Razvoj metoda umjetne inteligencije u obliku orkestriranog mikroservisa

Valjin, Luka

Undergraduate thesis / Završni rad

2022

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:211:729886

Rights / Prava: Attribution 3.0 Unported/Imenovanje 3.0

Download date / Datum preuzimanja: **2024-04-16**

Repository / Repozitorij:

Faculty of Organization and Informatics - Digital Repository

SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
VARAŽDIN

Luka Valjin

RAZVOJ METODA UMJETNE INTELIGENCIJE U OBLIKU ORKESTRIRANOG MIKROSERVISA

ZAVRNI RAD

Varaždin, 2022.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Luka Valjin

Matični broj: 45853/17–R

Studij: Poslovni sustavi

RAZVOJ METODA UMJETNE INTELIGENCIJE U OBLIKU ORKESTRIRANOG MIKROSERVISA

ZAVRNI RAD

Mentor :

Izv. prof. dr. sc. Markus Schatten

Varaždin, rujan 2022.

Luka Valjin

Izjava o izvornosti

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

Autor potvrdio prihvaćanjem odredbi u sustavu FOI-radovi

Sažetak

Proučavanje tehnologije implementacije orkestriranih mikroservisa (docker swarm/kubernetes) te implementacija metode umjetne inteligencije kao orkestriranog mikroservisa. Tehnologija implementacije mikroservisa je Docker. Mikroservis implementira metodu učenja podrškom koji upravlja letjelicom Lunar Lander iz okruženja Gym. Odabrana metoda implementirana je u programskom jeziku Python, a za implementaciju neuralne mreže oslanja se na biblioteku tensorflow. Cilj igre u okruženju je uspješno sletjeti letjelicu u što kraćem vremenu i sa što manjom potrošnjom goriva. Mikroservis komunicira s okruženjem preko POST metode API-ja, i zapakiran je u Docker kontejner. Završni rad definran je kao proof of concept riješenje i veže uz projekt HRZZ/O _HAI4Games.

Ključne riječi: umjetna inteligencija; orkestracija; mikroservis; docker; Kubernets; python; učenje podrškom; tensorflow

Sadržaj

1. Uvod 1

2. Metode i tehnike rada 2

3. Teorijski uvod 3

3.1. Učenje podrškom 3

3.1.1. Q-učenje 6

3.2. Mikroservisna arhitektura 7

4. Opis korištenih tehnologija 9

4.1. Docker 9

4.2. Docker swarm 10

4.3. Tensorflow / Keras 10

4.4. Gym enviroment 10

4.5. FastAPI 11

5. Opis implementacije 12

5.1. Agent 12

5.2. env only 16

5.3. API 17

5.4. Dockerfile 18

6. Zaključak 20

Popis literature 22

Slike 23

Popis kodova 24

1. Kod aplikacije 25

1. Uvod

Tema ovog rada je 'Razvoj metoda umjetne inteligencije u obliku orkestriranog mikroservisa' i veže se uz projekt HRZZ/O _HAI4Games. Cilj ovog projekta razvoj je orkestriranih hibridnih metoda umjetne inteligencije sa konkretnom primjenom u području računalnih igara. Budući da razvoj takvog sustava uvelike nadilazi opseg završnog rada ovaj će se rad usredotočiti na implementaciju samo jedne metode umjetne inteligencije, konkretno dubokog učenja podrškom.

Ideja je da se takva metoda implementira pomoću mikroservisa koji komunikaciju ostvaruje putem API-ja, jer se takav mikroservis može lako spojiti na centralni alat za orkestraciju. Drugi ključan dio je pokretanje unutar Docker kontejnera koje omogućava lako manipuliranje mikroservisom i njegovo pokretanje neovisno o sustavu na kojem se nalazi.

Motivacija za ovaj rad proizlazi iz moje uključenosti u AI lab na Fakultetu Organizacije i Informatike koji vodi Izv. prof. dr. sc. Markus Schatten, koji je potakao moje zanimanje za područje umjetne inteligencije.

2. Metode i tehnike rada

Glavna metoda ovog završnog rada izrada je funkcionalnog mikroservisa u području umjetne inteligencije, točnije učenja podrškom. Programski jezik korišten u izradi aplikacije je python. Za implementaciju neuronske mreže koristio se modul tensorflow, odnosno keras modul za neuronske mreže koji se nalazi unutar njega. Za implementaciju API-ja koristi se FastAPI modul pokretan sa uvicornom. Okruženje u kojem je trenirao agent je LunarLander okruženje koje je dio gym modula. Gym je open source modul sa raznim okruženjima poogodnim za treniranje agenata koji uče mehanizmom povratne sprege.

3. Teorijski uvod

Teorijski dio ovog završnog rada obrađuje učenje podrškom i specifičnije Q-učenje koje se koristi u implementaciji agenta. Teoretsko razumijevanje paradigme učenja podrškom potrebno je za razumijevanje problematike koja se iznosi i danog rješenja. To se razumijevanje nužno oslanja na neke matematičke formule i koncepte koji su objašnjeni dovoljno za razumijevanje implementacije.

Drugo teoretsko područje na kojem se zasniva ovaj rad je arhitektura same aplikacije. Teorijski uvod opisuje prednosti i mane mikroservisne arhitekture u odnosu na standardnu arhitekturu monolita kako bi razumjeli zbog čega je odabrana ta arhitektura i što se njome želi postići.

3.1. Učenje podrškom

Učenje podrškom (*reinforcement learning*) jedna je od 3 osnovne paradigme strojnog učenja, i bitno se razlikuje od nadgledanog (supervised) i nenadgledanog (unsupervised) strojnog učenja u samom pristupu problematici strojnog učenja. Korijeni ove ideje mogu se pronaći u grani psihologije, specifičnije u proučavanjima učenja metodom pokušaja i pogrešaka. Drugi izvor ove ideje zasniva se na problematici optimalne kontrole. Napretkom informacijskih tehnologija i područja umjetne inteligencije dolazi do spajanja ovih problematika u današnjem području učenja podrškom. [1]

Temelj ovog pristupa zasniva se na postojanju agenta i okruženja u kojem se taj agent nalazi. Smisao postojanja takvog agenta zadan je unaprijed definiranim ciljem koji agent pokušava ispuniti. Kreator agenta određuje pozitivne i negativne ciljeve postavljanjem nagradi i kazni koje agent dobiva ovisno o postignutom stanju unutar zadanog okruženja. Taj dio pristupa mogao bi se okarakterizirati kao nadgledano učenje. Iako bi se odabir akcije iz skupa mogućih akcija mogao okarakterizirati kao nenadgledano učenje budući da se ne oslanja na postojeće primjere točnog ponašanja, bitna je razlika da učenje podrškom pokušava maksimizirati nagradu dok nenadgledano učenje za cilj ima pronalazak unutarnje skrivene strukture. Iako su neki pokušali smjestiti ovakav pristup kao polunadgledano učenje većina se stručnjaka slaže kako je učenje podrškom zasebna paradigma.

Osim agenta i okruženja bitni elementi sistema za učenje podrškom su politika, signal nagrade, funkcija vrijednosti i opcionalno model okruženja.

Politiku u ovom kontekstu možemo definirati kao mapiranje stanja okruženja na akcije koje trebamo izvršiti u tim stanjima. Takva politika srž je samog agenta budući da definira njegovo ponašanje. Implementacija same politike može se bitno razlikovati od slučaja do slučaja a time i njena kompleksnost, koja seže od jednostavnog rječnika do naprednih algoritama za pretraživanje.

Signal nagrade reprezentacija je cilja u paradigmi učenja podrškom. U svakom vremenskom koraku okruženje agentu šalje povratnu informaciju u obliku vrijednosti nagrade. Agent

koristi tu informaciju za optimizaciju svog djelovanja u svrhu dugoročne maksimizacije ukupne nagrade.

Blizak pojam signalu nagrade je funkcija vrijednosti. Ona je pokušaj vremenske reprezentacije nagrade po čemu se razlikuje od signala nagrade koji se odnosi samo na trenutno stanje. Funkcija vrijednosti aproksimira buduća stanja okruženja i tako omogućava agentu da u obzir uzima i vremensku komponentu djelovanja, a ne samo trenutnu nagradu. Sukladno tome funkciju vrijednosti možemo zamisliti kao matematičku reprezentaciju psihološkog pojma odgađanja zadovoljstva u kojem agent odabire stanje manje nagrade zbog predviđanja da takvo stanje vodi u stanja s većom nagradom. Efektivno je to odabir suboptimalnog trenutnog stanja radi dugoročne maksimizacije nagrade. Za razliku od trenutne nagrade koja je poznata modelu, dugoročnu vrijednost neke akcije puno je teže odrediti jer zahtjeva da agent uzme u obzir sva buduća stanja koja će neka akcija prouzročiti. Budući da je ta procjena zadaća funkcije vrijednosti lako je uočiti njenu centralnu ulogu u uspješnosti nekog agenta.

Opcionalni element sustava za učenje podrškom je model okruženja. Prema tome koristi li se model sustave možemo podijeliti na sustave temeljene na modelima (model-based) i sustave bez modela (model-free). Metode zasnovane na korištenju modela okrenute su planiranju dok se metode bez modela baziraju na principu pokušaja i pogrešaka. Treba napomenuti da u praksi ta podjela nije izričito binarna već se nalazi na spektru između ove dvije opcije, ovisno koliko se neki sustav bazira na modelu. Prednost sustava bez modela njegova je široka primjena i generalizacija, budući da agent tretira okruženje po principu crne kutije, to jest procesira samo ulaze i izlaze bez potrebe za poznavanjem unutarnjeg modela okruženja. Takav je pristup naročito pogodan kod stohastičkih procesa koji nisu zasnovani na jasnim uzročno posljedičnim vezama.

Za dublje razumijevanje ove paradigme potrebno je ući u matematička i informatička načela na kojima je ova metoda zasnovana. Markovljev proces odlučivanja je model čija se problematika zasniva na povratnoj informaciji koju je moguće evaluirati i ovisnosti ishoda o akciji agenta za svako stanje. Markovljev proces odlučivanja je matematički idealiziran oblik problema učenja podrškom. To nam omogućava da formaliziramo proces sekvencijalnog donošenja odluka.

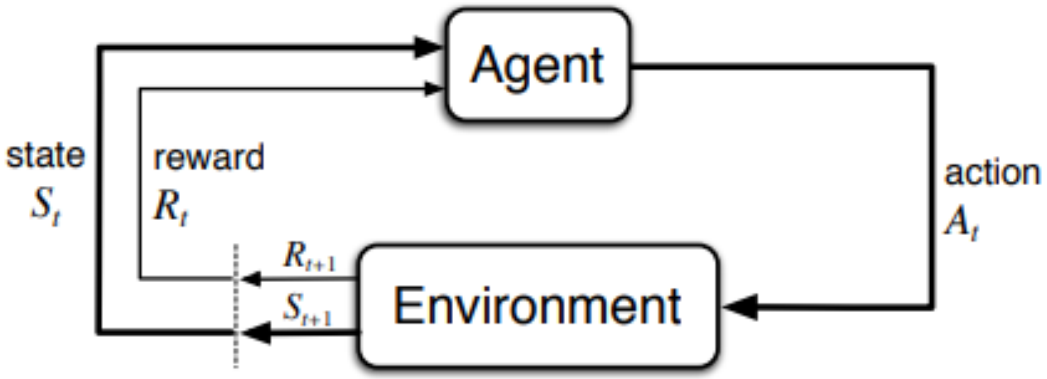
Markovljev proces zasniva se agentu, okruženju i njihovoj interakciji u slijedu diskretnih vremenskih koraka. Definira ga:

- skup stanja (S)
- skup akcija (A) - (A_s) je skup mogućih akcija u nekom stanju (s)
- vjerojatnost stanja i nagrade u sljedećem vremenskom koraku s obzirom na stanje i akciju u trenutnom vremenskom koraku:

$$p(s', r|s, a) = Pr\{S_t = s', R_t = r|S_{t1} = s, A_{t1} = a\},$$

Proces započinje tako da agent od okruženja dobiva $S_t \in S$,stanje s u trenutku t, iz skupa svih mogućih stanja. Agent zatim odabire akciju A_t iz skupa mogućih akcija u tom

stanju, $A_t \in A(s)$. Okruženje zbog te akcije prelazi u stanje S_{t+1} i agentu daje nagradu R_{t+1} . Agent se tada nalazi u stanju S_{t+1} i ukoliko epizoda nije gotova ponovno se započinje ovaj proces.



Slika 1: Interakcija agenta i okruženja u markovljevom procesu odlučivanja (izvor: Sutton i Barto, 2018)

Formalno definiranje kumulativne nagrade u Markovljevu procesu naziva se očekivani povrat G. On se definira kao zbroj svih nagrada do finalnog vremenskog koraka T

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \ + R_T$$

Ovakav pristup ima smisla ako djelovanje agenta ima prirodni zadnji vremenski korak. Takve cjeline od početnog do terminalnog stanja nazivamo epizode. Takve zadaće nazivamo i epizodne zadaće, te je njihova karakteristika da svaka nova epizoda započinje neovisno o završetku prethodne.

Suprotnost epizodnim zadaćama kontinuirane su zadaće. Interakcija agenta i okoline u takvim zadaćama nema prirodno raspoređene cjeline već kontinuirano nastavlja bez završetka. Problem nastaje kod pokušaja formulacije takvog problema gore navedenom jednadžbom. Budući da posljednji vremenski korak ne postoji očekivani vremenski povrat može biti beskonačan. Kako bi doskočili tom problemu uvodimo koncept umanjenja nagrade. U ovakvom pristupu agent pokušava odabrati akciju koja maksimizira očekivani umanjeni povrat po formuli:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

γ - gamma je parametar između 0 i 1 koji se naziva stopa umanjenja
γ određuje trenutnu vrijednost budućih nagrada na način da skorije nagrade imaju veću vrijednost. Na taj način možemo izbjegnuti beskonačni očekivani povrat.

Politika (π) je, u kontekstu Markovljevog procesa odlučivanja, funkcija koja mapira određeno stanje na vjerojatnosti odabira svake moguće akcije iz prostora akcija. Kako bi odredili ispravnu politiku koju model prati koristimo se funkcijama koje određuju koliko je dobro neko stanje ili neka akcija u određenom stanju. Takve funkcije nazivamo funkcijama vrijednosti. S

obzirom na to računamo li očekivani povrat za neko stanje ili akciju razlikujemo funkcije vrijednosti stanja i funkcije vrijednosti akcije.

Funkciju vrijednosti stanja definira funkcija

$$v_\pi(s) = E_\pi\left[G_t|S_t = s\right] = E_\pi\left[\sum_{k=0}^{\infty}\gamma^k R_{\mathbf{t+k+1}}|S_t = s\right], \forall s \in S$$

gdje je E_π očekivani povrat počinjemo li iz stanja s i pratimo li politiku π.

Sukladno tome, ukoliko želimo dobiti vrijednost akcije u pojedinom stanju koristimo funkciju vrijednosti akcije

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

koja računa očekivani povrat krećemo li iz stanja s, sa akcijom a i nakon toga pratimo politiku π.

Kako bi odredili koju politiku agent treba pratiti uvodimo pojam optimalne politike. Optimalna politika je ona koja ostvaruje najveći očekivani povrat kroz optimalnu funkciju vrijednosti stanja

$$v_*(s) = \max_\pi v_\pi(s), \forall s \in S$$

odnosno optimalnu funkciju vrijednosti akcije

$$q_*(s,a) = \max_\pi q_\pi(s,a), \forall s \in S, \forall a \in A$$

Svojstvo q_* je da zadovoljava

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

Ovo je Bellmanova jednadžba optimalnosti i tvrdi da je vrijednost akcije a u stanju u stanju s u vremenskom koraku t jednaka zbroju nagrade dobivene nakon te akcije i maksimalnog očekivanog umanjenog povrata koji možemo dobiti iz sljedećeg mogućeg para stanja i akcije. Bitno svojstvo ovo jednadžbe je njena rekurzivnost za svaki korak do završnog, tako da najbolja politika odabire akciju koja nas vodi u sljedeće stanje s' iz kojeg je moguće izvršiti najbolju akciju u koraku $t+1$.

3.1.1. Q-učenje

Jedan od najpoznatijih algoritama učenja podrškom je Q-učenje čiji je tvorac Chris Watkins. Algoritam spada u skupinu algoritama koja ne prati pohlepnu politiku, što znači da ne bira uvijek onu akciju za koju podrazumijeva da će dobiti najveću nagradu već je slobodan istražiti druge opcije u određenom postu slučajeva. Definira se kao funkcija:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right]$$

Ova funkcija vrijednosti akcije Q aproksimira optimalnu funkciju vrijednosti akcije, neovisno o tome koju politiku agent prati. $Q(S, A)$ funkcija koju algoritam računa označava očekivanu nagradu za akciju u zadanom stanju. Ažurirana Q-vrijednost računa se kao trenutna vrijednost na koju pridodajemo umnožak stope učenja i razliku vremenskog koraka. Razlika vremenskog koraka računa se kao dobivena nagrada uvećana za umnožak stope umanjenja i očekivane optimalne vrijednosti umanjena za trenutnu vrijednost. Dokazano je da ova funkcija konvergira u optimalnu funkciju vrijednosti q_*.

Algoritam Q-učenja izgleda ovako:
Inicijaliziramo $Q(s, a), \forall s \in S, \forall a \in A$ na bilo koju vrijednost (najčešće 0)
Ponovimo za svaku epizodu:

- Inicijaliziramo S

  Ponovimo za svaki korak u epizodi:

  - Odaberemo A iz S koristeći politiku izvedenu iz Q
  - Izvedemo akciju A i zabilježimo RiS'
  - $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right]$
  - $S \leftarrow S'$

  Dok S nije završno stanje

α označava stopu učenja, to je omjer u kojem nova dobivena vrijednost utječe na Q vrijednost za taj par stanja i akcije. Broj je između 0 i 1, gdje je 1 maksimalno korištenje nove vrijednosti.

Kod dubokog Q učenja za aproksimaciju funkcije koristi se q-neuronska mreža. Ulazni sloj neuronske mreže sačinjava stanje okruženja. Izlazni sustav veličine je prostora akcija. Neuronska mreža vraća aproksimaciju q vrijednosti za svaku akciju koja se može izvesti u tom stanju.

3.2. Mikroservisna arhitektura

Mikroservisna arhitektura zasniva se na skupu labavo povezanih, specijaliziranih servisa koji zajedno pružaju kompletnu uslugu. Svaki od tih servisa zaseban je samostojeći modul koji se može samostalno implementirati. Osnovna je pretpostavka mikroservisne arhitekture sposobnost različith mikroservisa da međusobno komuniciraju. Svaki taj mikroservis možemo promatrati kao zasebnu aplikaciju no nužno je na umu imati da je isti često dio neke veće cjeline. [2]

Suprotan pojam pojmu mikroservisne arhitekture je aplikacija kao monolit, što označava da je cijela aplikacija sadržana unutar jedne cjeline i neovisna o drugom aplikacijama. Takav pristup ima svoje prednosti no s razvojem sve većih i većih aplikacija dolazi do okretanja mikroservisnom pristupu. Prednost logičkog odvajanja aplikacije u više samostalnih cjelina su

povećana fleksibilnost i agilnost razvoja i implementacije. Budući da je mikroservis zasebna logička cjelina bitno je samo održavanje ulaza i izlaza iz mikroservisa stalnima kako bi se održala logika sustava. Unutarnja implementacija, pa čak i korištene tehnologije mogu se mijenjati bez da se promjene propagiraju na ostale dijelove aplikacije, što nije slučaj kod monolita. Neovisnost mikroservisa omogućava veću skalabilnost dodavanjem instanci samo pojedinog servisa a ne cijelog monolita, kao i lakšu implementaciju promjena bez rušenja kompletne aplikacije. Navedene promjene razlog su sve većeg korištenja ovog pristupa u modernim razvojima aplikacije.

Temeljna prednost ovog pristupa s obzirom na sadržaj ovog završnog rada je mogućnost orkestracije različitih mikroservisa. U ovom slučaju svaki mikroservis jedan je model strojnog učenja. Povezivanjem mikroservisa u ansambl pokušavamo povećati prediktivne sposobnosti cijelog sustava, zadržavajući jedinstvenost svakog mikroservisa koji po potrebi može biti uključen ili isključen. Time dobivamo model visoke preciznosti zadržavajući agilnost ansambla lakom implementacijom promjena ovisno o potrebama okoline.

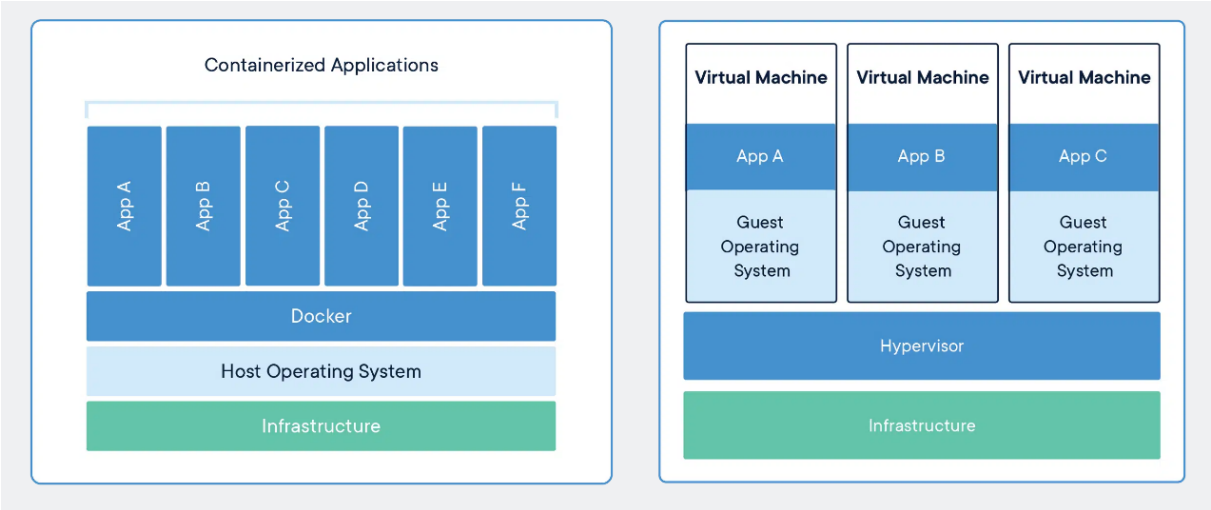
4. Opis korištenih tehnologija

U ovom se poglavlju opisuju tehnologije korištene u implementaciji ovog rješenja. To su Docker kao tehnologija za kontejnerizaciju, Docker Swarm kao alat za orkestraciju kontejnera, Teneorflow / Keras kao modul za neuronske mreže, Gym modul koji pruža okruženje za treniranje agenta i FastAPI modul za implementaciju API sučelja.

4.1. Docker

Docker je skup proizvoda koji se baziraju na principu virtualizacije na razini operacijskog sustava. Temeljni koncepti rada s dockerom su slika (image) i kontejner (container). Kao alat za pakiranje aplikacija koji pruža izolaciju okruženja, prenosivost i agnostički pristup kodu unutar aplikacije Docker je postao čest alat za razvoj i implementaciju mikroservisne arhitekture. [3]

Docker kontejner osnovna je runtime jedinica koja sadrži instancu neke aplikacije zajedno sa svim njezinim ovisnostima. To je izolirani proces koji se izvodi na operacijskom sustavu. Kontejner se zbog načina korištenja i arhitekture često uspoređuje s virtualnim mašinama, no ključno je uočiti razlike između ta dva koncepta. Najveća je razlika to što kontejneri virtualiziraju operacijski sustav dok virtualne mašine virtualiziraju hardver. Kontejneri zato dijele ljusku operacijskog sustava dok svaka virtualna mašina zahtjeva vlastitu kopiju operacijskog sustava. Posljedica takve razlike u arhitekturi je da kontejneri zahtijevaju manje vremena za podizanje sustava i zauzimaju manje prostora. [4]



Slika 2: arhitektura kontejnera i virtualne mašine (izvor: Docker službena dokumentacija)

Pojam koji se usko veže uz docker kontejner je docker slika. To je nepromjenjivi set naredbi kojim se jednoznačno definira način izgradnje pripadajućeg kontejnera. Sastoji se od aplikacije skupa s ovisnostima i meta-podataka o načinu na koji se slika treba pokrenuti. Sliku tako možemo zamisliti kao nacrt prema kojem se gradi kontejner. Jednom izgrađena slika ne može se mijenjati, ukoliko želimo napraviti promjene moramo izgraditi novu sliku. Slika se sastoji od slojeva, svaki sloj izvršava neke promjene na okruženju. Na temelju neke slike

možemo kreirati od 0 do n kontejnera. [5]

4.2. Docker swarm

Docker Swarm je riješenje za orkestraciju kontejnera koje dolazi ugrađeno s Dockerom. Ovaj alat omogućava kreiranje i upravljanje klasterom Docker čvorova kao jednim virtualnim sistemom, čak i ako se kontejneri ne nalaze na jednom fizičkom uređaju. Swarm ima mogućnost balansiranja tereta. [6]

4.3. Tensorflow / Keras

Tensorflow je modul za python, dostupan i u nekim drugim programskim jezicima, čija je osnovna uporaba u području strojnog učenja i umjetne inteligencije. Modul je besplatan i otvorenog koda (open-source) a razvijen je od strane Googleovog Google Brain Teama. Unutar samog modula nalazi se i Keras modul koji je implementiran kao sučelje koje korisnicima omogućava kreiranje neuralnih mreže i sve pripadajuće komponente za njihovo treniranje i korištenje. Zbog svoje kvalitete i dostupnosti ovaj je modul najčešće korišteni modul u izradi neuronskih mreža. [7]

4.4. Gym enviroment

Gym je otvoreno razvojno okruženje koje nudi široki spektar okolina pogodnih za treniranje agenata koji uče povratnom spregom. U ovom radu konkretno se koristi LunarLander. Cilj ove okoline je istrenirati agenta da sleti letjelicu, detaljniji opis čega nalazi se kod same implementacije. [8]

LunarLander ima prostor od 4 akcije:

- koristi glavni motor (pogon se nalazi na dnu letjelice i kreće letjelicu prema gore)
- koristi lijevi motor za orijentaciju
- koristi desni motor za orijentaciju
- ne radi ništa

Svaki potisnik ima samo dva stanja, uključeno i isključeno, agent nema opciju uključivanja potiska na manjem postotku snage.

Stanje okruženja određeno je vektorom od 8 dimenzija:

- prva je x koordinata letjelice
- druga je y koordinata letjelice
- treća je brzina letjelice na x osi

- četvrta je brzina letjelice na y osi
- peta je kut nagiba letjelice
- šesta je kutna brzina
- zadnje dvije su boolean vrijednosti koje određuju je li pojedina noga letjelice u kontaktu sa površinom (letjelica ima dvije noge)

Sustav nagrađivanja temelji se na sistemu nagrada i kazni koji je sljedeći:

- nagrada za silazak s početne pozicije do mjesta za silaz je otprilike 100-140 bodova
- ako se letjelica udaljava od prostora za slijetanje gubi nagradu
- rušenjem letjelice smatra se ako njeno tijelo dotakne zemlju i rezultira gubitkom od 100 bodova
- ako letjelica dođe u poziciju mirovanja dobiva 100 bodova
- svaka noga u kontaktu sa zemljom donosi 10 bodova (maksimalno 20)
- korištenje glavnog motora rezultira penalom od -0.3 bodova po okviru (frame)
- korištenje bilo kojeg bočnog motora rezultira penalom od -0.03 bodova po okviru (frame)

Prostor za slijetanje nalazi se u koordinatama (0,0), omeđen je zastavicama. Letjelica započinje epizodu na vrhu ekrana sa nasumičnom silom generiranom na njezin centar mase. Moguće je i slijetanje izvan označenog prostora za slijetanje. Gorivo je beskonačno.

Epizoda završava ukoliko:

- dođe do rušenja letjelice, točnije njeno tijelo dodirne tlo
- letjelica napusti vidljivi prostor (prema x i y koordiantama)
- ukoliko dođemo do maksimalnog broja koraka koji je u našem slučaju postavljen na 100

4.5. FastAPI

FastAPI je web framework za izgradnju API-ja unutar Python programskog jezika. Odlikuje se visokom brzinom i koristi Pythonov type hinting za validaciju i serializaciju podataka. Budući da modul nema vlastiti WSGI server za njegovo pokretanje koristi se uvicorn. [9] Tvorac modula je Sebastian Ramirez a vlasnik licence MIT. [10]

5. Opis implementacije

Implementacija ovog mikroservisa sastoji se od nekoliko koraka. Prvi korak je kreiranje agenta koji je sposoban učiti kroz interakciju sa svojim okruženjem. Zatim slijedi proces treniranja agenta kroz niz pokušaja. Nakon što smo istrenirali agenta na okruženju potrebno je implementirati ga kao mikroservis. Konkretna implementacija mikroservisa u ovom je radu implementirana kao API pomoću kojeg agent komunicira s okruženjem. Agent se skupa s API-jem nalazi unutar Docker kontejnera.

5.1. Agent

Agent je skripta u kojoj se kreira agent u klasi Agent i provodi se njegovo treniranje unutar main funkcije. Za treniranje je potrebno i učitavanje LunarLander okruženja s kojim agent ima interakciju.

```
class Agent:
    def __init__(self, state_space, action_size, batch_size=64, memory_size=1000000)
        :
        self.state_space = state_space
        self.action_size = action_size
        self.batch_size = batch_size
        self.memory = deque(maxlen=memory_size)
        self.gamma = 0.98
        self.epsilon = 1  # exploration rate
        self.epsilon_min = 0.02
        self.epsilon_decay = 0.99
        self.learning_rate = 1e-3
        self.model = self._build_model()
```

Klasa Agent sadrži varijable:

- *state_space* definira veličinu prostora varijabli koje određuju trenutno stanje okruženja, u ovom je slučaju to 8
- *action_size* je prostor akcija koje agent može poduzeti
- *batch_size* je veličina serije koja se koristi prilikom serijskog retreniranja modela na zapamćenim primjerima
- *memory* je veličina memorije u koju se spremaju povjesni podatci za retreniranje modela
- gamma je stopa umanjenja i označava koliko je manja vrijednost nagrade u svakom sljedećem vremenskom koraku u odnosu na trenutni. Ako je gamma 0.98 vrijednost nagrade nakon n koraka u sadašnjosti ima vrijednost te nagrade pomnožene sa 0.98 na n-tu
- epsilon označava stopu istraživanja, točnije početnu stopu istraživanja budući da se konstantno ažurira tjekom izvođenja aplikacije. Stopa od 1 označava da će agent uvijek oda-

brati nasumičnu akciju, što je dobro za početak budući da agent nema dovoljno znanja za donošenja odluke o akciji.

- epsilon_decay je stopa opadanja stope istraživanja. To se implementira na načina da se nakon svake iteracije epsilon množi sa tom stopom. U ovom konkretnom primjeru epsilon je nakon svake iteracije 1% manji. Time se postiže da agent nakon svake iteracije bira sve manje nasumičnih akcija.
- learning_rate je stopa učenja. Ona određuje u kolikom omjeru nove vrijednosti utječu na promjene u težinama modela.

```
def _build_model(self):
    model = Sequential()
    model.add(Dense(128, input_dim=self.state_space, activation='relu'))
    model.add(Dense(128, activation='relu'))
    model.add(Dense(128, activation='relu'))
    model.add(Dense(self.action_size, activation='linear'))
    model.compile(loss='mse', optimizer=Adam(learning_rate=self.learning_rate),
        metrics=['accuracy'])
    return model
```

Isječak koda 5.1: agent: inicijalizacija klase

Definiramo neuralnu mrežu sa sa 3 unutranja sloja. Aktivacija unutarnjih slojeva je relu, gubitak je prosječna kvadrirana pogreška, optimizator je Adam. [11] [12] [13]

```
def memorize(self, state, action, reward, next_state, done):
    self.memory.append((state, action, reward, next_state, done))
```

Isječak koda 5.2: agent: funkcija memorizacije

Funkcija za memoriziranje sprema u memoriju informacije o stanju, akciji, nagradi, sljedećem stanju i tome je li epizoda gotova.

```
def act(self, state):
    if random.uniform(0, 1) < self.epsilon:
        return random.randrange(self.action_size)
    else:
        return np.argmax(self.model.predict(state)[0])
```

Isječak koda 5.3: agent: funkcija dobivanja akcije

Funkcija koja vraća akciju iz agenta vraća radnom funkciju u onom postotku slučaja jednakom trenutnom epsilonu. Prvo iz memorije dohvaćamo 32 nasumično odabrana spremljena stanja.

```
def experience_replay(self):
    if self.batch_size > len(self.memory):
        return

    random_batch = random.sample(self.memory, self.batch_size)
```

```
        state = np.zeros((self.batch_size, self.state_space))
        next_state = np.zeros((self.batch_size, self.state_space))
        action, reward, done = [], [], []

        for i in range(self.batch_size):
            state[i] = random_batch[i][0]
            action.append(random_batch[i][1])
            reward.append(random_batch[i][2])
            next_state[i] = random_batch[i][3]
            done.append(random_batch[i][4])

        target = self.model.predict(state)
        target_next = self.model(next_state)

        for i in range(len(random_batch)):
            if done[i]:
                target[i][action[i]] = reward[i]
            else:
                target[i][action[i]] = reward[i] + self.gamma * (np.amax(target_next
                    [i]))

        self.model.fit(
            np.array(state),
            np.array(target),
            batch_size=self.batch_size,
            verbose=0
        )
```

Isječak koda 5.4: agent: funkcija iskustva

Učenje modela zasniva se na uzimanju serije nasumičnih prošlih stanja. Veličina serije u našem je slučaju prethodno definirana na 32 stanja. Kako bi imali reprezentativno stanje iz kojeg agent može učiti iskustvo predstavljamo stanjem u koraku t, akcijom u tom koraku, nagradu koja se vratila nakon te akcije i stanje u koraku t+1. Nasumičan odabir primjera razbija korelaciju koju uzimanje uzastopnih primjera sadrži u sebi. Nakon simulacije izvođenja zadane aktivnosti u zadanom stanju pozovemo fit metodu modela kako bi se model istrenirao na zadanim podatcima.

```
if __name__ == "__main__":

    env = gym.make('LunarLander-v2')  # uzimanje enviromenta

    n_episodes = 300
    num_episode_steps = env.spec.max_episode_steps  # constant value
    action_size = env.action_space.n
    state_space = env.observation_space.shape[0]
    max_reward = 0

    agent = Agent(state_space=state_space, action_size=action_size)

    if os.path.exists("saved_model"):
```

```
        agent.model = keras.models.load_model('saved_model')
        print('model_loaded')

    for episode in range(n_episodes):
        total_reward = 0

        observation = env.reset()

        state = np.reshape(observation, [1, state_space])

        for episode_step in range(num_episode_steps):
            env.render(mode="human")

            action = agent.act(state)

            observation, reward, done, _ = env.step(action)
            total_reward += reward

            next_state = np.reshape(observation, [1, state_space])

            agent.memorize(state, action, reward, next_state, done)

            agent.experience_replay()

            state = next_state

            if done:
                print("Episode␣%d/%d␣finished␣after␣%d␣steps␣with␣total␣reward␣of␣%f
                    ."
                      % (episode + 1, n_episodes, episode_step + 1, total_reward))
                break

            elif episode_step >= num_episode_steps - 1:
                print("Episode␣%d/%d␣timed␣out␣at␣%d␣with␣total␣reward␣of␣%f."
                      % (episode + 1, n_episodes, episode_step + 1, total_reward))

        agent.epsilon = max(agent.epsilon_min, agent.epsilon * agent.epsilon_decay)

        if total_reward >= max_reward:
            max_reward = total_reward

    agent.save_model('saved_model')

    env.close()
```

Isječak koda 5.5: glavna funkcija sa petljama za treniranje

U početku glavne funkcije učitamo LunarLander okruženje Gym modula. Odredimo broj epizoda za treniranje, u našem slučaju 300, dok je broj koraka svake epizode limitiran na 100. Početna najveća nagrada postavljena je na 0. Zatim inicijaliziramo agenta i ako postoje podatci za trenirani model učitamo ga pomoću kerasove funkcije *load_model*.

Zatim krenemo u implementaciju algoritma q-učenja u pythonu. Za svaku epizodu pos-

tavimo nagradu na 0, od okruženja dobijemo informacije o njegovom stanju i spremimo ih u varijablu state. Zatim uđemo u for petlju koja za svaki vremenski korak u epizodi uzima akciju koju odredi agent, proslijedi je okruženju, za nju dobije povratne informacije o novom stanju, nagradi i mogućem završetku epizode. Nagrada se pribroji ukupnoj nagradi za tu epizodu i taj vremenski korak se pohrani u memoriju. Agent zatim uči na prošlim stanjima, novo dobiveno stanje postaje trenutno i iteracija se ponavlja za sljedeći vremenski korak ukoliko nije došlo do kraja epizode. Nakon svake epizode epsilon se umanji kako bi povećali iskorištavanje naučenog odnosno smanjili istraživanje agenta. Napredak treniranja za svaku epizodu se ispisuje kako bi mogli pratiti stanje agenta.

Naučeni model agenta se zatim sprema i okruženje se gasi.

5.2. env only

Skripta "env only" demonstrira mikroservisnu izvedbu implementacije agenta. Unutar nje se učitava samo okruženje bez kreiranja agenta. Agent je pokrenut kao API unutar Docker kontejnera i okruženje od njega zahtjeva akcije putem POST zahtjeva na API.

```
import requests

import gym


if __name__ == "__main__":

    env = gym.make('LunarLander-v2')  # uzimanje enviromenta
    URL = 'http://localhost:8007/action'

    N_EPISODES = 5
    num_episode_steps = env.spec.max_episode_steps
    action_size = env.action_space.n
    state_space = env.observation_space.shape[0]

    for episode in range(N_EPISODES):
        total_reward = 0

        observation = env.reset()

        for episode_step in range(num_episode_steps):
            env.render(mode="human")

            data = {'variables': observation.tolist()}

            response = requests.post(URL, json=data)
            action = int(response.content)

            observation, reward, done, _ = env.step(action)
            total_reward += reward
```

```
            if done:
                print("Episode␣%d/%d␣finished␣after␣%d␣steps␣with␣total␣reward␣of␣%f
                    ."
                      % (episode + 1, N_EPISODES, episode_step + 1, total_reward))
                break

            elif episode_step >= num_episode_steps - 1:
                print("Episode␣%d/%d␣timed␣out␣at␣%d␣with␣total␣reward␣of␣%f."
                      % (episode + 1, N_EPISODES, episode_step + 1, total_reward))
```

Isječak koda 5.6: skripta koja se spaja na API

Skripta env only otvara okruženje i koristi šalje POST zahtjeve na API agenta. U tijelu POST zahtjeva šalje podatke o okruženju i kao odgovore prima Akciju koju agent odredi. Glavnina koda jednaka je kao u klasi Agent pa je nema potrebe opet objašnjavati, ključno je uočiti razliku da je agent implemetniran kao mikroservis, odnosno zasebna aplikacija koja se spaja na okruženje.

5.3. API

API programska datoteka sadržava implementaciju aplikacijskog programskog sučelja (Application Programming Interface) pomoću kojeg komuniciraju agent i okruženje.

```
import os

from fastapi import FastAPI
from tensorflow import keras
from pydantic import BaseModel
import numpy as np

import model_training

app = FastAPI()
OBSERVATION_SPACE = 8


class EnvVariables(BaseModel):
    variables: list[float]


agent = model_training.Agent(state_space=8, action_size=4, )
if os.path.exists("saved_model"):
    agent.model = keras.models.load_model('saved_model')


@app.post('/action')
def get_action(observation: EnvVariables):
    observation = np.asarray(observation.variables)
    state = np.reshape(observation, [1, OBSERVATION_SPACE])
    action = np.argmax(agent.model.predict(state)[0])
```

```
    return int(action)
```

Isječak koda 5.7: API

API je izrađen uz pomoć FastAPI modula i implementira *endpoint* za jedan POST zahtjev koji prima listu *float* vrijednosti koja opisuje okruženje. API koristi već istrenirani model od kojeg dohvaća akcije koje vraća kao odgovor.

5.4. Dockerfile

Dockerfile sadrži potrebne instrukcije za izgradnju mikroservisa koji se pokreće unutar Docker kontejnera. Proces izrade ovog mikroservisa sastoji se od uzimanja bazične slike, instaliranja potrebnih modula, kopiranja foldera aplikacije, otvaranja porta za komunikaciju i zadavanja početne naredbe za pokretanje API-ja pomoću uvicorna.

```
FROM python:3.9.13-bullseye

RUN apt-get update && apt-get install -y python3-dev python3-pip libssl-dev locales
    git openssh-server cmake curl gnupg

RUN pip3 install fastapi[all] gym[all] tensorflow keras pydantic

COPY . .

EXPOSE 8007

CMD ["uvicorn", "app:app", "--host", "0.0.0.0", "--port", "8007"]
```

Isječak koda 5.8: Dockerfile

Dockerfile je tekstualni dokument koji sadrži naredbe za izradu docker slike. Pokretanjem dockerfilea slijedno se izvode zapisane instrukcije. [14]

FROM instrukcija definira bazičnu sliku na temelju koje se gradi slika s aplikacijom koju želimo izgraditi. Bazična slika sadrži operacijski sustav i u našem slučaju Python budući da je slika 'python:3.9.13-bullseye'.

Instrukcija RUN služi za pokretanje instrukcija u terminalu, koristimo je kako bi instalirali potrebne pakete za pokretanje naše aplikacije.

COPY služi za kopiranje podataka sa lokalnog sistema na sliku. Prvi argument označava lokaciju na našem računalu a drugi lokaciju na docker slici. Lokacija se šalje u obliku putanje direktorija. Točka koju mi navodimo (".") posebno je definirana kao naredba koja uzima sve foldere u lokalnom direktoriju. Prilikom kreiranja slike nalazimo se u direktoriju aplikacije kako bi kopirali sve što se u njemu nalazi.

EXPOSE je naredba kojom omogućavamo konekciju kontejnera kreiranih iz ove slike preko navedenog porta. U našem slučaju to je port 8000. Treba napomenuti da time ne stvaramo direktnu konekciju između kontejnera i hosta (domaćina?) na kojem se on pokreće. To radimo pomoću zastavice -p ili –publish prilikom izvođenja "docker container run" naredbe.

Instrukcija CMD pruža kontejnru informacije za njegovo zadano izvršavanje. U ovom slučaju to je naredba za pokretanje uvicorna koji pokreće API.

6. Zaključak

Tema ovog završnog rada bio je razvoj metoda umjetne inteligencije u obliku orkestriranog mikroservisa u sklopu projekta O-HAI 4Games. Rad demonstrira implementaciju učenja podrškom u obliku mikroservisa. Za definiranje ulaza i izlaza modela koristio se API, ali u budućnosti se mogu pokušati koristiti i neki drugi načini povezivanja.

Rad ima prostora za napredak u pogledu orkestracije na način da se mikroservisni model spoji s drugim modelima strojnog učenja te tako sačinjava ansambl i da se to implementira pomoću Docker Swarma ili Kubernetesa. Model može napredovati i u smislu veće prilagodljivosti okruženju na način da se model spaja na okruženje različitih ulaznih veličina i tek onda kreira i trenira.

Problematično je što je tijekom izrade ovog završnog rada ukinuta podrška za windows operacijski sustav od strane Gym modula je bilo zahtjevno osposobiti sve module da rade bez međusobnih konflikata.

Ovaj rad dokazuje mogućnost implementacije metode umjetne inteligencije, specifično dubokog Q-učenja, u obliku mikroservisa koji se može spajati na neko okruženje. Orkestracija takvih metoda je moguća ali implementacija istog nadilazi opseg ovog završnog rada.

Dobrim se je pokazao odabir tehnologija koje su se koristile. Python kao programski jezik bio je dobar izbor zbog velikog broja javno dostupnih modula koju omogućavaju lako kreiranje agenta, posebice tensorflow i keras za kreiranje neuronske mreže, kao i numpy modul za baratanje numeričkim podacima. Docker se je pokazao kao odličan alat za pakiranje i implementiranje aplikacije u obliku mikroservisa.

Popis literature

[1] R. Sutton i A. Barto, *Reinforcement Learning, second edition: An Introduction* (Adaptive Computation and Machine Learning series). MIT Press, 2018., ISBN: 9780262352703. adresa: https://books.google.hr/books?id=uWV0DwAAQBAJ.

[2] J. Buelta, *Hands-On Docker for Microservices with Python: Design, deploy, and operate a complex system with multiple microservices using Docker and Kubernetes*. Packt Publishing, 2019., ISBN: 9781838822552. adresa: https://books.google.hr/books?id=EBrADwAAQBAJ.

[3] DockerInc., *Docker overview*, [pristupano: 14.09.2022]. adresa: https://docs.docker.com/get-started/overview/.

[4] DockerInc., *Docker container*, [pristupano: 14.09.2022]. adresa: https://www.docker.com/resources/what-container/.

[5] DockerInc., *Docker develop*, [pristupano: 14.09.2022]. adresa: https://docs.docker.com/develop/.

[6] DockerInc., *Docker Swarm*, [pristupano: 14.09.2022]. adresa: https://docs.docker.com/engine/swarm/.

[7] Google, *Tensorflow Keras official documentation*, [pristupano: 14.09.2022]. adresa: https://www.tensorflow.org/api_docs/python/tf/keras.

[8] openai, *Gym Lunar Lander github*, [pristupano: 14.09.2022]. adresa: https://github.com/openai/gym/blob/master/gym/envs/box2d/lunar_lander.py.

[9] Encode, *Uvicorn official documentation*, [pristupano: 14.09.2022]. adresa: https://www.uvicorn.org/.

[10] MIT, *FastAPI official documentation*, [pristupano: 14.09.2022]. adresa: https://fastapi.tiangolo.com/.

[11] Google, *Tensorflow Keras official documentation - Sequential*, [pristupano: 14.09.2022]. adresa: https://www.tensorflow.org/api_docs/python/tf/keras/Sequential.

[12] Google, *Tensorflow Keras official documentation - Adam optimizator*, [pristupano: 14.09.2022]. adresa: https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Adam.

[13] Google, *Tensorflow Keras official documentation - ReLu aktivacijska funkcija*, [pristupano: 14.09.2022]. adresa: https://www.tensorflow.org/api_docs/python/tf/keras/layers/ReLU.

[14] DockerInc., *Dockerfile reference*, [pristupano: 14.09.2022]. adresa: https://docs.docker.com/engine/reference/builder/.

Popis slika

1. Interakcija agenta i okruženja u markovljevom procesu odlučivanja (izvor: Sutton i Barto, 2018) 5

2. arhitektura kontejnera i virtualne mašine (izvor: Docker službena dokumentacija) 9

Sadržaj

5.1. agent: inicijalizacija klase 13
5.2. agent: funkcija memorizacije 13
5.3. agent: funkcija dobivanja akcije 13
5.4. agent: funkcija iskustva 13
5.5. glavna funkcija sa petljama za treniranje 14
5.6. skripta koja se spaja na API 16
5.7. API 17
5.8. Dockerfile 18

1. Kod aplikacije

```
import random
import os
from collections import deque

import gym
from tensorflow import keras
import numpy as np
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam


class Agent:
    def __init__(self, state_space, action_size, batch_size=64, memory_size=1000000)
        :
        self.state_space = state_space
        self.action_size = action_size
        self.batch_size = batch_size
        self.memory = deque(maxlen=memory_size)
        self.gamma = 0.98  # discount rate
        self.epsilon = 1  # exploration rate
        self.epsilon_min = 0.02
        self.epsilon_decay = 0.99
        self.learning_rate = 1e-3
        self.model = self._build_model()

    def _build_model(self):
        model = Sequential()
        model.add(Dense(128, input_dim=self.state_space, activation='relu'))
        model.add(Dense(128, activation='relu'))
        model.add(Dense(128, activation='relu'))
        model.add(Dense(self.action_size, activation='linear'))
        model.compile(loss='mse', optimizer=Adam(learning_rate=self.learning_rate),
            metrics=['accuracy'])
        return model

    def memorize(self, state, action, reward, next_state, done):
        self.memory.append((state, action, reward, next_state, done))

    def act(self, state):
        if random.uniform(0, 1) < self.epsilon:
            return random.randrange(self.action_size)
        else:
            return np.argmax(self.model.predict(state)[0])

    def experience_replay(self):
        if self.batch_size > len(self.memory):
            return
```

```
        random_batch = random.sample(self.memory, self.batch_size)

        state = np.zeros((self.batch_size, self.state_space))
        next_state = np.zeros((self.batch_size, self.state_space))
        action, reward, done = [], [], []

        for i in range(self.batch_size):
            state[i] = random_batch[i][0]
            action.append(random_batch[i][1])
            reward.append(random_batch[i][2])
            next_state[i] = random_batch[i][3]
            done.append(random_batch[i][4])

        target = self.model.predict(state)
        target_next = self.model(next_state)

        for i in range(len(random_batch)):
            if done[i]:
                target[i][action[i]] = reward[i]
            else:
                target[i][action[i]] = reward[i] + self.gamma * (np.amax(target_next
                    [i]))

        self.model.fit(
            np.array(state),
            np.array(target),
            batch_size=self.batch_size,
            verbose=0
        )

    def save_model(self, model_file):
        self.model.save(model_file)


if __name__ == "__main__":

    env = gym.make('LunarLander-v2')

    n_episodes = 300
    num_episode_steps = env.spec.max_episode_steps
    action_size = env.action_space.n
    state_space = env.observation_space.shape[0]
    max_reward = 0

    agent = Agent(state_space=state_space, action_size=action_size)

    if os.path.exists("saved_model"):
        agent.model = keras.models.load_model('saved_model')
        print('model_loaded')

    for episode in range(n_episodes):
        total_reward = 0
```

```
    observation = env.reset()

    state = np.reshape(observation, [1, state_space])

    for episode_step in range(num_episode_steps):
        env.render(mode="human")

        action = agent.act(state)

        observation, reward, done, _ = env.step(action)
        total_reward += reward

        next_state = np.reshape(observation, [1, state_space])

        agent.memorize(state, action, reward, next_state, done)

        agent.experience_replay()

        state = next_state

        if done:
            print("Episode␣%d/%d␣finished␣after␣%d␣steps␣with␣total␣reward␣of␣%f
                ."
                  % (episode + 1, n_episodes, episode_step + 1, total_reward))
            break

        elif episode_step >= num_episode_steps - 1:
            print("Episode␣%d/%d␣timed␣out␣at␣%d␣with␣total␣reward␣of␣%f."
                  % (episode + 1, n_episodes, episode_step + 1, total_reward))

    agent.epsilon = max(agent.epsilon_min, agent.epsilon * agent.epsilon_decay)

    if total_reward >= max_reward:
        max_reward = total_reward

agent.save_model('saved_model')

env.close()
```

```
import os
import pickle

from fastapi import FastAPI
from tensorflow import keras
from pydantic import BaseModel
import numpy as np

import model_training

app = FastAPI()
OBSERVATION_SPACE = 8


class EnvVariables(BaseModel):
    variables: list[float]


agent = model_training.Agent(state_space=8, action_size=4, )
if os.path.exists("saved_model"):
    agent.model = keras.models.load_model('saved_model')


@app.post('/action')
def get_action(observation: EnvVariables):
    observation = np.asarray(observation.variables)
    state = np.reshape(observation, [1, OBSERVATION_SPACE])
    action = np.argmax(agent.model.predict(state)[0])
    return int(action)
```

```
import requests

import gym


if __name__ == "__main__":
    env = gym.make('LunarLander-v2')
    URL = 'http://localhost:8007/action'

    N_EPISODES = 5
    num_episode_steps = env.spec.max_episode_steps  # constant value
    action_size = env.action_space.n
    state_space = env.observation_space.shape[0]

    for episode in range(N_EPISODES):
        total_reward = 0

        observation = env.reset()

        for episode_step in range(num_episode_steps):
            env.render(mode="human")

            data = {'variables': observation.tolist()}

            response = requests.post(URL, json=data)
            action = int(response.content)

            observation, reward, done, _ = env.step(action)
            total_reward += reward

            if done:
                print("Episode␣%d/%d␣finished␣after␣%d␣steps␣with␣total␣reward␣of␣%f
                    ."
                      % (episode + 1, N_EPISODES, episode_step + 1, total_reward))
                break

            elif episode_step >= num_episode_steps - 1:
                print("Episode␣%d/%d␣timed␣out␣at␣%d␣with␣total␣reward␣of␣%f."
                      % (episode + 1, N_EPISODES, episode_step + 1, total_reward))
```

```
FROM python:3.9.13-bullseye

RUN apt-get update && apt-get install -y python3-dev python3-pip libssl-dev locales
    git openssh-server cmake curl gnupg

RUN pip3 install fastapi[all] gym[all] tensorflow keras pydantic

COPY . .

EXPOSE 8007

CMD ["uvicorn", "app:app", "--host", "0.0.0.0", "--port", "8007"]
```