Primjena potpomognutog učenja na video igru Tekken 3

Iletić, Karlo

Sveučilište Josipa Jurja Strossmayera u Osijeku

Fakultet primijenjene matematike i informatike

Sveučilišni diplomski studij matematike
smjer: Matematika i računarstvo

Primjena potpomognutog učenja na video igru Tekken 3

Diplomski rad

Mentor:
Domagoj Ševerdija

Kandidat:
Karlo Iletić

Osijek, 2023

Sadržaj

1 | Uvod

U svijetu video igara pametni agenti služe kao dodatan izazov i zanimljivije iskustvo igranja za igrača. Osim samog igranja, video igre kroz vrijeme postaju odlično sredstvo za testiranje umjetne inteligencije jer pružaju kompleksne situacije i omogućuju potpuno sigurno okruženje u kojem je moguće bilo kakvo testiranje. Izazovi koje video igre pružaju često možemo povezati s izazovima iz pravog svijeta, pa razvoj pametnih agenata u ovom kontekstu doprinosi razvoju istraživanja umjetne inteligencije koje onda možemo primijeniti i na ostale domene.

Krajem 20. stoljeća počinju prvi pokušaji razvoja pametnih agenata u video igrama. Na početku su to bili sustavi temeljeni na predefiniranim pravilima koji odlučuju ponašanje agenta u igri. Kao primjer možemo uzeti duhove iz video igre *Pac-Man* (1980) koji hvataju igrača kada su dovoljno blizu, a kreću se nasumično kada su daleko. Nadalje, 1992. godine je razvijen program pod nazivom *TD-Gammon* koji je naučio igrati igru *Backgammon* dovoljno dobro da se rangira malo ispod najboljih ljudskih igrača tog vremena. Ovaj program je od iznimne važnosti jer je jedan od prvih primjena strojnog učenja i neuronskih mreža na igre. Unazad zadnjih 10 godina je potpomognuto učenje u kombinaciji s dubokim učenjem i raznim tehnikama imalo puno uspjeha. Za primjere možemo uzeti agenta koji postiže bolje rezultate od ljudskih eksperata na nekoliko *Atari* igara [10], agenta koji pobjeđuje najbolji tim u svijetu u video igri *Dota 2* [3] i tako dalje.

Potpomognuto učenje je oblik strojnog učenja koji se bazira na nagrađivanju poželjnih ponašanja, te kažnjavanju neželjenih. Agent potpomognutog učenja je dio okruženja kojeg je u stanju percipirati, te može donositi akcije koje imaju utjecaja na to okruženje. Akcija koju agent poduzme nosi nagradu koja je direktna posljedica promjene okruženja nakon izvršavanja te akcije. Na taj način agent uči optimalno ponašanje kroz pokušaje i pogreške.

Unazad zadnjih nekoliko godina potpomognuto učenje se pokazalo kao moćna tehnika za treniranje inteligentnih agenata pri donošenju optimalnih odluka u kompleksnim okruženjima. Algoritmi potpomognutog učenja su demonstrirali dobre rezultate u raznim domenama poput igranja društvenih igara [14], biologiji [8], samovozećim automobilima [18], robotici [9] kao i video igrama [16] [10] [1]. U ovom radu bavit ćemo se primjenom potpomognutog učenja na video igru pod nazivom *Tekken 3*.

U drugom poglavlju upoznat ćemo se s video igrom i načinom kontroliranja igrača.

U trećem poglavlju definirat ćemo proces potpomognutog učenja i vidjet ćemo

kako to možemo modelirati. Također, predložit ćemo nekoliko algoritama potpomognutog učenja koje ćemo kasnije i primijeniti.

U četvrtom poglavlju objasnit ćemo kako smo implementirali softversko sučelje koje koristimo za primjenu potpomognutog učenja na Tekken 3.

U zadnjem poglavlju ćemo vidjeti da je agent potpomognutog učenja koji koristi algoritam iz trećeg poglavlja bio relativno uspješan u učenju.

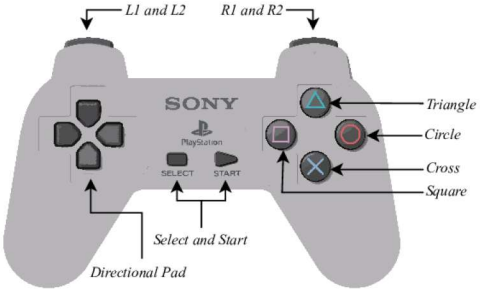
2 | Okruženje

Kako bi primijenili algoritme potpomognutog učenja bitno je da prvo razumijemo okruženje i načine na koji agent može utjecati na njega.

2.1 PlayStation

Sony PlayStation 1 (skraćeno PSX) je konzola video igara napravljena od strane *Sony Computer Entertainment* koja je izašla 1994. godine u Japanu, a 1995. godine u ostatku svijeta. PSX nudi raznoliku ponudu žanrova igara, od pucačina, akcijskih i borilačkih do RPG-ova, strateških i avanturističkih igara. Sveukupno je napravljeno oko 8000 igara za ovu konzolu [19]. Za igranje se koristimo kontrolerom koji ima sveukupno 14 tipki, prikazanih na Slici 2.1. Tako i agentima potpomognutog učenja akcije koje su dostupne odgovaraju tih 14 tipki, te razne kombinacije nekih od njih.

Unatoč velikoj popularnosti i velikom rasponu dostupnih igara, igre na ovoj konzoli dobivaju jako malo pozornosti iz perspektive potpomognutog učenja. Pogledamo li stanje istraživanja u ovoj domeni nije teško uočiti da je fokus više usmjeren na jednostavnije konzole poput *Atari 2600* (primjerice [10] [11] [7] i mnogi drugi). *M. G. Bellemare* i ostali [2] su 2012. godine predložili *Arcade Learning Environment* (*ALE*) koji pruža softversko sučelje za kontrolu i iščitavanje podataka iz *Atari 2600* igara, omogućujući brzu i laku primjenu potpomognutog učenja. Radovi koji obuhvaćaju PSX i potpomognuto učenje gotovo da ne postoje sve do 2019. godine, kada su *C. Purves* i ostali [13] predložili *PlayStation Reinforcement Learning Environment* (*PSXLE*) koji je nastao kao inspiracija od ALE. Ipak, u ovom radu se nećemo koristiti sa PSXLE, nego radimo svoju implementaciju softverskog sučelja.



Slika 2.1: PSX kontroler (slika preuzeta iz [13])

2.2 Tekken 3

Tekken je niz borilačkih video igara koje je razvio i objavio *Bandai Namco Entertainment*. Princip takvih igara je vrlo jednostavan – dva ili više lika sudjeluje u borbi. Borilačke igre obično sadržavaju mehanike poput blokiranja napada, protunapada, te ulančavanja napada u takozvane "komboe".

Trodimenzionalna kretnja je vrlo ograničena u prethodnim Tekken igrama, ali Tekken 3 dodaje naglasak na kretanje u trećoj osi dopuštajući svakom borcu takozvani "sidestep" gdje se pomakne ili prema ili od pozadine dajući osjećaj dubine. Sveukupno u igri postoji 23 borca od kojih su 10 dostupni za igranje od samog početka, a ostale je potrebno otključati. Tekken 3 je ponudio svojim igračima raznolik niz stilova borbe, gdje svaki borac ima svoj jedinstveni skup poteza i strategija. Kao posljedica, Tekken 3 pruža igračima nagrađujuće iskustvo koje zahtijeva i vještinu i precizan tajming.

Igra dolazi s 3 opcije težine - easy, medium i hard. Jedna od opcija igranja je *Arcade Mode* gdje kontroliramo borca protiv pretprogramiranog računala (*CPU*). Cilj je pobijediti 10 različitih boraca koje nam igra ponudi, a svaka borba se igra do 2 pobjede. Dakle, izgubimo li od nekog borca 2 puta to znači da je igra gotova, te je potrebno pobijediti 2 puta da bi prešli na sljedećeg borca. Osim *Arcade Mode* igra nudi puno drugih opcija o kojima se može detaljnije pročitati na [20].

Slika 2.2: Primjer borbe između 2 borca - King i Lei. Lijevi borac King je borac kojeg mi kontroliramo, a Lei je protivnički borac. Iznad svakog borca se nalazi traka koja indicira koliko životnih bodova je tom borcu preostalo. Traka će biti potpuno zelena ako borac ima sve životne bodove, a što više štete primi to će više postajati crvena. U sredini vidimo 27 što indicira da je preostalo 27 sekundi do kraja borbe.

2.2.1 Pravila

Nekoliko pravila postoji, od kojih navodimo samo 3 najbitnija:

- kada je borac uspješno ozlijeđen od strane protivnika njegovi životni poeni se smanjuju, a ako dostignu 0 tada borac gubi rundu.
- postoji vremensko ograničenje za svaku rundu. Ako vrijeme istekne borac koji posjeduje više životnih poena je pobjednik. Vremensko ograničenje je standardno naređeno na 40 sekundi, no ova vrijednost se može promijeniti u opcijama.
- neriješeno se događa u slučaju da oba borca u isto vrijeme dostignu 0 životnih bodova. Također, neriješeno je isto ishod ako vrijeme istekne a oboje imaju istu količinu životnih bodova.

2.2.2 Napadi i obrane

Tipke na desnoj strani kontrolera, koji su trokut, krug, kocka, iks - odgovaraju svakom od 4 uda borca i koriste se za obične napade.

- *kocka* - udarac lijevom rukom
- *iks* - udarac lijevom nogom
- *trokut* - udarac desnom rukom
- *krug* - udarac desnom nogom

Razne kombinacije gornjih napada s tipkama smjera (lijeva strana kontrolera) nam daju posebne napade. Kao što je već spomenuto svaki borac u igri ima svoj jedinstven skup takvih napada pa ako neka kombinacija radi na jednom borcu ne mora značiti da će raditi i na drugom.

Napadi su podijeljeni u tri kategorije ovisno o tome koji dio tijela neprijateljskog borca želimo pogoditi: visoki, srednji i niski raspon.

- napadi visokog raspona - pogađaju protivnika koji stoji. Promašit će protivnika koji čuči, a pogađa bez nanošene štete protivnika koji je u stojećem gardu.
- napadi srednjeg raspona - pogađaju protivnika koji stoji ili čuči. Pogađa bez nanošene štete protivnika koji je u stojećem gardu.
- napadi niskog raspona - pogađaju protivnika koji stoji ili čuči. Mogu se blokirati u čučećem gardu ili ako protivnik dobro tajmira skok pa preskoči napad.

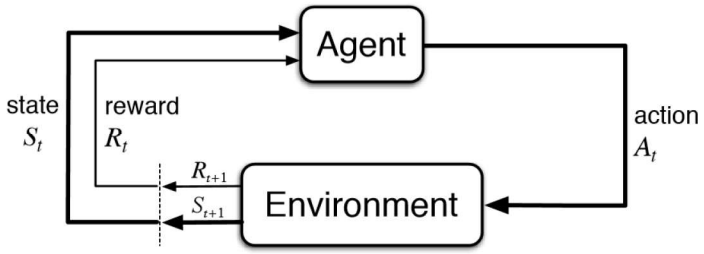
Gore spomenuti gardovi se postižu držanjem lijevom/desnom tipkom za stojeći gard, a dolje + lijevo/desno za čučeći gard.

Također, svaki borac može primijeniti bacanje protivnika na način da drži jednu od sljedeće dvije opcije tipki u isto vrijeme: + ili + . Bacanja ostavljaju borca

privremeno potpuno ranjivima, ali se od njih nije moguće zaštititi. Postoje tri bacanja koja dijele svi borci: prednje bacanje, lijevo/desno bacanje i stražnje bacanje, sve ovisno o tome gdje se nalazimo u odnosu na neprijatelja. Ako neprijatelj čuči ili je na zemlji, ne može se baciti.

3 | Potpomognuto učenje

Potpomognuto učenje možemo opisati kao proces interakcije agenta i okruženja. Agent odlučuje napraviti neku akciju koja posljedično mijenja okruženje, a promjena nosi sa sobom nagradu koja govori koliko je dobra ili loša ta promjena. Na taj način agent uči kako se ponašati u okruženju u kojem se nalazi. Taj proces se ponavlja sve dok agent ne nauči optimalno ponašanje u tom okruženju u smislu odabira akcije koja kroz vrijeme maksimizira nagradu.



Slika 3.1: Interakcija agenta i okruženja. Slika preuzeta iz [15]

Ovaj proces modeliramo pomoću Markovljevog procesa odlučivanja kojeg formalno definiramo u nastavku.

3.1 Markovljev proces odlučivanja

Definicija 1. ***Markovljev proces odlučivanja*** **(*MPO*)** *je četvorka* (S, A, P, R) *gdje:*

- S *predstavlja skup stanja*
- A *predstavlja skup akcija*
- $P(s' \mid a, s)$ *predstavlja vjerojatnost prelaska u stanje* s' *iz stanja* s *nakon izvršavanja akcije* a
- $R(s, a, s') \in \mathbb{R}$ *predstavlja nagradu promjene stanja iz* s *u* s' *nakon akcije* a

Agentovo ponašanje u okruženju opisujemo s funkcijom $\pi\colon S \to A$ koju nazivamo **strategija**. Dakle, ako se agent trenutno nalazi u stanju $s \in S$, tada strategija $\pi(s)$ govori koju akciju treba poduzeti. Rekli smo da agent pokušava naučiti kako se optimalno ponašati u okruženju, a to možemo formalizirati tako da pronalazimo strategiju koja maksimizira izraz

$$\pi^* = \arg\max_{\pi} \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t, s_{t+1}) \mid \pi\right]. \tag{3.1}$$

U gornjem izrazu se spominje $\gamma \in [0, 1]$ koji predstavlja koliko su agentu bitne nagrade dugoročno. Ako je $\gamma = 1$ tada je svaka od nagrada jednako bitna, a što je γ bliži 0 to su nagrade u budućnosti sve manje i manje bitne.

Kada govorimo o MPO u kontekstu nekog konačnog ponovljivog procesa tada razlikujemo posebna stanja $s_n \in S$ koja predstavljaju završna stanja iz kojeg nije moguće doći u nijedno drugo stanje. Niz tranzicija počevši iz $s_0 \in S$ u neko od završnih stanja $s_n \in S$ prateći strategiju π nazivamo **epizoda** koja traje $n \in \mathbb{N}$ koraka. U nastavku ćemo pretpostaviti da govorimo o MPO s konačnim ponovljivim procesom gdje iz bilo kojeg početnog stanja u konačan broj koraka dolazimo do završnog stanja.

3.2 Funkcija vrijednosti

Funkciju vrijednosti stanja $V_\pi \colon S \to \mathbb{R}$ definiramo kao očekivanu nagradu ako počnemo iz stanja $s \in S$ i pratimo strategiju π, odnosno

$$V_\pi(s) = \mathbb{E}\left[\sum_{t=0}^{n} \gamma^t R(s_t, a_t, s_{t+1}) \mid s_0 = s, \pi\right].$$

Slično, **funkciju vrijednosti akcije** $Q_\pi \colon S \times A \to \mathbb{R}$ definiramo kao očekivanu nagradu ako počnemo iz stanja s, poduzmemo akciju a, te nakon toga pratimo strategiju π, odnosno

$$Q_\pi(s, a) = \mathbb{E}\left[\sum_{t=0}^{n} \gamma^t R(s_t, a_t, s_{t+1}) \mid s_0 = s, a_0 = a, \pi\right].$$

Uočimo da onda vrijedi

$$\begin{aligned} V_\pi(s) &= \max_{a \in A} Q_\pi(s, a) \\ &= \max_{a \in A} \mathbb{E}\left[\sum_{t=0}^{n} \gamma^t R(s_t, a_t, s_{t+1}) \mid s_0 = s, a_0 = a, \pi\right] \\ &= \max_{a \in A} \mathbb{E}\left[R(s, a, s_1) + \sum_{t=1}^{n} \gamma^t R(s_t, a_t, s_{t+1}) \mid \pi\right] \\ &= \max_{a \in A} \mathbb{E}\left[R(s, a, s_1) + \gamma V_\pi(s_1)\right] \\ &= \max_{a \in A} \sum_{s' \in S} P(s' \mid s, a)(R(s, a, s') + \gamma V_\pi(s')). \end{aligned} \tag{3.2}$$

Sada imamo da vrijedi

$$Q_\pi(s,a) = \sum_{s' \in S} P(s' \mid s,a)(R(s,a,s') + \gamma V_\pi(s')).$$

Onda iz (3.2) imamo da vrijedi

$$Q_\pi(s,a) = \sum_{s' \in S} P(s' \mid s,a)(R(s,a,s') + \gamma \max_{a' \in A} Q_\pi(s',a')).$$

Za agenta je korisnost funkcije vrijednosti akcije (koju još i nazivamo **Q-vrijednost**) u tome što može zaključiti optimalnu strategiju kao

$$\pi(s) = arg\max_{a \in A} Q(s,a).$$

3.3 Tablično Q-učenje

S obzirom na to da je Q-vrijednost rekurzivna relacija, možemo definirati iterativni postupak na sljedeći način

$$Q_{t+1}(s,a) = \sum_{s' \in S} P(s' \mid s,a)(R(s,a,s') + \gamma \max_{a' \in A} Q_t(s',a')). \tag{3.3}$$

No, problem iza ovog načina učenja Q-vrijednosti je taj da moramo znati model okruženja $P(s' \mid s,a)$, što je jako velika pretpostavka. Ako nam model okruženja nije dostupan, problem pokušavamo riješiti na način da (3.3) ipak vratimo u stari oblik očekivanja

$$Q_{t+1}(s,a) = \mathbb{E}_{s' \sim P(s'|s,a)} \left[R(s,a,s') + \gamma \max_{a' \in A} Q_t(s',a') \right]. \tag{3.4}$$

Onda to možemo aproksimirati uzorkovanjem jednog novog stanja $s' \sim P(s' \mid s,a)$. U praktičnom smislu, ovo znači da se agent u okruženju trenutno nalazi u stanju s, napravio je akciju a, što je uzrokovalo prelazak u stanje s' sa nagradom $R(s,a,s')$ koje dobivamo nazad iz okoline. Prethodnu tranziciju $(s,a,s',R(s,a,s'))$ nazivamo **iskustvo**. Tada aproksimaciju Q-vrijednosti možemo napisati kao

$$Q_{t+1}(s,a) = R(s,a,s') + \gamma \max_{a' \in A} Q_t(s',a').$$

Ipak, u kompleksnijim okruženjima jedno uzorkovanje može biti nedovoljno. Pokušaj bolje aproksimacije radimo na način da pamtimo staru Q-vrijednost koju samo djelomično ažuriramo novim iskustvom

$$Q_{t+1}(s,a) = (1-\alpha)Q_t(s,a) + \alpha\left[R(s,a,s') + \gamma \max_{a' \in A} Q_t(s',a')\right],\ \alpha \in (0,1). \quad (3.5)$$

Intuicija iza toga je da se kroz vrijeme sakupljaju nova iskustva s kojima dobivamo bolji dojam o tome kako akcije utječu na okruženje, te koje akcije su bolje od drugih. Sa svakim novim iskustvom dobivamo sve bolju i bolju aproksimaciju očekivanja iz (3.4).

No, preostaje problem skaliranja. Iz (3.5) vidimo da ažuriramo Q-vrijednost za par (s,a). To znači da moramo napraviti tablicu veličine $|S| \times |A|$, što za bilo koje kompleksnije okruženje nije moguće napraviti. Uzmemo li za primjer PSX gdje stanje reprezentiramo kao stanje radne memorije, to bi značilo da postoji sveukupno $2^{16000000}$ stanja jer je količina radne memorije sveukupno 2 megabajta.

3.4 Duboko Q-učenje

Umjesto tablice možemo reprezentirati Q-funkciju s parametrom θ. Kada dobijemo novo iskustvo $(s,a,s',R(s,a,s'))$, umjesto direktnog ažuriranja Q-vrijednosti za par (s,a), možemo ažurirati samo θ i tako reprezentirati drugu Q-vrijednost koja ima različite aproksimacije za (s,a) parove od prethodne aproksimacije. Ako je θ neuronska mreža tada govorimo o **dubokom Q-učenju**, a Q-funkciju nazivamo **Q-mreža**. Definirajmo funkciju gubitka u k-tom koraku kao

$$\mathcal{L}(s';\theta_k) = \begin{cases} (R(s,a,s') - Q(s,a;\theta_k))^2, \text{ako je } s' \text{ završno stanje,} \\ ((R(s,a,s') + \gamma \max_{a' \in A} Q(s',a';\theta_k)) - Q(s,a;\theta_k))^2, \text{inače.} \end{cases}$$

Ideja iza toga je da usporedimo trenutnu aproksimaciju očekivane nagrade ako iz stanja s napravimo akciju a, odnosno $Q(s,a;\theta_k)$, s točnijom aproksimacijom gdje znamo nagradu u sljedećem koraku, no još uvijek nagađamo očekivanu nagradu budućnosti iz sada novog stanja s', odnosno $R(s,a,s') + \gamma \max_{a' \in A} Q(s',a';\theta_k)$. Naravno, htjeli bismo pronaći takav θ da su razlike između tih vrijednosti što manje, a to znači da proces učenja θ možemo s pomoću gradijentnog spusta jednostavno zapisati kao

$$\theta_{k+1} = \theta_k - \nabla_{\theta_k} \mathcal{L}(s';\theta_k).$$

Kompletan izraz od $\nabla_{\theta_k} \mathcal{L}(s';\theta_k)$ nećemo raspisivati jer se to u praksi računa algoritmima automatskog deriviranja.

Prije nego li napišemo formalni algoritam dubokog Q-učenja podsjetimo se kako cijeli proces učenja funkcionira. Kao prvo, cilj nam je da se agent ponaša optimalno u okolini u kojoj se nalazi. U kontekstu dubokog Q-učenja agent je potpuno opisan s parametrom θ. Kada se agent nalazi u nekom stanju s on će tada izabrati akciju za koju misli da će donijeti najviše nagrade dugoročno, odnosno $\text{argmax}_{a \in A} Q(s,a;\theta)$. Nakon što se akcija izvrši dobivamo nagradu i na temelju

nagrade popravljamo aproksimaciju koliko je akcija koju je agent izabrao zapravo bila dobra.

Kada bi agent uvijek izabirao akciju na ovaj način to bi potencijalno moglo rezultirati u loše aproksimacije jer se može desiti da ne isproba neke akcije koje bi mu mogle donijeti više nagrade. Ono što moramo ugraditi u našeg agenta je proces istraživanja, odnosno ne dopustiti mu da uvijek prati što misli da je najbolje. Zato, agentov odabir akcija tijekom učenja opisujemo s ϵ_{greedy} funkcijom:

$$\epsilon_{greedy}(s, \theta, \varepsilon) = \begin{cases} \text{nasumična akcija } a \in A, & \text{sa vjerojatnosti } \varepsilon, \\ \text{argmax}_{a \in A}\, Q(s, a; \theta), & \text{inače.} \end{cases} \tag{3.6}$$

U praksi bi epsilon na početku stavili da bude blizu 1 i onda kroz vrijeme ga smanjivati do neke određene manje vrijednosti. Sada proces učenja dajemo kao formalni algoritam.

Algoritam 1 Duboko Q-učenje

1: Inicijaliziraj Q-mrežu θ s nasumičnim vrijednostima
2: Inicijaliziraj početno stanje s ▷ Ovo može biti dano algoritmu
3: Inicijaliziraj broj koraka učenja K ▷ Ovo može biti dano algoritmu
4: Inicijaliziraj $0 \leq \varepsilon \leq 1$ ▷ Ovo može biti dano algoritmu
5: **for** $k := 1, \ldots, K$ **do**
6: $\quad a := \epsilon_{greedy}(s, \theta, \varepsilon)$
7: $\quad$ Pošalji akciju a okolini koja vraća novo stanje s' i nagradu $R(s, a, s')$
8: $\quad$ Izračunaj $\nabla_\theta \mathcal{L}(s'; \theta)$ i ažuriraj $\theta := \theta - \nabla_\theta \mathcal{L}(s'; \theta)$
9: $\quad$ Ažuriraj stanje $s := s'$

Postoji nekoliko potencijalnih problema s gornjim algoritmom koje ćemo u nastavku opisati i predložiti rješenja.

3.4.1 Ciljna mreža

Jedna od stvari koja je problematična je način na koji ažuriramo našu Q-mrežu. Neuronske mreže, u svrhu generalizacije, daju slične izlaze za slične ulaze. To znači da za razliku od tabličnog Q-učenja, kada ažuriramo aproksimaciju u dubokom Q-učenju nismo samo promijenili aproksimaciju za trenutno stanje s i izabranu akciju a, nego i za sva slična stanja. Ako je okruženje takvo da akcije ne mogu na drastične načine promijeniti stanje, što je većinom i slučaj, tada iz trenutnog stanja s idemo u njemu slično stanje s'. Recimo da je nagrada pozitivna, tada zbog načina ažuriranja idemo u sljedeće stanje gdje će aproksimacije za to stanje biti veće zbog prirode neuronskih mreža, pa će opet aproksimacije očekivane buduće nagrade isto biti veće. Ovo nas može dovesti u situaciju gdje mreža pokušava pratiti sve veće i veće aproksimacije koje je teško popraviti i potencijalno stvara učenje puno nestabilnijim, a u najgorem slučaju moguća je i divergencija.

Kao rješenje *V. Mnih* i ostali [11] su predložili dodavanje dodatne, takozvane ciljne mreže. Dodavanjem učenje postaje stabilnijim, a ideja je da fiksiramo aproksimaciju očekivane buduće nagrade i ažuriramo je s trenutnom aproksimacijom

svakih nekoliko koraka (hiperparametar). Recimo da je nagrada pozitivna, tada zbog načina ažuriranja idemo u sljedeće stanje gdje će aproksimacije za to stanje biti veće zbog prirode neuronskih mreža, ali aproksimacije očekivane buduće nagrade se neće promijeniti jer koristimo fiksiranu mrežu koju još nismo ažurirali.

Ciljnu mrežu označimo s $\bar{\theta}$, te definirajmo funkciju gubitka kao

$$\mathcal{L}(s';\theta_k,\bar{\theta}_k) = \begin{cases} \left(R(s,a,s') - Q(s,a;\theta_k)\right)^2, \text{ako je } s' \text{ završno stanje,} \\ \left(\left(R(s,a,s') + \gamma \max_{a' \in A} Q(s',a';\bar{\theta}_k)\right) - Q(s,a;\theta_k)\right)^2, \text{inače.} \end{cases}$$

Tada duboko Q-učenje s ciljnom mrežom izgleda na sljedeći način.

Algoritam 2 Duboko Q-učenje s ciljnom mrežom

Inicijaliziraj glavnu Q-mrežu θ s nasumičnim vrijednostima
Inicijaliziraj ciljnu Q-mrežu $\bar{\theta} = \theta$
Inicijaliziraj početno stanje s ▷ Ovo može biti dano algoritmu
Inicijaliziraj broj koraka učenja K ▷ Ovo može biti dano algoritmu
Inicijaliziraj $0 \leq \varepsilon \leq 1$ ▷ Ovo može biti dano algoritmu
Inicijaliziraj frekvenciju ažuriranja ciljne mreže t ▷ Ovo može biti dano algoritmu
for $k := 1, \ldots, K$ **do**
    $a := \epsilon_{greedy}(s, \theta, \varepsilon)$
    Pošalji akciju a okolini koja vraća novo stanje s' i nagradu $R(s,a,s')$
    Izračunaj $\nabla_\theta \mathcal{L}(s';\theta,\bar{\theta})$ i ažuriraj $\theta := \theta - \nabla_\theta \mathcal{L}(s';\theta,\bar{\theta})$
    Ažuriraj stanje $s := s'$
    if $k \equiv 0 \mod t$ **then**
        Ažuriraj ciljnu mrežu $\bar{\theta} := \theta$

3.4.2 Pamćenje iskustva

Učenje iz uzastopnih iskustava je neučinkovito zbog velike korelacije između njih. Gradijent može biti previše usmjeren u jednu stranu jer trenutna trajektorija nije reprezentativna cijele distribucije. Na primjer, ako je optimalna akcija da se agent pomakne lijevo u nekom okruženju, tada će iskustva biti dominirana samo s okruženjem na lijevoj strani.

Način na koji su *V. Mnih* i ostali [10] riješili taj problem je s pamćenjem prijašnjih iskustva i ponovnim učenjem na njima. Ideja je da imamo neki spremnik u koji dodajemo naša iskustva kroz vrijeme, i kada želimo učiti onda uzorkujemo uniformno nasumično jedan podskup iskustava iz tog spremnika i napravimo gradijentni spust na njima. Na ovaj način podskup iskustava je puno bliži stvarnoj distribuciji pa je učenje stabilnije. Duboko Q-učenje s ciljnom mrežom i pamćenjem iskustva možemo vidjeti u Algoritmu 3.

Algoritam 3 Duboko Q-učenje s ciljnom mrežom i pamćenjem iskustva

Inicijaliziraj glavnu Q-mrežu θ s nasumičnim vrijednostima
Inicijaliziraj ciljnu Q-mrežu $\bar{\theta} = \theta$
Inicijaliziraj početno stanje s ▷ Ovo može biti dano algoritmu
Inicijaliziraj broj koraka učenja K ▷ Ovo može biti dano algoritmu
Inicijaliziraj $0 \leq \varepsilon \leq 1$ ▷ Ovo može biti dano algoritmu
Inicijaliziraj frekvenciju ažuriranja ciljne mreže t ▷ Ovo može biti dano algoritmu
Inicijaliziraj prazni spremnik iskustava M
for $k := 1, \ldots, K$ **do**
    $a := \epsilon_{greedy}(s, \theta, \varepsilon)$
    Pošalji akciju a okolini koja vraća novo stanje s' i nagradu $R(s, a, s')$
    Dodaj iskustvo $(s, a, s', R(s, a, s'))$ u spremnik iskustava M
    Nasumično uzorkuj m iskustava iz M
    Za svako iskustvo $(s^{(i)}, a^{(i)}, s'^{(i)}, R(s^{(i)}, a^{(i)}, s'^{(i)}))$, $i = 1, \ldots, m$ izračunaj $\nabla_\theta \mathcal{L}(s'^{(i)}; \theta, \bar{\theta})$ i ažuriraj $\theta := \theta - \nabla_\theta \mathcal{L}(s'^{(i)}; \theta, \bar{\theta})$
    Ažuriraj stanje $s := s'$
    if $k \equiv 0 \mod t$ **then**
        Ažuriraj ciljnu mrežu $\bar{\theta} := \theta$

3.5 Duplo duboko Q-učenje

U praksi se pokazalo da Algoritam 3 i Q-učenje generalno imaju problem pristranosti precjenjivanja akcija. Na Q-vrijednosti u dubokom Q-učenju možemo gledati kao da predstavljaju prave Q-vrijednosti $\pm$ neki dodatak, gdje očito dodatak pokušavamo smanjiti s učenjem. Tada zbog *max* operatora ćemo vrlo vjerojatno precijeniti koliko neka akcija zapravo vrijedi. Valja napomenuti da je to zapravo otvoren problem je li precjenjivanje zapravo loše za mrežu. Kada bi se svaka akcija precijenila otprilike na isti način, tada bi relativni poredak između akcija ostao isti pa ne bi bilo nikakvih problema. Ipak, u praksi se pokazalo da pokušaji smanjivanja precjenjivanja nose u konačnici bolje rezultate.

Način na koji su *H. V. Hasselt* i ostali [6] riješili ovaj problem je tako što su odvojili odabir i aproksimaciju akcije. Ideja je da glavnu Q-mrežu θ pitamo koja je akcija najbolja, a onda za izabranu akciju pitamo ciljnu Q-mrežu $\bar{\theta}$ da aproksimira koliko je zapravo dobra ta akcija. Zamjenu matematički zapisujemo kao

$$\max_{a' \in A} Q(s', a'; \bar{\theta}_k) \quad \rightarrow \quad Q(s', \arg\max_{a' \in A} Q(s', a'; \theta_k); \bar{\theta}_k).$$

Funkcija gubitka tada izgleda kao

$$\mathcal{L}_{\text{double}}(s'; \theta_k, \bar{\theta}_k) = \begin{cases} (R(s, a, s') - Q(s, a; \theta_k))^2, & \text{ako je } s' \text{ završno stanje,} \\ ((R(s, a, s') + \gamma Q(s', \arg\max_{a' \in A} Q(s', a'; \theta_k); \bar{\theta}_k) \\ \quad - Q(s, a; \theta_k)))^2, & \text{inače.} \end{cases}$$

Algoritam 4 Duplo duboko Q-učenje s ciljnom mrežom i pamćenjem iskustva

Inicijaliziraj glavnu Q-mrežu θ s nasumičnim vrijednostima
Inicijaliziraj ciljnu Q-mrežu $\bar{\theta} = \theta$
Inicijaliziraj početno stanje s ▷ Ovo može biti dano algoritmu
Inicijaliziraj broj koraka učenja K ▷ Ovo može biti dano algoritmu
Inicijaliziraj $0 \leq \varepsilon \leq 1$ ▷ Ovo može biti dano algoritmu
Inicijaliziraj frekvenciju ažuriranja ciljne mreže t ▷ Ovo može biti dano algoritmu
Inicijaliziraj prazni spremnik iskustava M
for $k := 1, \ldots, K$ **do**
    $a := \epsilon_{greedy}(s, \theta, \varepsilon)$
    Pošalji akciju a okolini koja vraća novo stanje s' i nagradu $R(s, a, s')$
    Dodaj iskustvo $(s, a, s', R(s, a, s'))$ u spremnik iskustava M
    Nasumično uzorkuj m iskustava iz M
    Za svako iskustvo $(s^{(i)}, a^{(i)}, s'^{(i)}, R(s^{(i)}, a^{(i)}, s'^{(i)}))$, $i = 1, \ldots, m$ izračunaj $\nabla_\theta \mathcal{L}_{\text{double}}(s'^{(i)}; \theta, \bar{\theta})$ i ažuriraj $\theta := \theta - \nabla_\theta \mathcal{L}_{\text{double}}(s'^{(i)}; \theta, \bar{\theta})$
    Ažuriraj stanje $s := s'$
    if $k \equiv 0 \mod t$ **then**
        Ažuriraj ciljnu mrežu $\bar{\theta} := \theta$

4 | Implementacija

Kako bi primijenili potpomognuto učenje na PSX igre, potrebno je implementirati programsko sučelje s kojim možemo slati akcije koje se reflektiraju u video igri, te dobivati povratnu informaciju dok se igra izvodi u pravom vremenu. Kao prvo, potrebno je moći emulirati video igru. Srećom, postoji mnoštvo implementacija emulatora koji uspješno emuliraju PSX igre pa ovaj, nimalo trivijalan pothvat, ne moramo iznova implementirati. Pomoću emulatora kojeg izaberemo bismo morali moći kontrolirati PSX kontroler, te iščitavati informacije iz igre direktno iz memorije, kao i dobiti sliku ekrana. Nakon malo istraživanja odlučili smo se za *PCSX-Redux* kao odabir emulatora [1].

PCSX-Redux [21] je emulator otvorenog izvornog koda na kojem je 2018. godine počeo rad, a koji prima ažuriranja i nadogradnje i dan danas. Dva su glavna razloga iza odluke o odabiru baš ovog emulatora.

Kao prvo, ima ugrađen debugger s nekoliko mogućnosti koji su nama u interesu. Moguće je pratiti stanje registara, gledati assembly kod igre, stavljati breakpoint na izvršavanje linije ili ako se neka memorijska adresa čita ili piše, gledati i mijenjati stanje memorije, te mnoštvo drugih opcija koje nećemo nabrajati jer ih ima previše. Često su nam potrebne neke informacije iz igre da bi odredili stanje ili funkciju nagrade. Za primjere možemo uzeti životne bodove igrača, pozicije entiteta u igri, je li igra pauzirana itd. Svakako, neke informacije možda možemo iščitati s preslike ekrana pomoću nekih OCR tehnika, no one su nepouzdane i nepraktične. Sve ove opcije koje nam ugrađeni debugger pruža nam mogu pomoći u prikupljanju informacija koje nam trebaju.

Kao drugo, postoji softversko sučelje za kontrolu emulatora napisanog u skriptnom programskom jeziku Lua s opširnom dokumentacijom [2]. S obzirom na jednostavnost i lakoću korištenja Lua jezika, te odličnoj dokumentaciji, razvoj našeg projekta je ubrzan.

Neuronske mreže koje algoritmi iz ovog rada koriste implementiramo pomoću *PyTorcha*. To znači da će agent i emulator biti potpuno odvojeni pa će biti potrebno postaviti komunikaciju između njih.

[1] Emulator dostupan preko sličnog projekta: https://github.com/NDR008/TensorFlowPSX

[2] Dokumentacija se može pronaći na https://pcsx-redux.consoledev.net/Lua/introduction/

4.1 Minimalan primjer

U nastavku možemo vidjeti minimalan primjer korištenja razvijenog programskog sučelja. Dizajn je uvelike inspiriran od OpenAI-ovog Gyma [4] zbog elegantnosti i jednostavnosti za korištenje.

```
from env import Environment
from agent import RandomAgent

env = Environment()
agent = RandomAgent()

for _ in range(100):
  state, info = env.reset(1)
  done = False
  while not done:
    action = agent.generate_action(state)
    new_state, reward, done, info = env.step(action)
    loss = agent.train(state, action, reward, new_state, done, info
    )
    state = new_state
```

U gornjem primjeru smo kreirali agenta koji se bori protiv borca Law na najlakšoj težini igre i sveukupno igraju 100 mečeva. Agent o kojem se radi je *RandomAgent* što znači da šalje samo nasumične akcije i ne uči ništa (iako pozivamo `train` funkciju u njoj se ništa ne dešava ako se radi o RandomAgentu). U nastavku ćemo malo detaljnije pojasniti što se tu točno događa.

4.2 Akcije

Sa Slike (2.1) možemo vidjeti da sveukupno postoji 14 mogućih tipki na kontroleru. Izbacit ćemo select, start, L1, L2, R1, R2 jer nemaju nikakvog utjecaja na borca u našoj igri. Sada nam preostaje 8 tipki — 4 za kretnju i 4 za udarce. Mogli bismo dopustiti agentu da šalje jednu po jednu od tih 8 tipki, no na taj način ne bi mogli napraviti jako velik broj poteza u igri koji zahtijevaju da se dvije tipke odjednom pritisnu. Kao primjer smo mogli vidjeti u Poglavlju 2.2.2 da bacanje zahtijeva dvije tipke odjednom. Stoga, agentu ćemo dati mogućnost kombiniranja nekih od 8 dostupnih tipki. Kada izbacimo kombinacije koje nemaju smisla — gore + dolje i desno + lijevo, preostaje nam 34 mogućih akcija. Uz to dodajemo još jednu akciju koja predstavlja da agent ne napravi ništa što nas dovodi sveukupno do 35 akcija.

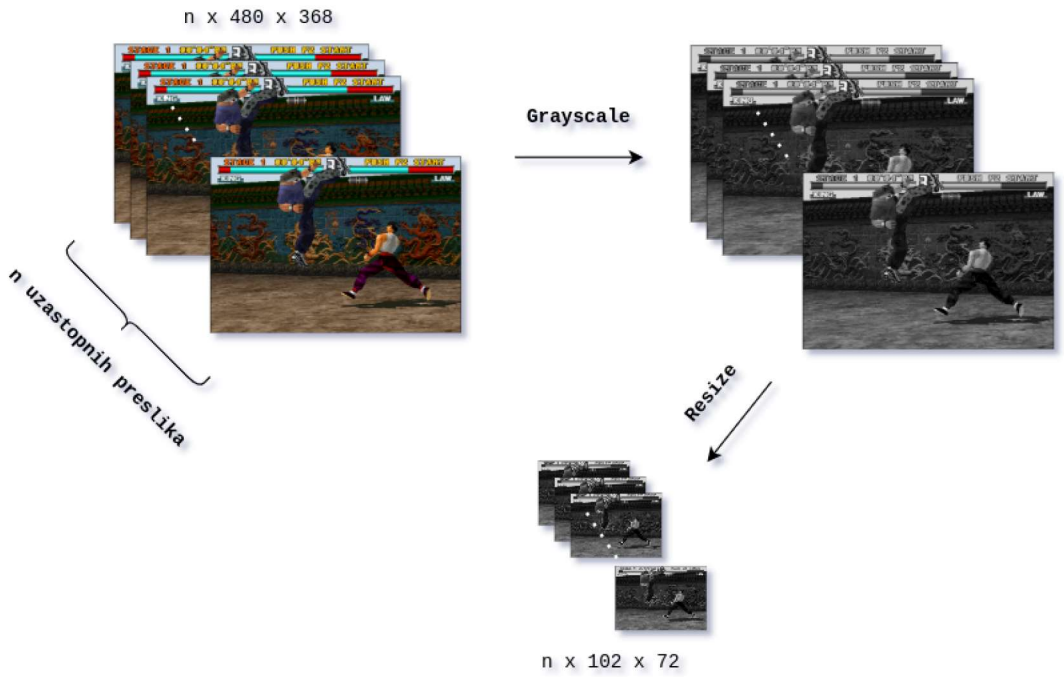
Sve vezano za akcije je implementirano u `action.py` (vidi poveznicu) gdje implementiramo funkciju za generiranje nasumične akcije i dodatnu mogućnost zabrane pojedinih tipki na kontroleru ili akcija.

4.3 Stanje

Stanje reprezentiramo kao uzastopni niz okvira (engl. frame) igre. Vrijedi napomenuti da postoje drugi načini reprezentiranja stanja poput nekih varijabli iz igre kao što su pozicije igrača, njihovi životni poeni, vektor smjera kretnje, udaljenost od rubova ekrana i tako dalje. Postoje radovi koji pokazuju da slični pokušaji nose dobre rezultate [5] [12]. Ipak, odlučujemo se za okvire jer na ovaj način agent i ljudska osoba imaju potpuno isti način percepcije igre, pa je situaciju možda ipak malo zanimljivija.

Umjesto da stanje bude samo jedan okvir uzet ćemo više uzastopnih. Iako je jedan okvir dovoljan da razumijemo pozicije igrača, nije dovoljan da razumijemo njihove kretnje. Ne znamo u kojem smjeru se kreću, jesu li u procesu pravljenja poteza i tako dalje.

Uzastopni niz okvira pretvaramo u stanje na način da svaki okvir pretvorimo iz *RGB* u *grayscale*, zatim smanjimo im veličinu s 480×368 u 102×72 i onda ih naslažemo u jedan tenzor dimenzija $n \times 102 \times 72$ kojeg možemo dati našoj Q-mreži kao input.

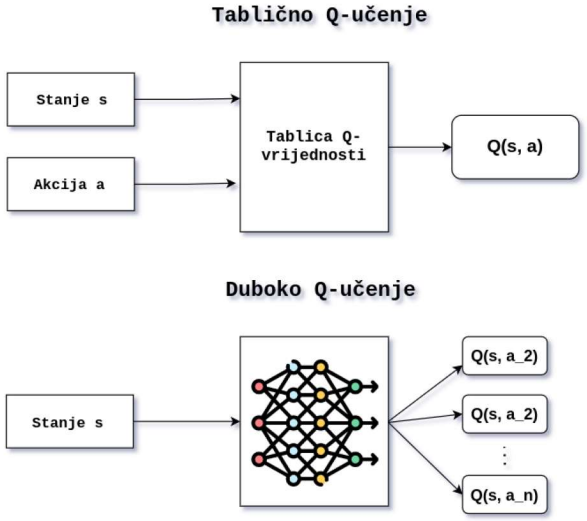


Slika 4.1: Proces pretvorbe niza od n okvira u stanje.

4.4 Mreža

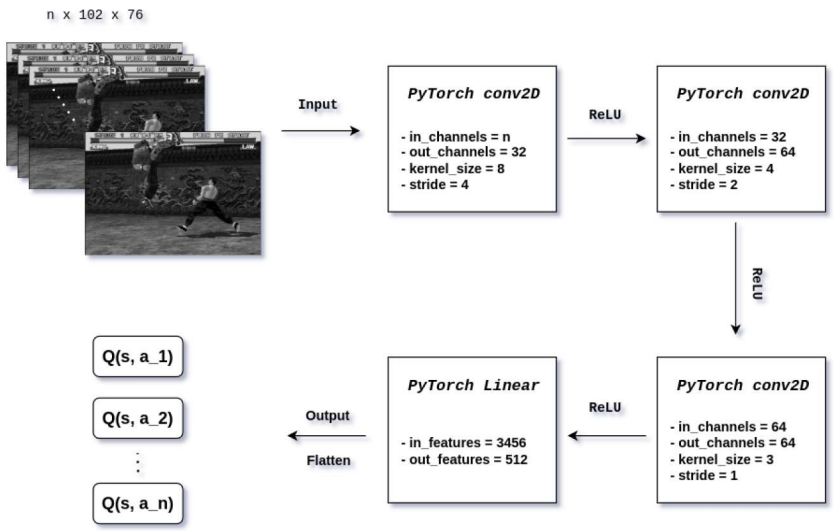
Jedan bitan tehnički detalj koji se tiče implementacije duboke Q-mreže je taj da ulaz mreže nije par stanje i akcije (s, a) te izlaz odgovarajuća Q-vrijednost $Q(s, a; \theta)$,

nego je ulaz samo stanje s, a izlaz su Q-vrijednosti za svaku akciju. Razlog iza ove odluke je taj što bi prilikom odabira najbolje akcije za trenutno stanje inače morali napraviti onoliko prolaza unaprijed (engl. forward pass) kroz mrežu koliko je i akcija. Puno efikasnije je napraviti jedan prolaz unaprijed pa proći kroz cijeli output i uzeti akciju kojoj odgovara najveća Q-vrijednost.



Slika 4.2: Razlika u implementaciji Q-vrijednosti između tabličnog i dubokog Q-učenja.

Arhitektura Q-mreže koju koristimo je direktno preuzeta iz [10]. Jedine razlike su što oni fiksiraju broj naslaganih okvira na 4, dok mi koristimo različite vrijednosti koje ćemo vidjeti u sljedećem poglavlju. Druga razlika je što su kod njih okviri dimenzija 84×84, a naši su 102×76.



Slika 4.3: Tok prolaza unaprijed Q-mreže.

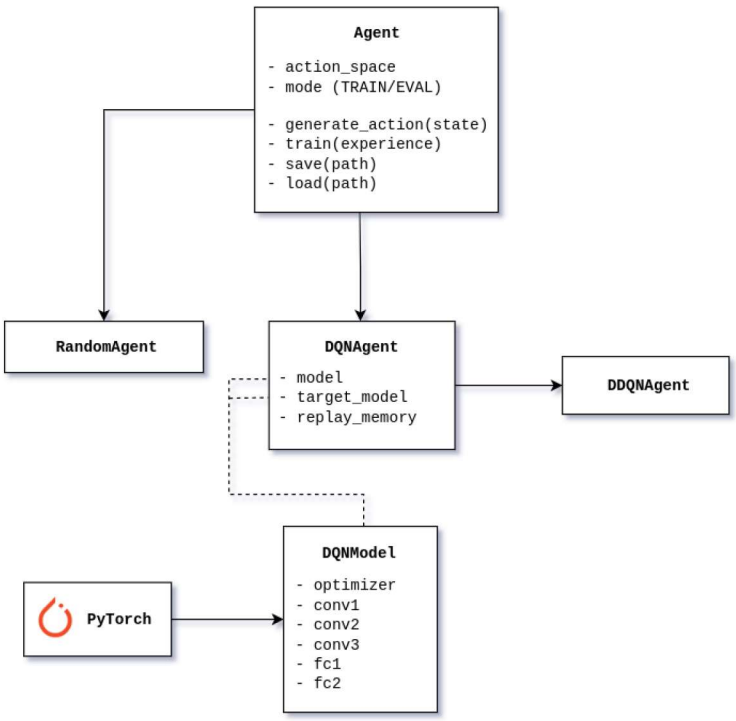
4.5 Agent

Agente implementiramo na način da definiramo baznu klasu koju pojedine implementacije agenata nasljeđuju. Bazna klasa sadrži generalne stvari koje bi u principu svaki agent trebao imati poput pristupa mogućim akcijama, funkciji za generiranje akcije, spremanju i učitavanju. Još jedna bitna stvar koju ima je stanje koje određuje da li se agent nalazi u evaluacijskom stanju ili stanju treniranja. Ovo je bitno jer prilikom evaluacije agenta ne želimo koristiti ϵ-pohlepno uzorkovanje akcija kao što je slučaj kod treniranja nego potpuno pohlepno.

Implementiramo tri vrste agenta:

- *RandomAgent* - agent koji šalje samo nasumične poteze
- *DQNAgent* - agent koji uči na temelju Algoritma 3
- *DDQNAgent* - agent koji uči na temelju Algoritma 4

Svaki od navedenih agenata implementira funkcije `train` i `generate_action`. Funkcija `generate_action` prima stanje i vraća izabranu akciju na temelju tog stanja. Tijekom treniranja ova funkcija je implementirana kao ϵ-pohlepna strategija (3.6). Tijekom evaluacije se koristi pohlepna strategija što za DQNAgenta i DDQNAgenta znači jedan prolaz unaprijed kroz DQNMrežu gdje je input mreži dano stanje, a iz outputa mreže odabiremo akciju koja pripada najvećoj vrijednosti. Sve vezano za agenta je implementirano u `agent.py` (vidi poveznicu).



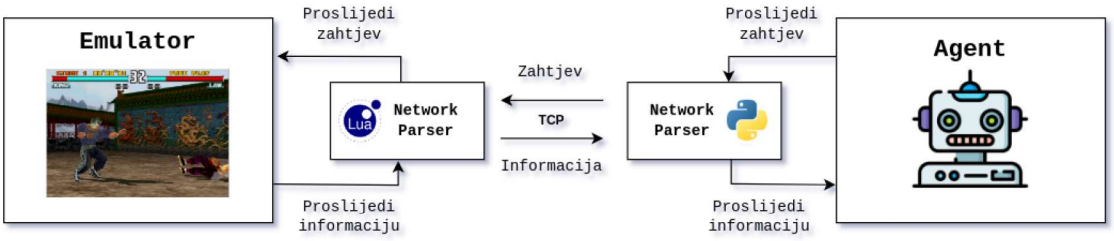
Slika 4.4: Pojednostavljen prikaz agenata.

4.6 Komunikacija između agenta i emulatora

Za postizanje interakcije agenta i igre očito je da nam je potrebna neka vrsta međuprocesne komunikacije. Za ovaj zadatak postoji puno standardnih rješenja koje možemo odabrati, a odlučili smo se na komunikaciju slanjem poruka preko mreže koristeći *Transmission Control Protocol* (TCP). Korištenjem protokola TCP, proces na nekom poslužitelju uključenom u računalnu mrežu stvara virtualnu vezu prema drugom procesu, te putem te ostvarene veze prenosi podatke.

Za slanje i primanje poruka smo se u Pythonu koristili s bibliotekom *socket*, a u emulatoru smo koristili *File* API koji u pozadini koristi biblioteku *luvit*.

Iz emulatora sve informacije koje agenta zanimaju se dobivaju isključivo na zahtjev. Kada agent odluči poduzeti akciju, mora zatražiti da se ta akcije pošalje do emulatora i onda ponovno zatražiti da dobije nazad novo stanje i nagradu.



Slika 4.5: Pojednostavljen prikaz interakcije agenta i emulatora.

4.7 Okruženje

Klasa *Environment* nam služi kao posrednik između agenta i emulatora. Instanciranjem ove klase otvaramo vezu sa strane Pythona na koji se jedan emulator tada može spojiti. Agent kada izabere akciju bira na koji emulator je želi poslati preko specifične instance Environment klase. Na taj način možemo imati agenta koji upravlja s više emulatora od jednom. Environment klasa pruža dvije glavne funkcije — `reset` i `step`.

Pozivom na funkciju `reset` resetiramo emulator na neko predefinirano stanje. Ovo postižemo tako što koristimo mogućnost emulatora da spremi i učita stanje cijele memorije. `reset` funkcija prima numeričku varijablu koja odgovara određenom spremljenom stanju na emulatoru. Tako možemo birati početna stanja proizvoljno, na primjer jednog agenta možemo trenirati na laganoj težini igre pozivajući `env.reset(x)`, a drugog agenta možemo trenirati na srednjoj težini igre ili protiv potpuno drugog protivnika pozivajući `env.reset(y)`. Funkcija `reset` vraća novo stanje i neke informacije o novom stanju u formatu koji je pogodan za ljude (za detalje pogledati poveznicu).

Pozivom na funkciju `step` šaljemo akciju u emulator i dobivamo nazad novo stanje. Zajedno sa stanjem, `step` funkcija vraća i odgovarajuću nagradu koja je rezultat poslane akcije. Kako bi agent bio više fer pokušali smo mu ugraditi efekt

inherentnog kašnjenja ljudske reakcije. Zato, trenutna akcija koja se šalje na emulator zapravo pripada stanju koji kasni sveukupno n okvira od trenutnog stanja igre. Pretpostavimo li da igramo na 60 fps i uzmemo li $n = 15$ tada agent kasni 250 ms sa svojim akcijama, što je otprilike i kašnjenje ljudske reakcije. Također, moguće je birati kojeg igrača u igri želimo kontrolirati s trenutnom akcijom što omogućuje da postavimo igranje agenta protiv agenta. Za detalje pogledati poveznicu.

4.8 Čitanje informacija iz memorije

Memory observer je alat koji je korišten za pronalazak memorijskih adresa varijabli iz igre. Ovaj alat je jedan od mnogih koji dolazi kao dio PCSX-Redux debuggera i pruža mogućnost pronalaska svih memorijskih adresa u memoriji emulirane igre koje su jednaki traženoj vrijednosti. Uz to, još je i moguće filtriranje pronađenih memorijskih adresa s obzirom na neki uvjet. S ovim alatom cijeli proces zapravo postaje vrlo jednostavan i opisat ćemo ga ukratko na primjeru pronalaska životnih bodova prvog igrača.

Poznata je činjenica da je maksimum životnih bodova igrača u Tekken 3 jednak 130. S tom informacijom možemo ući u borbu i prije nego primimo ikakvu štetu pauzirati igru i pronaći sve adrese s vrijednosti 130. Lista pronađenih adresa je obično jako dugačka s obzirom na to da je količina memorije za PSX jednaka 2 megabajta [17], pa proces nastavljamo tako što nastavimo igru, primimo bilo kakvu štetu i ponovno je pauziramo. Sada možemo filtrirati sve pronađene adrese s uvjetom da se vrijednost zapisana na toj adresi smanjila jer je naš igrač primio štetu pa smo u to sigurni. Taj proces zatim ponavljamo nekoliko puta sve dok nemamo razumno malu listu adresa koje možemo ručno provjeriti.

Adresa	Opis
0x800a961e	životni bodovi prvog igrača
0x800aaeaa	životni bodovi drugog igrača
0x8009547c	indikator je li borba u tijeku - 0 ako nije, 1 ako je
0x8009548c	indikator je li igra pauzirana - 0 ako nije, 1 ako je

Tablica 4.1: Informacije iz igre u memoriji.

5 | Eksperimentalni rezultati

Sada ćemo opisati rezultat najuspješnijeg eksperimenta kojeg smo proveli. Svi eksperimenti su provedeni na osobnom računalu s *Intel core i7-12700H* procesorom i *Nvidia RTX 3050* (*Laptop*) grafičkom karticom.

Odabrali smo da je agentov borac *King*, a protivnik je *Law* na laganoj težini igre. Eksperiment koristi *DDQNAgent* klasu koja implementira učenje iz Algoritma 4. Proces treniranja i evaluacije je sljedeći — agent prvo uči kroz 15 mečeva, a zatim se evaluira kroz 3 meča. Kada evaluiramo agenta koristimo pohlepnu strategiju za odabir akcije, odnosno uzimamo onu akciju kojoj odgovara najveća Q-vrijednost. Nagrada se sastoji od dva dijela — jedan dio nagrade je količina štete koju je agent nanio protivniku minus količina štete koju je agent primio, a drugi dio nagrade se dobiva samo ako je meč završio i iznosi +100 za pobjedu ili -100 za gubitak. Na taj način pozitivno podupiremo agentovo napadanje protivnika i izbjegavanje njegovih napada. S obzirom na to da je 130 maksimalna količina životnih bodova to znači da je ukupna nagrada kroz jedan meč između -230 i +230.

Metrike koje smo koristili su količina napravljene štete protivniku, količina primljene štete, nagrada, te postotak pobjeda. Ove vrijednosti računamo samo tijekom evaluacije i računamo ih u prosjeku kroz sva 3 meča evaluacije. Na primjer, ako bi naš agent u prvom meču napravio 33 štete protivniku, u drugom 120 i u trećem 65, tada bi zabilježili da je prosječna napravljena šteta te evaluacije $\frac{33+120+65}{3} = 72.67$.

Ovaj eksperiment smo trenirali 1.15M koraka što se prevodi u 32 sata treniranja. Pravo vrijeme je zapravo trostruko veće jer treniramo na 180 fps umjesto 60 fps pa je treniranje otprilike 3 puta brže nego u pravom životu. Agent je u tih 32 sata odigrao oko 17250 mečeva.

U Tablici 5.1 možemo vidjeti značenje svakog od parametra ovog eksperimenta. Parametre eksperimenta koji je postigao najbolji rezultat možemo vidjeti u Tablici 5.2, a rezultate na Slici 5.1. U rezultatima pratimo 4 informacije koje opisujemo u Tablici 5.3 i to protiv protivnika na laganoj težini igre koja ima oznaku (easy) na slici, te teškoj težini igre s oznakom (hard).

Kao što možemo vidjeti iz slike, agent je postigao malo preko 90% postotak pobjeda protiv protivnika na laganoj težini, a oko 33% na teškoj težini. Za usporedbu možemo uzeti RandomAgenta koji šalje samo nasumične akcije, koji postiže 79% na laganoj težini, a 25% na teškoj težini. Iz toga zaključujemo da naš agent uspješno uči.

Parametar	Opis
observations_per_state	Koliko uzastopnih okvira smatramo kao jedno stanje.
update_target_model	Frekvencija ažuriranja ciljne mreže iz algoritma 2
eps_func	Funkcija koja prima trenutni vremenski korak treniranja i vraća epsilon koji koristimo za (3.6).
learning_rate	Početna brzina treniranja koju optimizator koristi.
batch_size	Veličina podskupa iskustava koje uzorkujemo iz spremnika iskustava opisanog u Algoritmu 3.
memory_capacity	Kapacitet spremnika iskustava iz Algoritma 3.
discount	γ iz (3.1).

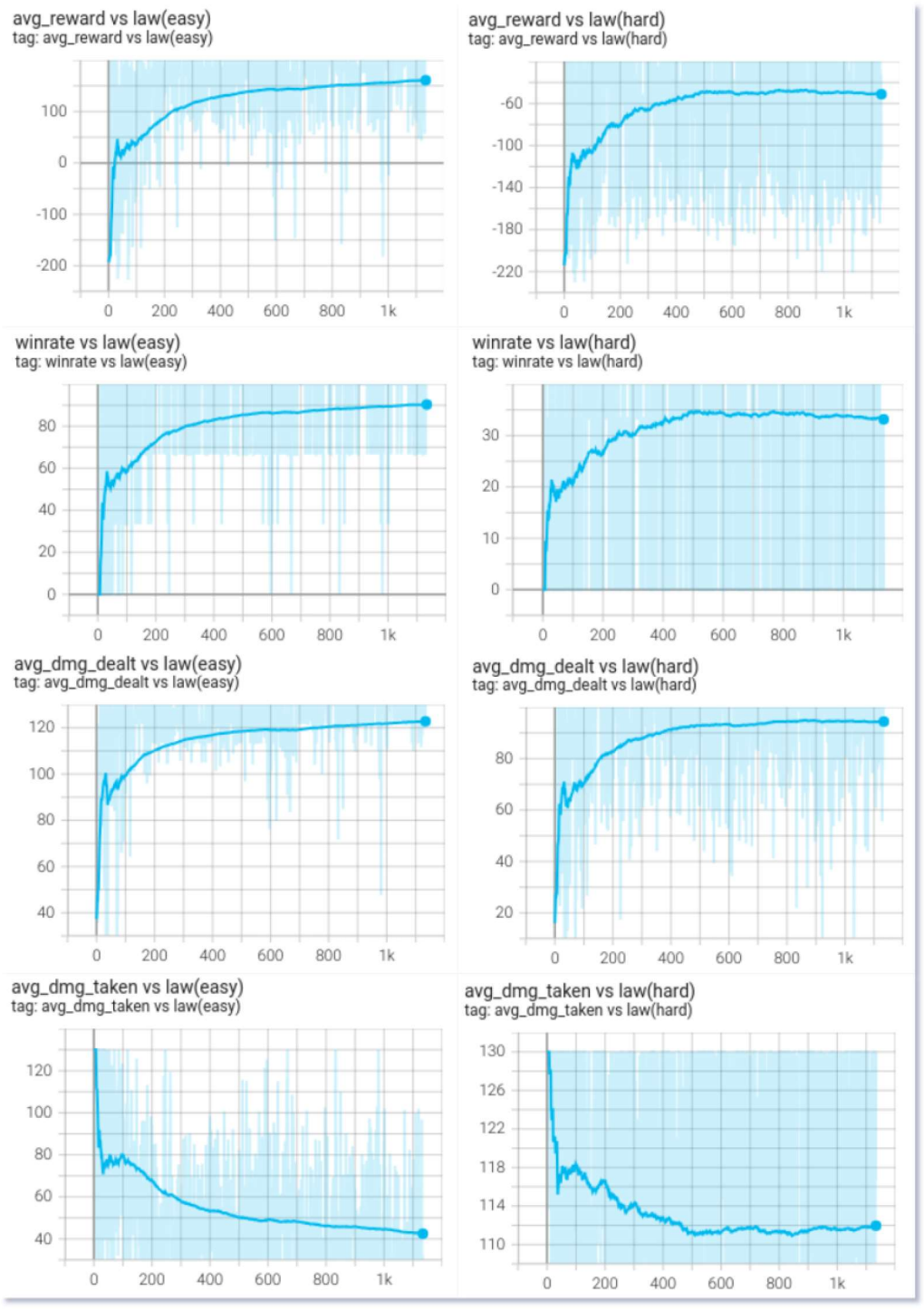
Tablica 5.1: Opisi parametara.

Parametar	Vrijednost
observations_per_state	15
update_target_model	15000
eps_func	$\epsilon_t = \max(0.2, 0.9 - \frac{t}{150000})$
learning_rate	1e-6
batch_size	32
memory_capacity	3000
discount	0.99

Tablica 5.2: Parametri najboljeg eksperimenta.

Oznaka	Značenje
avg_reward	Prosječna nagrada koju je agent dobio.
winrate	Postotak pobjede agenta.
avg_dmg_dealt	Prosječna šteta koju je agent nanjeo protivniku.
avg_dmg_taken	Prosječna šteta koju je agent primio od protivnika.

Tablica 5.3: Značenje rezultata.



Slika 5.1: Rezultat najboljeg eksperimenta.

6 | Zaključak

Važnost potpomognutog učenja u video igrama je višestruko. Potpomognuto učenje u video igrama poboljšava iskustvo igranja, ali i pomaže u boljem razumijevanju i daljnjem razvoju novih algoritama. Igrači mogu koristiti pametne agente da poboljšaju svoje vještine trenirajući protiv njih ili otkrivanjem novih strategija analizom njihove igre.

U ovom radu smo pokazali da je moguće uz relativno malo treniranja i količine računanja, trenirajući isključivo na osobnom računalu, dobiti obećavajuće rezultate koristeći algoritme potpomognutog učenja. Osim toga, dizajnirali smo i implementirali softversko sučelje za brzi i laki razvoj pametnih agenata za video igru Tekken 3.

Ipak, razina igranja našeg najboljeg agenta je još uvijek dosta daleko od razine ljudskog eksperta. Očito, jedan od načina poboljšanja je da treniramo duže vremena. Isto tako, uz pristup većoj količini računanja, možemo ubrzati treniranje tako da pokrenemo više paralelnih instanci igre i na taj način puno brže generiramo raznolika iskustva na kojima se agent trenira. Također, ovo softversko sučelje je ipak napravljeno primarno za korištenje uz video igru Tekken 3, ali uz nekoliko promjena se može generalizirati za korištenje za bilo koju video igru na PSX konzoli.

Literatura

[1] B. Baker, I. Akkaya, P. Zhokhov, J. Huizinga, J. Tang, A. Ecoffet, B. Houghton, R. Sampedro, J. Clune, *Video PreTraining (VPT): Learning to Act by Watching Unlabeled Online Videos*, arxiv:2206.11795 (2022)

[2] M. G. Bellemare, Y. Naddaf, J. Veness, M. Bowling, *The Arcade Learning Environment: An Evaluation Platform for General Agents*, arxiv:1207.4708 (2012)

[3] OpenAI : C. Berner, G. Brockman, B. Chan, V. Cheung, P. Dębiak, C. Dennison, D. Farhi, Q. Fischer, S. Hashme, C. Hesse, R. Józefowicz, S. Gray, C. Olsson, J. Pachocki, M. Petrov, Henrique P. d. O. Pinto, J. Raiman, T. Salimans, J. Schlatter, J. Schneider, S. Sidor, I. Sutskever, J. Tang, F. Wolski, S. Zhang, *Dota 2 with Large Scale Deep Reinforcement Learning*, arxiv:1912.06680 (2019)

[4] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, W. Zaremba, *OpenAI Gym*, arXiv:1606.01540 (2016)

[5] V. Firoiu, W. F. Whitney, J. B. Tenenbaum, *Beating the World's Best at Super Smash Bros. with Deep Reinforcement Learning*, arxiv:1702.06230 (2017)

[6] H. V. Hasselt, A. Guez, D. Silver, *Deep Reinforcement Learning with Double Q-learning*, arxiv:1509.06461 (2015)

[7] M. Hessel, J. Modayil, H. van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, D. Silver, *Rainbow: Combining Improvements in Deep Reinforcement Learning*, arxiv:1710.02298 (2017)

[8] Jumper, J., Evans, R., Pritzel, A. et al., *Highly accurate protein structure prediction with AlphaFold*, Nature **596**, 583–589 (2021)

[9] Kober J., Bagnell J.A., Peters J., *Reinforcement learning in robotics: A survey.*, The International Journal of Robotics Research. 2013;32(11):1238-1274. doi:10.1177/0278364913495721

[10] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, M. Riedmiller *Playing Atari with Deep Reinforcement Learning*, arxiv:1312.5602 (2013)

[11] Mnih, V., Kavukcuoglu, K., Silver, D. et al., *Human-level control through deep reinforcement learning*, Nature **518**, 529–533 (2015)

[12] I. Oh, S. Rho, S. Moon, S. Son, H. Lee, J. Chung, *Creating Pro-Level AI for a Real-Time Fighting Game Using Deep Reinforcement Learning*, arxiv:1904.03821 (2019)

[13] C. Purves, C. Cangea, P. Veličković, *The PlayStation Reinforcement Learning Environment (PSXLE)*, arxiv:1912.06101 (2019)

[14] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai. A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, D. Hassabis, *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm*, arxiv:1712.01815 (2017)

[15] R. S. Sutton, A. G. Barto, *Reinforcement Learning: An Introduction*, MIT Press, Cambridge, MA, 2018

[16] Vinyals, O., Babuschkin, I., Czarnecki, W.M. et al., *Grandmaster level in StarCraft II using multi-agent reinforcement learning*, Nature **575**, 350–354 (2019)

[17] J. Walker, *Everything You Have Always Wanted to Know about the Playstation But Were Afraid to Ask*, Web izvor dostupan na:

`https://fabiensanglard.net/doom_psx/psx.pdf`

[18] S. Wang, D. Jia, X. Weng, *Deep Reinforcement Learning for Autonomous Driving*, arxiv:1811.11329 (2019)

[19] *Sony Computer Entertainment Inc. Cumulative Software Titles (as of June 30, 2008)* Web izvor dostupan na

`https://web.archive.org/web/20080921062349/http://www.scei.co.j`
`↪ p/corporate/data/bizdatatitle_e.html`

[20] *Tekken 3 Manual*, Web izvor dostupan na:

`https://secure.cdn.us.playstation.com/manuals/classic/games/tek`
`↪ ken-3-manual-en.pdf`

[21] Grumpycoders, *PCSX-Redux* (2018), Web izvor dokumentacije

`https://pcsx-redux.consoledev.net/`

Sažetak

U ovom radu se bavimo primjenom potpomognutog učenja na video igru Tekken 3. Prezentiramo teoriju i implementaciju algoritama dubokog potpomognutog učenja, te implementaciju programskog sučelja preko kojeg algoritam i video igra mogu komunicirati.

Ključne riječi

potpomognuto učenje, tekken, video igre, playstation, pytorch, python, lua, pcsx-redux, q-učenje, duboko potpomognuto učenje, duboko q-učenje

Applying reinforcement learning on the video game Tekken 3

Summary

In this paper we apply reinforcement learning on the video game Tekken 3. We present the theory and implementation of deep reinforcement learning algorithms, as well as the implementation of a programming interface through which the algorithms and the video game can communicate.

Keywords

reinforcement learning, tekken, video games, playstation, pytorch, python, lua, pcsx-redux, q-learning, deep reinforcement learning, deep q-learning

Životopis

Rođen sam 1999. godine u Osijeku. Srednju školu završavam u Đakovu 2018. te se iste godine upisujem na preddiplomski studij matematike i računarstva na Odjelu za matematiku u Osijeku. Godine 2021. završavam preddiplomski studij i upisujem se na diplomski studij matematike, smjer matematika i računarstvo isto na Odjelu za matematiku.