # Sample a batch from the replay memory and concatenate so that there are
    # four tensors in total: observations, actions, next observations and rewards.
    observations, actions, next_observations, rewards = memory.sample(dqn.batch_size)

    # Transform every tuple into tensors.
    observations = torch.cat(observations)
```

```
127
128     non_terminal_next_observations = [next_obs for next_obs in
        next_observations if next_obs is not None]
129     non_terminal_next_observations = torch.cat(
        non_terminal_next_observations).float()
130
131     actions = torch.stack(actions, dim=0)
132     actions = torch.unsqueeze(actions, 1)
133
134     rewards = torch.stack(rewards, dim=0)
135     rewards = torch.unsqueeze(rewards, 1)
136
137     # Compute the current estimates of the Q-values for each state-
        action pair (s,a).
138     output = dqn.forward(observations)
139     output = output.to(device)
140     if dqn.env_name == "CartPole-v0":
141         q_values = torch.gather(input=output, index=actions, dim=1)
142     if dqn.env_name == "Pong-v0":
143         q_values = torch.gather(input=output, index=actions - 2, dim=1)
144
145     # Compute the Q-value targets.
146     non_terminal_mask = torch.BoolTensor(list(map(lambda obs: obs is not
        None, next_observations))) # indices for non terminal transitions
147     terminal_mask = ~non_terminal_mask # indices for terminal
        transitions
148     target_dqn_qpred = target_dqn.forward(non_terminal_next_observations
        )
149     target_dqn_qpred_max = torch.max(target_dqn_qpred, axis=1)[0].
        unsqueeze(1)
150
151     q_value_targets = torch.zeros(dqn.batch_size, 1)
152     q_value_targets = q_value_targets.to(device)
153     q_value_targets[non_terminal_mask] = rewards[non_terminal_mask] +
        dqn.gamma * target_dqn_qpred_max
154     q_value_targets[terminal_mask] = rewards[terminal_mask]
155
156     # Compute loss.
157     loss = F.mse_loss(q_values, q_value_targets)
158
159     # Perform gradient descent.
160     optimizer.zero_grad()
161
162     loss.backward()
163     optimizer.step()
164
165     return loss.item()
```

Listing 2: dqn.py

```
1 import argparse
2 import random
3
4 import gym
5 import torch
6 import torch.nn as nn
7
8 import config
9 from utils import preprocess
```

```
10
11 device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
12
13 parser = argparse.ArgumentParser()
14 parser.add_argument('--env', default='Pong-v0', choices=['CartPole-v0',
       'Pong-v0'])
15 parser.add_argument('--path', type=str, help='Path to stored DQN model.'
       )
16 parser.add_argument('--n_eval_episodes', type=int, default=1, help='
       Number of evaluation episodes.', nargs='?')
17 parser.add_argument('--render', dest='render', action='store_true', help
       ='Render the environment.')
18 parser.add_argument('--save_video', dest='save_video', action='
       store_true', help='Save the episodes as video.')
19 parser.set_defaults(render=False)
20 parser.set_defaults(save_video=False)
21
22 # Hyperparameter configurations for different environments. See config.
       py.
23 ENV_CONFIGS = {
24     'CartPole-v0': config.CartPole,
25     'Pong-v0': config.Pong
26 }
27
28 def evaluate_policy(dqn, env, env_config, args, eps_start, n_episodes,
       discounted=False, render=False, verbose=False):
29     """Runs {n_episodes} episodes to evaluate current policy."""
30     total_return = 0
31
32     for i in range(n_episodes):
33         obs = preprocess(env.reset(), env=args.env).unsqueeze(0)
34         if args.env == "Pong-v0":
35             obs_stack = torch.cat(env_config["obs_stack_size"] * [obs]).
       unsqueeze(0).to(device)
36
37         done = False
38         rewards_list = []
39
40         while not done:
41             if render:
42                 env.render()
43
44             if args.env == "Pong-v0":
45                 action = dqn.act(obs_stack, eps_start, exploit=True).
       item()
46             else:
47                 action = dqn.act(obs, eps_start, exploit=True).item()
48
49             obs, reward, done, info = env.step(action)
50             obs = preprocess(obs, env=args.env).unsqueeze(0)
51             if args.env == "Pong-v0":
52                 obs_stack = torch.cat((obs_stack[:, 1:, ...], obs.
       unsqueeze(1)), dim=1).to(device)
53                 obs_stack = preprocess(obs_stack, env=args.env)
54
55             rewards_list.append(reward)
56
57         if discounted:
```

```
            episode_return = sum([env_config['gamma']**t * rewards_list[t] for t in range(len(rewards_list))])
        else:
            episode_return = sum(rewards_list)
        total_return += episode_return

        if verbose:
            print(f'Finished episode {i+1} with a total return of {episode_return}')


    return total_return / n_episodes

if __name__ == '__main__':
    args = parser.parse_args()

    # Initialize environment and config
    env = gym.make(args.env)
    env_config = ENV_CONFIGS[args.env]

    if args.save_video:
        env = gym.wrappers.Monitor(env, './video/', video_callable=lambda episode_id: True, force=True)

    # Load model from provided path.
    dqn = torch.load(args.path, map_location=torch.device('cpu'))
    dqn.eval()

    mean_return = evaluate_policy(dqn, env, env_config, args, dqn.eps_start, args.n_eval_episodes, render=args.render and not args.save_video, verbose=True)
    print(f'The policy got a mean return of {mean_return} over {args.n_eval_episodes} episodes.')

    env.close()
```

Listing 3: evaluate.py

```
import argparse

import gym
from gym.wrappers import AtariPreprocessing
import torch
import torch.nn as nn

import config
from utils import preprocess, init_weights, plot_result
from evaluate import evaluate_policy
from dqn import DQN, ReplayMemory, optimize

import matplotlib.pyplot as plt
import pickle

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

parser = argparse.ArgumentParser()
parser.add_argument('--env', default='CartPole-v0', choices=['CartPole-v0', 'Pong-v0'])
parser.add_argument('--evaluate_freq', type=int, default=25, help='How
```

```
    often to run evaluation.', nargs='?')
parser.add_argument('--evaluation_episodes', type=int, default=5, help='
    Number of evaluation episodes.', nargs='?')
parser.add_argument('--model_name', default='Pong-v0', help='Model to
    load if present.', nargs='?')

# Hyperparameter configurations for different environments. See config.
    py.
ENV_CONFIGS = {
    'CartPole-v0': config.CartPole,
    'Pong-v0': config.Pong
}

if __name__ == '__main__':
    args = parser.parse_args()

    # Initialize environment and config.
    env = gym.make(args.env)
    if args.env == "Pong-v0":
        env = AtariPreprocessing(env, screen_size=84, grayscale_obs=True
    , frame_skip=1, noop_max=30, scale_obs=True)
    env_config = ENV_CONFIGS[args.env]
    params = f"lr_{env_config['lr']}_gamma_{env_config['gamma']}_tg_{
    env_config['target_update_frequency']}_ann_{env_config['anneal_length
    ']}"

    # Load Model.
    try:
        model_name = f"{args.env}_best_{params}.pt"
        target_model_name = f"{args.env}_target_{params}.pt"
        dqn = torch.load(f'models/{model_name}')
        target_dqn = torch.load(f'models/{target_model_name}')
        print("Loading model...")
    except:
        print("Initialize model...")

        # Initialize deep Q-networks.
        dqn = DQN(env_name=args.env, env_config=env_config).to(device)
        dqn.apply(init_weights)

        # Create and initialize target Q-network.
        target_dqn = DQN(env_name=args.env, env_config=env_config).to(
    device)
        target_dqn.load_state_dict(dqn.state_dict())

    # Load results pickle file to add new results from the pretrained
    model.
    results_file = f"results/{args.env}/{args.env}_results_{params}.pkl"
    try:
        with open(results_file, 'rb') as file:
            results_dict = pickle.load(file)
        print("Load pickle file")
        return_list = results_dict['return']
        eval_return_list = results_dict['eval_return']
        loss_list = results_dict['loss']
        step_list = results_dict['step']
        epsilon_list = results_dict['epsilon']
        dqn.eps_start = results_dict['eps_start']
```

```
        best_mean_return = results_dict['best_mean_return']
        memory = results_dict['memory']
        print(f"best_mean_return = {best_mean_return}")
    except:
        # Initialize lists to keep track of episodes' loss, return, number of steps, and epsilon values throughout the training.
        loss_list, return_list, eval_return_list, step_list, epsilon_list = [], [], [], [], []

        # Keep track of best evaluation mean return achieved so far.
        best_mean_return = -float("Inf")

        # Create replay memory.
        memory = ReplayMemory(env_config['memory_size'])

    # Initialize optimizer used for training the DQN. We use Adam rather than RMSProp.
    optimizer = torch.optim.Adam(dqn.parameters(), lr=env_config['lr'])

    # Compute the value for epsilon linear annealing.
    eps = dqn.eps_start
    eps_end = dqn.eps_end
    anneal_length = dqn.anneal_length
    eps_step = (eps - eps_end) / anneal_length

    # Keep track of the total number of steps.
    total_steps = 0

    for episode in range(env_config['n_episodes']):
        done = False

        episode_loss = 0
        episode_rewards = []
        episode_steps = 0

        obs = preprocess(env.reset(), env=args.env).unsqueeze(0).to(device)
        if args.env == "Pong-v0":
            obs_stack = torch.cat(env_config["obs_stack_size"] * [obs]).unsqueeze(0).to(device)

        while not done:
            total_steps += 1
            episode_steps += 1

            # Get action from DQN.
            if args.env == "Pong-v0":
                action = dqn.act(obs_stack, eps).item()
            else:
                action = dqn.act(obs, eps).item()

            # Act in the true environment.
            next_obs, reward, done, info = env.step(action)
            episode_rewards.append(reward)

            # Preprocess incoming observation.
            if not done:
                # Preprocess the non-terminal state.
```

```
                next_obs = preprocess(next_obs, env=args.env).unsqueeze
    (0).to(device)
                if args.env == "Pong-v0":
                    next_obs_stack = torch.cat((obs_stack[:, 1:, ...],
    next_obs.unsqueeze(1)), dim=1).to(device)
                    next_obs_stack = preprocess(next_obs_stack, env=args
    .env).to(device)
            else:
                # Set to None the terminal state.
                next_obs = None
                if args.env == "Pong-v0":
                    next_obs_stack = None

            # Add the transition to the replay memory. Everything has
    been move to GPU if available.
            action = torch.as_tensor(action).to(device)
            reward = torch.as_tensor(reward).to(device)
            if args.env == "CartPole-v0":
                memory.push(obs, action, next_obs, reward)
            if args.env == "Pong-v0":
                memory.push(obs_stack, action, next_obs_stack, reward)

            # Run DQN.optimize() every env_config["train_frequency"]
    steps.
            if total_steps % env_config["train_frequency"] == 0:
                loss = optimize(dqn, target_dqn, memory, optimizer)
                episode_loss += loss

            # Update the target network every env_config["
    target_update_frequency"] steps.
            if total_steps % env_config["target_update_frequency"] == 0:
                target_dqn.load_state_dict(dqn.state_dict())

            # Update the current observation.
            obs = next_obs
            if args.env == "Pong-v0":
                obs_stack = next_obs_stack

            # Update epsilon after each step.
            epsilon_list.append(eps)
            if eps > eps_end:
                eps -= eps_step

        # Compute the episode return.
        episode_undiscounted_return = sum(episode_rewards)

        # Add results to lists.
        loss_list.append(episode_loss/episode_steps)
        return_list.append(episode_undiscounted_return)
        step_list.append(episode_steps)

        # Evaluate the current agent.
        if episode % args.evaluate_freq == 0:
            mean_return = evaluate_policy(dqn, env, env_config, args,
    eps, n_episodes=args.evaluation_episodes)
            eval_return_list.append(mean_return)

            print(f'Episode {episode}/{env_config["n_episodes"]}: {
```

```
    mean_return}')
174
175             # Save current agent if it has the best performance so far.
176             if mean_return >= best_mean_return:
177                 best_mean_return = mean_return
178
179                 print('Best performance so far! Saving model.')
180                 torch.save(dqn, f'models/{args.env}_best_{params}.pt')
181                 torch.save(target_dqn, f'models/{args.env}_target_{
    params}.pt')
182
183             # Save results.
184             results_dict = {
185                 'return': return_list,
186                 'eval_return': eval_return_list,
187                 'loss': loss_list,
188                 'step': step_list,
189                 'epsilon': epsilon_list,
190                 'eps_start': eps,
191                 'best_mean_return': best_mean_return,
192                 'memory': memory
193             }
194             print("Save pickle file")
195             with open(results_file, 'wb') as file:
196                 pickle.dump(results_dict, file)
197
198             # Plot
199             # x_axis = range(len(results_dict['return']))
200             # plot_result(x_axis, results_dict['return'], "Episodes", "
    Return", title="Return vs. Number of episodes", save_path=f"results/{
    args.env}", env_name=args.env, params=params)
201             # plot_result(range(len(results_dict['eval_return'])),
    results_dict['eval_return'], "Episodes", "Return", title="Evaluation
    Return every 25 episodes", save_path=f"results/{args.env}", env_name=
    args.env, params=params)
202             # plot_result(x_axis, results_dict['loss'], "Episodes", "
    Loss", title="Loss vs. Number of episodes", save_path=f"results/{args
    .env}", env_name=args.env, params=params)
203             # plot_result(x_axis, results_dict['step'], "Episodes", "
    Number of steps", title="Number of steps per episode", save_path=f"
    results/{args.env}", env_name=args.env, params=params)
204             # plot_result(range(len(results_dict['epsilon'])),
    results_dict['epsilon'], "Steps", "Epsilon", title="Epsilon values at
     each step of the training", save_path=f"results/{args.env}",
    env_name=args.env, params=params)
205
206     # Close environment after training is completed.
207     env.close()
208
209     # Save results.
210     results_dict = {
211         'return': return_list,
212         'eval_return': eval_return_list,
213         'loss': loss_list,
214         'step': step_list,
215         'epsilon': epsilon_list,
216         'eps_start': eps,
217         'best_mean_return': best_mean_return,
```

```
        'memory': memory
    }
    print("Save pickle file")
    with open(results_file, 'wb') as file:
        pickle.dump(results_dict, file)

    # Plot.
    x_axis = range(len(results_dict['return']))
    plot_result(x_axis, results_dict['return'], "Episodes", "Return",
    title="Return vs. Number of episodes", save_path=f"results/{args.env}
    ", env_name=args.env, params=params)
    plot_result(range(len(results_dict['eval_return'])), results_dict['
    eval_return'], "Episodes", "Return", title="Evaluation Return every
    25 episodes", save_path=f"results/{args.env}", env_name=args.env,
    params=params)
    plot_result(x_axis, results_dict['loss'], "Episodes", "Loss", title=
    "Loss vs. Number of episodes", save_path=f"results/{args.env}",
    env_name=args.env, params=params)
    plot_result(x_axis, results_dict['step'], "Episodes", "Number of
    steps", title="Number of steps per episode", save_path=f"results/{
    args.env}", env_name=args.env, params=params)
    plot_result(range(len(results_dict['epsilon'])), results_dict['
    epsilon'], "Steps", "Epsilon", title="Epsilon values at each step of
    the training", save_path=f"results/{args.env}", env_name=args.env,
    params=params)
```

Listing 4: train.py

```
import random

import gym
from gym.wrappers import AtariPreprocessing
import torch
import torch.nn as nn

import matplotlib.pyplot as plt

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

def preprocess(obs, env):
    """Performs necessary observation preprocessing."""
    if env in ['CartPole-v0', 'Pong-v0']:
        return torch.tensor(obs, device=device).float()
    else:
        raise ValueError('Please add necessary observation preprocessing
    instructions to preprocess() in utils.py.')

def init_weights(model):
    if isinstance(model, torch.nn.Linear):
        torch.nn.init.normal_(model.weight, mean=0, std=0.01)
        torch.nn.init.zeros_(model.bias)

def plot_result(x, y, xlabel, ylabel, title, save_path, env_name, params
    ):
    plt.plot(x, y)
    plt.title(title)
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.savefig(f"{save_path}/{env_name}_{title}_{params}.png")
```

```
30      plt.show()
```

Listing 5: utils.py

References

[1] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (Feb. 2015), pp. 529–533. ISSN: 00280836. URL: http://dx.doi.org/10.1038/nature14236.

[2] Volodymyr Mnih et al. "Playing Atari with Deep Reinforcement Learning". In: *CoRR* abs/1312.5602 (2013). arXiv: 1312.5602. URL: http://arxiv.org/abs/1312.5602.