UNIVERZA V LJUBLJANI
FAKULTETA ZA MATEMATIKO IN FIZIKO

Finančna matematika – 1. stopnja

Tim Kalan

Spodbujevalno učenje pri igranju namiznih iger

Delo diplomskega seminarja

Mentorica: izr. prof. dr. Marjetka Knez

Ljubljana, 2021

Kazalo

Spodbujevalno učenje pri igranju namiznih iger

Povzetek

V povzetku na kratko opišite vsebinske rezultate dela. Sem ne sodi razlaga organizacije dela – v katerem poglavju/razdelku je kaj, pač pa le opis vsebine.

Reinforcement learning in board games

Abstract

Prevod zgornjega povzetka v angleščino.

Math. Subj. Class. (2010): navedite vsaj eno klasifikacijsko oznako – dostopne so na www.ams.org/mathscinet/msc/msc2010.html

Ključne besede: Spodbujevalno učenje, Markovski proces odločanja

Keywords: Reinforcement learning, Markov decision process

1. Uvod

Namizne igre ljudje igramo že od prazgodovine. Na Kitajskem je bila igra Go znana kot ena izmed štirih umetnosti Kitajskega učenjaka poleg igranja inštrumenta s strunami, kaligrafije in slikanja. Spremljajo nas že zelo dolgo časa, zato je naravno, da jih želimo ljudje čim bolje igrati.

Z adventom računalnika in računalništva je bil ta problem postavljen v novi luči. Vprašanje ni bilo več samo, kako dobro lahko človek igra igro sam, temveč tudi do kakšnega nivoja lahko spravi računalnik. Izkazalo se je, da nam pri tem problemu (in mnogih drugih) zelo dobro koristi »umetna inteligenca« oz. metode strojnega učenja (SU). Eno izmed vej SU bomo predstavili v tem delu in pogledali, kako nam lahko pomaga pri igranju namiznih iger.

Ideja, da bi nek stroj igral igre, ni nova, in kompleksnosti takega stroja so se zavedali ljudje že pred obstojem računalnika. Za konec uvodnega dela morda zabeležimo še citat iz eseja ameriškega pisatelja in pesnika Edgarja Allana Poea, ki govori o mehaničnem igralcu šaha:

> »Če prej omenjenemu [igralcu šaha] rečemo čisti stroj, moramo biti pripravljeni priznati, da je zunaj vseh primerjav, najbolj čudovit izum človeštva.«

1.1. **Motivacija.** Spodbujevalno učenje ima zelo lepo motivacijo, in sicer izhaja iz psihologije. Znana psihologa Thorndike in Skinner, sta na živalih izvajala eksperimente; postavila sta jih v neko novo situacijo, kjer je lahko žival naredila akcijo, ki je rezultirala v neki nagradi. Ko je bila žival ponovno postavljena v to situacijo, je hitreje ugotovila, katero akcijo mora storiti, da pride do nagrade.

Koncept, ki je opisan v zgornjem odstavku, se imenuje instrumentalno pogojevanje. Z njim se srečamo tudi ljudje; tako se namreč učijo otroci, odrasli ljudje pa se bolj zanesejo na logično razmišljanje. Vseeno pa je to motiviralo utemeljitelje spodbujevanega učenja.

1.2. **Strojno učenje.** To relativno novo raziskovalno področje se deli na tri glavne veje:

- **Nadzorovano učenje** se ukvarja s tem, kako iz nekih označenih podatkov naučimo računalnik, da prepoznava razne signale (slike, govor, tekst, ...) in to znanje uporabi za razpoznavo novih, neoznačenih podatkov.
- **Nenadzorovano učenje** odstrani označevanje iz podatkov in v njih poskuša odkriti skrite vzorce.
- **Spodbujevalno učenje** se ukvarja z »učenjem iz izkušenj«.

1.3. **Struktura naloge.** Naloga je razdeljena na štiri glavne dele. Na začetku so predstavljeni osnovni koncepti spodbujevalnega učenja in nekateri glavni algoritmi s tega področja. Potem se osredotočimo na namizne igre in ob nekaj malega teorije iger povzamemo osnovne koncepte, na katere naletimo. V naslednjem odseku združimo znanje iz prejšnjih dveh in predstavimo, kako nam teorija iger pripomore pri spodbujevalnem učenju v tem konteksu. Na koncu pa so predstavljeni nekateri empirični rezultati, ki sledijo iz zgoraj navedene teorije.

Spodbujevalno učenje se ukvarja s t. i. učenjem iz interakcije oz. izkušenj. Čeprav se to na prvi pogled ne zdi kot računska metoda, pač pa stvar psihologije, bomo kmalu dognali, kako prevesti to idejo v računalniku razumljiv jezik.

2.1. **Osnovni koncepti.** V osnovi nas zanima precej preprosta stvar: kako preslikati neko opazovano situacijo v akcijo na tak način, da učenec, ki mu v spodbujevalnem učenju pravimo **agent**, maksimizira neko numerično nagrado. Pri tem ne obstaja opazovalec, ki bi agentu povedal ali pa namignil, katere akcije so dobre, to mora ugotoviti sam, s poskušanjem in napakami. V tem dejstvu se skriva bistvena razlika med spodbujevalnim učenjem in ostalimi vejami strojnega učenja.

Še ena razlika tiči v pomembnosti časa pri spodbujevalnem učenju. Pri drugih oblikah strojnega učenja se ponavadi ukvarjamo s tabelaričnimi podatki, tu pa modeliramo nek dinamičen proces, zato je naravno, da je pomemben čas. Čeprav se ga da v tem kontekstu modelirati zvezno, je za naše namene dovolj, da ga opazujemo kot diskretne točke $t \in \{1, \ldots, T\}$, kjer T označuje nek končni čas (v splošnem je lahko seveda $T = \infty$).

2.1.1. *Nagrada.* Prvi pomemben koncept pri spodbujevanem učenju je nagrada. Kot smo že zgoraj omenili, to za nas pomeni neko numerično vrednost, kjer pozitivno število indicira »pozitivno nagrado«, negativno pa »kazen«. S pomočjo tega koncepta formaliziramo *cilj* učenja. Edini cilj agenta je maksimizacija te nagrade, pri čemer je vredno omeniti, da na nagrado agent lahko vpliva samo s svojimi akcijami (ne more recimo spremeniti načina, na katerega dobi nagrado).

Posebej pomembno je na tem mestu poudariti, da akcije nimajo nujno neposredne nagrade. Le-te lahko pridejo v poljubnem kasnejšem časovnem obdobju. To je smiselno, če gledamo z vidika namiznih iger: pri šahu ne razmišljamo samo o neposrednih akcijah, temveč razvijamo neko dolgoročno strategijo, ki nas na koncu nagradi z zmago.

Zgled 2.1 (Križci in krožci)**.** *Pri tej znani otroški igri (in pri mnogo drugih namiznih igrah) premik v času pomeni igranje poteze enega od igralcev, zato je diskreten čas popolnoma zadosten. Nagrado modeliramo na preprost način: če zmagamo, prejmemo nagrado* 1*, če izgubimo pa* -1*. V vseh ostalih situacijah, torej za izenačenje in po vsaki potezi, prejmemo nagrado* 0*.*

Zavedati se moramo tudi potencialnih omejitev oz. pomanjkljivosti takega modela nagrad in ciljev.

Hipoteza 2.2 (Hipoteza o nagradi)**.** *Vse cilje je mogoče opisati kot maksimizacijo neke kumulativne numerične nagrade.*

Zgled 2.3 (Protiprimera hipotezi o nagradi)**.** *Problem je, da hipoteza dovoljuje samo enodimezionalnost:*

- *Ko kupujemo hamburger, nam je pomemben okus in cena; kaj nam več pomeni?*
- *Država želi med epidemijo ohraniti življenja in gospodarstvo; v kolikšni meri naj prioritizira ti dve katergoriji?*

Na tem mestu poudarimo še, da se da tudi v takih situacijah modelirati nagrado na zgoraj opisani način in da je ta koncept vseeno dovolj splošen, da zajame zelo velik razred problemov.

2.1.2. *Okolje.* Okolje predstavlja del našega sistema, na katerega agent nima nobenega vpliva. Funkcija okolja je, da agentu pokaže **stanje** (angl. *state*) in mu da nagrado glede na **akcijo**, ki jo prejme od njega. Če se ponovno osredotočimo na namizne igre, bi lahko rekli, da je okolje igralna plošča *in* nasprotnik - tudi nanj namreč nimamo vpliva. Okolje nam služi tudi kot sodnik akcij oz. stanj. V kontekstu programa za igranje iger torej okolje izbira nasprotnikove akcije, odloča katero stanje pomeni zmago in dodeljuje nagrade.

Zgled 2.4 (Križci in krožci)**.** *Okolje za nas pomeni* 3×3 *igralno polje* ***in*** *našega nasprotnika - to je torej človek ali nek algoritem.*

2.1.3. *Agent.* Kot smo že omenili zgoraj, je agent naš učenec. Njegov cilj je torej maksimizacija numerične nagrade, to težnjo pa dosega s pomočjo **strategije** (angl. *policy*), ki mu pove, katero akcijo naj izbere v določenem stanju. Za ocenjevanje stanja si pomaga z **vrednostno funkcijo** (angl. *value function*). Kot ime implicira, je to funkcija, ki določa vrednosti stanjem (in akcijam).

Nagrada nam pove takojšnjo vrednost stanja, vrednostna funkcija pa to vrednost gleda na dolgi rok. Je izpeljanka nagrade, a veliko bolj primerna za maksimizacijo kot nagrada sama, saj upošteva, da so tudi stanja, ki ne prinesejo takojšnje nagrade lahko veliko vredna (na tem mestu se ponovno spomnimo šaha in grajenja strategije, ki nagrado prinese šele ob koncu igre).

Poleg tega je v splošnem lažje učenje prek vrednostnih funkcij kot prek strategij neposredno, saj je ponavadi stanj mnogokrat manj kot možnih strategij agenta.

Zapišimo zgoraj opisane pojme bolj formalno:

Definicija 2.5. Naj $\mathcal{S}$ označuje množico vseh stanj, $\mathcal{A}$ pa množico vseh akcij. Naj R_t, S_t, A_t zaporedoma označujejo nagrado, stanje in akcijo ob času t. Definiramo naslednje pojme:

- Agentova **strategija** (angl. *policy*) je preslikava, ki agentu pove, katero akcijo naj izbere v katerem stanju. Strategije delimo v dve skupini:
  - **Deterministična strategija** je preslikava $\pi : \mathcal{S} \to \mathcal{A}$, ki za vsako stanje pove, katero akcijo agent v njem izbere
  $$a = \pi(s).$$
  - **Stohastična strategija** je preslikava $\pi : \mathcal{A} \times \mathcal{S} \to [0, 1]$ ki za vsako stanje pove verjetnost, da se igra določena akcija. To označimo
  $$\pi(a|s) = P(A_t = a \mid S_t = s).$$

  Seveda lahko vsako deterministično strategijo predstavimo kot stohastično, kjer je verjetnost akcije a 1, verjetnosti ostalih akcij pa so enake 0.
- Naj bodo $R_{t+1}, ..., R_T$ nagrade, ji jih bomo prejeli od trenutka t do končnega časa T. **Povračilo** (angl. *return*) G_t v splošnem definiramo za $T = \infty$,
  $$G_t = R_{t+1} + \gamma R_{t+2} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1},$$
  kjer je $\gamma \in [0, 1]$ **diskontni faktor**. Predstavlja dejstvo, da imamo raje nagrade, ki bodo prišle prej. Formalno gledano, je cilj učenja maksimizacija pričakovanega povračila.

- Naj bo π dana strategija agenta. **Vrednostna funkcija stanja** (angl. *state value function*) glede na strategijo π je pogojno matematično upanje glede na stanje

$$v_\pi(s) = \mathrm{E}[G_t \mid S_t = s].$$

  Predstavlja torej pričakovani izplen, če se vedemo skladno s strategijo π.
- Naj bo π še vedno dana strategija agenta. **Vrednostna funkcija akcije** (tudi stanja-akcije) (angl. *action-value function*) glede na strategijo π je definirana kot

$$q_\pi(s, a) = \mathrm{E}[G_t \mid S_t = s, A_t = a].$$

  Pove nam pričakovani izplen, če ob času t naredimo akcijo a, nato pa se vedemo skladno s strategijo π.

Zgled 2.6 (Križci in krožci)**.** *Agent je v tem primeru računalniški program, ki prejme igralno ploščo, nasprotnikove poteze in nagrade, nato pa prek poskušanja in učenja vrne optimalno strategijo, t.j. za vsako stanje najboljšo možno akcijo.*

2.1.4. *Model.* Model je nenujen del našega sistema. Predstavlja znanje, ki ga ima agent o svojem okolju. Če imamo model, da lahko uporabimo, ga napovemo, kako se bo vedlo okolje in s tem premaknemo agentovo učenje iz čistih poskusov in napak na *načrtovanje* (angl. *planning*). Model je torej poleg strategije in vrednostne funkcije še tretja komponenta agenta. Na podlagi modela lahko agent »preračuna« smiselnost svojih akcij, brez da bi dejansko karkoli storil.

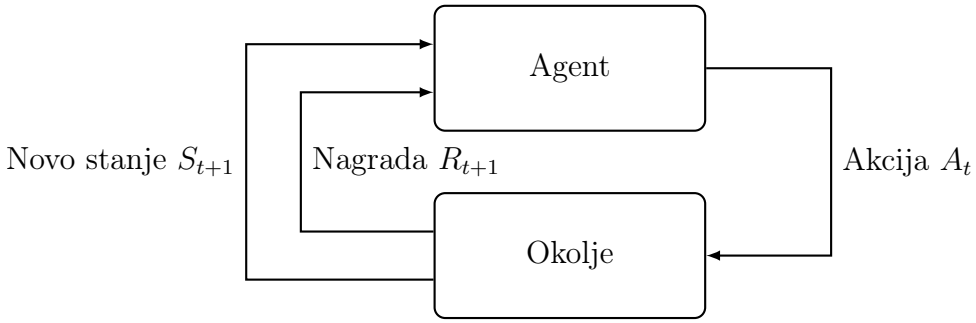
Prisotnost modela je glavna ločnica med dvema velikima, a zelo različnima vejama spodbujevalnega učenja. Če modela ni, govorimo o spodbujevalnem učenju brez modela (angl. *model-free reinforcement learning*), v nasprotnem primeru pa govorimo o učenju z modelom (angl. *model-based reinfocement learning*). Narava namiznih iger za dva igralca, kakeršne obravnavamo tu je, da lahko predvidimo v kakšno stanje nas prinese naša akcija, zato načeloma imamo omejen model okolja.

2.2. **Korak spodbujevalnega učenja.** Spodbujevalno učenje se pogosto ukvarja s procesi, ki naravno razpadejo v t. i. **epizode**. Tak proces so recimo namizne igre, kjer so epizode precej naravno posamezne igre. Če bi probavali robota naučiti hoje, bi lahko za epizode vzeli neko časovno okno ali pa morda hojo do prvega padca. Ni pa nujno, da je delitev tako naravna (ali pa sploh možna oz. smiselna). Za namene te diplomske naloge lahko privzamemo, da delitev na epizode obstaja.

Ideja učenja je, da agenta spustimo v okolje in mu dovolimo, da doživi (igra) mnogo epizod. Nato na nek način (s pomočjo spodbujevalnega učenja) ob nekih določenih časih (npr. po koncu posamezne epizode) posodobi svojo strategijo (in/ali vrednostno funkcijo).

Dejanski korak (npr. poteza enega od igralcev v namizni igri) v epizodi pa formalno gledano opredelimo na naslednji način.

- Agent naredi akcijo A_t ob prejetem stanju S_t in prejme nagrado R_t.
- Okolje prejme akcijo A_t, posreduje agentu stanje S_{t+1} in nagrado R_{t+1}.



2.2.1. *Raziskovanje in izkoriščanje.* Eden izmed glavnih problemov, s katerim se srečamo pri spodbujevalnem učenju, je problem raziskovanja in izkoriščanja. Ko se agent uči, začne dojemati, katere akcije ali pa kombinacije akcij mu pripeljejo nagrado. Ko to ugotovi, seveda lahko začne te akcije *izkoriščati* in prejemati vso nagrado, ki akcijam pripada. Pri tem pa naletimo na problem. Agent lahko izkorišča te akcije in nikoli ne ugotovi, da neka druga akcija prinese še višjo nagrado; tega ne izve, ker ne *raziskuje*. Če pa samo raziskuje, pa nikoli ne izkoristi potencialnih nagrad, ki jih sreča, torej se ničesar ne nauči.

Uravnoteženje raziskovanja in izkoriščanja je pomemben problem, a se izkaže, da ima dokaj enostavno rešitev (ki deluje dovolj dobro). Spoznali jo bomo v nadaljevanju.

Zgled 2.7 (Raziskovanje v namiznih igrah)**.** *Pri namiznih igrah agent lahko odkrije strategijo, ki premaga določenega nasprotnika. To strategijo lahko potem izrablja in nasprotnika vedno premaga, ko pa naleti na drugega nasprotnika se lahko izkaže, da je bila strategija učinkovita samo proti prvemu - naša strategija ni bila optimalna. Zato je pomembno, da tudi ob odkritju dobre strategije agent še vedno raziskuje prostor strategij, to najenostavneje dosežemo tako, da agenta prisilimo, da občasno igra naključne poteze.*

Morda se nekaterim bralcem zdi, da smo zaenkrat preveč »mahali z rokami«. To je zato, ker želimo, da se do te točke razvije intuicija o predstavljenih pojmih. V nadaljevanju bomo do sedaj opisane stvari bolj formalizirali.

2.3. **Markovski proces odločanja.** Spomnimo se najprej procesa spodbujevalnega učenja in ga poskusimo opisati bolj formalno: imamo zaporedje časovnih korakov $t = 0, 1, 2, \ldots$, ob katerih med sabo interaktirata agent in okolje. Ob koraku t agent prejme od okolja stanje (oz. reprezentacijo stanja) $S_t \in \mathcal{S}$, kjer $\mathcal{S}$ označuje množico vseh stanj. Na podlagi stanja in strategije, ki jo ima, izbere akcijo $A_t \in \mathcal{A}(S_t)$, kjer $\mathcal{A}(S_t)$ predstavlja množico akcij, ki jih ima agent na voljo v stanju S_t. Rezultat te akcije je nagrada $R_{t+1} \in \mathcal{R}$, kjer $\mathcal{R}$ označuje množico vseh nagrad, in novo stanje S_{t+1}.

Čeprav se da vse opisane koncepte posplošiti na števne in celo neštevne množice stanj in akcij, se bomo mi omejili na končne množice. To je pri namiznih igrah dovolj.

2.3.1. *Markovska veriga.* Dogajanje pri spodbujevalnem učenju lahko v grobem opišemo z zaporedjem slučajnih spremenljivk $S_0, S_1, \ldots$ v diskretnem času, to je, s slučajnim procesom stanj $(S_t)_{t=0}^{T}$. Zato je pomembno, da si natančno pogledamo nekaj lastnosti, ki jih lahko pričakujemo.

Definicija 2.8 (Markovska veriga)**.** Slučajni proces $(S_t)_{t=0}^T$ na končnem verjetnostnem prostoru $(\Omega, \mathcal{F}, P)$ je **Markovska veriga** oz. **Markovski proces** (angl. *Markov chain*), če zanj velja Markovska lastnost

$$P(S_{t+1} = s_{t+1} \mid S_t = s_t, \ldots, S_0 = s_0) = P(S_{t+1} = s_{t+1} \mid S_t = s_t).$$

Na kratko to *prehodno verjetnost* označimo $p_{ss'} := P(S_{t+1} = s' \mid S_t = s)$. Opazimo, da lahko te verjetnsoti zložimo v matriko $\mathcal{P} := [p_{ss'}]_{s,s' \in \mathcal{S}}$, ki ji pravimo *prehodna matrika*.

Zdaj Markovsko verigo predstavimo še na alternativni način: kot dvojico $(\mathcal{S}, \mathcal{P})$, kjer je $\mathcal{P}$ zgoraj definirana prehodna matrika, $\mathcal{S}$ pa množica vseh stanj.

Markovska lastnost pomeni, da je prihodnost neodvisna od preteklosti, če poznamo sedanjost. Spodbujevalno učenje se ukvarja predvsem s problemi, kjer to dejstvo drži. Tudi pri našem ciljnem problemu to načeloma velja: če pogledamo igralno ploščo na katerikoli točki, pogosto izvemo enako o trenutnem stanju, kot če bi opazovali igro od začetka.

2.3.2. *Markovski proces nagrajevanja.* Podoben koncept, ki se malo bolj približa dejanski situaciji v spodbujevalnem učenju, je *Markovski proces nagrajevanja*. Kot že ime morda namigne, je precej podoben Markovski verigi, le da v njem nastopajo *nagrade*.

Definicija 2.9 (Markovski proces nagrajevanja)**.** Pričakovani nagradi glede na stanje s pravimo **nagradna funkcija** (angl. *reward function*) in jo označimo

$$\mathcal{R}_s = E[R_{t+1} \mid S_t = s].$$

Naboru $(\mathcal{S}, \mathcal{P}, \mathcal{R}, \gamma)$, za katerega velja

- $\mathcal{S}$ je (končna) množica stanj,
- $\mathcal{P}$ je prehodna matrika $\mathcal{P}_{ss'} = P(S_{t+1} = s' \mid S_t = s)$,
- $\mathcal{R}$ je *nagradna funkcija* $\mathcal{R}_s = \mathrm{E}[R_{t+1} \mid S_t = s]$,
- $\gamma \in [0, 1]$ je *diskontni faktor*,

pravimo **Markovski proces nagrajevanja** (angl. *Markov reward process*).

Od navadne Markovske verige se torej razlikuje samo v prisotnosti nagrad ob vsakem koraku. Če te nagrade postavimo na $R_t = 0$ za vsak t, dobimo navadno Markovsko verigo. Pomembna razlika je še prisotnost diskontnega faktorja γ. Če velja $\gamma < 1$, potem procesu pravimo *diskontirani Markovski proces nagrajevanja*.

Diskontiranje je motivirano iz različnih vidikov: po eni strani nam pomaga, da se v primeru cikličnih procesov izognemo neomejenim povračilom. Poleg tega pa je diskontiranje v mnogo pogledih naraven način za opis situacije: pogosto imamo raje nagrade, ki pridejo prej. Primer tega poznamo recimo iz ekonomije; denar, ki ga dobimo kasneje nam pomeni manj, kot tisti, ki ga dobimo takoj.

Še vedno pa tovrsten proces ne opiše situacije, v kateri se znajdemo pri spodbujevalnem učenju, saj ne vsebuje koncepta akcij. Je pa že dovolj »globok«, da v njem lahko definiramo vrednostno funkcijo $v(s) = \mathrm{E}[G_t \mid S_t = s]$, ki je v tem primeru neodvisna od strategije π, saj strategija v tem modelu nima pomena.

Za vrednostno funkcijo lahko izpeljemo rekurzivno enačbo, s pomočjo katere lahko »rešimo« Markovski proces nagrajevanja. S tem mislimo, da vsakemu stanju dodelimo pravo vrednost. To je vrednost, ki jo stanju določata nagrada in nagradna funkcija.

$$
\begin{aligned}
v(s) &= \mathrm{E}[G_t \mid S_t = s] \\
&= \mathrm{E}[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s] \\
&= \mathrm{E}[R_{t+1} + \sum_{k=1}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s] \\
&= \mathrm{E}[R_{t+1} + \gamma(\sum_{k=1}^{\infty} \gamma^{k-1} R_{t+k+1}) \mid S_t = s] \\
&= \mathrm{E}[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \\
&= \mathrm{E}[R_{t+1} + \gamma v(S_{t+1}) \mid S_t = s].
\end{aligned}
$$

Dobili smo torej

$$v(s) = \mathrm{E}[R_{t+1} + \gamma v(S_{t+1}) \mid S_t = s]$$

oziroma

$$v(s) = \mathcal{R}_s + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'} v(s'), \tag{1}$$

kjer smo upoštevali aditivnost in idempotentnost pogojnega matematičnega upanja. Enačba (1) je **Bellmanova enačba za Markovkse procese nagrajevanja**.

Ker se ukvarjamo s primerom, ko je stanj končno mnogo, recimo n, jih lahko brez škode za splošnost označimo kar $1, \ldots, n$. Potem lahko Bellmanovo enačbo prepišemo v matrični obliki

$$
\begin{bmatrix} v(1) \\ \vdots \\ v(n) \end{bmatrix} = \begin{bmatrix} \mathcal{R}_1 \\ \vdots \\ \mathcal{R}_n \end{bmatrix} + \gamma \begin{bmatrix} \mathcal{P}_{11} \ldots \mathcal{P}_{1n} \\ \vdots \\ \mathcal{P}_{n1} \ldots \mathcal{P}_{nn} \end{bmatrix} \begin{bmatrix} v(1) \\ \vdots \\ v(n) \end{bmatrix}
$$

ali krajše

$$v = \mathcal{R} + \gamma \mathcal{P} v.$$

Opazimo, da je to *linearna* enačba, ki jo lahko eksplicitno rešimo:

$$
\begin{aligned}
v &= \mathcal{R} + \gamma \mathcal{P} v \\
(I - \gamma \mathcal{P}) v &= \mathcal{R} \\
v &= (I - \gamma \mathcal{P})^{-1} \mathcal{R}.
\end{aligned}
$$

Ta rešitev predpostavlja obrnljivost matrike $(I - \gamma \mathcal{P})$ in računanje njenega inverza, kar zahteva $O(n^3)$ operacij, zato je smiselna samo za majhne procese. Za večje obstajajo iterativni algoritmi in metode, nekatere izmed njih bomo spoznali v naslednjem odseku.

TA DEL LAHKO MATEMATIZIRAŠ - ENOLIČNOST REŠITEV

2.3.3. *Markovski proces odločanja.* Kot ime že nakazuje, **Markovski proces odločanja** (angl. *Markov decision process (MDP)*) razširja koncept procesa nagrajevanja z dodatkom odločanja - akcij. S tem dodatkom imamo v popolnosti opisan problem spodbujevalnega učenja: proučujemo torej nek proces, kjer ima agent možnost odločanja, izid pa je vsaj delno slučajen in odvisen od okolja.

Opomba 2.10. Ker v Markovskem procesu odločanja nastopajo (in imajo poglavitno) vlogo akcije, seveda pomembno vplivajo na nagradno funkcijo in prehodno matriko, zato moramo ta koncepta ustrezno prilagoditi:

$$\mathcal{R}_s^a = E[R_{t+1} \mid S_t = s, A_t = a],$$
$$\mathcal{P}_{ss'}^a = P(S_{t+1} = s' \mid S_t = s, A_t = a)$$

Definicija 2.11 (Markovski proces odločanja)**.** Naboru $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$, za katerega velja

- $\mathcal{S}$ je (končna) množica stanj,
- $\mathcal{A}$ je (končna) množica akcij,
- $\mathcal{P}$ je prehodna matrika $\mathcal{P}_{ss'}^a = P(S_{t+1} = s' \mid S_t = s, A_t = a)$,
- $\mathcal{R}$ je *nagradna funkcija* $\mathcal{R}_s^a = \mathrm{E}[R_{t+1} \mid S_t = s, A_t = a]$,
- $\gamma \in [0, 1]$ je *diskontni faktor*,

pravimo **Markovski proces odločanja** (angl. *Markov decision process*).

V kontekstu MDP-jev lahko sedaj formalno definiramo strategijo agenta π, ki je zahvaljujoč Markovski lastnosti odvisna od enega (trenutnega) stanja in ne od celotne zgodovine procesa. Definiramo tudi vrednostno funkcijo stanja $v_\pi(s)$ in vrednostno funkcijo akcije $q_\pi(s, a)$. Njihove definicije se ne spremenijo, so pa sedaj formalno vmeščene v model.

Opazimo, da je v MDP-ju pri fiksni strategiji π zaporedje oz. proces stanj $S_1, S_2, \ldots$ Markovska veriga $(\mathcal{S}, \mathcal{P}^\pi)$. Če v zaporedje stanj pomešamo še nagrade, torej $S_1, R_2, S_2, R_3, \ldots$, dobimo Markovski proces nagrajevanja $(\mathcal{S}, \mathcal{P}^\pi, \mathcal{R}^\pi, \gamma)$, kjer smo označili

$$\mathcal{P}^\pi = \sum_{a \in \mathcal{A}} \pi(a|s) \mathcal{P}_{ss'}^a$$
$$\mathcal{R}^\pi = \sum_{a \in \mathcal{A}} \pi(a|s) \mathcal{R}_s^a.$$

Opomba 2.12. Morda je nekatere bralce zbodlo, da v zaporedju stanj in nagrad S_1 ne sledi R_1, temveč R_2. Za tako notacijo smo se odločili, da poudarimo, da okolje agentu sočasno poda stanje S_2 in nagrado R_2, glede na stanje S_1 pa se agent odloča o akciji A_1.

MDP-ji so splošna orodja za obravnavo stohastičnih procesov, ki vključujejo odločitve v diskretnem času. Njihov utemeljitelj je Richard Bellman, ki je znan predvsem po izumu dinamičnega programiranja, zato morda ni presenetljivo, da nam prav dinamično programiranje poda osnovo za njihovo reševanje. Kot pri procesih nagrajevanja, lahko tudi tu izpeljemo Bellmanovo enačbo. Najprej izpeljimo rekurzivni zvezi za vrednostni funkciji stanja in akcije:

$$v_\pi(s) = \mathrm{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s],$$
$$q_\pi(s, a) = \mathrm{E}[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) \mid S_t = s, A_t = a].$$

Opazimo, da v_π in q_π povezujta zvezi

$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) q_\pi(s, a),$$
$$q_\pi(s, a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s').$$

Smiselno se zdi, da bi povprečje $q_\pi(s,a)$ po akcijah, uteženo z ustrenimi verjetnostmi

Če zvezi združimo, dobimo

$$v_\pi(s) = \sum_{a\in\mathcal{A}} \pi(a|s) \left[\mathcal{R}_s^a + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s')\right], \tag{2}$$

$$q_\pi(s,a) = \mathcal{R}_s^a + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a \left[\sum_{a'\in\mathcal{A}} \pi(a'|s') q_\pi(s',a')\right]. \tag{3}$$

Enačbo (2) imenujemo **Bellmanova enačba pričakovanja** (angl. *Bellman expectation equation*), ki ima ponovno matrično obliko in eksplicitno (ENOLIČNO - ZAKAJ, TU DODATNA RAZLAGA) rešitev:

$$v_\pi = \mathcal{R}^\pi + \gamma \mathcal{P}^\pi v_\pi$$
$$v_\pi = (I - \gamma\mathcal{P}^\pi)^{-1}\mathcal{R}^\pi$$

Ta enačba je sicer pomembna, a nas najbolj zanima »rešitev« MDP-ja. Želimo namreč najti optimalno strategijo. Pri tem nam pomaga naslednje:

Definicija 2.13.

- **Optimalna vrednostna funkcija stanja** (angl. *optimal state-value function*) $v_*(s)$ je največja med vsemi vrednostnimi funkcijami stanj
$$v_*(s) = \max_\pi v_\pi(s).$$
- **Optimalna vrednostna funkcija akcije** (angl. *optimal action-value function*) $q_*(s,a)$ je največja med vsemi vrednostnimi funkcijami akcij
$$q_*(s,a) = \max_\pi q_\pi(s,a).$$
- Na množici vseh možnih strategij Π definiramo delno urejenost na naslednji način:
$$\pi \geq \pi', \text{ če } v_\pi(s) \geq v_{\pi'}(s)\ \forall s.$$
Če za strategiji velja kot zgoraj, pravimo, da je π *boljša ali enaka* kot π'.

Sedaj lahko uporabimo naslenji izrek.

Izrek 2.14. *Za vsak Markovski proces odločanja velja:*

- *Obstaja optimalna strategija π_*, ki je boljša ali enaka kot vse ostale strategije;* $\pi_* \geq \pi\ \forall \pi \in \Pi$.
- *Vedno obstaja deterministična optimalna strategija. PREJ RAZLOŽI NATANČNO STRATEGIJE*
- *Vse optimalne strategije določajo optimalno vrednostno funkcijo stanja in optimalno vrednostno funkcijo akcije;*
$$v_{\pi*}(s) = v_*(s)$$
$$q_{\pi*}(s,a) = q_*(s,a).$$

OPOMBA: SKLIC NA DOKAZ

Izrek nam pove, da če poznamo $q_*(s,a)$, poznamo tudi optimalno deterministično strategijo, ki jo dobimo kot $\pi_*(a|s) = \mathbb{1}(a = \arg\max_{a\in\mathcal{A}} q_*(s,a))$.

Podobno kot za poljubne vrednostne funkcije, lahko tudi za optimalne vrednostne funkcije izpeljemo Bellmanovo enačbo, ki jo sedaj imenujemo **Bellmanova enačba optimalnosti** (angl. *Bellman optimality equation*). Sprva opazimo zvezi med q_* in v_*:

IZPELJI TI DVE ZVEZI

$$v_*(s) = \max_a q_*(s,a),$$
$$q_*(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s).$$

Ti zvezi nato vstavimo eno v drugo, in dobimo enačbi:

$$v_*(s) = \max_a \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s), \tag{4}$$

$$q_*(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \max_{a'} q_*(s',a'). \tag{5}$$

Opazimo, da sta enačbi tokrat nelinearni, kar močno oteži direktno reševanje. Na srečo pa tudi v tem primeru pridejo prav metode iz naslednjega odseka.

3. Algoritmi pri spodbujevalnem učenju

Algoritmov za reševanje MDP-jev je precej. Mi se bomo omejili predvsem na algoritme, ki se učijo prek vrednostne funkcije in izhajajo iz dinamičnega programiranja, a naj na tem mestu omenimo, da obstajajo tudi algoritmi, ki posodabljajo strategijo neposredno (MOGOČE KAKŠEN REFERENCE ALI PA KEJ).

Izjemnega pomena pri reševanju je tudi dejstvo, ali je znana matrika $\mathcal{P}$ in nagradna funkcija $\mathcal{R}_s^a$. Izkaže se, da je večina problemov takih, da ti dve stvari nista znani. Eden izmed takih problemov je tudi igranje namiznih iger – ne poznamo strategije našega nasprotnika, zato ne poznamo verjetnosti prehodov med stanji.

Poleg tega naj na tem mestu omenimo, da reševanju celotnega problema spodbujevalnega učenja pravimo **načrtovanje** (angl. *planning*), le-ta pa se deli na dve stopnji:

- **Napovedovanje** - pri tem podproblemu podamo nek MDP in strategijo, algoritem nam vrne vrednostno funkcijo stanja (glede na strategijo).
- **Upravljanje** - to je bolj poln problem: podan je MDP, algoritem pa nam vrne optimalno vrednostno funkcijo in optimalno strategijo.

3.1. Dinamično programiranje - reševanje poznanih MDP-jev.

Dinamično programiranje je optimizacijska metoda, ki deluje na principu deljenja velikega problema na manjše prekrivajoče podprobleme. V jedru reševanja je **Bellmanova enačba**, ki opisuje odnos med vrednostmi podproblemov in vrednostjo glavnega problema. Ker je idejo dinamičnega programiranja (DP) in MDP-jev dobil Bellman ob istem času, je naravno, da je DP metoda, ki je prilagojena prav situaciji v MDP-ju.

Problem mora v splošnem zadoščati dvema lastnostima, da je zanj primerno reševanje z dinamičnim programiranjem. Prvi pravimo **lastnost optimalne podstrukture**, ki pravi, da lahko problem razstavimo na podprobleme in rešitve podproblemov lahko potem sestavimo nazaj v rešitev celotnega problema. Druga pomembna

lastnost pa so **prekrivajoči podproblemi**, ki implicira, da lahko že poznane rešitve podproblemov večkrat uporabimo.

Opazimo, da obe lastnosti veljata za MDP-je; Bellmanova enačba, nam razdeli problem na podprobleme, vrednostna funkcija pa hrani in ponovno uporablja rešitve. Lastnosti veljata tudi za MRP-je in tudi metode, ki jih bomo spoznali so aplikativne na njih, a mi se bomo posvetili predvsem procesom odločanja.

(MORDA BOLJ FORMALNO O BELLMANOVIH ENAČBAH - NI TREBA?)

3.1.1. *Iterativno ocenjevanje strategije.* Algoritem deluje na MDP-ju, katerega prehodna matrika je znan podatek. Poleg tega potrebuje tudi fiksno strategijo π. Algoritem torej rešuje zgorlj parcialni problem, to je problem napovedovanja. V ozadju je glavna ideja Bellmanova enačba pričakovanja, z njo na vsakem koraku posodobimo $v_k(s)$ v $v_{k+1}(s)$ s pomočjo $v(s')$, kjer je s' naslednje stanje od s. Enačba iteracije se potem glasi

$$v_{k+1}(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left[\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_k(s') \right] \text{ ali na kratko}$$
$$v_{k+1} = \mathcal{R}^\pi + \gamma \mathcal{P}^\pi v_k.$$

Izrek 3.1. *Iterativno ocenjevanje strategije konvergira proti vrednostni funkciji, ki jo določa strategija π.*

Dokaz. Pokazali bomo, da iz iteracije lahko pridobimo skrčitev, iz izrekov o fiksni točki potem sledi naš rezultat (MOGOČE DEJMO NAVEST IZREK? - BANACH, CONTRACTION MAPPING) - LAHKO SKLIC NA LITERATURO.

Definiramo **Bellmanov operator pričakovanja** kot

$$T_\pi(v) = \mathcal{R}_\pi + \gamma \mathcal{P}_\pi v.$$

POVEŠ KAKO JE DEFINIRANA OPERATORSKA NESKONČNA NORMA Naj bosta u in v vrednostni funkciji. Potem je zgornji operator skrčitev glede na neskončno normo:

$$\begin{aligned} ||T_\pi(v) - T_\pi(u)||_\infty &= ||\mathcal{R}_\pi + \gamma \mathcal{P}_\pi v - \mathcal{R}_\pi + \gamma \mathcal{P}_\pi u||_\infty \\ &= ||\gamma \mathcal{P}_\pi (v - u)||_\infty \\ &\leq ||\gamma \mathcal{P}_\pi||_\infty ||v - u||_\infty \\ &\leq \gamma ||v - u||_\infty, \end{aligned}$$

kjer smo upoštevali, da velja $||\mathcal{P}_\pi||_\infty \leq 1$, saj so elementi $\mathcal{P}_\pi$ verjetnosti.

□

S tem procesom tako res dobimo vrednostno funkcijo, ki jo določa strategija, a strategija je statična; to reši problem napovedovanja.

Da rešimo problem upravljanja, uporabimo enostavno idejo: strategijo izboljšamo **požrešno**. To pomeni, da izberemo tako akcijo, ki nas pripelje v stanje, ki je dosegljivo in ima trenutno največjo vrednost v_π.

3.1.2. *Iteracija strategije.* Algoritem, ki ga dobimo s tem, da združimo iterativno ocenjevanje strategije in požrešno izboljšavo strategije, imenujemo iteracija strategije (angl. *policy iteration*). S tem algoritmom lahko sedaj rešimo problem upravljanja in s tem celoten problem poznanega MDP-ja.

Trditev 3.2. *Iteracija strategije vedno konvergira k optimalni strategiji π_*.*

Dokaz. Ukvarjali se bomo samo z determinističnimi strategijami, saj so zaradi požrešne izbire strategije le-te vedno deterministične, razen začetne. Ker je izbira začetne strategije za trditev nepomembna, lahko izberemo neko naključno deterministično strategijo $a = \pi(s)$.

Požrešno vedenje zdaj za nas pomeni, da vzamemo $\pi'(s) = \arg\max_{a\in\mathcal{A}} q_\pi(s,a)$. To očitno izboljša vrednost za vsako stanje

$$q_\pi(s,\pi'(s)) = \max_{a\in\mathcal{A}} q_\pi(s,a) \geq q_\pi(s,\pi(s)) = v_\pi(s).$$

Posledično izboljša tudi vrednostno funkcijo:

$$\begin{aligned} v_\pi(s) &\leq q_\pi(s,\pi'(s)) = \mathrm{E}_{\pi'}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s] \\ &\leq \mathrm{E}_{\pi'}[R_{t+1} + \gamma q_\pi(S_{t+1}\pi'(S_{t+1})) \mid S_t = s] \\ &\leq \mathrm{E}_{\pi'}[R_{t+1} + \gamma R_{t+2} + \gamma^2 q_\pi(S_{t+2}\pi'(S_{t+2})) \mid S_t = s] \\ &\leq \mathrm{E}_{\pi'}[R_{t+1} + \gamma R_{t+2} + \ldots \mid S_t = s] = v_{\pi'}(s). \end{aligned}$$

Če vrednost preneha rasti, dobimo

$$q_\pi(s,\pi'(s)) = \max_{a\in\mathcal{A}} q_\pi(s,a) = q_\pi(s,\pi(s)) = v_\pi(s).$$

To se zgodi, saj zaradi končnih nagrad tudi vrednostne funkcije zavzemajo končne vrednosti. Potem opazimo, da je zaradi $v_\pi(s) = \max_{a\in\mathcal{A}} q_\pi(s,a)$ zadoščeno Bellmanovi enačbi optimalnosti in zato po izreku 2.14 velja $v_\pi(s) = v_*(s)\ \forall s \in \mathcal{S}$, torej je π optimalna strategija. □

3.1.3. *Iteracija vrednosti.* K problemu lahko pristopimo tudi neposredno prek Bellmanove enačbe optimalnosti in se s tem izognemo neposredni uporabi strategije, vseeno pa rešimo problem upravljanja. Ideja je podobna kot pri iterativnem ocenjevanju strategije, le da imamo drugačno enačbo za iteracijo:

$$v_{k+1}(s) = \max_{a\in\mathcal{A}} \left[\mathcal{R}_s^a + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a v_k(s') \right]$$

oziroma

$$v_{k+1} = \max_{a\in\mathcal{A}} \mathcal{R}^a + \gamma \mathcal{P}^a v_k.$$

Podobno kot pri iterativnem ocenjevanju strategije pokažemo, da zgornja iteracija konvergira k v_*.

3.1.4. *Časovna zahtevnost.* Za korak obeh iteracij je potrebnih $O(mn^2)$ operacij, kjer je m število akcij in n število stanj. Popolnoma enake algoritme lahko uporabimo za iteracijo $q_\pi(s,a)$ oz. $q_*(s,a)$, le da sedaj pridemo do časovne zahtevnosti $O(m^2n^2)$.

3.1.5. *a sm js pozabu na generalised policy evaluation.*

3.2. **Monte Carlo.** Do sedaj smo se ukvarjali s poznanim MDP-jem, kar pa ni naš končen cilj. V nadaljevanju se bomo posvetili nepoznanim. Metode, ki jih bomo spoznali so tako bolj splošno aplikativne, svoje bistvo pa črpajo pri do sedaj poznanem.

Prva taka metoda je Monte Carlo. Spopada se s problemom napovedovanja z dodatno izboljšavo, da ne rabi poznati matrike $\mathcal{P}$. Potrebujemo pa strategijo π, na podlagi katere opazujemo epizode dogajanja. Ideja delovanja je, da se namesto na direktno računanje pričakovanega povračila $G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$, osredotočimo na računanje *empiričnega* povračila. To storimo tako, da za vsako stanje s definiramo števec $N(s)$, ki beleži, kolikokrat je bilo stanje obiskano. Ta števec ne beleži zgolj obiskov znotraj epizode, temveč skozi celoten proces učenja. Za vsako stanje hranimo še $S(s)$, ki mu ob vsakem obisku prištejemo pričakovano povračilo. To seveda pomeni, da se vrednost stanja ne more posodobiti takoj po obisku, ampak šele ob koncu epizode, ko je G_t znan. Po koncu učenja enostavno za vsako stanje delimo $S(s)$ z $N(s)$ in tako dobimo vzorčno povprečje za dejanski $v(s)$.

- Ob vsakem (ali samo prvem) obisku stanja s:

$$N(s) \leftarrow N(s) + 1$$
$$S(s) \leftarrow S(s) + G_t.$$

- Po zadostni količini epizod (ob koncu učenja):

$$V(s) \leftarrow S(s)/N(s)$$

TOLE MALO BOLJ RAZDELAJ - MONTE CARLO IMA KVADRATIČNO KONVERGENCO, SUTTON BARTO Opazimo, da vrednost $V(s)$ konvergira proti $v_\pi(s)$, ko gre $N(s) \to \infty$ po krepkem zakonu velikih števil. Opazimo tudi, da metoda deluje samo za probleme, ki imajo končne epizode, saj šele takrat lahko poznamo G_t. Če za vsako stanje s hranimo $N(s)$ in $S(s)$, tako dobimo oceno za $v_\pi(s)$ za vsak $s \in \mathcal{S}$.

Prostorsko zahtevnost lahko izboljšamo, če upoštevamo, da povprečje μ nekega zaporedja slučajnih spremenljivk $X_1, X_2, \ldots$ izračunamo inkrementalno:

$$\begin{aligned}
\mu_k &= \frac{1}{k} \sum_{i=1}^{k} X_i \\
&= \frac{1}{k} \left[X_k + \sum_{i=1}^{k-1} X_i \right] \\
&= \frac{1}{k} \left[X_k + (k-1)\mu_{k-1} \right] \\
&= \mu_{k-1} + \frac{1}{k} \left(X_k - \mu_{k-1} \right).
\end{aligned}$$

Na podlagi te zveze lahko implementiramo inkrementalni Monte Carlo, tako da po končani epizodi za vsako stanje S_t s povračilom G_t posodobimo (ODLOČI SE ZA S_t ALI s):

$$\begin{aligned}
&N(S_t) \leftarrow N(S_t) + 1, \\
&V(S_t) \leftarrow V(S_t) + \frac{1}{N(S_t)}(G_t - V(S_t)).
\end{aligned}$$

Izkaže se, da lahko zgornje še malo posplošimo in uporabimo

$$V(S_t) \leftarrow V(S_t) + \alpha(G_t - V(S_t)), \tag{6}$$

kjer α imenujemo **velikost koraka** oz. **hitrost učenja** (angl. *learning rate*). Posodobitveno pravilo (6) je konkreten primer bistvene ideje spodbujevalnega učenja prek vrednostne funkcije. Vsi ostali algoritmi nam dajo pravilo podobne oblike:

$$\textit{nova ocena} \leftarrow \textit{stara ocena} + \textit{korak}\ (\textit{tarča} - \textit{stara ocena}). \tag{7}$$

3.3. **Algoritem** $TD(0)$**.** V spodbujevalnem učenju obstaja skupina algoritmov, ki se imenujejo **algoritmi učenja s časovno razliko** (angl. *temporal-difference learning*). Delujejo v podobnih pogojih kot Monte Carlo in tudi dosegajo enak cilj. Pomembna razlika je, da ne potrebujejo povračila pri posodobitvi. Delujejo lahko tudi v sistemih, ki se ne delijo na epizode oz. ne potrebujejo končanih epizod za svoje delovanje.

Najpreprostejši algoritem učenja s časovno razliko je $TD(0)$. Ideja je, da v stanju ocenimo povračilo kot $G_t \approx R_{t+1} + \gamma V(S_{t+1})$. To potem uporabimo za *tarčo* v pravilu (7). To nam da pravilo

$$V(S_t) \leftarrow V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - V(S_t)). \tag{8}$$

Iz tega zelo hitro ugotovimo, od kod pride poimenovanje. Algoritem za svoje posodabljanje uporablja podatek iz naslednjega trenutka - torej se uči s časovno razliko. Pomembno je omeniti tudi, da za učenje uporablja oceno, torej ocenjuje na podalgi ocene. Temu pravimo *zankanje* (angl. *bootstrapping*).

3.3.1. *Primerjava z Monte Carlo.*

- Tarča pri MC je nepristranska ocena $v_\pi(S_t)$, medtem ko je pri TD(0) tarča pristranska ocena. Ta prednost MC je uravnotežena s tem, da ima TD(0) tarča nižjo varianco, saj je odvisna od ene same slučajne spremenljivke, medtem ko je pri MC odvisna od mnogih slučajnih dogodkov.
- Prav tako je pomembna razlika v dejstvu, da MC nujno rabi popolne epizode in se lahko izvede le po končani epizodi, TD(0) pa se lahko uči sproti (angl. *online*) in sploh ne potrebuje končnih epizod.
- Razlika je tudi v tem, kam konvergirata algoritma, če imamo na voljo samo končno količino epizod. Naj K označuje število epizod in k nakazuje, v kateri epizodi smo. Potem MC konvergira k rešitvi z minimalno povprečno kvadratno napako (MSE)

  $$\sum_{k=1}^{K} \sum_{t=1}^{T_k} (G_t^k - V(s_t^k))^2,$$

  torej se najbolje prilega opazovanim povračilom. TD(0) pa konvergira k rešitvi modela največjega verjetja Markova; to je rešitev MDP-ja, ki se najbolje prilega podatkom. Dobimo torej naslednji cenilki: (TO SI ŠE POGLEJ)

  $$\hat{\mathcal{P}}_{ss'}^a = \frac{1}{N(s,a)} \sum_{k=1}^{K} \sum_{t=1}^{T_k} \mathbb{1}(s_t^k, a_t^k, s_{t+1}^k = s, a, s')$$

  $$\hat{\mathcal{R}}_s^a = \frac{1}{N(s,a)} \sum_{k=1}^{K} \sum_{t=1}^{T_k} \mathbb{1}(s_t^k, a_t^k = s, a) r_t^k$$

- Zadnja pomembna razlika je, da TD izrabi Markovsko lastnost, medtem ko je MC ne. Zato je vsaj v teoriji TD veliko bolj učinkovit v MDP-jih.

Zgled 3.3 (Ti si sodnik - IZ SUTTON BARTO UČBENIKA)**.** *Recimo, da imamo na voljo za učenje naslednjih 8 epizod: A* 0*, B,* 0*; B* 1*; B* 1*; B* 1*; B* 1*; B* 1*; B* 0

To pomeni recimo, da smo prvo epizodo začeli v stanju A, nato dobili nagrado 0 *in prešli v stanje B, kjer smo ponovno dobili nagrado* 0*, tu pa se je epizoda končala. Če bi se učili samo iz teh osmih epizod, bi se verjetno vsi strinjali, da je* $V(B) = 3/4$*, saj je v šestih opažanjih B zavzel vrednost* 1*, v dveh pa* 0*. Z nami se strinjata tudi* $TD(0)$ *in MC.*

Razlike se začenjo pri stanju A. MC, ki minimizira povprečno kvadratno napako in zato ob enem opažanju A, ki je dal nagrado 0*, enostavno reče* $V(A) = 0$ *- to ima celo ničelno napako glede na podatke.*

$TD(0)$ *pa pristopi nekoliko drugače. Vedno, ko smo opazili A, smo potem takoj opazili tudi B (vmes smo prejeli nagrado* 0*), zato je smiselno, da je vrednost stanja A enaka vrednosti stanja B, tj.* $V(A) = 3/4$*. V tem pristopu smo si pomagali tako, da smo problem modelirali kot markovski proces.*

3.4. **TD(λ).** Algoritma, ki smo ju spoznali do sedaj mogoče na prvi pogled nista najbolj podobna, a bomo kmalu videli, da pripadata isti družini algoritmov. V ta namen si poglejmo povračila, ki gledajo n- korakov v prihodnost. Tarča se primerno prilagodi v:

$$G_t^{(n)} = R_{t+1} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n V(S_{t+1}).$$

To tarčo lahko uporabimo v pravilu (7). S tem dobimo algoritem, ki se imenuje *učenje s časovno razliko z n koraki.* Opazimo tudi, da z večanjem n proti neskončno pridemo na neki točki do že znanega MC pravila. Hitro tudi vidimo, da to pravilo sicer res poveže $TD(0)$ in MC, a ne predstavlja nobene bistvene izboljšave. Pojavi pa se še dodaten problem. Če smo v času, ki je manj kot n korakov stran od konca epizode, ne moremo porabiti celotnega $G_t^{(n)}$. V tem primeru enostavno vzamemo toliko korakov naprej, kot nam čas dopušča. To pomeni, da se vedemo enako kot pri MC algoritmu.

Nadgradnjo algoritma nam da dejstvo, da lahko skupaj združujemo podatke iz različnih časov prek tega, da povprečimo $G_t^{(n)}$ za različne n. V nadaljevanju bomo pogledali, kako združimo podatke iz vseh časov, ki nastopijo. Najti moramo ustrezne uteži, ki se seštejejo v ena in so smiselne glede na obravnavan problem. Primernega kandidata sta našla CITIRAJ ITD ITD, in sicer:

$$G_t^{\lambda} = (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{(n-1)} G_t^{(n)}.$$

Od tu takoj sledi algoritem **TD(λ) s pogledom naprej** (angl. *forward-view TD(λ)*). To poimenovanje izhaja iz dejstva, da kot MC, tudi ta algoritem potrebuje znanje prihodnosti, da posodobi vrednsoti; z drugimi besedami, uči se lahko samo iz popolnih epizod.

Čeprav je teoretično zanimiv algoritem, je v praksi veliko bolj uporaben algoritem, ki ne potrebuje vedenja o prihodnosti in se tako lahko uči *»online«*. Izkaže se, da je zgornji algoritem mogoče popraviti in ga spremeniti v bolj praktično verzijo.

ZAKAJ JE TO POVEZAVA MED TD(0) IN MC!!!!!

3.4.1. *Sledi upravičenosti (angl. Eligibility traces).* Prek tega koncepta lahko spremenimo algoritem tako, da se lahko uči po vsaki potezi sproti. Ideja je, da za vsako stanje hranimo njegovo sled upravičenosti. Konceptualno to za nas pomeni, da sproti ugotavljamo, kako zasluženo je stanje za morebitno nagrado - koliko je pripomoglo k njeni pridobitvi.

Praktično pa je sled upravičenosti slučajna spremenljivka $E_t : \mathcal{S} \to \mathbb{R}^+$. Pri sledeh sta pomembni dve stvari: **čas**, ki je pretekel od obiska in **pogostost** obiska stanja. Sledi upravičenosti združijo oboje. Za stanje tako lahko napišemo enostavno enačbo, kjer je z $E_t(s)$ označena sled v času t za stanje s:

$$\begin{aligned} E_0(s) &= 0 \\ E_t(s) &= \gamma\lambda E_{t-1}(s) + \mathbb{1}(S_t = s), \end{aligned}$$

kjer sta γ in λ že poznana parametra. Pomensko torej sled propada z oddaljevanjem od obiska s, v primeru, da je s ponovno obiskan, pa skoči za 1. S tem lahko torej dodelimo nagrade oz. kazni samo tistim stanjem, ki to zaslužijo. V tem pogledu je to res osnovni mehanizem za dodeljevanje nagrad, ko smo dodatno vezani še na čas.

S tem novim konceptom lahko posodobimo naš algoritem. Če označimo napako kot $\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$, potem zapišemo:

$$V(s) \leftarrow V(s) + \alpha\delta_t E_t(s). \tag{9}$$

Poleg zgoraj definiranih sledi, ki jim pravimo *akumulativne sledi*, obstajata še dve alternativi:

- *Nadomeščujoče sledi*, ki so podobne akumulativnim, le da ob obisku ni skoka za 1, pač pa skok za 1, drugače pa imajo enako lastnost propadanja kot akumulativne.
- *Nizozemske sledi*, ki jih definiramo $E_t(s) = (1-\alpha)\gamma\lambda E_{t-1}(s) + \mathbb{1}(S_t = s)$. Torej podobno kot akumulativne, le da so dodatno pomnožene z $(1-\alpha)$. Ideja je, da s tem dodatnim členom dobimo sled, ki je med akumulativno in nadomeščujočo.

V zgornji formulaciji pravila TD(λ) je tudi bolj očitno, zakaj ta algoritem poveže $TD(0)$ in MC:

- Če je $\lambda = 0$, potem velja $E_t(s) = \mathbb{1}(S_t = s)$, torej se posodobi samo trenutno stanje: $V(S_t) \leftarrow V(s_t) + \alpha\delta_t$, kar pa je natančno posodobitev pri $TD(0)$.
- Za $TD(1)$ pa je za nagrado zaslužno vsako stanje, tako kot pri MC. Pravo ekvivalenco nam pomaga videti naslednji izrek.

Izrek 3.4. *Vsota posodobitev vrednosti ob koncu epizode je enaka za TD(λ) s pogledom naprej in pogledom nazaj*

$$\sum_{t=1}^{T} \alpha\delta_t E_t(s) = \sum_{t=1}^{T} \alpha \left(G_t^\lambda - V(S_t)\right) \mathbb{1}(S_t = s).$$

Dokaz. Oglejmo si napako, pri TD(λ) s pogledom nazaj:

$$\begin{aligned}
G_t^\lambda - V(S_t) &= -V(S_t) \\
&\quad + (1-\lambda)\lambda^0(R_{t+1} + \gamma V(S_{t+1})) \\
&\quad + (1-\lambda)\lambda^1(R_{t+1} + \gamma R_{t+2} + \gamma^2 V(S_{t+2})) \\
&\quad + \dots \\
&= -V(S_t) \\
&\quad + (\gamma\lambda)^0(R_{t+1} + \gamma V(S_{t+1}) - \gamma\lambda V(S_{t+1})) \\
&\quad + (\gamma\lambda)^1(R_{t+2} + \gamma V(S_{t+2}) - \gamma\lambda V(S_{t+2})) \\
&\quad + \dots \\
&= (\gamma\lambda)^0(R_{t+1} + \gamma V(S_{t+1}) - V(S_t)) \\
&\quad + (\gamma\lambda)^1(R_{t+2} + \gamma V(S_{t+2}) - V(S_{t+1})) \\
&\quad + \dots \\
&= \delta_t + \gamma\lambda\delta_{t+1} + (\gamma\lambda)^2\delta_{t+2} + \dots
\end{aligned}$$

Oglejmo si sedaj epizodo, kjer je stanje s obiskano natanko enkrat, in sicer ob času k. Sled upravičenosti je potem enaka

$$E_t(s) = \gamma\lambda E_{t-1}(s) + \mathbb{1}(S_t = s) = (\gamma\lambda)^{t-k}\mathbb{1}(t \geq k).$$

Skozi celotno epizodo se napaka posodablja kot

$$\sum_{t=1}^{T} \alpha\delta_t E_t(s) = \alpha \sum_{t=1}^{T} (\gamma\lambda)^{t-k}\delta_t = \alpha(G_k^\lambda - V(S_k)),$$

kar pa je ravno enako kot TD(λ) s pogledom naprej. Če je stanje obiskano večkrat, pa se napaka akumulira v več takih napak pogleda naprej. □

3.5. **Posplošena iteracija strategije.** Do sedaj smo se osredotočali samo na problem napovedovanja - o strategiji nismo veliko govorili, smo pa ugotovili, kako ob dani strategiji dobimo pravo funkcijo vrednosti. Pri dinamičnem programiranju smo to reševali s pomočjo požrešne izboljšave strategije, ki bi tu imela enačbo

$$\pi'(s) = \arg\max_{a \in \mathcal{A}} \mathcal{R}_s^a + \mathcal{P}_{ss'}^a V(s'),$$

a to sedaj ne pride v poštev, saj želimo problem rešiti, brez da poznamo MDP (torej brez da poznamo $\mathcal{P}_{ss'}^a$ in $\mathcal{R}_s^a$). Opaizmo, da lahko prek zamenjave vrednostne funkcije stanj v funkcijo akcij $Q(s,a)$ odpravimo potrebo po poznavanju in dobimo enačbo požrešne izboljšave strategije kot

$$\pi'(s) = \arg\max_{a \in \mathcal{A}} Q(s,a). \tag{10}$$

Tako res lahko uporabimo enega od zgoraj opisanih algoritmov v kombinaciji s tem in rešimo problem upravljanja, torej celoten problem spodbujevalnega učenja. Naletimo pa na novo težavo. Če se vedemo požrešno ni nujno, da raziščemo celoten prostor stanj $\mathcal{S}$. Prišli smo do problema *rasikovanja in izkoriščanja.* Na srečo pa ima problem enostavno rešitev.

Če se agent ne vede popolnoma požrešno, pač pa z verjetnostjo ϵ izbere naključno akcijo, je zagotovljeno, da bo z neničelno verjetnostjo obiskal vsa stanja, hkrati pa se bo strategija vseeno izboljševala. če bo le ϵ dovolj majhen. To nas pripelje do

koncepta ϵ-**požrešnega raziskovanja**, ki deterministično strategijo spremeni v stohastično, kot sledi:

$$\pi(a|s) = \begin{cases} \epsilon/m + 1 - \epsilon & \text{če } a^* = \arg\max_{a \in \mathcal{A}} Q(s,a) \\ \epsilon/m & \text{sicer,} \end{cases} \tag{11}$$

kjer m označuje število akcij, ki so na voljo v stanju s.

Izrek 3.5. *Za vsako ϵ-požrešno strategijo π, je ϵ-požrešna strategija π', pridobljena glede na q_π, izboljšava. Torej $v_{\pi'}(s) \geq v_\pi(s)$.*

Dokaz. □

3.6. **Funkcijska aproksimacija.** Glavni problem zgornjih algoritmov je shranjevanje v tabele. Vsi delujejo na principu grajenja velihih tabel; en vnos za vsako stanje (oz. en vnos za vsako kombinacijo stanja in akcije). To pri velikih MDP-jih ne pride v poštev, saj učenje postane zelo prostorsko zahtevno in pa tudi dolgotrajno - stanje, ki ni obiskano, nima dodeljene vrednosti, torej o njem ne vemo nič. Tabelarne rešitve zato potrebujejo veliko kolino podatvko, oz. veliko količino epizod.

Zato je pomembna nadgradnja algoritmov uporaba funkcijskih aproksimatorjev. Ta koncept je izčrpno preoučevan na področju numerike, statistike in nadzorovanega učenja in načeloma se lahko uporabi katerkoli od znanih aproksimatorjev. Ideja je, da z nekaj utežmi opišemo funkcijo vrednosti, nato pa namesto funkcije neposredno, prilagajamo te uteži. To zmanjša prostorsko zahtevnost algoritmov in pa tudi omogoča, da dodelimo neke (smiselne) vrednosti stanjem, ki še niso bila obiskana. Ocene za funkcije vrednsoti stanja in stanja-akcije sedaj zapišemo

$$\hat{v}(s,w) \approx v_\pi(s),$$
$$\hat{q}(s,a,w) \approx q_\pi(s,a),$$

kjer w zdaj predstavlja vektor uteži, ki ga bomo dejansko spreminjali. Videli bomo, da so koristni predvsem funkcijski aproksimatorji, ki so *odvedljivi* po parametru w. Najpogosteje sta tako v uporabi **linearna aproksimacija** in **nevronske mreže**.

3.6.1. *Gradientni spust.* Najpogosteje uporabljena metoda »treniranja« aproksimatorjev je gradientni spust. Ideja je, da imamo neko funkcijo $J(w)$, ki je odvedljiva po vektorju parametrov w. Poiskati želimo njen (lokalni) minimum. To storimo tako, da izračunamo njen gradient $\nabla_w J(w)$ in nato premaknemo w v negativni smeri tega gradienta

$$\Delta w = -\alpha \nabla_w J(w)$$

kjer je α parameter hitrosti učenja (oz. velikost koraka). Ker gradient pove, v katero smer funkcija narašča, bo premik v negativni smeri povročil, da bomo to vrednost zmanjšali. Parameter α je tu pomemben, saj prek njega lahko dosežemo, da se ne premaknemo predaleč in s tem zgrešimo naš minimum. Uporaba gradienta pa je tudi kriva za to, da v splošnem tako lahko doežemo le lokalni minimum.

Za uporabo gradientnega spusta seveda potrebujemo neko kriterijsko funkcijo, ki jo želimo minimizirati. V našem primeru je smiselna izbira *srednja kvadratična napaka* med pravo vrednostjo stanja in ocenjeno vrednostjo oz.

$$J(w) = E_\pi[(v_\pi(s) - \hat{v}(s,w))^2]$$

in tako premikati uteži v skladu z

$$\Delta w = \alpha E_\pi[(v_\pi(s) - \hat{v}(s,w)) \nabla_w \hat{v}(s,w)],$$

kjer smo pri odvajanju pridobljeno 1/2 skrili v parameter α.

Seveda pa tu pridemo do dveh težav. Prva je uporaba pričakovane vrednosti, ki bi spet pomenila polno znanje o sistemu. To odpravimo z uporabo **stohastičnega gradientnega spusta** tako, da enostavno vzorčimo gradient

$$\Delta w = \alpha(v_\pi(s) - \hat{v}(s,w))\nabla_w \hat{v}(s,w).$$

V povprečju vseeno dobimo enako posodobitev uteži.

Druga težava je zahteva po poznavanju $v_\pi(s)$, kar seveda pri problemu spodbujevalnega učenja ni možno. To težavo rešimo z dosedaj podano teorijo: $v_p i(s)$ zamenjamo z eno izmed tarč, ki smo jih spoznali v poglavju 3. Najbolj pogosto je to tarča algoritma TD(λ), ki nam posodablja uteži v skladu z

$$\Delta w = \alpha(G_t^\lambda - \hat{v}(s,w))\nabla_w \hat{v}(s,w),$$

če uporabljamo algoritem s pogledom naprej oz.

$$\begin{aligned}\delta_t &= R_{t+1} + \gamma\hat{v}(S_{t+1},w) - \hat{v}(S_t,w)\\ E_t &= \gamma\lambda E_{t-1} + \nabla_w \hat{v}(S_t,w)\\ \Delta w &= \alpha\delta_t E_t\end{aligned}$$

če uporabljamo pogled nazaj.

3.6.2. *Linearna aproksimacija.* Najbolj enostaven način aproksimacije funkcij je seveda linerana aproksimacija. Ideja je, da stanje predstavimo z vektorjem

$$x(S) = \begin{bmatrix} x_1(S) \\ x_2(S) \\ \vdots \\ x_n(S) \end{bmatrix}.$$

Zgled 3.6 (Vektor $x(S)$ pri križcih in krožcih)**.** *Igralno ploščo križcev in krožcev lahko predstavimo kot vektor devetih števil, kjer* 1 *pomeni križec,* 0 *pomeni prazno polje in* -1 *pomeni krožec. Če želimo pridelati malo daljši vektor in s tem več uteži, se standardno uporabi binarni vektor stanja, kjer je ideja, da se čez ploščo sprehodimo trikrat: prvih devet števil ima enice povsod, kjer so na plošči križci in drugod ničle, drugih devet ima enice na mestih krožcev in zadnjih devet na praznih mestih.*

Sedaj lahko predstavimo vrednost stanja kot linearno kombinacijo komponent vektorja stanja

$$\hat{v}(s,w) = x(S)^T w = \sum_{i=1}^{n} x_i(S) w_i.$$

V tej situaciji stohastićni gradientni spust konvergira k globalnemu minimumu (CITIRAJ!!!). Dobimo tudi zelo enostavno pravilo posodabljanja, saj velja $\nabla_w \hat{v}(s,w) = x(S)$. To zelo poenostavi računanje in razmislek o teoriji v ozadju. Zato je tudi linearna aproksimacija najbolj teoretično obdelana – konvergenčni dokazi za funkcijske aproksimacije so večinoma narejeni samo za linearni primer.

3.6.3. *Nevronske mreže.* Umetne nevronske mreže so v zadnjem času zelo popularne v nadzorovanem učenju zaradi uspehov pri problemih klasifikacije (MORDA KAK CITAT). Konceptualno so modelirane po možganih: vsebujejo skupke *nevronov*, ki so organizirani v *sloje*. Prvi sloj se imenuje vhodni in ima toliko nevronov, kot je velik naš vhodni podatek. Za problem križcev in krožcev smo v zgledu 3.6 videli, da je to potem 9 ali 27, če uporabljamo binarni vektor. Zadnji sloj je izhodni, vmes pa so *skriti* sloji, ki jih je lahko poljubno mnogo in so lahko poljubne velikosti. Vsi sloji so med sabo povezani, in sicer tako, da so povezani vsi nevroni iz enega sloja z vsemi nevroni iz sosednjega sloja. Izhodni sloj je velik

mal bolj matematični odstavek

problemi odtavek

3.6.4. *Konvergenca pri funkcijskem aproksimiranju.*

3.6.5. *Serijsko učenje.*

4. Namizne igre

Namizne igre (npr. šah, Go, križci in krožci, ...) so proces, ki se zelo lepo poda modeliranju z Markovskim procesom odločanja. Množico stanj $\mathcal{S}$ enostavno zapolnimo z vsemi možnimi konfiguracijami igralne plošče. Množico akcij $\mathcal{A}$ sestavljajo vse legalne poteze, ki jih definirajo pravila igre in trenutno stanje. Prehodno matriko definirajo akcije našega nasprotnika oz. okolja, če je igra stohastične narave (npr. Backgammon - met kocke). Torej vidimo, da so namizne igre okolje, kjer je prehodna matrika neznana in zato učenje prek dinamičnega programiranja ne pride v poštev. Prav tako samo delno poznamo nagradno funkcijo, saj je tudi ta odvisna od dinamike okolja.

4.1. **Nagrajevanje.** Pri namiznih igrah je najpogosteje uporabljena zelo enostavna struktura nagrad. Zmagi in porazu sta dodeljeni vrednosti 1 oz. -1, za izenačenje se vzame vrednost 0. Vsem ostalim ne-končnim stanjem se prav tako dodeli vrednost 0 in tako se doseže, da imamo opravka z igro z vsoto nič tudi znotraj MDP-ja. Občasne spremembe te strukture nagrad so možne, najpogosteje se dodeli neko negativno nagrado vsem stanjem (npr. $-0,1$). Konceptualno naj bi to pomenilo pritisk na agenta da naj čim prej pride do konca igre, t.j. naj čim prej zmaga. Glede na posamezno igro se lahko prilagodi tudi nagrada za izenačenje (če je to želen rezultat, potem ima lahko pozitivno nagrado).

Posebnost pri namiznih igrah je tudi v diskontnem faktorju. Čeprav so vseeno zaželene nagrade, ki pridejo prej, torej krajše igre, se zaradi pojavitve neničelnih nagrad samo ob koncu igre pogosto uporablja diskontni faktor $\gamma = 1$.

4.2. **Pregled konceptov teorije iger.** Formalno gledano imamo z vidika teorije iger pri namiznih igrah opravka z ekstenzivno (kombinatorno) igro za dva igralca z vsoto 0.

4.2.1. *Nashevo ravnotežje.*

4.2.2. *Igre z vsoto nič.*

4.2.3. *Ekstenzivne igre.*

4.3. **Kompleksnost iger.**

4.3.1. *Game tree ...*

Literatura

[1] Richard E. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, 1957.

[2] Richard E. Bellman. A markov decision process. *Journal of Mathematical Mechanics*, (6), 1957.

[3] Imran Ghory. Reinforcement learning in board games. 2004.

[4] David Silver. Introduction to reinforcement learning. https://deepmind.com/learning-resources/-introduction-reinforcement-learning-david-silver, 2015.

[5] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An introduction*. The MIT Press, Cambridge, Massachusetts, 2 edition, 2015.

[6] Csaba Szepesvari. *Algorithms for Reinforcement Learning*. Morgan & Claypool Publishers, Alberta, Canada, 2009.