# Q-učenje kao algoritam metode pojačanog učenja i njegova primjena

**Hinić, Leon Hrid**

**SVEUČILIŠTE U ZAGREBU**
**FAKULTET ORGANIZACIJE I INFORMATIKE**
**VARAŽDIN**

**Leon Hrid Hinić**

# Q-UČENJE KAO ALGORITAM METODE POJAČANOG UČENJA I NJEGOVA PRIMJENA

**ZAVRŠNI RAD**

**Varaždin, 2020.**

**SVEUČILIŠTE U ZAGREBU**

**FAKULTET ORGANIZACIJE I INFORMATIKE**

**V A R A Ž D I N**

**Leon Hrid Hinić**

**Matični broj: 44922/16-R**

**Studij: Informacijski sustavi**

# Q-UČENJE KAO ALGORITAM METODE POJAČANOG UČENJA I NJEGOVA PRIMJENA

**ZAVRŠNI RAD**

**Mentor :**

Dr. sc. Bogdan Okreša Đurić

**Varaždin, srpanj 2020.**

*Leon Hrid Hinić*

**Izjava o izvornosti**

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

*Autor potvrdio prihvaćanjem odredbi u sustavu FOI-radovi*

---

**Sažetak**

Rad obrađuje koncept Q-učenja, algoritma pojačanog učenja, u kontekstu umjetne inteligencije. Uz kratki pregled metoda umjetne inteligencije usredotočen na metode strojnog učenja, rad naglasak stavlja na algoritam Q-učenja. Primjena algoritma prikazana je u praktičnom dijelu rada, u domeni računalnih igara.

**Ključne riječi**: umjetna inteligencija; strojno učenje; q-učenje; pojačano učenje; agent; inteligentni agent;

# Sadržaj

# 1. Uvod

Q-učenje jedna je od metoda pojačanog učenja, koje su samo dio grane umjetne inteligencije koje se naziva strojno učenje. Jedna od najzanimljivijih primjena Q-učenja je izrada kompjuterskih programa koji mogu kroz interakciju sa nekim okruženjem naučiti kako doći do zadanog cilja, bez ikakvih preddefiniranih uputa. Daljnji razvoj ovakve metode mogao bi dovesti do automatizacije širokog spektra aktivnosti zato što funkcionira čak i ako program nema nikakvih saznanja o okruženju u kojem se nalazi što otvara razne mogućnosti. Nisam pronašao niti jedan objavljen znanstveni rad koji na stvarnom primjeru prikazuje funkcioniranje i efektivnost ove metode strojnog učenja. Iz tih razloga ovo područje mi se činilo kao iznimno zanimljivo za daljnje istraživanje.

Nakon kratkog pregleda teorijskih osnova umjetne inteligencije, strojnog učenja i pojačanog učenja kojem pripada i samo Q-učenje, prikazana je primjena Q-učenja u obliku programskog koda. U ovom radu prikazana su četiri programa koji savladavaju video igre pomoću metode Q-učenja. Prvi program pokušat će naučiti kako doći do cilja u jednostavnoj igri pod nazivom *labirint*. Ostala tri će igrati igru *Starcraft 2*.

# 2. Metode i tehnike rada

U ovom radu glavna metoda, kojom su se provodile istraživačke aktivnosti, bila je izrada programskog koda agenata koji igraju određene video igre, te interpretiranja načina na koji su agenti naučili igrati igre.

Sav programski kod koji je korišten za izradu ovog rada napisan je u programskom jeziku Python. Službena stranica opisuje Python kao jasan i moćan objektno orijentirani programski jezik, usporediv s jezicima kao što su Perl, Ruby, Scheme ili Java [1].

Za izradu dijela za Q-učenje korišteni su dodatci *Numpy* i *Pandas*. *Numpy* se koristi zbog određenih matematičkih funcija koje nudi, dok se iz *Pandas* koristi struktura podataka u koju se spremaju podaci o Q - učenju. Za interakciju agenata iz primjera i okruženja koriste se još dva dodatka za Python, a to su Tkinter i pysc2. Tkinter je dodatak za Python koji pruža mogućnost izrade grafičkog sučelja koje u jednom od primjera služi kao okruženje koje agent istražuje. PySC2 je DeepMindova Python komponenta StarCraft II okruženja za učenje (SC2LE). Izlaže API za strojno učenje StarCraft II tvrtke Blizzard Entertainment kao Python RL okruženje [2].

# 3. Umjetna inteligencija i strojno učenje

Da bi mogli objasniti što je točno Q-učenje, prvo trebamo razjasniti pojmove umjetna inteligencija, inteligentni agent i strojno učenje, kakve sve vrste strojnog učenja postoje, te u konačnici nešto više o onoj vrsti kojoj Q-učenje pripada - pojačanom učenju.

## 3.1. Umjetna inteligencija

Umjetna inteligencija je znanstvena disciplina koja se bavi proučavanjem dizajna inteligentnih agenata. [3, str. 1]. U ovom kontekstu kao agent smatramo bilo što, što je u interakciji s okolinom. Inteligentni agent bio bi onaj agent koji pokazuje ponašanje koje možemo smatrati inteligentnim, poduzima korake potrebne za postizanje svojih ciljeva uzimajući u obzir svoju okolinu, uči iz iskustava, te je prilagodljiv na promjenu okoline i ciljeva.[3, str. 1]

Prema Lynton Poole [3, str. 1] središnji znanstveni cilj umjetne inteligencije je razumjeti principe koji omogućuju inteligentno ponašanje u prirodnim ili umjetnim sustavima. U ranim danima razvoja ove znanosti to je uglavnom uključivalo rasuđivanje, predstavljanje znanja, planiranje i učenje, dok se u novije vrijeme koriste za specifičnije zadatke kao što su obrada prirodnog jezika, percepcija i sposobnost kretanja i manipulacije predmetima.

Neapolitan i Jiang [4] navode da su 1955–1956 Allen Newell i Herbert Simon razvili program nazvan Teoretičar logike (eng. *Logic Theorist*) koji je bio namijenjen oponašanju vještina rješavanja problema čovjeka i smatra se prvi program umjetne inteligencije. Program je naučio dokazivati logičke teoreme, te je uspio pronaći kraće dokaze za neke od njih.

Većina inteligentnih agenata napravljena je da rješava točno određen problem zadanog opsega i nije u stanju rješavati probleme izvan svog opsega. To nije slučajnost nego je, zbog trenutnog stanja tehnologije i poznatih metoda, nužnost. Takav oblik umjetne inteligencije se često naziva slaba umjetna inteligencija (eng. *Weak AI*) [5]. Umjetna inteligencija još uvijek nije na razini na kojoj bi se jedan specifični agent mogao primjeniti na više različitih problema, ali se definitivno prema tome teži, te se ta hipotetska umjetna inteligencija naziva jaka umjetna inteligencija (eng. *Strong AI*) ili Umjetna opća inteligencija (eng. *Artificial general intelligence*, kraće AGI).

Hodson [6] navodi da AGI označava umjetnu opću inteligenciju, hipotetički računalni program koji može obavljati intelektualne zadatke kao i čovjek ili bolje od čovjeka. Naravno to se odnosi na sve zadatke koje čovjek može izvršavati zato što su u određenim područjima pojedine umjetne inteligencije već nadišle čovjekove sposobnosti. Jedno od tih područja je i područje video igara, o čemu će biti govora u kasnijim poglavljima ovog rada.

## 3.2. Strojno učenje

Pojam strojnog učenja se veže uz umjetnu inteligenciju, čak se često pogrešno koriste kao sinonimi. Od ova dva pojma, umjetna inteligencija je nadređeni pojam nad pojmom strojnog

učenja u smislu da je strojno učenje samo dio umjetne inteligencije.

Murphy [7, str. 1] definira strojno učenje kao skup metoda koje mogu automatski otkriti obrasce u podacima, a zatim upotrijebiti otkrivene obrasce za predviđanje budućnosti podataka ili da izvrši druge vrste odlučivanja u nesigurnosti (poput planiranja kako prikupiti više podataka!).

### 3.2.1. Pristupi strojnom učenju

Kratki pregled glavnih pristupa strojnom učenju. Ovi pristupi označavaju različite principe izrade algoritama učenja agenata.

#### 3.2.1.1. Učenje pod nadzorom

U prediktivnom ili pod nadzorom pristup učenju, cilj je naučiti uzorak preslikavanja od ulaza $x$ do izlaza $y$, s označenim skupom ulazno-izlaznih parova $D = \{(xi, yi)\}_{i=1}^{N}$ [7, str. 2]. Ulazi $x_i$ se nazivaju značajke, atributi ili kovarijable, dok se $y_i$ nazivaju varijable odgovora (eng. *response variable*).

#### 3.2.1.2. Učenje bez nadzora

Druga glavna vrsta strojnog učenja je opisno ili nenadgledano učenje pristup. Ovdje su nam dati samo unosi,$D = \{x_i\}_{i=1}^{N}$, a cilj je pronaći zanimljive obrasce u podacima [7, str. 2]. To se ponekad naziva otkrivanjem znanja. Ovo je puno slabije definiran problem, jer nije rečeno koje obrasce trebamo tražiti, a ne postoji očit način mjerenja pogreške. Kao primjer Russell i Norvig [8, str. 695] navode da u učenju bez nadzora, taksi agent može postupno razviti koncept dobrih prometnih dana i loših prometnih dana, bez prethodno zadanih etiketiranih primjera.

## 3.3. Pojačano učenje

Za pojačano učenje Sutton i Barto [9, str. 1] navode da je učenje što učiniti da bi maksimizirali vrijednost nagradnog signala pridruživanjem stanja i akcija. Prvi bitan aspekt ovog pristupa je da agent koji uči ovom metodom nema saznanja o tome koliku nagradu donosi pojedina akcija, nego mora sam otkriti iz iskustva, dok je drugi taj da u većini slučajeva poduzimanje određene akcije mijenja i nagrade svih akcija koje slijede nakon.

Agent mora biti u stanju osjetiti stanje svoje okoline u određenoj mjeri i mora biti u stanju poduzeti radnje koje utječu na stanje, to jest mijenjati stanje. Agent također mora imati cilj ili ciljeve koji su u nekoj vezi s okolinom. Ova tri aspekta pojačanog učenja, prepoznavanje trenutnog stanja, poduzimanje akcija koje mijenjaju stanje i neki određeni cilj, možemo promatrati kao Markovljev proces odlučivanja (eng. *Markov decision process*).

Markovljevi procesi odlučivanja (MDP), koji se nazivaju i stohastičkim dinamičkim programiranjem, opsežno su proučavani otkako su prvi put predstavljeni 1960. MDP-ovi su se

uglavnom koristili za modeliranje i rješavanje problema dinamičkog odlučivanja s višerazdobljem pod stohastičkim okolnostima [10, str. 1].

Svaki markovljev proces odlučivanje je uređena četvorka $(S, A, P_a, R_a)$. U kontekstu pojačanog učenja $S$ je skup svih stanja koje okolina i agent mogu poprimiti, $A$ je skup svih akcija koje agent može izvršiti, $P_a$ koji se ponekad označava i kao $P_a(s, s^{'})$ označava vjerojatnost da će akcija $a$ u stanju $s$ dovesti do prilaza u stanje $s^{'}$, a $R_a$ ili konkretnije $R_a(s, s^{'})$ predstavlja numeričku vrijednost nagrade koja se dobija izvršavanjem akcije $a$ i prijelaskom iz stanja $s$ u $s^{'}$. Također bitno je napomenuti da za potrebe strojnog učenja, ili konkretno pojačanog učenja kao markovljev proces odlučivanja promatramo diskretan markovljev proces odlučivanja, što znači da interakciju agenta i okoline promatramo u diskretnim vremenskim koracima.

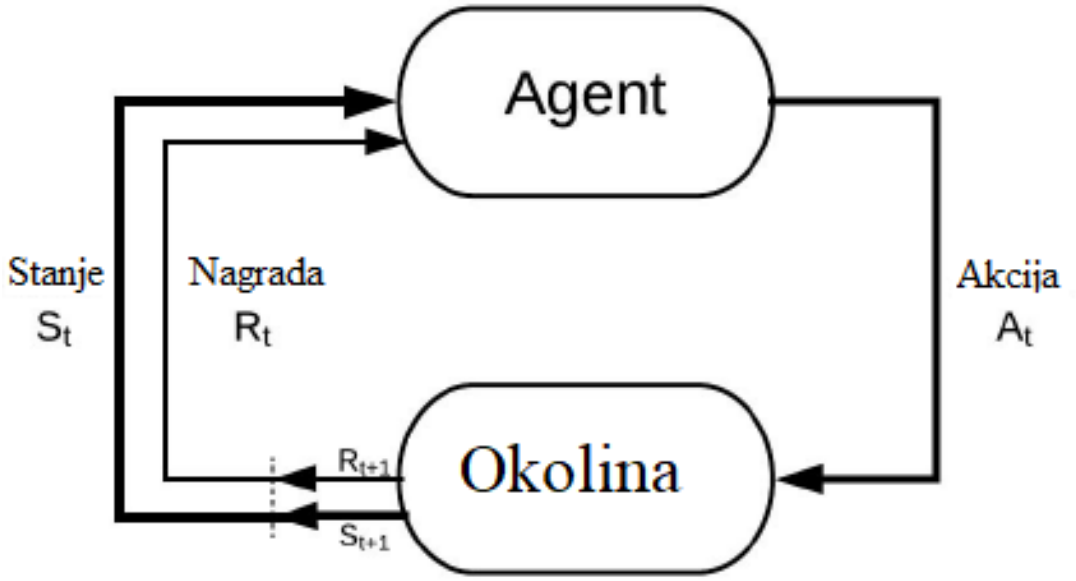
Jedan od izazova koji se javljaju u metodi pojačanog učenja, a ne u drugim vrstama strojnog učenja, je pronaći ravnotežu između istraživanja nepoznatog i korištenja već poznatog. Da bi maksimizirao nagradu, agent mora preferirati radnje koje je pokušao u prošlosti i za koje je utvrdio da su učinkovite u stvaranju nagrade. No, da bi otkrio takve radnje, mora pokušati radnje koje prije nije odabrao.

Prema Sutton i Barto [9, str. 6] uz agenta i okolinu, mogu se prepoznati četiri glavna podskupa sustava pojačanog učenja: politika, nagradni signal, funkcija vrijednosti i, po želji, model okruženja. Politika definira način ponašanja agenta u određenom trenutku. Politiku agenta možemo objasniti kao ono što je agent naučio, koju akciju izvesti ovisno o trenutnom stanju agenta i okoline. Različite vrste pojačanog učenja na različite načine dolaze do politike agenta, dok je neke vrste uopće ne koriste, kao što je i Q-učenje. Nagradi signal je ono što, nakon svakog diskretnog koraka, okolina vraća agentu u obliku broja koji prikazuje dobivenu nagradu za prošlu akciju.

Sutton i Barto [9, str. 6] definiraju funkciju vrijednosti kao ono što specificira što je dugoročno dobro. Grubo govoreći, vrijednost stanja je ukupni iznos nagrade koju agent može očekivati da će akumulirati u budućnosti, počevši od tog stanja.

Model okruženja se koristi za predviđanje budućih stanja te promjene okoline. Neke metode pojačanog učenja koriste modele i nazivaju se metode na bazi modela (eng. *model-based*), dok se one metode koje ne koriste model nazivaju metode bez modela (eng. *model-free*).

Na sljedećoj slici (slika 1) prikazano je međudjelovanje između agenta i okruženja u pojačanom učenju.




Slika 1: Međudjelovanje agenta i okruženja [11]

Agent poduzima akciju (eng. *Action*) $A_t$ na koju okruženje (eng. *Environment*) odgovara promjenom stanja (eng. *State*), to jest podatkom koji opisuje u kojem se stanju okruženje trenutno nalazi $S_t$, i nagradom za izvršenu akciju $R_t$ koja je u pravilu u brojčanom obliku.

Te tri navedene varijable uz još sljedeće stanje, ono u koje okolina prelazi nakon izvršenja akcije, čine uređenu četvorku koju promatramo kao MDP koji treba rješiti, to jest kao uređenu četvorku $(S_t, A_t, R_t, S_{t+1})$ .

# 4. Q-učenje

Q-učenje je oblik pojačanog učenja bez korištenja modela [12]. U svom najosnovnijem obliku Q-učenje se bazira na zapisivanju određenih vrijednosti u tablicu, koje se također nazivaju i Q vrijednosti, te na temelju tih vrijednosti odabire sljedeću akciju koju će izvršiti. Nakon svakog diskretnog koraka, okolina daje povratnu informaciju agentu koji onda procesira te informacije u obliku promjene vrijednosti u tablici.

"Q" u Q-učenju označava kvalitetu (eng. *quality*). Kvaliteta ovdje predstavlja koliko je dana radnja korisna u stjecanju neke buduće nagrade [11].

Kao što je ranije spomenuto promatramo problem kao MDP koji se sastoji od uređene četvorke $(S_t, A_t, R_t, S_{t+1})$, na čijem se temelju određuju Q vrijednosti.

Sutton i Barto [9, str. 131] definiraju određivanje Q vrijednosti sljedeći način:

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

$Q(S_t, A_t)$ predstavlja trenutnu vrijednost u tablici za izvršenu akciju $A_t$ u stanju $S_t$. $\alpha$ se naziva koeficijent učenja, dok je $R_{t+1}$ nagrada koju je agent dobio od okoline za izvršenu akciju. $\gamma$ je koeficijent smanjenja buduće nagrade, svaka sljedeća akcija donosi manju nagradu zato što je agentu bilo potrebno više vremena da postigne određeni cilj, $maxQ(S_{t+1}, a)$ je maksimalna vrijednost sljedećeg stanja u tablici koje se može postići određenom akcijom.

Dio unutar uglate zagrade odnosno $[R_{t+1} + \gamma maxQ(S_{t+1}, a) - Q(S_t, A_t)]$ proizlazi iz pristupa strojnom učenju koji je prethodio Q - učenju, a naziva se učenje po vremenskoj razlici (eng. *Temporal difference learning*) ili često skraćeno kao TD. TD je vrsta koja je nastala kao spoj druge dvije prethodne vrste učenja Monte Carlo učenja i dinamičkog programiranja.

Sutton i Barto [9, str. 119] objašnjava da poput Monte Carlo metoda, TD metode mogu izravno učiti iz sirovog iskustva bez modela dinamike okoliša. Poput dinamičkog programiranja, TD metode ažuriraju procjene djelomično na temelju ostalih naučenih procjena, ne čekajući konačni ishod. To znači da je agent sposoban funkcionirati bez ikakvog prethodnog znanja o svojem okruženju, te nema potrebe za čekanjem do kraja epizode da bi se izveo korak učenja (određivanja Q vrijednosti), nego do učenja dolazi nakon svake izvršene akcije. U ovom smislu jednu epizodu podrazumjeva niz stanja od početnog $S_{t_0}$ , sve dok $S_{t+1}$ nije stanje u kojem staje igra, odnosno završno stanje.

Prema Morvanu [13], uz promjenjenu notaciju da bude konzistentna sa onom kojom su do sada bile opisane varijable, cijeli algoritam Q-učenja izgleda ovako:

Inicijaliziraj $Q(S_t, A_t)$ proizvoljno

Ponavljaj (za svaku epizodu):

Inicijaliziraj početno stanje $S_{t_0}$

Ponavljaj (za svaki korak u epizodi):

Izaberi akciju $A_t$ koristeći politiku odabira prema $Q$ (npr. $\epsilon$ - greedy)

Izvrši akciju $A_t$, promotri $R_t$ i $S_{t+1}$

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

$S_t \leftarrow S_{t+1}$

Dok $S_t$ nije konačan

Inicijalizacija $Q(S_t, A_t)$ je proizvoljna zato što će već u prvom koraku biti zamjenjena stvarnim vrijednostima koje će agent dobiti od okoline. U određenim situacijama ovaj korak nije potreban, moguće je dodavati nove vrijednosti u Q tablicu tek kada prvi put dođemo u to stanje, na tom principu funkcioniraju primjeri koje ćemo objasniti u sljedećem poglavlju. Što je točno jedan korak unutar epizode definira se ovisno o potrebi okruženja koje agent pokušava savladati, u primjerima korak će biti definiran isto kao i u gore navedenom primjeru, to jest izvršavanje akcije i promatranje nagrade i stanja koju okruženje zauzima.

Na samom početku igre agent nema nikakvih saznanja o svom okruženju. U svakom sljedećem koraku igre agent bira najbolju moguću akciju, onu koja dovodi u stanje sa najvećom nagradom, u slučaju Q-učenja onu sa najvećom Q vrijednosti, a u slučaju da takvog stanja nema, pseudoslučajno izabire akciju, to se naziva politika agenta. Pseudoslučajan odabir akcije je iznimno važan dio Q-učenja čak i kada agent zna za postojanje akcije koja vodi u stanje s višom nagradom. Bez pseduoslučajnog odabira agent bi ostao zaglavljen u prvom nizu akcija koji mu je donio nagradu iako bi taj niz gotovo sigurno bio daleko od onog optimalnog kojeg agent zapravo pokušava pronaći.

Nakon izvršavanja akcije i promatranja reakcije okoline dolazi korak učenja koji se sastoji samo od izračunavanja Q vrijednosti i zapisivanja iste u Q tablicu. Sljedeće stanje postaje trenutno stanje i kreće sljedeći korak, i tako sve do kraja epizode.

Klasično Q-učenje ima dva velika problema, prvi je da se agenti koji koriste Q-učenje slabo prilagođavaju na povećanje mogućih stanja i akcija, rezultira znatno sporijim učenjem, što će se najbolje vidjeti u rezultatima praktičnih primjera u ovom radu, a drugi se naziva pristranost maksimizacije (eng. *maximization bias*).

Da biste vidjeli zašto dolazi do pristranosti maksimizacije, razmotrite jedno stanje s gdje postoji mnogo radnji $a$ čije su istinske vrijednosti $q(s, a)$ jednake nuli, ali čije su procijenjene vrijednosti $Q(s, a)$ nesigurne, pa su neke raspodijeljene iznad i neke ispod nule. Maksimum stvarnih vrijednosti je nula, ali maksimum procjena je pozitivan, što izaziva pozitivnu pristranost. [9, str. 134]

Postoje različita poboljšanja metode Q-učenja koja rješavaju ove probleme koji neće biti spomenuti zato što će svi praktični primjeri u ovom radu koristiti samo klasičnu verziju Q-učenja.

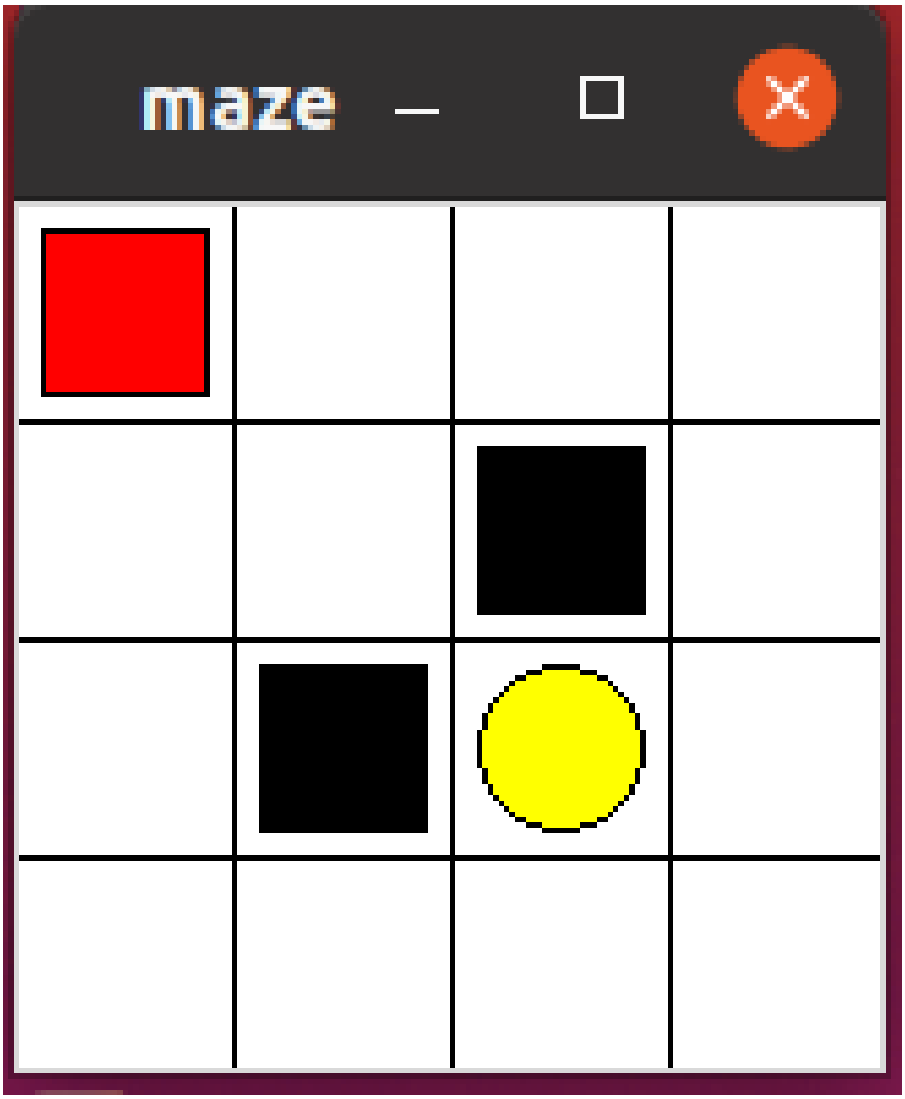
Da bi agent bio barem donekle uspješan u realnim okruženjima potreban je veliki broj epizoda za trening. Primjere realnih okruženja i način na koje ih agenti savladavaju prikazani su u sljedećem poglavlju.

# 5. Praktični primjer: Izrada agenta koji igra videoigre

U ovom poglavlju prikazan je razvoj agenta koji igra video igre te uči metodom Q-učenja, prvo na jednostavnom primjeru u igri labirint, a kasnije na znatno kompleksnijoj igri kao što je Starcraft 2.

## 5.1. Labirint

Da bi na primjeru demonstrirali metodu Q-učenja koristit ćemo jednostavni labirint kao igru koju agent pokušava naučiti. Izgled labirinta prikazan je na slici 2




Slika 2: Igra labirint u početnom stanju [Autorski rad]

Crveni kvadrat je onaj kojim agent upravlja i istražuje okolinu u kojoj se nalazi. Žuti krug je cilj do kojeg agent treba dovesti crveni kvadrat, dok su crni kvadrati prepreke u koje ako agent uđe automatski gubi igru. Agent može birati između 4 akcije, u kojem od 4 moguća smjera će se pomaknuti, te se taj odabir još smanjuje ovisno o lokaciji na kojoj se agent trenutno nalazi,

na primjer u slučaju da se nalazi u gornjem lijevom kutu kao na slici moguća su samo 2 smjera kretanja, dolje i desno.

Zbog malog broja mogućih stanja, akcija, i činjenice da se okruženje ne mijenja, agent u samo par epizoda pronalazi optimalan put do kraja, te ga konstantno ponavlja osim u slučaju pseudoslučajnog izbora akcije.

Sav kod prikazan u ovoj sekciji napravljen je po uzoru na upute iz videa [14]. Za početak prikazat ćemo na koji način agent uči kroz klasu QLearning.

```
import numpy as np
import pandas as pd

class QLearning:
    def __init__(self, actions, learningRate = 0.01, rewardDecay = 0.9, eGreedy =
        0.9):
        self.actions = actions
        self.learnRate = learningRate
        self.gamma = rewardDecay
        self.epsilon = eGreedy
        self.qTable = pd.DataFrame(columns = self.actions, dtype = np.float64)
```

Na početku uvozimo module *Numpy* i *Pandas* da bi ih kasnije mogli koristiti. Funkcija *init* je konstruktor u python programskom jeziku, ona se poziva kada se generira objekt klase *QLearning* . Kao parametri proslijeđuju se *self, actions, learningRate, rewardDecay i eGreedy* . *Self* označavu samu klasu *QLearning* te pomoću toga pristupamo atributima koji pripadaju klasi *QLearning*. Parametar *actions* označava skup svih akcija koje agent može izvršiti, koristi se za generiranje Q tablice prosljeđivanjem kao jedan od parametara *DataFrame()*funkciji iz *Pandas* dodatka.

*LearningRate* i *rewardDecay* predstavljaju $\alpha$ i $\gamma$ koji će se koristit za izračunavanje Q vrijednosti prilikom učenja agenta. *eGreedy* je koeficijent pseudoslučajnog odabira akcije, te utječe na politiku agenta. Vrijednost ova tri parametra definiraju se prilikom definicije funkcije da bi kasnije lakše mijenjali. U samom kodu funkcije svi parametri se spremaju u atribute klase i generira se Q tablica.

```
def CheckStateExist(self, state):
        if state not in self.qTable.index:
            #dodaj novo stanje u Q tablicu
            self.qTable = self.qTable.append(
                pd.Series(
                    [0]*len(self.actions),
                    index = self.qTable.columns,
                    name = state,

                )
            )
```

Ova funkcija se koristi isključivo unutar drugih funkcija klase *QLearning* za provjeru postojanja stanja proslijeđenog kao parametar. Ako stanje ne postoji dodaje novo stanje u Q tablicu. Ovime zaobilazimo postavljanje tablice svih mogućih stanja unaprijed. Umjesto toga

u init funkciji generiramo tablicu čija je jedna dimenzija sve moguće akcije koje agent može poduzeti zato što nam je to poznato, a ovu funkciju koristimo da svako stanje dodamo kao drugu dimenziju u tu istu tablicu tek kada se agent nađe u tom stanju.

```
def ChooseAction(self, observation):
        self.CheckStateExist(observation)
        if np.random.uniform() < self.epsilon:
            #najbolja akcija
            stateAction = self.qTable.loc[observation, :]

            #random ako imaju istu vrijednost
            action = np.random.choice(stateAction[stateAction == np.max(stateAction)
                ].index)

        else:
            #random akcija
            action = np.random.choice(self.actions)
        return action
```

Funkcija *ChooseAction* služi za odabir akcije koju će agent izvršiti, prima dva parametra, *self* koji smo već ranije objasnili i *observation* koji predstavlja stanje koje igra vraća agentu. Na početku poziva se druga funkcija *CheckStateExists()* kojoj se prosljeđuje *observation* za provjeru postojanja proslijeđenog stanja.

```
if np.random.uniform() < self.epsilon:
```

U ovoj linij koda definira se u kolikom će omjeru agent istraživati nepoznato ili koristiti poznate vrijednosti. *np.random.unfirom()* je funkcija iz *pandas* dodatka koja vraća jednoliko raspoređene pseudoslučajne vrijednosti u rasponu od 0 do 1, te provjeravamo je li ta vrijednost manja od *self.epsilon* varijeble u koju je u konstruktoru spremljena vrijednost paramtera *eGreedy = 0.9*, što znači da se u približno 10 posto slučajeva neće izvršiti ovaj *if* blok koda.

U slučaju da se ne izvrši akcije se odabire pseudoslučajno iz svih mogućih akcija, dok se u slučaju ulaska u *if* blok koda odabire najbolja akcija po tome koja dovodi u stanje sa najvećom Q vrijednosti u tablici, a u slučaju da postoji više akcija koje imaju maksimalnu Q vrijednost jedna od njih se odabire pseudoslučajno.

```
def learn(self, state, action, reward, stateNext):
        self.CheckStateExist(stateNext)
        qPredicted = self.qTable.loc[state,action]

        if stateNext != 'terminal':
            qTarget = reward + self.gamma * self.qTable.loc[stateNext,:].max()
        else:
            qTarget = reward

        #update tablice
        self.qTable.loc[state,action] += self.learnRate * (qTarget - qPredicted)
```

Ovaj dio koda zadužen je za učenje agenta, ima pet parametara *self, state, action, reward i stateNext*. Izuzevši parametar *self* ostali parametri su redom stanje, akcija, nagrada i sljedeće stanje u koje igra prelazi. Kao i u ostalim funkcijama ove klase na samom početku

provjerava se postoji li zapis o stanju u koje igra prelazi u Q tablici i dodaje se prema potrebi. Radi lakšeg zapisa u kodu funkciju oblika:

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

podijeljena je na više dijelova. Varijabla *qPredicted* zamjenjuje $Q(S_t, A_t)$, a *qTarget* zamjenjuje $R_{t+1} + \gamma maxQ(S_{t+1}, a)$. U slučaju da je akcija dovela u konačno stanje, tada vrijednost *qTarget* postaje jednaka nagradi, zato što vrijednost sljedećeg stanja je nula i shodno tome cijeli dio $\gamma maxQ(S_{t+1}, a)$ je jednak nuli. Zadnja linija funkcije predstavlja cijelu formulu samo sa određenim dijelovima zamjenjenima, odnosno već izračunatima u prethodnim linijama funkcije.

U ovom primjeru labirinta postoje još dvije python skripte, jedna pod imenom *maze.py* u kojoj se nalazi definicija okruženja odnosno sama igra koju agent igra i neće biti objašnjena u ovom radu, a druga je *Run.py* u kojoj se pozivaju funkcije iz *Brain.py* u kojoj se nalazi klasa *QLearning* i *maze.py*, to je ona skripta koju zapravo pokrećemo i pomoću koje se sve odvija. Sastoji se od jedne funkcije *update()* i bloka koda koji se izvršava kada pokrenemo skriput unutar *if* bloka. Skripta izgleda ovako:

```
from maze import Maze
from RL_brain import QLearning

def update():
    for episode in range(100):
        #čpoetna observacija
        observation = env.reset()

        while True:
            #novi env
            env.render()

            #odabir akcija
            action = RL.ChooseAction(str(observation))

            observationNext, reward, done = env.step(action)

            RL.learn(str(observation), action, reward, str(observationNext))

            #zamjena observation
            observation = observationNext

            if done:
                break
    #kraj igre
    print('kraj igre')
    env.destroy()


if __name__ == "__main__":
    env = Maze()
    RL = QLearning(actions = list(range(env.n_actions)))

    env.after(100, update())
```

```
env.mainloop
```

Prve dvije linije koda odnose se na uvoz druge dvije već spomenute skripte da bi ih mogli koristiti. Funkcija *update()* izvodi cijeli proces interakcije agenta i okruženja. Na početku otvaramo *for* blok koji se ponavlja sto puta, broj je proizvoljan i definira koliko epizoda agent izvršava. Epizoda može završiti i u manjem broju koraka ako agent uđe u crni kvadrat čime gubi igru ili ako uđe u žuti krug čime pobjeđuje igru, a informacija o tome je li gotova epizoda ili ne, spremljena je u varijablu *done* koju okruženje vraća pozivom funkcije *step (action)*. Taj parametar se dobio pozivanjem funkcije *ChooseAction* iz klase *QLearning*. Ostale dvije informacije koje igra vraća su *observationNext* i *reward* koje sadrže sljedeće stanje i nagradu za izvršenu akciju. Nakon toga se ti podaci koriste u pozivu funkcije *learn(str(observation), action, reward, str(observationNext))* gdje predstavljaju sve potrebne informacije za izračunavanje Q vrijednosti, a to su redom trenutno stanje, izvršena akcija, nagrada i sljedeće stanje, te se stanje u koju je igra prešla zapisuje kao trenutno stanje u kojem se igra nalazi.

U *if* bloku nalaze se definicije varijabli i poziv funkcije *update()* , što predstavlja onaj kod koji se stvarno izvršava na onaj način kako je bilo definirano o svom prijašnjem kodu.

Na primjeru ove jednostavne igre prikazan i objašnjen stvarni način na koji agent savladava jednostavnu igru, u obliku programskog koda jezika python. Iako je ovo jednostavan primjer, programski kod Q-učenja će ostati isti čak i sljedećim, kompliciranijim primjerima.

## 5.2. Starcraft 2

StarCraft II je znanstveno-fantastična videoigra strategije u stvarnom vremenu koju je razvila i objavila tvrtka Blizzard Entertainment, u svijetu je objavljena u srpnju 2010. godine [15]. Igra se bazira na ratu između tri rase, Terran, Protoss i Zerg. Svaka rasa ima zasebne građevine i jedinice koje može izgraditi, te shodno tome i različite načine i strategije kojima dolaze do pobjede.

Igra se odvija u stvarnom vremenu što znači da igrači trebaju istovremeno izvršavati veliki broj radnji kao što su izgradnja vlastite baze, izviđanje protivničke baze, te u kasnijim stadijima igre čak i borbu protiv neprijateljske vojske. Navedene činjenice dovode do vrlo kompleskne igre gdje u svakom trenutku ima velik broj odluka koje igrač treba donijeti, iz tog razloga će u praktičnim primjerima agent biti ograničen na manji broj akcija, te će na primjeru jednog agenta biti prikazano što se dogodi kada agent ima više akcija na izbor.

U nastavku prikazana su tri različita agenta koji igraju igru Starcraft 2. Sva tri agenta koriste Q-učenje da bi savladali igru, što u slučaju ovih primjera znači uništenje svih neprijateljskih građevina i jedinica, ali svaki agent to postiže na jedinstven način.

Sav kod prikazan u sljedećim sekcijama napravljen je po uputama koje je Brown objavio na svojim blogovima, [16] [17] [18] [19] [20], uz osobne dodatke koda i korištenje koda sa službene stranice pysc2 okruženja [21] i kodovima za prepoznavanjem svih jedinica na ekranu u igri pronađenih u obliku Pastebin-a. [22].

Svaki od agenata sastoji se od četiri datoteke. Dvije *Python* skripte *Brain.py* i *Run.py* i

dva oblika u kojima su spremljene Q tablice koji agenti koriste da bi mogli nastaviti učenje i ne kretati ispočetka kod svakog pokretanja koda, *QTABLE.gz* koji je oblik koji agenti koriste za učitavanje prilikom pokretanja i *QTABLE.csv* koji je u obliku čitljivom ljudima da bi na jednostavan način mogli vidjeti stvarni izgled Q tablica.

*Run.py* skripta je ista kod svih agenata, u njoj su samo definirani paremetri potrebni prilikom pokretanja igre, na primjer koje frakcije unutar igre igraju agent i njegov protivnik, koja je težina mape i slično. *Brain.py* je ona skripta koja se mijenja od agenta do agenta jer je u njoj opisano na koji način agent uči, kako se ponaša i dobiva povratne informacije od okruženja odnosno video igre.

U prvom agentu naveden je i objašnjen sav kod agenta, a u ostalim primjerima prikazani su i objašnjeni samo oni djelovi koda koji su drugačiji za tog agenta. Također klasa koja je zadužena za dio Q-učenja i biranja akcija *QLearningTable*, neće biti prikazana zato što je identična kao i klasa *QLearning* iz primjera agenta koji savladava labirint.

## 5.2.1. Agent 1

Prvi agent je najjednostavniji, ima najmanji mogući broj akcija, ne prati pozicije gdje se neprijatelj nalazi i napada samo točno predodređene koordinate na mapi. Zbog tih čimbenika nije u stanju efektivno naučiti igrati niti jednu drugu mapu osim one za koju je izrađen, a to je mapa pod nazivom *Simple64*.

### 5.2.1.1. Run.py

Kako je skripta *Run.py* ista kod sva tri agenta ona će biti prva objašnjena i izgleda ovako:

```
from Brain import QLearningAgent
from pysc2.agents import base_agent
from pysc2.env import sc2_env
from pysc2.lib import actions, features
from absl import app
```

U prvom dijelu definiramo sve module koji će nam biti potrebni za pokretanje skripte. *Brain* je naša skripta u kojoj je definiran agent i to u klasi *QLearningAgent*. Ostali moduli koji počinju sa pysc2, odnosno *pysc2.agents* ,*pysc2.env*, *pysc2.lib*, su oni koji implementiraju okruženje kojim će naš agent igrati i komunicirati s video igrom, nisu dodani svi odjednom zato što su ti moduli veoma opsežni pa koristimo samo one dijelove koji su nam potrebni. Zadnji modul je *absl* koji se koristi samo za pokretanje ove skripte na samome kraju.

```
def main(unused_argv):
    agent = QLearningAgent()
    try:
        while True:
            with sc2_env.SC2Env(
                map_name="Simple64",
                players=[sc2_env.Agent(sc2_env.Race.terran),
```

```
                    sc2_env.Bot(sc2_env.Race.zerg,
                                sc2_env.Difficulty.very_easy)],
            agent_interface_format=features.AgentInterfaceFormat(
                feature_dimensions=features.Dimensions(screen=84, minimap=64)),
            step_mul=16,
            game_steps_per_episode=0,
            visualize=True) as env:

            agent.setup(env.observation_spec(), env.action_spec())


            timesteps = env.reset()
            agent.reset()


            while True:
                step_actions = [agent.step(timesteps[0])]
                if timesteps[0].last():
                    break
                timesteps = env.step(step_actions)

    except KeyboardInterrupt:
      pass
```

U ovom dijelu definirana je *main()* funkcija čiji je parametar potreban da bi se funkcija mogla izvršiti putem *app* dijela *absl* dodatka. Na početku definiramo i inicijaliziramo agenta koji će igrati igru. Koristimo *try - except* programski konstrukt da bi omogućili izlaz iz funkcije u bilo kojem trenutku pritiskom jedne od kombinacija koje šalju prekidni signal (npr. ctrl + Q). Ostatak koda definira parametre za pokretanje igre, tko su igrači, koju će mapu igrati, brzina kojom agent može igrati, te hoće li se prikazivati pojednostavljena grafika igre.

```
if __name__ == "__main__":
  app.run(main)
```

Ovdje se samo poziva *main()* funkcija koju smo prethodno definirali svim ostalim prikazanim kodom.

#### 5.2.1.2. Agent 1 Brain.py

Nakon zajedničkog *Run.py* , sada će biti prikazan specifični *Brain.py* za prvog agenta.

```
import random
import math
import numpy as np
import pandas as  pd
import os.path

from pysc2.agents import base_agent
from pysc2.lib import actions, features, units
```

```
DATA_FILE = 'QTABLE_1'
```

Ovaj dio koda je gotovo identičan za sva tri agenta zato što koriste iste osnovne module za *pysc2* kao i ostale potrebne za izračunavanje određenih varijabli (*random, math*) i jedan o kojem nije bilo riječi do sada, *os.path* a koristi se za učitavanje i spremanje generiranih Q tablica, na mjesto definiranu u *DATAFILE* varijabli.

```
Action_No_Action = actions.FUNCTIONS.no_op.id
Action_Select_Point = actions.FUNCTIONS.select_point.id
Action_Build_Supply_Depot = actions.FUNCTIONS.Build_SupplyDepot_screen.id
Action_Build_Barracks = actions.FUNCTIONS.Build_Barracks_screen.id
Action_Build_Factory = actions.FUNCTIONS.Build_Factory_screen.id
Action_Train_Marine = actions.FUNCTIONS.Train_Marine_quick.id
Action_Train_Hellion = actions.FUNCTIONS.Train_Hellion_quick.id
Action_Select_Army = actions.FUNCTIONS.select_army.id
Action_Attack_Minimap = actions.FUNCTIONS.Attack_minimap.id
Action_Build_Refinery = actions.FUNCTIONS.Build_Refinery_screen.id
Action_Train_Reaper = actions.FUNCTIONS.Train_Reaper_quick.id
```

U ovom dijelu spremamo prava imena funkcija koje agent može izvršiti (koje vidimo se desne strane znaka = ) u varijable koje smo sami imenovali radi lakšeg korištenja.

```
#akcije za odabir
ACTION_DO_NOTHING = 'donothing'
ACTION_SELECT_SCV = 'selectscv'
ACTION_BUILD_SUPPLY_DEPOT = 'buildsupplydepot'
ACTION_BUILD_BARRACKS = 'buildbarracks'
ACTION_SELECT_BARRACKS = 'selectbarracks'
ACTION_BUILD_MARINE = 'buildmarine'
ACTION_SELECT_ARMY = 'selectarmy'
ACTION_ATTACK = 'attack'



possible_actions = [
    ACTION_DO_NOTHING,
    ACTION_SELECT_SCV,
    ACTION_BUILD_SUPPLY_DEPOT,
    ACTION_BUILD_BARRACKS,
    ACTION_SELECT_BARRACKS,
    ACTION_BUILD_MARINE,
    ACTION_SELECT_ARMY,
    ACTION_ATTACK,
]
```

U prvom djelu definiramo listu akcija mogućih akcije iz kojih će agent birati jednu po jednu za izvršavanje, a u polju *possible actions* samo spremamo tu listu akcija. Ove akcije nisu iste one iz prošlog dijela koje se koriste za stvarne pozive funkcija, ove koriste isključivo za odabir jedne od opcija u koraku biranja akcije za izvršavanje.

```
if __name__ == "__main__":
  #Definicije unit-a po ID
```

```
Unit_Terran_Commandcenter = 18
Unit_Terran_Scv = 45
Unit_Terran_Supply_Depot = 19
Unit_Terran_Barracks = 21

#potrebne konstante
Player_Id = features.SCREEN_FEATURES.player_id.index
Player_Relative = features.SCREEN_FEATURES.player_relative.index
Player_Self = 1
Unit_Type = features.SCREEN_FEATURES.unit_type.index
Not_Queued = [0]
Queued = [1]
```

Ovdje također zapisujemo neke konstante i stvarne nazive funkcija u oblike koje ćemo koristiti radi lakšeg pristupanja tim varijablama. U prvom djelu spremamo brojčane vrijednosti elemenata igre (onako kako ih pysc2 dodatak vidi) u varijable imenovane na način da bi mogli razumjeti što točno koja predstavlja. Agent 1 ima najmanji broj varijabli u ovom dijelu, zato što ima najmanji broj akcija i elemenata koje koristi, da bi imao što brži proces učenja.

```
Reward_Unit_Destroyed = 0.2
Reward_Building_Destroyed = 0.5
```

Ovdje definiramo koliko vrijedi koja nagrada koju ćemo kasnije pridružvati određenim akcijama. Iz samog imena već možemo vidjeti da će je nagrada za uništavanje neprijateljske vojske 0.2 dok je 0.5 za uništavanje neprijateljske zgrade.

```
class QLearningAgent(base_agent.BaseAgent):
    def __init__(self):
            super(QLearningAgent, self).__init__()
            self.qLearn = QLearningTable(actions=list(range(len(possible_actions))))
            self.killed_unit_score_past = 0
            self.killed_building_score_past = 0
            self.previous_state = None
            self.previous_action = None

            if os.path.isfile(DATA_FILE + '.gz'):
                self.qLearn.qTable = pd.read_pickle(DATA_FILE + '.gz', compression='
                    gzip')
```

U ovoj funkciji definiramo što se odvija prilikom kreiranja objekta klase *QLearningAgent* koja naslijeđuje funkcije iz klase *baseagent.BaseAgent* koja se nalazi unutar pysc2 okruženja. Definira se *qLearn* koji je objekt klase sa *QLearningTable* preko kojeg će se pozivati ostale funkcije za Q-učenje, i varijable koje pamte prošle uništene jedinice i građevine kao i prethodno stanje i akciju, koje će se koristiti u dijelu učenja. Na kraju *if* provjerava postoji li već datoteka u koju je spremljena Q tablica koji agent može učitati, ako postoji učitava je kao *qTable* koji je dio *qLearn* objekta.

```
def transformLocation(self, x, x_distance, y, y_distance):
        if not self.base_top_left:
            return [x - x_distance, y - y_distance]
        return [x + x_distance, y + y_distance]
```

Prikazana je jednostavna funkcija koja pretvara koordinata proslijeđene kao paramtere u vrijednosti koje ćemo upotrijebiti kao stvarne koordinate. Koristi se prilkom izgradnje građevina, i osigurava da agent ne pokuša sagraditi građevinu izvan mape. U pravom pozivu ove funkcije *x* i *y* bit će koordinate neke već postojeće građevine kraj koje iz strateških razloga želimo sagraditi građevinu, dok će *x distance* i *y distance* biti udaljenosti od te referentne građevine, te će ova funkcija provjeriti u kojem dijelu mape se nalazi agentova baza i shodno tome dodati ili oduzeti koordinate od referentne točke.

```
def step(self,obs):
        super(QLearningAgent, self).step(obs)
        player_y, player_x = (obs.observation['feature_minimap'][Player_Relative] ==
             Player_Self).nonzero()
        if player_y.any() and player_y.mean() <= 31:
            self.base_top_left = 1
        else:
            self.base_top_left = 0
```

Ovo je početak treće i zadnje funkcije u klasi *QLearningAgent*. Na samo početku agent pronalazi sam svoju poziciju na mapi pomoću *observation* dijela pysc2 okruženja, koristeći ranije definirane konstante *PlayerRelative* i *PlayerSelf* i mapu unutar igre, te se taj podatak o lokaciji zapisuje u varijablu *base top left* (1 za istinu i 0 za neistinu).

```
unit_type = obs.observation['feature_screen'][Unit_Type]

        depot_y, depot_x = (unit_type == Unit_Terran_Supply_Depot).nonzero()
        if depot_y.any():
            supply_depot_count = 1
        else:
            supply_depot_count = 0
        barracks_y, barracks_x = (unit_type == Unit_Terran_Barracks).nonzero()
        if barracks_y.any():
            barracks_count = 1
        else:
            barracks_count = 0

        supply_limit = obs.observation['player'][4]
        army_supply = obs.observation['player'][5]

        killed_unit_score = obs.observation['score_cumulative'][5]
        killed_building_score = obs.observation['score_cumulative'][6]

        current_state = [
            supply_depot_count,
            barracks_count,
            supply_limit,
            army_supply,
        ]
```

Nakon pronalaska baze agenta na red dolazi prepoznavanje trenutnog stanja, koje je iznimno važno zato što će agent putem Q-učenja promjeni stanja pridruživati vrijednost, isplati li se prelazak u to stanje ili ne. Također koristeći prethodno definirane konstante provjeravamo

posjeduje li agent depo (eng. *depot*) ili vojarnu (eng. *barracks*). Također se od okruženja dobivaju informacije o tome koliko maksimalno agent može imati vojske (varijabla *supplay limit*) i koliko trenutno ima vojske (varijabla *army supply*. Ove četiri varijable konstituiraju trenutno stanje u svakom koraku. Isto tako ovdje se spremaju još dvije informacije o uništenim neprijateljskim jedinicama (varijabla *killed unit score*) i građevinama (varijabla *killed building score*) da bi kasnije mogli provjeriti je li se taj broj povećao, da možemo u koraku učenja agentu pridružiti odgovarajuću nagradu.

```
if self.previous_action is not None:
            reward = 0

            if killed_unit_score > self.killed_unit_score_past:
                reward += Reward_Unit_Destroyed

            if killed_building_score > self.killed_building_score_past:
                reward += Reward_Building_Destroyed

            self.qLearn.Learn(str(self.previous_state), self.previous_action, reward
                , str(current_state))
            self.qLearn.qTable.to_pickle(DATA_FILE + '.gz', 'gzip')
            self.qLearn.qTable.to_csv(DATA_FILE + '.csv')
```

Ovaj dio odnosi se na agentovu sposobnost učenja. Da bi mogli točno znati u koje stanje je igra prešla nakon izvršene akcije moramo prvo preći u njega, iz tog razloga ovaj agent uči na početku svog sljedećeg koraka, ono što se dogodilo u prošlom koraku. Prvo trenutnu nagradu postavljamo na nulu i provjeravamo je li agent u ovom koraku uništio neprijateljsku zgradu ili jedinicu, te u slučaju da je, pridružujemo preddefiniranu nagradu za te akcije. Nakon toga pozivamo *Learn* funkciju kojoj proslijeđujemo prošlo stanje, akciju, nagradu i trenutno stanje, ista četiri parametra kao i u primjeru sa labirintom samo za prošli korak. Nakon učenja se Q tablicu zapisuju rezultati učenja, u svrhu kasnijeg ponovnog korištenja.

```
        action_to_choose = self.qLearn.ChooseAction(str(current_state))
        chosen_action = possible_actions[action_to_choose]

        #postavljanje šprolih podataka
        self.killed_unit_score_past  = killed_unit_score
        self.killed_building_score_past = killed_building_score
        self.previous_state = current_state
        self.previous_action = action_to_choose
```

Nakon učenja dolazimo na odabir akcije za koji se koristi ista metoda iz *qLearn* objekta kao i u primjeru s labirintom, te se nakon toga ta ista akcija odabire ali u formatu akcija koji je definiran na početku skripte. Sada slijedi najveći dio koda, koji se odnosi na izvršavanje odabrane akcije u prethodnom koraku.

```
if chosen_action == ACTION_DO_NOTHING:
            return actions.FunctionCall(Action_No_Action, [])

        elif chosen_action == ACTION_SELECT_SCV:
            unit_type = obs.observation['feature_screen'][Unit_Type]
            unit_y, unit_x = (unit_type == Unit_Terran_Scv).nonzero()
```

```
    if unit_y.any():
        i = random.randint(0, len(unit_y) - 1)
        target = [unit_x[i], unit_y[i]]

        return actions.FunctionCall(Action_Select_Point, [Not_Queued, target
            ])

elif chosen_action == ACTION_BUILD_SUPPLY_DEPOT:
    if Action_Build_Supply_Depot in obs.observation['available_actions']:
        unit_type = obs.observation['feature_screen'][Unit_Type]
        unit_y, unit_x = (unit_type == Unit_Terran_Commandcenter).nonzero()

        if unit_y.any():
            target = self.transformLocation(int(unit_x.mean()), 0, int(
                unit_y.mean()), 20)

            return actions.FunctionCall(Action_Build_Supply_Depot, [
                Not_Queued, target])

elif chosen_action == ACTION_BUILD_BARRACKS:
    if Action_Build_Barracks in obs.observation['available_actions']:
        unit_type = obs.observation['feature_screen'][Unit_Type]
        unit_y, unit_x = (unit_type == Unit_Terran_Commandcenter).nonzero()

        if unit_y.any():
            target = self.transformLocation(int(unit_x.mean()), 20, int(
                unit_y.mean()), 0)

            return actions.FunctionCall(Action_Build_Barracks, [Not_Queued,
                target])

elif chosen_action == ACTION_SELECT_BARRACKS:
    unit_type = obs.observation['feature_screen'][Unit_Type]
    unit_y, unit_x = (unit_type == Unit_Terran_Barracks).nonzero()

    if unit_y.any():
        target = [int(unit_x.mean()), int(unit_y.mean())]

        return actions.FunctionCall(Action_Select_Point, [Not_Queued, target
            ])

elif chosen_action == ACTION_BUILD_MARINE:
    if Action_Train_Marine in obs.observation['available_actions']:
        return actions.FunctionCall(Action_Train_Marine, [Queued])

elif chosen_action == ACTION_SELECT_ARMY:
    if Action_Select_Army in obs.observation['available_actions']:
        return actions.FunctionCall(Action_Select_Army, [Not_Queued])

elif chosen_action == ACTION_ATTACK:
    if Action_Attack_Minimap in obs.observation["available_actions"]:
        if self.base_top_left:
```

```
                return actions.FunctionCall(Action_Attack_Minimap, [Not_Queued,
                    [39, 45]])

            return actions.FunctionCall(Action_Attack_Minimap, [Not_Queued, [21,
                24]])

    return actions.FunctionCall(Action_No_Action, [])
```

Svaki od *if* odnosno *elif* programskih konstrukata u ovom dijelu podudara se sa odabirom jedne akcije te izvršavanjem iste. Kada se otkrije koja je odabrana akcija, na početku svakog dijela prvo se provjerava je li ta akcija trenutno dozvoljena, izostanak ove provjere dovodi do kritične pogreške za vrijeme igre, odnosno cijeli sustav se sruši u slučaju da akcija nije trenutno dozvoljena iz bilo kojeg razloga. Iako je akcija puno, ima samo par vrsta koje sve koriste isti slijed. Sve akcije mogu se podijliti u dvije skupine, one koje mogu čekati u redu (eng. *queued*) i one koje ne mogu (eng. *not queued*), svaka od ove dvije skupine imaju svoj kod koji treba proslijediti prilikom poziva funkcije, te smo iz tog razloga to ranije definirali kao konstantu.

Sve akcije koje grade kao referentnu točku gradnje uzimaju komandni centar (eng. *Command Center*) i grade na određene koordinate oko te točke koristeći funkciju *transformLocation*, u samom pozivu ovih funkcija treba proslijediti ime funkcije koje smo definirali na početku (npr. ACTION BUILD BARRACKS), konstantu *NOT QUEUED* jer su sve akcije gradnji*not queued* i mjesto na kojem će se izgraditi koje se nalazi u varijabli *target*.

Akcija odabira jedinice pod nazivom SCV koja služi za izgradnju građevina i crpljenje sirovina, nasumično odabire koordinate jednog od svih SCV jedinica, posprema koordinate u varijablu target, te na toj lokaciji poziva funkciju *ACTION SELECT POINT* koja efektivno simulira lijevi klik na određenom mjestu. Tek kada je selektirana jedna SCV jedinica dozvoljene su akcije gradnje, iz tog razloga je bitna provjera dozvoljenih akcija na početku svakog odabira akcije. Na gotovo identičan način funkcionira akcija selektiranja bilo koje građevine ili jedinice, te se jednako tako ovisno o selekciji mijenjaju dozvoljene akcije, na primjer da bi mogli stvarati vojnike marince (eng. *marines*) treba biti označena vojarna.

Dok je akcija selektiranje sve vojske vrlo jednostavna (samo jedan poziv funkcije) akcija za napadanje s tom istom vojskom bila bi znatno kompliciranija, ali u ovom jednostavnom primjeru agent je ograničen na samo jednu mapu te su poznate točne koordinate neprijateljske baze što čini i ovu akciju trivijalnom.

Slijedeći agent koristi suprotnu strategiju od ovoga, umjesto malog broja akcija na poznatoj lokaciji, ima proširene mogućnosti koje ćemo vidjeti u sljedećoj sekciji.

### 5.2.2. Agent 2

Ovaj agent proširen je većim brojem akcija koje može izvršiti, većim brojem stanja koje može poprimiti, te većim brojem jedinica i građevina kojima može upravljati.

Uz sve akcije koje može izvršiti agent broj 1, ovaj agent ima još i slijedeće:

```
Action_Build_Factory = actions.FUNCTIONS.Build_Factory_screen.id
Action_Train_Hellion = actions.FUNCTIONS.Train_Hellion_quick.id
```

```
Action_Build_Refinery = actions.FUNCTIONS.Build_Refinery_screen.id
Action_Train_Reaper = actions.FUNCTIONS.Train_Reaper_quick.id
```

To su dvije nove zgrade i dva nova tipa jedinica za borbu.

```
ACTION_BUILD_REFINERY = 'buildrefinery'
ACTION_BUILD_FACTORY = 'buildfactory'
ACTION_TRAIN_HELLION = 'trainhellion'
ACTION_SELECT_FACTORY = 'selectfactory'
ACTION_TRAIN_REAPER = 'trainreaper'
```

Što rezultira svim ovim akcijama za odabir, kako je prije objašnjeno da bi trenirali vojsku treba prvo označiti zgradu u kojoj se ona i trenira (iako je mjesto za trening jedne od novih jedinica u istoj građevini gdje se treniraju marinci)

```
possible_actions = [
    ACTION_DO_NOTHING,
    ACTION_SELECT_SCV,
    ACTION_BUILD_SUPPLY_DEPOT,
    ACTION_BUILD_BARRACKS,
    ACTION_SELECT_BARRACKS,
    ACTION_BUILD_MARINE,
    ACTION_BUILD_FACTORY,
    ACTION_TRAIN_HELLION,
    ACTION_SELECT_ARMY,
    ACTION_BUILD_REFINERY,
    ACTION_TRAIN_REAPER
]

for mm_x in range(0, 64):
    for mm_y in range(0, 64):
        if (mm_x + 1) % 16 == 0 and (mm_y + 1) % 16 == 0:
            possible_actions.append(ACTION_ATTACK + '_' + str(mm_x - 8) + '_' + str(
                mm_y - 8))
```

Ovo je popis svih starih akcija i dodanih novih akcija. Najbitnija razlika je što se ovdje u *for* petlji dodaju akcije koje napadaju određene koordinate. Zbog sposobnosti vojske da napadaju bilo što, što se nalazi u određenoj blizini jedinica nije potrebno imati odvojenu akciju za napadanje svake točke na mapi, nego možemo podijeliti mapu u zone koje su iste veličine kao i zona u kojoj vojska napada neprijatelje oko sebe.

```
Player_Hostile = 4
Unit_Terran_Factory = 27
Unit_Terran_Refinery = 20
Unit_Vespene_Geyser = 342
Unit_Vespene_Geyser_Rich = 344
```

Da bi mogli imati interakciju sa novim elementima potrebno ih je definirati putem njihovih kodova da bi im mogli pristupati. *Factory* i *Refinery* su dvije nove građevine koje agent ima mogućnost izgraditi, dok je *Vespene* nova sirovina potrebna za izgradnju novih zgrada i jedinica.

```
def TransformDistance(self, x, x_distance, y, y_distance):
        if not self.base_top_left:
```

```
            return [x - x_distance, y - y_distance]
        return [x + x_distance, y + y_distance]

    def transformLocation(self, x, y):
        if not self.base_top_left:
            return [64 - x, 64 - y]

        return [x, y]
```

Dodana je nova funkcija *transformLocation* koja se koristi isključivo za izglađivanje koordinata napada, funkciju koju je imala *transformLocation* zamjenjuje nova metoda *TransformDistance*.

```
depot_y, depot_x = (unit_type == Unit_Terran_Supply_Depot).nonzero()
        if depot_y.any():
            supply_depot_count = 1
        else:
            supply_depot_count = 0
        barracks_y, barracks_x = (unit_type == Unit_Terran_Barracks).nonzero()
        if barracks_y.any():
            barracks_count = 1
        else:
            barracks_count = 0

        factory_y, factory_x = (unit_type == Unit_Terran_Factory).nonzero()
        if factory_y.any():
            factory_count = 1
        else:
            factory_count = 0
```

U dijelu prepoznavanja građevina dodan je još i dio za prepozavanje postoji li tvornica (eng. *Factory*), jer će se i ona kasnije dodati u trenutno stanje.

```
 current_state = np.zeros(20)
        current_state[0] = supply_depot_count
        current_state[1] = barracks_count
        current_state[2] = supply_limit
        current_state[3] = army_supply
        current_state[4] = factory_count




        hot_squares = np.zeros(16)
        enemy_y, enemy_x = (obs.observation['feature_minimap'][Player_Relative] ==
            Player_Hostile).nonzero()
        for i in range(0, len(enemy_y)):
            y = int(math.ceil((enemy_y[i] + 1) / 16))
            x = int(math.ceil((enemy_x[i] + 1) / 16))

            hot_squares[((y - 1) * 4) + (x - 1)] = 1

        if not self.base_top_left:
            hot_squares = hot_squares[::-1]
```

```
        for i in range(0, 16):
            current_state[i + 4] = hot_squares[i]
```

U dijelu koji definira trenutno stanje ima značajnih promjena u odnosu na prošlog agenta. Osim što se dodaje broj tvornica u trenutno stanje, ostatak koda služi za dodavanje neprijateljskih pozicija u trenutno stanje u nadi da će se agent time poslužiti da nauči braniti se i napadati neprijateljske lokacije.

```
action_to_choose = self.qLearn.ChooseAction(str(current_state))
chosen_action = possible_actions[action_to_choose]
x = 0
y = 0
if '_' in chosen_action:
    chosen_action, x, y = chosen_action.split('_')
```

Agent broj 2 nakon odabira akcije pomoću *if* - a određuje je li odabrana akcija jedna od onih za napad ili nije, u slučaju da je pamti njene koordinate u varijable *x* i *y*.

```
    elif chosen_action == ACTION_BUILD_REFINERY:
            if Action_Build_Refinery in obs.observation['available_actions']:


                unit_type = obs.observation['feature_screen'][Unit_Type]
                vespene_y, vespene_x = (unit_type == Unit_Vespene_Geyser).nonzero()


                i = random.randint(0,len(vespene_y)-1)
                x = vespene_x[i]
                y = vespene_y[i]
                target = [x,y]
                return actions.FunctionCall(Action_Build_Refinery, [Not_Queued,
                    target])


    elif chosen_action == ACTION_BUILD_FACTORY:
            if Action_Build_Factory in obs.observation['available_actions']:
                unit_type = obs.observation['feature_screen'][Unit_Type]
                unit_y, unit_x = (unit_type == Unit_Terran_Commandcenter).nonzero()

                if unit_y.any():
                    target = self.TransformDistance(int(unit_x.mean()), 10, int(
                        unit_y.mean()), 30)

                    return actions.FunctionCall(Action_Build_Factory, [Not_Queued,
                        target])

    elif chosen_action == ACTION_SELECT_FACTORY:
            unit_type = obs.observation['feature_screen'][Unit_Type]
            unit_y, unit_x = (unit_type == Unit_Terran_Factory).nonzero()

            if unit_y.any():
                target = [int(unit_x.mean()), int(unit_y.mean())]
```

```
            return actions.FunctionCall(Action_Select_Point, [Not_Queued, target
                ])

    elif chosen_action == ACTION_TRAIN_HELLION:
            if Action_Train_Hellion in obs.observation['available_actions']:
                return actions.FunctionCall(Action_Train_Hellion, [Queued])

    elif chosen_action == ACTION_TRAIN_REAPER:
            if Action_Train_Reaper in obs.observation['available_actions']:
                return actions.FunctionCall(Action_Train_Reaper, [Queued])


    elif chosen_action == ACTION_ATTACK:
            do_it = True

            if len(obs.observation['single_select']) > 0 and obs.observation['
                single_select'][0][0] == Unit_Terran_Scv:
                do_it = False

            if len(obs.observation['multi_select']) > 0 and obs.observation['
                multi_select'][0][0] == Unit_Terran_Scv:
                do_it = False

            if do_it and Action_Attack_Minimap in obs.observation["available_actions
                "]:

                return actions.FunctionCall(Action_Attack_Minimap, [Not_Queued, self
                    .transformLocation(int(x), int(y))])

        return actions.FunctionCall(Action_No_Action, [])
```

Sve navedene nove akcije funkcioniraju na istom principu kao i one koje su već bile dio agenta 1, one koje grade i označavaju samo biraju nove koordinate i koriste drugu varijablu za definiranje elementa koji se gradi ili označava. Najveća promjena je u akciji koje nije nova, a to je *ACTION ATTACK*. Sam krajnji poziv je ostao identičan, ali prije toga je dodano par provjera, zato što je agent ponekad prilikom slanja vojske na novi način u većini slučajeva slao i SCV jedinice koje nisu sposobne za borbu, te je taj dio trebalo onemogućiti, što je i učinjeno provjerama je li selektirana SCV jedinica i onemogućavanjem napada ako je.

U sljedećem poglavlju objašnjeno je kako su navedene promjene utjecale na efikasnost ovog agenta u savladavanju igre. Prije toga u sljedećoj sekciji prikazan je zadnji agent koji pomoću Q-učenja igra Starcraft 2.

### 5.2.3. Agent 3

Ovaj agent je optimizirana verzija kombinacije prošla dva agenta. Nije ograničen samo na jednu mapu, ima veći broj akcija od prvog agenta, ali manji od drugog, uz još nekoliko dodataka koji će biti prikazani kroz programski kod agenta.

```
Action_No_Action = actions.FUNCTIONS.no_op.id
Action_Select_Point = actions.FUNCTIONS.select_point.id
```

```
Action_Build_Supply_Depot = actions.FUNCTIONS.Build_SupplyDepot_screen.id
Action_Build_Barracks = actions.FUNCTIONS.Build_Barracks_screen.id
Action_Train_Marine = actions.FUNCTIONS.Train_Marine_quick.id
Action_Select_Army = actions.FUNCTIONS.select_army.id
Action_Attack_Minimap = actions.FUNCTIONS.Attack_minimap.id
Action_Harvest_Gather = actions.FUNCTIONS.Harvest_Gather_screen.id

#akcije za odabir
ACTION_DO_NOTHING = 'donothing'
ACTION_BUILD_SUPPLY_DEPOT = 'buildsupplydepot'
ACTION_BUILD_BARRACKS = 'buildbarracks'
ACTION_BUILD_MARINE = 'buildmarine'
ACTION_ATTACK = 'attack'
```

Možemo primjetiti znantu razliku između broja definiranih akcija koje će se pozivati i onih koje agent ima na odabir, zato što agent broj 3 ima implementiran odabir i izvršavanje akcija na drugačiji način od prošla dva agenta.

```
def __init__(self):
    self.cc_y = None
    self.cc_x = None

    self.move_number = 0
```

Uz sve varijable iz prijašnjih agenata, ovaj agent ima tri nove varijable, *cc y , cc x* koje služe za pamćenje x i y koordinata komandnog centra i *move number* koja će se koristiti za novu metodu izvršavanja akcija.

```
 def splitAction(self, action_id):
        possible_action = possible_actions[action_id]

        x = 0
        y = 0
        if '_' in possible_action:
            possible_action, x, y = possible_action.split('_')

        return (possible_action, x, y)
```

Kod definiranu u ovoj funkciji s imenom *splitAction* je već postojao u agentu broj 2, samo je ovdje izdvojen da bi se mogao lakše koristiti na više mjesta.

```
def step(self,obs):
        super(QLearningAgent, self).step(obs)

        if obs.last():
            reward = obs.reward

            self.qLearn.Learn(str(self.previous_state), self.previous_action, reward
                , 'terminal')

            self.qLearn.qTable.to_pickle(DATA_FILE + '.gz', 'gzip')
            self.qLearn.qTable.to_csv(DATA_FILE + '.csv')
```

```
        self.previous_action = None
        self.previous_state = None

        self.move_number = 0

        return actions.FunctionCall(Action_No_Action, [])

    unit_type = obs.observation['feature_screen'][Unit_Type]

    if obs.first():
        player_y, player_x = (obs.observation['feature_minimap'][Player_Relative
            ] == Player_Self).nonzero()
        self.base_top_left = 1 if player_y.any() and player_y.mean() <= 31 else
            0

        self.cc_y, self.cc_x = (unit_type == Unit_Terran_Commandcenter).nonzero
            ()
```

Ovaj agent koristi nove mogućnosti pysc2 okruženja, a to je prepoznavanje je li trenutni korak prvi ili zadnji u epizodi. Ako je zadnji, odnosno ako je igra završila, dodijeljuje se nagrada za završetak igre (ovisno je li pobjeda ili poraz) te se poziva *Learn* funkcija za Q-učenje, rezultati učenja spremaju se u sekundarnu memoriju, te se radi sigurnosti određene varijable postavljaju na inicijalne vrijednosti da bi bile spremne za korištenje u sljedećoj epizodi. U slučaju da je prvi korak izvode se operacije prepoznavanja lokacije igrača i komandnog centra, zato što ih je potrebno samo jednom izvršiti, a koje su se u ostalim agentima izvodile prilikom svakog poziva *step* funkcije.

Da bi ubrzali agentovo učenje, svaka akcija koju agent može izabrati podjeljena je u najviše tri zasebne akcije od kojih se svaka izvodi u jednom koraku, te se učenje odvija nakon sva tri koraka. Pomoću varijable koja je prethodno definirana *move number* provjeravamo u kojem od tri koraka unutar akcije se agent trenutno nalazi, te se izvršava onaj dio akcije koji je potreban za taj korak. Na primjer ako agent izabere akciju izgradnje vojarne u prvom koraku će se izvršiti akcija označavanja SCV jedinice koja je potrebna za izgrdanju, u drugom koraku će se SCV poslati da izgradi vojarnu, a u trećem koraku će se SCV vratiti nazad na svoj početni položaj koji je crpljenje minerala. Ovako to izgleda u kodu:

```
    if self.move_number == 0:
        self.move_number += 1

        current_state = np.zeros(12)
        current_state[0] = cc_count
        current_state[1] = supply_depot_count
        current_state[2] = barracks_count
        current_state[3] = obs.observation['player'][Army_Supply]

        hot_squares = np.zeros(4)
        enemy_y, enemy_x = (obs.observation['feature_minimap'][Player_Relative]
            == Player_Hostile).nonzero()
        for i in range(0, len(enemy_y)):
            y = int(math.ceil((enemy_y[i] + 1) / 32))
```

```
        x = int(math.ceil((enemy_x[i] + 1) / 32))

        hot_squares[((y - 1) * 2) + (x - 1)] = 1

    if not self.base_top_left:
        hot_squares = hot_squares[::-1]

    for i in range(0, 4):
        current_state[i + 4] = hot_squares[i]

    green_squares = np.zeros(4)
    friendly_y, friendly_x = (obs.observation['feature_minimap'][
        Player_Relative] == Player_Self).nonzero()
    for i in range(0, len(friendly_y)):
        y = int(math.ceil((friendly_y[i] + 1) / 32))
        x = int(math.ceil((friendly_x[i] + 1) / 32))

        green_squares[((y - 1) * 2) + (x - 1)] = 1

    if not self.base_top_left:
        green_squares = green_squares[::-1]

    for i in range(0, 4):
        current_state[i + 8] = green_squares[i]

    if self.previous_action is not None:
        self.qLearn.Learn(str(self.previous_state), self.previous_action, 0,
             str(current_state))

    excluded_actions = []
    if supply_depot_count == 2 or worker_supply == 0:
        excluded_actions.append(1)

    if supply_depot_count == 0 or barracks_count == 2 or worker_supply == 0:
        excluded_actions.append(2)

    if supply_free == 0 or barracks_count == 0:
        excluded_actions.append(3)

    if army_supply == 0:
        excluded_actions.append(4)
        excluded_actions.append(5)
        excluded_actions.append(6)
        excluded_actions.append(7)

    rl_action = self.qLearn.ChooseAction(str(current_state),
        excluded_actions)

    self.previous_state = current_state
    self.previous_action = rl_action

    chosen_action, x, y = self.splitAction(self.previous_action)
```

Kao i u prošlom primjeru dodajemo neprijateljske pozicije u trenutno stanje kao polje pod nazivom *hot squares* , ali ovdje dodajmo još i *green squares* koje označavaju pozicije agentove vojske, te se nakon određivanja trenutnog stanja kao i ranije poziva metoda za Q-učenje. Nakon učenja pojavljuje se još jedna optimizacija specifična za ovog agenta, a to je popis nedozvoljenih akcija u varijabli *excluded actions*. Prošla dva agenta prilikom odabira mogli su izabrati bilo koju akciju neovisno o tome je li ju moguće izvršiti u trenutku odabira i tek se nakon odabira provjeravala mogućnost izvršenja akcije, što je bilo neefikasno i dovodilo do velikog broja koraka u kojima su se akcije preskakale jer je agent odabrao neku koju nije moguće izvršiti. Ovdje se putem nekoliko *if* provjera određene akcije dodaju u varijablu *excluded actions* te se te akcije proslijeđuju funkciji za odabir i privremeno se brišu iz popisa akcija između kojih agent može izabrati.

```
if chosen_action == ACTION_BUILD_BARRACKS or chosen_action ==
    ACTION_BUILD_SUPPLY_DEPOT:
    unit_y, unit_x = (unit_type == Unit_Terran_Scv).nonzero()

    if unit_y.any():
        i = random.randint(0, len(unit_y) - 1)
        target = [unit_x[i], unit_y[i]]

        return actions.FunctionCall(Action_Select_Point, [Not_Queued, target])

elif chosen_action == ACTION_BUILD_MARINE:
    if barracks_y.any():
        i = random.randint(0, len(barracks_y) - 1)
        target = [barracks_x[i], barracks_y[i]]

        return actions.FunctionCall(Action_Select_Point, [Action_Select_All, target
            ])

elif chosen_action == ACTION_ATTACK:
    if Action_Select_Army in obs.observation['available_actions']:
        return actions.FunctionCall(Action_Select_Army, [Not_Queued])
```

U prvom koraku sve moguće akcije svode se na izvršavanje samo ove tri navedene akcije, označavanje određenih građevina za izgradnju ili trening, i označavanje vojske u slučaju za napad.

```
elif self.move_number == 1:
            self.move_number += 1

            chosen_action, x, y = self.splitAction(self.previous_action)

            if chosen_action == ACTION_BUILD_SUPPLY_DEPOT:
                ...

                return actions.FunctionCall(Action_Build_Supply_Depot, [Not_Queued,
                    target])

            elif chosen_action == ACTION_BUILD_BARRACKS:
                ...
```

```
                return actions.FunctionCall(Action_Build_Barracks, [Not_Queued,
                    target])

            elif chosen_action == ACTION_BUILD_MARINE:
                if Action_Train_Marine in obs.observation['available_actions']:
                    return actions.FunctionCall(Action_Train_Marine, [Queued])

            elif chosen_action == ACTION_ATTACK:
               ...
                    return actions.FunctionCall(Action_Attack_Minimap, [Not_Queued,
                        self.TransformLocation(int(x) + (x_offset * 8), int(y) + (
                        y_offset * 8))])
```

U drugom koraku odabrane akcije izvršavaju se na sličan način kao i u prošlim agentima, te je većini akcija to pravi zadnji korak, iako će ipak doći do sljedećeg koraka.

```
elif self.move_number == 2:
            self.move_number = 0

            chosen_action, x, y = self.splitAction(self.previous_action)

            if chosen_action == Action_Build_Barracks or chosen_action ==
                Action_Build_Supply_Depot:
                if Action_Harvest_Gather in obs.observation['available_actions']:
                    unit_y, unit_x = (unit_type == Unit_Neutral_Mineral_Field).
                        nonzero()

                    if unit_y.any():
                        i = random.randint(0, len(unit_y) - 1)

                        m_x = unit_x[i]
                        m_y = unit_y[i]

                        target = [int(m_x), int(m_y)]

                        return actions.FunctionCall(Action_Harvest_Gather, [Queued,
                            target])

            return actions.FunctionCall(Action_No_Action, [])
```

Jedine akcije koje stvarno koriste sva tri koraka su one koje zahtjevaju gradnju neke građevine gdje se u trećem koraku vraća na početni položaj SCV jedinica koja je izgradila građevinu, ali zbog jednostavnosti algoritma učenja sve akcije promatramo kao da imaju tri koraka, ako nemaju treću korak pozvat će se akcija *Action no Action* koja će samo prebaciti stanje u sljedeći korak bez izvršavanja akcije.

```
class QLearningTable:
    def ChooseAction(self, observation, excluded_actions=[]):
        self.disallowed_actions[observation] = excluded_actions
        state_action = self.qTable.loc[observation, :]
        for excluded_action in excluded_actions:
            del state_action[excluded_action]
    def Learn(self, state, action, reward, stateNext):
```

```
if state == stateNext:
        return
```

Agent 3 također ima promjenjenu klasu *QLearningTable* sa samo par dodanih linija koda. Prva izmijenjena funkcije je *ChooseAction* kojoj je dodan još jedan parametar putem kojeg se proslijeđuju nedozvoljene akcije, te se iste brišu u *for* petlji, da ih agent ne bi mogao izabrati. Druga promjenjena funkcija je *Learn* u kojoj je dodana jedan *if* blok u kojem se provjerava je li stanje promijenjeno, ako nije ne izvršava se dio funkcije koji se odnosi na učenje. Ova promjena temelji se na činjenici da agent u malom vremenskom razdoblju izvršava veliku količinu akcija koje ne utječu na stanje igre što nepotrebno komplicira proces učenja.

# 6. Ponašanje agenta

Svaki od agenata naučio je rješavati problem na svoj način, u nastavku će biti objašnjeno ponašanje koje rezultira kodom koji je bio prikazan u prošlom poglavlju.

## 6.1. Agent 1

Prvi agent vrlo je jednostavan, ali je u konačnici uspio izvršiti zadatak koji mu je bio dodijeljen, a to je pobijediti neprijatelja. Na početku svog treniranja agent šalje jednog po jednog vojnika na neprijatelja, dok kasnije nauči generirati najveći dozvoljeni broj jedinica te ih tek onda poslati u napad, i odmah generirati nove kada je to moguće. Kako je jedina nagrada koju ovaj agent dobija za uništavanje protivničkih jedinica, te nema negativne nagrade za gubitak vlastitih jedinica, u jednoj od svojih iteracija agent je naučio maksimizirati svoju nagradu tako da konstantno šalje svoje jedinice da uništavaju protivničke, ali na takav način da ne pobijedi igru nego da konstatno dobija nagradu za uništavanje, što na kraju dovodi do toga da agent izgubi igru. Nakon kreiranja nove Q tablice koja diktira politiku agenta, agent je naučio igrati igru na očekivani način, gdje jednostavno pobjedi kada je u mogućnosti.

## 6.2. Agent 2

Tijekom razvoja agenta 2 bilo je više različitih načina na koje je agent učio. Ideja je bila poboljšati funkcionalnosti agenta 1, prvo način na koji uči, a kasnije dodavanjem više mogućnosti na izbor agentu.

Promjena načina učenja bazirala se na tome da osim samo nagrađivanja agenta za uništavanje neprijateljskih jedinica i građevina, agent dobije nagradu i za generiranje vlastitih jedinica i građevina. Ovaj pristup nije se pokazao uspješan, jer je agent vrlo rano naučio dobivati nagradu za treniranje (generiranje) vojske i konstantno je gradio vojsku bez napadanja. Sljedeće poboljšanje pokušalo je sustav nagrađivanja u potpunosti promijeniti, umjesto nagrađivanja agenta za određene akcije, korišten je sustav nagrađivanja koji je već definiran unutar same igrice. Ovo također nije bio uspješan pristup zato što su sada nagrade ovisile o previše različitih fakotra o kojima agent nije imao saznanja, te bi mu trebao iznimno velik broj epizoda da nauči igrati, što nije uspjelo.

Dodavanjem više akcija pokušala se povećati agentova šansa za pobjedu kroz mogućnost treniriranja jačih jedinica za borbu. Da bi agent to postigao također je bilo potrebno dodati akcije koje omogućavaju crpljenje nove vrste sirovine koja se koristi za izgradnju i trenining novih jedinica. Ovaj pristup se također nije pokazao dobar, agentu je trebalo znatno više vremena da nauči izgrađivati dodatne jedinice i koristiti ih u borbi. Veći broj akcija bez sustava koji sprječava odabir akcija koje nisu dozvoljene dovodi do agenta koji veliki dio vremena provodi ne radeći ništa, jer je jako velika šansa da je akcija koju je agent odabrao nedozvoljena u trenutnom stanju. U konačnici je agent uspio pobijediti protivnika čak i sa svim ovim otežavajućim faktorima, ali pobjeđuje u manjoj mjeri od agenta 1, od kojeg je trebao biti znatno efektivniji,

jedina prednost ovog agenta je ta da agent 2 ima mogućnost naučiti napadati neprijateljske jedinici po koordinatama što omogućava prilagodbu na bilo koju mapu.

## 6.3. Agent 3

Agent 3 koristi razne metode optimizacije da bi bio najefektivniji od spomenutih agenata koji igraju Starcraft 2. Prva metoda je suportna od one koja je korištena kod agenta 2, umjesto dodavanja akcija za odabir, agent 3 ima smanjeni broj akcije između kojih može birati. Druga metoda je korištenje dvije nove metode za provjerju je li trenutni korak prvi ili zadnji u epizodi, što smanjuje količinu koda koja se mora izvoditi u svakom koraku. Treća metoda je dodavanje provjere nedozvoljenih akcija prije nego što agent izabire akciju, te onemogućavanje izbora nedozvoljenih akcija. Ova metoda osigurava da agent ne gubi vrijeme pokušavajući izvesti akcije koje ne može izvesti zbog stanja igre. Četvrta metoda je podijela jedne akcije koju agent može izabrati u više akcija koje će se izvršiti u uzastopnim koracima, što zajedno sa brisanjem nedozvoljenih akcija osigurava da će se svaka akcija koju agent izabere moći izvršiti. Ovaj agent također dobiva pozitivnu nagradu za pobjedu i negativnu nagradu za gubitak igre, što rješava problem u kojem agent nauči samo ubijati protivničke jedinice bez pobjede.

Sve navedene metode dovode do agenta koji može naučiti biti iznimno efektivan, pobijediti u puno manjem broju epizoda od ostala dva agenta, te pobjeđivati u većemo postotku epizoda koje odigra. Agent 3 nema najvećih nedostataka prva dva agenta, ne gubi vrijeme na nemoguće akcije, može napadati neovisno o mapi, a nema veliki broj akcija koje ograničavaju učenje. Nakon treninga agentov slijed akcije je izgrađivanje građevina potrebnih za generiranje vojske, generiranje same vojske i slanje iste u istraživanje po mapi da pronađe neprijatelja, dok istovremeno u bazi gradi zgrade koje povećavaju maksimalnu količinu vojske koju agent može imati, ili ako je neka od zgrada uništena u neprijateljskom napadu ponovo je izgrađuje. Možda i najbitnija činjenica je da je agent sposoban naučiti napadati neprijateljske pozicije koje vidi na mapi te braniti svoje jedinice i građevine u slučaju neprijateljskog napada.

Iako sva tri agenta mogu pobijediti protivnika, agent 3 čini to s najmanjim vremenom treninga i u najvećem postotku.

# 7. Zaključak

Kroz primjer četiri veoma različita agenta koji koriste Q-učenje prikazali smo kako Q-učenje rješava generalni problem svog pojačanog učenja, koji je kako naučiti agenta da poduzima akcije koje dovode do maksimalne moguće nagrade. Q-učenje to rješava na način da u svakom koraku unutar okruženja izračunava Q vrijednost koja predstavlja kvalitetu tog koraka, što je veća vrijednost to je očekivana nagrada koju taj korak donosi veća. Isprobavajući razne akcije, agent od okruženja dobiva povratne informacije u obliku nagrade ( bila ona pozitivina ili negativna) te po tim informacijama izračunava Q vrijednosti. Birajući akcije koje dovode u stanje sa najvećom Q vrijednosti agent indirektno uči koja je optimalna politika biranja akcija. Naravno, agent ne može samo birati akcije koje donose najveću nagradu zato što bi time konstantno ponavljao jedan te isti niz akcija, iz tog razloga potreban je pseudoslučajan izbor akcija barem u nekoj mjeri.

Očit je i nedostatak ove metode, kako se povećava broj stanja koje okruženje može poprimiti i broj akcija koje agent može izvršiti agent sporije uči i teže se snalazi u okruženju. U slučaju s labirintom gdje se okruženje u niti jednom trenutku ne mijenja agent u manje od 20 epizoda nalazi optimalan put do pobjede, a prvi put pobjeđuje već u prvih par epizoda. Agenti koji igraju Starcraft 2 ne dolaze do prve pobjede u manje od sto epizoda ( i to samo agent 1 koji je najjednostavniji), a da postignu optimalno rješenje trebala bi tolika količina epizoda da to nije izvedivo u sklopu ovakvog rada. Vidljivo je iz primjera agenta 2 da omogućavanjem korištenja više različitih mogućnosti (čak i ako su bolje od osnovnih) ne garantira agenta koji će biti sposobniji u igranju igre, u ovom primjeru dogodilo se upravo suprotna situacija gdje je agent s najvećim mogućnostima i najlošiji u smislu pobjede nad neprijateljem. Ovaj zaključak podupire primjer agenta broj 3, koji ima najmanje akcija na odabir od sva tri agenta, a pokazuje najveću uspješnost u igri.

Kao metoda pojačanog učenja, Q-učenje pokazala se iznimno korisna te su u svim primjerima agenti naučili kako doći do pobjede u određenoj igri, bez ikakvih početnih saznanja o okruženju u kojem se nalaze i isključivo kroz vlastito istraživanje tog okruženja, što je i cilj metode Q-učenja.

# Popis literature


[1] *BeginnersGuide/Overview - Python Wiki*. adresa: https://wiki.python.org/moin/BeginnersGuide/Overview (pogledano 4. 9. 2020).

[2] O. Vinyals, T. Ewalds, S. Bartunov, P. Georgiev, A. S. Vezhnevets, M. Yeo, A. Makhzani, H. Küttler, J. Agapiou, J. Schrittwieser i dr., „Starcraft ii: A new challenge for reinforcement learning", *arXiv preprint arXiv:1708.04782*, 2017.

[3] D. Lynton Poole, *Computational Intelligence: A Logical Approach*, ISBN: 978-0-19-510270-3.

[4] R. E. Neapolitan i X. Jiang, *Artificial intelligence: With an introduction to machine learning*. CRC Press, 2018.

[5] *BBC The Next Big Thing - Artificial Intelligence*, rujan 2009. adresa: https://web.archive.org/web/20090925043908/http://www.open2.net/nextbigthing/ai/ai_in_depth/in_depth.htm (pogledano 17. 9. 2020).

[6] H. Hodson, „DeepMind and Google: the battle to control artificial intelligence", *The Economist*, ISSN: 0013-0613. adresa: https://www.economist.com/1843/2019/03/01/deepmind-and-google-the-battle-to-control-artificial-intelligence (pogledano 28. 8. 2020).

[7] K. P. Murphy, *Machine learning: a probabilistic perspective*. MIT press, 2012.

[8] S. Russell i P. Norvig, „Artificial intelligence: a modern approach", 2002.

[9] R. S. Sutton i A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[10] Q. Hu i W. Yue, *Markov decision processes with their applications*. Springer Science & Business Media, 2007, sv. 14.

[11] C. Shyalika, *A Beginners Guide to Q-Learning*, en, studeni 2019. adresa: https://towardsdatascience.com/a-beginners-guide-to-q-learning-c3e2a30a653c (pogledano 7. 9. 2020).

[12] C. J. Watkins i P. Dayan, „Q-learning", *Machine learning*, sv. 8, br. 3-4, str. 279–292, 1992.

[13] Morvan, *Q-learning - (Reinforcement Learning)*, en. adresa: https://mofanpy.com/tutorials/machine-learning/reinforcement-learning/tabular-q1/ (pogledano 8. 9. 2020).

[14] *#4 Q Learning Reinforcement Learning (Eng python tutorial) - YouTube*. adresa: https://www.youtube.com/watch?v=qPE4CPQY7mc&list=PLXO45tsB95cIplu-fLMpUEEZTwrDNh6Ba&index=4 (pogledano 8. 9. 2020).

[15] *StarCraft II: Wings of Liberty*, en, Page Version ID: 976499352, rujan 2020. adresa: https://en.wikipedia.org/w/index.php?title=StarCraft_II:_Wings_of_Liberty&oldid=976499352 (pogledano 6. 9. 2020).

[16] S. Brown, *Building a Smart PySC2 Agent*, en, kolovoz 2018. adresa: https://chatbotslife.com/building-a-smart-pysc2-agent-cdc269cb095d (pogledano 19. 8. 2020).

[17] ——, *Add Smart Attacking to Your PySC2 Agent*, en, kolovoz 2018. adresa: https://itnext.io/add-smart-attacking-to-your-pysc2-agent-17fd5caad578 (pogledano 9. 9. 2020).

[18] ——, *Build a Sparse Reward PySC2 Agent*, en, kolovoz 2018. adresa: https://itnext.io/build-a-sparse-reward-pysc2-agent-a44e94ba5255 (pogledano 9. 9. 2020).

[19] ——, *How to Locate and Select Units in PySC2*, en, kolovoz 2018. adresa: https://itnext.io/how-to-locate-and-select-units-in-pysc2-2bb1c81f2ad3 (pogledano 9. 9. 2020).

[20] ——, *Refine Your Sparse PySC2 Agent*, en, kolovoz 2018. adresa: https://itnext.io/refine-your-sparse-pysc2-agent-a3feb189bc68 (pogledano 9. 9. 2020).

[21] *deepmind/pysc2*, en. adresa: https://github.com/deepmind/pysc2 (pogledano 9. 9. 2020).

[22] *SC2 Unit IDs*, Paste Site. adresa: https://pastebin.com/KCwwLiQ1 (pogledano 9. 9. 2020).

# Popis slika