**МИНОБРНАУКИ РОССИИ**
**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**
**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**
**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**
**Кафедра МО ЭВМ**

# ОТЧЕТ

**по лабораторной работе №1**
**по дисциплине «Обучение с подкреплением»**
**Тема: Реализация DQN для среды CartPole-v1**

Студент гр. 0310 _____ Низовцов Р.С.

Преподаватель _____ Глазунов С.А.

Санкт-Петербург
2025

**Цель работы**

Реализовать алгоритм DQN для обучения агента в среде CartPole.

**Задача**

Задания для эксперимента:

1. Измените архитектуру нейросети (например, добавьте слои).
2. Попробуйте разные значения gamma и epsilon_decay.
3. Проведите исследование как изначальное значение epsilon влияет на скорость обучения

**Выполнение работы**

1)Алгоритм DQN реализован с использованием библиотеки PyTorch на языке Python. Код приведен в Приложении А.

За значения по умолчанию были взяты следующие показатели:

- Gamma – 0.99
- Epsilon – 1
- Epsilon_decay – 0.955
- Epsilon_min – 0.01
- Num_steps – 200
- Nem_episodes – 350
- Layers – 4 → 64 → 32 → 2

Результат работы приведен на рис. 1:

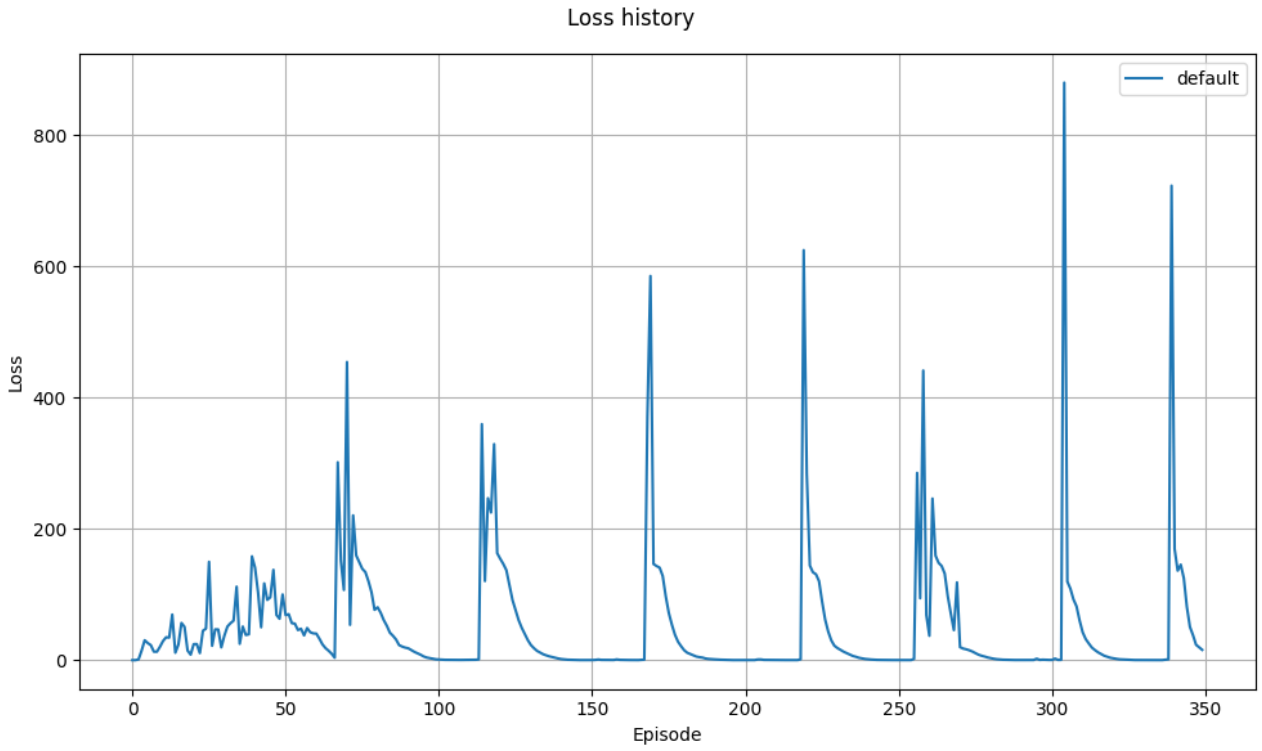


Рисунок 1 – Потери при обучении на значениях по умолчанию

По графику можно предположить, что во время обучения произошел ряд выбросов с падением точности. Постепенно агент адаптировался к данным и возвращал желаемую точность.

2)Были протестированы несколько вариантов набора слоев:

- Стандартный - 4 → 64 → 32 → 2
- С увеличением узлов в слое - 4 → 64 → 128 → 64→ 2
- С уменьшением количества слоев - 4 → 64 → 2
- С увеличением количества слоев - 4 → 64 → 32 → 64→ 32 → 2

Результат работы представлен на рис. 2:

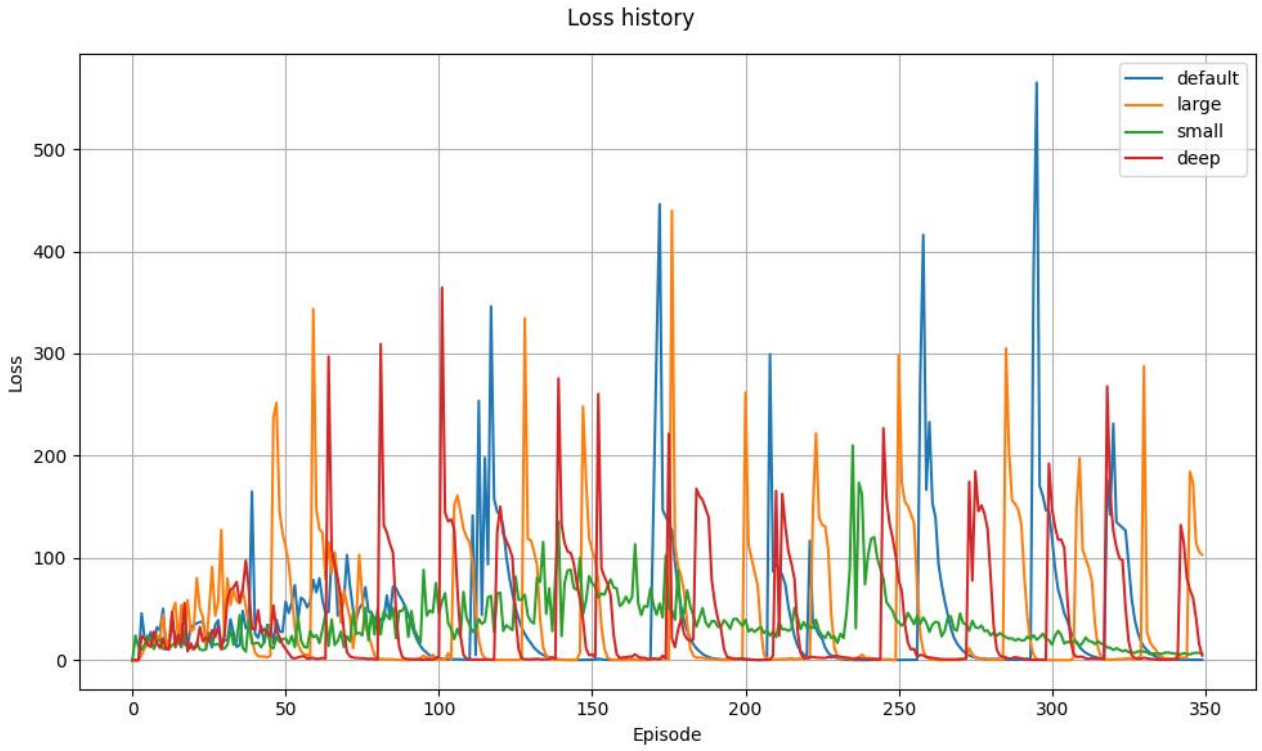


Рисунок 2 – Потери при различных вариантах наборов слоев

Из графика можно отметить, самый стабильный результат был при использовании уменьшенного набора.

3)Далее были протестированы все возможные пары гаммы и коэффициента уменьшения эпсилона с данными значениями:

- Гамма: 0.99, 0.9, 0.85, 0.8, 0.75
- Коэфф. Эпсилон: 0.995, 0.9, 0.85, 0.8, 0.75

На рис. 3-7 представлены результаты всех вариантов коэффициентов с закрепленным значением гаммы:

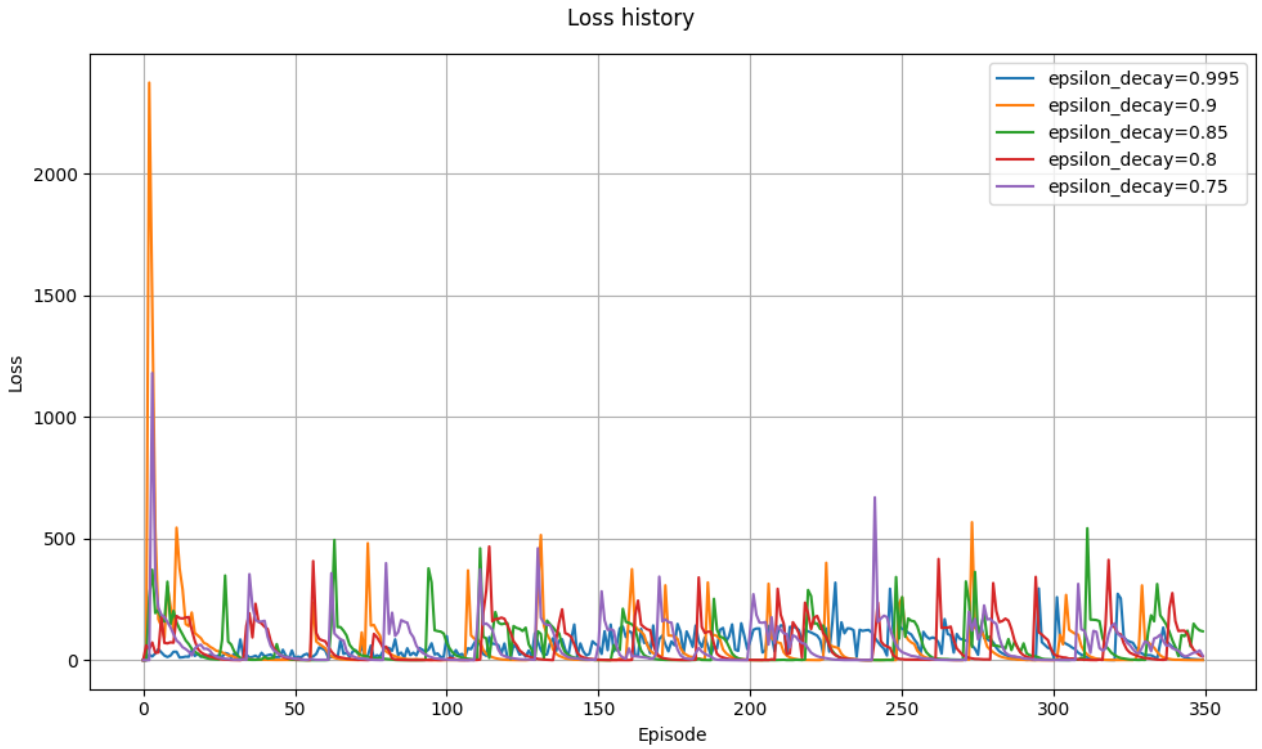


Рисунок 3 – Потери при различных вариантах коэфф. и гаммой 0.99

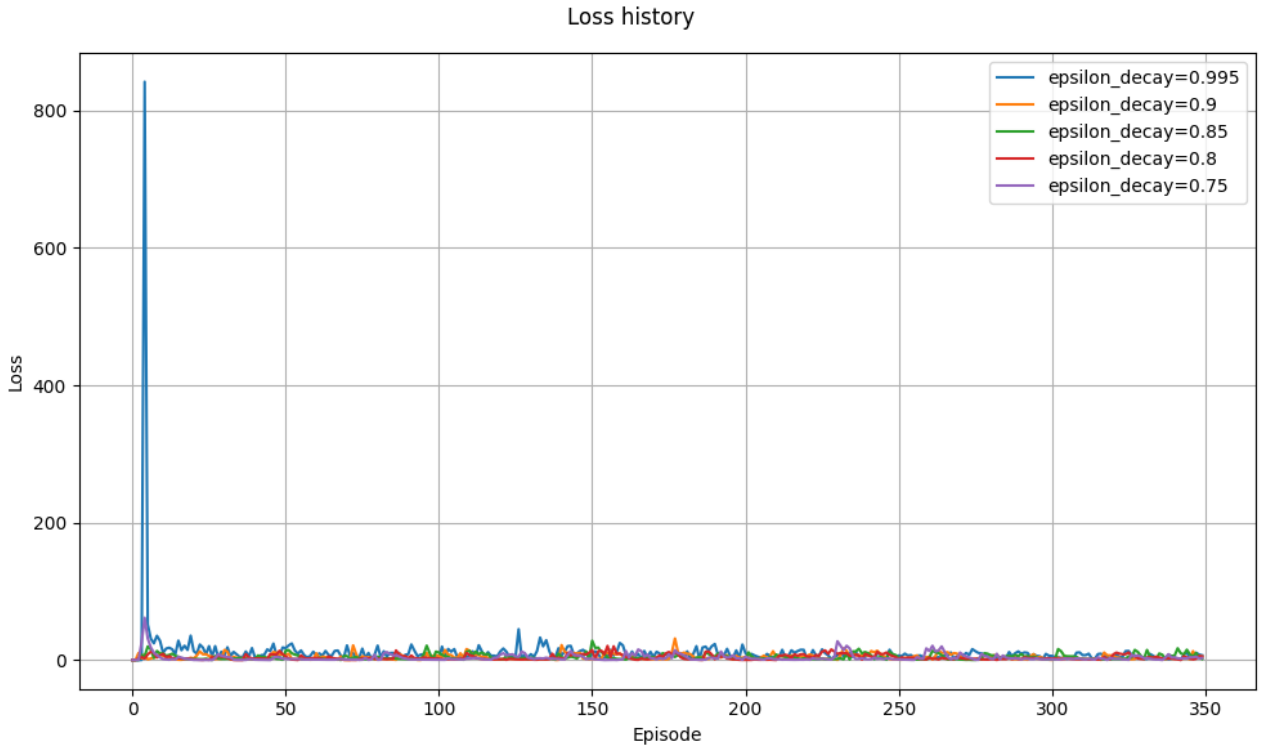


Рисунок 4 – Потери при различных вариантах коэфф. и гаммой 0.9

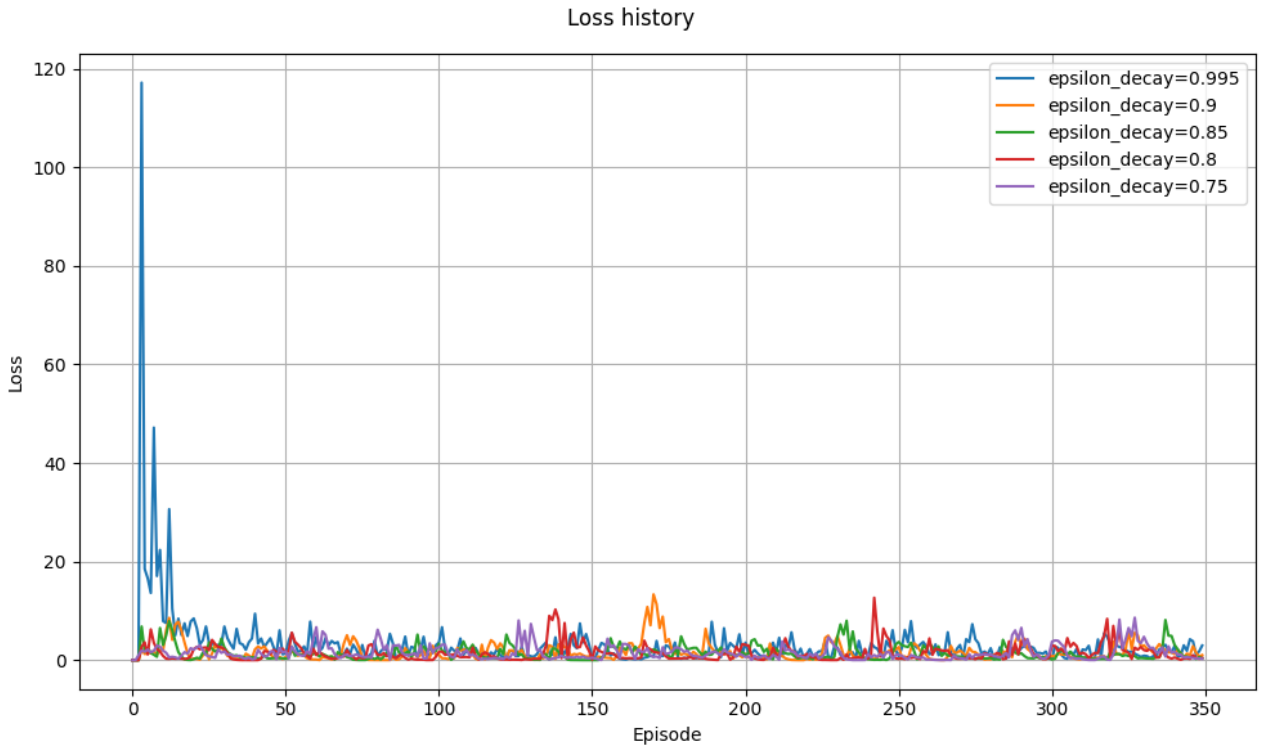


Рисунок 5 – Потери при различных вариантах коэфф. и гаммой 0.85

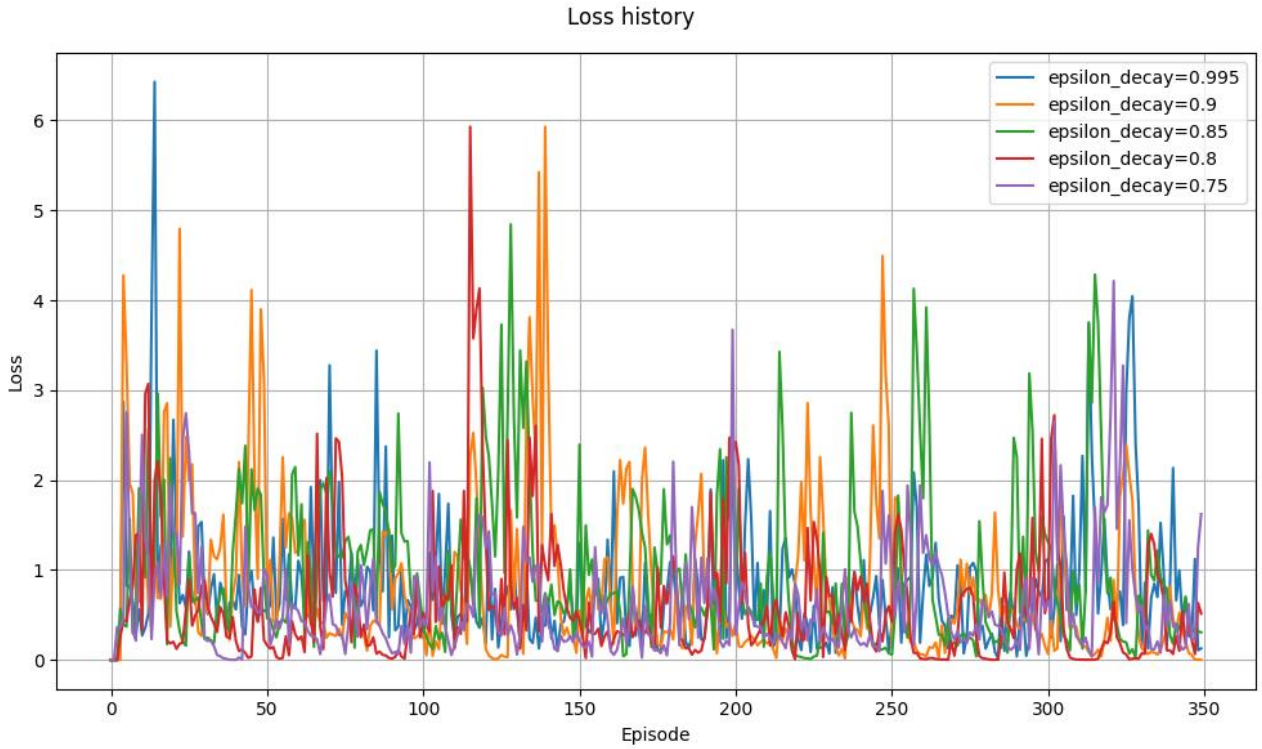


Рисунок 6 – Потери при различных вариантах коэфф. и гаммой 0.8

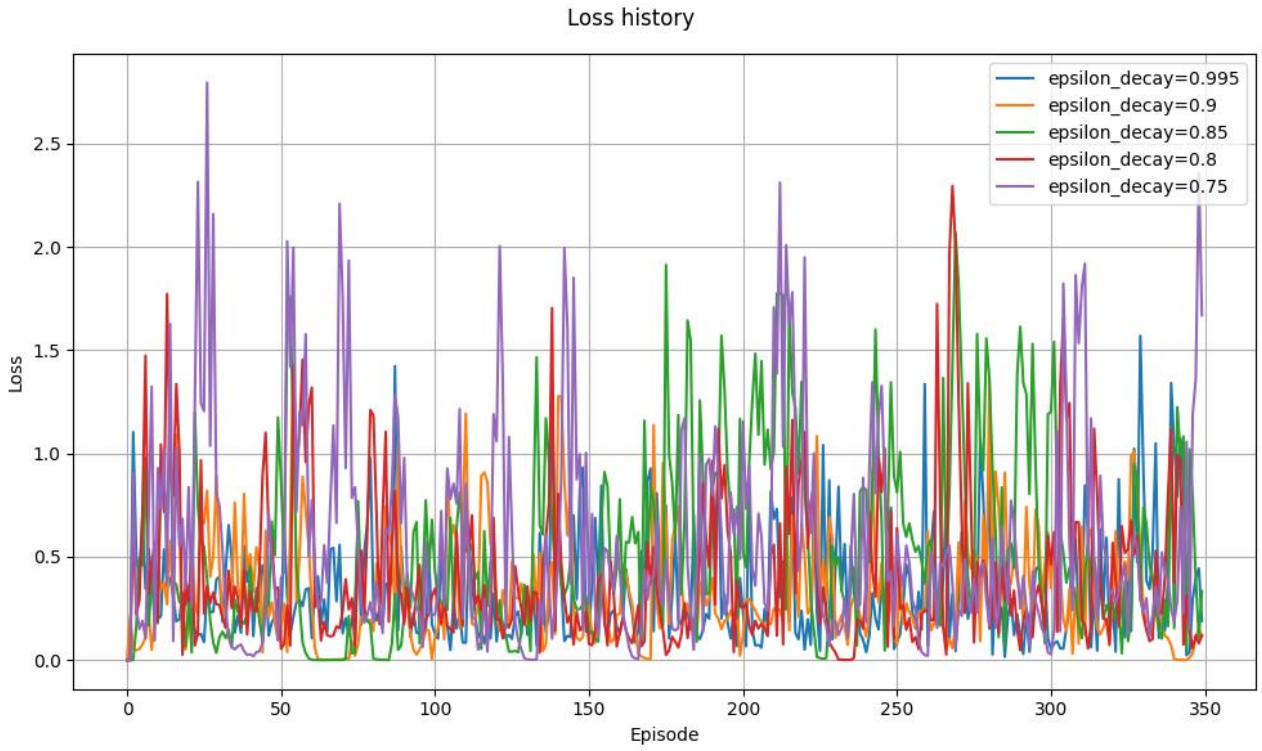


Рисунок 7 - Потери при различных вариантах коэфф. и гаммой 0.75

Из графиков можно сделать вывод, что маленькие значения гаммы и большие значения коэффициента увеличивают количество выбросов и уменьшают точность системы.

4)Далее были протестированы различные значения эпсилон (1, 0.9, 0.7, 0.5, 0.3, 0.1, 0.05, 0.01). Результат представлен на рис. 8:

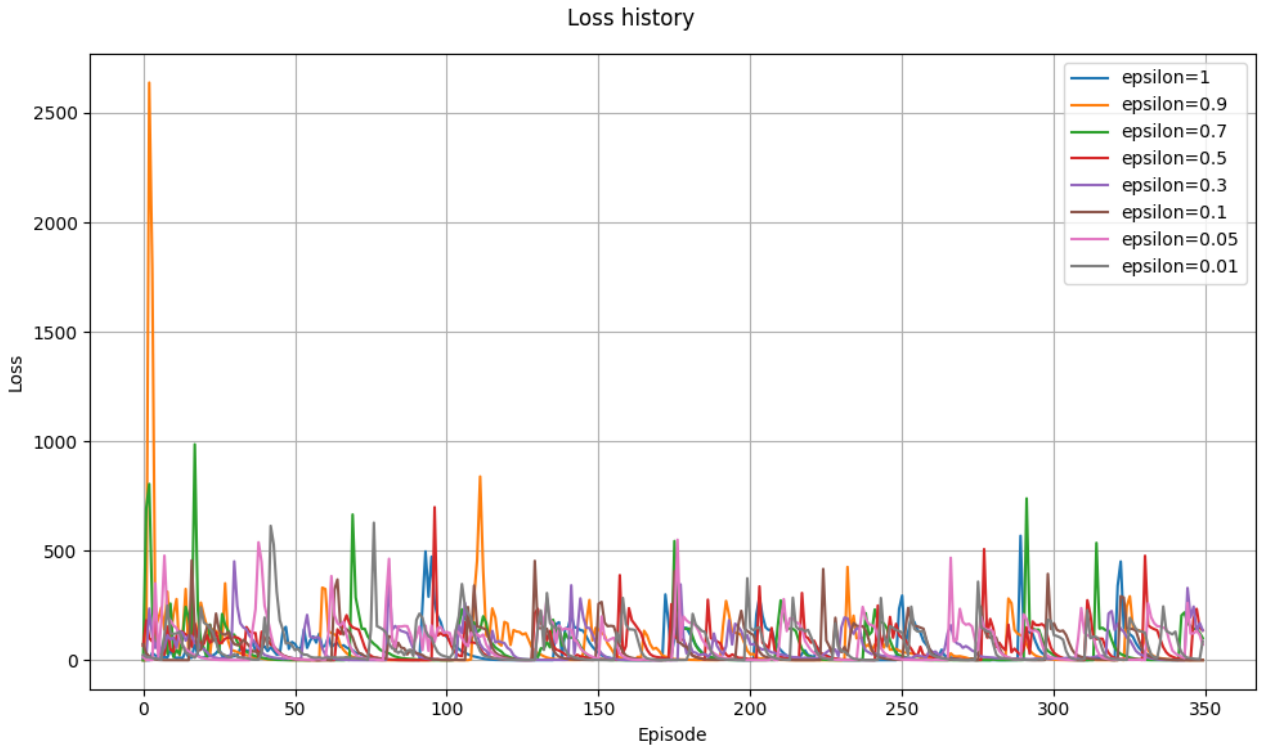


Рисунок 8 – Потери при различных значениях эпсилон

В результате можно сделать вывод, что слишком большие значения эпсилон приводят к большему количеству выбросов.

**Выводы.**

В ходе выполнения лабораторной работы был реализован алгоритм DQN и исследовано влияние различных параметров и архитектур нейросети на процесс обучения агента в среде CartPole.

В результате было установлено, что наилучшим набором является “набор с уменьшенным количеством слоев”. При выборе параметров эпсилон и гамма следует выбирать большие значения, но при этом максимальные из рассмотренных значений также отрицательно влияют на результат.

## ПРИЛОЖЕНИЕ А
## ИСХОДНЫ КОД ПРИЛОЖЕНИЯ

```
import random
from collections import deque

import gymnasium as gym
import matplotlib.pyplot as plt
import numpy as np
import torch
from gymnasium.core import Env
from torch import nn, optim
from tqdm import tqdm


class ReplayBuffer:
    def __init__(self, capacity=1000):
        self.buffer = deque(maxlen=capacity)

    def push(self, state, action, reward, next_state, done):
        self.buffer.append((state, action, reward, next_state, done))

    def sample(self, batch_size):
        batch = random.sample(self.buffer, batch_size)
        state, action, reward, next_state, done = zip(*batch)
        return (
            torch.tensor(np.array(state), dtype=torch.float32),
            torch.tensor(np.array(action), dtype=torch.float32),
            torch.tensor(np.array(reward), dtype=torch.float32),
            torch.tensor(np.array(next_state), dtype=torch.float32),
```

```
            torch.tensor(np.array(done), dtype=torch.float32),
        )

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


class QNetwork(nn.Module):
    def __init__(self, layers: list[nn.Module]):
        super(QNetwork, self).__init__()
        self.net = nn.Sequential(*layers)

    def forward(self, x):
        return self.net(x)


class DQNAgent:
    def __init__(
        self,
        layers: list[nn.Module],
        gamma: float = 0.99,
        epsilon: float = 1.0,
        epsilon_decay: float = 0.955,
        epsilon_min: float = 0.01,
        batch_size: int = 64,
    ):
        self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

        self.q_net = QNetwork(layers).to(self.device)
        self.net_target = QNetwork(layers).to(self.device)
```

```
        self.net_target.load_state_dict(self.q_net.state_dict())

        self.optimizer = optim.Adam(self.q_net.parameters(), lr=1e-3)

        self.gamma = gamma
        self.batch_size = batch_size
        self.epsilon = epsilon

        self.epsilon_decay = epsilon_decay
        self.epsilon_min = epsilon_min

        self.replay_buffer = ReplayBuffer(1000)
        self.loss = 0.0

    def select_action(self, state):
        if random.random() < self.epsilon:
            return random.randint(0, 1)
        else:
            with torch.no_grad():
                state_tensor = torch.tensor(state, dtype=torch.float32,
device=self.device)

                q_values = self.q_net(state_tensor)

            return torch.argmax(q_values).item()

    def train(self):
        if len(self.replay_buffer) < self.batch_size:
            return
```

```
            state, action, reward, next_state, done = self.replay_buffer.sample(self.batch_size)

            state = torch.tensor(state, dtype=torch.float32, device=self.device)
            action = torch.tensor(action, dtype=torch.int64, device=self.device)
            reward = torch.tensor(reward, dtype=torch.float32, device=self.device)
            next_state = torch.tensor(next_state, dtype=torch.float32, device=self.device)
            done = torch.tensor(done, dtype=torch.float32, device=self.device)

            """ dqn update """

            q_values = self.q_net(state).gather(1, action.unsqueeze(1)).squeeze(1)

            next_q_values = self.net_target(next_state).max(1)[0]

            target_q_values = reward + self.gamma * next_q_values * (1 - done)

            loss = nn.MSELoss()(q_values, target_q_values)

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

        def update_target(self):
            self.net_target.load_state_dict(self.q_net.state_dict())


    class Experiment:
```

```
def __init__(
    self,
    env: Env,
    agent: DQNAgent,
    num_episodes: int = 1000,
    num_steps: int = 200,
):
    self.env = env
    self.agent = agent

    self.num_episodes = num_episodes
    self.num_steps = num_steps

    self.loss_history = []

def run(self):
    for _ in tqdm(range(self.num_episodes)):
        state, _ = self.env.reset()

        episode_loss: float = 0.0

        for step in range(self.num_steps):
            action = self.agent.select_action(state)
            next_state, reward, done, _, _ = self.env.step(action)
            self.agent.replay_buffer.push(state, action, reward, next_state, done)

            state = next_state
            self.agent.train()
            episode_loss += float(self.agent.loss)
```

```
                if done:
                    break

            self.agent.update_target()

            self.agent.epsilon = max(self.agent.epsilon * self.agent.epsilon_decay, self.agent.epsilon_min)

            self.loss_history.append(episode_loss)



    def plot(results, file_name):
        fig, ax = plt.subplots(figsize=(10, 6))
        fig.suptitle("Loss history")

        for name, result in results.items():
            ax.plot(result["loss"], label=f"{name}")
            ax.set_xlabel("Episode")
            ax.set_ylabel("Loss")

        ax.legend()
        ax.grid()
        fig.tight_layout()
        fig.savefig(f"{file_name}_loss_history.png")


    layers_pack = {
        "default": [
            nn.Linear(4, 64),
```

```
        nn.ReLU(),
        nn.Linear(64, 32),
        nn.ReLU(),
        nn.Linear(32, 2),
    ],
    "large": [
        nn.Linear(4, 64),
        nn.ReLU(),
        nn.Linear(64, 128),
        nn.ReLU(),
        nn.Linear(128, 64),
        nn.ReLU(),
        nn.Linear(64, 2),
    ],
    "small": [
        nn.Linear(4, 64),
        nn.ReLU(),
        nn.Linear(64, 2),
    ],
    "deep": [
        nn.Linear(4, 64),
        nn.ReLU(),
        nn.Linear(64, 32),
        nn.ReLU(),
        nn.Linear(32, 64),
        nn.ReLU(),
        nn.Linear(64, 32),
        nn.ReLU(),
        nn.Linear(32, 2),
    ],
```

```
}

default_params = {
    "gamma": 0.99,
    "epsilon": 1,
    "epsilon_decay": 0.955,
    "epsilon_min": 0.01,
    "num_steps": 200,
    "num_episodes": 350,
}


def run_default_experiment():
    env = gym.make("CartPole-v1")

    results = {}

    print("Default experiment")
    agent = DQNAgent(
        layers=layers_pack["default"],
        gamma=default_params["gamma"],
        epsilon=default_params["epsilon"],
        epsilon_decay=default_params["epsilon_decay"],
    )

    experiment = Experiment(
        env=env,
        agent=agent,
        num_steps=default_params["num_steps"],
        num_episodes=default_params["num_episodes"],
```

```
    )

    experiment.run()

    results["default"] = {
        "loss": experiment.loss_history,
    }

    plot(results, file_name="default_experiment")


def run_architecture_experiment():
    env = gym.make("CartPole-v1")

    results = {}

    for layers_name, layers in layers_pack.items():
        print(f"Architecture experiment: {layers_name}")

        agent = DQNAgent(
            layers=layers,
            gamma=default_params["gamma"],
            epsilon=default_params["epsilon"],
            epsilon_decay=default_params["epsilon_decay"],
        )

        experiment = Experiment(
            env=env,
            agent=agent,
            num_steps=default_params["num_steps"],
```

```
            num_episodes=default_params["num_episodes"],
        )

        experiment.run()

        results[layers_name] = {
            "loss": experiment.loss_history,
        }

    plot(results, file_name="architecture_experiment")


def run_gamma_decay_experiment():
    env = gym.make("CartPole-v1")
    gammas = [0.99, 0.9, 0.85, 0.8, 0.75]
    epsilon_decays = [0.995, 0.9, 0.85, 0.8, 0.75]

    for gamma_idx in range(0, len(gammas)):

        results = {}

        for epsilon_decay_idx in range(0, len(epsilon_decays)):
            print(
                f"Gamma decay experiment: gamma={gammas[gamma_idx]}," +
                f"epsilon_decay={epsilon_decays[epsilon_decay_idx]}" +
                f"[{len(epsilon_decays)*gamma_idx + epsilon_decay_idx +
1}/{len(gammas) * len(epsilon_decays)}]"
            )

            agent = DQNAgent(
```

```
                layers=layers_pack["default"],
                gamma=gammas[gamma_idx],
                epsilon=default_params["epsilon"],
                epsilon_decay=epsilon_decays[epsilon_decay_idx],
            )

            experiment = Experiment(
                env=env,
                agent=agent,
                num_steps=default_params["num_steps"],
                num_episodes=default_params["num_episodes"],
            )

            experiment.run()

            results[f"epsilon_decay={epsilon_decays[epsilon_decay_idx]}"] = {
                "loss": experiment.loss_history,
            }

        plot(results, file_name=f"gamma_{gamma_idx}_experiment")


def run_epsilon_experiment():
    env = gym.make("CartPole-v1")
    epsilons = [1, 0.9, 0.7, 0.5, 0.3, 0.1, 0.05, 0.01]

    results = {}

    for idx in range(0, len(epsilons)):
```

```
        print(f"Epsilon experiment: epsilon={epsilons[idx]} [{idx+1}/{len(epsilons)}]")

        agent = DQNAgent(
            layers=layers_pack["default"],
            gamma=default_params["gamma"],
            epsilon=epsilons[idx],
            epsilon_decay=default_params["epsilon_decay"],
        )

        experiment = Experiment(
            env=env,
            agent=agent,
            num_steps=default_params["num_steps"],
            num_episodes=default_params["num_episodes"],
        )

        experiment.run()

        results[f"epsilon={epsilons[idx]}"] = {
            "loss": experiment.loss_history,
        }

    plot(results, file_name="epsilon_experiment")


if __name__ == "__main__":
    run_default_experiment()
    run_architecture_experiment()
    run_gamma_decay_experiment()
```

```
run_epsilon_experiment()
```