

Lab6

June 22, 2023

```
[ ]: import gymnasium as gym
import math
import random
import matplotlib.pyplot as plt
from collections import namedtuple, deque
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
```

```
[ ]: #
CONST_ENV_NAME = 'CartPole-v1'

#      GPU
CONST_DEVICE = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

#      ReplayMemory
Transition = namedtuple('Transition', ('state', 'action', 'next_state', '
↳ 'reward'))
```

0.0.1 Relay Memory

```
[ ]: #      Replay Memory
class ReplayMemory(object):
    def __init__(self, capacity):
        self.memory = deque([], maxlen=capacity)

    def push(self, *args):
        """
        ReplayMemory
        """
        self.memory.append(Transition(*args))

    def sample(self, batch_size):
        """
        batch_size
        """
```

```

        return random.sample(self.memory, batch_size)

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

```

0.0.2 DQN Model

```

[ ]: class DQN_Model(nn.Module):
    def __init__(self, n_observations, n_actions):
        '''
        '''

        super(DQN_Model, self).__init__()
        self.layer1 = nn.Linear(n_observations, 128)
        self.layer2 = nn.Linear(128, 64)
        self.layer3 = nn.Linear(64, n_actions)

    def forward(self, x):
        '''
        '''

        # batch

        x = F.relu(self.layer1(x))
        x = F.relu(self.layer2(x))
        return self.layer3(x)

```

0.0.3 DQN Agent

```

[ ]: class DQN_Agent:
    def __init__(
        self,
        env,
        BATCH_SIZE = 128,
        GAMMA = 0.99,
        EPS_START = 0.1,
        EPS_END = 0.5,
        EPS_DECAY = 1000,
        TAU = 0.005,
        LR = 0.0001,
    ):
        #
        self.env = env
        # Q-

```

```

self.n_actions = env.action_space.n
state, _ = self.env.reset()
self.n_observations = len(state)
#
self.BATCH_SIZE = BATCH_SIZE
self.GAMMA = GAMMA
self.EPS_START = EPS_START
self.EPS_END = EPS_END
self.EPS_DECAY = EPS_DECAY
self.TAU = TAU
self.LR = LR

#
#
self.policy_net = DQN_Model(self.n_observations, self.n_actions).
↳to(CONST_DEVICE)

#
#
#
self.target_net = DQN_Model(self.n_observations, self.n_actions).
↳to(CONST_DEVICE)
self.target_net.load_state_dict(self.policy_net.state_dict())

#
self.optimizer = optim.AdamW(self.policy_net.parameters(), lr=self.LR,
↳amsgrad=True)

# Replay Memory
self.memory = ReplayMemory(10000)

#
self.steps_done = 0

#
self.episode_durations = []

def select_action(self, state):
    '''

    '''

    sample = random.random()
    eps = self.EPS_END + (self.EPS_START - self.EPS_END) * math.exp(-1. * self.
↳steps_done / self.EPS_DECAY)
    self.steps_done += 1
    if sample > eps:

```

```

        with torch.no_grad():
            # eps
            # Q-
            # t.max(1)
            # [1]
            return self.policy_net(state).max(1)[1].view(1, 1)
        else:
            # eps
            #
            return torch.tensor([[self.env.action_space.sample()]], device=CONST_DEVICE, dtype=torch.long)

def plot_durations(self, show_result=False):
    plt.figure(1)
    durations_t = torch.tensor(self.episode_durations, dtype=torch.float)
    if show_result:
        plt.title(' ')
    else:
        plt.clf()
        plt.title(' ')
        plt.xlabel(' ')
        plt.ylabel(' ')
        plt.plot(durations_t.numpy())
        plt.pause(0.001) #

def optimize_model(self):
    '''

    '''

    if len(self.memory) < self.BATCH_SIZE:
        return

    transitions = self.memory.sample(self.BATCH_SIZE)
    # batch'
    # batch- Transition
    # Transition batch-
    batch = Transition(*zip(*transitions))

    # batch'
    non_final_mask = torch.tensor(tuple(map(lambda s: s is not None, batch.  

    next_state))), device=CONST_DEVICE, dtype=torch.bool)
    non_final_next_states = torch.cat([s for s in batch.next_state if s is not   

    None])
    state_batch = torch.cat(batch.state)
    action_batch = torch.cat(batch.action)
    reward_batch = torch.cat(batch.reward)

```

```

#            $Q(s_t, a)$ 
state_action_values = self.policy_net(state_batch).gather(1, action_batch)

#            $V(s_{t+1})$ 
next_state_values = torch.zeros(self.BATCH_SIZE, device=CONST_DEVICE)

with torch.no_grad():
    next_state_values[non_final_mask] = self.
    ↪target_net(non_final_next_states).max(1)[0]

#            $Q$ 
    expected_state_action_values = (next_state_values * self.GAMMA) +
    ↪reward_batch

#           Huber loss
criterion = nn.SmoothL1Loss()
loss = criterion(state_action_values, expected_state_action_values.
    ↪unsqueeze(1))

#
self.optimizer.zero_grad()
loss.backward()

# gradient clipping
torch.nn.utils.clip_grad_value_(self.policy_net.parameters(), 100)
self.optimizer.step()

def play_agent(self):
    '''
    '''

    env2 = gym.make(CONST_ENV_NAME, render_mode='human')
    state = env2.reset()[0]
    state = torch.tensor(state, dtype=torch.float32, device=CONST_DEVICE).
    ↪unsqueeze(0)
    res = []

    terminated = False
    truncated = False

    while not terminated and not truncated:
        action = self.select_action(state)
        action = action.item()
        observation, reward, terminated, truncated, _ = env2.step(action)
        env2.render()

```

```

        res.append((action, reward))

        state = torch.tensor(observation, dtype=torch.float32,
↪device=CONST_DEVICE).unsqueeze(0)

        print('done!')
        print('          : ', res)

    def train(self):
        """

        """

        if torch.cuda.is_available():
            num_episodes = 600
        else:
            num_episodes = 50

        for i_episode in range(num_episodes):
            #
            state, info = self.env.reset()
            state = torch.tensor(state, dtype=torch.float32, device=CONST_DEVICE).
↪unsqueeze(0)

            terminated = False
            truncated = False

            iters = 0
            while not terminated and not truncated:
                action = self.select_action(state)
                observation, reward, terminated, truncated, _ = self.env.step(action.
↪item())
                reward = torch.tensor([reward], device=CONST_DEVICE)

                if terminated:
                    next_state = None
                else:
                    next_state = torch.tensor(observation, dtype=torch.float32,
↪device=CONST_DEVICE).unsqueeze(0)

                #          Replay Memory
                self.memory.push(state, action, next_state, reward)

                #
                state = next_state

            #

```

```

self.optimize_model()

#           target-
#   ←      + (1 - )
target_net_state_dict = self.target_net.state_dict()
policy_net_state_dict = self.policy_net.state_dict()

for key in policy_net_state_dict:
    target_net_state_dict[key] = policy_net_state_dict[key] * self.TAU +
    ↪target_net_state_dict[key] * (1 - self.TAU)

self.target_net.load_state_dict(target_net_state_dict)
iters += 1

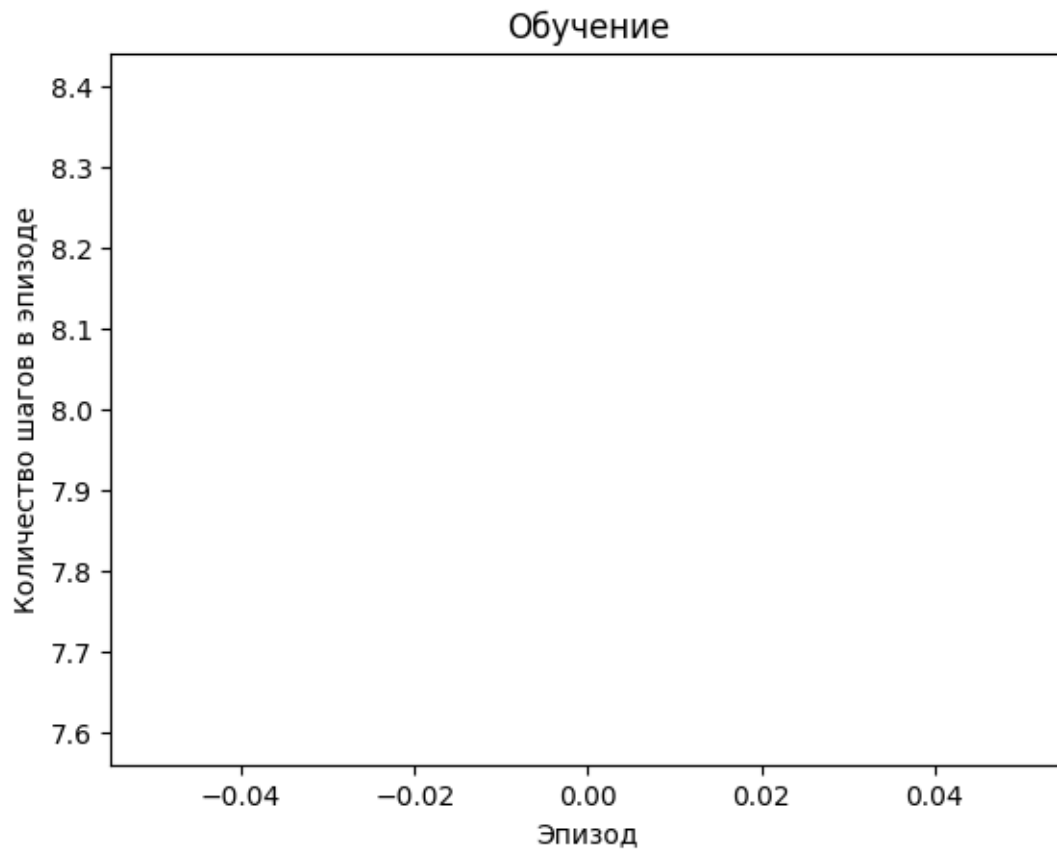
self.episode_durations.append(iters)
self.plot_durations()

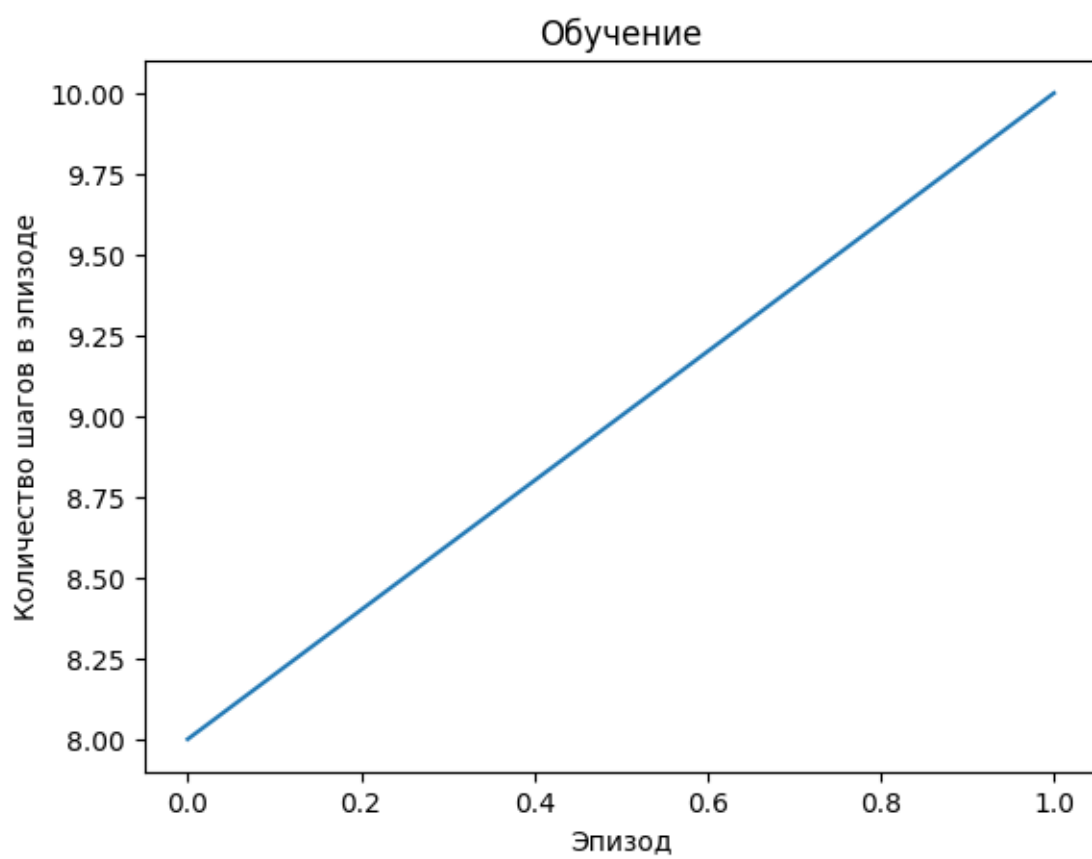
```

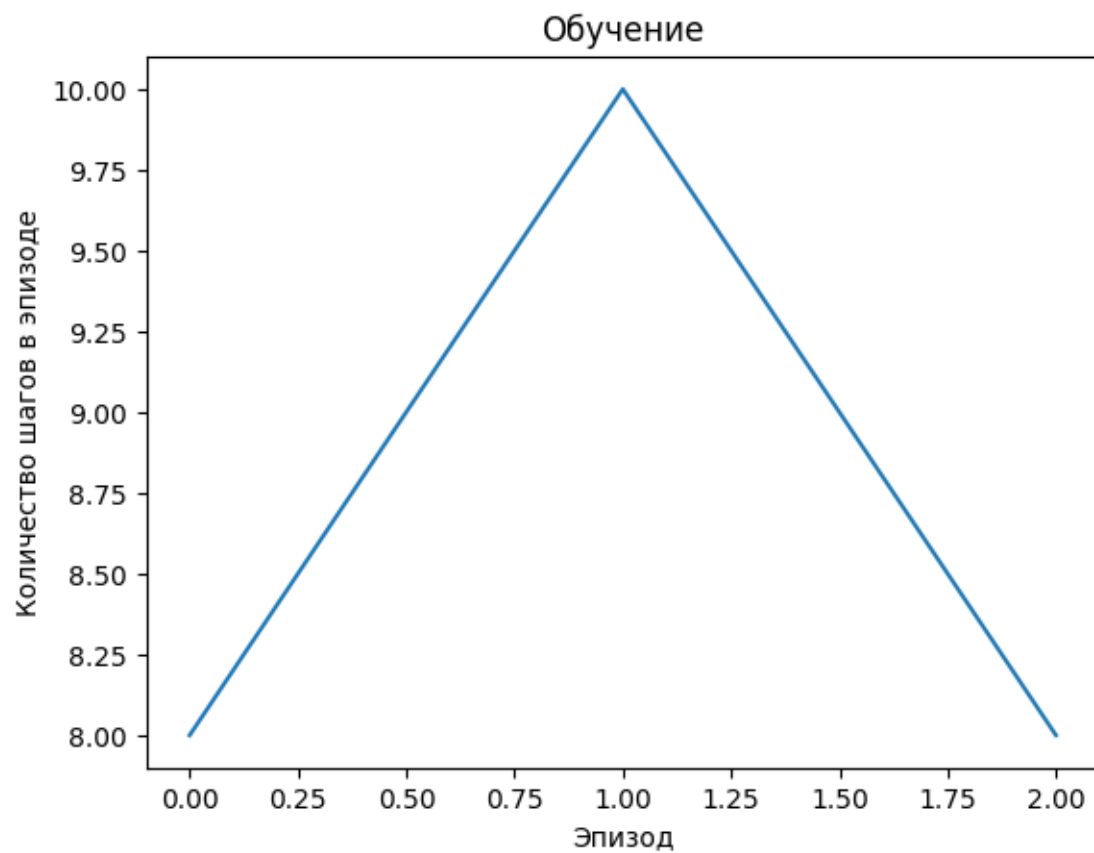
```

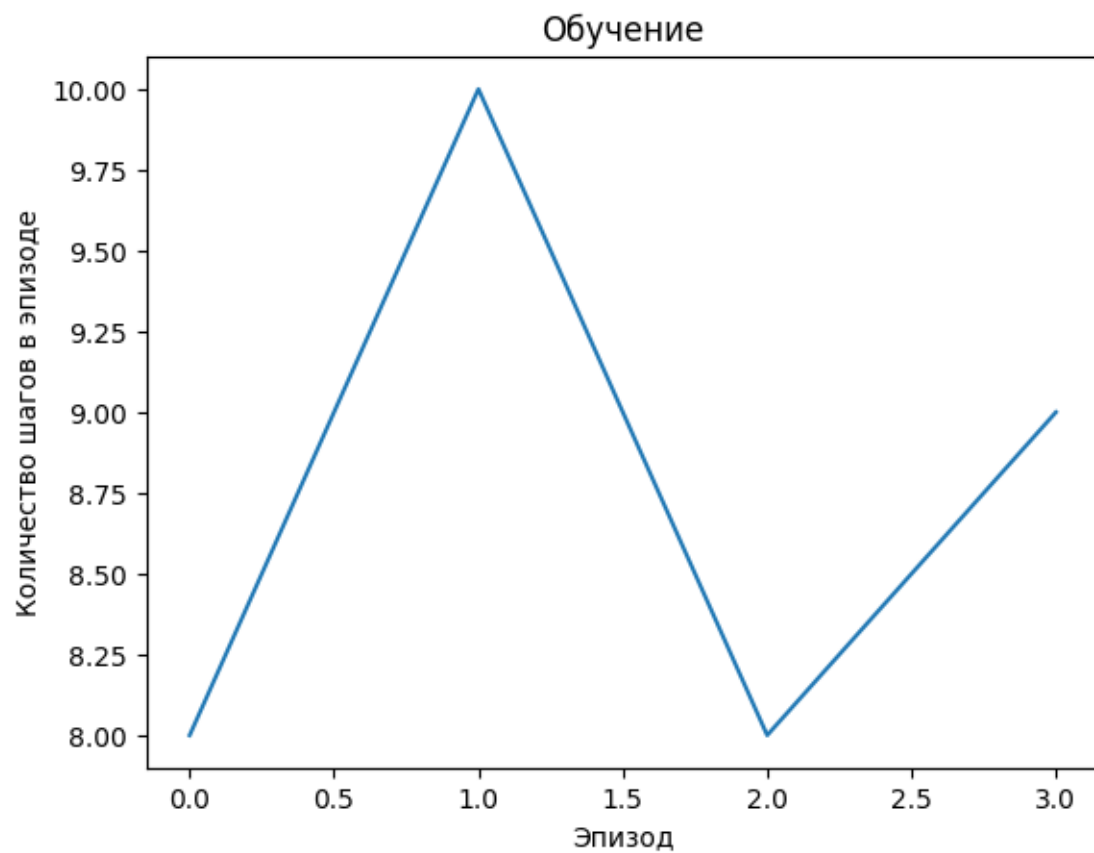
[ ]: env = gym.make(CONST_ENV_NAME)
agent = DQN_Agent(env)
agent.train()
agent.play_agent()

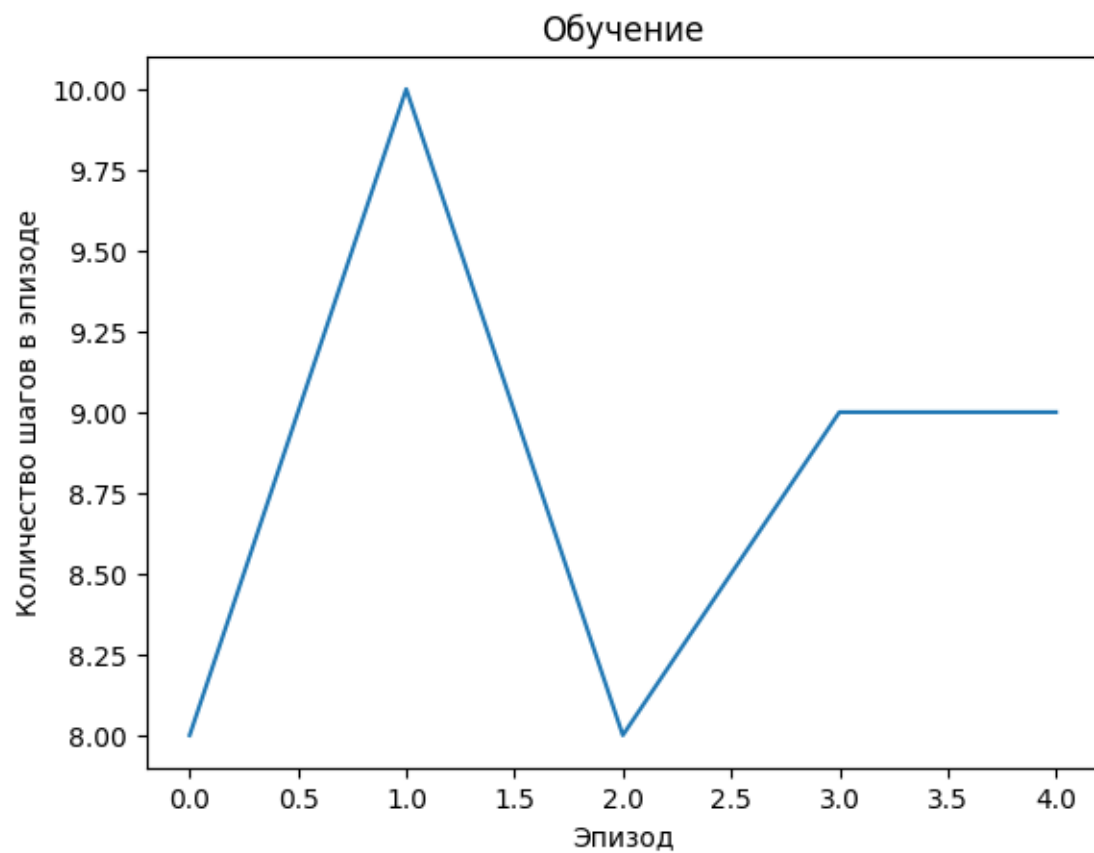
```

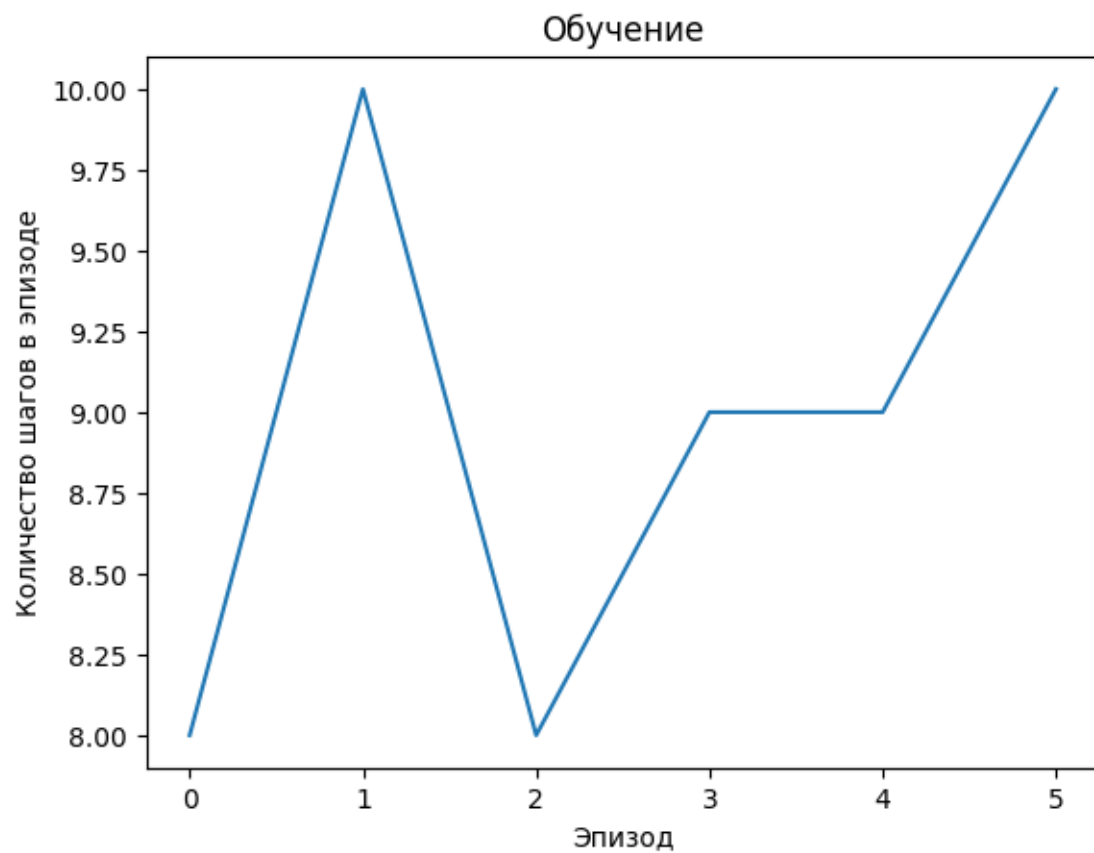


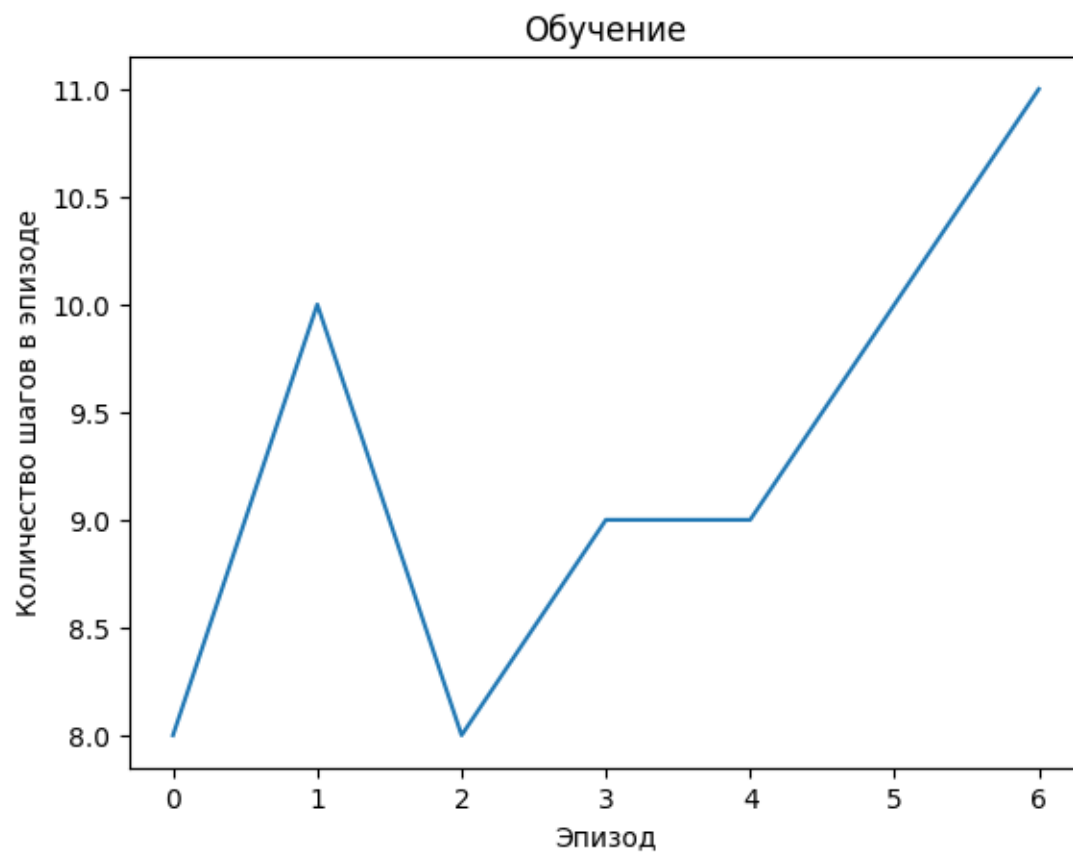


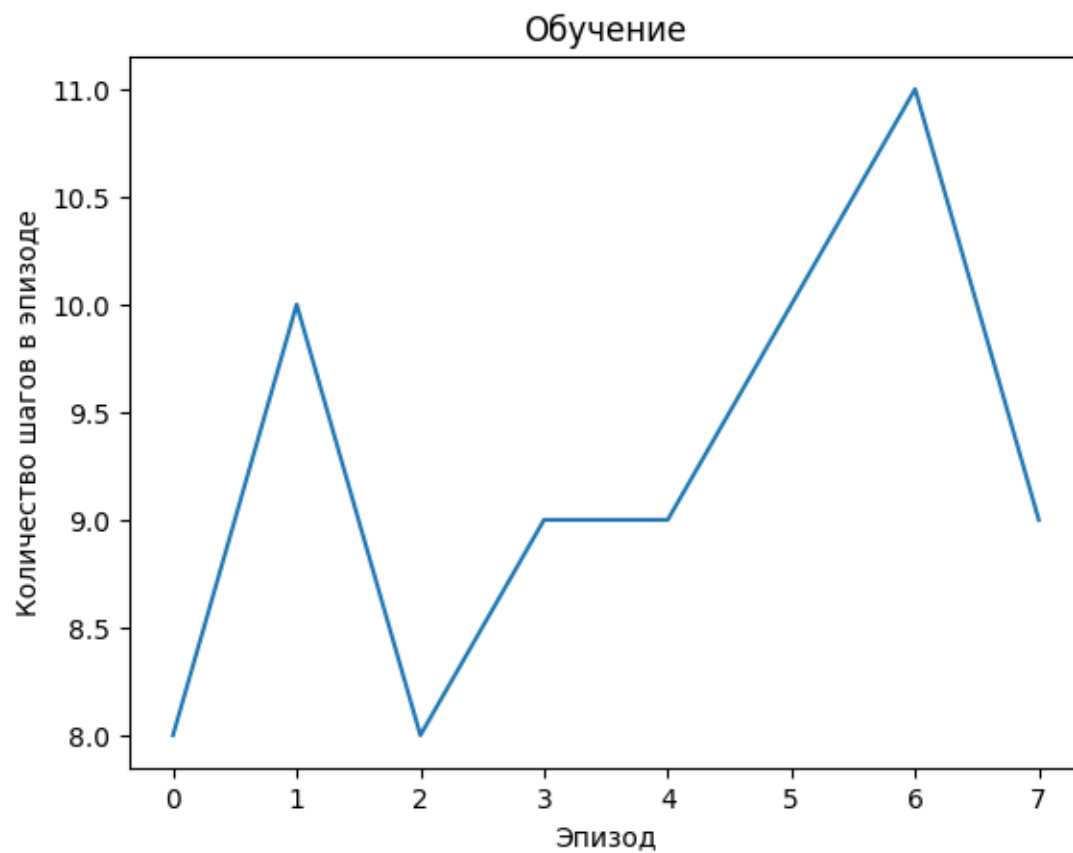


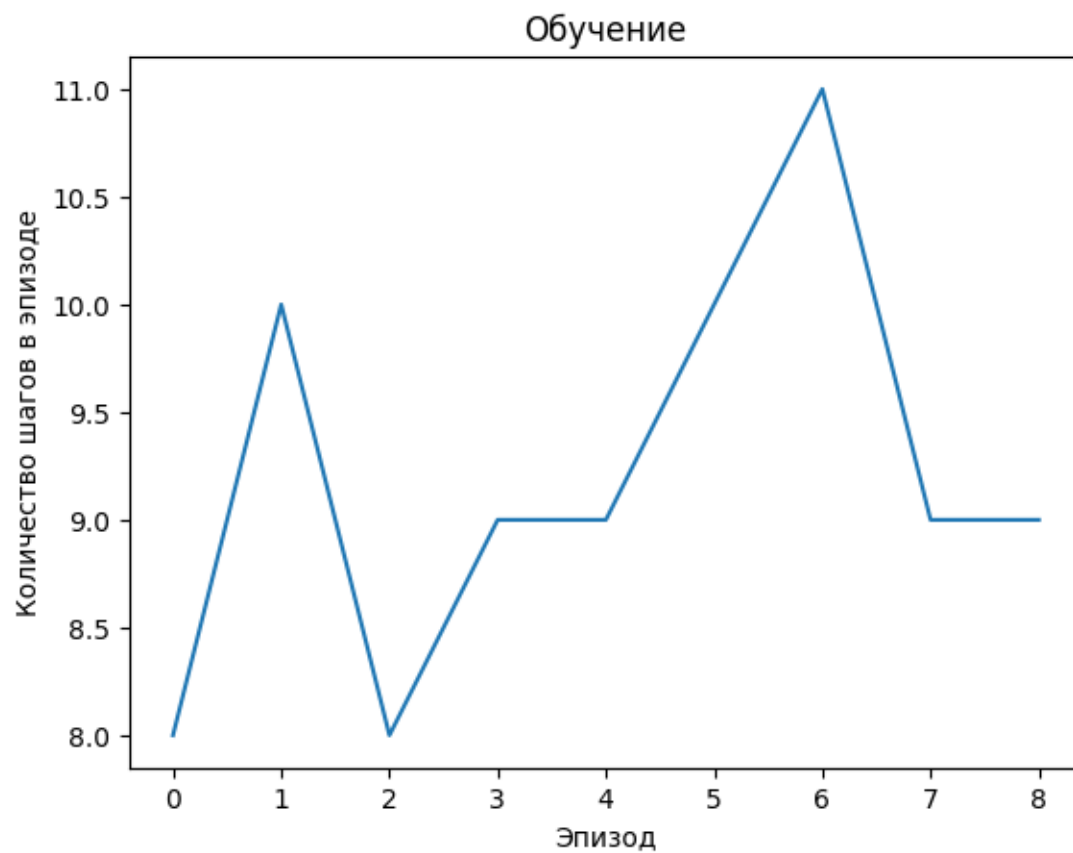


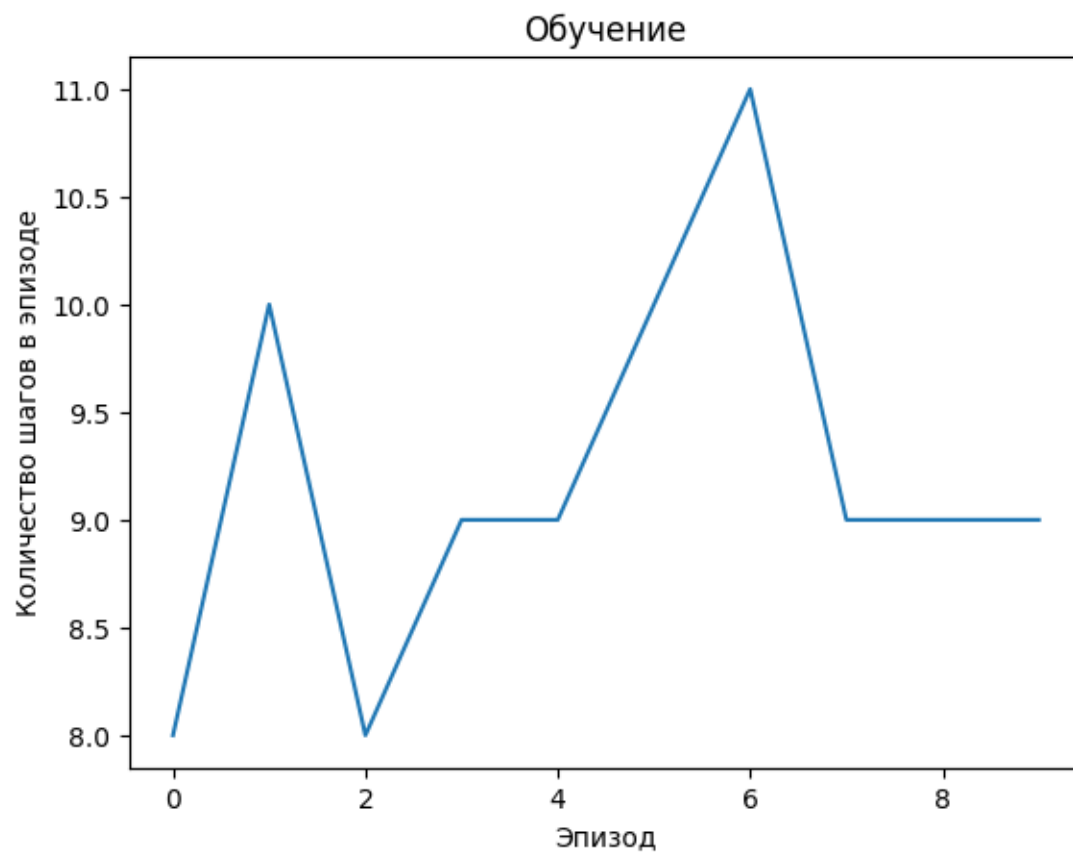


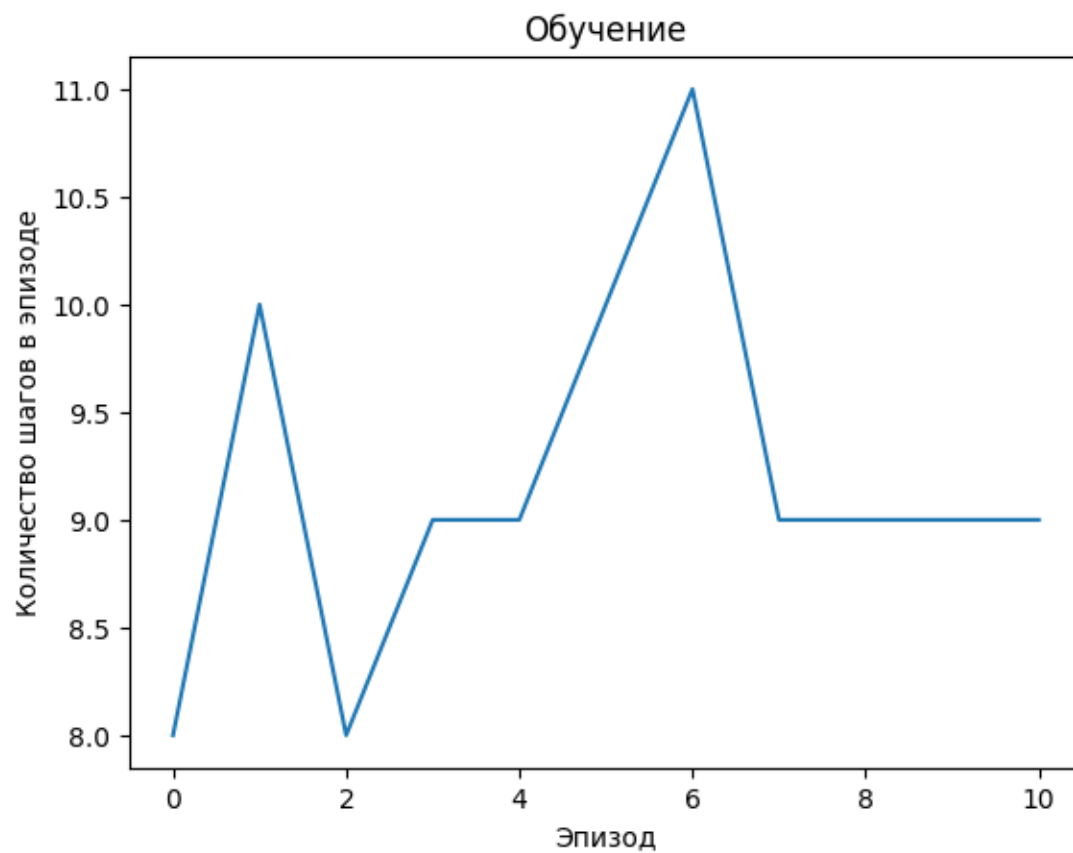


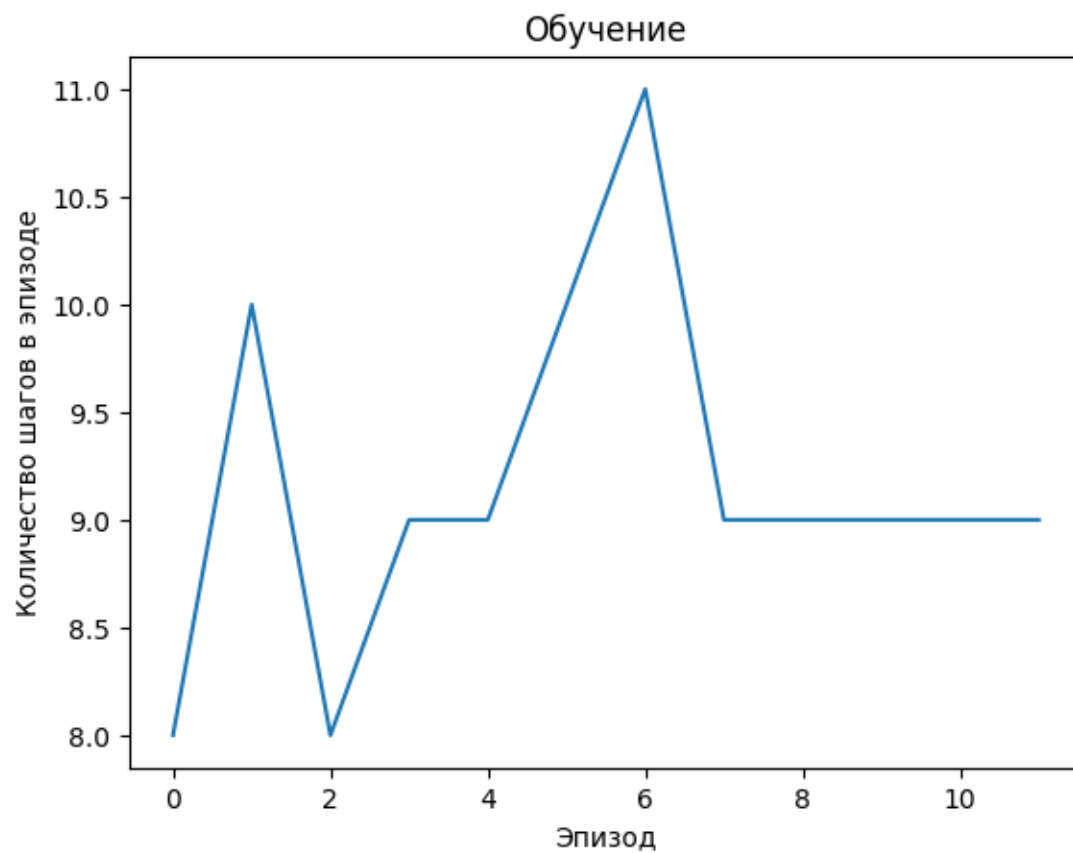


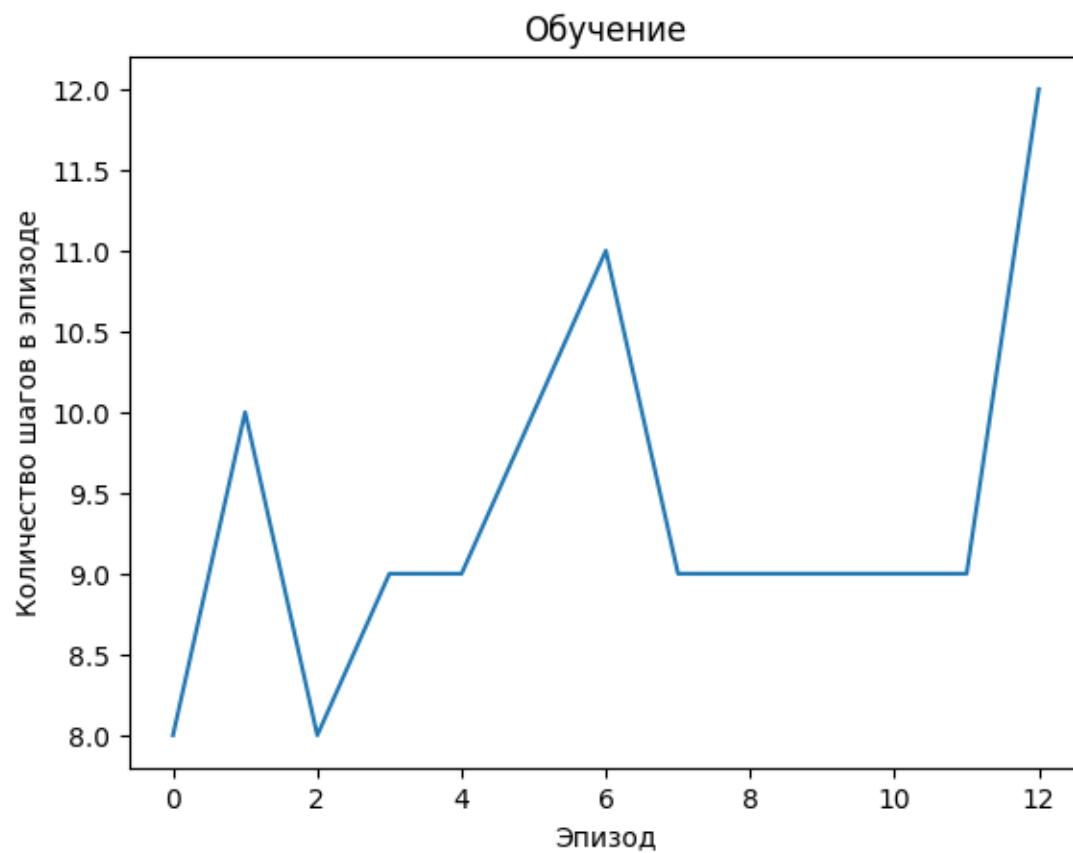


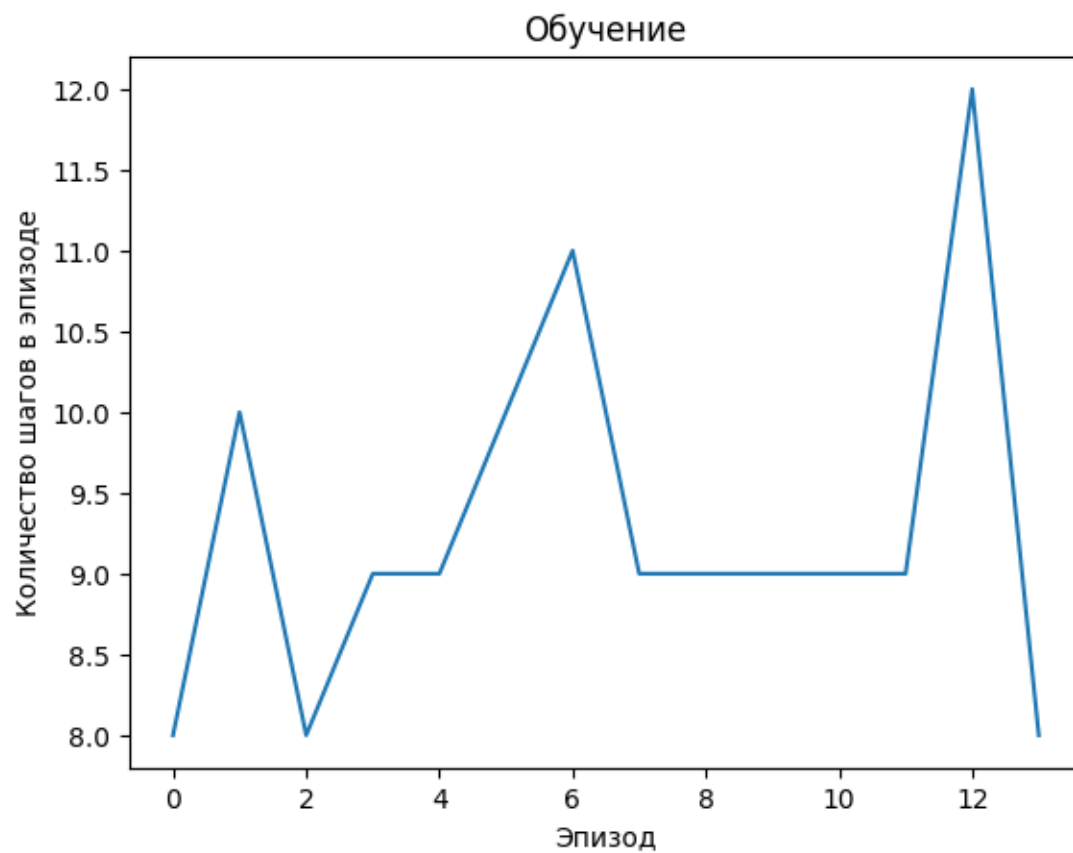


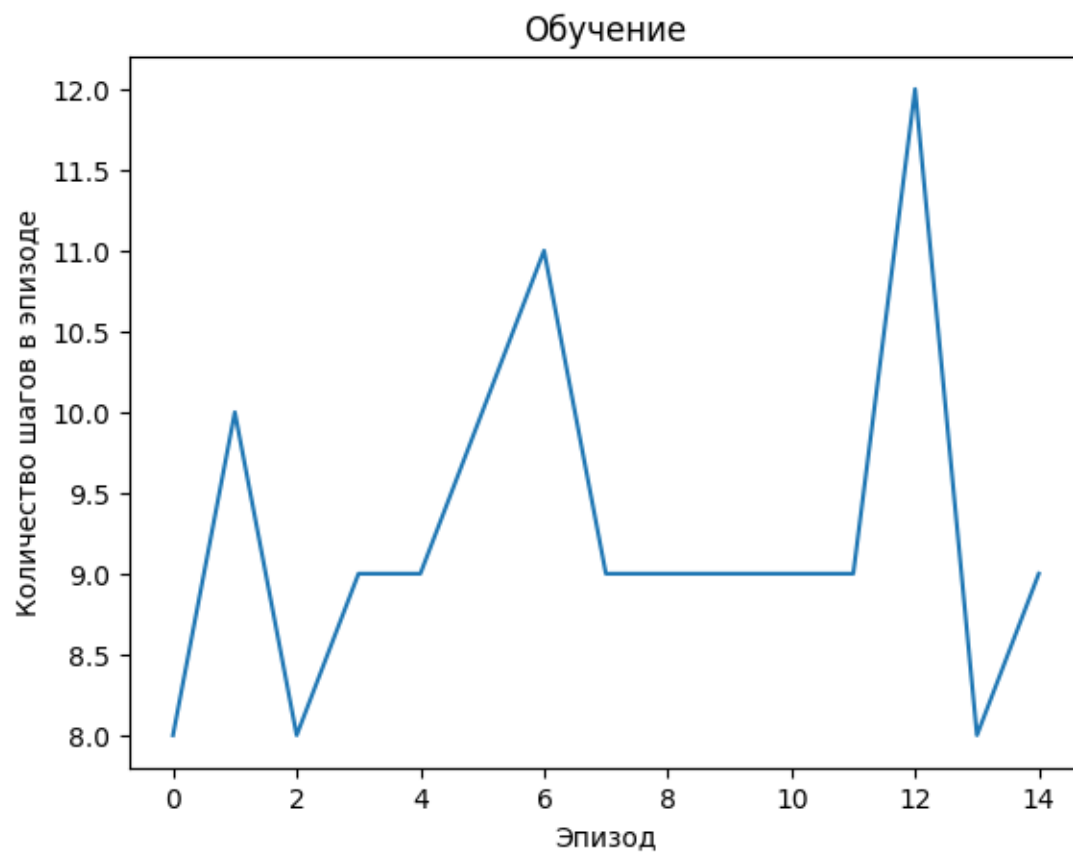


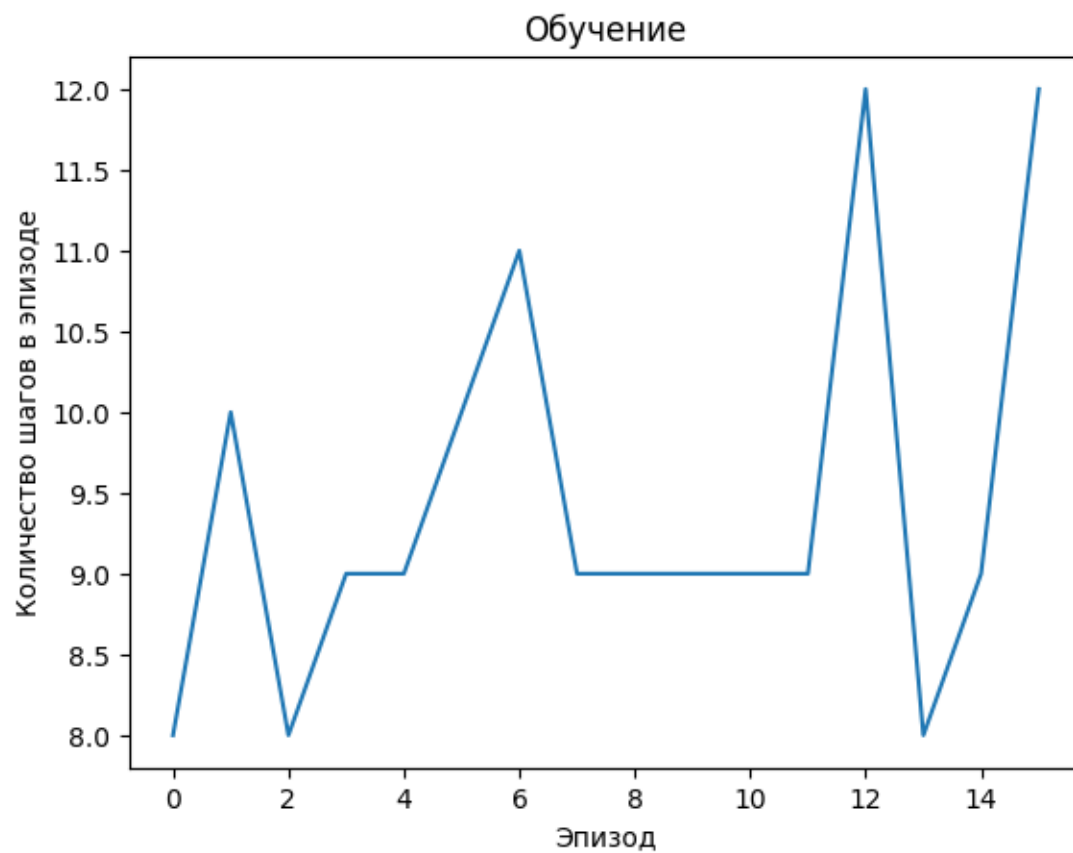


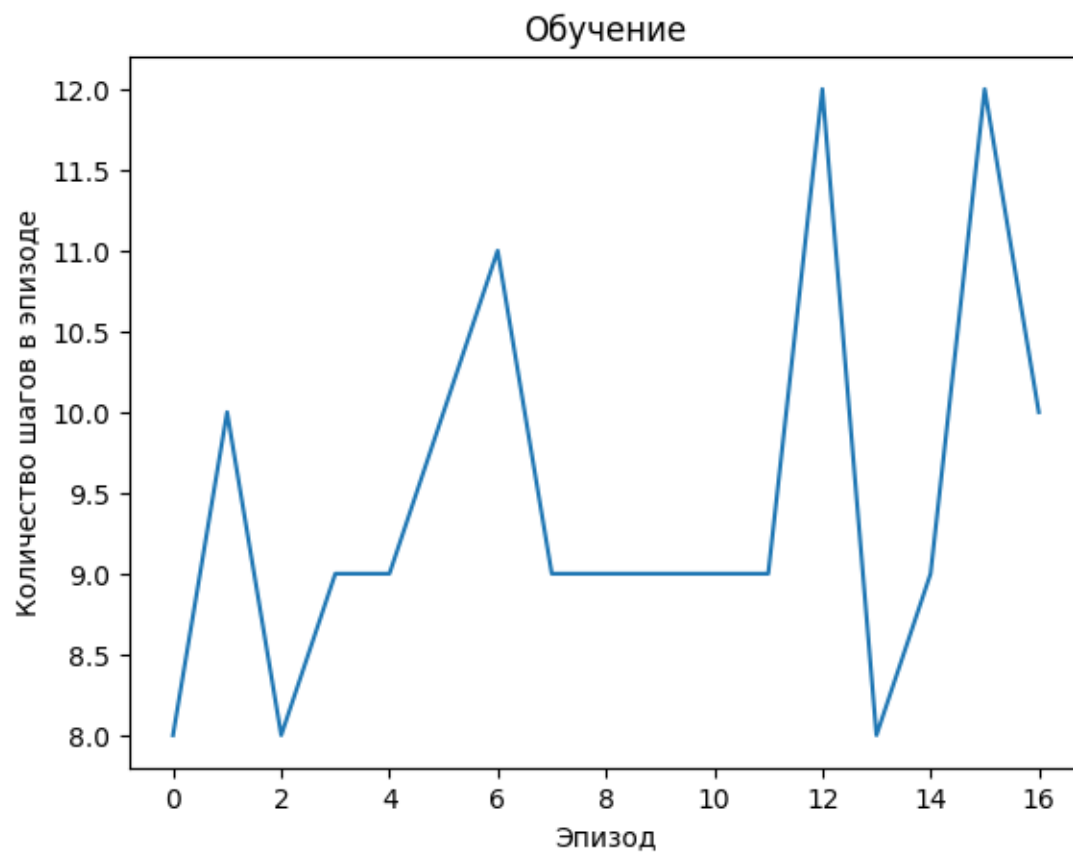


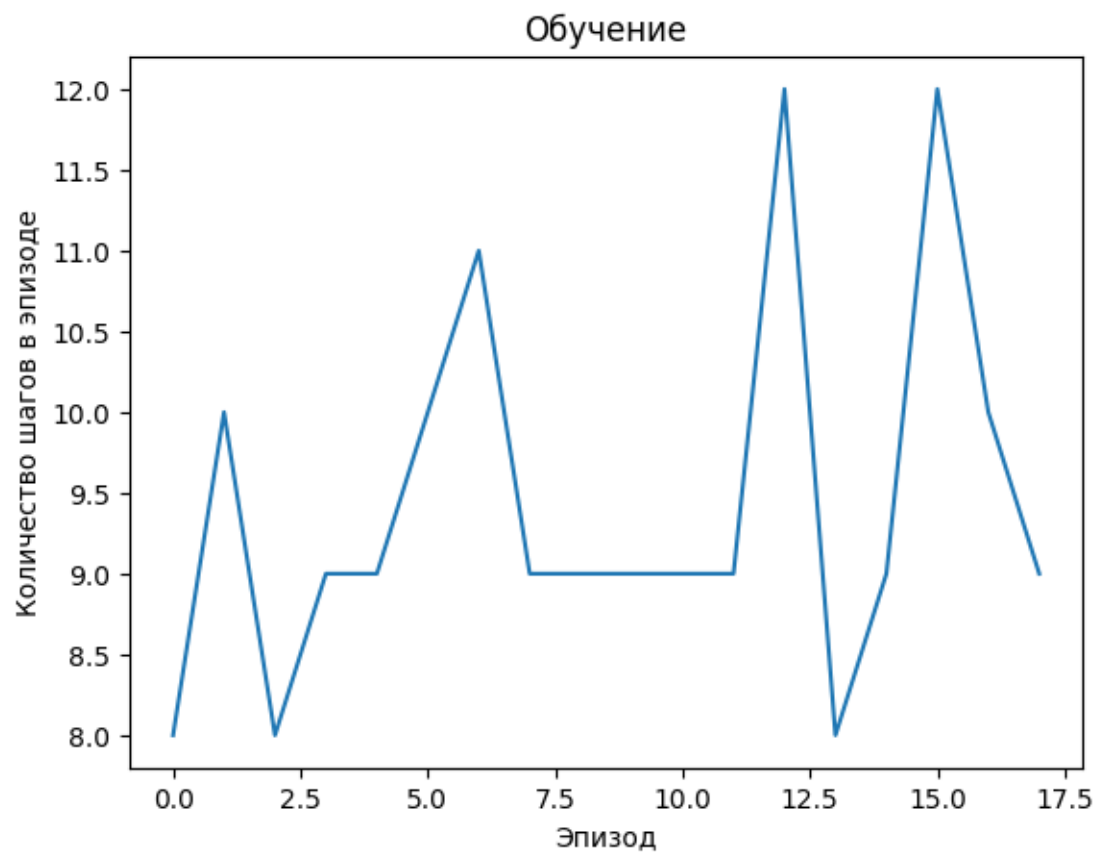


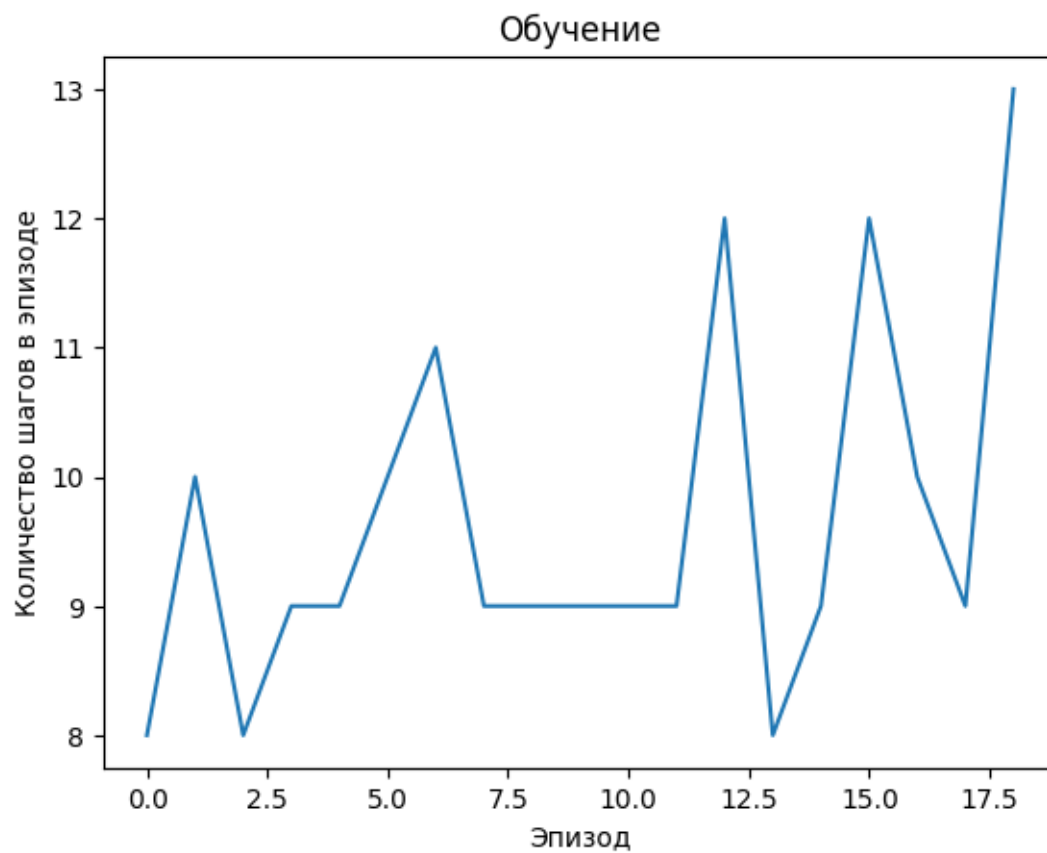


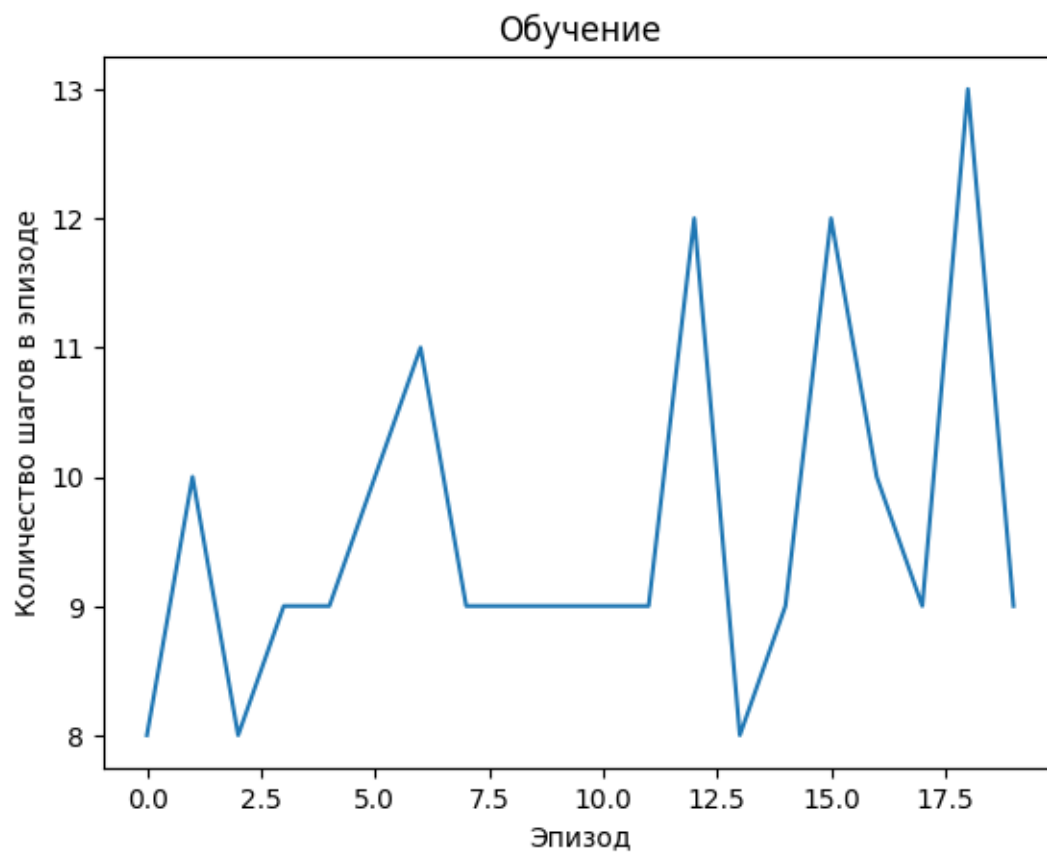


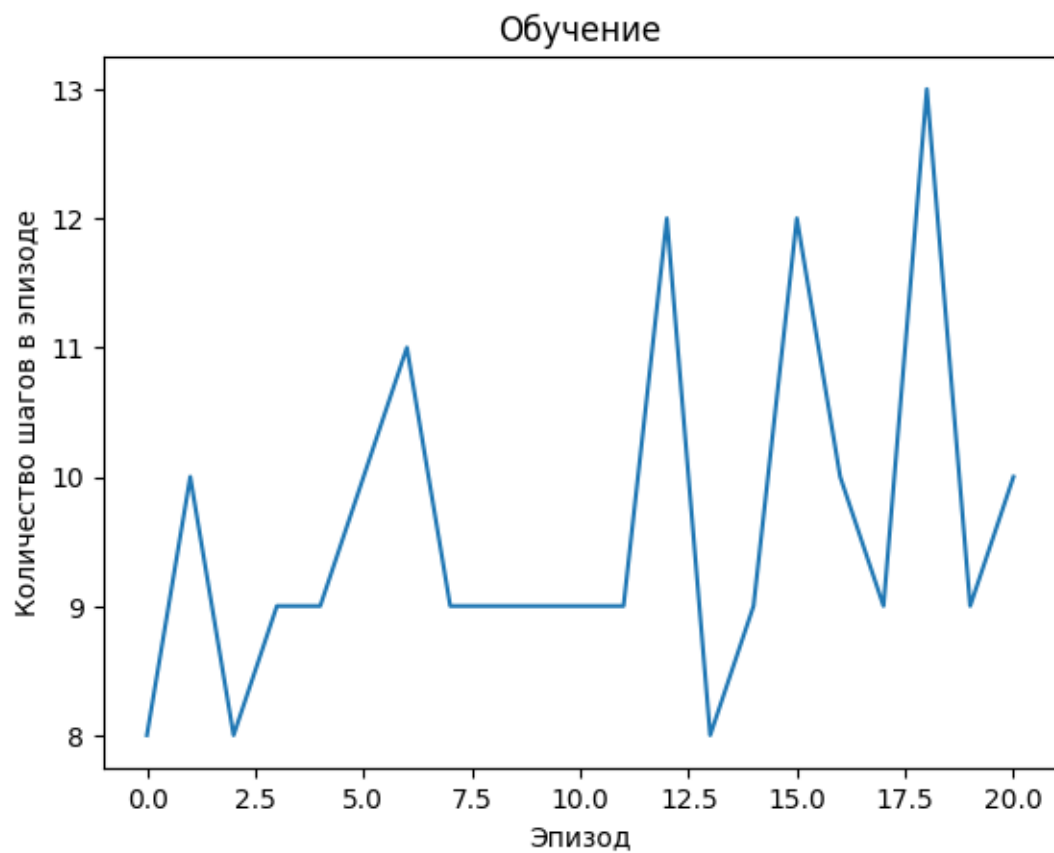


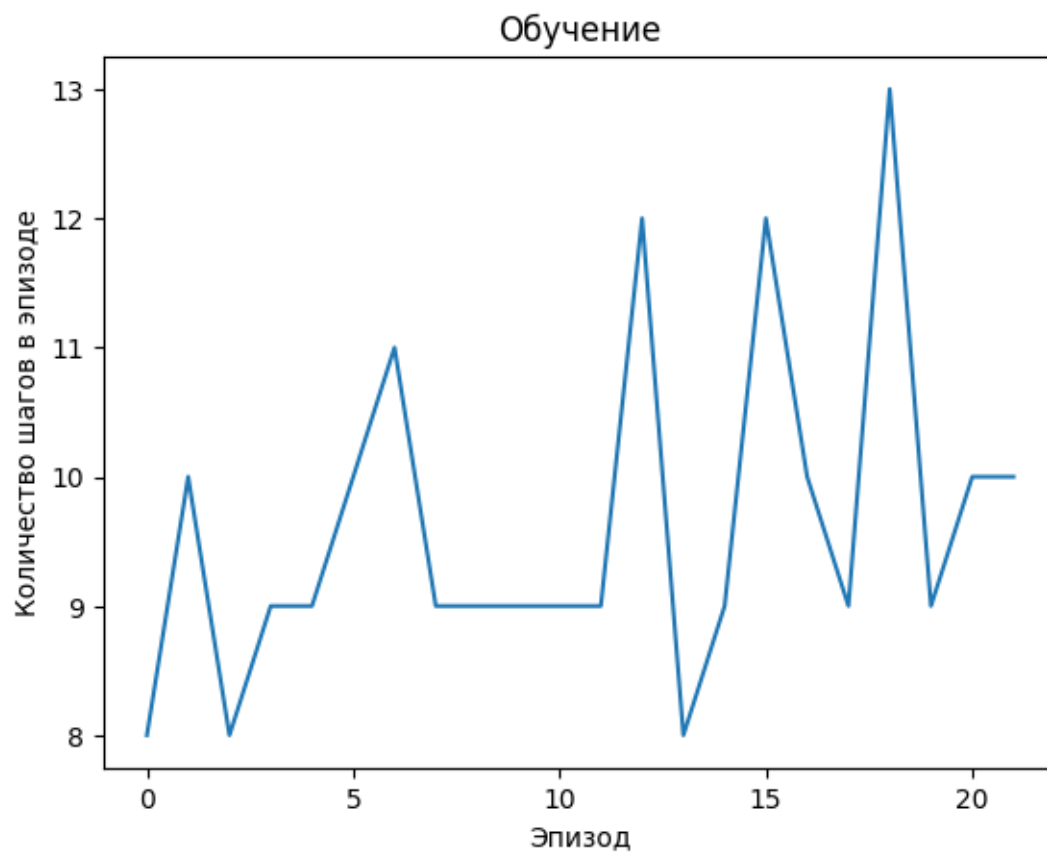


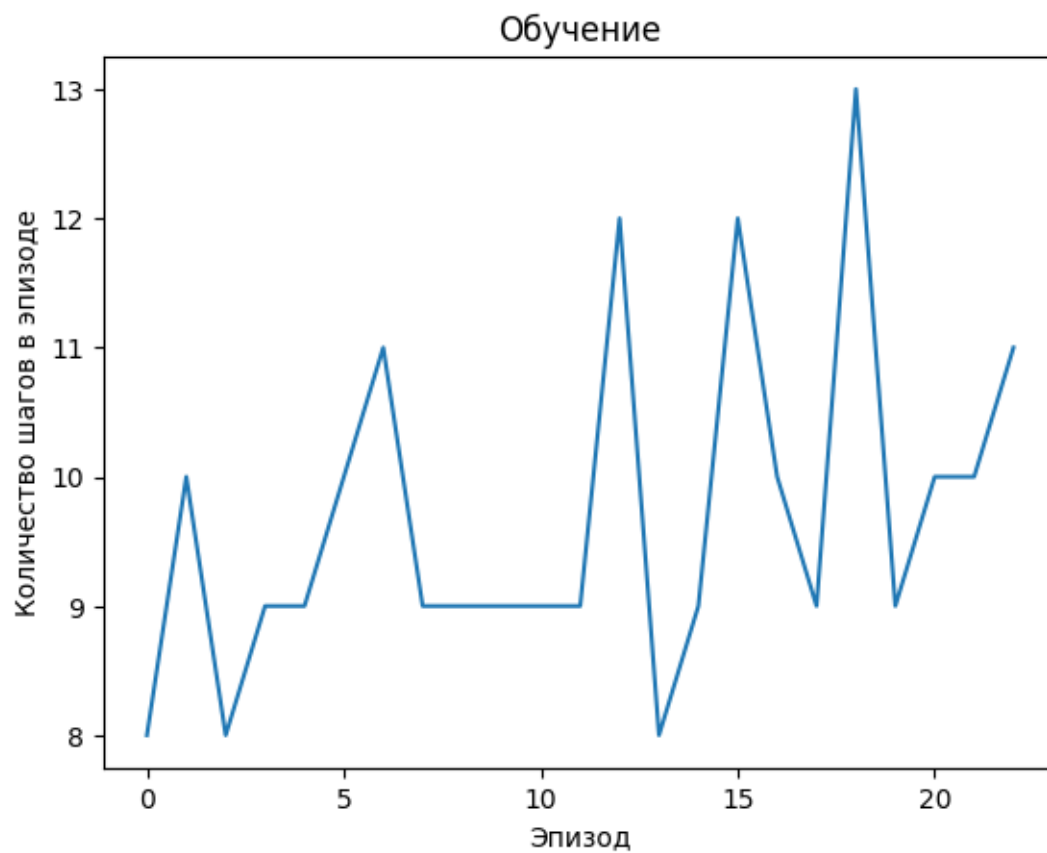


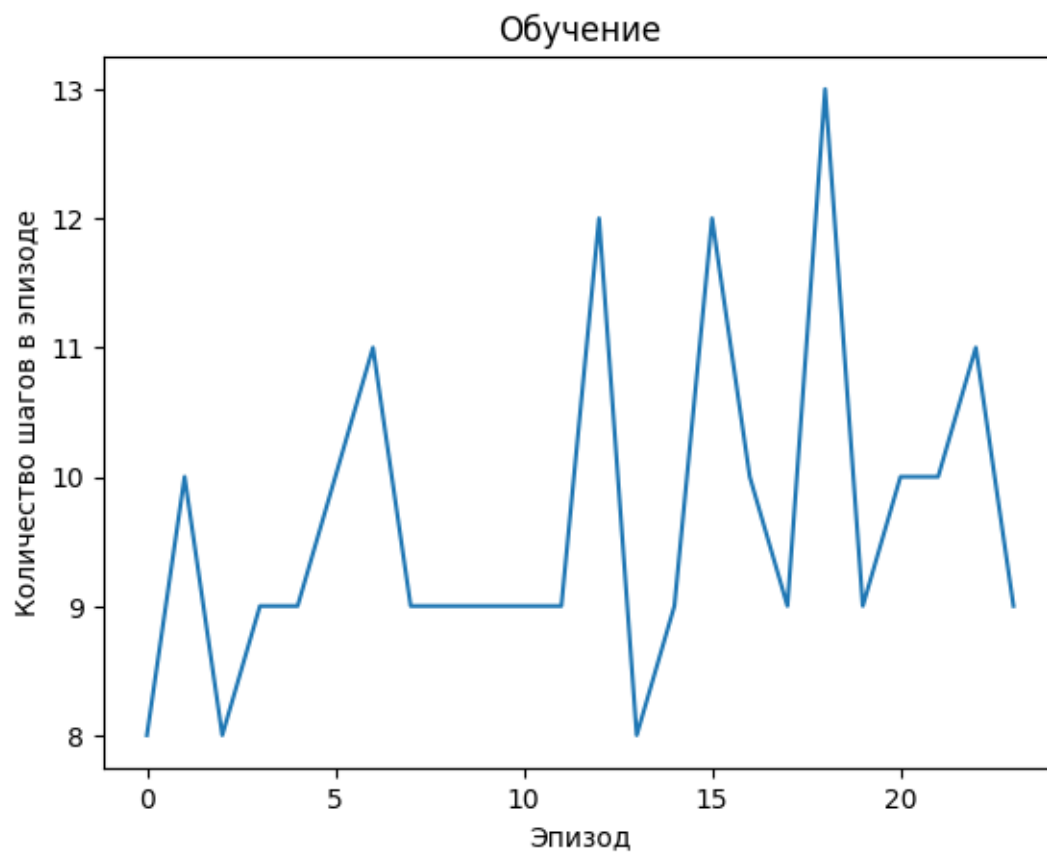


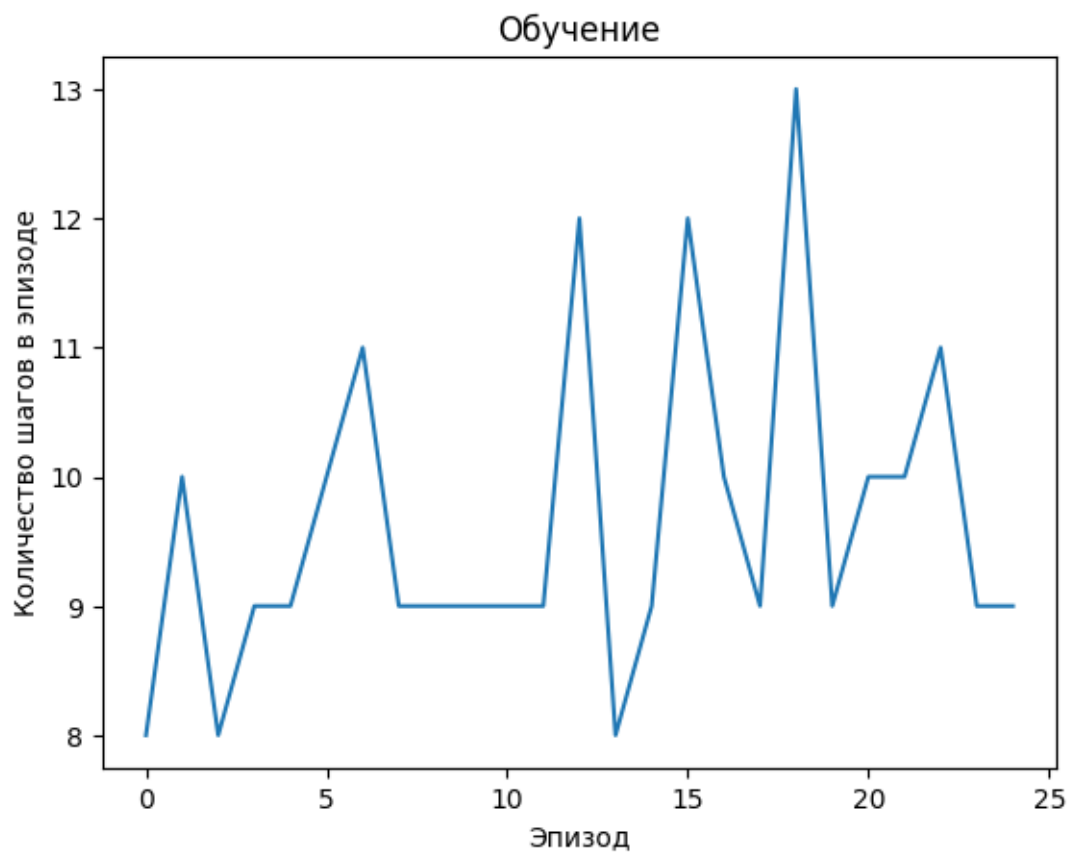


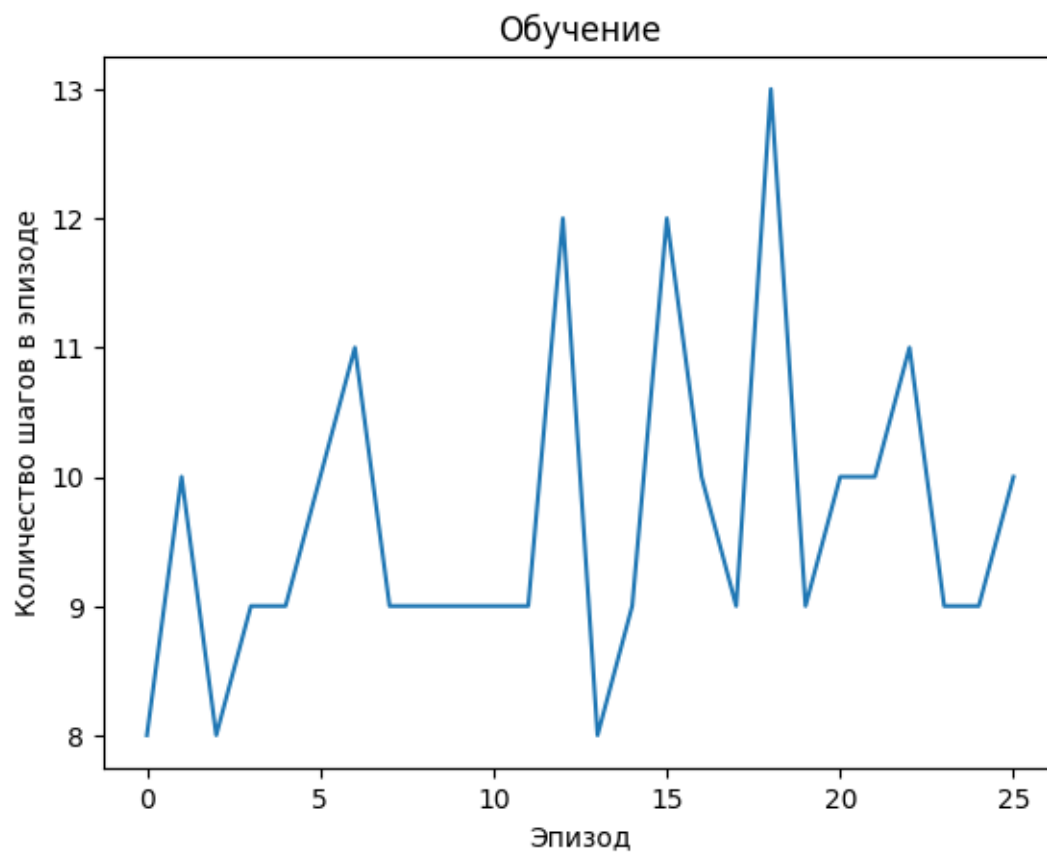


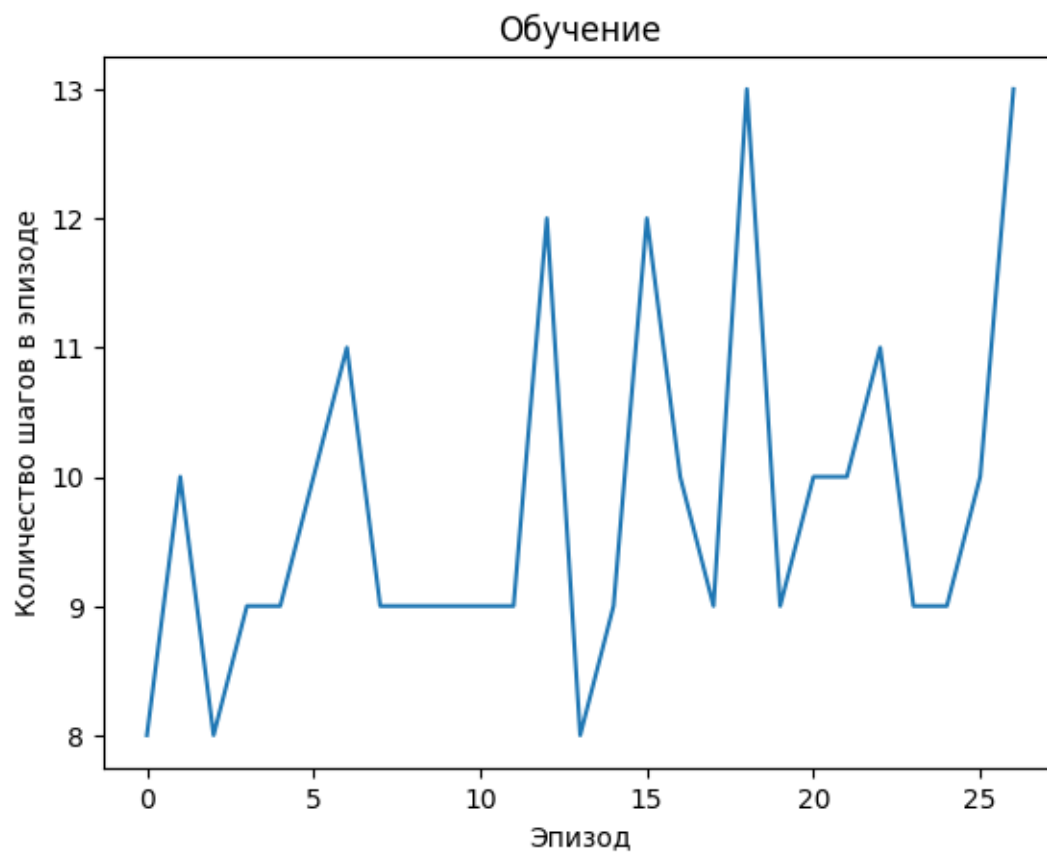


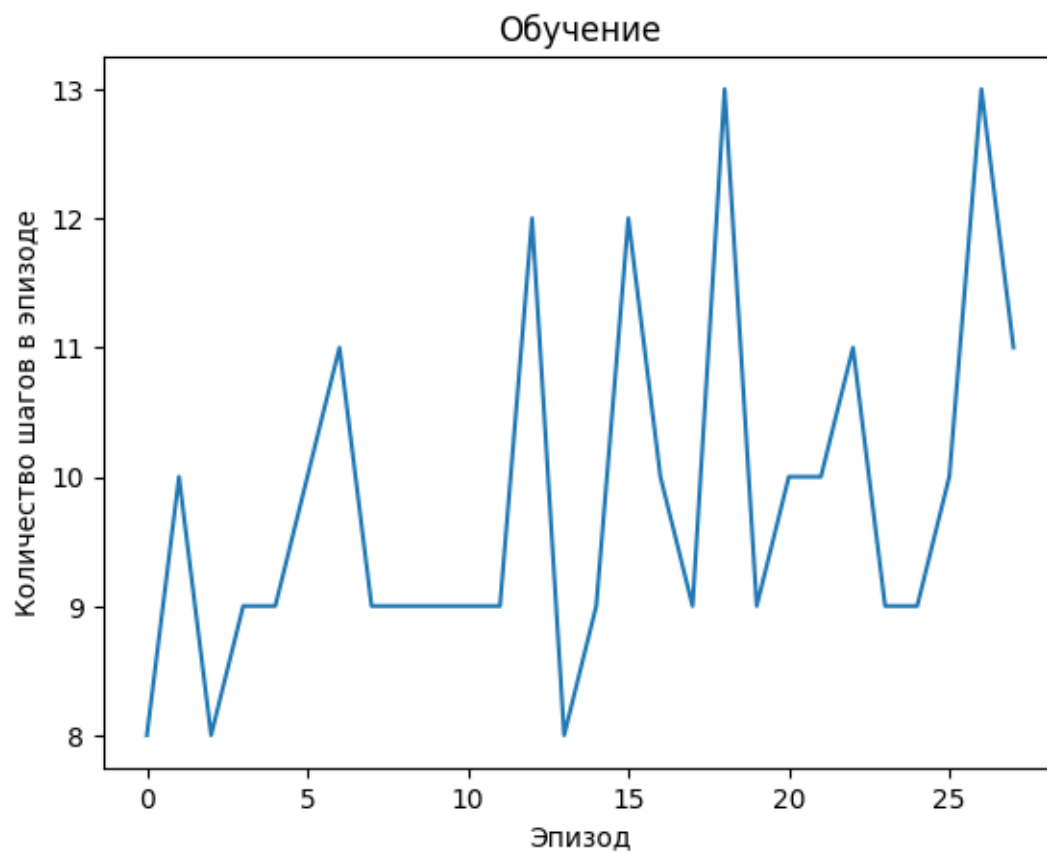


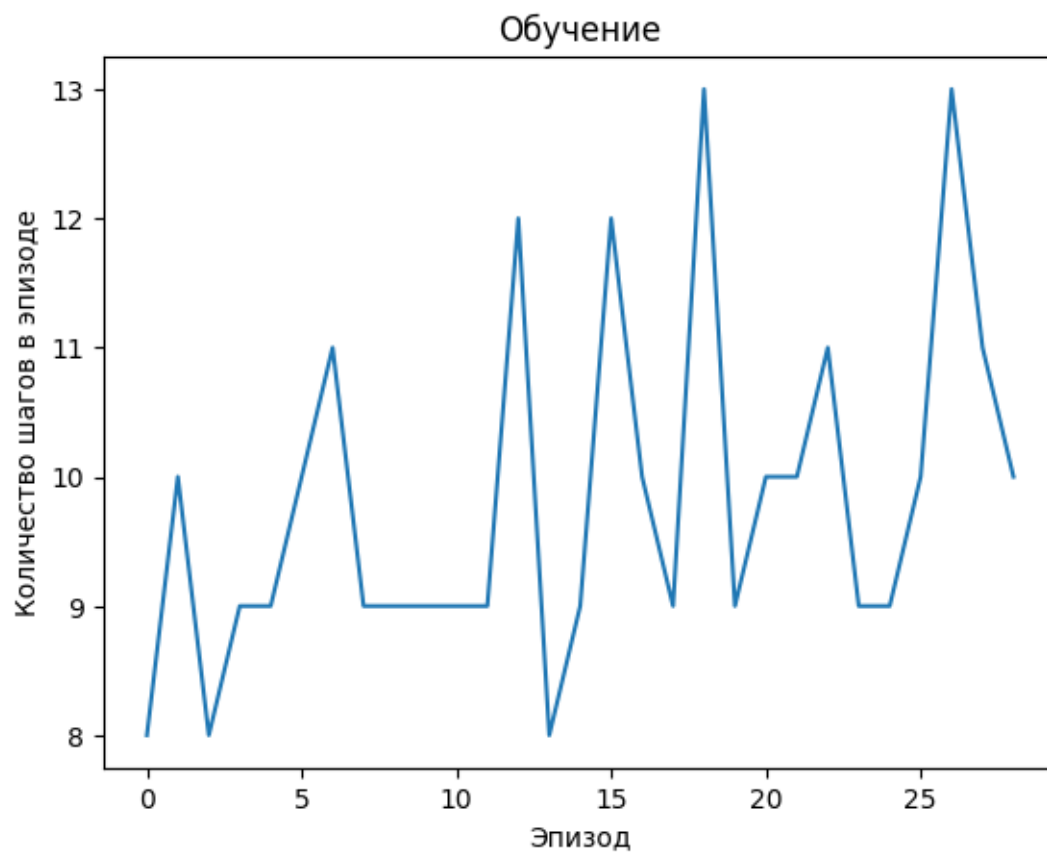


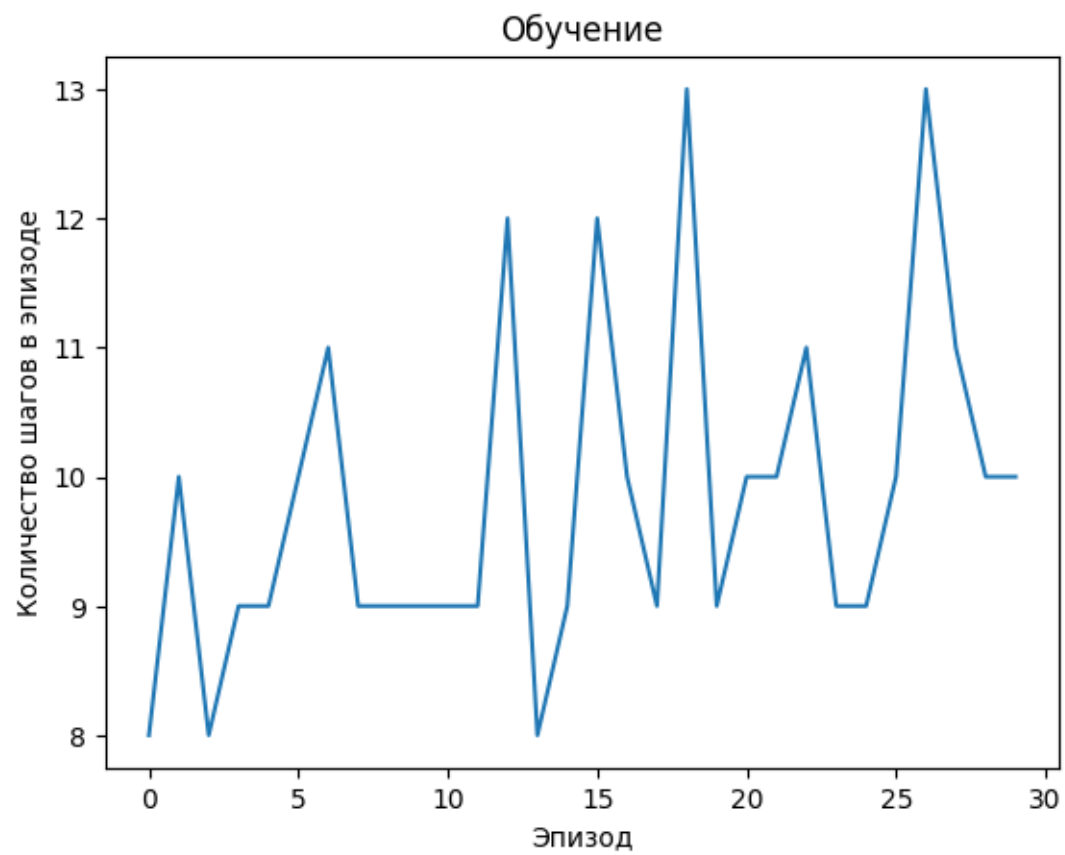


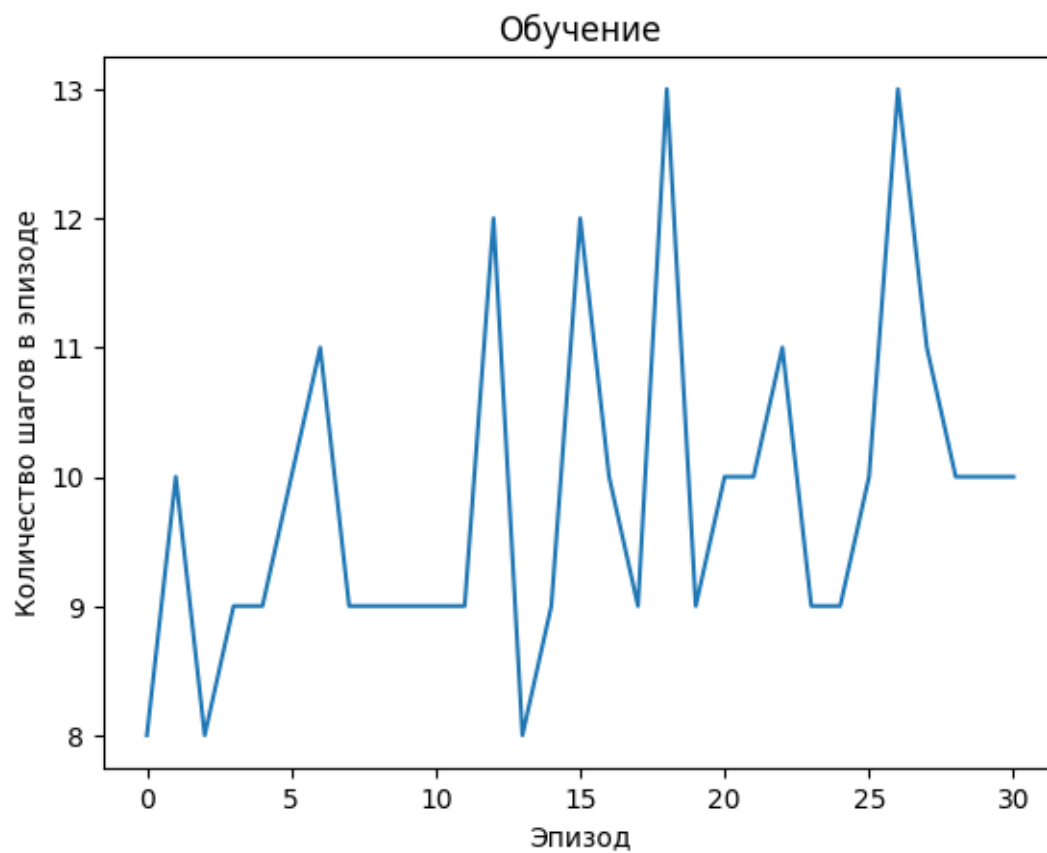


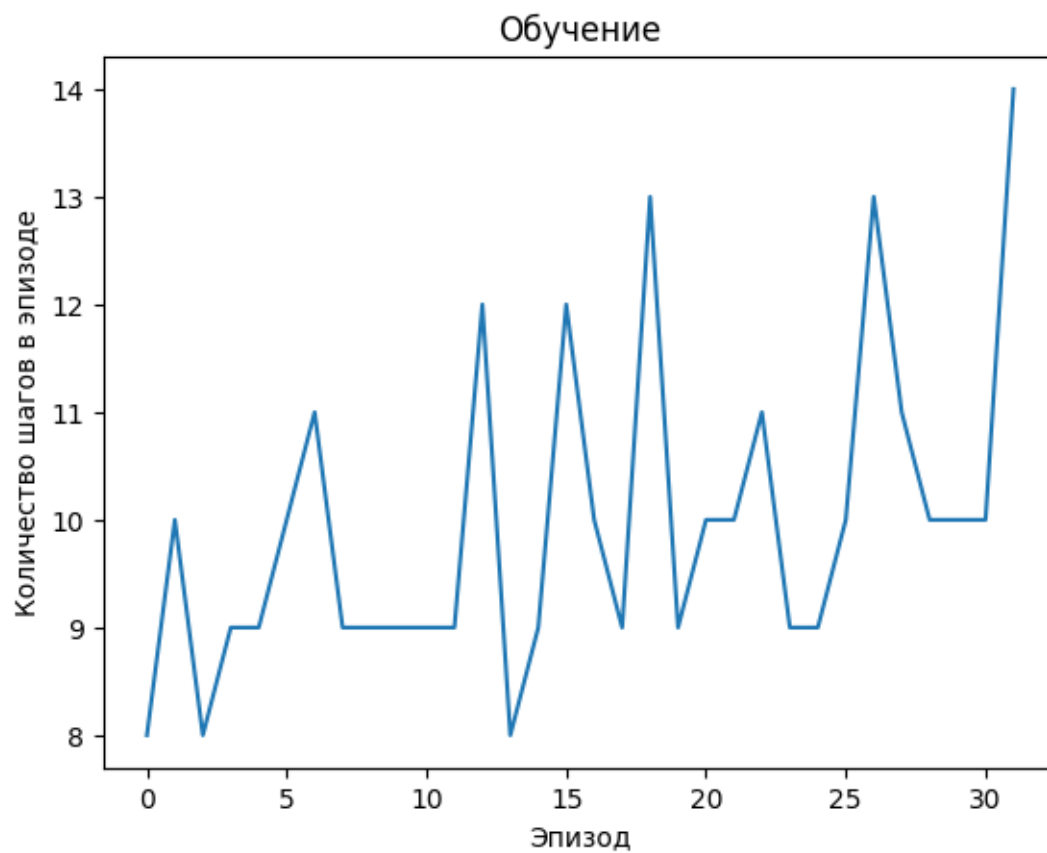


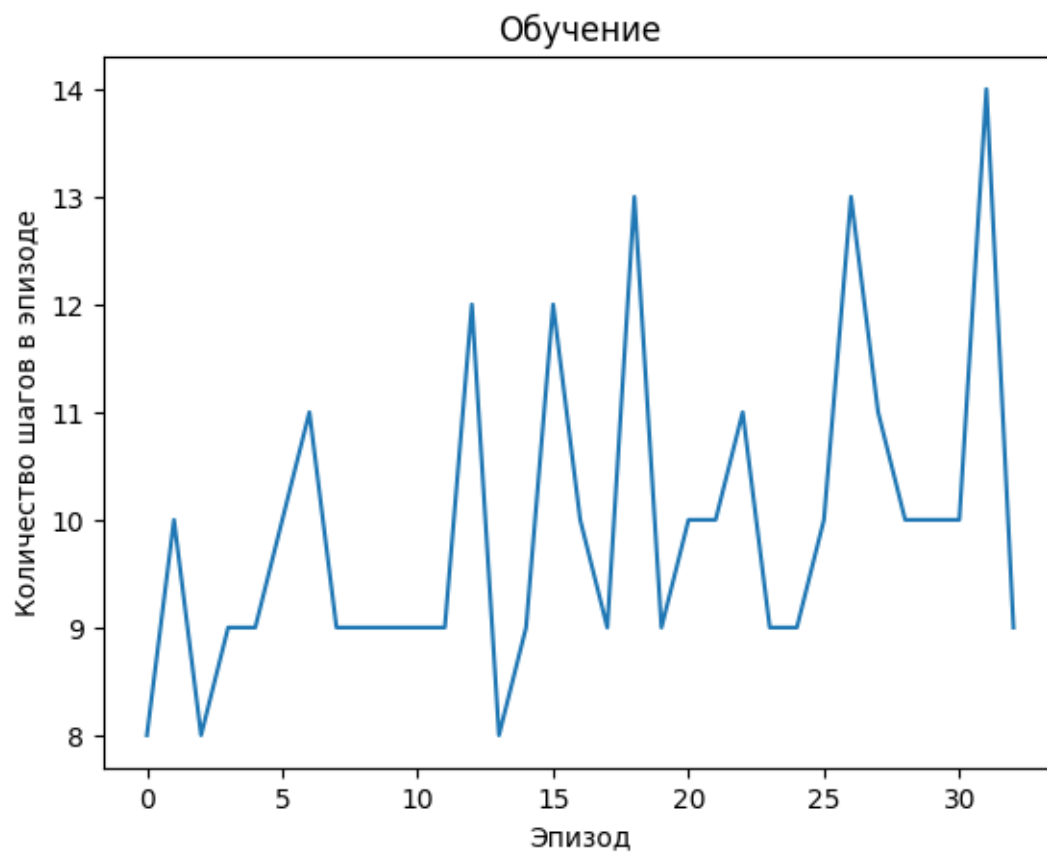


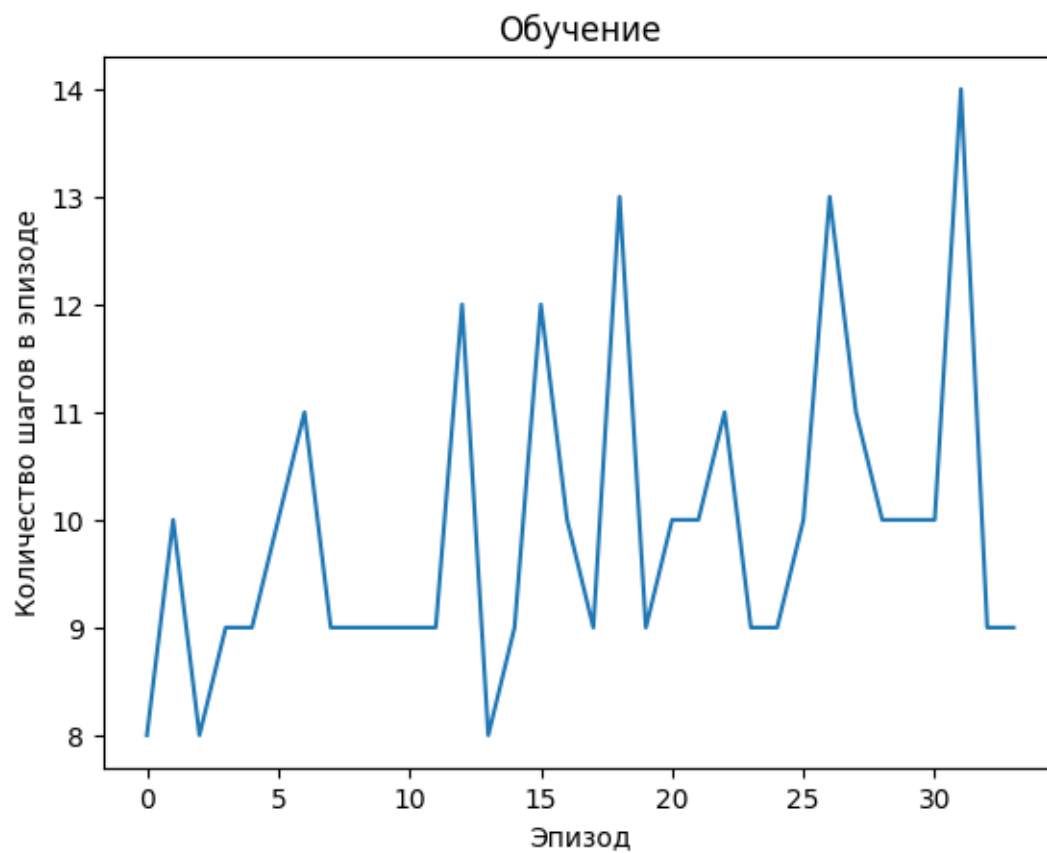


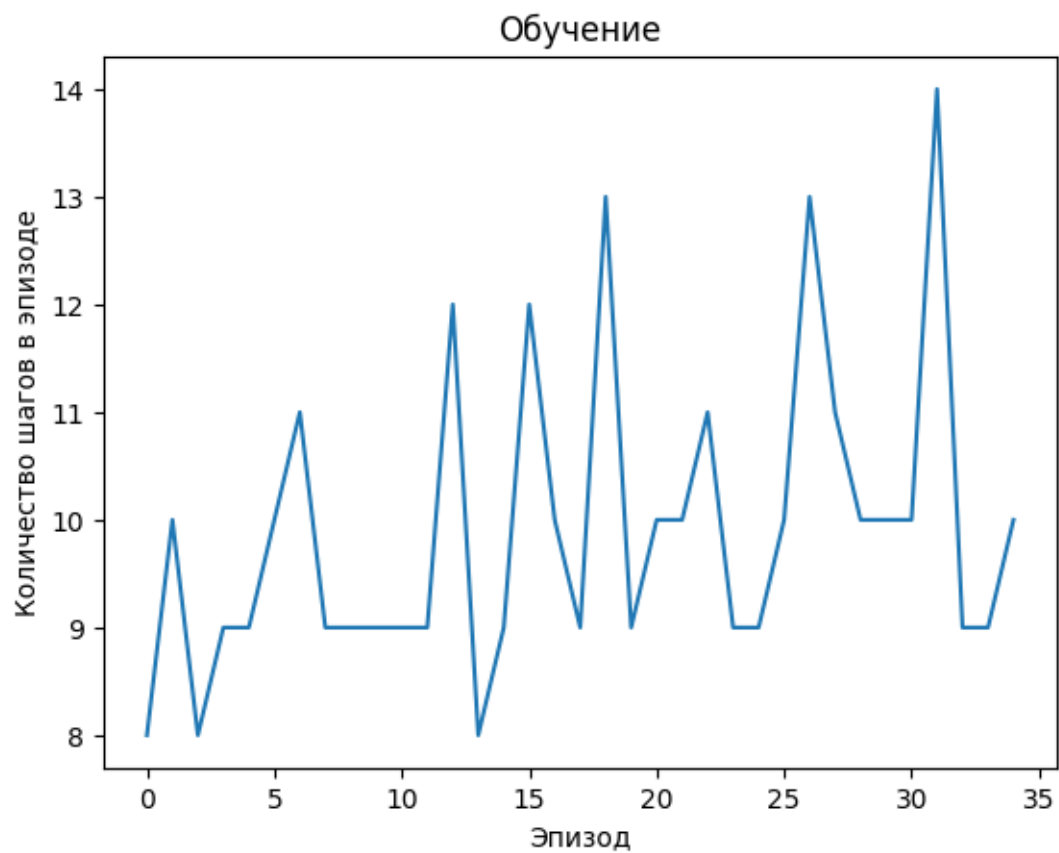


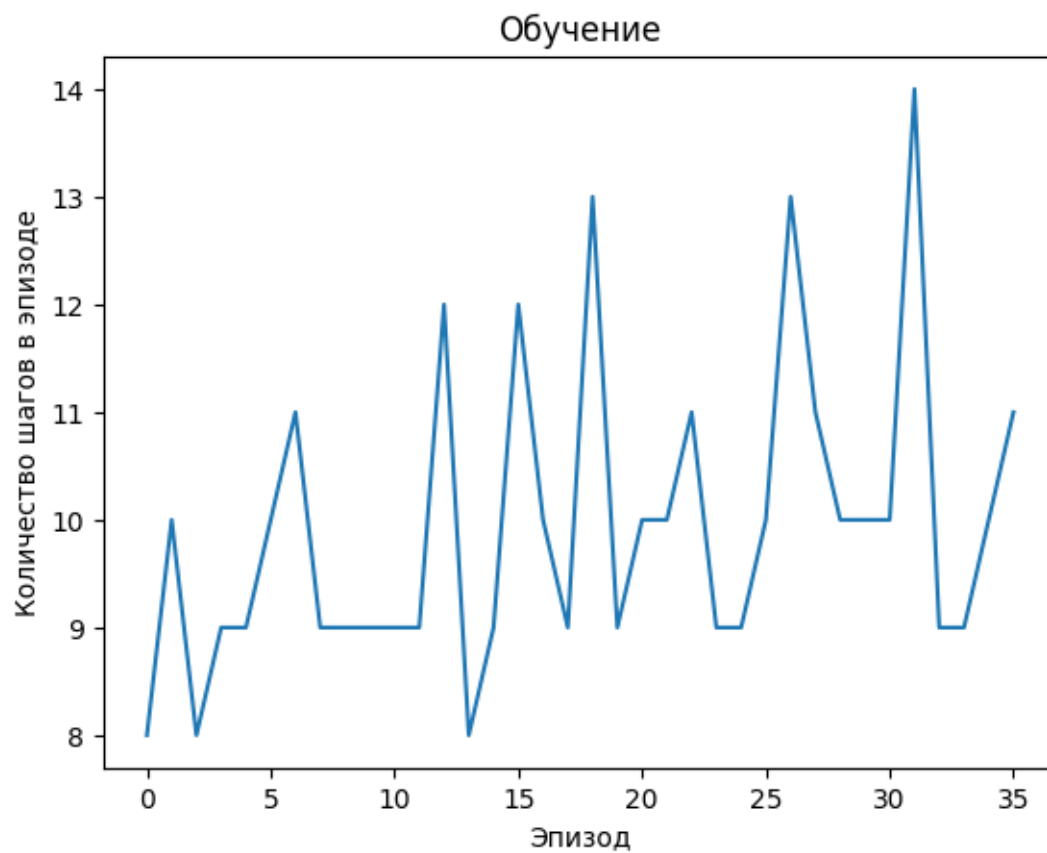


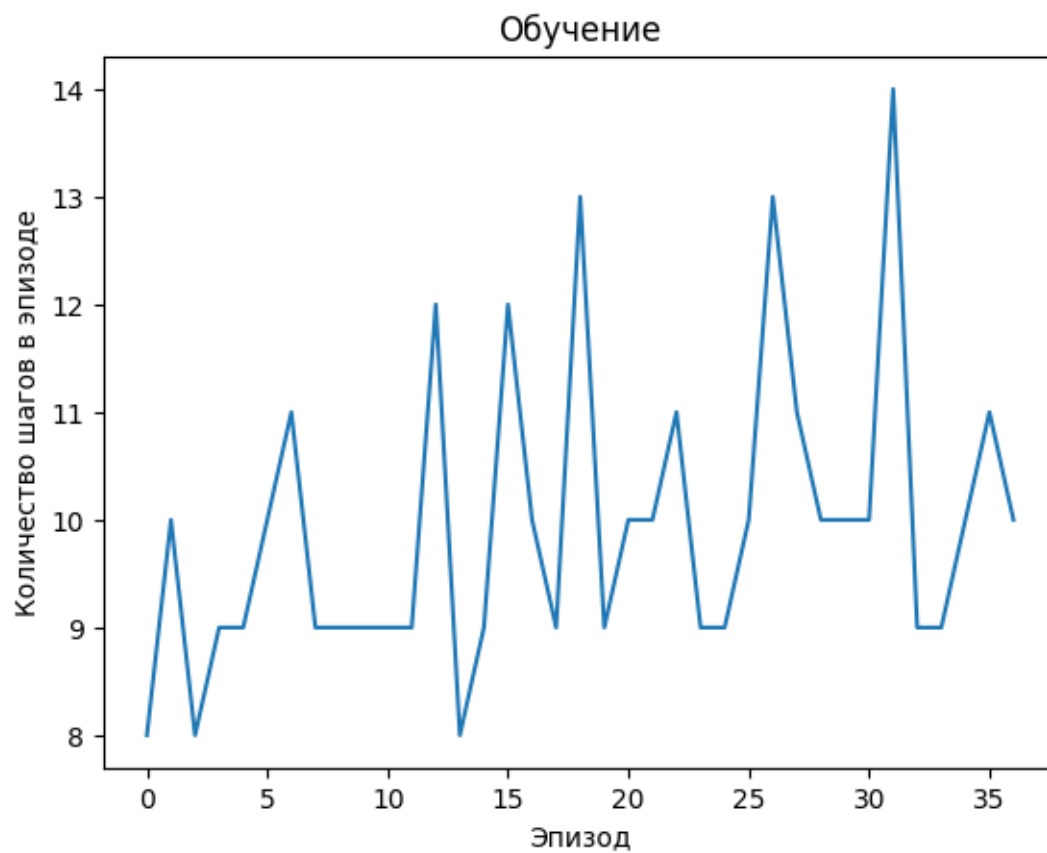


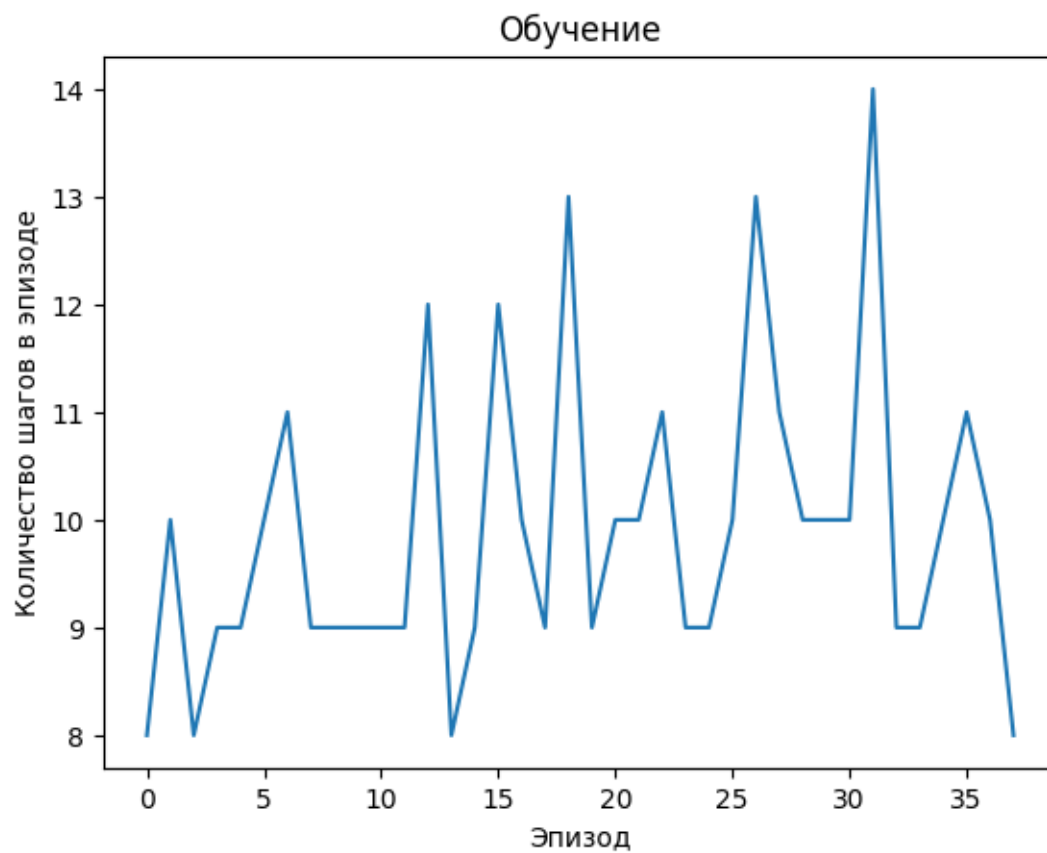


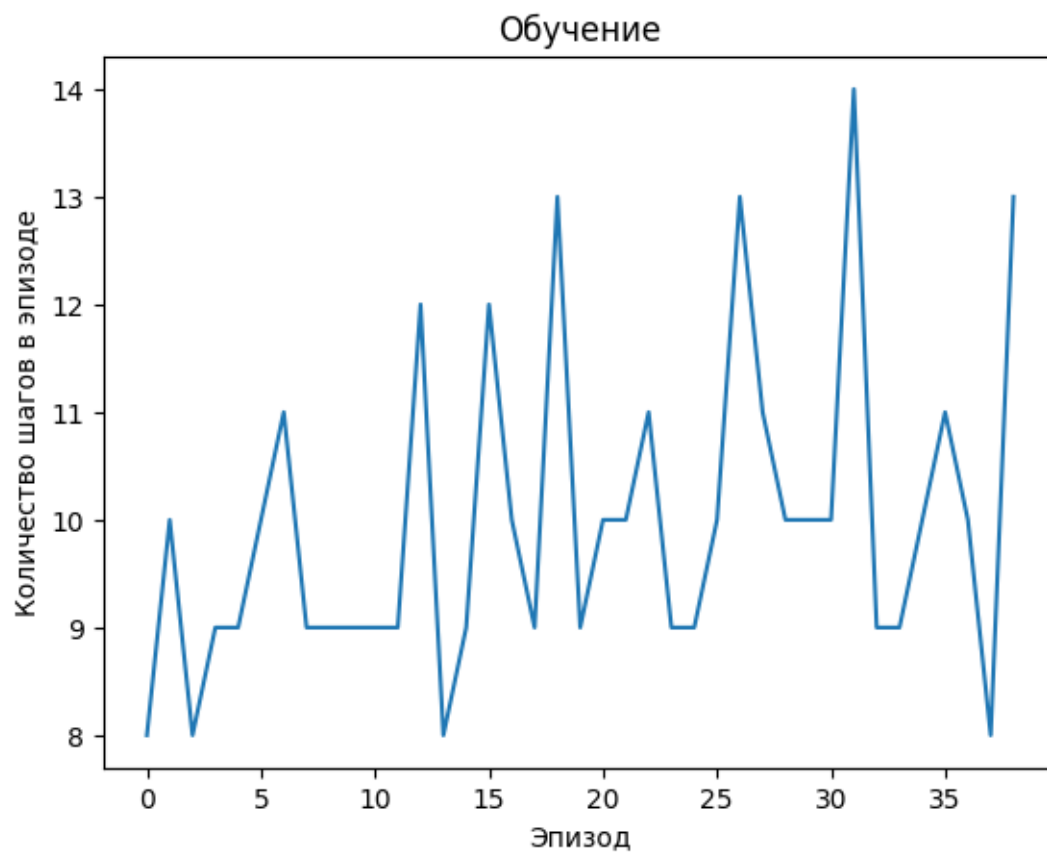


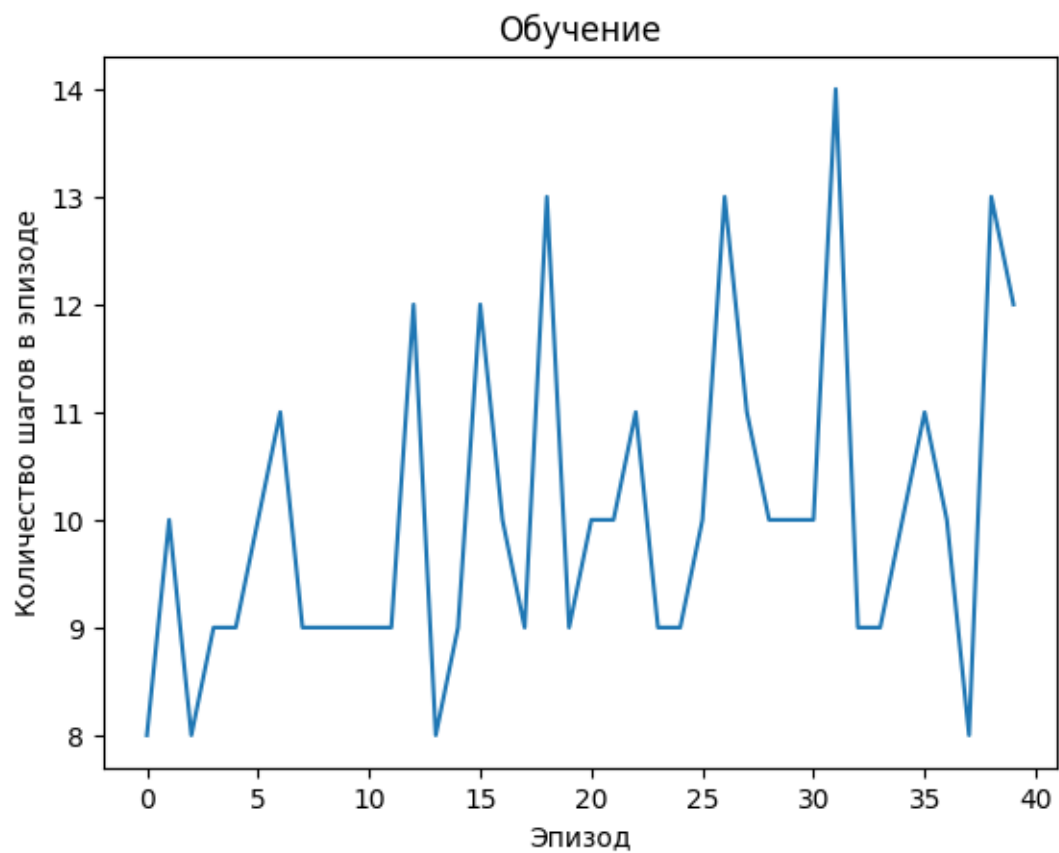


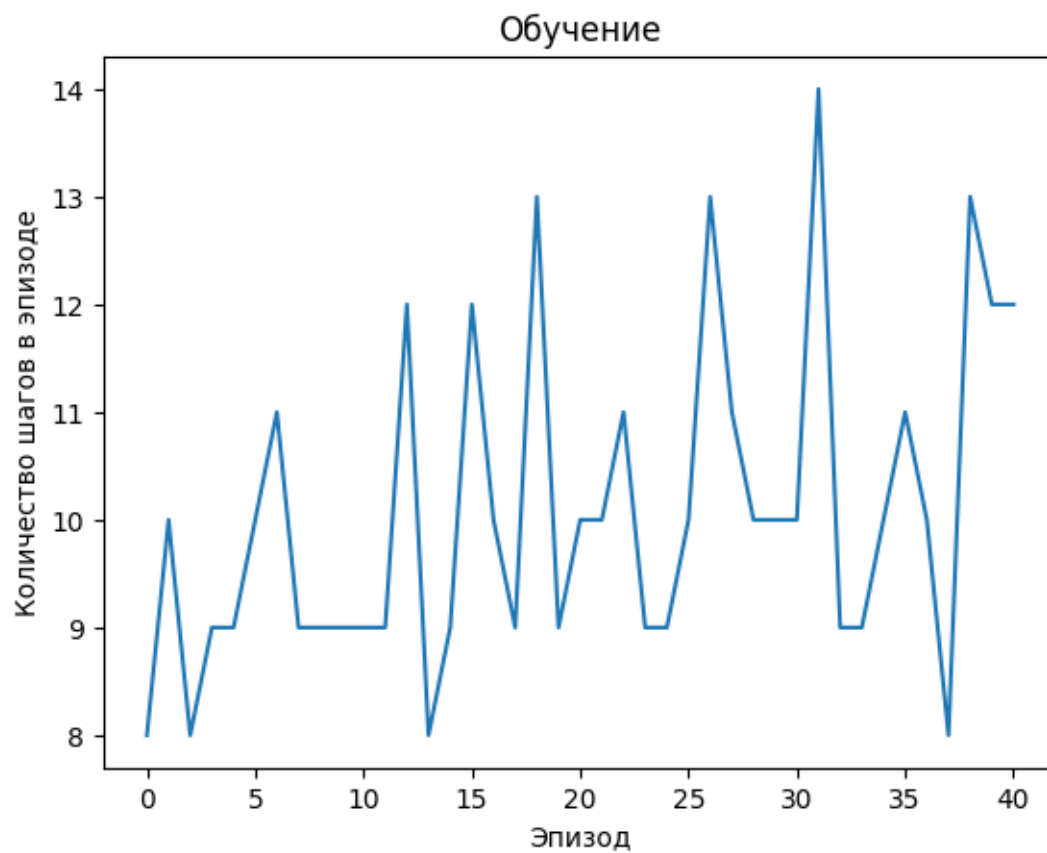


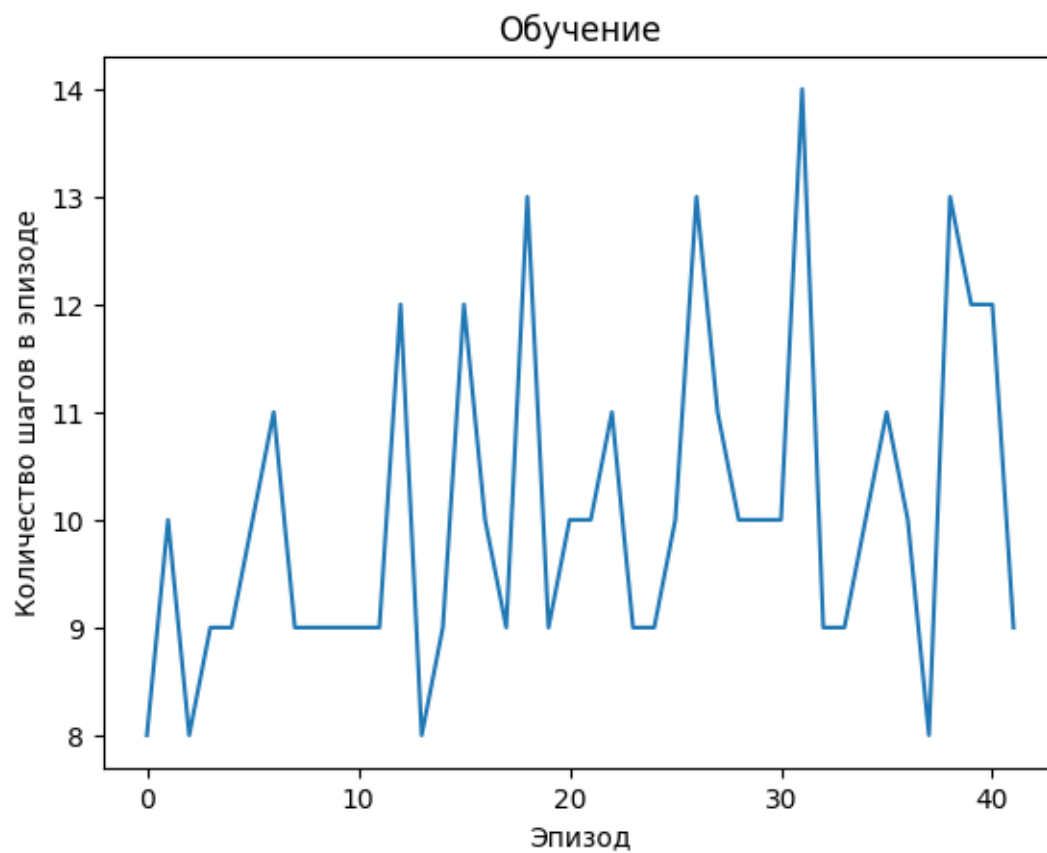


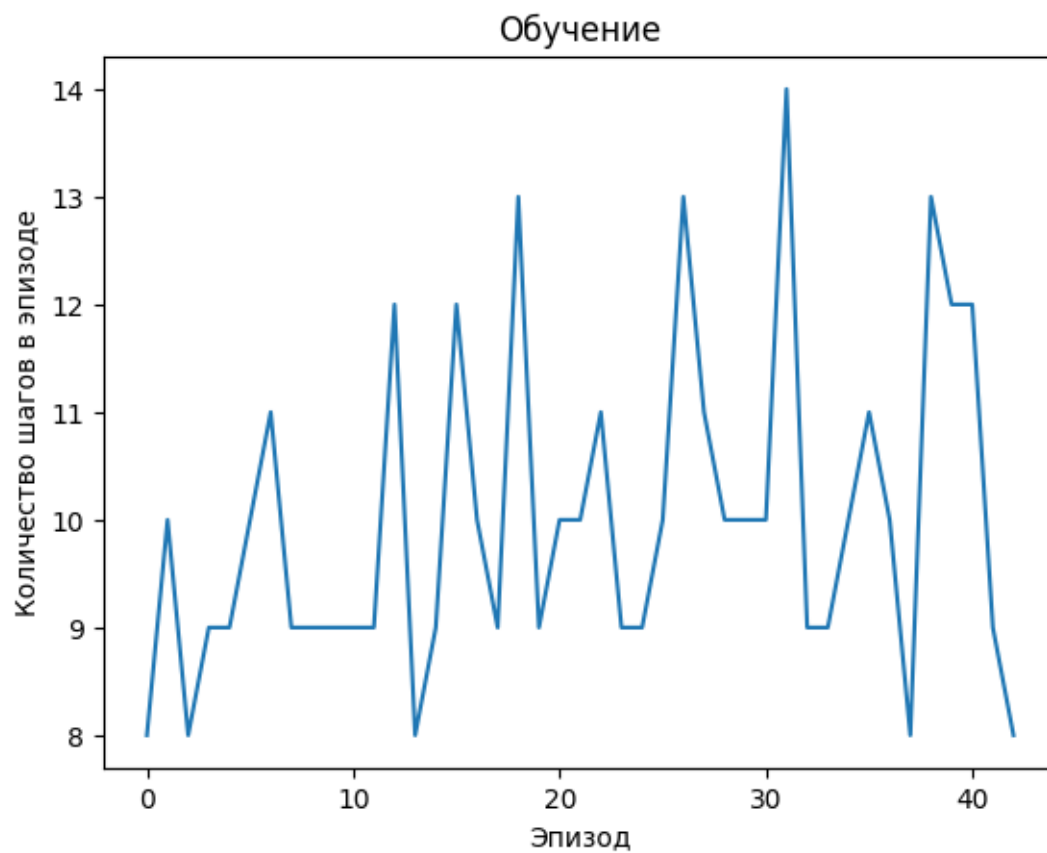


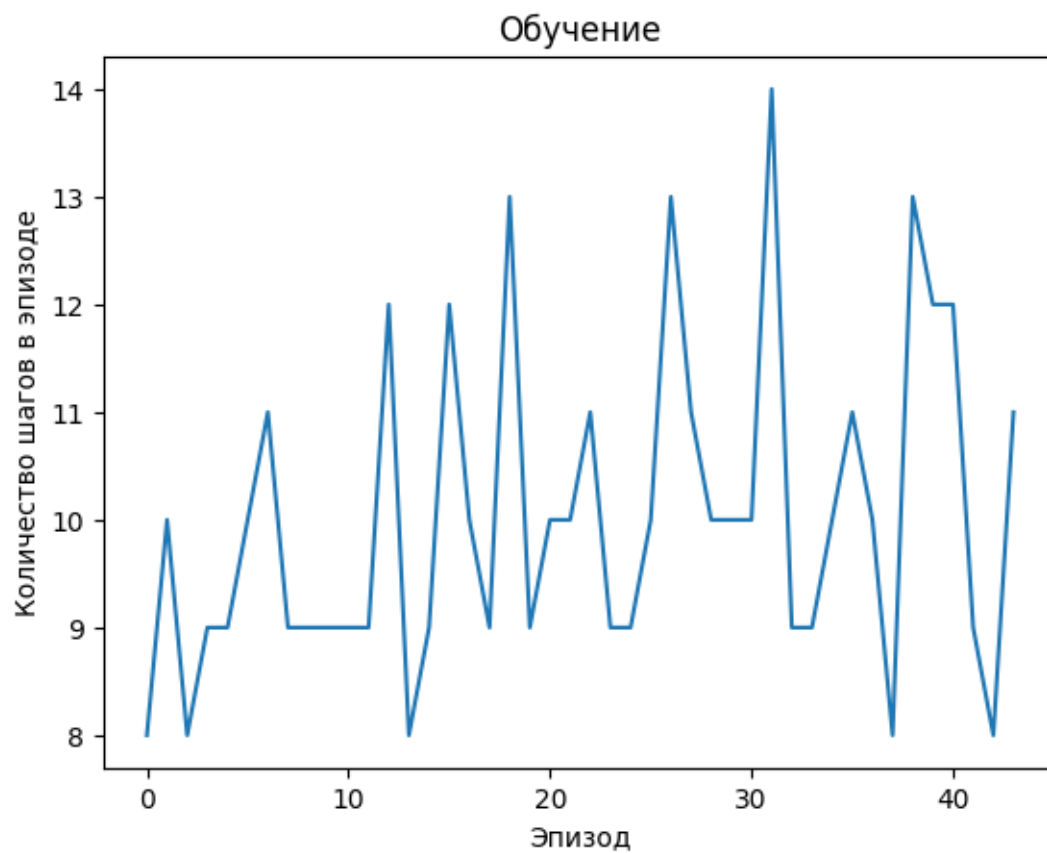


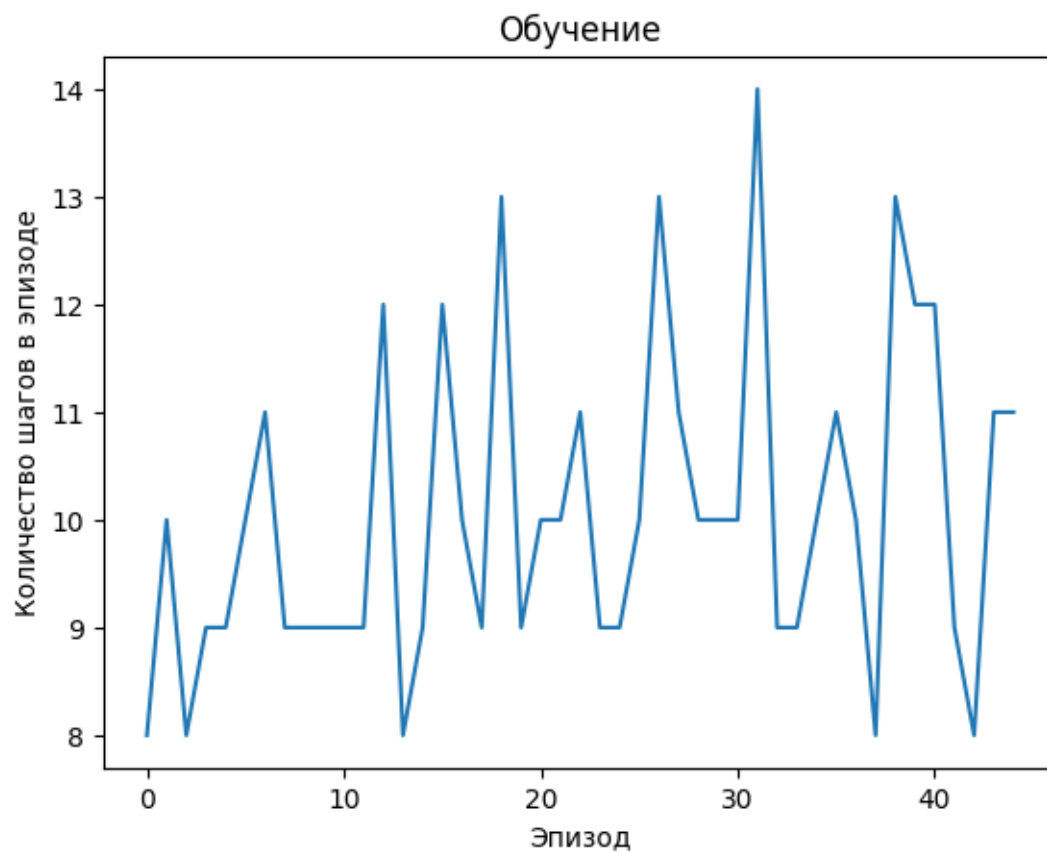


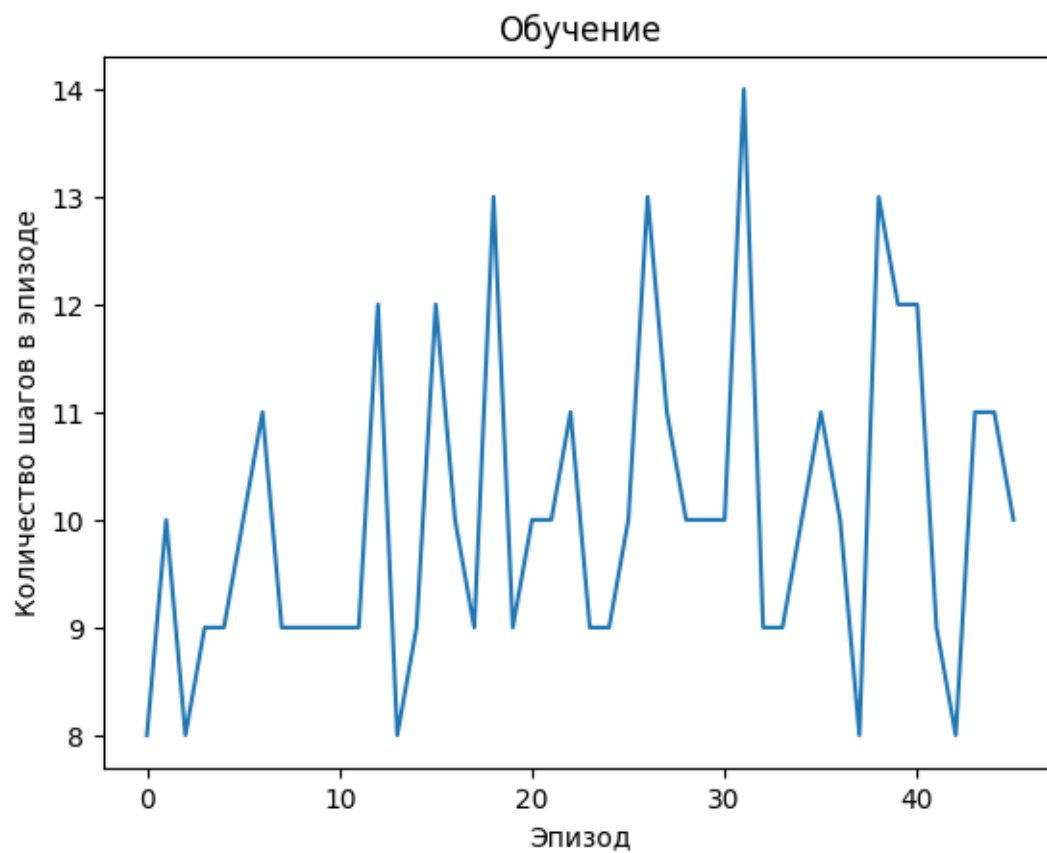


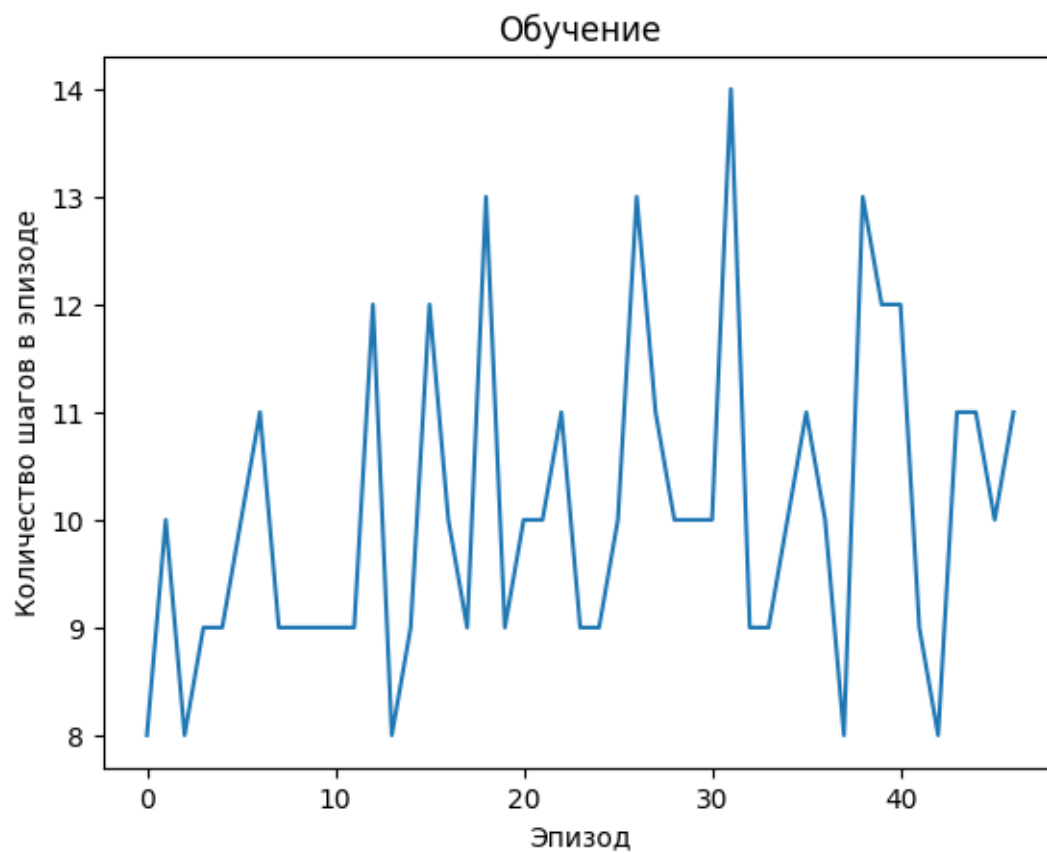


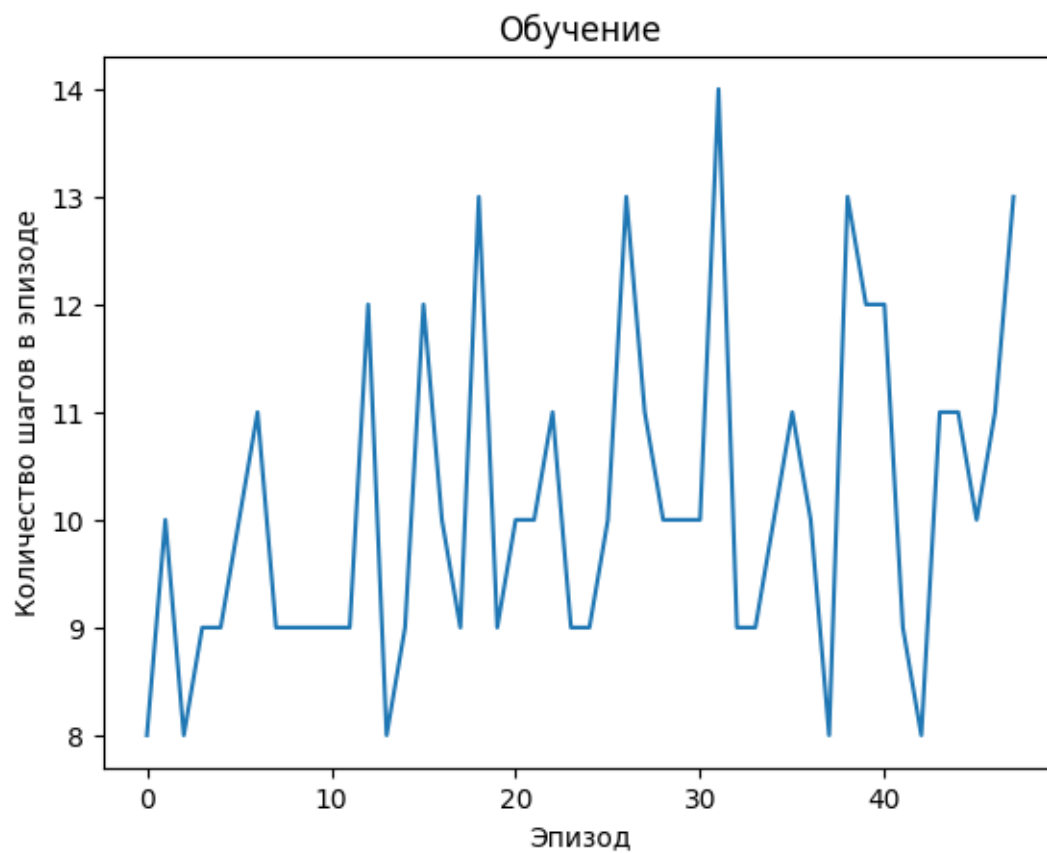


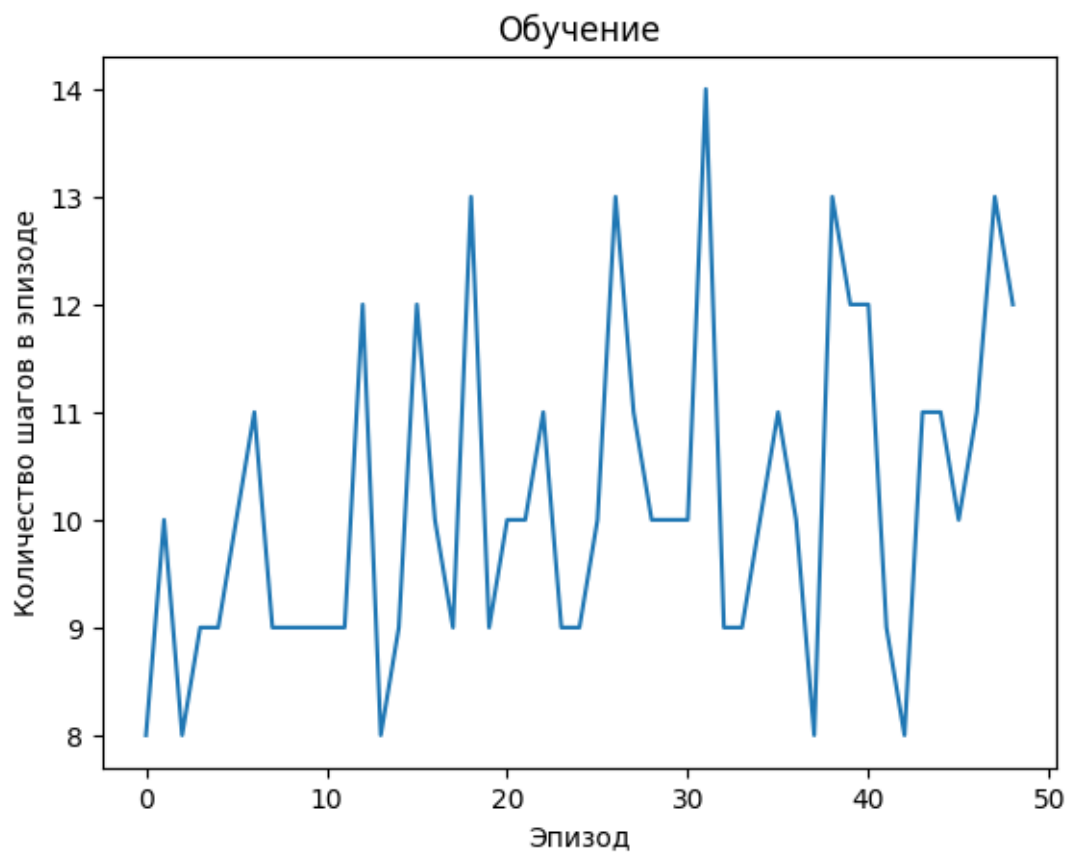


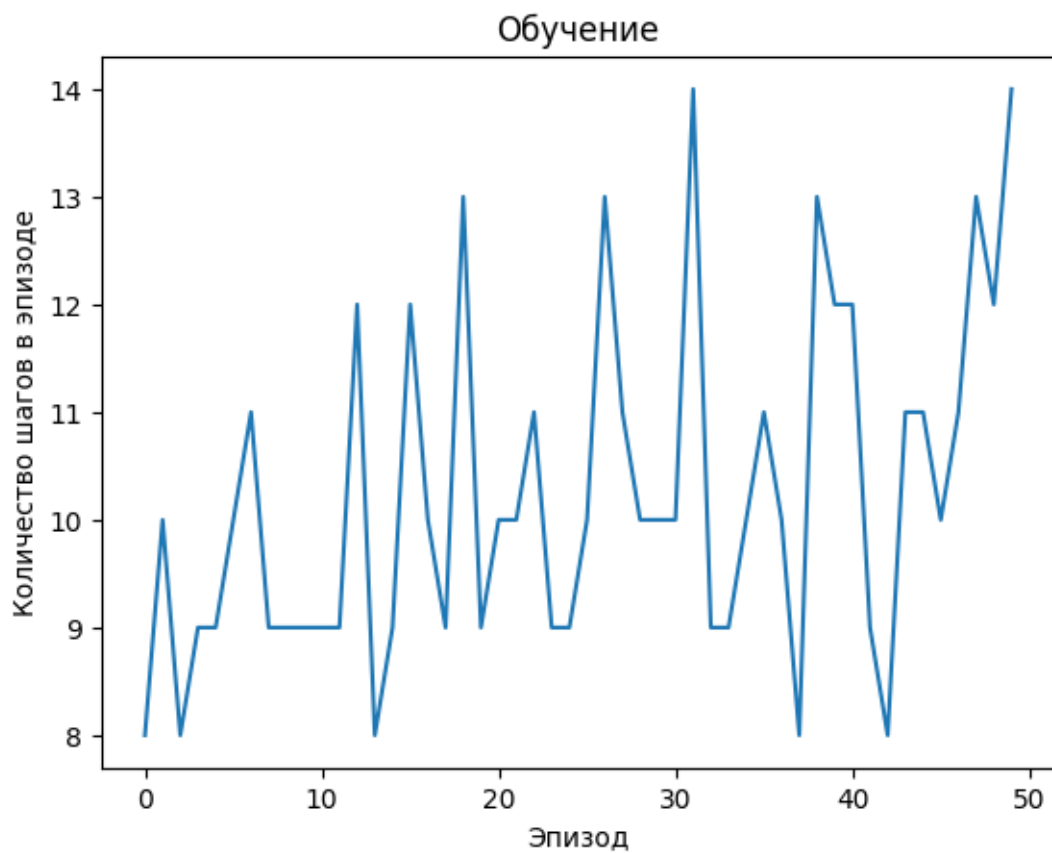












done!

: [(0, 1.0), (0, 1.0), (0, 1.0), (0, 1.0), (0, 1.0), (0, 1.0),
 (1, 1.0), (1, 1.0), (0, 1.0), (0, 1.0), (0, 1.0), (0, 1.0)]