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](#_Toc167133708)

[Введение 4](#_Toc167133709)

[1. Проектирование приложения 5](#_Toc167133710)

[1.1. Разработка связи между видеоигрой и приложением 5](#_Toc167133711)

[1.2. Создание gym окружения видеоигры для обучения 9](#_Toc167133712)

[1.3. Реализация нейронной сети 10](#_Toc167133713)

[2. Тестирование приложения и моделей 16](#_Toc167133714)

[Заключение 17](#_Toc167133715)

[Список использованных источников 18](#_Toc167133716)

[ПРИЛОЖЕНИЕ 20](#_Toc167133717)

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. Разработка связи между видеоигрой и приложением

Использовалась пользовательская модификация видеоигры TrackMania Nations Forever под названием TMInterface, которая позволяет наладить связь с собой с помощью одноименной библиотеки для Python.

Основными классами библиотеки, которые использовались в приложении, являются:

* + Client – класс, напрямую связанный с клиентом игры. Предназначен в первую очередь для наследования от себя и переопределения методов обратного вызова.
  + TMInterface – основной класс, который позволяет получать состояние игры или менять его, а также взаимодействовать с самими процессами видеоигры.
  + SimStateData – класс, который хранит в себе состояние видеоигры.
  + HmsDynaStruct – класс, который хранит в себе состояние об автомобиле в конкретный момент времени.
  + CheckpointData – класс, который хранит в себе состояние о пройденных чекпоинтах.

Для наследования от класса Client был создан класс SimStateClient. Он хранит в себе поля, отвечающие за хранение ссылки на объекты классов TMInterface, SimStateData и CheckpointData, данные, которые должны быть введены в игру, факт завершения гонки и флаг, при поднятии которого клиент должен перезапустить гонку.

Было переопределено 2 метода класса Client:

* + on\_run\_step – вызывается каждый физический тик в гонке, имеет в параметрах TMInterface, который хранит все текущие состояния. Метод используется для сохранения в клиенте ссылок на TMInterface, SimStateData и CheckpointData. Также при наличии ввода передает его в игру, а при необходимости перезапуска гонки – перезапускает её.
  + on\_checkpoint\_count\_changed – вызывается при прохождении чекпоинта в гонке, в том числе финишной черты. Используется для того, чтобы гонка не завершалась при проезде финиша, при этом меняя поле класса, отвечающее за факт конца гонки.

Также был создан класс SimStateInterface, обеспечивающий связь с клиентом. При инициализации он создает по экземпляру классов SimStateClient и TMInterface, регистрирует интерфейс в клиенте, а также хранит обработанные данные о состоянии машины. С помощью своих методов он также позволяет обращаться к клиенту при необходимости ввода каких-либо данных или для их получения.

Для хранения обработанных данных о машине был создан класс SimState. Он имеет в себе поля, которые уже передаются в нейронную сеть. Основные хранимые данные:

* + linear\_speed – линейная скорость автомобиля, то есть скорость по основной оси машины.
  + angular\_speed – боковая скорость автомобиля, то есть скорость по оси машины, перпендикулярной основной.
  + distance\_to\_centerline – расстояние до центра дороги.
  + angle\_to\_centerline – угол основной оси машины относительно дороги, где 0 означает, что машина смотрит в сторону, в которую нужно двигаться, а 1 по модулю – что машина смотрим в обратную сторону.
  + next\_curve\_distance – расстояние до ближайшего поворота.
  + next\_curve\_direction – направление следующего поворота (-1 – влево, 1 – вправо)
  + next\_curve\_distance2, next\_curve\_distance3 – длина следующих прямых отрезков дороги.
  + next\_curve\_direction2, next\_curve\_direction3 – направление следующих поворотов.

Помимо сложных методов подсчета этих данных класс также нормализует их до отрезка от -1 до 1.

Во время разработки было создано несколько сущностей подобных классов, отвечающих за разные типы трасс. Самый простой для трека, состоящего из одного прямого участка прямо до финиша, принимал только первые 4 указанных поля, так как не было необходимости знать, сколько осталось до поворотов и их направление по причине их отсутствия.

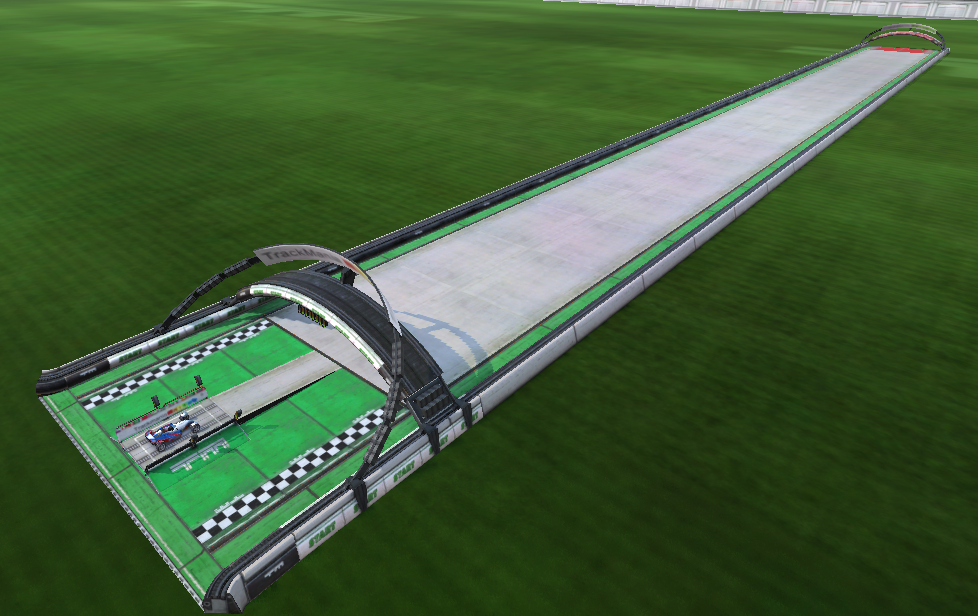


Рисунок 1. Прямая трасса

Класс, позволяющий получать данные для карты вида «змейка», имел в себе первые 6 указанных параметров, то есть теперь имел в себе значения, позволяющие понять, когда нужно поворачивать и в какую сторону.



Рисунок 2. Карта вида «змейка»

Самый большой класс для карт с различными перпендикулярными поворотами, имеет все 10 указанных полей, чтобы нейросеть понимала, стоит ли сразу разгоняться после поворота или нужно притормозить, так как скоро придется поворачивать ещё.



Рисунок 3. Карта с перпендикулярными поворотами

Для удобства разработки, средствами TMInterface были включены отображение текущего ввода и основные данные об автомобиле



Рисунок 4. Интерфейс запущенного TMInterface

## 1.2. Создание gym окружения видеоигры для обучения

Для обучения нейронной сети было решено создать окружение с помощью библиотеки gym. gym – это библиотека для разработки и тестирования алгоритмов обучения с подкреплением. Она предоставляет различные окружения и возможности создания окружений, в которых агенты могут обучаться и взаимодействовать, что позволяет исследователям и разработчикам создавать, тестировать и сравнивать различные алгоритмы обучения с подкреплением.

Библиотека предоставляет класс Env, предполагаемый для наследования и переопределения методов обратного вызова. В связи с этим был создан класс TMEnv. Основные методы окружений gym:

* + step – метод, который вызывается каждый шаг алгоритма обучения и обычно вызывает остальные методы обратного вызова.
  + render – метод, обычно используемый для отображения каких-либо данных во время обучения.
  + reset – метод, предназначенный для приведения состояния окружения в свое обычное состояние.
  + reward – метод, возвращающий подкрепление, или награду, которое используется в алгоритме обучения.

Этот класс инициализирует и содержит в себе экземпляр класса SimStateInterface, что позволяет получать актуальные данные из состояния видеоигры и вводить действия для машины.

Самой сложной для создания являлась функция награды, так как от нее зависит то, как именно нейронная сеть будет менять веса. Было решено давать подкрепление за высокую скорость, высокое ускорение, прохождение трассы в правильном направлении, прохождение поворотов и трека в целом. Отрицательная награда даётся за отрицательную скорость, за непрохождение трассы или прохождение трассы и поворотов в неправильном направлении, за падение из трека и за факт слишком сильного поворота машины в неправильном направлении.

Другой важной функцией является функция step, вызываемая каждый шаг обучения. Она принимает ввод для видеоигры, получаемый от модели, обрабатывает его, вводит в игру, сохраняет полученные данные, проверяет, не нужно ли перезапустить обучение и перезапускает при необходимости, а также возвращает все необходимые для нейронной сети данные:

* + observation – обработанные и нормализованные данные, полученные из клиента (SimStateClient), интерфейса (SimStateInterface) и класса обработки состояния (SimState) видеоигры.
  + reward – награда, полученная алгоритмом обучения за текущий шаг.
  + done – булевая переменная, означающая, закончилась ли текущая сессия обучения и было ли перезапущено окружение.

Помимо этого, TMEnv также хранит множество вспомогательных параметров, большинство из которых позволяют функции награды более корректно определять подкрепление, как, например, previous\_distance\_to\_curve, отвечающая за расстояние до поворота в предыдущий шаг обучения и с которой сравнивается текущее расстояние до поворота.

Этот класс также обеспечивает факт запущенности видеоигры, устанавливает ее скорость, хранит в себе значения, используемые нейронной сетью. Основная из таких - observation\_space, определяющая, сколько входных и параметров будет принимать модель нейронной сети. Она имеет определение в виде

OBSERVATION\_SPACE = Box(-1, 1, (10,)).

Это означает, что нейросеть принимает 10 параметров, все из которых принимают значения в диапазоне от -1 до 1.

## 1.3. 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.

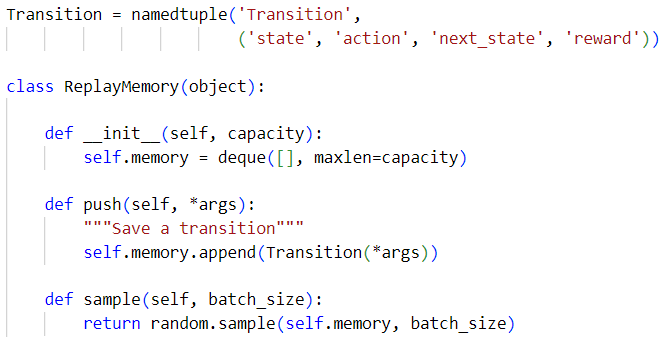


Рисунок 5. Реализация классов памяти воспроизведения опыта

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.

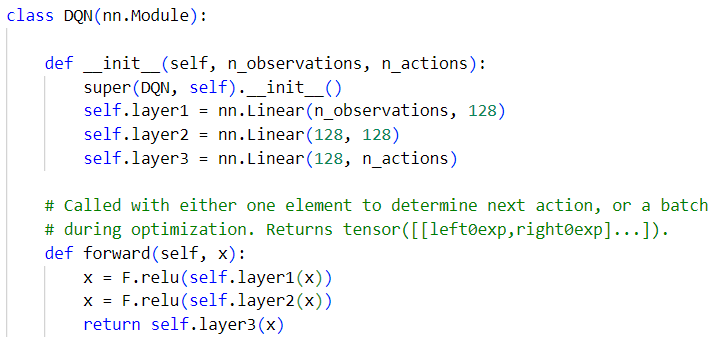


Рисунок 6. Реализация класса нейронной сети

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.
  + 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.

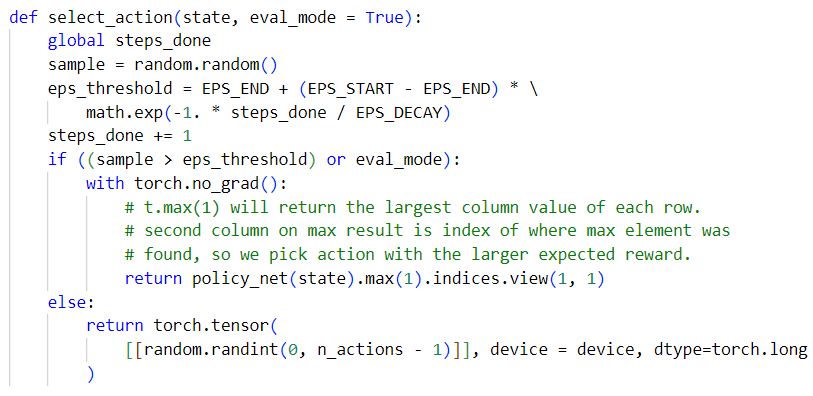


Рисунок 7. Реализация функции select\_action

## 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)

TMEnv.py

from gym import Env

from gym.spaces import MultiBinary, Box

import numpy as np

import time

from GameLaunch import GameLauncher

from TMInterface import SimStateInterface

from constants import LEVEL

# gas, break, steer

ACTION\_SPACE = MultiBinary((4,))

OBSERVATION\_SPACE = Box(-1, 1, (6,)) if LEVEL != 2 else Box(-1, 1, (10,))

GAME\_SPEED = 1 # default = 1

COMMAND\_SPEED = 0.05

class TMEnv(Env):

    def \_\_init\_\_(self):

        self.actual\_max\_distance = 1

        self.actual\_max\_reward = 0

        self.action\_space = ACTION\_SPACE

        self.observation\_space = OBSERVATION\_SPACE

        game\_launcher = GameLauncher()

        if not game\_launcher.game\_started:

            game\_launcher.start\_game()

            print("game started")

            input("press enter when game is ready")

            time.sleep(4)

        self.interface = SimStateInterface()

        time.sleep(0.15)

        self.interface.interface.set\_speed(GAME\_SPEED)

        self.saved\_state = np.array(self.interface.get\_state())

        # self.simthread = ThreadedClient()

        self.total\_reward = 0.0

        self.n\_steps = 0

        self.max\_steps = (100 / GAME\_SPEED) if LEVEL == 2 else (65 / GAME\_SPEED)

        self.command\_frequency = 0.05 / GAME\_SPEED

        # PrevStepValues

        self.added\_steps = 0

        self.last\_action = None

        self.previous\_distance\_to\_curve = 0

        self.previous\_speed = 0

        self.previous\_curve\_direction = 0

        self.saved\_cur = 0

    def action\_to\_command(self, action):

        gas, braking, left, right = action

        if (left == right):

            left = right = 0

        actions = {

            'accelerate': gas,

            'brake': braking,

            'left': left,

            'right': right

        }

        # print(f'actions: {actions}')

        self.interface.set\_actions(\*\*actions)

    def \_restart\_race(self):

        self.interface.reset()

    def step(self, input\_action):

        action\_correspondance = {

            i + 2 \* j + 4 \* k + 8 \* l: [i, j, k, l]

            for i in range(2)

            for j in range(2)

            for k in range(2)

            for l in range(2)

        }

        action = action\_correspondance[input\_action.item()]

        self.last\_action = action

        self.saved\_state = np.array(self.interface.get\_state())

        # print(f'action: {action}')

        self.last\_action = action

        self.action\_to\_command(action)

        done = False

        if (self.n\_steps >= self.max\_steps + self.added\_steps or self.total\_reward < -10 or self.total\_reward > 1500):

            done = True

            if (self.total\_reward > 1000):

                print("yay!")

            elif (self.total\_reward < -10):

                print("meh :(")

        cur\_reward = self.reward

        self.total\_reward += cur\_reward

        self.n\_steps += 1

        info = {}

        time.sleep(self.command\_frequency)

        return self.observation, cur\_reward, done, info

    def \_manual\_step(self):

        self.saved\_state = np.array(self.interface.get\_state())

        done = False

        if (self.n\_steps >= self.max\_steps + self.added\_steps or self.total\_reward < -10 or self.total\_reward > 1500):

            done = True

            if (self.total\_reward > 1000):

                print("yay!")

            elif (self.total\_reward < -10):

                print("meh :(")

        cur\_reward = self.reward

        self.total\_reward += cur\_reward

        self.n\_steps += 1

        info = {}

        time.sleep(self.command\_frequency)

        return self.observation, cur\_reward, done, info

    def render(self):

        pass

        # print(f"total reward: {self.total\_reward}")

    def reset(self):

        # print("reset")

        # print(f"total\_reward: {round(self.total\_reward, 3)}")

        # print(f"previous\_distance\_to\_curve: {round(self.previous\_distance\_to\_curve, 3)}")

        print(f"{time.strftime('%H:%M:%S', time.localtime())} | total\_reward: {self.total\_reward}")

        print()

        if (self.total\_reward > self.actual\_max\_reward):

            self.actual\_max\_reward = self.total\_reward

            print(f"{time.strftime('%H:%M:%S', time.localtime())} | actual\_max\_reward: {self.actual\_max\_reward}")

        if (self.previous\_distance\_to\_curve < self.actual\_max\_distance):

            self.actual\_max\_distance = self.previous\_distance\_to\_curve

            print(f"{time.strftime('%H:%M:%S', time.localtime())} | actual\_max\_distance: {self.actual\_max\_distance}")

        self.added\_steps = 0

        self.total\_reward = 0.0

        self.n\_steps = 0

        self.time = 0

        self.last\_action = None

        self.previous\_distance\_to\_curve = 0

        self.previous\_speed = 0

        self.previous\_curve\_direction = 0

        self.\_restart\_race()

        # print("reset done\n")

        # time.sleep(1.5 / GAME\_SPEED)

        return self.observation

    @property

    def state(self):

        return self.interface.client.sim\_state

    @property

    def observation(self):

        return self.saved\_state

    @property

    def reward(self):

        cur\_state = self.observation

        reward = 0

        speed = cur\_state[0]

        # print(f"speed: {speed}")

        # print(f"state: {cur\_state}")

        if (speed > 0.05):

            reward += speed

        elif (speed > 0.03):

            reward += speed / 1.1

        elif (speed > 0.02):

            reward += speed / 2.2

        elif (speed > 0.01):

            reward += speed / 3.3

        elif (speed > -0.01):

            reward += (-abs(speed)) / 3.3

        else:

            reward += speed

        if (self.previous\_distance\_to\_curve != 0):

            if ((LEVEL != 2 and self.previous\_curve\_direction == cur\_state[5])

                or (LEVEL == 2 and self.saved\_cur == self.interface.state.cur)):

                if (self.previous\_distance\_to\_curve - cur\_state[4] > 0.005):

                    reward += (self.previous\_distance\_to\_curve - cur\_state[4]) \* 500

                else:

                    reward += (self.previous\_distance\_to\_curve - cur\_state[4]) \* 1000 - 0.005

            else:

                if (LEVEL != 2):

                    self.added\_steps += 65 / GAME\_SPEED

                    reward += 1

                else:

                    if self.saved\_cur > self.interface.state.cur:

                        reward -= 25

                    else:

                        self.added\_steps += 65 / GAME\_SPEED

                        reward += 7

        if (LEVEL == 2):

            self.saved\_cur = self.interface.state.cur

        self.previous\_speed = speed

        self.previous\_distance\_to\_curve = cur\_state[4]

        self.previous\_curve\_direction = cur\_state[5]

        bad\_angle\_c = 0.75 - abs(cur\_state[3])

        if (bad\_angle\_c <= 0):

            reward += bad\_angle\_c \* 50

        if (self.interface.client.finished and self.n\_steps > 5):

            reward += 1000

        if (LEVEL == 0):

            if (cur\_state[4] > 0.855):

                reward -= 20

        elif (LEVEL == 1 or LEVEL == 2):

            if (self.state.dyna.current\_state.position[1] < (40 if LEVEL == 1 else 160)):

                reward -= 20

        # print(f'reward: {reward}')

        return reward

TMInterface.py

from tminterface.client import Client

from tminterface.interface import TMInterface, SimStateData

from tminterface.structs import CheckpointData

from time import sleep

from SimState2 import SimState2

from SimStates import SimState0, SimState1

from constants import LEVEL

"""

speed

acceleration

turning\_rate

lateral\_velocity

distance\_to\_centerline

angle\_to\_centerline

next\_curve\_distance

next\_curve\_direction

"""

"""

        0 - straight

        1 - curvy

        2 - ???

"""

class SimStateClient(Client):

    """

    Client for a TMInterface instance.

    Its only job is to get the simulation state that is used by the gym env for reward computation.

    """

    def \_\_init\_\_(self):

        super().\_\_init\_\_()

        self.iface: TMInterface = None

        self.sim\_state: SimStateData = None

        self.cp\_data: CheckpointData = None

        self.actions = None

        self.finished = True

        self.should\_reset = False

    def on\_run\_step(self, iface: TMInterface, \_time: int):

        self.iface = iface

        self.sim\_state = iface.get\_simulation\_state()

        self.cp\_data = iface.get\_checkpoint\_state()

        if self.should\_reset:

            self.should\_reset = False

            self.finished = False

            self.iface.respawn()

        if self.actions != None:

            self.iface.set\_input\_state(\*\*self.actions)

    def on\_checkpoint\_count\_changed(self, iface, current: int, target: int):

        self.iface.prevent\_simulation\_finish()

        if (current == target):

            self.finished = True

        return super().on\_checkpoint\_count\_changed(iface, current, target)

class SimStateInterface():

    def \_\_init\_\_(self):

        self.client = SimStateClient()

        self.interface = TMInterface()

        self.interface.register(self.client)

        if (LEVEL == 0):

            self.state = SimState0()

        elif (LEVEL == 1):

            self.state = SimState1()

        elif (LEVEL == 2):

            self.state = SimState2()

        print('SimStateInterface Initiated')

    def get\_state(self):

        self.step()

        return self.state.get\_state()

    def step(self):

        pos = self.client.sim\_state.dyna.current\_state.position

        rot = self.client.sim\_state.dyna.current\_state.rotation

        linear\_speed = self.client.sim\_state.dyna.current\_state.linear\_speed

        actual\_linear\_speed = linear\_speed[0] \* rot[0, 2] + linear\_speed[2] \* rot[0, 0]

        actual\_angular\_speed = - linear\_speed[0] \* rot[0, 0] + linear\_speed[2] \* rot[0, 2]

        if (LEVEL == 2):

            self.state.get\_info(pos, rot, actual\_linear\_speed, actual\_angular\_speed)

        else:

            self.state.get\_info\_from\_center\_line(pos, rot)

            self.state.linear\_speed = actual\_linear\_speed

            self.state.angular\_speed = actual\_angular\_speed

        self.state.normalize()

    def reset(self):

        self.client.should\_reset = True

    def set\_actions(self, \*\*kwargs):

        self.client.actions = kwargs

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

    interface = SimStateInterface()

    sleep(0.5)

    # interface.reset()

    while True:

        sleep(1)

        interface.step()

        print(interface.state)

        # print(interface.client.sim\_state.dyna.current\_state.position)