Poticano učenje te njegova primjena u modeliranju igrača za računalnu igru Doom

Vuk, Ilija

Master's thesis / Diplomski rad

2023

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Organization and Informatics / Sveučilište u Zagrebu, Fakultet organizacije i informatike**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:211:408801

SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
VARAŽDIN

Ilija Vuk

POTICANO UČENJE TE NJEGOVA PRIMJENA U MODELIRANJU IGRAČA ZA RAČUNALNU IGRU DOOM

DIPLOMSKI RAD

Varaždin, 2023.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Ilija Vuk

Matični broj: 46198/17–R

Studij: Informacijsko i programsko inženjerstvo

POTICANO UČENJE TE NJEGOVA PRIMJENA U MODELIRANJU IGRAČA ZA RAČUNALNU IGRU DOOM

DIPLOMSKI RAD

Mentor:

Dr. sc. Bogdan Okreša Đurić

Varaždin, rujan 2023.

Ilija Vuk

Izjava o izvornosti

Izjavljujem da je moj diplomski rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor potvrdio prihvaćanjem odredbi u sustavu FOI-radovi

Sažetak

U ovom radu obrađena je tema poticanog učenja. Rad opisuje proces implementacije poticanog učenja na stvarnom primjeru. Za izvedbu tog procesa potrebno je kreirati tzv. "Wrapper" klasu koja služi kao omotač oko okoline u kojoj agent djeluje. Pomoću wrapper klase definiramo sučelje pomoću kojeg agent može utjecati na svoju okolinu. Osim wrapper klase, potreban nam je i agent. Pošto se rad bazira na poticanom učenju, agent uči igrati igru kroz metode poticanog učenja koje su detaljnije opisane u radu.

Za izradu ovog rada korišteno je nekoliko alata i programa. Sami kod pisan je i izvođen u Visual Studio Code-u. Za implementaciju korištena je Python u posljednjoj verziji Python-a u vrijeme pisanja ovog rada (v.3.10.8). Za upravljanje Python bibliotekama korišteno je Anaconda Notebook okružje. Biblioteke koje je vrijedno istaknuti su Tensorflow v2.0, OpenAI Gym, matplotlib, te ViZDoom. Tensorflow te OpenAI Gym su besplatne, open-source, biblioteke koje služe za strojno učenje i umjetnu inteligenciju. Matplotlib je biblioteka koja omogućava iscrtavanje grafova u Pythonu. Možda i najvažnija biblioteka je ViZDoom. Ona nam omogućava kreiranje instance Doom videoigre. Osim toga olakšava nam rad s videoigrom jer nam olakšava kretanje u igri korištenjem agenta. Uz to je dostupan i veliki broj scenarija kreiranih od kreatora biblioteke u kojima možemo trenirati svog agenta.

Ključne riječi: Strojno učenje; Poticano učenje; Umjetna inteligencija; Python; Doom; Samostalni agent

Sadržaj

1. Uvod

Prvo pojavljivanje riječi inteligencija (engl. *Intelligence*) u zabilježenoj povijesti, kako navodi Middle English Dictionary [1], je krajem 14. stoljeća u knjizi „Canterbury Tales„ koju je napisao Geoffrey Chaucer. Geoffrey u ovom djelu nekoliko puta likove opisuje kao inteligentne, te tu riječ koristi u približnom značenju riječi koje ta riječ podrazumijeva danas.

Najraniji tekst u kojem se pojavljuje opis nečega što bismo mogli shvatiti kao inteligenciju je Aristotelovo djelo iz 4. st. pr. Kr. pod nazivom "Nichomachean Ethics" [Aristotle. (1980). Nicomachean ethics. Translated by T. Irwin. Indianapolis, IN: Hackett. (Original work written in 4th century BCE)]. U knjizi (Book VI, 1139a) objašnjava razliku između teoretskog razuma i praktičnog razuma te postavlja temelje za svoju filozofiju o etici i moralnosti.

Razum o kojem Aristotel govori je ipak nešto drukčiji od današnjeg shvaćanja inteligencije. Taj pojam ni dan danas nema striktno definirano značenje te se često kaže da je inteligencija jednostavno ono što se mjeri IQ testom. No, iako ne postoji konkretna definicija, često se taj pojam može svesti na sposobnost prilagodbe na nove okoline, kao i sposobnost učenja iz iskustava. Iz same definicije inteligencije (ili njenog nedostatka) vidimo da ona nije ograničena na ljudska bića.

Razvitkom čovječanstva ljudi su počeli promišljati i o umjetnoj inteligenciji (engl. *Artificial Intelligence*). Preduvjet za umjetnu inteligenciju bio je značajan razvitak tehnoloških znanosti koji se desio krajem prošlog stoljeća. Prvi spomen pojma umjetna inteligencija je 1956. godine. John McCarthy glasi kao izumitelj pojma umjetne inteligencije, a pojam je nastao tijekom radionice na Dartmouth koledžu gdje je John bio jedan od organizatora [2].

Cilj umjetne inteligencije, kako navodi John McCarthy 1955. godine [3], je razviti softver koji je sposoban donositi inteligentne odluke svojevoljno, bez vanjskog utjecaja ljudi. Cilj ovog rada je detaljnije se upoznati s metodologijama umjetne inteligencije, kao i izraditi inteligentnog agenta koji zna izabrati koju akciju želi napraviti kada se nađe u određenoj situaciji. Agent se mora snalaziti u okružju videoigre DOOM, te mora moći pobijediti u "deathmatch" scenariju.

2. Razrada teme

Sveučilište Stanford [4] navodi kako je Advice Taker vjerojatno prvi program koji je logiku koristio ne kao temu oko koje se program vrti, već kao način reprezentacije u memoriji računala. John McCarthy navodi da se ta razlika između njegovog programa i drugih programa (npr. Newell, Simon i Shawov "the Logic Theory Machine" i Gelernterov "the Geometry Program") može vidjeti u načinu na koji je program kreiran. U programima prije njegovog logika je bila pohranjena u samom programu, dok se u Advice Takeru logika nalazi u jeziku. Time se teoretski omogućava programu da sam formira nova znanja iz rečenica koje dobije kao ulaz. Advice Taker se vodi logikom prvog reda [5].

2.1. Strojno učenje

Strojno učenje je područje računalne znanosti koje se bavi razvojem računalnog softvera (agenata), koji mogu učiti iz podataka te iz tih podataka izvoditi zaključke [6]. Strojno učenje koristi se za različite zadatke, poput klasifikacije, regresije, klasterizacije, pa čak i za sintetiziranje novih podataka [7]. Algoritmi strojnog učenja koriste se u različitim industrijama, od trgovine do financija, od javnog sektora do zdravstva [8]. Strojno učenje je važan dio umjetne inteligencije i može se koristiti za rješavanje složenih problema koje bi bilo teško riješiti ručno. Prednost strojnog učenja je to što agent često uči samostalno. Agentu se ne mora eksplicitno programirati svaki djelić njegovog ponašanja što bi bilo potrebno u implementaciji programa bez uporabe tehnika umjetne inteligencije [9, str. 4].

Pojam agenta u literaturi ima konfliktne definicije. Konkretna definicija ovisi o mjestu na kojem se postavi granica između agenta i njegovog okružja.

Definicija 1 *Russell i Norvig [6] agenta definiraju kao entitet koji može promatrati svoje okružje kroz niz senzora te ima mogućnost interakcije sa okružjem kroz aktuatore [6, str. 34]. Agentovo ponašanje implementirano je uz pomoć agentovog programa. Agentov program je konkretna implementacija agentne funkcije (engl.* agent function*) koja uzima ulazne vrijednosti senzora te ih mapira na akciju. Sutton i Barto [10, str. 2] navode nešto generalniju definiciju. Agent mora moći vidjeti stanje u kojem se nalazi te mora moći donositi odluke koje utječu na samo stanje. Također mora imati ciljeve vezane za stanje okružja. Granica između agenta i okružje je stavljena nešto bliže agentu [10, str. 50]. U slučaju inteligentnog robota, njegovi motori, senzori te ostali mehanički dijelovi smatraju se dijelom okružja. Isto tako se i nagrade (iako su dijelom ukomponirane u samog agenta) smatraju eksternim. Za granicu između agenta i okružje vrijedi generalna ideja koja glasi: sve što agent ne može izmijeniti proizvoljno smatra se dijelom okružja.*

2.1.1. Nadzirano strojno učenje

Nadzirano strojno učenje je vrsta algoritama za strojno učenje. Jason Browniee [7] navodi da se nadzirano strojno učenje bazira na tome da imamo dva seta podataka. Prvi, veći

set, koristi se za treniranje agenta, dok se drugi, manji set, koristi za evaluaciju istreniranog agenta, odnosno kvalitete njegovih predviđanja. Veći set se sastoji od skupa podataka s već poznatom labelom. Ova vrsta učenja može se koristiti za nekoliko različitih zadataka.

Jedna od čestih svrha ovog učenja je klasifikacija [6, str. 696]. U tom slučaju model predviđa kategorizacijsku grupu kojoj objekt pripada. Jedan od početnih primjera koji početnici sretnu u ovoj vrsti učenja je korištenje MNIST baze podataka koja sadrži rukom pisane znamenke. Model u tom konkretnom zadatku iz dobivene slike određuje o kojoj se znamenci radi. MNIST baza podataka proširena je i drukčijim vrstama zapisa, kao na primjer baza podataka odjevnih predmeta. U rješavanju tog zadatka model uči povezati sliku odjevnog predmeta s njegovom oznakom. Nakon što nauči prepoznati odjevni predmet agent se testira na testnom setu podataka.

Drugi problem koji se rješava nadziranim strojnim učenjem je problem regresije. Pojam regresije izmislio je Sir Francis Galton [11], engleski polihistor čiji se utjecaj može vidjeti u psihologiji, biometrici, geografiji i mnogim drugim poljima. Osim svojeg značajnog doprinosa navedenim znanostima, značajno je doprinio i statistici, te se neki od njegovih pristupa koriste i dan danas. Michael Costello iz američke statističke udruge (engl. *American Statistic Association*) regresiju definira kao vezu između ovisnih varijabli (varijabla Y) i jedne ili više neovisnih varijabli (varijabla X). Regresijom se pokušava estimirati vrijednost varijable Y uz preddefinirane varijable X. Odnosno, pokušava se estimirati vrijednost funkcije f(x) = y [12]. Time je moguće odrediti jačinu veza između neovisnih varijabli i ovisne varijable. Regresija u smislu nadziranog strojnog učenja opisuje vrstu problema gdje je potrebno odrediti vrijednost Y ovisno o varijablama X. Kao klasični problem regresije uzima se problem predviđanja cijene nekretnina. U tom slučaju cijena nekretnine bila bi varijabla Y, dok bi varijable X bile lokacija, broj kvadrata, godina izgradnje, itd. Osim cijena nekretnina mogu se predviđati i mnoge druge brojčane vrijednosti kao što su na primjer postotak masnog tkiva osobe prema težini, visini, obujmu vrata, ruku i ostalim mjerama osobe[1].

2.1.2. Nenadzirano strojno učenje

Nenadzirano strojno učenje funkcionira na sličan način kao nadzirano strojno učenje. Kao i kod nadziranog strojnog učenja, agent dobiva dataset na kojem se izučava. Najveća razlika između nadziranog i nenadziranog učenja je zapravo u samom datasetu. Kod nadziranog učenja dataset za treniranje mora biti prethodno kategoriziran, odnosno svakom objektu u datasetu mora biti pridružena oznaka. Kod nenadziranog strojnog učenja podatci nisu prethodno kategorizirani, već algoritam sam mora pronaći veze između podataka.

J. Browniee navodi kako modeli iz dataseta deduciraju postojeće strukture podataka. Time se dobivaju generalizirana pravila koja vrijede za dataset. Moguće je smanjiti redundanciju u datasetu pronalaženjem najvažnijih stavki u datasetu te izbacivanjem onih koje su redundantne ili koje model proglasi neinformativnima. Treći pristup koji Browniee navodi je organiziranje podataka po sličnosti. Nenadzirano strojno učenje koristi se za rješavanje nekoliko vrsti problema kao npr. clustering, smanjenje dimenzionalnosti (engl. *dimensionality reduc-*

[1] https://www.kaggle.com/datasets/fedesoriano/body-fat-prediction-dataset

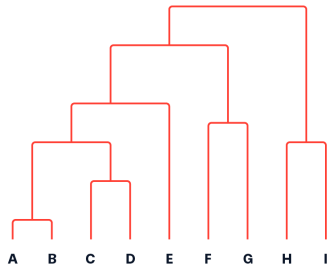
tion), pronalaženje pravila asocijacije (engl. *Association rule learning*) i detektiranje anomalija (engl. *Anomaly Detection*).

2.1.2.1. Klasteriranje

Klasteriranje (engl. *Clustering*) je metoda istraživačke analize podataka. Jain i Dubes [13] navode da cilj clusteringa nije generirati set pravila kojima dataset podliježe te s tim pravilima kategorizirati podatke. Clustering analizu opisuju kao zgodan način za pronaći validnu strukturu u podatcima. Cluster je skup nekoliko objekata koji su slični jedan drugom te su grupirani zajedno. Clustering metode se najčešće razvrstavaju u dvije skupine. To su centoridalne i hijerarhijske. Centroidalne metode kreiraju tzv. centroide koji se nalaze u sredini svakog clustera. Objekti u datasetu se zatim pridružuju najbližem centroidu. Hijerarhijske metode iz podataka generiraju strukturu u obliku stabla u kojem su clusteri ugniježđeni jedan u drugom. U hijerarhijskim metodama postoji još jedna kategorizacija prema načinu grupiranja kojim se clustering izvodi:

- Aglomerativno hijerarhijsko grupiranje
- Divizivno hijerarhijsko grupiranje

Aglomerativno hijerarhijsko grupiranje započinje s svakim objektom iz dataseta u svom odvojenom clusteru. Algoritam zatim iterativno spaja slične clustere sve dok ne dobijemo jedan cluster u obliku stabla. Divizivno hijerarhijsko grupiranje funkcionira na obrnuti način. Svi objektu u datasetu započinju u jednom clusteru te se iterativno odvajaju u podclustere. Rezultat obje metode je dendogram koji predstavlja hijerarhijsku strukturu podataka. Primjer dendograma može se vidjeti na slici ispod.



Slika 1: Prikaz dendograma, izvor: [14].

Postoji puno algoritama za clustering, scikit-learn [15] navodi k-means, DBSCAN, OPTICS, i još mnoge druge.

2.1.2.2. Smanjenje dimenzionalnosti

Smanjenje dimenzionalnosti je metoda u strojnom učenju koja pokušava smanjiti broj ulaznih varijabli u datasetu. Taj cilj je potrebno izvršiti uz zadržavanje što više informacija što je moguće. Česta svrha smanjenja dimenzionalnosti je jednostavno ubrzanje naknadnih transformacija dataseta. Algoritmi za smanjenje dimenzionalnosti pokušavaju pronaći najbitnije uzorke u podatcima, a to čine kroz jedan od dva načina:

- Feature selection
- Feature extraction

Feature selection odabire podset varijabli ovisno o odabranim kriterijima koje želimo ispuniti. Često je to relevantnost varijable u cijelom modelu. Feature extraction pak funkcionira na drukčiji način. Ovdje se iz kombinacija postojećih varijabli kreiraju nove varijable. Kombiniranjem varijabli se pokušava dobiti što veća varijancija podataka. Smanjenje dimenzionalnosti je čest korak prije daljnje obrade dataseta. Iz tog razloga je bitno provjeriti točnost podataka prije i poslije procesa smanjenja dimenzionalnosti kako bi osigurali da je točnost podataka ostala ista. Smanjenjem dimenzionalnosti se uvijek gubi određeni dio informacija, stoga je bitno provjeriti da smo izgubili samo varijable koje nisu relevantne.

2.1.2.3. Pronalaženje pravila asocijacije

Pronalaženje pravila asocijacija mogli bismo kategorizirati pod strojno učenje, ali i pod data mining. Naime, ono se često koristi u data miningu kako bi se pronašle veze između predmeta u datasetu. Problem koji se inače rješava s ovom metodom je problem košarice. Korištenjem pronalaženja pravila asocijacija moguće je pronaći pravila u datasetu koja prikazuju ponašanje "ako osoba kupi predmet A, tada kupi i predmet B". U ovoj metodi koriste se dva svojstva kojom se opisuje pravilo asocijacije između dva predmeta. To su povjerenje (engl. *confidence*) i podrška (engl. *support*). Povjerenje od 50% između predmeta A i B označava da u slučaju da osoba kupi predmet A, postoji 50% šanse da će kupiti i predmet B. Podrška od 2% posto označava da se u 2% od svih praćenih transakcija predmet A i predmet B kupuju zajedno. [16]

2.1.2.4. Detekcija anomalija

Anomalije označavaju odstupanje jedne instance u datasetu od ostatka podataka. Anomalije su, dakle, rijetke instance u kojima se dogodilo značajno odstupanje. Samim time možemo pretpostaviti da podatak koji značajno odstupa od ostatka podataka može biti ili pogreška u unosu/mjerenju podataka ili može biti događaji koji se treba daljnje istražiti. Primjeri takvih događaja su detekcija krađe kod praćenja transakcija na računu, detekcija zdravstvenih problema praćenjem anomalija u zdravstvenim mjerenjima, ili npr. praćenje računalnih mreža kako bi otkrili DDoS napade [17]. Algoritmi detekcije anomalija funkcioniraju tako da se model uči na

datasetu s "normalnim" podatcima, to jest podatci u datasetu ne sadrže anomalije. Iz tog razloga je detekcija anomalija od iznimne važnosti. Time agent s velikim pouzdanjem može za svaki budući događaj odrediti radi li se o anomaliji ili ne. U ovom slučaju izlazi do izražaja prednost nenadziranog strojnog učenja u odnosu na nadzirano strojno učenje. Ovaj problem bi bilo iznimno teško riješiti nadziranim strojnim učenjem jer bi tada trebali imati sve moguće anomalije kategorizirane u datasetu za treniranje.

2.1.3. Poticano učenje

Poticano učenje (engl. *Reinforcement Learning*) je grana koja postoji već duže vrijeme. To je metoda koja se bazira na prirodnom fenomenu poticanog učenja. Taj fenomen životinjski psiholozi istražuju već preko 70 godina. Poticano učenje opisuje mehanizam kojim se (u stvarnom svijetu) biće motivira kako bi pronalazilo hranu te naučilo izbjegavati opasnosti u budućnosti. Taj mehanizam primarno funkcionira kroz dvije kategorije, nagrade i kazne. U slučaju da životinja pojede obrok koji je iznimno bogat nutrijentima, mozak otpušta hormone kao što su dopamin te ostale hormone koji signaliziraju zadovoljstvo. Ti hormoni su nagrada zbog kojih životinja zna da su koraci koje je poduzela netom prije dobri za njeno preživljavanje.

Korijeni poticanog učenja počinju sa psiholozima koji su proučavali psihologiju životinja. Ivan Pavlov [18] je poznati psiholog koji je pokazao zanimljiv efekt gdje su životinje naučile povezivati okidače s budućim nagradama. Pavlov u svom dijelu koji je u ruskom originalu objavljen još 1899. godine opisuje svoje eksperimente sa psima i otkrivanje klasičnog uvjetovanja (engl. *Classical conditioning*). Eksperiment pokazuje pse kojima se prije svakog hranjenja zazvonilo zvonce. Psi su naučili povezivati zvuk zvonca (u ovom slučaju okidač) s pojmom hranjenja (nagrada). Nakon nekog vremena je psima refleksno na zvuk zvonca krenula slina, iščekivajući svoju nagradu [19]. Istraživanje poticanog učenja u psihologiji krenulo je 1898. godine kada je E. Thorndike postavio neke od temelja psiholoških principa. Tzv. "Zakon učinka" (engl. *The Law of Effect*) govori nam da su za ona ponašanja nakon kojih slijede ugodne, pozitivne, posljedice, šanse ponavljanja u budućnosti veća nego za ona ponašanja za koja to ne vrijedi. Isto tako, za ponašanja nakon kojih slijede negativne posljedice je vjerojatnost ponavljanja u budućnosti manja. Životinja želi minimizirati negativne posljedice, uz maksimiziranje pozitivnih posljedica [20].

Rad na poticanom učenju kao grani strojnog učenja započeo je kasne 1979. godine s veoma jednostavnom, no dotad zanemarenom idejom. Ideja je bila agent koji nešto želi te svoje ponašanje mijenja kako bi maksimizirao signal koji dobiva iz svog okružja. Poticano učenje je učenje što učiniti u kojoj situaciji. Odnosno odabir jedne akcije za dobiveni state kako bi se maksimizirala nagrada koju agent dobije od svog okruženja. Poticano učenje bazira se na principima MDP-a te na principima dinamičkog programiranja. Točnije, bazira se na verziji MDP-a koji je djelomično opažljiv. Kako bi mogli implementirati poticano učenje na određenom okruženju ono mora zadovoljavati određene uvjete. Ti uvjeti su:

- Agent ima mogućnost osjetiti u kojem se stanju nalazi
- Agent može poduzeti akcije koje mijenjaju njegovo stanje

- Agent ima cilj ili ciljeve vezane za stanje njegovog okružja

Sutton i Barto [10, str. 2] navode kako se svaka metoda koja je sposobna riješiti takve probleme može kategorizirati kao metoda poticanog učenja.

Za razliku od drugih metoda u strojnom učenju, poticano učenje ne služi za kategorizaciju podataka ili pronalaženje podložne strukture u podatcima. Poticano učenje jednostavno pokušava maksimizirati nagradu koju okružje vraća. Još jedna bitna razlika koju uočavamo u poticanom učenju, a da nije prisutna u drugim vrstama učenja je dilema istraživanja i iskorištavanja (engl. *Exploration-exploitation dilemma*) [6, str. 840]. Za maksimiziranje finalne nagrade potrebno je birati akcije koje nam donose najbolju nagradu, no, moguće je imati akciju koja za trenutno stanje ne donosi maksimalnu nagradu, ali u isto vrijeme maksimizira nagradu za cijelu epizodu. To znači da je ponekad potrebno odabrati akciju koja ne daje maksimalnu nagradu (engl. *reward*) za to stanje, već je potrebno otvarati i druge opcije stanja eksperimentirajući s drugim akcijama. U tom slučaju agent istražuje, dok se u slučaju odabira maksimalne nagrade agent odlučuje za opciju iskorištavanja. Agent ponekad mora istraživati kako bi se otključale potencijalne buduće nagrade. Dilema je dakle ta da nije moguće doći do optimalnog rješenja, odnosno optimalne politike ponašanja, sa samo jednim pristupom (iskorištavanje, ili istraživanje). Potrebno je koristiti kombinaciju oba pristupa u određenim situacijama.

Sutton i Barto [10, str. 4] navode kako je poticano učenje grana strojnog učenja koja je najbliža učenju kojeg ljudi iskuse. Samim time, poticano učenje obuhvaća mnoge discipline znanosti, kao što su matematika, statistika, psihologija te neuroznanost. Poticano učenje je dio većeg trenda u umjetnoj inteligenciji koja se vraća na učenje generaliziranih činjenica, za razliku od prikupljanja enormne količine znanja. U samim početcima poticanog učenja se pristup generalizacije otpisao pod pretpostavkom da je inteligencija zapravo samo skup specifičnih znanja, trikova te procedura. Isto tako, pretpostavljalo se da se do inteligencije može doći tako da se u program ugradi dovoljno relevantnih informacija. Metode koje su se bazirale na generalizacijama i učenju i istraživanju nazivale su se slabim metodama (engl. *weak methods*), dok su se metode bazirane na specifičnim znanjima nazivale jake metode (engl. *strong methods*) [10, str. 4]. Napretkom poticanog učenja vraćamo se pristupu generalnim principima (primjenjivim na široki broj inteligentnih ponašanja) u umjetnoj inteligenciji. Cilj takvog pristupa je otkriti manji set jednostavnijih principa pomoću koji su generalnijeg oblika te tako dozvoljavaju pravo učenje (učenje u kojem agent nešto sam nauči) za razliku od ugrađenog znanja.

Postoji nekoliko kategorizacija poticanog učenja. Neke od kategorizacija su prema vrsti učenja (učenje vrijednosti, učenje politike), prema vremenu ažuriranja vrijednosti (monte carlo metode - nakon cijele epizode, metode temporalne diferencije - nakon svakog timestepa), prema postojanju modela (bazirano na modelu ili tzv. model-free), te još mnoge druge [10]. S. Russell i P. Norvig [6] rade podjelu na Pasivno poticano učenje (engl. *Passive Reinforcement Learning*) i Aktivno poticano učenje (engl. *Active Reinforcement Learning*). No osim te podjele, postoje još mnoge:

- Postojanost modela (Bazirano na modelu (engl. *Model-based*) i bez modela (engl. *Model-free*)

- Metoda na kojoj se agent bazira (Monte-Carlo ili Temporal Difference metoda), odnosno sample-based ili bootstrapping
- Epizodičke ili kontinuirane
- Vrsta politike (On-policy ili off-policy)

2.1.3.1. Pasivno/Aktivno poticano učenje

Pasivno poticano učenje je jedna podvrsta poticanog učenja. U ovoj vrsti učenja agent nema interakciju sa svojim okružjem kako je dosad opisivano. Kod pasivnog poticanog učenja agent ima fiksnu politiku kojom se vodi te ju ne mijenja tijekom svog učenja [6, str. 832]. Agent jednostavno promatra kako se kreće kroz svoje okružje praćenjem predodređene fiksne politike. Tijekom svog učenja prati prijelaze iz jednog stanja u drugo te nagrade (dobivene procijenjenom vanjske politike) koje dobiva promjenom stanja. Time se evaluira politika te se može odrediti koliko je ta politika dobra. Pasivno poticano učenje koristi se u slučajevima kada ne želimo da agent sam istražuje okolinu već samo želimo vidjeti rezultat predodređene politike. Postoji nekoliko slučajeva u kojima bi to bilo poželjno ponašanje:

- Evaluiranje politike koju je kreirao čovjek - U ovom slučaju politiku generira stručnjak neke domene, te se pasivno poticano učenje koristi kako bi pronašli područja za poboljšanje
- Sustavi u kojima je sigurnost prioritet - ponekad se agentu ne smije dozvoliti samostalno ponašanje zbog opasnosti posljedica njegove akcije. Tada se agentu da preddefinirana politika u kojoj kontroliramo točan put kojim se on kreće.

Pasivno poticano učenje koristimo kada želimo samo evaluirati postojeću politiku. Cilj pasivnog poticanog učenja nije pronaći optimalnu politiku kroz metodu pokušaja i pogrešaka. Evaluacija se često radi pomoću metode vremenskog odmaka (engl. *Temporal Difference method*) [6, str. 837]. Ona omogućava evaluaciju trenutne politike bez odabira akcije.

Aktivno poticano učenje koristimo kada želimo postići upravo suprotno. Cilj agenta koji se vodi aktivnim podržanim učenjem je naučiti optimalnu politiku kroz metodu pokušaja i pogrešaka. Agent se kreće kroz svoje okružje te bira akcije bazirano na svom trenutnom razumijevanju svog okružja. Tada se agent vodi politikom koja se konstantno mijenja te uči preko ishoda koje dobije odabirom određene akcije. U aktivnom poticanom učenju cilj je maksimizirati kumulativnu nagradu.

2.1.3.2. Bazirano na modelu / Bez modela

Postoje dva različita pristupa pri dizajniranju agenta u kontekstu svijeta u kojem djeluje. U prvom pristupu je agentu potreban model njegovog okruženja kako bi učio. Primjeri takvih metoda su dinamičko programiranje te heurističko pretraživanje. Postoje i metode kojima nije potreban model okruženja te pod te metode spadaju Monte Carlo metode i Temporal Difference metode. Pod modelom okruženja podrazumijeva se sučelje pomoću kojeg agent može predvidjeti kako će okružje reagirati na njegovu akciju. Uz pomoć trenutnog stanja i odabrane akcije,

model generira predikciju stanja u kojem će se agent nalaziti te nagradu koju agent može očekivati. Sutton i Barto modele dijele na dvije kategorije [10, str. 160], to su distribucijski modeli (engl. *Distribution models*) te modeli uzorka (engl. *Sample models*)

Distribucijski modeli generiraju potpuni set mogućih stanja i vjerojatnosti da agent akcijom a prijeđe iz stanja s u stanje s' [21, str. 195]. Kako bi uspješno generirali potpuni set mogućih stanja, potrebna je detaljna reprezentacija okruženja u kojem se agent nalazi i dinamike prijelaza iz stanja u druga stanja. U kompleksnim okruženjima može biti izazovno imati takvu detaljnu reprezentaciju. Unatoč tome, tako detaljna reprezentacija okruženja može dati preciznija predviđanja buduće nagrade te tako pomoći agentu da brže nauči rješavati zadani zadatak. Modeli uzorka se razlikuju tako da generiraju samo jedan uzorak iz seta mogućih stanja i vjerojatnosti [21, str. 160-161]. Time se minimiziraju negativne strane distribucijskih modela jer je model onda jednostavniji te zahtijeva manje računalnih resursa. Ova vrsta modela manje je efikasna od distribucijskih modela. Efikasnost se može poboljšati kombiniranjem s različitim strategijama istraživanja okruženja, kao što su Monte Carlo Pretraga Stabla (engl. *Monte Carlo Tree Search*) te Dyna-Q. Modeli uzorka također nude i veću fleksibilnost jer se brže prilagođavaju novim informacijama.

Metode bazirane na modelu također se često nazivaju i planirajuće (engl. *planning*) [21, str. 8, 195] metode. Planiranje u ovom smislu odnosi se na bilo koji proces koji kao ulaz dobiva model, te generira politiku za interakciju s modeliranim okruženjem. Planirajući modeli često se dijele na dva specifična pristupa. Prvi pristup je planiranje u prostoru stanja (engl. *state-space planning*) [21, str. 196]. U ovom pristupu planiranjem se primarno želi navigirati kroz prostor stanja te pronaći put do optimalnog rješenja. Akcijama se prelazi iz jednog stanja u drugo te se vrijednosna funkcija izračunava u svakom stanju. Kao suprotni pristup ovome postoji planiranje u prostoru planova (engl. *plan-space planning*) [21, str. 196]. U planiranju u prostoru planova planiranjem se navigira kroz prostor planova. Operatorima se jedan plan transformira u drugi. Na isti način se i vrijednosna funkcija transformira iz jedne u drugu. Planiranje u prostoru planova kreće s inicijalnim djelomičnim planom koji se sastoji od početnog stanja, završnog stanja te niza akcija kojima se dolazi iz početnog u završno stanje. Metode planiranja u prostoru planova zatim inkrementalno dodaju akcije u niz akcija. Cilj je dobiti potpuni plan koji dolazi do završnog stanja te zadovoljava sve željene uvjete. Neke od popularnih metoda koje se bave planiranjem u prostoru planova su Nonlinear Planning with Abstraction Spaces (NPAS) sustav, Systematic Nonlinear Planning (SNLP) algoritam, i Partial-Order Causal-Link (POCL) algoritam. Ovakve metode izvrsne su u deterministčkim okruženjima gdje mogu inkrementalno dodavati akcije u niz akcija. Stabilnost deterministčkih okružja odlična je podloga za takve metode, u takvim problemima dinamika tranzicija je fiksna i poznata. Zbog kompleksnosti i nepredvidljivosti stohastičkih okruženja ne funkcioniraju najbolje u problemima stohastičke prirode. U takvim okruženjima vjerojatnosti prijelaza iz stanja u stanja, kao i funkcije nagrada su često ili nepoznate ili samo djelomično poznate.

Metode koje se ne baziraju na modelu često se nazivaju metodama koje uče. To ime dobivaju zbog načina na koji se poboljšava politika koju prate. One uče interakcijom sa svojim okružjem te se ne oslanjaju na model svog okružja. Metode se baziraju na povezivanju stanja i odabrane akcije s dobivenom nagradom te se takvim pristupom na dovoljno velikoj količini

podataka dobije veza stanje -> ispravna akcija. Postoji nekoliko ključnih metoda koje spadaju pod "metodama koje uče". To su Temporal Difference metode, Monte Carlo metode, Policy Gradient metode, model-free deep RL metode. Više o njima u kasnijim poglavljima.

2.1.3.3. Bazirano na uzorku/Bootstrapping

Pristup baziran na uzorku (engl. *Sample-based*) i bootstrapping su dva pristupa u predviđanju vrijednosnih funkcija u području poticanog učenja [10, str. 89, 91]. Te dvije vrste metoda mogu se smatrati kao polarne suprotnosti u svom pristupu.

Sample-based metode se za estimiranje vrijednosne funkcije oslanjaju na stvarnim iskustvima interakcije agenta i okružja. One koriste stvarne primjerke epizoda ili putanja agenta. Za estimiranje vrijednosne funkcije koristi se prosjek kumulativnih nagrada tijekom niza epizoda ili putanja. Školski primjer sample-based metode je monte carlo metoda. U poglavlju 5 "Monte Carlo Methods" Sutton i Barto navode kako iako je model potreban, sample-based metodama ne treba potpuno poznavanje okružja. Isto tako nije potrebno apsolutno nikakvo poznavanje dinamike tranzicija u okružju u kojem se agent nalazi. Model je potreban kako bi generirao primjerke epizoda ili putanja. Uz to agentu nije potreban niti set distribucija za svako stanje. Često je u stvarnim primjenama teško i doći do seta distribucija. Bootstrapping je pojam posuđen iz statistike. U statistici bootstrapping označava metodu kreiranja novih uzoraka uz pomoć postojećeg seta podataka. Originalno značenje riječi bootstrapping potiče iz izreke "To pull oneself up by one's bootstraps". Izreka opisuje ideju poboljšanja svoje situacije bez tuđe pomoći, odnosno oslanjanje na samog sebe. To značenje prenosi se u područje poticanog učenja. U bootstrapping pristupu, agent poboljšava vrijednosnu funkciju bez korištenja uzoraka. Za razliku od sample-based pristupa, estimiranje vrijednosne funkcije ne izvršava se nakon svake epizode. Vrijednosna funkcija se estimira nakon svakog koraka ili akcije, te se za estimiranje koristi kombinacija stvarne nagrade dobivene izvršavanjem akcije i agentovo trenutna predikcija nagrade budućeg koraka. Klasični primjer bootstrapping pristupa je Temporal Difference metoda. Za razliku od sample-based pristupa ovaj pristup je efikasniji s obzirom na veličinu uzorka pošto se vrijednosna funkcija reestimira nakon svakog koraka, umjesto nakon svake epizode. Druga prednost ovog pristupa u odnosu na sample-based pristup je to da može raditi i s djelomičnim epizodama kao i s ne-epizodičnim okružjima.

2.1.3.4. Epizodičke/Kontinuirane

Problemi u poticanom učenju dijele se na epizodičke probleme i na kontinuirane probleme koji se još nazivaju i sekvencijalnim problemima [6, str. 43]. U epizodičkim problemima proces učenja moguće je podijeliti na jasno definirane epizode.

Definicija 2 *Epizoda se smatra jasno definiranim nizom koraka koji je ograničen s početnim i završnim stanjem. Agent pokušava maksimizirati kumulativnu nagradu u svakoj epizodi. Nakon svake epizode stanje se resetira na početno te tako agent ima priliku primijeniti naučeno na početnom stanju.*

Kao primjer epizodičkog problema možemo uzeti primjer igre Uno. Igra započinje početnim stanjem, sastoji se od niza koraka/akcija te se igra završava terminalnim stanjem.

Kontinuirani problemi sastoje se od početnog stanja te beskonačnog niza koraka. Agentovo iskustvo se ne dijeli u epizode te ne postoji završni događaj kojim bi se epizoda završila. U ovom scenariju agent pokušava maksimizirati kumulativnu nagradu na beskonačnoj vremenskoj skali. Odnosno agent u okružju odrađuje beskonačan broj koraka te pritom pokušava dobiti što veću nagradu. Kao primjer ove vrste problema možemo uzeti sustav za detekciju anomalija koji prati bankovne transakcije kako bi detektirao novčane prijevare. Agent konstantno prati promet u okružju koje nema završnog događaja niti završnog stanja problema.

Kao rezime, epizodički problemi baziraju se na strukturi u kojoj se agentovo iskustvo dijeli na određene epizode koje imaju početno i završno stanje s nizom akcija koje ih povezuju. Kontinuirani problemi nemaju završno stanje već se sastoje od početnog stanja i beskonačnog niza akcija. Važno je napomenuti da su rijetko problemi striktno epizodički ili kontinuirani. Primjer kontinuiranog problema koji smo naveli može se prilagoditi kako bi bio epizodički. Agent bi mogao pratiti novčani promet na dnevnoj bazi te bi se jedan dan mogao smatrati jednom epizodom. Vrsta problema bitan je čimbenik u odabiru metode kojom će se rješavati problem. Sample-based metode prikladnije su za epizodičke probleme, dok se bootstrapping metode mogu koristiti i za epizodičke kao i za kontinuirane probleme.

2.1.3.5. On-policy/Off-policy

Podjela on-policy/off-policy temelji se na načinu na koji metoda uči i ažurira svoju vrijednosnu funkciju. U on-policy učenju agent uči tako da poboljšava onu politiku koja određuje njegovo ponašanje. Time agent u stvarnom vremenu osjeti posljedice akcija koje odabire u okružju. On-policy metode baziraju se na odabiru akcije prateći trenutnu politiku te učenje iz nagrade koju agent dobije od te akcije. Primjer algoritma koji uči na on-policy način je SARSA (State-Action-Reward-State-Action) algoritam. Kao što vidimo iz naziva algoritma, algoritam se bazira na nizu gdje agent kreće iz trenutnog stanja S, određuje akciju A, te okružje prema toj akciji generira nagradu R. Okružje osim nagrade generira i iduće stanje S' ovisno o vjerojatnostima prijelaza. Agent u tom trenutku zna koju akciju A' će izabrati iduću. SARSA algoritam zatim ažurira politiku koja je odredila akcije A i A', te se nastavlja dalje koristiti izmijenjenom politikom. Kod off-policy učenja, učenje se sastoji od dvije različite politike. Dvije politike koje se koriste u off-policy pristupu su behavior politika i target politika. Behavior politika zaslužena je za generiranje iskustava iz kojih agent uči. Iskustva generira implementacijom nekog od rješenja exploration-exploitation dileme. Jedan od češćih pokušaja rješavanja exploration-exploitation dileme je implementacija ϵ-greedy politike. ϵ-greedy politika bazira se na uvođenju nasumičnosti u određivanje akcije koju će agent poduzeti u određenom stanju. Kao mehanizam nasumičnosti funkcionira ϵ parametar. U ϵ% slučajeva agent akciju bira nasumično, dok se u $1 - \epsilon$% slučajeva akcija bira pomoću target politike. Time se osigurava da agent neće uvijek odabrati "najbolju" akciju, već će često i izabirati akcije koje inače ne bi izabrao. Tako se poboljšava agentovo upoznavanje okružja. Neke od ostalih implementacija mehanizma istraživanja o kojima ćemo govoriti kasnije su: Boltzmann istraživanje, Bayesian istraživanje, ϵ-decreasing,

Target politika zaslužena je za određivanje "ispravne" odluke u određenom stanju. Cilj off-policy učenja je generirati ispravnu target politiku s kojom agent može riješiti zadani problem.

2.2. Markovljev proces odlučivanja

Markovljev proces odlučivanja (engl. *Markov Decision Process*, u daljnjem tekstu MDP) klasična je formalizacija sekvencijalnog odlučivanja. Akcije ne utječu samo na trenutne nagrade, već indirektno i na sve buduće nagrade kroz izmjene budućih situacija i stanja [10, str. 47]. Na taj se način u MDP uključuje i problem odgođenog zadovoljstva (engl. *Delayed gratification*). Pojam odgođenog zadovoljstva dolazi iz područja psihologije. Ne postoji pojedini izvor za kojeg se pouzdano može reći da je glavni izumitelj ovog pojma. Pojam je vremenom evoluirao u jedan od temeljnih pojmova bihevioralne psihologije. Najveću popularnost pojmu dodijelio je eksperiment kojeg je vodio Stanfordski profesor Walter Mischel. Eksperiment naziva "Stanford Marshmallow Experiment" [22] uključivao je praćenje ponašanja djece u sobi s malom nagradom (kolačićem) te njihove odlučnosti odricanja kolačića kako bi naknadno dobili veću nagradu (2 kolačića). Cilj eksperimenta bio je proučiti u kojoj dobi se razvije mehanizam odgađanja nagrade kod djece predškolske dobi. U kontekstu MDP-a, odgađanje zadovoljstva je jedan od bitnijih koncepata. Cilj MDP-a je maksimizirati kumulativ nagrada tijekom rješavanja nekog problema. Maksimiziranje kumulativa nagrada često ne znači birati maksimalnu nagradu u svakom stanju. Zbog dinamike tranzicija vrijedi sljedeće:

$$
\begin{aligned}
&q_*(S_1, a_1) + \max_{a_2} q_*(S_2, a_2) = R_1 + R_2 \\
&q_*(S_1, a_3) + \max_{a_4} q_*(S_3, a_4) = R_3 + R_4 \\
&R_1 > R_3, R_4 > R_2 \implies R_1 + R_2 < R_3 + R_4
\end{aligned}
$$

Dinamika tranzicija omogućava situaciju u kojoj maksimiziranje lokalnih nagrada u pojedinom stanju nužno ne mora značiti maksimiziranje kumulativa nagrada. Uzmemo li prošli primjer, agent u stanju S_1 može birati akcije a_1 i a_3. Nagrada R_1 koju dobijemo odabirom akcije a_1 veća je od nagrade R_3 dobivene biranjem akcije a_3. No, maksimiziranje nagrada u trenutnom stanju ne uzima u obzir nagrade stanja u kojima agent završi nakon odabira trenutno maksimalne nagrade. Biranjem akcije a_1 agent završava u stanju S_2 gdje se biranjem akcije s maksimalnom nagradom dolazi do nagrade R_2. Biranjem akcije a_3 okružje prelazi u stanje S_3 gdje se biranjem akcije vezane za najveću nagradu dobiva nagrada R_4. Zbrojimo li sve nagrade dobivene u svakoj od tranzicija ispravnije se prikazuje stvarna slika. Iako je u početnom stanju S_1 nagrada za prvi slučaj bila veća ($R_1 > R_3$). U idućem stanju nagrada može biti veća u drugom scenariju ($R_4 > R_2$), što, kada izračunamo kumulativ nagrada, dovodi do zaključka da je drugi scenarij bolji, iako se nije birala akcija koja donosi maksimalnu nagradu u trenutnom stanju ($R_1 + R_2 < R_3 + R_4$). Iako se biranje maksimalne nagrade u svakom stanju čini kao pouzdana taktika, ipak je bitno implementirati mehanizam istraživanja, kao i mehanizam odgođenog zadovoljstva kako bi se postigao najbolji kumulativ nagrada.

2.2.1. Komponente Markovljevog Procesa Odlučivanja

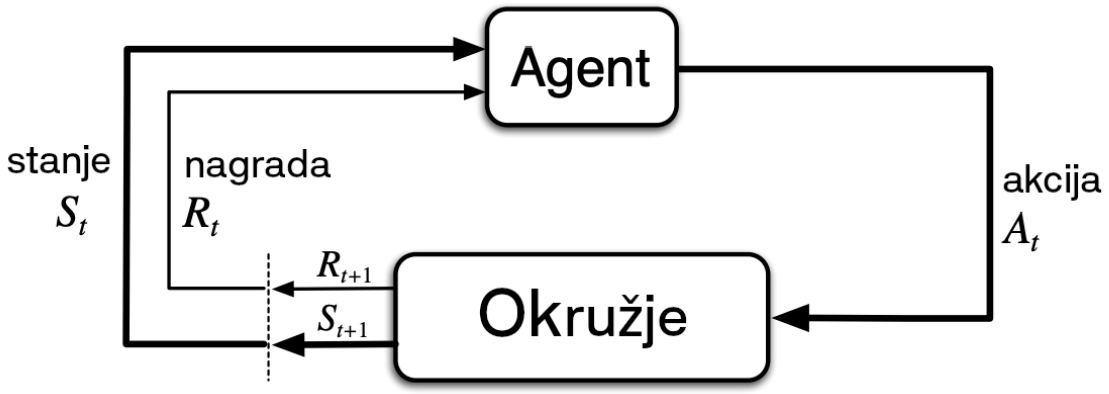
MDP je zamišljen kao jednostavan okvir za problem učenja kroz interakciju. Označava se kao okvir zbog načina na koji se problem transformira nakon što se pretvori u MDP. Transformiranje problema dešava se kroz opisivanje komponentama koje definira MDP. Te komponente su *agent*, *okružje*, *stanja*, *akcije*, *tranzicije* i njihove vjerojatnosti te *nagrade* dobivene pojedinom akcijom [6, str. 646].

Agent označava autonomni entitet koji samostalno donosi odluke te uči riješiti problem okružja. Okružje podrazumijeva sve ono s čime agent može imati interakciju. Agent i okružje u kontinuiranoj su interakciji. Agent odlučuje o akcijama u određenom stanju, te okružje reagira na te akcije i predstavlja nova stanja agentu. Osim što predstavlja nova stanja agentu, također mu daje i nagrade za učinjene akcije.

Stanje je reprezentacija situacije u kojoj se agent nalazi u okružju u određenom trenutku. Stanjem se agentu pobliže opisuje trenutna situacija te mu se olakšava proces donošenja odluka. Stanje može poprimiti razne oblike. Neki od čestih oblika stanja su niz numeričkih vrijednosti, na primjer X, Y i Z koordinate u prostoru, trenutna brzina, količina zdravlja, itd. Osim numeričkih vrijednosti mogu se pojavljivati i alfanumerički nizovi ili diskretne reprezentacije simbolima. Alfanumeričkim nizom može se tekstualno opisati agentova lokacija, npr "Dnevna soba", "Kuhinja", itd. Diskretnim simbolima mogu se opisivati objekti u igri te njihova pozicija u takozvanoj mreži (engl. *grid*) kojom se označava okružje. Također postoji i slika kao oblik reprezentacije stanja. Slika se može koristiti kao prikaz stanja objekata ispred agenta koji djeluje u 3D prostoru. Reprezentacija stanja ovisi o vrsti problema koji se rješava kao i o potrebama koje želimo zadovoljiti pri rješavanju problema. Set svih mogućih stanja u nekom okružju naziva se prostorom stanja.

Akcija je glavni mehanizam kojim se agent koristi kako bi izmijenio situaciju u kojoj se nalazi i transformirao okružje. Biranjem akcije agent okružju šalje signal koje okružju označava da je potrebno odraditi tranziciju iz stanja S pomoću akcije A te na taj način pomoću tranzicijskih vjerojatnosti iz stanja S u druga stanja izračunati iduće stanje.

Interakciju između agenta i okružja možemo vidjeti na dijagramu 2.



Slika 2: Prikaz interakcije između agenta i okružja; preuzeto iz [10]

Dijagram prikazuje interakciju između agenta i okružja u kojem djeluje. Interakcija se odvija u diskretnim vremenskim koracima označenim $t = 0, 1, 2, 3, ... \in \mathbb{N}$. Interakcija započinje stanjem $S_t \in \mathcal{S}$ u vremenskom trenutku t. Agent u ovom stanju bira akciju $A_t \in \mathcal{A}(s)$ kojom se kontrola daje okružju. Okružje generira nagradu $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$ te agenta prebacuje u stanje S_{t+1}. Niz se tako nastavlja s akcijom A_{t+1}, što je okružju signal da treba izračunati iduće stanje S_{t+2}, itd. Svaku interakcija između agenta i okružja moguće je opisati nizom stanja, akcija i nagrada. Takav niz naziva se tranzicijom. Tranzicija u ovom slučaju izgleda ovako:

$$S_0, A_0, R_1, S_1, A_1, R_2...$$

Konačni MDP bavi se problemima koji imaju *konačni* broj stanja ($\mathcal{S}$), akcija ($\mathcal{A}$) i nagrada ($\mathcal{R}$). Osim toga ovakvi problemi imaju i diskretne distribucije vjerojatnosti kojima se definira da nagrada R_t+1 i stanje S_t+1 ovise samo o prethodnom stanju S_t i izabranoj akciji A_t. Distribucija vjerojatnosti označava da se za svaku vrijednost $s' \in \mathcal{S}$ i $r \in \mathcal{R}$ može odrediti vjerojatnost pojavljivanja u određenom trenutku t [6, str. 486-487]. Sutton i Barto [10] navedeno zapisuju na sljedeći način [10, str. 48]:

$$p(s', r|s, a) \doteq Pr\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\}, \forall s', s \in S, r \in R, a \in A(s)$$

Prethodno prikazanom funkcijom p definira se dinamika MDP-a. U navedenom izrazu Pr označava vjerojatnost (engl. *Probability*) događanja matematičkog izraza u vitičastim zagradama. Matematički izraz koristi se operatorom uvjetne vjerojatnosti. Uvjetna vjerojatnost u primjeru $P(A|B)$ označava vjerojatnost događaja A ako je ispunjen uvjet B [23, str. 46]. To znači da matematički izraz $Pr(S_t = s', R_t = r|S_{t-1} = s, A_{t-1)} = a$ predstavlja vjerojatnost dolaska u stanje s' s nagradom r pod pretpostavkom da je prethodno stanje bilo s te je odabrana akcija a. Sutton i Barto [10] funkciju vjerojatnosti definiraju na sljedeći način $p : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$. Funkcija p kao ulaz prihvaća četiri argumenta. Argumenti koje prihvaća su stanje $s \in \mathcal{S}$, nagradu $r \in \mathcal{R}$, iduće stanje $s' \in \mathcal{S}$ te akciju $a \in \mathcal{A}$. Ulazni argumenti se mapiraju na vjerojatnost u rasponu od 0 do 1. Ovakav redoslijed ulaznih argumenata može izgledati zbunjujuće pošto argumenti nisu u ispravnom kronološkom redoslijedu. U stvarnosti vrijedi redoslijed izvršavanja naveden u primjeru tranzicije 2.2.1 (S, A, R, S). Nagrada R i iduće stanje S dolaze u isto vrijeme tako da se prethodno napisani primjer može napisati i na sljedeći način S, A, S, R. Sutton i Barto koristili su takav zapis zbog načina zapisivanja u definiciji funkcije $p(s', r|s, a)$. U navedenom zapisu koristi se operator uvjetne (kondicionalne) vjerojatnosti $a|b$ kojom se označava da a slijedi iz b [23, str. 46]. Ako okrenemo redoslijed pisanja operatora, vidimo da se i Sutton i Barto koriste navedenom notacijom S, A, S, R. Kondicionalni operator navodi sljedeće [10, str. 49]:

$$\sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r|s, a) = 1, \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$$

Ovaj matematički izraz objašnjava da suma vjerojatnosti idućih stanja (s') i nagrada (r) za dobiveno stanje s i akciju a mora iznositi 1. Osim toga, MDP uvodi pojam Markovljevog

svojstva. Markovljevo svojstvo postavlja temeljna ograničenja na definiciju stanja. Prethodni matematički izraz iznimno je bitan jer predstavlja ideju da se vjerojatnost budućeg stanja može izračunati uz pomoć trenutnog stanja i akcije te da nije potrebno nikakvo znanje o prethodnim stanjima ili akcijama. Stoga je definicija stanja ograničena tako da mora sadržavati sve informacije potrebne za određivanje idućeg stanja. Ako stanje zadovoljava taj uvjet tada se može reći da stanje, ali i cijeli problem, zadovoljava Markovljevo svojstvo [10, str. 49]. Postoje mnogi algoritmi koji se oslanjaju na postojanje Markovljevog svojstva, kao npr Monte Carlo metode te samim time i ekstenzije tog algoritma kao što su Q-Learning, i SARSA. Iako se ti algoritmi oslanjaju na postojanje Markovljevog svojstva, mogu se prilagoditi za probleme koji ne podliježu ograničenjima Markovljevog svojstva. U tom slučaju stanje ne sadrži sve relevantne informacije potrebne za odlučivanje idućeg stanja te se iz tog razloga implementiraju mehanizmi pamćenja koji sve relevantne informacije drže pohranjeno.

Sutton i Barto [10] navode kako je iz funkcije p s četiri ulazna argumenta moguće izvesti, odnosno izračunati, bilo kakvu informaciju o okolišu koja nekoga može zanimati. Na primjer, iz navedene funkcije može se izvesti funkcija s tri argumenta kojom se određuje vjerojatnost tranzicija između dva stanja i akcije. Notacija funkcije izgleda ovako: $p : \mathcal{S} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$, a potpuna funkcija glasi [10, str. 49]:

$$p(s'|s, a) \doteq Pr\{S_t = s'|S_{t-1}, A_{t-1} = a\} = \sum_{r \in \mathcal{R}} p(s', r|s, a)$$

Moguće je izvesti i formulu za izračun nagrade za par stanja i akcije $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ koja glasi[10, str. 49]:

$$r(s, a) \doteq \mathbb{E}[R_t|S_{t-1} = s, A_{t-1} = a] = \sum_{r \in \mathcal{R}} r \sum_{s' \in \mathcal{S}} p(s', r|s, a)$$

Također je moguće izvesti i nagradu za S, A, S' tranziciju. Ta nagrada znatno olakšava izračun u kojem se u stanju S bira niža nagradu, a u stanju S' se pohlepno bira najveća nagrada. Takav izraz zapisuje se na sljedeći način $r : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$. Vrijednost je moguće izračunati matematičkim izrazom [10, str. 49]:

$$r(s, a, s') \doteq \mathbb{E}[R_t|S_{t-1} = s, A_{t-1} = a, S_t = s'] = \sum_{r \in \mathcal{R}} r \frac{p(s', r|s, a)}{ps'|s, a}$$

2.2.2. Ciljevi, nagrade i povratne vrijednosti

Cilj agenta usko je povezan s nagradama i povratnim vrijednostima. Kao što je dosad spomenuto, cilj agenta je maksimizirati nagradu koju dobije [6, str. 37]. Ideja nagrade kao stimulirajućeg mehanizma je ključna ideja u poticanom učenju. No, bitno je razlikovati nagrade od povratnih vrijednosti. Cilj agenta je maksimizirati nagradu, a nagrada je kumulativ svih povratnih vrijednosti. Agent bi mogao pokušavati maksimizirati povratne vrijednosti, no to ne mora nužno značiti maksimiziranje kumulativne nagrade.

Sutton i Barto [10] upozoravaju o opasnostima manipuliranja signalom nagrade te navode sljedeće:

> "Posebice, signal nagrade nije mjesto za prenošenje prethodnog znanja agentu o tome kako postići ono što mi želimo da agent čini. Na primjer, agent koji igra šah bi trebao biti nagrađen samo za samu pobjedu, a ne za ostvarivanje podciljeva kao što su uzimanje protivnikovih figurica ili stjecanje kontrole nad središtem ploče. Ako bi se ostvarivanje takvih podciljeva nagrađivalo, tada bi agent mogao pronaći način da postigne podciljeve bez postizanja stvarnog cilja. Na primjer, agent bi mogao pronaći način kako da uzme protivnikove figure čak i po cijenu gubitka igre." [10, str. 54]

Prema njima, signalom nagrade ne treba se agentu reći *kako* nešto postići, već mu dati cilj, odnosno *što* postići i pustiti agenta da sam pronađe način. No, mislim da to nije u potpunosti ispravan pristup. Ovakav pristup u direktnoj je kontradikciji s konceptom oblikovanja nagrada. Konceptom oblikovanja nagrada moguće je poboljšati efikasnost učenja agenta. Oblikovanjem nagrada postojeća nagradu koju agent dobije od okružja oblikuje se tako da joj se dodaje vrijednost funkcije za oblikovanje nagrada. Nagrada se oblikuje tako što se u nju dodaju dodatne nagrade (ili kazne) za ostvarivanje podciljeva [24]. Stav koji iznose Sutton i Barto u direktnoj je koliziji s konceptom oblikovanja nagrada. No, prilikom oblikovanja nagrada veoma je bitno pažljivo odrediti funkciju oblikovanja, kako bi se spriječilo upravo to o čemu Sutton i Barto upozoravaju. Oblikovanje nagrada bi agenta trebalo samo potaknuti u pravom smjeru, ne u potpunosti preuzeti kontrolu nad agentom. Samo tako će agent moći naučiti riješiti cilj, bez da se pretjerano fokusira na podciljeve i zanemari glavni cilj.

Agent pokušava estimirati kumulativ nagrada u epizodi (Definicija 2). Kumulativ nagrada naziva se još i povratnom vrijednosti [10, str. 54]. Dakle, agent pokušava maksimizirati povratnu vrijednost koju označavamo s G_t. G_t označava zbroj nagrada od $t = 0$ do $t = T$, gdje T označava završni trenutak.

$$G_t \doteq R_0 + R_1 + R_2 + ... + R_T$$

Ovaj zapis izrazito je pogodan za epizodičke MDP-e. Kod epizodičkih MDP-a izvedba problema može se podijeliti na epizode, to jest postoje jasno određeni početni trenutak te završni trenutak. Set svih stanja bez završnog stanja u ovakvim problemima prikazuje se s $\mathcal{S}$, dok se set svih stanja uključujući završno stanje prikazuje s $\mathcal{S}^+$ Kod kontinuiranih MDP-a se ovaj izraz ne može primijeniti. Kod takvih oblika MDP-a ne postoji završni trenutak, odnosno moguće je definirati samo $\mathcal{S}$. U tom slučaju prethodno navedeni izraz izgleda ovako [10, str. 54]:

$$G_t \doteq R_{t+1} + R_{t+2} + R_{t+3} + ... + R_\infty$$

Samim time, s obzirom na to da se zbraja beskonačan broj nagrada i povratna vrijednost G_T bi bila beskonačno velika. Beskonačno velika povratna vrijednost narušava glavni

princip ovog pristupa, a to je maksimiziranje kumulativa nagrada. Kako bi prilagodili jednadžbu da funkcionira za kontinuirane zadatke, potrebno je implementirati mehanizam diskontnih nagrada. Diskontirane nagrade označavaju pojam je koji je usko povezan s ljudskim ponašanjem i načinom na koji ljudi ali i životinje djeluju [6, str. 649]. Mnogi psiholozi i bihevioralni znanstvenici su istraživali ljudsko odlučivanje te koncept odgođenih nagrada. Diskontirane nagrade spadaju u istu kategoriju. Diskontnim nagradama MDP-ima se dodaju dvije značajne karakteristike [6, str. 650]:

- Ograničavanje povratne vrijednosti u kontinuiranim problemima tako da se izbjegne beskonačno velika nagrada
- Uvodi se mogućnost prioritiziranja kratkoročnih nagrada. Takva vrsta djelovanja usko je povezana s ljudskom psihom

Diskontirane nagrade implementiraju se uvođenjem diskontnog faktora (γ) u jednadžbu povratne vrijednosti [6, str. 649]. Vrijednosti budućih nagrada množe se diskontnim faktorom te se tako smanjuje vrijednost nagrade X u budućnosti. Ta ista nagrada bi u kratkoročnoj budućnosti imala veću vrijednost. Jednadžba za G_t s diskontnim faktorom izgleda [10, str. 55]:

$$G_T \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + ... + \gamma^k R_{t+k+1} = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

U ovom prikazu γ je diskontni faktor u rasponu $[0, 1]$. Diskontnim faktorom označava se vrijednost nagrade u budućnosti u odnosu na istu nagradu u trenutku. Što je diskontni faktor manji, agent postaje sve više kratkovidan (engl. *myopic*). Kada se diskontni faktor približava vrijednosti 1, tada agent postaje manje kratkovidan. Ako je $\gamma = 1$ tada agent podjednako vrednuje nagradu i sada i u budućnosti. Sutton i Barto [10, str. 55] navode kako agent s većim diskontnim faktorom postaje dalekovidan. National Eye Institute [25] kao definiciju dalekovidnosti navodi "Dalekovidnost je refrakcijski poremećaj zbog kojeg bliski objekti izgledaju mutno". U kontekstu vrednovanja nagrada bi dalekovidni agent višu vrijednost dodjeljivao nagradama u budućnosti u odnosu na trenutne nagrade. Pošto je maksimalna vrijednost diskontnog faktora 1 tada agent ne može nikada postati istinski dalekovidan. U najgorem slučaju ($\gamma = 1$) podjednaku vrijednost imaju buduće nagrade kao i trenutne. Kratkovidnost agenta označava karakteristiku u kojoj agent prioritizira trenutnu, kratkoročnu, nagradu te se ne fokusira na buduću nagradu. Iz toga je vidljivo da diskontni faktor ima značajan utjecaj na ranije spomenutu dilemu istraživanja i iskorištavanja. Postoji značajna korelacija između diskontnog faktora i mehanizma iskorištavanja i istraživanja. Što je γ-faktor manji, to se agent više fokusira na aspekt iskorištavanja, dok se kod većeg γ-faktora agent više fokusira na istraživanje.

2.2.3. Vrijednosne funkcije

Vrijednosne funkcije (engl. *value functions*) [10, str. 58] su ključne u procesu učenja agenta. Pomoću vrijednosnih funkcija moguće je odrediti koliko je neko stanje "dobro" ili koliko

je neka akcija u pojedinom stanju "dobra". Vrijednosne funkcije određuju vrijednost stanja i akcije te se tako procjenjuje koliko je stanje ili akcija "dobra". Vrijednost stanja podrazumijeva kumulativ nagrada, odnosno povratnu vrijednost, koju će agent dobiti nakon izvršavanja sekvence akcija praćenjem politike kojom se trenutno vodi. Politika agenta definira se simbolom π, a ona je zapravo mehanizam koji vodi proces odlučivanja. Politikom se određuje koju akciju a izabrati u stanju s. Vjerojatnost biranja akcije a u stanju s vodeći se politikom π zapisuje se [10, str. 58]:

$$\pi(a|s)$$

Navedeni zapis koristi se već spomenutim operatorom uvjetne vjerojatnosti [23, str. 46]. U ovom slučaju operator označava vjerojatnost događanja akcije a u slučaju gdje je prethodno stanje bilo s. Određivanje akcije a prateći politiku π u stanju s zapisuje na sljedeći način:

$$\pi(s)$$

Politike agenta mogu biti stohastičke i deterministčke [6, str. 848-849][10, str. 79], što je diktirano samim okružjem [6, str. 43]. Kod stohastičkih politika u određenom stanju postoji vjerojatnost biranja bilo koje od mogućih akcija. Stohastičke politike kao vjerojatnosti odabira akcija u određenom stanju generiraju distribuciju u obliku liste s vjerojatnostima za svaku akciju:

$$\pi(s) = [p(a_1), p(a_2), ..., p(a_n)$$

Funkcijom p definira se vjerojatnost odabira pojedine akcije. Primjer ovakve distribucije bio bi:

$$\pi(s) = [0.3, 0.2, 0.5]$$

U ovom slučaju vidljivo je da je vjerojatnost odabira prve akcije u stanju s jednako 30%. Za drugu i treću akciju vjerojatnosti odabira su 20 i 50%. Zbroj svih distribucija mora biti jednak 1. Deterministčke politike uvijek u pojedinom stanju biraju jednu akciju. Za razliku od stohastičkih politika, deterministčke politike ne dozvoljavaju nasumičnost u odabiru akcija. U deterministčkim politikama agent uvijek bira istu akciju za određeno stanje. Takav primjer izgleda ovako: $\pi(s) = a_k$, gdje a_k predstavlja pojedinu akciju agenta.

Vrijednosna funkcija stanja određuje vrijednost pojedinog stanja u okruženju u kojem se agent nalazi [10, str. 58]. Tako se mogu definirati stanja prema kojima agent želi težiti, te stanja koja želi izbjeći. Očiti primjer dobrog stanja je primjer u igri šaha u kojoj je agent dva poteza do pobjede. Dok je primjer lošeg stanja ono stanje u kojem agent gubi igru. Vrijednost određenog stanja definira se kao povratna vrijednost od stanja s pa sve do završnog stanja s_T uz praćenje politike π. Tu vrijednost iskazujemo sljedećom matematičkom jednadžbom [10, str. 58]:

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t|S_t = s_t] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s\right], \forall s \in \mathcal{S}$$

U ovom slučaju $\mathbb{E}$ definira određivanje *očekivane* vrijednosti. Odnosno očekivanog kumulativa nagrada od stanja t do završnog stanja. Kumulativ nagrada prikazuje se simbolom G_t

za kojeg se kao preduvjet uzima stanje s_t. Navedeni kumulativ nagrade moguće je matematički raspisati kao sumu svih (diskontiranih γ^k) nagrada R_{t+k+1}.

Na isti način može se definirati i vrijednost para stanja i akcije (engl. *state-action pair*). Vrijednost para stanja i akcije definira se kao povratna vrijednost koju agent dobije biranjem akcije a u stanju s te vođenjem politikom π u daljnjim stanjima [10, str. 58]. Navedena funkcija za vrijednost para stanja i akcije, $q_\pi(s, a)$, matematički se zapisuje izrazom [10, str. 58]:

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

Prilikom proučavanja MDP-a, jedan od ključnih ciljeva je određivanje optimalne politike koja će maksimizirati očekivani kumulativ nagrade. Vrijednosne funkcije su u tom slučaju izrazito korisne pošto njima možemo odrediti vrijednost stanja i akcije. Za karakterizaciju optimalne politike moguće je koristiti Bellmanove jednadžbe. One također izražavaju vrijednost određenog stanja ili para stanja i akcije, no na malo drukčiji način. To su matematičke jednadžbe kojima se opisuje rekurzivna veza između stanja/stanja i akcije i susjednih stanja/stanja i akcija. Bellmanove jednadžbe povezuju trenutne vrijednosti s budućima te nam omogućuju da ažuriramo vrijednosne funkcije i iterativno dođemo do optimalnih vrijednosnih funkcija. Prethodno prikazanu vrijednosnu funkciju stanja možemo pretvoriti u Bellmanovu jednadžbu na sljedeći način :

$$\begin{aligned}
v_\pi(s) \doteq \mathbb{E}_\pi[G_t|S_t = s_t] &= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s\right], \forall s \in \mathcal{S} \\
&= \mathbb{E}_\pi\left[R_{t+1} + \gamma R_{t+2} + ... + \gamma^{T-t+1} R_T \middle| S_t = s_t\right] \\
&= \mathbb{E}_\pi\left[R_{t+1} + \gamma G_{t+1} | S_t = s_t\right] \\
&= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)[r + \gamma \mathbb{E}_\pi[G_{t+1}|S_{t+1} = s']] \\
&= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')], \forall s \in \mathcal{S}
\end{aligned}$$

Najbitnija značajka Bellmanovih jednadžbi je rekurzivnost koja je ugrađena unutar jednadžbe. Na posljednjoj liniji vidljiva je rekurzija gdje je prilikom izračuna $v_\pi(s)$ potrebno izračunati i $v_\pi(s')$. Kako bi izračunali $v_\pi(s')$ potrebna nam je vrijednost idućeg stanja te se taj niz nastavlja. Bellmanova jednadžba vrijednost stanja s izračunava tako da se za svaku akciju a u stanju s izračuna suma nagrade koja slijedi od $\pi(a|s)$ i diskontirane nagrade idućeg stanja s'.

Osim Bellmanova jednadžbe za vrijednost stanja, postoji i Bellmanova jednadžba za izračun vrijednosti para stanja i akcije. Tu jednadžbu označavamo sa $q(s, a)$. Jednadžba se izvodi od vrijednosne funkcije para stanja i akcije:

$$
\begin{aligned}
q_\pi(s,a) &\doteq \mathbb{E}_\pi[G_t|S_t=s,A_t=a] \\
&= \mathbb{E}_\pi\left[R_{t+1}+\gamma R_{t+2}+...+\gamma^{T-t+1}R_T\middle|S_t=s_t,A_t=a\right] \\
&= \mathbb{E}_\pi\left[R_{t+1}+\gamma G_{t+1}|S_t=s_t,A_t=a_t\right] \\
&= \sum_{s',r} p(s',r|s,a)[r+\gamma\sum_{a'}\pi(a'|s')q_\pi(s',a')]
\end{aligned}
$$

Bellmanova jednadžba za vrijednost para stanja i akcije koristi se kao model pomoću kojeg je moguće matematički prikazati vrijednost para stanja i akcije. U ovom slučaju izračunava se suma svih budućih stanja s', kao i pripadajućih nagrada r. Za svako buduće stanje izračunava se suma nagrade i diskontirane vrijednosti svakog budućeg stanja koje slijedi odabirom akcije a u stanju s'. Vrijednost budućeg stanja računa se tako da se odredi tranzicijska vjerojatnost u stanje s' pomoću akcije a' te se ta vjerojatnost množi s q vrijednosti stanja s' i akcije a'. Kao i u Bellmanovoj jednadžbi za vrijednost stanja, i u ovoj jednadžbi moguće je vidjeti rekurziju koja je presudna u Bellmanovim jednadžbama.

2.3. Dinamičko programiranje

Dinamičko programiranje je pojam za čije je stvaranje zaslužan Richard E. Bellman [26]. Dinamičko programiranje skup je metoda koji pronalazi rješenje problema razbijanjem problema na manje probleme. Sutton i Barto [10, str. 73] navode kako je dinamičko programiranje kolekcija algoritama koji pronalaze optimalnu politiku koristeći se savršenim modelom okružja u vidu MDP-a.

Dinamičko programiranje često se primjenjuje na probleme s rekurzijom. Česti primjer problema na koji je moguće primijeniti je problem Fibonaccijevog niza. U ovom školskom primjeru, implementacija s običnom rekurzijom nevjerojatno je spora i neefikasna. Sveučilište u Texasu [27, str. 11] navodi kako se za potrebe izračuna 40. Fibonaccijeva broja, četvrti broj u Fibonaccijevom nizu izračuna čak 24,157,817 puta. U slučaju rudimentarne rekurzije, implementacija algoritma u Python programskom jeziku izgleda:

Listing 2.1: [Primjer algoritma za izračun Fibonaccijevog niza korištenjem rudimentarne rekurzije] Ovo je primjer koda koji je preuzet iz [27, str. 5] te preveden u Python programski jezik

```
def fib(n):
    if(n <= 2):
        return 1
    else:
        return fib(n-1)+fib(n-2)
```

Rekurzija funkcionira tako da funkcija fib dobije ulazni argument n kojim se označava redni broj traženog broja iz Fibonaccijevog niza. Ako je redni broj manji ili jednak broju 2, tada se kao rezultat vraća broj 1. U protivnom se rekurzivno poziva ista funkcija nad prethodnom

broju (n-1) te broju prije prethodnog (n-2). Kao što je moguće zaključiti promatranjem samih uvjeta ponovnog poziva (ako je redni broj veći od 2, funkcija se poziva još dva puta), ovakav tip rekurzije izrazito je neefikasan. Na primjer, za potrebe računanja 43. Fibonaccijevog broja, redoslijed izvršavanja bio bi sljedeći. Pošto je $n = 43$ veće od 2, tada se ulazi u "else" dio koda. Rekurzivno se poziva funkcija fib za 42. i 41. broj. Pri izračunu 42. Fibonaccijevog broja ponovno se izvršava "else" dio koda gdje se funkcija fib poziva s parametrima 41 i 40. Već na drugoj razini rekurzije vidljiva je neefikasnost pri izvršavanju. Konstantno se izvršavaju ponovljeni izračuni već izračunatih brojeva. Rješenje ovog problema je krenuti u suprotnom smjeru. Umjesto naivne rekurzije, moguće je krenuti od manjih brojeva prema traženom [27, str. 11]. Tako je moguće poboljšati efikasnost programa tako da nema ponovnog izračuna brojeva.

Osim uklanjanja rekurzije moguća su i druga brojna unaprjeđenja. Jedan način unaprjeđenja algoritma je uvođenje mehanizma memoizacije. Memoizacija predstavlja spremanje međuvrijednosti kako bi se poboljšala efikasnost algoritma [27, str. 15][6, str. 780]. Prilikom daljnjih izračuna moguće je pogledati postoji li već vrijednost rezultat izračuna u lokalnom spremniku.

Za dinamičko programiranje problem mora zadovoljavati dva uvjeta [27, str. 19]: Optimalna podstruktura - moguće je riješiti potprobleme odvojeno i kombinirati rješenja kako bi dobili rješenje cijelog problema Mora sadržavati preklapajuće potprobleme - rješenja za potprobleme ovise jedan o drugome.

Uvjet optimalne podstrukture označava da se problem može podijeliti na manje cjeline istog problema. Pronalaskom optimalnog rješenja na svaki potproblem i kombiniranjem svih rješenja dobiva se optimalno rješenje problema. Optimalna podstruktura vidljiva je i u primjeru s Fibonaccijevim nizom. Rješenje za 42. član Fibonaccijevog niza jednak je kombiniranom rješenju 40. i 41. člana niza. Drugi uvjet za implementaciju dinamičkog programiranja je svojstvo problema u kojemu se potproblemi preklapaju. U takvim problemima moguće je koristiti metode memoizacije (engl. *memoization*) i tabulacije (engl. *tabulation*) za pohranjivanje međuvrijednosti [28]. Tako se međurješenja jednog problema mogu koristiti za ubrzavanje rješavanja drugih problema. To svojstvo omogućava da pristup dinamičkog programiranja bude značajno efikasniji od naivne rekurzije. I ovo svojstvo vidljivo je u primjeru Fibonaccijevog niza. Prilikom izračuna n-tog člana niza izračunavaju se svi prethodni članovi. Kao što je već prethodno spomenuto, time se dolazi do problema višestrukog izračuna u kojem se članovi niza izračunavaju više puta. Taj problem može se izbjeći implementacijom mehanizma pamćenja međuizračuna te je izračun finalnog rješenja znatno efikasniji.

Pri pronalasku optimalnog rješenja na probleme MDPa uz pomoć dinamičkog programiranja koriste se 5 ključnih algoritama [6]. Ovi algoritmi su međusobno povezani, te se često koriste u kombinaciji jedan s drugim. Ključni algoritmi nazivaju se evaluacija politike (engl. *Policy Evaluation*) što se u literaturi također naziva i korakom predviđanja (engl. *Prediction*) [10, str. 74], poboljšanje politike (engl. *Policy Improvement*), iteracija politike (engl. *Policy Iteration*), iteracija vrijednosti (engl. *Value Iteration*), te generalizirana iteracija politike (engl. *Generalized Policy Improvement*).

2.3.1. Evaluacija politike (Predviđanje)

Evaluacija politike označava izračun vrijednosne funkcije stanja v_π. Taj problem se u literaturi još naziva i problemom predviđanja. Problem se svodi na implementaciju prethodno objašnjenih Bellmanovih jednadžbi. Evaluacija politike koristi se Bellmanovom jednadžbom za izračun vrijednosti stanja (2.2.3) preformulirano kao pravilo ažuriranja [10, str. 74]:

$$\begin{aligned} v_{k+1}(s) &\doteq \mathbb{E}_\pi[R_{t+1} + \gamma v_k(S_t + 1)|S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_k(s')], \forall s \in \mathcal{S} \end{aligned}$$

Navedeni algoritam naziva se iterativnom evaluacijom politike (engl. *Iterative policy evaluation*) [10, str. 74]. Za svaku aproksimaciju v_{k+1} pomoću v_k potrebno je zamijeniti staru vrijednost stanja s novom vrijednošću dobivenom zbrajanjem stare vrijednosti stanja s' i sljedećih stanja i predviđenom nagradom. Bitno je naglasiti da svaka iteracija iterativne evaluacije politike ažurira vrijednosti svakog stanja jednom kako bi se izračunala nova aproksimativna vrijednosna funkcija [10]. Sutton i Barto [10] navode kako postoje dva moguća pristupa pri implementaciji sekvencijalnog programa za iterativnu evaluaciju politike.

U prvom pristupu algoritam se implementira uz pomoć dva niza. Jedan niz koristi se za pohranjivanje starih vrijednosti, dok se u drugi niz pohranjuju nove vrijednosti izračunate uz pomoć starih. Tako se tijekom samog algoritma stare vrijednosti ne mijenjaju.

Drugi pristup uključuje korištenje jednog niza i ažuriranje vrijednosti u mjestu. Odnosno, prilikom izračuna nove vrijednosti, stara vrijednost se "pregazi". Također navode kako oba pristupa konvergiraju prema v_π. To jest, oba pristupa rezultiraju optimalnim rješenjem. Bitna razlika je da drugi pristup konvergira brže, odnosno brže se dolazi do optimalnog izračuna vrijednosti. To je zato što se tijekom samog izračuna, u mjestu starih vrijednosti koriste novoizračunate vrijednosti čim se pojave u nizu. Time se znatno algoritam ubrzava jer u prvom pristupu nove vrijednosti ulaze u izračun tek prilikom sljedeće iteracije izračuna. U literaturi se često podrazumijeva da se prilikom spominjanja algoritama Dinamičkog Programiranja koristi algoritam koji vrijednosti ažurira u mjestu.

2.3.2. Poboljšanje politike

Poboljšanje politike iznimno je važan algoritam koji omogućava pronalaženje boljih politika. No, uspoređivanje različitih politika može biti kompleksan problem. Poboljšanje politike bazira se na principu izračuna vrijednosti biranja akcije a u stanju s te daljnjim praćenjem politike π. Time se generira politika π' koja je identična politici π, s razlikom da se u stanju s sada bira akcija a. Teorem poboljšanja politike (engl. *Policy improvement theorem*) navodi kako je politika π' bolja od politike π Ako je povratna vrijednost veća, matematički zapis glasi [10, str. 78]:

$$q_\pi(s, \pi'(s)) \geq v_\pi(s) \implies v'_\pi(s) \geq v_\pi$$

Teorem nam govori da, Ako je vrijednost odabira akcije a u stanju s prema politici π' te daljnjim praćenjem politike π veća ili jednaka od vrijednosti dobivene samim praćenjem politike π. Tada slijedi da je i sama politika π' bolja ili jednako dobra kao i politika π.

Teorem poboljšanja politike nam omogućava korištenje "pohlepnog pristupa" (engl. *greedy approach*) za izračun optimalne politike. Prethodno je navedeno kako je pohlepni pristup često miopičan te rezultira suboptimalnim rješenjem, odnosno suboptimalnom politikom. s obzirom na to da pohlepni pristup ne uzima u obzir buduće nagrade, rješenje ne može biti optimalno. Inkorporiranjem teorema poboljšanja politike moguće je pohlepno izabirati akcije u svakom stanju. Teorem uspoređuje pohlepnu politiku i trenutnu politiku. Zbog usporedbe nagrada svih stanja, uklanja se problem miopičnosti te teorem garantira da je pohlepna politika bolja od trenutne. Poboljšanje politike izvodi se iterativno kroz cijeli proces učenja te se time osigurava da će pohlepna politika uzimati u obzir buduće nagrade.

2.3.3. Iteracija politike

Bazom ovog algoritma smatraju se Bellmanove jednadžbe. U MDP-u koji se sastoji od n stanja, postoji po jedna Bellmanova jednadžba za svako stanje, odnosno sveukupno n Bellmanovih jednadžbi. Rješavanje seta od n linearnih jednadžbi nije izuzetno težak problem uz korištenje algebarskih tehnika. No, u slučaju Bellmanovih jednadžbi radi se o ne-linearnim jednadžbama. Stoga se ovaj problem rješava iterativnim pristupom. Iterativni pristup započinje inicijalnom pretpostavkom vrijednosti stanja. Vrijednosti iz inicijalne pretpostavke uvrštavaju se u desnu stranu Bellmanovih jednadžbi. Time se dobiva ažurirana vrijednost stanja te se vrijednosti iz inicijalne pretpostavke zamjenjuju izračunatom vrijednošću. Tako je moguće iterativno ažurirati vrijednosti. Teorem poboljšanja politike garantira nam da je moguće ovakvim pristupom konvergirati do optimalnog rješenja. [6]

Iteracija politike kombinira korake estimacije politike i poboljšanja politike. Poboljšanje politike π uz pomoć v_π koje rezultira politikom π' moguće je izračunati poboljšani $v_{\pi'}$ te tako generirati još bolju politiku π''. Tako se algoritmi ponavljaju te kao rezultat imamo sljedeću sekvencu poboljšanih politika i vrijednosnih funkcija:

$$\pi_0 \xrightarrow{E} v_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} v_{\pi_1} \xrightarrow{I} \ldots \xrightarrow{I} \pi_* \xrightarrow{E} v_{\pi_*}$$

Uz pomoć simbola $\xrightarrow{E}$ opisuje se poziv algoritma estimacije politike, dok simbol $\xrightarrow{I}$ označava poboljšanje politike.

2.3.4. Iteracija vrijednosti

Velika mana algoritma iteracije politike je njegova kompleksnost i složenost kako u implementacijskom smislu, tako i u smislu zahtjevnosti resursa. Komputacijski troškovi ovog al-

goritma mogu biti iznimno veliki zbog potrebe evaluiranja politike u svakoj iteraciji. Evaluiranje politike može zahtijevati nekoliko prolaza (engl. *sweep*) kroz set stanja, što povećava komputacijske troškove [10]. Zbog načina na koji iterativni pristup funkcionira, do optimalnog rješenja dolazi se kada broj iteracija teži u beskonačnost. No, smatra se kako problem može imati više rješenja koji su dovoljno blizu optimalnom rješenju, te se zbog toga nakon određenog broja iteracije (nakon što promjena postane dovoljno mala) ovaj postupak zaustavlja. Dakle, broj izvršavanja potreban za pronalazak optimalnog rješenja nalazi se u rasponu od 1 do $\inf$.

Dok se iteracija politike sastoji od ponavljajućeg niza algoritama "Evaluacija politike" i "Poboljšanje politike", iteracija vrijednosti funkcionira nešto drukčije. Kod iteracije vrijednosti koraci su sljedeći [10, str. 83][29]:

- Pronalazak optimalne vrijednosne funkcije
  - Poboljšanje politike
  - Skraćena verzija evaluacije politike
- Ekstrakcija politike iz optimalne vrijednosne funkcije

Iteracija vrijednosti Bellmanovu jednadžbu pretvara u pravilo ažuriranja koji izražen kao matematički izrazi ima sljedeći oblik:

$$\begin{aligned} v_{k+1}(s) &\doteq \max_a \mathbb{E}_\pi[R_{t+1} + \gamma v_k(S_t + 1)|S_t = s, A_t = a] \\ &= \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_k(s')], \forall s \in \mathcal{S} \end{aligned}$$

Ovaj postupak izvršava se samo jednom te ga nije potrebno ponavljati kao kod algoritma Iteracije Politike. Postupak nije potrebno ponavljati jer se iz optimalne vrijednosne funkcije vrlo lako dobiva optimalna politika. Skraćenom verzijom evaluacije politike smatra se verzija u kojoj se izvršava samo jedan prolaz kroz set stanja. Iteracija Vrijednosti je time veoma sličan algoritam Iteraciji Politike. Primarna razlika između ova dva algoritma je to da Iteracija Vrijednosti kombinira korake Evaluacija Politike i Poboljšanje Politike u jedan korak. Iteracija Vrijednosti se također koristi i Bellmanovim max operatorom u koraku evaluacije politike. Time se u ovom algoritmu izabire akciju koja rezultira maksimalnom nagradom, dok se kod Iteracije Politike maksimizacija vrijednosti akcija događa u poboljšanju politike.

2.4. Monte Carlo metode

Algoritmi obrađeni u ovom radu do sad bazirali su se na potpunom ili djelomičnom modelu agentovog okružja. Za potrebe učenja pomoću Monte Carlo metoda agentu nije potreban model. Ovaj pristup može naučiti optimalnu politiku i bez modela te mu je za učenje potrebno

samo iskustvo (engl. *experience*). Pod iskustvom podrazumijevaju se sekvence stanja, akcija i nagrada. Iskustvo može biti ili stvarno, dobiveno direktno od okružja, ili simulirano, dobiveno simulacijom interakcije s okružjem. Za pristup sa simuliranim okružjem potreban je model okružja, no nije nužno da model generira potpuni set distribucija kao što je to slučaj s dinamičkim programiranjem, već je dovoljno da generira sekvencu. Monte Carlo metode funkcioniraju izračunavanjem prosjeka povratnih vrijednosti te su stoga korisne samo za epizodične probleme [10]. Ove metode kreiraju estimacije vrijednosti bez korištenja bootstrapping-a. To jest, u koracima evaluacije politike ili optimizacije, Bellmanove jednadžbe se ne pozivaju u slučajevima gdje s obje strane jednadžbe imamo nepoznanice. Monte Carlo metode koriste se nasumičnim povratnim vrijednostima iz uzorkovane sekvence kako bi se direktno estimirale vrijednosti [30].

Velika prednost Monte Carlo metoda u odnosu na dinamičko programiranje je ne korištenje Bellmanovih jednadžbi i bootstrappinga [31]. Tako se ovim metodama smanjuje kompleksnost i komputacijski zahtjevi. Vrijednost jednog stanja u Monte Carlo metodama ne ovisi o vrijednostima drugih stanja, te stoga trošak izračuna vrijednosti jednog stanja ne ovisi o ukupnom broju stanja. Također, estimacije stanja mogu se fokusirati na ona stanja koja nam doprinose cilju. Kod Dinamičkog programiranja svaka iteracija estimacije politike radi prolaz kroz sva stanja neovisno o njihovoj intrinzičnoj vrijednosti. Tako se također mogu ostvariti znatna ubrzanja algoritma kao i poboljšanja u efikasnosti.

2.4.1. Monte Carlo predviđanje

Monte Carlo predviđanje (engl. *Monte Carlo Prediction*) je algoritam koji pokušava naučiti vrijednosti stanja za definiranu politiku. Kao što je već rečeno, vrijednost stanja s definira se veoma jednostavno te definicija glasi kao očekivana povratna vrijednost stanja s, odnosno očekivani kumulativ diskontnih nagrada dobivenih praćenjem politike u slučaju kada se kreće iz stanja s. Glavna ideja ovog algoritma je generiranje uzoraka u smislu epizoda i izračunavanje prosječne povratne vrijednosti nakon što je stanje s posjećeno. Nakon dovoljnog broja prolaza kroz određeno stanje, vrijednost tog stanja konvergira prema stvarnoj vrijednosti. Tako i predviđanja postaju sve točnija [10].

Postoje dva pristupa unutar estimiranja vrijednosti stanja, a to su Monte Carlo metoda prvog posjeta (engl. *first-visit Monte Carlo method*) te Monte Carlo metoda svakog posjeta (engl. *every-visit Monte Carlo method*) [10, str. 92]. Implementacije ove dvije metode su veoma slične, razlika je u načinu na koji rukuju s ponovljenim dolascima u određeno stanje. Okružjem je moguće navigirati tako da se u pojedino stanje dođe nekoliko puta. Time se dolazi do pitanja što napraviti kod ponovnih dolazaka. Kod first-visit MC metode za izračun vrijednosti stanja u obzir se uzima samo prvi dolazak u stanje. Tako ova metoda ostaje nepristrana zato što se svaki dolazak broji samo jednom. Kod every-visit MC metode svaki dolazak u stanje ulazi u izračun prosjeka. Obje metode imaju svoje teoretske i praktične primjene te izabir implementacije ovisi o specifičnostima okružja u kojem se problem nalazi.

Algoritam [10, str. 92] funkcionira tako da se pomoću politike π generira sekvenca akcija, stanja i nagrada koja ima sljedeći oblik:

$$S_0, A_0, R_1, S_1, A_1, ..., S_{T-1}, A_{T-1}, R_T$$

Politika π se u početku algoritma postavlja proizvoljno, to jest prilikom prve iteracije akcije se mogu birati nasumičnim odabirom. Pseudokod every-visit verzije algoritma izgleda:

```
G = 0
Za svaki korak t u generiranoj sekvenci:
    G = γG + R_{t+1}
    Dodaj G u PovratneVrijednosti(S_t)
    V(S_t) = prosjek(PovratneVrijednosti(S_t))
```

Verzija algoritma s implementacijom first-visit funkcionira na podosta sličan način, no s jednom bitnom razlikom. Razlika između algoritama je u tome da je u ovoj verziji algoritma potrebno provjeriti postoji li već povratna vrijednost za određeno stanje. Pseudokod slijedi:

```
G = 0
Za svaki korak t u generiranoj sekvenci (od T, prema 0):
    G = γG + R_{t+1}
    Ako se S_t nije pojavio u sekvenci dosad:
        Dodaj G u PovratneVrijednosti(S_t)
        V(S_t) = prosjek(PovratneVrijednosti(S_t))
```

Osim vrijednosti stanja, Monte Carlo metode nude nam algoritme i za procjenu vrijednosti para stanje-akcija. Odnosno predviđanje vrijednosti za q_*, to jest $q_\pi(s, a)$. Kao što je obrađeno u prethodnim poglavljima, vrijednost para stanje-akcija označava povratnu vrijednost dobivenu kada agent kreće iz stanja s, napravi akciju a te se u daljnjim vremenskim koracima vodi politikom π.

Rješenje ovog problema veoma je slično rješenju za vrijednost stanja te su algoritmi koji se koriste za predviđanje vrijednosti para stanje-akcija zapravo samo prilagođeni algoritmi za predviđanje vrijednosti stanja. Kao i u predviđanju vrijednosti stanja, i u ovom slučaju postoje first-visit i every-visit pristupi za rješavanje ovog problema te se vrijednost para predviđa praćenjem uzorkovane sekvence [10].

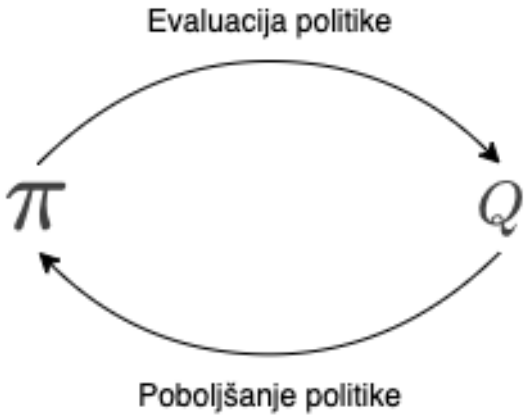
2.4.2. Monte Carlo kontrola

Monte Carlo metode povezane su s generaliziranom iteracijom politike. Izgled generalizirane iteracije politike (u daljnjem tekstu GPI, engl. *Generalized Policy Iteration*) [10, str. 86] prilagođen za Monte Carlo metode vidljiv je na dijagramu 3.

Slika prikazuje generaliziranu iteraciju politike u kojoj se u fazi evaluacije politike ažuriraju predviđene vrijednosti Q-funkcije. Q-funkcija označava vrijednost određene akcije u pojedinom stanju. U fazi poboljšanja politike se predviđene vrijednosti Q-funkcije koriste kako bi se poboljšala politika. Tranzicijska sekvenca ovog prilagođenog GPI-a izgleda ovako:

$$\pi_0 \longrightarrow Q_{\pi_0} \longrightarrow \pi_1 \longrightarrow Q_{\pi_1} \longrightarrow ... \longrightarrow Q_{\pi_*} \longrightarrow \pi_*$$

Sekvenca se ponavlja sve dok optimalna politika π_* i optimalne Q-vrijednosti Q_{π_*} nisu



Slika 3: Prikaz generalizirane iteracije politike

generirani. Time se dolazi do velikog problema koji iskazuje značaj istraživanja. Politika se poboljšava prema iskustvu koje agent dobije od okružja. Iskustvo ovisi o odabranim akcijama, a odabir akcija ovisi o trenutnoj politici. Politika akcije odabire prema predviđenim vrijednostima ($Q(s, a)$). Predviđene vrijednosti mogu biti točne kada je agent pri kraju učenja, no mogu biti i netočne, npr kada je agent tek na samom početku učenja. To znači da može postojati akcija a u stanju s za koju je predviđena vrijednost $Q(s, a)$ na početku mala. U tom slučaju akcija a neće nikad biti izabrana, te će optimalan izbor biti u potpunosti preskočen [32]. Ovaj problem se u literaturi naziva problemom *održavanja istraživanja* (engl. *maintaining exploration*) [10, str. 96].

Rješenje problema održavanja istraživanja leži u dva moguća pristupa [10, str. 96]. Prvi pristup naziva se *početak istraživanjem* (engl. *exploring start*). Početak istraživanjem omogućava istraživanje tako da definira početak svake epizode. Svaka epizoda započinje nasumično odabranim stanjem $S_0 \in \mathcal{S}$ te nasumično odabranom akcijom $A_0 \in \mathcal{A}$. Time se osigurava da se u beskonačno velikom broju ponavljanja/epizoda svaki par stanja-akcije ponovi beskonačan broj puta te se tako Q-vrijednost svakog stanja i akcije ažurira u nekom trenutku.

Drugi pristup se često koristi zbog ograničenja koje ima pristup početka istraživanjem. Početak istraživanjem pristup nije prikladan za svaku vrstu problema. U nekim slučajevima je početno stanje predefinirano te se ne može izabrati nasumičnim odabirom. Kod takvih problema do izražaja dolaze stohastičke politike koje su spomenute u ranijem poglavlju. Kao što je rečeno, kod stohastičkih politika se za svako stanje definiraju vjerojatnosti odabira za svaku od akcija. Tu tvrdnju matematički izražavamo kao:

$$\pi(a|s) > 0, \forall a \in \mathcal{A}(s), s \in \mathcal{S}$$

Stohastičkim politikama za svaku akciju mogu se definirati početne vrijednosti. Tako se garantira da će svaka akcija biti isprobana barem jednom u svakom stanju.

2.4.3. On-policy

Postoje dva ključna pristupa implementacije Monte Carlo metoda [6, str. 844]. Pristupi se razlikuju po načinu na koji se ažurira politika tijekom učenja, odnosno o broju politika koje agent primjenjuje. On-policy pristup učenje provodi koristeći se samo jednom politikom. Politika zaslužna za donošenje odluka je ujedno politika i koja se evaluira i poboljšava. On-policy MC metode često se implementiraju korištenjem stohastičke politike. Sutton i Barto [10] takve stohastičke politike (politike u kojima je $\pi(a|s) > 0$ za svaki set stanja-akcije s, a) nazivaju još i "mekanim" politikama. Takve vrste politika su "mekane" u kontrastu na deterministčke, "čvrste", politike. Deterministčke politike nazivaju se "čvrstima" zbog konkretnosti vjerojatnosti. U svakom stanju samo jedna akcija ima vjerojatnost $\pi(a|s) = 1$, dok su sve ostale vjerojatnosti jednake 0. Stohastičke politike započinju s početnim vjerojatnostima između 0 i 1 za svako stanje, te se u procesu učenja vjerojatnosti nepoželjnih akcija u stanju smanjuju, dok vjerojatnosti poželjnih akcija rastu.

Stohastički on-policy pristup implementira se kao ϵ-pohlepna (engl. ϵ-*greedy*) politika [10, str. 100]. Pohlepna politika označava da se bira akcija koja donosi najbolju nagradu. Parametar ϵ definira vjerojatnost u kojoj se akcija ne izabire pohlepno već se odabire nasumičnim odabirom. Vrijednost parametra ϵ definira se u rasponu od 0 do 1, gdje veći broj označava veću vjerojatnost da se akcija izabere nasumično, odnosno agent više istražuje. Minimalna vjerojatnost izabira svake od akcija je:

$$\frac{\epsilon}{|\mathcal{A}(s)|}$$

Vjerojatnost odabira greedy akcije jednaka je:

$$1 - \epsilon + \frac{\epsilon}{|\mathcal{A}(s)|}$$

Algoritam on-policy pristupa glasi:

Listing 2.2: Kod preuzet iz [10, str. 101] te preveden na hrvatski jezik

```
π = proizvoljna "mekana" politika
Q(s,a) = proizvoljne vrijednosti za sve akcije u svakom stanju
PovratneVrijednosti(s,a) = prazna lista povratnih vrijednosti za sve akcije u
    svakom stanju

Za svaku epizodu:
    Generiranje sekvence politikom π: S_0, A_0, R_1, S_1, ..., S_{T-1}, A_{T-1}, R_T
    G = 0
    Za svaki korak t u generiranoj sekvenci (od T, prema 0):
        G = γG + R_{t+1}
        Ako se par S_t, A_t nije pojavio u sekvenci dosad:
            Dodaj G u PovratneVrijednosti(S_t, A_t)
            A(S_t, A_t) = prosjek(PovratneVrijednosti(S_t, A_t))
            A* = argmax_a Q(S_t, a)
            Za svaki a ∈ A(S_t):
                π(a|S_t) = { 1 - ε + ε/|A(S_t)|,   ako a = A*
                           { ε/|A(S_t)|,           ako a ≠ A*
```

2.4.4. Off-policy

Za razliku od on-policy pristupa gdje se koristi samo jedna politika, u off-policy pristupu postoje 2 različite politike [10, str. 103]. Jedna politika zaslužena je za generiranje uzoraka sekvenci, dok se evaluacije i poboljšanja izvršavaju na politici koju pokušavamo izučiti da bude optimalna. Politika zaslužna za istraživanje (engl. *exploratory policy*) zapisuje se kao: $b(a|s)$. U literaturi se još naziva i politikom ponašanja (engl. *behaviour policy*), s obzirom na to da generira "ponašanje", odnosno generira iskustvo. Ciljna politika (engl. *target policy*), ona koja se poboljšava, zapisuje se kao i do sada: $\pi(a|s)$ [32].

Prilikom implementacije off-policy pristupa pretpostavlja se pretpostavka pokrivenosti (engl. *coverage*). Kako bi predviđali vrijednosti za politiku π pomoću epizoda koje generira politika b, pretpostavlja se da se svaka akcija izabrana u politici π bira (barem ponekad) u politici b. To jest, pretpostavka pokrivenosti glasi: $\pi(a|s) > 0 \implies b(a|s) > 0$. Iz pretpostavke slijedi da politika b mora biti stohastička u onim stanjima gdje nije identična politici π. Kao što je navedeno u prošlom poglavlju, ciljna politika često kreće sa stohastičkim pristupom te se vrijednosti smanjuju dok ne postane deterministička. I u off-policy pristupu vrijedi isto. Ciljna politika često kreće kao stohastička te tijekom procesa učenja postaje sve više deterministička. Stoga je bitno da politika ponašanja, ona koja generira iskustvo i osigurava rješenje problema istraživanja, ostane deterministička kako bi se mogla istraživati stanja koja trenutnom ciljnom politikom nisu optimalna [10].

Kao što je rečeno, off-policy pristup koristi politiku ponašanja b za generiranje uzoraka, dok se pomoću tih uzoraka poboljšava ciljna politika π. Time formula za estimaciju izgleda [10, str. 104]:

$$\mathbb{E}_b[G_t|S_t = s, A_t = a] = q_b(s, a)$$

Iz prethodne formule vidljivo je da se estimacija odnosi na politiku b, te se estimacijom određuju q-vrijednosti za politiku ponašanja. Takve vrijednosti nije moguće iskoristiti za poboljšanje ciljne politike. s obzirom na to da distribucije akcija u stanjima i njihove vjerojatnosti mogu biti različite, direktnom primjenom tih vrijednosti dobili bi netočnu ciljnu politiku. Rješenje ovog problema je mehanizam pod nazivom *uzorkovanje po važnosti* (engl. *importance sampling*) [10, str. 104]. Uzorkovanje po važnosti prilagođava težine povratnih vrijednosti prema relativnim vjerojatnostima pojavljivanja njihovih tranzicija u obje politike (politika ponašanja i ciljna politika). Taj omjer se u literaturi naziva *importance-sampling ratio* [10, str. 104]. Vjerojatnost pojavljivanja određene tranzicije s početnim stanjem S_t u politici π izražava se kao [10, str. 104]:

$$\begin{aligned}
&Pr\{A_t, S_{t+1}, A_{t+1}, ..., S_T|S_t, A_{t:T-1} \sim \pi\} \\
&= \pi(A_t|S_t)p(S_{t+1}|S_t, A_t)\pi(A_{t+1}|S_{t+1})...p(S_T|S_{T-1}, A_{T-1}) \\
&= \prod_{k=t}^{T-1} \pi(A_k|S_K)p(A_{k+1}|S_k, A_k),
\end{aligned}$$

Lijeva strana matematičkog izraza predstavlja definiciju vjerojatnosti događanja odre-

đene tranzicije (npr. $A_0, S_1, A_1, S_2, ...$) uz uvjetnu vjerojatnost da su stanja i akcije određene prema politici π. Desna strana izraza prikazuje produkt u kojemu se vjerojatnost svake akcije A_t u stanju S_t ($\pi(A_t|S_t)$ množi s funkcijom vjerojatnosti prijelaza iz stanja S_t u stanje S_{t+1} (p). Desnu stranu izraza moguće je skraćeno napisati koristeći se kompaktnim operatorom za direktni umnožak ($\prod$). Uz pomoć desne strane izraza moguće je prikazati i importance-sampling omjer. Omjer se dobiva tako da se dijeli vjerojatnost pojavljivanja tranzicije u politici π s vjerojatnošću pojavljivanja tranzicije u politici b. Matematički izraz glasi:

$$\begin{aligned}\rho_{t:T-1} &\doteq \frac{\prod_{k=t}^{T-1} \pi(A_k|S_k)p(A_{k+1}|S_k, A_k)}{\prod_{k=t}^{T-1} b(A_k|S_k)p(A_{k+1}|S_k, A_k)} \\ &= \prod_{k=t}^{T-1} \frac{\pi(A_k|S_k)}{b(A_k|S_k)}\end{aligned}$$

Definirani omjer koristi se pri izračunu vrijednosne funkcije kao i q-vrijednosti. Primjenom omjera formula za vrijednost stanja prolazi sljedeće matematičke operacije:

$$\begin{aligned}v_b(s) &= \mathbb{E}_b[G_t|S_t = s] \\ v_\pi(s) &= \mathbb{E}[\rho_{t:T-1}G_t|S_t = s]\end{aligned}$$

Na isti način se definira i prilagođena q-funkcija:

$$\begin{aligned}q_b(s, a) &= \mathbb{E}_b[G_t|S_t = s, A_t = a] \\ q_\pi(s, a) &= \mathbb{E}[\rho_{t:T-1}G_t|S_t = s, A_t = a]\end{aligned}$$

Pseudokod za off-policy algoritam sličan je pseudokodu za on-policy algoritam. Do razlika dolazi u dodatnoj politici ponašanja. Osim toga se još primjenjuje i importance-sampling omjer kojeg prikazujemo kao W_t. Pseudokod glasi:

Listing 2.3: Kod je preuzet iz [10, str. 111] te preveden na hrvatski jezik

```
Q(s,a) ∈ R = proizvoljne vrijednosti za sve akcije u svakom stanju
π = argmax_a Q(s,a)
b = proizvoljna politika koja pokriva π
C(s,a) = 0 - lista suma importance-sampling omjera za sve akcije u svakom stanju

Za svaku epizodu:
    Generiranje sekvence politikom π: S_0, A_0, R_1, S_1, ..., S_{T-1}, A_{T-1}, R_T
    G = 0
    W = 1
    Za svaki korak t u generiranoj sekvenci (od T-1, do 0):
```

```
G = γG + R_{t+1}
C(S_t, A_t) = C(S_t, A_t) + W
Q(S_t, A_t) = Q(S_t, A_t) + W/C(S_t,A_t) [G-Q(($*S_t, A_t$*))]
ako promjene izmjene akciju poduzetu ciljnom politikom (A_t ≠ π(S_t)):
    Nastavi na t+1 epizodu
W = W 1/b(A_t|S_t)
```

2.5. Policy Gradient metode

Dosadašnje metode koristile su se estimacijom vrijednosti para stanja i akcije kako bi se izabrala akcija u pojedinom stanju. No, osim tog pristupa postoje i razni drugi. Jedan od tih pristupa temelj je Policy Gradient metoda, u pristupu policy gradient metoda koristi se numerička preferenca koja se daje akciji a u vremenskom trenutku t [10, str. 322]. Za pojam numeričke preference koristi se notacija $H_t(a) \in \mathcal{R}$. Preferenca označava samo da se određenoj akciji daje prednost, no nema nikakav utjecaj na nagradu koja se dobiva akcijom. Za određivanje vjerojatnosti izabira akcije a u trenutku t koristi se takozvana *soft-max distribucija* (engl. *soft-max distribution*) [6, str. 848]. Soft-max distribucija u literaturi se naziva još i Gibbsovom ili Boltzmannovom distribucijom [10, str. 37]. Vjerojatnost odabira akcije matematički glasi [10]:

$$Pr\{A_t = a\} \doteq \frac{e^{H_t(a)}}{\sum_{b=1}^{k} e^{H_t(b)}} \doteq \pi_t(a)$$

Policy gradient metode umjesto učenja vrijednosti para stanja i akcija uče parametriziranu politiku (engl. *parameterized policy*) kojoj za odabir akcije nije potrebna vrijednosna funkcija ($v_\pi(s)$). I dalje vrijednosna funkcija može biti korisna u procesu učenja parametra politike, no nije potrebna za odabir akcije. Za matematički opis vektora parametara politike koristi se notacija $\theta \in \mathcal{R}^{d'}$ [10]. Notacija $\mathcal{R}^{d'}$ označava Euklidski prostor dimenzije d'. Svaki vektor u tom prostoru sastoji se od d' realnih brojeva. Tako se definira i vektor parametara θ. Vektor se sastoji od d' parametara koji imaju utjecaj na odabir akcije u određenom stanju.

Vjerojatnost odabira akcije a u trenutku t i stanju t uz parametar θ glasi [10, str. 321]:

$$\pi(a|s, \theta) = Pr\{A_t = a|S_t = t, \theta_t = \theta\}$$

Također postoje metode koje koriste i vrijednosnu funkciju, u tom slučaju koristi se težinski vektor za vrijednosnu funkciju koji označavamo $w \in \mathcal{R}^d$. Vrijednosna funkcija koja koristi težinski faktor glasi $\hat{v}(s, w)$ [10]. Takva funkcija koristi se karakteristikama stanja te uz pomoć težinskog faktora izračunava vrijednost određenog stanja. Primjer estimacije takvog stanja mogao bi biti scenarij igre u kojoj agent ima tri karakteristike. To su količina zdravlja, količina oklopa te količina metaka koje agent još ima. Uzme li se kao primjer težinski vektor $[0.5, 0.2, -0.1]$ gdje vrijednosti pripadaju ovim karakteristikama: [zdravlje, oklop, metci] te trenutnog stanja koje je $[68, 23, 10]$. Tada bi izračun vrijednosti stanja izgledao:

Početne vrijednosti: $\theta = [0.5, 0.2, -0.1]$, $stanje = [68, 23, 10]$.

$$\begin{aligned} v\{s,\theta\} &= \theta[zdravlje] * stanje[zdravlje] + \theta[oklop] * stanje[oklop] + \theta[metci] * stanje[metci] \\ &= (0.5 * 68) + (0.2 * 23) + (-0.1 * 10) \\ &= 34 + 4.6 - 1 \\ &= 37.6 \end{aligned}$$

Dakle, vrijednost stanja $[68, 23, 10]$ uz težinski vektor $[68, 23, 10]$ glasi 37.6. Izračunatu vrijednost stanja moguće je koristiti u procesu učenja za poboljšanje agenta.

Policy Gradient metode koriste se skalarnom mjerom performansi $J(\theta)$ za učenje parametra politike. Ove metode pokušavaju maksimizirati tu skalarnu mjeru, iz tog razloga predviđa se uspon gradijenta (engl. *gradient ascent*) po mjeri performansi J. To se zapisuje na sljedeći način:

$$\theta_{t+1} = \theta_t + \alpha\widehat{\nabla J(\theta_t)}$$

U ovom slučaju $\widehat{\nabla J(\theta_t)} \in \mathbb{R}^{d'}$ označava stohastičku aproksimaciju gradijenta performansi ovisno o parametru θ_t. Također postoji i podobitelj ovih metoda koje uče aproksimacije za politike kao i za vrijednosne funkcije. Takve metode nazivaju se Actor-critic metodama. U tom kontekstu "actor" označava naučenu politiku, dok "critic" nazivamo naučenu vrijednosnu funkciju, npr. vrijednosnu funkciju stanja [10, str. 331].

Postoje brojni načini parametriziranja politike. Jedan od čestih pristupa, barem u slučajevima gdje je prostor akcija diskretan i ne prevelik, je parametriziranje numeričkih preferenci, dosad zapisano notacijom $h(s, a)$. Takve numeričke preference zapisujemo notacijom: $h(s, a, \theta) \in \mathbb{R}$. Vjerojatnost odabira akcije uzimanjem u obzir numeričku preferencu uz soft-max distribuciju označavamo na sljedeći način:

$$\pi(a|s, \theta) \doteq \frac{e^{h(s,a,\theta)}}{\sum_b e^{h(s,b,\theta)}}$$

Jedna od prednosti parametriziranja politike prema soft-max distribuciji u preferencama akcija je taj da se politika vremenom konvergira u determinističu politiku. Takav pristup u prednosti je u odnosu na ϵ-pohlepni pristup jer u tom pristupu uvijek postoji ϵ vjerojatnost da će akcija biti izabrana nasumično. Konkretna prednost dobiva se u specifičnim slučajevima gdje stohastičnost pristupa gradijenta politike dolazi do izražaja. Kod ovog pristupa moguće je definirati vjerojatnosti za odabir svake akcije, dok se kod pohlepnih ili ϵ-pohlepnih metoda izabire akcija koja donosi najveću nagradu. Problem nastaje u slučaju gdje se vrijednosti pomaknu za malu vrijednost u jednom smjeru. To znači da će se kod metoda gradijenta politike jednostavno izmijeniti vjerojatnosti odabira svake akcije, dok će se kod pohlepnih metoda potencijalno u potpunosti izmijeniti odabir akcije. Tako, iako se vrijednost određenog para stanja i akcije promijenila za samo malu količinu, krajnji rezultat (odabir akcije u određenom stanju) promijenio

se veoma značajno. Metode gradijenta politike nude upravo tu granularnost pomoću koje je moguće za malu količinu "pogurati" određenu akciju prema optimalnosti, dok pohlepne metode ne nude toliku razinu detaljnosti, nijansiranosti niti razine preciznosti.

Najčešće korišteni procjenitelj gradijenta politike koji se koristi u algoritmima stohastičkog uspona gradijenta ima sljedeći oblik [33, str. 2]:

$$\hat{g} = \hat{E}_t[\nabla_\theta log \pi_\theta(a_t|s_t)\hat{A}_t]$$

Pri čemu je π_θ stohastička politika, a $\hat{A}_t$ je procjena prednosti (engl. *advantage*) u vremenskom trenutku t. Cijeli izraz $\hat{E}_t$ označava estimaciju prosječne vrijednosti izračunate iz uzorkovanog seta podataka. Implementacije policy gradient metoda koje koriste softver za automatiziranu diferencijaciju (npr. biblioteke ili okvire koji pružaju alate za efikasan izračun gradijenata ovisno o ulaznim varijablama) funkcioniraju tako da se kreira ciljna funkcija čiji gradijent je estimator gradijenta politike ($\hat{g}$). Taj se gradijent dobiva diferencijacijom ciljne funkcije: [33].

$$L^{PG}(\theta) = \hat{E}_t[\log \pi_\theta(a_t|s_t)\hat{A}_t$$

Izračunom diferencijacije (odnosno izračunom derivacije/gradijenta funkcije (L^{PG}) u odnosu na θ) dobiva se gradijent navedene funkcije koji prikazuje smjer i veličinu "najstrmijeg" uspona u ciljnoj funkciji. Schulman et al. [33] navode da iako zvuči primamljivo izvoditi nekoliko koraka optimizacije na ovoj funkciji gubitka koristeći se istom tranzicijom/putanjom, u praksi to nije preporučljivo. Takva optimizacija vodi do "destruktivno velikih ažuriranja politike" [33, str. 2].

2.5.1. Policy Gradient Theorem

Teorem gradijenta politike ključni je koncept za razumijevanje obitelji metoda gradijenta politike.

Kontinuirani zadatci i epizodički zadatci definiraju skalarnu mjeru performansi na različite načine. U epizodičkim zadatcima mjera performansi definira se kao vrijednost početnog stanja ($v(s)$) u epizodi [10, str. 324]. Skalarna mjeru performansi prikazujemo kao: $J(\theta) \doteq v_{\pi_\theta}(s_0)$ gdje se s v_{π_θ} označava vrijednosna funkcija politike određene parametrom, odnosno π_θ. Aproksimacija funkcije može biti izazovan zadatak jer promjene parametra politike imaju utjecaj na odabir akcije ali i na distribucije stanja u kojima se akcije biraju. Tim dvjema utjecajima direktno se utječe na performanse trenutne politike. Utjecaj parametra politike na odabir akcija je opće poznat te je stoga vrlo lagano izračunati akcije koje će biti odabrane vodeći se određenim parametrom te tako i povratnom nagradom. No, utjecaj politike na distribuciju stanja nije opće poznat s obzirom na to da se radi o komponenti samog okružja. Postavlja se pitanje kako je moguće predvidjeti gradijent performansi prema parametru politike kada ta komponenta nije poznata. U razumijevanju tog problema pomaže nam teorem gradijenta politike. Teorem služi kao vrsta matematičkog izraza za zapis gradijenta performansi prema

parametru politike. Teorem za epizodičke zadatke glasi [10]:

$$\nabla J(\theta) \, \alpha \sum_{s} \mu(s) \sum_{a} q_{\pi}(s,a) \nabla_{\pi}(a|s,\theta)$$

U ovom izrazu $\nabla J(\theta)$, kao što je već objašnjeno, predstavlja gradijent skalarne mjere performansi J prema parametru θ. Simbol α predstavlja operator proporcionalnosti. U ovom slučaju lijeva strana (gradijent skalarne mjere performansi) direktno je proporcionalna desnoj strani. Točnije, proporcionalna je sumi svih stanja $\sum_s$ i njihovih distribucija $\mu(s)$ pomnoženo sa sumom po svim akcijama $\sum_a$ u kojoj se sumiraju q vrijednosti para stanje-akcija prema politici π, odnosno $q_{\pi}(s,a)$ pomnoženo s gradijentom odabira akcije a u stanju s uzimajući u obzir parametar θ $\nabla\pi(a|s,\theta)$.

2.5.2. REINFORCE

REINFORCE [10, str. 326] algoritam ključan je algoritam u obitelji metoda gradijenta politike. Njegova važnost potječe iz toga što je to algoritam na kojemu su naprednije metode gradijenta politike bazirane. Naprednije metode nadograđuju ovaj algoritam te rješavaju njegove brojne probleme. Naziv ove metode dolazi iz jednadžbe koja se koristi u ovom algoritmu. Formula glasi: "**RE**ward **I**ncrement = **N**onnegative **F**actor × **O**ffset **R**einforcement × **C**haracteristic **E**ligibility". U prijevodu naziv označava "Povećanje nagrade = Nenegativni faktor * Pomak nagrade * Karakteristična važnost". Pomak nagrade u ovom slučaju odnosi se na razliku između predviđene vrijednosti i stvarne vrijednosti, dok karakteristična važnost predstavlja udio u kojoj pojedina karakteristika ima utjecaj na vrijednost.

REINFORCE algoritam može se smatrati nadogradom Monte Carlo metode. Bitna karakteristika Monte Carlo metoda je stvaranje uzoraka. Na isti način i ova metoda generira uzorke te procjenjuje povratnu vrijednost i ovisno o povratnoj vrijednosti ažurira parametre politike. Sutton i Barto [10] navode kako je način generiranja uzoraka od ključne važnosti. Gradijenti uzoraka ne moraju biti jednaki stvarnom gradijentu, već je dovoljno da budu proporcionalni. Teorem gradijenta politike daje nam izraz koji je proporcionalan stvarnom gradijentu, te on u sebi sadrži komponentu α koja označava veličinu koraka te se u njega može pohraniti bilo koja konstanta proporcionalnosti. Desna strana teorema gradijenta politike može se zapisati i na sljedeći način [10, str. 326]:

$$\begin{aligned} \nabla J(\theta) \, \alpha \, & \sum_{s} \mu(s) \sum_{a} q_{\pi}(s,a) \nabla\pi(a|s,\theta) \\ & = \mathbb{E}_{\pi} \left[\sum_{a} q_{\pi}(s,a) \nabla\pi(a|s,\theta) \right] \end{aligned}$$

Iz toga slijedi da se algoritam stohastičkog uspona gradijenta može definirati kao [10,

str. 326]:

$$\theta_{t+1} \doteq \theta_t + \alpha \sum_a \hat{q}(S_t, a, w) \nabla \pi(a|S_t, \theta)$$

Rastavljanje izraza na dijelove smanjuje kompleksnost izrazu te olakšava razumijevanje. Za korištenje ovog pravila ažuriranja te kreiranje novih parametara politike θ_{t+1} u trenutku $t+1$ potrebna su dvije vrijednosti. Prva je vrijednost parametara u trenutku t. Druga potrebna vrijednost je suma po svakoj akciji u kojoj se predviđena vrijednost stanja S_t uz akciju a te težinskim faktorom w množi s gradijentom vjerojatnosti odabira same akcije a u stanju S_t uz parametre θ. Cilj navedenog izraza je promijeniti parametre θ tako da se potaknu akcije s većim predviđenim vrijednostima. Ovakav algoritam naziva se metodom svih akcija (engl. *all-actions*) čisto zato što ažuriranje uključuje sve akcije. No, također postoji i klasični pristup REINFORCE algoritmu koji je razvio Williams 1992. godine [34]. U ovom pristupu ažuriranje u trenutku t uključuje samo izabranu akciju u tom trenutku, to jest A_t [10]. Klasični pristup REINFORCE algoritmu dobiva se tako da se u prethodni matematički izraz ukomponira izabiranje akcije a. Dosadašnji pristup nije ponderirao vrijednosti akcija u pojedinom stanju. Mehanizam ponderiranja dodaje se na sljedeći način [10, str. 327]:

$$\begin{aligned}
\nabla J(\theta) \; \alpha \; \mathbb{E}_\pi \left[\sum_a \pi(a|S_t, \theta) q_\pi(s, a) \frac{\nabla \pi(a|S_t, \theta)}{\pi(a|S_t, \theta)} \right] \\
= \mathbb{E}_\pi \left[q_\pi(S_t, A_t) \frac{\nabla \pi(A_t|S_t, \theta)}{\pi(A_t|S_t, \theta)} \right] \\
= \mathbb{E}_\pi \left[G_t \frac{\nabla \pi(A_t|S_t, \theta)}{\pi(A_t|S_t, \theta)} \right]
\end{aligned}$$

Tako se dobiva izraz pomoću kojeg je moguće izračunati očekivanu vrijednost koja je proporcionalna gradijentu. Algoritam stohastičkog gradijenta za klasični pristup REINFORCE metodi glasi:

$$\theta_{t+1} \doteq \theta_t + \alpha G_t \frac{\nabla \pi(A_t|S_t, \theta)}{\pi(A_t|S_t, \theta)}$$

U prethodnom izrazu gradijent vjerojatnosti odabira akcije A_t u stanju S_t uz parametar θ dijeli se s vjerojatnošću odabira akcije A_t u stanju S_t s parametrom θ. U ovom matematičkom izrazu nalazi se suptilna razlika jer se u brojniku nalazi *gradijent* vjerojatnosti, dok se u nazivniku nalazi konkretna vjerojatnost. Rezultat takvog dijeljenja je vektor smjera u parametarskom prostoru (R^d) koji povećava vjerojatnost odabira akcije A_t u stanju S_t. Ažuriranje parametara takav vektor koristi za pomicanje proporcionalno povratnoj vrijednosti te inverzno proporcionalno vjerojatnosti. Prva činjenica znači da se povećava vjerojatnost akcijama s većim vrijednostima. Druga činjenica znači da će akcija s manjom vjerojatnošću odabira imati veći utjecaj na veličinu ažuriranja. Odnosno, ako imamo dvije akcije a_1 i a_2 s vjerojatnostima odabira 0.1 i 0.9, ukoliko se pri odabiru obje akcije dobije velika povratna vrijednost X, tada će se vjerojatnost za odabir akcije a_1 više povećati nego vjerojatnost za odabir akcije a_2. Takvo ponašanje veoma je po-

željno jer bi bez tog mehanizma akcije koje imaju visoku vjerojatnost bile izabrane vrlo često te bi njihova vjerojatnost rasla vrlo brzo. Rezultat takvog pristupa je taj da se gubi poželjno istraživanje te algoritam teži iskorištavanju (odnosno algoritam izbjegava druge akcije) [32].

s obzirom na to da REINFORCE metoda ima svojstva metoda stohastičkog gradijenta, prema svojoj teoretskoj podlozi ima veoma dobra svojstva konvergiranja. No, upravo zbog svoje teoretske podloge ima i neke veoma nepoželjne karakteristike. Mana Monte Carlo metoda je visoka varijanca što rezultira nestabilnim ažuriranjima vrijednosti te samim time i sporim učenjem. Sporo učenje simptom je buke koja je prisutna u podatcima te je za ispravno učenje potrebna veća količina podataka kako bi se uočili stvarni uzorci u setu podataka. Problem visoke varijance moguće je kompenzirati implementacijom takozvane referentne vrijednosti (engl. *baseline*) $b(s)$ u teorem gradijenta politike. Takav teorem izgleda [10, str. 329]:

$$\nabla J(\theta)\ \alpha \sum_{s} \mu(s) \sum_{a} \Big(q_\pi(s,a) - b(s) \Big) \nabla \pi(a|s,\theta)$$

Pri čemu referentna vrijednost $b(s)$ može biti bilo kakva funkcija sve dok se ne mijenja ovisno o akciji a. Pravilo ažuriranja koje ukomponira referentnu vrijednost izgleda [10, str. 329]:

$$\theta_{t+1} \doteq \theta_t + \alpha \Big(G_t - b(S_t) \Big) \frac{\nabla \pi(A_t|S_t,\theta)}{\pi(A_t|S_t,\theta)}$$

2.5.3. AC metode

Actor-Critic metode mogu se smatrati nadogradom REINFORCE metode. Iako REINFORCE nudi veoma dobar okvir za poticano učenje, neke od mana ovog algoritma ograničavaju njegove performanse i učinkovitost u pojedinim scenarijima. Glavni cilj Actor-critic metoda je akomodirati neka od ograničenja te poboljšati stabilnost učenja kao i njegovu efikasnost. Velika mana REINFORCE algoritma je visoka varijanca što rezultira nestabilnim i sporijim učenjem. Visoka varijanca potječe iz Monte Carlo estimacije gradijenta politike. Time se svakom uzorkovanom tranzicijom unose nestabilnosti te proces učenja može biti iznimno osjetljiv na sami set uzorkovanih tranzicija.

Konda i Tsitsiklis [35] navode kako se većina metoda u poticanom učenju, kao i u Neuro-Dinamičkom programiranju vode jednim od dva pristupa [35, str. 1]:

- Samo-akterski (engl. *Actor-only*) pristup u kojem postoji parametrizirana obitelj politika. Gradijent performansi u odnosu na parametre se direktno estimiraju simulacijom. Navedeni parametri zatim se "guraju" u smjeru poboljšanja. Mane ovog pristupa su moguća visoka varijanca te nedostatak pravog učenja, to jest stare informacije se ne iskorištavaju. Prilikom kreiranja novog gradijenta nakon izmjene politike u obzir se ne uzimaju prethodni gradijenti.
- Samo-kritičarske (engl. *Critic-only*) metode oslanjaju se isključivo na aproksimaciju vrijednosne funkcije uz pomoć Bellmanovih jednadžbi. s obzirom na to da ova obitelj metoda ne poboljšava politiku na direktan način, moguće je da je rezultat ovakvog učenja dobra

aproksimacija vrijednosne funkcije no rezultirajaća politika ne mora nužno biti optimalna.

Cilj actor-critic metoda je kombinirati dva navedena pristupa kako bi se iskoristile jake strane ovih pristupa, te minimizirale mane svakog pristupa pojedinačno. Takozvani critic koristi se aproksimacijom i simulacijom kako bi naučio vrijednosnu funckciju uz pomoć koje se ažuriraju parametri politike na actor strani metode. Tako je moguće garantirati konvergenciju u optimalnu politiku čime se uklanja mana critic-only metoda. Za razliku od actor-only metoda varijanca je u ovom pristupu znatno manja što rezultira bržim učenjem, odnosno bržom konvergencijom [35].

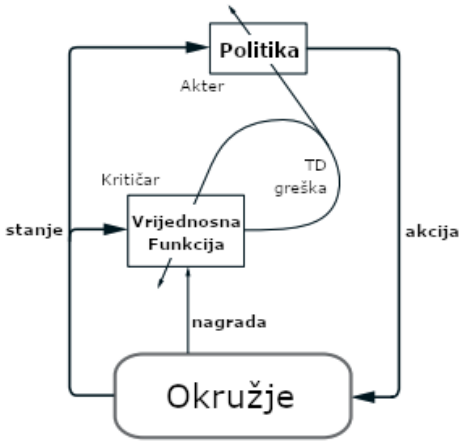
Kod REINFORCE metode s referentnom vrijednosti, postojeća vrijednosna funkcija estimira vrijednost samo prvog stanja u svakoj tranziciji. Takva procjena provodi se na početku tranzicije što znači da se pomoću nje ne može izračunati vrijednost prve akcije, pošto se izvršava prije akcije. Actor-critic metode taj problem rješavaju tako da se vrijednosna funkcija poziva i za drugo stanje tranzicije, čime se u vrijednost stanja uključuju set prva dva stanja te prva akcija poduzeta u toj tranziciji. Predviđenom vrijednošću drugog stanja diskontiranom i dodanom u povratnu vrijednost dobivamo povratnu vrijednost jednog koraka što pišemo $G_{t:t+1}$. Ova metoda predviđanja vrijednosti akcija naziva se *critic* metodom. Actor-critic metode dijele reprezentaciju vrijednosne funkcije od same politike, te se dvije strane ove metode koriste komplementarno. Actor, od engleskog korijena riječi djelovati (engl. *act*) označava strukturu koja djeluje, odnosno koji izabire akcije. Critic je dio ove arhitekture koja kritizira akcije koje actor poduzima, odnosno vrednuje te akcije. Ovakav pristup uvijek se obavlja on-policy pristupom. Critic vrednuje trenutnu politiku kojom se actor vodi. Njegove kritike izražuju se kao skalarni signal u obliku pogreške vremenske (TD) razlike (engl. *Temporal difference error*). Pogreška vremenske razlike opisuje razliku u vrijednosti između trenutnog stanja i budućeg stanja. Matematički ju pišemo:

$$\delta_t = R_{t+1} + \gamma V_t(S_{t+1}) - V(S_t)$$

Dakle, TD pogreška u trenutku t izračunava se kao zbroj vrijednosti dobivene nagrade i razlike između vrijednosti trenutnog stanja te vrijednosti predviđenog budućeg stanja. Simbolom V_t označavamo vrijednosnu funkciju koju trenutno critic implementira. Ako je δ_t pozitivna to označava da se odabir ove akcije u budućnosti treba povećati. Ako je δ_t negativna tada bi se izabrana akcija trebala manje izabirati u budućnosti. Sljedeća slika ilustrira interakciju između actora, critica i okružja.

2.5.4. Proximal Policy Optimization

Metoda proksimalne optimizacije politike (u daljnjem tekstu PPO, engl. *Proximal Policy Optimization*) [33] najnoviji je standard u području poticanog učenja. Originalna dokumentacija koja je uvela ovu metodu objavljena je 2017. godine. Algoritam su razvili John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, te OpenAI. Algoritam je objavio OpenAI pod nazivom "Proximal Policy Optimization Algorithms" [33]. Motivacija za ovim algoritmom bilo je nezadovoljstvo postojećim algoritmima poticanog učenja. Većina pristupa imali su značajne mane koje su smanjivale korisnost algoritama. Većina pristupa imalo je mane u veoma važnim



Slika 4: Prikaz actor-critic arhitekture, izvor: [21].

aspektima kao što su stabilnost algoritma, efikasnost sa stajališta podataka te mogućnost rukovanja kompleksnim okružjima [33, str. 1]. Schulman et al. [33] kao preporučene pristupe u vrijeme objavljivanja algoritma navode navode Duboko Q-učenje (skraćeno "DQN", engl. *Deep Q-Learning*), "vanilla" policy gradient metode te trust region metode. Kao mane DQN pristupa navode [33, str. 1] kako nije primjenjiv ni na mnogim jednostavnim problemima te osobito pati u situacijama gdje se algoritam susreće sa kontinuiranim problemima, vanilla policy gradient metode su neefikasne te nisu robusne/stabilne, TRPO (engl. *Trust Region Policy Optimization*) metoda na kojoj se zapravo i bazira PPO je prekomjerno kompleksna, teška za implementirati te nije prikladna na arhitekture koji uključuju buku u kontekstu nasumičnosti.

Cilj PPO metode je adresirati ograničenja i izazove s navedenim pristupima te uvesti novi pristup koji je komputacijski efikasan, stabilan, te prilagodljiv za veliki broj problema i okružja. Ideja je bila imati algoritam koji je koristan i za jednostavne probleme, ali ima i mogućnost rukovanja sa kompleksnim i visoko-dimenzionalnim problemima. Osim toga PPO pokušava pronaći rješenje dileme istraživanja i iskorištavanja te tako pruža efikasna ažuriranja politike uz zadržavanje stabilnosti.

Kao što je već rečeno, algoritam se bazira na TRPO metodi. TRPO metoda poznata je po svojoj kompleksnosti te stoga ova metoda pokušava zadržati podatkovnu efikasnost i stabilnost TRPO-a koristeći se samo optimizacijom prvog reda. TRPO metoda u svom originalnom obliku [36] koristi se optimizacijom drugog reda u kojoj se koriste derivacije drugog reda koje su značajno "skuplje" komputacijski. Uz veliki trošak, derivacije drugog reda uvode nepotrebnu kompleksnost u algoritam te se uklanjanjem tog dijela olakšava razumijevanje i implementacija metode. U policy gradient metoda obrađenim dosad vidljivo je da se optimizacije izvršavaju tako da se direktno pokušava maksimizirati povratna vrijednost. Naime, PPO u optimizacijsku funkciju uvodi takozvani "Clipping" mehanizam te se omjeri vjerojatnosti (omjer vjerojatnosti odabira akcije prema novoj politici u odnosu na staru politiku). Clipping mehanizam ograničava ovaj omjer u određenom rasponu te se tako ne dozvoljavaju velike promjene. Mehanizmom clippinga kreira se pesimistična procjena performansi politike te se tako prikazuje konzervativniji pogled na performanse politike. Točno taj mehanizam spriječava nestabilnost u učenju te drastične promjene u distribucijskim vrijednostima politike. Za optimizaciju politike provodi

se ponavljajući proces u kojem se kreiraju uzorkovani podatci pomoću politike te se zatim provodi nekoliko epoha optimizacije na uzorkovanim podatcima [33]. Epoha označava jedan potpuni prolaz kroz sve podatke za treniranje agenta te izračun vrijednosti potrebnih za ažuriranje agenta te samo ažuriranje modela odnosno samih parametara modela. U ovom pristupu na jednom setu uzorkovanih podataka provodi se nekoliko iteracija treniranja. Tako se osigurava efikasnost s obzirom na uzorkovane podatke.

U TRPO algoritmu maksimizira se ciljna funkcija (takozvanog "surrogate" cilja), s ograničenjem na veličinu samih ažuriranja politike [33]. Surogatna ciljna funkcija označava izmijenjenu ciljnu funkciju koja služi kao surogat/proxy za stvarnu ciljnu funkciju kojoj je cilj maksimizirati povratnu vrijednost. Surogatna ciljna funkcija dizajnirana je kako bi bila lakša za optimizirati te podliježe ograničenju veličini koraka kako bi promjene politike ostale unutar regije povjerenja (engl. *Trust Region*). Matematički izraz korišten za izračun vrijednosti glasi [33, str. 2]:

$$\max_{\theta} \hat{E}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} \hat{A}_t \right]$$
$$\text{subject to } \hat{E}_t[KL[\pi_{\theta_{\text{old}}}(\cdot|s_t], \pi_\theta(\cdot|s_t)]] \leq \delta$$

Pri čemu $\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}(a_t|s_t)}}$ označava omjer nove distribucije politike u odnosu na staru distribuciju politike. Ovaj omjer označava veličinu promjene uvedene ažuriranjem, što je omjer dalji od 1 to je promjena veća. Prednost (engl. *Advantage*) označava se s $\hat{A}_t$ te označava razliku između vrijednosti nove akcije i stare akcije. Odnosno prednost označava koliko je izabrana akcija *bolja* od akcije koju trenutna politika preporučuje. Novouvedeni pojam KL označava KL divergenciju (engl. *KL divergence*), odnosno Kullback-Leiblerova divergenciju koja predstavlja mjeru različitosti između dvije distribucije vjerojatnosti. Pomoću KL divergencije kvantificira se razlika između dvije distribucije vjerojatnosti. U ovom kontekstu KL divergencija se koristi kako bi dobili razliku između stare i nove neuralne mreže.

U prethodnom zapisu omjer vjerojatnosti može se zapisati kao $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}(a_t|s_t)}}$, odnosno $r(\theta_{\text{old}}) = 1$. TRPO maksimizira surogatni cilj na sljedeći način:

$$L^{CPI}(\theta) = \hat{E}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} \hat{A}_t \right] = \hat{E}_t[r_t(\theta)\hat{A}_t]$$

Oznaka CPI predstavlja konzervativnu iteraciju politike (engl. *Conservative Policy Iteration*). CPI se odnosi na proces iteracije politike u kojem se naizmjence provode koraci evaluacije politike te poboljšanja politike. No, u konzervativnom pristupu iteraciji politike provode se konzervativna ažuriranja politike čime se ograničavaju veličine ažuriranja te se osigurava da nova politika neće biti prekomjerno različita od stare politike [37]. Bez ograničenja prethodnog izraza L^{CPI} pokušaj maksimiziranja rezultirao bih velikim ažuriranjima te se stoga uvode restrikcije na veličinu ažuriranja za bilo koje promjene koje su predaleko od $r_t(\theta) = 1$. Takva ciljna funkcija glasi:

$$L^{CLIP}(\theta) = \hat{E}_t[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t]$$

Navedeni izraz može se smatrati nadogradom L^{CPI} izraza. Prvi pojam unutar $\min$ funkcije zapravo je L^{CPI} izraz. Drugi pojam $\text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)$ označava modificiranu verziju surogatne funkcije u kojoj se omjer vjerojatnosti ograničava tako da $r_t(\theta)$ ostane unutar granica $[1-\epsilon, 1+\epsilon]$. U izrazu simbol ϵ označava hiperparametar koji se najčešće postavlja na vrijednost u rasponu od 0.1 do 0.3 [33]. Korištenjem $\min$ funkcije uzima se manji od dva izraza. Razlog iza korištenja manjeg izraza je taj da se time osigurava stabilnost. Bez $\min$ funkcije ne bi postojalo ograničenje veličine ažuriranja te bi sama ažuriranja mogla biti znatno velika. Problem kod takvih velikih ažuriranja je taj da se njima uvode znatne nestabilnosti u učenju. Prevelikim koracima agent može promašiti cilj te se pomaknuti u krivom smjeru. Navedena funkcija također ograničenja primjenjuje samo na poboljšanja politike, odnosno kada je $\hat{A}_t > 0$. U slučaju kada je $\hat{A}_t < 0$, izmjene na politici se ne ograničavaju. Takvo ponašanje je poželjno zbog višestrukog korištenja istih uzorkovanih podataka. Poboljšanja je potrebno limitirati jer samo zato što je u pojedinom slučaju akcija a bila dobra, ne znači nužno da će ta akcija biti u svakom slučaju dobra. No, u slučaju da se neka akcija pokaže nepoželjnom, tada ažuriranje politike treba biti mnogo značajnije jer ta akcija nije poželjna.

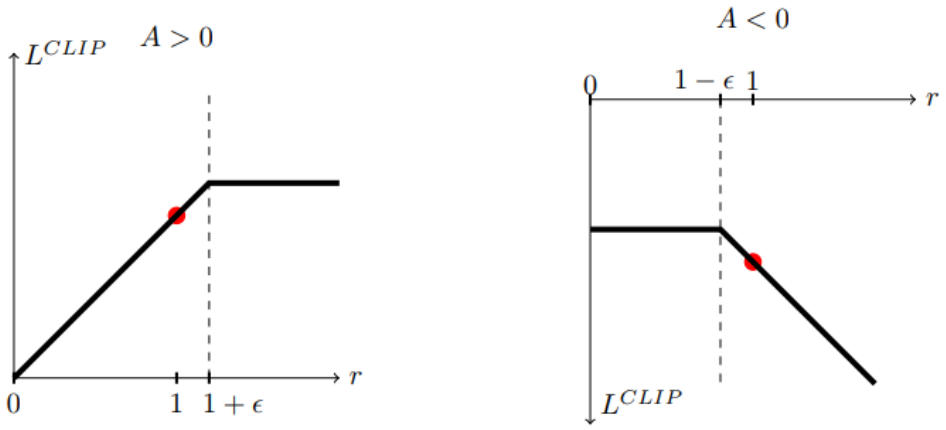
Kao što je već rečeno, omjer r predstavlja omjer vjerojatnosti akcija prema novoj politici u odnosu na staru politiku. Ako je omjer bliže vrijednosti 0, to znači da je vjerojatnost akcija u novoj politici ekstremno mala u odnosu na staru politiku. Kada je vrijednost omjera r na drugom ekstremu (npr. $+\infty$, odnosno veći od 1) to označava da je vjerojatnost akcija u novoj politici znatno veća nego vjerojatnost akcija u staroj politici. Dijagram potanje prikazuje mehanizam clippinga te je vidljivo da je clippinga asimetričan. Clipping omjera na $1+\epsilon$ primjenjuje se kada je $A > 0$, to jest kada je nova akcija bolja od stare akcije. Tada se omjer ograničava na $1+\epsilon$ kako ne bi uveli preznačajnu promjenu. U slučaju kada je $A < 0$, odnosno kada je nova akcija lošija od stare akcije, primjenjuje se $1-\epsilon$ ograničenje omjera. Donja granica označava da nova politika postavlja znatno manje vjerojatnosti odabira akcije u odnosu na staru politiku te se i takvo ažuriranje treba ograničiti. Također se može vidjeti i mehanizam naveden u prošlom paragrafu, ako je prednost $A < 0$, te je omjer iznad 1, vrijednost se ne ograničava. Odnosno, negativne akcije koje nova politika podržava se kažnjavaju.

Osim PPO Clipping metode postoji i drugi pristup, a to je pristup s adaptivnim KL koeficijentom kazne (engl. *Adaptive KL Penalty Coefficient*) [33, str. 4]. U ovom pristupu umjesto clipping mehanizma implementira se kazna proporcionalna veličini KL divergencije. Ovakav pristup često postiže lošije performanse od clipping pristupa, a pristup se bazira na sljedećim koracima [33, str. 4]:

- Optimiziranje ciljne funkcije korištenjem KL-penalizacije uz nekoliko epoha

$$L^{KLPEN}(\theta) = \hat{E}_t\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}\hat{A}_t - \beta KL[\pi_{\theta_{\text{old}}}(\cdot|s_t), \pi_\theta(\cdot|s_t)]\right]$$

- Izračun $d = \hat{E}_t[KL[\pi_{\theta_{\text{old}}}(\cdot|s_t), \pi_\theta(\cdot|s_t)]]$



Slika 5: Prikaz mehanizma clippinga u slučaju pozitivne prednosti (lijevo), te u slučaju negativne prednosti (desno), izvor: [33]

- Ako je $d < d_{\text{targ}}/1.5, \beta \leftarrow \beta/2$
- Ako je $d > d_{\text{targ}}/1.5, \beta \leftarrow \beta \times 2$

Ažurirana vrijednost β parametra koristi se u idućem ažuriranju politike. Schulman et al. [33] navode kako je ponekad moguće pronaći slučaj u kojem je KL divergencija značajno drukčija od d_{targ}, no takvi slučajevi su rijetki te se β parametar vrlo brzo prilagodi. Također, parametri 1.5 te 2 su izabrani heuristički, no algoritam nije osjetljiv na navedene parametre.

PPO algoritam smatra se nadogradom policy gradient metoda te njena prednost leži upravo u jednostavnosti implementacije. Prethodno obrađene funkcije mogu se implementirati u obliku PPO algoritma jednostavnim izmjenama klasične policy gradient metode. Za implementacije koje se koriste automatiziranom diferencijacijom (kao pristup obrađen ranije) moguće je konstruirati funkciju gubitka L^{CLIP} ili L^{KLPEN} umjesto L^{PG} te na istoj provesti nekoliko koraka metode stohastičkog gradijenta uspona.

Algoritam za PPO metodu glasi:

```
Za svaku iteraciju
    Za svakog actora (1, ..., N)
        Izvršavaj politiku π_θold u okružju T vremenskih koraka
        Izračunaj procjene prednosti Â_1, ..., Â_T
    Optimiziraj surogatni cilj L prema θ, sa K epoha i veličinom minibatcha
        M ≤ NT
    θ_old ← θ
```

2.6. DOOM franšiza

Doom franšiza, često pisano stilizirano "DOOM", službeno je započela davne 1993. godine. Danas se franšiza sastoji od 9 igara glavne serije te 9 dodatnih igara koje nisu uključene u glavnu seriju. Osim navedenih igara, franšiza uključuje i 7 romana (*Knee Deep in the Dead*, *Hell on Earth*, *Infernal Sky*, *Endgame Sama*, *Doom 3: Worlds on Fire*, *Doom 3: Maelstrom*, te dodatnog romana nastalom prema filmu), stripa, društvenih igara (engl. *tabletop*) te filmova.

Ideja za originalnom videoigrom DOOM nastala je nedugo nakon objavljivanja popularne videoigre Wolfenstein 3D. Wolfenstein 3D smatra se "djedom" žanra 3D pucačina kao i pucačina iz prvog lica. Ova igra ustanovila je osnove žanra kao što su brzi tempo te potrebe brzih refleksa i ostalih motoričkih sposobnosti. Također je služila kao veliki katalizator za popularnost žanra. Iz same popularnosti ove videoigre studio koji je odgovoran za stvaranje ovog remek djela, Id Software, odlučio je kreirati novu pucačinu iz prvog lica. Plan za novu igru bio je kreirati 3D videoigru uz pomoć game engine-a kojeg je razvijao glavni programer u Id Software studiju, John Carmack. Prije nego što je DOOM kreiran kao koncept, postojala je mogućnost da će se game engine koristiti u novom izdanju igre "Commander Keen". No, suosnivači Id Software-a, John Romero i Adrian Carmack koji su također bili zaduženi za dizajn te vođenje likovnog tima, htjeli su razvijati nešto u mračnijem stilu od Commander Keen igara. Game engine koji je Carmack razvio naziva se *id Tech 1* odnosno "Doom engine". Ta tehnologija išla je u smjeru igre bržeg tempa te igre iz prvog lica. Commander Keen igre nisu posjedovale niti jednu od te dvije karakteristike. Osim Commander Keen također je postojala i ideja igre u *Aliens* svijetu. Svi članovi id software studija su bili veliki obožavatelji ovog filma. Smatrali su da bi takva igra bila odlična te su iznenađujuće prava za takvu videoigru bila dostupna. No, nakon dugog razmišljanja odlučili su da bi bilo bolje stvoriti igru koja nije vezana za već postojeći film te bi tako imali veću slobodu biranja smjera u kojem će igra ići. Tehnologija koju je Carmack osmislio bila je preimpresivna za bilo kakve kompromise. [38].

Konkretna ideja za DOOM igrom došla je Carmacku nakon dugog konceptualiziranja, sama ideja bazirala se na demonima. Inspiracija za takvom idejom bila je sveprisutna. Od demona u katolicizmu do anegdote u kojoj je Romero, dizajner i suosnivač id Softwarea, u jednoj *Dungeons and Dragons* igri sasvim slučajno stvorio ogroman broj demona koji je preplavio zemlju te na kraju i uništio cijeli svijet u toj društvenoj igri. Carmackova ideja bila je iskoristiti izvanrednu tehnologiju koju su izdizajnirali te pomoću nje kreirati igru u kojoj se susreću demoni i tehnologija. U takvoj igri igrač bi koristio oružja visoke tehnologije kako bi porazio najezdu demona te ostale zvijeri koji su došle direktno iz pakla. Romero, kao i ostatak tima bili su oduševljeni ovom idejom. Ideja je bila inovativna te je imala veliki potencijal. Nakon ideje sve što je bilo potrebno je bio naziv. Neki od prvotnih ideja za nazive bili su "Mutants from Hell!", "Die, Mutants, Die", "3-Demons", te "It's Green and Pissed". Finalni naziv ove igre, kao i cijele franšize, inspiriran je filmom iz 1986. godine producenta Martina Scorsesea pod nazivom "Boja novca" (engl. *Color of Money*). U glavnoj ulozi nalazi se mladić pod imenom Vincent Lauria koji je talentirani igrač biljara te kroz svoje avanture u filmu ostvaruje spektakularne rezultate na biljarskim natjecanjima. U jednoj sceni koja je bila inspiracija za nazivom, Tom Cruise u ulozi Vincenta Lauria ulazi u prostoriju s biljarskim stolom te svoj najdraži biljarski štap nosi u crnoj torbi. Njegov suigrač ga zatim upita "Što nosiš u toj torbi?". Vincent nakon osmijeha koji ukazuje na samouvjerenost u svoju pobjedu upita "Ovdje?". Nakon otvaranja torbe kratko izjavljuje "Doom", u prijevodu "Propast" [38].

Tim je odlučio nastaviti s razvojem Doom igre te se početni razvojni tim sastojao od petero ljudi; programera Johna Carmacka te Johna Romera, umjetnika Adriana Carmacka i Kevina Clouda te dizajnera Toma Halla. Nakon početka razvoja, prebacili su se u novi ured prikladnog mračnog interijera. Također su odlučili prestati surađivati s tvrtkom Apogee Software

koja je služila kao izdavačka kuća za njihova prethodna djela. Iako im je Tom, tadašnji direktor tvrtke Apogee, bio bliski prijatelj, odlučili su da im Apogee više nije potreban te su smatrali kako bi bilo bolje igru Doom izdati samostalno. Osim što je ova videoigra popularizirala žanr pucačina iz prvog lica, također je bila pionir svojim fenomenalnim 3D grafikama te igranjem s drugim igračima putem mreže. Također je nudila revolucionarnu potporu za ručnim prilagodbama mapa i razina kao i stvaranjem. Sve to omogućeno je tipom datoteke pod nazivom WAD što je zapravo akronim za "Where's All the Data", odnosno "gdje su svi podatci". Datoteka takvog tipa sadrži sve potrebne informacije o pojedinoj razini, kao što su svi objekti na razini kao i definiranje svih sekcija mape [38].

Osim populariziranja žanra pucačina iz prvog lica, ova videoigra zajedno sa cijelom franšizom bila je kritična za cijelu industriju videoigara. Osim postavljanja temeljnih standarda za pucačine, bila je revolucionarna i po tehnološkim naprecima korištenim pri izradi ove videoigre. Korištenje raycasting tehnologije bilo je nevjerojatno otkriće u to vrijeme, kao i inovativna tehnologija mapiranja tekstura. Iako videoigra izgleda 3D, sve teksture u videoigri su zapravo 2D. Zbog tehnologije mapiranja tekstura bilo je moguće korisniku pružiti doživljaj u kojem se igrača uranja u DOOM svijet. Takav pristup imao je značajan utjecaj na videoigre koje su došle nakon te je postavio standarde vizualnog iskustva u industriji videoigara. DOOM videoigra bila je jedan od pionira koji su popularizirali videoigre na osobnim računalima (dalje u tekstu *PC*, engl. *Personal Computer*). Osim igre DOOM, veliki utjecaj na PC industriju videoigara imali su i sljedeće videoigre: već spomenuti Wolfenstein 3D (1992) te Quake (1996) koji je također razvio id Software, Half-Life (1998) razvijen od strane Valve Corporation te Blizzardovi StarCraft (1998) i World of Warcraft (2004).

Za DOOM bi se moglo reći da je postao kulturološki fenomen već i prije samog izlaska videoigre, no sa sigurnošću se može reći da je bio kulturološki fenomen već nekoliko sati nakon što je objavljen. Kao što je već spomenuto, id Software odlučio je preuzeti izdavanje videoigre u svoje ruke. Odlučili su se poći sa "shareware" pristupom u kojem se početno korištenje softvera daje besplatno te se softver često naplaćuje nakon određenog perioda. U slobodnijem pristupu sharewarea se korisniku daje sloboda da plati softver ako mu se sviđa. Izdavanje DOOM igre najavljeno je u ponoć 10. prosinca 1993. godine. Za objavljivanje videoigre koristila bi se mreža fakulteta Parkside u Wisconsinu koja je u to vrijeme imala mrežu visoke propusnosti te je mogla podnijeti 125 spojenih korisnika u isto vrijeme. Plan je bio da id Software prenese datoteke za videoigru te da igrači te datoteke preuzmu. No, prilikom izvedbe ovog plana došlo je do problema. Zbog najave ove videoigre te fenomena koji je nastao najavom, mreža fakulteta popunila se korisnicima koji su nestrpljivo čekali prijenos datoteka. Zbog nedostatka sredstva komunikacije na serveru kao improvizirano sredstvo komunikacije koristili su nazive datoteka. Na serveru su nastajale datoteke s imenima kao što su "KADA JE DOOM" (engl. *WHEN IS DOOM*) te "MI ČEKAMO" (engl. *WE ARE WAITING*). U ponoć je zbog velikog broja spojenih korisnika bilo nemoguće id Softwareu spojiti se na server te prenijeti datoteke. Ostvarivanje konekcije nije bilo moguće čak ni nakon povećanja ograničenja za mogući broj korisnika na serveru te je jedino rješenje bilo izbaciti sve korisnike s mreže kako bi se Jay Wilbur, koosnivač id Softwarea, mogao spojiti na mrežu. Potražnja za DOOM-om bila je za to vrijeme nevjerojatnih razmjera, desetak tisuća korisnika pokušalo je u isto vrijeme pristupiti mreži. Od silnog

broja korisnika mreža fakulteta, kao i računalo fakulteta u Wisconsinu, se srušila te je takav interes za videoigrom dotad bio neviđen. Interes za videoigrom DOOM uzrokovao je mnogim zabranama zbog opsjednutosti igrača za DOOM-om. Intel, fakultet Texas AM te sveučilište u Louisvilleu zabranili su DOOM na njihovim računalima te su mnogi čak i osmislili programe koji bi periodički brisali instance DOOM videoigre. Nedugo nakon objavljivanja, DOOM su nazivali "cyber opijatom". Već nakon jednog dana id Software ostvario je dobitke. Iako je procijenjeno da je samo 1 posto osoba koji su isprobali shareware verziju zapravo odlučio kupiti punu verziju, već u prvom danu ostvareno je 100,000 dolara u prodaji.

U ovom radu fokus se stavlja na originalnu DOOM videoigru iz 1993. godine. Originalna videoigra ove serije prvotno je podijeljena u tri epizode koje igrač može igrati. Naknadno je dodana još jedna epizoda pod nazivom "Thy Flesh Consumed". Početak svake razine izgleda dosta slično; igra započinje s igračem smještenim u određenu prostoriju, odnosno na određenu lokaciju, te je cilj da igrač dođe do obilježene sobe koja označava kraj razine. Igrač se nalazi u ulozi neimenovanog borca posade svemirskih marinaca u bazama na Marsu, njegovim mjesecima, na zemlji ili u paklu. Igraču se često kolokvijalno daje imena "Doomguy" te "Doomslayer". Bezimenost lika kojim igrač upravlja bila je svjesna odluka. Članovi id Softwarea smatrali su kako se takvim pristupom olakšava igraču da se poistovjeti s glavnim likom te da glavnog lika napravi svojim avatarom [38]. Vrijeme radnje ove igre teško je konkretizirati jer postoji nekoliko različitih vremenskih sljedova, odnosno crta, u svemiru odnosno svemirima DOOM franšize. Pretpostavlja se da je vrijeme radnje originalne videoigre 21. stoljeće, te je prema romanima moguće da se radnja dešava već 2004. godine [39].

2.6.1. Vrste čudovišta

Čudovišta u DOOM videoigri temeljni su dio igračevog iskustva, oni su neprijatelji s kojima se igrač susreće i bori. Pod kategorijom čudovišta spadaju sva bića u videoigri koja ne postoje ako se pri igranju koristi *nomonsters* parametar. Čudovišta u igri su ili postavljena u razinu od dizajnera razine ili kreirana od strane drugog čudovišta, najčešće *boss* čudovišta. Odnosno, glavnog čudovišta pojedine razine, ako takvo postoji. Svako čudovište je u početku razine u stanju neaktivnosti te se aktivira nakon određenog okidača, odnosno kada se zadovolji jedan od sljedećih uvjeta:

- Čudovište ugleda igrača
- Čudovište primi ozljede
- Čudovište čuje napad igrača (bilo koje oružje, pa čak i šake koje su po svojoj prirodi tihe)

Nakon što se čudovište aktivira, kreće u napad prema igraču. Čudovište igrača osjeti čak i kroz zidove te pokušava doći do igrača čak i ako se nalazi na suprotnom kraju razine. Ako čudovište dođe do prepreke, pokušava ju zaobići. Nakon što neprijatelj prijeđe u aktivirano stanje, u njemu u pravilu ostaje zauvijek, no postoje određene iznimke. Ako čudovište ne vidi/čuje igrača tada u aktiviranom stanju ostaje sve dok ne ubije svoju metu. Riječ "metu" je u ovom slučaju ključna. Moguće su situacije u kojima jedno čudovište slučajno ozlijedi drugo te

tako dolazi do zanimljivog događaja u kojem se čudovišta bore između samih sebe. Nakon što jedno čudovište ubije drugo, čudovište koje je preostalo vraća se u prvotno, neaktivno, stanje. Također u određenim verzijama igre postoje i greške u samoj igri pomoću kojih je moguće vratiti sva čudovišta u prvotno stanje nakon spremanja i učitavanja stanja igre. Čudovišta pokazuju izrazito nisku razinu inteligencije, no unatoč tome mogu koristiti neke od naprava namijenjene za igrače kao što su teleporteri, vrata te dizala [39].

Čudovišta se prema njihovim vizualnim karakteristikama dijele u dvije grupe, a to su bivši ljudi te čudovišta iz pakla. Kategorija bivših ljudi u prvoj igri dijeli se na dvije vrste, takozvani "Zombieman" te "Shotgun guy" koji se u priručnicima za igru još naziva i "Sergeant". Čudovišta iz pakla znatno su mnogobrojnija. To su "Imp", "Demon" koji se kolokvijalno još naziva i "Pinky", "Spectre", "Lost Soul", "Cacodemon", "Baron of Hell", "Spider Mastermind" [40]. Iz originalnog priručnika namjerno je izostavljen još jedan neprijatelj, a to je "Cyberdemon". Cyberdemon se pojavljuje kao boss neprijatelj u razini E2M8 te slovi kao najteži protivnik u originalnoj igri. Njegovo izostavljanje iz priručnika omogućilo je klimaktičko iznenađenje na toj razini.

2.6.1.1. Zombieman

Zombieman predstavlja najjednostavnijeg neprijatelja u igrici. Priručnik [40] navodi kako se radi o bivšim vojnicima koji su ili ubijeni te reanimirani, posjednuti ili pretvoreni u zombija tijekom invazije demona. Zombieman je prva vrsta neprijatelja s kojim se igrač susreće te se tako upoznaje s mehanikama borbe. Prepoznatljiv je po svojoj vojničkoj odori svijetlo-smeđe boje uprljanu krvlju te zelenoj kosi i crvenim očima. Zombiemana je relativno lagano ubiti s njegovih 20 zdravlja (u daljnjem tekstu HP, od engl. *Health Points*), a polu-automatskom puškom kojom je naoružan nanosi 3-15 štete po hitcu. Kreće se brzinom od 70 mu/s (map units / sekundi), odnosno 8 mu/sličici (engl. *frame*). Prilikom pucanja na igrača, zombieman prvo cilja 10 vremenskih jedinica (engl. *tic*), te nakon toga ispali jedan metak. Njegova puška ima široki radius raspršivanja (standardna devijacija oko 9° do maksimalno $\pm$22°) [40] te najčešće promaši, posebice na srednje-velikim ili velikim udaljenostima. Vrsta napada kojom se koristi u priručniku se naziva HITSCAN napadom. Hitscan označava da se u trenutku pucanja "skenira" (*scan*) putanja metka te se već u tom trenutku određuje radi li se o pogotku ili promašaju. To efektivno znači da se igrač (a niti čudovište) ne može izmaknuti napadu. Bitno je napomenuti da se u borbi "prsa u prsa" igračevi napadi šakama također smatraju hitscan napadom, dok su ugrizi i napadi šakama čudovišta programirani kao jednostavna provjera udaljenosti - ako se igrač nalazi u određenoj udaljenosti, napad pogađa. Prilikom smrti zombieman ostavlja spremnik za streljivo u kojem se nalazi 5 metaka, odnosno 10 metaka na težinama 1 i 4 [39].

2.6.1.2. Shotgun guy

Druga vrsta neprijatelja koji su prethodno bili ljudi je "shotgun guy", odnosno narednik. Narednik je veoma sličan zombiemanu, pošto se radi o zombificiranom vojniku koji je imao čin narednika. Doom priručnik [40] kao opis ovog neprijatelja navodi: "Isti kao Bivši Ljudi, no puno zlobniji i žilaviji. Ako niste oprezni, ove hodajuće sačmarice napravit će vam još nekoliko

rupa!". Izgledom je sličan zombiemanu, također nosi vojničku odoru, no u slučaju narednika radi se o odori sivo-crne boje. Također ima crvene oči te je ćelav. Narednik kao brzinu kretanja također ima 8 mu/sličici, no zbog neobičnog ponašanja u zombiemanovom kretanju, narednik se kreće brzinom od 93.3 mu/s. Shotgun guy raspolaže s 30 HP-a, a sačmarica kojom se koristi nešto je slabija verzija od igračeve sačmarice. Prilikom pucanja streljiva sačmarice ispaljuju se 3 projektila od kojeg svaki može napraviti štetu od 3-15. Odnosno, šteta koju narednik može napraviti ispaljivanjem jednog metka u rasponu je od 9 do 45. Napad narednika sličan je zombiemanu. Narednik prvo nišani igrača 10 tic-ova te pritom ispaljuje 3 puščana zrna od kojih svaki ima radijus raspršivanja standardne devijacije $9°$ do $\pm 22°$. Svaki od ispaljenih zrna također se koristi hitscan algoritmom. Nakon smrti kod narednikovog tijela ostaju 4 streljiva za sačmaricu.

2.6.1.3. Imp

Imp je vjerojatno prvi demon s kojim se igrač ima priliku susresti. Radi se o humanoidnim demonima ljudskih proporcija. Priručnik za igru [40] navodi kako imp nije stereotipični slatki mali vražićak u crvenom odijelu koje se može pretpostaviti za čudovište naziva "vražićak" (engl. *imp*). Već se radi o ružnom, smeđem čudovištu [40]. Osim toga, moguće ih je prepoznati po krvavo-crvenim očima i ustima te bijelim šiljcima koji vire iz njihovog tijela. Impovi su jedni od najučestalijih neprijatelja s kojima se igrač susreće te se smatraju osloncem stvorenja iz pakla po kojima je igra poznata. Kada primijete igrača ispuštaju glasni zmijoliki siktaj, a u neaktivnoj fazi proizvode frktajući zvuk nalik jaguaru. Impovi su posebice opasni zbog svojeg napada bacanja vatrene kugle, kao i zbog svoje izdržljivosti. Impovi imaju 60 HP-a što ih čini znatno izdržljivijim od drugih početnih neprijatelja. Za ubijanje impa potrebno je upucati ga otprilike 6 puta s pištoljem. Brzina kretnje impa jednaka je brzini kretnje narednika, odnosno 93.3 mu/s, a šteta koju nanose jednim udarcem u rasponu je od 3-24. Prilikom borbe na daljinu koriste se napadima vatrenim kuglama koje putuju u ravnini sve dok ne udare u objekt, zid ili metu. Pri borbi prsa u prsa koriste se kandžama kako bi napali metu [39].

2.6.1.4. Demon (Pinky)

Priručnik za igru [40] navodi kako su demoni nešto slično obrijanim gorilama, no imaju rogove, veliku glavu, mnoštvo zubiju te ih je teže ubiti [39]. Kolokvijalno se nazivaju "Pinky" zbog njihove ružičaste boje kože. Demoni su izrazito mišićavi stvorovi sa žutim svjetlećim očima te se kreću digitigradnim kretanjem, a na svakoj nozi imaju još i 3 kandže. Kada postanu svjesni igračevog prisustva proizvode glasnu riku poput lava. Zbog nedostatka napada na daljinu, u borbi im je cilj približiti se igraču te iskoristiti svoj jaki ugriz kako bi načinili štetu u rasponu od 4-40 zdravlja. Pritom se koriste svojom velikom brzinom od 175 mu/s, a tijekom trčanja izvode i manevre izbjegavanja trčanjem u cik-cak uzorku. No, iako demoni imaju znatno veću količinu zdravlja (150 HP) od drugih čudovišta spomenutih dosad, njihova veličina čini ih lakim metama za pogodak. Zbog svoje veličine kao i zbog kratkog dometa svojih ugriza nisu velika prijetnja igraču. U samom implementaciji ove igre postoji zanimljiva situacija gdje se demoni nazivaju kodnim imenom "MT_SERGEANT". Njihov naziv u izvornom kodu upućuje na to da su demoni

uvedeni u videoigru prije nego narednici te da im je izvorno ime bilo Demon Narednik (eng *Demon Sergeant*)

2.6.1.5. Spectre

Spectre je vrlo zanimljiva vrsta neprijatelja. Oni posjeduju mogućnost da budu djelomično nevidljivi. Priručnik [40] pod sekcijom Spectre navodi: "Odlično. Točno ono što ti je trebalo. Nevidljivo (skoro) čudovište." [40]. Time se upućuje na to da iako posjeduju moć nevidljivosti, ona ne radi besprijekorno. Nevidljivost funkcionira tako da savija svjetlost oko sebe te tako postaju nevidljivi. U mračnijim sobama ili u pozadinama određene teksture (npr. sivi zid) postaje znatno teže uočiti ovaj efekt. No, uočavanje ovih neprijatelja je puno lakše u svijetlim sobama ili sobama gdje su pozadine izrađene od raznolikih tekstura. Osim svoje nevidljivosti, ostale karakteristike ovih čudovišta identične su demonima [40].

2.6.1.6. Lost Soul

Svi neprijatelji spomenuti do sada su neprijatelji koji hodaju. Lost Soul prvi je leteći neprijatelj s kojim se igrač susreće. Igrač ih prvi put susreće u epizodi 2, na prvoj ili drugoj mapi, ovisno o težini na kojoj igrač igra. Lost Soul leteća je lubanja koja gori. Ima dva roga, oštre zube te narančaste oči sa žutim zjenicama. Za razliku od drugih čudovišta, ova vrsta ne proizvodi zvukove kada spazi igrača. Zbog toga, zajedno s njihovom mogućnosti letenja, mogu se vrlo lagano prišuljati igraču. Njihov napad svodi se na udarac glavom. U svom neaktivnom stanju kreću se brzinom od 46.7 mu/s. Kada opaze igrača napadaju ga velikom brzinom od 175 mu/s. Jednim napadom rade štetu u rasponu od 3 do 24, a oni sami imaju 100 HP-a.

2.6.1.7. Cacodemon

Cacodemoni su uobičajeni demoni koji se susreću u svakoj videoigri. Također su i maskota originalne DOOM igre. Njihov izgled inspiriran je stvorenjima iz popularne stolne igre "Dungeons and Dragons". Cacodemon je crveno okruglo čudovište s velikim tijelom, rogovima te jednim zelenim okom. Također imaju ogromna usta s mnoštvom zubiju i smiješkom koji podsjeća na češirsku mačku. Svojim izgledom podsjećaju na čudovište iz Dungeons and Dragons stolne igre zvano Beholder. Također je prepoznata sličnost između Cacodemona te čudovišta vrste "Astral Dreadnought" s korica jedne od ekspanzija Dungeons and Dragons igre pod nazivom "Manual of the Planes". Cacodemon izgleda kao odrezana verzija Astral Dreadnought-a. Borbene karakteristike cacodemona su poprilično jednostavne. Glavni napad kojim se koriste je ispaljivanje opasne kugle plazme iz njihovih usta. Kugla putuje veoma brzo brzinom od 350 mu/s, a jedan pogodak čini štetu u rasponu od 5-40. Osim razlike u snazi, kugla funkcionira na identičan način kao i napad vatrenom kuglom koji impovi koriste. Ako se dovoljno približe svojoj meti, koriste se svojim jakim ugrizom kojim nanose štetu od 10 do 60 HP-a. Njihova debela koža čini ih veoma izdržljivim neprijateljima s njihovih 400 HP-a. Brzina kojom se kreću je 93.3 mu/s, a njihovo lebdenje omogućava im da prelete preko prepreka koje bi hodajući neprijatelji morali zaobići.

2.6.1.8. Baron of Hell

Baron of Hell jedno je od najsnažnijih bića s kojima se igrač može susresti. Prvi susret s ovim čudovištem je u epizodi 1 mapa 8 gdje dva barona pod internim nazivom "Bruiser Brothers" služe kao boss borba cijele prve epizode [39]. Priručnik [40] navodi kako je baron of hell jak kao kamion kiper (samoistovarivač), te da je skoro toliko i visok. Priručnik [40] također komično izjavljuje kako se radi o najgorim bićima na dvije noge od tiranosaurusa rexa [40]. Izgledom uzima formu kozolikog demona crvenog torza i smeđih nogu s kopitima. Kada uoči igrača ispušta glasnu riku bika. Borbenim ponašanjem su veoma slični impovima. Kada se radi o borbi na udaljenost rukama bacaju zelene vatrene kugle, a pri borbi prsa u prsa metu napadaju kandžama. Jedan napad kuglom čini štetu od 8 do 64 HP-a, a jedan napad kandžama uzrokuje 10 do 80 HP-a štete. Kreću se brzinom od 93.3 mu/s, a njihovih 1000 zdravlja čini ih jednim od najizdržljivijih čudovišta u igrici.

2.6.1.9. Cyberdemon

Cyberdemon drugo je boss čudovište protiv kojeg se igrač bori. Zbog klimaktičkog iznenađenja ne postoji unos ovog čudovišta u priručniku. Igrač se prvi put susreće sa cyberdemonom u završnici druge epizode igre na mapi 8. Cyberdemona moguće je prepoznati po njegovom visokom stasu i izgledu koji podsjeća na minotaura. Naziva se "cyber" demonom jer je velikim dijelom i on kibernetički. Abdomen mu je zamijenjen velikim brojem crvenih žica, desna noga je također kibernetička. Na desnoj ruci moguće je vidjeti nekoliko žica i metalnih dijelova, a lijeva ruka u potpunosti je zamijenjena bacačem raketa. Njegov napad svodi se na korištenje kibernetičke ruke za ispaljivanje 3 rakete koje se kreću brzinom od 700 mu/s,a svaki napad radi štetu od 20-160 HP-a direktne štete te dodatnih 0-128 HP-a od eksplozije, što sveukupnu štetu svodi na 148-288 HP-a. Cyberdemon se kreće brzinom od 186.7 mu/s te ima nevjerojatnu količinu zdravlja od 4000 HP-a. Osim toga, cyberdemon je imun na svaki tip eksplozije osim eksplozije kod oružja BFG9000. Ubijanje cyberdemona izuzetno je težak zadatak. Najlakši način je 2-4 hitca s oružjem BFG9000. S plazma puškom potrebno je 150-200 hitaca kako bi ubili Cyberdemona, a bacačem raketa potrebno je oko 45 raketa [39].

2.6.1.10. Spider mastermind

Cyberdemon nije jedino tajno čudovište prema priručniku za igru. Zadnje boss čudovište originalne igre DOOM, kao i nadograđene verzije Ultimate DOOM koja ima dodatnu epizodu, također nije spomenuto u samom priručniku. Poznato i pod nazivom Spiderdemon iako inferiorniji po svim statistikama od cyberdemona, i dalje predstavlja veliku opasnost za igrača. Radi se o velikom mozgu i licu s crvenim očima koje trepću žutom bojom kada napada. Čudovište se koristi kibernetičkom šasijom od četiri noge koje izgledom podsjeća na pauka - stoga njegovo ime "Spider mastermind". Kao oružje koristi se lančanom puškom čije metke ispaljuje na prvi tren kada ugleda igrača. Njegova lančana puška funkcionira na sličan način kao i narednikova sačmarica. Svakim metkom ispaljuju se 3 kuglice koja se koriste hitscan algoritmom te svaka od kuglica radi 3-15 štete, odnosno 9-45 sveukupno po metku. No, za

razliku od narednika, ovo čudovište ispaljuje metke nevjerojatnom brzinom od 467 metaka po minuti. Nakon što spider mastermind počne pucati na svoju metu, s pucanjem ne prestaje sve dok meta nije uništena, sakrije se iza prepreke odnosno razbije liniju vidnog polja ili ako se čudovište ošamuti ili ubije. Spiderdemoni, kao i cyberdemoni, imuni su na eksplozivnu štetu od oružja te primaju samo štetu direktnog pogotka. No zbog svojih velikih proporcija vrlo je lagano pogoditi ga sa svim zrakama BFG9000 oružja, te su za ubijanje potrebna jedan ili dva direktna hitca iz bliza. Spiderdemon nešto je manje izdržljiv od cyberdemona sa svojih 3000 HP-a, te se kreće sporijom brzinom od 140 mu/s. No, iako se po svojim statistikama spiderdemon čini slabijim neprijateljem od cyberdemona, u praksi su dosta podjednako jaki. U slučajevima kada dođe do borbe između ova dva neprijatelja (npr. na mapi MAP20), oba neprijatelja imaju otprilike podjednaku šansu za pobjedom.

2.6.2. Vrste oružja

Postoji nekoliko vrsti oružja u DOOM-u, a svrha svake vrste je borba s čudovištima i ostalim neprijateljima. Oružje se nabavlja skupljanjem oružja sa zemlje, a u razinu može biti ili stavljeno po dizajnu (prilikom dizajniranja mape) ili može biti oružje ispušteno nakon što jedan od neprijatelja umre. Nakon što igrač jednom pokupi oružje, svako drugo skupljanje oružja samo povećava količinu streljiva. Za korištenje većine oružja potrebno je navedeno streljivo koje ima određenu rijetkost ovisno o jačini oružja. Stoga ima smisla da se za slabije neprijatelje koristi slabije oružje čije se streljivo češće nalazi. Igrač igru započinje samo s dva bazična oružja, a to su njegove šake te pištolj.

2.6.2.1. Šake

Šake su najbazičnije oružje u igrici. Pretežito se koriste samo kao zadnja opcija u slučajevima kada igrač nema streljiva ili ga želi štedjeti. Igrač na svojoj lijevoj ruci nosi takozvani "bokser", odnosno metalni okvir koji štiti prste te nanosi veću štetu. Udarac šakom otprilike je jednake snage kao i pucanj pištolja. Jedan udarac nanosi štetu u rasponu od 2-20, a s "Berserker" supermoći (engl. *powerup*) jačina šake povećava se deseterostruko te je tada u rasponu od 20-200. Brzina udaraca šake je otprilike 2 udarca po sekundi, točnije 123.5 udaraca po minuti. Udarac šakom smatra se hitscan tipom pogotka te se iz tog razloga nije moguće izmaknuti udarcu šake [39].

2.6.2.2. Motorna pila

Osim šaka, jedino drugo oružje za borbu prsa u prsa je motorna pila. Smatra se četiri puta boljom šakom zbog četverostruko brže brzine udaraca. Motorna pila radi jednaku štetu kao i šaka po udarcu (2-20 HP-a), a brzina udaraca je 525 udaraca po minuti. Velikom brzinom napadanja motorna pila nanosi otprilike 94.47 štete po sekundi. Zbog brzine napada može biti pogodno oružje kada igrač želi čuvati streljivo. Sa svojih 525 udaraca po minuti moguće je držati čudovište u potpunosti "ošamućeno", pogotovo kada se radi o neprijateljima s visokom šansom boli (engl. *pain chance*). Šansa boli predstavlja postotak da biće prijeđe u status

boli prilikom primanja udarca u kojem se ne može kretati niti uzvratiti udarac. Motorna pila se pronalazi u raznim tajnim sobama [39].

2.6.2.3. Pištolj

Pištolj se smatra bazičnim oružjem te je ekvivalent šakama na daljinu. Igrač započinje igru s pištoljem te 50 metaka koji se dijele između pištolja te lančane puške. Svaki pucanj iz pištolja ispaljuje jedan metak koji nanosi 5 do 15 štete. Iz pištolja je moguće metke pucati individualno (jedan pritisak tipke - jedan metak), no moguće je i držati tipku za pucanje pri čemu je prvi metak precizan, a sljedeći imaju radijus raspršivanja standardne devijacije oko 2° do $\pm 5.5°$. Pištolj se uobičajeno smatra poprilično beskorisnim oružjem, te se u praksi koristi samo kada igrač štedi streljivo za sačmaricu ili kada se bori s najslabijim neprijateljima kao što su zombieman ili imp [39].

2.6.2.4. Sačmarica

Jedno od najkorisnijih i najsvestranijih oružja dolazi u obliku sačmarice. Svaki pucanj iz sačmarice ispaljuje 7 zrnaca od kojih svaka nanosi štetu u rasponu od 5 do 15 HP-a, odnosno 35-105 štete ako svako zrno sačme pogodi metu. Zbog raspršivanja zrna pri pucanju, sačmarica je posebno pogodna za borbu u uskim prostorima te se može koristiti do udaljenosti srednjeg dometa. Pri pucanju na visoku udaljenost raspršenost zrnaca onemogućava nanošenje velike štete.

2.6.2.5. Lančana puška

Lančana puška (engl. *chaingun*), još poznata i pod nazivom Gatlingova strojnica je automatsko oružje vrlo visoke brzine paljbe. Nanosi jednaku štetu po metku kao i pištolj, no puca znatno većom brzinom od 525 metaka po minuti. Prilikom svakog pucnja ispaljuju se dva metka od kojih svaki nanosi štetu od 5-15 te se vodi hitscan algoritmom. Prva dva metka u svakoj seriji pucanja stopostotno su precizni, a daljnji metci imaju jednak radijus raspršenosti kao i kod pištolja [39].

2.6.2.6. Bacač raketa

Bacač raketa (engl. *Rocket launcher*) jedno je od najjačih oružja u igri. No, samim time je i municija za bacač raketa rijetka. Svaka od raketa nanosi od 20 do 160 direktne štete te dodatnih 0 do 128 štete eksplozije ovisno o blizini eksplozije. Smatra se najefektivnijim oružjem na veće udaljenosti zbog brzine projektila od 700 mu/s kao i zbog svoje brzine pucanja od 105 raketa po minuti. Zbog činjenice da se šteta od eksplozije primjenjuje i na samog igrača, korisnost bacača raketa u uskim prostorima kao i u borbi na malu udaljenost znatno opada [39].

2.6.2.7. Plazma puška

Plazma puška moderna je vrsta oružja koja ispaljuje bijelo plavu plazmu u obliku sferičnih projektila. Svaka od kugli čini štetu od 5 do 40 HP-a, a puška ima mogućnost ispaljivanja 700 kugli po minuti. Plazma puška otprilike je tri puta jača od lančane puške. Svojim rafalom može uništiti rulje neprijatelja, a ni jači neprijatelji nisu velika prijetnja kada je igrač naoružan plazma puškom. plazma puška smatra se veoma jakim oružjem te sveukupno veoma dobrim oružjem neovisno o situaciji u kojoj se igrač nalazi. Za ispaljivanje plazme koriste se ćelije punjene energijom kao streljivo. Ova puška dijeli streljivo sa oružjem BFG9000 [39].

2.6.2.8. BFG9000

BFG9000 slovi kao najmoćnije oružje u cijeloj videoigri. Inicijali ovog oružja iščitavaju nekoliko imena. Prema filmu iz 2005. godine puno ime ovog oružja glasi "Bio Force Gun", dok se u različitim materijalima još naziva i "Big Fancy Gun" te mnoštvom drugih imena [39]. Kao i u slučaju plazma puške, radi se o futurističkom oružju koje koristi energetske ćelije kako bi stvorilo plazmu. Ispaljuje kugle zelene plazme te pri svakom ispaljivanju koristi 40 ćelija energije. Velika kugla plazme koju ispaljuje kreće se brzinom od 875 mu/s, te čini štetu u rasponu od 100 do 800. Šteta je uvijek zaokružena na višekratnik od 100. Nakon kratke pauze od 16 tikova, otprilike 0.46 sekundi, ispaljuju se dodatnih 40 nevidljivih zraka tragača iz smjera igrača prema smjeru u kojem je kugla ispaljena u obliku stošca radijusa 45°. Svaka od zraka čini štetu u rasponu od 49-87 HP-a te ima veliki domet od 1024 mu. Minimalna šteta od ovog oružja je 0 ako igrač promaši metu kuglom kao i zrakama. Maksimalna šteta koju igrač može nanijeti u teoriji iznosi 800 direktne štete + 40x87 štete od zraka, to jest sveukupno 4280 HP-a štete. U praksi zbog pseudo-nasumičnosti takav broj neće nikada biti postignut, no ogromna količina štete koju BFG9000 može nanijeti čini ga najjačim oružjem u igri [39].

2.7. OpenAI Gym

OpenAI Gym složeno je aplikacijsko programsko sučelje (engl. *Application Programming Interface*) koje služi za standardiziranje okružja poticanog učenja. Također sadrži mnogobrojne scenarije koji olakšavaju proces učenja i implementacije algoritama poticanog učenja. Službena beta verzija objavljena je 27. travnja 2016. godine [41]. Osim ponuđenih alata za implementaciju poticanog učenja, kao i scenarija u kojima je moguće implementirati poticano učenje, OpenAI Gym sastoji se još i od web stranice na kojoj je moguće objaviti svoje rezultate u rješavanju nekog problema. Tako se kultivira društveni aspekt te se pruža jedno mjesto na kojemu je moguće usporediti različite pristupe poticanom učenju te njihove performanse. OpenAI Gym fokusira se na epizodičnost poticanog učenja, te se fokus stavlja na samo okružje u kojemu se agent nalazi. Sljedeći kod prikazuje primjer naveden u originalnom informacijskom dokumentu za Open AI Gym [41].

```
observation0 = env.reset()
action0 = agent.act(observation0)
observation1, reward1, done0, info1 = env.step(action0)
```

```
action1 = agent.act(observation1)
observation2, reward1, done1, info1 = env.step(action1)
...
action99 = agent.act(observation99)
observation100, reward99, done99, info99 = env.step(action99)
```

Navedeni kod započinje s pozivom "reset" metode na objektu okružja "env". Tako scenarij ponovno započinje te se vraća prvo promatranje (engl. *observation*), odnosno stanje okružja. Nakon što imamo stanje moguće je agentu to stanje proslijediti te agent odlučuje o prikladnoj akciji koju zatim vraća. Izvršavanje akcije u okružju provodi se uz pomoć "step" metode okružja kojoj se kao argument prosljeđuje sama akcija.

Ključne stavke u svakom procesu poticanog učenja su agent te okružje. OpenAI Gym nudi samo apstrakciju okružja, ali ne i agenta. To je bila svjesna odluka jer je cilj bio olakšati korištenje te omogućiti različite implementacije agenata. Pružanje apstrakcije agenta ograničavalo bi korisnike biblioteke na određene implementacije agenata. Prednosti biblioteke koja standardizira okružja su mnogobrojne. Jedna od tih prednosti je omogućavanje verzioniranja okružja. Centralizacija okružja u jednu biblioteku olakšava verzioniranje okružja tako da se sa svakom velikom promjenom može promijeniti naziv. Npr, ako se početna verzija zadatka "Cartpole-v0" značajno promijeni tako da se izmjene ključne funkcionalnosti, moguće je preimenovati okružje u "Cartpole-v1". OpenAI Gym također pruža i praćenje performansi agenta bez dodatnih izmjena. Na svakom okružju implementiran je takozvani *Monitor* koji prati svaki vremenski korak u procesu učenja kao i informacije o agentu i okružju u tom vremenskom koraku [41].

Prva beta verzija izašla je 2016. godine, a 25.11.2022. godine najavljeno je osnivanje nove neprofitne organizacije pod nazivom *The Farama Foundation*. Tim koji je uključen u novoosnovanu Farama organizaciju preuzeo je razvoj OpenAI Gym paketa još početkom 2021. godine, a najavom osnivanja organizacije prethodno nazvani OpenAI Gym paket mijenja naziv u Gymnasium. Gymnasium je kompatibilan sa starijim verzijama te su verzija 0.26.2 gymnasium-a i verzija 0.26.2 gym-a identične. Paket gym smatra se napuštenim te se daljnji razvoj odvija na gymnasium paketu [42].

2.8. ViZDoom

ViZDoom je prvo objavljeno okružje bazirano na DOOM-u čiji je cilj bio pružiti kompleksno 3D okružje za istraživanje poticanog učenja [43, str. 2]. ViZDoom je baziran na DOOM-u, odnosno na portiranoj (prilagođenoj za okolinu kojoj program nije originalno namijenjen) verziji pod nazivom ZDOOM. Prva verzija kreirana je rane 2016. godine. Motivacija za kreiranjem nečega takvoga bio je veliki nedostatak 3D okružja, a pogotovo onih iz prvog lica. Većina istraživanja odrađenih do tad baziralo se na 2D okružjima kao što su učenje igranja arkadnih igara kao što su Breakout te Pinball, ili učenje na kompleksnijim igrama na ploči kao što su Šah ili Go. No, agenti u 3D okružjima moraju biti još inteligentniji i kompleksniji. Kako bi agent bio kompetentan u 3D svijetu, mora riješiti nekoliko različitih problema te utilizirati različite vještine u isto vrijeme. Kako bi se uspješno riješilo okružje potrebno je istovremeno navigirati okružjem,

istraživati, znati prepoznati neprijatelja te biti dovoljno precizan kako bi ga pogodili. 3D okružja specifična su zbog perspektive u kojoj se agent nalazi. U 2D okružjima ne postoji vidno polje, u 3D okružjima vrlo je lako moguće da se iza agenta nalazi neprijatelj ili neki objekt koji mu može pomoći. Stoga je pamćenje stanja koje je izvan njegovog vidnog polja važan aspekt svakog agenta [43].

Kreiranje platforme ViZDoom izazvalo je veliki interes za istraživanjem poticanog učenja u 3D okružjima, ali i kreiranje novih 3D platformi baziranih na drugim igrama. Neke od drugih platformi su *DeepMind Lab* koji je baziran na igri *Quake III Arena* također od kompanije id Software [44] te *Project Malmo* baziran u svijetu igre *Minecraft* [45]. Osim kreiranja navedene platforme, također je kreirano i natjecanje pod nazivom *Visual Doom AI Competition (VDAIC)* u kojemu se agenti svih natjecatelja natječu u takozvanom *Deathmatch* događaju. Deathmatch je vrsta igre u kojoj se natjecatelji bore jedni protiv drugih te se nakon smrti ponovno stvaraju na mapi. Cilj ovog natjecanja bio je potaknuti interes za ovim poljem ali i pronaći agenta koji je najbolji u područjima navigacije okružjem, pucanja, izmicanja projektilima te ostalim stavkama koje se vrednuju [43].

ViZDoom je imao i određene zahtjeve koje su autori postavili. Kako bi ViZDoom bila uspješna platforma potrebno je da bude bazirano na igri otvorenog izvora (engl. *open-source*), nevjerojatne brzine generiranja uzoraka, prilagodljive rezolucije i drugih parametara prikaza, nudi mogućnost igre s više stvarnih igrača, podržava razne platforme kao što su Windows, Linux, MacOS. Uz navedene zahtjeve postojali su još mnogi drugi [43].

Natjecanje je bilo podijeljeno u dvije staze gdje se prva sastojala od deathmatch igre na poznatoj mapi. Mapa se sastojala od puno uskih hodnika i drugih prostora te je zbog kombinacije uskih prostora te bacača raketa kojim su svi agenti bili naoružani bilo zaista lagano usmrtiti protivnika. No, zbog eksplozije koju raketa kreira prilikom kontakta s površinom bilo je moguće i usmrtiti samog sebe. Druga staza sastojala se od 3 neviđene razine te su agenti prvotno naoružani pištoljima. Razine su sadržavale brojno oružje te su, za razliku od prve staze, razine bile otvorenog tipa s velikim udaljenostima na kojima je trebalo utilizirati dobru preciznost. Odabir neviđenih razina znatno je utjecao na performanse svih agenata pošto agenti nisu mogli prethodno naučiti razinu. Uz to je značajno na performanse utjecala i otvorenost razina gdje nije bilo dovoljno nasumično ispaliti raketu te pukom slučajnošću dobiti tzv. *frag*, odnosno uspješno ubijanje protivnika. Kao pobjednik staze 1 proglašen je bot "F1" koji je ostvario 559 ubojstava te je umro 413 puta. Drugo mjesto osvojio je bot "Arnold" sa svojih 413 ubojstava no s nevjerojatnih 217 smrti što je najmanje od svih 9 natjecatelja. Bot "F1" također je imao i najmanje samoubojstava, 38, dok je Arnold imao 119 samoubojstava. Stazu 2 osvojio je bot "IntelAct" sa svojih 256 ubojstava, te je drugo mjesto ponovno osvojio bot "Arnold" sa 164 ubojstava, no s nevjerojatnim brojem smrti te je imao omjer ubojstava/smrti 32.8, dok je bot "IntelAct" ostvario omjer ubojstava i smrti od 3.08 [43]

Agent "F1", pobjednik staze 1, implementiran je pomoću varijante A3C algoritma uz mehanizam kurikulumskog učenja. Kurikulumsko učenje označava treniranje u lakšem okruženju sa slabijim i sporijim neprijateljima, jednostavnijim kartama te progresivnim otežavanjem okružja nakon što se agent prilagodi te svlada lakše okružje [46]. Agent "IntelAct", najbolji agent

u stazi 2, koristio je implementaciju algoritma *Direct Future Prediction (DFP)*. DFP se razlikuje od algoritama poticanog učenja te se naginje više algoritmima nadziranog učenja tako da se umjesto monolitskog stanja i skalarne nagrade promatra tok senzorskih unosa (engl. *stream of sensory input*) s_t te tok mjera (engl. *stream of measurements*) m_t. Tok senzorskih unosa sastoji se od informacija o vizualnim, auditornim te taktilnim ulazima koji se dobivaju od senzora, a tok mjera sastoji se od informacija kao što su količina HP-a, količina streljiva te broju frag-ova [47]. Agent "Arnold" ostvario je dobre rezultate veoma zanimljivim pristupom. Implementiran je uz pomoć dva modula, jedan namijenjen za navigaciju implementiran kao *Deep Q Learning (DQN)* algoritam, te drugi namijenjen za borbu s neprijateljima implementiran koristeći *Deep Recurrent Q Learning (DRQN)* algoritam. DRQN neuralna mreža smatra se glavnom te ona ima dodatnu provjeru nalazi li se neprijatelj na ekranu, te ako neprijatelj ne postoji onda se odlučivanje o akciji predaje mreži za navigaciju. Navigacijska mreže nagrađivala se za uspješno kretanje mjereno brzinom, a penalizirala se za hodanje po lavi ili drugim zamkama na podu. Mreža za borbu nagrađivala se za uspješna ubojstva te skupljanje objekata, a penalizirala se za gubitak zdravlja ili streljiva. Postoji jedna činjenica koja Arnolda čini veoma zanimljivim te mu je u ovom natjecanju dala ogromnu prednost. Agenti u ovom scenariju nisu imali mogućnost vertikalnog ciljanja te je Arnold pronašao način kako to iskoristiti u svoju korist. Prilikom promatranja Arnolda u borbi moguće je primijetiti da se cijelo vrijeme nalazi u položaju čučnja te je upravo zbog toga u svakom scenariju imao izuzetno visok omjer ubojstava i smrti. Zbog konstantnog čučnja koje je bilo ugrađeno u kod, Arnold je bio u velikoj prednosti jer je većina projektila prelazila preko njega [43].

ViZDoom pruža mogućnost igranja u nekoliko modova. Prva podjela odnosi se na sinkronost igre. U asinkronom modu igra teče u konstantnih 35 sličica po sekundi te Ako je agent prespor, takozvani frame mu može promaknuti te ima priliku reagirati na sljedeći. Na drugu ruku, ako je agent prebrz tada je blokiran sve do idućeg framea. U sinkronom modu igra čeka na odluku agenta te agent u tom slučaju nije vremenski ograničen, odnosno može odlučivati o odluci svojim tempom. ViZDoom također pruža i podršku za igru s jednim igračem, kao i mod s više igrača (do maksimalno 16). Igre s više igrača uključuju deathmatch, timski deathmatch te kooperativne scenarije. Također se podržavaju i scenariji izrađeni od strane korisnika uz pomoć alata Doom Builder 2 te SLADE3 gdje se može direktno utjecati na mehanike okružja. Također se pruža mogućnost za mnogim posebnim slojevima prikaza kao što su automatsko označavanje objekata, prikaz dubine u prostoru te pogled razine odozgo. Uz navedene mogućnosti pružaju se i još mnoge druge funkcionalnosti vezane za DOOM igru [43].

Originalna dokumentacija pruža sljedeći uzorak koda kojim se pokreće agent s nasumično odabranim akcijama u preddefiniranoj konfiguraciji pod nazivom "custom_config.cfg". Akcije koje se nudi u konfiguraciji su okret u lijevo, okret u desno te pucanje. Kod započinje kreiranjem nove igre, učitavanjem scenarija te se nasumičnim odabirom izvršavaju akcije sve dok epizoda ne završi [43].

Listing 2.4: Kod preuzet iz [43, str. 4]

```
import vizdoom as vzd
from random import choice
game = vzd.DoomGame()
```

```
game.load_config("custom_config.cfg")
game.add_game_args("+name RandomBot")
game.init()
actions = [[1, 0, 0], [0, 1, 0], [0, 0, 1]]
while not game.is_episode_finished():
    if game.is_player_dead():
        game.respawn_player()
    state = game.get_state()
    screen = state.screen_buffer
    health, ammo = state.game_variables
    game.make_action(choice(actions))
```

2.9. Implementacija agenta

2.9.1. Instaliranje okružja

Za kreiranje agenta potrebni su nam mnogi paketi i biblioteke. Neki od najkorisnijih paketa koje koristimo su već spomenuti ViZDoom, cv2, gym, stable_baselines, matplotlib te još mnogi drugi.

2.9.1.1. Conda

Prva stavka koja nam je potrebna je softver za upravljanje paketima o kojima naš agent ovisi. Pogodan program za ovu svrhu je *Conda*. Conda je sustav za upravljanje paketima za mnogobrojne programske jezike. Conda nudi podršku za Python, R, Ruby, Lua, Scala, Java, Javascript, C/C++, Fortran te još mnoge druge programske jezike [48]. Osim upravljanja paketima, također nudi opcije upravljanja okružjima paketa. Okružje paketa podrazumijeva instancu okružja u kojem se može pristupiti paketima u određenoj verziji. Tako je moguće na istom računalu imati različite verzije istih paketa. Velika prednost Conde je ta da je platformski agnostik, odnosno moguće ju je pokretati na raznim operacijskim sustavima kao što su Windows, MacOS te Linux. Condu je moguće instalirati putem službenog instalera dostupnog na njihovim stranicama te je proces instalacije poprilično jednostavan. Nakon uspješne instalacije moguće je koristiti *conda* komandu za kreiranje okružja i instalaciju paketa. Za izvršavanje okružja prikladna je 3.7 verzija Pythona. Sve novije verzije imaju velikih problema u pokretanju ViZDooma ili nekih od drugih paketa koji su potrebni. Conda sadrži bazično okružje koje sva dodatna okružja nasljeđuju prilikom kreiranja. Svi paketi instalirani u bazičnom okružju bit će instalirani i u dodatnim okružjima. U dodatnom okružju instaliraju se paketi potrebni za pokretanje jupyter notebook datoteka. Jupyter Notebook pruža mogućnost kreiranja dokumenta u kojemu je moguće isječke koda podijeliti u ponovno izvršive sekcije. Tako se postojeći kôd segregira u odvojene isječke koje je moguće zasebno pokrenuti.

Za aktiviranje bazičnog okružja te instalaciju potrebnih paketa pokrenuti sljedeće komande [49]:

```
conda activate
conda install -c conda-forge notebook
```

```
conda install -c conda-forge nb_conda_kernels
conda install -c conda-forge jupyter_contrib_nbextensions
conda install pip
```

Navedenim komandama aktiviramo bazično okružje, instaliraju se svi paketi vezani za jupyter notebook. To su paketi notebook, nb_conda_kernels, jupyter_contrib_nbextensions te na posljetku pip. Paket notebook omogućava pokretanje jupyter notebook datoteka. Paket nb_conda_kernels je ekstenzija za Jupyter Notebook paket koja omogućava conda okružju pristup instancama takozvanih "kernela" za Python, R, ili drugi programski jezik koji pokreće kod u tom programskom jeziku. Posljednji paket za instalaciju je jupyter_contrib_nbextensions koji uključuje nekoliko ekstenzija za Jupyter Notebook kreiranih od strane Jupyter zajednice. Svaka od ekstenzija nudi dodatne funkcionalnosti koje poboljšavaju iskustvo rada u Notebooku. To su funkcionalnosti kao što su zatvaranje i otvaranje sekcije koda što poboljšava čitljivost, generiranje tablice sadržaja, mogućnost provjere gramatike teksta te još mnoge druge funkcionalnosti.

Zatim je potrebno kreirati conda okružje u kojem se pokreće instanca ViZDoom-a. Kao što je navedeno, eksperimentiranjem je pronađeno da je za najstabilnije pokretanje pogodna 3.7 verzija Pythona. Za kreiranje i pokretanje okružja koriste se sljedeće dvije komande:

```
conda create --name py37 python=3.7
conda activate py37
```

Vrlo je bitno pronaći pravu verziju pip-a (onu instaliranu u conda py37 okružju)

Putanja do tog pipa trebala bi izgledati nešto slično ovome: "/anaconda/envs/py37/bin/pip". Za instalaciju pomoću pipa koristi se komanda sljedećeg oblika:

```
/anaconda/envs/py37/bin/pip install [ime_paketa]
```

Konkretnije, instalacija se pokreće uz sljedeće dvije komande:

```
pip install --upgrade setuptools
pip install dlib==19.17.0
```

2.9.1.2. ViZDoom

Već spomenuti paket ViZDoom ključan je za implementaciju agenta. To je paket pomoću kojeg možemo kreirati instance Doom igre te nam nudi API za upravljanje igrom. Za instalaciju ViZDoom-a potrebno je instalirati pakete koji služe kao preduvjet za uspješnu instalaciju ViZDoom-a. Ti paketi su cmake, git, boost, openal-soft te sdl2. Instaliramo ih sa sljedećom komandom:

```
conda install -c conda-forge boost cmake sdl2 openal-soft
```

Postoji nekoliko načina instalacije ViZDoom paketa. U slučaju da se koristi Conda za upravljanje okružjima, ViZDoom je potrebno ručno preuzeti te ručno instalirati. Također postoji mogućnost instalacije pomoću pip-a, no takav pristup je u direktnoj koliziji s Conda okružjima te se ne preporučuje u originalnoj dokumentaciji. Nakon instalacije preduvjetnih paketa, potrebno je preuzeti zadnju verziju ViZDoom-a s njihovog GitHub repozitorija. Zatim se navigira

u ViZDoom folder te se pokreću skripte za build te install. Za uspješnu instalaciju bitno je da je verzija cmakea jednaka ili veća "3.1.0", te da su instalirani i paketi "setuptools" te "wheel".

```
git clone https://github.com/mwydmuch/ViZDoom.git --recurse-submodules
cd ViZDoom
python setup.py build && python setup.py install
```

U slučaju da je proces instalacije uspješno prošao, ViZDoom je instaliran te ga je moguće koristiti u projektu sljedećom komandom za uvoz biblioteke:

```
from vizdoom import DoomGame
```

2.9.1.3. Gym

OpenAI Gym je paket koji nudi API za upravljanje okružjem u kojem budući agent postoji i djeluje. Za instalaciju ovog paketa pokrećemo sljedeću komandu:

```
conda install -c conda-forge gym
```

Iz gym biblioteke potrebna je klasa pod nazivom *Env* koja služi za kreiranje takozvane Wrapper klase. Wrapper klasom moguće je definirati svoje okružje "po mjeri", a proširivanje Env klase omogućava zadržavanje njenih funkcionalnosti. Konkretne funkcionalnosti koje ova klasa nudi su atributi *action_space*, *observation_space* te *reward_range*. Također se definiraju i sljedeće metode:

- step
- reset
- render
- close
- seed

Osim Env klase, gym sadrži još mnoge klase i tipove objekata. Klasom *Discrete* definira se konačni broj elemenata, odnosno cijelih brojeva. Točnije, klasa Discrete opisuje prostor sljedećeg oblika: $a, a+1, ..., a+n-1$. Uz pomoć klase *Box* moguće je definirati multidimenzionalni prostor u $\mathcal{R}^n$. Box se definira kao Kartezijski produkt n zatvorenih intervala. Svaki interval ima jedan od sljedećih oblika: $[a, b]$, $(-\infty, b]$, $[a, \infty)$, ili $(-\infty, \infty)$. Navedene klase iz gym paketa u kod se uvoze na sljedeći način:

```
from gym import Env
from gym.spaces import Discrete, Box
```

2.9.1.4. Stable Baselines3

Stable Baselines3 zadnja je verzija popularne biblioteke za Python pod nazivom Stable Baselines. Radi se o implementacijama popularnih algoritama za poticano učenje u okviru

(engl. *framework*) za poticano učenje, PyTorch. Algoritmi koji se nalaze u paketu Stable Baselines poboljšane su verzije algoritama baziranih na implementacijama iz paketa *Baselines* koji je razvio OpenAI. Osim Baselines, postoji još mnogo implementacija biblioteka za poticano učenje kao što su Tensorforce, KerasRL, SpinningUp te još mnogi drugi. Baselines paket bazira se na Tensorflow implementaciji. Od Stable Baselines3 verzije, implementacija je prebačena sa Tensorflowa na PyTorch. PyTorch je često fleksibilniji za rad te ima sintaksu koja je više nalik Pythonu. Operacije u PyTorchu implementirane su pomoću tenzora (engl. *tensor*) čija je implementacija veoma slična operacijama u biblioteci NumPy. Time se olakšava proces prilagodbe osobama koje su upoznate sa sintaksom Pythona te NumPy-a. Stable Baselines uveo je značajna poboljšanja u Baselines implementaciju. Sve implementacije prošle su kroz proces refaktoriranja koda (engl. *Code Refactoring*) prilikom kojeg se kod "čisti" bez mijenjanja funkcionalnosti. Navedeni kod je pisan prema PEP8 stilističkim smjernicama koje čine službeni standard pisanja Python koda. Također je implementirana unificirana struktura za sve algoritme. Time se implementacija svih algoritama bazira na istim nazivima metoda te se olakšava postavljanje eksperimenata u različitim algoritmima kao i prenošenje znanja s jednog algoritma na drugi. Stable Baselines3 ima implementiranu integraciju s Tensorboard paketom. Tensorboard je alat za vizualizaciju koji omogućava praćenje procesa učenja te generiranje grafova iz prikupljenih podataka. Alat je iznimno koristan za bolju vizualizaciju procesa učenja te pravovremeno uočavanje potencijalnih problema. Za korištenje Tensoboard praćenja podataka potrebno je prilikom kreiranja modela definirati vrijednost parametra pod nazivom "tensorboard_log" [50].

Kao preduvjet za instalaciju Stable Baselines3 paketa potrebna je instalacija paketa PyTorch. Oba paketa moguće je instalirati pomoću pip instalatora, ali i uz pomoć conda upravljača paketima. Za instalaciju oba paketa izvršavaju se sljedeće dvije komande:

```
conda install -c pytorch pytorch
conda install -c conda-forge stable-baselines3
```

Stable Baselines3 omogućava jednostavno korištenje poznatih algoritama za poticano učenje. Za implementaciju učenja na okružju registriranog u Gym okružjima, moguće je koristiti sljedeći kod u jednoj liniji naveden u službenoj dokumentaciji [50]:

```
from stable_baselines3 import A2C
model = A2C("MlpPolicy", "CartPole-v1").learn(10000)
```

2.9.2. Python programski kôd

2.9.2.1. EnvironmentConfigurations.py

Zbog bolje čitljivosti koda kao i zbog bolje fleksibilnosti implementiran je mehanizam konfiguracija. Konfiguracije se pohranjuju u zasebnu python datoteku pod nazivom *EnvironmentConfigurations*. Datoteka sadrži određene konstante kao što su broj epizoda u slučaju testiranja, prefiks putanja u kojima će se pohranjivati modeli agenta kao i putanja do tensorboard zapisa. Osim toga sadrži i index trenutne konfiguracije kao i različite konfiguracije kojima se predstavljaju različiti scenariji.

```
EPISODES_NUM = 10
AGENT_MODEL_PATH_PREFIX = './agents/agent_for_'
TENSORBOARD_LOG_PATH_PREFIX = './logs/logs_for_'
CURRENT_CONFIGURATION_INDEX = 0
EVALUATION_FREQUENCY = 25000
MODEL_SAVING_FREQUENCY = 25000
EXPECTED_IMAGE_SHAPE = (240, 320, 3)

configurations = [{
        'name': 'basic',
        'scenarioConfigFilePath': 'VizDoom/scenarios/basic.cfg',
        'actionNumber': 3,
    }, {
        'name': 'defend_the_center',
        'scenarioConfigFilePath': 'VizDoom/scenarios/defend_the_center.cfg',
        'actionNumber': 3,
    }, {
        'name': 'deadly_corridor',
        'scenarioConfigFilePath': 'VizDoom/scenarios/deadly_corridor.cfg',
        'actionNumber': 7,
    }, {
        'name': 'deathmatch',
        'scenarioConfigFilePath': 'VizDoom/scenarios/deathmatch.cfg',
        'actionNumber': 4,
    }, {
        'name': 'defend_the_center_expanded',
        'scenarioConfigFilePath': 'VizDoom/scenarios/defend_the_center_expanded.
            cfg',
        'actionNumber': 4,
    }]
```

Konfiguracije se zapisuju kao polje objekata, a svaki objekt u polju prati identičnu strukturu. Konfiguracija se sastoji od naziva koji se koristi u putanjama kod pohrane agenta te tensoboard zapisa. Tako se datoteke segregiraju prema konfiguraciji u kojoj su izvršene. U svakoj konfiguraciji definirani su i putanja do ViZDoom konfiguracijske datoteke. Svaka konfiguracijska datoteka sastoji se od raznih postavki kao što su naziv mape, postavki renderinga kao što su rezolucija slike, prikaz raznih efekata u igri, bazičnim nagradama scenarija. Također se definira maksimalno vrijeme nakon kojeg se epizoda zaustavlja, kao i dozvoljene akcije agenta i varijable dostupne putem *get_state()* metode. U objektu konfiguracije definira se i broj akcija koje agent može izvršavati u toj konfiguraciji. Ovo informaciju je potrebno ponoviti jer se u datoteci EnvironmentConfigurations.py ne može pristupiti ViZDoom konfiguracijskoj datoteci. Definirana su 5 scenarija, bazični scenarij, obrana središta (engl. *Defend the center*), smrtonosni hodnik (engl. *Deadly corridor*), borba na smrt (engl. *Deathmatch*) te dodatan scenarij u kojem je obrana središta proširena s mogućnosti kretanja.

2.9.2.2. RewardShapingFactors.py

Datoteka RewardShapingFactors sastoji se od varijabli potrebnih za implementaciju mehanizma oblikovanja nagrada. U ovoj datoteci sadržani su faktori za nagrađivanje i penaliziranje

agenta ovisno o stanjima varijabli prije i nakon izvođenja akcije. Datoteka također sadrži i stanja VizDoom varijabli za količinu streljiva kao i stanja oružja. Iz tog razloga je potrebno iz VizDoom biblioteke uvesti varijablu GameVariable.

```
from vizdoom.vizdoom import GameVariable

class RewardShaping:
    DAMAGE_REWARD_FACTOR = 0.01
    DISTANCE_REWARD = 5e-4
    DISTANCE_PENALTY = -2.5e-3
    DISTANCE_REWARD_THRESHOLD = 3.0
    AMMO_REWARD_FACTOR = 0.02
    AMMO_PENALTY_FACTOR = -0.01
    HEALTH_REWARD_FACTOR = 0.02
    HEALTH_PENALTY_FACTOR = -0.01
    ARMOR_REWARD_FACTOR = 0.01

REWARD_SHAPING = RewardShaping()

AMMO_VARIABLES = [GameVariable.AMMO0, GameVariable.AMMO1,
                  GameVariable.AMMO2, GameVariable.AMMO3,
                  GameVariable.AMMO4, GameVariable.AMMO5,
                  GameVariable.AMMO6, GameVariable.AMMO7,
                  GameVariable.AMMO8, GameVariable.AMMO9]

WEAPON_VARIABLES = [GameVariable.WEAPON0, GameVariable.WEAPON1,
                    GameVariable.WEAPON2, GameVariable.WEAPON3,
                    GameVariable.WEAPON4, GameVariable.WEAPON5,
                    GameVariable.WEAPON6, GameVariable.WEAPON7,
                    GameVariable.WEAPON8, GameVariable.WEAPON9]
```

2.9.2.3. VizdoomGymWrapper.py

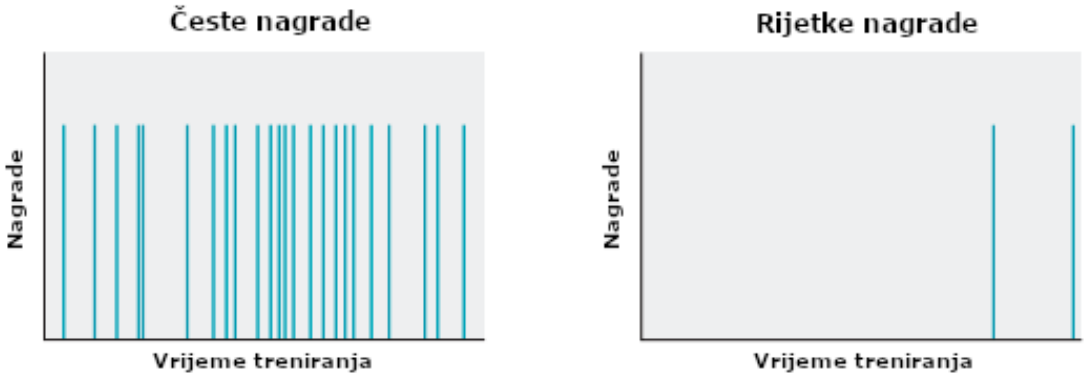
Prilikom implementacije procesa poticanog učenja na nekom stranom okružju, često je dobra odluka koristiti se već postojećom strukturom koja može olakšati implementaciju ali je i standardizirati. Korištenjem OpenAI Gym sučelja moguće je koristiti se metodama unutar klase Gym pomoću takozvane Wrapper klase. Wrapper klasa u tom slučaju služi kao poveznica između metoda Gym klase te okružja u kojem želimo trenirati agenta. Takvo ponašanje u Objektno Orijentiranom Programskom jeziku implementira se nasljeđivanjem. U tom slučaju klasa VizDoomGym (wrapper klasa) ima sve metode i ponašanja koja su očekivana od Env klase. To su na primjer metode close, step, render, reset, itd. VizDoomGym smatra se podklasom klase Env, te klasa za navedene metode nudi svoju implementaciju ili kroz nadogradnju funkcionalnosti ili kroz potpuno nadjačavanje metoda.

Datoteka VizdoomGymWrapper.py započinje s raznim uvozima paketa te uvozom faktora nagrade i penalizacije za funkcionalnost oblikovanja nagrade. Također se uvoze i konfiguracijske varijable vezane za okružje.

```
from vizdoom import DoomGame
import numpy as np
```

```
from gym import Env
from gym.spaces import Discrete, Box
import cv2
from vizdoom.vizdoom import GameVariable
import EnvironmentConfigurations as EnvConfig
from RewardShapingFactors import REWARD_SHAPING, AMMO_VARIABLES,
    WEAPON_VARIABLES
```

Svi scenariji u kojima se agent može pronaći već sadrže pojedine nagrade. U bazičnom scenariju agenta se nagrađuje za ubijanje cacodemona, dok se u smrtonosnom hodniku igrača nagrađuje za ubijanje neprijatelja, ali i inverzno proporcionalno udaljenosti krajnjeg cilja što je u tom slučaju oklopni prsluk. U deathmatch scenariju jedina nagrada koju igrač dobiva je nagrada za ubijanje protivnika. U tom scenariju, ali i donekle u prethodnim scenarijima dolazi se do velikog problema poticanog učenja. Taj problem naziva se problemom rijetke nagrade (engl. *Sparse Reward problem*) [51].



Slika 6: Usporedba čestih i rijetkih nagrada, izvor: [52]

Dijagram 6 prikazuje učestalost nagrada u sustavu gdje su nagrade česte (lijevo) te u sustavu gdje su nagrade rijetke (desno). U jednakoj implementaciji agenta može se očekivati da će učenje agenta, kao i rezultati samog učenja, biti bolji u sustavu s čestim nagradama.

Problem rijetke nagrade pojavljuje se u slučajevima kada je za nagradu potrebno napraviti nekoliko uzastopnih akcija točnim redoslijedom kako bi se dobila nagrada. Bilo koji krivi potez dovodi do ne primanja nagrade. Taj problem rijetko se pojavljuje u ljudskom ponašanju, pošto se ljudi često vode internom motivacijom te odrađuju aktivnosti za koje ne postoji nagrada. No, agentu rijetke nagrade predstavljaju iznimno velik problem. Nagrade služe kao mehanizam kojim se određene akcije potiču kako bi ih agent ponovno izabrao u istim ili sličnim stanjima. Ako agent ne dobije nagradu, akcije koje su na pravom putu nikada nisu nagrađene te se samim time ne ohrabruju. Iz tog razloga se nagrade u okružju moraju veoma pažljivo implementirati kako bi se postiglo optimalno ponašanje [51]. Postoji nekoliko rješenja koji se ističu pri rješavanju ovog problema. U jednom pristupu problem se rješava oblikovanjem nagrada. Oblikovanje nagrada podrazumijeva prilagodbu originalne funkcije nagrade s dodatnom oblikovanom funkcijom nagrade. Oblikovana funkcija nagrade implementira ljudsko, domensko, znanje te se tako agenta može potaknuti dodatnim stavkama koje nisu uključene u originalnu nagradu. Takva modificirana nagrada matematički se izražava kao $r' = r + F$, pri čemu je

r originalna nagrada, F je funkcija oblikovane nagrade, a r' predstavlja oblikovanu nagradu [53]. Oblikovanje nagrada u slučaju Doom agenta funkcionira tako da se nagradi za ubijanje pridodaje dodatna nagrada za ozljeđivanje neprijatelja, nagrada ili penali za kretanje, nagrada ili penali za streljivo te još mnoga druga oblikovanja. Prilikom svake akcije agentu se izračunava delta vrijednost za svaku bitnu statistiku te ga se nagrađuje ili penalizira ovisno o razlici u prijašnjem te sadašnjem stanju.

Drugi pristup kojim je moguće riješiti ili barem mitigirati problem je takozvano kurikulumsko učenje [46]. Mehanizam kurikulumskog učenja već je spomenut pri analizi agenta "F1" iz ViZDoom natjecanja. Kurikulumsko učenje bazira se na strukturi učenja kojim se često ljudi koriste u procesu učenja. Bazična ideja u ljudi svodi se na postepeno učenje koncepata koji se nadograđuju težim konceptima pri čemu se već naučene stvari ponovno iskorištavaju te poboljšavaju za skupljanje kompleksnijih znanja. Takav princip primjenjiv je i na umjetnu inteligenciju. Učenje započinje s izrazito ograničenim setom informacija, kao u primjeru agenta koji uči gramatiku s vrlo jednostavnim početnim pravilima, te se naknadno nadograđuje kroz učenje. Bengio et al. [46] navode kako, prema njihovim eksperimentalnim rezultatima, kurikulumsko učenje može biti korisno u sferi strojnog učenja. Postoje strategije koje su pogodne za određene zadatke, te strategije koje su apsolutno beskorisne ili čak kontraproduktivne za neke zadatke. Također kao prednost predstavljaju brže učenje te je brža konvergencija dokazana kroz eksperimente [46]. U kontekstu Doom agenta, kurikulumsko učenje moguće je implementirati pomoću komandi kojima je moguće direktno utjecati na neprijatelje u igri. Na početku procesa učenja neprijateljima se postavlja brzina od na primjer 25% te im se zdravlje postavlja na 25% od trenutne vrijednosti. To znači da agent umjesto da mora neprijatelja pogoditi 4 puta kako bi ga usmrtio i dobio nagradu, neprijatelja mora pogoditi samo jednom. Time se značajno mitigira problem rijetke nagrade jer se smanjuje broj akcija koje se moraju izvršiti točnim redoslijedom kako bi agent došao do nagrade. Nakon što agent svlada okružje u kojima su neprijatelji na 25% snage, moguće je kroz isto sučelje modificirati neprijatelje kako bi se nalazili na npr. 35% snage. Taj proces se nastavlja sve dok agent ne nauči pobjeđivati neprijatelje koji su u stopostotnoj snazi. U istoj datoteci dalje se nalazi definicija navedene klase te njen konstruktor u kojoj se inicijaliziraju početne vrijednosti.

```
class VizDoomGym(Env):
  def __init__(self, env_config, is_reward_shaping_on=False, is_game_window_visible=
      False):
    super().__init__()
    self.game = DoomGame()
    self.game.load_config(env_config["scenarioConfigFilePath"])
    self.game.set_window_visible(is_game_window_visible)
    self.game.set_automap_buffer_enabled(True)
    self.game.init()

    self.is_reward_shaping_on = is_reward_shaping_on
    self.action_number = env_config["actionNumber"]
    self.action_space = Discrete(self.action_number)
    self.actions = np.identity(self.action_number, dtype=np.uint8)

    height, width, channels = self.game.get_screen_height(), self.game.
        get_screen_width(), self.game.get_screen_channels()
```

```
    new_height, new_width, new_channels = self.process_frame(np.zeros((height, width
        , channels))).shape
    self.observation_space = Box(low=0, high=255, shape=(new_height, new_width,
        new_channels), dtype=np.uint8)

    self.health = 100
    self.x, self.y = self.get_player_position()
    self.weapon_state = self.get_weapon_state()
    self.ammo_state = self.get_ammo_state()
    self.total_reward = self.damage_dealt = self.deaths = self.frags = self.armor =
        0

  def get_player_position(self):
    return self.game.get_game_variable(GameVariable.POSITION_X), self.game.
        get_game_variable(
        GameVariable.POSITION_Y)

  def get_weapon_state(self):
    return np.array([self.game.get_game_variable(WEAPON_VARIABLES[i]) for i in range
        (len(WEAPON_VARIABLES))])

  def get_ammo_state(self):
    return np.array([self.game.get_game_variable(AMMO_VARIABLES[i]) for i in range(
        len(AMMO_VARIABLES))])
```

Okružju se prilikom inicijalizacije prosljeđuju parametri bitni za uspostavljanje okružja. Jedan od tih parametara je *env_config* - jedan od objekata iz niza u konfiguracija implementiranim u EnvironmentConfigurations.py datoteci. Postoji i parametar *is_reward_shaping_on* koji služi za uključivanje te isključivanje oblikovanja nagrada pri treniranju. Posljednji parametar u ovoj metodi je *is_game_window_visible* koji također može biti True ili False te se ovisno o proslijeđenoj vrijednosti prikazuje prozor igre ili ne. Tako je moguće uštedjeti resurse računala pri procesu treniranja jer u tom trenu nije bitan vizualni prikaz agentovog odlučivanja u smislu promatranja samog agenta. U samom konstruktoru dolazi do kreiranja nove instance igre Doom, učitava se konfiguracija za scenarij te se inicijalizira sama igra. Nakon toga postavljaju se početne vrijednosti potrebne za daljnju operaciju agenta, a to su broj akcija, prostor obzervacija te početne vrijednosti za zdravlje, poziciju, streljivo, itd.

Klasa također definira i *close* metodu kojom se nadjačava close metoda iz Gym Env klase. U slučaju poziva close komande zatvara se prozor igre. Također je definirana i *step* metoda koja je prisutna i u Env klasi. Implementacija te dvije metode vidljiva je u sljedećem isječku koda.

```
  def close(self):
    self.game.close()

  def step(self, action):
    action_reward = self.game.make_action(self.actions[action], 5)

    done = self.game.is_episode_finished()
    state = self.game.get_state()
```

```
    if not state:
      return np.zeros(self.observation_space.shape), action_reward, done, {}

    reward = action_reward
    if self.is_reward_shaping_on:
        reward += self.shape_rewards()

    img = self.process_frame(state.screen_buffer)

    return img, reward, done, {}
```

Step metoda kao argument dobiva *action* što je zapravo indeks akcije koju agent poduzima. Na objektu igre (*self.game*) poziva se metoda *make_action* kojoj se kao prvi argument šalje akcija iz jedinične matrice (*self.actions)* pod retkom *action*. Kao drugi argument šalje se takozvani frame-skip parametar koji prilikom odabira akcije tu istu akciju ponavlja sljedećih *frame-skip* tic-ova. Tako se olakšava ubijanje protivnika kada agent uspješno puca na neprijatelja. No, prevelika frame-skip vrijednost otežava agentovo nišanjenje jer u tom slučaju agent često pomakne pušku previše te u potpunosti prijeđe preko neprijatelja [54]. Prilikom odrađivanja akcije, sama igra vraća povratnu vrijednost koja označava nagradu za navedenu akciju. U ovoj vrijednosti reflektirat će se bilo koje ubojstvo kojeg agent ostvari. Ako je uključena opcija oblikovanja nagrada, na nagradu dobivenu za pojedinu akciju nadodaje se i oblikovana nagrada pozivom *shape_rewards* metode.

```
def shape_rewards(self):
       return sum([
           self.calculate_damage_reward(),
           self.calculate_ammo_reward(),
           self.calculate_health_reward(),
           self.calculate_armor_reward(),
           self.calculate_distance_reward(),
       ])

 def calculate_damage_reward(self):
     damage_dealt = self.game.get_game_variable(GameVariable.DAMAGECOUNT)
     delta_damage_dealt = damage_dealt - self.damage_dealt
     self.damage_dealt = damage_dealt

     reward = REWARD_SHAPING.DAMAGE_REWARD_FACTOR * delta_damage_dealt
     return reward

 def calculate_distance_reward(self):
     x, y = self.get_player_position()
     delta_x = self.x - x
     delta_y = self.y - y

     distance_moved = np.sqrt(delta_x ** 2 + delta_y ** 2)
     self.x = x
     self.y = y

     reward = REWARD_SHAPING.DISTANCE_REWARD if distance_moved > REWARD_SHAPING.
         DISTANCE_REWARD_THRESHOLD else REWARD_SHAPING.DISTANCE_PENALTY
```

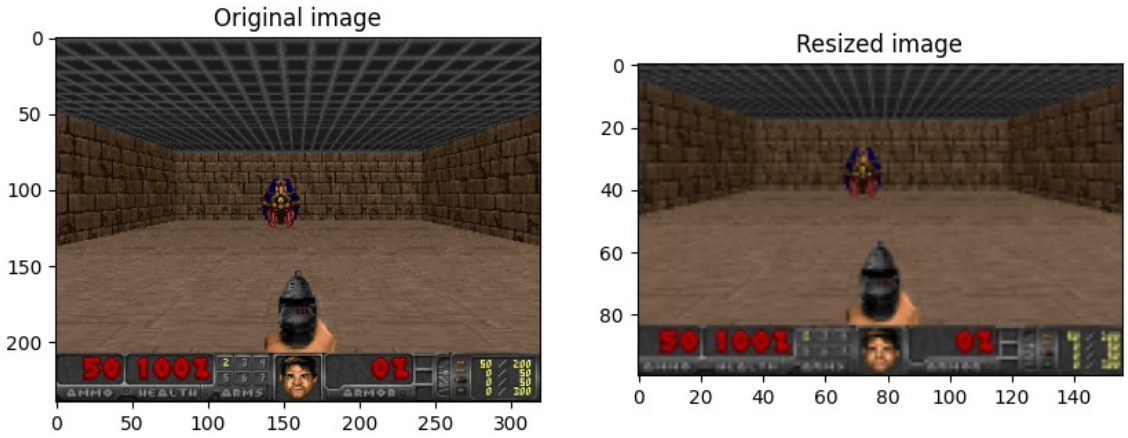
```
    return reward
```

Metoda *shape_rewards* izrazito je jednostavna. Poziva sve metode za oblikovanje nagrade te vraća sumu svih vraćenih vrijednosti. Metode za oblikovanje nagrada koje postoje su *calculate_damage_reward*, *calculate_ammo_reward*, *calculate_health_reward*, *calculate_armor_reward*, te *calculate_distance_reward*. Svaka od njih operira na sličnom principu, a to je usporedba stare vrijednosti (koja se u slučaju učinjene štete pohranjuje u *self.damage_dealt*) te nove vrijednosti dobivene pozivanjem self.game.get_game_variable komande. Dobivena razlika množi se s faktorom nagrade te metoda vraća tu vrijednost. U slučajevima gdje postoji i penalizacija uzima se maksimalna vrijednost između 0 i delte (odnosno same promjene između stare i nove vrijednosti) koja se množi s faktorom nagrade te minimalna vrijednost između 0 i delte koja se množi s faktorom penalizacije. Razlika između ta dva umnoška je sama nagrada agenta. Razlike postoje u metodi za izračun nagrade za streljivo gdje se za izračun delta vrijednosti uzima suma streljiva za sva oružja. Razlika također postoji i u izračunu nagrade za udaljenost. Nagrada (odnosno penali) za udaljenost postoji kako bi se spriječilo ponašanje u kojem agent ne istražuje već čeka ispred mjesta gdje se neprijatelji stvaraju. Takvo ponašanje naziva se kampiranjem (engl. *camping*) te je izrazito nepoželjno u slučaju kad na razini postoji nekoliko portala iz kojih neprijatelji izlaze. Ova klasa definira još 3 metode, metodu *reset*, *render* i *process_frame*.

```
def reset(self):
  self.game.new_episode()
  state = self.game.get_state()

  self.health = 100
  self.x, self.y = self.get_player_position()
  self.armor = self.frags = self.total_reward = self.deaths = self.damage_dealt =
      0

  return self.process_frame(state.screen_buffer)

def render(self):
  pass

def process_frame(self, observation):
  if observation.shape != EnvConfig.EXPECTED_IMAGE_SHAPE:
      raise ValueError(f"Unexpected␣observation␣shape.␣Expected␣{EnvConfig.
          EXPECTED_IMAGE_SHAPE},␣but␣got␣{observation.shape}.")

  resized = cv2.resize(observation[40:, 4:-4], None, fx=.5, fy=.5, interpolation=
      cv2.INTER_AREA)
  return resized
```

Metoda reset brine se o postavljanju svih vrijednosti na početne. Metoda je veoma slična kôdu implementiranom u konstruktoru klase. Postavljaju se početne vrijednosti koordinata igrača, zdravlja, oklopa, načinjene štete, itd. Render metoda služi kao prazna metoda kojom se nadjačava render metoda u Env klasi. Budući da igra funkcionira kroz ViZDoom biblioteku sama render metoda nije niti potrebna. Metoda za obradu vizualnih podataka provjerava dimenzije slike kako bi bile pravog oblika te baca pogrešku ako uvjet nije zadovoljen. U pro-

tivnom se slika obrađuje vertikalnim (gornjih 40 redova slike) te horizontalnim obrezivanjem (4 stupca s lijeve i desne strane). Nadalje se slika također manipulira i *resize* metodom iz *cv2* biblioteke. Kao faktor skaliranja po x i y osi postavlja se 0.5. Tako se dimenzije slike zapravo smanjuju za 50%. Tako se mijenja kvaliteta slika kako bi se izbjegle nepotrebne informacije. Interpolacija slike pri smanjenju kvalitete obavlja se cv2 *INTER_AREA* algoritmom koji odlično funkcionira pri smanjenju dimenzija slike. Izgled slike prije te izgled slike poslije moguće je vidjeti na sljedećem dijagramu.



Slika 7: Usporedba originalne slike i prilagođene slike

Kao što je vidljivo na prikazu, desna, prilagođena, slika je značajno mutnija te je iz slike jasno vidljivo da je dio stropa odrezan. Strop agentu predstavlja nepotrebne informacije koje on svakako mora obraditi. Rezanje nepotrebnih dijelova slike može znatno ubrzati agentovo učenje zbog smanjene količine podataka koje agent uzima u obzir.

2.9.2.4. EnvironmentHelpers.py

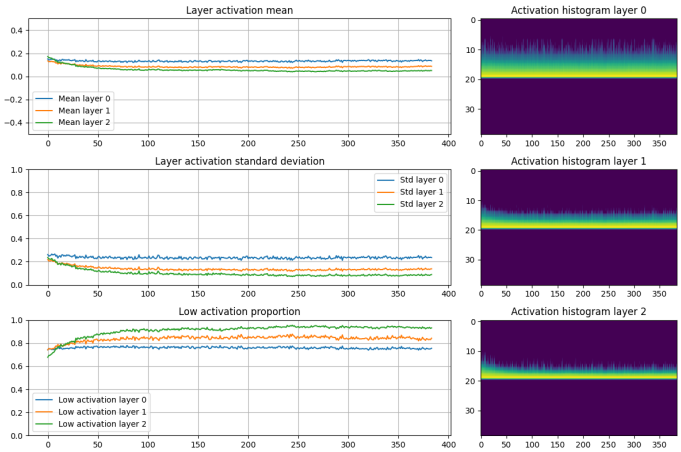
EnvironmentHelpers jednostavna je Python datoteka koja služi kao pomoćnik u kreiranju agentovog okružja. Radi se o datoteci koja sadrži jednu metodu pod nazivom *create_vectorised_environment*. Njena dužnost je omatanje instance VizdoomGym klase u klasu DummyVecEnv te naposljetku u VecTransposeImage klasu. Pri tom metoda prima dva argumenta, *n_envs* koji označava broj okružja koji je potrebno kreirati za paralelno izvršavanje, te argument *params* koji se destrukturira uz pomoć tzv. operatora raspakiravanja ***params*. Params predstavlja parametre koji se prosljeđuju samoj instanci VizdoomGym klase.

```
from stable_baselines3.common import vec_env
from stable_baselines3.common.vec_env import DummyVecEnv, VecTransposeImage
from VizdoomGymWrapper import VizDoomGym

def create_vectorised_environment(n_envs=1, **params) -> vec_env.
    VecTransposeImage:
    env = VecTransposeImage(
        DummyVecEnv([lambda: VizDoomGym(**params)] * n_envs)
    )
    return env
```

Navedena metoda instancira novi objekt klase VizDoomGym te ga pretvara u lambda funkciju. Lambda funkcija multiplicira se *n_envs* puta te se niz lambda funkcija prosljeđuje kao argument *DummyVecEnv* klasi iz biblioteke Stable Baselines3. DummyVecEnv kreira jednostavni vektorizirani omot oko više okružja. Tako je moguće u isto vrijeme izvršavati nekoliko instanci igre te višestruko ubrzati proces učenja kroz paralelno učenje. Klasa VecTransposeImage koristi se kao korak obrade slike. PyTorch konvolucijski slojevi rade s ulaznim slikama koje imaju format CxHxW pri čemu C označava slojeve slike (engl. *Channel*), a H i W visinu (engl. *Height*) i širinu (engl. *Width*). ViZDoom omogućava podešavanje formata ekrana, odnosno direktan utjecaj na rezoluciju pomoću *screen_resolution* opcije u konfiguracijama scenarija te na broj slojeva slike uz pomoć *screen_format* opcije u konfiguracijama. Po zadanim postavkama slika se prikazuje u CRCGCB formatu pri čemu se broj slojeva nalazi na prvom mjestu. No, prema uputama u ViZDoom konfiguracijskim datotekama, takav format nije pogodan za rad s OpenCV (odnosno cv2) bibliotekom. Stoga se kao zadani format u ViZDoom okružju postavlja RGB24, a transponiranje slike izvodi se kroz VecTransposeImage klasu.

2.9.2.5. Cnn.py

Na jednostavnijim okružjima agent pokazuje odlično snalaženje u okružju te vrlo brzo shvaćanje cilja. Nakon kratkog vremena uspješno savladava okružje. No, u kompleksnijim okružjima to nije slučaj. Uz pomoć *hooks* mehanizma na *features_extractor* dijelu politike moguće je na svaki aktivacijski sloj trenutne konvolucijske neuralne mreže (engl. *Convoluted Neural Network*, skraćeno *CNN*) dodati bilo kakvu funkcionalnost. U ovom slučaju implementira se praćenje aktivacije svakog aktivacijskog sloja. Objašnjenje o implementaciji hook-ova obrađeno je u besplatnom FastAI kursu u lekciji 10 [55]. Iz sljedeće slike vidljivo je kako je aktivacija pojedinih slojeva neuronske mreže veoma loša.



Slika 8: Prikaz aktivacije slojeva uz bazični CNN, izvor: vlastita izrada

Poboljšanje aktivacije svakog sloja može dovesti do kvalitetnijeg procesa učenja. Jedan od načina poboljšanja aktivacije je umjesto postojeće CNN implementacije pružiti svoju implementaciju. Time je moguće imati potpunu kontrolu nad svim slojevima te je slojeve moguće posložiti tako da se osigurava potpuna aktivacija. Vlastita implementacija CNN-a moguća je implementacijom vlastite klase koja proširuje klasu *BaseFeaturesExtractor* iz biblioteke Stable

Baselines3. Implementacija glasi:

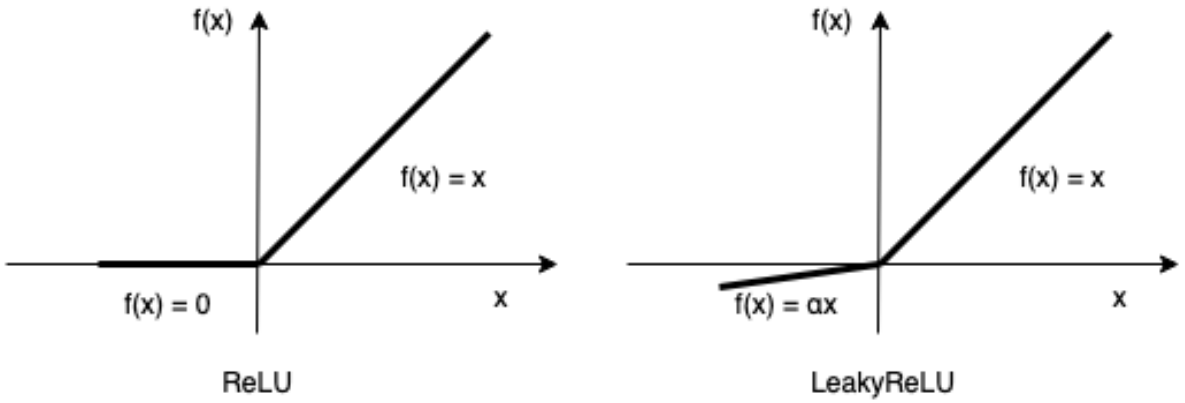
Listing 2.5: Ovo je kod za implementaciju prilagođene neuralne mreže [55], [56]

```
from stable_baselines3.common.torch_layers import BaseFeaturesExtractor
import torch as th
import torch.nn as nn
import gym


class CustomCNN(BaseFeaturesExtractor):
    def __init__(self, observation_space: gym.spaces.Box, features_dim: int = 128,
        **kwargs):
        super().__init__(observation_space, features_dim)

        self.cnn = nn.Sequential(
            nn.LayerNorm([3, 100, 156]),

            nn.Conv2d(3, 32, kernel_size=8, stride=4, padding=0, bias=False),
            nn.LayerNorm([32, 24, 38]),
            nn.LeakyReLU(**kwargs),

            nn.Conv2d(32, 64, kernel_size=4, stride=2, padding=0, bias=False),
            nn.LayerNorm([64, 11, 18]),
            nn.LeakyReLU(**kwargs),

            nn.Conv2d(64, 64, kernel_size=3, stride=1, padding=0, bias=False),
            nn.LayerNorm([64, 9, 16]),
            nn.LeakyReLU(**kwargs),

            nn.Flatten(),
        )

        self.linear = nn.Sequential(
            nn.Linear(9216, features_dim, bias=False),
            nn.LayerNorm(features_dim),
            nn.LeakyReLU(**kwargs),
        )

    def forward(self, observations: th.Tensor) -> th.Tensor:
        return self.linear(self.cnn(observations))
```

Konstruktor klase BaseFeaturesExtractor kao argumente prima veličinu ulazne slike (varijabla *observation_space*) te broj izlaznih dimenzija koje CNN proizvodi (varijabla *features_dim*). Iz tog razloga i vlastita implementacija prima iste argumente kao i dodatni argument pod nazivom *kwargs* koji se raspakirava u svakom LeakyReLU sloju. Vlastita implementacija CNN-a kreira se uz pomoć *Sequential* metode iz biblioteke *torch.nn*, a metoda kao argument prima niz slojeva koji čine završnu neuralnu mrežu. Implementacija vlastitog CNN-a bazira se na arhitekturi od 3 konvolucijska sloja te jednim potpuno konektiranim slojem. Takav dizajn standardan je za zadatke koji uključuju obradu vizualnih podataka. Nakon svakog konvolucijskog sloja dodaje se i sloj normalizacije koji često može pružiti dodatnu stabilnost u učenju.

Prvi sloj u mreži služi kao sloj normalizacije. Ulazi u ovaj sloj oblika su [3, 100, 156]. Takav oblik dobije se kada početnu dimenziju slike od [240, 320, 3] oblikujemo obrezivanjem nakon čega oblik postaje [200, 312, 3], mijenjamo veličinu slike faktorom od 0.5. Nakon smanjenja, slika prima oblik od [100, 156, 3] te naposljetku slika prolazi kroz VecTransposeImage klasu koja mijenja oblik slike u [3, 100, 156].

Normalizacijski sloj služi za normaliziranje ulaznih podataka te omogućava brže učenje te bolju preciznost u generalizaciji [57]. Zatim slijedi 2D konvolucijski sloj koji primjenjuje 32 filtera na ulaznim podatcima te se filter miče po 4 piksela u jednom koraku. Konvolucijski sloj pogodan je za pronalaženje uzoraka u ulaznim podatcima [56]. Nakon konvolucijskog sloja se podatci normaliziraju koristeći se sa LayerNorm te se normalizirani podatci prosljeđuju aktivacijskoj funkciji LeakyReLU (engl. *Leaky Rectified Linear Unit*). LeakyReLU prilagođena je verzija ReLU aktivacijske funkcije. ReLU funkcija se vrlo lako opisuje - ona funkcionira tako da za negativnu ulaznu vrijednost vraća 0, a za pozitivne ulazne vrijednosti funkcionira linearno iz čega je i dobila naziv *Rectified Linear Unit*. Implementacija ove funkcije u programskom jeziku također je jednostavna, za ulazni parametar *input* kôd funkcije glasi: `return max(0, input)`. U slučaju LeakyReLU funkcije radi se o maloj prilagodbi gdje se za negativne ulazne vrijednosti ne vraća 0 već se vrijednost množi s faktorom α. Programska implementacija glasi: `return max (a*input, input)` [58]. Razlika između ReLU aktivacijske funkcije te LeakyReLU aktivacijske funkcije vidljiva je na sljedećem dijagramu.



Slika 9: Prikaz ReLU (lijevo) i LeakyReLU (desno) aktivacijske funkcije, izvor: vlastita izrada

Navedena struktura (Conv2D, LayerNorm, LeakyReLU) ponavlja se još dva puta s različitim parametrima te se naposljetku izlaz posljednjeg sloja pretvara u jednodimenzionalni tensor uz pomoć *Flatten* funkcije. Time se završavaju konvolucijski slojevi neuralne mreže koji su odlični za prepoznavanje uzoraka [56]. Zatim se u *self.linear* svojstvo klase pohranjuje niz potpuno povezanih linearnih slojeva. Njihova svrha je kombiniranje uzoraka koje konvolucijski slojevi nauče te korištenje istih kako bi se iz prepoznatih karakteristika izabrala završna izlazna vrijednost [59].

2.9.2.6. LearningAgent.ipynb

Datoteka LearningAgent najbitnija je datoteka u cijelom direktoriju projekta. Radi se o Jupyter Notebook datoteci koja sadrži logiku potrebnu za podučavanje agenta u pojedinom ViZDoom okružju. Sav potreban kod za podučavanje agenta razlama se u tri Jupyter Notebook ćelije. Pokrenute zajedno, ćelije će inicijalizirati sve parametarske varijable, pokrenuti učenje agenta na dva okružja za učenje i jednom evaluacijskom okružju te nakon završetka učenja zatvoriti kreirana okružja.

Listing 2.6: Prva ćelija u datoteci LearningAgent.ipynb

```
from Cnn import CustomCNN
import EnvironmentConfigurations as EnvConfig

model_save_path = f"{EnvConfig.AGENT_MODEL_PATH_PREFIX}{EnvConfig.configurations
    [EnvConfig.CURRENT_CONFIGURATION_INDEX]['name']}";
tensorboard_log_path = f"{EnvConfig.TENSORBOARD_LOG_PATH_PREFIX}{EnvConfig.
    configurations[EnvConfig.CURRENT_CONFIGURATION_INDEX]['name']}"

env_params = {
    "env_config": EnvConfig.configurations[EnvConfig.CURRENT_CONFIGURATION_INDEX
        ],
    "is_reward_shaping_on": True,
    "is_game_window_visible": False
}

evaluation_env_params = {
    "env_config": EnvConfig.configurations[EnvConfig.CURRENT_CONFIGURATION_INDEX
        ],
    "is_reward_shaping_on": False,
    "is_game_window_visible": False
}

agent_params = {
    "tensorboard_log": tensorboard_log_path,
    "verbose": 1,
    "n_epochs": 3,
    "n_steps": 4096,
    "learning_rate": 1e-4,
    "batch_size": 64,
    "seed": 0,
    'policy_kwargs': {'features_extractor_class': CustomCNN}
}
```

U prvoj ćeliji 2.6 uvoze se potrebne biblioteke za kreiranje parametara za okružje, evaluacijsko okružje te agenta. Prilikom učenja se za okružje za učenje uključuje opcija oblikovanja nagrada te se ne prikazuje prozor videoigre. Tako se mogu poboljšati performanse učenja jer je potrebno manje resursa zbog uklanjanja vizualnog prikaza. Za evaluacijsko okružje također se ne prikazuje prozor igre, no u ovom slučaju se i oblikovanje nagrada isključuje. U evaluacijskom okružju poželjno je imati čistu nagradu (što u deathmatch scenariju uključuje samo broj ubojstava koje agent postigne). Oblikovanje nagrada često prolazi kroz nekoliko iteracija dok

se ne dođe do optimalnih faktora za oblikovane nagrade. Iz tog razloga oblikovana nagrada može značajno varirati te se ne smatra dobrim kandidatom za usporedbu između različitih implementacija istog agenta. Broj ubojstava, na drugu ruku, je savršeni kandidat za usporedbu performansi jer su baš agentove performanse ključna stavka koja utječe na broj ubojstava. Iz tog razloga se na evaluacijskom okružju ne primjenjuje oblikovanje nagrade te se prikupljeni podatci mogu prikazati u obliku grafa. Kao parametre agenta postavljaju se iznimno bitni hyper-parametri *n_epochs*, *learning_rate* te *batch_size*. Nazivaju se hyper-parametrima jer su upravo oni na vrhu hijerarhije parametara te su zaslužni za proces treniranja kao i za krajnji rezultat koji se postiže. Broj epoha (n_epochs) označava broj potpunih prolaza kroz cijeli dataset prikupljenih podataka. U jednoj epohi agent je napravio jedan potpuni prolaz kroz set podataka te prošao jednu iteraciju učenja. Postavljanjem broja epoha na tri ostvaruje se bolja efikasnost agenta s obzirom na količinu podataka kao i bolje performanse. Stopa učenja (learning_rate) predstavlja veličinu koraka u kojem se parametri modela (CNN-a) pomiču tijekom optimizacijskog procesa. Radi se o potencijalno najvažnijem hyper-parametru jer njegova vrijednost značajno utječe na sami proces učenja. Previsoka stopa učenja može dovesti do nestabilnosti u procesu učenja jer proces optimizacije može napraviti preveliki korak te "preletjeti" preko optimalnog rješenja. Preniska stopa učenja na drugu ruku može značajno usporiti proces optimizacije te zahtijevati veći broj koraka za konvergenciju u optimalno rješenje. U procesu učenja se često primjenjuje i varijabilna stopa učenja, pri čemu se stopa učenja povećava sve dok je učenje stabilno. Kada dođe do nestabilnosti u učenju stopa učenja se vraća na prethodnu vrijednost [60]. Batch size označava broj uzoraka koji se uključuje u svaku iteraciju učenja. U slučaju da se učenje sastoji od 1000 uzoraka u setu podataka, ako se batch size postavi na 32 tada model obrađuje set podataka po 32 uzorka u jednom trenu. Batch size se postavlja kao vrijednost koja je višekratnik broja 2. Često se preporučuje 32 ili 64. Niže vrijednosti daju bržu konvergenciju no u proces učenja se uvodi "buka" zbog toga što male količine uzoraka nisu reprezentativne za cijeli set podataka. Veće vrijednosti batch sizea smanjuju količinu buke u procesu učenja no i usporavaju konvergiranje [61].

Listing 2.7: Druga ćelija u datoteci LearningAgent.ipynb

```
from EnvironmentHelpers import create_vectorised_environment
from utils.AutomaticModelSavingCallback import AutomaticModelSavingCallback
from utils.Initialisation import initialise_network_weights
from stable_baselines3 import PPO
from stable_baselines3.common import policies
from stable_baselines3.common.callbacks import EvalCallback

env = create_vectorised_environment(**env_params, n_envs=2)
evaluation_env = create_vectorised_environment(**evaluation_env_params, n_envs
    =1)

automatic_model_saving_callback = AutomaticModelSavingCallback(
            check_freq=EnvConfig.MODEL_SAVING_FREQUENCY,
            save_path=model_save_path)

evaluation_callback = EvalCallback(
            evaluation_env,
            n_eval_episodes=10,
```

```
            eval_freq=EnvConfig.EVALUATION_FREQUENCY,
            log_path=tensorboard_log_path,
            best_model_save_path=f'models/{EnvConfig.configurations[EnvConfig.
                CURRENT_CONFIGURATION_INDEX]["name"]}')

model = PPO(policies.ActorCriticCnnPolicy, env, device="cuda", **agent_params)
initialise_network_weights(model.policy)
model.learn(total_timesteps=30000000, callback=[automatic_model_saving_callback,
     evaluation_callback])

# model = PPO.load(f"{model_save_path}/best_model", **agent_params)
# model.set_env(env)
# model.learn(total_timesteps=30000000, callback=[agentCallback,
    evaluation_callback], reset_num_timesteps=False)
```

Druga ćelija 2.7 služi za kreiranje okružja za učenje i evaluacijskog okružja te kreiranje takozvanih *callback* funkcija. Callback funkcije su funkcije koje se pozivaju tijekom procesa učenja. Uz pomoć njih moguće je pristupiti internom stanju samog modela tijekom učenja te se omogućava praćenje stanja, automatsko spremanje, izmjene modela, itd. U isječku koda callback funkcije koriste se za automatsko pohranjivanje modela te za evaluaciju modela koja svakih nekoliko vremenskih koraka provjerava performanse agenta te ako dođe do novog rekorda automatski pohranjuje najbolji model. Parametar *n_eval_episodes* u EvalCallback-u predstavlja broj epizoda na kojima se agent testira. Postavljanjem ovog parametra smanjuje se šansa da se agentu "posreći" prilikom evaluacije te da iz tog razloga ostvari dobar rezultat [50]. Zatim se kreira nova instanca PPO algoritma. Algoritmu se prosljeđuju prethodno definirani parametri agenta, kao i samo okružje. Također se omogućava definiranje uređaja. Kada je uređaj postavljen na "cuda", tada se učenje prvo pokušava izvesti na grafičkoj kartici ako ona podržava CUDA set alata. U protivnom se proces učenja izvršava na mikroprocesoru. Nakon kreiranja instance algoritma prvo se registriraju mehanizmi praćenja statistike te se, prema uputama FastAI kursa (lekcija 8) [55], inicijaliziraju početne vrijednosti neuralne mreže koristeći se Kaiming inicijalizacijom [62]. Kaiming normalizacija poboljšava performanse te osigurava model od gubitka gradijenta kada se radi o ReLU (ili LeakyReLU) slojevima u neuralnoj mreži. Kod za inicijalizaciju kaiming normalizacije glasi:

```
from torch import nn

def initialise_network_weights(module: nn.Module):
    if isinstance(module, nn.Conv2d) or isinstance(module, nn.Linear):
        nn.init.kaiming_normal_(
            module.weight,
            a=0.1,
            mode='fan_in',
            nonlinearity='leaky_relu'
        )

    for submodule in module.children():
        initialise_network_weights(submodule)
```

Nakon inicijalizacije početnih parametara pokreće se učenje agenta metodom *learn* na

modelu. Metodi se prosljeđuju callback funkcije te ukupni broj vremenskih koraka za treniranje agenta. Također postoji opcija učitavanja prethodno treniranog agenta te nastavak treniranja. U isječku koda su te linije koda zakomentirane. Učitavanje prethodnog modela izvodi se pozivom *PPO.load* metode, a također je potrebno i postaviti okružje pozivom *model.set_env* metode. Iznimno je bitno da se ne poziva *initialise_network_weights* metoda jer ona postavlja težine (engl. *weights*), odnosno parametre, neuralne mreže na inicijalnu vrijednost. Time se gubi svo znanje koje je prethodno naučeno te agent pokreće proces učenja iznova. Zadnja ćelija sastoji se od poziva *close* metoda nad okružjima za učenje i evaluaciju.

2.9.2.7. TestingAgent.ipynb

Datoteka TestingAgent također je Jupyter Notebook datoteka. Sastoji se od kôda koji se koristi za testiranje agenta i jednostavno pokretanje bilo kojeg scenarija te bilo kojeg prethodno treniranog agenta. Datoteka sadrži 3 Notebook ćelije u kojima se dešava kreiranje parametara za okružje, učitavanje željenog modela te naposljetku pokretanje N broja epizoda (definirano u konfiguracijskoj datoteci) vodeći se predikcijama koje model napravi.

```
import EnvironmentConfigurations as EnvConfig

env_params = {
    "env_config": EnvConfig.configurations[EnvConfig.CURRENT_CONFIGURATION_INDEX
        ],
    "is_reward_shaping_on": False,
    "is_game_window_visible": True
}
```

Isječak koda uvozi konfiguracije okružja te kreira parametre za jedno okružje. U ovom slučaju oblikovanje nagrada nema smisla pošto se taj mehanizam odnosi na konkretno učenje agenta. Parametar za vidljivost prozora videoigre postavlja se na True kako bi se videoigra prikazala.

```
from EnvironmentHelpers import create_vectorised_environment
from stable_baselines3 import PPO
import glob, os

model_save_path = f"{EnvConfig.AGENT_MODEL_PATH_PREFIX}{EnvConfig.configurations
    [EnvConfig.CURRENT_CONFIGURATION_INDEX]['name']}"
all_model_files = glob.glob(f"{model_save_path}/*")
latest_model_path = max(all_model_files, key=os.path.getctime)

env = create_vectorised_environment(**env_params, n_envs=1)
model = PPO.load(f"{latest_model_path}")
```

Druga ćelija koda zaslužna je za pronalazak posljednje datoteke agenta iz direktorija koji je naveden u samoj konfiguraciji okružja. Pronalazak putanja svake datoteke u direktoriju moguć je korištenjem *glob* metode iz istoimene biblioteke. Koristi se i *max* metoda kojoj se prosljeđuje lista datoteka te funkcija koja vraća vrijeme kreiranja datoteke koje služi kao ključ. Rezultat te dvije metode su pronalazak datoteke koja je po vremenu kreiranja zadnja u direktoriju. Zatim se kreira vektorizirano okružje s brojem okružja jedan, te se pomoću *PPO.load*

metode učitava posljednji model iz direktorija modela.

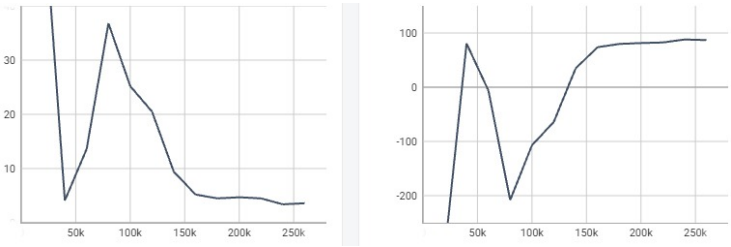
```
import time

for episode in range(EnvConfig.EPISODES_NUM):
    obs = env.reset()
    done = False
    total_reward = 0
    while not done:
        action, second = model.predict(obs)
        obs, reward, done, info = env.step(action)
        time.sleep(0.05555)
        total_reward += reward
    print("Total reward: ", total_reward)
    time.sleep(1)
```

Za pokretanje agenta u igri potrebna je jednostavna *for* petlja. Izvršava se N epizoda prema konfiguraciji te se u svakoj epizodi okružje postavlja na početno stanje. Također se postavljaju vrijednosti varijabli *done* koja označava da je uvjet završnog stanja ispunjen i *total_reward* u koju se upisuje ukupna nagrada za epizodu. Svaka epizoda se izvršava dok završno stanje nije postignuto (agent je ubijen, uvjet prelaska razine je ispunjen ili je isteklo vrijeme definirano u konfiguraciji za pojedini scenarij). Predviđanje akcije prema trenutnom stanju moguće je uz *predict* metodu na modelu. Modelu se pritom prosljeđuje trenutna slika koju agent vidi. Agent izvršava akciju pomoću *step* metode na okružju, te okružje vraća iduće stanje (sljedeća slika), nagradu i status igre (igra je gotova ili ne). Kako bi se igra mogla promatrati te da igra ne bi bila prebrza, nakon izvršavanja svake akcije program se pauzira na 0.05555 sekundi. U vrijednost ukupne nagrade dodaje se nagrada trenutnog vremenskog trenutka te se proces ponavlja sve dok epizoda ne završi. Na kraju epizode se u konzolu ispisuje ukupna nagrada za epizodu te se igra pauzira na 1 sekundu.

2.9.3. Eksperimenti na različitim scenarijima

2.9.3.1. Bazični scenarij

Bazični scenarij je početni scenarij u kojem agent može djelovati. Radi se o jednostavnom scenariju s jednim neprijateljem koji ima mogućnost micanja lijevo i desno. Agent je naoružan pištoljem te su mu dostupne 3 akcije, korak u lijevo, korak u desno te pucanje pištoljem. Agent ovaj scenarij svladava veoma brzo te brzo shvaća cilj samog okružja. Krajnji rezultat učenja je taj da agent ubija neprijatelja u samo nekoliko akcija što se dešava u nekoliko stotinki sekunde. Uspješno treniranje agenta u ovom scenariju traje 200-tinjak tisuća koraka. Nakon treniranja agent postiže prosječnu nagradu oko 87. Prosječno trajanje epizode je 3.5 vremenska koraka okružja. Odnosno, agent u prosjeku u 3.5 akcije završi epizodu. Stvarno trajanje epizode može varirati ovisno o poziciji u kojoj se neprijatelj nasumično stvori.



Slika 10: Prosječno trajanje epizode te prosječna nagrada u basic scenariju, izvor: vlastita izrada Tensorboard alatom

2.9.3.2. "Deadly corridor" scenarij

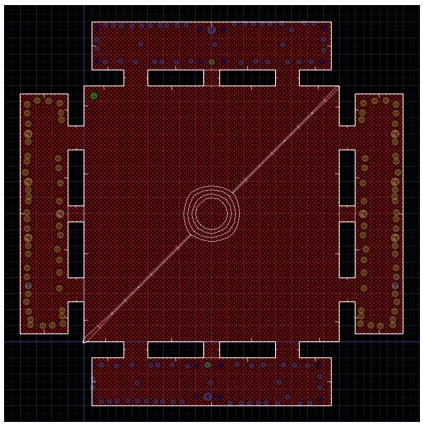
Scenarij smrtonosnog hodnika kompleksniji je od bazičnog scenarija u kojem se agent vrlo lako snašao. U ovom scenariju radi se o dugačkom hodniku s tri proširenja hodnika, odnosno manje prostorije. U svakom proširenju nalaze se dva neprijatelja, pri čemu su oba neprijatelja u prvoj prostoriji vrste Zombieman. Kompleksnost ovog scenarija nalazi se u specifičnom obliku prostorija koji osigurava smanjeno vidno polje. Hod u ravnoj liniji znači da oba neprijatelja u prostoriji u isto vrijeme mogu pucati u agenta, a on može pucati samo u jednog od njih. U takvom scenariju agent mora ili biti izrazito brz u svom ciljanju, ili mora primijeniti strategiju "grljenja" zida pri čemu agent hoda uz jedan od bočnih zidova kako bi jednom neprijatelju zatvorio kut vidnog polja te borbu 1 naprema 2 pretvorio u borbu 1 na 1. Izgled smrtonosnog hodnika iz ptičje perspektive te iz perspektive agenta moguće je vidjeti na sljedećoj slici.

Slika 11: Smrtonosni hodnik iz ptičje perspektive (gore) te agentove perspektive (dolje), izvor: vlastita izrada uz pomoć matplotlib biblioteke te alata SLADE 3 i GIMP

2.9.3.3. "Deathmatch" scenarij

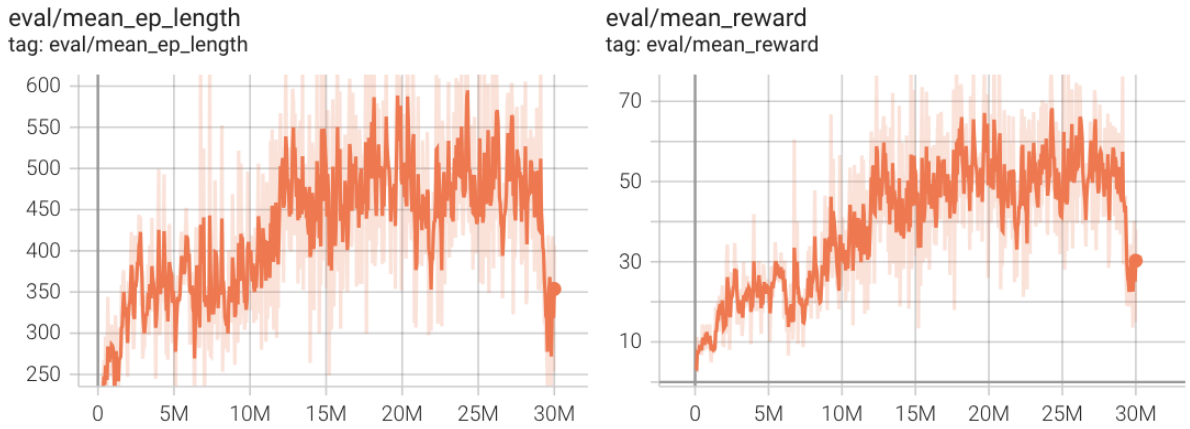
Deathmatch scenarij je bez sumnje najkompleksniji scenarij u kojem se agent može pronaći. Zbog nekoliko prostorija s nekoliko ulaza, više vrsti oružja kao i nekoliko vrsti predmeta

koje agent može pokupiti te brojnih neprijatelja s kojima se može susresti, ovo je scenarij koji je najteži za svladati. Kako bi agent uspješno svladao scenarij mora kombinirati nekoliko različitih oblika ponašanja kao što su navigacija prostorom, skupljanje oklopa, paketa zdravlja i streljiva kada je to potrebno te borba s neprijateljima. Izgled same karte za deathmatch scenarij vidljiv je na idućoj slici.



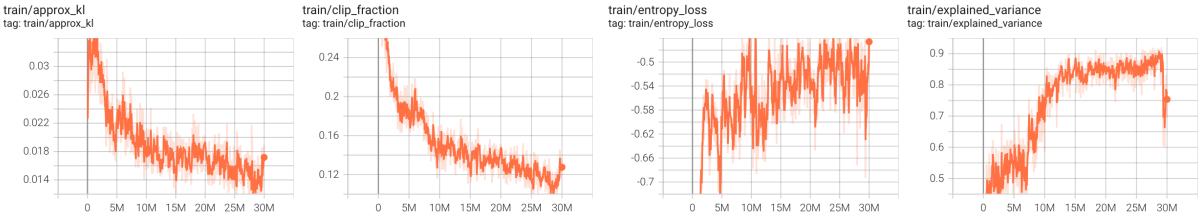
Slika 12: Izgled karte za deathmatch razinu, izvor: vlastita izrada putem besplatnog alata SLADE 3

Agent je na ovom scenariju podučavan 30 milijuna vremenskih koraka te nakon tog treniranja ostvaruje izvrsne rezultate u borbi s čudovištima. Sljedeća slika prikazuje grafove s prosječnim trajanjem epizode te prosječnom nagradom dodijeljenom na kraju svake epizode. Grafovi iscrtavaju navedene vrijednosti kroz cjelokupni proces učenja.



Slika 13: Prosječno trajanje epizode te prosječna nagrada u deathmatch scenariju, izvor: vlastita izrada Tensorboard alatom

Iz navedenih vrijednosti, kao i iz grafova na prethodnoj slici, moguće je zaključiti da je proces učenja bio veoma stabilan. Sljedeći grafovi prikazuju vrijednost predviđene KL divergencije te usko povezani postotak "clippinga". Također je vidljiv i graf s objašnjenjem varijance (engl. *explained variance*). Objašnjenje varijance statistička je metrika koja definira performanse samog modela. Ona opisuje proporciju varijance u krajnjem rezultatu koja je objašnjena predikcijom modela. U drugim riječima, objašnjenje varijance opisuje u kolikoj mjeri se poduda-raju agentove predikcije sa stvarnim rezultatom određenog stanja. Vrijednost parametra nalazi



Slika 14: Statistike za Deathmatch scenarij, izvor: vlastita izrada alatima Tensorboard te GIMP

se u rasponu od 0 do 1, pri čemu 0 označava da agent ne zna predvidjeti nijanse određenog stanja te u svakom slučaju predvidi prosječnu nagradu u okružju. Kada je vrijednost parametra jednaka 1, to označava da agent u svakoj situaciji sa stopostotnom sigurnošću ispravno određuje krajnji rezultat [50]. U svim navedenim grafovima moguće je vidjeti stabilnost procesa učenja.

2.9.4. Moguća poboljšanja

Postoje brojni načini na koji bi se potencijalno mogle poboljšati performanse agenta. Neki od načina uključuju olakšavanje korištenja istog agenta u scenarijima s različitim konfiguracijskim datotekama, odnosno različitim brojem mogućih akcija. Također postoje načini koji potencijalno poboljšavaju agentovo strategiranje ili jednostavno poboljšavaju njegovu spretnost u mnogim zadatcima. Neka od potencijalnih poboljšanja su:

- Dodavanje maske za akcije (engl. *action mask*) kako bi istog agenta bilo lakše implementirati u raznim scenarijima [63].
- Implementacija mehanizma za pamćenje stanja, odnosno pamćenje što se nalazi iza agenta [64][65]
- Poboljšanje treniranja kroz učenje pomoćnih zadataka [66]
- Poboljšanje treniranja u sustavu s nekoliko instanci istog agenta (ili različitih agenata s drukčijim hiperparametrima/faktorima za oblikovanje nagrada)

3. Zaključak

Cilj ovog rada bio je implementirati inteligentnog agenta u revolucionarnoj videoigri "DOOM". Cilj je postignut te je agent u stanju samostalno poraziti najjače neprijatelje u igri. No, za stjecanje takvih rezultata agent je morao provesti veliku količinu vremena u procesu treniranja. Samim time, vrijedno je napomenuti da je agent mehanički sposobniji od većine igrača, no protiv pravog igrača gubi bitku zbog nedostatka strateškog razmišljanja. Može se reći da agent "luta" razinom sve dok ne naiđe na protivnika nakon čega ga sa stopostotnom preciznošću i nevjerojatnim refleksima pogađa. Potencijalno bi se agenta moglo naučiti i strateškom razmišljanju, no za to bi bili potrebni znatno veći resursi.

OpenAI tim je za razvoj svojeg OpenAI Five [67] projekta koristio 256 grafičkih kartica te 128000 jezgri procesora. OpenAI Five se bazira na kompleksnijoj igri s većim fokusom na strateško razmišljanje, no svakako se u obzir mogu uzeti ogromni razmjeri resursa potrebnih za razvoj inteligentnog agenta sposobnim za smišljanje strategije.

Za potrebe ovog rada korišteni su razni alati kao što su GIMP za uređivanje slika i dijagrama. Za pisanje koda korišteno je razvojno okruženje Visual Studio Code uz razne ekstenzije koje pomažu pri pisanju Python koda u Jupyter Notebook okruženju. Upravljanje bibliotekama provedeno je conda okružjem, a od bitnijih biblioteka korištene su ViZDoom, OpenAI Gym, Stable Baselines3 te razne druge pomoćne biblioteke. Proces učenja agenta implementiran je pomoću modernog algoritma pod nazivom PPO.

Popis literature

[1] University of Michigan, *Middle English Compendium*, 2000. adresa: http://quod.lib.umich.edu/m/middle-english-dictionary/.

[2] M. Negnevitsky, *Artificial intelligence: a guide to intelligent systems*. Pearson education, 2005.

[3] C. Manning, *Artificial Intelligence Definitions*, Stanford University Human Centered AI, rujan 2020. adresa: https://hai.stanford.edu/sites/default/files/2020-09/AI-Definitions-HAI.pdf (pogledano 20. 8. 2023.).

[4] „Professor John McCarthy - Father of AI," *Stanford*, adresa: http://jmc.stanford.edu/articles/mcc59.html (pogledano 9. 8. 2023.).

[5] J. McCarthy, „Programs with Common Sense," *Proceedings of the Teddington Conference on the Mechanization of Thought Processes*, London: Her Majesty's Stationary Office, 1959., str. 75–91.

[6] S. Russell, P. Norvig, E. Davis i dr., *Artificial Intelligence: A Modern Approach*, 3rd. Pearson, 2016.

[7] J. Brownlee, „A Tour of Machine Learning Algorithms," kolovoz 2019. adresa: https://machinelearningmastery.com/a-tour-of-machine-learning-algorithms/ (pogledano 20. 8. 2023.).

[8] R. Chahar i D. Kaur, „A systematic review of the machine learning algorithms for the computational analysis in different domains," *International Journal of Advanced Technology and Engineering Exploration*, sv. 7, br. 71, str. 147, 2020.

[9] J. G. Carbonell, R. S. Michalski i T. M. Mitchell, „An overview of machine learning," *Machine learning*, str. 3–23, 1983.

[10] R. S. Sutton i A. G. Barto, *Reinforcement learning: An introduction, second edition*. MIT press, 2018.

[11] N. W. Gillham, „Sir Francis Galton and the birth of eugenics," *Annual review of genetics*, sv. 35, br. 1, str. 83–101, 2001.

[12] M. Costello, *An Introduction to Crossbow Hunting*, 2023. adresa: https://www.amstat.org/asa/files/pdfs/POL-CostelloRegressionSlides.pdf.

[13] A. K. Jain i R. C. Dubes, *Algorithms for clustering data*. Prentice-Hall, Inc., 1988.

[14] *Dendogram*, 2023. adresa: https://datavizproject.com/data-type/dendrogram/.

[15] F. Pedregosa, G. Varoquaux, A. Gramfort i dr., „Scikit-learn: Machine Learning in Python," *Journal of Machine Learning Research*, sv. 12, str. 2825–2830, 2011.

[16] J. Han, J. Pei i H. Tong, *Data mining: concepts and techniques*. Morgan kaufmann, 2022.

[17] V. Bajaj, „Unsupervised Learning for Anomaly Detection," *Towards Data Science*, kolovoz 2020. adresa: https://towardsdatascience.com/unsupervised-learning-for-anomaly-detection-44c55a96b8c1 (pogledano 11. 9. 2023.).

[18] W. H. Gantt. „Ivan Pavlov," Encyclopedia Britannica. (lipanj 2023.), adresa: https://www.britannica.com/biography/Ivan-Pavlov (pogledano 20. 8. 2023.).

[19] I. P. Pavlov, *Conditioned Reflexes: An Investigation of the Physiological Activity of the Cerebral Cortex*, prijevod G. V. Anrep. London: Oxford University Press, 1927., str. 142.

[20] E. L. Thorndike, *Animal Intelligence: An Experimental Study of the Associative Processes in Animals*. Columbia University Press, 1898. DOI: 10.1037/10780-000.

[21] R. S. Sutton i A. G. Barto, *Reinforcement learning: An introduction, second edition, in progress*. MIT press, 2015.

[22] W. Mischel i E. B. Ebbesen, „Attention in delay of gratification," *Journal of Personality and Social Psychology*, sv. 16, br. 2, str. 329–337, 1970. DOI: 10.1037/h0029815.

[23] J. K. Blitzstein i J. Hwang, *Introduction to probability*. Crc Press, 2019.

[24] A. Y. Ng, D. Harada i S. Russell, „Policy invariance under reward transformations: Theory and application to reward shaping," *Icml*, Citeseer, sv. 99, 1999., str. 278–287.

[25] N. E. Institute, „Farsightedness (Hyperopia)," rujan 2020. adresa: https://www.nei.nih.gov/learn-about-eye-health/eye-conditions-and-diseases/farsightedness-hyperopia (pogledano 17. 8. 2023.).

[26] R. E. Bellman, *Dynamic programming*. Princeton university press, 2010.

[27] University of Texas, *Topic 26 Dynamic Programming*, Prezentacija za kolegij, 2023.

[28] *Overlapping Sub-problems*, Webpage. adresa: https://www.javatpoint.com/overlapping-sub-problems (pogledano 22. 6. 2023.).

[29] zyxue i nbro, *Answer to: What is the difference between value iteration and policy iteration? [closed]*, Stack Overflow, Original post by zyxue on Feb 27, 2017; Revised by nbro on Jan 22, 2018, veljača 2017. adresa: https://stackoverflow.com/revisions/42493295/8.

[30] N. Jiang, *Reinforcement Learning and Monte-Carlo Methods*. University of Illinois Urbana-Champaign, 2021.

[31] R. Warren. „Reinforcement Learning Introduction Part 3," GitHub. (travanj 2020.), adresa: https://richard-warren.github.io/blog/rl_intro_3/ (pogledano 22. 8. 2023.).

[32] E. V. Labs, *Reinforcement Learning beginner to master - AI in Python*, Online tečaj, Udemy, 2022. adresa: https://www.udemy.com/course/beginner-master-rl-1/ (pogledano 22. 8. 2023.).

[33] J. Schulman, F. Wolski, P. Dhariwal, A. Radford i O. Klimov, *Proximal Policy Optimization Algorithms*, 2017. arXiv: 1707.06347 [cs.LG].

[34] R. J. Williams, „Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine learning*, sv. 8, str. 229–256, 1992.

[35] V. Konda i J. Tsitsiklis, „Actor-critic algorithms," *Advances in neural information processing systems*, sv. 12, 1999.

[36] J. Schulman, S. Levine, P. Moritz, M. I. Jordan i P. Abbeel, „Trust Region Policy Optimization," *CoRR*, sv. abs/1502.05477, 2015. arXiv: 1502.05477.

[37] S. Kakade i J. Langford, „Approximately optimal approximate reinforcement learning," *Proceedings of the Nineteenth International Conference on Machine Learning*, 2002., str. 267–274.

[38] D. Kushner, *Masters of Doom: How Two Guys Created an Empire and Transformed Pop Culture*. USA: Random House Inc., 2003., ISBN: 1588362892.

[39] Fandom, *Doom Wiki*, https://doom.fandom.com/, 2023.

[40] id Software, Inc., *Doom Manual*, 1993. adresa: https://www.starehry.eu/download/action3d/docs/Doom-Manual.pdf.

[41] G. Brockman, V. Cheung, L. Pettersson i dr., „OpenAI Gym," *CoRR*, sv. abs/1606.01540, 2016. arXiv: 1606.01540.

[42] F. Foundation. „Announcing The Farama Foundation: The future of open source reinforcement learning." (listopad 2022.), adresa: https://farama.org/Announcing-The-Farama-Foundation (pogledano 15. 6. 2023.).

[43] M. Wydmuch, M. Kempka i W. Jaśkowski, „ViZDoom Competitions: Playing Doom from Pixels," *IEEE Transactions on Games*, sv. 11, br. 3, str. 248–259, 2019., The 2022 IEEE Transactions on Games Outstanding Paper Award. DOI: 10.1109/TG.2018.2877047.

[44] C. Beattie, J. Z. Leibo, D. Teplyashin i dr., „DeepMind Lab," *CoRR*, sv. abs/1612.03801, 2016. arXiv: 1612.03801.

[45] M. Johnson, K. Hofmann, T. Hutton i D. Bignell, „The Malmo Platform for Artificial Intelligence Experimentation," *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, serija IJCAI'16, New York, New York, USA: AAAI Press, 2016., str. 4246–4247, ISBN: 9781577357704.

[46] Y. Bengio, J. Louradour, R. Collobert i J. Weston, „Curriculum Learning," *Proceedings of the 26th Annual International Conference on Machine Learning*, serija ICML '09, Montreal, Quebec, Canada: Association for Computing Machinery, 2009., str. 41–48, ISBN: 9781605585161. DOI: 10.1145/1553374.1553380.

[47] A. Dosovitskiy i V. Koltun, „Learning to Act by Predicting the Future," *CoRR*, sv. abs/1611.01779, 2016. arXiv: 1611.01779.

[48] *Anaconda Software Distribution*, verzija Vers. 2-2.4.0, 2020. adresa: https://docs.anaconda.com/.

[49] J. Güse, „How to set up Anaconda and Jupyter Notebook the right way," *Towards Data Science*, siječanj 2021. adresa: https://towardsdatascience.com/how-to-set-up-anaconda-and-jupyter-notebook-the-right-way-de3b7623ea4a (pogledano 16. 6. 2023.).

[50] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus i N. Dormann, „Stable-Baselines3: Reliable Reinforcement Learning Implementations," *Journal of Machine Learning Research*, sv. 22, br. 268, str. 1–8, 2021.

[51] J. Hare, *Dealing with Sparse Rewards in Reinforcement Learning*, 2019. arXiv: 1910.09281 [cs.LG].

[52] A. Zai i B. Brown, *Deep reinforcement learning in action*. Manning Publications, 2020.

[53] Y. Hu, W. Wang, H. Jia i dr., „Learning to Utilize Shaping Rewards: A New Approach of Reward Shaping," *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan i H. Lin, ur., sv. 33, Curran Associates, Inc., 2020., str. 15 931–15 941.

[54] C. Schulze i M. Schulze, „ViZDoom: DRQN with Prioritized Experience Replay, Double-Q Learning, & Snapshot Ensembling," *CoRR*, sv. abs/1801.01000, 2018. arXiv: 1801.01000.

[55] J. Howard i dr., *fastai*, https://github.com/fastai/fastai, 2018.

[56] A. Krizhevsky, I. Sutskever i G. E. Hinton, „Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, sv. 25, 2012.

[57] J. Xu, X. Sun, Z. Zhang, G. Zhao i J. Lin, „Understanding and improving layer normalization," *Advances in Neural Information Processing Systems*, sv. 32, 2019.

[58] Y. Wang, Y. Li, Y. Song i X. Rong, „The influence of the activation function in a convolution neural network model of facial expression recognition," *Applied Sciences*, sv. 10, br. 5, str. 1897, 2020.

[59] K. Simonyan i A. Zisserman, *Very Deep Convolutional Networks for Large-Scale Image Recognition*, 2015. arXiv: 1409.1556 [cs.CV].

[60] M. H. Beale, M. T. Hagan i H. B. Demuth, „Neural network toolbox," *User's Guide, MathWorks*, sv. 2, str. 77–81, 2010.

[61] M. Li, T. Zhang, Y. Chen i A. J. Smola, „Efficient mini-batch training for stochastic optimization," *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2014., str. 661–670.

[62] K. He, X. Zhang, S. Ren i J. Sun, „Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," *Proceedings of the IEEE international conference on computer vision*, 2015., str. 1026–1034.

[63] C.-Y. Tang, C.-H. Liu, W.-K. Chen i S. D. You, „Implementing action mask in proximal policy optimization (PPO) algorithm," *ICT Express*, sv. 6, br. 3, str. 200–203, 2020., ISSN: 2405-9595. DOI: https://doi.org/10.1016/j.icte.2020.05.003.

[64] M. J. Hausknecht i P. Stone, „Deep Recurrent Q-Learning for Partially Observable MDPs," *CoRR*, sv. abs/1507.06527, 2015. arXiv: 1507.06527.

[65] G. Lample i D. S. Chaplot, „Playing FPS Games with Deep Reinforcement Learning," *Proceedings of the AAAI Conference on Artificial Intelligence*, sv. 31, br. 1, veljača 2017. DOI: 10.1609/aaai.v31i1.10827.

[66] P. Vafaeikia, K. Namdar i F. Khalvati, „A Brief Review of Deep Multi-task Learning and Auxiliary Task Learning,” *CoRR*, sv. abs/2007.01126, 2020. arXiv: 2007.01126.

[67] C. Berner, G. Brockman, B. Chan i dr., „Dota 2 with Large Scale Deep Reinforcement Learning,” *CoRR*, sv. abs/1912.06680, 2019. arXiv: 1912.06680.

Popis slika