Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное учреждение

высшего образования

«Казанский (Приволжский) федеральный университет»

*Институт вычислительной математики и информационных технологий*

**ОТЧЁТ**

**по производственной технологической (проектно-технологической) практике**

|  |  |
| --- | --- |
| Обучающийся Близнюк В.Ю., 09-052  *(ФИО, группа)* | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  *(подпись)* |

Руководитель практики от КФУ

ст. преподаватель кафедры анализа данных

и технологий программирования Жажнева И.В. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

*(должность, ФИО) (подпись)*

Оценка за практику \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(подпись руководителя)

Дата сдачи отчета \_\_23.05.2024\_\_\_\_\_

Казань, 2024

Содержание

[АННОТАЦИЯ 3](#_Toc167657720)

[Введение 4](#_Toc167657721)

[1. Проектирование приложения 5](#_Toc167657722)

[1.1. Реализация нейронной сети 5](#_Toc167657723)

[1.2. Заключение по приложению и циклу его работы 13](#_Toc167657724)

[2. Тестирование приложения и моделей 16](#_Toc167657725)

[Заключение 17](#_Toc167657726)

[Список использованных источников 18](#_Toc167657727)

[ПРИЛОЖЕНИЕ 20](#_Toc167657728)

АННОТАЦИЯ

Цель настоящей работы заключается в разработке информационной системы, которая позволяет создавать модели нейронной сети, способные проходить гоночные трассы в видеоигре TrackMania Nations Forever. В рамках данной работы была исследована предметная область, проведен сравнительный анализ готовых решений, составлен ряд функциональных и нефункциональных требований к системе, спроектирована схема базы данных. Была затронута проблема проектирования алгоритма обучения нейронной сети, для решения которой было решено использовать DQN-алгоритм обучения с подкреплением.

Обучающая часть системы была разработана с применением алгоритма DQN. Создание приложения производилось с использованием библиотеки создания окружения gym, библиотеки для машинного обучения Torch и ряда дополнительных библиотек.

Связь между приложением и видеоигрой была реализована с помощью TMInterface и соответствующей Python библиотеки.

Ключевые слова: Машинное обучение с подкреплением, DQN, TrackMania Nations Forever, TMInterface, Python, Torch, gym.

# Введение

Целью производственной технологической (проектно-технологической) практики направления подготовки 09.03.03 «Прикладная информатика» согласно программе практики является закрепление теоретических знаний, практических умений и навыков, полученных в процессе освоения дисциплин основной образовательной программы приобретение практического опыта решения профессиональных задач. При прохождении данной практики обучающийся опирается на материалы ранее освоенных дисциплин (модулей) и/или практик: Информатика, Информационные технологии, Архитектура информационных систем, Управление данными, Технологии программирования, Ознакомительная практика.

Освоение данной практики способствует эффективному выполнению и защите выпускной квалификационной работы.

Основными задачами производственной технологической (проектно-технологической) практики являются:

-овладеть навыками использования отечественных и международных стандартов, норм и правил; навыками разработки технической документации на всех стадиях жизненного цикла информационной системы;

- овладеть практическими навыками разработки программного обеспечения;

- овладеть навыками выполнения работ на всех стадиях жизненного цикла создания информационных систем; навыками использования инструментальных средств управления проектами в области разработки информационных систем.

Технологической (проектно-технологической) практика проходила на базе Института вычислительной математики и информационных технологий Казанского (Приволжского) федерального университета.

Сроки прохождения практики: с 08.04.2024 по 23.05.2024.

## 1. Проектирование приложения

## 1.1. Реализация нейронной сети

Для обучения с подкреплением было решено использовать метод обучения DQN (Deep Q-Network). Его основными компонентами являются:

* + Q-значение – Q(s,a) предствавляет собой ожидаемую суммарную награду, которую агент получит, начиная из состояния s, выполняя действие a и следуя некоторой стратегии в будущем.
  + Нейронная сеть – для аппроксимации функций Q-значений используется глубокая нейронная сеть. Входом сети является состояние s, а на выходе сеть выдает Q-значения для всех возможных действий a в этом состоянии.
  + Целевая сеть – DQN использует две сети: основную (policy) сеть и целевую (target) сеть. Целевая сеть копирует веса основной сети через определенные интервалы времени. Это помогает стабилизировать обучение, так как целевые значения изменяются медленнее.
  + Память воспроизведения опыта (Replay Memory) – опыт агента (переходы) сохраняется в буфере воспроизведения. Переходы случайным образом выбираются из буфера для обучения. Это снижает корреляцию между последовательными шагами и делает обучение более стабильным и эффективным.

Далее каждый из компонентов будет описан более подробно.

В памяти воспроизведения опыта (Replay Memory) хранятся переходы, которые наблюдает агент, что позволяет повторно использовать эти данные в дальнейшем. Путем случайной выборки из нее переходы, составляющие партию, декоррелируются, что значительно стабилизирует и улучшает процедуру обучения сети.

Для этого понадобилось 2 класса:

* + Transition – именованный кортеж, представляющий один переход в окружении. Он сопоставляет пары (состояние, действие) с их результатом (следующее\_состояние, награда).
  + ReplayMemory – циклический буфер ограниченного размера, в котором хранятся переходы, наблюдавшиеся в последнее время. Он также реализует метод sample для выбора случайной партии переходов для обучения.

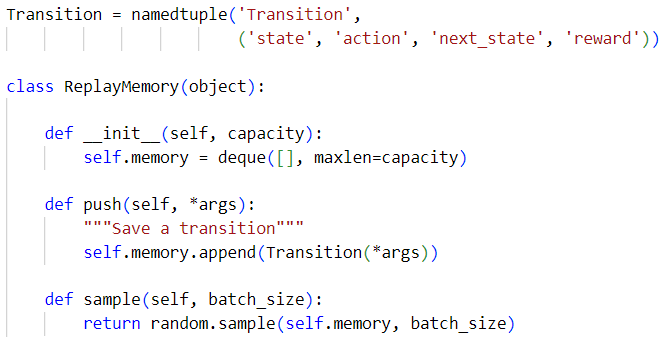


Рисунок 1. Реализация классов памяти воспроизведения опыта

Основной целью алгоритма обучения с подкреплением является обучение политики, которая пытается максимизировать дисконтированное кумулятивное вознаграждение

,

где дисконт γ является константой от 0 до 1. Низкие значения дисконта делают вознаграждения из неопределенного далекого будущего менее важными для агента, чем те, которые он может получить в ближайшем будущем и в которых он может быть достаточно уверен. Они также побуждают агентов получать вознаграждения ближе по времени, чем эквивалентные вознаграждения, которые находятся в будущем.

Основная идея Q-обучения заключается в том, что если бы была функция Q\*: State×Action→R, которая могла бы сказать, какой будет награда, если будет предпринято действие в данном состоянии, то можно было бы легко построить политику, которая максимизирует вознаграждение:

.

Однако агент не знает всего о мире, поэтому у него нет доступа к Q\*. Но поскольку нейронные сети являются универсальными аппроксиматорами функций, можно просто создать такую сеть и обучить ее, чтобы она напоминала Q\*.

Для правила обновления обучения было решено использовать факт, что каждая функция Q для некоторой политики подчиняется уравнению Беллмана:

.

Разность между двумя сторонами равенства называется погрешностью временной разности, δ:

.

Чтобы минимизировать эту ошибку, мы будем использовать потерю Хьюбера. Функция потерь Хьюбера действует как средняя квадратичная ошибка, когда ошибка мала, и как средняя абсолютная ошибка, когда ошибка велика – это делает его более устойчивым к выбросам, когда оценки Q очень зашумлены. Это вычисляется для пакетов переходов B, взятых из памяти воспроизведения опыта:

,

где

Модель представляет собой нейронную сеть с прямой передачей данных, которая учитывает разницу между текущим и предыдущим состояниями. У нее есть два выхода, представляющие собой Q(s,left) и Q(s,right) (где s - вход сети). По сути, сеть пытается предсказать ожидаемую прибыль от каждого действия с учетом текущих входных данных.

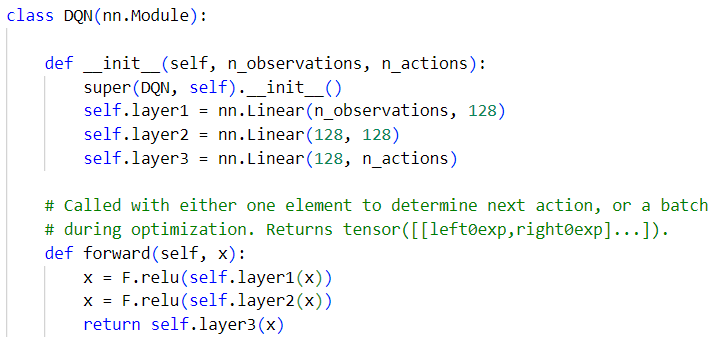


Рисунок 2. Реализация класса нейронной сети

Скрипт обучения имеет много различных параметров, основные из которых было решено выделить в отдельный класс, как гиперпараметры.

* + BATCH\_SIZE = 128 – размер пакета переходов, которые выбираются из памяти воспроизведения опыта для каждого шага обучения. Большой размер пакета может стабилизировать обучение за счет более точной оценки градиента, но также требует больше памяти и вычислительных ресурсов. Выбранный размер пакета в 128 является распространенным выбором, который балансирует эти факторы.
  + GAMMA = 0.99 – дисконтирующий фактор для подкрепления, описанный ранее.
  + EPS\_START = 0.95 – начальное значение эпсилон в эпсилон-жадной политике. Эпсилон управляет балансом между исследованием (пробованием новых действия) и эксплуатацией (выбором наилучшего известного действия). Высокое начальное значение эпсилон означает, что агент будет много исследовать в начале, пробуя разные действия для сбора информации об окружении.
  + EPS\_END = 0.05 – конечное значение эпсилон после его снижения. Это значение, до которого эпсилон уменьшится со временем. Низкое конечное значение эпсилон означает, что в конечном итоге агент будет в основном использовать то, что он уже выучил, но иногда будет исследовать, чтобы избежать застревания в субоптимальном поведении.
  + EPS\_DECAY = 2500 – скорость, с которой эпсилон уменьшается от EPS\_START до EPS\_END.

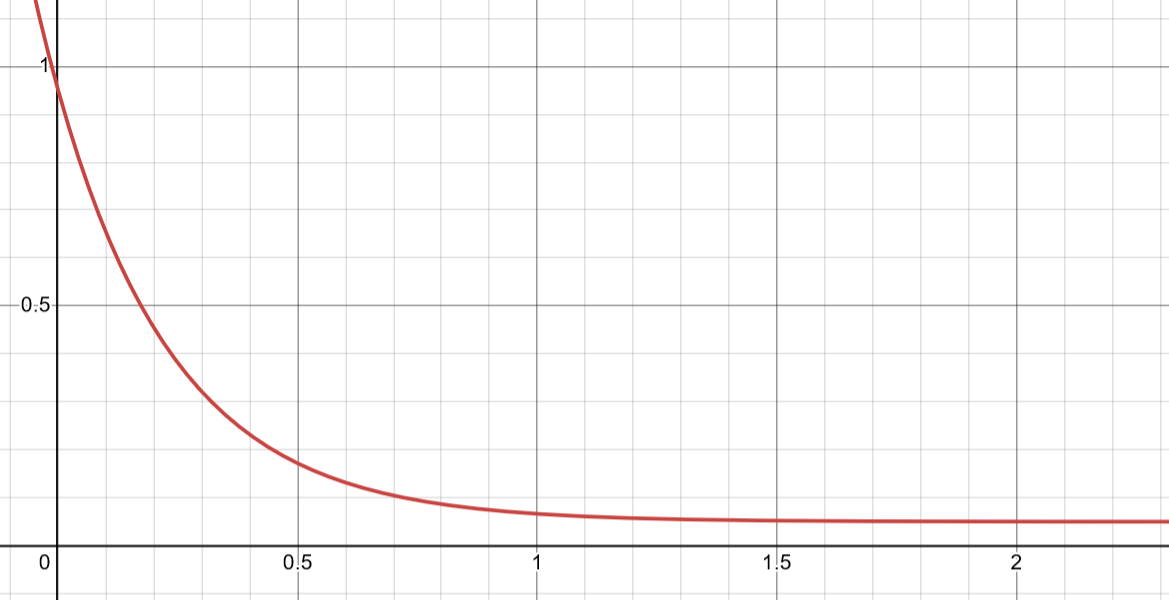


Рисунок 3. Функция эпсилон, где ось x – число выборов действий в десятках тысяч

* + TAU = 0.005 - скорость обновления целевой сети. Параметр τ управляет тем, как быстро целевая сеть обновляется, чтобы соответствовать основной сети. Малое значение τ означает, что целевая сеть обновляется медленно, что приводит к более стабильному обучению.
  + LR (Learning Rate) = 0.0001 - скорость обучения оптимизатора, конкретно AdamW. Скорость обучения определяет размер шагов, которые делает оптимизатор в направлении градиента. Меньшая скорость обучения означает, что агент будет делать более точные обновления параметров сети, что может привести к более стабильному и точному обучению, но это может также замедлить сходимость.

Основной функцией выбора действия является функция select\_action, которая выбирает действие в соответствии с эпсилон-жадной политикой.

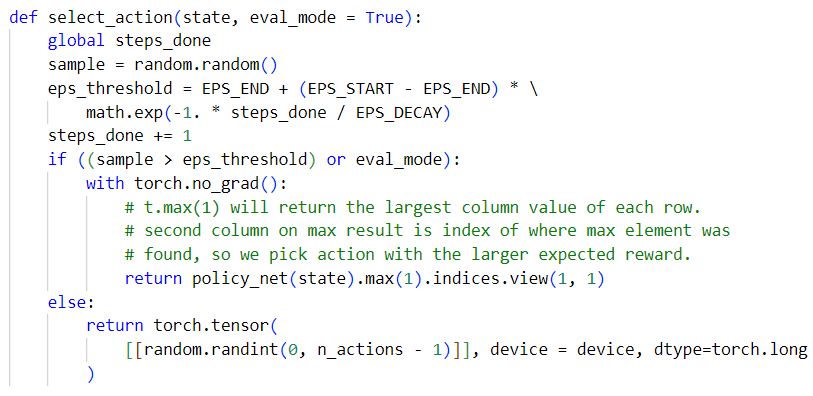


Рисунок 4. Реализация функции select\_action

Основное обучение нейронной сети происходит в функции optimize\_model, вызываемая каждый шаг обучения после совершения действия и получения результатов выполненного действия. Сначала она делает выборку, объединяет все тензоры в один, вычисляет Q(st,at) и V(st+1)=maxaQ(st+1,a) и объединяет их в нашу потерю. По определению, V(s)=0, если s - терминальное состояние. Для повышения устойчивости также используется целевая сеть для вычисления V(st+1). Целевая сеть обновляется на каждом шаге с помощью мягкого обновления, управляемого гиперпараметром TAU.

Обучение начинается с момента, когда в памяти переходов набирается больше элементов, чем прописано в гиперпараметре размера пакета переходов BATCH\_SIZE. Если это условие выполнено, то из памяти берутся случайные переходы и трансформируется в удобный формат.

После этого из выборки отдельно вычисляются состояния, которые являются конечными. В обучении с подкреплением конечное состояние — это состояние, в котором эпизод заканчивается. В нашем случае, конец эпизода — это финиш трассы или проигрыш, определяемый концом отведенного для прохождения временем или сильно отрицательной аккумулированной награды. В конечном состоянии нет следующего состояния, на которое можно было бы опираться для прогнозирования будущих наград. Поэтому в этих состояниях значение функции Q устанавливается равным 0.

После этого для всех состояний выборки с помощью основной модели создается массив значений Q – state\_action\_values. Он будет использован в функции потерь, как существующие значения функции.

Для неконечных состояний целевая сеть вычисляет значения V, которые потом используются для вычисления ожидаемых значений функции Q, как суммы награды и дисконтированного значения V для следующего состояния.

После этого функция потерь вычисляется с использованием потери Хьюбера. Затем оптимизатор обновляет параметры модели, используя вычисленный градиент. Градиенты ограничиваются по величине для предотвращения скачков значений.

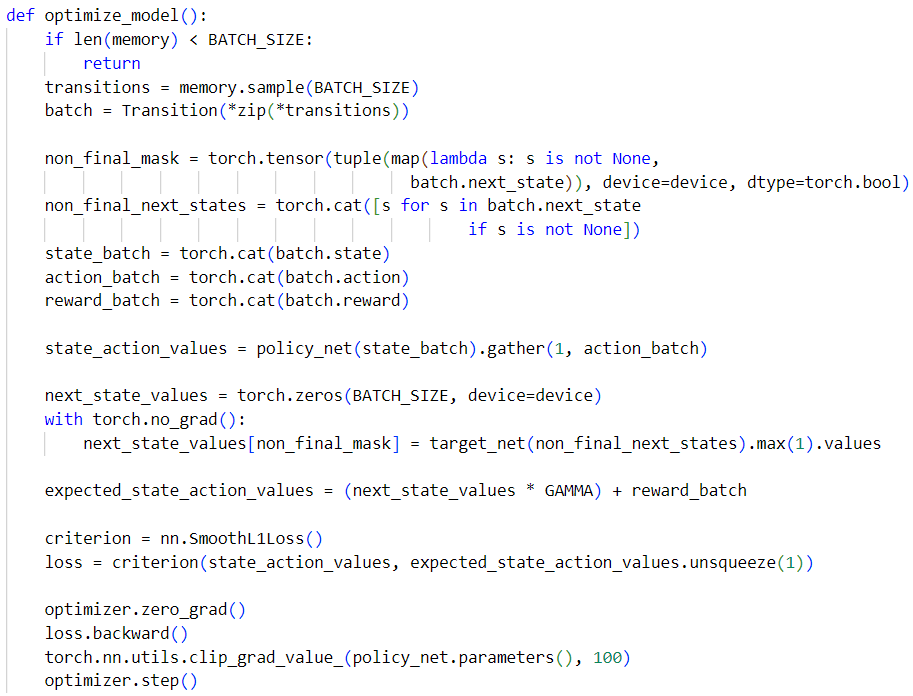


Рисунок 5. Реализация функции optimize\_model

Теперь можно определить основной цикл обучения. В начале окружение сбрасывается и получается тензор начального состояния. Затем выбирается и исполняется действие, наблюдается следующее состояние и вознаграждение, после чего модель оптимизируется. Когда эпизод заканчивается, мы цикл перезапускается.

Действия выбираются либо случайным образом, либо на с помощью модели, получая значения следующего шага из окружения. Мы записываем результаты в память воспроизведения, а также запускаем шаг оптимизации на каждой итерации. Оптимизация выбирает случайную выборку из памяти воспроизведения для обучения модели. «Старая» целевая сеть также используется в оптимизации для вычисления ожидаемых значений Q. Мягкое обновление ее весов выполняется на каждом шаге.

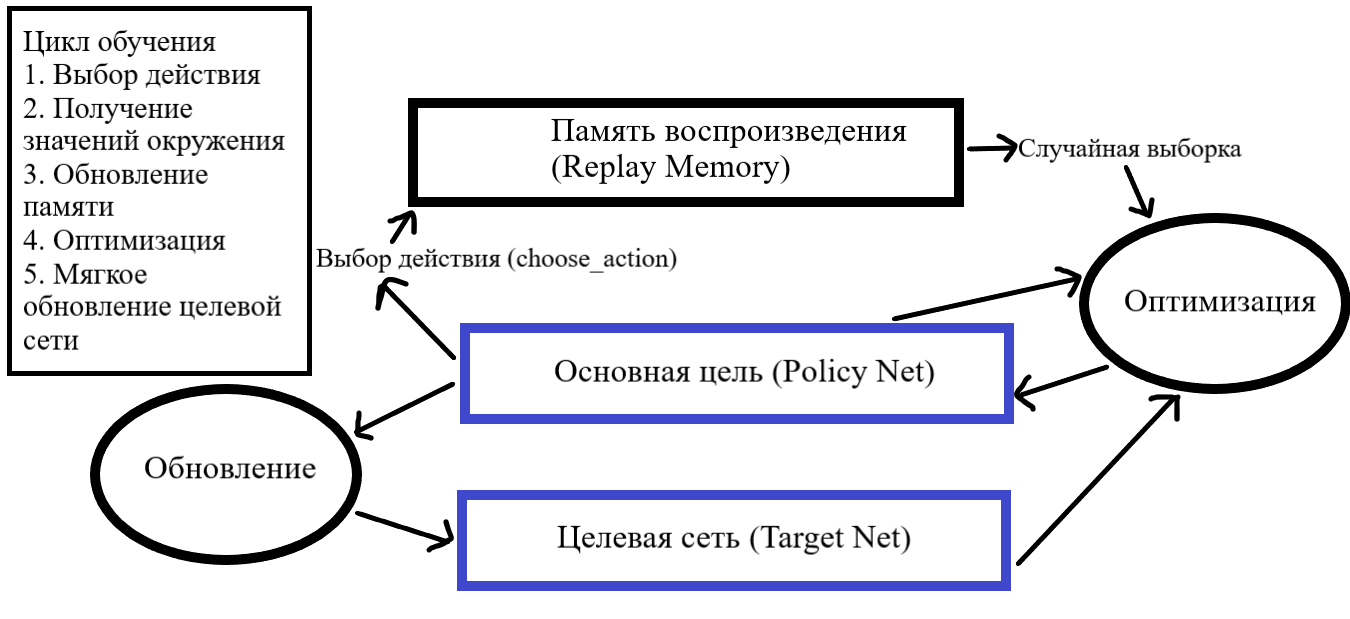


Рисунок 6. Упрощенная схема цикла обучения

Также были созданы средства для сохранения и загрузки моделей, чтобы в дальнейшем их использовать для последующего дообучения, переобучения или демонстрации.

Так, каждые 50 эпизодов сохраняются словари состояний обеих моделей и оптимизатора, вся память воспроизведения и количество выполненных шагов (для корректной работы эпсилон-жадной стратегии).

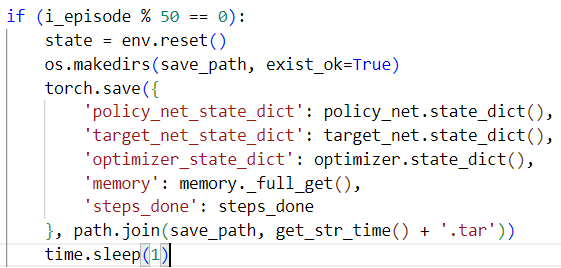


Рисунок 7. Реализация сохранения чекпоинта обучения

Для дообучения используются все данные, но для переобучения при необходимости можно указать свое число выполненных шагов, чтобы агент опять начал больше исследовать, чем эксплуатировать.

Для демонстрации достаточно лишь словаря состояний основной модели. Также при демонстрации убраны шаги оптимизации модели и использование полученной награды. Эпсилон-жадная стратегия тоже не имеет влияния, так как агент при демонстрации всегда эксплуатирует, а не исследует.

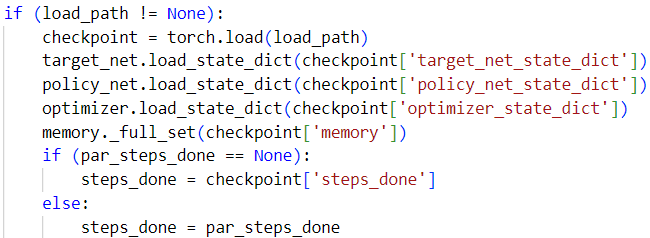


Рисунок 8. Реализация загрузки чекпоинта обучения

## 1.2. Заключение по приложению и циклу его работы

Цикл работы приложения в конечном итоге выглядит следующим образом:

* + Инициализируются основная и целевая модели, память воспроизведения и оптимизатор.
  + Запускается и инициализируется окружение, в котором будет происходить обучение.
  + Запускается и инициализируется интерфейс, связывающий клиент видеоигры и окружение.
  + Запускается и инициализируется клиент, связывающий интерфейс и видеоигру. Предполагается, что видеоигра запущена и выбран подходящий трек, на котором будет происходить обучение.
  + Начинается цикл обучения, в начале которого окружение сбрасывается и записывается тензор начального состояния.
  + Основываясь на состоянии, выбирается действие (либо случайное, либо выбранное моделью, что зависит от эпсилон), которое отправляется в окружение, после чего окружение отправляет данные о своем состоянии приложению.
  + Полученные от окружения данные о факте конца эпизода используются для конца эпизода.
  + Полученные от окружения данные и введенное управление добавляются в память воспроизведения, после чего используются для оптимизации модели целевой модели.
  + Обновленная целевая модель используется для мягкого обновления основной модели.
  + Полученные от окружения данные о состоянии видеоигры сохраняются для последующего выбора управления.

Получение состояния окружения происходит следующим образом:

* + Приложение делает запрос данных окружения путем сброса окружения или после ввода игрового управления.
  + При сбросе окружения оно обнуляет все свои значения, делает команду интерфейсу на перезапуск трека, после чего возвращает обнуленные данные приложению.
  + Иначе окружение принимает ввод и передает его интерфейсу.
  + Происходит ожидание в 0.05 секунд для изменения состояния видеоигры.
  + Окружение запрашивает данные о состоянии видеоигры у интерфейса.
  + Окружение вычисляет функцию награды на основе введенных данных.
  + Окружение проверяет, не должен ли эпизод обучения быть закончен (например, при финише трека, концу отведенного времени или отрицательной аккумулированной награде)
  + Окружение передает данные о состоянии видеоигры, награду и факт конца эпизода приложению.

Получение состояния видеоигры происходит следующим образом:

* + Окружение запрашивает данные о состоянии видеоигры у интерфейса.
  + Интерфейс запрашивает данные у клиента.
  + Клиент каждый физический тик игры сохраняет состояние видеоигры, благодаря чему может ответить на запрос интерфейса.
  + Интерфейс обрабатывает полученные данные, приводит их в нормализованное состояние и отвечает на запрос окружения.

Итоговая диаграмма классов без указания их полей и методов выглядит следующим образом.

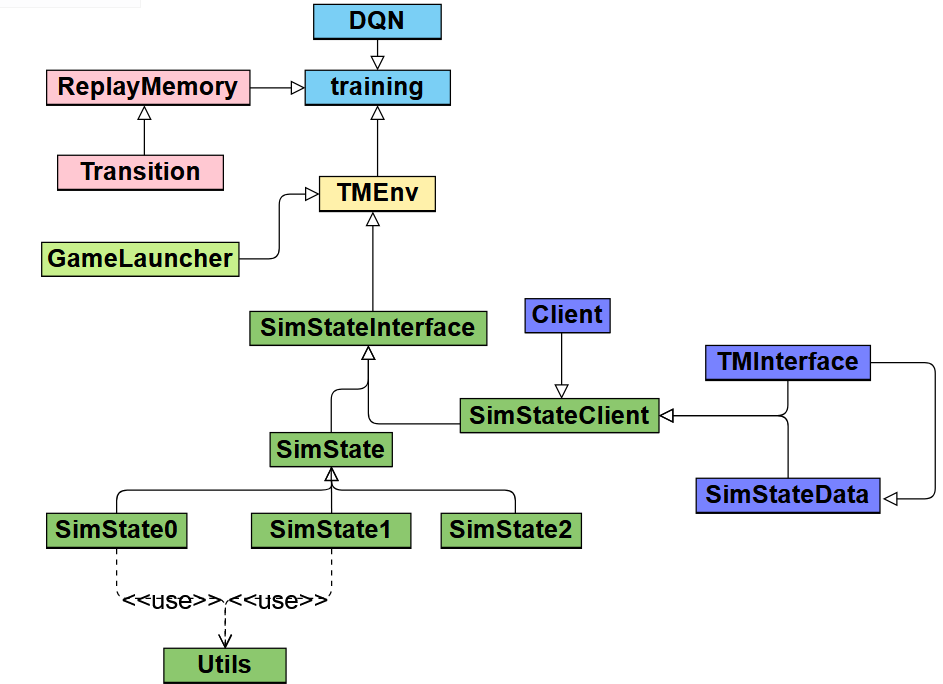


Рисунок 9. Диаграмма классов приложения

## 2. Тестирование приложения и моделей

Тестирование приложения проводилось вручную. На всех создаваемых картах с правильно вводимыми данными приложение работало корректно, начиная и успешно заканчивая обучение модели. Вспомогательные для разработки и использования средства также не выдавали ошибок при корректном вводе данных.

Тестирование моделей считалось пройденным при успешном прохождении моделью трассы. На всех создаваемых картах с правильно вводимыми данными обученные модели могли сориентироваться на трассе и успешно ее закончить. При проезде участков треков, похожих на те, что были при обучении, модель использовала оптимальные линии, что подтверждает успешное обучение.

# Заключение

За время прохождения производственной технологической (проектно-технологической) практики мной приобретены следующие компетенции:

| **Шифр компетенции** | **Расшифровка приобретаемой компетенции** |
| --- | --- |
| ОПК-4 | Способен участвовать в разработке стандартов, норм и правил, а также технической документации, связанной с профессиональной деятельностью |
| ОПК-7 | Способен разрабатывать алгоритмы и программы, пригодные для практического применения |
| ОПК-8 | Способен принимать участие в управлении проектами создания информационных систем на стадиях жизненного цикла |

# Список использованных источников

1. Exploration Strategies in Deep Reinforcement Learning [Электронный ресурс]. URL: https://lilianweng.github.io/posts/2020-06-07-exploration-drl/ (дата обращения: 10.04.2024).
2. Epsilon-Greedy Algorithm in Reinforcement Learning [Электронный ресурс]. URL: https://www.geeksforgeeks.org/epsilon-greedy-algorithm-in-reinforcement-learning/ (дата обращения: 10.04.2024)
3. Reinforcement Learning (DQN) Tutorial [Электронный ресурс]. URL: https://pytorch.org/tutorials/intermediate/reinforcement\_q\_learning.html (дата обращения: 11.04.2024)
4. TMInterface [Электронный ресурс]. URL: https://donadigo.com/tminterface/ (дата обращения: 14.04.2024)
5. Save and Load the Model [Электронный ресурс]. URL: https://pytorch.org/tutorials/beginner/basics/saveloadrun\_tutorial.html (дата обращения: 17.04.2024)
6. Saving and loading a general checkpoint in PyTorch   
   [Электронный ресурс]. URL: https://pytorch.org/tutorials/recipes/recipes/saving\_and\_loading\_a\_general\_checkpoint.html (дата обращения: 17.04.2024)
7. Tips for Loading an nn.Module from a Checkpoint   
   [Электронный ресурс]. URL: https://pytorch.org/tutorials/recipes/recipes/module\_load\_state\_dict\_tips.html?highlight=loading%20nn%20module%20from%20checkpoint (дата обращения: 17.04.2024)
8. Saving and Loading Models [Электронный ресурс]. URL: https://pytorch.org/tutorials/beginner/saving\_loading\_models.html (дата обращения: 17.04.2024)
9. How Forza's Drivatar Actually Works | AI and Games #60 [Электронный ресурс]. URL: https://www.youtube.com/watch?v=JeYP9eyIl4E&t=1s (дата обращения: 30.04.2024)
10. Drivatar in Forza Motorsport [Электронный ресурс]. URL: http://web.archive.org/web/20140208083127/http://research.microsoft.com/en-us/projects/drivatar/forza.aspx (дата обращения: 30.04.2024)
11. TMRL [Электронный ресурс]. URL: https://github.com/trackmania-rl/tmrl (дата обращения: 10.05.2024)
12. Функция потерь Хьюбера [Электронный ресурс]. URL: https://ru.wikipedia.org/wiki/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F\_%D0%BF%D0%BE%D1%82%D0%B5%D1%80%D1%8C\_%D0%A5%D1%8C%D1%8E%D0%B1%D0%B5%D1%80%D0%B0 (дата обращения: 15.05.2024)

# ПРИЛОЖЕНИЕ

training.py

import datetime

import math

from os import path

import os

import random

from itertools import count

import time

import torch

import torch.nn as nn

import torch.optim as optim

from dqn import DQN

from replay\_memory import ReplayMemory, Transition

from hyperparameters import \*

from TMEnv import TMEnv

def get\_str\_time():

    return str(datetime.datetime.now()).replace('.', '').replace('-', '').replace(':', '').replace(' ', '')

base\_path = path.join(path.curdir, 'model\_dicts')

episode\_durations = []

env = TMEnv()

# if GPU is to be used

device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")

# Training utils

# Get number of actions from

n\_actions = 16

# Get the number of state observations

state = env.reset()

n\_observations = len(state)

policy\_net = DQN(n\_observations, n\_actions).to(device)

target\_net = DQN(n\_observations, n\_actions).to(device)

target\_net.load\_state\_dict(policy\_net.state\_dict())

optimizer = optim.AdamW(policy\_net.parameters(), lr=LR, amsgrad=True)

memory = ReplayMemory(10000)

steps\_done = 0

def select\_action(state, eval\_mode = True):

    global steps\_done

    sample = random.random()

    eps\_threshold = EPS\_END + (EPS\_START - EPS\_END) \* \

        math.exp(-1. \* steps\_done / EPS\_DECAY)

    steps\_done += 1

    if ((sample > eps\_threshold) or eval\_mode):

        with torch.no\_grad():

            # t.max(1) will return the largest column value of each row.

            # second column on max result is index of where max element was

            # found, so we pick action with the larger expected reward.

            return policy\_net(state).max(1).indices.view(1, 1)

    else:

        return torch.tensor(

            [[random.randint(0, n\_actions - 1)]], device = device, dtype=torch.long

        )

# Training Loop

def optimize\_model():

    if len(memory) < BATCH\_SIZE:

        return

    transitions = memory.sample(BATCH\_SIZE)

    # Transpose the batch (see https://stackoverflow.com/a/19343/3343043 for

    # detailed explanation). This converts batch-array of Transitions

    # to Transition of batch-arrays.

    batch = Transition(\*zip(\*transitions))

    # Compute a mask of non-final states and concatenate the batch elements

    # (a final state would've been the one after which simulation ended)

    non\_final\_mask = torch.tensor(tuple(map(lambda s: s is not None,

                                          batch.next\_state)), device=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)

    # Compute Q(s\_t, a) - the model computes Q(s\_t), then we select the

    # columns of actions taken. These are the actions which would've been taken

    # for each batch state according to policy\_net

    state\_action\_values = policy\_net(state\_batch).gather(1, action\_batch)

    # Compute V(s\_{t+1}) for all next states.

    # Expected values of actions for non\_final\_next\_states are computed based

    # on the "older" target\_net; selecting their best reward with max(1).values

    # This is merged based on the mask, such that we'll have either the expected

    # state value or 0 in case the state was final.

    next\_state\_values = torch.zeros(BATCH\_SIZE, device=device)

    with torch.no\_grad():

        next\_state\_values[non\_final\_mask] = target\_net(non\_final\_next\_states).max(1).values

    # Compute the expected Q values

    expected\_state\_action\_values = (next\_state\_values \* GAMMA) + reward\_batch

    # Compute Huber loss

    criterion = nn.SmoothL1Loss()

    loss = criterion(state\_action\_values, expected\_state\_action\_values.unsqueeze(1))

    # Optimize the model

    optimizer.zero\_grad()

    loss.backward()

    # In-place gradient clipping

    torch.nn.utils.clip\_grad\_value\_(policy\_net.parameters(), 100)

    optimizer.step()

def train(save\_path, load\_path=None, par\_steps\_done=None):

    global target\_net, policy\_net, optimizer, steps\_done

    steps\_done = 0

    if (load\_path != None):

        checkpoint = torch.load(load\_path)

        target\_net.load\_state\_dict(checkpoint['target\_net\_state\_dict'])

        policy\_net.load\_state\_dict(checkpoint['policy\_net\_state\_dict'])

        optimizer.load\_state\_dict(checkpoint['optimizer\_state\_dict'])

        memory.\_full\_set(checkpoint['memory'])

        if (par\_steps\_done == None):

            steps\_done = checkpoint['steps\_done']

        else:

            steps\_done = par\_steps\_done

    target\_net.train()

    policy\_net.train()

    if torch.cuda.is\_available():

        num\_episodes = 600

    else:

        num\_episodes = 50

    for i\_episode in range(num\_episodes):

        # print(f'i\_episode: {i\_episode}')

        # Initialize the environment and get its state

        if (i\_episode % 5 == 0):

            state = env.reset()

            os.makedirs(save\_path, exist\_ok=True)

            torch.save({

                'policy\_net\_state\_dict': policy\_net.state\_dict(),

                'target\_net\_state\_dict': target\_net.state\_dict(),

                'optimizer\_state\_dict': optimizer.state\_dict(),

                'memory': memory.\_full\_get(),

                'steps\_done': steps\_done

            }, path.join(save\_path, get\_str\_time() + '.tar'))

            time.sleep(2)

        state = env.reset()

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

        for t in count():

            action = select\_action(state, eval\_mode=False)

            observation, reward, terminated, info = env.step(action)

            reward = torch.tensor([reward], device=device)

            done = terminated

            if terminated:

                next\_state = None

            else:

                next\_state = torch.tensor(observation, dtype=torch.float32, device=device).unsqueeze(0)

            # Store the transition in memory

            memory.push(state, action, next\_state, reward)

            # Move to the next state

            state = next\_state

            # Perform one step of the optimization (on the policy network)

            optimize\_model()

            # Soft update of the target network's weights

            # θ′ ← τ θ + (1 −τ )θ′

            target\_net\_state\_dict = target\_net.state\_dict()

            policy\_net\_state\_dict = policy\_net.state\_dict()

            for key in policy\_net\_state\_dict:

                target\_net\_state\_dict[key] = policy\_net\_state\_dict[key]\*TAU + target\_net\_state\_dict[key]\*(1-TAU)

            target\_net.load\_state\_dict(target\_net\_state\_dict)

            if done:

                episode\_durations.append(t + 1)

                break

def eval(load\_path):

    global target\_net, policy\_net, optimizer

    checkpoint = torch.load(load\_path)

    target\_net.load\_state\_dict(checkpoint['target\_net\_state\_dict'])

    policy\_net.load\_state\_dict(checkpoint['policy\_net\_state\_dict'])

    optimizer.load\_state\_dict(checkpoint['optimizer\_state\_dict'])

    target\_net.eval()

    policy\_net.eval()

    for i\_episode in range(100):

        # print(f'i\_episode: {i\_episode}')

        # Initialize the environment and get its state

        state = env.reset()

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

        for t in count():

            action = select\_action(state)

            observation, reward, terminated, info = env.step(action)

            reward = torch.tensor([reward], device=device)

            done = terminated

            if terminated:

                next\_state = None

            else:

                next\_state = torch.tensor(observation, dtype=torch.float32, device=device).unsqueeze(0)

            # Move to the next state

            state = next\_state

            if done:

                episode\_durations.append(t + 1)

                break

if \_\_name\_\_ == '\_\_main\_\_':

    save\_path = path.join(base\_path, get\_str\_time())

    load\_path = path.join(base\_path, '20240510124210553633.tar')

    # train(save\_path)

    train(save\_path, load\_path=load\_path)

    # eval(load\_path=load\_path)