Primjena učenja podrškom na problem dvonožnog hodanja

Čaleta, Paula

PRIRODOSLOVNO–MATEMATIČKI FAKULTET
SVEUČILIŠTA U SPLITU

PAULA ČALETA

PRIMJENA UČENJA PODRŠKOM NA PROBLEM DVONOŽNOG HODANJA

DIPLOMSKI RAD

Split, srpanj 2022.

PRIRODOSLOVNO–MATEMATIČKI FAKULTET
SVEUČILIŠTA U SPLITU

ODJEL ZA MATEMATIKU

PRIMJENA UČENJA PODRŠKOM NA PROBLEM DVONOŽNOG HODANJA

DIPLOMSKI RAD

Studentica:
Paula Čaleta

Mentor:
doc. dr. sc. Ivo Ugrina

Split, srpanj 2022.

Uvod

Učenje kroz interakciju s okolinom vjerojatno je prvo što nam pada na pamet kada razmišljamo o prirodi učenja. Kada se beba igra, maše ručicama ili gleda okolo, nema izravnog učitelja, ali ima direktnu senzomotoričku vezu sa svojom okolinom. Ostvarivanje te veze nudi obilje informacija o uzroku i posljedicama radnji te o tome što treba napraviti da bismo ostvarili svoje ciljeve. Tijekom naših života takve su interakcije nedvojbeno neizostavni izvor znanja o našoj okolini i o nama samima. Neovisno o tome učimo li voziti automobil ili održati razgovor, konstantno smo svjesni odgovora okoline na naše postupke i nastojimo utjecati na posljedice našeg ponašanja.

Učenje podrškom (engl. *reinforcement learning*) grana je strojnog učenja koja se temelji upravo na tim principima. Ona ujedinjuje koncepte iz teorije kontrole, strojnog učenja i psihologije. Postoji konsenzus među psiholozima da je navedena pojednostavljena analogija povratne informacije o nagradi usko povezana s načinom na koji ljudski mozak radi. Ova teorija povezana je s hipotezom koja tvrdi da dopamin igra ključnu ulogu u subjektivnom užitku povezanim uz pozitivne nagrade [10]. Od tad, područje učenja podrškom procvjetalo je iz teoretskog okruženja u matematički koncept kojim se navedene teorije mogu implementirati i testirati u računalnim simulacijama ili na pravim robotima.

Unatoč izuzetnim naporima velikih majstora i profesionalaca, igre poput

šaha, Goa, Atarija i čak Catana podlegle su, mnogo ranije nego očekivano, sirovoj snazi računanja i učenja podrškom. Druga domena problema je ona koja uključuje neprekidnu dinamiku akcija za kontrolu robota u pravom svijetu. Taj tip problema još nije u potpunosti "riješen" i čovjek je daleko superiorniji u tim zadacima od najboljeg robota ikad napravljenog. Težina kontroliranja robota leži u beskonačnom broju mogućih akcija i stanja u kojima se on može naći, što čini istraživanje i generalizaciju nemogućima (ako upotrebljavamo determinističke tehnike koje rješavaju igre poput šaha). Unatoč mnogim izazovima, u zadnjih par desetljeća vidi se izniman napredak u problemima neprekidnih akcija.

Primjerice, 2021. godine tim znanstvenika sa Sveučilišta u Kaliforniji, Berkley, izgradio je dvonožnog robota koji je naučio hodati pomoću učenja podrškom. Ono što je revolucionarno kod ovog robota, nazvanog Cassie, jest što je to uspio potpuno sam, bez oponašanja ili direktnog programiranja. Proces učenja počeo je u virtualnom svijetu gdje je unutar simulacije naučen, između ostalog, hodati uspravno. Ta su znanja prenesena iz virtualnog u stvarni svijet, a simulacija je omogućila da robot nauči hodati u kraćem vremenu i bez uništavanja hardvera. Prvo je Cassie hodala poput bebe i baš kao dijete nastavila učiti. Naučila je kako ostati uspravno pri spoticanju ili kako se oporaviti od podražaja guranja. Također je naučila kako kompenzirati kada su joj dva motora bila oštećena.

U ovom radu demonstrirat ćemo kako naučiti simuliranog dvonožnog robota hodati implementacijom dva učinkovita algoritma učenja podrškom.

Sadržaj

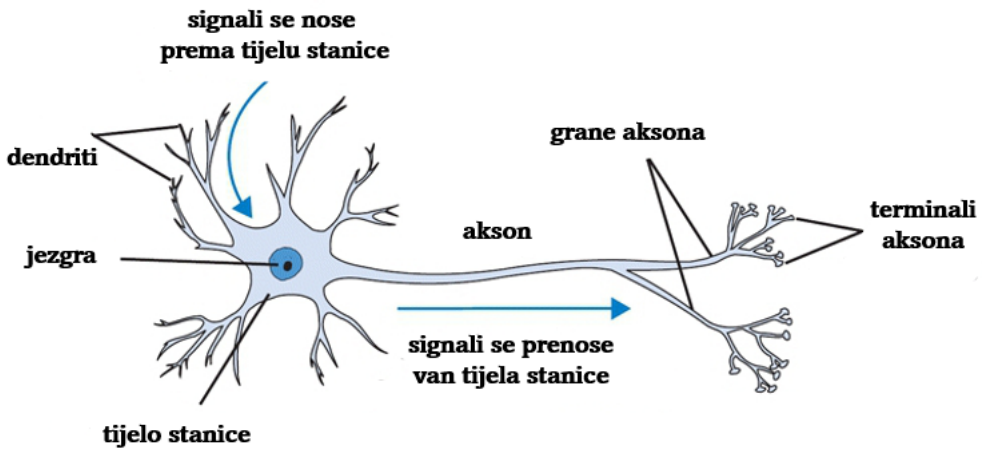
Poglavlje 1

Neuronske mreže

1.1 Biološka motivacija i osnovni model

Namijenjena oponašanju neuronskih mreža ljudskog mozga, struktura umjetnih neuronskih mreža slična je onoj bioloških neuronskih mreža. Ljudski mozak sadrži mrežu milijardi gusto povezanih neurona. Ona je iznimno kompleksna, nelinearna i sadrži trilijune veza između neurona. Sastoji se od dendrita, aksona, tijela stanice, sinapsi i jezgre (Slika 1.1). *Dendriti* su odgovorni za primanje ulaza od ostalih neurona, a *aksoni* za prijenos s jednog neurona na drugi. Molekularna i biološka mašinerija neuronskih mreža temelji se na elektrokemijskoj signalizaciji. Dendriti prenose električni signal stanici, zatim tijelo stanice računa nelinearnu funkciju ulaza i ispaljuje signal samo ako je primljen dobar ulaz. Akson tada prenosi izlaz funkcije ostalim neuronima. Dendrit drugog neurona prima ulaz i proces se nastavlja. Vezu između aksona neurona i dendrita drugog neurona nazivamo *sinapsa*. Ona regulira kemijsku vezu čija težina ima utjecaj na ulaz stanice. Svaki neuron ima više dendrita čime prima ulaz od više susjednih neurona.

1.1. Biološka motivacija i osnovni model



Slika 1.1: Biološka neuronska mreža

Navedeni biološki mehanizam simuliran je u *umjetnim* neuronskim mrežama koje sadrže jedinice računanja koje također nazivamo neuroni. Računalne jedinice su međusobno spojene težinama koje igraju istu ulogu kao jačina sinaptičkih veza u biološkim organizmima. Točnije, svaki ulaz ima težinu w koja je pridružena obzirom na njenu relativnu važnost u odnosu na ostale ulaze. Neuron primjenjuje tzv. *aktivacijsku funkciju* na težinsku sumu njegovih ulaza. Ideja je da se sinaptičke jačine (težine w) mogu naučiti i da kontroliraju snagu utjecaja i njegovog smjera: ekscitoran (pozitivna težina) ili inhibitoran (negativna težina) jednog neurona na drugi. U osnovnom modelu dendriti nose signal tijelu stanice gdje se sve sumira. Ako konačna suma prelazi određeni prag, neuron može 'ispaliti' signal. Tu stopu pucanja neurona modeliramo aktivacijskom funkcijom. Jednostavni matematički model koji radi na navedenim principima nazivamo *jednostavni g-perceptron*. Navedimo formalnu definiciju.

Definicija 1.1 ***Jednostavni g-perceptron*** *je uređena četvorka* $(X, Y, g, s_{(w_1,\cdots,w_n,\theta)})$, *gdje je* $X \subset \mathbb{R}^n$, *za neki* $n \in \mathbb{N}$, $Y \subset \mathbb{R}$, $s_{(w_1,\cdots,w_n,\theta)}$ *funkcija koju zovemo* ***integralnom funkcijom***

$$s_{(w_1,\cdots,w_n,\theta)} : X \to \mathbb{R}$$

dana pravilom

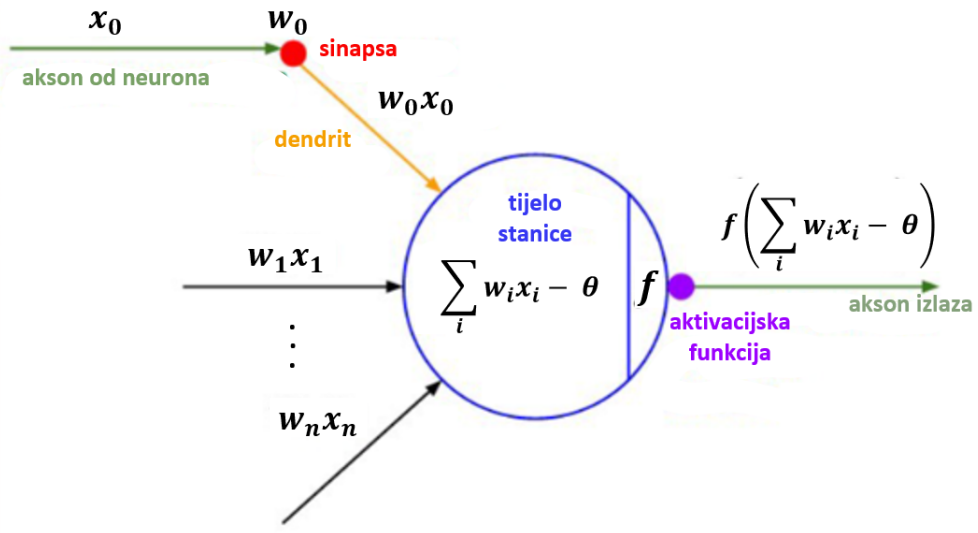
$$s_{(w_1,\cdots,w_n,\theta)}(x) = s_{(w_1,\cdots,w_n,\theta)}(x_1,\cdots,x_n) := \sum_{j=1}^{n} w_j x_j - \theta,$$

gdje su $w_1,\cdots,w_n,\theta \in \mathbb{R}$ *parametri te* g *proizvoljna funkcija*

$$g : \mathbb{R} \to Y$$

koju nazivamo ***aktivacijska funkcija****.*

Parametar θ je računarski ekvivalent praga podražljivosti u živčanoj stanici.



Slika 1.2: Perceptron i njegova usporedba s biološkim modelom

1.2 Aktivacijske funkcije

Izbor aktivacijske funkcije bitan je dio izgradnje neuronske mreže. U slučaju perceptrona, možemo koristiti signum funkciju za predviđanje binarne oznake klase. Tada bi izlaz aktivacijske funkcije, tj. predikcija bila

$$\hat{y} = \text{sign}\{\overline{W} \cdot \overline{X}\} = \text{sign}\{\sum_{j=1}^{n} w_j d_j\}$$

S druge strane, ako želimo predvidjeti realnu vrijednost, ima smisla koristiti identitetu kao aktivacijsku funkciju u završnom neuronu. Primjetimo da jednadžba $\overline{W} \cdot \overline{X} = 0$ definira linearnu hiperravninu i ona dijeli prostor u dvije klase, što znači da ovakav tip modela ima dobre performanse kada su podaci linearno separabilni. To uvodi motivaciju za nelinearnim aktivacijskim funkcijama.

Sigmoid aktivacijska funkcija dana s:

$$\Phi(x) = \frac{1}{1 + e^{-x}}$$

daje vrijednosti u intervalu $\langle 0, 1\rangle$, što je zgodno ako izlaz želimo interpretirati kao vjerojatnost.

Tanh aktivacijska funkcija ima oblik sličan sigmoid funkciji, samo što je horizontalno skalirana i vertikalno translatirana u $[-1, 1]$, tj. dana je s:

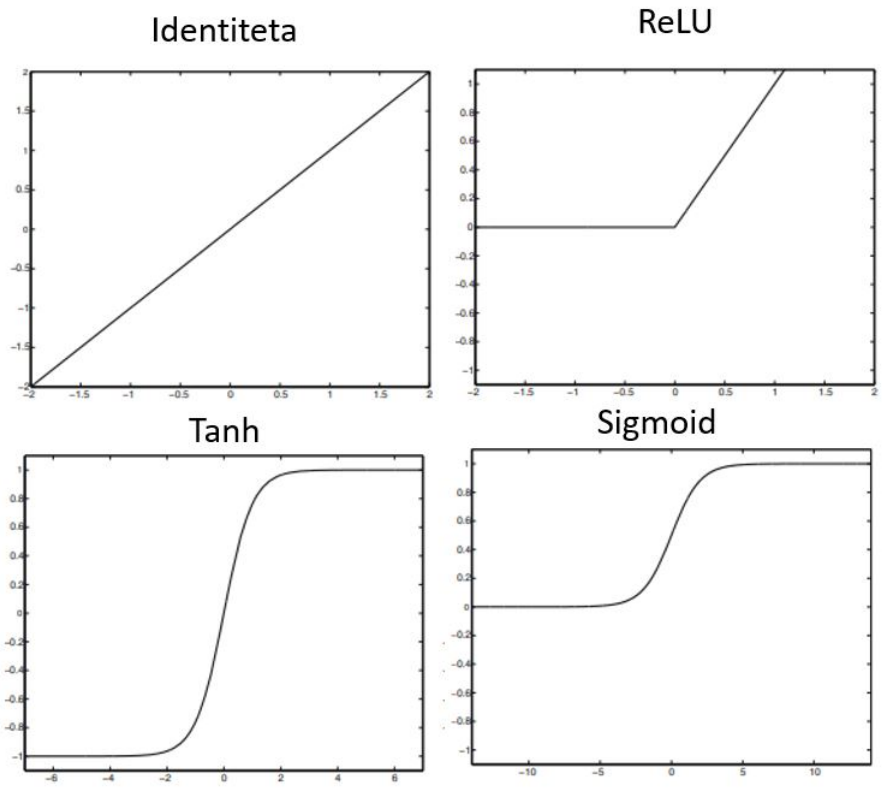
$$\Phi(x) = \frac{e^{2x} - 1}{e^{2x} + 1}.$$

Tanh preferiramo nad sigmoid funkcijom kada želimo da je izlaz i pozitivan i negativan. Osim toga, sigmoid funkcija ima veći gradijent pa je lakše trenirati (više o tome kasnije).

Sigmoid i tanh funkcija bile su prvi izbor za uvođenje nelinearnosti u neuronske mreže, no u novije vrijeme često se koristi **ReLU** aktivacijska funkcija. Ona je dana s

$$\Phi(x) = \max\{x, 0\}$$

i koristi se u višeslojnim neuronskim mrežama zbog jednostavnijeg treniranja i veće brzine konvergencije.



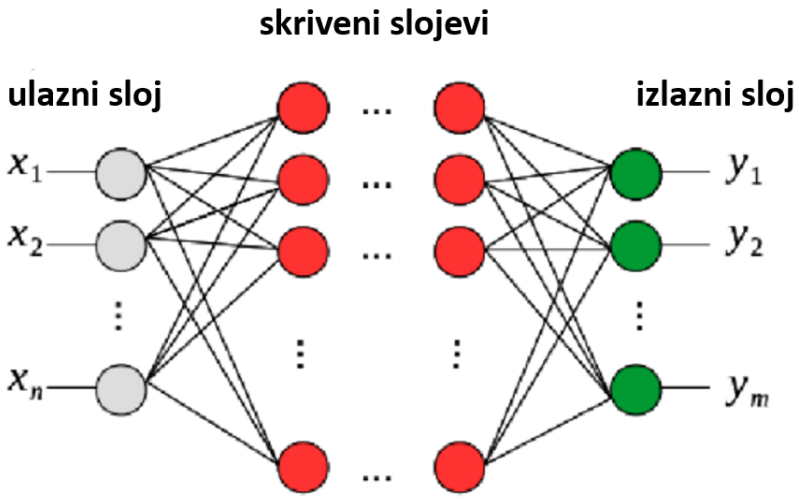
Slika 1.3: Aktivacijske funkcije

Primjetimo da su sve ovdje prikazane aktivacijske funkcije monotone. Nadalje, osim identitete i ReLU na pozitivnoj strani x osi, aktivacijske funkcije su zasićene pri visokim apsolutnim vrijednostima, tj. povećanje argumenta ne mijenja puno vrijednost funkcije. Ovakve nelinearne aktivacijske funkcije jako su korisne kod višeslojnih mreža jer pomažu stvoriti snažne kompozicije različitih tipova funkcija. Također, primijetimo da ovakve funkcije sažimaju ulaz u ograničeni interval. Korištenje nelinearnih aktivacijskih funkcija ima važnu ulogu u povećanju snage modeliranja mreže. Ako mreža koristi samo

linearne aktivacije, neće pružati veću snagu modeliranja od jednoslojne linearne mreže ([2], poglavlje 1.5).

1.3 Višeslojne neuronske mreže

Perceptron ima dva sloja, ulazni sloj i izlazni čvor. No, kako je samo drugi sloj računski, kažemo da je perceptron jednoslojna mreža. Višeslojne neuronske mreže imaju više računskih slojeva. Dodatne međuslojeve (između ulaznog i izlaznog) nazivamo *skriveni slojevi* jer račun koji se izvršava nije vidljiv korisniku. Za specifičnu arhitekturu višeslojnih neuronskih mreža kažemo da je *feed-forward* mreža jer se uzastopni slojevi međusobno "hrane" u smjeru prema naprijed, tj. iz ulaza u izlaz. Zadana arhitektura *feed forward* neuronskih mreža ima pretpostavku da su svi čvorovi jednog sloja povezani sa svim čvorovima sljedećeg sloja.



Slika 1.4: Višeslojna neuronska mreža

Ako neuronska mreža sadrži $p_i, i = 1, \cdots, k$ jedinica (neurona) u i-tom

sloju, tada vektor stupci reprezentacije izlaza, u oznaci $\bar{h_i}$ imaju dimenzionalnost p_i. Dakle, broj jedinica sloja nazivamo dimenzionalnost tog sloja. Težine veza između ulaznog sloja i prvog skrivenog sloja sadržane su u matrici W_1 dimenzija $d \times p_1$, a težine između i-tog skrivenog sloja i $(i+1)$-vog skrivenog sloja su elementi matrice W_i dimenzija $p_i \times p_{i+1}$. Ako izlazni skoj sadrži o jedinica, tada je zadnja matrica W_{k+1} dimenzija $p_k \times o$. d-dimenzionalni ulazni vektor $\bar{x}$ transformira se u izlaz koristeći sljedeće rekurzivne jednadžbe:

$$\bar{h}_1 = \Phi(W_1^T \bar{x}) \qquad \text{[Ulazni sloj prema skrivenom sloju]}$$
$$\bar{h}_{i+1} = \Phi(W_{i+1}^T \bar{h}_i), \forall i \in \{1, \cdots, k-1\} \qquad \text{[Skriveni sloj prema skrivenom sloju]}$$
$$\bar{h}_o = \Phi(W_{k+1}^T \bar{h}_k) \qquad \text{[Skriveni sloj prema izlaznom sloju]}$$

Aktivacijske funkcije Φ primjenjuju se na svaku koordinatu ulaznog vektora.

1.4 Proces učenja

Iz prethodnog znamo da svaki neuron prima ulazne vrijednosti pomnožene s težinama koje predstavljaju snagu te veze. No, da bismo ažurirali težine u pravom smjeru i počeli proces učenja, trebamo prvo opisati koliko je izlaz mreže "dobar" u odnosu na problem koji rješavamo. U tu svrhu uvodimo **funkciju gubitka** E čiji su argumenti upravo sinaptičke težine i koja formalno izražava to svojstvo. Odabir funkcije gubitka ovisi o prirodi problema. Primjerice, za problem predviđanja izlaza za dani ulaz, funkcija gubitka daje razliku između željenog i dobivenog izlaza. Intuitivno, želimo promijeniti težine mreže na način da smanjimo tu razliku što je više moguće.
Time proces učenja postaje optimizacijski problem, tj. želimo odrediti globalni minimum funkcije gubitka $E\colon \mathbb{R}^n \to \mathbb{R}$. U praksi se najčešće, zbog

vremenske i memorijske povoljnosti, koriste metode iterativnog tipa. Osnovna ideja tih metoda je generiranje niza točaka $(x_k)_{k\in\mathbb{N}\cup\{0\}}$ tako da vrijedi:

$$(x_{k+1} = x_k + \alpha_k d_k), \text{za svaki } k \in \mathbb{N}\cup\{0\},$$

gdje je $x_0 \in \mathbb{R}^n$ proizvoljna točka, $d_k \in \mathbb{R}^n$ vektor koji određuje smjer kretanja iz točke x_k, a $\alpha_k > 0$ realni parametar koji određuuje duljinu koraka iz točke x_k u smjeru vektora d_k. Parametre niza $(x_k)_{k\in\mathbb{N}\cup\{0\}}$ biramo na način da je niz $(E(x_k))_{k\in\mathbb{N}\cup\{0\}}$ monotono padajući te da konvergira nekom od lokalnih minimuma funkcije E, po mogućnosti baš globalnom minimumu.
Može se pokazati da je $\nabla E(x)$ smjer najvećeg rasta funkcije E u točki x pa je $-\nabla E(x)$ smjer najvećeg pada funkcije f u točki x, tj. minimum funkcije ćemo tražiti u smjeru negativnog gradijenta i tu iterativnu metodu zvati **metoda gradijentnog spusta**. Analogno, za maksimizacijski problem tražit ćemo maksimum funkcije u smjeru pozitivnog gradijenta i tu metodu zvati **metoda gradijentog uspona**.
Dakle, imamo:

$$W_{k+1} \leftarrow W_k \pm \alpha\nabla E(W_k),$$

gdje je W_k matrica težina u k-toj iteraciji učenja, a hiperparametar α obično uzimamo iz intervala $\langle 0, 1\rangle$ i još ga nazivamo **stopa učenja** (engl. *learning rate*).
Kod jednoslojnih neuronskih mreža, proces treniranja je relativno jednostavan jer se funkcija gubitka može izračunati kao direktna funkcija težina, što omogućava lagan izračun gradijenta. No, u slučaju višeslojnih neuronskih mreža, problem je što je gubitak komplicirana kompozicija funkcija težina iz prethodnih slojeva. Gradijent kompozicija funkcija računamo tzv. **algoritmom s povratnim proslijeđivanjem** (engl. *Backpropagation algorithm*) koji koristi pravilo derivacije kompozicije za izračun svih lokalnih gradijenta gubitka.

Poglavlje 2

Učenje podrškom

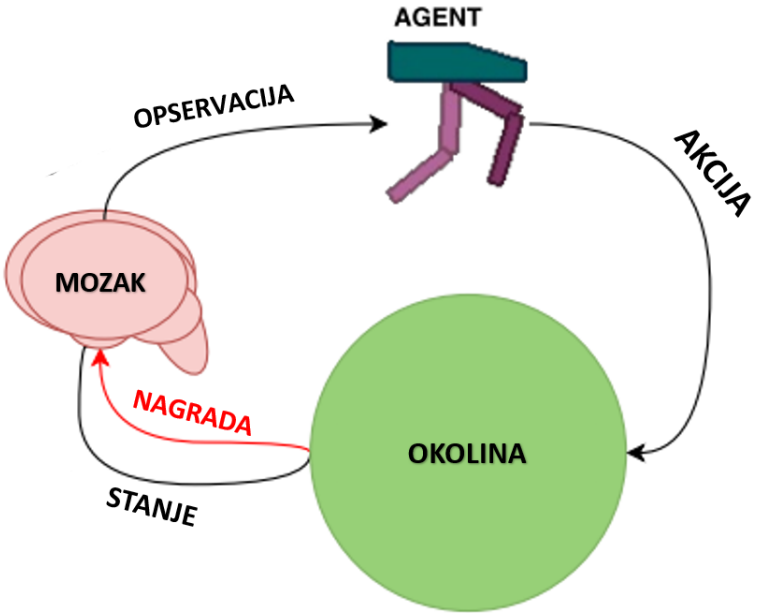
2.1 Uvod u problem

Problem učenja podrškom opisan je međusobnom interakcijom agenta i njegove okoline kroz stanja, akcije i signale nagrade. Okolina je sustav, u našem slučaju dvodimenzionalni svijet, kojeg agent percipira i u kojem reagira.

Okolina i njen agent zajedno se kreću kroz vrijeme u pojedinačnim jedinicama vremena. U svakom vremenskom koraku okolina E daje agentu informacije o njegovom stanju s u vremenskom koraku t pomoću opservacije o i agent izvršava akciju a u okolini te dobija nagradu r. Ciklus se zatim ponavlja (Slika 2.1).

Stanje s je potpun opis stanja svijeta, tj. ne postoji informacija o svijetu koja mu je nedostupna. **Opservacija** o je parcijalni opis stanja koji može izostaviti informacije. U dubokom učenju podrškom gotovo uvijek stanja i opservacije predstavljamo vektorom, matricom ili tenzorom realnih brojeva. Primjerice, vizualna opservacija može biti prikazana RGB matricom vrijednosti piksela ili, kao u našem slučaju, stanje robota može biti prikazano kutovima i brzinama zglobova. Agent može imati parcijalnu ili potpunu op-

servaciju okoline.



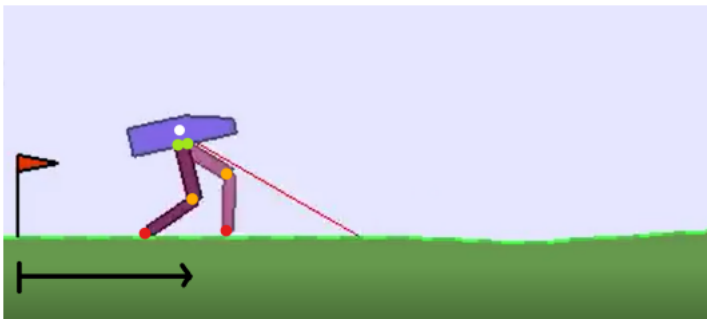
Slika 2.1: Dijagram koji opisuje interakciju između agenta i njegove okoline za problem učenja podrškom

Različite okoline dopuštaju različite akcije. Skup svih valjanih akcija u danoj okolini nazivamo **akcijski prostor**. Neke okoline, poput Atari igara, imaju **diskretni akcijski prostor** u kojem je samo konačan broj koraka dostupan agentu. Druge okoline, poput one u kojoj agent kontrolira robota u fizičkom svijetu, imaju **neprekidni akcijski prostor**. U neprekidnim prostorima akcije su vektori realnih brojeva. Navedena podjela igra veliku ulogu za metode u dubokom učenju podrškom. Neke familije algoritama mogu se direktno primijeniti u jednom slučaju, ali bi se trebale bitno modificirati za primjenu u drugom slučaju.

2.2 Opis radnog okruženja

OpenAI Gym je "open source framework" i nudi razna gotova okruženja za razvoj algoritama učenja podrškom. Mi ćemo raditi na klasičnom *BipedalWalker* okruženju gdje je teren relativno gladak. Robot ima determinističku dinamiku, ali teren je nasumično generiran na početku svake epizode. Osim toga, okolina ima neprekidni prostor akcija i opservacija.

Robot ima dvije noge, a svaka noga ima dva zgloba. Trebamo ga naučiti hodati primjenom rotacijske sile na navedene zglobove. Stanje robota je vektor duljine 24 (Slika 2.2 i Slika 2.3) : četiri parametra centra mase trupa (bijelo), kut i brzina svakog zgloba (dva zelena i dva narančasta), kontakt sa tlom (crveno) i 10 LiDAR skenova tla (crvena linija). Prostor akcija je dakle vektor duljine četiri koji predstavlja moment sile motora na svakom od četiri zgloba (Slika 2.4). LiDAR skenovi pomažu detektirati potencijalnu neravninu tla i oni su jedina informacija o terenu.



Slika 2.2: Robot u *BipedalWalker* okruženju. Crvena zastava označava početnu poziciju

2.2. Opis radnog okruženja

Num	Observation	Min	Max	Mean
0	hull_angle	0	2*pi	0.5
1	hull_angularVelocity	-inf	+inf	-
2	vel_x	-1	+1	-
3	vel_y	-1	+1	-
4	hip_joint_1_angle	-inf	+inf	-
5	hip_joint_1_speed	-inf	+inf	-
6	knee_joint_1_angle	-inf	+inf	-
7	knee_joint_1_speed	-inf	+inf	-
8	leg_1_ground_contact_flag	0	1	-
9	hip_joint_2_angle	-inf	+inf	-
10	hip_joint_2_speed	-inf	+inf	-
11	knee_joint_2_angle	-inf	+inf	-
12	knee_joint_2_speed	-inf	+inf	-
13	leg_2_ground_contact_flag	0	1	-
14-23	10 lidar readings	-inf	+inf	-

Slika 2.3: Prostor opservacija s 24 parametra

Num	Name	Min	Max
0	Hip_1 (Torque / Velocity)	-1	+1
1	Knee_1 (Torque / Velocity)	-1	+1
2	Hip_2 (Torque / Velocity)	-1	+1
3	Knee_2 (Torque / Velocity)	-1	+1

Slika 2.4: Prostor akcija s 4 parametra

Zadatak je da robot dođe do cilja u ograničenom vremenu. Za svaki korak koji robot napravi okolina vraća odgovarajuću nagradu i sljedeće stanje. Malo pomicanje naprijed (s lijeva na desno) rezultira malom nagradom, dok primjena zakretnog momenta na zglobove rezultira malom negativnom nagradom. Svrha negativne nagrade u tom slučaju je da robot nauči što glađe hodati s minimalnim okretnim momentom. Ako robot brzo stigne do cilja

i minimizira ukupni zakretni moment zglobova, dobit će rezultat od barem +300. Ako u bilo kojem trenutku padne, dobiva negativnu nagradu -100 i epizoda završava. Epizodu definiramo kao niz uzastopnih stanja, akcija i nagrada koji završava u terminalnom stanju ili ako je agent prešao maksimalan broj koraka.

2.3 Strategija

Strategija (engl. *policy*) je pravilo pomoću kojeg agent odlučuje koju akciju poduzeti. Može biti **deterministička** i u tom je slučaju označavamo s μ. Dakle, postoji točno jedna akcija za svako dano stanje. Drugim riječima deterministička strategija je funkcija

$$\mu : S \to A$$

definirana s

$$a_t = \mu(s_t), a_t \in A, s_t \in S,$$

gdje je A skup akcija, a S skup stanja. Strategija može biti i **slučajna** i tada je označavamo s π. Ona za dano stanje vraća vjerojatnosnu distribuciju akcija iz akcijskog prostora, tj.

$$a_t \sim \pi(\cdot \mid s_t), a_t \in A, s_t \in S$$

U dubokom učenju podrškom radimo s **parametriziranim strategijama**, tj. sa strategijama čiji su izlazi izračunjive funkcije koje ovise o skupu parametara (npr. težine neuronske mreže) koje možemo prilagoditi promjeni ponašanja koristeći neki optimizacijski algoritam. Najčešće parametre takve strategije označavamo s θ ili ϕ i pišemo kao potpisni znak uz simbol strategije da bismo naznačili vezu:

$$a_t = \mu_\theta(s_t)$$

$$a_t \sim \pi_\theta(\cdot \mid s_t).$$

2.4 Putanja i funkcija nagrade

Putanja τ je niz stanja i akcija, tj.

$$\tau = (s_0, a_0, s_1, a_1, ...).$$

Prvo stanje s_0 je nasumično uzorkovano iz distribucije start-stanja koju često označavamo s ρ_0:

$$s_0 \sim \rho_0(\cdot).$$

Prijelazima stanja (što se događa svijetu između stanja u vremenu t,s_t, i stanja u vremenu $t+1$,s_{t+1}) upravljaju prirodni zakoni okoline i ovise samo o zadnjoj akciji a_t. Mogu biti deterministički:

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

ili slučajni:

$$s_{t+1} \sim P(\cdot \mid s_t, a_t).$$

Funkcija nagrade R ključan je dio u učenju podrškom. Ona ovisi o trenutnom stanju, upravo poduzetoj akciji i sljedećem stanju:

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

iako je često pojednostavljeno na ovisnost samo o trenutnom stanju $r_t = R(s_t)$ ili paru stanja i akcije $r_t = R(s_t, a_t)$. Cilj agenta je maksimizirati kumulativnu nagradu kroz putanju, što može značiti više stvari. Jedna vrsta povrata je **konačni nediskontirani povrat**, tj. zbroj nagrada r_t u konačnom broju koraka $T \in \mathbb{N}_0$ putanje τ:

$$R(\tau) = \sum_{t=0}^{T} r_t.$$

Druga vrsta povrata je **beskonačni diskontirani povrat**, tj. zbroj svih nagrada koje je ikad stekao agent, ali koje su diskontirane obzirom na to koliko daleko u budućnosti su stečene:

$$R(\tau) = \sum_{t=0}^{\infty} \gamma^t r_t,$$

gdje je $\gamma \in [0, 1]$ **faktor diskontiranja**.

Zašto nam treba faktor diskontiranja? Zar ne želimo dobiti sve nagrade? Želimo, ali faktor diskontiranja je ujedno intuitivno privlačan i matematički zgodan. Na intuitivnoj razini: novac sada je bolji nego novac kasnije. Matematički, beskonačna suma nagrada ne mora konvergirati u konačnu vrijednost, ali uz faktor diskontiranja i pod razumnim uvjetima, beskonačna suma konvergira.

2.5 Markovljev proces odlučivanja

Okolina i agentove interakcije modelirane su Markovljevim procesom odlučivanja koji se temelji na činjenici da sustav zadovoljava **Markovljevo svojstvo**: prijelazi ovise samo o zadnjem stanju i akciji, a ne i o prošlosti. Jednostavije, možemo reći i "Budućnost je neovisna o prošlosti uz danu sadašnjost". Pogledajmo formalnu definiciju.

Definicija 2.1 ***Markovljev proces odlučivanja*** *je uređena petorka* (S, A, R, P, ρ_0) *gdje je:*

- S *skup stanja*
- A *skup akcija*
- $R\colon S \times A \times S \to \mathbb{R}$ *funkcija nagrade, gdje je* $r_t = R(s_t, a_t, s_{t+1})$

- $P : S \times A \to \mathcal{P}(S)$ *vjerojatnosna funkcija prijelaza, gdje je* $P(s'|s,a) = P(s_{t+1} = s' \mid s_t = s, a_t = a)$ *vjerojatnost prijelaza u stanje* s' *ako smo počeli u stanju* s *i napravili akciju* a

- ρ_0 *je distribucija početnog stanja.*

2.6 Problem učenja podrškom

Koji god izbor mjerila povrata uzeli i koju god strategiju izabrali, cilj učenja podrškom je odabrati strategiju koja maksimizira **očekivani povrat** kada agent djeluje sukladno njoj. Da bismo pričali o očekivanom povratu, prvo trebamo spomeniti vjerojatnosne distribucije putanja.
Pretpostavimo da su i prijelazi okoline i strategija slučajni. Tada je vjerojatnost putanje τ od T koraka pod strategijom π

$$P(\tau \mid \pi) = \rho_0(s_0) \prod_{t=0}^{T-1} P(s_{t+1} \mid s_t, a_t)\pi(a_t \mid s_t).$$

Očekivani povrat (za koju god metriku), označen s $J(\pi)$ je tada:

$$J(\pi) = \int_{\tau} P(\tau \mid \pi)R(\tau) = E_{\tau\sim\pi}[R(\tau)].$$

Centralni optimizacijski problem učenja podrškom tada možemo izraziti s:

$$\pi^* = \arg\max_{\pi} J(\pi),$$

gdje je π^* **optimalna strategija**.

2.7 Funkcije vrijednosti

Često je korisno znati *vrijednost* stanja ili para stanja i akcije. Pod vrijednost mislimo na očekivani povrat ako krenemo u tom stanju ili paru stanja i

akcije i zatim uvijek djelujemo u skladu s određenom strategijom. **Funkcije vrijednosti** koriste se, na jedan ili drugi način, skoro u svakom algoritmu učenja podrškom. Spomenimo četiri glavne funkcije.

1. **On-policy funkcija vrijednosti**, $V^\pi(s)$, koja daje očekivani povrat ako počnemo u stanju s i uvijek djelujemo u skladu sa strategijom π:

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

2. **On-policy funkcija akcije i vrijednosti**, $Q^\pi(s, a)$, koja daje očekivani povrat ako počnemo u stanju s, napravimo proizvoljnu akciju a (koja ne mora biti iz strategije) i onda uvijek djelujemo prema strategiji π:

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

3. **Funkcija optimalne vrijednosti**, $V^*(s)$, koja daje očekivani povrat ako počnemo u stanju s i uvijek djelujemo u skladu s *optimalnom* strategijom u okolini:

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

4. **Funkcija optimalne akcije i vrijednosti**, $Q^*(s, a)$, koja daje očekivani povrat ako počnemo u stanju s, napravimo proizvoljnu akciju a i onda uvijek djelujemo u skladu s *optimalnom* strategijom u okolini:

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

2.7.1 Optimalna Q-funkcija i optimalna akcija

Postoji važna veza između funkcije optimalne akcije i vrijednosti ($Q^*(s, a)$) i akcije koju odabire optimalna strategija. Po definiciji $Q^*(s, a)$ daje očekivani

povrat ako počnemo u stanju s, napravimo (proizvoljnu) akciju a i zatim uvijek djelujemo prema optimalnoj strategiji. Optimalna strategija će za s izabrati akciju koja maksimizira povrat pri startu u s. Zato, ako znamo Q^*, možemo direktno dobiti optimalnu akciju $a^*(s)$ iz:

$$a^*(s) = \arg\max_a Q^*(s, a).$$

Napomena 2.2 *Može biti više akcija koje maksimiziraju $Q^*(s, a)$. U tom slučaju sve su one optimalne i optimalna strategija može nasumično odabrati neku od njih. Međutim,* ***uvijek*** *postoji optimalna strategija koja deterministički bira akciju.*

2.7.2 Bellmanove jednadžbe

Sve četiri funkcije vrijednosti zadovoljavaju jednadžbe samodosljednosti koje nazivamo **Bellmanove jednadžbe**. Osnovna ideja Bellmanovih jednadžbi je sljedeća:

> *Vrijednost početne točke je nagrada koju očekujemo dobiti od činjenice da smo na toj poziciji plus vrijednost onoga gdje god bili u sljedećem koraku.*

Bellmanove jednadžbe on-policy funkcija vrijednosti su:

$$V^\pi(s) = \mathop{\mathbb{E}}_{\substack{a\sim\pi \\ s'\sim P}} \big[r(s, a) + \gamma V^\pi(s')\big],$$

$$Q^\pi(s, a) = \mathop{\mathbb{E}}_{s'\sim P}\Big[r(s, a) + \gamma \mathop{\mathbb{E}}_{a'\sim\pi} \big[Q^\pi(s', a')\big]\Big],$$

gdje je $\gamma \in [0, 1]$ faktor diskontiranja, $s' \sim P$ kratica za $s' \sim P(\cdot \mid s, a)$, što ukazuje na to da je sljedeće stanje s' uzorkovano iz pravila prijelaza okoline; $a \sim \pi$ je kratica za $a \sim \pi(\cdot \mid s)$ i $a' \sim \pi$ je kratica za $a' \sim \pi(\cdot \mid s')$.
Bellmanove jednadžbe za funkcije optimalne vrijednosti su:

$$V^*(s) = \max_a \mathop{\mathbb{E}}_{s'\sim P} \big[r(s, a) + \gamma V^*(s')\big],$$

$$Q^*(s,a) = \mathop{\mathbb{E}}_{s' \sim P}\Big[r(s,a) + \gamma \max_{a'} Q^*(s',a')\Big],$$

Ključna razlika između Bellmanovih jednadžbi za on-policy funkcije vrijednosti i funkcije optimalne vrijednosti je izostanak ili prisutnost funkcije max nad akcijama. Njena važnost leži u tome što kad god agent bira akciju, da bi djelovao optimalno, mora odabrati akciju koja vodi najvećoj vrijednosti.

2.8 Vrste algoritama učenja podrškom

Jedna od najvažnijih točaka grananja u algoritmu učenja podrškom je pitanje *ima li agent pristup modelu okoline ili ga mora naučiti.* Pod model okoline mislimo na funkciju koja predviđa prijelaze stanja i nagrade.

Glavna prednost postojanja modela je što agent tada može *planirati* razmišljanjem unaprijed jer on u tom slučaju može vidjeti što će se dogoditi za niz različitih odluka i odabrati najbolju opciju. Agent tada može prenijeti razultate planiranja unaprijed u naučenu strategiju. Kada ovaj pristup radi, može rezultirati znatnim napretkom u efikasnosti uzorkovanja u usporedbi s metodama bez modela. S druge strane, glavni nedostatak leži u činjenici da *model temeljne istine okoline često nije dostupan agentu.* Ako agent želi koristiti model u tom slučaju ga mora naučiti potpuno iz iskustva, što stvara mnoge izazove. Najveći izazov je što agent može iskoristiti pristranost modela, zbog čega će dati dobre rezultate obzirom na naučeni model, ali se ponaša suboptimalno (ili jako loše) u stvarnoj okolini. Učenje modela je jako teško i čak izniman trud se ne mora isplatiti.

Za algoritme koji koriste model kažemo da su **modelom-utemeljeni**, a za one koji ga ne koriste kažemo da su **model-slobodni**. Iako model-slobodne metode nemaju potencijalnu prednost u efikasnosti uzorkovanja pri korištenju modela, one imaju tendenciju biti lakše za implemenaciju i

podešavanje. Mi ćemo se baviti upravo model-slobodnim algoritmima.

Postoje dva glavna pristupa reprezentaciji i treniranju agenata model-slobodnim učenjem podrškom.

1. **Optimizacija strategije**. Metode u ovoj familiji označavaju strategiju eksplicitno s $\pi_\theta(a \mid s)$. One optimiziraju parametre θ direktno gradijentnim usponom nad ciljem $J(\pi_\theta)$ ili indirektno maksimizacijom lokalnih aproksimacija od $J(\pi_\theta)$. Navedena optimizacija skoro se uvijek izvodi "on-policy", što znači da svako ažuriranje koristi samo podatke koji su prikupljeni djelovanjem prema najnedavnijoj verziji strategije. Optimizacija strategije također često uključuje učenje aproksimatora $V_\phi(s)$ on-policy funkcije vrijednosti $V^\pi(s)$, što se koristi u načinu na koji se strategija ažurira.

2. **Q-učenje**. Metode u ovoj familiji uče aproksimatora $Q_\theta(s, a)$ optimalne funkcije akcije i vrijednosti ($Q^*(s, a)$). One tipično koriste funkciju cilja temeljenu na Bellmanovoj jednadžbi. Navedena optimizacija skoro se uvijek radi "off-policy", što znači da svako ažuriranje može koristiti podatke prikupljene u bilo kojem trenutku tijekom treniranja, neovisno o tome na koji je način agent istraživao okolinu kada su podaci dobiveni. Pripadnu strategiju dobivamo pomoću veze između Q^* i π^*: akcije koje radi agent Q-učenja dane su sa:

$$a(s) = \arg\max_a Q_\theta(s, a).$$

Kompromis između Optimizacije strategije i Q-učenja

Primarna snaga metoda optimizacije strategije leži u tome što *direktno optimiziramo za onu stvar koju želimo*, što ih čini stabilnim i pouzdanim.

Za razliku od toga, metode Q-učenja samo *indirektno* optimiziramo za agentovo izvođenje, treniranjem Q_θ da zadovolji jednadžbu samodosljednosti. Za ovakav tip učenja postoji više mogućnosti neuspjeha, zbog čega je manje stabilan. Ipak, metode Q-učenja imaju prednost zbog toga što su znatno više uzorkovano efikasne kada rade jer mogu ponovno iskoristiti podatke učinkovitije od tehnika optimizacije strategije.

Interpolacija između Optimizacije strategije i Q-učenja.

Srećom, optimizacija strategije i Q-učenje nisu nekompatibilni, štoviše, pod nekim okolnostima su i ekvivalentni. Postoji niz algoritama koji žive između ova dva ekstrema. Algoritmi koji se nalaze na tom spektru imaju mogućnost opreznog balansiranja između snaga i slabosti obiju strana. Upravo takav algoritam prvi je od dva kojeg ćemo demonstrirati u ovom radu.

Poglavlje 3

Prvi algoritam

Glumac-Kritičar (engl. *Actor-Critic*) metode kombiniraju prednosti metoda temeljenih na učenju funkcije vrijednosti i onih temeljenih na učenju strategije. Točnije, imamo model *glumca* koji uči strategiju agenta i model *kritičara* koji govori glumcu kako se može poboljšati. U ovom radu baziramo se na glumac metodama koje koriste gradijent strategije za učenje modela strategije. Glumac uči od kritičara dok kritičar koristi glumčeve interakcije s okolinom da poboljša kvalitetu svojih evaluacija.

3.1 Deep Deterministic Policy Gradient (DDPG)

Deep Deterministic Policy Gradient ili **DDPG** je algoritam koji paralelno uči Q-funkciju i strategiju. Koristi off-policy podatke i Bellmanovu jednadžbu za učenje Q-funkcije, a Q-funkciju koristi za učenje strategije.

Pristup je usko povezan s Q-učenjem i ima jednaku motivaciju: ako znamo optimalnu funkciju akcije i vrijednosti $Q^*(s, a)$, onda za svako stanje s optimalnu akciju $a^*(s)$ možemo dobiti rješavanjem

$$a^*(s) = \arg\max_a Q^*(s, a).$$

DDPG ispripliće učenje aproksimatora od $Q^*(s,a)$ s učenjem aproksimatora od $a^*(s)$ i to radi na način posebno prilagođen okolinama s neprekidnim akcijskim prostorom. No, što to točno znači da je DDPG prilagođen *posebno* za okoline s neprekidnim akcijskim prostorima? To je povezano s načinom na koji računamo maksimum nad akcijama u $\max_a Q^*(s,a)$. Kada imamo konačan broj diskretnih akcija, maksimum ne predstavlja nikakav problem jer možemo jednostavno izračunati Q-vrijednosti odvojeno za svaku akciju i usporediti ih. No, kada je akcijski prostor neprekidan, ne možemo iscrpno ocijeniti prostor i rješavanje optimizacijskog problema je vrlo netrivijalno. Računanje $\max_a Q^*(s,a)$ bila bi jako "skupa" subrutina kad bi koristili klasičan algoritam optimizacije. Obzirom na to da bi je trebali pokrenuti svaki put kada agent želi napraviti akciju u okolini, to je neodrživo.

Kako je akcijski skup neprekidan, pretpostavljamo da je funkcija $Q^*(s,a)$ diferencijabilna obzirom na argument akcije. To nam dopušta da stvorimo efikasno pravilo učenja temeljeno na gradijentu za strategiju $\mu(s)$ koje iskorištava tu činjenicu. Tada, umjesto da pokrećemo skupu optimizacijsku subrutinu svaki put kada želimo izračunati $\max_a Q^*(s,a)$ možemo je aproksimirati s $\max_a Q(s,a) \approx Q(s,\mu(s))$.

3.1.1 Q-učenje kod DDPG-a

Prvo, sjetimo se Bellmanove jednadžbe koja opisuje optimalnu funkciju akcije i vrijednosti, $Q^*(s,a)$. Dana je s:

$$Q^*(s,a) = \mathop{\mathbb{E}}_{s' \sim P}\Big[r(s,a) + \gamma \max_{a'} Q^*(s',a')\Big],$$

gdje $s' \sim P$ znači da je sljedeće stanje s' uzorkovano u okolini iz distribucije $P(\cdot \mid s,a)$. Bellmanova jednadžba je početna točka učenja aproksimatora od $Q^*(s,a)$. Pretpostavimo da je aproksimator neuronska mreža $Q_\phi(s,a)$ (s

parametrima ϕ) i da imamo skup $\mathcal{D}$ tranzicija (s, a, r, s', d) (gdje d označava je li stanje s' završno). Postavimo **MSBE (mean-squared Bellman error)** funkciju koja nam približno govori koliko je funkcija Q_ϕ blizu toga da zadovolji Bellmanovu jednadžbu:

$$L(\theta, \mathcal{D}) = \underset{(s,a,r,s',d)\sim\mathcal{D}}{\mathbb{E}} \left[\left(Q_\phi(s,a) - \left(r + \gamma(1-d) \max_{a'} Q_\phi(s', a') \right) \right)^2 \right]$$

Ovdje, kod evaluacije $(1 - d)$ koristimo Pythonovu konvenciju evaluacije True u 1 i False u 0. Tada, ako je s' u završnom stanju, Q-funkcija treba pokazati da agent ne dobiva dodatne nagrade nakon trenutnog stanja. Ovaj izbor notacije odgovara kasnijoj implementaciji u kodu.
Algoritmi Q učenja za aproksimatore funkcija većinski se temelje na minimizaciji MSBE funkcije gubitka. Postoje dva glavna trika koje koriste svi takvi algoritmi:

1. **Međuspremnik za ponavljanje**. Svi standardni algoritmi za treniranje dubokih neuronskih mreža za aproksimaciju Q^* koriste međuspremnik za ponavljanje. To je skup $\mathcal{D}$ prethodnih iskustava. Da bi algoritam bio stabilan, međuspremnik mora biti dovoljno velik za držanje širokog raspona iskustava, ali možda nije uvijek dobra ideja sve zadržati. Ako samo koristimo najnedavnije podatke, možemo je pretrenirati na njima; ako koristimo previše iskustva, možemo usporiti učenje.

   Napomena 3.1 *Spomenuli smo da je DDPG off-policy algoritam. Objasnimo zašto i kako. Primjetimo da međuspremnik treba sadržavati stara iskustva iako su ona možda prikupljena koristeći staru strategiju. Zašto ih uopće možemo koristiti? Jer Bellmanovu jednadžbu "nije briga" koje uređene trojke tranzicija su korištene ili kako su te akcije izabrane ili što se događa nakon dane tranzicije jer optimalna Q funkcija mora zadovoljavati Bellmanovu jednadžbu za* ***sve*** *moguće tranzicije.*

2. **Ciljne mreže**. Algoritmi Q učenja koriste ciljne mreže. Član

$$r + \gamma(1-d)\max_{a'} Q_\phi(s', a')$$

nazivamo **cilj** jer kada minimiziramo MSBE gubitak pokušavamo da Q funkcija bude što sličnija njemu. No, cilj ovisi o istim parametrima koje pokušavamo trenirati (ϕ), što čini MSBE optimizaciju nestabilnom. Rješenje je koristiti skup parametara koji su blizu ϕ, ali uz vremensku odgodu, tj. drugu mrežu koju nazivamo **ciljna mreža** i koja zaostaje nakon prve. Parametri ciljne mreže označavamo s ϕ_{targ}.

Ciljna mreža se ažurira jedanput po ažuriranju glavne mreže *polyak* prosjekom:

$$\phi_{targ} \leftarrow \rho\phi_{targ} + (1-\rho)\phi,$$

gdje je ρ hiperparametar između 0 i 1 (obično je blizu 1).

Kao što smo spomenuli ranije, računanje maksimuma nad akcijama u cilju je izazov u neprekidnim akcijskim prostorima. DDPG se nosi s ovim problemom koristeći **ciljnu mrežu strategije** za izračun akcije koja približno maksimizira $Q_{\theta_{targ}}$. Ciljnu mrežu strategije dobivamo na isti način kao ciljnu Q funkciju : polyak prosjekom parametara strategije tijekom treniranja.

Dakle, Q učenje u DDPG-u izvodi se minimizacijom sljedećeg MSBE gubitka stohastičkim gradijentnim spustom:

$$L(\theta, \mathcal{D}) = \mathop{\mathbb{E}}_{(s,a,r,s',d)\sim\mathcal{D}}\left[\left(Q_\phi(s,a) - \left(r + \gamma(1-d)Q_{\phi_{targ}}(s', \mu_{\theta_{targ}}(s'))\right)\right)^2\right],$$

gdje je $\mu_{\theta_{targ}}$ ciljna strategija.

3.1.2 Učenje strategije kod DDPG-a

Učenje strategije u DDPG-u je prilično jednostavno. Želimo naučiti deterministička strategiju $\mu_\theta(s)$ koja daje akciju koja maksimizira $Q_\phi(s,a)$. Kako je

akcijski prostor neprekidan, pretpostavljamo da je Q funkcija diferencijabilna obzirom na akciju pa možemo napraviti gradijentni uspon (samo obzirom na parametre strategije) da bi riješili

$$\max_{\theta} \underset{s\sim\mathcal{D}}{\mathbb{E}} \left[Q_\phi(s, \mu_\theta(s))\right].$$

Primjetimo da parametre Q funkcije ovdje tretiramo kao konstante.

3.1.3 Istraživanje vs. eksploatacija

DDPG trenira deterministička strategiju na off-policy način. Zbog toga što je strategija deterministička, kad bi agent istraživao u skladu sa strategijom u početku vjerojatno ne bi probao dovoljnu raznolikost akcija za pronalazak korisnih signala učenja. Da bi DDPG strategije istraživale bolje dodajemo šum akcijama tijekom treniranja. Autori originalnog DDPG rada [7] preporučuju vremenski koreliran *OU šum*, no recentniji rezultati sugeriraju da Gaussov šum srednje vrijednosti nula radi sasvim dobro. Druga opcija je jednostavnija pa je zbog toga i preferirana.

3.1. Deep Deterministic Policy Gradient (DDPG)

Algoritam 1 DDPG

Ulaz: inicijalni parametri strategije θ i Q funkcije ϕ, prazan međuspremik $\mathcal{D}$

Neka su ciljni parametri jednaki glavnim parametrima $\theta_{targ} \leftarrow \theta$,$\phi_{targ} \leftarrow \phi$

repeat

  Uoči stanje s i odaberi akciju $a = clip(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, gdje je $\epsilon \sim \mathcal{N}$

  Izvrši akciju a u okolini

  Uoči sljedeće stanje s', nagradu r i signal kraja d za indikaciju je li s' krajnje stanje.

  Spremi (s, a, r, s', d) u međuspremnik $\mathcal{D}$

  Ako je s' krajnje stanje, resetiraj okolinu

  if je vrijeme za ažuriranje **then**

    for koliko god ažuriranja **do**

      Nasumično uzorkuj skup prijelaza $\mathcal{B} = (s, a, r, s', d)$ iz $\mathcal{D}$

      Izračunaj ciljeve

$$y(r, s', d) = r + \gamma(1 - d)Q_{\phi_{targ}}(s', \mu_{\theta_{targ}}(s'))$$

      Ažuriraj Q funkciju jednokoračnim gradijentnim spustom koristeći

$$\nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} (Q_\phi(s, a) - y(r, s', d))^2$$

      Ažuriraj strategiju jednokoračnim gradijentnim usponom koristeći

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_\phi(s, \mu_\theta(s))$$

      Ažuriraj ciljne mreže:

$$\phi_{targ} \leftarrow \rho\phi_{targ} + (1 - \rho)\phi,$$

$$\theta_{targ} \leftarrow \rho\theta_{targ} + (1 - \rho)\theta$$

    end for

  end if

until konvergencija

3.2 Twin Delayed Deep Deterministic Policy Gradients (TD3)

Iako DDPG može pružiti izvrsne rezultate, ima svojih mana. Poput mnogih algoritama učenja podrškom, treniranje DDPG-a može biti nestabilno i može puno ovisiti o pronalasku pravih hiperparametara za ponuđeni zadatak. Čest tip neuspjeha algoritma je kada naučena Q funkcija jako precijenjuje Q vrijednosti. Te greške procjene nagomilaju se kroz vrijeme i mogu dovesti do toga da agent zapne u lokalnom optimumu ili iskusi katastrofalan zaborav. **Twin Delayed Deep Deterministic Policy Gradients (TD3)** rješava ovaj problem fokusirajući se na reduciranje pristranosti precijenjivanja. To radi pomoću tri trika:

1. **Clipped Double-Q Learning**. TD3 uči *dvije* Q funkcije umjesto jedne (odakle slijedi "twin" u imenu) i koristi manju od dvije Q vrijednosti za formiranje ciljeva u Bellmanovim funkcijama gubitka i greške.

2. **"Odgođena" ažuriranja strategije**. TD3 ažurira strategiju (i ciljne mreže) rjeđe nego Q funkciju. Originalni rad [9] preporučuje jedno ažuriranje strategije za svaka dva ažuriranja Q funkcije.

3. **Zaglađivanje ciljne strategije**. TD3 dodaje šum ciljnoj akciji da bi strategiji bilo teže iskorištavati greške Q funkcije zaglađivanjem Q niz promjene akcije.

Navedena tri pristupa rezultiraju značajno poboljšanom uspješnosti u usporedbi s osnovnim DDPG-om. TD3 paralelno uči dvije Q funkcije, Q_{ϕ_1} i Q_{ϕ_2}, pomoću MSBE minimizacije, skoro na isti način kao što DDPG uči jednu Q funkciju. Da bismo pokazali kako to TD3 radi i kako se razlikuje od klasičnog DDPG-a, počet ćemo od unutarnjeg dijela funkcije gubitka prema van.

3.2.1 Zaglađivanje ciljne strategije

Zaglađivanje ciljne strategije služi kao regularizator algoritma. Ono uzima u obzir određeni način kvara koji se može dogoditi u DPPG-u: ako aproksimator Q funkcije razvije netočni oštri šiljak za neke akcije, strategija će brzo iskoristiti šiljak i tada imati krhko ili nepravilno ponašanje. To se može izbjeći zaglađivanjem Q funkcije nad sličnim akcijama. U idealnom slučaju neće biti varijance među ciljnim vrijednostima, uz to da sličnim akcijama pridružujemo slične vrijednosti. TD3 reducira varijancu dodavanjem "malo" slučajnog šuma cilju. Raspon šuma ograničavamo da osiguramo da je ciljna vrijednost što bliže originalnoj akciji. Nakon dodavanja ograničenog šuma, ciljna akcija se također ograničava tako da leži unutar intervala akcija (sve valjane akcije a zadovoljavaju $-1 = a_{Low} \leq a \leq a_{High} = 1$). Ciljne akcije su tada:

$$a'(s') = clip\Big(\mu_{\theta_{targ}}(s') + clip(\epsilon, -c, c), a_{Low}, a_{High}\Big), \epsilon \sim \mathcal{N}(0, \sigma).$$

Dodavanjem šuma procjeni vrijednosti strategije su stabilnije jer ciljna vrijednost vraća višu vrijednost za robusnije akcije.

3.2.2 Ograničeno dvostruko Q učenje

Obje Q funkcije koriste jedan cilj koji računamo na način da uzmemo onu Q funkciju koja daje manju ciljnu vrijednost ("manje od dva zla"):

$$y(r, s', d) = r + \gamma(1 - d) \min_{i=1,2} Q_{\phi_i, targ}(s', a'(s')),$$

i obje učimo regresijom sljedećem cilju:

$$L(\phi_1, \mathcal{D}) = \mathop{\mathbb{E}}_{(s,a,r,s',d)\sim\mathcal{D}} \left[\left(Q_{\phi_1}(s, a) - y(r, s', d) \right)^2 \right],$$

$$L(\phi_2, \mathcal{D}) = \mathop{\mathbb{E}}_{(s,a,r,s',d)\sim\mathcal{D}} \left[\left(Q_{\phi_2}(s, a) - y(r, s', d) \right)^2 \right].$$

Korištenjem manje Q vrijednosti za cilj i regresijom prema tome pomaže u izbjegavanju precijenjivanja u Q funkciji. Dakle, ova metoda favorizira podcijenjivanje Q vrijednosti. Pristranost podcijenjivanju nije problem jer niske vrijednosti se neće propagirati kroz algoritam za razliku od visokih vrijednosti. To osigurava stabilniju aproksimaciju, čime se poboljšava stabilnost cijelog algoritma.

3.2.3 Odgođena ažuriranja strategije

Ciljne mreže su izvrstan alat za uvođenje stabilnosti agentovom treniranju, međutim u slučaju glumac-kritičar metoda postoje problemi u navedenoj tehnici. Oni leže u interakciji između mreže strategije (glumac) i mreže vrijednosti (kritičar). Treniranje agenta divergira kada je loša strategija precijenjena. Strategija našeg agenta će se tada pogoršavati dok se ažurira na stanjima s puno grešaka.

Kako bismo to popravili, jednostavno moramo provoditi ažuriranja mreže strategije rjeđe nego vrijednosne mreže. U praksi se mreža strategije ažurira nakon određenog vremenskih koraka, dok se mreža vrijednosti nastavlja ažurirati nakon svakog vremenskog koraka. Ova rjeđa ažuriranja strategije imat će procjenu vrijednosti s manjom varijancom i stoga bi trebala rezultirati boljom strategijom.

Algoritam 2 TD3

Ulaz: inicijalni parametri strategije θ i Q funkcije ϕ_1, ϕ_2, prazan međuspremik $\mathcal{D}$
Neka su ciljni parametri jednaki glavnim parametrima $\theta_{targ} \leftarrow \theta$,$\phi_{targ,1} \leftarrow \phi_1$, $\phi_{targ,2} \leftarrow \phi_2$
repeat
  Uoči stanje s i odaberi akciju $a = clip(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, gdje je $\epsilon \sim \mathcal{N}$
  Izvrši akciju a u okolini
  Uoči sljedeće stanje s', nagradu r i signal kraja d za indikaciju je li s' krajnje stanje.
  Spremi (s, a, r, s', d) u međuspremnik $\mathcal{D}$
  Ako je s' krajnje stanje, resetiraj okolinu
  if je vrijeme za ažuriranje **then**
    for j u rasponu (koliko god ažuriranja) **do**
      Nasumično uzorkuj skup prijelaza $\mathcal{B} = (s, a, r, s', d)$ iz $\mathcal{D}$
      Izračunaj ciljne akcije

$$a'(s') = clip(\mu_{\theta_{targ}}(s') + clip(\epsilon, -c, c), a_{Low}, a_{High}), \epsilon \sim \mathcal{N}(0, \sigma)$$

      Izračunaj ciljeve

$$y(r, s', d) = r + \gamma(1 - d) \min_{i=1,2} Q_{\phi_{targ,i}}(s', a'(s'))$$

      Ažuriraj Q funkcije jednokoračnim gradijentnim spustom koristeći

$$\nabla_{\phi_i} \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} (Q_{\phi_i}(s, a) - y(r, s, d))^2, i = 1, 2$$

      if j mod odgoda_strategije $= 0$ **then**
        Ažuriraj strategiju jednokoračnim gradijentnim usponom koristeći

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_{\phi_1}(s, \mu_\theta(s))$$

        Ažuriraj ciljne mreže:

$$\phi_{targ,i} \leftarrow \rho\phi_{targ,i} + (1 - \rho)\phi_i, i = 1, 2$$

$$\theta_{targ} \leftarrow \rho\theta_{targ} + (1 - \rho)\theta$$

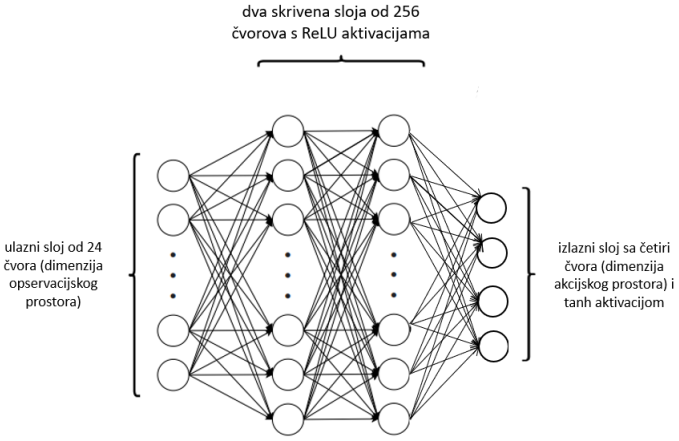
      end if
    end for
  end if
until konvergencija

3.3 Implementacija

Pogledajmo sada implementaciju TD3 algoritma na BipedalWalker problem preuzetu s GitHub repozitorija [12]. Kod je u programskom jeziku *Python* u kombinaciji s programskim okvirom za strojno učenje *PyTorch*. Neke dijelove koda izostavit ćemo i/ili spomenuti samo opisno radi fokusa na problematici samog algoritma.

3.3.1 Arhitektura kritičara i glumca

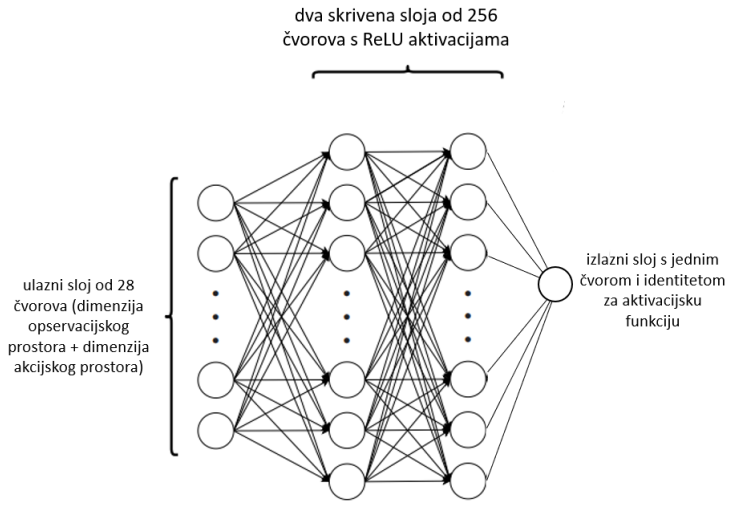
Trebaju nam dvije neuronske mreže, jedna koja nam služi za aproksimaciju strategije i nju nazivamo **glumac** (engl. *Actor*), a druga za aproksimaciju funkcije vrijednosti i nju nazivamo **kritičar** (engl. *Critic*). Za arhitekturu obje mreže biramo potpuno povezanu mrežu sa dva skrivena sloja s ReLU aktivacijskim funkcijama. Mreže se razlikuju u ulaznom i izlaznom sloju.



Slika 3.1: Arhitektura mreže glumac

Glumac za ulaz koji predstavlja stanje svijeta treba dati odabranu akciju. Dimenzija ulaznog sloja je dakle jednaka dimenziji prostora opservacija (24), a dimenzija izlaznog sloja jednaka je dimenziji akcijskog prostora (4). Za

aktivacijsku funkciju izlaznog sloja biramo tanh funkciju jer akcije poprimaju vrijednosti u $\langle -1, 1\rangle^4$.



Slika 3.2: Arhitektura mreže kritičar

S druge strane, kritičar za ulaz koji je uređeni par stanja s i akcije a daje vrijednost iz $\mathbb{R}$ koja predstavlja očekivani povrat. Zato je dimenzija ulaznog sloja zbroj dimenzija prostora akcija i opservacija (28), a dimenzija izlaznog sloja je 1. Koristimo linearnu aktivacijsku funkciju jer predviđamo realnu vrijednost.

Za optimizator uzimamo Adam, jedan od najčešće korištenih algoritama optimizacije zbog dobrih performansi, brzine i niskog trošenja memorije.

3.3.2 Klasa Agent

Klasa Agent sadrži sve potrebne metode i varijable za učenje. Konstruktor klase definira njene atribute i pridružuje im vrijednosti. Atribut `observationDim` (`actionDim`) sadrži dimenziju opservacijskog (akcijskog) prostora iz varijable okoline `Env`. Atribut `gamma` predstavlja faktor diskontiranja γ iz Bellmanove

jednadžbe, a atribut `tau` je hiperparametar iz polyak prosjeka kod ažuriranja ciljnih mreža.

```
class Agent():
    def __init__(
        self, env: Env, learningRate: float, gamma: float, tau: float,
        shouldLoad: bool=True, saveFolder: str='saved'
    ):
        self.observationDim = env.observation_space.shape[0]
        self.actionDim = env.action_space.shape[0]
        self.gamma = gamma
        self.tau = tau
```

Atribut `envName` sadrži mapu za spremanje modela, `device` je li koristimo CPU ili GPU, a `buffer` sadrži međuspreminik za ponavljanje kojeg ili učitavamo ako već postoji ili ga instaciramo iz klase `Buffer`. Definiciju klase `Buffer` nećemo detaljno opisivati, bitno je da znamo čemu služi.

```
        if not os.path.isdir(saveFolder):
            os.mkdir(saveFolder)
        self.envName = os.path.join(saveFolder, env.name + '.')
        name = self.envName
        self.device = T.device('cuda' if T.cuda.is_available() else 'cpu')
        self.buffer = pickle.load(open(name + 'Replay', 'rb'))\
            if shouldLoad and os.path.exists(name + 'Replay') else Buffer(
                self.observationDim, self.actionDim
            )
```

Inicijaliziramo glumca i kritičare koristeći klasu `Network` ili učitavanjem već postojećeg modela ako postoji.

```
        self.actor = pickle.load(open(name + 'Actor', 'rb'))\
            if shouldLoad and os.path.exists(name + 'Actor') else Network(
                [self.observationDim, 256, 256, self.actionDim],
```

```
            nn.Tanh,
            learningRate,
            self.device
        )
    self.critic1 = pickle.load(open(name + 'Critic1', 'rb'))\
        if shouldLoad and os.path.exists(name + 'Critic1') else
        ↪ Network(
            [self.observationDim + self.actionDim, 256, 256, 1],
            nn.Identity,
            learningRate,
            self.device
        )
    self.critic2 = pickle.load(open(name + 'Critic2', 'rb'))\
        if shouldLoad and os.path.exists(name + 'Critic2') else
        ↪ Network(
            [self.observationDim + self.actionDim, 256, 256, 1],
            nn.Identity,
            learningRate,
            self.device
        )
```

Inicijaliziramo ciljne mreže glumca i kritičara dubokim kopiranjem već definiranih atributa glumca i kritičara (čime stvaramo potpuno nove objekte neovisne o prvom) ili, ako postoje, učitavanjem već postojećih modela.

```
self.targetActor = pickle.load(open(name + 'TargetActor', 'rb'))\
    if shouldLoad and os.path.exists(name + 'TargetActor') else\
    deepcopy(self.actor)
self.targetCritic1 = pickle.load(open(name + 'TargetCritic1',
↪ 'rb'))\
    if shouldLoad and os.path.exists(name + 'TargetCritic1') else\
    deepcopy(self.critic1)
self.targetCritic2 = pickle.load(open(name + 'TargetCritic2',
↪ 'rb'))\
```

```
        if shouldLoad and os.path.exists(name + 'TargetCritic2') else\
        deepcopy(self.critic2)
```

Pogledajmo sada metode klase Agent. Metoda `getDeterministicAction()` za dano stanje vraća akciju koju dobivamo primjenom mreže glumca na stanje, tj. unaprijednom propagacijom mreže strategije.

```
def getDeterministicAction(self, state: np.ndarray) -> np.ndarray:
    actions: T.Tensor = self.actor.forward(T.tensor(state,
    ↪  device=self.device))
    return actions.cpu().detach().numpy()
```

Metoda `getNoisyAction()` dodaje šum izračunatoj akciji iz prethodne metode i ograničava dobivenu vrijednosti u intervalu $\langle -1, 1\rangle$ pomoću funkcije `clip`. Kao što smo spomenuli prije, uzimamo Gaussov šum srednje vrijednosti 0 , a $\langle -1, 1\rangle = \langle a_{Low}, a_{High}\rangle$.

```
def getNoisyAction(self, state: np.ndarray, sigma: float) ->
↪  np.ndarray:
    deterministicAction = self.getDeterministicAction(state)
    noise = np.random.normal(0, sigma, deterministicAction.shape)
    return np.clip(deterministicAction + noise, -1, +1)
```

Metoda `computeTargets()` vraća ciljne vrijednosti. Prvo računa ciljne akcije unaprijednom propagacijom kroz ciljnu mrežu glumca uz sljedeće stanje iz međuspremnika za ponavljanje kao ulaz. Zatim ciljnim akcijama dodaje Gaussov šum srednje vrijednosti 0, ograničava šum u intervalu $\langle -\text{trainingClip}, +\text{trainingClip}\rangle$ pa ograničava ciljne akcije u intervalu $\langle -1, 1\rangle$.

```
def computeTargets(
    self, rewards: T.Tensor, nextStates: T.Tensor, dones: T.Tensor,
    trainingSigma: float, trainingClip: float
) -> T.Tensor:
```

```
        targetActions = self.targetActor.forward(nextStates.float())
        noise = np.random.normal(0, trainingSigma, targetActions.shape)
        clippedNoise = T.tensor(
            np.clip(noise, -trainingClip, +trainingClip),
            ↪ device=self.device
        )
        targetActions = T.clip(targetActions + clippedNoise, -1, +1)
```

Nadalje, unaprijednom propagacijom kroz ciljne mreže kritičara računa ciljne Q vrijednosti, uzima manju od te dvije vrijednosti i, konačno, vraća ciljnu vrijednost koristeći odgovarajuću formulu.

```
        targetQ1Values = T.squeeze(
            self.targetCritic1.forward(T.hstack([nextStates,
            ↪ targetActions]).float())
        )
        targetQ2Values = T.squeeze(
            self.targetCritic2.forward(T.hstack([nextStates,
            ↪ targetActions]).float())
        )
        targetQValues = T.minimum(targetQ1Values, targetQ2Values)
        return rewards + self.gamma*(1 - dones)*targetQValues
```

Sada definiramo i metode za vraćanje gubitka za glumca i kritičare iz koraka 14 i 16 u pseudokodu TD3 algoritma. Za kritičare uzimamo MSE gubitak, a za glumca srednju vrijednost Q vrijednosti. Kako za glumca imamo problem maksimizacije, vraćamo negativnu vrijednost da bi ga preveli u problem minimizacije.

```
    def computeQLoss(
        self, network: Network, states: T.Tensor, actions: T.Tensor,
        ↪ targets: T.Tensor
    ) -> T.Tensor:
```

```
        QValues = T.squeeze(network.forward(T.hstack([states,
        ↪  actions]).float()))
        return T.square(QValues - targets).mean()

    def computePolicyLoss(self, states: T.Tensor):
        actions = self.actor.forward(states.float())
        QValues = T.squeeze(self.critic1.forward(T.hstack([states,
        ↪  actions]).float()))
        return -QValues.mean()
```

Definiramo i `updateTargetNetwork()` metodu za ažuriranje ciljne mreže polyak prosjekom.

```
    def updateTargetNetwork(self, targetNetwork: Network, network:
    ↪  Network):
        with T.no_grad():
            for targetParameter, parameter in zip(
                targetNetwork.parameters(), network.parameters()
            ):
                targetParameter.mul_(1 - self.tau)
                targetParameter.add_(self.tau*parameter)
```

Naposlijetku definiramo metodu za ažuriranje parametara mreža. Prvo uzimamo slučajan uzorak određene veličine iz međuspremnika za ponavljanje pa spremimo akcije, stanja, nagrade, sljedeća stanja i oznaku kraja iz uzorka u odgovarajuće varijable tipa tenzor.

```
    def update(
        self, miniBatchSize: int, trainingSigma: float, trainingClip:
        ↪  float,
        updatePolicy: bool
    ):
        miniBatch = self.buffer.getMiniBatch(miniBatchSize)
```

```
states = T.tensor(miniBatch["states"], requires_grad=True,
↪  device=self.device)
actions = T.tensor(miniBatch["actions"], requires_grad=True,
↪  device=self.device)
rewards = T.tensor(miniBatch["rewards"], requires_grad=True,
↪  device=self.device)
nextStates = T.tensor(
    miniBatch["nextStates"], requires_grad=True,
    ↪  device=self.device
)
dones = T.tensor(miniBatch["doneFlags"], requires_grad=True,
↪  device=self.device)
```

Nadalje, pomoću gore definiranih metoda računamo ciljeve pa gubitke Q funkcija. Zatim radimo korak gradijentog spusta za obje Q funkcije (tj. mreže kritičara). Ako je vrijeme za ažuriranje strategije (glumca), analogno računamo gubitak pa radimo korak gradijentog spusta mreže glumca te ažuriramo ciljne mreže.

```
targets = self.computeTargets(
    rewards, nextStates, dones, trainingSigma, trainingClip
)
Q1Loss = self.computeQLoss(self.critic1, states, actions, targets)
self.critic1.gradientDescentStep(Q1Loss, True)
Q2Loss = self.computeQLoss(self.critic2, states, actions, targets)
self.critic2.gradientDescentStep(Q2Loss)
if updatePolicy:
    policyLoss = self.computePolicyLoss(states)
    self.actor.gradientDescentStep(policyLoss)
    self.updateTargetNetwork(self.targetActor, self.actor)
    self.updateTargetNetwork(self.targetCritic1, self.critic1)
    self.updateTargetNetwork(self.targetCritic2, self.critic2)
```

Time je definirana klasa `Agent`.

3.3.3 Main funkcija

Definirajmo hiperparametre modela koje sami biramo i možemo ih mijenjati ovisno o performansama.

Za faktor diskontiranja γ uzimamo vrijednost što bliže 1 jer time izbjegavamo "zanemarivanje" budućih nagrada. Za stopu učenja biramo (proizvoljno) 0.0003. Za parametar kod ažuriranja ciljnih mreža uzimamo $0.005 =$ `tau` $= 1-\rho$, što znači da parametre ciljne mreže množimo koeficijentom blizu 1 i dodajemo parametre mreže pomnožene s koeficijentom blizu nule. Time dobivamo sporije ažuriranje ciljne mreže. Nadalje, definiramo `actionSigma` (`trainingSigma`) koji predstavljaju standardnu devijaciju u normalnoj distribuciji iz koje definiramo aditivni šum strategiji (ciljnim akcijama). Te vrijednosti postavljamo na 0.1 (0.2) jer akcije poprimaju vrijednosti u intervalu $\langle -1, 1\rangle$. Iz istog razloga parametar ograničavanja šuma ciljnih akcija `trainingClip` postavljamo na 0.5 da ograničimo aditivni šum na $\langle -0.5, 0.5\rangle$ da budu što bliže stvarnim akcijama. Veličinu uzorka koju uzimamo iz međuspremnika postavljamo na 100, a parametre glumca ažuriramo svaki drugi put u odnosu na ažuriranje parametara kritičara.

```
gamma = 0.99
learningRate = 3e-4
tau = 0.005
actionSigma = 0.1
trainingSigma = 0.2
trainingClip = 0.5
miniBatchSize = 100
policyDelay = 2
```

Prvo stvorimo gym okolinu pa instanciramo agenta iz klase Agent. Trenutno

stanje dobivamo iz okoline pomoću funkcije `reset()`. Uključujemo i kod za potencijalni nastavak prethodnog treniranja spremljenog u **csv** datoteku (ako postoji).

```
resume = True      #nastavi iz prethodnog "checkpointa" ako postoji
render = True      #prikaz na ekranu
envName = "BipedalWalker-v3"
env = gym.make(envName)
csvName = envName + '-data.csv'
agent = Agent(env, learningRate, gamma, tau, resume)
state = env.reset()
step = 0
runningReward = None
numEpisode = 0
if path.exists(csvName):
    fileData = list(csv.reader(open(csvName)))
    lastLine = fileData[-1]
    numEpisode = int(lastLine[0])
rewards_list = []
```

Neka treniramo nad 2000 epizoda. Maksimalan broj koraka agenta epizode postavljen je od strane okoline na 1600. Svaka epizoda sastoji se od sljedećeg:

```
while numEpisode <= 2000:
```

1. odredi akciju i dodaj joj šum pomoću metode `getNoisyAction()`

```
    action = agent.getNoisyAction(torch.from_numpy(state).float(),
    ↪  actionSigma)
```

2. napravi korak u okolini i skupi informacije (definirane varijablama `nextState`, `reward`, `done`, `info`)

```
    nextState, reward, done, info = env.step(action)
```

3. spremi podatke u međuspremnik

```
agent.buffer.store(state, action, reward, nextState, done)
```

4. Ako je agent u terminalnom stanju, evaluiramo ga na testnoj epizodi. Računamo i spremamo u **csv** datoteku redni broj testne epizode, ukupnu nagradu u testnoj epizodi te pomični prosjek nagrada svih testnih epizoda do trenutne. Ako smo u zadnjih 100 epizoda prosječno postigli nagrade veće od 300, završavamo treniranje.

```
if done:
    numEpisode += 1
    sumRewards = 0
    state = env.reset()
    done = False
    while not done:
        action = agent.getDeterministicAction(np.float32(state))
        nextState, reward, done, info = env.step(action)
        if render:
            env.render()
        state = nextState
        sumRewards += reward
    state = env.reset()
    runningReward = sumRewards\
        if runningReward is None\
        else runningReward*0.99 + sumRewards*0.01
    fields = [numEpisode, sumRewards, runningReward]
    with open(env.name + '-data.csv', 'a', newline='') as f:
        writer = csv.writer(f)
        writer.writerow(fields)
    agent.save()
    print(
        f"episode {numEpisode:6d} --- " +
        f"total reward: {sumRewards:7.2f} --- " +
        f"running average: {runningReward:7.2f}",
```

```
            flush=True
        )
        rewards_list.append(sumRewards)
        if (np.mean(rewards_list[-100:]) >= 300):
                        break
    else:
        state = nextState
    step += 1
```

5. Ažuriraj parametre mreža kritičara pomoću metode `update()` klase `Agent`. Metodi šaljemo i parametar `shouldUpdatePolicy` tipa `bool` koji određuje je li vrijeme za ažuriranje glumca ili ne.

```
shouldUpdatePolicy = step % policyDelay == 0
agent.update(
    miniBatchSize, trainingSigma, trainingClip, shouldUpdatePolicy
)
```

Poglavlje 4

Drugi algoritam

ARS (eng. *Augmented Random Search*) prezentiran je 2018. godine u [15] kao algoritam koji je barem 15 puta efikasniji od tadašnjih najbržih konkurentnih model-free metoda na MuJoCo referentnoj točki za probleme neprekidne kontrole . On se temelji na poboljšanju postojećeg algoritma koji se zove Basic Random Search.
Kao što znamo, rješavanje problema učenjem podrškom zahtijeva pronalazak strategije za kontrolu dinamičkih sustava s ciljem maksimiziranja prosječne nagrade. Napišimo formulaciju problema na malo drugačiji način nego prije:

$$\max_{\theta \in \mathbb{R}^d} \mathop{\mathbb{E}}_{\xi} \big[r\big(\pi_\theta, \xi\big) \big], \tag{4.1}$$

gdje $\theta \in \mathbb{R}^n$ parametrizira strategiju $\pi_\theta : \mathbb{R}^n \to \mathbb{R}^p$. Slučajna varijabla ξ kodira nasumičnost okoline, tj. slučajna početna stanja i stohastičke prijelaze. Vrijednost $r(\pi_\theta, \xi)$ je nagrada dobivena strategijom π_θ na jednoj putanji generiranoj u sustavu. Općenito bi mogli koristiti stohastičke strategije, ali metoda koju izlažemo koristi deterministične strategije.

4.1 Basic Random Search (BRS)

Primjetimo da formulacija problema (4.1) ima za cilj optimizirati nagradu direktno optimizacijom parametara strategije θ. Mi ćemo razmatrati metode koje istražuju u prostoru parametara umjesto u akcijskom prostoru. Taj izbor čini treniranje učenjem podrškom ekvivalentno optimizaciji bez gradijenta s evaluacijama funkcije šuma. Jedna od najstarijih i najjednostavnijih takvih optimizacijskih metoda je **nasumična pretraga (eng. random search)**. Nasumična pretraga ravnomjerno i nasumučno izabire smjer na sferi u prostoru parametara pa optimizira funkciju niz odabrani smjer.

Primitivan oblik nasumične pretrage jednostavno računa aproksimaciju konačnim razlikama niz nasumični smjer i onda radi korak u tom smjeru bez korištenja linijske pretrage. Naš algoritam **ARS** temelji se upravo na ovoj jednostavnoj strategiji.

Za ažuriranje parametara θ strategije π_θ, naša metoda iskorištava smjerove ažuriranja u obliku:

$$\frac{r(\pi_{\theta+\nu\delta}, \xi_1) - r(\pi_{\theta-\nu\delta}, \xi_2)}{\nu}, \tag{4.2}$$

za dvije neovisne i jednako distribuirane normalne varijable ξ_1 i ξ_2, pozitivan realan broj ν i Gaussov vektor δ srednje vrijednosti 0. Poznato je da je takav prirast ažuriranja nepristran estimator gradijenta obzirom na θ od $\mathbb{E}_\delta \mathbb{E}_\xi[r(\pi_{\theta+\nu\delta}, \xi)]$, glatku verziju cilja (4.1) koja je blizu originalnom cilju (4.1) kada je ν mali.

Dakle, ideja BRS-a je izabrati parametriziranu strategiju π_θ, perturbirati parametre θ primjenom $+\nu\delta$ ili $-\nu\delta$, gdje je $\nu < 1$ konstantan šum i δ slučajan broj generiran iz normalne distribucije. Zatim izvršavamo akcije obzirom na $\pi_{\theta+\nu\delta}$ i $\pi_{\theta-\nu\delta}$ pa računamo nagrade akcija $r(\theta + \nu\delta)$ i $r(\theta - \nu\delta)$.

Kada imamo nagrade perturbiranog θ, računamo prosjek

$$\triangle = \frac{1}{N}\sum_{\delta}[r(\theta + \nu\delta) - r(\theta - \nu\delta)]\delta$$

za sve δ te ažuriramo parametre θ koristeći $\triangle$ i stopu učenja α:

$$\theta^{j+1} = \theta^j + \alpha\triangle$$

Algoritam 3 BRS

1: **Hiperparametri:** duljina koraka α, broj smjerova N po iteraciji, standardna devijacija šuma istraživanja ν
2: **Inicijaliziraj:** $\theta_0 = 0$ i $j = 0$
3: **while** kriterij zaustavljanja nije zadovoljen **do**
4: Uzorkuj $\delta_1, \delta_2, \cdots, \delta_N$ iste veličine kao θ_j, neovisne i jednako distribuirane slučajne varijable standardne normalne distribucije
5: Napravi $2N$ izvođenja horizonta H i izračunaj pripadne nagrade koristeći strategije

$$\pi_{j,k,+}(x) = \pi_{\theta_j + \nu\delta_k(x)}$$

i

$$\pi_{j,k,-}(x) = \pi_{\theta_j - \nu\delta_k(x)},$$

za $k \in \{1, 2, \cdots, N\}$.
6: Izvrši korak ažuriranja:

$$\theta_{j+1} = \theta_j + \frac{\alpha}{N}\sum_{k=1}^{N}[r(\pi_{j,k,+}) - r(\pi_{j,k,-})]\delta_k.$$

7: $j = j + 1$
8: **end while**

4.2 Augmented Random Search (ARS)

Predstavimo sada tri poboljšanja BRS-a. Kroz ostatak rada koristit ćemo M za oznaku parametara strategije jer naša metoda koristi linearne strategije, dakle M je $p \times n$ matrica.

4.2.1 Skaliranje standardnom devijacijom σ_R

Kako treniranje strategija napreduje, nasumična pretraga u prostoru parametara strategija može dovesti do velikih varijacija u nagradama kroz iteracije. Kao rezultat, teško je odabrati fiksnu duljinu koraka α koji ne dopušta štetne promjene između malih i velikih koraka. Taj problem velikih varijacija između razlika $r(\pi_{M+\nu\delta}) - r(\pi_{M-\nu\delta})$ rješavamo skaliranjem koraka ažuriranja standardnom devijacijom σ_R $2N$ nagrada koje su skupljene u svakoj iteraciji. Intuitivno je jasno da će standardna devijacija nagrada imati ulaznu putanju što vrijeme ide dalje jer perturbacije težina strategije pri visokim nagradama mogu natjerati našeg agenta da padne ranije, što dovodi do većih varijacija u skupljenim nagradama. Dakle, kad ne bi skalirali s σ_R, naša metoda bi npr. u iteraciji 300 radila korake koji su tisuću puta veći od koraka na početku treniranja.

4.2.2 Normalizacija stanja

Normalizacija stanja intuitivno osigurava da strategija stavi jednaku težinu na različite komponente stanja. Da steknemo intuiciju zašto to pomaže, pretpostavimo da jedna koordinata stanja poprima vrijednosti u rasponu $[90, 100]$, a druga u rasponu $[-1, 1]$. Tada bi prva koordinata stanja dominirala u računu, dok druga ne bi imala nikakav utjecaj. Dakle, za težine M strategije i smjer perturbacije δ imamo

$$(M + \nu\delta)\text{diag}(\Sigma)^{-\frac{1}{2}}(x - \mu) = \left(\tilde{M} + \nu\delta\text{diag}(\Sigma)^{-\frac{1}{2}}\right)(x - \mu), \tag{4.3}$$

gdje je $\tilde{M} = M\text{diag}(\Sigma)^{-\frac{1}{2}}$

4.2.3 Korištenje smjerova s najboljim rezultatima

Sjetimo se da je naš cilj maksimizirati skupljene nagrade. Međutim, računamo prosječnu nagradu 2N epizoda nad strategijama s perturbiranim parametrima trenutne strategije u svakoj iteraciji. To stvara probleme jer ako su npr. nagrade $r(\pi_{j,k,+})$ i $r(\pi_{j,k,-})$ male u usporedbi s ostalim nagradama, to znači da će pomak M_j u bilo kojem od smjerova δ_k i $-\delta_k$ smanjiti prosjek. Jedan način za rješavanje ovog problema je silazno sortiranje smjerova perturbacije δ_k obzirom na $\max\{r(\pi_{j,k,+}), r(\pi_{j,k,-})\}$ i onda uzeti samo prvih b smjerova za ažuriranje težina strategije. Primjetimo da je za $b = N$ algoritam jednak onome bez ovog poboljšanja.

Algoritam 4 ARS

1: **Hiperparametri:** duljina koraka α, broj smjerova N po iteraciji, standardna devijacija šuma istraživanja ν, broj smjerova s najboljim rezultatima $b < N$

2: **Inicijaliziraj:** $M_0 = 0 \in \mathbb{R}^{p\times n}$, $\mu_0 = 0 \in \mathbb{R}^n$, $\Sigma_0 = I_n \in \mathbb{R}^{n\times n}$, $j = 0$

3: **while** kriterij zaustavljanja nije zadovoljen **do**

4: Uzorkuj $\delta_1, \delta_2, \cdots, \delta_N \in \mathbb{R}^{p\times n}$ neovisne i jednako distribuirane slučajne varijable standardne normalne distribucije

5: Napravi $2N$ izvođenja horizonta H i izračunaj pripadne nagrade koristeći $2N$ strategija

$$\pi_{j,k,+}(x) = (M_j + \nu\delta_k)\text{diag}(\Sigma_j)^{-\frac{1}{2}}(x - \mu_j)$$

i

$$\pi_{j,k,-}(x) = (M_j - \nu\delta_k)\text{diag}(\Sigma_j)^{-\frac{1}{2}}(x - \mu_j),$$

za $k \in 1, 2, \cdots, N$.

6: Sortiraj smjerove δ_k prema $\max\{r(\pi_{j,k,+}), r(\pi_{j,k,-})\}$, označi s $d_{(k)}$ k-ti najveći smjer i s $\pi_{j,(k),+}$ i $\pi_{j,(k),-}$ pripadne strategije.

7: Izvrši korak ažuriranja:

$$Mj + 1 = M_j + \frac{\alpha}{b\sigma_R}\sum_{k=1}^{b}[r(\pi_{j,(k),+}) - r(\pi_{j,(k),-})]\delta_{(k)},$$

gdje je σ_R standardna devijacija $2b$ nagrada korištenih u koraku ažuriranja.

8: Neka su μ_{j+1}, Σ_{j+1} srednja vrijednost i kovarijanca od $2NH(j+1)$ stanja s kojima smo se susreli od početka treniranja.

9: $j = j + 1$

10: **end while**

4.3 Implementacija

Pogledajmo sada implementaciju ARS algoritma na BipedalWalker problem preuzetu s GitHub repozitorija [16]. Kod je u programskom jeziku *Python*. Neke dijelove koda izostavit ćemo i/ili spomenuti samo opisno radi fokusa na problematici samog algoritma.

4.3.1 Klasa strategije

U konstruktoru klase strategije `Policy` definiramo atribute `theta` (matrica težina mreže dimenzija (dimenzija akcija)×(dimenzija stanja), tj. parametri koje treniramo), `learning_rate` (stopa učenja), `num_directions` (broj smjerova istraživanja), `num_best_directions` (broj b najboljih smjerova koje uzimamo u obzir) i `noise` (parametar šuma μ kojim množimo perturbacije δ_k).

```
import numpy as np
class Policy():
    def
    ↪ __init__(self,num_observations,num_actions,lr,num_dirs,num_dirs_best,noise):
        self.theta = np.zeros((num_actions,num_observations))
        self.learning_rate = lr
        self.num_directions = num_dirs
        self.num_best_directions = num_dirs_best
        self.noise = noise
```

Metoda `evaluate()` računa vrijednost strategije nad stanjem sa ili bez dodanog šuma (ovisno o tome u kojem trenutku pozivamo metodu). Primjetimo da strategiju zapravo modeliramo perceptronom. Imamo težine `theta` (sa ili bez dodanog šuma) koje matrično množimo s ulazom `state`. Aktivacijska funkcija je u ovom slučaju identiteta.

```
    def evaluate(self,state, delta=None, direction=None):
        if direction is None:
            return self.theta.dot(state)
        elif direction == "positive":
            return (self.theta+self.noise*delta).dot(state)
        else:
            return (self.theta-self.noise*delta).dot(state)
```

Metoda `sample_deltas()` uzorkuje perturbacije, tj. matrice δ_i, $i = 1, \ldots, N$ istih dimenzija kao težine `theta` s elementima iz standardne normalne distribucije.

```
   def sample_deltas(self):
        return [np.random.randn(*self.theta.shape) for i in
        ↪  range(self.num_directions)]
```

Metoda `update()` radi korak ažuriranja (korak 7 algoritma)

```
    def update(self,rollouts,sigma_r):
        step = np.zeros(self.theta.shape)
        for positive_reward,negative_reward,d in rollouts:
            step += (positive_reward-negative_reward)*d
        self.theta += self.learning_rate/(self.num_best_directions *
        ↪  sigma_r)*step
```

4.3.2 Klasa Normalizator

Klasa `Normalizer` služi nam za normalizaciju stanja. U njenom konstruktoru definiramo atribute `n` (redni broj stanja), `mean` (srednja vrijednost), `mean_difference` (suma kvadratnih razlika koja nam služi za izračun varijance), `variance` (varijanca).

```
import numpy as np
class Normalizer():
    def __init__(self,num_inputs):
```

```
        self.n = np.zeros(num_inputs)
        self.mean = np.zeros(num_inputs)
        self.mean_difference = np.zeros(num_inputs)
        self.variance = np.zeros(num_inputs)
```

Metoda `normalize()` normalizira stanje.

```
    def normalize(self,state):
        observation_mean = self.mean
        observation_standard_deviation = np.sqrt(self.variance)
        return (state-observation_mean)/observation_standard_deviation
```

Metoda `observe()` ažurira varijancu i aritmetičku sredinu pri prijelazu u novo stanje. `clip` metodom u izračunu varijance osiguravamo da varijanca ne bude 0 (da ne dođe do dijeljenja s nulom kod normalizacije stanja).

```
    def observe(self,state):
        self.n += 1.
        last_mean = self.mean.copy()
        self.mean += (state - self.mean)/self.n
        self.mean_difference += (state-last_mean)*(state-self.mean)
        self.variance = (self.mean_difference/self.n).clip(min=1e-2)
```

4.3.3 Main funkcija

Definiramo broj epoha treniranja (`epochs`), stopu učenja (`lr`) stavljamo na 0.02, za broj smjerova N po epohi (`num_dirs`) biramo 32, definiramo 2000 kao maksimalan broj koraka (`total_episodes`) pri svakom od $2N$ evaluacija strategija, za broj b najboljih smjerova (`num_dirs_best`) uzimamo 16, a za standardnu devijaciju šuma istraživanja (`noise`) uzimamo 0.03.

```
epochs = 2000
lr = 0.02
num_dirs = 32
```

```
total_episodes = 2000
num_dirs_best = 16
noise = 0.03
```

Funkcija `exploration` evaluira danu strategiju $\pi_{j,k,+}$ ili $\pi_{j,k,-}$. Dakle, radi korake u okolini dok ne dođe u završno stanje ili broj koraka prijeđe 2000. U svakom koraku normalizira stanje, primjeni strategiju na stanje za dobijanje akcije pa izvršava tu akciju u okolini. Nagradu koju nam vraća okolina prvo skaliramo u segment $[-1, 1]$ na način da visoke negativne nagrade (manje od -1) poprimaju vrijednost -1, a visoke pozitivne nagrade (veće od 1) poprimaju vrijednost 1. Pretpostavka je da ćemo time smanjiti pristranost modela prema jako velikim pozitivnim ili negativnim nagradama koje će imati velik utjecaj na konačnu sumu. Funkcija vraća sumu takvih nagrada.

```
def exploration(env, normalizer, policy, direction=None, delta=None):
    state = env.reset()
    done = False
    episode = 0.
    sigma_rewards = 0
    while not done and episode < total_episodes:
        normalizer.observe(state)
        state = normalizer.normalize(state)
        action = policy.evaluate(state, delta, direction)
        state, reward, done, _ = env.step(action)
        reward = max(min(reward, 1), -1)
        sigma_rewards += reward
        episode += 1
    return sigma_rewards
```

Funkcija `train_ars_agent` glavna je funkcija za treniranje modela. Prvo uzorkujemo matrice δ_i, $i = 1, \ldots, N$ koje nam služe za aditivni šum, zatim evaluiramo svaku od perturbiranih strategija $\pi_{j,k,+}$ i $\pi_{j,k,-}$ dobivene dodava-

njem spomenutog šuma uz konstantni multiplikativni koeficijent ν parametrima strategije.

```
def train_ars_agent(env,policy,normalizer,epochs,lr,num_dirs,
↪  total_episodes, num_dirs_best,noise = 0.03):
    for epoch in range(epochs):
        deltas = policy.sample_deltas()
        positive_rewards = [0] * num_dirs
        negative_rewards = [0] * num_dirs
        for i in range(num_dirs):
            positive_rewards[i] =
            ↪  exploration(env,normalizer,policy,direction="positive",delta=deltas[i])
        for i in range(num_dirs):
            negative_rewards[i] =
            ↪  exploration(env,normalizer,policy,direction="negative",delta=deltas[i])
```

Nadalje, računamo standardnu devijaciju nagrada dobivenih evaluacijama perturbiranih strategija pa sortiramo nagrade prema $\max\{r(\pi_{j,k,+}), r(\pi_{j,k,-})\}$. Konačno, ažuriramo parametre strategije koristeći prvih $b = 16$ sortiranih parova nagrada i evaluiramo ažuriranu strategiju na testnoj epizodi za praćenje performansi modela. Ako smo u prethodnih 100 epoha prosječno ostvarili ukupnu nagradu veću od 300, po definiciji smo riješili Bipedal Walker problem te završavamo treniranje.

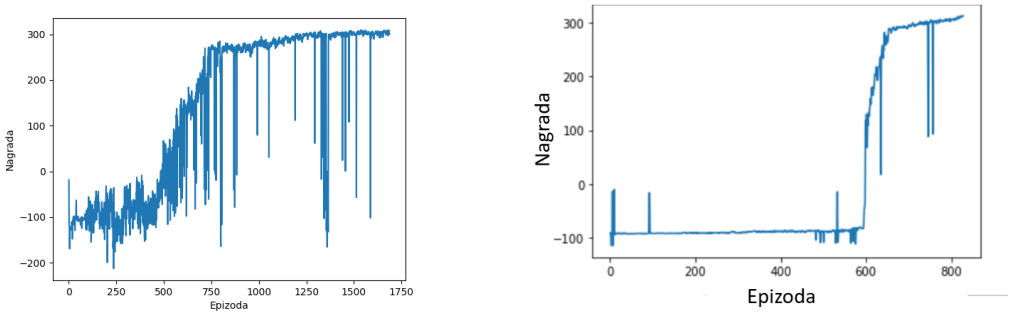
```
        final_rewards = np.array(positive_rewards+negative_rewards)
        sigma_r = final_rewards.std()
        scores = {score:max(positive_reward, negative_reward) for score,
        ↪  (positive_reward, negative_reward) in
        ↪  enumerate(zip(positive_rewards, negative_rewards))}
        order = sorted(scores.keys(), key=lambda x: scores[x],
        ↪  reverse=True)[:num_dirs_best]
        rollouts = [(positive_rewards[k], negative_rewards[k], deltas[k])
        ↪  for k in order]
```

```
policy.update(rollouts, sigma_r)
reward_eval = exploration(env,normalizer,policy)
print('Epoch:', epoch, '   -------------X--------------
↪  Reward:', reward_eval)
if(np.mean(rewards_list[-100:]) >= 300):
  break
```

Poglavlje 5

Rezultati

Pri evaluaciji rezultata algoritama uzet ćemo u obzir sljedeće: Je li algoritam rezultirao modelom koji rješava Bipedal Walker problem? Je li algoritam brži ili sporiji u treniranju od drugog algoritma? Konačno, kao manje formalna metrika, kakva je subjektivna kvaliteta hoda modela?
Prvo usporedimo rezultate jednog treniranja algoritama (Slika 5.1).



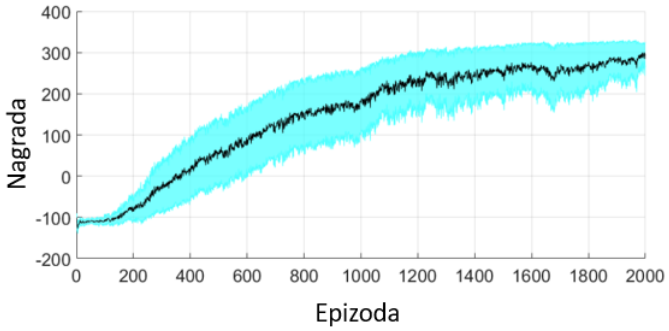
(a) TD3 (b) ARS

Slika 5.1: Evaluacija jednog treniranja oba algoritma.

Napomenimo da su, u svrhe usporedbe, prikazane nagrade ARS algoritma bez koraka u kojem ograničavamo nagrade u segment $[-1, 1]$ (naravno, taj

korak nismo izostavili u treniranju). Vidimo da ARS algoritmu treba relativno mali broj koraka za rješavanje Bipedal Walker problema. Kod toga uzimamo u obzir činjenicu da unatoč tome što za jedan korak (ažuriranje) modela radimo čak 64 epizode naspram jedne kod TD3, ARS-u u prosjeku treba manje vremena za taj jedan korak. S druge strane, TD3 također uspijeva naučiti agenta hodati, ali mu za to treba više vremena. Iako TD3 uspijeva formalno riješiti problem Bipedal Walker tek u 1750 epizoda, on potiže dobre rezultate već od približno 800-te epizode, tj. dobiva nagrade jako blizu 300. Razlog nižih nagrada unatoč stizanju do cilja ili blizu cilja leži u načinu na koji agent hoda, tj. radi akcije koje okolina manje preferira ili ne stigne do cilja prije nego što dosegne maksimalan broj koraka. Također, može se dogoditi da agent u više epizoda tijekom treniranja "zapne" u stanju u kojem stoji na mjestu, što samo usporava treniranje jer ne napredujemo, samo čekamo da istekne maksimalan broj koraka epizode.

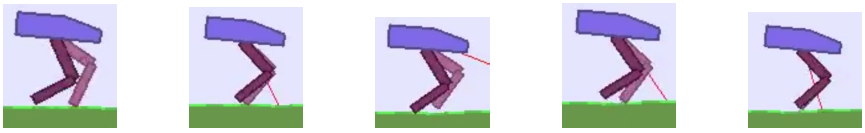
Da steknemo bolji dojam o performansama TD3 algoritma, pogledajmo sada prosječne performanse 64 različitih treniranja. Osjenčano područje predstavlja standardnu devijaciju prosječnih evaluacija svih treniranja.



Slika 5.2: Prosječne performanse 64 treniranja modela TD3 algoritmom

Vidimo da je ponekad potrebno i skoro 2000 epizoda za rješavanje pro-

blema. Dakle, treniranje modela TD3 algoritmom vremenski je puno zahtjevnije od treniranja ARS algoritmom. ARS algoritmom rješavamo BipedalWalker problem već u nekoliko sati, a TD3-u za to treba puno više vremena. No, nije nagrada i vrijeme treniranja jedini način na koji možemo uspoređivati performanse algoritama. Ovdje se ipak radi o hodanju čiju tehniku možemo vizualno ocijeniti na manje formalan način. Treba uzeti u obzir da način hodanja koji algoritam ocijeni visokom nagradom ne mora nužno nama izgledati prirodno. Isto tako, način hodanja jednog modela ne mora biti sličan onome drugog modela, iako su trenirani istim algoritmom. No, ipak probajmo vizualno ocijeniti hodanje dva istrenirana modela:



(a) (b) (c) (d) (e)

Slika 5.3: hod TD3 agenta

(a) (b) (c) (d) (e)

Slika 5.4: hod ARS agenta

Iz snimke hoda i gornjih isječaka možemo reći da nijedan ni drugi algoritam ne stvara agenta koji hoda prirodno, tj. ne izmjenjuje noge. Noga svjetlije boje je uvijek ispred noge tamnije boje ili u ravnini s njom. Zbog toga oba robota više izgledaju kao da poskakuju nego hodaju, iako se drže relativno blizu tla. TD3 agent više spaja noge, dok ARS agent naizgled vuće tamniju nogu za svjetlijom. Vizualnim pregledom teško je zaključiti

koji robot "bolje" hoda, no malu prednost bi mogli dati TD3 agentu. Dakle, treniranje TD3 algoritmom daleko je vremenski zahtijevnije od treniranja ARS algoritmom, no, u prikazana dva istrenirana modela, TD3 rezultira za nijansu boljim hodom.

Za dodatno stjecanje intuicije, uočimo ključne razlike između korištenih algoritama:

1. ARS, za razliku od TD3, *ne iskorištava odmah* nagradu dobivenu izvršavanjem neke akcije kao povratnu informaciju o tome kako napreduje. Umjesto toga, tek nakon što završi epizodu za određenu strategiju, koristi akumuliranu nagradu cijele epizode za evaluaciju strategije. Dakle, ARS istražuje prostor strategija, dok TD3 istražuje prostor akcija. Ovo je vrlo zanimljivo, jer unatoč tome što ne iskorištava potpuno mogućnosti povratnih informacija okoline, ARS postiže izvrsne rezultate.

2. Način na koji ažuriramo težine modela. ARS koristi jednostavnu metodu konačnih razlika, dok TD3 korsti gradijetni spust, puno netrivijalniju metodu koja stvarno računa derivacije funkcije gubitka umjesto same aproksimacije. Razlog zašto ne možemo koristiti gradijentni spust kod ARS-a jer uopće nemamo funkciju vrijednosti.

3. ARS koristi perceptron, dakle plitko učenje, dok TD3 korsti višeslojne neuronske mreže, tj. radi se o dubokom učenju.

Uzevši sve navedeno u obzir, očito je da je TD3 daleko kompleksniji algoritam od ARS-a i zato zahtijeva više vremena za treniranje. S druge strane, ARS je primjer da metoda ne mora biti složena da bi davala jako lijepe rezultate. Treba naglasiti da oba algoritma uspijevaju, prije ili kasnije, riješiti zadani problem. Najbitnije je da u oba slučaja dobivamo robota koji *zna hodati*, ne ulazeći u kvalitetu samog hoda. Naravno, postoji mogućnost boljih

rezultata testiranjem različitih vrijednosti hiperparametara algoritama, kao što je, primjerice, broj smjerova i broj najboljih smjerova kod ARS-a. Osim toga, obzirom na to da su prezentirani 2018. godine, valja napomenuti da implementirani algoritmi trenutno vjerojatno nisu najbolji za dani problem, no sigurno su vrijedni spomena i bitan korak prema rješavanju problema neprekidne kontrole.

Literatura

[1] Barto, Sutton (2018.), *Reinforcement Learning: An Introduction*

[2] Charu C. Aggarwal, *Neural Networks and Deep Learning, A Textbook*

[3] Keerthana V, *Artificial Neural Network, Its inspiration and the Working Mechanism*
https://www.analyticsvidhya.com/blog/2021/04/artificial-neural-network-its-inspiration-and-the-working-mechanism/

[4] David Fumo, *A Gentle Introduction To Neural Networks Series*
https://towardsdatascience.com/a-gentle-introduction-to-neural-networks-series-part-1-2b90b87795bc

[5] Matea Kalinić (2017.), *Matematičke osnove neuronskih mreža*

[6] OpenAI, Spinning up, *Introduction to RL, Algorithm Docs*
https://spinningup.openai.com

[7] Lillicrap, Hunt, Pritzel, Heess, Erez, Tassa, Silver, Wierstra, *Continuous control with deep reinforcement learning*
https://arxiv.org/pdf/1509.02971.pdf

[8] OpenAI Gym, *BipedalWalker-v2 dokumentacija*
https://github.com/openai/gym/wiki/BipedalWalker-v2

[9] Fujimoto, van Hoof, Meger (2018.), *Addressing Function Approximation Error in Actor-Critic Methods*
https://arxiv.org/pdf/1802.09477.pdf

[10] Levi Fussel, *Exploring Bipedal Walking Through Machine Learning Techniques* https://project-archive.inf.ed.ac.uk/ug4/20181201/ug4_proj.pdf

[11] Dibachi,Azoulay (2021.), *Teaching a Robot to Walk Using Reinforcement Learning*
https://arxiv.org/pdf/2112.07031.pdf

[12] Momin Haider (2022.), *TD3 Bipedal Walker*
https://github.com/hmomin/TD3-Bipedal-Walker

[13] Dobal Byrne, *TD3: Learning To Run With AI*
https://towardsdatascience.com/td3-learning-to-run-with-ai-40dfc512f93

[14] Ziad Salloum, *Introduction to Augmented Random Search*
https://towardsdatascience.com/introduction-to-augmented-random-search-d8d7b55309bd

[15] Mania, Guy, Recht (2018.), *Simple random search provides a competitive approach to reinforcement learning*
https://arxiv.org/pdf/1803.07055.pdf

[16] Unnikrishnan Menon (2020.), *BipedalWalker*
https://github.com/7enTropy7/BipedalWalker

TEMELJNA DOKUMENTACIJSKA KARTICA

PRIRODOSLOVNO–MATEMATIČKI FAKULTET
SVEUČILIŠTA U SPLITU
ODJEL ZA MATEMATIKU

DIPLOMSKI RAD

PRIMJENA UČENJA PODRŠKOM NA PROBLEM DVONOŽNOG HODANJA

Paula Čaleta

Sažetak:

Cilj je ovog diplomskog rada predstaviti model slobodne metode učenja podrškom primjenjive na rješavanje problema neprekidne kontrole, točnije dvonožnog hodanja. U tu je svrhu prvo opisana struktura i proces učenja neuronskih mreža. Definiramo i osnovne koncepte učenja podrškom, vrste algoritama te dajemo uvid u BipedalWalker-v3 radno okruženje dvonožnog robota u kojem implementiramo algoritme. Zatim prezentiramo TD3 algoritam počevši od DDPG algoritma kao osnove istog te ARS algoritam. Konačno, opisujemo implementaciju u Pythonu i rezultate oba algoritma.

Ključne riječi:

Neuronske mreže, Q-funkcija, Bellmanova jednadžba, Glumac-kritičar metode, DDPG, TD3, ARS

Podatci o radu:

59 stranica, 13 slika, 16 literaturnih navoda, napisano na hrvatskom jeziku

Mentor: *doc. dr. sc. Ivo Ugrina*

Članovi povjerenstva:

doc. dr. sc. Tanja Vojković

mag. math. Domagoj Jelić

TEMELJNA DOKUMENTACIJSKA KARTICA

Povjerenstvo za diplomski rad je prihvatilo ovaj rad *7.7.2022*

TEMELJNA DOKUMENTACIJSKA KARTICA

FACULTY OF SCIENCE, UNIVERSITY OF SPLIT
DEPARTMENT OF MATHEMATICS

MASTER'S THESIS

APPLYING REINFORCEMENT LEARNING ON THE BIPEDAL WALKING PROBLEM

Paula Čaleta

Abstract:

The goal of this thesis is to introduce model free reinforcement learning methods applicable on solving continuous control problems, i.e. bipedal walking. For that purpose, the neural networks structure and learning process is first defined. We also define the basic concepts and algorithm types of reinforcement learning and give insight into the BipedalWalker-v3 environment in which we implement the algorithms. Subsequently, we present the TD3 algorithm (starting with the DPPG algorithm as its basis) and the ARS algorithm. Finally, we describe the Python implementation and results of both algorithms.

Key words:

neural networks, Q-function, Bellman equation, Actor-Critic methods, DDPG, TD3, ARS

Specifications:

59 pages, 13 pictures, 16 references, written in Croatian

Mentor: *Assistant professor Ivo Ugrina*

Committee:

Assistant professor Tanja Vojković

TEMELJNA DOKUMENTACIJSKA KARTICA

Domagoj Jelić, *mag. math.*

This thesis was approved by a Thesis committee on *7.7.2022.*