SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 2857

Automatizirani razvoj agenata za različita okruženja

Paulo Sanković

Zagreb, lipanj 2022.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 11. ožujka 2022.

DIPLOMSKI ZADATAK br. 2857

Pristupnik:	**Paulo Sanković (0036509900)**
Studij:	Računarstvo
Profil:	Računarska znanost
Mentor:	doc. dr. sc. Marko Đurasević

Zadatak: **Automatizirani razvoj agenata za različita okruženja**

Opis zadatka:

Proučiti različita okruženja poput jednostavnih igara ili fizikalnih simulatora za koje je moguće razviti agente. Proučiti različite metode i modele poput umjetnih neuronskih mreža koje se mogu iskoristiti za upravljanje agentom u takvim okruženjima. Proučiti postojeća programska rješenja koja pružaju simulaciju pojedinih okruženja te istražiti mogućnosti povezivanja s takvim okruženjima. Razviti programski okvir koji omogućuje automatski razvoj agenata za odabrana okruženja te grafički prikazuje ponašanje razvijenih agenata. Ocijeniti efikasnost razvijenih agenata za odabrana okruženja. Predložiti moguća poboljšanja predloženog postupka s ciljem razvoja efikasnijih agenata. Radu priložiti izvorne programske kodove, dobivene rezultate uz potrebna objašnjenja i korištenu literaturu.

Rok za predaju rada: 27. lipnja 2022.

Zahvaljujem se mentoru doc. dr. sc. Marku Đuraseviću na pomoći, korisnim savjetima i razumijevanju tijekom izrade ovog diplomskog rada, kao i svim profesorima i kolegama s kojima sam proveo svoje osnovnoškolske, srednjoškolske i studentske dane. Posebno bih se zahvalio svojoj obitelji na pruženoj podršci i razumijevanju.

SADRŽAJ

1. Uvod **1**

2. Podržano učenje **2**

2.1. Ključni koncepti 3

2.2. Duboki modeli 5

2.2.1. Unaprijedni potpuno povezani modeli 5

2.2.2. Konvolucijski modeli 7

2.3. Algoritmi podržanog učenja 8

2.3.1. Q učenje 10

2.3.2. Duboko Q učenje 11

2.3.3. Dvostruko duboko Q učenje 13

2.3.4. Gradijent politike 15

2.3.5. Akter-kritičar 16

2.3.6. Prednosni akter-kritičar 17

3. OpenAI Gym **19**

3.1. Struktura 19

3.1.1. Okolina 20

3.1.2. Interakcija s okolinom 20

3.1.3. Prostor akcija i prostor stanja 21

3.1.4. Omotači 22

3.1.5. Vektorizirana okruženja 22

3.2. Determinizam atari okruženja 23

3.3. Okruženja 24

3.3.1. Okruženje CartPole 24

3.3.2. Okruženje Breakout 26

4. Stable Baselines3 **28**
4.1. Korištenje predtreniranih modela 28

5. Implementacija **30**
5.1. Moduli 30
5.1.1. Instanciranje dubokih modela 30
5.1.2. Serijalizacija i deserijalizacija agenata 32
5.1.3. Omotači 32
5.2. Implementacijski detalji 33
5.2.1. Duboko Q učenje 33
5.2.2. Dvostruko duboko Q učenje 36
5.2.3. Prednosni akter-kritičar 36
5.3. Okolina CartPole 38
5.3.1. Implementacija Deep Q Learning algoritma 39
5.3.2. Implementacija Double Deep Q Learning algoritma 39
5.3.3. Implementacija Actor Critic algoritma 39
5.3.4. Usporedba 39
5.4. Okolina Breakout 39
5.4.1. Implementacija Double Deep Q Learning algoritma 39
5.4.2. Implementacija Deep Q Learning algoritma 39
5.4.3. Implementacija Actor Critic algoritma 39
5.4.4. Usporedba 40
5.5. Usporedba algoritama 40

6. Daljnji rad **41**

7. Zaključak **43**

Literatura **44**

Popis slika **49**

Popis tablica **50**

Popis programskih isječaka **51**

1. Uvod

Inteligentni sustavi sposobni su analizirati, razumjeti i učiti iz dostupnih podataka putem posebno dizajniranih algoritama umjetne inteligencije. Zahvaljujući dostupnosti velikih skupova podataka, razvoja tehnologije i napretka u algoritmima došli smo do razine gdje nam razvijeni modeli mogu uvelike poslužiti u svakodnevnom životu.

Posebno je zanimljiva problematika u kojoj je bez znanja o pravilima i funkcioniranju specifične okoline, potrebno konstruirati specijaliziranog agenta koji se nalazi u određenom stanju okoline i ponavlja korake izvršavanja optimalne akcije i prijelaza u novo stanje okoline. Za svaku akciju agent prima određenu nagradu - mjeru koja označava koliko su akcije agenta prigodne i koliko je napredak agenta kvalitetan. Agent izvršava akcije i prelazi u nova stanja sve dok se ne nađe u terminalnom (završnom) stanju. Dakle, cilj agenta u okolini jest pronaći optimalnu strategiju koja će maksimizirati očekivanu dobit (nagradu) u određenom vremenskom okviru.

Područje strojnog učenja koje se bavi prethodno navedenom problematikom naziva se podržano učenje (engl. *reinforcement learning*). Agenti koji se prilično dobro ponašaju u takvim okolinama možemo implementirati pomoću različitih algoritama podržanog učenja koji se temelje na umjetnim neuronskim mrežama (engl. *artificial neural networks*). Umjetne neuronske mreže su dobri aproksimatori funkcija i najbolje se ponašaju u okolini koja ima kompozitnu strukturu gdje vrlo kvalitetno duboki model predstave kao slijed naučenih nelinearnih transformacija.

U sklopu ovog rada bilo je potrebno proučiti i razumjeti metode i algoritme podržanog učenja i funkcioniranje umjetnih neuronskih mreža. Nadalje, bilo je potrebno istražiti, proučiti i naposljetku implementirati neke od algoritama podržanog učenja koji se zasnivaju na umjetnim neuronskim mrežama i koje je trebalo uklopiti u okoline koje su prikladne za simulaciju i testiranje ponašanja naučenih agenta.

Programsko rješenje implementirano je u programskom jeziku *Python*, primarno koristeći *PyTorch* biblioteku (engl. *library*), te ostale korisne biblioteke poput *numpy*, *cv2*, *stable-baselines3*, *tensorboard*... Za simulaciju i testiranje ponašanja agenata u posebnom okruženju korištena je biblioteka *OpenAI Gym*.

2. Podržano učenje

Strojno učenje (engl. *Machine learning*) je grana umjetne inteligencije (engl. *artificial inteligence*) koje se može definirati kao skup metoda koje u podatcima mogu automatski otkrivati obrasce, i potom te otkrivene obrasce iskorištavati pri budućem predviđanju podataka, ili obavljati druge zadatke odlučivanja u prisustvu nesigurnosti [39]. Drugim riječima, bez eksplicitnog programiranja moguće je napraviti sustave koji funkcioniraju kao ljudski mozak - imaju pristup podatcima, koriste ih za učenje i samim time bolje razumiju entitete, domene i veze između podataka.

Strojno učenje dijeli se na 3 podvrste: nadzirano učenje, nenadzirano učenje i podržano (ojačano) učenje. Nadzirano učenje (engl. *supervised learning*) karakterizira učenje modela nad testnim podatcima koji su označeni. Model točno zna da za određeni ulaz mora vratiti izlaz koji je istovjetan unaprijed pridruženoj oznaci. Algoritam mjeri točnost kroz funkciju gubitka, prilagođavajući se sve dok se izračunata razlika izlaza modela i stvarnog izlaza (pogreška) ne smanji do određene mjere. S druge strane, u nenadziranom učenju (engl. *unsupervised learning*) posjedujemo podatke bez zadanog izlaza - podatci su dani bez ciljne vrijednosti i u tim situacijama treba pronaći određenu pravilnost. Postupci poput grupiranja, smanjenja dimenzionalnosti, otkrivanja veza između primjeraka... pripadaju nenadziranom učenju.

Posebna i nama najzanimljivija podvrsta strojnog učenja jest podržano učenje (engl. *reinforcement learning*). Podržano učenje bavi se optimizacijom ponašanja agenta koji je u interakciji s okolinom (u kojoj se nalazi) i koji izvršava akcije na temelju informacija koje dobiva iz okoline. Agent pri svakom koraku od okoline dobiva povratnu informaciju u obliku nagrade ili kazne. Za razliku od prethodne dvije navedene podvrste koje mapiraju ulazne podatke na određeni format izlaza, u podržanom učenju je najizraženije učenje iz iskustva koje je čovjeku i drugim živim bićima veoma blisko.

2.1. Ključni koncepti

Za potpuno razumijevanje podržanog učenja, bitno je u navesti i pojasniti ključne koncepte i terminologiju. Okolina (engl. *environment*) označava svijet u kojem se agent nalazi i s kojim interaktira. Stanje s_t (engl. *state*) reprezentira kompletni opis okoline u određenom trenutku t. S druge strane, opservacija okoline o_t (engl. *observation*) predstavlja prilagođeni i ograničeni opis okoline koji agent dobije u nekom trenutku. Kada se agent nađe u nekom stanju s_t i kada od okoline dobije opservaciju o_t, tada agent poduzima akciju a_t (engl. *action*) i samim time u idućem vremenskom koraku inicira promjenu stanja s_{t+1} i opservacije okoline o_{t+1}.

Način na koji agent odabire akciju iz skupa svih dostupnih akcija naziva se politika (engl. *policy*). To je funkcija koja definira ponašanje agenta. Politika π može biti deterministička ili stohastička. Veza između akcije, determinističke politike i stanja predstavljena je izrazom 2.1 [22]. U našem slučaju koristit ćemo politike predstavljene dubokim modelima - aproksimacije funkcije odluke čiji se parametri uče optimizacijskim algoritmima.

$$a_t = \pi(s_t) \tag{2.1}$$

Nadalje, putanja τ (engl. *trajectory*) je pojam koji označava niz stanja i pripadajućih akcija i predstavljen je izrazom 2.2 [11]. Interakcija s okolinom započinje u trenutku $t = 0$ kada pomoću funkcije za inicijalizaciju okoline ρ_0 poštujući pravila okoline, nasumično generiramo stanje s_0.

$$\tau = (s_0, a_0, s_1, a_1, ...) \tag{2.2}$$

Najvažnija i najkorisnija informacija koju agent dobiva od okoline jest nagrada r_t (engl. *reward*) koju generira funkcija R (engl. *reward function*) i koja u obzir uzima trenutno i iduće stanje te akciju koja je izazvala promjenu stanja. Povezanost između generiranja iznosa nagrade i same nagrade prikazana je izrazom 2.3 [11].

$$r_t = R(s_t, a_t, s_{t+1}) \tag{2.3}$$

Želimo dobiti što bolji pregled koliko su bile dobre akcije koje je agent poduzeo. Cilj agenta je dobiti što više nagrada, odnosno maksimizirati povrat (engl. *return*)

$R(\tau)$ u svakom koraku [22]. Tu informaciju možemo predstaviti na dva različita načina. Sumom nekorigiranih nagrada zbrajamo samo nagrade koje su dobivene u fiksnom vremenskom intervalu T (izraz 2.4) [11]. Na taj način dobivamo informaciju o tome kolika je bila prosječna nagrada u zadnjih T koraka. Ako pak želimo pokazati da su nam nagrade u trenutnom stanju vrjednije nego nagrade koje ćemo dobiti u budućim stanjima, moramo uvesti korekcijski faktor (diskontni faktor) $\gamma \in (0, 1)$ (engl. *discount factor*). Navedeni pristup sumiranja korigiranih nagrada prikazan je izrazom 2.5 [11].

$$R(\tau) = r_0 + r_1 + ... + r_T = \sum_{t=0}^{T} r_t \tag{2.4}$$

$$R(\tau) = r_0 + \gamma \cdot r_1 + \gamma^2 \cdot t_2 + \gamma^3 \cdot t_3 + ... = \sum_{t=0}^{\infty} \gamma^t r_t \tag{2.5}$$

U našem slučaju agent interaktira sa stohastičkom okolinom i stoga funkciju politike (engl. *policy function*) (izraz 2.6) predstavljamo kao probabilističku funkciju koja u obzir uzima stanje i akciju, i vraća vjerojatnost poduzimanja dane akcije a_t u stanju s_t.

$$\pi(s, a) = P(a_t = a \mid s_t = s) \tag{2.6}$$

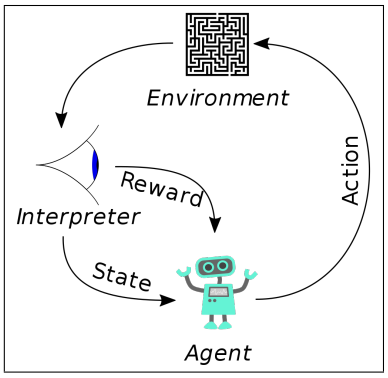
Za slučaj kada agent odabire putanju τ koja je u skladu sa politikom π počevši od stanja s, moguće je procijeniti očekivani povrat $R(\tau)$ [22]. Riječ je o funkciji vrijednosti stanja (engl. *value function*) (izraz 2.7) koja nam u suštini govori koliko je dobro biti u određenom stanju poštujući politiku [11]. Stanja vrednujemo prema tome koliko nagrade očekujemo da slijedi iz njih [22].

$$V^{\pi}(s) = \mathop{\mathbb{E}}_{\tau \sim \pi} [R(\tau) \mid s_0 = s] \tag{2.7}$$

S druge strane, imamo funkciju vrijednosti akcije (engl. *action-value function*) (izraz 2.8) [11] koja predstavlja procjenu očekivanog povrata $R(\tau)$ za slučaj da se agent nalazi u stanju s i poduzme akciju a (koja možda nije dio politike) i zatim uvijek postupa prema politici π [22].

$$Q^{\pi}(s, a) = \mathop{\mathbb{E}}_{\tau \sim \pi} [R(\tau) \mid s_0 = s, a_0 = a] \tag{2.8}$$

Dakle, cilj podržanog učenja jest pronaći optimalnu funkciju vrijednosti stanja ili funkciju vrijednosti akcije (ovisno o vrsti algoritma) te na taj način dobiti optimalnu politiku koja maksimizira povrat. U svakom koraku interakcije agenta s okolinom, agent prima opis stanja okoline u kojoj se nalazi. S obzirom na to stanje, izvršava akciju koja vrši neku promjenu nad okolinom i prebacuje ju u novo stanje. Agent prima povratnu informaciju od okoline koja reprezentira koliko je odabrana akcija dobra.



Slika 2.1: Ciklus interakcije agenta s okolinom [35]

2.2. Duboki modeli

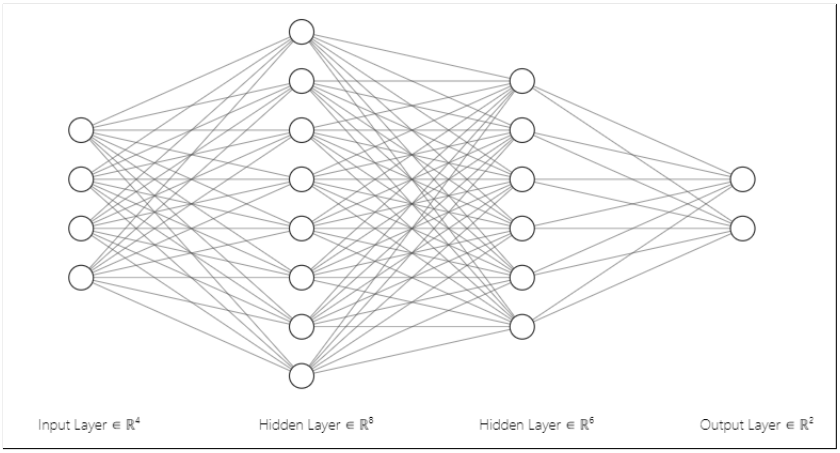
Duboko učenje (engl. *Deep learning*) jest tip strojnog učenja (točnije, podskup strojnog učenja) koje nastoji oponašati način zaključivanja i obrasce koje ljudski mozak koristi za učenje i donošenje odluka. Veliku ulogu u cijeloj ideji dubokog učenja imaju duboke neuronske mreže (engl. *deep neural networks, DNN*) pomoću kojih se povezivanjem više slojeva procesnih elemenata (čvorova, neurona), dobivaju duboki modeli koji su sposobni učiti i baratati s podatcima kompozitne strukture. Primjenom dubokih modela dolazimo do slijeda naučenih nelinearnih transformacija kojima aproksimiramo funkciju odluke, učimo mapiranje između ulaznih podataka i izlazih podataka, te nastojimo postići dobru generalizaciju nad stvarnim podatcima [19].

2.2.1. Unaprijedni potpuno povezani modeli

Unaprijedni potpuno povezani modeli (engl. *Fully connected neural network*) (poznatiji i pod nazivom višeslojni perceptron (engl. *Multi-layer perceptron*)) sastoje se od lanaca potpuno povezanih slojeva. Svaki neuron iz prethodnog sloja povezan je s neuronom idućeg sloja.

Sastoje se od tri vrste slojeva - ulaznog sloja, izlaznog sloja i skrivenih slojeva.

Ulaznom sloju dovode se podatci koje je potrebno obraditi. Izlaz neuronske mreže (u najosnovnijem obliku) predstavljen je logitima (engl. *logits*) - vektorima logaritama nenormaliziranih vrijednosti. Specifičnije, za slučaj da želimo provesti klasifikaciju podataka ili drugačije organizirati izlazne vrijednosti na izlaz dodajemo posebni sloj (npr. *Softmax* funkcija za klasifikaciju). Samo su ulaz i izlaz specificirani dimenzijama. Model ima slobodu da iskoristi skrivene slojeve na način koji osigurava najbolju aproksimaciju funkcije. Neuronskim mrežama želimo izgraditi modele koji nisu linearno odvojivi i zato koristimo nelinearnu aktivacijsku funkciju - najčešće ReLU (Rectified Linear Unit). Svaki od slojeva modelira jednu nelinearnu transformaciju.



Slika 2.2: Arhitektura potpuno povezanog modela

Slika 2.2 [23] prikazuje arhitekturu potpuno povezanog modela koji je sastavljen od sveukupno 4 potpuno povezana sloja - ulaznog (dimenzije 4), izlaznog (dimenzije 2) i dva skrivena sloja (svaki dimenzije 8). Kodom 1 prikazana je implementacija navedenog modela u biblioteci *PyTorch*.

```
import torch.nn as nn

model = nn.Sequential(
    nn.Linear(in_features=4, out_features=8, bias=True),
    nn.ReLU(),
    nn.Linear(in_features=8, out_features=6, bias=True),
    nn.ReLU(),
    nn.Linear(in_features=6, out_features=2, bias=True),
)
```

Kôd 1: Implementacija potpuno povezanog modela na slici 2.2 koristeći biblioteku *PyTorch*

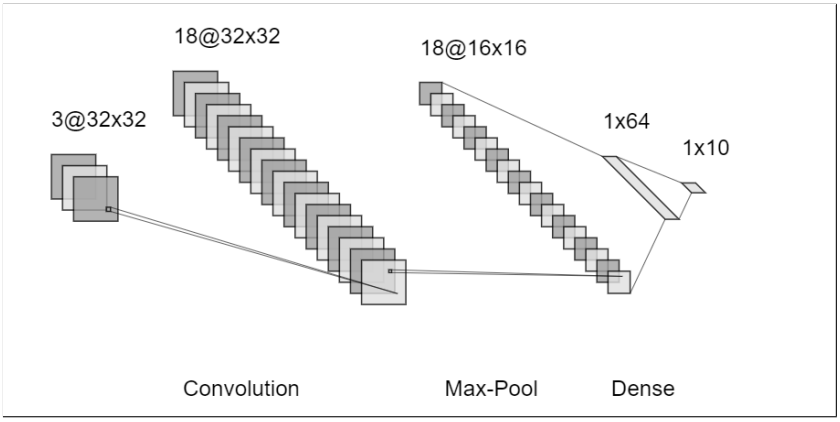
2.2.2. Konvolucijski modeli

Konvolucijski modeli (engl. *Convolutional neural networks*) su modeli koji uz potpuno povezane slojeve imaju najmanje jedan konvolucijski sloj (engl. *convolution layer*). Osim spomenutih slojeva, konvolucijski modeli sadrže i slojeve sažimanja (engl. *pooling layers*), slojeve u kojima provodimo nelinearnost podataka te sloj koji višedimenzionalni ulaz pretvara u jednodimenzionalni i pritom priprema podatke za obradu u potpuno povezanim slojevima (engl. *flatten layer*).

Operacija konvolucije provodi se nad ulazom i jezgrom (engl. *kernel*) (slobodnim parametrima koje učimo) gdje kao rezultat dobivamo mapu značajki koja pokazuje gdje se koja značajka nalazi u ulaznim podatcima (npr. slici). Dimenzije mapa značajki i njihovu robusnost korigiramo korištenjem atributa koraka konvolucije (engl. *stride*) i nadopunjavanja ulaznih podataka (engl. *padding*). Slično konvolucijskom sloju, sloj sažimanja odgovoran je za smanjenje prostora značajki (smanjenje dimenzionalnosti podataka) i dodatno za izdvajanje dominantnih značajki. Razlikujemo dvije vrste sažimanja: sažimanje maksimalnom vrijednosti (engl. *max pooling*) i sažimanje srednjom vrijednosti (engl. *average pooling*). Prilikom sažimanja značajki maksimalnom vrijednošću u obzir uzimamo samo značajku najveće vrijednosti te na taj način uklanjamo šum ulaza i potencijalno biramo značajku najveće važnosti.

Korištenje konvolucijskih modela biti će nam iznimno potrebno u situacijama kada su ulazni podatci u formi slike, odnosno kada su nam važne lokalne interakcije između ulaznih podataka (piksela) te njihova vremenska i prostorna ovisnost.

Slika 2.3 [23] prikazuje jednostavni konvolucijski model koji se sastoji od konvolucijskog sloja, sloja sažimanja maksimalnom vrijednosti, sloja koji 3-dimenzionalne podatke pretvara u 1-dimenzionalne, te dva potpuno povezana sloja. Ulaz u konvolucijski sloj predstavlja RGB slika (3 kanala) dimenzije 32×32. Primjenom konvolucije (veličina jezgre 3, korak 3, nadopuna 1) izvlačimo 18 kanala značajki dimenzije 32×32 (dimenzija se nije promijenila iz razloga što nadopunjavamo ulaz). Primjenom sažimanja maksimumom (veličina jezgre 2, korak 2) smanjujemo broj značajki na dimenziju 16×16. Prvi potpuno povezani sloj na svoj ulaz dobije vektor dimenzije 4608 kojeg pretvara u vektor izlaza dimenzije 64. Posljednji potpuno povezani sloj koji je ujedno i posljednji sloj u ovom konvolucijskom modelu za izlaz predaje vektor dimenzije 10. Isječak koda koji prikazuje implementaciju jednostavnog konvolucijskog modela koristeći bibilioteku *PyTorch* prikazan je kodom 2.



Slika 2.3: Arhitektura konvolucijskog modela

```
import torch.nn as nn

model = nn.Sequential(
    nn.Conv2d(in_channels=3, out_channels=18, kernel_size=(3, 3),
    ↪  stride=(1, 1), padding=1),
    nn.MaxPool2d(kernel_size=2, stride=2),
    nn.Flatten(),
    nn.Linear(in_features=18 * 16 * 16, out_features=64),
    nn.Linear(in_features=64, out_features=10)
)
```

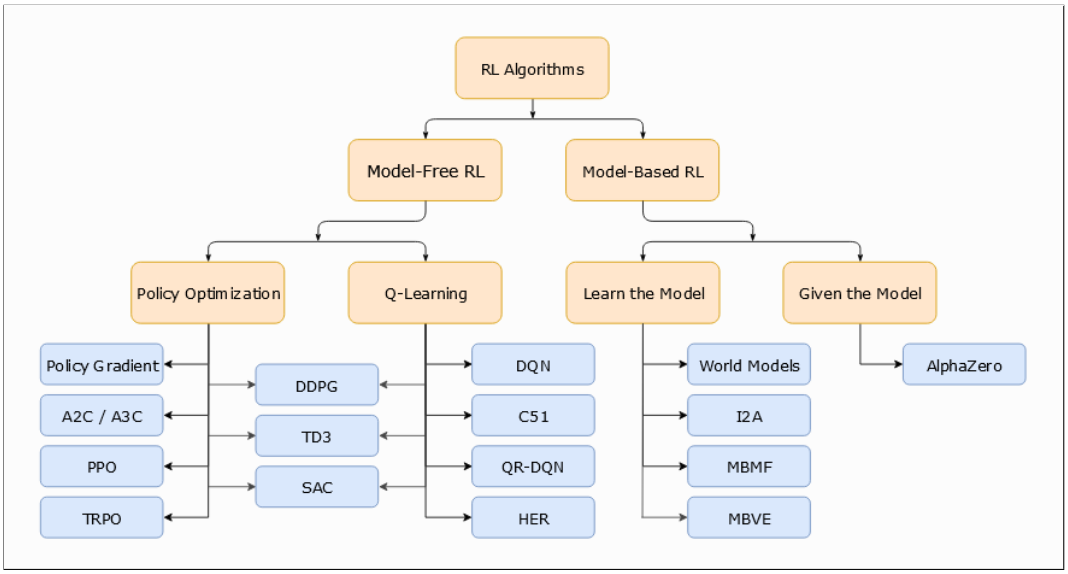
Kôd 2: Implementacija konvolucijskog modela na slici 2.3 koristeći biblioteku *PyTorch*

2.3. Algoritmi podržanog učenja

Način na koji pristupamo problemu pronalaženja optimalne politike prvenstveno ovisi o informacijama koje okolina pruža agentu i o tome je li poznato unutarnje djelovanje okoline - ima li agent pristup modelu okoline (pristup internoj funkciji okoline koja predviđa prijelaze stanja i nagrade). Pritom razlikujemo pristupe koji su temeljeni na modelu - poznaju funkciju predviđanja (engl. *model-based methods*) i pristupe koji okolinu promatraju kao crnu kutiju i pritom nisu upoznati s njenim pravilima niti principima funkcioniranja (engl. *model-free methods*).

Podržano učenje često pokušava riješiti probleme iz svakodnevnog života čiji temeljni model okruženja nije dostupan ili ga je pak teško implementirati. Samim time, nije u potpunosti praktično razmatrati postupke koji se temelje na modelu, koji svoje

korake mogu unaprijed planirati i pritom ne zahtijevaju interakciju s okolinom. S druge strane, metode koje u obzir ne uzimaju model moraju jedino znati točnu reprezentaciju stanja okoline i skup akcija koje agent može poduzimati. Gruba podjela algoritama podržanog učenja prikazana je na slici 2.4



Slika 2.4: Gruba podjela algoritama podržanog učenja [9]

Postoji i podjela algoritama s obzirom na to koje akcije agent izvršava prilikom učenja. Ako agent prilikom učenja izvršava akcije koje su dio trenutne politike koju prati, riječ je o učenju s uključenom politikom (engl. *on-policy*). S druge strane, učenje s isključenom politikom (engl. *off-policy*) znači da agent prilikom učenja može odabrati akcije koje nisu dio trenutne politike. Na taj način agent uči o više politika dok slijedi jednu [22].

Svaki algoritam strojnog učenja definiran modelom, gubitkom i metodom optimizacije [40]. Model je postupak obrade (odnosno skup funkcija) sa slobodnim parametrima koji za određen ulaz daje pripadajući izlaz. Gubitak je mjera koja na formaliziran način vrednuje slobodne parametre modela, odnosno pokazuje u kojoj mjeri se mi ne slažemo s onim što je model predstavio kao izlaz. Metoda optimizacije (optimizacijski postupak) jest način na koji pronalazimo optimalne parametre koji su važni kako bi minimizirali prethodno navedenu komponentu - gubitak. Navedene tri glavne komponente biti će važno napomenuti pri svakom predstavljaju algoritma jer su to glavne odrednice pri analizi algoritama strojnog učenja.

2.3.1. Q učenje

Algoritam Q učenje (engl. *Q Learning*) pripada skupini algoritama s isključenom politikom, jer uči iz akcija koje nisu dio trenutne politike. Algoritam procjenjuje kvalitetu (engl. *quality*) poduzimanja određene akcije i govori koliko je ona korisna pri dobivanju buduće nagrade. Osnovna inačica algoritma koristi jednostavnu strukturu podataka zvanu Q tablica (engl. *Q Table*) i pogodna je za korištenje u okolinama s ograničenim i relativno malim brojem stanja i akcija [24].

Q tablica sastoji se uređenih dvojki $[stanje, akcija]$ i pridruženih vrijednosti (Q vrijednosti) koje su prvotno inicijalizirane na 0. Izgled ove strukture prikazan je na slici 2.5 (gornja struktura, pridružena algoritmu Q učenja). Učenje se provodi iterativno: odabire se i izvršava akcija koja u trenutnom stanju ima najveću Q vrijednost, izračunava se nagrada, i na posljetku, uz pomoć Bellmanove jednadžbe izračunava se nova Q vrijednost 2.9. Na kraju učenja, Q tablica nam zapravo pokazuje koja je najbolja akcija koju možemo poduzeti u određenom stanju.

Kako bi optimizirao politiku, agent ima opciju istraživanja novih stanja i maksimiziranja svoje ukupne nagrade. Mora pronaći kompromis između istraživanja (odabira nasumične akcije) (engl. *exploration*) i odabira akcije s najvećom Q vrijednošću (engl. *exploitation*) [24]. Najbolja strategija bila bi uvesti hiperparametar ϵ koji označava vjerojatnost nasumičnog odabira akcije (umjesto odabira najbolje akcije) kojim bi se model kratkoročno žrtvovao ali sve u svrhu donošenja najbolje cjelokupne odluke u budućnosti. Navedena tehnika naziva se *epsilon-greedy exploration strategy*.

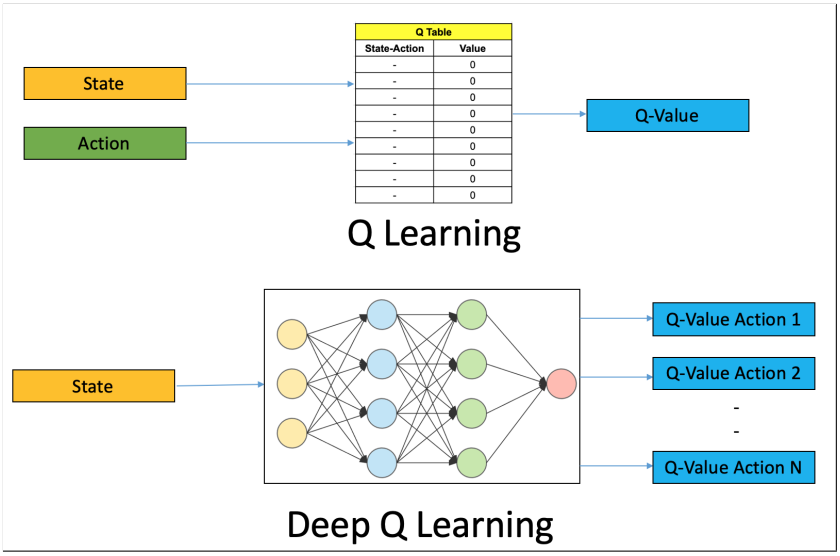
Uzevši u obzir prethodno navedene činjenice, pri početku treniranja akcije biramo nasumično kako bi agent mogao istražiti okolinu. Daljnjim treniranjem, smanjujemo vrijednost hiperparametra ϵ i samim time smanjujemo utjecaj strategije nasumičnog istraživanja i povećavamo utjecaj odabira akcije s najvećom Q vrijednošću. Na taj način, agent sve više koristi informacije koje je skupio prethodnim istraživanjem.

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha \left[r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right] \tag{2.9}$$

Prikazana Bellmanova jednadžba 2.9 iskazuje ukupnu korisnost poduzimanja akcije a_t kada se nalazimo u stanju s_t [22]. Sastoji se od nekoliko hiperparametara. Koeficijent α označava stopu učenja (engl. *learning rate*), parametar s kojim iskazujemo u kojoj mjeri želimo da nova vrijednost pridonese ukupnom zbroju vrijednosti. Koeficijent γ predstavlja korekcijski faktor (engl. *discount factor*) koji je već predstavljen u poglavlju Ključni koncepti.

Algoritam Q učenja koristi postupak učenja s vremenskom razlikom [22] (engl. *Temporal difference learning*) koji uče smanjivanjem odstupanja između procjena u različitim vremenskim trenucima [18]. Dakle, ažurirana procjena Q vrijednosti sastoji se od zbroja trenutne procjene i skalirane korekcijske vrijednosti (engl. *temporal difference error*). Korekcijska vrijednost je razlika između ciljane vrijednosti Q učenja (engl. *temporal difference target*) te procjene trenutne vrijednosti i u suštini govori u kojoj mjeri se trenutna vrijednost treba ažurirati kako bi smanjili odstupanje između procjene Q vrijednosti susjednih stanja. Ukupnu nagradu nastojimo procijeniti zbrojem trenutne nagrade (dobivene izvođenjem akcije a_t) i skalirane optimalne vrijednosti buduće akcije (ciljane politike).

Kao što je već rečeno, algoritam Q učenja pripada skupini algoritama s isključenom politikom i to je upravo vidljivo iz dijela Bellmanove jednadžbe. Pri postupku učenja, neovisno o tome koja se trenutno politika slijedi, pri računanju uzimamo u obzir buduću akciju čija je Q vrijednost optimalna. Ta akcija može, ali i ne mora biti dio politike koja se trenutno slijedi.



Slika 2.5: Razlika u strukturama koje koriste Q učenje i Duboko Q učenje [14]

2.3.2. Duboko Q učenje

Korištenje Q tablica nije pogodno za kompleksne okoline, okoline s velikim brojem stanja i akcija. U takvim okolinama koristi se inačica Q učenja, duboko Q učenje (engl. *Deep Q Learning*), čija je ideja, opis, niz objašnjenih formula i algoritam predstavljen u članku [27]. Osnovna ideja ovog algoritma jest naučiti duboki model da za određeno stanje okoline na ulazu, generira skup svih akcija i pripadajućih vrijednosti

funkcije stanja i akcije, i pritom izabere akciju s pripadajućom najvećom vrijednošću $Q(s, a)$ funkcije 2.8. Slika 2.5 upravo pokazuje tu razliku u korištenim strukturama za generiranje i procjenu vrijednosti funkcije kvalitete.

U procesu učenja koriste se 2 neuronske mreže jednake arhitekture i različitih vrijednosti težina parametara. Razlikujemo neuronsku mrežu koja prati trenutnu politiku (i pritom provodi stalno ažuriranje parametara) (engl. *policy neural network, online neural network*) i neuronsku mrežu koja prati ciljanu politiku (engl. *target neural network*). Svakih N koraka vrijednosti težina parametara ciljne neuronske mreže zamjenjuju se vrijednostima parametara mreže koja prati trenutnu politiku. Samo treniranje (optimizacija parametara) provodi se nad mrežom koja prati trenutnu politiku. Korištenje dviju neuronski mreža pospješuje stabilnost i učinkovitost procesa učenja. Druga neuronska mreža služi nam pri izračunu ciljane vrijednosti q učenja. Njene parametre ne optimiziramo. Samim time osigurava se da izračun ciljane vrijednosti Q učenja bude stabilan, sve dok se njeni parametri zamijene parametrima učene mreže.

Funkciju kvalitete $Q(s, a)$ aproksimirat ćemo neuronskom mrežom. Stoga, prikladno ju je zapisati kao funkciju parametriziranu težinama neuronske mreže, θ 2.10 [18]. Nadalje, operacija ažuriranja vrijednosti funkcije akcije pomoću neuronskih mreža može se izraziti jednadžbom 2.11 [5]. Funkcija gubitka 2.12 predstavljena je kvadratom razlika ciljane vrijednosti Q učenja (koju aproksimira ciljna neuronska mreža u ovisnosti o njenim težinama θ^-) i procjene trenutne vrijednosti akcije (koju aproksimira trenutna neuronska mreža u ovisnosti o njenim težinama θ). Vrijednosti su nasumično uniformno uzorkovane iz prethodnog iskustva D (točnije, iz spremnika za ponavljanje) [5].

$$Q(s, a) \approx Q(s, a; \theta) \tag{2.10}$$

$$Q(s_t, a_t; \theta) = Q(s_t, a_t; \theta) + \alpha \, \mathbb{E}\left[r_t + \gamma \max_a Q(s_{t+1}, a; \theta^-) - Q(s_t, a_t; \theta)\right] \tag{2.11}$$

$$\mathcal{L}(\theta) = \mathop{\mathbb{E}}_{(s,a,r,s') \sim U(D)} \left[\left(r_t + \gamma \max_a Q(s_{t+1}, a; \theta^-) - Q(s_t, a_t; \theta)\right)^2\right] \tag{2.12}$$

Grupno (engl. *batch*) učenje pozitivnije utječe na optimizacijski postupak od pojedinačnog učenja, zbog preciznijeg gradijenta. Uzorci za grupu trebali bi biti odabrani slučajnim uzorkovanjem kako bi bili nezavisni (želimo odrediti nepristranu procjenu gradijenta) [38]. To nam je motivacija za uvođenje metode ponavljanja iskustva (engl.

experience replay). U spremnik za ponavljanje (engl. *replay buffer*) pohranjuju se sve akcije, stanja i nagrade koje je agent poduzeo od početka. Konačno, pri svakoj iteraciji postupka učenja, iz liste se izvlači određeni broj uzoraka (grupa).

Algoritam 2.6 prikazuje pseudokod dubokog Q učenja s metodom ponavljanja iskustva, dvije neuronske mreže za aproksimaciju Q funkcija, zamjenu vrijednosti težina parametara, metodu ponavljanja iskustva, izračun funkcije gubitka.

Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
For episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For t = 1,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

Slika 2.6: Algoritam dubokog Q učenja s ponavljanjem iskustva [27]

2.3.3. Dvostruko duboko Q učenje

Algoritam dubokog Q učenja ima tendenciju precjenjivanja vrijednosti funkcije akcije. Odabir akcije koja ju maksimizira i procjenu njene vrijednosti (u ovisnosti o akciji odabranoj u prethodnom koraku), provodi se samo nad vrijednostima težinskih parametara θ^- ciljne neuronske mreže, kao što je i prikazano jednadžbom 2.13 [6]. Precijenjene vrijednosti ne predstavljaju problem u slučajevima kada su sve vrijednosti ravnomjerno uvećane. Međutim, ako precjenjivanje nije ujednačeno, tada bi ono moglo negativno utjecati na kvalitetu rezultirajuće politike, posebice u okolinama s većim brojem akcija. Navedeni problem predstavljen je u članku [34] uz potencijalno rješenje u vidu algoritma dvostrukog dubokog Q učenja.

$$\max_a Q(s_{t+1}, a; \theta^-) = Q(s_{t+1}, \underset{a}{\text{argmax}}\, Q(s_{t+1}, a, \theta^-); \theta^-) \tag{2.13}$$

Ideja dvostrukog dubokog Q učenja jest smanjiti precjenjivanje vrijednosti funkcije akcije na način da postupke odabira akcije provodi neuronska mreža koja slijedi politiku (s parametrima θ), a izračun funkcije akcije provodi ciljana neuronska mreža (s parametrima θ^-). Modificirana Bellmanova jednadžba prilagođena za provođenje algoritma dvostrukog dubokog Q učenja i gubitak prikazani su jednadžbama 2.14 i 2.15 [5].

$$Q(s_t, a_t; \theta) = Q(s_t, a_t; \theta) + \alpha \, \mathbb{E}\left[r_t + \gamma Q(s_{t+1}, \underset{a}{\operatorname{argmax}} \, Q(s_{t+1}, a, \theta); \theta^-) - Q(s_t, a_t; \theta) \right] \tag{2.14}$$

$$\mathcal{L}(\theta) = \underset{(s,a,r,s') \sim U(D)}{\mathbb{E}} \left[\left(r_t + \gamma Q(s_{t+1}, \underset{a}{\operatorname{argmax}} \, Q(s_{t+1}, a, \theta); \theta^-) - Q(s_t, a_t; \theta) \right)^2 \right] \tag{2.15}$$

Cilj opisanog algoritma dvostrukog dubokog Q učenja bio je smanjiti efekt precjenjivanja vrijednosti funkcije akcije pritom ne mijenjajući osnovni kostur algoritma dubokog Q učenja kako bi mogli na jednostavan način usporediti algoritma bez pretjeranog utroška resursa u računanju. Usporedba je pokazala da u okolinama s većim brojem akcija, algoritam dvostrukog dubokog Q učenja postiže bolje rezultate [34]. Algoritam 2.7 prikazuje pseudokod dvostrukog dubokog Q učenja s metodom ponavljanja iskustva, dvije neuronske mreže za aproksimaciju Q funkcija, zamjenu vrijednosti težina parametara, metodu ponavljanja iskustva, izračun funkcije gubitka.

Algorithm 1 : Double Q-learning (Hasselt et al., 2015)

Initialize primary network Q_θ, target network $Q_{\theta'}$, replay buffer $\mathcal{D}$, $\tau << 1$
for each iteration **do**
  for each environment step **do**
    Observe state s_t and select $a_t \sim \pi(a_t, s_t)$
    Execute a_t and observe next state s_{t+1} and reward $r_t = R(s_t, a_t)$
    Store (s_t, a_t, r_t, s_{t+1}) in replay buffer $\mathcal{D}$
  for each update step **do**
    sample $e_t = (s_t, a_t, r_t, s_{t+1}) \sim \mathcal{D}$
    Compute target Q value:
      $Q^*(s_t, a_t) \approx r_t + \gamma \; Q_\theta(s_{t+1}, argmax_{a'} Q_{\theta'}(s_{t+1}, a'))$
    Perform gradient descent step on $(Q^*(s_t, a_t) - Q_\theta(s_t, a_t))^2$
    Update target network parameters:
      $\theta' \leftarrow \tau * \theta + (1 - \tau) * \theta'$

Slika 2.7: Algoritam dvostrukog dubokog Q učenja s ponavljanjem iskustva

2.3.4. Gradijent politike

Prethodno opisani algoritmi podržanog učenja temeljili su se na aproksimaciji funkcije vrijednosti (kvalitete (engl. *quality*) pojedine akcije i/ili stanja). Veća vrijednost funkcije vrijednosti označavala je bolju kvalitetu promatrane akcije u određenom stanju. Optimalnu politika pritom implicitno dobijemo izvođenjem niza akcija za koje je model procijenio da su najkvalitetnije. S druge strane, postoje i algoritmi koji se oslanjaju na direktnu aproksimaciju funkcije politike (izraz 2.6). Funkcija politike vraća vjerojatnosnu distribuciju akcija koje agent može poduzeti ovisno o stanju u kojem se nalazi [1]. Označena je parametrima θ i predstavljamo ju izrazom 2.16.

$$\pi(s, a) \approx \pi_\theta(s, a) = P(s, a; \theta) \tag{2.16}$$

Metode zasnovane na politici (engl. *policy-based*) imaju brojne prednosti: podržavaju stohastičke i deterministčke politike, vrlo su učinkoviti u visokodimenzionalnim i kontinuiranim prostorima akcija, te ih karakteriziraju dobra svojstva i izrazito brza konvergencija pod uvjetom korištenja stohastičkog gradijentnog uspona (engl. *Stochastic Gradient Ascent*) [22]. U suštini, metode podržanog učenja zasnovane na politici predstavljaju optimizacijski problem. Stohastički gradijentni uspon koristi se zbog ideje maksimiziranja ciljne funkcije politike (engl. *policy objective function*) $J(\theta)$ definirane izrazom 2.17 koja predstavlja vrijednost očekivanog povrata (kojeg smo izrazom 2.5 definirali kao sumu diskontiranih nagrada) [37]. Vrijednost parametara θ ažurira se u smjeru pozitivnog gradijenta ciljne funkcije politike $J(\theta)$ s korakom učenja α 2.18. Ažuriranje parametara vrši se na kraju epizode, za razliku od Q učenja gdje su se težine ažurirale na svakom koraku.

$$J(\theta) = \mathop{\mathbb{E}}_{\tau \sim \pi_\theta} [R(\tau)] \tag{2.17}$$

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta) \tag{2.18}$$

Korištenje neuronskih mreža i derivabilnih aktivacijskih funkcija omogućava pretpostavku da je i politika π_θ koju nastojimo aproksimirati, derivabilna. Na slici 2.8 nalazi se sažeti prikaz algoritama koji se baziraju na gradijentu politike (engl. *Policy Gradient*) i njihovih izraza za gradijent ciljne funkcije politike.

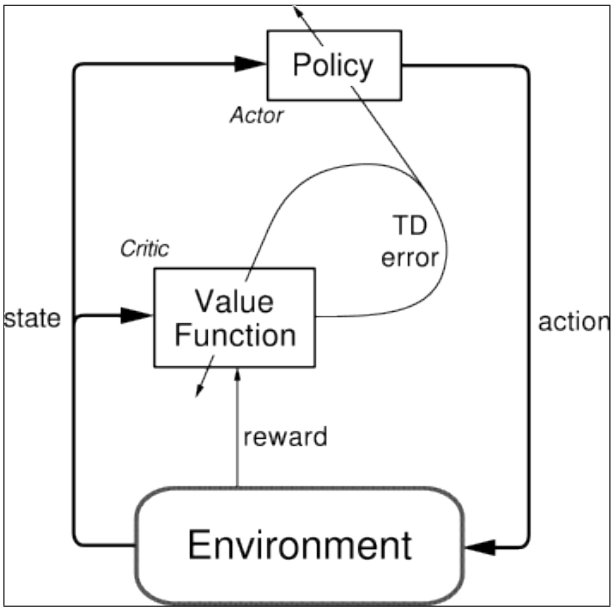
$$
\begin{aligned}
\nabla_\theta J(\theta) &= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a)\, G_t\right] && \text{REINFORCE} \\
&= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a)\; Q^w(s,a)\right] && \text{Q Actor-Critic} \\
&= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a)\; A^w(s,a)\right] && \text{Advantage Actor-Critic} \\
&= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a)\; \delta\right] && \text{TD Actor-Critic}
\end{aligned}
$$

Slika 2.8: Algoritmi gradijenta politike [13]

2.3.5. Akter-kritičar

Algoritmi akter-kritičar (engl. *Actor-Critic*) kombiniraju oba pristupa - aproksimiraju funkciju vrijednosti i funkciju politike. Osnovna ideja jest podijeliti model na dva dijela – akter (engl. *Actor*), koji uči parametriziranu politiku i kritičar (engl. *Critic*), koji procjenjuje funkciju vrijednosti. Modeli s vremenom (učenjem) u svojoj ulozi postaju sve bolji - akter uči odabirati bolje akcije (počinje učiti politiku), a kritičar postaje bolji u procjeni tih akcija i evaluaciji aktera [20].

Akter-kritičar algoritmi pripadaju skupini algoritama koji uče s uključenom politikom i nužno je da se do kraja epizode cijelim putem prati odabrana politika. Slika 2.9 prikazuje strukturu akter-kritičar algoritama sa specificiranom kritikom u obliku signala greške vremenske razlike (*temporal difference error*) koji je specifičan za svaki pojedini akter-kritičar algoritam. Uzevši u obzir navedeni signal, riječ je o *TD Actor-Critic* algoritmu čija je formula prikazana na slici 2.8.



Slika 2.9: Struktura akter-kritičar algoritma [12]

Metoda aktor-kritičar ima dobru i intuitivnu analogiju sa svakodnevnim životom i primjerom odrastanja u kojem dijete (koje ima ulogu aktera) istražuje okolinu oko sebe i stalno isprobava nove stvari, dok ga istovremeno njegovi roditelji (koji imaju ulogu kritičara) promatraju, čuvaju i pokušavaju mu dati na znanje koji su njegovi postupci opravdani i dobri, a koji nisu. Dijete zaključuje na temelju povratnih informacija svojih roditelja i prilagođava svoje ponašanje. Kako dijete raste, ono uči koji su postupci dobri i loši, te u suštini uči igrati igru zvanu život.

2.3.6. Prednosni akter-kritičar

Algoritam prednosni akter-kritičar (engl. *Advantage Actor Critic*) (kraće *A2C*) nasljeđuje sve prednosti osnovnih implementacija akter-kritičar metode. Svrha kritičara je redukcija visoke varijance gradijenta koja je prisutna u običnim metodama zasnovanim na gradijentu politike (npr. u algoritmu gradijenta politike *REINFORCE* prikazanom na slici 2.8 koji koristi kumulativni povrat 2.5).

U algoritmu *A2C* kritičar ocjenjuje trenutnu akciju korištenjem funkcije prednosti (engl. *Advantage function*). Ona ukazuje na prednost poduzimanja određene akcije u odnosu na onu koja prati politiku. Izražavamo ju kao razliku vrijednosti akcije (Q vrijednosti) i vrijednosti stanja 2.19. Dakle, zadatak kritičara je procijeniti funkciju prednosti i pritom agentu pružiti informaciju o benefitu poduzimanja te akcije u odnosu na akciju koja prati politiku.

$$A(s_t, a_t) = Q(s_t, a_t) - V(s_t) \tag{2.19}$$

Vrijednost akcije aproksimiramo greškom vremenske razlike 2.20. Ako poopćimo trenutnu situaciju da za svaku poduzetu akciju trebamo izračunati vrijednost funkcije prednosti, i uzmemo u obzir da se evaluacija i ažuriranje parametara radi tek na kraju epizode, shvatimo da funkcija $Q(s_t, a_t)$ predstavlja diskontni kumulativni povrat od stanja t pa sve do kraja epizode. Navedena tvrdnja prikazana je izrazom 2.21.

$$A(s_t, a_t) = r_t + \gamma V(s_{t+1}) - V(s_t) \tag{2.20}$$

$$A(s_t, a_t) = R_t - V(s_t) \tag{2.21}$$

Akter i kritičar su predstavljeni dubokim modelima. Kritičar ažurira parametre w korištenjem funkcije greške vremenske razlike, dok akter ažurira parametre θ korištenjem gradijenta politike. Budući da koristimo i aktera i kritičara, ukupna funkcija

gubitka izražava se kao zbroj pojedinačnih gubitaka svakog sudionika 2.22. Definirana i opisana funkcija prednosti zajedno s akterovim aproksimacijama vjerojatnosti akcija sudjeluje u maksimiziranju ciljne funkcije politike (odnosno minimiziranju gubitka) i ažuriranju parametara dubokog modela kojim je akter predstavljen 2.23 [1]. Kritičaru želimo da mu je aproksimacija funkcije stanja što bliža stvarnom povratu i zato njegov gubitak predstavljamo izrazom 2.24 gdje L_δ predstavlja huberov gubitak (engl. *Hubert loss*) [1].

$$L = L_{actor} + L_{critic} \tag{2.22}$$

$$L_{actor} = -\sum_{t=1}^{T} \log \pi_\theta(a_t \mid \ s_t) \left[R_t - V_w^\pi(s_t)\right] \tag{2.23}$$

$$L_{critic} = L_\delta(R_t, V_w^\pi) \tag{2.24}$$

Pseudokod *A2C* algoritma prikazan je slici 2.10. Dobivena nagrada, vjerojatnost akcija i vrijednost stanja pamte se sve do kraja epizode ili trenutka definiranog parametrom t_{max} kada se izračunavaju funkcije gubitka i ažuriraju parametri dubokih modela [28].

Algorithm 1 Advantage actor-critic - pseudocode

// Assume parameter vectors θ and θ_v
Initialize step counter $t \leftarrow 1$
Initialize episode counter $E \leftarrow 1$
repeat
  Reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$.
  $t_{start} = t$
  Get state s_t
  repeat
    Perform a_t according to policy $\pi(a_t|s_t;\theta)$
    Receive reward r_t and new state s_{t+1}
    $t \leftarrow t+1$
  until terminal s_t **or** $t - t_{start} == t_{max}$
  $R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t, \theta_v) & \text{for non-terminal } s_t \text{ //Bootstrap from last state} \end{cases}$
  for $i \in \{t-1, \ldots, t_{start}\}$ **do**
    $R \leftarrow r_i + \gamma R$
    Accumulate gradients wrt θ: $d\theta \leftarrow d\theta + \nabla_\theta \log \pi(a_i|s_i;\theta)(R - V(s_i;\theta_v)) + \beta_e \partial H(\pi(a_i|s_i;\theta))/\partial\theta$
    Accumulate gradients wrt θ_v: $d\theta_v \leftarrow d\theta_v + \beta_v(R - V(s_i;\theta_v))(\partial V(s_i;\theta_v)/\partial\theta_v)$
  end for
  Perform update of θ using $d\theta$ and of θ_v using $d\theta_v$.
  $E \leftarrow E+1$
until $E > E_{max}$

Slika 2.10: Algoritam prednosni akter-kritičar (*A2C*) [28]

3. OpenAI Gym

OpenAI Gym je *Python* biblioteka (engl. *library*) otvorenog koda (engl. *open source*) koja služi za razvijanje i usporedbu agenata u odabranim okolinama. Iznimno je popularna u sferi simpatizera i programera koji se bave razvijanjem modela podržanog učenja zbog jednostavnosti korištenja, velikog broja dostupnih okolina i jednostavnog stvaranja novih okolina, te jednostavne interakcije agenta i okoline. *OpenAI Gym* biblioteka se redovito održava i trenutno je u verziji `0.24.1`.

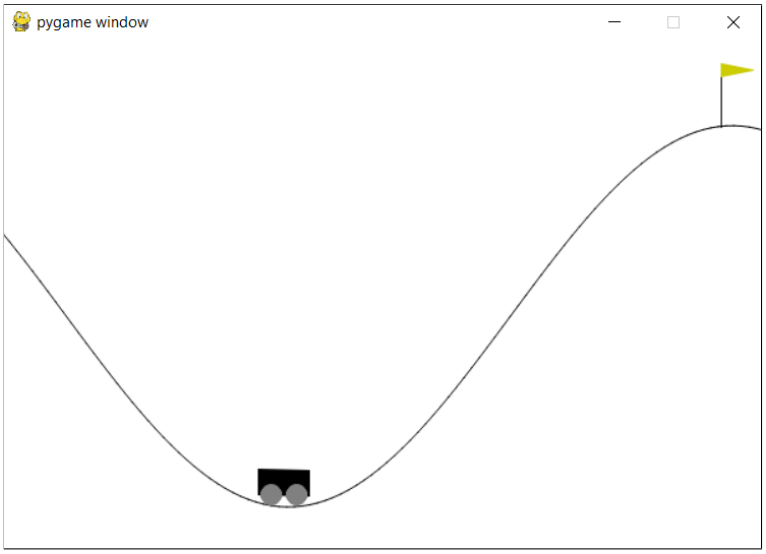
3.1. Struktura

Interakcija agenta i okoline podijeljena je na epizode. Na početku svake epizode, početno stanje se nasumično uzorkuje iz distribucije, i interakcija se nastavlja sve dok se okolina nađe u terminalnom stanju [17].

```
import gym

env = gym.make("MountainCar-v0")

observation = env.reset()

done = False
while not done:
    action = env.action_space.sample()
    observation, reward, done, info = env.step(action)

    env.render()

env.close()
```

Kôd 3: Jednostavan primjer integracije agenta i *Gym* okoline (1 epizoda)

Kod 3 prikazuje potpunu implementaciju jednostavne interakcije agenta i okoline. Agent u ovom jednostavnom slučaju nasumično odabere akciju iz skupa svih dostupnih akcija za tu okolinu (linija 9). Osnovni kostur sastoji se od koraka specifikacije okoline (linija 3), inicijalizacije okoline (linija 5) te interakcija okoline i agenta - agent predaje okolini odabranu akciju, okolina vraća povratnu informaciju (linije 7 - 14).



Slika 3.1: Rezultat pokretanja koda 3

3.1.1. Okolina

Temelj oko kojeg se zasniva *OpenAI Gym* biblioteka jest razred (engl. *class*) `Env` koji u suštini implementira simulator koji pokreće okruženje u kojem naš agent može interaktirati s okolinom. Točnije rečeno, enkapsulira sva potrebna ponašanja i metode koje su potrebne za jednostavnu interakciju. Objekt tipa `Env` stvara se pozivanjem funkcije `gym.make()` (kod 3 linija 3) kojoj se predaje identifikator okoline (`id`) zajedno s opcionalnim argumentima (metapodatcima).

3.1.2. Interakcija s okolinom

Kao što je vidljivo iz koda 3 osnovne metode koje se pozivaju nad instancom razreda `Env` su `reset` i `step`. Funkcija `reset` postavlja okruženje u početno stanje i vraća njegovu vrijednost (kod 3. linija 5). S druge strane, funkciji `step` (kod 3. linija 10) predaje se jedna od ispravnih akcija koja inicira prijelaz okoline iz jednog stanja u drugo. Funkcija vraća 4 vrijednosti: vrijednost prostora stanja (engl. *observation*), iznos nagrade kao rezultat poduzimanja određene akcije (engl. *reward*), zastavicu koja

signalizira jesmo li došli u završno stanje okoline (engl. *done*), te neke dodatne informacije.

Još jedna vrlo često korištena funkcija jest `render` (kod 3. linija 12) koja služi kako bi se u određenom formatu prikazala okolina. Dostupni formati su: `human` (otvara se skočni prozor sa slikom stanja okoline - slika 3.1), `rgb_array` (`numpy` array RGB vrijednosti) i `ansi` (string reprezentacija okoline).

3.1.3. Prostor akcija i prostor stanja

Osnovna struktura okruženja opisana je atributima `observation_space` i `action_space` koji su dio razreda `Env` i čija se vrijednost može razlikovati zavisno o okolini. Atribut `action_space` opisuje numeričku strukturu svih legitimnih akcija koje se mogu izvesti nad određenom okolinom. S druge strane, atribut `observation_space` definira strukturu objekta koje predstavlja opis stanja u kojem se okolina nalazi.

Format validnih akcija i stanja okoline, odnosno struktura tih podataka, definirana je razredima `Box`, `Discrete`, `MultiBinary` i `MultiDiscrete`. Svi navedeni razredi nasljeđuju i implementiraju glavne metode nadrazreda `Space`.

Razred `Box` predstavlja strukturu podataka u kontinuiranom n-dimenzionalnom prostoru. Prostor i njegove validne vrijednosti omeđene su gornjim i donjim granicama koje se jednostavno postave pri inicijalizaciji strukture pridruživanjem željenih vrijednosti atributima `high` i `low` [21]. Kod 4 prikazuje inicijalizaciju `Box` strukture podataka koja je sastavljena od 3-dimenzionalnog vektora čije su vrijednosti omeđene odozdo i odozgo vrijednostima -1 i -2. Metoda `sample(self)` nasumično uzorkuje element iz prostora koristeći različite distribucije ovisno o ograničenjima prostora.

```
1 >>> import numpy as np
2 >>> from gym.spaces import Box
3 >>> b = Box(low=-1.0, high=2.0, shape=(3,), dtype=np.float32)
4 >>> b.sample()
5 array([-0.3791686 , -0.35007873,  0.8138365 ], dtype=float32)
```

Kôd 4: Primjer korištenja strukture kontinuiranog prostora `Box`

Razred `Discrete` s druge strane, predstavlja strukturu podataka u diskretnom n-dimenzionalnom prostoru gdje su validne vrijednosti sve cjelobrojne vrijednosti unutar intervala $[0, n-1]$ (početna vrijednost se može specificirati). Kod 5 prikazuje inicijalizacije `Discrete` strukture podataka ovisno o specificiranoj početnoj vrijednosti.

```
>>> from gym.spaces import Discrete
>>> d = Discrete(3)              # {0, 1, 2}
>>> d = Discrete(3, start=-1)    # {-1, 0, 1}
>>> d.sample()
1
```

Kôd 5: Primjer korištenja strukture diskretnog prostora `Discrete`

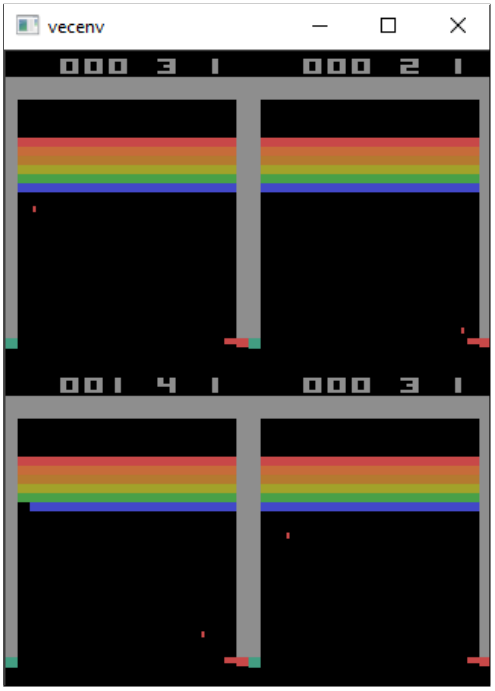
3.1.4. Omotači

Omotači (engl. *Wrappers*) su prikladne strukture koje omogućavaju izmjenu elemenata postojećeg okruženja bez potrebe za mijenjanjem originalnog koda. Omotači omogućavaju modularnost, mogu se implementirati prema vlastitim potrebama i ulančavati [21]. Ova funkcionalnost vrlo se često koristi u situacijama kada pri treniranju modela želimo normalizirati ulaze (skalirati vrijednosti slikovnih jedinica), provesti regularizaciju (podrezivanje vrijednosti nagrade), transformirati ulaze u *PyTorch* dimenzije, implementirati metodu preskakanja slikovnih okvira... Navedene funkcionalnosti moguće je postići tako da definiramo vlastiti omotač koji će nasljeđivati ili obični `Wrapper` nadrazred ili specifičnije razrede poput `ObservationWrapper`, `RewardWrapper`, `ActionWrapper`...

3.1.5. Vektorizirana okruženja

Koristeći standardne metode stvaranja i interakcije s Gym okruženjem, pokrećemo samo jednu instancu okruženja i na taj način ne iskorištavamo računalnu snagu koja nam je dostupna u potpunosti. Vektorizirana okruženja (engl. *Vectorized environments*) su okruženja koja paralelno pokreću više kopija istog okruženja u svrhu poboljšanja učinkovitosti i ubrzanja procesa učenja agenta i njegove interakcije s kopijama okolina.

Python biblioteka *Stable Baselines3* [3] pruža gotove omotače koji omogućavaju pokretanje n paralelnih neovisnih instanci okolina. Paralelna obrada interno je implementirana korištenjem *multiprocessing* paketa. Zbog paralelne interakcije agenta s više okolina potrebno je prilagoditi strukturu objekata. Agent predaje omotanoj okolini niz od n akcija i dobiva listu povratnih informacija (stanje pojedinačne okoline, nagrada i zastavica terminalnog stanja) u formi liste n vrijednosti.



Slika 3.2: Vektorizirano okruženje sa 4 paralelne asinkrone instance

3.2. Determinizam atari okruženja

Upravljanje agentima i njihovo izvođenje akcija u određenoj atari okolini interno je implementirano koristeći objektno orijentiranu razvojnu cjelinu (engl. *framework*) *The Arcade Learning Environment* (skraćeno *ALE*). Na taj način razdvajaju se slojevi emulacije okruženja (koristeći Atari 2600 emulator *Stella*), sloj koji kontrolira samog agenta *(ALE)* i sloj koji pruža jednostavnu programsku implementaciju i interakciju *(OpenAI Gym)* [16].

Originalna Atari 2600 konzola nije imala izvor entropije za generiranje pseudoslučajnih brojeva i iz tog razloga okruženje je bilo u potpunosti deterministično - svaka igra počinje u istom stanju, a ishodi su u potpunosti određeni stanjem i akcijom [25]. Iz tog razloga vrlo je jednostavno postići visoku uspješnost agenta u okolini pamćenjem dobrog niza akcija. Nama to ne odgovara jer želimo postići da agent nauči donositi dobre odluke. Pristupi kojima se nastoji uvesti određen stupanj stohastike su *sticky actions*, *frame skipping*, *initial no-ops*, te *random action noise*.

Frame skipping pristup pri svakom koraku okoline uzima zadnju akciju koju je agent odabrao i ponavlja ju n broj puta (kroz n slikovnih okvira). Ovisno o verzijama okoline, broj ponavljanja može se specificirati tipom `int` - cijelobrojnom fiksnom vrijednošću (determinizam), ili tipom `tuple(2)` - bira se slučajno odabrana vrijednost

unutar specificiranog intervala (stohastika). Osim potencijalnog uvođenja stohastike, ovaj pristup pojednostavljuje problem podržanog učenja i ubrzava izvođenje [25]. Tehnika *initial no-ops* označava da na početku epizode okolina ignorira akcije agenta slučajan broj puta iz intervala $[0, n]$ - izvršava akciju `NOOP`. Osim što na početku agent ne zna kada će okolina početi koristiti njegove akcije, ostatak interakcije s okolinom je i dalje deterministički [25]. Sličan pristup bio bi i da potičemo raznolikost početnih stanja objekta s kojim upravljamo. To bi postigli nasumičnim poduzimanjem niza akcija kojima pomičemo objekt prije samog aktiviranja posebne akcije `FIRE` [26].

Tehnika *random action noise* zamjenjuje odabranu akciju agenta nasumičnom akcijom samo ako je vjerojatnost zamjene manja od specificirane vrijednosti. Ovo uspješno uvođenje stohastike dolazi s negativnim posljedicama poduzimanja slučajne akcije koja može značajno poremetiti politiku agenta i smanjiti učinak [25]. Odabir slučajne akcije bolje je zamijeniti pristupom ponavljanja prethodno odabrane akcije uzimajući u obzir parametar vjerojatnosti zamjene akcije - *stickiness*. Riječ je o *sticky actions* pristupu koji uvodi stohastiku, te ne ometa odabir akcije agenta koji može biti siguran da njegove akcije neće biti pogubne za ostatak iterakcije s okolinom.

Pristupi *sticky actions* i *frame skipping* dio su *OpenAI Gym* biblioteke i mogu se specificirati prilikom instanciranja okoline, dok je *random action noise* i *initial no-ops* tehnike potrebno manualno implementirati ili se osloniti na izvedbu u poznatim *Python* bibliotekama poput *Stable Baselines3*.

3.3. Okruženja

U OpenAI Gym ekosustavu dostupno je puno okruženja koja omogućuju interakciju s agentom. Neki od njih su: *Atari* - skup od Atari 2600 okolina, *MuJoCo* (punim nazivom *Multi-Joint dynamics with Contact*) - skup okolina za provođenje istraživanja i razvoja u robotici, biomehanici, grafici i drugim područjima gdje je potrebna brzina i točna simulacija, *Classic Control* - skup okolina koje opisuju poznate fizikalne eksperimente. Opisat ćemo neke od okolina koje se koriste u daljnjoj implementaciji.

3.3.1. Okruženje CartPole

Ovo okruženje modelira fizikalni problem održavanja ravnoteže. Inačica je sličnog fizikalnog problema pod nazivom *obrnuto njihalo* (engl. *inverted pendulum*). Za pomična kolica zakvačen je stupić. Njegovo težište nalazi se iznad središta mase i na taj način osigurava da sustav nije stabilan. Zglob, odnosno dodirna točka između stupića

i kolica nema trenja niti drugih gubitaka. Također, kolica koja se kreću vodoravno po putanji u 2 smjera nemaju trenja niti drugih gubitaka. Cilj ovog fizikalnog problema jest uravnotežiti stup primjenom sila i pomicanjem kolica u lijevom ili desnom smjeru.

Za svaki poduzeti korak okolina dodjeljuje nagradu u vrijednosti $+1$. Struktura valjanih akcija koje agent može poduzeti (`action_space`) instanca je razreda `Discrete(2)` - skup akcija je diskretan i u svakom koraku je moguće odabrati 1 od maksimalno 2 dostupne akcije. Opis značenja svake akcije prikazan je u tablici 3.1. S druge strane, objekt koji predstavlja strukturu stanja okoline u određenom vremenskom trenutku (`observation_space`) instanca je razreda `Box(4)` – stanje se sastoji od 4 kontinuirane vrijednosti od kojih su neke ograničene i odozdo i odozgo. Točan opis i granice vrijednosti predočeni su u tablici 3.2.

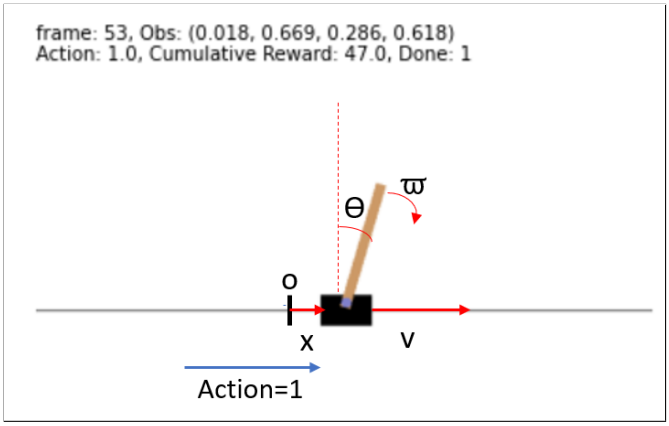
Tablica 3.1: Opis valjanih akcija okoline CartPole - atribut `action_space`

Akcija	Opis akcije
0	Pomak kolica ulijevo
1	Pomak kolica udesno

Tablica 3.2: Opis strukture okoline okoline CartPole - atribut `observation_space`

Indeks	Opis	Donja granica	Gornja granica
0	Pozicija kolica	-4.8	4.8
1	Brzina kolica	$-\infty$	∞
2	Nagib štapića i kolica	-0.418rad	0.418rad
3	Brzina štapića na vrhu	$-\infty$	∞

Početno stanje okoline inicijalizira se pozivom metode `reset()` slučajnim vrijednostima iz uniformne razdiobe na intervalu $[-0.05, 0.05]$. Okolina podržava 3 uvjeta zaustavljanja (uvjeti koji označuju da je riječ o terminalnom stanju): nagib štapića i kolica je izvan intervala $[-0.2095, 0.2095]$rad, pozicija sredine kolica je izvan intervala $[-2.4, 2.4]$ (sredina kolica dotiče rub vidljivog prostora) i duljina epizode veća je od 500 koraka. Na slici 3.3 prikazana je okolina CartPole zajedno s vrijednostima okoline.



Slika 3.3: Izgled i vrijednosti CartPole okoline [36]

3.3.2. Okruženje Breakout

Ova okolina simulira poznatu Atari 2600 igru u kojoj je cilj sakupiti što više bodova pomičući platformu i održavajući lopticu na ekranu. Platforma je postavljena na dnu ekrana, na fiksnoj visini i moguće ju je pomicati u dva smjera. Loptica se odbija između zidova, platforme i 6 razina *ciglenih* blokova čijim razbijanjem se sakupljaju bodovi. Ako loptica padne ispod platforme koju igrač kontrolira, gubi se život. Igra završava kada igrač potroši 5 života, odnosno kada 5 puta loptica padne ispod platforme [2].

Skup validnih akcija je instanca razreda `Discrete(4)` - skup akcija je diskretan i u svakom koraku je moguće odabrati 1 od maksimalno 4 dostupne akcije. Opis značenja svake akcije prikazan je u tablici 3.3. Kao opis trenutnog stanja okoline moguće je dobiti RGB vrijednosti svakog piksela slike (slike dimenzije 210×160) ili vrijednosti radne memorije *ALE* okoline (128 bajta) - što je korisno jer možemo preskočiti korak učenja reprezentacije okoline (preskačemo dio gdje algoritmi učenja moraju iz piksela slike naučiti reprezentaciju). Razlike u strukturi objekta okoline prikazane su u tablici 3.4.

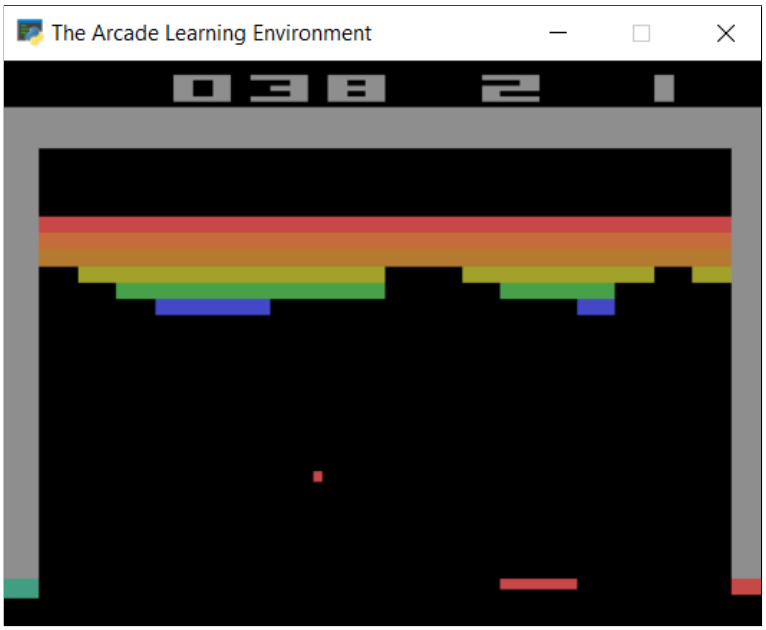
Tablica 3.3: Opis valjanih akcija okoline Breakout - atribut `action_space`

Akcija	Opis akcije	Detaljniji opis akcije
0	NOOP	Ne poduzima se nikakva akcija
1	FIRE	Akcija koja pokreće igru
2	RIGHT	Platforma se pomiče udesno
3	LEFT	Platforma se pomiče ulijevo

Tablica 3.4: Opis strukture objekta okoline Breakout - atribut `observation_space`

Indeks	Struktura
RAM vrijednosti	`Box(0, 255, (128,), uint8)`
Vrijednosti RGB slike	`Box(0, 255, (210, 160, 3), uint8)`

Nagrada dolazi u obliku bodova koji se dobivaju uništavajući *ciglene* blokove. Vrijednost nagrade ovisi o boji cigle. Izgled same okoline prikazan je na slici 3.4.



Slika 3.4: Izgled Breakout okoline

4. Stable Baselines3

Poznatije biblioteke otvorenog koda za programski jezik *Python* koje prvenstveno pružaju implementacije algoritama podržanog učenja su *Stable Baselines3*, *RLlib*, *Tensorforce*... Njihove implementacije algoritama i naučene modele korisno je koristiti pri usporedbi performansi vlastitih implementacija. Navedene biblioteke koriste *OpenAI Gym* okruženja za razvijanje, testiranje i usporedbu agenata.

Stable Baselines3 (SB3) biblioteka sadrži skup implementacija algoritama podržanog učenja. Za oblikovanje dubokih modela korištena je *PyTorch* biblioteka. Algoritmi su neovisni o modelu (engl. *model-free*) i ograničeni na interakciju jednog agenta s okolinom (engl. *single-agent*). SB3 predstavlja nadogradnju *Stable Baselines* i *OpenAI Baselines* biblioteka koje za oblikovanje dubokih modela koriste *TensorFlow* biblioteku. U odnosu na njih, SB3 je bolje dokumentiran, 95% koda je pokriveno testovima (engl. *test coverage*), te sadrži više implementiranih algoritama [30].

Odsječak koda 6 prikazuje jednostavnost korištenja SB3 biblioteke. U dvije linije koda postavili smo tip dubokog modela, identifikator okoline, specificirali algoritam učenja, te pokrenuli učenje modela sa specificiranim ukupan broj koraka.

```
1 from stable_baselines3 import A2C
2
3 env = A2C('MlpPolicy', 'CartPole-v1', verbose=1)
4 model = env.learn(total_timesteps=100_000)
```

Kôd 6: Jednostavan primjer korištenja *Stable Baselines3* biblioteke

4.1. Korištenje predtreniranih modela

Predtrenirane modele koji su objavljeni u javnom *GitHub* repozitoriju [29] poželjno je koristiti pri usporedbi ponašanja naših manualno implementiranih algoritama i pripadajućih treniranih agenata.

Za uspješno pokretanje agenata potrebno je proučiti spomenuti *GitHub* repozitorij radnog okvira *RL Baselines3 Zoo*, klonirati projekt i pokrenuti naredbu 7 koja pokreće *Python* skriptu s dodatnim argumentima u kojima je specificiran algoritam, identifikator okoline i direktorij u kojem se nalazi prednaučen agent. *RL Baselines3 Zoo* jest radni okvir koji koristi *Stable Baselines3* biblioteku i pruža niz korisnih skripti za treniranje i evaluiranje agenata, podešavanje hiperparametara, iscrtavanje rezultata i snimanje videa.

```
1 python enjoy.py --algo a2c --env CartPole-v1 --folder agents/
```

Kôd 7: Naredba za izvođenje *Python* skripte i pokretanje predtreniranog modela

5. Implementacija

Kompletna implementacija napisana je u programskom jeziku *Python* koji je odabran zbog svoje jednostavnosti, sažetosti, fleksibilnosti, neovisnosti o operacijskom sustavu, te velikoj popularnosti. Postoji veliki broj dobro dokumentiranih i iznimno korisnih biblioteka od kojih se za problematiku ovog rada izdvajaju biblioteke za oblikovanje i učenje dubokih modela (*PyTorch*), biblioteka za iznimno brzo i učinkovito provođenje matematičkih operacija (*Numpy*), te biblioteka za jednostavnu interakciju agenata i implementiranih okolina s kojima agent interaktira i dobiva povratnu informaciju (*OpenAI Gym*).

Implementirani su duboki modeli prethodno predstavljenih algoritma dubokog Q učenja, dvostrukog dubokog Q učenja, te prednosnog akter-kritičara. Agenti u svojoj implementaciji interaktiraju s okolinama CartPole i Breakout.

5.1. Moduli

Implementacija je podijeljena u nekoliko modula: modul za generičko instanciranje dubokih modela, modul za generičku serijalizaciju i deserijalizaciju objekata (spremanje i učitavanje agenata), modul za *OpenAI Gym* omotače i modul u kojem se nalazi implementacija agenata i algoritama.

5.1.1. Instanciranje dubokih modela

Modul `networks` sastoji se od metoda i struktura koje omogućuju generičko instanciranje dubokih modela, točnije unaprijednih potpuno povezanih modela i konvolucijskih modela opisanih u poglavlju Duboki modeli. Prilikom poziva metode za instanciranje unaprijedne potpuno povezane mreže kao argument predaje se lista cijelih brojeva koje predstavljaju dimenzije pojedinih slojeva odvojenih nelinearnom aktivacijskom funkcijom ReLU (kao što je prikazano odsječkom 8).

```
from networks import fc

net = fc([4, 128, 64, 2])
print(net)

# Sequential(
#   (0): Linear(in_features=4, out_features=128, bias=True)
#   (1): ReLU()
#   (2): Linear(in_features=128, out_features=64, bias=True)
#   (3): ReLU()
#   (4): Linear(in_features=64, out_features=2, bias=True)
# )
```

Kôd 8: Generičko instanciranje unaprijedne potpuno povezane mreže

S druge strane, pri instanciranju konvolucijske neuronske mreže, za definiranje atributa konvolucijskog sloja koristi se posebna struktura `CnnStructure` 9 koja definira broj ulaznih i izlaznih kanala, dimenziju jezgre (slobodnih parametara koje učimo), veličinu koraka i nadopunu. Nakon konvolucijskog sloja provodi se nelinearnost aktivacijskom funkcijom ReLU. Na posljetku, značajke se transformiraju u vektor i prosljeđuju potpuno povezanom sloju. Pri prijelazu iz konvolucijskog sloja u potpuno povezani sloj, potrebno je točno izračunati broj parametara koji se prenose iz jednog sloja u drugi. Umjesto iscrpnog računanja, broj parametara je određen tako da se napravio jedan unaprijedni prolaz kroz definirane konvolucijske slojeve. Generični poziv funkcije za instanciranje konvolucijske neuronske mreže prikazan je odsječkom 10.

```
class CnnStructure:
    def __init__(self, in_channels: int, out_channels: int,
    ↪  kernel_size: int, stride: int, padding: int = 0):
        ...
```

Kôd 9: Struktura za definiranje atributa konvolucijskog sloja

Svi parametri konvolucijskog i potpuno povezanog sloja inicijalizirani su koristeći Kaiming uniformnu distribuciju koja u obzir uzima funkciju nelinearnosti koju koristimo.

```
from networks import cnn, CnnStructure

cnn_layers = [
    CnnStructure(in_channels=4, out_channels=32, kernel_size=8,
    ↪  stride=4),
]
net = cnn(observation_space, cnn_layers, [512], 4)
print(net)

# Sequential(
#   (0): Conv2d(4, 32, kernel_size=(8, 8), stride=(4, 4))
#   (1): Flatten(start_dim=1, end_dim=-1)
#   (2): Linear(in_features=12800, out_features=512, bias=True)
#   (3): ReLU()
#   (4): Linear(in_features=512, out_features=4, bias=True)
# )
```

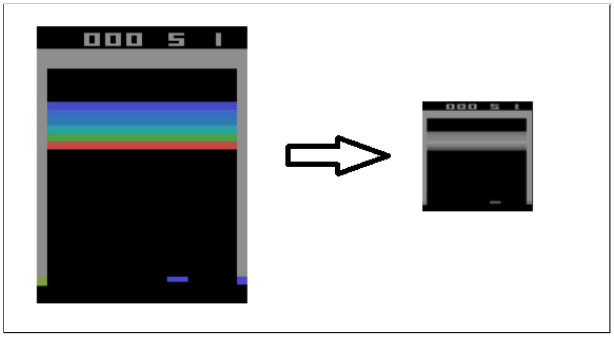
Kôd 10: Generičko instanciranje konvolucijske neuronske mreže

5.1.2. Serijalizacija i deserijalizacija agenata

Modul `serialization` sadrži metode za učitavanje i spremanje *Python* objekata, točnije serijalizaciju objekata u tok okteta podataka (engl. *byte stream*) i njegovu deserijalizaciju. Za samo spremanje koriste se pomoćne funkcije biblioteke *PyTorch* (`torch.save` i `torch.load`) koje u internoj implementaciji koriste biblioteku za serijalizaciju i deserijalizaciju - *Pickle*. U implementaciji koristimo navedene funkcionalnosti kako bi sačuvali i učitali parametre naučenih dubokih modela.

5.1.3. Omotači

Prema uzoru na biblioteku *Stable Baselines3* i članke koji opisuju dobre i kvalitetne pristupe pri rješavanju problema podržanog učenja [27], modul `wrappers` sadrži zbirku implementiranih *OpenAI Gym* omotača. Svi korišteni omotači prilagođeni su uporabi na *Atari* okolinama. Ideja je da originalnu okolinu omotamo u strukture koje omogućavaju izmjenu elemenata postojećeg okruženja bez potrebe za mijenjanjem originalnog koda. Omotač `WarpFrame` koristeći biblioteku *cv2* (*OpenCV*) skalira sliku okoline na okvir dimenzija 84×84 i pretvara RGB vrijednosti piksela u nijanse sive boje (engl. *grayscale*), kao što je i prikazano na slici 5.1.



Slika 5.1: Breakout okoline prije i nakon korištenja omotača `WarpFrame`

Obratimo pažnju ponovno na prikaz 5.1. Dimenzije (širina - W, visina - H i broj kanala - C) slike lijevo su $160 \times 210 \times 3$ a slike desno $84 \times 84 \times 1$. Iz razloga što okolina vraća slike u dimenziji $H \times W \times C$, a *PyTorch* konvolucijski sloj prima dimenzije oblike $C \times H \times W$, napravljen je posebni omotač `TransposeImageObs` koji uređuje dimenzije.

Omotači mogu poslužiti i za ugrađivanje stohastike u atari okruženjima kao što je i opisano u poglavlju 3.2. Omotač `NoopResetEnv` osigurava da na početku epizode okolina ignorira akcije agenta (tehnika *initial no-ops*). Omotač `MaxAndSkipEnv` provodi tehniku *frame skipping* i n puta vraća istu povratnu informaciju agentu.

Od ostalih korisnih omotača, bitno je spomenuti omotač `ClipRewardEnv` ograničava nagrade okoline na vrijednosti $\{-1, 0, 1\}$, `FireResetEnv` koji na početku epizode izvršava operaciju `FIRE`, `BatchedPytorchFrameStack` koji u kombinaciji s `SubprocVecEnv` osigurava istovremeno izvođenje nekoliko instanci okruženja koristeći biblioteku *multiprocessing* [30].

5.2. Implementacijski detalji

5.2.1. Duboko Q učenje

Na samom početku izvođenja algoritma potrebno je napuniti spremnik za ponavljanje D i u njega pohraniti sve akcije, stanja i nagrade koje je agent poduzeo od početka. Iz tog razloga korisno je napraviti strukturu koja predstavlja uređenu n-torku `Transition` i razred `ReplayMemory` kao što je prikazano kodom 11. Razred interno za pohranu koristi kolekciju `deque` (engl. *Doubly Ended Queue*) koji se više preferira od listi zbog bržih operacija unosa i brisanja s početka i kraja reda. Kapacitet spremnika za ponavljanje određen je parametrom `capacity`. Razred sadrži metode za ubacivanje podataka i nasumično uzorkovanje određenog broja prijelaza.

```python
import random
from collections import deque, namedtuple

Transition = namedtuple('Transition', ('state', 'action', 'done',
↪  'next_state', 'reward'))

class ReplayMemory:
    def __init__(self, capacity):
        self.memory = deque([], maxlen=capacity)

    def push(self, *args):
        self.memory.append(Transition(*args))

    def sample(self, batch_size):
        return random.sample(self.memory, batch_size)

    def __len__(self):
        return len(self.memory)
```

Kôd 11: Instanciranje neuronskih mreža

Slijedi instanciranje dviju neuronskih mreža: mreže koja prati trenutnu politiku i provodi stalno ažuriranje parametara `online_net` (s parametrima θ), i mreže koja prati ciljanu politiku `target_net` (s parametrima θ^{-}). Nakon instanciranja, obje mreže trebaju imati jednaku vrijednost parametara. Navedeno je prikazano odsječkom koda 12.

```python
from playground.agents import DeepQNetworkAgent

online_net = DeepQNetworkAgent(env, net)
target_net = DeepQNetworkAgent(env, net)

target_net.load_state_dict(online_net.state_dict())
```

Kôd 12: Struktura spremnika za ponavljanje

U svakom koraku potrebno je odabrati iduću akciju koju će agent poduzeti. Provodimo tehniku istraživanja (pritom odabiremo nasumičnu akciju) i odabira akcije s

najvećom Q vrijednošću (potencijalno najkorisniju akciju). Opisana strategija *epsilon-greedy exploration* prikazana je kodom 13.

```
q_values = self(observations)       # online_net unaprijedni prolaz
actions = torch.argmax(q_values, dim=1).tolist()

for i in range(len(actions)):
    if random.random() <= epsilon:
        actions[i] = random.randint(0, self.n_actions - 1)
```

Kôd 13: Implementacija *epsilon-greedy exploration* strategije

Gubitak se izračunava postupkom 14 prema formuli 2.12. Prvo se provodi unaprijedni prolaz nad mrežom `target_net` i pritom se dobivaju vrijednosti $Q(s_{t+1}, a; \theta^-)$. Računa se njihova maksimalna vrijednost i skalira hiperparametrom `gamma`. Provodi se unaprijedni prolaz nad mrežom `online_net` (vrijednosti $Q(s_t, a_t; \theta)$) i od prikupljenih vrijednosti računa huberov gubitak za kojeg se pokazalo da je bolji od srednje kvadratne pogreške (engl. *Mean Squared Error*) zbog manje osjetljivosti na odustupanja (engl. *outliers*). Nakon izračuna gubitka, provodi se izračun gradijenta i ažuriranje parametara `online_net` mreže. Nakon što prođe određeni broj koraka, provodi se zamjena vrijednosti parametara, odnosno ažuriranje parametara `target_net` mreže.

```
with torch.no_grad():
    target_q_values = target_net(next_states)
    max_target_q_values = target_q_values.max(dim=1,
    ↪  keepdim=True)[0]

    targets = rewards + gamma * (1 - dones) * max_target_q_values

q_values = self(states)

action_q_values = torch.gather(input=q_values, dim=1, index=actions)
loss = F.huber_loss(action_q_values, targets)

optimizer.zero_grad()
loss.backward()
optimizer.step()
```

Kôd 14: Izračun gubitka dubokog Q učenja

5.2.2. Dvostruko duboko Q učenje

Algoritam dvostrukog dubokog Q učenja rješava problem precjenjivanja vrijednosti funkcije akcije. Razlika između njega i algoritma dubokog Q učenja očituje se u izrazu i računanju funkcije gubitka 2.15. Upravo to je bio i cilj - riješiti problem precjenjivanja pritom ne mijenjajući osnovni kostur algoritma. Prema algoritmu, izračun funkcije gubitka počinje unaprijednim prolazom kroz `online_net` mrežu i dobivanjem vrijednosti $Q(s_{t+1}, a, \theta)$. Nad dobivenom vrijednostima provodimo operaciju argmax - uzimamo najveće Q vrijednosti i zapamtimo akcije kojima su one pridružene. Nad `target_net` mrežom računamo unaprijedni prolaz i Q vrijednosti izračunate s parametrima θ^- pridružujemo ranije zapamćenim akcijama. Daljnje računanje istovjetno je postupcima u dubokom Q učenju - izračun $Q(s_t, a_t; \theta)$, huberovog gubitka, gradijenta i ažuriranje parametara θ `target_net` mreže

```
with torch.no_grad():
    online_q_values = self(next_states)
    max_online_actions = online_q_values.argmax(dim=1, keepdim=True)

    target_q_values = target_net(next_states)
    action_q_values = torch.gather(input=target_q_values, dim=1,
    ↪  index=max_online_actions)

    targets = rewards + gamma * (1 - dones) * action_q_values

q_values = self(states)
action_q_values = torch.gather(input=q_values, dim=1, index=actions)

loss = F.huber_loss(action_q_values, targets)
...
```

Kôd 15: Izračun gubitka dvostrukog dubokog Q učenja

5.2.3. Prednosni akter-kritičar

Implementacija prednosnog akter-kritičar algoritma započinje konstruiranjem dubokog modela s idejom da i akter i kritičar dijele sve slojeve iste neuronske mreže osim posljednjeg sloja. Dimenzija izlaznog sloja aktera odgovara broju akcija koje su akteru na raspolaganju u određenoj okolini, dok istovremeno kritičar ima samo jedan izlaz koji odgovara procjeni funkcije stanja. Odsječak koda 16 prikazuje inicijalizaciju

potpuno povezanog dubokog modela s dva različita izlazna sloja.

```
self.fc1 = nn.Linear(in_features=input_size, out_features=128)
self.actor = nn.Linear(in_features=128, out_features=output_size)
self.critic = nn.Linear(in_features=128, out_features=1)
```

Kôd 16: Inicijalizacija potpuno povezanog dubokog modela algoritma akter-kritičar

Prilikom unaprijednog prolaza, potrebno je prvo izračunati izlaz iz zajedničkog dijela mreže i potom rezultat provući kroz sloj aktera i sloj kritičara. Akter izračunava vjerojatnost poduzimanja akcija, dok kritičar procjenjuje funkciju stanja. Odsječak unaprijednog prolaza prikazan je kodom 17.

```
x = F.relu(self.fc1(x))

action_probability = F.softmax(self.actor(x), dim=-1)
state_value = self.critic(x)
```

Kôd 17: Unaprijedni prolaz algoritma akter-kritičar

Pri svakoj iteraciji potrebno je odrediti akciju koju će izvršiti agent. Taj odabir akcije prikazan je kodom 18. Unaprijednim prolazom mreže dobijemo vjerojatnost poduzimanja svake od mogućih akcija i vrijednost funkcije stanja. Agent će poduzeti akciju koja ima najveću vjerojatnost. Dobivena nagrada, vjerojatnost akcija i vrijednost stanja pohranjuju se sve do kraja epizode kada se provodi evaluacija, izračunava gubitak, provodi izračun gradijenata i ažuriranje parametara.

```
action_probability, state_value = self.forward(state)

m = Categorical(action_probability)
action = m.sample()

self.actions_history.append((m.log_prob(action), state_value))
```

Kôd 18: Odabir akcije i pohrana podataka za evaluaciju algoritma *A2C*

Konačno, na kraju epizode provodi se postupak evaluacije prikupljenog znanja 19. Prvo za svaki korak izračunavamo diskontni povrat od stanja t pa sve do kraja epizode

(vrijednost R_t). Slijedi računanje gubitka. Za svaku iteraciju izračunava se funkcija prednosti 2.21. Funkcija prednosti je razlika izračunatih diskontnih povrata i procjene funkcije stanja koju je kritičar procijenio prilikom poduzimanja svake akcije 18. Nadalje, gubitak aktera i kritičara računa se prema izrazima 2.23 i 2.24. Gubitci se zbroje, provede se izračun gradijenata i ažuriranje parametara dubokog modela.

```
cum_reward = 0
expected_returns = []
for reward in self.rewards[::-1]:
    cum_reward = reward + self.gamma * cum_reward
    expected_returns.insert(0, cum_reward)

...

actor_losses = []
critic_losses = []
for (action_prob, state_value), reward in zip(self.actions_history,
↪  expected_returns):
    advantage = reward - state_value.item()

    actor_losses.append(-action_prob * advantage)
    critic_losses.append(F.huber_loss(state_value,
    ↪  torch.tensor([reward])))

optimizer.zero_grad()
loss = torch.stack(actor_losses).sum() +
↪  torch.stack(critic_losses).sum()

loss.backward()
optimizer.step()
```

Kôd 19: Evaluacija aktera i kritičara u algoritmu *A2C*

5.3. Okolina CartPole

- učenje na grafičkoj xx - prikaz grafa i opis (sve na istom grafu) - zaključak sa slike - korišteni parametri

5.3.1. Implementacija Deep Q Learning algoritma

https://github.com/PauloSankovic/retro-ai/blob/master/models/DeepQNetworkAgent-state_env%3Dcartpole_gamma%3D0.99_lr%3D5e-4_bs%3D32_es%3D1_ee%3D0.02.pickle

5.3.2. Implementacija Double Deep Q Learning algoritma

5.3.3. Implementacija Actor Critic algoritma

https://github.com/PauloSankovic/retro-ai/blob/master/models/ActorCriticAgent_env%3Dcarpole_gamma%3D0.99_lr%3D3e-2.pickle

video igranja: https://raw.githubusercontent.com/PauloSankovic/retro-ai/master/playground/video/cart-pole/openaigym.video.0.9768.video000000.mp4

5.3.4. Usporedba

5.4. Okolina Breakout

5.4.1. Implementacija Double Deep Q Learning algoritma

https://github.com/PauloSankovic/retro-ai/blob/master/models/DoubleDeepQNetworkAgent-state_env%3Dbreakout_step%3D1400000_v%3D2.pickle

video igranja: https://github.com/PauloSankovic/retro-ai/tree/master/playground/video/breakout-ddqn

5.4.2. Implementacija Deep Q Learning algoritma

https://github.com/PauloSankovic/retro-ai/blob/master/models/DeepQNetworkAgent-state_env%3Dbreakout_step%3D1080000.pickle

5.4.3. Implementacija Actor Critic algoritma

Intuitively, expected return simply implies that rewards now are better than rewards later. In a mathematical sense, it is to ensure that the sum of the rewards converges.

To stabilize training, the resulting sequence of returns is also standardized (i.e. to have zero mean and unit standard deviation).

5.4.4. Usporedba

5.5. Usporedba algoritama

6. Daljnji rad

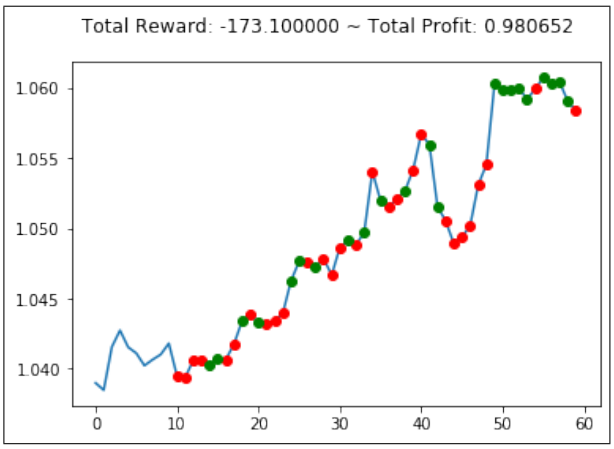
U prethodnim poglavljima predstavljeni su i implementirani pojedini algoritmi dubokog podržanog učenja koristeći programski jezik *Python*, te primarno biblioteku *PyTorch* za oblikovanje i učenje dubokih modela, te okoline biblioteke *OpenAI Gym* u kojima je agent bio smješten i s kojima je interaktirao i dobivao povratne informacije. Osim obrađenih algoritama dubokog Q učenja, dvostrukog dubokog Q učenja, akter-kritičara, te prednosnog akter-kritičara, u daljnjem radu bilo bi korisno predstaviti, implementirati i isprobati algoritme poput suparničkog dvostrukog Q učenja (engl. *Dueling Double Deep Q Network*) (skraćeno *D3QN*), asinkronog prednosnog akter-kritičara (engl. *Asynchronous Advantage Actor-Critic*) (skraćeno *A3C*), mekog akter-kritičara (engl. *Soft Actor-Critic*) (skraćeno *SAC*), Trust Region Policy Optimization (skraćeno *TRPO*), Proximal Policy Optimization (skraćeno *PPO*)... koji u određenim slučajevima daju bolje rezultate od obrađenih algoritama.

Pronalazak optimalnih hiperparametara dubokih modela ovisnima o okolini u kojoj se nalaze je veoma iscrpan i vremenski zahtjevan postupak, no ispravan odabir hiperparametara kao i arhitekture dubokog modela, može uvelike poboljšati performanse agenta. Osim hiperparametara, performanse dubokog modela može poboljšati i korištenje različitih optimizacijskih postupaka. Prilikom implementacije koristio se optimizacijski postupak Adam (engl. *Adaptive Moments*), no moguće je koristiti i obični stohastički gradijentni spust (engl. *Stohastic Gradient Descent*) (skraćeno *SGD*), RMSProp, AdaGrad... i s njima kombinirati upotrebu momenta i restarta kod promjene stope učenja. Također, bilo bi korisno pri prolasku podataka kroz konvolucijske slojeve primijeniti sažimanje maksimumom. Ono u obzir uzima samo značajku najveće vrijednosti te na taj način uklanja šum ulaza i potencijalno bira značajku najveće važnosti. Korisno bi bilo i nakon određenog broja iteracija učenja provesti validacijsku epizodu nad trenutnim stanjem agenta (bez ikakvih omotača) i na taj način procijeniti koliko se ponašanje u međuvremenu poboljšalo ili pogoršalo.

Također, u daljnjem radu korisno bi bilo smjestiti agente te provjeriti kako se ponašaju u novim okolinama i suočavaju s njihovom problematikom. Posebno se zanim-

ljivima čine *OpenAI Gym* okoline *Double Dunk* [7], *Space Invaders* [10], *Car Racing* [4], *Pong* [8], te cijela porodica *MuJoCo* okolina za simulaciju i kontrolu. *OpenAI Gym* ima podršku za jednostavan razvoj i ručnu implementaciju novih igara i njihovih okolina, poput igrica *Flappy Bird* [32] ili *Dino Game* [31].

Podržano učenje ima veliku primjenu i u trgovanju dionicama, valutama (engl. *forex*), kriptovalutama za čije bi se treniranje agenata i interakciju mogle koristiti biblioteke *Gym AnyTrading* (slika 6.1) [15] i *TensorTrade* [33].



Slika 6.1: *Gym AnyTrading* okolina s označenim kupovnim i prodajnim pozicijama

7. Zaključak

LITERATURA

[1] Playing cartpole with the actor-critic method. URL https://www.tensorflow.org/tutorials/reinforcement_learning/actor_critic.

[2] Breakout - atari - atari 2600. URL https://atariage.com/manual_html_page.php?SoftwareID=889.

[3] Vectorized environments. URL https://stable-baselines3.readthedocs.io/en/master/guide/vec_envs.html.

[4] URL https://www.gymlibrary.ml/environments/box2d/car_racing/.

[5] . URL https://nn.labml.ai/rl/dqn/index.html.

[6] Deep learning paper implementations, . URL https://nn.labml.ai/rl/dqn/index.html.

[7] URL https://www.gymlibrary.ml/environments/atari/double_dunk/.

[8] URL https://www.gymlibrary.ml/environments/atari/pong/.

[9] Part 2: Kinds of rl algorithms. URL https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html. Pristupljeno 25. lipnja 2022.

[10] URL https://www.gymlibrary.ml/environments/atari/space_invaders/.

[11] Part 1: Key concepts in rl. URL https://spinningup.openai.com/en/latest/spinningup/rl_intro.html.

[12] Actor-critic methods. URL http://www.incompleteideas.net/book/ebook/node66.html.

[13] Deep reinforcement learning and control fall 2018, cmu 10703 - policy gradient methods. URL https://www.andrew.cmu.edu/course/10-703/.

[14] Deep q-learning: An introduction to deep reinforcement learning, Apr 2020. URL https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/.

[15] AminHP. Aminhp/gym-anytrading: The most simple, flexible, and comprehensive openai gym trading environment (approved by openai gym). URL https://github.com/AminHP/gym-anytrading.

[16] M. G. Bellemare, Y. Naddaf, J. Veness, i M. Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, jun 2013.

[17] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, i Wojciech Zaremba. Openai gym, 2016.

[18] Steve Brunton. Q-learning: Model free reinforcement learning and temporal difference learning. URL https://www.youtube.com/watch?v=0iqz4tcKN58.

[19] Ian Goodfellow, Yoshua Bengio, i Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[20] Sergios Karagiannakos. The idea behind actor-critics and how a2c and a3c improve them, Nov 2018. URL https://theaisummer.com/Actor_critics/.

[21] Ayoosh Kathuria. Getting started with openai gym, Apr 2021. URL https://blog.paperspace.com/getting-started-with-openai-gym/.

[22] Zvonko Kostanjčar. Predavanja - podržano učenje. 2021.

[23] Alexander LeNail. Nn-svg: Publication-ready neural network architecture schematics. *Journal of Open Source Software*, 4(33):747, 2019. doi: 10.21105/joss.00747. URL https://doi.org/10.21105/joss.00747.

[24] Patrick Loeber. Reinforcement learning with (deep) q-learning explained, Feb 2022. URL https://www.assemblyai.com/blog/reinforcement-learning-with-deep-q-learning-explained/.

[25] Marlos C. Machado, Marc G. Bellemare, Erik Talvitie, Joel Veness, Matthew J. Hausknecht, i Michael Bowling. Revisiting the arcade learning environment: Evaluation protocols and open problems for general agents. *CoRR*, abs/1709.06009, 2017. URL http://arxiv.org/abs/1709.06009.

[26] Nicolas Maquaire. Are the space invaders deterministic or stochastic?, Sep 2020. URL https://towardsdatascience.com/are-the-space-invaders-deterministic-or-stochastic-595a30becae2.

[27] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, i Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518 (7540):529—533, February 2015. ISSN 0028-0836. doi: 10.1038/nature14236. URL https://doi.org/10.1038/nature14236.

[28] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, i Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *CoRR*, abs/1602.01783, 2016. URL http://arxiv.org/abs/1602.01783.

[29] Antonin Raffin. Rl baselines3 zoo. https://github.com/DLR-RM/rl-baselines3-zoo, 2020.

[30] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, i Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021. URL http://jmlr.org/papers/v22/20-1364.html.

[31] Nicholas Renotte. Build a chrome dino game ai model with python | ai learns to play dino game. URL https://www.youtube.com/watch?v=vahwuupy81A.

[32] Talendar. Flappy bird for openai gym. URL https://github.com/Talendar/flappy-bird-gym.

[33] Tensortrade-Org. Tensortrade-org/tensortrade: An open source reinforcement learning framework for training, evaluating, and deploying robust trading agents. URL https://github.com/tensortrade-org/tensortrade.

[34] Hado van Hasselt, Arthur Guez, i David Silver. Deep reinforcement learning with double q-learning, 2015. URL https://arxiv.org/abs/1509.06461.

[35] Wikipedia. Reinforcement learning — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Reinforcement%20learning&oldid=1090221087, 2022. Pristupljeno 25. lipnja 2022.

[36] Jian Xu. How to beat the cartpole game in 5 lines, Feb 2021. URL https://towardsdatascience.com/how-to-beat-the-cartpole-game-in-5-lines-5ab4e738c93f.

[37] Chris Yoon. Deriving policy gradients and implementing reinforce, May 2019. URL https://medium.com/@thechrisyoon/deriving-policy-gradients-and-implementing-reinforce-f887949bd63.

[38] Marko Čupić. Duboko učenje - optimizacija parametara modela. 2019.

[39] Marko Čupić. *Umjetna Inteligencija - Uvod u strojno učenje*. 2022. http://java.zemris.fer.hr/nastava/ui/ml/ml-20220430.pdf.

[40] Jan Šnajder. Strojno učenje: 2. osnovni koncepti. 2020.

Automatizirani razvoj agenata za različita okruženja

Sažetak

Ovaj rad proučava problematiku podržanog učenja gdje se bez znanja o pravilima i funkcioniranju specifične okoline, želi konstruirati agent kojemu je cilj pronaći optimalnu strategiju koja maksimizira očekivanu dobit u određenom vremenskom okviru. Osim opisa podržanog učenja, dubokih modela i algoritama koji se baziraju na dubokim modelima, detaljno je prezentirana struktura *OpenAI Gym* biblioteke i njenih okolina, te napravljena usporedba u kojoj se analizira uspješnost naučenih agenata.

Programsko rješenje implementirano je u programskom jeziku *Python*, primarno koristeći *PyTorch* radni okvir za dizajn i učenje dubokih neuronskih mreža, te *OpenAI Gym* biblioteka za simulaciju i testiranje ponašanja agenata.

Ključne riječi: Podržano učenje, duboko učenje, neuronske mreže, OpenAI Gym, algoritmi neovisni o modelu

Automated design of agents for various environments

Abstract

This paper studies the issue of reinforcement learning where without knowing the rules and internal dynamics of environment, the goal is to build an agent whose aim is to find the optimal strategy that maximizes the expected reward in a given time frame. Aside from describing the basics of reinforcement learning, deep learning models and model free algorithms, the key concepts of *OpenAI Gym* library and its environments are broken down in detail, and a comparison is made between agents implemented using....

Software implementation is written in *Python*, primarily using the *PyTorch* framework for designing and learning deep neural networks, and the *OpenAI Gym* library for simulating and testing agent behavior.

Keywords: Reinforcement learning, deep learning, neural networks, OpenAI Gym, model-free algorithms

POPIS SLIKA

2.1. Ciklus interakcije agenta s okolinom [35] 5
2.2. Arhitektura potpuno povezanog modela 6
2.3. Arhitektura konvolucijskog modela 8
2.4. Gruba podjela algoritama podržanog učenja [9] 9
2.5. Razlika u strukturama koje koriste Q učenje i Duboko Q učenje [14] . 11
2.6. Algoritam dubokog Q učenja s ponavljanjem iskustva [27] 13
2.7. Algoritam dvostrukog dubokog Q učenja s ponavljanjem iskustva . . 14
2.8. Algoritmi gradijenta politike [13] 16
2.9. Struktura akter-kritičar algoritma [12] 16
2.10. Algoritam prednosni akter-kritičar (*A2C*) [28] 18

3.1. Rezultat pokretanja koda 3 20
3.2. Vektorizirano okruženje sa 4 paralelne asinkrone instance 23
3.3. Izgled i vrijednosti CartPole okoline [36] 26
3.4. Izgled Breakout okoline 27

5.1. Breakout okoline prije i nakon korištenja omotača `WarpFrame` . . . 33

6.1. *Gym AnyTrading* okolina s označenim kupovnim i prodajnim pozicijama 42

POPIS TABLICA

3.1. Opis valjanih akcija okoline CartPole - atribut `action_space` . . . 25
3.2. Opis strukture okoline okoline CartPole - atribut `observation_space` 25
3.3. Opis valjanih akcija okoline Breakout - atribut `action_space` . . . 26
3.4. Opis strukture objekta okoline Breakout - atribut `observation_space` 27

POPIS PROGRAMSKIH ISJEČAKA

1. Implementacija potpuno povezanog modela na slici 2.2 koristeći biblioteku *PyTorch* 6
2. Implementacija konvolucijskog modela na slici 2.3 koristeći biblioteku *PyTorch* 8
3. Jednostavan primjer integracije agenta i *Gym* okoline (1 epizoda) . . . 19
4. Primjer korištenja strukture kontinuiranog prostora `Box` 21
5. Primjer korištenja strukture diskretnog prostora `Discrete` 22
6. Jednostavan primjer korištenja *Stable Baselines3* biblioteke 28
7. Naredba za izvođenje *Python* skripte i pokretanje predtreniranog modela 29
8. Generičko instanciranje unaprijedne potpuno povezane mreže 31
9. Struktura za definiranje atributa konvolucijskog sloja 31
10. Generičko instanciranje konvolucijske neuronske mreže 32
11. Instanciranje neuronskih mreža 34
12. Struktura spremnika za ponavljanje 34
13. Implementacija *epsilon-greedy exploration* strategije 35
14. Izračun gubitka dubokog Q učenja 35
15. Izračun gubitka dvostrukog dubokog Q učenja 36
16. Inicijalizacija potpuno povezanog dubokog modela algoritma akter-kritičar 37
17. Unaprijedni prolaz algoritma akter-kritičar 37
18. Odabir akcije i pohrana podataka za evaluaciju algoritma *A2C* 37
19. Evaluacija aktera i kritičara u algoritmu *A2C* 38