Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Žiga Franko Gorišek

Odkrivanje znanj igre Black jack z metodami spodbujevanega učenja

diplomsko delo

visokošolski strokovni študijski program
prve stopnje
računalništvo in informatika

Mentor: viš. pred. dr. Aleksander Sadikov

Ljubljana, 2018

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Odkrivanje znanj igre Blackjack z metodami spodbujevanega učenja:

Kandidat naj na primeru igre Blackjack razišče sodobne tehnologije za odkrivanje znanj in zakonitosti v podatkih in dinamičnih sistemih kot so igre. Spozna se naj z (globokimi) nevronskimi mrežami, spodbujevanim učenjem in kombiniranjem obeh tehnik. Nauči naj se igrati Blackjack, kar pomeni tako odločanje glede stav kot tudi odločanje med samo igro. Rezultate naj primerja z obstoječo teorijo igre.

Kazalo

Seznam uporabljenih kratic

kratica	**angleško**	**slovensko**
SVM	Support vector machine	Metoda podpornih vektorjev
DQN	Deep Q newtork	Globoke Q mreže
...	...	...

Povzetek

Naslov: Odkrivanje znanj igre Black jack z metodami spodbujevanega učenja

Avtor: Žiga Franko Gorišek

Skozi študijski proces, sem pridobil veliko znanja ter zanimanja na področjih, kot so algoritmi, optimizacija, stave, finance ter umetna inteligenca. Tako sem razvil željo, da bi se specializiral na področju Globokega učenja (Deep learning).

Želel sem doseči stopnjo znanja in razumevanja kako naučeno specializacijo izkoristiti z najnovejšimi tehnologijami strojnega učenja.

Izredno me je tudi prevzela zgodba o MIT študentih, ki so odkrili in izkoristili luknjo v igri Black jack, s katere so si odprli dobičkonosno igro.

Zato sem se odločil narediti sistem, ki se nauči odkriti zakonitosti igre Black jack, se tako naučiti tehnike DQN in poskušati izboljšati osnovo njihovega sistema.

Ključne besede: DQN, Nevronske mreže, Black jack, delivec, igralec, model.

Abstract

Title: Discovering the laws of Black jack game with reinforcement learning

Author: Žiga Franko Gorišek

Through the study process, I gained a lot of knowledge and interests in fields such as algorithms, optimization, betting, finance and artificial intelligence. So I developed a desire to specialize in Deep Learning.

I wanted to achieve a level of knowledge and understanding of how to use the learned specialization with the latest machine learning technologies.

I was also extremely impressed by the story about MIT students who discovered and used the Black jack hole from which they opened a profitable game.

So I decided to make a system that learns how to discover the legitimacy of the Black jack game, to learn DQN techniques and try to improve the basis of their system with it.

Keywords: DQN, Neural network, Black jack, dealer, player, model.

Poglavje 1

Uvod, motivacija in igra Black jack

1.1 Uvod

Intuicija diplomske naloge izhaja z nekdanje MIT skupine študentov, ki je okoli leta 1979 ugotovila, kako in s kakšnimi pravili lahko premagamo igro Black jack. Njihova zgodba temelji na odličnih matematikih, ki so odkrili in izkoristili štetje kart v svojo prid. Več o njih si lahko ogledate v prispevku: The True Story of The MIT Black jack Team [1].

Tako bomo sprva spoznali celotno igro Black jack, kako se je razvila, kakšna so njena osnovna pravila ter kakšna pravila potrebujemo za dobičkonosno igro. Preden se dotaknemo oblikovanja celotnega sistema, bo potrebno razumeti tudi različne statistične mere, s katerimi bomo ocenjevali naše odigrane igre. Tu bomo tudi spoznali približne željene ocene modelov po učenju.

Po celotnem spoznanju igre Black jack in vseh ocenjevan v njej, bomo spoznali metodo strojnega učenja, imenovano nevronske mreže in odkrili zakaj so prišle do izraza le nekaj let nazaj. V njej bomo spoznali nekaj ključnih razvojnih metod, kot tudi metod s katerimi so postale uspešne.

Ker pa so modeli nevronskih mrež narejeni za nadzorovano učenje, se bomo dotaknili metode spodbujevanega učenja. Obrazloženo je celotno delo-

vanje, tako starih metod Q-učenje (Q-Learning), kot tudi najnovejše metode Globokih Q mrež (Deep Q-Nework).

Po postavljenih temeljih same igre Black jack, kot tudi metode Deep Q-Network, se bomo lotili gradnje sistema. Tu se sistem deli na dva dela. Stavno in igralno odločanje. Za oba bomo zgradili modela, katera se bosta sama prilagajala igri Black jack.

Ker je v igralnem modelu spekter parametrov prevelik, je opisana tudi trenutno najnovejša tehnika iskanja najboljše kombinacije parametrov imenovana Populacijsko iskanje.

Na koncu bomo primerjali kombinacije vseh modelov, s katerimi bomo prišli do končnih ugotovitev.

Moja pričakovanja diplomske naloge so, da se naučim tako teorije tehnologij nevronskih mrež kot tudi jih znati praktično uporabiti. To znanje uporabiti pri gradnji igralnega modela in stavnega modela. Za igralni model želim, da bi se naučil igrati optimalno igro, ki nam jo priporoča tudi igralnica. Stavni model je cilj naučiti igrati dobičkonosno igro.

1.2 Black jack

Black jack ali 21, je igra kart med igralci ter delivcem, kjer vsak igralec igra proti delivcu, vendar ne proti ostalim igralcem. Igra se igra z več kupčki, po 52 kart, običajno je število kupčkov med štiri ter osem in je ena najbolj igranih bančniških iger v igralnicah.

Prve oblike igre so bile omenjene v knjigi Rinconete y Cortadillo španskega avtorja Miguel de Cervantes [2], ki govori o goljufanju pri igri veintiuna, kar po špansko pomeni 21. Opisuje jo kot igro, kjer moraš doseči vrednost 21 s kartami, ki je ne smeš preseči in kjer As opisuje dve stanji 1 ter 11.

1.3 Osnovna pravila igre

Pravila so se skozi leta spreminjala, namreč sprva igra ni bila optimalna za igralnico. Razlike med pravili najdemo tudi po igralnicah. Namreč, vsaka igralnica ima napisana svoja pravila, ki jih je treba pri igranju upoštevati. Zato bomo najprej spoznali osnovne standarde igre Black jack, nato pa se dotaknili pravil, ki jih potrebujemo, da odpremo možnost dobičkonosni igri.

1.3.1 Vrednosti kart

Vsaka karta predstavlja svojo vrednost v igri. Poznamo:

- karte od 2 do 10 imajo vrednost enako njeni številki,
- karte od J do K imajo vrednost enako 10,
- karta A pa je izjema, ker lahko predstavlja vrednost 1 in 11. Njegova vrednost pa je dodeljena na podlagi drugih kart, ki jih imamo.

1.3.2 Stava

Ob začetku igre ima vsak igralec določeno vsoto vloženega denarja, ki ga zamenjajo za čipe, kjer vsak čip predstavlja svojo vrednost (enako kot kovanci/bankovci). Pred vsako igro tako igralci stavijo, ali bodo premagali delivca v naslednji igri ali ne. Vložijo lahko katero koli vsoto, ki pa je omejena na naše trenutno stanje ter najmanjše in največje omejitve stave, ki jih predpisuje vsaka igralnica drugače.

1.3.3 Odločanja v igri

Po položenih stavah se po vrsti vsakemu igralcu dodeljuje karta, tako da ima na koncu vsak igralec dve karti in delivec ima eno karto. Drugo dobi po odigranih potezah igralcev. Med igro lahko v različnih položajih uporabimo različne izbire, kako se bo igra nadaljevala. Tako poznamo štiri osnovna odločanja: Ne povleci, Dodatna karta, Podvojitev ter Deljenje.

Izbiri Ne povleci ter Dodatna karta je možno odigrat v vsakem položaju igre. Kot pa izbiri Podvoji, ki je na voljo, ko ima igralec dve karti in Razdeli, ki je tudi na voljo, ko ima igralec dve karti, vendar samo takrat, ko sta obe enaki.

Ne povleci (Stand) je možnost, ki pove delivcu, da smo zadovoljni z našim seštevkom kart. Tako se igralčeva igra zaključi.

Dodatna karta (Hit) delivec nam dodeli še eno karto, po kateri se ponovno odločamo, kako bomo nadaljevali samo igro.

Podvojitev (Double) naš trenutni vložek podvojimo. Posledično nam delivec dodeli še eno karto, po dobljeni karti se naša poteza zaključi.

Deljenje (Split) možnost, nam razdeli karti v dve polji, na katerih je stava enaka začetni stavi. Torej vložimo še enkrat toliko kot smo, nato odigramo oba polja posebej.

***Predaja** (Surrender) dodatna možnost, ki jo je večina igralnic zavrglo, a je potrebna za dobičkonosno igro. Opis sledi v nadaljevanju.

Osnovno odločanje, ki se je izkazalo za najbolj učinkovito v sami igri, je predstavljeno v tabeli 1.1.

1.3.4 Igranje delivca

Ko zaključijo svoja odločanja v igri vsi igralci, se dodeli delivcu še eno karto, da ima dve karti. Njegovo odločanje je dokaj preprosto. Vedno se odloča izključno z dvema izbirama Ostani in Povleci.

Možnost Ostani, se izbere samo v primeru, da je njegova vrednost kart večja ali enaka 17. V vseh ostalih primerih kliče opcijo Povleci. Obstaja pa tudi izjema, ko kliče Povleci, kljub temu da ima vrednost kart enako 17, vendar ima v karto A.

1.4 Dodeljevanje nagrad

Nagrade se razdelijo po odigrani igri delivca. Od dobljene nagrade, pa je odvisno, kakšna je končna vrednost kart, ki jih ima igralec in kolikšna je bila njegova začetna stava. Nagrade se tako dodeljujejo po naslednjem postopku, kjer primerjamo vsakega igralca posebej z delivcem.

- če je igralec odigral napačno potezo avtomatično izgubi vložek,
- če je igralec odigral Predajo, dobi pol vložka nazaj,
- če je vrednost kart, tako igralca, kot tudi delivca enaka, dobi igralec nazaj celoten vložek,
- če je vrednost kart igralca enaka 21, dobi 2.5 kratnik vložka,
- če je vrednost kart igralca večja od 21, igralec izgubi vložek,
- če je vrednost kart delivca večja kot 21, igralec dobi dvakratnik vložka,
- če je vrednost kart delivca večja od igralčevih, igralec izgubi vložek,
- in če je vrednost kart igralca večja, kot vrednost kart delivca, igralec dobi dvakratnik vložka.

1.5 Uporabljena pravila

Da je moč premagati samo igro Black jack, je treba v igri uporabljati določena pravila, ki so se že uporabljala v preteklosti, vendar jih večina igralnic ne uporablja več.

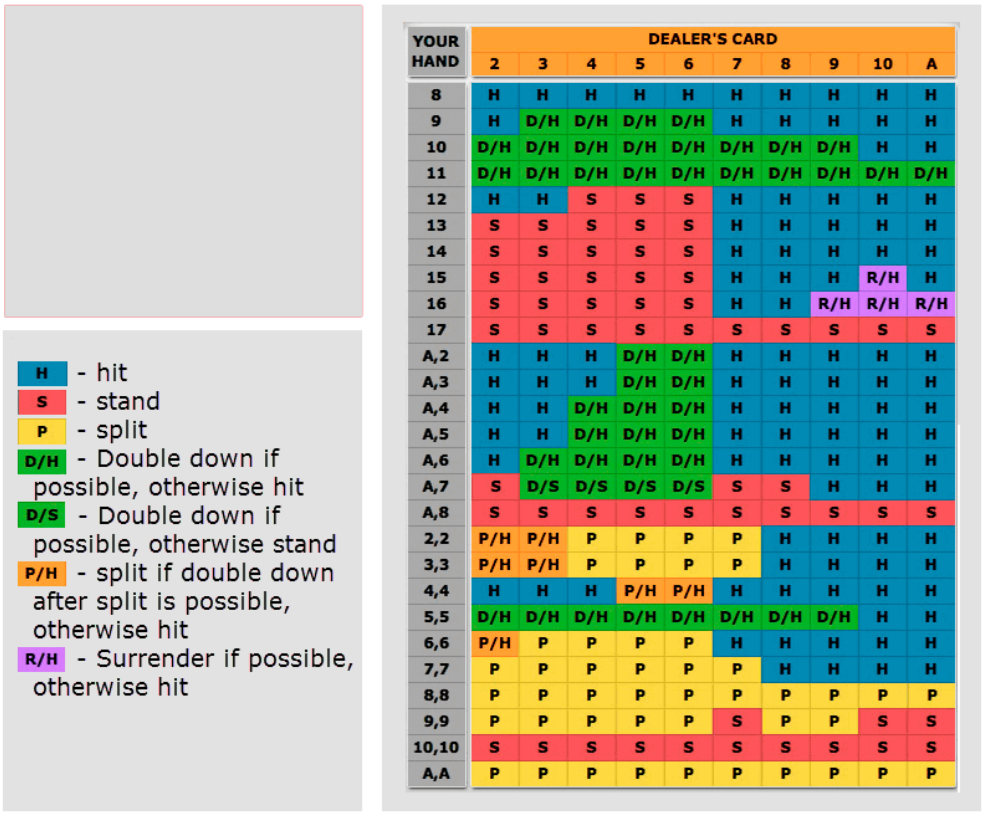
Treba je dodati možnost izbire v samem odločanju v igri in zagotoviti, da delivec uporablja kupček kart.

Uporabljena izbira je tako imenovana Predaja (Surrender). Na voljo je samo, ko ima igralec v roki 2 karti. Omogoča, pa nam, da predamo igro in dobimo polovico svoje stave nazaj. Pri štetju kart je to zelo uporabno v

primeru velikega vložka, kjer dobimo seštevek obeh kart enak 15 ali 16, in da ima delivec eno od kart med 10 in karto A.

Za igranje dobičkonosne igre je treba tudi zagotoviti, da delivec ne zmeša kart med vsako igro. V primeru, da jih, ne moremo uporabljati štetja kart, ker se števec vsako igro postavi na nič. Več o tem v nadaljevanju.

Primer tabele odločanja, ki nam jo priskrbi igralnica, je prikazana na sliki 1.1.



YOUR HAND	DEALER'S CARD 2	3	4	5	6	7	8	9	10	A
8	H	H	H	H	H	H	H	H	H	H
9	H	D/H	D/H	D/H	D/H	H	H	H	H	H
10	D/H	D/H	D/H	D/H	D/H	D/H	D/H	D/H	H	H
11	D/H	D/H	D/H	D/H	D/H	D/H	D/H	D/H	D/H	D/H
12	H	H	S	S	S	H	H	H	H	H
13	S	S	S	S	S	H	H	H	H	H
14	S	S	S	S	S	H	H	H	H	H
15	S	S	S	S	S	H	H	H	R/H	H
16	S	S	S	S	S	H	H	R/H	R/H	R/H
17	S	S	S	S	S	S	S	S	S	S
A,2	H	H	H	D/H	D/H	H	H	H	H	H
A,3	H	H	H	D/H	D/H	H	H	H	H	H
A,4	H	H	D/H	D/H	D/H	H	H	H	H	H
A,5	H	H	D/H	D/H	D/H	H	H	H	H	H
A,6	H	D/H	D/H	D/H	D/H	H	H	H	H	H
A,7	S	D/S	D/S	D/S	D/S	S	S	H	H	H
A,8	S	S	S	S	S	S	S	S	S	S
2,2	P/H	P/H	P	P	P	P	H	H	H	H
3,3	P/H	P/H	P	P	P	P	H	H	H	H
4,4	H	H	H	P/H	P/H	H	H	H	H	H
5,5	D/H	D/H	D/H	D/H	D/H	D/H	D/H	D/H	H	H
6,6	P/H	P	P	P	P	H	H	H	H	H
7,7	P	P	P	P	P	P	H	H	H	H
8,8	P	P	P	P	P	P	P	P	P	P
9,9	P	P	P	P	P	S	P	P	S	S
10,10	S	S	S	S	S	S	S	S	S	S
A,A	P	P	P	P	P	P	P	P	P	P

Slika 1.1: Pravilna tabela odločanja pri igri Black jack

V narejeni simulaciji je implementirana izbira, da lahko igralec predstavlja več igralcev.

Poglavje 2

Kako igrati dobičkonosno igro?

Ker hočemo v nadaljevanju naučiti računalnik igrati napredno Black jack igro (brez predznanja), je dobro razumeti, kako pridemo tudi sami do napredne igre. Tako pridemo znova do MIT ekipe in njihovega odločanja. Najprej je treba razumeti, da je vsak sistem edinstven, namreč vsaka igralnica ima svoja pravila (limite, različne igralne izbire ...), prav tako je treba upoštevati, kakšno vsoto denarja igralec nameni vsaki igri.

Da predstavljen sistem deluje je treba zagotoviti, da igralnica dopušča možnost Predaj, igra s štiri do osem kupčki kart, ki jih ponovno zmeša, ko je še 10 kart v celotnem kupčku ter omejitev stave $2.000EUR$.

Zdaj, ko vemo želene lasnosti, je treba rešiti dve težavi. Kako bomo igrali samo igro Black jack in kako bomo postavljali stave, da bo sistem čez čas dobičkonosen?

2.1 Zmagovalna stopnja

Preden preidemo na igralna odločanja in stave v igri, je treba vedeti, kako bomo ocenjevali uspešnost svojih metod. V Black jacku se nagrade dodeljujejo, kot je opisano v poglavju 1.4.

To lahko posplošimo na *loss* (vse izgubljene stave, katerih je ln), *push* (rezultate med igralcem in delivcem je bil neodločen, katerih je lp) in *win*

(igralec je premagal delivca, katerih je wn). Tako velja, da je končen rezultat enak izračunu 2.3.

$$score_{positive} = \sum_{w=1}^{wn} win_w \tag{2.1}$$

$$score_{negative} = \sum_{l=1}^{ln} loss_l \tag{2.2}$$

$$money_{end} = score_{positive} + score_{negative} \tag{2.3}$$

Vendar, končen denar ($money_{end}$) nam ne da vedeti, kakšen je procentualen dobiček po odigranih igrah. Za izračun uporabimo formulo 2.4, ki nam pove, koliko odstotkov smo zaslužili v primerjavi z začetnim vložkom ($money_{start}$).

$$P_{win} = \frac{money_{end} - money_{start}}{money_{start}} \tag{2.4}$$

2.2 Odločanje v igri

Pri samem odločanju ni velike umetnosti, namreč igralnica je optimizirala igro tako, da bo vedno negativna. Zato kot osnovo skozi samo igro vzamemo tabelo, ki nam jo priskrbi igralnica (primer tabele odločanja: 1.1). To nam zagotovi, v našem primeru 49,7 % zmagovalno stopnjo (upoštevano je, da je pri vsakem stavnem odločanju uporabljena konstantna stava).

2.3 Enotna stava

Ena zelo uporabnih tehnik pri stavljenju je ta, da določimo velikost stave, glede na svoj trenuten položaj, v našem primeru, trenuten denar. Na samem začetku si izberemo, kolikšen delež trenutnega denarja bomo namenili eni stavi. Da lahko vsakič izračunamo trenuten $unit$, opredelimo $unit_{devider}$, ki je v našem primeru $unit_{devider} = 1000$, ter trenuten denar $money$. Torej,

vzeli bomo 0,1 % delež trenutne vsote kot trenuten "bet". Dobro je, da *unit* nesvebuje več kot 25 % trenutnega deleža, namreč želimo imet dovolj čipov za Podvojitev ali Deljenje.

$$unit = \frac{money}{unit_{devider}} \tag{2.5}$$

Zakaj Enotna stava? Dobra stran uporabe Enotne stave je, da nam teoretično ne more zmanjkati denarja, če igramo negativno igro (zmagovalna stopnja < 50 %), ter v primeru pozitivne igre (zmagovalna stopnja > 50 %) se nam denar nenehno veča.

2.4 Stavno odločanje

Čar dobičkonosne igre je v stavnem odločanju. Da vemo, kdaj se nam splača vplačati večjo vsoto čipov, lahko to izračunamo z že padlimi kartami. Te hranimo v $cards_{fallen}$, katerih je fn. Karte razdelimo na slabe (1), nevtralne (0) in dobre (-1) karte. Tako lahko izračunamo trenuto stanje $count_{real}$, ki je opredeljeno v 2.6.

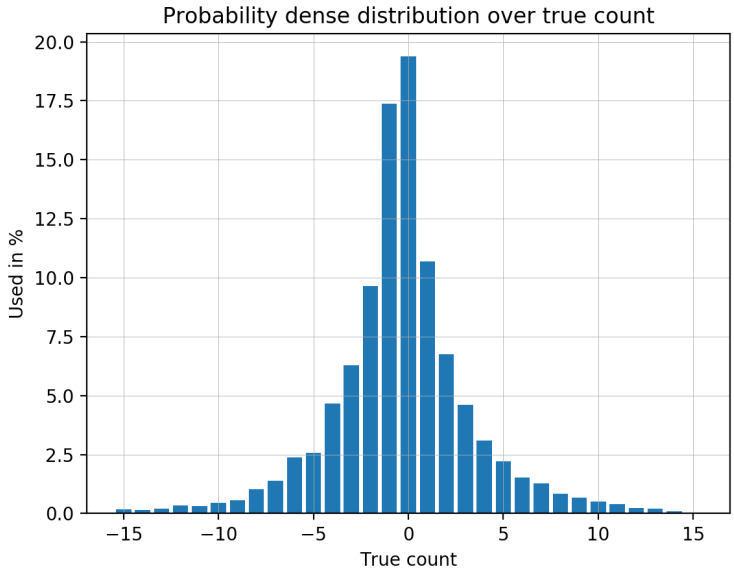
$$count_{real}(cards_{fallen}) = \sum_{card=1}^{fn} \begin{cases} 1 & cards_{fallen}[card] \in [2,3,4,5,6] \\ 0 & cards_{fallen}[card] \in [7,8,9] \\ -1 & cards_{fallen}[card] \in [10,J,Q,K,A] \end{cases} \tag{2.6}$$

Težavo, ki jo imamo v primeru, da se odločamo na podlagi $count_{real}$ je ta, da je varianca veliko večja, ko je delivčev kupček poln in pada, ko se kupček manjša. Da bi rešili to linearno težavo dodamo $count_{real}$ še nelinearno funkcijo, ki jo definiramo kot $count_{true}$, a potrebujemo še dodatno sprejemlivko $deck_{len}$, ki pove koliko kupčkov delivec še ima (ta vrednost je zaokrožena na celo število!). Tako lahko definiramo $count_{true}$ kot 2.7.

$$count_{true} = \frac{count_{real}}{deck_{len}} \tag{2.7}$$

2.4.1 Monte Carlo simulacija

Zdaj, ko vemo, kako se izračuna $count_{true}$ nas zanima, kakšne so verjetnosti zmage, ter kakšna je distribucija gostote določenega stanja. Monte Carlo simulacija [12] je bila narejena prav za ta namen. Tako lahko z njo ugotovimo, kakšna je odstotek gostote (probabilety densety distribution) vsake vrednosti v $count_{true}$ z visokim zaupanjem. Za lažje razumevanje bomo $count_{true}$ omejili med -15 in 15, ostale vrednosti pa zanemarili.

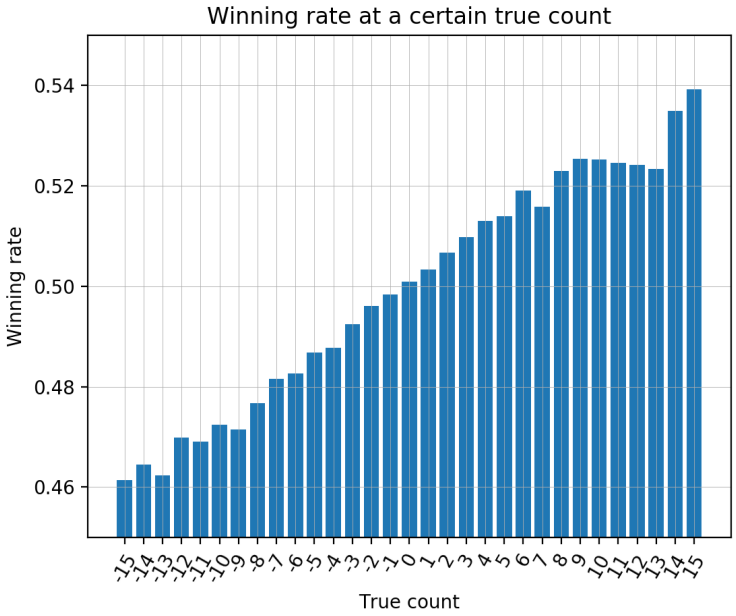


Slika 2.1: True count procentualna distribucija

Kot je razvidno s histograma, je $count_{true}$ več kot $95,68\%$ časa med -8 ter 8. Poglejmo si še, kakšne verjetnosti zmage nas čakajo v določenem položaju $count_{true}$. Pri tem bomo primerjali, razliko med dobljenimi pozitivnimi in negativnimi rezultati za vsak $count_{true}$ po formuli 2.8.

Upoštevati moramo, da so v formuli 2.2 vse vrednosti negativne.

$$P_{win} = \frac{score_{positive}}{score_{positive} - score_{negative}} \tag{2.8}$$



Slika 2.2: Zmagovalna distribucija skozi $count_{true}$

Odkrili smo, kje je prestopna točka našega sistema!

Vidimo, da podatki na grafu še ne ležijo povsem uravnoteženo, želim opomniti, da so podatki grafa iz 1.500 iger po 1.000 stav. V nadaljevanju bi bilo treba izvesti testiranje na večjem naboru iger.

Poglavje 3

Deep Q learning

V naslednjem poglavju bomo spoznali teorije tehnologij, ki se uporabljajo v Deep Q Network algoritmu. Najprej se bomo dotaknili nevronskih mrež, kako se uporabljajo v nadzorovanem učenju ter vse tehnologije katere se uporabljajo. Nato bomo spoznali osnoven algoritem za spodbujevano učenje po imenu Q-učenje. Na koncu pa spoznali kombinacijo Q-učenja ter nevronskih mrež, imenovano Deep Q Network.

3.1 Nevronske mreže

V strojnem učenju poznamo kar nekaj učnih algoritmov, ki so v zadnjih desetletjih pripomogle k razvoju računalnika in prav tako človeka. V prejšnjem desetletju so se po večini uporabljale naslednje tehnike, kot so: Logistic regression, SVM, Random forest, Boosting in še bi jih lahko naštevali. A do ne davnega se je velikost informacij drastično povečala. Prišli smo v čas, kjer imamo milijone podatkov, na katerih lahko izvajamo razna učenja.

Tako so dobile svoj pomen tudi nevronske mreže [9], ki sta ga predstavila nevrofiziolog Warren McCulloch in matematik Walter Pitts, leta 1943. Zakaj so nevronske mreže dobile pomen šele zdaj, ko imamo na voljo milijone podatkov? Izkazalo se je, da so nevronske mreže izredno močne pri interpretaciji bolj obširnih podatkov. Recimo opis slike (CNN [16]), predstavitev besede

(RNN - word embading [8]), prevajanje (Attention model [11]), prepoznavanje govora (RNN, LSTM [14]). Vendar pri tako zahtevnih nalogah potrebuješ ali dobre minimalistične opise podatkov, ki jih je treba sprogramirati ali pa velike količine podatkov in učni algoritem, ki jih lahko obdela.

3.1.1 Delovanje

Za lažji opis, kako nevronske mreže delujejo, lahko to interpretiramo kot skupino ljudi, v kateri se predaja informacija po ravneh (imamo L nivojev). Skupina prejema informacije, po katerih se odloči za določeno akcijo. Vsaka oseba v skupini je predstavljena kot krogec v sliki 3.1 po prvi ravni. Prva raven, so dobljene informacije X ali $A^{[0]}$ (vsak krogec predstavlja svojo informacijo). Vsaka oseba nato obdela vse informacije, ki jim doda svoje mnenje(osebe imajo različna mnenja). Za interpretacijo mnenja je uporabljena linearna funkcija 3.2. Preden oseba odda svoje mnenje nadrejenemu, spusti svoje mnenje skozi, tako imenovano aktivacijo 3.3 (ne-linearno funkcijo označeno kot $G^{[l]}$). Postopek se ponavlja do zadnje osebe, ki se končno odloči za akcijo.

Zapišemo lahko formulo za raven oseb:

$$A^{[0]} = X \tag{3.1}$$

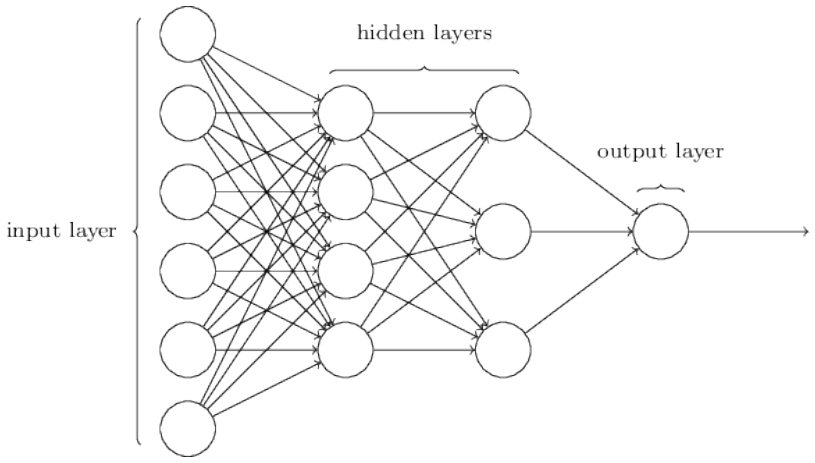
$$Z^{[l]} = A^{[l-1]} \times W^{[l]} + b^{[l]} \tag{3.2}$$

$$A^{[l]} = G(Z^{[l]}) \tag{3.3}$$

Končno odločitev zadnje osebe lahko, tako predstavimo kot: $y_{pred} = A^{[L-1]}$.

3.1.2 Aktivacije

Kar naredi nevronske mreže tako močne, je aktivacija na vsaki ravni. Kot smo videli že razliko med $count_{real}$, ki je bil linearni seštevek, ter $count_{true}$, ki smo ga dobili s tem, da smo razbili $count_{real}$ z $deck_{len}$. Prav to delajo tudi nevronske mreže, ki predelajo informacije z linearno funkcijo, jo preslikajo



Slika 3.1: Simbolična predstavitev nevronskih mrež

z nelinearno funkcijo. S tem dobijo drugačen opis informacij, ki ga nato uporabimo za boljšo napoved. Aktivacija je potrebna, a je lahko na vsakem nevronu drugačna, namreč brez nje bi dobili samo veliko linearno funkcijo. Poglejmo jih nekaj.

Sigmoid Najbol standardna funkcija v samih začetkih je bila Sigmoidna funkcija.

$$f(x) = \frac{1}{1 + e^{-x}} \tag{3.4}$$

Trenutno se ta funkcija najbolj uporablja v zadnji ravni, ker nam omogoča napovedovanje verjetnosti določene akcije. Ta funkcija pa se ne uporablja več med vmesnimi oziroma skritimi ravnmi, namreč, izkazalo se je, da ima funkcija veliko težavo preiti z ene skrajne točke v drugo. To pa izhaja z izredno majhnih gradientov na končnih točkah. Če želimo prit z ene skrajne točke v drugo, potrebujemo veliko korakov, zaradi spreminjanja gradienta.

Tanh Prav tako je bila zelo uporabljena tudi $Tanh$ funkcija.

$$f(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{3.5}$$

Zdaj je njegova uporaba najbol vidna v LSTM, GRU celicah. Uporablja se pa tudi v skritih ravneh.

ReLU Da so nevronske mreže res zaživele, je bil en od dejavnikov tudi ReLU funkcija [4]. Do uporabe je prišla zaradi računanja gradienta, saj so njegovi gradienti direktena razdalja od željene.

$$f(x) = max(0, x) \tag{3.6}$$

Uporaba te funkcije se najde v vseh skritih ravneh nevronskih mrež, ter tudi na zadnjem nivoju pri regresiji.

PReLU V našem modeliranju bomo uporabili aktivacijo PReLU. Ker ReLU funcija ne prenaša šlabe"oziroma negativne informacije, želimo to popraviti z dodatnim učnim parametrom *alpha*. Le ta nam omogoča uporabiti, majhen odstotek negativnih vrednosti.

$$f(x) = max(x \times alpha, x) \tag{3.7}$$

3.1.3 Inicializacija

Sprva, ko naredimo model, še nimamo naučenih uteži. Uporablja se nekaj metod:

Zero inicializacija. Težava, pri zero inicializaciji je, da v primeru, če imamo vse uteži W nastavljene na 0, so gradienti vseh uteži enaki! Tako se vsi nevroni na določeni ravni naučijo samo ene stvari.

Random inicializacija. Zgornjo težavo rešimo z random inicializacijo, tako so že na začetku različna mnenja posameznikov (tudi sam, ko prideš prvič v službo meniš, da je nekaj bolše, kljub temu da tega še nisi nikoli delal). Z to metodo dobimo, pri zelo globokih nevronskih mrežah, problem izginjanja/eksplozije gradienta.

He inicializacija. Zaradi problem izginjanja/eksplozije gradienta, je bila razvita He inicializacija, ali drugače Xavier normal inicializacija [3]. Ta nam omogoča naključno inicializacijo uteži, ki ga zmanjša.

$$Var(W) = \frac{2}{n_{in} + n_{out}} \tag{3.8}$$

Izračun 3.8 nam pove varianco naključnosti na določeni ravni, kjer sta n_{in} število nevronov prejšnji ravni in n_{out} število nevronov naslednje ravni.

3.1.4 Izračun napake

Model želimo čez čas posodabljati, da bodo njegove odločitve boljše. Zato je treba poznat, pravilne odgovore stanj y in napovedane akcije modela y_{pred} za njih. Vseh stanj je n. Ko to vemo, lahko izračunamo za vsak nevron posebej, kje je bila njegova napaka.

Nekaj uporabnih funkcij napak:

- Povprečna kvadratna napaka:

$$J(y, y_{pred}) = \frac{1}{n}\sum_{i=1}^{n}(y_{pred_i} - y_i)^2 \tag{3.9}$$

- Povprečna absolutna napaka:

$$J(y, y_{pred}) = \frac{1}{n}\sum_{i=1}^{n}|y_{pred_i} - y_i| \tag{3.10}$$

- Binarna navzkrižna entropija:

$$J(y, y_{pred}) = \frac{1}{n}\sum_{i=1}^{n} -(y_i log y_{pred_i} + (1 - y_i) log(1 - y_{pred_i})) \tag{3.11}$$

3.1.5 Gradient Descent

Zdaj ko vemo, kje je bila naša napaka, jo lahko prenesemo na model z algoritmom gradient descent. Razvila sta ga Herbert Robbins in Sutton Monro, že leta 1951 kot Stochastic gradient descent [13].

Stochastic gradient descent. Ideja metode je posodabljati model nevronske mreže, vsakega učnega vzorca posamično, z učno oceno α. Prikaz algoritma 1.

Algorithm 1 Stochastic gradient descent

```
1: procedure STOCHASTIC GRADIENT DESCENT(iterations)
2:     for iter in Iterations do
3:         for xi, yi in X, y do
4:             y_pred = model.predict(xi) #predikcija modela
5:             error = loss(yi, y_pred) # izračun napake predikcije
6:             dW, db # izračun gradientov za atributa W ter b
7:             W = W − α × dW # posodobitev vektorja uteži W
8:             b = b − α × db # posodobitev vektorja konstante b
9:         end for
10:    end for
11: end procedure
```

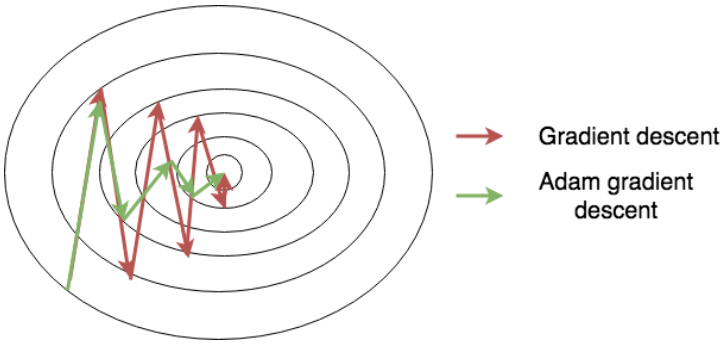
Gradient descent. oziroma Mean gradient descent je bila prva modifikacija, ki je izboljšala Stochastic gradient descent. Razlikuje se samo, da neučimo modela v enem trenutku, ne samo, enem trening primeru, ampak na vseh trening primerih. Tako ocenimo povprečen gradient dW, db vsakega nevrona in s to oceno posodabljamo W in b.

Mini-batch gradient descent. Vendar nastane težava, da imamo kar naenkrat 5.000.000 trening primerov n. Zato je pri gradient descentu nastala težava, ker mora čez vse primere in šele nato posodobi W ter b vsakega nevrona.

Da bi pohitrilil zgornje učenje, je bil razvit Mini-batch gradient descent, ki razdeli trening primere na podmnožice in na vsaki naredi učenje. S tem pridobimo veliko hitrejše učenje na veliki množici podatkov.

Tipična uporaba: Kot podmnožice v mini-batchu se tipično uporabljajo velikosti: 64, 128, 256, 512 ter 1024. Če je $m < 2000$ se še vedno upoprablja gradient descent in v primeru, da izberemo Mini-batch z velikostjo 1, se izvaja Stochastic gradient descent.

Adam optimizer. Trenuten algoritem, ki se je izkazal za zelo uporabnega pa je Adam optimization [7]. Osnovna ideja deluje še vedno, kot gradient descent vendar upoštevamo tudi lasnosti momentum gradient descenta in RMS prop. Oba nam dodelita vsak svoj moment. Adam pa uporablja kombinacijo obeh. Prikaz na sliki 3.2.



Slika 3.2: Prikaz učenja gradient descenta ter Adam optimizer-ja.

Momentum gradient descent upošteva še moment, ki ga izračuna z eksponentno uteženega povprečja (Exponentially weighted average).

Za izračun momenta potrebujemo β_1, ki je običajno $\beta_1 = 0,9$ in unčno oceno α. Na začetku pa tudi inicializaeramo $VdW = 0$, ki je moment za W, ter $Vdb = 0$, ki je moment za b. Nato pa jih posodabljamo po formuli 3.12, kjer Vd predstavlja VdW ali Vdb, ter d odvod W ali b.

$$Vd = Vd \times \beta_1 + (1 - \beta_1) \times d \tag{3.12}$$

Upoštevamo tudi moment RMSprop algoritma, ki deli učno oceno s kvadriranim eksponentno razpadajočim povprečjem gradienta. Prav tako potrebujemo tudi sprejemlivke β_2, SdW ter Sdb, ki jih na začetku inicializeramo z $SdW = 0$ in $Sdb = 0$. Nato pa ju posodabljamo po formuli 3.13, kjer Sd predstavlja SdW ali Sdb, ter d odvod W ali b.

$$Sd = \beta_2 \times Sd + (1 - \beta_2) \times d^2 \tag{3.13}$$

Zaradi začetne napake v eksponentnem uteženem povprečju, tako v Momentum gradiend descenta kot v RMSprop, je treba ocene popraviti.

$$Vd_{corr} = \frac{Vd}{1 - \beta_1^t} \tag{3.14}$$

$$Sd_{corr} = \frac{Sd}{1 - \beta_2^t} \tag{3.15}$$

Ko imamo popravljene rezultate, lahko z α posodobimo uteži 3.16, 3.17. Upoštevamo pa tudi, da je lahko SdW ali Sdb enaka 0. Zato dodamo še $\epsilon = 10^{-8}$.

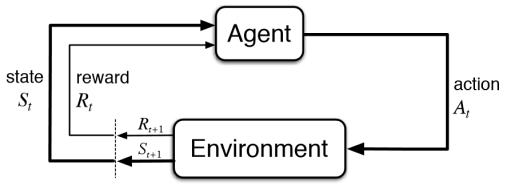
$$W = W - \alpha \times \frac{VdW_{corr}}{\sqrt{SdW_{corr} + \epsilon}} \tag{3.16}$$

$$b = b - \alpha \times \frac{Vdb_{corr}}{\sqrt{Sdb_{corr} + \epsilon}} \tag{3.17}$$

Običajno za parametre $\beta_1 = 0,9$, $\beta_2 = 0,999$ in $\epsilon = 10^{-8}$ ne potrebujemo optimizacije. Treba pa je najti dobro α konstanto. Adam optimizer se je izkazal odlično v praksi in je skoraj vedno boljši od gradient descenta.

3.2 Spodbujevano učenje

Čez čas se je strojno učenje razvijalo. Nastajalo je vse več težav, ki jih delimo na tri sklope. Nadzorovano učenje, kjer imamo učne podatke in znane rezultate, kar se želimo naučiti. Nenadzorovano učenje kjer so samo podatki, s katerih želimo ugotoviti odvisnosti, vendar rezultati niso znani. Ter spodbujevano učenje oziroma reinforcement learning, pri katerem želimo model naučiti obnašanja v določenem okolju 3.3.



Slika 3.3: Simbolični prikaz spodbujevanega učenja.

Zakaj spodbujevano učenje? Izkazalo se je, da zbiranje podatkov in izdelava lasnosti, zna biti za določene primere, izredno težko opravilo, hkrati pa

States:	Action 1	Action 2	....	Action m
State 1	Q(s1, a1)	Q(s1, a2)	....	Q(s1, am)
State 2	Q(s2, a1)	Q(s2, a2)	....	Q(s2, am)
State 3	Q(s3, a1)	Q(s3, a3)	....	Q(s3, am)
....	....	....	....	....
State n	Q(sn, a1)	Q(sn, a2)	....	Q(sn, am)

Tabela 3.1: Q-learning matrix

želimo spodbujati simulacijo enakega učenja, kot ga uporabljamo sami. Zato je leta 1957 Belman razvil izračun za optimizacijo, ki posodablja nagrade čez čas.

3.2.1 Q-learning

Po Bellmanovem izračunu optimalnosti je bil razvit tudi eden prvih algoritmov, imenovan Q-learning [15]. Njegova ideja je optimizirati nagrado akcije a_t v trenutnem stanju s_t, kjer upoštevamo nagrade naslednjih dogodkov. To storimo tako, da sprva naredimo Q matriko, v kateri stanje opisuje na določeno vrstico, vsaka akcija pa je stolpec zase. Prikaz Q-learning matrike 3.1.

Tako Q-lerning matrika vsebuje celice, ki so predstavljene s stanjem s_t ter a_t, le to pa želimo posodabljati. Posodabljanje izračunamo po računu 3.18.

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha * (r_t + \gamma * max(s_{t+1}, a)) \tag{3.18}$$

Opis uporabljenih parametrov:

- s_t - je vektor, ki opisuje trenutno stanje na katerem smo,
- a - vektor, ki vsebuje pričakovane nagrade za vsako akcijo posebej v nekem stanju,
- a_t - pričakovana nagrada izbrane akcije v stanju t,

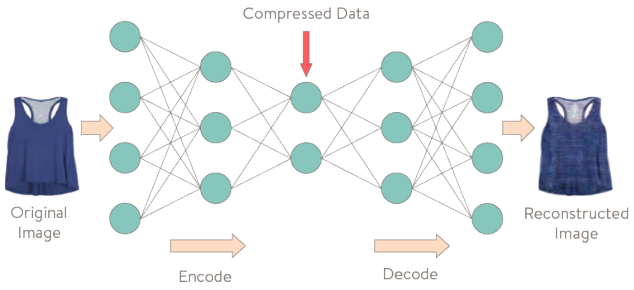
- r_t - dobljena nagrada, ki jih dodamo najvišjo nagrado akcije naslednjega stanja s_{t+1},
- γ - skrbi za odvisnost trenutne nagrade od nadaljnih nagrad.

Pri tem algoritmu pa nastane težava. Namreč, v določenih primerih število stan eksplodira, nakar algoritma ni mogoče izvajati.

3.2.2 Deep Q Network

Prišli smo v čas, kjer imamo na voljo ogromno podatkov, zato so dobile svoj namen tudi nevronske mreže, ki so se izkazale za izjemno uporabne pri interpretaciji kompleksnih zadev (slik, besed ...). V Deep mind-u so preoblikovali Q-learning algoritem, tako da namesto hranjenja vsakega stanja posebaj, uporabijo nevronske mreže, ki tako interpretirajo stanje s_t, hkrati pa izvede tudi klasifikacijo najboljše akcije a.

Za dokaz interpretacije je Ian Goodfellow izdelal Encoder-Decoder nevronsko mrežo [5], ki prikazuje interpretacijo (Encoder) v manjši vektor *CompressedData*. S katerega je tudi mogoče dobiti osnovno sliko (Decoder), kot je prikazano v 3.4.

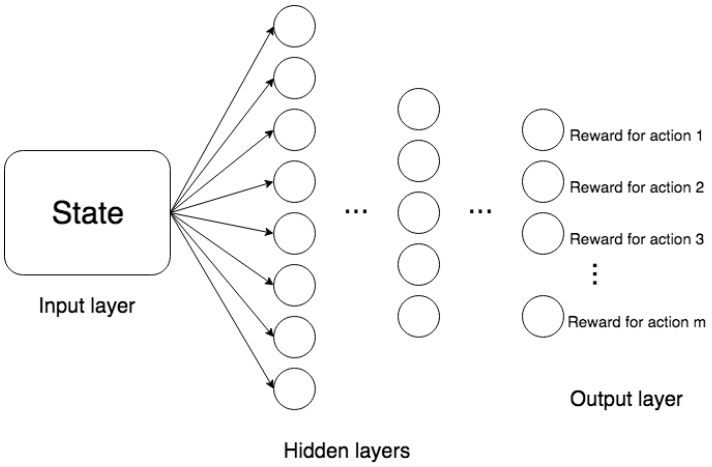


Slika 3.4: Simbolična slika s prikazom Encoder-Decoder interpretacijo nevronskih mrež.

Na podoben način lahko interpretiramo tudi stanje s Q-learning algoritmom.

Deep-mind je tako zasnoval osnovno idejo, da algoritem Deep Q Netowrk [10] zajame sliko stanja, jo interpretira in napove najboljšo akcijo.

Uporabili so Konvolucijsko nevronsko mrežo, po kateri lažje interpretiramo sliko, le to interpretacijo združimo v vektor in jo podamo v polno povezano nevronsko mrežo, ki napove pričakovano nagrado vsake akcije. Da bi posplošili delovanje DQN algoritma s Q-learning algoritmom, si ga lahko predstavljamo kot 3.5.



Slika 3.5: Prikaz delovanja DQN.

Vendar to ne zadostuje, da lahko DQN algoritem učimo. Moramo sestaviti podatke, s katerimi bomo učili svoj sistem in določiti strategijo učenja. Za to je Deep-mind predlagal še dva postopka.

Replay memory uporabljamo za hranjenje preteklih stan, skozi katere učimo svoj model. Določiti je potrebno velikost našega Replay memory-ja, ki jo bomo označili kot $replay_{size}$. V prvi predstavitvi sistema, so mu določili standardno velikost $replay_{size} = 1.000.000$. Podatke shranjujemo v obliki: $(state_t, action_t, reward_t, state_{t+1}, done)$.

ϵ **greedy policy** nam omogoča stabilnost učenja. Njegova ideja je, da ima sistem v prvih fazah učenja vse akcije naključne, ta pa se zmanjšuje z vsako iteracijo. S tem je mogoče doseči stabilno učenje, hkrati pa dodamo modelu možnost, da tudi sam poskuša odigrati poteze, s katerih se tudi uči.

Tako uporabljamo tri parametre:

- ϵ - določa verjetnost naključne akcije,
- ϵ_{min} - dodeljuje $\epsilon - u$ njegovo najmanšo vrednost (običajno 10%),
- ϵ_{decay} - nam določa hitrost zmanševanja ϵ-a.

Za vsako akcijo se *epsilon* zmanjša po $\epsilon = \epsilon * \epsilon_{decay}$, vendar ϵ ne zmanjšamo pod ϵ_{min}!

Akcija. Ko dobimo, neko stanje $state_t$, mu moram zagotoviti akcijo. Zanjo pa je treba upoštevati, odločitev našega DQN algoritma ter ϵ greedy policy. To lahko storimo z enostavnim pogojem 2.

Algorithm 2 Delovanje akcij z ϵ-greedy policy

```
function ACTION(state_t)
    action ← argmax(model.predict(state_t))
    if random.rand() < ε then
        return possible random action
    else
        return possible action
    end if
end function
```

Učenje v DQN algoritmu se izvaja po vsaki akciji, po ideji Mini-batch. Vsakič tako vzamemo z Replay memory-ja $batch_{size}$ položajev, jih shranimo v $batch_{mini}$, na katerih izvedemo gradient descent. Naša stanja so shranjena kot $(state_t, action_t, reward_t, state_{t+1}, done)$.

Za izvedbo učenja bomo tako uporabili vsa stanja *state* z $batch_{mini}$, kot naše inpute. Napovedovati pa želimo posodobljene y, kot opisuje algoritem 3.

Algorithm 3 Delovanje algoritma Deep Q Network

procedure Deep Q Network Training($batch_{mini}$)
  $X \leftarrow list([])$
  $y \leftarrow list([])$
  for $state$, $action$, $reward$, $state_{next}$, $done$ in $batch_{mini}$ **do**
    if $done = True$ **then**
      $target \leftarrow reward$
    else
      $target \leftarrow reward + gamma * max(model.predict(state_{next}))$
    end if
    $yi \leftarrow model.predict(state)$
    $yi[action] \leftarrow target$
    $X.append(\ state)$
    $y.append(\ yi)$
  end for
  $model.fit(X, y)$
end procedure

Poglavje 4

Gradnja modelov

Zdaj, ko imamo narejeno igro Black jack, za izvajanje simulacij, znanje o nevronskih mrežah in o algoritmu DQN, ga lahko izkoristimo in naredimo sistem, ki se bo sam naučil zakonitosti igre. Kot že vemo, je vsaka igra sestavljena iz dveh delov. Stavljenje ter odločitev pri samem igranju kart. Tako bomo v sistemu gradili dva modela: igralni model ter stavni model.

Ko gradimo modele, se moramo odločiti, kako bomo podatke postavili, kakšno strukturo bo imel naš učni algoritem in kako bomo predstavili nagrade. Igralni model se bo poskušal naučiti želeno tabelo, ki nam jo priskrbi igralnica 1.1. Stavni model pa bo na podlagi $count_{true}$ ter trenutnega denarja $money_{current}$, skušal ugotoviti kdaj in koliko se nam splača vložiti v igro, da bo dobičkonosna.

Oba modela bomo učili z uporabo prilagojenega Deep Q-Newtork algoritma.

4.1 Knjižnice

Za izdelavo in učenje modelov so v projektu uporabljene naslednje zunanje knjižnice.

Keras je knjižnica za hiter razvoj nevronskih mrež, ki deluje na Tensorflow podlagi. Njegova lasnost je graditi računski graf, to bo v našem primeru nevronska mreža, s katerega nam lahko vrne gradient vsakega elementa v njej. Prav tako pa vsebuje vse obstoječe metode, ki se uporabljajo za vse podskupine nevronskih mrež.

SKlearn knjižnica nam omogoča lahko uporabo in implementacijo učnih algoritmov, kot so: SVM, Ridge, Random Forest ..., prav tako pa vsebuje algoritme, ki nam omogočijo obdelavo podatkov tako za kategorične parametre kot tudi za številčne. Z nje je uporabljen One-hot-label-encoder.

Numpy je namenjen za vse ostale matematične operacije, ki se izvajajo izven modela.

4.2 Igralni model

Prvi model, ki ga potrebujemo, je igralni model. Ta nam bo vodil odločitve v igri in poskušal doseči raven, katero lahko dobimo z uporabo tabele 1.1. V sami igri je razvidno s tabele, da se odločamo na podlagi svojih kart, delivčeve karte in trenutne vrednosti naših kart. Tako se tudi igralni model odloča z teh parametrov. Napovedoti želimo naslednje odločitve: Ne povleci, Dodatna karta, Podvojitev, Predaja Deljenje.

Čemu se želimo približati oziroma kakšne so naša pričakovanja? Ker vemo, da je tabela odločitev, ki jo ponuja igralnica, najbolj idealna, se poskušamo približati njenim rezultatom, ki so $win_{rate} = 49{,}7\ \%$. Vemo pa tudi, da običajni Black jack igralci povprečno odigrajo z $win_{rate} = 48\ \%$. Pri tem je upoštevano, da ne igrajo z konstantno stavo.

4.2.1 Predstavitev podatkov

Kako se model uči, je zelo odvisno, kako pripravimo podatke. V našem primeru imamo tako kategorične sprejemlivke (karte) kot številčne (vrednost

kart). Zato je treba te informacije preoblikovati v normaliziran opis.

Za kategorične spremenljivke se običajno uporablja LabelEncoder ali One-hot vektor. V našem primeru bomo opisovali karte z One-hot-label-encoderjem.

Tako predstavimo vsako karto kot vektor velikosti 13, kjer vsak položaj predstavlja svojo karto. Vsaka karta ima na položaju le nje zapisano 1 vsi ostali položaji pa so zapolnjeni z 0.

Primer karte 10 v vektorskem zapisu: $[0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0]$.

Odločiti se je bilo treba tudi koliko kart bo model sprejel. V našem primeru je to 8 kart. Tako v primeru, da imaš samo 2 karti, le te predstavimo z One-hot vektorjem ostale pa z ničelnim vektorjem velikosti 10.

Na enak način predstavimo tudi delivčevo karto.

Potrebujemo samo še številčno predstavitev vrednosti kart. To dodamo za lažje odločanje modela. Vemo pa tudi, da je razpon vrednosti med 4-21. To vrednost normalizeramo s 17, namreč to je vrednost, ki jo upošteva delivec med svojimi odločitvami.

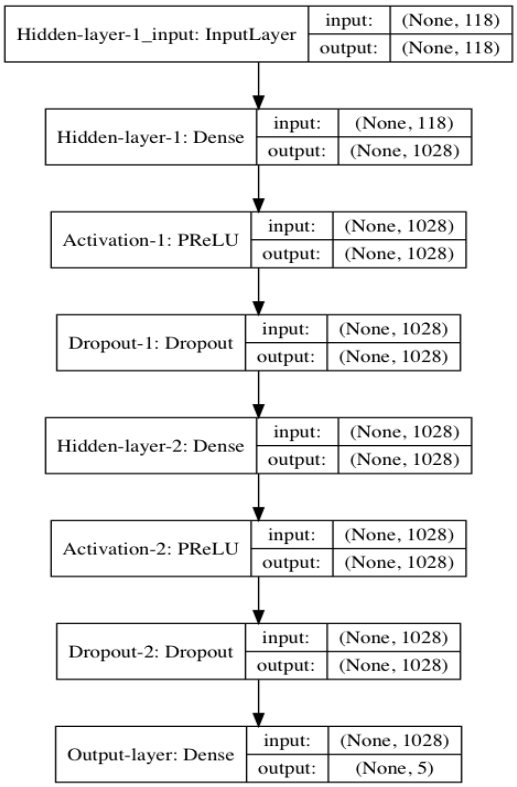
4.2.2 Model

Trenutno imamo že narejeno predstavitev podatkov za igralni model. Treba je še zasnovati strukturo nevronske mreže, ki bo skrbela za odločanje.

Uporabljena je bila, tako imenovana, plitva nevronska mreža (shallow neural network). To je poimenovanje za manjše nevronske mreže, ki imajo le nekaj skritih nivojev. Naš model vsebuje vnosni nivo, 2 skrita nivoja ter končni nivo, kot je prikazano na 4.1.

Kot začetni inicializator uteži, bomo uporabili Xaviar-initializer.

Zdaj, ko je model postavljen, je treba še določiti funkcijo napake ter optimizator, ki jo skuša minimizirati. Ker imamo na končnem nivoju linearno aktivacijo, lahko uporabimo kar srednjo kvadratno napako, ker skušamo napovedati, kakšen bo naš prihodek po odigrani stavni igri. Da bi le to napako minimizirati, pa uporabimo Adam-optimizer.



Slika 4.1: Game decision model

Osnovni parametri igralnega modela	
Parameter:	Vrednost:
Learning rate	0,01
Gamma	1,0
Over time gamma	0,99
Clipping value	0,5
Positive reward addition	0,0
Dropout rate	0,0
Epochs	50
Replay memory size	20000
Min replay memory size	3000
Mini-batch size	32
Start epsilon	1,0
Epsilon decay	0,99994
Min epsilon	0,1

4.2.3 Nagrade

Za uspešno učenje je treba tudi dobro nastaviti nagrade akcij. Tu imamo na voljo kar nekaj načinov, kako predstaviti dobljene nagrade in jih nato napovedovati.

Upoštevati želimo, da se ne ocenjuje vsaka pod igra (ena stavna) zase, temveč vsaka pod igra vpliva na celotno igro (1.000 stav). Tako za posodobitev trenutnih nagrad dodamo parameter $gamma_{ot}$. $gamma_{ot}$ nam bo omogočal, tako prenos nagrad s končne stave vse nazaj do nagrad začetne stave, kot tudi dodeljeval vpliv naslednje nagrade na trenutno.

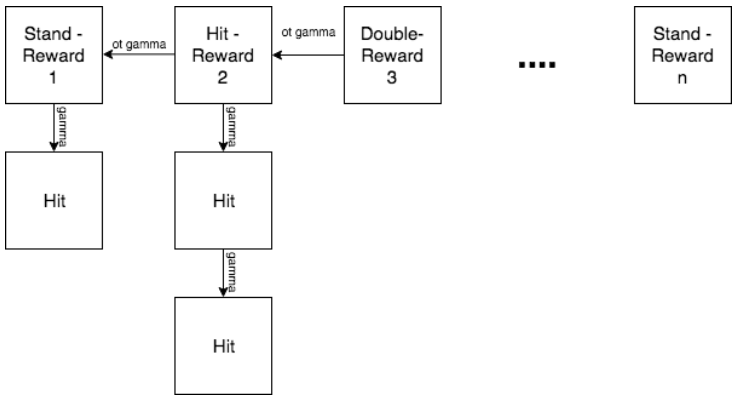
Ker pa igralni model odigra eno ali več akcij v eni stavi, je treba dodeliti tudi prenos posodobljene nagrade do začetne akcije. To nam bo omogočal še en dodaten parameter *gamma*, ki nam prav tako omogoča prenos nagrade od končne akcije do začetne. Kot kot tudi določal vpliv naslednje nagrade na trenutno. Prikaz nagrad je na sliki 4.2.

Pri samih nagradah bi tudi želeli povečati razmik med negativnimi in pozitivnimi igrami. Uporabimo dodaten parameter, ki nam opisuje dodatno nagrado $additional_{reward}$. Dodana je izbira, da je lahko $additional_{reward}$ tako pozitiven, kot negativen. V primeru, da je pozitiven se ta prišteje pozitivnim igram in če je negativen se prišteje negativnim igram. Vsako nagrado igre i lahko označimo kot $reward_i$, katera se posodobi po enačbi 4.1.

$$reward_i = reward_i + \begin{cases} additional_{reward} & reward_i > 0 \wedge additional_{reward} > 0 \\ additional_{reward} & reward_i < 0 \wedge additional_{reward} < 0 \\ 0 & otherwise \end{cases} \tag{4.1}$$

4.2.4 Spomin

Vsako potezo igre shranjujemo v obliki: trenutno stanje, akcija, nagrada, naslednje stanje, konec, naslednje polje. Ko se igra zaključi pa je potrebno popraviti obliko shranjenih korakov, namreč v primeru Delitve se zgodita



Slika 4.2: Predstavitev dodeljevanja nagrad za Game decision model.

dva položaja, ki oba vplivata na rezultat trenutne stave. Zato popravimo trenutno obliko v (trenutno stanje, akcija, nagrada, seznam naslednjih stanj, konec). Na koncu osnovne igre (1.000 stav) popravimo tudi končne nagrade z $gamma_{ot}$ po enačbi 3.18. Tako imamo na koncu nagrade predstavljene kot (trenutno stanje, akcija, nagrada, seznam naslednjih stan, konec) s katerega lahko začnemo učiti svoj model. Pozor: ena osnovna igra (1.000 stav), se doda v spomin šele po samem koncu vseh stav!

4.2.5 Učenje

Zdaj ko imamo postavljeno igro Black jack ter model, ga lahko prilagajamo igri. To naredimo, da po vsaki narejeni odločitvi izvedemo Mini-batch gradient descent s podatki od že odigranih iger, ki so shranjeni v spominu. Na začetku želimo raziskovati delovno površino učne težave. Zato nam $\epsilon-greedy$ strategija določa, kolikšen odstotek iger bomo odigrali naključno.

Tako naključne akcije kot tudi akcije modela se izvedejo samo nad možnimi akcijami, če akcija ni možna, jo prezremo!

Sprva se izvedeta dve popolnoma naključni igri, nato pa se začne ϵ zmanjševati do ϵ_{min}. Osnovni model učimo 100 iger po 1.000 stav.

4.2.6 Hiper-parameter iskanje

Ker je igralni model kar kompleksen, je zanj težko predvideti, s katerimi parametri bo dosegel najbolšje rezultate na testiranju. Zato se uporabljajo tehnike Hiper-parameter iskanja.

Včasih so se parametri iskali z izčrpnim iskanjem. Za vsak iskan parameter si izbral določene iskalne vrednosti, nato pa si jih sistematično pregledoval in primerjal rezultate med saboj. Ker je bil ta način za velika iskanja prepočasen, se je začela uporabljati naključna tehnika. V njej se določi vsakemu parametru iskalni prostor s katerega naredimo naključne parametre modela. Iskalni prostor lahko ročno spreminjamo med iteracijami.

Populacijsko iskanje Naključno iskanje se je izkazalo za solidno. Vendar ga je Deep-mind ekipa izboljšala s populacijskim iskanjem oziroma genetskim iskanjem [6]. Uporabili bomo le funkcionalnost iskanja njihovega algoritma. Prenosno funkcijo uteži pa zanemarili, ker je v našem iskanju všteta tudi dolžina učenja vsakega modela.

Ideja algoritma izhaja iz naključnega iskanja in približevanja populacije najboljšemu osebku. Algoritem vsebuje populacijo *population* osebkov *worker*. Sprva definirajmo osebek.

Vsak osebek predstavlja kombinacijo hiper-parametrov (learning rate, gamma, over time gamma, clipping value, positive reward addition, dropout rate in epochs), model nevronske mreže, funkcionalnost učenja nevronske mreže ter rezultate testiranja. Osebki pa vsebujejo tudi metodi *exploit* ter *explore* s katerimi raziskujemo parameterski prostor.

Metoda *explore* nam omogoča dodeljevanje šuma vsakemu hiper-parametru. Velikost šuma na vsakem hiper-parametru pa določimo s pomočjo standardne deviacije, ki je definirana za vsak hiper-parameter drugače.

Metoda *exploit* prejme celotno populacijo. Iz celotne populacije izbermo naključen osebek, katerega rezultate testiranja primerjamo z rezultati testiranja trenutnega osebka. V primeru, da so rezultati naključnega osebka boljši, prepišemo trenutnem osebku njegove hiper-parametre z hiper-parametri na-

ključnega osebka in kličemo metodo *explore*. V primeru, da so rezultati trenutnega osebka boljši od naključnega obdržimo trenutni osebek.

Tako na začetku naredimo populacijo *population* naključnih osebkov *worker* (naključni hiper-parametri) v določenem parameterskem prostoru 4.2.6 in definiramo število iteracij *iterations* algoritma. V prvi iteraciji zgradimo modele osebkov, jih učimo in testiramo (tako kot pri naključnem iskanju). Po prvi iteraciji pa algoritem kliče metodi *exploit* ter *explore* in uči ter testira samo novo narejene osebke. Prikaz delovanja algoritma je predstavljen v algoritmu 4.

Algorithm 4 Delovanje algoritma Populacijsko iskanje

```
procedure POPULATION BASED TRAINING(population, iterations)
    for iteration in iterations do
        for worker in population do
            if iteration Not 0 then
                new ← worker.exploit( population)
                if new is True then
                    worker.explore()
                end if
            end if
            worker.create model()
            result ← test model( worker)
        end for
    end for
end procedure
```

Celotno iskanje smo tako izvedli na spodnjem parametrskem prostoru z velikostjo populacije $population_{length} \leftarrow 15$ in $iterations \leftarrow 30$.

Parameterski prostor	
Parameter:	Prostorske omejitve:
Learning rate	[0,001; 0,0001]
Gamma	[0,5; 1,0]
Over time gamma	[0,0; 1,0]
Clipping value	[0.1; 1,5]
Positive reward addition	[-0,5; 0,5]
Dropout rate	[0,0; 0,7]
Epochs	[50; 150]

4.3 Stavni model

Kljub temu, da je pri sami logiki, kako igrati dobičkonosno igro, večina dela s stavnim odločanjem, v našem primeru ne bo tako. Namreč, pridobljeno znanje bomo uporabili in ga poskušali izboljšati z algoritmom Deep Q-Network. Tako nam bo stavni model reševal težavo, koliko staviti v trenutni situaciji, da bo igra čim bolj dobičkonosna.

Model učimo na dveh parametrih: $count_{true}$ ter $money_{current}$.

4.3.1 Predstavitev podatkov

V Stavnem modelu uporabljamo že kot zgoraj zapisano $count_{true}$ ter $money_{current}$ parametra, po katerima se odloča kolikšno stavo vplačati v določeni situaciji.

Tako kot v 2.2 je tudi tu $count_{true}$ omejen na interval $\in [-15, 15]$, s katerim zajamemo 99,73 % vseh vrednosti. Upoštevamo pa tudi $money_{current}$, ki tvori našo trenutno situacijo čipov. Oba parametra predstavimo številčno. Potrebna je še normalizacija. $count_{true}$ parameter normaliziramo s 15. Tako dobimo njegovo predstavitev med $\in [-1, 1]$. Za $money_{current}$ bomo uporabili začetni vložek $money_{start}$ in ga normalizirali po formuli 4.2. Tako normaliziran $money_{current}$ sega med $\in [-1, \infty]$.

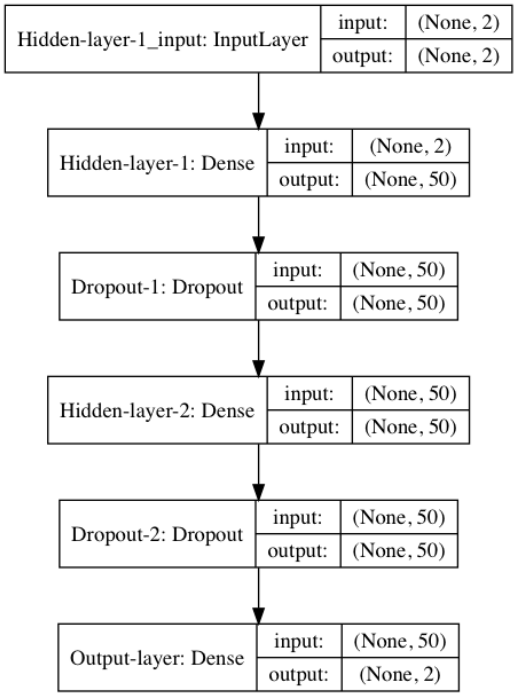
$$money_{normalized} = \frac{money_{current}}{money_{start}} - 1 \tag{4.2}$$

Kot vnos v nevronsko mrežo uporabimo zgoraj zapisan parametra, po katerih model napoveduje vplačila: $10, 100$. Njegovo napoved bomo nato uporabili kot vplačilo čipov ali pa večkratnik *multiplier* Enotne stave $unit_{bet}$ 4.3.

$$current_{bet} = unit \times multiplier \tag{4.3}$$

4.3.2 Model

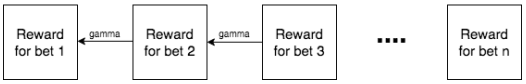
Uporabljena je Plitva nevronska mreža, katera vsebuje dva vnosa, dve skriti ravni s 50 nevrino, ter dva izhoda. Za samo regularizacijo sta bila dodana tudi Dropout nivoja na obeh skritih nivojih. Prikaz modela na sliki 4.3.



Slika 4.3: Predstavitev stavnega modela.

4.3.3 Nagrade

Pri stavnem modelu je dodeljevanje nagrad nekoliko lažje 4.4, kot v igralnem modelu. Namreč tu ne skrbimo za dodatne vmesne akcije.



Slika 4.4: Predstavitev stavnih nagrad.

Tu prav tako želimo, da se ena stava ne upošteva kot samostojna stava, temveč kot del celote ($n_{max-bet}$ stav). Prenos in vpliv nagrad nam bo predstavljal parameter *gamma*. Ker želimo, da ima začetna stava vpliv vse do končne stave, lahko le to izračunamo s spodnjim izračunom, v katerem upoštevamo $n_{max-bets}$, kot število stav v eni igri.

$$gamma = 1 - \frac{1}{n_{max-bets}} \tag{4.4}$$

Zaradi širokega spektra nagrad, želimo vsako nagrado normalizirati, s tem tudi zmanjšamo razpon napake pri učenju. Za normalizacijo bomo uporabili parameter $reward_{norm} = 25$. Vsako nagrado bomo tako pred shranjevanjem normalizirali po enačbi 4.5, to bomo skušali napovedati pri samem učenju.

$$reward_{final} = \frac{reward}{reward_{norm}} \tag{4.5}$$

V spominu stavnega modela nato hranimo stanja v obliki: trenutno stanje, akcija, normalizirana nagrada, naslednje stanje, konec igre. Konec igre je označen kot $True$ samo, če nam zmanjka čipov ali pa smo odigrali $n_{max-bet}$ stav.

4.3.4 Učenje

Modelu je treba dodati še nekaj učnih parametrov, nato ga lahko začnemo učiti. Učimo ga s pomočjo Mini-batch Adam optimizatorjem in $\epsilon - gredy$ strategijo, enako kot pri Igralnem modelu z drugačnimi parametri.

Osnovni parametri stavnega modela	
Parameter:	Vrednost:
Learning rate	0,0005
Mini-batch size	8000
Replay memory size	20000
Gamma	0,8
Epsilon start	1,0
Epsilon end	0,1
Epsilon decay	0,95
Reward std	25

Kar zadeva odločanje med igro, uporabljamo model odločanja, ki nam ga priskrbi igralnica 1.1. Na tak način si zagotovimo določene verjetnosti, ki jih imamo v posamezni situaciji $count_{true}$.

Poglavje 5

Testiranje modelov

Sprva bomo spoznali osnovne in željene rezultate, ki se jim želijo naši modeli približati oziroma, ki jih želijo izboljšati. Ko imamo naučene modele, želimo vedeti, kako uspešno se modeli odražajo s svojimi odločitvami med učenjem in testiranjem. Pregledali smo rezultate Hiper-parameter iskanja in jih primerjali z osnovnimi igralnimi modeli. Na koncu pa primerjali rezultate kombinacije najboljših modelov z osnovnim in željenim odločanjem.

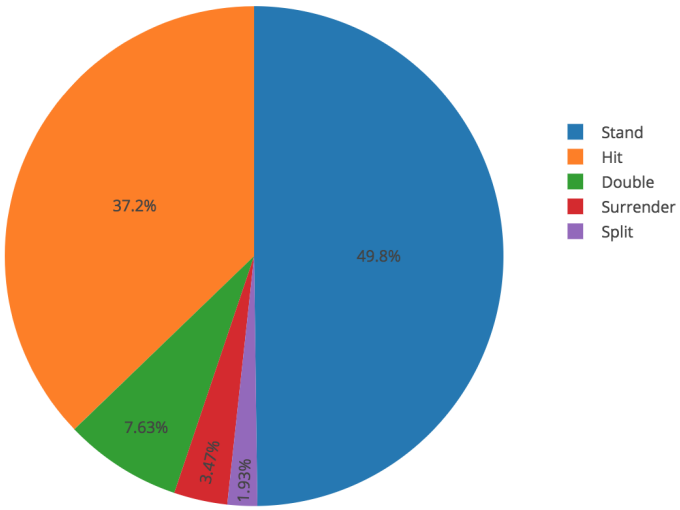
5.1 Base-line

Za sam začetek si bomo pogledali, kako se model 1.1 obnese v Monte-carlo simulaciji. Hranili smo podatke o vsaki stavi in odločitvah modela, ki so analizirane za boljšo predstavo, kako mora naš model delovat.

Poglejmo si najprej distribucijo akcij, ki jih naredi željen model. V tortnem diagramu 5.1 je prikaz procentualno razmirje med izvedenimi akcijami.

Kot vidimo, je večina odločitev modela akcija Ne povleci in to kar 50 % časa. Za tem si sledijo Dodatna karta, Podvoji, Predaj ter Deljenje. Zanima pa nas tudi, kakšna je pričakovana vrednost čipov v določenem trenutku. To nam prikazuje graf 5.2.

Več stav, kot odigramo, večji je razpon pričakovane vrednosti. Največji preskok se naredi v prvih 100 stavah, nato se standardna deviacija razpona



Slika 5.1: Base-line distribucija akcij.

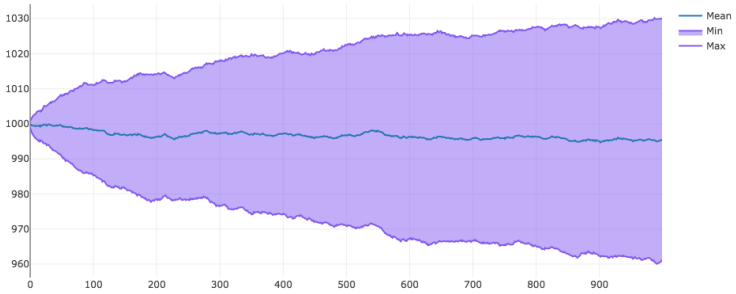
logaritemsko zmanšuje. Iz tega lahko sklepamo, da se igra v 100. stavah normalizira oziroma se spreobrne. Samo povprečje željenega modela pa je dokaj stabilno.

5.2 Igralni model

Vsak model, ki ga zgradimo, hranimo njegove rezultate na vsaki iteraciji učenja. Pri Igralnem modelu bomo pozorni predvsem na dve distribuciji, končen denar ter distribucijo akcij, ki jih je model naredil na trenutni iteraciji. Ker pa vemo, kakšen je naš želeni model, bomo poskušali obe omenjeni distribuciji zgoraj približati željenim.

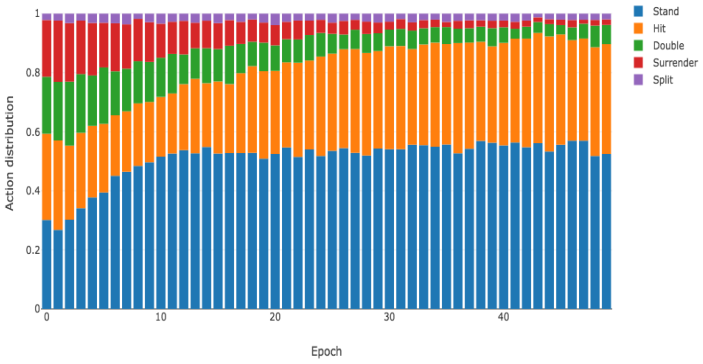
5.2.1 Učenje modela

Zanima nas tudi, kako se model odraža med samim učenjem. Ko gledamo rezultate treninga je pomembno vedeti, da so na začetku učenja vse poteze naključne, ta naključnost se na koncu zmanjša do ϵ_{min}! To pomeni, da na treningu ne bomo dosegali željenih rezultatov. Za točne rezultate je treba model testirati, brez naključnih potez, po koncu učenja.



Slika 5.2: Base-line pričakovanega denarja.

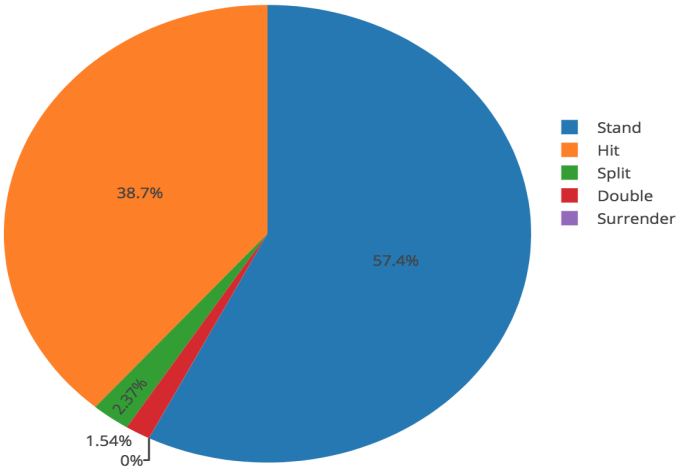
Poglejmo kako se spreminja distribucija akcij skozi čas 5.3, iz katere je razvidno približevanje željeni distribuciji akcij 5.1.



Slika 5.3: Prikaz spreminjanja distribucije akcij na treningu.

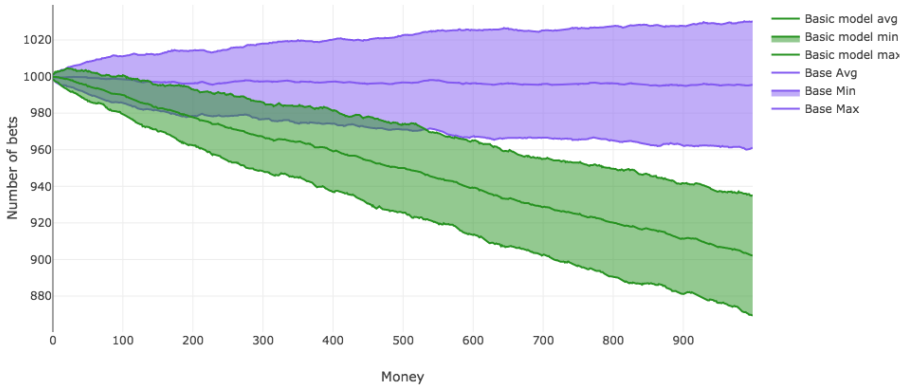
5.2.2 Testiranje modela

Ko imamo model naučen, želimo vedeti, kako se bo odražal v realnem svetu. Vemo pa tudi, da je pri učenju na koncu uporablja 10 % naključnosti. Najprej si za predstavitev poglejmo, kakšno distribucijo akcij doseže model, ko nima naključnih potez 5.4.



Slika 5.4: Testiranje distribucije akcij.

Razvidno je, da se je osnovni model naučil igrati igro in uporabljati možnosti: Ne povleci, Dodatna karta, Podvoji ter Deljenje, možnost Predaj pa je zanemaril. Na grafu 5.5, kjer je narejena primerjava osnovno naučenega modela ter odločitev, ki nam jo ponuja igralnica 1.1, se model ni naučil želene distribucije, vendar se ji približuje.



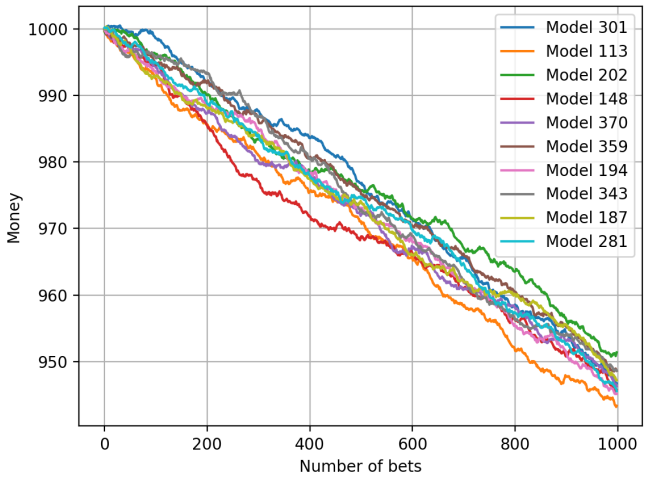
Slika 5.5: Primerjanje rezultatov modela z željenimi rezultati.

Izračunajmo še kakovost P_{win} osnovnega modela po formuli 2.8. Kjer je naš $money_{end} = 905$ in $money_{start} = 1.000$. Tako dobimo, da je naš osnovni

model igral z P_{win} =45,25 %. Izračunamo lahko tudi procentualen dobiček po formuli 2.4, ki znese -5,5 % povprečne izgube, glede na eno igro (1.000 stav).

5.3 Populacijsko iskanje

Po tednu dni učenja modelov s Populacijskim iskanjem, lahko primerjamo rezultate testiranih modelov med saboj.

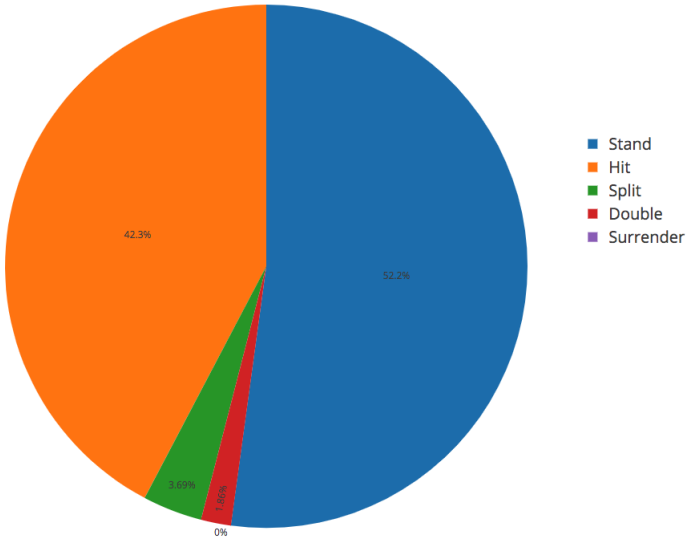


Slika 5.6: Prikaz 10 najboljših modelov Hiper-parameter iskanja.

Na grafu 5.6 je prikazano 10 najbolše modelov. V nadaljevanju bomo uporabljali prvi najbolje naučen model, si pogledali, kakšna je njegova akcijska distribucija izračunali njegovo kakovost po formuli2.8 ter povprečno dobičkonosnost po formuli 2.4.

Trenuten najbolši model je Model 202, ki je dosegel kakovost konstantne igre z P_{win} =47,55 %. Njegova povprečna dobičkonosnost doseže P_{win} =-4,9 % po vsaki igri.

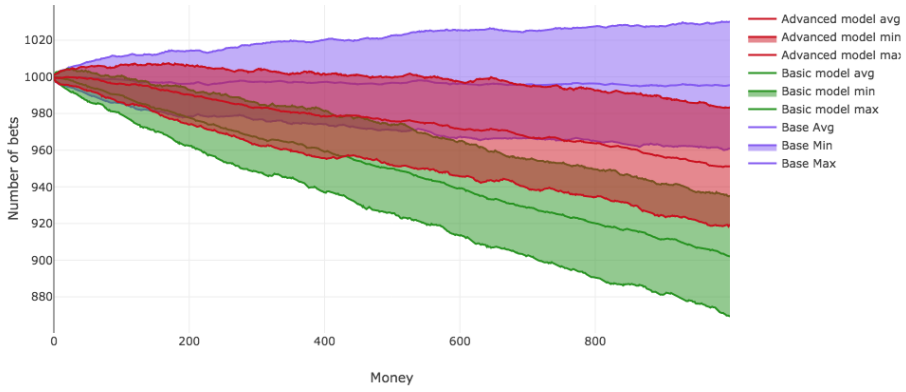
Poglejmo si še rezultate njegove distribucije akcij, ki je prikazana na grafu 5.7.



Slika 5.7: Testiranje distribucije akcij najboljšega modela v Populacijskem iskanju.

Pogledali smo tudi rezultate novega modela, v primerjavi z želenimi in osnovnimi rezultati, ki so prikazani na grafu 5.8.

V nadaljevanju bomo temu modelu dodali še stavni model in pogledali ali se da njegove rezultate s stavami izboljšati.



Slika 5.8: Primerjanje rezultatov novega modela s starimi in željenimi rezultati.

5.4 Stavni model

Pri testiranju stavnih modelov bomo upoštevali, da uporabljamo samo brezhiben igralni model. Namreč, po nekaj poizkusih učenja stavnega modela z naučenim igralnim modelom, se je izkazalo, da teorija brezhibne igre drži, in da brez nje ne moramo doseči dobičkonosne igre.

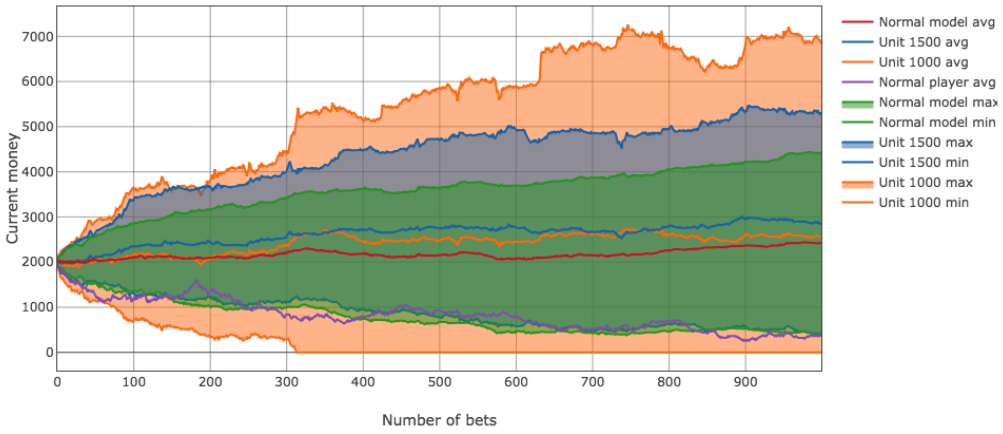
Primerjali bomo naučen stavni model, na katerem uporabimo enotsko stavljenje, ki se izračuna s formulama 2.5 ter 4.3. Tako bomo na stavnem modelu uporabili enotske stave z $unit_{devider}$ =1.000 ter $unit_{devider}$ =1.500. Rezultate bomo primerjali z ne uradnimi podatki povprečnih igralcev, ki je P_{win} =48 %. Podatki povprečnega igralca so bili narejeni s pomočjo naključne funkcije ter konstantne stave, ki je enaka 100.

Stavni modeli so bili testirani z začetnim denarjem 2.000 čipov in odigrali 50 iger po 1.000 stav.

Graf 5.9 prikazuje povprečne rezultate modelov ter pričakovan razpon čipov v določenem trenutku vsakega modela.

Zanima nas tudi povprečen dobiček iger, ki ga izračunamo po formuli 2.4.

- Naučeno stavno odločanje doseže povprečno P_{win} =21,5 % dobička.

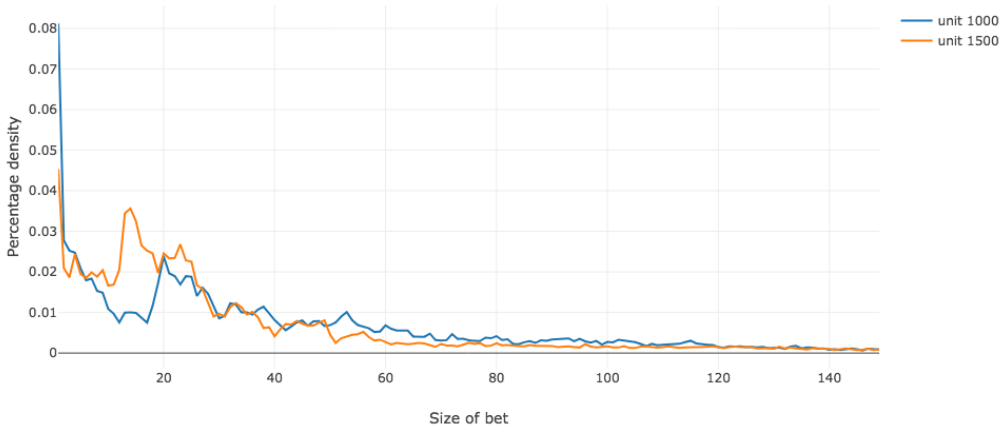


Slika 5.9: Primerjanje rezultatov stavnih modelov

- Enotsko stavljenje z naučenim modelom in $unit_{devider}$ =1.000 doseže povprečno P_{win} =27,8 % dobička.
- Enotsko stavljenje z naučenim modelom in $unit_{devider}$ =1500 pa doseže kar P_{win} =42,9 % povprečnega dobička.

Primerjajmo še procentualno distribucijo velikosti stav za vsak enotski stavni model posebaj v grafu 5.10. Modela ne bomo primerjali z naučenim modelom, ker vemo, da upošteva smo dve vrednosti. Enotski stavni model sicer napove večje stave kot 150 čipov, zaradi čistejšega prikaz.

Za omenjena modela pa velja, da je bilo največje vplačilo za Enotsko stavljenje z $unit_{devider}$ =1.000 enako 1.608 čipov, ter za Enotsko stavljenje z $unit_{devider}$ =1.500 enako 767 čipov.



Slika 5.10: Primerjanje distribucije stav enotskih modelov

Poglavje 6

Ugotovitve

Z diplomski nalogi sem želel spoznati najnovejše tehnologije učenja in iskanja umetne inteligence, kot sta Globoke Q mreže (Deep Q Newtork) ter Populacijsko učenje (Population based training). Slednje sem bolje spoznal in se jih naučil uporabljati v težavah spodbujevanega učenja.

Kar zadeva igralni model, je razvido, da model ni dosegel željenega standarda. Stabilnost učenja je vzrok zaradi kerega menim, da se model ni naučil igrati optimalne igre. Sam algoritem je bil predstavljen na Atari igrah, v katerih ne igra vloge verjetnost temveč konstantna nagrada. Tako menim, da težava nastane v spominu modela, ker se lahko zgodi, da kljub optimalni situaciji lahko le ta n krat izgubi. Če se to zgodi, se model poskuša prilagoditi rezultatom tako, da storjeno akcijo zamenja.

Težavo bi v nadaljevanju poskušal rešiti s selekcijo shranjevanja iger v spominu modela. Predlagal bi model, ki bi napovedoval, ali določeno igro shranit ali ne na podlagi trenutnih vnosov in že shranjenih iger. Nagrada modela bi bila razlika med uspešnostjo prejšnje igre in trenutne igre. Na ta način bi zmanjšal velikost spomina, hranil samo pomembne vnose in povečal stabilnost učenja.

S stavnim modelom pa sem dokazal, da je možno tako izračunati kot tudi naučiti model igrati dobičkonosno igro. Če primerjam najboljše rezultate modelov z željenimi cilji trgovanja na borzi, je algoritem dosegel abnormalne

dobičkonosne rezultate. Zato lahko zaključim, da sem s samo diplomsko nalogo zadovoljen.

Literatura

[1] The true story of the mit blackjack team. `https://www.youtube.com/watch?v=QflVqavHHM0&t=427s`.

[2] Miguel de Cervantes. *Novelas Ejemplares*. Novelas Ejemplares, 1613.

[3] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS'10). Society for Artificial Intelligence and Statistics*, 2010.

[4] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In Geoffrey Gordon, David Dunson, and Miroslav Dudík, editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, volume 15 of *Proceedings of Machine Learning Research*, pages 315–323, Fort Lauderdale, FL, USA, 11–13 Apr 2011. PMLR.

[5] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 2672–2680. Curran Associates, Inc., 2014.

[6] Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M. Czarnecki, Jeff Donahue, Ali Razavi, Oriol Vinyals, Tim Green, Iain Dun-

ning, Karen Simonyan, Chrisantha Fernando, and Koray Kavukcuoglu. Population based training of neural networks. *CoRR*, abs/1711.09846, 2017.

[7] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[8] Pengfei Liu, Xipeng Qiu, and Xuanjing Huang. Learning context-sensitive word embeddings with neural tensor skip-gram model. *Shanghai Key Laboratory of Data Science, Fudan University*, pages 1284–1290, 2015.

[9] Warren McCulloch and Walter Pitts. A logical calculus of ideas immanent in nervous activity. *W. Bulletin of Mathematical Biophysics*, 5:115–133, 1943.

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

[11] Thang Luong Hieu Pham and Christopher D. Manning. *Proceedings of the 2015 Conference on Empirical Methods in Language Processing.* 2015.

[12] Samik Raychaudhuri. Introduction to monte carlo simulation. *Winter Simulation Conference*, pages 91–100, 2008.

[13] Herbert Robbins and Sutton Monro. A stochastic approximation method. *The Annals of Mathematical Statistics*, 22:400–407, 1951.

[14] Jurgen Schmidhuber Sepp Hochreiter. Long short-term memory. *Neural Computation*, 9:1735–1780, 1997.

[15] Christopher J.C.H. Watkins and Peter Dayan. Technical note: Q-learning. *Machine Learning*, 8(3):279–292, May 1992.

[16] LeCun Y., Haffner P., Bottou L., and Bengio Y. Object recognition with gradient based learning. *Shape, Contour and Grouping in Computer Vision. Lecture Notes in Computer Science*, 1681, 1999.