Metode umjetne inteligencije za implementaciju agenata koji igraju računalne igre

Nina, Perić

Master's thesis / Diplomski rad

2020

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:211:165189

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Nina Perić

METODE UMJETNE INTELIGENCIJE ZA IMPLEMENTACIJU AGENATA KOJI IGRAJU RAČUNALNE IGRE

DIPLOMSKI RAD

Varaždin, 2020.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Nina Perić

Matični broj: 43989/15-R

Studij: Organizacija poslovnih sustava

METODE UMJETNE INTELIGENCIJE ZA IMPLEMENTACIJU AGENATA KOJI IGRAJU RAČUNALNE IGRE

DIPLOMSKI RAD

Mentor/Mentorica:

Izv. prof. dr. sc. Markus Schatten

Varaždin, srpanj 2020.

Nina Perić

Izjava o izvornosti

Izjavljujem da je moj završni/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/Autorica potvrdio/potvrdila prihvaćanjem odredbi u sustavu FOI-radovi

Sažetak

U ovom radu objasnit će se temelji pojmovi vezani uz razvoj umjetne inteligencije. Definirati će se što je agent kao nositelj umjetne inteligencije te svojstva okružja koja karakteriziraju probleme koje agent riješava. Obradit će se dva opća pristupa razvoju umjetne inteligencije te njihove pripadne metode, algoritmi i koncepti. U sklopu simboličkog pristupa objasnit će se načini na koje agenti riješavaju probleme pretraživanja i odlučivanja korišenjem logike, reprezentacije znanja te zaključivanja na temelju vjerojatnosti u uvjetima nesigurnosti. Kao dio statističkog pristupa opisat će se metode korištene u nadziranom (eng. *Supervised*), nenadziranom (eng. *Unsupervised*) te pojačanom (eng. *Reinforcement*) učenju. One uključuju metode klasifikacije, regresije, klasteriranja te metode učenja politika. U sklopu strojnog učenja opisat će se primjena umjetnih neuronskih mreža za ostvarivanje dubokog učenja (eng. *Deep Learning*). Nakon objašnjavanja različitih metoda umjetne inteligencije objasnit će se razvoj praktičnog primjera metode pojačanog učenja na primjeru razvoja SPADE agenta koji igra računalnu igru Cartople iz OpenAi Gym biblioteke. Agent će koristiti deep Q-learning metodu koja uključuje rad s neuronskom mrežom za čiju implementaciju će se koristiti Tensorflow biblioteka.

Ključne riječi: umjetna inteligencija, agent, pojačano učenje, neuronske mreže, SPADE, OpenAi Gym, Tensorflow

Sadržaj

1. Uvod

Umjetna inteligencija za mnoge je do nedavno bio pojam koji pripada znanstvenoj fantastici, no danas je to jedna od tehnologija na kojoj se ostvaruje značajan napredak u razvoju te ona nalazi sve veću primjenu u svijetu oko nas. Od virtualnih pomoćnika, samovozećih auti, tražilica te algoritama koji nam preporučuju sadržaj na platformama kao što su YouTube ili Netflix, možemo pretpostaviti da je većina tehnološki osviještenih ljudi došla u kontakt s umjetnom inteligencijom i bez da je o tome možda razmišljala. Ipak, mnogim ljudima umjetna inteligencija i dalje predstavlja misteriozan, nekima čak i zastrašujuć pojam. Njezine mogućnosti naizgled su ograničene samo računalnim resursima svog vremena i kreativnošću ljudi koji ju primjenjuju.

Motivacija za ovaj rad je razjasniti pojam umjetne inteligencije, te kroz shvaćanje različitih načina na koji se ona ostvaruje dobiti bolji uvid u sam pojam, njegove mogućnosti i prepreke kod implementacije. Objasniti će se što umjetna inteligencija obuhvaća zajedno s temeljim pojmovima. U prvoj polovici rada prikazat će se različiti pristupi razvoju umjetne inteligencije, metode koje pojedini pristup obuhvaća te koncepte i algoritme koji se koriste u njihovoj primjeni. Razmotrit će se kakvi problemi se mogu riješavati korištenjem umjetne inteligencije te koji su izazovi prilikom implementacije tih riješenja. Nakon što definiramo pojmove, pristupe, metode i koncepte, obrazložit ćemo pristup razvoju praktičnog djela ovog rada, tj. implementaciju agenta koji uči igrati računalnu igru koristeći metodu pojačanog učenja. Opisat će se korištene tehnologije i način na koji se koriste. Zatim će se pokazati njihova primjena na razvoju agenta sa sposobnošću učenja igranja odabrane igre. Kroz implementaciju objasnit će se korišteni koncepti te njihova primjena na konkretnom primjeru.

2. Metode razvoja umjetne inteligencije

Prije nego što krenemo s opisom pristupa i metoda, potrebno je razjasniti temeljne pojmove vezane uz umjetnu inteligenciju te dio povijesti njenog razvoja. Kao začetak umjetne inteligencije autori utjecajne knjige o području Russel i Norvig navode 1956. godinu kada je održana prva radionica za razmjenu ideja o racionalnim računalnim programima [1]. Od tog događaja polje umjetne inteligencije prošlo je kroz brojna razdoblja naglog razvoja i stagnacije. Bilo je neslaganja o cilju samog polja te načina na koji se trebaju ostvariti ti ciljevi. Dio neslaganja proizlazi iz pitanja treba li umjetna inteligencija emulirati ljudsko razmišljanje ili nadići taj princip i pronaći svoj optimalan pristup razvoju inteligencije. Također kao što je slučaj sa svakom novom i nedokazanom znanosti, njen je napredak ovisio o novčanom ulaganju u istraživanje, što pak ovisi o isplativosti primjene te znanosti u praksi. Kroz prethodnih 60 godina razvoja javile su se brojne isplative primjene umjetne inteligencije, no njezine mogućnosti nisu uvijek ispunjavala ambiciozna obećanja. Trenutno živimo u periodu u kojem umjetna inteligencija ima brojne isplative primjene, značajnije nego ikada u povijesti, no važno je održati razumna očekivanja u vidu ograničenja tehnologije i teorije. Kroz razvoj umjetne inteligencije osim računalne znanosti prepoznat je značaj primjene teorije i metoda brojnih drugih znanosti kao što su matematika, filozofija, logika, statistika, ekonomija, lingvistika i ostale. Brojne metode koje ćemo prikazati u ovom poglavlju temelje se na prethodnim istraživanjima u tim znanostima [1].

Temeljni pojmovi koje trebamo definirati i pojasniti su umjetna inteligencija (eng. *Artificial Intelligence - AI*), agenti i okružje (eng*. environment*). Od samog početka postoji rasprava o tome što zapravo jest umjetna inteligencija. Intuitivno znamo da postoji razlika između uobičajenog računalnog programa i nečeg što bi smo nazvali umjetnom inteligencijom, no strogo definirati jasnu granicu može predstavljati izazov. Naročito kad ne postoji univerzalni konsenzus oko jedinstvene definicije. Dane su mnoge definicije u kojima se očituje stav autora o cilju te znanosti i načinima na koje bi ona trebala težiti ispuniti te ciljeve. Stavovi o ciljevima se dijele na ostvarivanje računalnog programa koji emulira ljudsko ponašanje i razmišljanje ili onog koji razmišlja i djeluje racionalno. Kako je sam koncept napredne inteligencije nama poznat samo u ljudima, razumljivo je uspoređivati razvoj umjetne inteligencije s ljudskom, no i taj pristup ima svojih nedostataka. Dio razloga tome je da znanost o ljudskome mozgu i izvoru inteligencije u ljudima još uvijek nema sve odgovore. Tome suprotstavljeni pristup gleda na umjetnu inteligenciju kao razvoj agenata koji racionalno razmišljaju i djeluju. Njega karakterizira definicija Poolea i suradnika koja glasi:

„*Umjetna inteligencija je znanost usredotočena na dizajn računalnih agenata s inteligentnim sposobnostima*.“ [2]

Razmišljati i djelovati racionalno znači težiti djelovanju s najboljim mogućim očekivanim ishodom, s obzirom na raspoloživo znanje [1]. Iz navedene definicije vidimo da nositelj djelovanja jest agent. Prema značenju same riječi agent je svaki entitet koji na neki način djeluje, no u kontekstu umjetne inteligencije agent ima užu definiciju. Radi se o inteligentnom agentu koji percipira svoje okružje, prilagođava se promjenama te zna i postiže svoje ciljeve [1]. Definiraju ga svojstva autonomije, racionalnosti te fleksibilnosti u određenoj okolini. Autonomija agenta podrazumijeva da on samostalno djeluje te upravlja svojim stanjem i postupcima. Racionalnost podrazumijeva da agent djeluje i razmišlja racionalno što smo ranije pojasnili. Fleksibilnost pak podrazumijeva tri svojstva: reaktivnost, proaktivnost i društvenost. Reaktivnost obuhvaća kontinuirano razumijevanje svog okružja te pravovremeno reagiranje na promjene koje se u njemu događaju. Proaktivnost se odnosi na preuzimanje inicijative za vlastito ponašanje te generiranje i ostvarivanje vlastitih ciljeva. Svojstvo društvenosti se odnosi na sposobnost komunikacije i/ili suradnje s drugim agentima ili ljudima [3]. Ako uzmemo u obzir navedene karakteristike agenta kao nositelja inteligencije možemo razabrati između uobičajenog računalnog programa i nečega što nazivamo umjetnom inteligencijom.

Kao što je ranije spomenuto, svaki agent djeluje u pripadnom okruženju. Agent percipira svoje okružje kroz senzore te djeluje na njega kroz aktuatore [1]. Različita okružja mogu predstavljati značajan izazov prilikom implementacije inteligentnog agenta ovisno o njihovoj kompleksnosti, predvidljivosti te mogućnosti modeliranja u simulacijama. To je naročito problem za svakog agenta koji djeluje i u interakciji je sa stvarnim svijetom, jer kao što nam je poznato stvarni svijet nudi gotovo neograničen broj mogućih varijabli koje je moguće razmotriti u gotovo svakom slučaju. Jasno, svakome tko pokušava raditi s gotovo neograničenim brojem varijabli u praksi ne piše se dobro. Iz tog razloga okruženja temelje se na simulacijama ili ograničenim interpretacijama stvarnog svijeta kao reprezentacijom problema. Dizajn i izvedba tih simulacija ili odabir izvora podataka, kao i odabir relevantnih varijabli koje agenti interpretiraju predstavljaju značajan izazov sami za sebe prilikom implementacije nekog agenta. Okružja možemo opisati s obzirom na neka od njihovih ključnih svojstva u odnosu na agenta [1]. Prvo svojstvo jest transparentnost (eng. *observability*). Ono ovisi o agentovoj sposobnosti da u svakom trenutku posjeduje djelomično, potpuno ili čak nikakvo znanje o relevantnim aspektima svog trenutnog okružja. Drugo važno svojstvo jest broj agenata u tom okružju. Agent u okružju može djelovati sam, ili pored drugih agenata. U ovom kontekstu drugim agentima se smatraju svi drugi entitet koji djeluju aktivno s nekim vlastitim ciljem. Ovisno o njihovim ciljevima, oni mogu biti u konkurentskom ili suradničkom odnosu s našim agentom. U okolišu s više agenata otvara se mogućnost međusobne komunikacije no dovodi i element nesigurnosti ako njihove akcije nisu predvidive. Treće svojstvo okružja jest predvidljivost njegovih stanja. Ako agent može predvidjeti iduće stanje na

temelju trenutnog i akcija koje poduzme kažemo da je okruženje determinističko. U suprotnom ono je stohastičko, te mu je inherentno svojstvo nesigurnost. Primjer stohastičkog okruženja je stvarni svijet u čijem slučaju ako i djeluje predvidivo, postoji preveliki broj varijabli za uzeti u obzir pa se smatra stohastičkim. Unutar stohastičkih okruženja također razlikujemo i nedeterministička okruženja unutar kojih su poznati mogući ishodi, no ne i njihova vjerojatnost. Iduće svojstvo koje karakterizira okružje je epizodičnost. U epizodičnom okružju javljaju se diskretne epizode unutar kojih agent na temelju svoje percepcije okružja donosi odluku, te ta odluka nema utjecaja na iduću epizodu. Ako odluke agenta imaju dugoročne posljedice radi se o slijednom okružju (eng. *sequential*). Okružje se također može mijenjati bez utjecaja agenta te u tom slučaju kažemo da je dinamično, a u suprotnom statično. Ovisno o načinu na koji se odvijaju akcije i mijenjaju stanja u okružju, ono može biti diskretno ili kontinuirano. Ako se akcije odvijaju trenutačno i za vrijeme njihovog odvijanja ne mijenja se stanje okružja ono je diskretno. U slučaju da je za vrijeme odvijanja akcija potrebno uzeti u obzir vrijeme koje prolazi i kako ono utječe na stanje okružja, ili same akcije ili odluke ne karakteriziraju diskretne vrijednosti radi se o kontinuiranom okružju. Posljednje svojstvo kojim opisujemo okružje u odnosu na agenta ovisi o znanju agenta o tom okružju. Ako agent poznaje sva pravila koja vrijede u okružju radi se o poznatom (eng. *known*) okružju. U nepoznatom okružju agent mora prvo naučiti pravila koja opisuju i vrijede u njegovom okružju [1].

Nakon što smo pojasniti osnovne pojmove vezane uz razvoj umjetne inteligencije te kakve probleme ta znanost nastoji riješiti, možemo početi razmišljati o različitim pristupima rješavanja tih problema. Pristupe možemo podijeliti na simboličke i statističke [4]. Oba pristupa imaju svoje prednosti i nedostatke, te se u praksi često kombiniraju. Simbolički pristup temelji se na formalnoj logici. Dominirao je područjem u početku razvoja znanosti te su njegove metode i danas u značajnoj primjeni. Pristup se bavi s problemima reprezentacije znanja, rezoniranja, pretraživanja, planiranja i rada s vjerojatnošću u uvjetima nesigurnosti. Statistički pristup postojao je od samog početka razvoja područja no posljednjih godina postignut je značajan napredak u razvoju i primjeni njegovih metoda. Metode statističkog pristupa bave se problemima strojnog učenja i raspoznavanja uzoraka [4]. Aktualni obnovljeni entuzijazam za područje proizlazi iz napretka u strojnom učenju te kontinuiranog unaprjeđenja i rastućom dostupnošću računalnih resursa sve većem broju ljudi. U narednim potpoglavljima objasnit ćemo principe koji stoje iza pojedinih metoda oba pristupa, njihove prednosti i nedostatke te probleme skojima se susreću te kako ih rješavaju.

2.1. Simboličke metode

Simbolički pristup temelji se na razvoju racionalnih agenata koji traže riješenja na dane probleme. Riješenja se temelje na donošenju odluka kako bi se postigao definirani cilj. Ti problemi mogu varirati u složenosti, te se u skladu s time može odabrati pristup pronalaženja rješenja od jednostavnog pretraživanja, do rezoniranja u uvjetima nesigurnosti. Agenti kako bi pronašli riješenje mogu koristiti logiku kako bi rezonirali o najboljim akcijama te o stanjima njihovih okružja. Odluke se temelje na informacijama koje mogu biti pohranjene u bazama znanja ili se mogu temeljiti na procjenama vjerojatnosti u nedeterminističkim okružjima. U sljedećim podpoglavljima opisat ćemo na koje načine agenti donose odluke i riješavaju probleme različite složenosti primjenom koncepata pretraživanja, logičkog zaključivanja, reprezentacije znanja te procjenom vjerojatnosti.

2.1.1. Rješavanje problema

Jedan od ranih pristupa razvoju umjetne inteligencije bazirao se na implementaciji agenata koji su tražili rješenja na dane probleme. Agent mora razmotriti skup mogućih stanja, skup akcija koje može poduzeti i njihovih posljedica te ispuniti dani cilj. Rješenje danog problema zatim se reprezentira kao slijed akcija koje agent mora poduzeti kako bi ostvario svoj cilj [1]. Način na koji on može pronaći najbolje rješenje za ostvarivanje svog cilja svodi se na problem pretraživanja.

Najjednostavniji slučaj takvog problema je kada je okružje poznato (eng. *known*), determinističko, transparentno (eng. *observable*) i diskretno. U tom slučaju možemo definirati svaki problem kao skup od pet komponenti koje su agentu poznate. To su inicijalno stanje okružja u kojem se agent nalazi, skup definicija akcija za svako stanje unutar kojeg ih agent može poduzeti, skup definiranih rezultiranih stanja nakon svake akcije te funkcija korisnosti (eng. *utility function*) koja evaluira konačno stanje. U ovakvom slučaju pretraživanje se svodi na razmatranje svih mogućih puteva, te odabira najboljeg koji vodi do cilja kao rješenje. Za to se mogu koristiti jednostavni algoritmi kao što su BFS (eng. *Breadth First Search*), DFS (eng. *Depth First Search*), UCS (eng. Uniform Cost Search), obostrano pretraživanje (eng. Bidirectional search) i algoritmi koji koriste heurističke pristupe kao što su pohlepno pretraživanje (eng. Greedy best-first search) i A zvijezda (eng. A star, A*) [1]. Algoritmi se razlikuju po redoslijedu prema kojem razmatraju potencijalne puteve te time imaju različite performanse ovisno o promatranom problemu. Heuristički pristupi koriste funkciju kako bi procijenili vrijednost pojedinog koraka pretraživanja, kao što je duljina puta, te odabiru sljedeći korak koji ima najmanju cijenu. Ti algoritmi najčešće postižu najbolje performanse.

Kao što možemo pretpostaviti, većinu problema ne možemo pojednostaviti do te mjere da odgovaraju uvjetima takvog okružja bez da se zanemare njegovi bitni elementi, no takvi slučajevi postoje. Primjeri takvog problema su traženje najkraćeg puta, navigacija web mjestom te neke logističke operacije. Pretraživanjem možemo rješavati i probleme koji ne ispunjavaju sve uvjete takvog okružja.

U slučaju da nam način ostvarivanja rješenja nije bitan, već samo izgled konačnog stanja, možemo koristiti algoritme koji rade lokalno pretraživanje. Takav slučaj može biti optimizacija nekog rasporeda. Tada možemo koristiti algoritme kao što su hill-climbing pretraživanje, simulirano annealiranje, local-beam pretraživanje te genetski algoritam [1]. Ovi algoritmi ne donose uvijek konačno optimalno rješenje, već rade na tome da inkrementalno poboljšavaju i optimiziraju rješenje. Rješenje se zatim evaluira koristeći za to danu funkciju. Hill-climbing pretraživanje uvijek odabire samo akcije koje donose poboljšanje i time se dešava dolaženje do lokalnog maksimuma jer algoritam ne čini privremene kompromise kojima bi kasnije došao do boljeg rješenja. Simulirano anneliranje zato povremeno nasumično odabire akcije koje ne donose poboljšanje kako bi se izbjegao taj problem. Local beam pretraživanje je algoritam sličan BFS algoritmu, no počinje s n nasumično odabranih stanja i pamti samo n najboljih stanja do kojih je moguće doći iz početnih. Genetski algoritam je modifikacija local beam pretraživanja, te također počinje s n nasumično odabranih stanja. Ta stanja zatim procjenjuje na temelju funkcije korisnosti i kreira sljedeća stanja kombinacijom dijelova dvoje odabranih stanja koja su bliže optimalnom rješenju. U novonastalim stanjima zatim postoji šansa da se mutira jedna njihova komponenta. Genetski algoritam time nastoji emulirati koncept selektivne reprodukcije gdje se odabiru najprikladniji roditelji te je njihov produkt podložan mutaciji.

Lokalno pretraživanje možemo koristiti i u slučajevima u kojima je okružje kontinuirano. Jedan od načina na koji možemo rješavati takve probleme je da sami diskretiziramo okružje na diskretna stanja po vlastitom nahođenju ili metodom po izboru te koristimo jedan od ranije spomenutih algoritama [1]. U suprotnom možemo koristiti neke od metoda koje koriste gradijent ciljane funkcije kako bi se približili lokalnom maksimumu kontinuirane funkcije kao Newton-Rhapson metodu za računanje korijena funkcija. U slučaju da je okružje nedeterminističko agent ne može sa sigurnošću predvidjeti posljedice svojih akcija, te mora uzeti u obzir sve moguće posljedice. U tom slučaju agent ne može dati rješenje kao slijed akcija, jer posljedice bilo kojeg slijeda akcija ne moraju uvijek biti iste. Rješenje se tada daje u obliku plana koji opisuje koju akciju treba poduzeti u kojoj situaciji. U slučaju da okružje nije potpuno transparentno, tj. agent ne zna koje je stanje njegovog okružja, on radi na temelju popisa pretpostavljenih mogućih stanja. Zatim inkrementalno gradi moguće rješenje tako da sužava izbor na one akcije koje je moguće poduzeti u svim stanjima u kojima se on potencijalno može nalaziti u tom trenutku. Postoje slučajevi u kojima je okružje nepoznato,

odnosno u kojima agent ne zna koja su moguća stanja okružja niti posljedice svojih akcije. Tada agent ne može izračunati rješenje, već mora direktno istražiti takvo okružje kako bi mogao rješavati problem. Istraživanje problema takve vrste naziva se online istraživanje te obuhvaća pamćenje iskustava kroz istraživanje te korištenje jednog od ranije navedenih algoritama jednom kada agent stvori zadovoljavajuću percepciju o svom okružju [1].

Dosad smo samo razmotrili slučaj u kojem jedan agent sam radi na traženju rješenja problema. Kao što smo ranije spomenuli, okružja mogu biti višeagentna te ti agenti ne moraju surađivati s našim agentom. Štoviše, ako su njihovi ciljevi međusobno protivni oni mogu aktivno sabotirati našeg agenta koji nastoji pronaći rješenje. Primjer takvih okružja nalazi se u brojnim igrama s više igrača kao što su kartaške igre ili šah. U tom slučaju ti suprotstavljeni agenti dovode element nesigurnosti u okružje agenta jer je njihovo ponašanje nepredvidivo. Naš agent onda može postupati na način sličan kao u nedeterminističkom okružju, gdje kreira plan za moguće posljedice svojih akcija ovisno o reakcijama suprotstavljenih agenata. U slučaju da je moguće uzeti u obzir sve moguće poteze agenta i suprotstavljenog agenta može se primijeniti min-max algoritam. Najčešće postoji preveliki broj svih mogućih takvih ishoda da bi agent izračunao optimalnu strategiju u razumnom vremenu. Tada može koristiti algoritme pretraživanje u sklopu kojih određuje koje opcije može zanemariti jer pretpostavlja da neće dovesti do optimalne strategije, kao npr. alpha-beta pruning algoritam. Opcije koje zanemaruje može odrediti tako da ako su u usporedbi s već otkrivenim opcijama inferiorne, zna da ih neće odigrati u korist opcija koje su bolje [1].

Problemi kakve smo dosad razmatrali smatrali su rješenjem postizanje stanja koje se smatra optimalnim prema zadanoj funkciji korisnosti. U tom slučaju vrijednost stanja je predstavlja njegova vrijednost dobivena prema toj funkciji. No stanje možemo detaljnije promatrati kao skup više varijabli. Tada možemo razmišljati o problemima čiji je cilj zadovoljiti određene uvjete za te varijable. Takve probleme nazivamo problemima zadovoljavanja ograničenja (eng. *Constraint Satisfaction Problems* - CSP). Njih definiraju skup varijabli, skup uvjeta koje one moraju zadovoljiti te skup domena vrijednosti koje one mogu poprimiti. Postoje tri glavne vrste metoda koje se koriste za rješavanje takvih problema, propagacija ograničenja (eng. Constraint propagation), backtracking i lokalno pretraživanje [1]. Propagacija ograničenja obuhvaća određivanje ne dozvoljenih vrijednosti za pojedine varijable na temelju zadanih ograničenja. To može dovesti do rješenja samo po sebi ili se može kombinirati s algoritmima pretraživanja. Backtracking algoritam postepeno definira vrijednosti za tražene varijable. Počne od malog broja varijabli kojima dodjeljuje sve moguće kombinacije vrijednosti iz domena. Kad postigne kombinaciju tog malog broja varijabli koja zadovoljava ograničenja doda još jednu varijablu u tu kombinaciju i ponavlja isti princip dodjeljivanja vrijednosti toj varijabli. Algoritmi lokalnog pretraživanja na početku određuju vrijednosti svim varijablama te iterativno mijenjaju te vrijednosti dok ne zadovolje uvjete. U praksi najčešće se primjenjuje

propagacija ograničenja kako bi se smanjila domena mogućih rješenja te neka od metoda pretraživanja kako bi se postiglo konačno rješenje koje zadovoljava ograničenja.

2.1.2. Znanje, rezoniranje i planiranje

U prethodnom poglavlju opisali smo kako agenti mogu rješavati probleme koristeći različite algoritme pretraživanja kako bi pronašli rješenje u različitim okruženjima. Agenti implementirani tim pristupom nisu vrlo fleksibilni po pitanju zadataka koje mogu rješavati. Za rješavanje kompleksnijih zadataka na nešto zanimljiviji način od pretraživanja, agenta možemo unaprijediti da ima tri nove sposobnosti koje ćemo opisati u ovom potpoglavlju. To su upravljanjem bazom znanja koja sadrži informacije o okružju, sposobnost rezoniranja, tj. korištenja logike kako bi se zaključile nove informacije na temelju baze znanja te sposobnost planiranja akcija potrebnih da se ispuni cilj [1].

U bazi znanja agent sprema informacije o svom okružju u njemu razumljivom obliku. Agentovo znanje reprezentiraju činjenice za koje on smatra da su istinite u njegovom okružju. Taj pristup omogućava agentu mnogo fleksibilniji pristup u shvaćanju svog okružja u odnosu na samo zadani broj mogućih stanja. Kako bi agent mogao razumjeti informacije pohranjene u njegovoj bazi znanja, potreban mu je jezik koji definira reprezentaciju tog znanja. Kako će agent koristiti logiku da zaključi nove informacije na temelju onih eksplicitno pohranjenih u bazi znanja te informacije je potrebno pohraniti u obliku s tim u vidu. To znači da te informacije moraju biti precizne i formalno definirane. Najjednostavniji oblik logike koju možemo koristiti je propozicijska logika. Sintaksa propozicijske logike dozvoljava iznošenje tvrdnji ili propozicija koje su istinite ili neistinite. Tvrdnje bilježimo pomoću simbola, kao na primjer T_1. Tvrdnji možemo dodati negaciju (¬) kako bi smo označili da ona nije istinita. Te tvrdnje zatim mogu biti povezane veznicima i (∧), ili (∨), implikacijom (⇒) te ekvivalencijom (⇔) i činiti kompleksnije tvrdnje čija se istinitost dobiva putem pripadnih tablica istinitosti. Primjer takve kompleksne tvrdnje je sljedeći.

$$T_1 \wedge T_2 \Rightarrow T_3$$

On znači da ako su T_1 i T_2 istinite, tada je istinita i T_3, no u kontekstu problema pridaje im se konkretan značaj, kao npr. ako je stol mokar i šalica prazna slijedi da je netko prolio šalicu. Agent koji ima tvrdnje u tom obliku zabilježene u bazi znanja može zaključivati nove tvrdnje koje vrijede u njegovom okružju. Prvi način na koji to može postići je provjeravanjem da li je neka nova tvrdnja istinita u svim mogućim svjetovima u kojima vrijede sve druge tvrdnje u bazi podataka [1]. Mogući svjetovi obuhvaćaju sve moguće kombinacije varijabli koje su dio nekog problema. To je vrlo računalno zahtjevan pristup ovisno o složenosti svijeta, no postoji i drugi pristup. On obuhvaća dokazivanje novih tvrdnji na temelju onih već poznatih. Kompleksnije tvrdnje možemo shvatiti kao pravila na temelju kojih pomoću poznatih jednostavnih tvrdnji

možemo dokazati nove jednostavne tvrdnje. Tako na primjer ako znamo da vrijedi T_1i imamo pravilo $T_1 \wedge T_2$ možemo zaključiti da vrijedi i T_2. Uz čak ovakvu jednostavnu reprezentaciju znanja, agent može donositi informirane odluke te koristiti jedan od algoritma pretraživanja kako bi pronašao slijed akcija koje će ga dovesti do cilja. U praksi je ipak za tu svrhu češće korištena logika prvog reda (LPR) jer omogućuje jednostavniju reprezentaciju složenijih okružja. LPR podržava i definiciju odnosa između entiteta te kvantifikatore svaki (∀), postoji (∃) i jednakost (=). Za implementaciju baze znanja agenta koji rezonira o svom okružju možemo koristiti jezik Prolog. Ime Prolog kombinacija je riječi programiranje i logika iz čega možemo naslutiti da se jezik temelji na logici, konkretno na LPR [5]. Agent koristeći Prolog definira bazu znanja u koju može pohranjivati informacije o objektima, informacije o odnosima tih objekata te pravila. Primjer zapisa u Prologu je sljedeći.

Grad(Zagreb)

Grad(Split)

A1(Zagreb,Split)

U primjeru kao gradovi su definirani Zagreb i Split, te je definirana autocesta A1 koja ih povezuje. Nakon što agent ima takvu bazu u nju može pohranjivati dodatno znanje zaključivanjem na temelju već poznatih informacija ili na temelju novih iskustava s okružjem. Bazi može postavljati pitanja o nekoj tvrdnji vezanoj uz neki objekt te dobiti odgovor da li je tvrdnja istinita ili ne. Takvi agenti mogu se primijeniti za dokazivanje tvrdnji, a ta tvrdnja može biti matematički dokaz ili dokaz da u određenoj situaciji postoji slijed akcija koje vode do ciljanog stanja. Snaga ovakvog pristupa reprezentaciji znanja i rezoniranja o tom znanju je u njegovoj univerzalnosti. Nudi alate za opisivanje i rad s problemima u mnogo složenijim okružjima, te razumno intuitivan način za reprezentaciju takvih problema na agentu razumljiv način.

Ovakav pristup omogućava i rješavanje kompleksnijih problema planiranja. S obzirom na to da su varijable koje opisuju stanja i akcije reprezentirane na isti apstraktni način u sintaksi LPR, razvijen je jezik za opisivanje domene planiranja (eng *Planning Domain Definition Language* - PDDL) kako bi se definirala stanja, akcije i rezultati za specifični za problem [1]. Stanja u PDDL-u se definiraju kao skup tvrdnji koje su podložne promjeni kao posljedica akcija koje poduzima agent. Na taj način definira se inicijalno i ciljano stanje. Akcije se definiraju kroz skup preduvjeta koji moraju biti ispunjeni da bi se ona mogla izvršiti te skup posljedica koje nastaju nakon što se izvrši. Rudimentaran problem pronalaženja puta u svrhu ilustracije definicije stanja i akcija definiranih PDDL-om je prikazan u tablici 1.

Inicijalno stanje:	Lokacija(Zagreb)	
Akcija:	Putuj(polazište, odredište, cesta)	
	Preduvjet:	Lokacija(polazište) ∧ Cesta(polazište,odredište)
	Rezultat:	Lokacija(odredište) ∧ ¬ Lokacija(polazište)

Tablica 1: Primjer definicije domene problema PDDL-om

Za pronalaženje rješenja u toj reprezentaciji okružja razvijeni su brojni pristupi. Prvi koji ćemo spomenuti obuhvaća pretraživanje mogućih puteva od inicijalnog stanja do ciljanog, ili čak u obratnom smjeru. No kako se radi o složenijim problemima s iznimno velikim brojem mogućih stanja, potrebno je razviti heuristično pretraživanje za pojedini slučaj jer pretraživanje koje bi uzelo u obzir sve teoretske mogućnosti vrlo brzo može dovesti do kombinatoričke eksplozije. Te heuristike najčešće obuhvaćaju pojednostavljivanje specifičnog problema zanemarivanjem preduvjeta akcija ili grupiranjem više stanja u manji broj apstraktnih stanja. Korištenjem heurističnog pristupa za specifične probleme postiže se dekompozicija krupnog problema na više manjih rješivih problema, no to dolazi pod cijenu slabije primjenjivosti istog pristupa na druge probleme i zahtjeva pomoć čovjeka kako bi osmislio te heuristike.

Interes za rješavanje sve složenijih problema proizlazi ne samo iz akademske znatiželje, već i iz želje da se sposobnosti agenta koji mogu pamtiti znanje, rezonirati i planirati primjene na probleme iz stvarnog svijeta koji uvijek dolaze uz naizgled nepremostive izazove kao ranije spomenuta kombinatorička eksplozija. Stvarni problemi često dolaze uz okružja koja su nedeterministička što dodatno kompliciraju problem. U stvarnim problemima planiranja često se javljaju ograničenja vezana uz resurse kao što su novac, vrijeme, sastavni dijelovi ili korišteni alati. Neki od tih resursa mogu biti potrošni kao vrijeme, dok se drugi samo koriste na određeni period kao alati. Za rad s takvim resursima često se koristi grupiranje resursa kako bi se pojednostavio problem ako individualni resursi nisu bitno različiti s obzirom na njihovu ulogu u problemu [1]. Za rad s kompleksnijim problemima razvijen je pristup koji organizira planove u hijerarhije. On obuhvaća definiranje slijeda jednostavnih akcija u akcije više razine (eng. *High-level actions*). Te akcije isto imaju preduvjete i posljedice koji proizlaze iz jednostavnih akcija od kojih se sastoje. Tim pristupom s akcijama više razine možemo postupati kao i s jednostavnim akcijama te pojednostaviti postupak planiranja. Ako je cilj ostvariv korištenjem akcija više razine, ostvariv je i korištenjem samo jednostavnih akcija, no proces provjeravanja da li je ostvariv je znatno jednostavniji kada je potrebno raditi s manjim brojem mogućih akcija. Agentu te akcije više razine možemo sami zadati, ili on može pamtiti generalizirane akcije na temelju prethodnih iskustava kroz pretraživanje. Za rješavanje drugih izazova koji proizlaze iz problema temeljenih na stvarnom svijetu razvijeni su drugi principi koji se djelomično temelje na onima korištenim u problemima pretraživanja. U polutransparentnim okružjima agent može kreirati plan za svaki mogući slučaj stanja u kojem se može nalaziti, te

se držati dijelova plana koji odgovaraju novim informacijama o okružju koje dobiva kroz provođenje plana. U nedeterminističkim okružjima koristi se online planiranje. Ono uzima u obzir neplanirane posljedice akcija te prilagođava plan u skladu s njima. Okružje može biti nedeterminističko kao posljedica prisutnosti drugih agenata. U slučaju planiranja, naš agent može dogovoriti suradnju s tim agentima te kroz koordinaciju zajednički izgraditi plan. Koncept suradnje između agenata zanimljiv je sam po sebi jer nosi nove izazove koji proizlaze iz uspostavljanja komunikacije kako bi se razmjenjivale informacije, potencijalno suprotstavljenih ciljeva te pitanja povjerenja i agentove politike prema suradnji.

2.1.3. Nesigurno znanje i rezoniranje

Kao što smo već ranije naveli, u rješavanju stvarnih problema često se susrećemo s nesigurnošću koja dodatno komplicira već kompleksne probleme za naše agente. U stohastičkim, netransparentnim ili višeagentnim okružjima potrebno je voditi računa o svim stanjima za koje agent smatra da su moguća na temelju podataka koje ima. To obuhvaća planiranje za svaki od tih moguća stanja što vrlo lako postaje iznimno zahtjevno za svaki netrivijalni problem. Ako želimo rješavati kompleksne probleme moramo uvesti pojam vjerojatnosti kako bi smo mogli kvantificirati nesigurnost [1]. U određenim problemima za neku tvrdnju ne možemo uvijek znati da li je istinita ili neistinita, ali često možemo procijeniti vjerojatnost da bude jedno ili drugo. Tu vjerojatnost iskazujemo u intervalu od 0 do 1 te ona ovisi o trenutnom najboljem znanju agenta. Pojedine vjerojatnosti mogu biti dio početnog znanja agenta ili se mogu definirati kroz iskustvo. U nesigurnim uvjetima agent više ne mora biti potpuno siguran da će njegov plan dostići cilj kako bi se smatrao racionalnim, samo da je njegov plan najbolji po pitanju koristi za agenta s obzirom na njegovo znanje. Ljudi razmišljaju na sličan način i njihova racionalnost nije upitna ako se određen plan na kraju pokaže neuspješan ako uzevši u obzir njihovo znanje u trenutku odlučivanja je to bila najbolja odluka prema vjerojatnosti za uspjeh. Kako bi agent mogao donositi odluke koje se temelje na vjerojatnostima pojedinih tvrdnji nisu mu dovoljna samo pravila logike jer se ona bave samo zaključivanjem na temelju istinitih ili neistinitih tvrdnji. Njih možemo nadopuniti notacijom i pravilima posuđenih iz teorije vjerojatnosti. Tvrdnje kojima je dodijeljena vjerojatnost zapisujemo kao $P(tvrdnja)$ što označava vjerojatnost da je tvrdnja istinita. Kako bi agent mogao zaključivati na temelju takvih tvrdnja, mora moći raditi računske operacije s vjerojatnostima. Najjednostavnije pravilo koje vrijedi za vjerojatnosti je pravilo produkta koje se odnosi na vjerojatnost da su neke dvije tvrdnje obje točne. Ono tvrdi da je umnožak vjerojatnosti tvrdnja da zasebno budu točne jednaka vjerojatnosti da obje budu točne, što navedenom notacijom zapisujemo na sljedeći način.

$$P(a \wedge b) = P(a) \cdot P(b)$$

Iz teorije vjerojatnosti preuzet je i Bayesov teorem koji omogućava računanje vjerojatnosti da je nova tvrdnja istinita ako je poznata vjerojatnost te i neke druge tvrdnje zasebno te odnos vjerojatnosti između te dvije tvrdnje u obratnom smjeru. On glasi [1]:

$$P(a|b) = \frac{P(b|a)P(a)}{P(b)}$$

Bayesov teorem iskazuje da je vjerojatnost da tvrdnja a bude istinita ako je tvrdnja b istinita jednaka vjerojatnosti da je b istinita ako je a istinita množena s vjerojatnosti da je a istinita podijeljena s vjerojatnosti da je b istinita. Koristeći Bayesov teorem agent može temeljem znanja o vjerojatnosti stanja svog okružja donosit zaključke o vjerojatnostima novih tvrdnji vezanih za svoje okružje. Taj princip je najbolje pokazati na primjeru. Uzmimo agenta koji želi utvrditi vjerojatnost da će na autocesti biti gusti promet ako zna da je trenutno turistička sezona. Tvrdnjama možemo dodijeliti sljedeće nazive:

$P(sezona) = vjerojatnost\ da\ je\ trenutno\ turistička\ sezona$

$P(gužva) = vjerojatnost\ da\ je\ na\ autocesti\ gusti\ promet$

Ako agent zna da je vjerojatnost da je trenutno turistička sezona ako je na autocesti gust promet jednaka $P(gužva|sezona) = 0.6$, da je vjerojatnost da je trenutno turistička sezona $P(sezona) = 0.25$, te da je vjerojatnost gustog prometa na autocesti jednaka $P(gužva) = 0.2$ on može izračunati vjerojatnost gustog prometa za vrijeme turističke sezone, te ona za naš primjer iznosi:

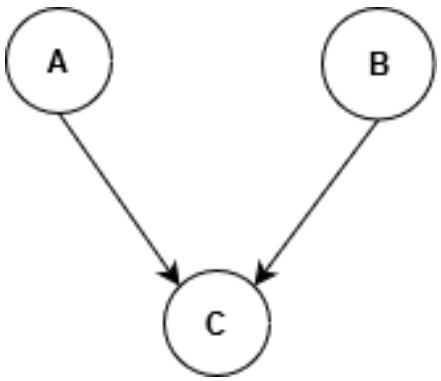
$$P(sezona|gužva) = \frac{P(gužva|sezona) \cdot P(sezona)}{P(gužva)} = \frac{0.6 \cdot 0.25}{0.2} = 0.75$$

Kroz ovakve izračune vjerojatnosti agent može donositi razumne pretpostavke vezane uz stanje svog okružja te posljedica svojih akcija. U ovom slučaju korištene su varijable čija vrijednost može biti istinita ili lažna, no pojedine varijable mogu poprimiti više diskretnih ili čak kontinuiranih vrijednosti. Vrijednosti vjerojatnosti za diskretne varijable možemo zapisivati kao vektore duljine broja mogućih diskretnih vrijednosti. Svaka vrijednost u vektoru tada predstavlja vjerojatnost jedne od moguće diskretne vrijednosti uz predodređeni redoslijed. U slučaju kontinuiranih varijabli koristi se funkcija koja definira distribuciju vjerojatnosti da varijabla poprimi neku od vrijednosti iz domene [1]. Kad uzimamo u obzir više varijabli i njihove kombinirane vrijednosti, te vjerojatnosti možemo prikazivati u tablici. Svaka dodatna varijabla dijeli tablicu na dva djela, dio u kojem je ona istinita te dio u kojem je neistinita. Svaka ćelija u toj tablici sadrži vrijednost vjerojatnosti kombinacije tvrdnji u stupcima i retku. Takvu tablicu nazivamo full joint distribucijom.

Kako smo ranije spomenuli, agent u uvjetima nesigurnosti uzima u obzir sva moguća stanja u kojima se nalazi ili se može nalaziti kao posljedica svojih akcija. Ta moguća stanja su iskazana u modelu mogućih svjetova. Ranije su se ti mogući svjetovi mogli definirati pomoću

tablica istinitosti kombinacijom svih razmatranih varijabli, no kako se u uvjetima nesigurnosti radi o vjerojatnostima potreban je drugi pristup. Za definiciju modela mogućih svjetova koristi se model vjerojatnost. On se može sveobuhvatno definirati full joint distribucijskom tablicom, no kako veličina tablice eksponencijalno raste dodavanjem varijabli u praksi postaje vrlo zahtjevno rješavati probleme korištenjem te tablice kao baze znanja. Ipak, kako takva tablica pohranjuje vrijednosti za kombinaciju svih varijabli u okružju, a možemo za neke od njih tvrditi da su neovisne ili međusobno uvjetno neovisne, tablicu možemo razdvojiti na više manjih tablica s kojima je lakše raditi. Ako su dvije varijable uvjetno neovisne, znači da iako možda obje ovise o nekoj istoj trećoj varijabli, ne moraju ovisiti međusobno jedna o drugoj. Ako pak se radi o jako velikom broju varijabli, ni taj pristup neće dovoljno smanjiti kompleksnost tablica kako bi se s njima moglo raditi u praksi. U tom slučaju može se koristiti naivni Bayes model [1]. On definira da su sve varijable posljedica međusobno uvjetno neovisne što značajno smanjuje kompleksnost vjerojatnosnih tablica, no njegov naziv kao naivni model naslućuje na njegovu temeljnu slabost. Većina problema ne može se pojednostaviti do te mjere bez da se izgubi dio bitnih veza između varijabla, no postoje slučajevi gdje taj princip donosi dostatne rezultate u rješavanju problema.

Kako je rad s tablicama full joint distribucije najčešće pretjerano zahtjevan po pitanju računalnih resursa, iako je najprecizniji pristup, možemo odabrati drugi pristup za prikaz ovisnosti varijabli u okružju. Bayesove mreže su aciklički usmjereni grafovi u kojima svaki vrh sadrži informacije vezane uz jednu varijablu [1]. Te informacije se prikazuju u obliku tablice koja sadrži vjerojatnosnu distribuciju. Ta distribucija određuje utjecaje drugih varijabla na promatranu i iskazuje se vrijednostima vjerojatnosti da je promatrana varijabla istinita ili lažna u različitim slučajevima kombinacija varijabli koje na nju utječu. Utjecaji među varijablama se označavaju strelicama koje povezuju točke u grafu. U ovakvom modelu najveća tablica pripada onoj varijabli na koju utječe najveći broj drugih varijabli, no i takva tablica je u većini slučajeva neusporedivo manja od tablice full join distribucije za isti problem. Prikaz jednostavnog primjera Bayesove mreže s pripadnom tablicom za jednu od varijabla možemo vidjeti na slici 1.



A	B	P(C)
T	T	0.25
T	F	0.75
F	T	0.15
F	F	0.1

Slika 1: Jednostavan primjer Bayesove mreže s pripadnom tablicom uvjetovane vjerojatnosti

Agent koji zaključuje na temelju vjerojatnosti dohvaća vrijednosti iz takvih tablica kako bi ustanovio vjerojatnost da vrijedi tvrdnja C ako zna vrijednosti varijabla A i B. Taj pristup može se proširiti i na složenije kombinacije varijabli, no izvršavanje takvih upita i dalje može predstavljati izazov u vrlo kompleksnim problemima. U tom slučaju mogu se koristiti algoritmi procjenjivanja koji daju približan odgovor kada precizno računanje vrijednosti nije realistično. Za to se koriste Monte Carlo algoritmi koji na temelju nasumičnih uzoraka iz Bayesove mreže procjenjuju traženu vrijednost [1]. Ti algoritmi razlikuju se prema pristupu odabira tih uzoraka te se po dobrim preformansama na velikim mrežama izdvajaju Markov Chain Monte Carlo algoritam te algoritam vaganja vjerojatnosti (eng. *Likelihood weighing*).

Kako bi smo agentu u uvjetima nesigurnosti olakšali rad s mogućim stanjima, možemo uvesti model prijelaza stanja (eng. *transition model*) te model senzora. Model prijelaza stanja opisuje moguće načine prijelaza iz trenutnog stanja u sljedeće. Agent može primijeniti model prijelaza stanja kako bi odredio koje je moguće buduće stanje te odabrao koje iduće stanje je izgledno na temelju informacija prikupljenih iz okružja. Model senzora definira vjerojatnost vrijednosti tih informacija u idućem stanju. Takvi modeli mogu postati iznimno složeni s obzirom na broj prethodnih stanja koje treba uzeti u obzir kako bi se izračunalo sljedeće moguće stanje. U tom slučaju radi se Markovljeva pretpostavka (eng. *Markov assumption*) koja definira da trenutno stanje ovisi samo o određenom broju prethodnih stanja. Poštujući tu pretpostavku dobiva se lanac prethodnih stanja o kojima ovisi sljedeće stanje, te ga se naziva Markovljevim lancem [1]. Svaku kariku u lancu čini stanje okružja koje predstavlja jedan trenutak u vremenu. Taj lanac može se i skratiti do točke da samo buduće stanje može predvidjeti na temelju samo trenutnog stanja. U slučaju skrivenih Markovljevih modela (eng. hidden Markov model – HMM) stanje predstavlja samo jedna varijabla koja se dobiva na temelju jedne ili više varijabli koje opisuju stanje okružja. Ta varijabla poprima diskretnu vrijednost za svako moguće stanje. HMM se može prikazati matricom dimenzija $S \, x \, S$ u kojoj je S broj mogućih stanja [1]. Osim HMM kao temporalni modeli mogu se koristiti i dinamične Bayesove mreže te Kalman filteri. Agent korištenjem jednog od navedenog modela procjenjuje trenutno stanje na temelju prethodnog, te predviđa sljedeće stanje na temelju poznatih ili procijenjenih vjerojatnosti. Korištenjem tih modela moguće je procjenjivati stanja više vremenskih koraka unaprijed te unatrag.

Agent nakon što ima sposobnost predviđanja stanja u nesigurnosti te odlučivanja na temelju vjerojatnosti, može donositi odluke s ciljem postizanjem stanja s najvećom korisnosti. Korisnost stanja je vrijednost koja se računa koristeći funkciju korisnosti koja je specifična za pojedini problem. Odluke koje agent može donositi kako bi maksimizirao korisnost možemo prikazati mrežom odlučivanja (eng. *decision network*). One su usmjereni grafovi u kojima jedna točka predstavlja jednu od varijabla (eng. *chance node*), odluku agenta ili funkciju korisnosti. Sve varijable koje utječu na korisnost stanja povezane su s funkcijom korisnosti. Agent u takvoj

mreži može izračunati očekivanu korisnost za svaku moguću kombinaciju odluka, te odabrati one koje imaju najveću korisnost.

2.2. Statističke metode

Prethodno opisane metode za razvoj umjetne temeljile su se na logici, algoritmima pretraživanja, teoriji korisnosti te teoriji vjerojatnosti. Dok teorija vjerojatnosti predstavlja uporište i za polje statistike, u ovom poglavlju opisat ćemo metode koje su dobile na značaju nakon što je prihvaćeno da statistika može značajno doprinijeti razvoju umjetne inteligencije krajem osamdesetih godina 20. stoljeća [1]. Značaj statistike za polje razvoja umjetne inteligencije donosi značajan napredak akumuliran kroz stoljeća ljudskog razvoja na metodama interpretacije, klasifikacije, analize i organizacije podataka. Kako agent koji želi rješavati kompleksne probleme usporedive s onima koje ljudi mogu rješavati mora uzeti u obzir ponekad goleme količine podataka možemo uvidjeti kako statističke metode mogu pomoći u rješavanju tih problema. Posebno značajan napredak donio je razvoj metoda strojnog učenja koje ne samo da omogućavaju rad s golemim količinama podataka već dostižu bolje rezultate što više podataka je dostupno. Uz pojavu eksponencijalnog rasta skupova podataka (eng. *Big Data*) koja se javila kao posljedica informatizacije većine industrija, postali su dostupni golemi skupovi podataka na kojima je moguće primjenjivati metode strojnog učenja. Iz tog razloga postoji značajan suvremeni interes za primjenjivanje metoda umjetnih inteligencija za rješavanje problema kakvi ranije nisu bili učinkovito ili isplativo rješivi primjenom klasičnih metoda. Strojno učenje temelji se na razvoju ili „učenju“ funkcije koja za određene ulazne podatke daje traženi rezultat. U narednim poglavljima opisat ćemo tri pristupa strojnom učenju ovisno o vrsti podataka koji se koriste za učenje te pripadne statističke metode koje oni obuhvaćaju. Značajan napredak za strojno učenje donijela je uspješna primjena umjetnih neuronskih mreža (eng. *Artificial Neural Networks*). Učenje koje uključuje neuronske mreže naziva se duboko učenje (eng. *Deep Learning*). One se mogu primjenjivati u sva tri pristupa strojnom učenju te ćemo ih objasniti u zasebnom poglavlju.

2.2.1. Nadzirano učenje

Nadzirano učenje (eng. *Supervised learning*) je oblik strojnog učenja u kojem je zadatak naučiti funkciju koja na temelju primjera ulaznih podataka (eng. *input)* i izlaznih podataka (eng. *output*) može za nove ulazne podatke pružiti odgovarajuće izlazne podatke. Podatke na kojima se uči funkcija čine parovi ulaza i odgovarajućih izlaza te ih možemo nazvati primjerima. Cilj je razviti funkciju koja na temelju samo ulaznih podataka može odrediti

odgovarajuće izlazne podatke. Matematički taj opis podataka možemo prikazati sljedećom notacijom [1]. Neka je sljedeći niz primjer skupa podataka na kojem se uči:

$$(x_1, y_1), (x_1, y_1), \dots (x_N, y_N)$$

Prvi član para u nizu x_i predstavlja instancu ulaznog podatka, dok y_i predstavlja pripadan izlazni podatak. Funkciju čiji je rezultat y_i za neki x_i možemo na jednostavan način zapisati sljedećim.

$$y = f(\mathrm{x})$$

Kako nam funkcija f nije poznata niti ju je moguće matematički izračunati cilj je pronaći funkciju h koja aproksimira stvarnu funkciju f. Ovaj pristup ovisi o kvaliteti skupa ulaznih podataka na kojem se uči jer vrijedi GIGO (eng. *Garbage In, Garbage out*) princip. Kvalitetan skup podataka mora imati pravilno i istinito označene parove ulaznih i izlaznih podataka, te dovoljan broj različitih primjera za razinu generalizacije koju se želi ostvariti. Ovaj princip strojnog učenja koristiti se u slučajevima u kojima je moguće prikupiti odgovarajući skup podataka s označenim ulaznim i izlaznim parovima. Primjeri praktičnih problema koji se mogu rješavati korištenjem nadziranog učenja su prepoznavanje teksta, jezika ili čak govora, prepoznavanje na temelju slika, te rješavanje brojnih problema za specifične domene poslovanja, financija, klasifikacije, rangiranja informacija i ostalih problema prepoznavanja uzoraka.

Ovisno o vrijednostima funkcije koju aproksimiramo metodom nadziranog učenja, probleme možemo podijeliti na probleme klasifikacije i probleme regresije. U slučaju da se radi o diskretnim izlaznim vrijednostima y koje ne moraju biti brojevi, govorimo o problemu klasifikacije. Primjer klasifikacijskog problema je prepoznavanje teksta gdje je rezultat funkcije jedno od slova dane abecede ili znamenka. Ako je funkcija kontinuirana radi se o problemu regresije. U tim problemima rezultat je neki broj, kao na primjer u problemu predviđanja cijena. U slučaju regresije nastoji se pronaći funkcija koja daje približnu ili prosječnu vrijednost y [1].

Najjednostavniji slučaj regresije je linearna regresija u kojoj se aproksimira linearna funkcija koja predviđa izlaznu vrijednost y. Tu funkciju zapisujemo na sljedeći način.

$$h(x) = w_1 x + w_0$$

Kako bi se dobila funkcija h potrebno je izračunati vrijednosti težina w_0 i w_1. U slučaju više ulaznih varijabli, jednadžba je suma svih vrijednosti x množenih s pripadnom težinom. Kako bi se izračunale vrijednosti težina, definira se funkcija gubitka (eng. *Loss function*) koju se potom nastoji minimizirati. Postoji više načina za definiranje funkcije gubitka, no najčešći je kao suma kvadrata razlike ciljane y vrijednosti i vrijednosti koju predviđa funkcija h za svaki primjer unutar skupa podataka. Takvu funkciju gubitka možemo zapisati na sljedeći način [1].

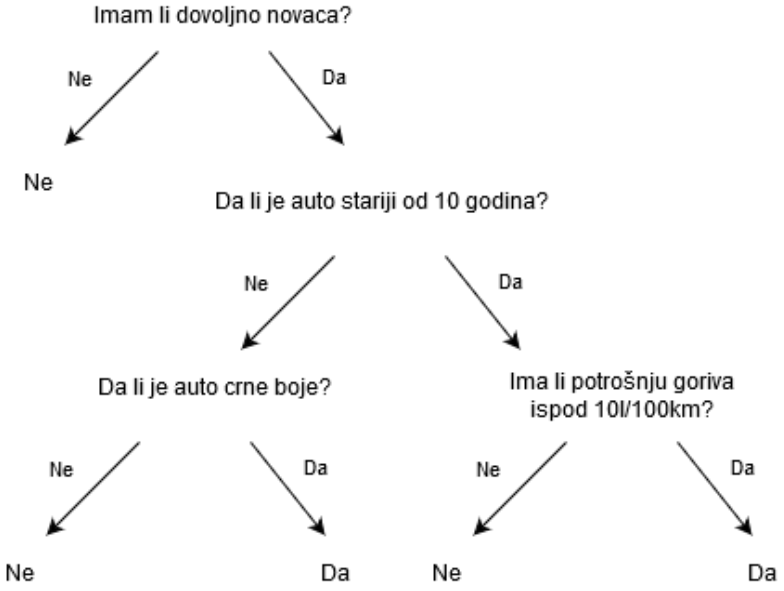
$$Loss(h) = \sum_{j=1}^{N} (y_j - h(x_j))^2$$

Kako bi smo minimizirali funkciju gubitka, potrebno je pronaći vrijednost težina w s kojima je rezultat funkcije što bliži nuli, jer je time funkcija h bliža stvarnoj funkciji f. Za minimizaciju funkcije gubitka može se koristiti neki od algoritama optimizacije, no najčešće korišteni je algoritam spuštanja po gradijentu (eng. *gradient descent*). Princip tog algoritma je postepeno prilagođavanje vrijednost težina za određeni iznos koji nazivamo brzinom učenja. Za svaki primjer iz ulaznog skupa podataka, optimizira se funkcija gubitka prema sljedećem principu gdje je j broj primjera a i broj ulazne varijable [1].

$$w_i \leftarrow w_i + \alpha \sum_j x_{j,i}(y_j - h(x_j))$$

Nakon optimizacije uvrštavanjem vrijednosti težina w u funkciju h možemo računati vrijednosti koje predstavljaju rješenje. Linearnom regresijom možemo rješavati i probleme klasifikacije tako da računamo funkciju h na isti princip, ali odredimo intervale izračunatih vrijednosti koji onda predstavljaju klase.

Jedan od pristupa koji se može primijeniti u obje vrste problema klasifikacije i regresije je učenje stabla odlučivanja (eng. *Decision tree learning*) [6]. Taj pristup obuhvaća razvoj stabla odlučivanja koje možemo prikazati kao usmjereni graf u kojem točke predstavljaju jedno pitanje ili test na neku od ulaznih varijabla. To pitanje može ispitivati da li je promatrana varijabla istinita, ili u slučaju kontinuiranih vrijednosti da li je manja ili veća od neke definirane granice. Veze u grafu predstavljaju moguće odgovore na to pitanje te usmjeravaju na sljedeće pitanje ili na konačan rezultat. Jednostavno stablo odlučivanja možemo prikazati na primjeru. Uzmimo pojednostavljeni problem odlučivanja o kupovini auta. Ulazne varijable x možemo zapisati kao vektor koji sadrži četiri vrijednosti oblika $(cijena, boja, starost, potrošnja\ goriva)$. Primjer jednostavnog stabla odlučivanja za taj problem prikazan je na slici 2.



Slika 2: primjer stabla odlučivanja za pitanje o kupovini auta

Cilj pristupa učenja stabla odlučivanja je razviti takvo stablo odlučivanja koje uspješno predviđa y odnosno odluku za dane ulazne varijable na temelju primjera parova ulaznih i izlaznih varijabli. Glavni problem kod razvoja stabla odlučivanja koje pouzdano daje točne rezultate je njegova veličina. Nastoji se razviti što manje stablo koje daje što pouzdanije rezultate. To se može postići primjenom algoritma koji na temelju vrijednosti varijabla u dobivenom skupu podataka, nastoji rangirati utjecaj pojedine varijable na konačni rezultat, te varijablu s najvećim utjecajem ispitati prvu. Utjecaj pojedine varijable na konačni rezultat procjenjuje na temelju koliko precizno ona sama može predvidjeti ishod. U procesu pojednostavljivanja stabla odlučivanja, neke varijable mogu se eliminirati ako se pokaže da svaki njihov ishod dovodi do jednake vjerojatnosti za svaki mogući odgovor. Ovakav pristup pruža jednostavan način za klasifikaciju brojnih problema, no uz njega se može javiti problem pretjeranog poklapanja (eng. *overfitting*) [6]. Taj problem se javlja kada dano stablo odlučivanja odlično predviđa rezultat na skupu podataka na kojem je razvijeno, no nepouzdano predviđa nove primjere. To je posljedica detekcije uzoraka koji ne postoje u općem problemu, no javljaju se unutar danog skupa podataka. Isti problem može se javiti primjenom drugih metoda učenja te je iz tog razloga dobar odabir skupa podataka za učenje ključan za razvoj primjenjivog modela odlučivanja.

U problemima klasifikacije jedan od principa koji često postiže dobre rezultate je korištenje naivnog Bayes modela (eng. *Naive Bayes*) kojeg smo diskutirali u poglavlju 2.1.3. U kontekstu problema klasifikacije, može ga se koristiti kako bi se odredila vjerojatnost da neka ulazna varijabla predviđa određenu klasifikaciju [7]. Za to se koristi Bayesov teorem, kojeg u ovom kontekstu možemo interpretirati na sljedeći način.

$$P(c|x) = \frac{P(x|c)P(c)}{P(x)}$$

Vjerojatnost da x pripada klasi c računa se kao umnožak vjerojatnosti da vrijedi x ako pripada klasi c i vjerojatnosti da vrijedi c, podijeljeno s vjerojatnosti da vrijedi x. Vjerojatnosti ulaznih varijabli računaju se dijeljenjem ukupnog broja varijabli s brojem slučaja u kojima je ta varijabla istinita. Na isti princip računaju se i vrijednosti $P(x|c)$ na temelju skupa ulaznih podataka. Iako se ovom metodom radi pretpostavka da ulazne varijable ne ovise jedna o drugoj, metoda u praksi često donosi vrlo dobre rezultate te ju je moguće koristiti na vrlo velikim skupovima podataka [8].

Osim opisanih metoda i algoritama za razvoj modela učenja, mogu se koristiti i neparametarski modeli koji koriste algoritme k-najbližeg susjeda (eng. *k-nearest neighbor, KNN*) te algoritme koje koriste Support Vector Machines (SVM). Neparametarski model učenja nemaju definiran broj parametara koji karakteriziraju funkciju, već koriste podatke iz danih primjera kako bi odredili rezultat[1]. KNN algoritam može se koristiti za probleme klasifikacije. On procjenjuje sličnost dane vrijednosti s k najbližih poznatih vrijednosti, te na temelju klasifikacije najsličnijih vrijednosti dodjeljuje klasifikaciju danoj vrijednosti. Linearni SVM

algoritmi kreiraju višedimenzionalne granice između klasa podataka na temelju poznatih primjera, te klasificiraju nove primjere prema granicama klasa [1].

Odabir prave metode nadgledanog učenja za dani problem ovisi o mnogo faktora. Nema algoritma koji za svaki problem daje najbolje rješenje, te je potrebno uzeti u obzir veličinu skupa podataka, raspoložive računalne resurse, svojstvo linearnosti podataka te da li su ulazni i izlazni podaci diskretni ili kontinuirani. Odabir algoritma kao i ulaznog skupa podataka često se svodi na vaganje prednosti i nedostataka za pojedini slučaj problema.

2.2.2. Nenadzirano učenje

Nenadzirano učenje (eng. *Unsupervised learning*) je vrsta strojnog učenja u kojem ulazni podaci nisu označeni kao što su u slučaju nadgledanog učenja. Cilj nenadziranog učenja je otkriti uzorke u danim podacima. To se može postići modeliranjem latentnih varijabli na temelju eksplicitnih varijabli unutar danog skupa podataka raznim algoritmima klasteriranja koji grupiraju dane podatke koji su međusobno slični ili određuju hijerarhiju unutar njih. Nenadgledano učenje može se koristiti i za otkrivanje anomalija unutar skupa podataka. Dok se modeli dobiveni nadziranim učenjem koriste za predviđanje funkcije koja daje rezultate za nove ulazne podatke, nenadzirano učenje se koristi za dobivanje novih informacija iz danog skupa podatka koji se koristi za učenje. U slučaju problema s velikim brojem različitih varijabli koje mogu poprimiti različite vrijednosti, možemo primijeniti HMM kako bi smo okružja s više podataka prikazali pomoću jedne varijable.

Latentne varijable su one koje nisu eksplicitno dane u skupu podataka, ali se mogu odrediti na temelju tih podataka. Primjer takve varijable je vjerojatnost pripadnosti grupama koje čine dane varijable. Ako na dane podatke gledamo kao distribuciju, ona ima više komponenata te pripadnost nekoj od komponenata označava pripadnost klasi. Te komponente su i same distribucije. Cilj je definirati vjerojatnost za pojedinu varijablu da pripada nekoj od tih komponenata tj. klastera. Opći pristup računanju tih vjerojatnosti dan je Expectation-Maximization (EM) algoritmom. Algoritam se sastoji od E i M koraka. U E koraku računaju se vjerojatnosti za svaku varijablu x_j da pripadaju komponenti C_i, te se time određuje koja varijabla pripada kojoj komponenti. Tu vjerojatnost iskazujemo na sljedeći način, te ju možemo računati Bayesovim teoremom [1].

$$p_{ij} = P(C = i|x_j)$$

M korak obuhvaća računanje novih prosjeka vrijednosti, kovarijance te težine za pojedine komponente koje su nastale određivanjem pripadnosti varijabla komponentama u prethodnom koraku. Na temelju tih vrijednosti računa se logaritam vjerojatnosti da je distribucija danih podataka nastala distribucijom izračunatom na temelju tih podataka.

Algoritmi klasteriranja kreiraju i grupiraju podatke u klase ne temelju uzoraka u danim podacima. Postoji više takvih algoritama od koji možemo izdvojiti K-means, DBSCAN te algoritme za hijerarhijsku analizu klasteriranja (eng. Hierarchical Cluster Analysis - HCA). K-means algoritam za klasteriranje grupira dane podatke u zadan k broj klasa koji koristi sličan pristupu kao i EM algoritam. Svaka klasa ima svoje teoretsko središte koje nazivamo centroid [9]. Na početku algoritma nasumično odabere varijable koje će biti centroidi te odabir centroida optimizira kroz iteraciju. Cilj k-means algoritma je odrediti centroid za svaku klasu koji ima što kraću udaljenost od svih članova koji joj pripadaju. Udaljenost predstavlja varijancu između varijable i centroida te se računa kao kvadrat razlike između pojedine vrijednosti te vrijednosti centroida. Funkciju D za računanje udaljenosti možemo matematički zapisati na sljedeći način, gdje je x_i jedna od varijabla koja pripada klasi k a μ_k je vrijednost centroida.

$$D = (x_i - \mu_k)^2$$

Iteracija algoritma je relativno jednostavna. Nakon što su dodijeljeni početni centroidi, računa se udaljenost svake varijable x do svakog centroida μ. Varijabla x pripada klasteru s centroidom do kojeg ima najmanju udaljenost. Nakon što su određene pripadnosti varijabla klasama, odabire se novi centroid koji ima najkraću sumu udaljenosti do svih pripadnih varijabla. Nakon što se kroz naizmjenično reklasificiranje varijabli te odabir centroida više ne mijenja izbor centroida, algoritam je završen [9]. Izazov u korištenju k-means klasteriranja u slučajevima gdje broj klasa nije unaprijed definiran problemom je odrediti optimalan broj klasa, te se iz tog razloga algoritam može više puta provesti kako bi se na temelju rezultata mogle evaluirati rezultirane klase.

DBSCAN (eng. *Density-based spatial clustering of applications with noise*) algoritam također služi za klasteriranje podataka na temelju udaljenosti, no u njemu nije potrebno definirati broj klasa. Udaljenost se računa na isti način kao i u k-means algoritmu. Umjesto broja klasa, definira se vrijednost epsilon (ε) koja predstavlja najmanju dozvoljenu udaljenost između dvije vrijednosti unutar klastera, te vrijednost $minPts$ koja definira minimalni broj susjednih vrijednosti koje čine klaster. Vrijednosti unutar klastera prema tome mogu biti središnje, granične ili ne pripadati ni jednom klasteru. Središnje vrijednosti su one koje imaju $minPts$ susjeda unutar ε udaljenosti s kojima čine dio istog klastera. Granične vrijednosti su one koje su manje od ε udaljene od neke središnje vrijednosti, no imaju manje od $minPts$ susjeda [10]. Algoritam počinje nasumičnim odabirom neke od vrijednosti, te ako ta vrijednost ima $minPts$ susjeda označava se kao središnja vrijednost s kojom počinje formacija klastera. Zatim se provjeravaju sve susjedne vrijednosti te ako i one zadovoljavaju uvjete također se označavaju kao središnje vrijednosti unutar istog klastera. Nakon što se dođe do graničnih vrijednosti i više nije moguće pronaći središnju vrijednost koja pripada istom klasteru, završava se pretraživanje tog klastera. Nakon toga ponovno se odabire jedna od preostalih neodređenih vrijednosti te se ponavlja postupak. Ako se odabere vrijednost koja nema susjede udaljene

manje od ε nju se označava kao šum u podacima, te se odabire druga početna varijabla. Algoritam se nastavlja dok nisu klasificirane sve vrijednosti [10].

HCA algoritmi određuju klastere unutar hijerarhije. To znači da se klasteri viših razina sastoje od pripadnih klastera nižih razina. Hijerarhiju klastera možemo prikazati dendogramom. Ovim pristupom klasteriranju ne moramo unaprijed odrediti broj klasa, već samo odabrati razinu s odgovarajućim brojem klasa nakon što je klasteriranje izvršeno. Postoje dva temeljna pristupa hijerarhijskom klasteriranju ovisno o smjeru grupiranja. Prvi pristup naziva se aglomerativno klasteriranje te obuhvaća grupiranje više klastera nižih razina u klastere viših razina. Niži klasteri koji tvore klaster više razine ne moraju biti međusobno istih razina, samo niže razine od onog kojeg tvore. Obratni smjer kreiranja hijerarhije kroz razdvajanje klastera viših razina na klastere nižih razina zove se divizivno klasteriranje [11]. Aglomerativno klasteriranje se radi tako da se odaberu dvije vrijednosti koje su najbliže. Udaljenost najčešće računamo kao i u prethodnim primjerima, no može se odabrati i neka druga funkcija. Nakon što su odabrane dvije najsličnije vrijednosti one se grupiraju u jedan klaster najniže razine. Nakon toga ponovno se računa udaljenost između svih vrijednosti, te ako se neka od vrijednosti može pridružiti vrijednostima u postojećem klasteru, oni zajedno čine kaster više razine. U suprotnom, ta vrijednost čini zaseban klaster iste razine. Taj postupak se ponavlja dok sve vrijednosti ne čine jedan klaster najviše razine [11]. Divizivno klasteriranje započinje od klastera najviše razine koji obuhvaća cijeli skup podataka. Taj skup se zatim dijeli na dva manja klastera koji se zatim evaluiraju kako bi se odredila njihova razina u hijerarhiji. Taj postupak se ponavlja do kad svaka vrijednost ne predstavlja vlastiti klaster. Postoji više metoda za razdvajanje klastera, od odabira dvije najudaljenije varijable kao početke novih klastera do korištenja k-means algoritma za kreiranje dva nova klastera. Razinu novih klastera možemo najjednostavnije odrediti pomoću redoslijeda kojim su nastali, ili računanjem najveće udaljenosti između dvije varijable unutar klastera te dodjeljivanjem ranga ovisno o toj udaljenosti [12]. Divizivno klasteriranje se rjeđe koristi u praksi zbog slabije preciznosti i učinkovitosti u odnosu na aglomerativno klasteriranje.

Osim otkrivanja latentnih varijabli i klasteriranja, nenadgledano učenje obuhvaća i otkrivanje anomalija u podacima. Anomalije su podaci koji na neki način odudaraju od ostatka podataka, te mogu ukazivati na neku zanimljivu pojavu. Detekcija anomalija nalazi veliku primjernu u sigurnosnim sustavima i analizi transakcija gdje anomalije mogu upućivati na sumnjivu aktivnost. Jedan princip izrade modela za detekciju anomalija obuhvaća izradu modela na testnim podacima koji će predstavljati „normalne“ podatke. Za izradu takvog modela možemo koristiti ranije spomenut DBSCAN algoritam. Nakon što su na testnim podacima uspostavljeni klasteri, možemo provjeravati udaljenost vrijednosti novih podataka od središnjih vrijednosti definiranih klastera. Ako ta vrijednost prelazi određeni prag, možemo ju klasificirati kao anomaliju [13]. LOF algoritam (eng. *Local Outlier Factor*) također se koristi za detekciju

anomalija. Na temelju gustoće podataka računa se faktor koji ukazuje na vjerojatnost da je neka varijabla anomalija. Algoritam započinje računanjem udaljenosti svakog podatka od njegovog k-tog najbližeg susjeda, gdje je k moguće zadati. Ta udaljenost predstavlja gornju granicu udaljenosti unutar koje se podatak smatra unutar dometa (eng. *Reachability distance*). Zatim se računa gustoća lokalnog dometa (eng. *Local reachability density - LDR*) za pojedinu varijablu prema sljedećoj formuli.

$$lrd(a) = \frac{1}{\frac{\sum reach\_dist(a,n)}{k}}$$

LRD je inverz količnika sume dometa vrijednosti a i susjeda n te odabranog broja k. LOF se zatim računa kao prosjek sume LRD-a svojih k susjeda. LOF vrijednost manja od 1 označava da se varijabla nalazi blizu svojih susjeda, dok LOF vrijednost veća od jedan označava da je varijabla značajno udaljena te predstavlja anomaliju [14].

2.2.3. Pojačano učenje

Pojačano učenje (eng. *Reinforcement learning - RL*) razlikuje se od prethodnih pristupa po tome da ne koristi unaprijed pripremljeni skup podataka. Kod RL učenje se odvija paralelno uz interakciju s okružjem. Agent koji uči principom RL na početku ne zna gotovo ništa o svom okružju, već uči kroz iskustvo. Cilj RL je na temelju prethodnih iskustava odabira akcija te njihovih posljedica, naučiti odabirati prave akcije u pojedinim situacijama koje pruže najbolje posljedice. Kako bi agent mogao razlikovati željene posljedice od neželjenih, definira se funkcija nagrade [1]. RL koristi se za rješavanje problema vožnje auta, upravljanja robotima, igranja igra, apstrakcije velikih tekstova te u brojnim drugim primjenama gdje je moguće dobiti povratnu informaciju o donesenoj odluci kako bi se na tom iskustvu učilo.

Problem RL možemo opisati modelom Markovljevog procesa odlučivanja (eng. Markov decision process - MDP). MDP služi za rješavanje problema slijedog odlučivanja u transparentnom, stohastičkom okružju. Modelom se opisuje skup mogućih stanja s definiranim inicijalnim stanjem, skup akcija za svako stanje i Markovljev model prijelaza stanja koji određuje vjerojatnosti ishoda akcija te funkciju nagrade [1]. Razlikujemo dvije vrste agenta koji rješavaju probleme RL. Refleksni agenti uče politiku koja za svako stanje definira koju akciju treba poduzeti. Učenje takvih agenata nazivamo pasivno RL gdje je cilj naučiti koja je optimalna politika, te djelovati po toj fiksnoj politici. Agenti koji uče funkciju korisnosti (eng. *utility-based agents*) za dano stanje definiraju modele stanja okružja te se akcija odabire na temelju funkcije. Politika kod tih agenata nije fiksna već se uči kroz iskustvo, i taj pristup nazivamo aktivno RL.

Jedan od pristupa pasivnom RL je izravna procjena korisnosti (eng. *direct utility estimation*). Agent nasumično izvršava akcije u više pokušaja (eng. *trial*) rješavanja problema

kako bi prikupio podatke o stanjima i dobivenim nagradama. Zatim računa korisnost svakog stanja koje je posjetio u prethodnim pokušajima. Korisnost se računa kao ukupna očekivana nagrada počevši od određenog stanja do kraja tog pokušaja. Iteracijom računanja korisnosti stanja tim principom na podacima svih pokušaja računa se prosjek očekivane nagrade za pojedino stanje. Ovaj pristup često nije učinkovit jer radi pretpostavku da su korisnosti stanja međusobno neovisna, što je rijetko točno [15].

ADP (eng. *Adaptive Dynamic Programming*) metoda unaprjeđuje ranije navedeni pristup korištenjem modela prijelaza stanja, te Bellmanove jednadžbe za računanje korisnosti stanja. Bellmanova jednadžba računa korisnost stanja kao sumu nagrade za to stanje te diskontiranu vrijednost očekivane nagrade sljedećeg stanja ako je agent poduzeo optimalnu akciju [1]. Iskazuje se na sljedeći način gdje korisnost U stanja s je jednaka sumi nagrade R za stanje s i umnoška diskontnog faktora γ s maksimalnom očekivanom korisnosti sljedećeg stanja s' množenog s vjerojatnosti da nastupi stanje s' nakon poduzete akcije a u stanju s. Diskontni faktor γ se zadaje procjenom na razini slučaja i određuje važnost budućih nagrada u odnosu na trenutne.

$$U(s) = R(s) + \gamma \ max \sum_{s'} P(s'|s,a)U(s')$$

Kombinacijom modela stanja iz kojeg se dobiva vjerojatnost da nastupi stanje s' i Bellmanove jednadžbe agent može mnogo preciznije procijeniti korisnost nekog stanja koje po sebi ne donosi veliku nagradu, ali vodi do stanja koje donosi. Ovaj pristup i dalje koristi fiksnu politiku tako da je djelom pasivnog RL. Politika se računa algoritmom za iteraciju vrijednosti po izboru te procjenjuje navedenim procesom [15].

Navedeni pristup iako rješava neke od problema izravne procjene korisnosti ne rezultira uvijek stvarno optimalnom politikom jer i dalje ovisi o početnom prikupljenom skupu pokušaja na temelju kojih gradi model prijelaza stanja. Taj model može obuhvaćati samo mali dio stvarnih mogućih stanja, a kako agent nema poticaj tražiti nova stanja jer za njih nema definiranu očekivanu korisnost nema ni motivacije da ih istraži. Agent će uvijek odabirati stanja za koja očekuje nagradu prije stanja za koja ju ne očekuje, te takvog agenta nazivamo pohlepnim (eng. *Greedy agent*). Iz tog razloga ADP metoda može se unaprijediti na način da potiče agenta na istraživanje novih stanja u okružju. Postoji više načina za poticanje agenta na istraživanje. Način koji se može uključiti u Bellmanovu jednadžbu obuhvaća dodjeljivanje više očekivane koristi kombinacijama stanja i akcija koje još nisu istražene. U tom slučaju vrijednost buduće očekivane vrijednosti u Bellmanovoj jednadžbi zamijenimo funkcijom koja u slučaju da se radi o kombinaciji stanja i akcije koja još nije istražena zadani broj puta vraća najveću očekivanu nagradu, a u suprotnom vraća stvarnu očekivanu vrijednost [1]. Agent koji na ovaj način računa očekivanu korisnost stanja uz istraživanje okružja uči aktivno kroz nova iskustva.

Alternativa ADP metodi koja ne koristi model prijelaza stanja je TD metoda (eng. *Temporal Difference*). U toj metodi očekivana korisnost stanja ažurira se na temelju jednog prijelaza iz stanja u novo stanje te dobivene nagrade. Za to nije potreban čitav model prijelaza kako se radi o samo dva stanja. Za računanje korisnosti stanja se tada koristi jednostavan slučaj Bellmanove jednadžbe uz dodatak faktora učenja α. Ta jednadžba se iskazuje na sljedeći način [1].

$$U(s) \leftarrow U(s) \ + \ \alpha(R(s) + \gamma U(s') - U(s))$$

Faktor učenja se zadaje te on određuje važnost koju pridajemo jednoj instanci prijelaza iz stanja s u s' iz koje se uči. Faktor može biti fiksan ili se može smanjivati svakog puta kad se računa za isto stanje. TD metoda se smatra pasivnim RL jer samo procjenjuje korisnost stanja na temelju fiksne politike kako bi se ona evaluirala. Iako TD metoda zanemaruje očekivanu korisnost budućih stanja i računa samo buduću korisnost sljedećeg stanja, u praksi donosi dobre rezultate jer nije računalno intenzivna što nadoknađuje relativno sporu konvergenciju prema stvarnoj korisnosti stanja [15].

Q-učenje (eng. *Q-Learning*) je metoda koja obuhvaća učenje optimalne politike kroz primjenu trenutno najbolje poznate politike te se stoga smatra aktivnim RL. Klasičan pristup Q-učenju uključuje učenje prijelaznog modela kako bi se izračunale Q vrijednosti za kombinacije stanja i akcija. Kod Q-učenja korisnost pojedinog stanja s je jednaka maksimalnoj očekivanoj korisnosti poduzimanja optimalne akcije a u sljedećem stanju. To se iskazuje ako funkcija na sljedeći način [1].

$$U(s) = \max Q(s, a)$$

Za računanje Q vrijednosti koristi se Bellmanova jednadžba kao i u slučaju prethodnih metoda, uz manje izmjene kao u nastavku [1].

$$Q(s, a) = R(s) + \gamma \sum_{s'} P(s'|s, a) max Q(s', a')$$

Ovaj pristup obuhvaća učenje prijelaznog modela kako bi se mogla izračunati vjerojatnost nastupanja stanja s' ako je u stanju s poduzeta akcija a. Postoji i pristup Q-učenju koji se zasniva na TD metodi gdje se Q vrijednosti računaju samo za pojedinu kombinaciju stanja i akcije koja se u njemu može poduzeti. Za taj pristup nije potrebno učiti prijelazni model kao i u slučaju TD metode. Q vrijednosti za pojedina stanja i akcije računaju se prema sljedećoj jednadžbi [1].

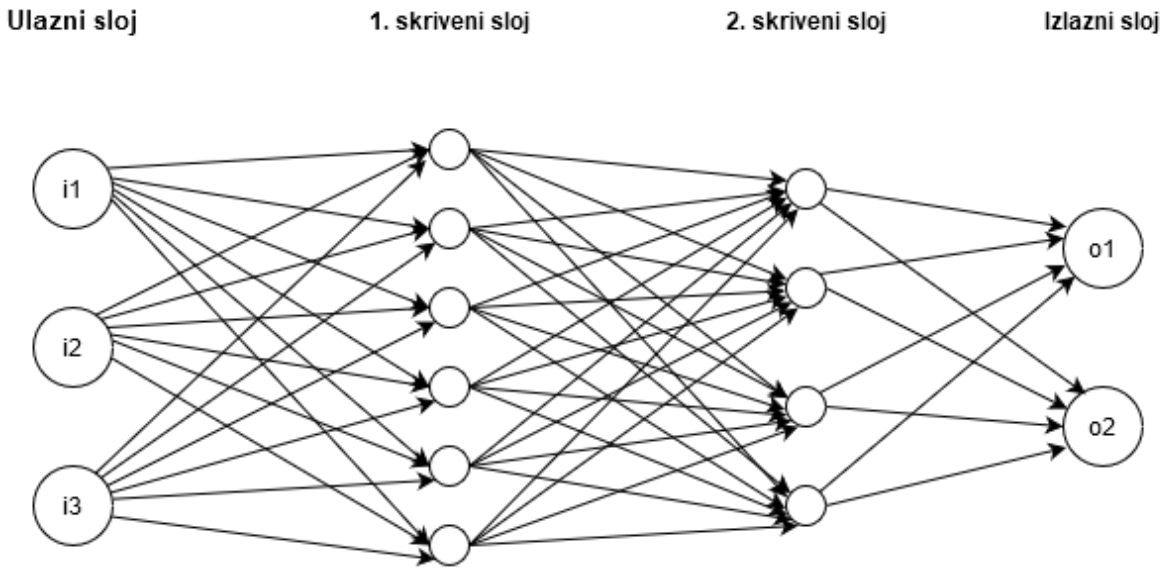
$$Q(s, a) \leftarrow Q(s, a) + \alpha(R(s) + \gamma \ max Q(s', a') - Q(s, a))$$

Kao i u slučaju TD metode, α je faktor učenja koji je potrebno odrediti. Politika agenta u Q učenju je odabirati akciju koja vodi do najveće Q vrijednosti za optimalnu akciju u sljedećem stanju $Q(s', a')$. Kako se radi o aktivnom RL učenju, također se preporuča korištenje funkcije za istraživanje okružja kako se ne bi propustilo uzeti u obzir neistražena stanja kao posljedica odabira poznatih akcija u stanjima kojima je izračunata viša Q vrijednost.

2.2.4. Duboko učenje

Duboko učenje (eng. *Deep Learning*) je vrsta strojnog učenja u kojem se koriste umjetne neuronske mreže (eng. *Artificial Neural Networks*). Neuronske mreže se mogu koristiti za nadgledano, nenadgledano i pojačano učenje same po sebi ili u kombinaciji s nekim od algoritama. Suština funkcioniranja neuronskih mreža jest da na temelju određenih ulaznih podataka koji se obrađuju u skrivenim slojevima daju rezultat u obliku izlaznih podataka. Te mreže se mogu trenirati na proizvoljnoj vrsti podataka od jednostavnih brojeva do slika te nude bolje performanse što je više podataka dostupno na kojima se može trenirati. Koncept neuronskih mreža postoji od četrdesetih godina prošlog stoljeća, no nije doživio raširenu primjenu u strojnom učenju do prethodnih desetljeća kada su velike količine podataka postale dostupne te je postignut značajan napredak po pitanju računalnih resursa [16]. Veliki broj novih tehnologija poput samovozećih automobila, virtualnih pomoćnika te sustava za razumijevanje i prevođenje jezika razvijeni su pomoću dubokog učenja.

Neuronske mreže su ključ dubokog učenja te kako bi smo shvatili njihovu primjenu u strojnom učenju potrebno je prvo definirati što one jesu. Neuronske mreže su strukture koje su apstraktno analogne strukturi ljudskog mozga. Sastoje se od međusobno povezanih čvorova (eng. Nodes, Units) koji reprezentiraju neurone [1]. Ti čvorovi su podijeljeni u slojeve koje povezuju u strukturu mreže te ih se može prikazati usmjerenim grafom. Svaka mreža ima ulazni, izlazni te jedan ili više skrivenih slojeva. Broj čvorova u ulaznom sloju odgovara ulaznim podacima, skriveni sloj može imati proizvoljan broj čvorova, a izlazni sloj ima broj čvorova koji odgovaraju izlaznim podacima. Prikaz neuronske mreže s dva skrivena sloja možemo vidjeti na slici 3.



Slika 3. Primjer jednostavne neuronske mreže

Na slici prikazana je mreža koja se sastoji od dva skrivena sloja, prvi sa šest čvorova i drugi sa četiri. Ovo je vrlo pojednostavljen primjer u svrhu ilustracije te se u praksi koriste mreže s više stotina čvorova u skrivenim slojevima.

Neuroni iz jednog sloja aktiviraju neurone u sljedećem prema zadanoj aktivacijskoj funkciji. Svaka veza između neurona ima pripadnu težinu koju označavamo kao $w_{i,j}$ gdje je s i označen ulazni čvor a s j izlazni. Težina predstavlja snagu veze između dva čvora te se koristi u računanju aktivacijske funkcije. Tu funkciju računamo na temelju sume svih aktivacija čvorova koji su s njom spojeni. Matematički tu funkciju g prikazujemo na sljedeći način [1].

$$a_j = g(in_j) = g(\sum_{i=0}^{n} w_{i,j}\, a_j)$$

Na temelju te funkcije čvor šalje signal neuronima s kojima je povezan. Razlikujemo linearne, nelinearne te aktivacijske funkcije binarnog koraka (eng. *binary step*). Najjednostavnije su funkcije binarnog koraka koje samo određuju minimalnu fiksnu vrijednost potrebnu da se čvor aktivira te u slučaju aktivacije uvijek šalje istu jačinu signala kakva je bila zaprimljena. Linearne i nelinearne funkcije dozvoljavaju aktivaciju neurona s različitim signalima aktivacije. Linearne funkcije su također jednostavne no ne omogućavaju korištenje algoritma spuštanja po gradijentu (eng. gradient descent) u sklopu minimizacije funkcije gubitka za treniranje mreže jer je njihova derivacija konstantna vrijednost. Nelinearne funkcije aktivacije dozvoljavaju korištenje tog algoritma te su neke od najčešće korištenih ReLU (eng. Rectified Linear Unit) i njezine varijante, sigmoidne funkcije te Swish aktivacijska funkcija [17]. ReLU funkcija vraća pozitivni signal jednak ulaznoj vrijednosti ako je ona pozitivna a u suprotnom se ne aktivira. Svaka funkcija ima svoje prednosti te je kod treniranja neuronske mreže vrlo važno odabrati prikladnu kako bi se ostvario što bolji rezultat učenja.

Kroz treniranje neuronske mreže na podacima prilagođavaju se vrijednosti težina veza između čvorova. To se može postići nekim od algoritama za optimizaciju za definiranu funkcije gubitka koju se nastoji minimizirati. U slučaju nadgledanog učenja, kao ulazni podaci koriste se označeni skupovi podataka. Na temelju uspoređivanja izlaznih vrijednosti mreže s poznatim traženim rezultatima optimiziraju se težine veza u mreži kako bi mreža davala tražene rezultate. U obradi slikovnih podataka najčešće se koristi ovaj pristup. U slučaju nenadgledanog učenja neuronske mreže mogu se koristiti za predviđanje vrijednosti na temelju obrađenih podataka. U pojačanom učenju neuronske mreže možemo koristit za optimizaciju politike agenta, kao što ćemo pokazati na praktičnom primjeru ovog rada.

3. Izrada agenta koji uči igrati Cartpole

U ovom poglavlju posvetit ćemo se primjeni metode dubokog pojačanog učenja (eng. *Deep Q-Learning*) u praksi. To ćemo ostvariti kroz razvoj agenta koji igra računalnu igru. Ideja za korištenje računalne igre kao okružje za agenta proizlazi iz činjenice da je računalna igra zapravo simulacija s definiranim pravilima, ciljem, preprekama te sustavom za rangiranje uspješnosti ostvarivanja tog cilja. Uloga igrača u računalnoj igri dobrim dijelom se preklapa s ulogom agenta u svom okružju. Računalne igre podrazumijevaju sposobnost učenja i odlučivanja od strane ljudskog igrača, što su upravo svojstva koja mi nastojimo postići kod našeg agenta. Istu podudarnost primijetili su i odlučili iskoristiti osnivači OpenAI korporacije kada su htjeli pripremiti univerzalni alat za one koji žele eksperimentirati s umjetnom inteligencijom, bez da moraju tratiti vrijeme na osmišljanje i implementaciju okružja. Koristeći biblioteku OpenAi Gym agentu ćemo omogućiti da igra Cartpole igru. Kako bi smo kreirali agenta koji može učiti kako igrati koristit ćemo metodu Q-učenja uz pomoć neuronske mreže za računanje Q vrijednosti.

Prije nego krenemo s implementacijom samog agenta potrebno je definirati radno okruženje u kojem to možemo ostvariti. Većina alata koja podržava rad s umjetnom inteligencijom moguće je koristiti na više operacijskih sustava te u različitim programskim jezicima. Ipak jasan favorit je linux operacijski sustav gdje većina alata nudi sve svoje funkcionalnosti i najbolje performanse. Također većina tih alata primarno je razvijena za rad u pythonu i za njih nudi najveću podršku, dok je tek kasnije omogućen rad s ostalim jezicima i to samo u pojedinim slučajevima. Iz tog razloga za implementaciju našeg agenta koristit će se Ubutnu operacijski sustav verzije 19.10, pomoću softvera za virtualizaciju VirtualBox. Virtualizacija operacijskog sustava nije ideala s obzirom na performanse te se u slučaju rada s intenzivni primjenama strojnog učenja svakako treba izbjeći, no u našem slučaju neće imati značajne posljedice. Kao programski jezik odabrat ćemo posljednju verziju pythona 3.7.5 jer je najbolje podržana od strane potrebni alata. Ubuntu dolazi s jednostavnim text editor alatom koji će biti dostatan za naše potrebe pisanja koda u pythonu. Postoje brojne biblioteke koje omogućavaju rad s agentima, no za potrebe ove implementacije odabran je SPADE radni okvir. SPADE je biblioteka za python koja omogućava jednostavnu implementaciju različitih agenta prema FIPA specifikaciji [12]. Kako smo odabrali raditi s neuronskim mrežama za to će nam biti potreban odgovarajući alat. Za tu primjenu Google je razvio Tensorflow biblioteku koju ćemo koristiti u našem rješenju.

Kako bi smo navedenu ideju za implementaciju agenta mogli provesti u djelo, potrebno je prvo temeljno shvatiti pravila same igre, te alate koje ćemo koristiti za razvoj agenta. U narednim potpoglavljima opisat ćemo način rada sa svakom spomenutom bibliotekom prije

nego pristupimo njihovoj implementaciji u kodu. Kroz razvoj agenta opisat ćemo njegovu osnovnu arhitekturu te različite strategije koje se koriste za poboljšanje učenja.

3.1. Opis Cartpole igre

Ideju za Cartpole igru osmislili su Barto, Sutton i Anderson u svom radu o prilagodljivim elementima sličnim neuronima koji rješavaju teške probleme učenja i upravljanja [13]. Ideja je balansirati štap koji je pričvršćen za kolica u samo jednoj točci. Kolica se nalaze na tračnicama te se po njima mogu kretati lijevo ili desno. Tračnice su fiksne duljine te kolica mogu pasti s njih ako odu predaleko u jednu od strana. Igrač upravlja kolicima te ih može pomicati u lijevo i desno. Cilj je što dulje balansirati štap bez da se previše nagne u stranu te bez da kolica padnu s tračnica.

Kao što je ranije spomenuto naš agent će igrati implementaciju igre koju pruža OpenAi Gym [14]. Najnovija dostupna verzija igre je Cartpole-v1 koja se razlikuje od prethodne samo po maksimalnom broju dozvoljenih koraka prije kraja igre. U toj implementaciji definirana su određena pravila koja će agent morati poštovati. Na početku igre štap započinje u uspravnom položaju. Duljina tračnica iznosi 4.8 jedinica a kolica se na početku nalaze približno na sredini. Kolica i štap imaju određenu težinu, tako da ovisno o njihovoj brzini mijenja im se njihov momentum. Brzinom upravlja agent tako da bira između broja 0 za ubrzanje u lijevu stranu te broja 1 za ubrzanje u desnu stranu. Igra započinje tek kada agent poduzme prvi korak. Trajanje igre ograničeno je na maksimalno 500 takvih koraka. Za svaki uspješan korak unutar kojeg igra nije izgubljena, to jest štap se nije previše nagnuo i kolica su i dalje na tračnicama dodjeljuje se +1 jedinica nagrade. Nakon inicijaliziranja igre i svakog sljedećeg koraka agentu se javlja novo stanje njegove okoline, tj. stanje u igri. Stanje je tipa Box(4) što znači da sadrži četiri decimalna broja koja predstavljaju položaj i ubrzanje kolica, te kut i brzinu vrha štapa. Prikaz stanja s minimalnim i maksimalnim dozvoljenim vrijednostima možemo vidjeti na tablici 2.

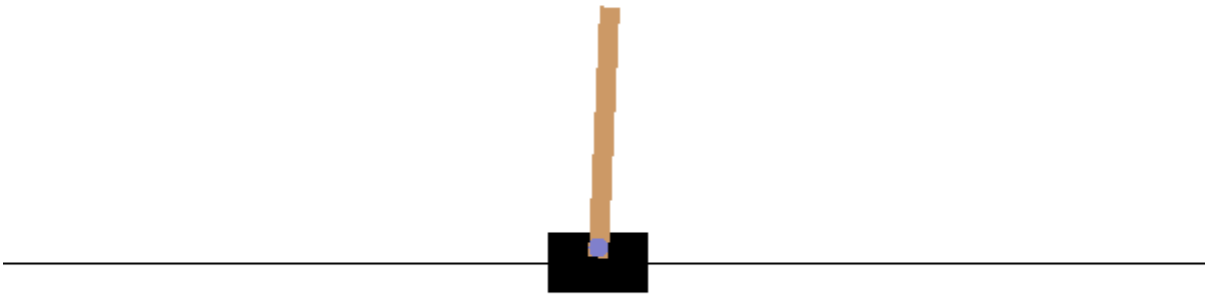
Indeks podatka	Promatrano svojstvo	Minimalna vrijednost	Maksimalna vrijednost
0	Položaj kolica	-2.4	2.4
1	Brzina kolica	- ∞	∞
2	Kut štapa	~ - 41.8°	~ 41.8°
3	Brzina vrha štapa	- ∞	∞

Tablica 2. Prikaz maksimalnih dozvoljenih vrijednosti koje karakteriziraju stanje igre [14]

Na početku igre nasumično se određuju položaj i brzina kolica, te kut i brzina vrha štapa unutar ±0.05 jedinica nulte vrijednosti za pojedino svojstvo. Primjer stanja koje agent dobije kao povratnu informaciju kod inicijalizacije igre izgleda ovako:

$$[0.03015496 \;\; 0.01370 \;\; 0.03009693 \;\; 0.06251934]$$

Podaci o stanju okružja su polje u kojem indeks pojedine vrijednosti odgovara opisima vrijednosti u tablici 2. Ostale vrijednosti koje okružje iz biblioteke vraća agentu nakon pojedinog koraka su iznos nagrade, da li je igra gotova te informacije za dijagnostiku vezane uz stanje igre. Biblioteka također omogućava grafički prikaz igre nakon svakog agentovog koraka koristeći pyglet biblioteku. Primjer takvog generiranog prikaza možemo vidjeti na slici 4.



Slika 4. Grafički prikaz igre Cartpole-v1

Nakon što smo opisali pravila igre te njenu implementaciju u OpenAi Gym biblioteci, možemo razmotriti kakvo je to okružje za našeg agenta. Kao što smo ranije rekli, agent prilikom svakog koraka zna kakvo je stanje njegovog okružja tako da je ono transparentno. Kako se radi o igri za jednog igrača, nije riječ o višeagentnom okružju. Također kako u igri postoje pravila o posljedicama svake odluke u obliku zakona fizike koje kolica i štap poštuju, te je agent jedini koji upravlja igrom, iduće stanje okružja može se predvidjeti na temelju njegovih odluka, stoga je ono determinističko. Iako se igra može odigrati u više epizoda, kroz tijek igre odluka agenta u jednom koraku svakako trebaju utjecati na odluku u sljedećem, stoga je naše okružje slijedno. Kako se idući korak u igri odigra tek kad agent donese odluku o svojem sljedećem potezu okružje je statično. S obzirom na kontinuirane vrijednosti koje opisuju stanje okružja, ono je kontinuirano, tj. ne poprima diskretni broj stanja. Kako agent na početku nema znanje o pravilima igre i posljedicama svojih akcija, ono je njemu nepoznato. Upravo zato koristit ćemo metodu Q-učenja kako bi agent naučio iz iskustva igranja igre koje poteze napraviti u kojoj situaciji.

3.2. Korištene biblioteke

Kako bi smo ostvarili implementaciju našeg agenta koristit ćemo više različitih biblioteka, no njih troje su od posebnog značaja za našeg agenta te ćemo ih posebno pojasniti prije nego krenemo s implementacijom. Prva biblioteka je SPADE okvir za rad s agentima. Omogućava kreaciju XMPP baziranog agenta kakvog ćemo koristiti za naše rješenje. Druga biblioteka je OpenAI Gym koja će nam omogućiti jednostavan pristup našoj odabranoj igri. Treća biblioteka je Tensorflow koja je također ključna za implementaciju dubokog učenja te ćemo njezine i funkcionalnost ostalih biblioteka opisati u sljedećim potpoglavljima.

3.2.1. SPADE

Smart Python Agent Development Enviroment (SPADE) je platforma koja omogućava razvoj višeagentnih sustava. Temelji se na pythonu i XMPP standardu za slanje poruka [15]. Ideju za takvom platformom predstavili su Javier Palanca Camara i suradnici u svom radu na temu Višeagentna sustavska platforma temeljena na Jabberu 2006. godine [12]. Od tad je platforma razvijana pod MIT licencom kao besplatan softver. SPADE podržava Foundation for Intelligent Physical Agents (FIPA) standard za komunikaciju agenata koristeći Agent Communication Language (ACL) [12]. Kako je to jedan od najprihvaćenijih standarda te on omogućava komunikaciju između brojnih agenata na različitim sustavima koji mogu biti pisani i u drugim jezicima, SPADE je prihvaćen kao jedno od najpopularnijih rješenja za implementaciju agenata u pythonu.Takva fleksibilnost u komunikaciji je postignuta korištenjem XMPP servera na kojeg je potrebno registrirati svakog agenta. Moguće je kreirati vlastiti takav server, no postoje besplatni takvi serveri na kojima je moguće registrirati vlastitog agenta putem interneta, kao što su https://jix.im/en/register/ i https://jabb.im/reg/. Svaki agent ima vlastiti Jabber ID (JID) i lozinku koje je potrebno specificirati prilikom implementacije agenta. Ako smo odabrali opciju besplatnog XMPP servera potrebna je veza na internet kako bi se on koristio.

Nakon što smo objasnili što je SPADE platforma i koja je njezina svrha, možemo proučiti kako uz nju kreiramo agente te kakvi oni mogu biti. Svaki agent implementira se kao klasa koja prima objekt agenta iz spade biblioteke. Svaka takva klasa definira setup metodu u kojoj možemo konfigurirati dodatne informacije vezane uz agenta. Unutar klase agenta definiraju se pojedine klase za njegova ponašanja, te unutar tih klasa run metoda ponašanja, te opcionalne on_start i on_end metode. Nakon što je klasa agenta definirana, ona se može instancirati uz proslijeđen JID i lozinku. Agenta pokrećemo njegovom metodom start() i završavamo metodom stop().

Agenti mogu imati različita ponašanja, te čak i više njih odjednom. Glavni način na koji programiramo agenta je da definiramo njegova ponašanja. Klase unutar kojih definiramo ponašanja primaju objekt pripadnog ponašanja iz SPADE biblioteke. SPADE platforma nudi nam cikličko (eng. *cyclic*), jednokratno (eng. *one-shot*), periodično (eng. *periodic*), jednokratno s odmakom (eng. *Timeout*) te ponašanje konačnog automata (eng. *Final State Machine* – FSM). Cikličko ponašanje kao što njegovo ime sugerira omogućuje agentu da ponavlja zadatak opisan unutar ponašanja u nedogled. Jednokratno ponašanje izvršava se samo jednom. Periodično ponašanje slično je cikličkom, no ponavlja se u fiksnim vremenskim razmacima odnosno periodima. Jednokratno ponašanje s odmakom također svoj zadatak izvršava samo jednom no nakon što prođe definirana količina vremena od kad je ponašanje pokrenuto. Ponašanje konačnog automata malo je zanimljivije. Unutar tog ponašanja definiraju se stanja unutar kojih agent obavlja određene aktivnosti. Stanje nakon što je gotovo može prebaciti agenta u neko drugo stanje ili agent može završiti s radom. U setup metodi FSM agenta potrebno je definirati sve dozvoljene prijelaze stanja. Prijelaze stanja moguće je prikazati dijagramom stanja koji opisuje mogući slijed rada agenta. Za našeg agenta odabrat ćemo FSM ponašanje te ćemo ga prikazati jednim takvim dijagramom. Kreirat ćemo ga poštujući gore opisanu proceduru te mu definirati stanja po potrebi. Možemo još napomenuti da agent može pamtiti objekte tako da ih spremimo kao varijable na self.agent.varijabla. Tim objektima agent može pristupati iz bilo kojeg stanja te se oni ne mijenjaju prijelazom iz stanja u stanje osim kada mi to definiramo. SPADE agenti nude još mnoge druge funkcionalnosti koje možemo pronaći u API dokumentaciji, no ovo su neke osnovne koje će nama biti potrebne [16].

3.2.2. OpenAi Gym

OpenAi Gym je biblioteka koja pruža resurse za rad s okružjima namijenjenih agentima koji koriste metodu pojačanog učenja. Ta okružja su različite vrste igara od jednostavnih toy text do kompleksnijih 3D igara, te okružja za rad s robotskim rukama. Biblioteku je kreirala OpenAi kompanija 2016. godine s ciljem da rad s metodama pojačanog učenja učini pristupačnijim i standardiziranijim [17]. OpenAi je neprofitna organizacija koje uviđa potencijal unaprijeđenja te znanosti kao opću društvenu dobit te je biblioteka Gym besplatna za korištenje. Poduzeće su osnovali Elon Musk, Greg Brockman, Ilya Sutskever, Sam Altman i Wojciech Zaremba 2015. godine te je od tada objavljen značajan broj istraživačkih radova te edukacijskog materijala u sklopu djelovanja poduzeća [18].

Gym biblioteka dijeli se na sedam kategorija okružja, uz mogućnost kreiranja i korištenja novih okružja od trećih strana. Okružja variraju s obzirom na kompleksnost i tematiku igara, te ih većina sadrži varijacije na istu igru.. Kategorija s najjednostavnijim okružjima zove

se Toy text namijenjena je početnicima koji se žele upoznati s radom biblioteke i metodom pojačanog učenja. Sadrži igre kao što su ajnc (eng. *Blackjack*), igra toplo-hladno i pogađanje brojeva. Igre se prikazuju tekstualno u terminalu. Druga kategorija igara svodi se na aproksimaciju ponašanja algoritama kao što su kopiranje i zbrajanje. Kategorija classic control sadrži igre koje su predložene u dosadašnjim radovima na temu pojačanog učenja kao što je naša odabrana igra Cartpole. Igre koriste pyglet biblioteku za 2D prikaz igre u zasebnom prozoru. Kategorija robotics sadrži okružja u kojima agent uči manipulirati robotsku ruku kako bi izvršavala jednostavne zadatke kao što su premještanje predmeta ili okretanje kocke, jajeta ili štapova. Kategorija Box2D sadrži igre koje se temelje na Box2D alatu za simulaciju fizike u dvodimenzionalnom prostoru. Uključuje igre kao što su hodanje na dvije noge, utrkivanje s autićem i slijetanje na Mjesec. MuJoCo kategorije slično prethodnoj kategoriji sadrži igre temeljene na alatu za simulaciju fizike u trodimenzinalnom prostoru MuJoCo. Nažalost MuJoCo alat nije besplatan, no postoji alternativni open source alat koji se može koristiti umjesto njega pod nazivom PyBullet [19]. Igre unutar te kategorije simuliraju hodanje, skakanje, ustajanje i plivanje različitih entiteta, te Cartpole igru i njezine varijacije. Posljednja kategorija igara sadrži više od stotinjak igara za Atari 2600 konzolu. Igre u okružju se simuliraju kroz Arcade Learning Environment koji koristi Stella Atari emulator. Ta okružja predstavljaju najveći izazov djelomično zbog kompleksnosti a djelomično zbog sučelja koje pružaju agentu. Naime, za gotovo svaku igru postoji varijacija gym okružja u kojem agent dobiva informaciju o okružju kao sliku jednog okvira u igri li kao sadržaj RAM-a.

Rad s Gym bibliotekom je relativno jednostavan ako uzmemo u obzir da je alternativa improvizirati sučelje između agenta i njegovog okružja, ili samostalno kreirati simulaciju kao agentovo okružje. Nakon što Gym biblioteku dodamo u naš projekt iz nje možemo pristupiti svakoj od navedenih igara kroz gym.make() funkciju tako da joj proslijedimo odgovarajuće ime igre i vraćeni objekt spremimo u env varijablu. Nakon toga iz env varijable možemo dobiti sve potrebne informacije o okružju, stvarati prikaz o igri, prosljeđivati akcije našeg agenta te dobivati povratne informacije o okružju. Svako okružje ima definiran prostor za akcije (eng. *action space*) i prostor za doživljaje iz okružja (eng. *observation space*). To znači da ovisno o igri agentove moguće akcije, te njegov doživljaj okružja su ograničeni unaprijed definiranim granicama i oblicima. Ograničenja akcija i doživljaja mogu biti reprezentirana dvjema vrstama prostora, to su Box i Discrete. Discrete prostor podrazumijeva određen prirodan broj akcija ili stanja okružja, te se označava kao npr. Discrete(4) za četiri diskretne akcije koje agent može poduzeti. Box prostor označava da doživljaj okružja ili akcije agenta mogu biti kontinuirane vrijednosti unutar nekog definiranog intervala. On se u biblioteci označava kao Box(4,) za 4 kontinuirane varijable koje npr. opisuju stanje nekog okružja u određenom trenutku. Svojstva takvih prostora za pojedinu igru možemo saznati ispisivanjem varijabli env.action_space i

env.observation_space nakon što smo odabrali igru. Ako želimo saznati ograničenja diskretnih prostora možemo ispisati gornje i donje granice na sličan način dodajući .high i .low na prethodne varijable, kao npr env.observation_space.low ako želimo ispisati donju granicu dozvoljenih vrijednosti za pojedina svojstva u zadanom okružju. Druga najvažnija funkcija za koju ćemo koristiti env varijablu je env.step() kojoj se prosljeđuje odabrana akcija agenta kao broj. Pozivom te funkcije izvršava se proslijeđena akcija te ona vraća novo stanje okružja, iznos nagrade, bool varijablu koja označava da li je igra gotova te info podatke za dijagnostiku potencijalnih problema. Nakon izvršavanja neke akcije možemo pozvati funkciju env.render() koja će generirati i prikazati jedan okvir grafičkog prikaza igre. Ako igra završi prebrzo može se desiti da se prikaz ne stigne prikazati pa se u tom slučaju preporučuje koristiti time.sleep() funkcija za proizvoljno kratku količinu vremena. Posljednja važna funkcija za koju koristimo env varijablu je env.close() koju koristimo nakon što je igra završena, tj. odigrana je jedna epizoda igre. Kod za svako okružje iz biblioteke dostupan je na githubu, uz opis svojstva okružja za popularnije igre.

3.2.3. Tensorflow

Tensorflow je besplatna biblioteka za rad s matematičkim metodama te metodama strojnog učenja u pythonu. Google je izdao biblioteku pod open source Apache 2 licencom 2015. godine [20]. Biblioteku je razvio Google Brain tim developera za kompaniju, no ubrzo su shvatili da je to alat od kojeg zajednica može imati veliku korist, te oni zauzvrat mogu imati veliku korist od zajednice koja će im pomoći dalje razvijati i usmjeravati razvoj alata. Uz potporu zajednice i jednog od najvećih titana računalne industrije ubrzo je postao najpopularniji alat za rad s modelima strojnog učenja [21]. Alat je razvijen kao python biblioteka no nudi podršku za druge jezike kao što su Java, C i Go korištenjem eksperimentalnih sučelja, no to sa sobom nosi dodatne komplikacije te potencijalno pogoršanje performansa [22]. Sama biblioteka je napisana u Pythonu, C++ i Nvidijinom CUDA jeziku kako bi se ostvarile što bolje performanse. Kao što možemo zaključiti iz čestog spominjanja performansi u vezi biblioteke, to je jedan od važnih faktora za razmotriti kod rada s bibliotekom. Dok će to uvijek pretpostavljati izazov u radu sa strojnim učenjem, Google nudi verziju biblioteke koja koristi samo procesor računala te verziju koja koristi procesor i grafičku karticu [23]. Biblioteka koja koristi i grafičku karticu ostvaruje bolje performanse kod rada s većim neuronskim mrežama, no podržava samo grafičke kartice koje proizvodi Nvidia i slabije podržava ostale jezike [24].

Tensorflow primarno koristi Keras biblioteku za rad s neuronskim mrežama. Keras biblioteka nije vrlo pristupačna pa je između ostalih sličnih alata dodana i u Tensorflow 2017. godine kao pozadinski alat za rad s neuronskim mrežama [25]. Keras omogućava kreiranje slojeva, korištenje optimizatora i aktivacijskih funkcija te mnoge druge funkcionalnosti potrebne

za rad s neuronskim mrežama. Moguće je Keras biblioteku koristiti samostalno no kroz Tensorflow je rad s neuronskim mrežama mnogo pristupačniji, bolje podržan i nudi brojne druge funkcionalnosti. Google je u rujnu 2019. izdao verziju 2.0 biblioteke koja je dodatno unaprijedila performanse na grafičkim karticama, uz broje druge promijene zbog kojih kod koji je radio s prijašnjim verzijama više nije bio kompatibilan [26]. Primjer toga je prelazak s rada sa sesijama na izravno izvršavanje (eng. *eager execution*). U ovom radu poslužit ćemo se starijom verzijom Tensorflowa 1.13 zbog većeg broja dostupne literature o radu s bibliotekom te ćemo objasniti osnove u njenom korištenju.

Rad s Tensorflow bibliotekom nije naročito jednostavan što je za očekivati s obzirom na prirodu njezinog zadatka. Biblioteka nudi veliki broj na prvi pogled zbunjujućih funkcionalnosti, no upravo to joj daje prilagodljivost i snagu koju koriste korporacije i istražitelji. Korištenje Tensorflow biblioteke svodi se na definiciju modela kroz kod koji onda implementiramo koristeći biblioteku. Prva stvar koju moramo znati jest da bi smo mogli raditi s bibliotekom moramo prvo pripremiti model naše neuronske mreže i pripadne funkcije. Mreža koju ćemo koristiti može biti jednostavna ili sadržavati više skrivenih slojeva. Moramo dakle pripremiti odabranu funkciju koju aproksimiramo, funkciju gubitka te potencijalno funkciju za izračun ciljnih vrijednosti. Nakon što smo to pripremili znamo koje su to vrijednosti koje želimo proslijediti modelu kod treniranja mreže. Za svaku takvu vrijednost kreiramo placeholder koristeći tf.placeholder() funkciju kojoj proslijedimo vrstu podataka koju pohranjujemo, najčešće int32 ili float32, te dimenzije podataka no to je opcionalno. Na sličan način možemo koristiti i konstante pomoću tf.constant() funkcije. Za vrijednosti funkcije koje želimo izračunati definiramo tensorflow varijable koristeći tf.Variable() funkciju kojoj proslijedimo inicijalnu vrijednost i vrstu podataka. Te varijable ne smijemo zaboraviti inicijalizirati pozivom na funkciju tf.global_variables_initializer() unutar sesije koristeći tf.run() funkciju prije nego ih model počne koristiti. Važno je napomenuti da vrijednosti s kojima model radi ne moraju biti jednostavni brojevi, već to mogu biti vektori, matrice ili podatkovne strukture većih dimenzija. Naziv biblioteke dolazi od naziva za tensore, a to su n-dimenzionalni objekti koji služe za pohranu podataka [27]. Matrica je dvodimenzionalni tensor, dok je vektor jednodimenzionalni tensor.

Nakon što imamo naše placeholdere i varijable možemo slijedno ispisati niz operacija iz naše funkcije koju aproksimiramo. Za to koristimo funkcije kao tf.add(), tf.subtract(), tf.multiply() ovisno o matematičkoj operaciji, a popis mogućih operacija možemo pronaći u dokumentaciji biblioteke [28]. Tim funkcijama proslijedimo potrebne varijable a posljednja varijabla u koju spremimo rezultat niza operacija predstavlja naš model. Na isti način koristeći tesorflow varijable, placeholdere i matematičke funkcije kreiramo i funkciju gubitka. Kako bi smo mogli trenirati naš model moramo ga pokrenuti unutar Tensorflow sessije koju možemo otvoriti koristeći tf.Session() funkciju i spremiti ju u varijablu. Istu funkciju na kraju zatvaramo

koristeći sess.close(). Nakon što smo definirali model, funkciju gubitka, placeholdere za podatke i otvorili sesiju možemo pokrenuti sesiju te iz nje dobiti rezultat funkcije koristeći tf.run() funkciju da provjerimo odgovara li rezultat očekivanom. Toj funkciji moramo proslijediti varijablu našeg modela, te rječnik koji mapira naše podatke na odgovarajuće kreirane placeholdere. Primjer poziva takve funkcije je:

```
tf.run(model, feed_dict={x_paceholder : x_data})
```

Ako rezultat poziva te funkcije s testnim vrijednostima odgovara rezultatu koji smo očekivali možemo pretpostaviti da je model ciljane funkcije dobro definiran u kodu. Nakon toga možemo odabrati optimizator koji ćemo koristiti za treniranje neuronske mreže. Biblioteka nudi veliki broj optimizatora čiji je popis dostupan u dokumentaciji biblioteke [29]. Ovisno o slučaju odabire se prikladan optimizator, no neki od češće korištenih su GradientDescentOptimizer i AdamOptimizer. Optimizator definiramo pozivajući njegovu pripadnu funkciju te mu prosljeđujemo brzinu učenja, najčešće vrijednosti 0.001 no kao i odabir optimizatora to je na diskreciju korisnika. Primjer dodjeljivanja optimizatora je sljedeći:

```
tf.train.AdamOptimizer(learning_rate=0.001)
```

Nakon što smo definirali optimizator odredimo mu da minimizira funkciju gubitka koristeći funkciju .minimize() i prosljeđujući joj funkciju gubitka. Nakon toga samo je preostalo napisati funkciju koja će u odabranom broju iteracija odraditi treniranje mreže. Unutar te funkcije koristimo naredbu tf.run() kojoj proslijedimo model funkcije koju aproksimiramo, te rječnik koji mapira ulazne podatke te ciljane podatke na odgovarajuće placeholdere. Primjer takve naredbe je sljedeći:

```
sess.run(model, feed_dict={x_placholder:x_data,
target_placeholder:target_data} )
```

Koristeći navedeni princip možemo kreirati jednostavan model aproksimirane funkcije, funkcije gubitka i ciljanih vrijednosti te istrenirati jednostavnu neuronsku mrežu pomoću tensorflow biblioteke. U slučaju da želimo kompleksniju neuronsku mrežu s više slojeva, biblioteka nam nudi funkciju tf.layers.dense() za dodavanje dense slojeva u našu mrežu. U tom slučaju nakon što smo definirali placeholdere i varijable pozovemo navedenu funkciju. Tu funkciju koristimo na sličan način kao i matematičke operacije. Kod kreiranja prvog dodatnog sloja funkciji prosijedimo ulazne podatke, broj čvorova koje će sadržavati te aktivacijsku funkciju. Biblioteka također korisniku nudi na izbor raznovrsne aktivacijske funkcije navedene u dokumentaciji biblioteke [30]. Primjer poziva funkcije za dodavanje sloja neuronskoj mreži je sljedeći:

```
tf.layers.dense.(x_placeholder, 100, activation=tf.nn.relu)
```

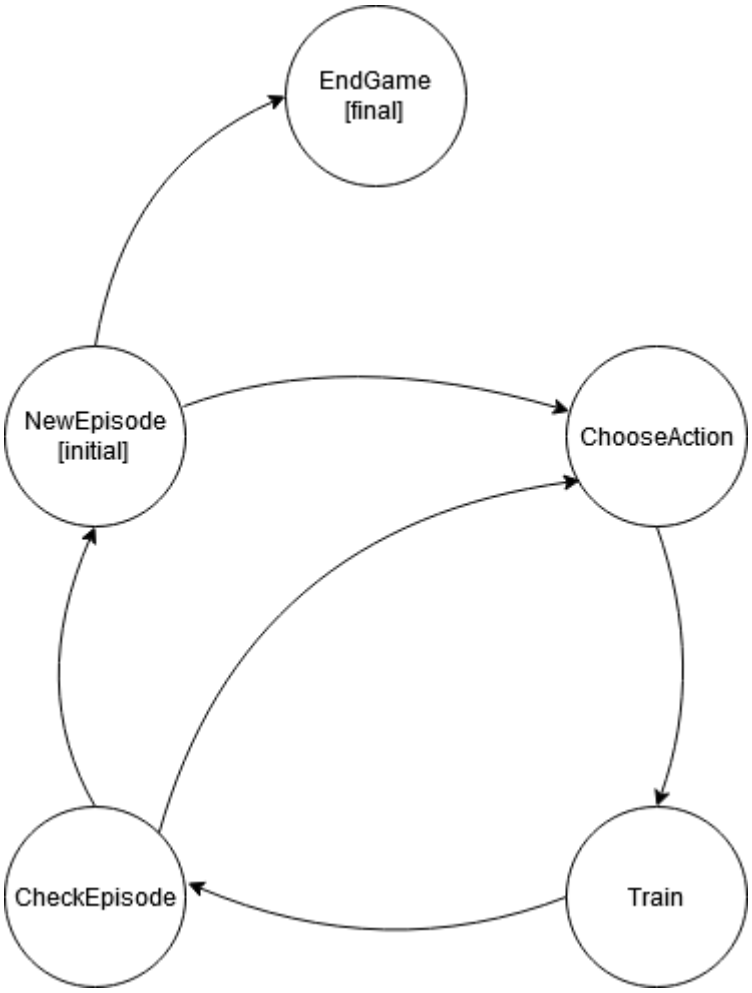
Korisnik koristeći tu funkciju može dodati proizvoljan broj dense slojeva, svaki idući put prosljeđujući prethodni sloj umjesto placeholdera. Biblioteka nudi još brojne funkcionalnosti za napredne korisnike no one spomenute u ovom poglavlju dovoljne su za uspješan rad s neuronskim mrežama u sklopu implementacije metode pojačanog učenja.

3.3. Implementacija agenta

Kako bi smo implementirali agenta koji koristi deep Q-learning metodu kako bi naučio igrati Cartpole igru potrebno je prvo pripremiti potrebne alate. Kao što je ranije spomenuto, koristit će se Ubuntu operacijski sustav koji ćemo virtualizirati pomoću VirtualBox alata. VirtualBox je open source alat za virtualizaciju operacijskih sustava te ga je moguće preuzeti sa stranice www.virtualbox.org. Koristit ćemo najnoviju verziju alata 6.1. Ubuntu operacijski sustav također je besplatan za korištenje te ćemo preuzeti verziju 19.10 s besplatne stranice www.osboxes.org. Nakon što smo uspostavili operacijski sustav na njega prvo moramo instalirati najnoviju verziju Python jezika, u ovom slučaju 3.7.5. Kako bi smo dalje mogli na jednostavan način instalirati ranije spomenute potrebne alate, instalirat ćemo i pip. Pomoću pip alata instalirat ćemo SPADE verziju 2.2.3, Gym verziju 0.17.2 i Tensorflow verziju 1.13.1. Nakon što smo osigurali sve potrebne alate možemo krenuti razvijati našeg osnovnog FSM agenta.

3.3.1. FSM agent

Kako bi smo kreirali FSM agenta, koristit ćemo SPADE biblioteku, no prije nego ga možemo implementirati moramo ga registrirati na jednom ranije spomenutih besplatnih stranica. Dodijelit ćemo mu ime Mat te proizvoljnu lozinku i time je on spreman za korištenje. Kako bi smo osmislili našeg FSM agenta moramo prvo definirati koja stanja ćemo mu implementirati, s obzirom na koje akcije želimo da u njima poduzima. Zasad samo želimo imati jednostavnog FSM agenta kojemu je omogućeno igranje Cartpool igre u više epizoda, no želimo imati prostora za proširiti njegove mogućnosti. Agentu ćemo odrediti pet ponašanja, prvo naziva NewEpisode u kojem će izvršiti potrebnu pripremu za igranje nove epizode, drugo naziva ChooseAction u kojem će agent odabrati svoju sljedeću akciju, treće naziva Learn u kojem će agent odigrati svoj potez, dobiti povratnu informaciju te kasnije nešto iz toga naučiti, četvrto naziva CheckEpisode u kojem će ovisno o tome da li je igra završena odlučiti odabire li novu akciju ili izvještava o rezultatu i posljednje stanje EndGame za nakon što su odigrane sve epizode. Prikaz stanja i njihovih veza možemo vidjeti na dijagramu stanja na slici 5.



Slika 5. Dijagram stanja FSM agenta

Kao što vidimo stanje NewEpisode je inicijalno, te ovisno o tome da li je odigran zadani broj epizoda vodi agenta u stanje odabira akcije ili završavanja igre. U stanju ChooseAction agent zna da igra i dalje traje te odabire akciju koju će poduzeti. U stanju Learn agent izvrši tu akciju te prelazi u stanje CheckEpisode. U tom stanju agent provjerava da li je trenutna epizoda završila, u kojem slučaju prelazi natrag u stanje NewEpisode, a u suprotnom odabire svoj sljedeći potez. Nakon što imamo arhitekturu stanja svojeg agenta možemo ga implementirati u kodu koristeći SPADE biblioteku za agenta te Gym biblioteku za kreiranje okružja igre. Implementacija opisanog osnovnog agenta slijedi u nastavku.

```
import spade
from spade.agent import Agent
from spade.behaviour import FSMBehaviour, State
import gym
import time

print("Gym:", gym.__version__) #0.17.2 installed
print("Setting up environment CartPole-v1")
env = gym.make('CartPole-v1')
print("Observation space:", env.observation_space) #Box(4,)
print("Action space:", env.action_space) #Discrete(2)
```

```
num_episodes = 5

class agent(Agent):
    class Ponasanje(FSMBehaviour):
        async def on_start(self):
            self.agent.episode = 0

        async def on_end(self):
            self.kill()

    class NewEpisode(State):
        async def run(self):
            self.agent.episode += 1
            if self.agent.episode <= num_episodes:
                print("zapocinjem novu epizodu.")
                env.reset()
                self.agent.ep_reward = 0
                self.agent.ep_done = False
                self.set_next_state("ChooseAction")
            else: self.set_next_state("EndGame")

    class ChooseAction(State):
        async def run(self):
            self.agent.action = env.action_space.sample()
            self.set_next_state("Learn")

    class Learn(State):
        async def run(self):
            next_state, reward, done, info = env.step(self.agent.action)
            env.render()
            time.sleep(0.1)
            self.agent.ep_reward += reward
            self.agent.ep_done = done
            self.set_next_state("CheckEpisode")

    class CheckEpisode(State):
        async def run(self):
            if self.agent.ep_done == False:
                self.set_next_state("ChooseAction")
            else:
                print("Episode: {}, Reward:
{:.2f}".format(self.agent.episode, self.agent.ep_reward))
                self.set_next_state("NewEpisode")

    class EndGame(State):
        async def run(self):
            env.close()
            print("Zavrsio igranje epizoda")

    async def setup(self):
        print("Agent setup...")
        fsm = self.Ponasanje()

        fsm.add_state(name="NewEpisode", state=self.NewEpisode(),
initial=True)
        fsm.add_state(name="CheckEpisode", state=self.CheckEpisode())
        fsm.add_state(name="ChooseAction", state=self.ChooseAction())
        fsm.add_state(name="Learn", state=self.Learn())
        fsm.add_state(name="EndGame", state=self.EndGame())

        fsm.add_transition(source="NewEpisode", dest="EndGame")
```

```
        fsm.add_transition(source="NewEpisode", dest="ChooseAction")
        fsm.add_transition(source="CheckEpisode", dest="ChooseAction")
        fsm.add_transition(source="CheckEpisode", dest="NewEpisode")
        fsm.add_transition(source="ChooseAction", dest="Learn")
        fsm.add_transition(source="Learn", dest="CheckEpisode")
        self.add_behaviour(fsm)

if __name__=='__main__':
    a = agent("mat@jix.im", "tajna")
    a.start()
    a.stop()
```

Na početku python skripte, prije klase agenta pomoću Gym biblioteke kreirano je okružje igre Cartpole naredbom gym.make() koje je spremljeno u varijablu env. Broj epizoda određen je na 5 kako se radi smo o primjeru za demonstraciju da agent može igrati više epizoda. U klasi agenta vidimo implementaciju pojedinih stanja. U stanju NewEpisode agent bilježi na vlastiti brojač epizoda da je započeta nova te inicijalizira ili resetira samo okružje, te zapamćene vrijednosti o nagradi i bool varijablu koja označava kraj epizode. U stanju ChooseAction agent nasumično odabire akciju od ponuđenih u akcijskom prostoru okružja, kako se radi o agentu koji samo demonstrira mogućnost da može igrati igru. Ovo stanje te stanje Learn ćemo unaprjeđivati u idućim iteracijama agenta kako bi smo mu omogućili da uči pametno odabirati svoje akcije. U ovakvom obliku agent u stanju Learn samo izvršava odabranu akciju koristeći naredbu env.step() te pamti podatke o dobivenoj nagradi i kraju igre. U tom stanju također odrađuje prikaz igre koristeći naredbu env.render() nakon koje je dodana naredba da uspava dretvu na desetinku sekunde jer bi se u suprotnom igra prebrzo odvijala što uzrokuje grešku prilikom prikaza. U idućem stanju CheckEpisode agent provjerava vlastitu varijablu o kraju igre koja u slučaju da nije istinita prelazi u stanje odabira akcije, a u suprotnom ispisuje rezultat odigrane epizode i prelazi u stanje u kojem započinje novu. Posljednje stanje EndGame samo zatvara okružje i izvještava o kraju. Kao i svako ponašanje SPADE agenta ima definirane funkcije on_start, on_end i setup. U on_start funkciji agent zasad samo postavlja vlastiti brojač odigranih epizoda na nulu, a u on_end se zaustavlja. U setup funkciji agent si dodaje navedena ponašanja te definira prijelaze između stanja analogno prijelazima prikazanim na dijagramu. Naposljetku u main funkciji klasa agenta se inicijalizira s proslijeđenim jabber ID-om i pripadnom lozinkom, te pokreće i zaustavlja nakon što je izvršio svoj zadatak. Ovakav jednostavan agent može igrati igru, no akcije odabire nasumično. Tim pristupom ostvaruje prosječni rezultat od približno 20 bodova po epizodi. To je loš rezultat, naročito ako uzmemo u obzir da je prihvaćeno da agent uspješno igra igru ako dosljedno postiže rezultat od prosječno 200 bodova. U narednim poglavljima unaprijedit ćemo ovog jednostavnog agenta koristeći deep Q-Learning metodu.

3.3.2. Deep Q-Learning agent

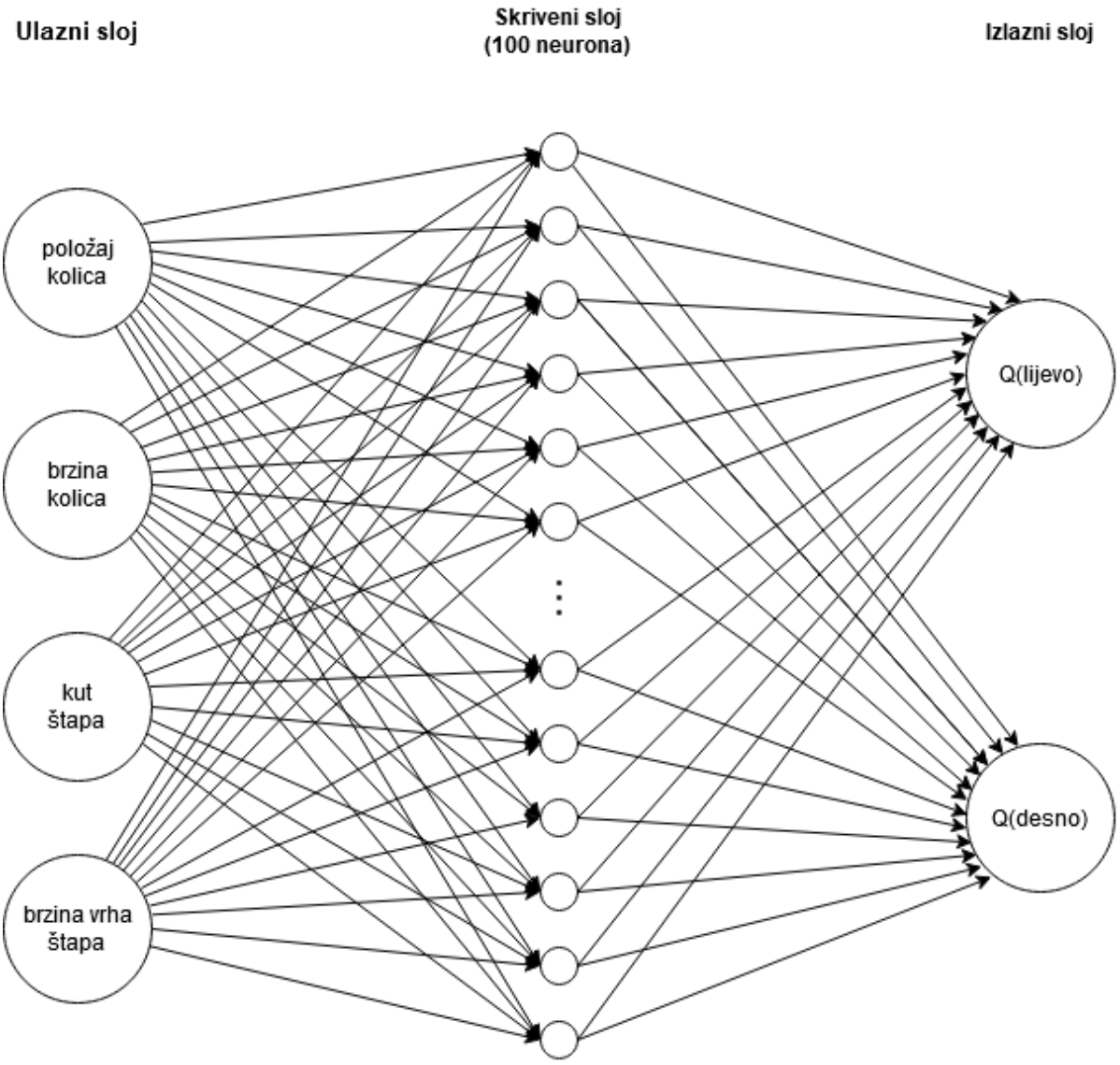
Nakon što smo kreirali osnovnog FSM agenta koji može igrati Cartpole igru, potrebno je primijeniti metodu deep Q-Learninga kako bi on mogao učiti iz svojih prijašnjih iskustava te s vremenom naučiti igrati igru. Prije nego krenemo na implementaciju u kodu moramo odrediti funkciju koju agent aproksimira, funkciju ciljane vrijednosti, funkciju gubitka te arhitekturu neuronske mreže. Kao što smo ranije spomenuli, u metodi Q-učenja funkcija koju ćemo aproksimirati neuronskom mrežom može se opisati Bellmanovom jednadžbom. Ona računa korisnost poduzete akcije u trenutnom stanja ovisno o dobivenoj nagradi i korisnosti sljedećeg stanja u kojem je poduzeta optimalna akcija.

$$Q(s,a) = R + \gamma \max\big(Q(s',a')\big)$$

Model neuronske mreže ćemo stoga prilagoditi tako da odgovara toj jednadžbi. Ciljanu vrijednost T također možemo izračunati koristeći Bellmanovu jednadžbu. Funkciju gubitka zatim računamo kao kvadriranu razliku između ciljane vrijednosti i q vrijednosti odabrane akcije u trenutnom stanju. Podsjetimo se da ona izgleda ovako:

$$Loss = (T - \max\,(Q(s',a'))^2$$

Na temelju ovih jednadžbi možemo odrediti vrijednosti koje ćemo davati mreži kao podatke te vrijednosti koje će ona aproksimirati. Naš agent iz svojeg okružja dobiva informacije o stanju okružja te dobivenoj nagradi. Vrijednost varijable gama (γ) koja predstavlja važnost buduće očekivane nagrade odredit ćemo na 0.97. Dakle preostaju nam Q vrijednost trenutnog stanja i akcije te predviđena maksimalna Q vrijednost budućeg stanja u kojem je poduzeta optimalna akcija. To su vrijednosti koje ćemo aproksimirati koristeći neuronsku mrežu. Ulaz u našu mrežu mora biti doživljaj stanja koje agent dobiva iz okružja, a izlaz akcija koji bi trebao poduzeti. Ulazni sloj će stoga odgovarati dimenzijama doživljaja. Između njega i izlaznog sloja stavit ćemo skriveni sloj. U kompleksnijim problemima može se koristiti više slojeva, no u ovom jedan može biti dovoljan. Skriveni sloj može sadržavati proizvoljan broj neurona, gdje veći broj može značiti bolje performanse učenja no predstavlja veće opterećenje za računalo. U ovom slučaju kreirat ćemo sloj sa 100 neurona. Posljednji sloj imat će samo 2 neurona koji odgovaraju agentovom izboru između dvije akcije u pojedinom stanju. Ti neuroni sadržavat će q vrijednost pojedine akcije, te će agent odabrati onu za koju će biti izražena najveća q vrijednost. Kako bi smo si lakše predočili opisanu arhitekturu neuronske mreže, možemo promotriti njezin prikaz na slici 6.



Slika 6. Prikaz slojeva neuronske mreže

Nakon što smo si odredili model neuronske mreže te potrebne funkcije, možemo razmisliti o implementaciji koristeći Tensorflow. Prvo ćemo morati kreirati potrebne placeholdere. Prvi placeholder biti će za opis okružja. Kao što smo rekli okružje je tipa Box što znači da sadrži kontinuirane vrijednosti, pa ćemo stoga kao tip podataka placeholdera definirati float32. Oblik podataka će odgovarati dimenzijama Box prostora koji opisuje okružje. Drugi placeholder koji će nam biti potreban je za akcije agenta. Tu se radi o diskretnim vrijednostima, konkretno samo 0 i 1 pa ćemo tip podataka postaviti na int32. Preostao je još placeholder za ciljanu vrijednost, koja također može biti kontinuirana vrijednost pa ćemo odrediti tip podataka float32. Nakon što imamo placeholdere možemo definirati model kakav smo zamislili. Kreirat ćemo skriveni sloj koji će primati ulazne vrijednosti stanja te imati 100 neurona. Odabrat ćemo ReLU aktivacijsku funkciju. To je funkcija koja će proslijediti ulaznu vrijednost do idućeg neurona ako je ulazna vrijednost pozitivna. Ona je najčešće korištena takva funkcija zbog

relativne jednostavnosti i dobrih performansa [31]. Ovaj sloj proslijedit ćemo u idući sloj koji će imati samo dva neurona te će predstavljati naš izlaz iz modela. Kako je rezultat iz tog sloja vektor od dva elementa, pomnožit ćemo ga a one hot vektorom koji predstavlja indeks akcije kao bi smo izolirali q vrijednost pojedine akcije u danom stanju. Time smo definirali model naše mreže, no potrebno je definirati i funkciju gubitka. Nju ćemo proslijediti optimizatoru s ciljem da ju se minimizira kroz treniranje mreže. Funkcija gubitka analogno ranije navedenoj formuli biti će kvadrat razlike ciljane vrijednosti te očekivane q vrijednosti za odabranu akciju u idućem stanju. Odabrat ćemo Adam optimizator. On je nadograđena verzija drugog najpopularnijeg Gradient Descent optimizatora, te omogućava brzo i učinkovito učenje [32]. Nakon što smo definirali model i potrebne funkcije, agentu je preostalo pokrenuti Tensorflow sesiju kojoj prosljeđuje model i promotreno stanje okružja, te iz modela dobiti akciju koju treba poduzeti. No kako bi ta akcija bila suvisla, mrežu je potrebno trenirati kroz više iteracija igre, odnosno epizoda. Stoga ćemo agentu omogućiti i da trenira svoju mrežu. Za to će agent nakon poduzete akcije izračunati ciljanu vrijednost koristeći Bellmanovu jednadžbu. Nakon što je već poduzeo akciju, agent može izračunati Q vrijednost prethodnog stanja i akcije jer mu je tada poznata nagrada postignuta tom akcijom, te rezultirano iduće stanje. Q vrijednost tog idućeg stanja i optimalne akcije može izračunati koristeći model neuronske mreže, a vrijednost game mu je poznata od početka. Na taj način izračunatu ciljanu vrijednost će agent koristiti prilikom treniranja mreže. Kako bi smo pokrenuli treniranje mreže moramo pokrenuti ranije definiran optimizator unutar Tensorflow sesije te mu proslijediti sve ranije definirane placeholdere. U kontekstu agenta to znači da nakon poduzete akcije, agent mu prosljeđuje svoje prethodno stanje, poduzetu akciju te izračunatu ciljanu vrijednost. Na taj način agent će nakon svakog poduzetog koraka trenirati svoju neuronsku mrežu koja će se na temelju tih podataka optimizirati i vraćati sve pouzdaniju procjenu o tome koji akciju treba poduzeti u trenutnom stanju.

Nakon što smo objasnili princip prema kojem ćemo raditi s Tensorflow bibliotekom, možemo krenuti s implementacijom na našem FSM agentu. Za rad s neuronskom mrežom koristit ćemo klasu preuzetu od slične implementacije agenta na githubu [33]. Ta klasa prilikom inicijalizacije definira placeholdere, model, funkciju gubitka te optimizator. Sadrži funkcije koje ažuriraju model mreže kroz treniranje i dohvaćaju q vrijednost stanja na ranije opisane načine. Implementacija te klase u kodu slijedi u nastavku.

```
class QNetwork():
    def __init__(self, state_dim, action_size):
        self.state_in = tf.placeholder(tf.float32, shape=[None,
*state_dim])
        self.q_target_in = tf.placeholder(tf.float32, shape=[None])
        self.action_in = tf.placeholder(tf.int32, shape=[None])
        action_one_hot = tf.one_hot(self.action_in, depth=action_size)
        self.hidden = tf.layers.dense(self.state_in, 100,
activation=tf.nn.relu)
        self.q_state = tf.layers.dense(self.hidden, action_size,
activation=None)
        self.q_state_action = tf.reduce_sum(tf.multiply(self.q_state,
action_one_hot), axis=1)

        self.loss = tf.square(self.q_target_in - self.q_state_action)
        self.optimizer =
tf.train.AdamOptimizer(learning_rate=0.001).minimize(self.loss)

    def update_model(self, session, state, action, q_target):
        feed = {self.state_in: state, self.action_in: action,
self.q_target_in: q_target}
        session.run(self.optimizer, feed_dict=feed)

    def get_q_state(self, session, state):
        q_state = session.run(self.q_state, feed_dict={self.state_in:
state})
        return q_state
```

U konstruktoru klase vidimo da su definirani ranije spomenuti placeholderi s pripadnim vrstama podataka. Svima je prva dimenzija definirana kao None kako ne bismo ograničavali broj podataka koje možemo odjednom pohraniti u mrežu, iako na trenutnom primjeru pohranjujemo samo po jedan podatak za svaki placeholder. Placeholderu za stanje naziva state_in definirane su dodatne dimenzije koje se podudaraju s dimenzijama observation_space varijable od Carpole okružja. Placeholder za akcije koje agent može poduzeti (0, 1), u kodu naziva action_in, potrebno je pretvoriti u one hot vektor. Taj vektor koristi se za dobivanje q vrijednosti pojedine akcije u danom stanju tako da se množi s rezultiranim vrijednostima iz neuronske mreže, te se reducira stupanj tog rezultata kako bi se vratio samo jedan broj kao vrijednost. Za to se koristi funkcija tf.reduce_sum kojoj je proslijeđen parametar axis=1, što znači da će reducirati vrijednost po stupcima tako da ih zbroji. Slojevi mreže implementirani su pomoću funkcija tf.layers dense. U varijabli hidden nalazi se ranije spomenuti skriveni sloj sa 100 neurona, koji je proslijeđen u idući sloj koji predstavlja rezultat mreže. Tom posljednjem sloju kao broj neurona proslijeđena je varijabla dimenzije prostora akcija koja osigurava da će output naše mreže odgovarati broju akcija, odnosno broju 2. U konstruktoru također je definirana funkcija gubitka kao varijabla loss, te se računa kao kvadrat razlike izračunate q vrijednosti akcije u stanju te ciljane vrijednosti. Ciljanu vrijednost izračunat će agent. Funkcija gubitka proslijeđena je optimizatoru koji će raditi s ciljem da ju minimizira. Optimizatoru određena je

brzina učenja kao 0.001. Ova vrijednost predstavlja iznos za koji će se u mreži prilagođavati vrijednosti u jednom koraku treniranja mreže. Iznos 0.001 najčešće se koristi, ali veće vrijednosti mogu brže donijeti rezultate uz rizik odabira manje optimalnog puta prerano. Korištenjem manjih vrijednosti agent će sporije učiti, ali zato neće vrlo rano pridavati preveliku važnost putevima koji nisu nužno najbolji. Funkcija get_q_state služi za dohvaćanje vektor q vrijednosti za akcije u proslijeđenom stanje tako da u Tensorflow sesiji pokrene model mreže spremljen u varijabli klase q_state uz proslijeđeno stanje. Tu funkciju agent koristi kada odabire akciju te kad računa ciljanu vrijednost. Funkcija update_model na temelju proslijeđenih vrijednosti pokreće optimizator kako bi trenirala mrežu, te ju agent poziva nakon što prikupi potrebne vrijednosti.

Nakon što smo opisali klasu koju će naš agent koristiti, potrebno je nadopuniti implementaciju našeg jednostavnog FSM agenta. Za početak potrebno mu je u on_start metodu inicijalizirati varijable koje će koristiti u radu s mrežom, kao i instancu same mreže. To ćemo postići tako da u on_start metodu dodamo sljedeće linije koda.

```
self.agent.state_dim = env.observation_space.shape
self.agent.action_size = env.action_space.n
self.agent.q_network = QNetwork(self.agent.state_dim,
self.agent.action_size)
self.agent.gamma = 0.97
self.agent.sess = tf.Session()
self.agent.sess.run(tf.global_variables_initializer())
```

Nakon inicijalizacije agent sada ima instancu QNetwork klase, vlastitu tensorflow sesiju te je inicijalizirao vrijednosti koje se nalaze u neuronskoj mreži. Također mu je definirana vrijednost game kako bi mogao kasnije računati ciljanu vrijednost pomoću Bellmanove jednadžbe. Dalje u stanju NewEpisode u kojem inicijaliziramo okružje, želimo da agent zapamti stanje koje mu bude vraćeno iz env varijable kao početno stanje. To stanje ćemo spremiti u agentovu varijablu current_state sljedećom naredbom.

```
self.agent.current_state = env.reset()
```

U stanju ChooseAction više neće nasumično odabirati akcije, već će koristiti funkciju get_q_state kako bi dobio q vrijednosti mogućih akcija za njegovo trenutno stanje. Od dobivenih q vrijednosti, odabrat će onu čija je vrijednost najveća, te vratiti njezin index kao akciju. Kako indeksi q vrijednosti odgovaraju vrijednostima koje predstavljaju akcije možemo ih kao takve koristiti. U stanju ChooseAction zamijenimo redak za nasumičan odabir akcije sa sljedećim:

```
q_state = self.agent.q_network.get_q_state(self.agent.sess,
[self.agent.current_state])
self.agent.action = np.argmax(q_state)
```

U stanju Learn ćemo sad omogućiti agentu ne samo da odigra akciju, već i da istrenira model mreže pomoću dobivenih informacija iz okružja. Nakon što agent odigra odabranu akciju pomoću metode get_q_state dobit će q vrijednost najbolje akcije u novom stanju, te pomoću te vrijednosti, vraćene nagrade i game koristeći Bellmanovu jednadžbu izračunat će ciljanu q vrijednost. U slučaju da je igra gotova, i znamo da neće biti idućeg stanja, varijablu koja sadrži q vrijednosti idućeg stanja postavit ćemo na 0 tako da ju pomnožimo s nulom ako je igra gotova, jer u tom slučaju možemo garantirati da neće biti budućih nagrada. Agent će zatim dobivenog iznosa ciljane q vrijednosti, novog stanja i poduzete akcije proslijediti metodi update_model kako bi trenirao mrežu s tim podacima. Nakon toga, ažurirat će vlastitu varijablu trenutnog stanja na novo stanje dobiveno nakon poduzimanja prethodne akcije. Kako bi agent mogao izvršavati navedeno, u stanje Learn potrebno je dodati sljedeće:

```
q_next_state = self.agent.q_network.get_q_state(self.agent.sess,
[next_state])
q_next_state = (1-done) * q_next_state
q_target = reward + self.agent.gamma * np.max(q_next_state)
self.agent.q_network.update_model(self.agent.sess,
[self.agent.current_state], [self.agent.action], [q_target])

self.agent.current_state = next_state
```

Na kraju potrebno je samo u stanju EndGame zatvoriti agentovu Tensorflow sesiju koja je otvorena u on_start metodi.

Agent kakvog smo sad implementirali ima mogućnost učenja na temelju svojih iskustava, te u skladu s time odabirati svoje akcije ovisno o stanju okružja. Ipak, ako pokrenemo novog i poboljšanog agenta, primijetit ćemo da ponekad postiže i lošije rezultate nego kada je samo nasumično odabrao akcije, te da se rezultati koje postiže ne mijenjaju kroz nove epizode već ostaju relativno konzistentni od početka. Ako ispišemo akcije koje on poduzima, možemo vidjeti da uvijek poduzima istu akciju, ili gotovo pravilno alternira između njih oboje. Na početku igre, agent još ne zna koje su optimalne akcije u kojem stanju okružja, no neuronska mreža koju on koristi kako bi odabrao svoju akciju inicijalizirana je s nekim nasumičnim početnim vrijednostima. Naš agent svakog puta odabire one akcije čije su q vrijednosti nasumično postavljene na veću vrijednost, te uopće ne istražuje ostale mogućnosti koje potencijalno mogu dovesti do dugoročno bolje nagrade. To je posljedica toga što se naš agent trenutno vodi pohlepnom (eng. *greedy*) politikom [34]. Zanima ga samo koja q vrijednost je najveća te će nju odabrati svakog puta. Kako bi smo dobili agenta koji ne pada u istu zamku, već istražuje druge potencijalno bolje mogućnosti u sljedećem potpoglavlju primijenit ćemo epsilon-greedy strategiju.

3.3.3. Epsilon-greedy strategija

Jedan od načina da prisilimo agenta da istraži mogućnosti u svom okružju je implementirajući epsilon-greedy strategiju. Strategija nalaže princip odabira akcija u kojem u početku agent češće odabire nasumične akcije i time stječe nova iskustva, a kasnije više se oslanja na neuronsku mrežu koju je kroz ta iskustva istrenirao. Epsilon-greedy strategija zamijenit će dosadašnju greedy strategiju, tako da postupak odlučivanja o akciji koju će agent poduzeti zamijenimo novom strategijom. Tu funkciju možemo matematički iskazati kao funkciju na sljedeći način [35]:

$$\pi(s) = \begin{cases} nasumična\ akcija & ako\ a < \varepsilon \\ aproksimirana\ optimalna\ akcija & ako\ a > \varepsilon \end{cases}$$

Iz funkcije možemo vidjeti da konačna odluka ovisi o nasumično odabranom broju a, te o broju epsilon (ε). Broj a nasumično se odabire prije svake odluke unutar intervala $0 \leq a \leq 1$. Broj ε je zadan broj unutar intervala $0 \leq \varepsilon \leq 1$, te se najčešće smanjuje s vremenom nakon što je agent dovoljno istražio okružje. Tim pristupom agenta se u početku prisiljava da istraži svoj okoliš, jer kada je ε bliže 1 veća je vjerojatnost da će a biti manji od ε te će agent postupiti na neki nepredvidivi način i naučiti nešto novo. Kasnije nakon što je agent istražio okoliš i poboljšao svoju aproksimaciju optimalne funkcije, konkretno q vrijednosti u neuronskoj mreži, ε se smanji kako bi se agentu dozvolilo da donosi odluke na temelju te aproksimacije.

Implementacija epsilon-greedy strategije na našem agentu vrlo je jednostavna. Prvo moramo agentu u on_start metodi dodijeliti varijablu koja sadržava vrijednost ε. Tu vrijednost ćemo u početku postaviti na 1.0 te kasnije smanjivati. Zatim u stanju ChooseAction moramo generirati nasumičan broj a pomoću random biblioteke. Taj broj zatim usporedimo s vrijednosti od ε, te u slučaju da je on manji vratimo nasumični broj 0 ili 1 kao odabranu akciju. U slučaju da je broj a već od ε, odaberemo akciju s najvećom q vrijednosti za dano stanje. Kako bi smo to implementirali u kodu, prethodni odabir akcije s najvećom q vrijednosti zamijenimo sa sljedećim kodom:

```
if random.random() < self.agent.eps:
    self.agent.action = np.random.randint(self.agent.action_size) #random
else: self.agent.action = np.argmax(q_state) #greedy
```

Preostalo je još samo smanjiti ε nakon svake epizode kako bi se agentu kasnije dozvolilo da odlučuje na temelju vrijednosti iz trenirane neuronske mreže. To možemo implementirati tako da u stanju NewEpisode varijablu epsilon pomnožimo s 0.99 kako bi se on postupno smanjivao kroz naredne epizode. Definirat ćemo i minimalnu vrijednost epsilona kao 0.01 kako bi naš

agent uvijek imao barem 1% šansu da istraži neku novu mogućnost u danoj situaciji. To implementiramo u kodu tako da u stanje NewEpisode na početku dodamo naredbu:

```
self.agent.eps = max(0.01, 0.99*self.agent.eps)
```

Nakon što smo implementirali epsilon-greedy strategiju, možemo vidjeti da naš agent u početku nasumično isprobava akcije te ne postiže neki značajan rezultat. Ipak rezultati koje postiže više nisu dosljedni, te ima epizoda u kojima postigne viši rezultat nego kada se vodio greedy politikom. Uz epsilon-greedy politiku naš agent dosljedno postiže rezultate više od 200 jedinica nagrade nakon što je odigrao prosječno 400 epizoda. Postizanjem dosljedno visokih rezultata nakon učenja na prethodnim iskustvima mogli bi smo smatrati da je naš agent uspješno naučio igrati Cartpole igru, no praćenjem rezultata kroz epizode vidimo da on uči relativno sporadično, i često ne ostaje dosljedan u svojim rezultatima. Ako želimo unaprijediti način na koji agent učit te poboljšati same performanse učenja možemo pimijeniti experience replay koncept. Taj koncept ćemo pojasniti i implementirati u sljedećem potpoglavlju.

3.3.4. Experience replay

Koncept korištenja experience replay obuhvaća pamćenje prethodnih iskustava, te nasumičan odabir prethodnih iskustava za treniranje neuronske mreže. Jedno iskustvo obuhvaća sve informacije vezane uz jedan korak agenta. Ono uključuje posljednje stanje i poduzetu akciju, te dobivenu nagradi i rezultirano sljedeće stanje. Niz takvih prikupljenih iskustava zapisuje se u memoriju, najčešće u obliku tablice. Jedno iskustvo možemo zapisati kao sljedeću uređenu četvorku [36]:

$$e_t = (s_t, a_t, r_{t+1}, s_{t+1})$$

Dok se pamćenje prethodnih iskustava čini kao dobra ideja, nasumičan odabir iskustava samo mijenja redoslijed kojim ih učimo te ne predstavlja odmah očitu prednost za poboljšanje procesa učenja. Razlog zašto to unaprjeđuje učenje jest da pomaže smanjiti korelaciju između uzoraka na kojima se uči [36]. Slijedno učenje na snažno koreliranim uzorcima iskustava može navesti agenta u jedan od lokalnih minimuma u slučaju kada se za redom javljaju slična iskustva za koje model tada prekompenzira. To možemo vidjeti kod našeg agenta koji kroz tijek svog učenja ima periode gdje iz epizode u epizodu preferira lijevu ili desnu stranu što ne dovodi do visokih dugoročnih nagrada. Nasumičnim odabirom iskustava na kojim se uči pomaže smanjit tu korelaciju. Druga prednost koju možemo dobiti korištenjem experience replaya proizlazi iz činjenice da naš agent sada pamti više od jednog iskustva. To znači da možemo koristiti više prethodnih iskustava odjednom prilikom treniranja naše mreže. Na taj način dobivamo više iz već postojećih iskustava i prije nego što agent stekne nova. U

vidu performansi, to može usporiti izvođenje programa no u kontekstu agenta on uči brže nakon manje izvedenih akcija. Taj pristup može imati i pozitivan učinak na performanse u slučajevima kada je stjecanje novih iskustava veće opterećenje od treniranje mreže na prethodnim iskustvima, te ako su ta iskustva iz nekog drugog razloga manje pristupačan resurs. Osim navedenog, još jedna prednost ovakvog pristupa jest pamćenje rijetkih iskustava iz kojih je dosad agent učio samo jednom i odbacio ih, no u ovom slučaju ima priliku učiti iz njih više puta.

Naš agent trenutno ne pamti prethodna iskustva, već ih jednom iskoristi kako bi trenirao neuronsku mrežu i zatim odbaci. Želimo agentu omogućiti da pohranjuje svoja iskustva te ih višestruko iskoristi kako bi iz njih dobio maksimalnu korist. Također želimo moći nasumično uzimati uzorke tih iskustava te ih koristiti za treniranje agentove neuronske mreže. Kako bi smo agentu omogućili pamćenje njegovih iskustava, koristit ćemo klasu preuzetu s githuba [33]. Implementacija te klase u kodu uz objašnjenje slijedi u nastavku.

```
class ReplayBuffer():
    def __init__(self, maxlen):
        self.buffer = deque(maxlen=maxlen)

    def add(self, experience):
        self.buffer.append(experience)

    def sample(self, batch_size):
        sample_size = min(len(self.buffer), batch_size)
        samples = random.choices(self.buffer, k=sample_size)
        return map(list, zip(*samples))
```

Klasa ReplayBuffer sprema stečena iskustava u varijablu buffer u obliku deque-a. Deque je objekt sličan listi, koji može sadržavati fiksan maksimalan broj elemenata. Kada se dosegne maksimalan broj elemenata, novi elementi se i dalje mogu dodavati no za svaki dodatni element briše se drugi sa suprotnog kraja liste. Klasa ima funkciju add() koja služi za dodavanje novih elemenata. Funkcija sample() služi za dohvaćanje određenog broja nasumično odabranih elemenata iz buffer varijable, te ih vraća u obliku zasebnih lista za stanja, akcije, nagrade i sljedeća stanja. To čini korištenjem zip funkcije na nasumično odabranim iskustvima te konvertiranjem ih u liste. U slučaju da u bufferu ima manje zapisa nego što je traženo, vraća samo toliko rezultata koliko ih se u njemu nalazi. Kako bi naš agent mogao koristiti klasu, u on_start metodi spremimo mu instancu te klase uz definiranu maksimalnu duljinu deque-a na 10 000 elemenata, koristeći sljedeću liniju koda.

```
self.agent.replay_buffer = ReplayBuffer(maxlen=10000)
```

Limitiranjem broja iskustava koje agent pamti osim što smanjujemo količinu memorije koju program koristi, agentu dajemo priliku da uči na novijim iskustvima. Kasnije kada agent nauči donositi bolje odluke, želimo da nastavi učiti na tim novijim iskustvima.

Nakon što agent ima instancu klase u koju će spremati iskustva, moramo mu to omogućiti u Learn stanju. Nakon što agent odigra odabranu akciju želimo da spremi sve informacije o tom iskustvu u ReplayBuffer klasu. Osim navedene četvorke koje opisuju iskustvo, naš agent nakon odrađene akcije također dobiva i podatak o tome da li je tom akcijom epizoda završena, te želimo da pamti i tu informaciju. Nakon što je agent spremio najnovije iskustvo, želimo da dobije skup od 50 nasumično odabranih iskustava iz memorije kako bi ih mogao koristiti za treniranje neuronske mreže i računanje ciljne funkcije. Za računanje ciljne funkcije možemo i dalje koristiti get_q_state metodu, samo joj moramo proslijediti sva dobivena nova stanja. Možemo napomenuti da nije potrebno mijenjati placeholdere definirane u klasi QNetwork iako sada spremamo po 50 podataka za jednu vrijednost odjednom jer nisu ograničene dimenzije podataka koje možemo unositi, tj. prva dimenzija oblika je u svim slučajevima None. Nakon što agent ima q vrijednosti za 50 mogućih sljedećih stanja, potrebno je provjeriti je li jedno od vraćenih stanja bilo posljednje u epizodi, u kojem slučaju njegovu je q vrijednost potrebno postaviti na 0 jer u tom slučaju ne može se očekivati buduća nagrada. To možemo postići tako da listu vrijednosti svih vraćenih elemenata q vrijednosti za buduća stanja indeksiramo pomoću liste dones koja sadrži bool vrijednosti ovisno o tome da li je to stanje bilo gotovo, te na indeksu gdje je ta bool vrijednost istinita postavimo q vrijednosti na nula. To možemo postići tako da ih izjednačimo s nulvektorom oblika [0,0] koji odgovara obliku prostora akcija. S obzirom da sada računamo više ciljanih vrijednosti za moguća buduća stanja, moramo ažurirati način na koji ih računamo koristeći Bellmanovu jednadžbu. Rezultat će sada biti lista q ciljanih vrijednosti koju dobivamo tako da svaki element iz liste nagrada zbrajamo s gamom množenom s maksimalnom q vrijednosti vektora pojedinog elementa iz liste q vrijednosti sljedećih stanja. Nakon što su izračunate q ciljane vrijednosti, funkciji update_model možemo proslijediti liste stanja i akcija dohvaćene iz memorije te pripadnu listu ciljanih vrijednosti kao bi se pomoću njih trenirao model neuronske mreže. Opisanu implementaciju ostvarujemo izmjenom koda u stanju Learn prema sljedećim.

```
self.agent.replay_buffer.add((self.agent.current_state, self.agent.action,
next_state, reward, done))

states, actions, next_states, rewards, dones =
self.agent.replay_buffer.sample(50)

q_next_states = self.agent.q_network.get_q_state(self.agent.sess,
next_states)

q_next_states[dones] = np.zeros([self.agent.action_size])
```

```
q_targets = rewards + self.agent.gamma * np.max(q_next_states, axis=1)

self.agent.q_network.update_model(self.agent.sess, states, actions,
q_targets)
```

Nakon što smo agentu omogućili da pamti svoja iskustva, te ažurira model neuronske mreže na temelju nasumičnog izbora od 50 iskustava u jednom koraku, možemo primijetiti da su se performanse učenja značajno poboljšale. Naš agent nakon već stotinjak epizoda postiže rezultate od približno 200 jedinica nagrade kakve je ranije postizao nakon prosječno 400 epizoda. Kroz agentovo učenje vidimo da postepeno ostvaruje sve bolji i bolji rezultat te je učenje manje sporadično. Nakon dvjestotinjak epizoda agent može postizati i maksimalan rezultat od 500 jedinica nagrade.

4. Zaključak

Umjetna inteligencija jedna je od najuzbudljivijih i brzorastućih znanosti našeg vremena. U ovom radu definirali smo ju kao razvoj agenata s inteligentnim sposobnostima koji racionalno i autonomno riješavaju probleme kakve su do sad rješavali samo ljudi. Kroz simbolički i statistički pristup nudi se široki spektar metoda razvijenih kroz desetljeća istraživanja za riješavanje takvih problema. Veliki broj tih metoda vuče temelje iz drugih znanosti. Simboličke metode temelje se na logici i teoriji vjerojatnosti te uz sposobnost agenta za reprezentaciju znanja mogu riješavati široki spektar problema odlučivanja čak i u uvjetima nesigurnosti, kakve nude problemi iz stvarnog svijeta. One predstavljaju temelj za racionalnog agenta koji autonomno može donositi oduke o svojem ponašanju. Statistički pristup obuhvaća metode koje se temelje na statistici te omogućavaju agentima učenje na golemim, ljudima ponekad i neshvatjlivim skupovima podataka. Strojno učenje metodama klasifikacije i regresije omogućava računalim sustavima učenje pojmova koje ne bi moglo biti moguće definirati klasičnim pristupom programiranja da li zbog ograničenja eksplicitne naravi programiranja ili ljudskog shvaćanja problema. Metodama klasteriranja u sklopu nenadgledanog strojnog učenja moguće je detektirati uzorke u podacima čija bi složenost i količina preopteretila bilokoju pojedinu osobu. Metode pojačanog učenja dozvoljavaju razvoj sustava odlučivanja bez eksplicitne definicije politike, već samo kroz nagrađivanje dobrih odluka. Naročiti napredak strojnom učenju dostignut je kroz primjenu umjetnih neuronskih mreža. Uz dostupnost velikih skupova podataka, razvijenih alata te dostatnih računalnih resursa danas metode dubokog učenja pronalaze široku primjenu u velikom broju industrija, te je rad s tim metodama pristupačniji nego ikad. Uz dosad postignuti napredak kontinuirano se pronalaze problemi koje je moguće riješiti primjenom metoda umjetne inteligencije.

Kroz praktičan primjer izrade agenta koji igra Cartpole igru pokazali smo primjer implementacije pojačanog učenja koristeći besplatno dostupne alate. Koristeći SPADE okvir za razvoj agenta, OpenAi Gym za rad s okružjem igre te Tensorflow biblioteke za rad s neuronskom mrežom implementirali smo FSM agenta koji je naučio igrati danu igru. Agent je kroz primjenu metode Q-učenja i neuronske mreže za optimizaciju q vrijednosti naučio donositi odluke o akcijama s kojima je uspješno naučio igrati igru bez su mu dane informacije o prirodi svog okružja te posljedici svojih akcija. Kroz primjenu epsilon-greedy strategije potaknuli smo ga na istraživanje svog okružja kako bi mogao učiti na što većem broju različitih iskustava. Implementacijom experience replay koncepta omogućili smo mu pamćenje iskustava u eksplicitnom obliku kako bi mogao iz njih više puta učiti. Kroz razvoj ovog agenta pokazali smo na kakve izazove se može naići kroz razvoj umjetne inteligencije koju ne možemo eksplicitno navoditi te što mogu biti odgovori na neke od tih izazova.

Popis literature

[1] S. Russel i P. Norvig, Artificial Intelligence: A Modern Approach, New Jersey: Pearson Education, Inc., 2010.

[2] D. Poole, A. Mackworth i R. Goebel, Computational Intelligence: A Logical Approach, New York: Oxford University Press, 1998.

[3] M. Schatten, *Višeagentni sustavi: Inteligentni agenti,* Varaždin: Fakultet organizacije i informatike, 2019..

[4] M. Schatten, »Novi val umjetne inteligencije,« Vidi, 26 11 2018. [Mrežno]. Available: https://m.vidi.hr/index.php/business-3-0/novi-val-umjetne-inteligencije. [Pokušaj pristupa 10 8 2020].

[5] A. Colmerauer i P. Roussel, »The birth of Prolog,« 11. 1992.. [Mrežno]. Available: http://alain.colmerauer.free.fr/alcol/ArchivesPublications/PrologHistory/19november92.pdf. [Pokušaj pristupa 29. 8. 2020.].

[6] P. Yadav, »Decision Tree in Machine Learning,« 13. 11. 2018.. [Mrežno]. Available: https://towardsdatascience.com/decision-tree-in-machine-learning-e380942a4c96. [Pokušaj pristupa 2. 9. 2020.].

[7] S. Ray, »6 Easy Steps to Learn Naive Bayes Algorithm with codes in Python and R,« 11. 9. 2017.. [Mrežno]. Available: https://www.analyticsvidhya.com/blog/2017/09/naive-bayes-explained/. [Pokušaj pristupa 3. 9. 2020.].

[8] M. Sidana, »Intro to types of classification algorithms in Machine Learning,« 28. 2. 2017.. [Mrežno]. Available: https://medium.com/sifium/machine-learning-types-of-classification-9497bd4f2e14. [Pokušaj pristupa 3. 9. 2020.].

[9] I. Dabbura, »K-means Clustering: Algorithm, Applications, Evaluation Methods, and Drawbacks,« Towards data science, 17. 9. 2018.. [Mrežno]. Available: https://towardsdatascience.com/k-means-clustering-algorithm-applications-evaluation-methods-and-drawbacks-aa03e644b48a. [Pokušaj pristupa 3. 9. 2020.].

[10] S. Yildirim, »DBSCAN Clustering — Explained,« Towards data science, 22. 4. 2020.. [Mrežno]. Available: https://towardsdatascience.com/dbscan-clustering-explained-97556a2ad556. [Pokušaj pristupa 3. 9. 2020.].

[11] S. Yildirim, »Hierarchical Clustering — Explained,« Towards data science, 3. 4. 2020.. [Mrežno]. Available: https://towardsdatascience.com/hierarchical-clustering-explained-e58d2f936323. [Pokušaj pristupa 3. 9. 2020.].

[12] M. Roux, »A comparative study of divisive hierarchical clustering,« 2015.. [Mrežno]. Available: https://www.google.hr/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&ved=2ahUKEwiCj9SDldPrAhUQ8KQKHbDoBbQQFjAMegQIBBAB&url=https%3A%2F%2Farxiv.org%2Fpdf%2F1506.08977&usg=AOvVaw1Z47iv5-5s6bpnIws1Hajw. [Pokušaj pristupa 3. 9. 2020.].

[13] M. G. Gumbao, »Best clustering algorithms for anomaly detection,« 3. 6. 2019.. [Mrežno]. Available: https://towardsdatascience.com/best-clustering-algorithms-for-anomaly-detection-d5b7412537c8. [Pokušaj pristupa 4. 9. 2020.].

[14] P. Wenig, »Local Outlier Factor for Anomaly Detection,« 5. 12. 2018.. [Mrežno]. Available: https://towardsdatascience.com/local-outlier-factor-for-anomaly-detection-cc0c770d2ebe. [Pokušaj pristupa 4. 9. 2020.].

[15] S. Bhatt, »Explaining Reinforcement Learning: Active vs Passive,« 2018.. [Mrežno]. Available: https://www.kdnuggets.com/2018/06/explaining-reinforcement-learning-active-passive.html. [Pokušaj pristupa 5. 9. 2020.].

[16] K. Strachnyi, »Brief History of Neural Networks,« 23. 1. 2019.. [Mrežno]. Available: https://medium.com/analytics-vidhya/brief-history-of-neural-networks-44c2bf72eec. [Pokušaj pristupa 5. 9. 2020.].

[17] »7 Types of Neural Network Activation Functions: How to Choose?,« Missinglink.ai, 2020.. [Mrežno]. Available: https://missinglink.ai/guides/neural-network-concepts/7-types-neural-network-activation-functions-right/. [Pokušaj pristupa 5. 9. 2020.].

[18] J. C. Palanca, A. G. Bada i M. E. Gregori, »A Jabber-based Multi-Agent System Platform,« 2006.. [Mrežno]. Available: https://www.researchgate.net/profile/Gustavo_Aranda/publication/221455273_A_jabber-based_multi-agent_system_platform/links/09e4150b7cc265afb9000000/A-jabber-based-multi-agent-system-platform.pdf?origin=publication_list. [Pokušaj pristupa 19. 8. 2020.].

[19] A. Barto, R. Sutton i C. Anderson, »Neuronlike Adaptive Elements That Can Solve Difficult Learning Control Problems,« 1983. [Mrežno]. Available: http://incompleteideas.net/papers/barto-sutton-anderson-83.pdf. [Pokušaj pristupa 14. 8. 2020.].

[20] »CartPole-v0 Github wiki,« OpenAI Inc., 2020.. [Mrežno]. Available: https://github.com/openai/gym/wiki/CartPole-v0. [Pokušaj pristupa 28 7 2020].

[21] J. Palanca, »SPADE documentation,« 2020.. [Mrežno]. Available: https://spade-mas.readthedocs.io/en/latest/readme.html. [Pokušaj pristupa 19. 8. 2020.].

[22] »SPADE API documentation,« Javi Palanca Revision, 2020.. [Mrežno]. Available: https://spade-mas.readthedocs.io/en/latest/spade.html. [Pokušaj pristupa 20. 8. 2020.].

[23] »Getting Started with Gym,« OpenAi, 2016.. [Mrežno]. Available: https://gym.openai.com/docs/. [Pokušaj pristupa 20. 8. 2020.].

[24] »OpenAi,« Crunchbase Inc., 2020.. [Mrežno]. Available: https://www.crunchbase.com/organization/openai. [Pokušaj pristupa 20. 8. 2020.].

[25] »OpenAi Gym Github README,« OpenAi, 2020.. [Mrežno]. Available: https://github.com/openai/gym. [Pokušaj pristupa 21. 8. 2020.].

[26] C. Metz, »Google Just Open Sourced TensorFlow, Its Artificial Intelligence Engine,« Code Nast, 11. 9. 2015.. [Mrežno]. Available: https://www.wired.com/2015/11/google-open-sources-its-artificial-intelligence-engine/. [Pokušaj pristupa 21. 8. 2020.].

[27] C. D. Costa, »Best Python Libraries for Machine Learning and Deep Learning,« Medium, 25. 3. 2020.. [Mrežno]. Available: https://towardsdatascience.com/best-python-libraries-for-machine-learning-and-deep-learning-b0bd40c7e8c. [Pokušaj pristupa 21. 8. 2020.].

[28] N. Ahmad i R. Gavali, »Which language should I use for tensorflow?,« Quora, 3. 2018.. [Mrežno]. Available: https://www.quora.com/Which-language-should-I-use-for-tensorflow. [Pokušaj pristupa 21. 8. 2020.].

[29] A. Baliuka, »Tensorflow performance (versions 1 vs 2 and CPU vs GPU),« Stackoverflow, 26. 8. 2019.. [Mrežno]. Available: https://stackoverflow.com/questions/57657651/tensorflow-performance-versions-1-vs-2-and-cpu-vs-gpu. [Pokušaj pristupa 21. 8. 2020.].

[30] »Advantages and Disadvantages of TensorFlow,« Techvidian team, 1. 8. 2020.. [Mrežno]. Available: https://techvidvan.com/tutorials/pros-and-cons-of-tensorflow/. [Pokušaj pristupa 21. 8. 2020.].

[31] S. Decubber, »Training and Serving ML models with tf.keras,« 17. 8. 2018.. [Mrežno]. Available: https://blog.tensorflow.org/2018/08/training-and-serving-ml-models-with-tf-keras.html. [Pokušaj pristupa 21. 8. 2020.].

[32] Tensorflow, »TensorFlow 2.0 is now available!,« 30. 9. 2019.. [Mrežno]. Available: https://medium.com/tensorflow/tensorflow-2-0-is-now-available-57d706c2a9ab. [Pokušaj pristupa 22. 8. 2020.].

[33] J. Brownlee, »A Gentle Introduction to Tensors for Machine Learning with NumPy,« 14. 2. 2018.. [Mrežno]. Available: https://machinelearningmastery.com/introduction-to-tensors-for-machine-learning/. [Pokušaj pristupa 22. 8. 2020.].

[34] Tensorflow, »Tensorflow 1.13 math dokumentacija,« 10. 6. 2019. [Mrežno]. Available: https://github.com/tensorflow/docs/tree/r1.13/site/en/api_docs/python/tf/math. [Pokušaj pristupa 22. 8. 2020.].

[35] Tensorflow, »Tensorflow optimizers,« 10. 6. 2019.. [Mrežno]. Available: https://github.com/tensorflow/docs/tree/r1.13/site/en/api_docs/python/tf/train. [Pokušaj pristupa 22. 8. 2020.].

[36] Tensorflow, »Tensorflow activation functions,« 10. 6. 2019.. [Mrežno]. Available: https://github.com/tensorflow/docs/tree/r1.13/site/en/api_docs/python/tf/nn. [Pokušaj pristupa 22. 8. 2020.].

[37] J. Brownlee, »A Gentle Introduction to the Rectified Linear Unit (ReLU),« 9. 1. 2019.. [Mrežno]. Available: https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/?fbclid=IwAR0l9bDyk1g99hzYedzYuN42vSREpNs5WUKcztcm5q1hFgqyuQaekSWf01k. [Pokušaj pristupa 24. 8. 2020.].

[38] J. Brownlee, »Gentle Introduction to the Adam Optimization Algorithm for Deep Learning,« 20. 8. 2020.. [Mrežno]. Available: https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/?fbclid=IwAR21LNQhx7jfbWboQwGVF649sa74572PCIrhgwAUkP2XyvTuS_hM-shLIKY. [Pokušaj pristupa 24. 8. 2020.].

[39] the-computer-scientist, »DeepQNetworksInOpenAIGym,« 1. 2. 2019.. [Mrežno]. Available: https://github.com/the-computer-scientist/OpenAIGym/blob/master/DeepQNetworksInOpenAIGym.ipynb. [Pokušaj pristupa 24. 8. 2020.].

[40] S. Zychlinski, »The Complete Reinforcement Learning Dictionary,« 23. 2. 2019.. [Mrežno]. Available: https://towardsdatascience.com/the-complete-reinforcement-learning-dictionary-e16230b7d24e#a76c. [Pokušaj pristupa 24. 8. 2020.].

[41] M. Tokic i G. Palm, »Value-Difference Based Exploration: Adaptive Control Between Epsilon-Greedy and Softmax,« 2011.. [Mrežno]. Available: http://www.tokic.com/www/tokicm/publikationen/papers/KI2011.pdf. [Pokušaj pristupa 26. 8. 2020.].

[42] »Replay Memory Explained - Experience for Deep Q-Network Training,« Deeplizard, 3. 11. 2018.. [Mrežno]. Available: https://deeplizard.com/learn/video/Bcuj2fTH4_4. [Pokušaj pristupa 26. 8. 2020.].

Popis slika

Popis tablica

Prilozi

Uz rad priložene su četiri datoteke koje sadrže python skriptu implementacije agenta. Prva datoteka naziva fsmSimple sadrži implementaciju jednostavnog FSM agenta kakav je opisan u poglavlju 3.3.1. Druga datoteka naziva fsmGreedy sadrži implementaciju agenta koji uči po principu Q-učenja uz pomoć neuronske mreže kakav je opisan u poglavlju 3.3.2. Treća datoteka je naziva fsmEps te sadrži agenta s implementiranom epsilon-greedy strategijom kakav je opisan u poglavlju 3.3.3. Četvrta datoteka sadrži posljednju verziju agenta kojem je implementiran experience replay koncept kao što je opisan u poglavlju 3.3.4.