IMPLEMENTACIJA IGRE I IZRADA AGENTA POMOĆU STROJNOG UČENJA

Baničević, Zdravko

Master's thesis / Specijalistički diplomski stručni

2021

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Split / Sveučilište u Splitu**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:228:857289

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-03-29**

Repository / Repozitorij:

Repository of University Department of Professional Studies

zir.nsk.hr

UNIVERSITY OF SPLIT

dabar
DIGITALNI AKADEMSKI ARHIVI I REPOZITORIJI

SVEUČILIŠTE U SPLITU

SVEUČILIŠNI ODJEL ZA STRUČNE STUDIJE

Specijalistički diplomski stručni studij Informacijske tehnologije

ZDRAVKO BANIČEVIĆ

Z A V R Š N I R A D

IMPLEMENTACIJA IGRE I IZRADA AGENTA POMOĆU STROJNOG UČENJA

Split, srpanj 2021.

SVEUČILIŠTE U SPLITU

SVEUČILIŠNI ODJEL ZA STRUČNE STUDIJE

Specijalistički diplomski stručni studij Informacijske tehnologije

Predmet: Primijenjena umjetna inteligencija

Z A V R Š N I R A D

Kandidat: Zdravko Baničević

Naslov rada: Implementacija igre i izrada agenta pomoću strojnog učenja

Mentor: Toma Rončević

Split, srpanj 2021.

Sadržaj

Sažetak **1**

1 Uvod **2**

2 Tehnologije **3**

2.1 Integrirano razvojno okruženje 3

2.2 Sustav za upravljanje verzijama 4

2.3 PyTorch 4

2.4 Tensorboard 6

2.5 Računalni oblak 9

3 Pojačano učenje **12**

3.1 Umjetna inteligencija 12

3.2 Strojno učenje 13

3.3 Povijest pojačanog učenja 14

3.4 Elementi pojačanog učenja 15

3.5 Konačan Markov proces odlučivanja 15

3.6 Nagrada 16

3.7 Politike i funkcije vrijednosti 17

3.7.1 Optimalne politike i funkcije vrijednosti 18

3.8 Bellmanova jednadžba optimalnosti 18

3.9 Q-Učenje 18

3.9.1 Epsilon pohlepna strategija 19

3.9.2 Ažuriranje q-vrijednosti 20

3.10 Neuronska mreža 22

3.10.1 Učenje neuronske mreže 22

3.10.2 Duboka Q-mreža 23

4 Implementacija **24**

4.1 Implementacija igre 24

4.1.1 Poker 24

4.1.2 Pravila igre 25

4.1.3 Program 27

4.1.4 Protivnici 33
4.2 Implementacija agenta 34
4.3 Učenje 38
4.4 Rezultati 38
4.5 Napredak 39

5 Zaključak **42**

Literatura **43**

6 Dodatak **44**

Sažetak

Cilj ovog rada je implementirati pametnog agenta koji pomoću pojačanog učenja nauči igrati igru Poker. Poker je popularna igra karata koja ima razne varijacije pravila igranja. U ovom radu je implementiran Limit Texas Hold'em turnir u Python programskom jeziku verzije 3.7.10. Za izradu agenta korišteni su razvojni okviri (eng. *Frameworks*) PyTorch 1.3.1 za sastavljanje i rad s neuronskom mrežom te Tensorboard 2.3.0 za vizualizaciju napretka i promjene neuronske mreže kroz učenje.

Ključne riječi: Pojačano učenje, Poker, PyTorch, Tensorboard, Umjetna inteligencija

Summary

Poker game implementation and agent creation using machine learning

The goal of this work is implementing an intelligent agent that through reinforcement learning is able to learn playing the game of Poker. Poker is a popular card game, that has many variations in game rules. This paper describes the implementation of Limit Texas Hold'em tournament in the programming language Python version 3.7.10. Frameworks used for building the agent are PyTorch 1.3.1 for assembling and manipulation of neural networks and Tensorboard 2.3.0 for tracking progress and value changes inside the network.

Keywords: Artificial intelligence, Poker, PyTorch, Reinforcement learning, Tensorboard

1 Uvod

U ovom se radu opisuje izrada agenta koji će pomoću metoda strojnog učenja vremenom poboljšavati svoju igru. U svijetu strojnog učenja igre su uvijek imale ključnu ulogu već od samih početaka kada se postavljao temelj umjetne inteligencije. Ključni dio agenta je neuronska mreža koja prima stanje igre kao ulazni parametar u mrežu, a na izlazu svakoj akciji dodijeli vrijednost, te akcija s najvišom vrijednosti se smatra potezom prikladnim za trenutno stanje. Za izradu i upravljanje neuronske mreže koristio se PyTorch, kojeg je bilo potrebno dodatno instalirati. Također se u ovom radu opisuje izradu igre karata Limit Texas Hold'em turnir u Python programskom jeziku. Za izradu igre se nije koristila niti jedna biblioteka koja se ne nalazi u Pythonovom standardnom skupu biblioteka. Poker je igra koja spada pod nepotpuno informirane igre. To znači da igrač koji sudjeluje u igri ne zna sveukupno stanje o igri, tj., nema informaciju o protivničkim kartama u ruci. Osim toga poker je specifičan jer uključuje mogućnost da se i s lošom rukom pobjedi krug, na način praćenja i povisivanja uloga tako da se iskazuje lažni dojam protivnicima o posjedovanju dobrih karata, tzv. *blef*.

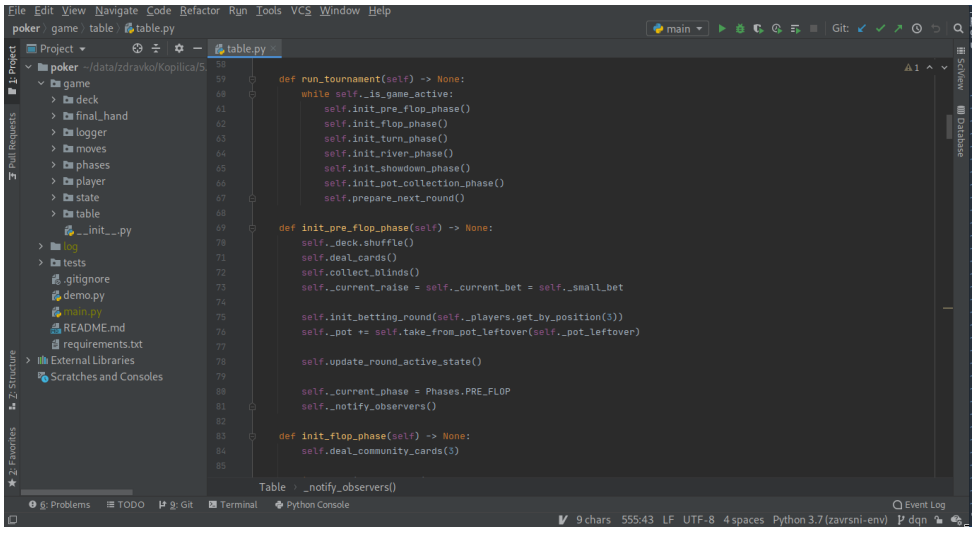
Glavni motiv ovog rada je demonstrirati da algoritam iz područja strojnog učenja može biti u stanju uspješno savladati jednu složenu igru, koja osim toga uključuje i određenu količinu nasumičnosti kao što je poker. Ovaj se problem pokušava riješiti pomoću neuronske mreže koja je trenutno najpopularniji pristup umjetne inteligencije.

2 Tehnologije

Za implementaciju rada izabran je Python programski jezik, koji je u posljednjih godina dosegnuo visoku popularnost i koristi se u gotovo svim vrstama aplikacija. Interpretirani je jezik visoke razine opće namjene te ima filozofiju dizajna koji naglašava čitljivost kôda. Podržava više paradigme programiranja, uključujući objektno orijentirane, imperativne, funkcionalne i proceduralne. Programski jezik koristi dinamičke tipove (eng. *dynamically typed*), što znači da se varijablama ne određuje tip, ali ako se želi postoji mogućnost izričito ga naglasiti. Posjeduje sveobuhvatnu biblioteku, koja između ostalog pokriva i područje umjetne inteligencije. Korištenjem se dosta olakšavaju obrade i pripreme podataka za inteligentne agente, te izradu istih. Također, kako se ne bi sukobili paketi iz ovog projekta s paketima iz drugog ili iz sustava, stvorilo se posebno virtualno okruženje samo za ovaj projekt. Pythonov modul koji to omogućuje zove se *virtualenv*.

2.1 Integrirano razvojno okruženje

Kôd je pisan u razvojnom okruženju PyCharm koji je jedan od mnogih okruženja Češke tvrtke JetBrains. Glavne značajke su analiza kôda, grafički program za otklanjanje pogrešaka, integrirani tester jedinica te integracija s kontrolnim sustavom za verzioniranje. Izgled PyCharm integriranog razvojnog okruženja prikazan je na slici 1.



Slika 1: Integrirano razvojno okruženje PyCharm

2.2 Sustav za upravljanje verzijama

Za praćenje promjena u izvornom kôdu, korišten je git. Slobodan je i otvorenog kôda. Prilagođen je za rad programera, bilo to da na istom projektu radi više ljudi, ili samo jedna osoba. Nije nužno da ga se koristi samo za praćenje promjene u izvornom kôdu, nego za bilo koju vrstu datoteka. Osnovne značajke su mu brzina, integracija podataka, te podrška za distribuirane, ne linearne tijekove rada. Također su se koristile poslužiteljske usluge GitHuba, na kojemu se nalazi izvorni kôd ovoga rada. Putanja za pristup kôdu je `https://github.com/zb46392/poker`. GitHub drži najveću količinu izvornog kôda na svijetu, te osim osnovnih funkcionalnosti gita, pruža dodatne usluge za upravljanje izvornim kôdom. Među dodatnim značajkama su i kontrola pristupa i prava, kolaboracijski alati za praćenje pogrešaka (eng. *bug tracking*), upravljanje zadataka, pisanje dokumentacije itd.

2.3 PyTorch

Jedna od biblioteka korištena izvan Pythonove standardne biblioteke je PyTorch 1.3.1. Biblioteka je otvorenog kôda razvijena u Facebookovom laboratoriju za istraživanje umjetne inteligencije (eng. *Facebook AI Research lab - FAIR*). Bazira se na Torch biblioteci koja se koristi u Lua skriptnom programskom jeziku. Torch se više aktivno ne razvija, međutim Pythonova verzija PyTorch je u aktivnom razvoju. Primarne funkcionalnosti koje PyTorch nudi su računalne operacije s tenzorima, kao i NumPy biblioteka. Kompatibilan je s NumPy bibliotekom, znači NumPy prepoznaje i prihvaća PyTorchov tenzor i obrnuto. Dodatne funkcionalnosti koje PyTorch nudi su duboke neuronske mreže građene na sustavu automatske diferencijacije temeljen na vrpci (eng. *tape-based autograd system*). Dodatno ima mogućnost izvršavati operacije na Cuda sposobnoj grafičkoj procesorskoj jedinici (*Nvidia grafičke kartice*) što znatno skrati vrijeme treniranja neuronske mreže. Cuda je aplikacijsko programsko sučelje (eng. *application programming interface - API*) koje omogućuje programerima izvršavanje instrukcije na grafičkoj kartici. PyTorch jako pojednostavljuje slaganje, modificiranje i treniranje neuronske mreže, te su sve osnovne operacije uključene u biblioteci. Sustav automatske diferencijacije u pozadini čuva graf neuronske mreže s kojim vrlo jednostavno izvršava povratno propagiranje pogreške (eng. *backpropagation*), te koristi tehniku autodiferencijacija u obrnutom načinu (eng. *reverse-mode auto-differentiation*). Naslanja se na nekoliko znanstvenih radova. Međutim PyTorchova implementacija se najbrže izvršava. Instalirati ga se može kompajliranjem (eng. *compiling*) izvornog kôda ili skidanjem već gotove binarne datoteke

spremne za izvršavanje. Najjednostavniji način je putem pip naredbe: `pip install pytorch` ili ako se koristi Anacondu `conda install pytorch -c pytorch`. U kôdu ga se uključuje sa `import torch`, te moduli koji se često koriste uključuju se sa `import torch.nn as nn` i `import torch.nn.functional as F`. Klasa koja implementira neuronsku mrežu trebala bi biti naslijeđena od klase `torch.nn.Module` te je potrebno implementirati funkciju `forward`, koja se poziva u trenutku kada se prosljeđuje neki ulaz u mrežu. Međutim ovu se funkciju eksplicitno ne poziva, nego se izvršava u pozadini kada se objekt poziva kao funkciju i kroz argument se proslijedi ulaz. Primjer je dan u ispisu 1.

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class MyNet(nn.Module):
   def __init__(self, input_size: int, output_size: int) -> None:
      super().__init__()
      self._fc1 = nn.Linear(in_features=input_size, out_features=64)
      self._fc2 = nn.Linear(in_features=64, out_features=32)
      self._out = nn.Linear(in_features=32, out_features=output_size)

   def forward(self, in_tensor: torch.Tensor) -> torch.Tensor:
      t = in_tensor
      t = F.relu(self._fc1(t))
      t = F.relu(self._fc2(t))
      t = self._out(t)
      return t


net = MyNet(3, 2)

X = torch.rand(3)
y = net(X)
```

Ispis 1: Nasljeđivanje PyTorch torch.nn.Module

Isto se može složiti mrežu sa `torch.nn.Sequental` modulom i u tom slučaju nije potrebno implementirati forward funkciju, nego se podrazumijeva da je izlaz prethodnog sloja ulaz u sljedeći. Primjer korištenja `torch.nn.Sequential` pokazan je ispisom 2

```
import torch
import torch.nn as nn

net = nn.Sequential()
net.add_module('fc_0', nn.Linear(in_features=10, out_features=64)
net.add_module('relu_0', nn.ReLU())
net.add_module('fc_1' nn.Linear(in_features=64, out_features=32)
net.add_module('relu_1', nn.ReLU())
net.add_module('fc_2', nn.Linear(in_features=32, out_features=2)
net.add_module('softmax_2', nn.Softmax(dim=1))

X = torch.rand(10)
y = net(X)
```

Ispis 2: Korištenje PyTorch torch.nn.Sequential

2.4 Tensorboard

Biblioteka je razvijena od tvrtke Google. Pruža alate vizualizacije potrebne za eksperimentiranje sa strojnim učenjem. Omogućuje:

- Praćenje i vizualizacija mjernih podataka kao što su gubitak i točnost
- Vizualizaciju grafa modela i njegove slojeve
- Prikaz histograma o vrijednostima u pojedinim slojevima koji se mijenjaju tijekom učenja
- Projektiranje ugrađivanja u prostor niže dimenzije
- Prikazivanje slika, teksta i audio podataka

Dodatno je vrlo jednostavno rezultate objaviti na stranici `https://tensorboard.dev/`, gdje se detaljno opisuje postupak.

PyTorcheva klasa `SummaryWriter` omogućuje zapisivanje rezultata u Tensorboard i uključuje se sa `from torch.utils.tensorboard import SummaryWriter`. Zapisivanje informacije o modelu i slojevima se odvija pozivanjem funkcije `add_graph`, dok se s funkcijom `add_scalar` dodaju vrijednosti gubitka i/ili točnosti a sa `add_histogram` se dodaju vrijednosti

elemenata pojedinih slojeva. Sveukupno korištenje ove biblioteke u ovom radu nalazi se u ispisu 3

```
def _init_summary_writer(self) -> None:
    dummy_input = self._generate_dummy_input()
    log_dir_path = self._generate_log_dir_path()

    self._summary_writer = SummaryWriter(log_dir=str(log_dir_path))
    self._summary_writer.add_graph(self._model, dummy_input)


def monitor(self) -> None:
    progress = self._progress
    self._summary_writer.add_scalar('Progress', progress, self._step)

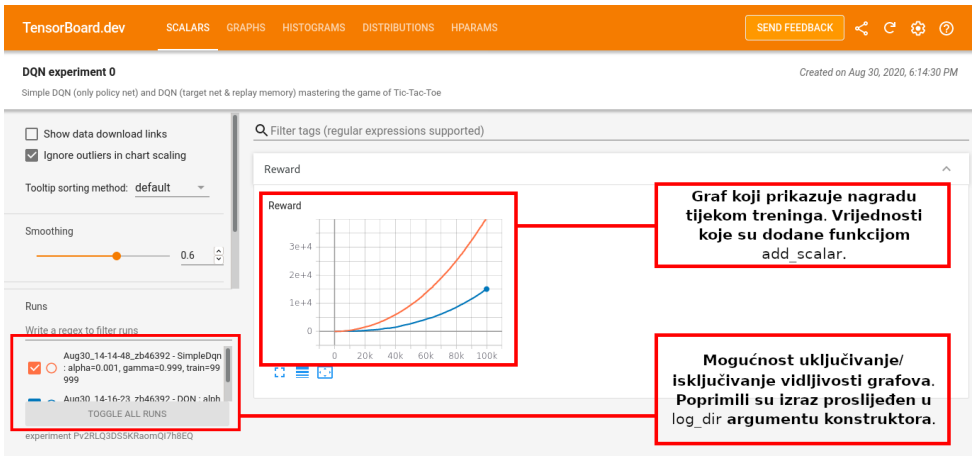
    for name, param in self._model.named_parameters():
        self._summary_writer.add_histogram(name, param, self._step)
        self._summary_writer.add_histogram(f'{name}_grad', param.grad,
                                           self._step)

    self._step += 1


def close(self) -> None:
    self._summary_writer.close()
```

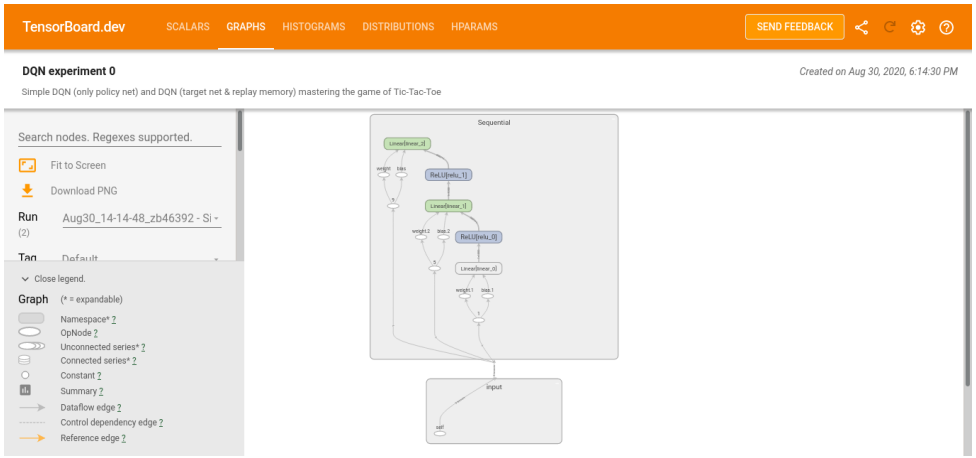
Ispis 3: Korištenje Pytorch SummaryWriter

Vidljivo je u ispisu 3 da su funkcije `_init_summary_writer`, `monitor` i `close` dio neke klase, koja je u potpunosti dana ispisom 14 u dodatku. Ta klasa je zadužena za zapisivanje napredka agenta tijekom učenja na TensorBoard. U konstruktoru od `SummaryWriter` je proslijeđen argument `log_dir`, koji predstavlja putanju datotečnog sustava do direktorija u kojemu se zapisuju podaci napredka. Ime direktorija bi trebalo na jedinstven način opisati trening, kako bi se u TensorBoardu razlikovalo više treninga koji se međusobno uspoređivaju. U ovom slučaju je naziv direktorija sastavljen od vremena izvođenja te osnovnih hiperparametara za taj trening. Na slici 2 je prikazan primjer Tensorborda, uz opise bitnih dijelova, u kojem su se pratili rezultati treniranja dvaju modela, te je vidljivo da naziv svakog grafa odgovara izrazu koji je proslijeđen u konstruktor.



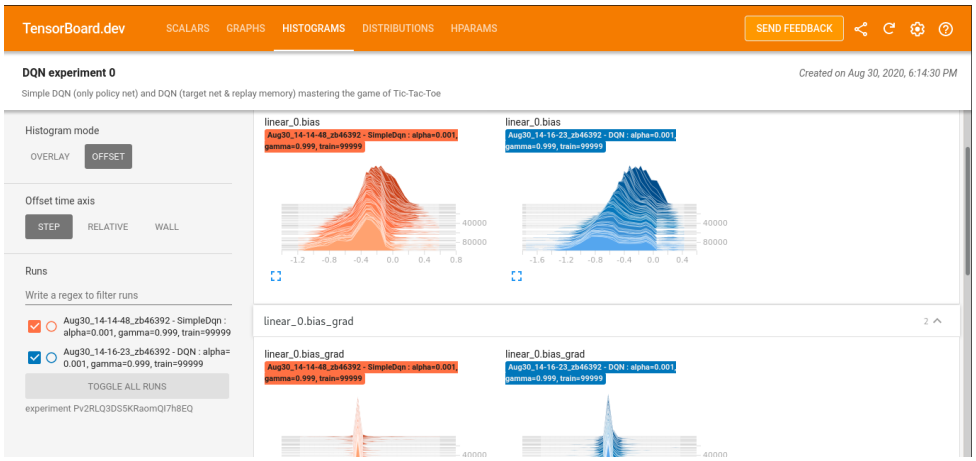
Slika 2: Izgled Tensorboard vizualizacija mjernih podataka.

Nakon poziva konstruktora poziva se funkcija `add_graph` koja zabilježava arhitekturu modela te podatke koje prima. Izgled tog područja prikazan je slikom 3. Dalje se tijekom učenja agenta zabilježava zbroj svih nagrada u nekom trenutku s funkcijom `add_scalar`, koji prima naziv vrijednosti, te sama vrijednost i korak u kojemu je postignuta ta vrijednost. Graf nagrade se nalazi na početnom području Tensorboarda vidljiv na slici 2.

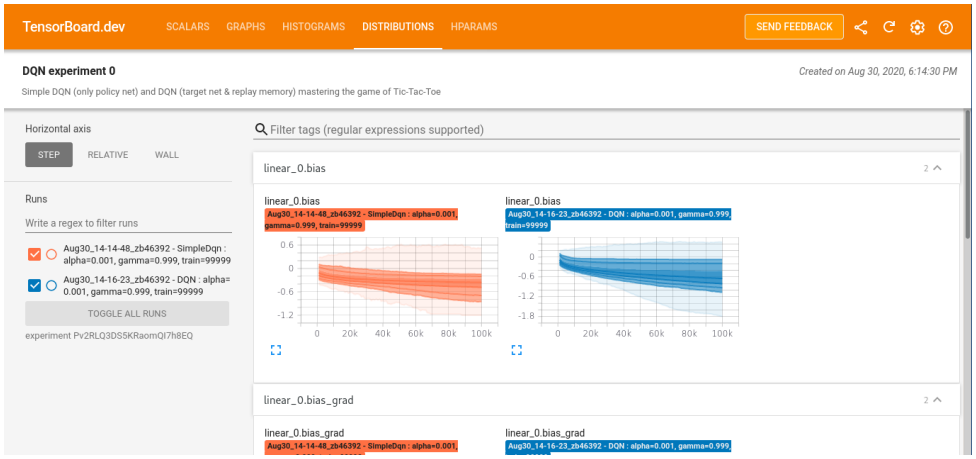


Slika 3: Tensorboard graf modela i njegovi slojevi.

Dodatno se tijekom učenja zabilježavaju i histogram parametra kako se mijenja, a to je moguće s funkcijom `add_histogram`. Parametri koje ta funkcija prima su slični kako i kod `add_scalar`, samo što se ne radi o jednoj vrijednosti nego o skupu vrijednosti. Prikaz histograma i njihova distribucija u Tensorboardu su prikazani slikama 4 i 5.



Slika 4: Tensorboard histogrami o vrijednostima u pojedinim slojevima.

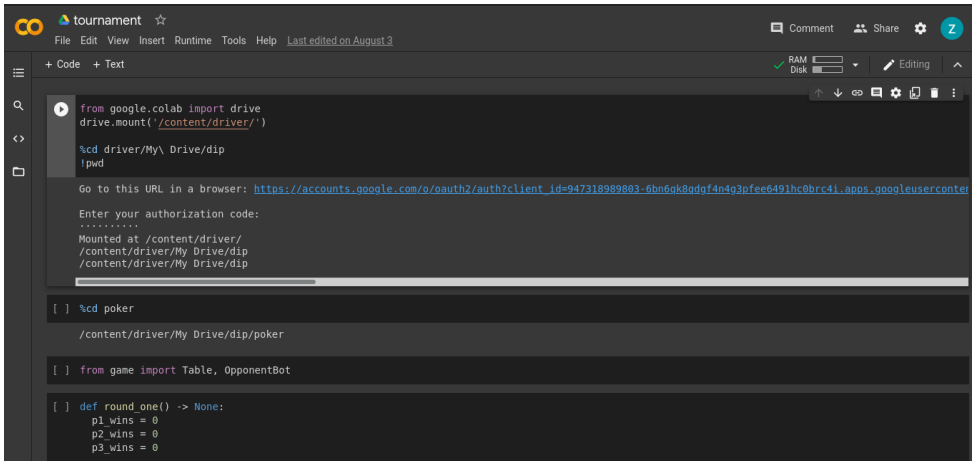


Slika 5: Tensorboard distribucije o vrijednostima u pojedinim slojevima.

2.5 Računalni oblak

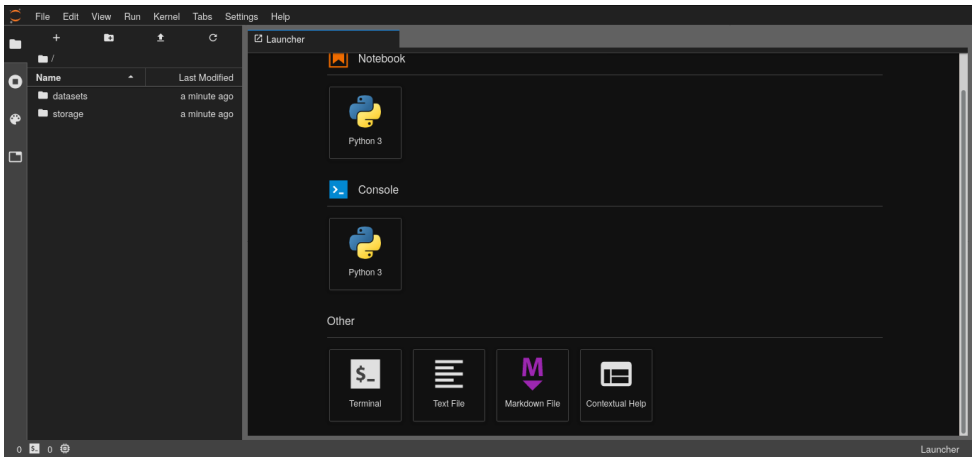
Učenje je dugotrajan proces, čak i sa suvremenim osobnim računalima. Tako da su se koristile dodatne računalne snage u oblaku, što može ubrzati proces učenja. Osim toga, što je veći broj računalne snage na raspolaganju, mogu i više različitih algoritama istovremeno učiti. Postoje razni ponuđači koji nude mogućnost učenja u oblaku, međutim ovisno o snazi i prostoru pojedinog komponenta računala, plaća se određena cijena. Pronađene su mogućnosti korištenja računalnih oblaka koji se ne plaćaju, ali imaju druga ograničenja. Bez obzira na njih, u svrhu ovoga rada, bili su sasvim dovoljni za korištenje. Jedan od korištenih je Google Colaboratory, koji je izrađen od tvrtke Google. Radno sučelje jako liči na Jupyter notebook. Jupyter notebook je alat za pisanje programskog kôda gdje se kôd piše u ćelije. Svaka se ćelija može posebno pokrenuti, što omogućuje brze promjene bez da se cijeli kôd iz početka pokreće. Dodatno se

u ćelije mogu i pisati Markdown oznake uz pomoć kojih se izvršavane ćelije ispravno formatiraju. Ovo omogućuje da se programski kôd i dokumentacija može nalaziti na istemu mjestu. Za korištenje Google Colaboratory alata, kako i za korištenje bilo kojeg Google alata, potrebno je izraditi Google korisnički račun. Izgled Google Colaboratory radnog sučelja prikazan je na slici 6.



Slika 6: Radno okruženje Google Colaboratory.

Ograničenja u korištenju bez plaćanja, su mogućnost korištenja grafičke procesorske jedinice samo na jedan dan i Google Colaboratory mora u web pregledniku biti aktivan. Nakon nekog vremena neaktivnosti svi se procesi prekidaju. Zapravo omogućuju izvršenje puno kratkih procesa, a ne jednog koji se vrti jako dugo. Druga usluga računalnog oblaka koja je korištena je Paperspace. Ovdje se nudi korištenje grafičke procesorske jedinice 6 sati bez plaćanja. Nakon isteka tog vremena, oblak se ruši bez mogućnosti nastavka korištenja istog. Može se novoga pokrenuti, međutim nisu uvijek svi resursi slobodni, ako ih drugi korisnici rezerviraju. Prvo se izabere koji će radni okviri biti unaprijed instalirani i onda se izabere model grafičke kartice. Nakon sastavljanja željenog oblaka podiže se JupyterLab radno okruženje vidljivo na slici 7.



Slika 7: Radno okruženje JupyterLab.

Može se primijetiti da u JupyterLab postoji mogućnost, osim otvaranja Jupyter Notebook dokumenata, otvoriti sučelje naredbenog retka (eng. *command line interface*). Korištenje sučelja naredbenog retka je dosta pojednostavnilo korištenje oblaka u odnosu na Jupyter notebook koji se u Paperspace nije ni koristio. Moguće je u Jupyter notebook izvršavati naredbe ljuske ako se na početak naredbe stavi uskličnik, npr. `!pwd` koja ispisuje potpunu putanju do trenutnog radnog direktorija. Za korištenje usluge kod Paperspace potrebno je izraditi korisnički račun, ili se može pak prijaviti s korisničkim računom od Google ili GitHub.

3 Pojačano učenje

Najistaknutije osobine pojačanog učenja su pokušaj i neuspjeh, te odgođena nagrada koju algoritam doživljava u okruženju. Algoritam bi trebao, do neke razine, imati osjećaj i informacije o okolini, te cilj ili ciljeve prema kojima teži, a odnose se na određeno stanje u okolini.

3.1 Umjetna inteligencija

Umjetna inteligencija je jako široko područje računalne znanosti. Bavi se proučavanjem i izradom algoritama koji nemaju eksplicitne naredbe kako se ponašati u nekom trenutku i/ili okolini, nego kroz određene procese sami uspiju donijeti zaključke i odluke. Umjetna inteligencija se dijeli na puno podgrana, koje se u prošlosti nisu smatrali toliko srodnim koliko danas. U današnje doba se iskazao potencijal umjetne inteligencije i sve više se ulaže u njezino istraživanje i razvoj.

Umjetna inteligencija se dijeli na više grana kao što su:

- Rasuđivanje i rješavanje problema: Gdje se pokušava implementirati razbijanje problema u logičke korake do rješenja i postepenim izvršavanjem tih koraka riješiti zadani problem. Ovakvi algoritmi se nisu iskazali korisnima u velikim problemima zbog kombinatorne eksplozije, znači rastom problema algoritam eksponencijalno usporava.

- Reprezentacija znanja: Ovo je glavno područje u istraživanju klasične umjetne inteligencije. Skupljaju se znanja od nekog područja i pohranjuju se u neku bazu u kojoj se istaknu pojmovi i veze između njih. Neke od stvari koje se nalaze u takvoj bazi su: predmeti, svojstva, vrste i odnosi između predmeta, situacija, događaja, stanja i vremena, uzroci i posljedice, znanje o znanju i još puno onih manje istraženih područja.

- Planiranje: Odnosi se na predviđanje nekih budućih događaja te donošenje određenih odluka koje utječu na ishod.

- Učenje: Predstavlja algoritme koji ispočetka jako loše rješavaju zadatak, međutim skupljanjem iskustva vremenom su sve bolji.

- Obrada prirodnog jezika: Omogućuje strojevima razumijevanje ljudskih jezika.

- Percepcija: Sposobnost pomoću određenih senzora primiti informacije o stvarnoj okolini te uspješno interpretirati te informacije u svoje zadaće.

- Kretanje i rukovanje: Upravljanje mehaničkih udova kako bi se riješio neki zadatak u stvarnoj okolini. Smatra se posebno kompleksnim, te obuhvaća paradoksni pojam (Moravec's paradox) u kojemu je teže implementirati radnju koju malo dijete može izvoditi bez ikakvih poteškoća, npr: primiti neki predmet u ruke i odložiti ga na određeni položaj, od radnje kojoj je odraslome čovjeku teško, npr: jako dobro odigrati partiju šaha. Izlazi iz činjenice da za razliku od šaha percepcija, kretanje i rukovanje su prirodnom selekcijom kroz milijune godina usađeni u čovjeka.

- Socijalna inteligencija: Uključuje poznavanje i prepoznavanje emocije i ljudskih međudjelovanja. Nekim algoritmima bi bilo vrlo korisno da uz pojmove teorije uključuje spoznaju ljudskih emotivnih stanja i motive za donijeti bolje odluke.

- Opća inteligencija: U prošlosti se pokušalo dizajnirati opće inteligentnog agenta koji pokriva široku ljudsku spoznaju. No odustalo se od toga zbog preogromne količine informacija koje su kombinirane iz raznih područja. Danas se razvija 'usku' umjetnu inteligenciju koja je usredotočena u jedno područje i smatra se da bi opća umjetna inteligencija trebala ujediniti hrpu tih usredotočenih područja.

3.2 Strojno učenje

U ovome završnom radu se razradilo područje iz učenja. I ovo se područje grana na više potpodručja. Među najpoznatijima su učenje s nadzorom (eng. *supervised learning*) u kojemu algoritam dobije skup podataka iz kojih treba donijet neke zaključke i skup rješenja, odnosno definirane zaključke koje bi trebao donijeti. Tako da kroz neko vrijeme koje je dodijeljeno za učenje, primi podatak, donese odluku i ovisno o tome koji je unaprijed definirani zaključak kojeg je trebao donijeti se prilagodi da u buduće donosi što ispravnije odluke. Druga grana se zove učenje bez nadzora (eng. *unsupervised learning*), gdje algoritam prima neki skup podataka i vremenom uspije pronaći razlike i sličnosti u podacima. Nema informacije o tome što podatak predstavlja ali zna podatke svrstati u svoje definirane kategorije. Izrađeni agent iz ovog projekta spada pod skupinom pojačanog učenja (eng. *reinforcement learning*). U ovoj grani algoritam djeluje u nekom okruženju pomoću određenih akcija, te u početku nema informacije o tome kako

akcije utječu na okruženje. Te vremenom mora sam otkriti koje akcije u kojem trenutku donose najveću nagradu.

3.3 Povijest pojačanog učenja

Što se tiče povijesti pojačanog učenja vrijedi spomenuti dvije glavne grane koje su tada postojale kao samostalne dok se danas isprepliću. Jedna od njih je nastala kao dio psihologije životinjskog učenja te se fokusirala na učenje metodom pokušaja i pogreške. Druga pak koristi vrijednosne funkcije i dinamičko programiranje te se bavi problemom optimalnog upravljanja i njegovog rješenja. Prva grana pojačanog učenja je zapravo i dovela do otkrića pojačanog učenja početkom 1980-tih godina te se može reći kako kao takva proizlazi iz nekih najranijih djela umjetne inteligencije, iako se sama ideja takvog načina učenja, metode pokušaja i pogreške, proteže unazad čak do 1850-tih godina. Edward Thorndike je bio prvi koji je uspio objasniti bit učenja metodom pokušaja i pogreške te ju je nazvao "Zakon učinka" (eng. *Law of effect*): "Od nekoliko odgovora danih na istu situaciju, oni koji su povezani ili pomno prate životinjsku volju, pod jednakim uvjetima, čvršće će biti povezani sa situacijom, tako da će, kad se ponovi, biti vjerojatnije da će se i oni ponoviti; oni koji su povezani ili pomno prate nelagodu životinje, pod jednakim uvjetima, imat će oslabljene veze s tom situacijom, pa će, kada se ponovi, biti manje vjerojatno da će se i ona ponoviti" [1].

Alan Turing je opisao sistem „zadovoljstvo – patnja" koji je funkcionirao u skladu sa Zakonom učinka. To je ujedno bila i jedna od najranijih ideja kako implementirati metodu pokušaja i učenja s umjetnom inteligencijom. Kod druge metode, pojam "optimalna kontrola" počeo se upotrebljavati za opisivanje problema dizajniranja regulatora koji minimalizira ili maksimizira mjeru ponašanja dinamičnog sustava tijekom vremena. Dinamičko programiranje smatra se jedinim izvedivim načinom rješavanja općenito stohastičkih problema optimalnog upravljanja. I dalje je učinkovitiji i više se primjenjuje od ijedne druge opće metode iako njezini računski zahtjevi eksponencijalno rastu s brojem varijabli stanja. Puno vremena je trebalo da bi se prepoznavale veze između optimalne kontrole i dinamičkog programiranja i učenja. Postupak djelovanja na pojačano učenje pomoću MDP (Markov Decision Processes) formalizma s kojim je radio Chris Watkins 1989. godine je zapravo vrijeme kada se dogodila potpuna integracija metoda dinamičkog programiranja s učenjem za vrijeme dobivanja novih podataka. Ove dvije glavne grane pojačanog učenja su se u kasnim 1980-tim godinama međusobno povezale s tre-

ćom, manje izraženom granom, koja se bavi metodom vremenske razlike.

3.4 Elementi pojačanog učenja

Osnovni građevni blokovi na kojem se temelje metode pojačanog učenja su:

- Politika: Opisuje ponašanje u određenom trenutku. Određuje koje radnje poduzeti ovisno o stanju okruženja, iskazuje se kao funkcija vjerojatnosti koja opisuje vjerojatnost sljedeće određene radnje.

- Signal nagrade: Govori algoritmu koliko je uspješan, a primarni cilj svakog algoritma pojačanog učenja je težnja prema najvećoj mogućoj nagradi.

- Funkcija vrijednosti: Daje određenoj situaciji konkretnu vrijednost. Na neki način procjenjuje buduću nagradu, što ga čini sekundarnim ciljem algoritmu pojačanog učenja. Uspješnim predviđanjem dugoročne nagrade poboljšava i ubrzava process izvršavanja primarnog cilja.

- Model okruženja: Neobavezni element, koji oponaša okruženje i pomaže algoritmu planirati buduća ponašanja. Algoritmi koji se služe modelom nazivaju se metode bazirane na modelu (eng. *Model based*), za razliku od jednostavnijih metoda, slobodni od modela (eng. *Model free*).

3.5 Konačan Markov proces odlučivanja

Formalizacija sekvencijalnog postupka donošenja odluka u kojima se uključuje, osim neposredne nagrade, odgođena nagrada. Procjenjuje se vrijednost za svaku radnju u nekom stanju $q_*(s, a)$ ili vrijednost stanja $v_*(s)$.

Osnovni elementi su:

- Agent

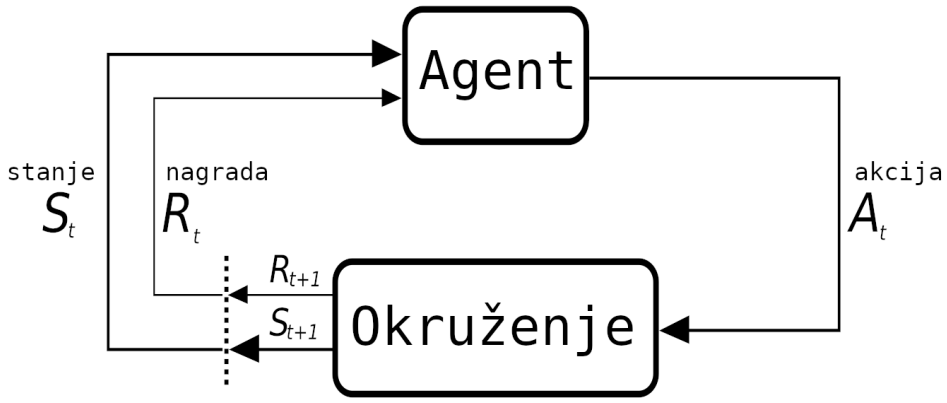
- Okruženje

- Stanje

- Akcija

- Nagrada

U Markovom procesu odlučivanja (eng. *Markov decision process*) okolina se definira kao skup stanja S, dozvoljene akcije kao skup A te skup R nagrade. Svi skupovi se smatraju konačnim, te u svakom vremenskom koraku $t = 0, 1, 2, ..$, agent proučava stanje $S_t \in S$ i izvršava akciju $A_t \in A$. U svakom vremenskom koraku pridružena akcija stanju daje par (S_t, A_t), te izvršavanjem agentove radnje okruženje prelazi u sljedeće stanje $S_{t+1} \in S$. U novom stanju agent primi nagradu $R_{t+1} \in R$ i proučava novo stanje kako bi primijenio sljedeću akciju. Ovaj proces se može opisati funkcijom $f(S_t, A_t) = R_{t+1}$, te interakcija agenta i okruženja prikazana je na slici 8.



Slika 8: Diagram markovog procesa odlučivanja

3.6 Nagrada

Očekivana nagrada G_t se računa kao zbroj svih budućih nagrada $G_t = R_{t+1} + R_{t+2} + R_{t+3} + ... + R_T$, gdje R_T predstavlja nagradu u konačnom stanju okruženja. Prolaženjem jednom od početnog stanja do krajnog naziva se *epizoda*, tako da pokretanjem nove epizode okruženje se ponovno namjesti na neko određeno stanje neovisno o tome kako je prijašnja epizoda završila. Međutim kako je agentu važnija neposredna nagrada, a osim toga postoje okruženja u kojima se neprekidno prelazi iz stanja u stanje i okruženje ne posjeduje konačno stanje, uvodi se pojam *popust budućih nagrada*. Tako da svaku sljedeću buduću nagradu uzme sve manje u obzir. Ovakva očekivana nagrada se definira kao $R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ...$, gdje gamma predstavlja

stopu popusta budućih nagrada, $0 \leq \gamma \leq 1$, tako da je sada G_t opisan jednadžbom 1.

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{1}$$

3.7 Politike i funkcije vrijednosti

Politika određuje agentov sljedeći potez, kroz skupljanje iskustva ta politika se može mijenjati. Označava se sa π i definira distribuciju vjerojatnosti preko $a \in A(s)$ za svaki $s \in S$, tako da konačne vjerojatnosti svake akcije $a = A_t$ u stanju $s = S_t$ su opisane politikom $\pi(a|s)$. Za opisivanje agentu dali se nalazi u dobrom stanju ili opisivanje agentu koliko je dobra akcija za neko stanje zadužene su funkcije vrijednosti. Koliko je dobro stanje, odnosno koliko je dobra neka akcija u nekom stanju agentu se daje do znanja pomoću vrijednosti očekivane nagrade. Funkcije vrijednosti se definiraju u odnosu na politiku, pošto ona utječe na donošenje agentove odluke. Jedna od funkcija vrijednosti je funkcija vrijednosti stanja i pomoću nje agent dobije informaciju o očekivanoj nagradi, dok prati politiku π, počevši od stanja s u trenutku t. Formalno se označava matematičnom jednadžbom 2.

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[G_t|S_t = s] \\ &= \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s \right] \end{aligned} \tag{2}$$

S druge strane postoji funkcija vrijednosti akcija $q_\pi(s, a)$ koja opisuje koliko je dobra akcija a u stanju s dok se prati politika π. Tako da funkcija računa vrijednost očekivane nagrade za svaku akciju i definirana je matematičkom jednadžbom 3.

$$\begin{aligned} q_\pi(s, a) &= \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \\ &= \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a \right] \end{aligned} \tag{3}$$

Slovo q opisuje kvalitetu (eng. *quality*) određene akcije i često se vrijednost naziva q-vrijednost, te funkcija koja je daje q-funkcijom. U jednadžbama 2 i 3 $\mathbb{E}_\pi$ definira očekivanu vrijednost, u ovom slučaju očekivanu nagradu, slučajne variable praćenjem politike π.

3.7.1 Optimalne politike i funkcije vrijednosti

Jedna od osnovnih značajki pojačanog učenja koja se ističe, je da algoritam ima neki definirani cilj, prema kojemu teži. Glavni cilj je pronalaženje optimalne politike, koju će agent pratiti, kako bi doveo nagradu do najveće moguće razine. Neka politika π se smatra boljom od neke druge politike π', ili bar jednako dobrom, ako za sva moguća stanja nudi višu, ili jednaku nagradu za agenta. Optimalna politika je ona koja za sva moguća stanja daje najveću nagradu i kada ne postoji neka druga politika koja za neko stanje daje veću nagradu. Također optimalna politika posjeduje njoj pripadajuću optimalnu funkciju vrijednosti stanja, odnosno optimalnu funkciju vrijednosti akcija. Optimalne funkcije se bilježe oznakom v_* gdje je $v_*(s) = \max_\pi v_\pi(s)$ za funkciju vrijednosti stanja, a za funkcije vrijednosti akcija q_* gdje je $q_*(s,a) = \max_\pi q_\pi(s,a)$ i to za sva $s \in S$, te u slučaju funkcije vrijednosti akcija još i za sva $a \in A$.

3.8 Bellmanova jednadžba optimalnosti

U nastavnom djelu ovoga rada usredočenje će biti na q_*, tako da se v_* neće razrađivati. To ne znači da to nije primjenjivo za v_*, ali izrada agenta ide u smjeru q-učenja (eng. *q-learning*), pa je sav daljan opis prilagođen tome. Po Bellmanovoj jednadžbi, koju bi trebao svaki algoritam kojemu je cilj pronalaženje optimalne funkcije vrijednosti zadovoljiti, svaka očekivana nagrada jednaka je trenutnoj nagradi dodana na maksimalne buduće nagrade s popustom. Ima za zadatak pronaći q_* koja zadovoljava jednadžbu 4.

$$q_*(s,a) = \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} q_*(s',a')\right] \tag{4}$$

Trenutno stanje i akcija predstavljeni su sa s i a, iz kojih proizlazi nagrada R_{t+1}. Pa je toj nagradi za bilo koje buduće stanje s' i buduće akcije a' dodana vrijednost maksimalne buduće nagrade s popustom. Pošto se donosi optimalnu akciju u stanju s, očekuje se da će se u sljedećemu stanju s' moći izvršiti optimalna akciju a'.

3.9 Q-Učenje

Ideja q-učenja jest tablica u kojoj je svaki redak određeno stanje u okolini, a svaki stupac određena akcija za tu okolini. Tako da se u svakoj ćeliji nalazi q-vrijednost za određenog para stanja i akcije. Potrebno je odrediti početne q-vrijednosti, za koje nema neko čvrsto pravilo.

Neki smisleni pristup je postaviti ih na nulu pošto agent još nije posjetio niti jedno stanje i nema nikakvih informacija o okolini. Neka istraživanja su pokazala da optimističnim pristupom na početno stanje, znači svakoj akciji u novom stanju se dodijeli najveća q-vrijednost, donijelo je bolje rezultate učenja. Prolaženjem kroz stanja okoline i izvršavanjem akcije se te q-vrijednosti ažuriraju, tako da češćim izvršavanjem iste akcije u nekom stanju ta q-vrijednost se sve više približava stvarnoj. Za ažuriranje q-vrijednosti u tablici koristi se Bellmannova jednadžba optimalnosti 4. Pošto tablica u početku nema korisnih informacija o tome koje akcije poduzeti uvode se pojmovi istraživanja (eng. *exploration*) i iskorištavanja (eng. *exploitation*). U fazi istraživanja agent donese nasumičnu odluku, tako da u sljedećemu stanju primi nagradu i ovisno o tome može ažurirati q-vrijednost u tablici. Istraživanje služi za prikupljanje informacija o okolini kako bi se u buduće moglo donijeti što bolju odluku. U fazi iskorištavanja agent izvrši po njemu najbolju akciju, odnosno pronađe redak u q-tablici koji odgovara trenutnom stanju i ovisno o tome koja akcija sadrži najvišu q-vrijednost, tu akciju izabere. Ta faza služi za prikupljanje najvišu moguću nagradu, što je ipak i agentov glavni zadatak.

3.9.1 Epsilon pohlepna strategija

Potreban je neki omjer koji raspoređuje koliko i kada će se istraživati, odnosno iskorištavati. Želi se izbjeći istraživanje kada se posjeduje dovoljno informacija o okolini i izostati maksimalnoj nagradi, a također se želi izbjeći iskorištavati s nedovoljno informacija o okolini, što isto tako dovodi do neuspješnog prikupljanje maksimalne nagrade. Taj omjer u epsilon pohlepnoj strategiji (eng. *epsilon greedy strategy*) je ϵ i broj je između 0 i 1. Odluka se donosi tako da se nasumično generira broj između 0 i 1, te ako je taj broj veći od ϵ onda se iskorištava, znači izabere se akcija koja bi trebala donijeti najveću nagradu. Ako je manji onda se istražuje i izvrši se nasumična akcija, što dovodi do zaključka da sa $\epsilon = 1$ je 100% sigurno da će se istraživati a kada je $\epsilon = 0$ se neće nikako. Kako se situacije razlikuju, da li agent posjećuje svoje prvo stanje ili je već neko vrijeme proveo u okolini i prošao dosta stanja, mora se prema tome i prilagoditi ϵ. Za prilagođavanje se koristi stopa propadajućeg epsilona (eng. *epsilon decay rate*), tako da na početku je $\epsilon = 1$ i vremenom opada i sve više se približava 0. Znači da je na početku agentu veći prioritet istraživati okolinu a vremenom sve više i više ju iskorištava. Po terminologiji algoritama se smatra nekim algoritmom *pohlepnim* kada uzme u obzir samo neposrednu vrijednost, odnosno nagradu, u obzir bez da se razmatraju dugoročne posljedice. Zbog toga se kaže da agent vremenom postane pohlepan. Najčešće ϵ nikad ne dosegne vrijednost nule, nego ga se prestane

smanjivati u trenutku kada $\epsilon = 0.1$, tako da u 10% slučajeva agent i dalje istražuje, kako mu ne bi nešto promaklo.

3.9.2 Ažuriranje q-vrijednosti

Kada se agent uči snalaziti u okolini, kroz svako stanje koje prolazi izvrši neku akciju i ovisno o nagradi ažurira q-vrijednost u tablici. Koristi se Bellmanova jednadžba optimalnosti kako bi se q-vrijednosti u tablici približile optimalnoj q_* vrijednosti. U svakom prolaženju kroz stanja se računa gubitak, odnosno razlika između $q_*(s,a)$ i $q(s,a)$, opisana jednadžbom 5.

$$\begin{gathered} q_*(s,a) - q(s,a) = Gubitak \\ \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} q_*(s',a')\right] - \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}\right] = Gubitak \end{gathered} \tag{5}$$

S tim da se izračunata nova q-vrijednost ne pohranjuje izravno u tablicu. Agent bi trebao češće prolaziti isto stanje kako bi se mogao efektivnije približiti optimalnoj q-vrijednosti. Uvodi se novi parametar stope učenja (eng. *learning rate*), koji određuje koliko će se nova vrijednost uzeti u obzir u odnosu na staru, te se označava sa α. Stopa učenja je broj između 0 i 1, gdje 1 znači da će nova q-vrijednost u potpunosti biti izračunata u tom trenutku a stara se neće uzeti u obzir, tj. bit će pregažena. Kako agent sve češće prolazi kroz stanja skuplja sve više iskustva i poželjno je da prijašnja iskustva uzme u obzir, tako da stvori novo iskustvo koje sadrži sažete informacije o starim iskustvima i novim. Potpuni izračun za ažuriranje nove q-vrijednosti prikazan je jednadžbom 6, te iterativnim postupkom ažuriranja q-funkcija bi trebala konvergirati optimalnoj, iz čega proizlazi optimalna politika.

$$q^{novi}(s,a) = (1-\alpha) \cdot \underbrace{q(s,a)}_{\substack{\text{staro} \\ \text{iskustvo}}} + \alpha \overbrace{\left(R_{t+1} + \gamma \max_{a'} q(s',a')\right)}^{\substack{\text{novo} \\ \text{iskustvo}}} \tag{6}$$

Primjer q-učenja u Pythonu s kojim se uspješno učilo igru križić-kružić dan je u ispisu 15 koji se nalazi u dodatku.

Klasa agenta naslijeđena je od bazne klase `Base` koja je apstraktna klasa dana u ispisu 4. Definira potpis metode `observe_environment`, `choose_action` te `prepare_for_episode`.

```
from abc import ABC, abstractmethod
from typing import Any, Optional, Tuple


class Base(ABC):
    @abstractmethod
    def observe_environment(self, env: Any) -> None:
        pass

    @abstractmethod
    def choose_action(self, actions: Tuple[Any, ...]) -> Optional[Any]:
        pass

    @abstractmethod
    def prepare_for_episode(self) -> None:
        pass
```

Ispis 4: Bazna klasa agenta

Ova implementacija q-učenja očekuje okruženje `ObservableEnvironment` što je jako jednostavna klasa vidljiva u ispisu 5. Sadrži tri podatka: stanje, nagrada i to, da li je posljedno stanje u epizodi.

```
from typing import NamedTuple, Tuple

class ObservableEnvironment(NamedTuple):
    state: Tuple[int, ...]
    reward: float
    is_terminal: bool
```

Ispis 5: Očekivano okruženje q-učenja

Može se primijetiti da nije obična klasa, pošto nema niti jedne metode, nego je zapravo struktura, a to se u Pythonu postiže nasljeđivanjem `NamedTuple`. Na ovaj način se generalizirao postupak q-učenja, da nije strogo vezan za križić-kružić, nego je u stanju učiti bilo koje okruženje koje prosljeđuje agentu informacije o stanju na ovakav način.

3.10 Neuronska mreža

U ovom području strojnog učenja se istražuju i primjenjuju umjetne neuronske mreže (eng. *artificial neural networks*). Neuronske mreže na jednoj apstraktnoj razini imitiraju rad i strukturu mozga. Mozak posjeduje ogromnu mrežu stanica, nazvanih neuroni, koji su međusobno povezani. Svaki neuron preko svojih hvataljki prima signal od drugih neurona, obrađuje i prosljeđuje dalje drugim neuronima za daljnju obradu. Što je češća razmjena signala između dva neurona, to njihova veza postaje čvršća te se tako utvrđuju pojmovi. Umjetna neuronska mreža složena je u slojevima, tako da je svaki neuron povezan s neuronima iz sljedećeg sloja a nikako iz istog. Mrežu s više od dva sloja smatra se dubokom. Posebni slojevi su ulazni i izlazni slojevi, gdje ulazni slojevi primaju podatke a izlazni daju neki zaključak. Između njih se može nalaziti nekoliko skrivenih slojeva, tako da sirovi podaci stižu samo prvom sloju neurona, koji obrađeni signal usmjeravaju drugom sloju. Takav signal prolazi kroz nekoliko slojeva prije nego se mogu donijeti zaključci. Postoje različite vrste slojeva koji na drugačiji način obrađuju podatke. Osnovni sloj koji se najčešće koristi je potpuno povezani sloj (eng. *fully connected layer*), ponekad i nazvan gusti sloj (eng. *dense layer*). Kod računalne neuronske mreže svaki neuron je neka funkcija, koja prima neki podatak kao ulaz, obrađuje ga te izbaci na izlazu obrađeni podatak koji se šalje sljedećem neuronu na ulaz. Cijela neuronska mreža je kompozicija mnogih funkcija što je također čini funkcijom. Svakom podatku koji ulazi u neuron pridodani su parametri, često nazvani i težine (eng. *weights*), pa se tijekom treninga mijenjaju parametri tako da se postigne što bolja točnost na izlazu. Na izlazu svakog neurona stoji aktivacijska funkcija koja brojem između nula i jedan opisuje koliko se taj neuron aktivira, tako da nula znači da se neuron nije aktivirao a jedan da se u potpunosti aktivirao. U ovom području do izražaja jako dolaze elementi iz linearne algebre, pošto su parametri i aktivacije vektori, matrice ili tenzori.

3.10.1 Učenje neuronske mreže

Glavni cilj učenja neuronske mreže je pronalaženje optimalnih parametara koji najbolje pretvaraju ulazne podatke u željeni izlaz. Postupak se odvija tako da se za neki ulaz u mrežu usporedi izlaz sa željenim izlazom, izračuna se razlika i na osnovu toga se ažuriraju parametri u neuronskoj mreži, tako da za sljedeći put za isti ili sličan ulaz ova razlika bude što manja. Prije početka učenja parametri se inicijaliziraju s nasumičnim vrijednostima. Za izračun razlike koristi se funkcija gubitaka (eng. *loss function*) koja kao ulaz prima izlaz neuronske mreže i željeni izlaz. S nasumičnim parametrima je ova razlika ogromna, a vremenom se ažuriranjem

parametara teži za što manjom razlikom, odnosno da gubitak bude što bliži nuli. Gubitak koji se često koristi je funkcija srednje kvadratne pogreške (eng. *mean squared error*) i opisana je formulom 7 gdje je y željeni izlaz a $\hat{y}$ izlaz koju je neuronska mreža izbacila.

$$\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{7}$$

Nakon izračuna gubitka povratnom se propagacijom (eng. *backpropagation*) ažuriraju svi parametri u neuronskoj mreži tako da se spuštanjem po gradijentu (eng. *gradient descent*) funkcije gubitaka povećavaju ili smanjivaju vrijednosti parametara. Prema tome za koliko se u svakoj povratnoj propagaciji parametri mjenjaju, se unaprijed definira stopa učenja. Za ubrzavanje konvergencije gubitka koriste se optimizacije kao stohastično spuštanje po gradijentu (eng. *stochastic gradient descent*) koji nasumičnim uzorkivanjem računa gubitak samo na maloj skupini podataka s kojom dobije neku procjenu stvarne funkcije gubitaka. Dodatno pomoću momentuma, koji oponaša akceleracijsku silu na stopi učenja, znači što češće se vrijednosti mijenjaju u nekom smjeru to su koraci veći, se taj postupak još ubrzaje. Vremenom su se mnogo raznih optimizacija razvile i ne postoji neko pravilo pomoću kojega se može odrediti što se koristi u kojemu trenutku, nego se izvode pokusi s raznim metodama i promatraju rezultati pomoću kojih se onda izabere metoda za daljnji rad. Od svih metoda koje se koriste u dubokom učenju, funkcija srednje kvadratne vrijednosti i stohastično spuštanje po gradijentu na neki su način početna točka s kojom se uči.

3.10.2 Duboka Q-mreža

S obzirom na to da neko okruženje posjeduje ogromnu količinu stanja, te pohranjivanje svih stanja bi moglo vrlo lako prelaziti raspoloživu memoriju računala, s dubokom se umjetnom neuronskom mrežom aproksimira q-tablicu. Sastavljena je tako, da na ulaznom sloju ulazi stanje a na izlaznom se sloju svakoj akciji pridodijeli njezinu q-vrijednost. S dubokom q-mrežom se također računa optimalna q-vrijednost pomoću Bellmannove jednadžbe optimalnosti.

4 Implementacija

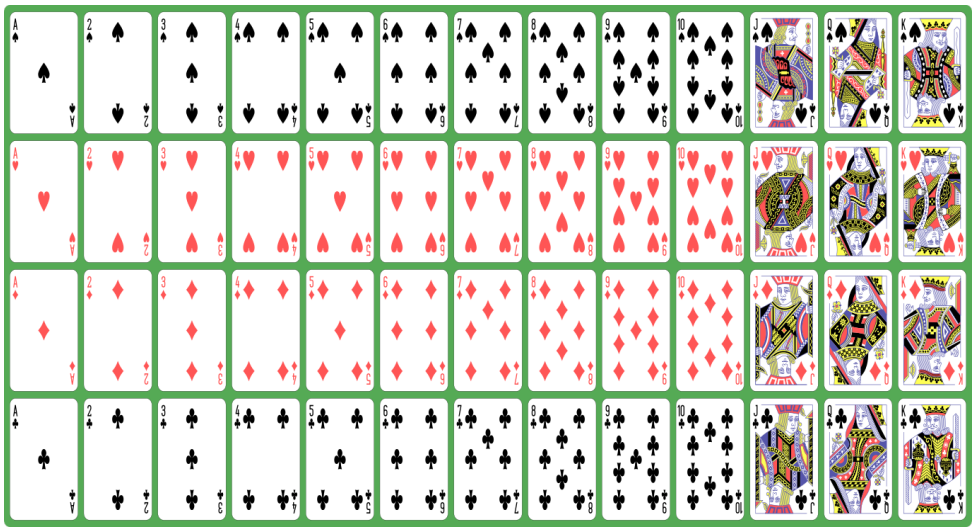
Implementacija se sastoji od dva dijela. Prvi dio opisuje igru Texas Hold'em poker te izradu igre u python programskom jeziku, a drugi dio se odnosi na izradu agenta koji treba naučiti igrati ovu igru i biti uspješan na jednoj osnovnoj razini.

4.1 Implementacija igre

4.1.1 Poker

Jako poznata igra karata, nastala u Americi, čiji korijeni dosežu od perzijske igre As-Nas. Kroz 19. stoljeće igra se razvijala s raznim dodacima i varijacijama. Postaje jako popularna kroz 20. stoljeće kada su se počeli održavati turniri. Ta popularnost se proširila svijetom uz igranje i održavanje turnira putem interneta, te turnira koji snimaju karte svih igrača tako da gledatelji mogu bolje pratiti igru.

Iako postoje razne verzije pokera, svi oni dijele jednaka osnovna pravila igre. Igra se sastoji od 52 igrajuće karte, koje se mogu vidjeti na slici 9, od kojih se 13 karata 4 puta ponavljaju samo s različitim simbolima. Vrijednosti tih 13 karata su: brojevi od 2 do 10, dečko, kraljica, kralj i as. Cilj svakog igrača je sebi složiti najjaču ruku kombinacijom od pet karata.



Slika 9: Izgled igračih karata za poker

Ruke poredane od najslabije do najjače:

- Visoka karta (eng. *high card*): Igrač nije uspio složiti odgovarajuće kombinacije i gleda

se karta s najjačom vrijednosti. Karte poredane po jačini (od najslabije prema najjače): brojevi od 2 do 10 (veći broj znači snažnija karta), dečko, kraljica, kralj pa as.

- Par (eng. *pair*): Igrač je od bilo koje vrijednosti uspio skupiti dvije iste karte.
- Dva para (eng. *two pairs*): Ruka se sastoji od dvije karte s jednom vrijednosti i dvije s drugom.
- Tris (eng. *tris*): Bilo koja vrijednost karte 3 puta.
- Skala (eng. *straight*): Pet različitih vrijednosti poredanih tako da je svaka sljedeća karta za jednu vrijednost jača od prethodne.
- Boja (eng. *flush*): Pet karata istog simbola.
- Puna kuća (eng. *full house*): Tri karte iste vrijednosti i dvije druge karte istih vrijednosti.
- Poker (eng. *poker*): Četiri karte istih vrijednosti.
- Skala u boji (eng. *straight flush*): Pet vrijednosti gdje je svaka sljedeća za jednu vrijednost jača od prethodne i sve u istim simbolima.
- Kraljevska skala u boji (eng. *royal flush*): 10, dečko, kraljica, kralj, as u istim simbolima.

4.1.2 Pravila igre

Igra koja se implementirala naziva se *Limit Texas Hold'em turnir*, i pravila su sljedeća. Potrebno je bar dva igrača, tri da bude zanimljivo pa do nekih desetak (nema neke fiksne gornje granice, ali najčešće je deset). Prije početka igre svi igrači uplaćuju dogovoreni iznos novca za sudjelovanje u igri, te dobiju za uzvrat žetone vrijednosti uloženog novca. Igrači sjede za ovalnim stolom i jednoga od igrača se određuje kao djelitelja. Djelitelj dobije poseban žeton koji ga vidljivo označava kao takvog. Karte se izmiješaju i svakom igraču se dodjele dvije karte, u smjeru kazaljke na satu. Ove dvije karte su za ostale igrače tajna, znači svaki igrač nastoji da jedini vidi te karte, zapamti ih i po mogućnosti do kraja kruga ih drži okrenute naopako na stolu. U prvoj fazi *pre flop* nakon dijeljenja karata prvi igrač lijevo od djelitelja je prisiljen uložiti žetone u visini pola vrijednosti minimalnog uloga, taj ulog se naziva *small blind*, a sljedeći ulaže *big blind* cjelokupan minimalni ulog, također prisiljeno. Prisiljeni ulozi služe kako bi u svakom krugu postojala neka dobit, a i da skrati igru koja i onako zna potrajati satima. Ostali igrači nakon

njih imaju mogućnost birati hoće pratiti (eng. *call*), što znači da moraju sve skupa uložiti koliki je trenutačni najveći ulog, hoće li odustati i preklopiti karte (eng. *fold*), gdje odbacuju karte iz ruku i isključeni su iz igre do kraja kruga ili će povisiti (eng. *raise*) ulog, u kojem slučaju trenutni ulog postane jednak prošlom plus minimalni iznos za ulaganje. U trenutku kada su ostali igrači napravili potez, prvi igrač do djelitelja mora nadoplatiti ostali iznos za pratiti ili izabrati neki od drugih poteza i na kraju igrač koji je morao prisilno uložiti minimalni ulog ima mogućnost napraviti potez, s tim da ako nije bilo povećavanja uloga umjesto da prati (eng. *call*) ima potez potvrde (eng. *check*), što znači da nema namjeru povisiti ulog niti izostati iz trenutnog kruga. Kada svi igrači potvrde potez djelitelj izbacuje jednu kartu iz igre, i kaže se da se karta izgori (eng. *burn*), te na sred stola okrene 3 karte vidljive svima, to je *flop* faza. Karte okrenute na srid stola su karte zajednice (eng. *community cards*) i svaki igrač ima pravo svoju ruku od pet karata složiti s bilo kakvom kombinacijom svoje vlastite ruke i karata zajednice. Slijedi potez svakog igrača, međutim u ostalim fazama nema prisilnog ulaganja. U *turn* fazi djelitelj opet izbacuje kartu i okrene sljedeću na stol, ponovno svi igrači određuju svoj potez. *River* faza je identična *turn* fazi, tako da na kraju je pet karata okrenuto na sredini stola i bar dva igrača u igri. Sada igrači okreću karte i izjasne se koju kombinaciju su sebi složili, i igrač s najjačom kombinacijom kupi ukupan ulog (eng. *pot*), ili ako imaju nekoliko igrača istu jačinu onda ga dijele. Ako se u bilo kojoj fazi dogodi da svi osim jednog igrača odustanu, preskaču se ostale faze i igrač skuplja ukupni ulog. Postoji još i poseban potez ulaganja svega (eng. *all in*) gdje igrač kada nema dovoljno žetona za uložiti, ulaže sve što ima i u slučaju gubitka izbačen je iz igre. Dobitnik je igrač koji zadnji ostaje u igri i nagrada mu je ukupan uplaćeni ulog svih igrača. Pošto je igra nepotpuno informirana igra, postoji mogućnost blefiranja, gdje igrač s lošom rukom ulaže i/ili diže ulog da stvori dojam da ima jaku ruku na račun koje će neki ili svi ostali igrači odustati u trenutnom krugu.

Limit označava da je svaki ulog fiksan i u svakoj fazi su dozvoljena tri ponovna povećavanja uloga (eng. *reraise*), gdje za razliku od *no limit* igrač može povećati najmanje minimalan ulog a najviše koliko hoće i nema ograničen broj ponovnog povećavanja uloga.

Texas Hold'em je način igre s dvije tajne karte i pet javnih, dok u klasičnom pokeru svaki igrač dobije pet karata (koje su drugim igračima tajna) i ima pravo ne izbaciti niti jednu, a može izbaciti koliko hoće, pa i sve karte i onoliko koliko izbaci dobije novih, te na kraju to postaje konačna ruka.

Turnir podrazumijeva da svi igrači na početku uplate jednak iznos i igra se dok svi ispadnu osim jednog, dok u *cash game* može igrač ponovno kupiti žetone kako bi nastavio igrati, može odustati od igre iako nije izgubio sve žetone te ih unovčiti, a mogu čak naknadno doći i novi igrači.

4.1.3 Program

Pisane su klase koje predstavljaju karte i špil. Klasa `Card`, koja je dana u ispisu 6, predstavlja jednu kartu koja ima 3 vrijednosti: oznaku, symbol i vrijednost. Dok klasa `Deck`, koja je dana u ispisu 7 stvara špil od 52 odgovarajuće karte za poker te implementira metode za miješanje, dijeljenje i izbacivanje karata.

```
class Card:
    def __init__(self, rank: str, suit: str, value: int) -> None:
        self._rank = rank
        self._suit = suit
        self._value = value

    @property
    def rank(self) -> str:
        return self._rank

    @property
    def suit(self) -> str:
        return self._suit

    @property
    def value(self) -> int:
        return self._value

    def __str__(self) -> str:
        return str(self._rank) + '(' + str(self._suit + ')')

    def __eq__(self, other) -> bool:
        return self._value == other.value and self._suit == other.suit

    def __lt__(self, other) -> bool:
        return self._value < other.value
```

```
    def __gt__(self, other) -> bool:
        return self._value > other.value

    def __repr__(self) -> str:
        return str(self)
```

Ispis 6: Klasa karta

```
from .card import Card
from random import shuffle
from typing import List

RANKS = ('2', '3', '4', '5', '6', '7', '8', '9', '10',
         'Jack', 'Queen', 'King', 'Ace')
SUITS = ('Heart', 'Diamond', 'Spade', 'Club')

class Deck:

    def __init__(self) -> None:
        self.cards = [Card(rank, suit, value + 1) for rank, value in
                      zip(RANKS, range(len(RANKS))) for suit in SUITS]

    def shuffle(self) -> None:
        shuffle(self.cards)

    def deal(self, amount: int = 1) -> List[Card]:
        to_deal = self.cards[:amount]
        self.cards = self.cards[amount:]

        return to_deal

    def burn(self, amount: int = 1) -> None:
        self.cards = self.cards[amount:]

    def create_new_deck(self) -> None:
        self.__init__()

    def get_cards(self) -> List[Card]:
        return self.cards
```

Ispis 7: Klasa špil

Svi igrači koji žele sudjelovati u poker turniru moraju biti naslijeđeni od apstraktne bazne klase. `Player` koja je prikazana u ispisu 8.

```
from abc import ABC, abstractmethod
from game import Card, Moves, State
from typing import List


class Player(ABC):
    def __init__(self, chips: int) -> None:
        self._chips = chips
        self._hand = []

    def receive_chips(self, amount: int) -> None:
        self._chips += amount

    def spend_chips(self, amount: int) -> int:
        self._chips -= amount
        return amount

    def receive_cards(self, cards: List[Card]) -> None:
        self._hand += cards

    def get_hand(self) -> List[Card]:
        return self._hand

    def destroy_hand(self) -> None:
        self._hand = []

    def get_amount_of_chips(self) -> int:
        return self._chips

    @abstractmethod
    def make_move(self, possible_moves: List[Moves],
                  game_state: State) -> Moves:
        pass
```

Ispis 8: Bazna klasa igrač

Bazna klasa prima količinu žetona kao parametar u konstruktor, implementira osnovne operacije svakog igrača, koje su: primanje žetona, trošenje žetona, primanje karata, pokazivanje ruke, uništavanje ruke i prikaz vrijednosti žetona. Bazna klasa definira apstraktnu metodu `make_move` za odrađivanje akcije koja prima trenutno stanje igre te niz mogućih, odnosno, dozvoljenih akcija. Trebala bi vratiti jednu od akcija koja se nalazi u nizu dozvoljenih. U suštini bi trebao inteligentni agent na osnovu određenog stanja u igri donijeti odluku o najboljem sljedećem potezu. Ostavlja se fleksibilnost za buduće pokuse gdje se na osnovu broja informacija o okruženju, odnosno, stanja u igri, mjeri kvaliteta odluke agenta o sljedećem koraku (agent se smatra inteligentnijim kad sa što manje informacija donese dobru odluku).

Glavni dio cijele igre je klasa `Table`, brine se o rasporedu igre te primjenjuje pravila u Limit Texas Hold'em turniru. Prima niz klasa kao ulazni parametar u konstruktor. Provjerava da li su sve klase naslijeđene od bazne klase igrača, inicijalizira sve igrače s proslijeđenim klasama, dodjeljuje im žetone i pokreće igru. Stol posjeduje vlastitu klasu igrača koja enkapsulira baznu klasu, te dodatne podatke koji služe za praćenje same igre. Ova pomoćna klasa prikazana je ispisom 16 u dodatku. Nakon što stol rasporedi igrače, započinje s turnirom, prolazi kroz sve faze igre dok ne ostane samo jedan igrač. Prati ulaganja svih igrača, i na kraju dijeli ukupni ulog odgovarajućim igračima. Klasa za igrače za stolom smještena je u datotečnom sustavu unutar direktorija za stol. Namjena je usko vezana za igru tako da njen domet ne bi smio prelaziti sami stol. Kvalitetan software bi se trebao sastojati od neovisnih elemenata, što omogućuje jednostavnije testiranje svakog elementa te razmjenu ili dodavanje novih elemenata. Iako se dosta toga u ovoj klasi razdvojilo (kao npr. izrađena je odvojena klasa koja određuje jačinu krajnje ruke), i dalje ima prostora za refaktoriranje. Tu bi sljedeći korak bio izbaciti podjelu žetona na kraju svakog kruga u zasebnu klasu i tako rasteretiti klasu stola. Najkompleksnija funkcija cijele igre je na kraju svakog kruga koja dijeli žetone dobitnicima. Pošto mora pokriti sve mogućnosti, dosta problematično postaje kada je to nekom ili nekoliko igrača posljednji potez *all in*, tada nastaje nekoliko razina dobitnika te se svakom dobitniku mora izračunati točan dobitak. Postoje slučajevi gdje nakon dijeljenja *pota* ostaju žetoni jer ih nije moguće podijeliti na broj dobitnika. U tom slučaju svaka kuća ima svoja pravila i većinom se igra po nekom dogovoru za taj slučaj. Ovdje se taj ostatak prenese u sljedeći krug, igra se normalno i na kraju

dobitnik tog kruga, ili dobitnici ako je moguće među njima podijeliti bez ostatka, pokupe ostatak iz prethodnog kruga. Kroz izradu igre, klasa `Table` je poprimila preveliku odgovornost i previše stvari se odvijaju u toj klasi, te se kroz refaktoriranje izdvojilo klasu koja računa jačinu ruke. Bodovni sustav je osmišljen tako da najjača ruka nekog tipa ruke ima manje bodova od najslabije ruke jačeg tipa. U dodatku se nalazi klasa koja pronalazi konačnu ruku i dana je u ispisu 17. Koeficijent svake vrste ruke dan je ispisom 9.

```
from enum import Enum



class FinalHandType(Enum):
   HIGH_CARD = 1
   PAIR = 20
   TWO_PAIRS = 300
   TRIS = 4000
   STRAIGHT = 15000
   FLUSH = 33000
   FULL_HOUSE = 500000
   POKER = 6600000
   STRAIGHT_FLUSH = 21500000
   ROYAL_FLUSH = 22000000
```

Ispis 9: Koeficijenti vrste ruke

Vrijedi još spomenuti prikaz informacija o igri, gdje se implementirao uzorak dizajna promatrača (eng. *observer design pattern*) isto poznat kao objava i preplata (eng. *publish and subscribe*). Koristi se najčešće kada više elemenata zahtijevaju iste informacije te promjene tih informacija, kao npr. za prikaz na drugačiji način. Efektivno razdvaja logiku izvršavanja i prikaz informacija. Sastoji se od dvije komponente, od objavitelja koji drži sve pretplatitelje i samog pretplatitelja. Objavitelj omogućuje dodavanje i odstranjivanje pretplatitelja te obavještava pretplatitelje o promjenama. Svaki pretplatitelj sam za sebe definira na koji način će koristiti informacije i kako se promjene izražavaju. Implementacija objavitelja u Python kôdu prikazana je ispisom 10. S time se oslobodila klasa `Table` da se brine o bilo kakvom načinu za prikaz informacije o stanju igre, samo mora definirati metode koje dodaju i odstranjivaju pretplatitelje i u određene trenutke pozvati `notify` funkciju od objavitelja.

```
from .subscriber import BaseSubscriber as Subscriber, State
```

```
class Publisher:
    def __init__(self) -> None:
        self._subscribers = []

    def attach(self, subscriber: Subscriber) -> None:
        self._subscribers.append(subscriber)

    def detach(self, subscriber: Subscriber) -> None:
        detached = 0

        for i in range(len(self._subscribers)):
            if self._subscribers[i - detached] == subscriber:
                self._subscribers.pop(i - detached)
                detached += 1

    def notify(self, state: State) -> None:
        for subscriber in self._subscribers:
            subscriber.update(state)
```

Ispis 10: Objavitelj

Implementirala su se dva pretplatitelja, jedan koji ispisuje stanje u terminal a drugi koji zapisuje stanje u tekstualnu datoteku. U buduće bi se mogao dodati neki grafički prikaz stanja što ovaj uzorak dizajna omogućuje bez da se mijenja postojeći kôd, samo se dodaje nova klasa koja nasljeđuje `BaseSubscriber`.

Za osiguranje sigurnosti kôda pisani su testovi, te u slučaju izmjene ili dodatka novih funkcionalnosti prikazuju, da li su izmjene utjecale negativno na postojeći kôd. Postoji disciplina razvoj upravljan testovima (eng. *test driven development*), u kojemu nije dozvoljeno pisati niti jednu liniju produkcijskog kôda prije nego je napisan test koji ne prolazi i ne piše se niti jedna linija testnog kôda dok postoji test koji ne prolazi. Nakon što postoji test koji ne prolazi piše se produkcijski kôd koji omogućuje prolaznost testa i vraća se pisanje novog testa. Nažalost ova disciplina nije primijenjena u ovom projektu, iz razloga, što je potrebno, kao u bilo kojoj disciplini, bar nekoliko mjeseci iskustva kako bi se efektivno primijenila. Bez obzira nije se izostavila važnost testova, pa su se za najvažnije funkcije implementirali testovi, iz jednog razloga za provjeru da li kôd radi u redu i iz drugog, da se za sve buduće promjene može provjeriti

ispravnost. Testovi su poslužili u nekoliko trenutaka a pogotovo u fazi refaktoriranja kôda. Za testiranje jedinica (eng: *unit test*) koristio se Pythonov modul `unittest` iz razloga jer se nalazi u Pythonovoj standardnoj biblioteci, nije potrebna nikakva instalacija i može se direktno koristiti. Kreiran je direktorij test, te po konvenciji se osigurava da ime svake skripte testa započinje sa `test_`. Svaka klasa koja testira neku jedinicu mora biti naslijeđena od `unittest.TestCase`, i ime svake funkcije koja testira jedinicu također započinje sa `test_`. `TestCase` klasa ima implementirane funkcije za provjeru kao što su `assertEqual`, `assertTrue`, `assertFalse`, ..., također implementiranu funkciju `setUp` koja se izvršava prije svake testne funkcije. U toj funkciji se pripremaju objekt/i za testiranje kako se ne bi trebalo u svakoj funkciji ponavljati iste postupke. Iako se testne funkcije nalaze u istoj klasi objekti kreirani u `setUp` funkciji se ne dijele među testnim funkcijama nego se za svaku funkciju kreira novi. Funkcija `setUp` se eksplicitno ne poziva nego sve to obavlja bazna klasa `TestCase` u pozadini. U dodatku se može promatrati potpuni primjer jednog testa koji je dan u ispisu 18. Osnovna pravila kojih bi se trebali svi testovi držati su da kao prvo moraju biti brzi, u izvršavanju i prikazivanju rezultata. Testovi kojima treba puno vremena se ne izvršavaju često i kao posljedica pojavljivaju se greške u kôdu. Moraju biti neovisni jedni o drugima. Znači, ne smije niti jedan test postojati koji se ne može pokrenuti prije nego se pokrene neki drugi test. U bilo kojemu trenutku bi trebala postojati mogućnost pokrenuti bilo koji test u bilo kojem redoslijedu. Trebali bi se moći pokrenuti u bilo kakvom okruženju. Dodatno svi testovi moraju samostalno prikazati da li je test prošao ili ne i koji nije prošao. Test ne smije izbaciti nikakvu informaciju gdje je potrebna ljudska evaluacija rezultata.

4.1.4 Protivnici

Za dokazivanje da inteligentni agent napreduje i zna igrati igru, potrebno je modelirati nekog protivnika na osnovu čega se mogu donijeti ovi zaključci. Najosnovniji protivnik s čime se mjeri agent u bilo kojem području strojnog učenja jest agent koji donosi nasumične odluke. U slučaju pokera agent prima dozvoljene akcije u nekom stanju te nasumično izabere jednu od njih. Specifično za poker je ovo previše loš protivnik, jer se u bilo kojemu stanju među dozvoljenim akcijama nalazi *fold*. Jako su rijetke situacije kada igrač odustane od trenutnog kruga a da nije nitko povećao ulog. Dodatno potpuno nasumičnim biranjem akcija, češće se prekida krug, jer su svi igrači osim jednoga odustali, što znači da agent rijetko posjećuje posljedno stanje kruga u kojemu se ocjenjuju ruke. Iz ovog razloga je osnovni protivnik, protiv kojega se inteligentni agent mora iskazati, polu nasumični. Polu nasumični protivnik će slučajnim odabirom izabrati

akciju *fold* samo u slučaju kada se povećao ulog. Dodatno se implementirao algoritam s nekom heuristikom. Početnu ruku u *preflop* fazi kada još nema karata na stolu se boduje na osnovu tablice koju su složili David Skalinsky and Mason Malmuth [3]. Tablica grupira sve početne ruke u razini od najjače prema najslabijoj. U ostalim fazama se računa jačina ruke metodama koje se naslanjaju na rad: Opponent modeling in poker [4]. U radu se opisuje izračun učinkovite jačine ruke (eng. *effective hand strength*) koja je dana formulom 8.

$$EHS = HS \cdot (1 - NPOT) + (1 - HS) \cdot PPOT \tag{8}$$

Gdje je HS trenutna jačina ruke, ne uzimajući u obzir poboljšanje ili pogoršanje buduće ruke. $NPOT$ je negativni potencijal koji uzima u obzir pogoršanje buduće ruke a $PPOT$ pozitivni potencijal koji zadrži poboljšanje buduće ruke. Potpuna implementacija ove formule je računalno jako zahtjevna. Za izračun trenutne jačine ruke potrebno je generirati sve moguće protivničkove karte u ruci te usporediti jačinu sa svojom rukom, što je još prihvatljivo. Međutim za izračun pozitivnog i negativnog potencijala potrebno je generirati za svaku sljedeću fazu sve moguće karte koje se mogu okrenuti na stolu i usporediti sve moguće protivnikove ruke sa svojom. Pa se iz tog razloga odlučilo koristiti samo trenutnu jačinu ruke.

4.2 Implementacija agenta

Agent se implementirao pomoću PyTorch radnog okvira. Definirani hiperparametri su globalni za cijelu klasu, za brzo i jednostavno mijenjanje. U konstruktoru se provjerava da li postoji Cuda sposobna grafička procesorska jedinica i pohranjuje tu informaciju na sljedeći način `self._device = 'cuda'if torch.cuda.is_available() else 'cpu'`. Stvaranje same umjetne neuronske mreže dano je ispisom 11, gdje se može primijetiti kako na osnovu varijable `self._device` sadržaj neuronske mreže pohranjuje u odgovarajućoj radnoj memoriji. Kako grafička procesorska jedinica ima svoju vlastitu radnu memoriju potrebno je u njoj inicijalizirati neuronsku mrežu u slučaju da se je želi koristiti za treniranje.

```
def _create_neural_network(self) -> nn.Sequential:
    in_size = self._state_interpreter.state_space
    out_size = self._state_interpreter.action_space

    network = nn.Sequential()
```

```
    network.add_module('linear_0', nn.Linear(in_features=in_size,
                                   out_features=1000))
    network.add_module('relu_0', nn.ReLU())
    network.add_module('linear_1', nn.Linear(in_features=1000,
                                   out_features=1000))
    network.add_module('relu_1', nn.ReLU())
    network.add_module('linear_2', nn.Linear(in_features=1000,
                                   out_features=out_size))
    network.to(self._device)
    return network
```

Ispis 11: Stvaranje neuronske mreže za DQN

Neuronska mreža je jako jednostavna, ima samo dva skrivena potpuno povezana sloja s ispravljenom linearnom jedinicom (eng. *rectified linear unit*) kao aktivaciju. Ova aktivacija svaki element koji je negativan postavlja na nulu, a pozitivne ne dira. Sastavljena neuronska mreža se pohranjuje u varijablu `self._policy_net`. Kroz trening se neuronska mreža ažurira na kraju svakog kruga, što znači da se svi potezi do tada moraju na neki način pamtiti. Ovaj način se pokazao da ubrzaje treniranje. Osim poteza pamti se i cjelokupno iskustvo u određenom stanju koje se sastoji od prethodnog stanja, prethodne akcije, prethodne moguće akcije i sljedećeg stanja. Funkcija koja ažurira neuronsku mrežu prikazana je u ispisu 12.

```
  def _update_policy_net(self, batch: BatchOfExperiences,
                         reward: float) -> None:
     previous_states, previous_actions, \
        previous_possible_actions, next_states = batch

     actual_pred = self._policy_net(torch.cat(previous_states))
     target_pred = self._generate_target_preds(batch, actual_pred, reward)

     loss = self._loss(actual_pred, target_pred)
     self._optim.zero_grad()
     loss.backward()
     self._optim.step()
```

Ispis 12: Ažuriranje neuronske mreže za DQN

Nagrada koja se prosljeđuje je razlika žetona prije početka kruga i nakon, tako da

ako je razlika pozitivna znači da je agent pobijedio, a ako je negativna onda je izgubio. Prije nego se počmu ažurirati vrijednosti u mreži potrebno je pozvati `self._policy_net.train()` funkciju iz PyTorch biblioteke, koja priprema neuronsku mrežu za treniranje. Kod ažuriranja neuronske mreže prvo raspakira iskustvo, potom funkcija `torch.cat` spaja sve elemente u nizu u jedan tensor, jer se to očekuje kao ulaz pozivanjem neuronske mreže. U ovom koraku se dohvaćaju sve q-vrijednosti za prethodno stanje tako da se poziva neuronska mreža s prethodnim stanjem, što se u slučaju običnog q-učenja s tablicom jednostavno radilo indeksiranjem u određeni redak. Sljedeća funkcija `_generate_target_preds` stvara novo izračunate q-vrijednosti za akcije koje se ažuriraju, a to su akcije koje su izabrane u tom stanju. Ostale akcije se ne mijenjaju, što je usporedivo kod običnog q-učenja gdje se ažurira samo jedna ćelija u tablici. Stvaranje ažurirane q-vrijednosti dano je u ispisu 13. Gubitak se računa pomoću funkcije srednje kvadratne pogreške, koja pruža PyTorch `torch.nn.MSELoss()`, na osnovu čega se u povratnoj propagaciji ažuriraju parametri u mreži. PyTorchev `torch.optim` sadrži klase koje su zadužene za ažuriranje neuronske mreže, a u ovom slučaju se koristio `torch.optim.Adam`. Sama inicijalizacija optimizatora se izvršila u konstruktoru klase naredbom `self._optim = optim.Adam(self._policy_net.parameters(), self._alpha)`. U konstruktoru optimizatora se prosljeđuju parametri mreže te stopa učenja. Prije pozivanja funkcije povratne propagacije potrebno je postaviti gradijente na nulu, kako se ne bi zbrajali s prethodno izračunatim, s funkcijom `zero_grad` od optimizatora. Funkcija `loss_backward` računa nove gradijente, a ažuriranje parametara se odvija u `self._optim.step()` ovisno o gradijentima. Na osnovu gradijenta, optimizator povećava ili smanjuje parametar a na osnovi stope učenja, koja je proslijeđena u konstruktor, određuje se količina smanjivanja ili povećavanja. Ovdje se može primijetiti kako PyTorch implicitno manipulira sa svojim grafom o neuronskoj mreži kojeg čuva u pozadini, također koliko pojednostavljuje rad s neuronskim mrežama.

```
def _generate_target_preds(self, batch: BatchOfExperiences,
                           preds: torch.Tensor, reward: float) -> torch.Tensor:
    previous_states, previous_actions, \
        previous_possible_actions, next_states = batch
    discounted_reward_sums = \
        self._calculate_discounted_reward_sums(
            len(previous_actions), reward)

    next_preds = self._policy_net(torch.cat(next_states))
    next_qs = next_preds.max(dim=1).values.detach()
```

```
    target_preds = torch.zeros(preds.shape).to(self._device)

    for i, ppa in enumerate(previous_possible_actions):
        target_preds[i][list(ppa)] = preds[i][list(ppa)]
        if i == len(previous_possible_actions) - 1:
            target_preds[i][previous_actions[i]] = \
                discounted_reward_sums[i]
        else:
            target_preds[i][previous_actions[i]] = \
                (next_qs[i] * self._gamma) + discounted_reward_sums[i]

    return target_preds
```

Ispis 13: Stvaranje ažurirane q-vrijednosti

Unutar funkcije koja stvara q-vrijednosti prema kojima se uspoređivaju trenutno koja mreža izbacuje, događa se nešto neobično. Obično se neuronske mreže treniraju tako da se proslijedi nešto, a onda uspoređuje rezultat s očekivanim i na osnovu rezultata se parametri u neuronskoj mreži ažuriraju. No u ovom slučaju po drugi put se šalje neko stanje u mrežu. Međutim nužno je zadovoljiti Bellmanovu jednadžbu optimalnosti zbog potrebe za $\max_{a'} q_*(s', a')$. U usporedbi s izračunom q-vrijednosti u klasičnom q-učenju, fali dobar dio jednadžbe. Razlog tome je što dio jednadžbe obavlja funkcija gubitka a dio optimizator koji drži stopu učenja.

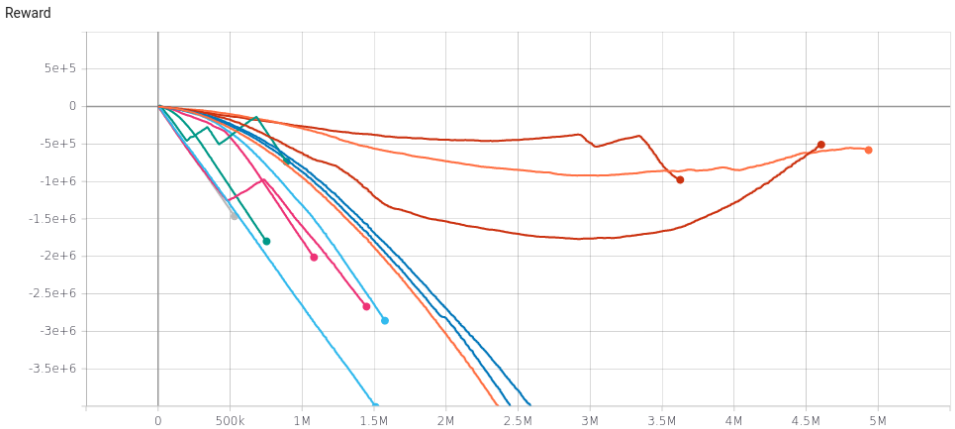
Izradila se klasa koja prevodi stanje igre u stanje koje je prihvatljivo za agenta. Stanje se prevodilo u jedno vruće kôdiranje (eng. *one hot encoding*), što znači da se na kraju dobije niz od nula i jedinica. Minimalni prostor stanja koji je izgledao prihvatljiv je od 83 elemenata, od kojih 52 elementa predstavljaju svaku kartu, gdje su jedinice poznate karte, bez obzira da li se nalaze na stolu ili u ruci. Sljedećih 30 elemenata su ukupan broj žetona, znači da se turnir ograničio na 3 igrača gdje svaki na početku posjeduje 10 žetona. Još se na kraju dodao jedan element koji se odnosi na mogućnost povećanja uloga, kako u Limit Texas Hold'em postoji mogućnost samo tri puta ponovno povećati ulog u svakom krugu.

4.3 Učenje

Klasa `SimpleDqnBot`, koja predstavlja agenta, sadrži globalne varijable koje drže hiperparametre treninga. To su alfa (stopa učenja), gamma (stopa popusta buduće nagrade), epsilon (stopa pohlepe) te oznaku da li epsilon propada tijekom treninga. Za učenje proširila se klasa `Table` s mogućnosti pokretanja turnira ispočetka s istim igračima. Sadrži metode koje postavljaju sve objekte u početni položaj s kojim se započinje turnir. Napisana je skripta u kojoj se određuju parametri treniranja, od kojih je najbitniji broj epizoda u kojima se odvija trening. Skripta inicijalizira turnir s agentom q-duboke mreže i dva polu nasumična. Nakon svake epizode pokreće novi turnir i na kraju treninga ispisuje informacije o treningu. Dodatno se na kraju treninga pohranjuje stanje neuronske mreže na tvrdi disk pomoću funkcije `torch.save(self._policy_net.state_dict(), file_name)`. Poslije se stanje mreže može pomoću `self._policy_net.load_state_dict(torch.load(file_path))` ponovno učitati u mrežu.

4.4 Rezultati

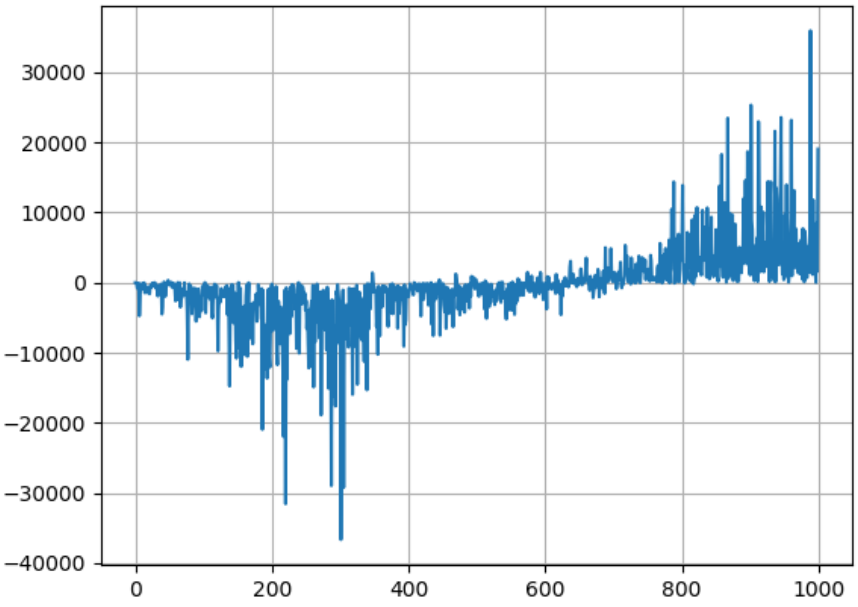
Graf nagrade je prikazan kao kumulativni zbroj dobivenih, odnosno izgubljenih, žetona. Mijenjanje raznih hiperparametara je dovelo do raznih rezultata koji se prikazuju u Tensorboardu i vidljivi su na slici 10, također su se rezultati prenijeli na tensorboard.dev koje se nalaze na `https://tensorboard.dev/experiment/yCxUmVc1TQeV03tQgJCD1g/`.



Slika 10: Rezultati učenja

Od svih rezultata bi se moglo odvojiti jednog agenta kojemu je vidljivo da mu se nagrada, nakon nekog vremena, stalno poboljšava. Taj jedan agent na kraju treninga, koji se vrtio

u periodu od milijun epizoda, posjeduje najvišu nagradu. Kada bi se tom agentu napredak prikazao kao razlika kumulativne nagrade, njegov graf bi izgledao kao na slici 11. Hiperparametri tog agenta su 10^{-4} za stopu učenja, 0.999 za stopu popusta buduće nagrade te 10^{-6} za propadanje stope istraživanja u svakoj epizodi do 0.1 tijekom treninga.



Slika 11: Napredak kao razlika nagrade

U igri protiv dva agenta neuronska mreža jako loše igra prije treninga, međutim nakon milijun epizoda treninga agent s neuronskom mrežom je u stanju pobijediti u skoro 60% slučajeva, što je prikazano slikom 12.

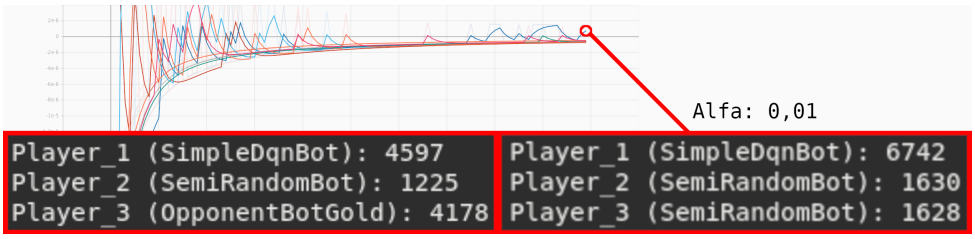


Slika 12: Usporedba agenta prije i poslije treninga

4.5 Napredak

Nadalje su se pokretali pokusi napredovanja mreže, tako da agent uči igrajući protiv protivnika koji donose odluke o sljedećoj akciji na osnovi jačine trenutne ruke. Svi pokusi sa različitim hiperparametrima su bili neuspješni. Detaljnim uvidom u izbor akcije nakon treniranja, primjetilo se da neuronska mreža pretežito izabere akciju *fold*. Znači mreža nije uspješno naučila igrati igru, te je zaključila da sa akcijom *fold* najduže ostaje u igri. Iz tog razloga se

odlučilo promijeniti algoritam učenja. U algoritam su se dodale nove mogućnosti kako bi se mogle pokušati razne vrste treninga. Sada je moguće umjesto jednog potpunog treninga, te provjere nakon toga, trening periodično prekinuti i provjeriti agentov napredak. Moguće je pratiti napredak na tensorboardu tokom treninga i/ili validacije. Dodatno postoji i mogućnost da pamti najbolji napredak tokom treninga i u slučaju da agent igra bolje to će se novo stanje mreže pohranjivati. Na taj način ostaje pohranjeno agentovo najbolje stanje. Ulaz u mrežu se promijenio tako da se odvojeno šalju karte koje su na stolu i karte koje su u ruci. Uz to su se dodale dozvoljene akcije i vrsta trenutne ruke u ulaz mreže, a izbacila se informacija o količini žetona i mogućnost povećavanja uloga. Agent sa najboljim rezultatom prikazan je na slici 13, gdje je napredak bilježen kao ukupni prosjek dobivenih žetona u tom trenutku. Vidljivo je da je agent na kraju sveukupno više dobio žetona nego ih je izgubio. Još se prikazuje validacija protiv dva polunasumična agenta, gdje je pobjedio u skoro 70% slučajeva. Isto tako nadmašuje igru protiv polunasumičnog agenta i agenta sa heuristikom.



Slika 13: Rezultati konačne validacije

Dodatni pokusi obuhvaćaju:

- Neuronska mreža sa 3 skrivena sloja: Dodao se još jedan sloj neurona, za povećavanje neuronske mreže. Neuronska mreža se može povećati u širinu i u dubinu. U širinu znači povećavanje broja neurona u pojedinim skrivenim slojevima a u dubinu povećavanje broja skrivenih slojeva.

- Heuristično istraživanje: Umjesto nasumične odluke tokom istraživanja, heuristični agent donosi odluku. Agenta kroz trening vodi heuristični agent, što u jednu ruku ubrzaje trening ali u drugu ga ograničava.

- Zajednička neuronska mreža: Više agenata koriste istu neuronsku mrežu, tako da se mreža u jednoj epizodi više puta ažurira.

Niti jedan od dodatnih pokusa nije donio znatno poboljšanje. Trening više agenata sa

zajedničkom neuronskom mrežom se prekinuo, pošto se pojavio već poznati problem u kojemu mreža zaključi da najduže ostaje u igri kada stalno izvrši akciju *fold*. Situacija pred samim krajem treninga je bila takva da svi igrači uvijek izvršavaju akciju *fold*, prisilni ulozi su kružili od igrača do igrača i epizoda nije imala kraja.

5 Zaključak

Područje umjetne inteligencije je jako zanimljivo područje koje je još u razvoju. Zbog toga još nema čvrstih pravila kojima se garantira dobar rezultat, nego postoji puno pristupa koje je potrebno prilagoditi za određeni problem, a za visoku razinu uspješnosti potrebno je pokušati više različitih pristupa ili različitih kombinacija pristupa. Upute za rješavanje neke problematike s umjetnom inteligencijom se mogu pronaći u raznim oblicima, no u takvim uputama se često radi sa vrlo ograničenom domenom, što bitno olakšava njihovu primjenu. Osim slaganja arhitekture modela za učenje, potrebno je uložiti i dosta vremena u pripremu podataka za učenje, jer ih se može dobiti i prikazati u raznim oblicima. Osim toga potrebno je pogoditi prikladne hiperparametre, za koje također nema čvrstih pravila, nego se radi na osnovi pokušaja i neuspjeha. Naravno, vremenom se dobije neka intuicija kako otprilike složiti neke stvari, ali i dalje se pokušavaju ostale kako bi ih se međusobno usporedilo. U ovom radu se vidjelo da uspješnost učenja uvelike ovisi o prikladnim hiperparametrima poput stope učenja ili arhitekture same neuronske mreže. Za bilo koju vrstu problema koja se mogu riješiti algoritmički bez umjetne inteligencije je obično brže koristiti algoritmičko riješenje nego pomoću umjetne inteligencije učiti novo rješenje. Što se tiče implementacije igre Texas Hold'em, ostaje još dosta prostora za daljnja istraživanja i poboljšanja. Jako je kompleksna igra koja kombinira sreću, poznavanje igre i procjenjivanje protivničke reakcije. Svakako je moguće složiti neuronsku mrežu koja uspješno igra ovu igru na visokoj razini. Pokazalo se da se s jednom jednostavnom arhitekturom može smisleno igrati, što je, u stvari, i sama bit ovog rada.

Literatura

[1] Richard S. Sutton & Andrew G. Barto, *Reinforcement learning: An introduction*, http://www.incompleteideas.net/book/RLbook2018.pdf, 2018, (posjećeno: 29.11.2020).

[2] Deeplizard, *Reinforcement learning - goal oriented intelligence*, https://deeplizard.com/learn/playlist/PLZbbT5o_s2xoWNVdDudn51XM8lOuZ_Njv, 2018, (posjećeno: 29.11.2020).

[3] David Skalinsky & Mason Malmuth, *Starting hand groups*, https://www.thepokerbank.com/strategy/basic/starting-hand-selection/sklansky-groups/, 1999, (posjećeno: 29.11.2020).

[4] Darse Billings, Denis Papp, Jonathan Schaeffer, Duane Szafron, *Opponent modeling in poker*, http://www.cs.virginia.edu/~evans/poker/wp-content/uploads/2011/02/opponent_modeling_in_poker_billings.pdf, 1998, (posjećeno: 29.11.2020).

[5] Wikipedia, *Artificial intelligence*, https://en.wikipedia.org/wiki/Artificial_intelligence, 2020, (posjećeno: 29.11.2020).

6 Dodatak

```
from game import Utils
import torch
from torch import nn
from torch.utils.tensorboard import SummaryWriter
from typing import Dict



class Monitor:
    def __init__(self, model: nn.Module,
                 monitor_comment_params: Dict[str, str]) -> None:
        self._model = model
        self._comment_params = monitor_comment_params
        self._progress = 0
        self._step = 0
        self._summary_writer = None
        self._device = next(model.parameters()).device

        self._init_summary_writer()

    def _init_summary_writer(self) -> None:
        dummy_input = self._generate_dummy_input()
        log_dir_path = self._generate_log_dir_path()

        self._summary_writer = SummaryWriter(log_dir=str(log_dir_path))
        self._summary_writer.add_graph(self._model, dummy_input)

    def _generate_dummy_input(self) -> torch.Tensor:
        dummy_data = [0 for _ in range(self._model[0].in_features)]
        return torch.as_tensor(dummy_data, dtype=torch.float32).to(self._device)

    def _generate_log_dir_path(self) -> str:
        comment = self._generate_comment()
        return str(Utils.get_base_dir().joinpath('runs').
                   joinpath(f'{Utils.get_now_as_str()}: {comment}'))

    def _generate_comment(self) -> str:
        comment = ''
        params_amount = len(self._comment_params)
```

```
        for i, k in enumerate(self._comment_params.keys()):
            comment += f'{k}={self._comment_params.get(k)}'
            if i < params_amount - 1:
                comment += ', '

        return comment

    def monitor(self) -> None:
        progress = self._progress
        self._summary_writer.add_scalar('Progress', progress, self._step)

        for name, param in self._model.named_parameters():
            self._summary_writer.add_histogram(name, param, self._step)
            self._summary_writer.add_histogram(f'{name}_grad', param.grad,
                                               self._step)

        self._step += 1

    def update_progress(self, new_progress: int) -> None:
        self._progress = new_progress

    def close(self) -> None:
        self._summary_writer.close()
```

Ispis 14: Klasa koja zapisuje na Tensorboard

Klasa `Monitor` pomoću klase `SummaryWriter` zapisuje napredak učenja agenta na Tensorboard.

```
from . import ObservableEnvironment as O_Env
from agent.base import Base
from random import random, randint
from typing import Tuple, Any, Optional
from datetime import datetime


class QLearning(Base):
    def __init__(self, training_amount: int = 100000):
        super(Base, self).__init__()
        self._q_table = {}
        self._previous_action = None
        self._current_state = None
        self._previous_state = None
        self._init_q_value = 0.0
        self._learning_rate = 0.3
        self._gamma = 0.9
        self._epsilon = 1.0
        self._epsilon_decay = self._epsilon / training_amount

    def prepare_for_episode(self) -> None:
        self._previous_action = None
        self._previous_state = None

    def observe_environment(self, env: O_Env) -> None:
        if env.state not in self._q_table.keys():
            self._add_state_to_q_table(env.state)

        self._update_q_table(env)
        self._current_state = env.state

        if env.is_terminal and self._epsilon > 0:
            self._epsilon -= self._epsilon_decay

    def choose_action(self, actions: Tuple[Any, ...]) -> Optional[Any]:
        for action in actions:
            if action not in self._q_table[self._current_state].keys():
                self._add_action_to_q_table(action)
```

```
    if random() < self._epsilon:
        action = self._choose_random_action(actions)
    else:
        action = self._choose_best_action(actions)

    self._previous_state = self._current_state
    self._previous_action = action

    return action

def _add_state_to_q_table(self, state: Tuple[int, ...]) -> None:
    self._q_table[state] = {}

def _add_action_to_q_table(self, action: Any) -> None:
    self._q_table[self._current_state][action] = self._init_q_value

def _update_q_table(self, env: O_Env) -> None:
    # Calculate new Q-Value (Bellman equation)
    if len(self._q_table[env.state].keys()) > 0 \
            and self._previous_state is not None \
            and self._previous_action is not None:
        current_action = self._get_actions_by_value(env.state)[0]
        old_q = self._q_table[
            self._previous_state][self._previous_action]

        new_q = (1 - self._learning_rate) * old_q + \
                self._learning_rate * (env.reward + self._gamma *
                                       self._q_table[env.state][current_action])

        self._q_table[self._previous_state][self._previous_action] = new_q
    elif env.is_terminal:
        self._q_table[self._previous_state][self._previous_action] = \
            env.reward

@staticmethod
def _choose_random_action(actions: Tuple[Any, ...]) -> Any:
    rnd = randint(0, len(actions) - 1)
```

```
        return actions[rnd]

    def _choose_best_action(self, actions: Tuple[Any, ...]) -> Any:
        actions_by_value = self._get_actions_by_value(self._current_state)
        h_value = self._q_table[self._current_state][actions_by_value[0]]
        h_actions = []
        for action in actions_by_value:
            if self._q_table[self._current_state][action] == h_value \
                    and action in actions:
                h_actions.append(action)

        if len(h_actions) > 0:
            rnd = randint(0, len(h_actions) - 1)
            return h_actions[rnd]

        return None

    def _get_actions_by_value(self, state: Tuple[int, ...]) -> \
            Tuple[Any, ...]:
        return tuple(
            sorted(
                self._q_table[state].keys(),
                key=lambda action: self._q_table[state][action],
                reverse=True)
        )
```

Ispis 15: Standardno q-učenje

Metoda `prepare_for_episode` priprema agenta za novu epizodu i u ovom slučaju samo poništava vrijednosti koje su bile pohranjene u varijablama koje predstavljaju prethodno stanje i akciju. Nadalje metoda `observe_environment` dodaje stanje u tablicu ako je prvi put posjećena, poziva funkciju `_update_q_table` i ažurira varijable trenutnog stanja i stope istraživanja. U ovoj cijeloj klasi najzanimljivija je funkcija `_update_q_table` u kojoj se pomoću Bellmanove jednadžbe optimalnosti ažuriranja ćelija u tablici. Još među bitnim metodama je `choose_action`, koja ovisno o tome dali se istražuje ili iskorištava stanje donese odluku o sljedećoj akciji.

```
from copy import deepcopy
from game import Card, FinalHandType, Moves, Player as Basic_Player, State
from typing import List, Optional


class Players:
    def __init__(self, basic_player: Basic_Player,
                 name: str = 'Player') -> None:
        self._basic_player = basic_player
        self._basic_player_type = basic_player.__class__.__name__
        self._name = name
        self._current_bet = 0
        self._total_bet = 0
        self._score = 0
        self._is_active = True
        self._current_move = None
        self._final_hand = None
        self._final_hand_type = None
        self._next = None

    @property
    def player_type(self) -> str:
        return self._basic_player_type

    @property
    def name(self) -> str:
        return self._name

    @property
    def current_bet(self) -> int:
        return self._current_bet

    @current_bet.setter
    def current_bet(self, bet: int) -> None:
        self._current_bet = bet

    @property
    def total_bet(self) -> int:
```

```
        return self._total_bet

    @total_bet.setter
    def total_bet(self, bet: int) -> None:
        self._total_bet = bet

    @property
    def current_move(self) -> Moves:
        return self._current_move

    @current_move.setter
    def current_move(self, move: Moves) -> None:
        self._current_move = move

    @property
    def final_hand(self) -> List[Card]:
        return self._final_hand

    @final_hand.setter
    def final_hand(self, hand: List[Card]) -> None:
        self._final_hand = hand

    @property
    def final_hand_type(self) -> FinalHandType:
        return self._final_hand_type

    @final_hand_type.setter
    def final_hand_type(self, hand_type: FinalHandType) -> None:
        self._final_hand_type = hand_type

    @property
    def score(self) -> int:
        return self._score

    @score.setter
    def score(self, score: int) -> None:
        self._score = score

    @property
```

```
def is_active(self) -> bool:
    return self._is_active

@is_active.setter
def is_active(self, state: bool) -> None:
    self._is_active = state

@property
def next(self) -> 'Players':
    return self._next

@next.setter
def next(self, player: 'Players') -> None:
    self._next = player

def receive_cards(self, cards: List[Card]) -> None:
    self._basic_player.receive_cards(cards)

def get_amount_of_chips(self) -> int:
    return self._basic_player.get_amount_of_chips()

def spend_chips(self, amount: int) -> int:
    return self._basic_player.spend_chips(amount)

def make_move(self, possible_moves: List[Moves],
              game_state: State) -> Moves:
    return self._basic_player.make_move(possible_moves, game_state)

def get_hand(self) -> List[Card]:
    return self._basic_player.get_hand()

def receive_chips(self, amount: int) -> None:
    self._basic_player.receive_chips(amount)

def destroy_hand(self) -> None:
    self._basic_player.destroy_hand()

def reset(self) -> None:
    self._basic_player.destroy_hand()
```

```
        self._current_move = None
        self._current_bet = 0
        self._total_bet = 0
        self._final_hand = None
        self._final_hand_type = None
        self._score = 0
        self._is_active = True

    def append(self, player: 'Players') -> None:
        tmp = self
        is_closed_list = False

        while tmp.next is not None:
            if tmp.next == self:
                is_closed_list = True
                tmp.next = None
            else:
                tmp = tmp.next

        tmp.next = player

        if is_closed_list:
            player.next = self

    def get_by_position(self, position: int = 1) -> 'Players':
        next_player = self

        for _ in range(position):
            next_player = next_player.next

        return next_player

    def remove_player(self, player: 'Players') -> None:
        for p in self:
            if p.next == player:
                p.next = player.next
                player.next = None

    def find(self, player_to_find: 'Players') -> Optional['Players']:
```

```
        if player_to_find == self:
            return player_to_find

        player = self._next

        while player != player_to_find:
            if player == self or player is None:
                return None

            player = player.next

        return player

    def find_by_move(self, move: Moves) -> List['Players']:
        players = []

        for player in self:
            if player.current_move is move:
                players.append(player)

        return players

    def count(self) -> int:
        player = self._next
        cnt = 1

        while player != self and player is not None:
            cnt += 1
            player = player.next

        return cnt

    def count_active(self) -> int:
        active_amount = 0

        for player in self:
            if player._is_active:
                active_amount += 1
```

```
        return active_amount

    def clone(self) -> 'Players':
        return deepcopy(self)

    def __iter__(self) -> 'Players':
        self._tmp = self
        self._has_head_been_returned = False
        return self

    def __next__(self) -> 'Players':
        if self._tmp is not None:
            if self._tmp == self:
                if not self._has_head_been_returned:
                    self._has_head_been_returned = True
                else:
                    raise StopIteration

            tmp = self._tmp
            self._tmp = self._tmp.next

            return tmp
        else:
            raise StopIteration
```

Ispis 16: Pomoćna klasa igrači za stolom

Implementirana je stilom klasične povezane liste, s kojom se omogućuje zatvaranje kruga, odnosno, igrač nakon posljednoga je prvi. Dodatno na ovaj način nije potrebno pratiti tko je trenutno diler, jer je to uvijek glava liste. Za intuitivno korištenje ove klase u petljama potrebno je implementirati Pythonove metode `__iter__` i `__next__` koje se brinu i o tome da petlja završava s posljednjim igračem. Podaci koje klasa čuva za svakog igrača su: referenca na objekt igrača, ime klase od tog objekta, ime igrača, trenutni ulog, sveukupni ulog, bodove (na kraju svakog kruga ovisno o jačini ruke dobije se određeni broj bodova), da li je aktivan (u smislu trenutnog kruga, da li aktivno sudjeluje u ulaganju/povećavanju uloga itd.), posljednji doneseni potez, krajnja ruka i kojeg je tipa ta ruka. Ova klasa implementira povezanu listu tako da ima referencu na sljedećeg igrača (što olakšava držanje redoslijeda igrača te mijenjanje dilera).

```
from . import FinalHandType, FinalHand
from game import Card
from typing import Dict, List, Optional


class StrongestFinalHandFinder:
    @staticmethod
    def find(cards: List[Card]) -> FinalHand:
        hand = sorted(cards, reverse=True)
        final_hand = StrongestFinalHandFinder._try_find_flush(hand)

        if final_hand is not None:
            bigger_hand = StrongestFinalHandFinder._try_find_straight(
                final_hand)
            if bigger_hand is not None:
                if StrongestFinalHandFinder._is_royal_flush(bigger_hand):
                    return StrongestFinalHandFinder._create_final_hand(
                        bigger_hand, FinalHandType.ROYAL_FLUSH)
                else:
                    return StrongestFinalHandFinder._create_final_hand(
                        bigger_hand, FinalHandType.STRAIGHT_FLUSH)
            else:
                return StrongestFinalHandFinder._create_final_hand(
                    final_hand, FinalHandType.FLUSH)

        final_hand = StrongestFinalHandFinder._try_find_poker(hand)
        if final_hand is not None:
            return StrongestFinalHandFinder._create_final_hand(
                final_hand, FinalHandType.POKER)

        final_hand = StrongestFinalHandFinder._try_find_full_house(hand)
        if final_hand is not None:
            return StrongestFinalHandFinder._create_final_hand(
                final_hand, FinalHandType.FULL_HOUSE)

        final_hand = StrongestFinalHandFinder._try_find_straight(hand)
        if final_hand is not None:
            return StrongestFinalHandFinder._create_final_hand(
```

```
            final_hand, FinalHandType.STRAIGHT)

    final_hand = StrongestFinalHandFinder._try_find_tris(hand)
    if final_hand is not None:
        return StrongestFinalHandFinder._create_final_hand(
            final_hand, FinalHandType.TRIS)

    final_hand = StrongestFinalHandFinder._try_find_two_pair(hand)
    if final_hand is not None:
        return StrongestFinalHandFinder._create_final_hand(
            final_hand, FinalHandType.TWO_PAIRS)

    final_hand = StrongestFinalHandFinder._try_find_pair(hand)
    if final_hand is not None:
        return StrongestFinalHandFinder._create_final_hand(
            final_hand, FinalHandType.PAIR)

    return StrongestFinalHandFinder._create_final_hand(
        hand[:5], FinalHandType.HIGH_CARD)

@staticmethod
def find_stronger_hand(final_hand1: FinalHand,
                       final_hand2: FinalHand) -> Optional[FinalHand]:
    if final_hand1.score == final_hand2.score:
        hand1 = sorted(final_hand1.hand, reverse=True)
        hand2 = sorted(final_hand2.hand, reverse=True)

        nbr_of_cards = len(hand1) if len(hand1) < len(hand2) else len(hand2)

        for i in range(nbr_of_cards):
            if hand1[i].value > hand2[i].value:
                return final_hand1

            if hand2[i].value > hand1[i].value:
                return final_hand2

        return None

    elif final_hand1.score > final_hand2.score:
```

```
            return final_hand1

        else:
            return final_hand2

    @staticmethod
    def _create_final_hand(hand: List[Card],
                           hand_type: FinalHandType) -> FinalHand:
        if hand_type is FinalHandType.TWO_PAIRS \
                or hand_type is FinalHandType.FULL_HOUSE:
            score = hand[0].value * hand_type.value + hand[3].value
        else:
            score = hand[0].value * hand_type.value

        return FinalHand(hand=hand, type=hand_type, score=score)

    @staticmethod
    def _try_find_flush(sorted_cards: List[Card]) -> Optional[List[Card]]:
        suits = dict()

        for card in sorted_cards:
            if card.suit not in suits:
                suits[card.suit] = list()

            suits[card.suit].append(card)

        for suit in suits:
            if len(suits[suit]) >= 5:
                return suits[suit][:5]

        return None

    @staticmethod
    def _try_find_straight(sorted_cards: List[Card]) -> Optional[List[Card]]:
        cnt = 0
        previous_card = sorted_cards[0]

        for i in range(1, len(sorted_cards)):
            if sorted_cards[i].value != previous_card.value - 1:
```

```
            cnt = 0
        else:
            cnt += 1
            if cnt == 4:
                return sorted_cards[i - 4:i + 1]

        previous_card = sorted_cards[i]

    if cnt == 3 \
            and sorted_cards[0].value == 13 \
            and sorted_cards[-1].value == 1:
        return sorted_cards[len(sorted_cards) - 4:-1] + [sorted_cards[0]]

    return None

@staticmethod
def _is_royal_flush(cards: List[Card]) -> bool:
    return (cards[0].value == 13
            and cards[1].value == 12
            and cards[2].value == 11
            and cards[3].value == 10
            and cards[4].value == 9
            and cards[0].suit
            == cards[1].suit
            == cards[2].suit
            == cards[3].suit
            == cards[4].suit)

@staticmethod
def _try_find_poker(sorted_cards: List[Card]) -> Optional[List[Card]]:
    same_value_cards = StrongestFinalHandFinder._find_same_value_cards(
        sorted_cards, 4)

    if same_value_cards is not None:
        high_value_card = StrongestFinalHandFinder. \
            _find_high_value_cards_not_in_cards(same_value_cards,
                                                sorted_cards, 1)
        return same_value_cards + high_value_card
```

```
        return None

    @staticmethod
    def _find_same_value_cards(cards: List[Card],
                               amount: int) -> Optional[List[Card]]:
        cards_by_value = StrongestFinalHandFinder. \
            _divide_cards_by_value(cards)

        for same_value_cards in cards_by_value:
            if len(cards_by_value[same_value_cards]) >= amount:
                return cards_by_value[same_value_cards][:amount]

        return None

    @staticmethod
    def _divide_cards_by_value(cards: List[Card]) -> Dict[int, List[Card]]:
        same_value_cards = dict()

        for card in cards:
            if card.value not in same_value_cards:
                same_value_cards[card.value] = list()

            same_value_cards[card.value].append(card)

        return same_value_cards

    @staticmethod
    def _find_high_value_cards_not_in_cards(
            selected_cards: List[Card], cards: List[Card],
            amount: int) -> List[Card]:
        high_value_cards = list()
        for card in cards:
            if card not in selected_cards:
                high_value_cards.append(card)
                if len(high_value_cards) == amount:
                    break
        return high_value_cards

    @staticmethod
```

```
def _try_find_full_house(
        sorted_cards: List[Card]) -> Optional[List[Card]]:
    same_3_cards = StrongestFinalHandFinder. \
        _find_same_value_cards(sorted_cards, 3)

    if same_3_cards is not None:
        same_2_cards = StrongestFinalHandFinder._find_same_value_cards(
            [card for card in sorted_cards if card not in same_3_cards], 2)

        if same_2_cards is not None:
            return same_3_cards + same_2_cards

    return None

@staticmethod
def _try_find_tris(sorted_cards: List[Card]) -> Optional[List[Card]]:
    same_value_cards = StrongestFinalHandFinder. \
        _find_same_value_cards(sorted_cards, 3)

    if same_value_cards is not None:
        high_value_cards = StrongestFinalHandFinder. \
            _find_high_value_cards_not_in_cards(same_value_cards,
                                                sorted_cards, 2)
        return same_value_cards + high_value_cards

    return None

@staticmethod
def _try_find_two_pair(sorted_cards: List[Card]) -> Optional[List[Card]]:
    first_pair = StrongestFinalHandFinder._find_same_value_cards(
        sorted_cards, 2)

    if first_pair is not None:
        second_pair = StrongestFinalHandFinder._find_same_value_cards(
            [card for card in sorted_cards if card not in first_pair], 2)

        if second_pair is not None:
            high_value_cards = StrongestFinalHandFinder.\
                _find_high_value_cards_not_in_cards(first_pair + second_pair,
```

```
                                        sorted_cards, 1)
            return first_pair + second_pair + high_value_cards

        return None

    @staticmethod
    def _try_find_pair(sorted_cards: List[Card]) -> Optional[List[Card]]:
        same_value_cards = StrongestFinalHandFinder._find_same_value_cards(
            sorted_cards, 2)

        if same_value_cards is not None:
            high_value_cards = StrongestFinalHandFinder.\
                _find_high_value_cards_not_in_cards(same_value_cards,
                                                    sorted_cards, 3)
            return same_value_cards + high_value_cards

        return None
```

Ispis 17: Klasa za pronalaženje najjače ruke

Ova se klasa uglavnom sastoji od statičkih metoda i nema potrebe da bilo što drži za sebe. Prima proizvoljan niz karata i vraća kombinaciju karata koja predstavlja najjaču ruku. Uspoređuje ruku s najjačom prema najslabijom, tako da u trenutku kada se pronađe odgovarajuća ruka nije više potrebno dalje povjeravati nego se ju vrati. Nije potrebno inicijalizirati objekt ove klase nego se funkcije direktno pozivaju preko klase.

```
from game.player.dqn import ReplayMemory
import unittest
from unittest import TestCase


class TestReplayMemory(TestCase):
    def setUp(self) -> None:
        super().setUp()
        self._mem_replay = ReplayMemory()

    def test_initialization(self) -> None:
        mem_size = 64000
        batch_size = 128

        mem_replay = ReplayMemory(memory_size=mem_size, batch_size=batch_size)

        self.assertEqual(mem_size, mem_replay.memory_size)
        self.assertEqual(batch_size, mem_replay.batch_size)

    def test_insertion(self) -> None:
        experience = ((1, 2, 4), ('a', 'b', 'r'), (71, '000', True))
        self._mem_replay.insert(experience)

        memory = self._mem_replay._memory[0]

        self.assertEqual(experience, memory)

    def test_insertion_at_beginning_on_overflow(self):
        reply_mem = ReplayMemory(memory_size=2, batch_size=1)

        first = (('testing', 'replay memory'), ('insertion', 'on overflow'))
        second = ((111, 222, 333), (True, False, True),
                  ([44, 55], [66, 77, 88]))
        third = ('x', 'y', 'z')

        reply_mem.insert(first)
        reply_mem.insert(second)
        reply_mem.insert(third)
```

```
        reply_mem.insert(first)

        memory_one = reply_mem._memory[0]
        memory_two = reply_mem._memory[1]

        self.assertEqual(third, memory_one)
        self.assertEqual(first, memory_two)

        reply_mem.insert(second)
        memory_three = reply_mem._memory[0]

        self.assertEqual(second, memory_three)

    def test_batch_size_greater_than_memory_size(self) -> None:
        memory_size = 32
        batch_size = 64

        self.assertRaises(Exception, ReplayMemory, memory_size=memory_size,
                          batch_size=batch_size)

    def test_can_sample(self) -> None:
        replay_mem = ReplayMemory(memory_size=8, batch_size=2)

        memory_one = (1, 2, 3)
        memory_two = ('a', 'b', 'c')

        replay_mem.insert(memory_one)
        self.assertFalse(replay_mem.can_sample())

        replay_mem.insert(memory_two)
        self.assertTrue(replay_mem.can_sample())

    def test_sample(self) -> None:
        memory_size = 8
        batch_size = 2
        replay_mem = ReplayMemory(memory_size, batch_size)
        memories = [i for i in range(10)]

        replay_mem.insert(memories[0])
```

```
        self.assertRaises(Exception, replay_mem.get_sample)

        replay_mem.insert(memories[1])

        batch = replay_mem.get_sample()
        self.assertEqual(batch_size, len(batch))

        for i in range(2, 10):
            replay_mem.insert(memories[i])

        batch = replay_mem.get_sample()
        self.assertEqual(batch_size, len(batch))

    def test_flush_memory(self) -> None:
        first = 1
        second = 'x'
        third = {'1x': 10}

        self._mem_replay.insert(first)
        self._mem_replay.insert(second)
        self._mem_replay.insert(third)

        flushed = self._mem_replay.flush()

        self.assertEqual([first, second, third], flushed)
        self.assertTrue(len(self._mem_replay._memory) == 0)


if __name__ == '__main__':
    unittest.main()
```

Ispis 18: Primjer test jedinica