PROCES IZRADE UMJETNE INTELIGENCIJE PODRŽANIM UČENJEM

Garmaz, Petar

Sveučilište Josipa Jurja Strossmayera u Osijeku

Ekonomski fakultet u Osijeku

Diplomski studij Poslovna ekonomija/Poslovna informatika

Petar Garmaz

PROCES IZRADE UMJETNE INTELIGENCIJE PODRŽANIM UČENJEM

Diplomski rad

Osijek, 2022

Sveučilište Josipa Jurja Strossmayera u Osijeku

Ekonomski fakultet u Osijeku

Diplomski studij Poslovna ekonomija/Poslovna informatika

Petar Garmaz

PROCES IZRADE UMJETNE INTELIGENCIJE PODRŽANIM UČENJEM

Diplomski rad

Kolegij: Sustavi poslovne inteligencije

JMBAG: 0010218592

e-mail: pgarmaz@efos.hr

Mentor: doc.dr.sc. Slobodan Jelić

Osijek, 2022

Josip Juraj Strossmayer University of Osijek

Faculty of Economics in Osijek

Graduate Study Business economics/Business informatics

Petar Garmaz

CREATION PROCESS OF AN ARTIFICIAL INTELLIGENCE USING REINFORCEMENT LEARNING

Graduate paper

Osijek, 2022

IZJAVA

O AKADEMSKOJ ČESTITOSTI, PRAVU PRIJENOSA INTELEKTUALNOG VLASNIŠTVA, SUGLASNOSTI ZA OBJAVU U INSTITUCIJSKIM REPOZITORIJIMA I ISTOVJETNOSTI DIGITALNE I TISKANE VERZIJE RADA

1. Kojom izjavljujem i svojim potpisom potvrđujem da je diplomski (navesti vrstu rada: završni / diplomski / specijalistički / doktorski) rad isključivo rezultat osobnoga rada koji se temelji na mojim istraživanjima i oslanja se na objavljenu literaturu. Potvrđujem poštivanje nepovredivosti autorstva te točno citiranje radova drugih autora i referiranje na njih.
2. Kojom izjavljujem da je Ekonomski fakultet u Osijeku, bez naknade u vremenski i teritorijalno neograničenom opsegu, nositelj svih prava intelektualnoga vlasništva u odnosu na navedeni rad pod licencom *Creative Commons Imenovanje – Nekomercijalno – Dijeli pod istim uvjetima 3.0 Hrvatska*. 
3. Kojom izjavljujem da sam suglasan/suglasna da se trajno pohrani i objavi moj rad u institucijskom digitalnom repozitoriju Ekonomskoga fakulteta u Osijeku, repozitoriju Sveučilišta Josipa Jurja Strossmayera u Osijeku te javno dostupnom repozitoriju Nacionalne i sveučilišne knjižnice u Zagrebu (u skladu s odredbama Zakona o znanstvenoj djelatnosti i visokom obrazovanju, NN br. 123/03, 198/03, 105/04, 174/04, 02/07, 46/07, 45/09, 63/11, 94/13, 139/13, 101/14, 60/15).
4. izjavljujem da sam autor/autorica predanog rada i da je sadržaj predane elektroničke datoteke u potpunosti istovjetan sa dovršenom tiskanom verzijom rada predanom u svrhu obrane istog.

Ime i prezime studenta/studentice: Petar Garmaz

JMBAG: 0010218592

OIB: 52496061863

e-mail za kontakt: petar.garmaz@gmail.com

Naziv studija: Diplomski studij Poslovna ekonomija/Poslovna informatika

Naslov rada: Proces izrade umjetne inteligencije podržanim učenjem

Mentor/mentorica diplomskog rada: doc.dr.sc. Slobodan Jelić

U Osijeku, 2022. godine

Potpis Petar Garmaz

Proces izrade umjetne inteligencije podržanim učenjem

SAŽETAK

U ovom radu je prikazan proces izrade umjetne inteligencije podržanim učenjem u Python programskom jeziku. Opisana su temeljna znanja i tehnologije korištene za izradu projekta ovog rada. Nadalje je prikazan način izrade projekta diplomskog rada te prikazani rezultati navedenog projekta.

Ključne riječi: umjetna inteligencija, podržano učenje, strojno učenje, DQN

Creation process of an artificial intelligence using reinforcement learning

ABSTRACT

This graduate paper shows the creation process of an artificial intelligence using reinforcement learning in Python programming language. It describes fundamental knowledge and technologies that were used in the creation of the paper's project. Furthermore, it presents way the paper's project was made and shows the results of said project.

Keywords: artificial intelligence, reinforcement learning, machine learning, DQN

Sadržaj

1. Uvod 1
2. Teorijska podloga i prethodna istraživanja 2
   2.1. Strojno učenje 2
   2.2. Podržano učenje 2
   2.3. Q-Učenje 5
   2.4. Duboko Q-Učenje 6
   2.5. PyTorch i Pygame 7
3. Metodologija rada 11
4. Opis istraživanja i rezultati istraživanja 12
   4.1. Opis projekta diplomskog rada 12
   4.2. Izrada procesa treniranja 13
      4.2.1. Glavna petlja treninga 15
      4.2.2. Treniranje kretanja i nišana 17
      4.2.3. Model 27
   4.3. Rezultat 30
5. Rasprava 33
   5.1. Prednosti podržanog učenja 33
   5.2. Nedostaci podržanog učenja 33
   5.3. Primjenjivost podržanog učenja 33
6. Zaključak 35
7. Literatura 36

Popis slika 37

1. Uvod

Umjetna inteligencija (engl. Artificial intelligence, tj. AI) je pojam koji se koristi kako bi se opisala mogućnost računala da obavlja zadatke koji bi inače zahtijevali ljudsku inteligenciju, poput učenja ili rješavanja problema. Umjetna inteligencija se najčešće odnosi na računala koja imaju mogućnost „razmišljanja“ sama za sebe te donošenje odluka temeljene na prikupljenim i analiziranim podacima. Zadnjih nekoliko desetljeća je došlo do rasta interesa za umjetnu inteligenciju te je došlo do izvanrednog napretka u polju strojnog učenja.

Strojno učenje je grana računalne znanosti koja omogućuje računalima učenje, analizu te odluka na temelju prikupljenih podataka. Računalo uči kroz proces treniranja, u kojem računalo prepoznaje određene obrasce u podacima, te ih koristi za predviđanje i donošenje odluka.

Svrha ovog rada je prikazati proces izrade umjetne inteligencije podržanim učenjem u Python programskom jeziku. Podržano učenje je tip strojnog učenja koji koristi sustav nagrade i kazne kako bi umjetna inteligencija znala je li posljednja aktivnost bila pozitivna ili negativna.

Zadatak ovog rada je bio izraditi model neuronske mreže te ga primijeniti na primjer jednostavne video igre.

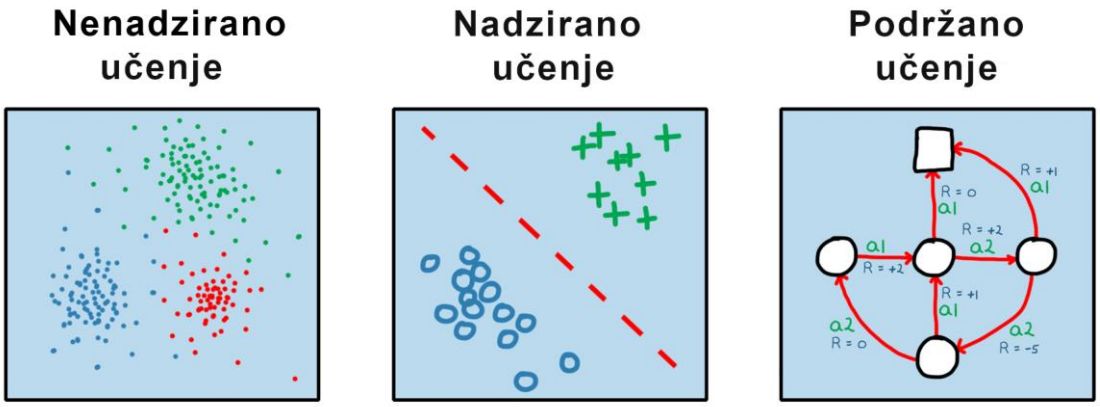
2. Teorijska podloga i prethodna istraživanja

2.1. Strojno učenje

Strojno učenje je postalo jedno od najvažnijih tema unutar razvojnih organizacija koje traže inovativne načine korištenja podataka kako bi se poslovanje podiglo na novu razinu razumijevanja (J. Hurwitz, D. Kirsch, 2018).

Postoje 3 pristupa strojnom učenju:

1. Nadzirano učenje
2. Nenadzirano učenje
3. Podržano učenje



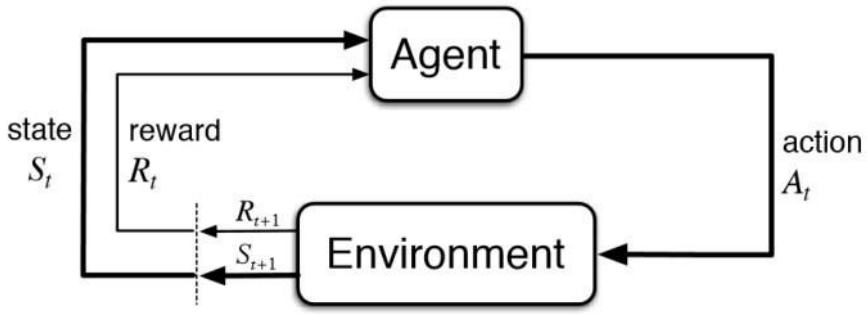
Slika 1: Vrste strojnog učenja
Izvor: https://se.mathworks.com/discovery/reinforcement-learning.html

2.2. Podržano učenje

„Podržano učenje, kao nadzirano i nenadzirano učenje, je jedan od glavnih područja strojnog učenja koje se bavi proučavanjem procesa učenja nekog entiteta, tj. agenta u svijetu u kojem postoji, tj. okolina. Agent pokušava maksimizirati nagradu koju prima od okoline te izvršava akcije kako bi postigao što više nagrada u jednoj epizodi“ (Zychlinski, 2019). „Cilj svakog algoritma podržanog učenja je utvrditi optimalnu politiku koja će imati maksimalnu nagradu“ (Karunakaran, 2020).

Podržano učenje sadrži sljedeće elemente:

1. Agent
2. Okolina
3. Stanje
4. Akcija
5. Politika
6. Epizoda
7. Nagrada ili kazna



Slika 2: Ciklus podržanog učenja
Izvor: https://www.kdnuggets.com/2018/03/5-things-reinforcement-learning.html

Agent je dio podržanog učenja koji pokušava učiti i donositi odluke tako da maksimizira nagrade koje dobije od okoline. U projektu ovog diplomskog rada, agenta predstavlja oklopno vozilo agenta koje se sastoji od 2 dijela (kupola i trup).

Okolina je dio podržanog učenja koji uključuje sve sa čime agent može imati interakciju, bilo to direktno ili indirektno. Okolina se mijenja čim agent odradi neku akciju. U projektu ovog diplomskog rada, okolinu predstavljaju neprijateljska oklopna vozila.

Stanje predstavlja scenarij u kojem agent susreće okolinu. Agent ima mogućnost prijelaza između dva stanja izvođenjem *akcija*. U projektu ovog diplomskog rada, stanje i akcija ovise o dijelu agenta; za kupolu agenta u stanje ulazi odnos pozicija između oklopnih vozila u okolini i agenta, dok za trup agenta u stanje ulazi trenutna pozicija agenta, trenutni vektor kretanja agenta, trenutna pozicija oklopnih vozila u okolini te udaljenost između agenta i oklopnih vozila

u okolini; akcija kupole agenta se odnosi na pomicanje nišana po x i y osi koordinatnog sustava, a akcija trupa agenta se odnosi na pomicanje trupa agenta po x i y osi koordinatnog sustava.

Politika definira način ponašanja agenta u određenom trenutku, tj. politika je preslikavanje percipiranih stanja okoline na akcije koje treba poduzeti u tim stanjima. U nekim slučajevima politika može biti jednostavna funkcija ili pregledna tablica, dok u drugim može uključivati opsežna izračunavanja. Postoje 2 vrste algoritama politike u podržanom učenju: „Na politici" (eng. On-policy) i „Izvan politike" (eng. Off-policy). Algoritmi podržanog učenja koji koriste politike koje su donesene na temelju prijašnjih stanja i akcija su *On-policy* algoritmi, a algoritmi koji ignoriraju prijašnja stanja i akcije su *Off-policy*. U projektu ovog diplomskog rada politika ne postoji, jer algoritam Q-Učenja (jedan od Off-policy algoritama) koristi tzv. Q-Vrijednosti kako bi izračunao najbolju akciju agenta u trenutnom stanju.

Epizodu predstavljaju sva provjerena stanja između inicijalnog stanja (prvo učitano stanje) i terminalnog stanja (zadnje učitano stanje). U projektu ovog diplomskog rada epizode predstavljaju svakih 10 sličica po sekundi, od inicijalizacije video igre do trenutka uništenja agenta.

Nagrada je brojčana vrijednost koju agent prima od strane okoline kao odgovor na napravljenu radnju ili akciju. Nagrade Agentov cilj je maksimizirati cjelokupnu nagradu koju primi tijekom jedne epizode.

Podržano učenje funkcionira na način da agent očita trenutno stanje okoline te donese odluku (tj. akciju) temeljenu na trenutnom stanju okoline. Svaka napravljena akcija donosi svoje nagrade (ako je akcija pozitivna) ili kazne (ako je akcija negativna). No, nije svaka odluka donesena na temelju trenutnog stanja, nego se nekada odluke mogu donijeti na temelju stohastičko, tj. nasumično. Takvo donošenje odluka se zove „Kompromis istraživanja i iskorištavanja" (eng. Exploration vs. exploitation tradeoff) te je opisan u postotku, tj. što je veći postotak to je veća šansa da se dogodi nasumična akcija ili donese nasumična odluka. Nakon svake iteracije učenja se taj postotak smanjuje sve dok ne dođe do nule. Kada je taj broj nula, agent će uvijek donositi odluke temeljene na dobivenom iskustvu. Kompromis istraživanja i

iskorištavanja se koristi kako bi agent mogao što više naučiti o svojoj okolini u početku simulacije, jer se tako agent dobiva određeno iskustvo u različitim situacijama.

Nakon napravljene akcije, očitava se novo stanje okoline te zajedno sa trenutnim stanjem i napravljenom akcijom učitava u model.

2.3. Q-Učenje

Q-Vrijednost (eng. Q-Value), čije ime dolazi od riječi kvalitete (eng. Quality), je mjera sveukupne očekivane nagrade pod pretpostavkom da je agent u trenutnom stanju „*s*“ te da izvodi trenutnu akciju „*a*“.

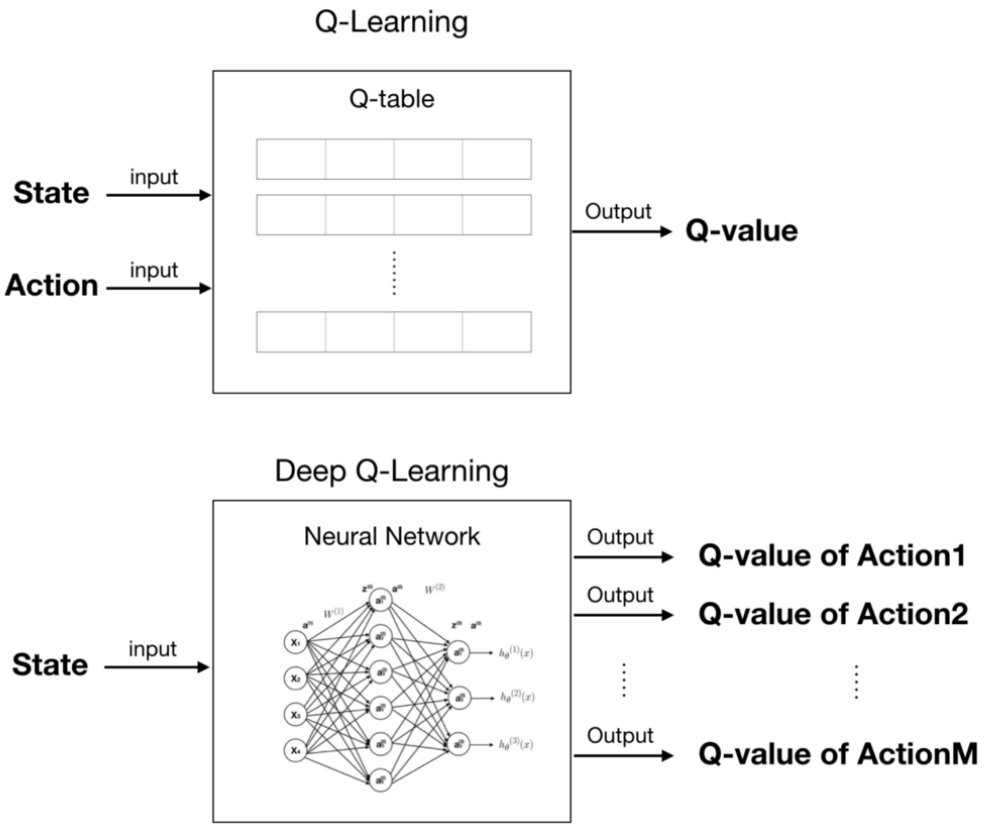
Q-Učenje (eng. Q-Learning) je jedan od najkorištenijih algoritama podržanog učenja. Q-Učenje koristi tablice kako bi sortirao sve Q-Vrijednosti svih mogućih parova izvršenih akcija i trenutnih stanja. Za ažuriranje tablice, Q-Učenje koristi *Bellmanovu jednadžbu*.

$$Q(s,a) = r(s,a) + \gamma max_a Q(s',a)$$

Bellmanova jednadžba tvrdi da je Q-Vrijednost dobivena iz stanja s odabirom akcije a neposredna primljena nagrada r(s,a) na koju se dodaje najveća moguća Q-Vrijednost iz stanja s' (tj. iz budućeg stanja koje je očitano nakon izvršene akcije a u stanju s). Dobit ćemo najveću Q-Vrijednost od s' odabirom akcije koja maksimizira Q-Vrijednost. Također uvodimo γ, koji se obično naziva diskontni faktor (eng. Discount factor), koji kontrolira važnost dugoročnih nagrada u odnosu na neposredne. Diskontni faktor varira u vrijednostima od 0 do 1, gdje 1 znači da se stavlja najveća moguća važnost na dugoročne nagrade, a 0 gdje se najveća vrijednost daje kratkoročnim nagradama. (Zychlinski, 2019)

2.4. Duboko Q-Učenje

Duboko Q-Učenje (eng. Deep Q-Learning) ili Duboke Q Mreže (eng. Deep Q Networks) je kombinacija Q-Učenja i dubokog učenja. Kao što je spomenuto u radu, Q-Učenje koristi tablice kako bi sortirao sve Q-Vrijednosti svih mogućih parova izvršenih akcija i trenutnih stanja, no u dubokom Q-Učenju te tablice se mijenjaju sa neuronskim mrežama koje pokušavaju naći približne Q-Vrijednosti. Takva neuronska mreža se često naziva *aproksimator* ili *aproksimacijska funkcija*.



Slika 3: Razlika između Q-Učenja i Dubokog Q-Učenja
Izvor: https://medium.com/@qempsil0914/zero-to-one-deep-q-learning-part1-basic-introduction-and-implementation-bb7602b55a2c

2.5. PyTorch i Pygame

Za izradu projekta ovog diplomskog rada su korištene biblioteke PyTorch (korišten u izradi neuronskih mreža) i Pygame (korišten za izradu okoline i agenta).

PyTorch je softver otvorenog koda koji služi kao okvir za strojno učenje temeljen na Torch bibilioteci koja je razvijena od strane Meta AI (Yegualalp, 2017). PyTorch je besplatan softver objavljen pod BSD licencom. Kako bi se pohranile ogromne količine homogenih višedimenzionalnih niza brojeva, koristi se PyTorch klasa koja se zove Tensor. Pytorch tenzor je jako sličan NumPy listama, razlika je u tome što PyTorch tenzori, osim što mogu obavljati kalkulacije i upravljati podacima na CPU-u, mogu obavljati kalkulacije i upravljati podacima na Nvidia GPU-u koji podržava CUDA API.

Instalacija PyTorch softvera je vrlo jednostavna, na web stranici PyTorch softvera se može naći pomagalo u kojem korisnik može odabrati koju verziju želi instalirati, za koji se operacijski sustav želi korisnik instalirati, kroz koji paket (conda, pip, LibTorch), u kojem će se jeziku koristiti (Python, C++, Java) te koja će se platforma koristiti za kalkulacije (CPU, CUDA). Nakon što korisnik odabere svoje preferencije, web stranica će izbaciti točnu naredbu koja se može iskoristiti unutar Windowsove naredbene ploče kako bi se instalirao softver.

Za svrhe projekta ovog diplomskog rada je korištena sljedeća naredba za instalaciju PyTorch softvera:

```
pip3 install torch torchvision torchaudio
```

Kako bi se koristio PyTorch, u datoteci u kojoj se koristi se mora dodati sljedeća linija koda:

```
import torch
```

U projektu ovog diplomskog rada su korišteni sljedeći moduli PyTorch-a:

1. torch.nn – neuronske mreže
2. torch.optim - optimizacija
3. torch.nn.functional – funkcije neuronske mreže

Torch.nn modul nudi različite alate kojima se lakše mogu definirati neuronske mreže tako da se samo definiraju slojevi.

Torch.optim modul koji nudi različite optimizacijske algoritme koji se koriste u izradi neuronskih mreža. Većinom svi standardni optimizatori su podržani, poput SGD, Adam, LBFGS, itd.

Torch.nn.functional služi kako bi se definirali prilagođeni slojevi neuronske mreže. Slojevi definirani torch.nn.funcional modulom nisu potpuni slojevi, nego se moraju koristiti uz torch.nn modul. Također, torch.nn.functional sadrži brojne korisne funkcije poput aktivacijskih funkcija (ReLU, Sigmoid, Tanh, itd.) i konvolucijskih operacija.

Pygame je kolekcija modula koji se mogu koristiti kako bi se lakše napravile video igre u Pythonu. Pygame omogućuje izrađivanje potpuno funkcionalnih video igara i ostalih multimedijalnih programa u Python programskom jeziku (Pygame, 2020.). Pygame je portabilan te se može pokretati na gotovo svakoj platformi i operacijskom sustavu.

Pygame sadrži sljedeće značajke zbog kojih je jedan od najboljih okvira za izradu jednostavnih 2D igara u Pythonu, a to su:

- Jednostavnost – jednostavan i lagan za korištenje, savršeno za početnike u Pythonu
- Pouzdanost – mnogo se video igara objavilo, koje su koristile Pygame, što znači da je sami modul bio testiran par miliona puta
- Kontrola – mogućnost kontrole glavne petlje te veća kontrola kod korištenja drugih rječnika
- Kompaktnost – pygame ne zahtijeva stotine i tisuće linija koda kako bi program radio neke sitnice, jezgra pygame-a je jednostavna te dodatne stvari poput GUI rječnika i različitih efekata se mogu nadodati posebno izvan igre
- Modularnost – pygame omogućava posebno korištenje zasebnih dijelova modula (Pygame, 2020.).

Instalacija Pygame-a se također vrši pomoću naredbene ploče, no može se i instalirati tako da se preuzme instalacijska datoteka sa Pygame-ove web stranice. Za svrhe projekta ovog diplomskog rada je korištena sljedeća naredba za instalaciju Pygame-a:

```
pip install pygame
```

Kako bi se koristio Pygame, u datoteci u kojoj se koristi se mora dodati sljedeća linija koda:

```
import pygame
```

Također postoji i Pygame-ov modul za grafičko sučelje zvan *Pygame GUI* koji služi kako bi se izradilo grafičko sučelje za video igre rađene sa Pygame-om.

Pygame GUI sadrži sljedeće značajke:

- Tematiziran – mogućnost korištenja JSON datoteka kako bi se zadala jedinična tema korisničkog sučelja cijelog projekta
- Podržava HTML – podržava neke od HTML-ovih podskupova poput omotavanja, stiliziranja te ostalih HTML-ovih markup-a.
- Standardni elementi – podržava standardne elemente korisničkog sučelja, poput gumbova, paragrafa, tekstnih unosa, traka za pomicanje i padajući izbornici
- Lokalizacija – podržava mogućnost lakše lokalizacije za različite jezike
- Kompatibilnost – uvažava Pygame-ov način programiranja, tako da se ne osjeti razlika između dva modula

Instalacija Pygame GUI-a se vrši pomoću naredbene ploče sa sljedećom naredbom:

```
pip install pygame_gui
```

Kako bi se koristio Pygame, u datoteci u kojoj se koristi se mora dodati sljedeća linija koda:

```
import pygame_gui
```

3. Metodologija rada

Za izradu ovog diplomskog rada je korištena metoda deskripcije. Deskripcija je opis pojava koje se istražuju. Svako istraživanje bi trebalo započeti s deskripcijom svih temeljnih pojmova ili pojava. U znanstvenom radu velika pozornost posvećuje se detaljnom opisivanju činjenica, pojava ili podataka kako bi se povećala objektivnost i točnost u svim drugim fazama istraživanja. (Žugaj i dr., 2006).

Glavna hipoteza ovog rada jest testiranje implementabilnosti podržanog strojnog učenja u okruženju video igara. Verifikaciju hipoteze temeljimo na izrađenom modelu neuronske mreže korištenjem dubokog Q-Učenja, koji je jedan od najkorištenijih algoritama podržanog učenja te primjenom tog modela na video igri.

Glavni izvori podataka za opis ovog diplomskog rada je korišten zadatak diplomskog rada, te ostali izvori literature koji su korišteni za izradu tog zadatka i izvori koji su korišteni za razumijevanje podržanog učenja.

Tijekom izrade zadatka diplomskog rada najviše poteškoća se pronalazilo kod pisanja koda samog modela, zbog ne razumijevanja literature i velikih količina informacija koje se podrazumijevaju kao temelj razumijevanja podržanog učenja. No poteškoće su se riješile kombinacijom postupka pokušaja i pogreške te ponovnim proučavanjem literature.

4. Opis istraživanja i rezultati istraživanja

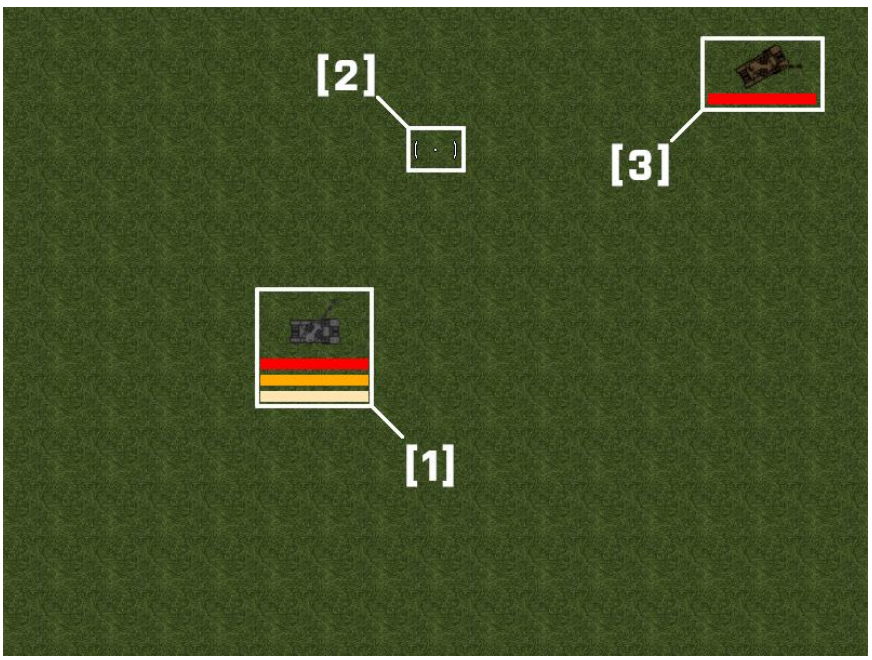
4.1. Opis projekta diplomskog rada

Zadatak projekta diplomskog rada je bio izraditi model neuronske mreže te ga primijeniti na primjer jednostavne video igre. Video igra je vrlo jednostavnog tipa u kojoj je cilj uništiti što više neprijateljskih oklopnih vozila. Nakon svakog uništenog neprijateljskog oklopnog vozila započinje nova runda sa novim neprijateljem.

Igrač može upravljati svojim oklopnim vozilom na 3 načina, upravljanje pozicijom oklopnog vozila (tj. pokretanje oklopnog vozila) prema *x* i *y* osi, upravljanje pozicijom nišana (tj. pokretanje nišana) prema *x* i *y* osi te ispaljivanje projektila. Ulogu igrača u ovom slučaju preuzima tzv. *Agent*, tj. entitet sličan igraču kojeg kontrolira neuronska mreža. Agent je podijeljen na 2 dijela

Okolina uzima ulogu neprijateljskih oklopnih vozila koji se pomiču po koordinatnom sustavu nasumično. Neprijateljska oklopna vozila se pomiču tako da im se dodijeli nasumična točka na koordinatnom sustavu te se pomiču u pravcu od trenutne točke do dodijeljene točke.

Projekt ovog diplomskog rada je dostupan na GitHub-u na sljedećoj poveznici: https://github.com/PetarGarmaz/Proces-izrade-umjetne-inteligencije-podrzanim-ucenjem



Slika 4: Isječak video igre i entiteti u igri. 1) Agentov trup; 2) Agentov nišan; 3) Neprijateljsko oklopno vozilo
Izvor: Autor

4.2. Izrada procesa treniranja

Projekt ovog diplomskog rada se sastoji od sljedećih modula:

- Main
- Agent
- Enemy
- TrainAim
- TrainMovement
- Model
- Plot

Main je glavni modul koji je rađen u MVC arhitekturi, te je zbog toga podijeljen na 3 klase: model, view i controller. Program započinje tako da se prvo pozove Controller klasa, koja upravlja cijelim programom. Kada se ta klasa pozove, inicijaliziraju se početne varijable koje se koriste kako bi video igra mogla funkcionirati te se učitavaju varijable koje se koriste za treniranje neuronske mreže. U klasi model se definiraju svi korišteni objekti i njihove varijable uključujući agenta i neprijateljske objekte, te logika video igre, dok se u klasi view definira

Agent i *Enemy* moduli se sastoje od više klasa, od kojih sve služe kako bi dodavale funkcionalnost agentu i neprijateljskim objektima, poput pomicanja trupa, pomicanja nišana, ispaljivanja projektila.

TrainAim i *TrainMovement* moduli sadrže klase pod istim imenom u kojima se nalaze metode i funkcije koje će se koristiti tijekom treniranja nišanjenja i kretanja, respektivno.

Model je također jedna od glavnih modula, u kojem se trenira neuronska mreža te se sastoji od 2 klase, jedna klasa definira neuronsku mrežu, dok ju druga trenira. Također ovaj modul ima mogućnost spremanja i učitavanja modela neuronske mreže.

Plot modul je pomoćni modul kojim se definira objekt grafa kojim se pokazuju rezultati trenutne i prijašnjih epizoda.

```
class GameController():
    def __init__(self):
        #Initializing pygame and pygame sound settings
        pygame.init()
        pygame.mixer.init(frequency = 44100, size = 32, channels = 2, buffer = 512)
        pygame.mixer.set_num_channels(16)

        self.model = GameModel()
        self.view = GameView()

        #Misc
        self.done = False
        self.menuDone = False
        self.paused = False

        self.frameIteration = 0
        self.steps = 0
        self.gameNum = 1

        self.isTraining = False
        self.isNewModel = True
        self.roundOver = False

        self.movementTrainerObject = TrainMovement()
        self.aimTrainerObject = TrainAim()

        #Plot
        self.plotScores = [0]
        self.plotMeanScores = [0]
        self.totalScore = 0
        self.record = 0

        #Start with main menu
        self.UpdateMainMenu()
```

Slika 5: Isječak koda inicijalizacije Controller-a
Izvor: Autor

4.2.1. Glavna petlja treninga

Glavna petlja treninga se odnosi na *while* petlju koja će se ponavljati sve dok korisnik ne odluči da su neuronske mreže dovoljno uvježbane. Prema postavkama video igre, petlja se odvija 60 puta u sekundi, tj. 60 sličica po sekundi (eng. FPS, tj. frames per second). No prije nego što se petlja pokrene, treba prvo inicijalizirati objekt grafa te se učitavaju postojeći modeli neuronskih mreža (rezultat postoje li već napravljeni modeli neuronskih mreža će se spremiti u varijablu *isNewModel* u obliku *boolean-a*).

Tek nakon inicijalizacije grafa i učitavanja postojećih neuronskih mreža se pokreće *while* petlja, koja će se ponavljati do trenutka kada varijabla *done* bude točna. U glavnoj petlji se rade glavne provjere poput trenutnih životnih bodova agenta, postoji li neprijatelj na polju te se vrte osnovne operacije video igre, poput provjeravanja kolizije, tj. dodirivanja objekata agenta i neprijatelja, te crtanje sličica, tj. Sprite-ova. Nakon obavljenih osnovnih radnji program prelazi na operacije treniranja.

Treniranje se odvija svakih 10 sličica, tj. svaku 1/6 sekunde, zbog velikog broja računanja koje računalo mora obaviti u procesu treniranja modela neuronskih mreža. Smatralo se da je svakih 10 sličica dovoljno vremena kako bi računalo moglo relativno pouzdano moglo uzimati informacije o okolini te izvesti neku akciju. U slučaju da se uzelo 5 sličica ili niže vrijeme treninga bi bilo nešto duže, a ako bi se uzelo 30 ili više sličica moglo bi doći do toga da postoji previše zaostajanja između učitavanja informacija o okolini i izvođenja akcija.

Svakih 10 sličica se pokreće proces treniranja zasebnih dijelova agenta. Kako bi se pokrenuo taj proces, pokreće se funkcija *Train()* u svakom objektu treniranja (*movementTrainerObject* i *aimTrainerObject*).

```
def UpdateGame(self):
        self.UpdatePlot()

        movementLoaded = self.movementTrainerObject.Load()
        aimLoaded = self.aimTrainerObject.Load()

        if(movementLoaded and aimLoaded):
            self.isNewModel = False

        while not self.done and self.menuDone:
            self.GameEventHandler()

            if(self.model.agent.health <= 0):
                self.roundOver = True
                self.ResetGame()

            if(not self.paused):
                self.model.GameEnemySpawner()
                self.model.GameLogic()
                self.model.GameDraw(self.view)

                if(self.frameIteration % 10 == 0):
                    if(self.isTraining):
                        self.movementTrainerObject.Train(self.steps, self.gameNum,
self.roundOver, self.model.enemies, self.model.agent, self.isNewModel)
                        self.aimTrainerObject.Train(self.steps, self.gameNum, self.roundOver,
self.model.enemies, self.model.agentCrosshair, self.model.agentTurret, self.isNewModel)
                    else:
                        self.movementTrainerObject.Test(self.steps, self.gameNum,
self.model.enemies, self.model.agent)
                        self.aimTrainerObject.Test(self.steps, self.gameNum,
self.model.enemies, self.model.agentCrosshair, self.model.agentTurret)

                    self.steps += 1

                self.frameIteration += 1
```

Slika 6: Isječak koda glavne petlje
Izvor: Autor

4.2.2. Treniranje kretanja i nišana

Treniranje kretanja i nišana su odvojene u 2 klase: *TrainMovement* i *TrainAim*. Obje klase dijele sličnu inicijalizaciju. U inicijalizaciji ovih klasa se nalaze različite varijable koje će se koristiti za treniranje. Varijable poput *oldState* i *newMove* će se koristiti kako bi se spremila trenutna okolina i izvršena akcija, respektivno. Nadalje, *batchSize* definira koliko će biti velika hrpa podataka na kojima će se neuronska mreža ponovo trenirati nakon završetka epizode te se smatra da je vrijednost od 10.000 optimalna jer dozvoljava dovoljan broj podataka za ponovno treniranje, a nije toliko velika da usporava cijeli proces, *maxMemory* definira maksimalnu količinu podataka koju će program spremati, *learnRate* (hrv. stopa učenja) definira koliko će se model promijeniti kao odgovor na procijenjenu pogrešku svaki put kada se težine modela promijene, vrijednost od 0.001 je procijenjena optimalna vrijednost gdje je učenje nije previše nestabilno. Stopa učenja se može objasniti na primjeru učenja kretanja, npr. može postojiti slučaj u kojem se agent samo zna pomicati prema gore kao reakcija na približavanje neprijatelja. Nakon nekoliko iteracija, agent će kad tad naučiti se pomaknuti u drugim smjerovima uz optimalnu stopu učenja, no ako je stopa učenja preniska može doći do toga da će agent odbaciti pokretanje u drugim smjerovima kao izuzetak te neće ponavljati takve akcije, a ako je stopa učenja prevelika onda će agent odmah početi vjerovati kako je zadnja napravljena akcija (koja nije pokret prema gore) najbolja akcija te ubuduće više neće koristiti akciju pomicanja prema gore.

Varijabla *discountRate (γ)* definira važnost dugoročnih nagrada te se smatra da je vrijednost od 0.9 optimalna jer većina situacija u video igri zahtijevaju „razmišljanje unaprijed“, te kratkoročne nagrade neće uvijek biti dostupne. Diskontni faktor se može objasniti na primjeru nišanjenja, npr. uz optimalni diskontni faktor (0.8 – 0.9), nišan će bez problema moći preći veliku udaljenost kako bi došlo do svog cilja te će ga uspješno pratiti na kratkim udaljenostima, no ako je taj broj velik (1) može se dogoditi da će nišan uspješno preći velike udaljenosti do svog cilja, no kad stigne do kraćih udaljenosti neće se moći odlučiti za sljedeću akciju jer razmišlja previše unaprijed te se ne fokusira na kratkoročne nagrade kada treba, a ako je diskontni faktor prenizak (0 - 0.5) onda će nišan teže dolaziti do svog cilja, jer će samo razmišljati o kratkoročnim nagradama. Varijabla *memory* definira mjesto gdje se pohranjuju podaci skupljeni tijekom epizode, *reward* definira trenutne nagrade ili kazne koje se skupljaju tijekom epizode.

Varijabla *trainerModel* definira samu neuronsku mrežu, gdje svaka stavka u zagradi predstavlja veličine ulaznog, skrivenog te izlaznog sloja respektivno. Veličina ulaznog i izlaznog sloja

mora biti iste veličine kao i veličina niza okoline i niza mogućih akcija, dok skriveni sloj može biti bilo koje veličine (promjena veličine skrivenog sloja nije donijela velike promijene u procesu učenja, te se smatra da je veličina od 512 neurona sasvim dovoljna).

Varijabla *trainer* definira algoritam treniranja koji će se koristiti za neuronsku mrežu, te joj se prosljeđuje varijabla *trainerModel*, *learnRate* i *discountRate*.

```
class TrainMovement():
    def __init__(self):
        self.oldState = None
        self.newMove = None

        self.batchSize = 1000
        self.maxMemory = 100000
        self.learnRate = 0.001
        self.discountRate = 0.9                 #Gamma
        self.memory = deque(maxlen=self.maxMemory)
        self.reward = 0

        self.trainerModel = Linear_QNet(7, 512, 5)
        self.trainer = QTrainer(self.trainerModel, lr=self.learnRate, gamma=self.discountRate)

#...

class TrainAim():
    def __init__(self):
        self.oldState = None
        self.newMove = None

        self.batchSize = 1000
        self.maxMemory = 100000
        self.learnRate = 0.001
        self.discountRate = 0.9                 #Gamma
        self.memory = deque(maxlen=self.maxMemory)
        self.reward = 0

        self.trainerModel = LinearQNN(5, 512, 5)
        self.trainer = QTrainer(self.trainerModel, lr=self.learnRate, gamma=self.discountRate)
```

Slika 7: Isječak koda klasa TrainMovement i TrainAim
Izvor: Autor

Nakon inicijaliziranih klasi dolazi pozivanje funkcije Train. U toj funkciji se postepeno pozivaju ostale funkcije kojima se dohvaća trenutno stanje okoline, nova akcija te se izvode

nove akcije i pridonose nagrade, kasnije se uzima novo stanje okoline nakon izvedene akcije te se poziva funkcija treniranja kratkotrajne memorije i pamćenja.

```
    def Train(self, steps, gameNum, roundOver, enemies, entity, isNewModel):
        if(steps % 2 == 0):
            self.oldState = self.GetState(enemies, entity)

            self.newMove = self.GetAction(gameNum, isNewModel)

            self.PerformMove(entity)

            self.AssignRewards(enemies, entity)

        elif(steps % 2 != 0):
            newState = self.GetState(enemies, entity)

            self.TrainShortMemory(self.oldState, self.newMove, self.reward, newState,
roundOver)

            self.Remember(self.oldState, self.newMove, self.reward, newState, roundOver)
```

Slika 8: Isječak koda funkcije Train, u klasi TrainMovement
Izvor: Autor

Također bitno je primijetiti kako se funkcija Train() dijeli na 2 dijela, u prvom dijelu, gdje je varijabla *steps* parni broj, se odvijaju radnje uzimanja trenutnog stanja, uzimanja i izvođenja nove akcije te pridonošenja nagrada, a u drugom dijelu, gdje je varijabla *steps* neparni broj, se odvijaju radnje uzimanja novog stanja, treniranja kratkotrajne memorije i pamćenja. Varijabla *steps* predstavlja broj koliko puta se pozvala funkcija Train().

Ta podjela se koristi kako bi se dalo malo vremena između izvođenja neke akcije i uzimanja novog stanja, jer da se izvođenje akcije i novo stanje izvode u jednoj sličici (tj. u istoj iteraciji) trenutno stanje se ne bi razlikovalo od novog stanja te neuronska mreža ne bi bila dobro istrenirana.

Funkcija *GetState()* dohvaća trenutno stanje okoline kako bi agent znao što se događa oko njega. Postoji razlika između GetState funkcije kod klase TrainMovement i TrainAim u tome što svaka funkcija uzima samo one podatke koje su bitne za tu klasu.

Kod TrainMovement klase je bitna pozicija agenta, vektor kretanja agenta, pozicija neprijatelja, te udaljenost između neprijatelja i agenta, dok kod TrainAim klase se gleda odnos između nišana agenta i neprijatelja sa 4 strane agentovog nišana (iznad nišana, desno od nišana, ispod nišana, lijevo od nišana), npr. ako je neprijatelj iznad nišana agenta, onda će prva vrijednost niza biti 1, ako ne onda je ta vrijednost 0. Isto je kod TrainAim klase bitan podatak o vremenu od zadnjeg ispaljivanja projektila, kako bi agent znao samog sebe tempirati, pošto je potrebno 7 sekundi kako bi se ponovo napunila cijev za paljbu.

Na kraju, funkcija vraća sve te podatke u obliku numpy niza.

```
def GetState(self, enemies, entity):
    vector = entity.movement
    pos = entity.position
    x, y = pos
    vx, vy = vector

    state = [x, y, vx, vy]

    if(len(enemies) > 0):
        for enemy in enemies:
            enemyPos = enemy.position
            ex, ey = enemyPos
            distance =
enemyPos.distance_to(pos)

            state.append(ex)
            state.append(ey)
            state.append(distance)

    else:
        defaultPos = Vector2(0, 0)
        distance =
pos.distance_to(defaultPos)
        state.append(0)
        state.append(0)
        state.append(distance)

    npState = np.array(state, dtype=int)
    return npState
```

```
def GetState(self, enemies, entity,
entityTurret):
    pos = entity.position
    x, y = pos

    state = [int(entityTurret.cooldown /
7)]

    if(len(enemies) > 0):
        for enemy in enemies:
            enemyPos = enemy.position
            ex, ey = enemyPos
            distance =
enemyPos.distance_to(pos)

            if(distance > 50):
                state.append(int(abs(y -
ey) > 25 and y > ey))      #UP
                state.append(int(abs(y -
ey) <= 25 and x < ex))     #RIGHT
                state.append(int(abs(y -
ey) > 25 and y < ey))      #DOWN
                state.append(int(abs(y -
ey) <= 25 and x > ex))     #LEFT
            else:
                state.append(1)
                state.append(1)
                state.append(1)
                state.append(1)
    else:
        state.append(0)
        state.append(0)
        state.append(0)
        state.append(0)

    npState = np.array(state, dtype=int)
    return npState
```

Slika 9: Isječci koda funkcije GetState u TrainMovement (lijevo) i TrainAim (desno) klasama
Izvor: Autor

Funkcija *GetAction()* služi kako bi se dohvatila neka akcija koju agent mora izvršiti. Kao što je spomenuto u poglavlju Teorijske podloge, postoji određeni kompromis koji se mora napraviti između istraživanja i iskorištavanja, u ovoj funkciji to je varijabla *epsilon.*

Varijabla epsilon se mijenja nakon svake epizode, tj. runde (predstavljena varijablom *gameNum*), te se postepeno smanjuje. On predstavlja kolika je šansa da će sljedeća akcija biti nasumična, u početku ta je šansa 100%, no nakon 100 epizoda ta šansa se spušta na 0%.

Ako je akcija nasumična, onda se uzima nasumična radnja, ako akcija nije nasumična, onda se pokušava predvidjeti koja bi akcija bila najkvalitetnija, tj. bit će odabrana ona akcija koja ima najveću predviđenu Q-Vrijednost. Predviđanje se radi pozivanjem funkcije *forward* u objektu neuronske mreže, tj. *trainerModel*. Od dobivenog niza funkcije *forward* se uzima najveći broj u nizu te će taj broj predstavaljati predviđenu akciju.

Predviđena akcija se vraća u obliku niza, gdje je broj koji dijeli indeks predviđene akcije jednak 1, npr. ako niz dobiven funkcijom *forward* je jednak *[48.9784, 49.7803, 43.5083, 64.4452, 38.8251]*, predviđena akcija će biti broj indeksa najvećeg broja, u ovom slučaju indeks 3 koji iznosi *65.4452*. Tada će se u varijablu *newMove*, koja je trenutno *[0, 0, 0, 0, 0]* postaviti vrijednost 1 pod indeks najvećeg broja, tj. pod indeks 3, te će ta varijabla biti *[0, 0, 0, 1, 0]*.

```
def GetAction(self, gameNum, isNewModel):
    epsilon= 100 - gameNum
    newMove = [0, 0, 0, 0, 0]
    move = 0

    if (random.randint(0, 100) < epsilon and isNewModel):          #Do random
        move = random.randint(0, 4)
    else:                                                          #Do NN
        state0 = torch.tensor(self.oldState, dtype=torch.float)
        move = self.trainerModel.forward(state0)
        move = torch.argmax(move).item()

    newMove[move] = 1

    return newMove
```

Slika 10: Isječak koda funkcije GetAction() u TrainMovement() klasi
Izvor: Autor

Funkcija *PerformMove()* služi kako bi se izvršila predviđena ili nasumična nova akcija. Oba dijela agenta mogu zasebno izvršavati akcije korištenjem ove funkcije.

Agentov trup uključuje sljedeće akcije: pomicanje prema gore, pomicanje prema desno, pomicanje prema dolje, pomicanje prema lijevo i stajanje na mjestu.

Agentov nišan uključuje sljedeće akcije: pomicanje prema gore, pomicanje prema desno, pomicanje prema dolje, pomicanje prema lijevo i ispaljivanje projektila.

U ovoj funkciji postoji jednostavna provjera u kojoj se gleda koja je od vrijednosti u nizu predviđene akcije jednaka broju 1 te se u skladu s time dio agenta koji se kontrolira u toj klasi pomiče prema zakazanom smjeru, npr. ako je predviđena akcija jednaka [0, 1, 0, 0, 0], onda će se dio agenta pomicati prema desno, ovisno u kojoj se klasi priziva ova funkcija.

```
def PerformMove(self, entity):
    #Move action
    if(self.newMove[0] == 1):
        entity.movement = Vector2(0, -
2)
    elif(self.newMove[1] == 1):
        entity.movement = Vector2(2,
0)
    elif(self.newMove[2] == 1):
        entity.movement = Vector2(0,
2)
    elif(self.newMove[3] == 1):
        entity.movement = Vector2(-2,
0)
    elif(self.newMove[4] == 1):
        entity.movement = Vector2(0,
0)
```

```
def PerformMove(self, entityTurrent):
    #Aim action
    if(self.newMove[0] == 1):
        entityTurrent.movement =
Vector2(0, -3)
    elif(self.newMove[1] == 1):
        entityTurrent.movement =
Vector2(3, 0)
    elif(self.newMove[2] == 1):
        entityTurrent.movement =
Vector2(0, 3)
    elif(self.newMove[3] == 1):
        entityTurrent.movement =
Vector2(-3, 0)
    elif(self.newMove[4] == 1):
        entityTurrent.movement =
Vector2(0, 0)



    #Fire action
    entityTurrent.doFireMain =
bool(self.newMove[4])
```

Slika 11: Isječak koda funkcije PerformMove() u klasi TrainMovement (lijevo) i TrainAim (desno)
Izvor: Autor

Funkcija *AssignRewards()* služi kako bi se dodijelile nagrade ili kazne agentu za napravljenu akciju.

Agentov trup može dobiti nagradu ako je dovoljno daleko udaljen od neprijatelja, tj. ako je udaljen više od 400 piksela od neprijatelja, a ako pokušava izaći iz postojećeg igraćeg polja (koje je široko 800 piksela i visoko 600 piksela) agentov trup dobiva kaznu.

Agentov nišan može dobiti nagradu ako je udaljen manje od 50 piksela od neprijatelja i još veću nagradu ako ispali projektil kada je dovoljno blizu neprijatelja.

```
def AssignRewards(self, enemies,
entity):
    self.reward = 0
    pos = entity.position
    x, y = pos

    if(y <= 10 and self.newMove[0] ==
1):
        self.reward = -10
    elif(x >= 790 and self.newMove[1]
== 1):
        self.reward = -10
    elif(y >= 590 and self.newMove[2]
== 1):
        self.reward = -10
    elif(x <= 10 and self.newMove[3]
== 1):
        self.reward = -10

    for enemy in enemies:
        enemyPos = enemy.position
        distance =
enemyPos.distance_to(pos)

        if(distance >= 400):
            self.reward = 10
```

```
def AssignRewards(self, enemies,
entity):
        self.reward = 0
        crosshairPos = entity.position

        for enemy in enemies:
            enemyPos = enemy.position
            distance =
enemyPos.distance_to(crosshairPos)

            if(distance <= 50):
                self.reward = 10

                if(self.newMove[4] ==
1):
                    self.reward = 20
```

Slika 12: Isječak funkcije AssignRewards() u klasi TrainMovement (lijevo) i TrainAim(desno)
Izvor: Autor

Funkcija *Remember()* služi kako bi se spremilo trenutno stanje, predviđena akcija, ukupna nagrada, sljedeće stanje i stanje epizode (je li završena ili nije) u memoriju koja će se kasnije koristiti za treniranje dugotrajne memorije.

```
def Remember(self, state, action, reward, nextState, roundOver):
    self.memory.append((state, action, reward, nextState, roundOver))
```

Slika 13: Isječak funkcije Remember()
Izvor: Autor

Funkcija *TrainShortMemory()* služi kako bi se pokrenuo proces treniranja modela neuronske mreže korištenjem algoritma dubokog Q-Učenja, ali samo u trenutnoj iteraciji.

```
def TrainShortMemory(self, state, action, reward, nextState, roundOver):
    self.trainer.TrainStep(state, action, reward, nextState, roundOver)
```

Slika 14: Isječak funkcije TrainShortMemory
Izvor: Autor

Funkcije *Save()* i *Load()* služe kako bi se spremio model ili učitao model iz *.pth* datoteke.

```
def Save(self):
    self.trainerModel.save("driverBrain.pth")

def Load(self):
    isLoaded = self.trainerModel.load("driverBrain.pth")
    return isLoaded
```

Slika 15: Isječak funkcije Save() i Load() u klasi TrainMovement
Izvor: Autor

Funkcija *TrainLongMemory()* je funkcija koja se poziva na kraju epizode, tj. kada se agent uništi te služi kako bi se model istrenirao na podacima koji su spremljeni u varijabli *memory*.

```
def ResetGame(self):
    self.gameNum += 1
    self.frameIteration = 0
    self.steps = 0

    if(self.isTraining):
        self.movementTrainerObject.TrainLongMemory()
        self.aimTrainerObject.TrainLongMemory()

        if self.model.score > self.record:
            self.record = self.model.score
            self.movementTrainerObject.Save()
            self.aimTrainerObject.Save()

    self.UpdatePlot()

    self.model.__init__()
    self.model.agent.maxHealth = self.view.healthSlider.get_current_value()
    self.model.agent.health = self.model.agent.maxHealth
    self.roundOver = False
```

Slika 16: Isječak ResetGame() funkcije u klasi GameController u Main modulu, prikazuje gdje se poziva funkcija TrainLongMemory
Izvor: Autor

No pošto nije realistično uzeti sve podatke iz memorije, zbog njene moguće veličine, uzima se nasumični uzorci te se isti premještaju i slažu u zasebne varijable, tj. sva trenutna stanja se slažu u varijablu *states*, sve napravljene akcije se slažu u varijablu *actions*, itd.

Nakon što se premjeste podaci, isti se šalju u model neuronske mreže gdje će se koristiti za treniranje.

```
def TrainLongMemory(self):
    smallSample = None

    if len(self.memory) > self.batchSize:
        smallSample = random.sample(self.memory, self.batchSize)
    else:
        smallSample = self.memory

    states, actions, rewards, newStates, dones= zip(*smallSample)

    self.trainer.TrainStep(states, actions, rewards, newStates, dones)
```

Slika 17: Isječak funkcije TrainLongMemory()
Izvor: Autor

4.2.3. Model

Model je odvojen u 2 klase: *LinearQNN* i *Qtrainer*. *LinearQNN* klasa se koristi za jednostavniju izradu neuronskih mreža pomoću torch.nn modula koje pruža PyTorch.

U inicijalizaciji LinearQNN klase, nasljeđuje se torch.nn modul te se rade 2 varijable koje će predstavljati vezu između zadanih slojeva.

```
class LinearQNN(nn.Module):
    def __init__(self, inputSize, hiddenSize_1, outputSize):
        super().__init__()
        self.linear1 = nn.Linear(inputSize, hiddenSize_1)
        self.linear2 = nn.Linear(hiddenSize_1, outputSize)
```

Slika 18: Isječak inicijalizacije klase LinearQNN
Izvor: Autor

Pošto je LinearQNN klasa naslijedila torch.nn modul, može se postaviti *forward()* funkcija koja se prema zadanim postavkama koristi za predviđanje izlaza iz nekog ulaza.

Kako bi pretvorili ulaz u izlaz, potrebno je prvo ulaz provući kroz aktivacijsku funkciju koja definira kako će se ulaz pretvoriti u izlaz, u ovom slučaju se koristi ReLU (eng. Rectified Linear Unit) funkcija. Nakon toga se input, koji je provučen kroz aktivacijsku funkciju, provlači kroz drugi sloj koji je definiran u inicijalizaciji, kako bi izlaz bio pravilne forme, te se taj izlaz vraća return funkcijom.

```
def forward(self, x):
    xInput = self.linear1(x)
    xHidden_1 = F.relu(xInput)
    xOutput = self.linear2(xHidden_1)

    return xOutput
```

Slika 19: Isječak forward() funkcije
Izvor: Autor

U inicijalizaciju *QTrainer* klase ulazi model neuronske mreže, learning rate (stopa učenja) i gamma (diskontni faktor), koji su preneseni iz *TrainAim* i *TrainMovement* klasi. Nadalje se definira varijabla *optimizator* koji će se koristiti za optimizaciju učenja i varijabla *criterion* koja definira kriterij koji mjeri srednju kvadratnu pogrešku između elemenata ulaza i elemenata izlaza. U ovom slučaju se koristi Adamov optimizator, koji je jedan od najboljih optimizatora te efikasnije postiže dobre rezultate.

```
class QTrainer:
    def __init__(self, model, lr, gamma):
        self.model = model
        self.lr = lr
        self.gamma = gamma
        self.optimizer = optim.Adam(model.parameters(), lr=self.lr)
        self.criterion = nn.MSELoss()
```

Slika 20: Isječak inicijalizacije QTrainer klase
Izvor: Autor

U klasi QTrainer se definira funkcija *TrainStep()* koja je glavna funkcija u kojoj se odvijaju sve operacije za treniranje neuronske mreže. Kod prizivanja funkciji se moraju priložiti trenutno stanje, izvedena akcija, ukupna nagrada, novo stanje i status epizode, koji se moraju preurediti kako bi se mogli koristiti u neuronskoj mreži tako da se podaci pretvore u PyTorchove tenzore. Nakon pretvorbe varijabli u tenzore, isti se moraju urediti tako da svaka varijabla bude jedan niz koristeći PyTorchovu funkciju *unsqueeze()*.

```
def TrainStep(self, state, action, reward, newState, done):
    state = torch.tensor(state, dtype=torch.float)
    newState = torch.tensor(newState, dtype=torch.float)
    action = torch.tensor(action, dtype=torch.float)
    reward = torch.tensor(reward, dtype=torch.float)

    if len(state.shape) == 1:
        state = torch.unsqueeze(state, 0)
        newState = torch.unsqueeze(newState, 0)
        action = torch.unsqueeze(action, 0)
        reward = torch.unsqueeze(reward, 0)
        done = (done, )

    #Predict Q of current action
    prediction = self.model.forward(state)
    targets = prediction.clone()

    for index in range(len(state)):
        newQ = reward[index]

        if(not done[index]):
            newQ = reward[index] + self.gamma *
torch.max(self.model.forward(newState[index]))

         targets[index][torch.argmax(action[index]).item()] = newQ

    #Optimize
    self.optimizer.zero_grad()
    loss = self.criterion(targets, predictions)
    loss.backward()

    self.optimizer.step()
```

Slika 21: Isječak funkcije TrainStep()
Izvor: Autor

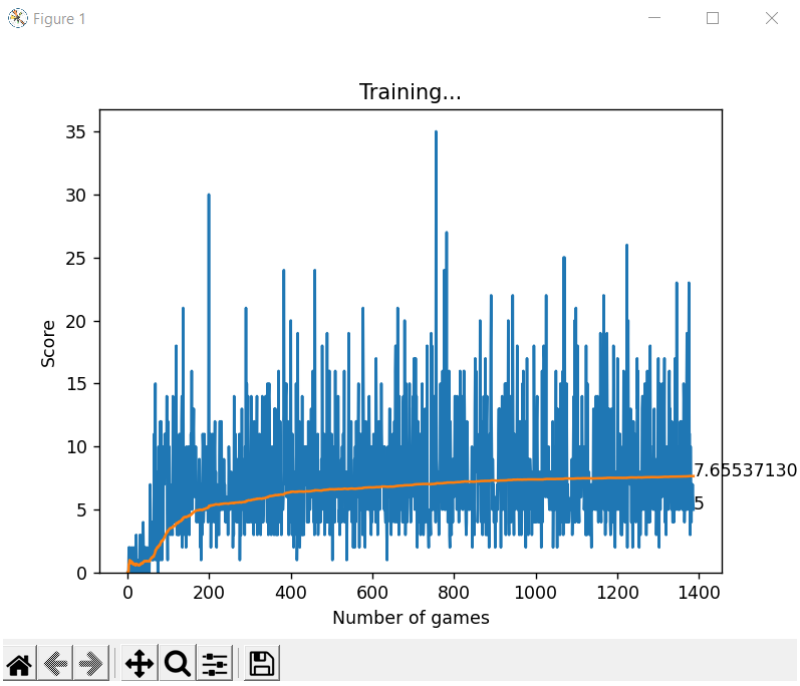
Nakon uređivanja podataka, dolazi samo predviđanje, prvo se uzima predviđena Q-Vrijednost te se sprema u varijablu *prediction*. Ta se varijabla kopira u varijablu *targets* u koju će se postaviti predviđene Q-Vrijednosti trenutne akcije na budućem stanju.

Nadalje se provjerava je li epizoda završena, ako je onda se jednostavno uzima cijela nagrada kao buduća Q-Vrijednost, ako ne onda se izračunava buduća Q-Vrijednost primjenom *Bellmanove jednadžbe*. Ta buduća Q-Vrijednost se uvrštava u *targets* varijablu pod odgovarajućim indeksom.

Optimizacija se vrši nakon što se *targets* i *predictions* varijable provuku kroz kriterij srednje kvadratne pogreške.

4.3. Rezultat

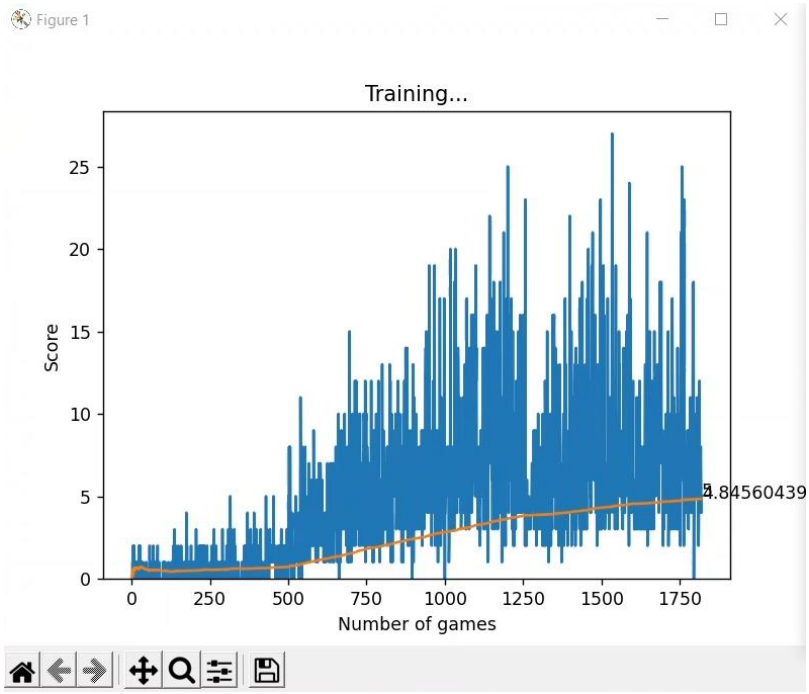
Rezultat projekta ovog diplomskog rada je temeljen na 2 sesije treniranja modela neuronske mreže, prva sesija koja je trajala oko 20 sati (nije bila snimljena) i druga sesija koja je trajala oko 48 sati te je snimak (ubrzan 25 puta kako bi se smanjio obujam datoteke) dostupan na sljedećoj poveznici: https://youtu.be/HbZ1wlpTmqU



Slika 22: Isječak grafa prve sesije treniranja
Izvor: Autor

Na grafu su prikazane sljedeće stavke: na x osi se nalazi broj odigranih rundi, tj. epizodi, na y osi se nalazi broj skupljenih bodova u rundi. Plave linije prikazuju trenutan broj bodova y skupljenih u rundi x, dok narančasta linija prikazuje prosječan broj bodova y koji su bili odigrani do runde x.

U prvoj sesiji maksimalni broj bodova je bio 34 nakon otprilike 750 rundi. Agent je uspio do tog broja bodova doći na način da si je trup premjestio na desnu granicu, tj. otprilike negdje u poziciji gdje je x ≈ 800 piksela, a y ≈ 300, dok je relativno preciznim ciljanjem uspio uništiti neprijatelje dovoljno brzo prije nego što mu se približe. Agent je saznao da se neprijatelji najmanje kreću oko te pozicije te je srećom uspio skupiti toliko bodova u toj rundi. Općenito, prosječan broj bodova skupljenih u jednoj rundi na taj način rada je 7.66, što je poprilično loše. Problem je nastao kada agent nije znao gdje treba ići kada se neprijatelj želi približiti te je agent stoga samo čekao na mjestu.



Slika 23: Isječak grafa druge sesije treniranja
Izvor: Autor

U drugoj sesiji maksimalni broj bodova je bio 28 nakon otprilike 1500 rundi. Agent je uspio do tog broja bodova doći na način da si je trup držao blizu svih 4 granica, dok je relativno preciznim ciljanjem uspio uništiti neprijatelje dovoljno brzo prije nego što mu se približe. Agent je saznao da se neprijatelji najmanje kreću oko granica te kad su mu se neprijatelji približili, pokušao se je izmaknuti na neku drugu granicu. Općenito, prosječan broj bodova skupljenih u jednoj rundi na taj način rada je 4.85, što je poprilično loše. Problem u ovoj sesiji je što se agent nije mogao dovoljno brzo izmaknuti neprijateljima i što nekad ne bi ni probao napraviti.

Iako je druga sesija bila 2 puta duža, ona nije bila bolja od prve zbog promijenjenih parametara. Postojala je velika razlika u vremenu sesija zbog promijenjene varijable *batchSize,* koja je bila 10.000 u prvoj sesiji, a u drugoj 100.000, što je znatno usporilo proces treniranja nakon svake epizode (gdje je na kraju trebalo gotovo 10 do 15 sekundi kako bi prošao proces dubokog treninga). Također se između dvije sesije promijenio kompromis istraživanja i iskorištavanja, u kojem bi u prvoj sesiji na početku treniranja varijabla *epsilon* iznosila 60%, a u drugoj sesiji 100%. Isto tako vezano za kompromis istraživanja i iskorištavanja, varijabli *epsilon* puno duže trebalo za smanjivanje na 0% (tek nakon 1000 odigranih rudni bi varijabla bila napokon 0%).

5. Rasprava

5.1. Prednosti podržanog učenja

Podržano učenje za razliku od ostalih pristupa strojnog učenja se može koristiti kako bi se riješili kompleksni problemi koji se ne mogu riješiti konvencionalnim rješenjima. Podržano učenje razvija razne tehnike rješavanja problema koje moraju biti orijentirane na ostvarivanje dugoročnih ciljeva. Zbog toga se može reći da je podržano učenje jako blizu učenja kod ljudi.

Također jedna od prednosti podržanog učenja je u tome što može popraviti greške koje se pojave u procesu treniranja, a i kad se te greške poprave, one se više ne ponavljaju. To je sve zahvaljujući mogućnosti učenja iz okoline, za razliku od ostalih pristupa strojnog učenja, gdje se učenje odvija isključivo iz nekog postojećeg skupa podataka.

5.2. Nedostaci podržanog učenja

Jedan od glavnih nedostataka podržanog učenja je u „gladi“ za podatcima. Podržanom učenju je potrebna ogromna količina podataka te je za to potrebno puno računanja za što je potrebno imati dovoljno dobro računalo sa dobrim procesorom. Zato se često ovaj pristup koristi u video igrama, jer video igre imaju mogućnost ponovnog igranja, što znači na podržano učenje može dobiti velike količine podataka. Također jedan od nedostataka je da podržanom učenju treba jako puno vremena kako bi se došlo do zadovoljavajućih rezultata (ovisno o kompleksnosti okoline).

5.3. Primjenjivost podržanog učenja

Glavna polja u kojima se podržano učenje može primijeniti su:

- Video igre
- Auto industrija
- Robotika

U video igrama se podržano učenje koristi kako bi se došlo do najbolje ili najoptimalnije akcije u igri umjesto ili protiv igrača, npr. mogućnost igranja ljudskog igrača protiv umjetne inteligencije učena podržanim učenjem u šahu.

U auto industriji se podržano učenje može koristiti kod izrade autonomne vožnje ili autonomnog parkiranja, u kojem se prvo treba napraviti simulacijski model u obliku video igre gdje se može trenirati umjetna inteligencija te kasnije primijeniti u stvarnom svijetu.

U robotici se može podržano učenje koristiti kako bi se istrenirali roboti kako bi obavljali određene zadatke, npr. mehaničke robotske ruke u proizvodnji moraju znati koji objekt moraju uhvatiti i gdje ga kasnije staviti.

6. Zaključak

U ovom diplomskom radu je istraživan proces izrade umjetne inteligencije podržanim strojnim učenjem u Python programskom jeziku. Izrađena umjetna inteligencija se primijenila na primjeru video igre. U radu su se spomenuli osnovni pristupi strojnog učenja koji uključuju nadzirano, nenadzirano i podržano strojno učenje. Objasnili su se osnovni elementi podržanog učenja, koji uključuju agenta, okolinu, stanje, akciju, politiku, epizodu i nagradu ili kaznu, te najkorištenije algoritme podržanog učenja, poput Q-Učenja i Dubokog Q-Učenja. Pri izradi projekta diplomskog rada korištena je PyTorch biblioteka, koja je uvelike olakšala proces izrade neuronskih mreža.

Promatranjem rezultata projekta se može zaključiti kako se mogu ostvariti određena poboljšanja. Jedno od tih poboljšanja bi bilo poboljšanje pomicanja trupa agenta, jer prema dobivenim rezultatima, trup agenta je jedan od glavnih razloga zašto agent ima mali broj bodova, zato što agent previše ovisi o prosječnoj poziciji neprijatelja, a ne o trenutnoj poziciji neprijatelja. To se može riješiti promjenom nagradne funkcije (dodavanjem više načina dobivanja nagrade), dodavanjem više informacija o okolini (npr. vektor kretanja neprijatelja, ili trenutni kut neprijatelja) ili pak pojednostavljenjem informacija o okolini.

7. Literatura

1. Bhatt S. (2018). 5 Things You Need to Know about Reinforcement Learning. Dostupno na: https://www.kdnuggets.com/2018/03/5-things-reinforcement-learning.html [pristupljeno 15. kolovoza 2022.]
2. Hurwitz J., Kirsch D. (2018). Machine Learning for dummies. [Online] IBM Limited Edition. Dostupno na: https://www.ibm.com/downloads/cas/GB8ZMQZ3 [pristupljeno: 13. srpnja 2022.]
3. Karunakaran D. (2020). Key concepts in Reinforcement Learning. Dostupno na: https://medium.com/intro-to-artificial-intelligence/key-concepts-in-reinforcement-learning-2af715dfbfa [pristupljeno 21. srpnja 2022.].
4. MathWorks. (2022). What Is Reinforcement Learning?. Dostupno na: https://se.mathworks.com/discovery/reinforcement-learning.html [pristupljeno 16. kolovoza 2022.].
5. Medium. (2019). (Deep) Q-learning, Part1: basic introduction and implementation. Dostupno na: https://medium.com/@qempsil0914/zero-to-one-deep-q-learning-part1-basic-introduction-and-implementation-bb7602b55a2c [pristupljeno 15. kolovoza 2022.]
6. Pygame. (2020). About – wiki. Dostupno na: https://www.pygame.org/wiki/about [pristupljeno 10. srpnja 2022.]
7. Pygame GUI. (2020). Quick Start Guide. Dostupno na: https://pygame-gui.readthedocs.io/en/latest/quick_start.html [pristupljeno 10. srpnja 2022.]
8. PyTorch. (2016). PyTorch Features. Dostupno na: https://pytorch.org/features/ [pristupljeno 10. srpnja 2022.]
9. Yegulalp, S. (2017). Facebook brings GPU-powered machine learning to Python. Dostupno na: https://www.infoworld.com/article/3159120/facebook-brings-gpu-powered-machine-learning-to-python.html [pristupljeno 11. kolovoza 2022.].
10. Zychlinski, S. (2019). Qrash Course: Reinforcement Learning 101 & Deep Q Networks in 10 Minutes. Dostupno na: https://towardsdatascience.com/qrash-course-deep-q-networks-from-the-ground-up-1bbda41d3677 [pristupljeno 15. srpnja 2022.].
11. Zychlinski, S. (2019). The Complete Reinforcement Learning Dictionary. Dostupno na: https://towardsdatascience.com/qrash-course-deep-q-networks-from-the-ground-up-1bbda41d3677 [pristupljeno 15. srpnja 2022.].

Popis slika

Slika 1: Vrste strojnog učenja 2

Slika 2: Ciklus podržanog učenja 3

Slika 3: Razlika između Q-Učenja i Dubokog Q-Učenja 6

Slika 4: Isječak video igre i entiteti u igri 12

Slika 5: Isječak koda inicijalizacije Controller-a 14

Slika 6: Isječak koda glavne petlje 16

Slika 7: Isječak koda klasa TrainMovement i TrainAim 18

Slika 8: Isječak koda funkcije Train, u klasi TrainMovement 19

Slika 9: Isječci koda funkcije GetState u TrainMovement i TrainAim klasama 21

Slika 10: Isječak koda funkcije GetAction() u TrainMovement() klasi 22

Slika 11: Isječak koda funkcije PerformMove() u klasi TrainMovement i TrainAim 23

Slika 12: Isječak funkcije AssignRewards() u klasi TrainMovement i TrainAim 24

Slika 13: Isječak funkcije Remember() 25

Slika 14: Isječak funkcije TrainShortMemory() 25

Slika 15: Isječak funkcije Save() i Load() u klasi TrainMovement 25

Slika 16: Isječak ResetGame() funkcijegdje se poziva funkcija TrainLongMemory() 26

Slika 17: Isječak funkcije TrainLongMemory() 27

Slika 18: Isječak inicijalizacije klase LinearQNN 27

Slika 19: Isječak forward() funkcije 28

Slika 20: Isječak inicijalizacije QTrainer klase 28

Slika 21: Isječak funkcije TrainStep() 29

Slika 22: Isječak grafa prve sesije treniranja 30

Slika 23: Isječak grafa druge sesije treniranja 31