Факултет инжењерских наука Универзитета у Крагујевцу

Никола Н. Килибарда

Реализација модела за играње игре Таблић помоћу подржаног учења

дипломски рад

Крагујевац, 2020.

Факултет инжењерских наука Универзитета у Крагујевцу

Назив студијског програма: Рачунарска техника и софтверско инжењерство
Ниво студија: Основне академске студије
Предмет: Оперативни системи
Број индекса: 557/2016

Никола Н. Килибарда

Реализација модела за играње игре Таблић помоћу подржаног учења

дипломски рад

Комисија за преглед и одбрану:
1. **доц. др Владимир М. Миловановић - ментор**

2._____ Датум одбране:_____
3._____ Оцена:_____

Садржај

Резиме

Циљ овог дипломског рада јесте да се направи рачунарски програм који приближно на нивоу човека може играти популарну карташку игру "Таблић", а заснован је на принципима машинског учења и вештачке интелигенције. Најпре ће бити написан детерминистички програм за играње Таблића заснован на похлепном алгоритму. Након тога ће бити описан модел заснован на такозваном подржаном учењу. Подржано учење или учење с подршком је уз надгледано и ненадгледано учење једна од три основних парадигми машинског учења. У подржаном учењу испитивани систем, односно агент, бива обучаван кроз узајамно дејство са окружењем. Типичан сценарио је да такозвани агенти вуку одређене потезе унутар окружења. Сви добри потези агента бивају награђени и на тај начин стимулисани и у будућности. Окружење је обично моделовано кроз такозвани Марковљев процес одлучивања. Биће испитан и образложен утицај различитих параметара на квалитет добијеног модела. Добијени модели биће упоређени међусобно, али и са похлепним моделом и човеком.

Кључне речи: Таблић, машинско учење, похлепни алгоритам, подржано учење.

Abstract

The purpose of this thesis is to make a computer program that can play the popular card game "Tablić" on a human level, and is based on the principles of machine learning and artificial intelligence. First, a deterministic program for playing Tablić, based on a greedy algorithm will be written. After that a model based on reinforcement learning will be made. Reinforcement learning, alongside supervised and unsupervised learning is one of the three basic paradigms of machine learning. In reinforcement learning, the examined system, i.e. agent, is trained through interacting with the environment. In a typical scenario, the so-called agent, makes certain moves within the environment. All of the agent's good moves are rewarded and thus stimulated in the future. The environment is usually modeled through the so-called Markov decision-making process. The influence of different parameters on the quality of the obtained model will be examined and explained. The obtained models will be compared with each others, but also with a greedy model and a human player.

Keywords: Tablić, machine learning, greedy algorithm, reinforcement learning.

1 Увод

Учење интеракцијом са окружењем је један од првих видова учења које човек присваја и које користи током целог свог живота. Иако нема експлицитног учитеља, човек има директну сензомоторну везу са окружењем. Помоћу ове везе човек може закључити мноштво информација о узроку и последицама његових радњи као и шта треба учинити да би постигао своје циљеве. Током човековог живота такве интеракције су несумњиво главни извор знања о његовом окружењу а и њему самом. Без обзира да ли човек учи да вози бициклу, да свира клавир или како да води разговор, он је веома свестан његовог окружења и како оно реагује на његово понашање, и својим понашањем се труди да утиче на оно што се дешава.

Када човек учи да игра неку игру, он прво научи правила а после тога почне да игра ту игру. Сваком одиграном партијом, он добија искуство и полако унапређује своје вештине и највеће знање управо стекне играњем, покушавањем разних потеза и посматрањем како они утичу на исход игре.

Игре представљају један од веома популарних домена за истраживање у области вештачке интелигенције. Крајем 2017. године, компанија *DeepMind* је направила програм под називом *AlphaZero* који је успео да игра на надљудском нивоу, три различите игре: Го, Шах и Шоги. За ово је коришћен алгоритам за подржано учење за који се осим правила игре није користило никакво друго знање о игри коју је учио. Целокупно знање које је *AlphaZero* стекао је било кроз играње игре против самог себе.

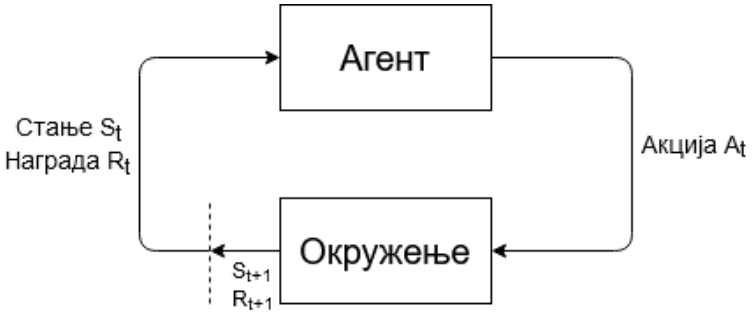
Подржано учење или учење поткрепљивањем представља једну од три основних парадигми машинског учења, која је базирана на учењу интеракцијом са окружењем. У овом раду ће се принципи подржаног учења применити на игру Таблић и реализовати модел који ће научити да игра ову игру играјући против самог себе.

2 Проблем подржаног учења

У овом поглављу ће детаљно бити описани проблеми који се решавају подржаним учењем. Биће постављена формулација ових проблема помоћу Марковљевог процеса одлучивања и на крају ће бити описане методе за решавање ових проблема.

2.1 Агент и окружење

Проблем подржаног учења представља проблем код кога се учење извршава кроз интеракцију. Ученик и доносилац одлука назива се *агент*. Све оно са чиме он интерагује, назива се *окружење*. Они непрекидно интерагују, агент бира и извршава акције на основу тренутног стања окружења, а окружење одговара на те акције. Окружење такође даје награде које агент покушава да максимизује временом.



Слика 1: Интеракција између агента и окружења

Прецизније, агент и окружење узајамно делују једни на друге у сваком дискретном временском кораку, $t = 0, 1, 2, 3, ...$ У сваком временском кораку t, агент добија неку представу стања окружења, $S_t \in \mathcal{S}$, где је $\mathcal{S}$ скуп свих могућих стања, и на основу тога бира акцију, $A_t \in \mathcal{A}(S_t)$, где је $\mathcal{A}(S_t)$ скуп акција које су доступне у стању S_t. Један временски корак касније, као последица агентове акције, он добија нумеричку награду, $R_{t+1} \in \mathcal{R} \subseteq \mathbb{R}$, и налази се у новом стању, S_{t+1}.

2.2 Циљеви и награде

У подржаном учењу, сврха или циљ агента је формализована помоћу специјалног сигнала награде који он добија од окружења. У сваком временском кораку, награда је нека нумеричка вредност, $R_t \in \mathbb{R}$. Циљ агента представља максимизацију награде коју он добија. Ово не значи максимизацију непосредне награде, већ максимизацију укупне дугорочне награде.

Иако формулисање циљева у виду сигнала награде може да се чини ограничавајућим, у пракси се показало као веома флексибилно и широко применљиво. Агент увек учи како да максимизује своју награду. Самим тим како би агент радио оно што се од њега тражи, награде морају бити дефинисане на такав

начин да максимизацијом њих агент такође остварује и задати циљ. Због тога је веома битно да су награде добро дефинисане и да указују на оно што треба бити остварено.

Сигнал награде не треба да пружа било какву информацију о томе *како* нешто треба да се уради, већ само информацију о томе *шта* треба да се уради.

На пример, агент који учи да игра шах, треба бити награђен само онда када победи партију, а не и када оствари неке подциљеве као што су узимање противничких фигура. Уколико би остваривање ових подциљева било награђивано, агент би можда нашао начин да их оствари без да оствари стварни циљ. На пример, агент би можда нашао начин да узме противникове фигуре чак и по цену изгубљене партије.

2.3 Добитак

Већ је речено да циљ агента представља максимизација дугорочне награде. Потребно је формално дефинисати ову дугорочну награду, односно оно што агент треба да максимизује. Агент треба да максимизује *очекивани добитак* који ће бити означен са G_t, и који је дефинисан као функција награда остварених после тренутка t. У најједноставнијем случају ова функција може бити представљена као:

$$G_t = R_{t+1} + R_{t+2} + \cdots + R_T = \sum_{k=t+1}^{T} R_k$$

где T представља временски корак последњег тзв. *завршног стања*. Завршно стање представља крај интеракције између агента и окружења, у примеру игара означава стање после последњег потеза. Период од почетног до завршног стања назива се *епизода*. Овакав приступ дефинисања функције добитка није применљив у ситуацијама када епизоде нису коначне, тј. када је $T = \infty$. У том случају добитак би веома лако могао да тежи бесконачности. Због тога се уводи концепт умањења и функција умањеног добитка је представљена као:

$$G_t = R_{t+1} + \gamma * R_{t+2} + \gamma^2 * R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+1+k}$$

где γ представља параметар који се назива *фактор умањења* и за који важи $0 \leq \gamma \leq 1$.

Фактор умањења одређује колико је тренутно вредна награда из будућности. У кораку t награда коју ће агент добити у кораку $t + k$ вреди γ^{k-1} пута мање него награда која се добија одмах. У случају када су награде ограничене и ако је $\gamma < 1$, ова вредност конвергира и у случају бесконачних епизода. Уколико важи $\gamma = 0$, агент би максимизовао само тренутну награду, односно агент би био похлепан.

2.4 Марковљев процес одлучивања

Марковљев процес одлучивања представља формализацију проблема који се решавају подржаним учењем. Окружење у овим проблемима је углавном представљено као Марковљев процес одлучивања.

Дефиниција 1 Марковљев процес одлучивања *је уређена шесторка* $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \tau, p, \gamma)$ *где је* $\mathcal{S}$ скуп стања, $\mathcal{A}$ *је* скуп акција, $\mathcal{R} \subseteq \mathbb{R}$ *је* скуп награда, τ *је* трајекторија, *односно низ случајних променљивих*

$$S_0, A_0, R_0, S_1, A_1, R_1, S_2, A_2, R_2, \ldots$$

таквих да важи $S_t \in \mathcal{S}, A_t \in \mathcal{A}$ *и* $R_t \in \mathcal{R}$ *за свако* $t \geq 0, p : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ *је* функција преласка *таква да за свако* $t \geq 0$ *важи*

$$p(s', r|s, a) = P(S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a)$$

и γ *је реалан број из интервала* $[0, 1]$ *који се назива* фактором умањена.

2.4.1 Марковљево својство

У Марковљевом процесу одлучивања динамичност система треба да буде потпуно дефинисана помоћу фунцкије преласка p

$$p(s', r|s, a) = P(S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a)$$

Ово својство се зове *Марковљево својство*. Марковљево својство говори да вероватноћа преласка из једног стања у друго стање зависи само од тренутног стања и акције. Односно, историја стања која је довела до тренутног стања нема утицаја на понашање окружења у тренутном стању. Математички то може да се напише и на други начин. Да би Марковљево својство било задовољено треба да важи:

$$P(S_t, R_t|S_0, A_0, R_0, ..., S_{t-1}, A_{t-1}) = P(S_t, R_t|S_{t-1}, A_{t-1})$$

Ово не значи да Марковљевим процесом одлучивања не могу да се формализују окружења која зависе од претходних стања, већ само значи да стање треба да буде дефинисано тако да садржи све потребе информације за даље понашање окружења.

На пример, у шаху, ако би стање било представљено само позицијом фигура на табли, таква дефиниција окружења не би задовољала Марковљево својство, с обзиром да без знања о претходним стањима не би могло да се одреди који играч је тренутно на потезу. Самим тим како би игра шаха задовољавала Марковљево својство, потребно је да тренутно стање садржи информацију о играчу који је тренутно на потезу као и информацију о дозвољеним рокадама играча.

2.4.2 Политика

Епизоде представљају реализацију трајекторија, односно низове реализација случајних променљивих које их чине. Генерисање једне епизоде јесте последица интеракције агента са окружењем. Начин генерисања епизоде зависи једним делом од понашања окружења, односно функције преласка која дефинише окружење, а другим делом зависи од понашања агента. Понашање агента може се формализовати *политиком* $\pi(a|s)$, расподелом вероватноће над акцијама за дато стање, односно сматра се да ће агент у стању s предузети акцију a са вероватноћом $\pi(a|s)$. Уколико је вероватноћа једне од акција једнака 1, док су вероватноће осталих једнаке 0, та политика је детерминистичка.

2.4.3 Функције вредности

Под решавањем Марковљевог процеса одлучивања подразумева се проналажење политике која води максималном очекиваном добитку агента.

Очекивани добитак који агент добија пратећи политику π почевши из неког стања s:

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s]$$

назива се *функција вредности стања* где $\mathbb{E}_\pi$ представља очекивану вредност случајне променљиве уколико агент прати политику π. У завршним стањима, вредност ове функције је увек 0. Поред функције вредности стања, битно је дефинисати и *функцију вредности акције у стању*

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a]$$

Ове функције се могу лако повезати:

$$\begin{aligned} v_\pi(s) &= \mathbb{E}[G_t|S_t = s] \\ &= \sum_a \pi(a|s)\mathbb{E}[G_t|S_t = s, A_t = a] \\ &= \sum_a \pi(a|s)q_\pi(s, a) \end{aligned}$$

Приметимо интуитивност ове везе. Очекивани добитак у стању s је просек очекиваних добитака када се у стању s предузме акција a, при чему се сваки од тих добитака при упросечавању отежава вероватноћом акције a чији је резултат. Могуће је изразити и функцију q преко функције v:

$$\begin{aligned} q_\pi(s, a) &= \mathbb{E}[G_t|S_t = s, A_t = a] \\ &= \mathbb{E}[R_t + \gamma G_{t+1}|S_t = s, A_t = a] \\ &= \sum_{s',r} p(s', r|s, a)[r + \gamma\mathbb{E}[G_{t+1}|S_{t+1} = s']] \\ &= \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')] \end{aligned}$$

Приметимо да је лакше из функције q добити функцију v, него обрнуто. Наиме, први смер подразумева да је позната политика π што је у пракси типично случај, пошто се политика користи за управљање агентом. Други смер подразумева да је позната функција преласка, односно да је познато функционисање окружења, што често није тачно јер је окружење осим у ретким случајевима тешко прецизно описати.

2.4.4 Оптимална политика

Функције вредности омогућавају мерење квалитета политика. Наиме, уколико за сва стања $s \in \mathcal{S}$ важи

$$v_{\pi 1}(s) \geq v_{\pi 2}(s)$$

јасно је да је прва политика боља од друге или јој је бар једнака. Тиме је дефинисан парцијални поредак над политикама $\pi_1 \geq \pi_2$. За сваки Марковљев процес одлучивања постоји бар једна *оптимална политика*, политика која је боља или једнака од свих осталих политика. Иако је могуће да постоји више оптималних политика, све оптималне политике се означавају са π_*. Све оптималне политике деле једну функцију вредности стања која се назива *оптимална функција вредности стања*, она се означава са v_* и дефинисана је као:

$$v_*(s) = \max_\pi v_\pi(s)$$

за све $s \in \mathcal{S}$.

Оптималне политике такође деле и *оптималну функцију вредности акције у стању*, која се означава са q_* и дефинисана је као:

$$q_*(s, a) = \max_\pi q_\pi(s, a)$$

за све $s \in \mathcal{S}$ и $a \in \mathcal{A}$.

Познавање функције q_* омогућава израчунавање једне оптималне политике:

$$\pi_*(a|s) = \begin{cases} 1, & \text{ако важи } a = \arg\max_a q_*(s, a) \\ 0, & \text{иначе} \end{cases}$$

У сваком стању, ова оптимална политика је детерминистичка и доноси акцију која даје највећи добитак. Односно оптимална политика доноси акцију за коју је вредност $q_*(s, a)$ највећа. Коришћењем ове политике можемо извести и везу између оптималне функције стања и оптималне функције акције у стању:

$$\begin{aligned} v_*(s) &= \sum_a \pi_*(a|s) q_*(s, a) \\ &= q_*(s, \arg\max_a q_*(s, a)) \\ &= \max_a q_*(s, a) \end{aligned}$$

2.4.5 Белманове једнакости

Пожељно је имати начин израчунавања функције q_*. Један начин да се то постигне пружају такозване *Белманове једнакости* које изражавају рекурентне везе функција вредности. Конкретно, за функцију v_* важи:

$$\begin{aligned} v_*(s) &= \max_a q_*(s,a) \\ &= \max_a \mathbb{E}_{\pi_*}[G_t|S_t = s, A_t = a] \\ &= \max_a \mathbb{E}_{\pi_*}[R_t + \gamma * G_{t+1}|S_t = s, A_t = a] \\ &= \max_a \sum_{s',r} p(s',r|s,a)[r + \gamma \mathbb{E}_{\pi_*}[G_{t+1}|S_{t+1} = s']] \\ &= \max_a \sum_{s',r} p(s',r|s,a)[r + \gamma v_*(s')] \end{aligned}$$

Белманова једнакост за функцију q је:

$$q_*(s,a) = \sum_{s',r} p(s',r|s,a)[r + \gamma \max_{a'} q_*(s',a')]$$

Добитак при преузимању акције a у стању s зависи од непосредно добијене награде и највећег добитка који се може добити из стања у коме се агент нађе након предузимања акције. Притом, како су различити преласци у наредна стања различито вероватни, потребно је непосредно добијене награде и будуће добитке отежати вероватноћама тих прелаза. Ове једначине омогућавају израчунавање функција вредности једноставним итеративним поступцима

$$v_{i+1}(s) \leftarrow \max_a \sum_{s',r} p(s',r|s,a)[r + \gamma v_i(s')]$$

за свако s и

$$q_{i+1}(s,a) \leftarrow \sum_{s',r} p(s',r|s,a)[r + \gamma \max_{a'} q_i(s',a')]$$

за свако s и a. Ови поступци конвергирају за произвољне иницијалне вредности v_o и q_o у случају коначних скупова $\mathcal{S}$ и $\mathcal{A}$. Међутим, ако су ови скупови велике кардиналности, ова конвергенција ће бити превише спора за практичну употребу. Ако су бесконачни, онда овај алгоритам није применљив.

2.5 Алгоритам *q-учења*

Претходно описани поступак решавања Марковљевог процеса одлучивања је применљив само када је познат начин функционисања окружења. С обзиром да је то ретко случај, мора се наћи други начин решавања овог проблема.

Један од начина решавања овог проблеме јесте да се функција вредности акције у стању q_* научи. Један од алгоритама за учење ове функције је алгоритам *q-учења* који је описан у алгоритму 1.

Алгоритам 1: Алгоритам *q-учења*

иницијализуј $Q(s,a), \forall s \in \mathcal{S}, \forall a \in \mathcal{A},$ произвољно;
иницијализуј $Q(\text{завршно стање}, \cdot) = 0$;
за *сваку епизоду* **уради**
  иницијализуј S;
  понављај
    изабери A из S користећи ε-похлепну политику добијену из Q;
    изврши акцију A и опази R, S';
    $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma \max_a Q(S',a) - Q(S,A)]$;
    $S \leftarrow S'$;
  докле год важи S *није завршно*;
крај петље

У првом кораку произвољно се иницијализују вредности функције Q за сва стања и акције. Такође се иницијализује функција Q тако да у завршним стањима она има вредност 0. Након иницијализације агент креће да интерагује са околином кроз епизоде. На почетку сваке епизоде иницијализује се почетно стање S.

Агент користећи неку политику интерагује са околином, извршавајући акције, док не дође у завршно стање када се епизода завршава. Политика која се најчешће користи при *q-учењу* је тзв. ε-похлепна политика и она је дефинисана као:

$$\pi_{q,\varepsilon}(a|s) = \begin{cases} 1 - \varepsilon + \frac{\varepsilon}{|\mathcal{A}(s)|}, & \text{ако важи } a = \arg\max_a q(s,a) \\ \frac{\varepsilon}{|\mathcal{A}(s)|}, & \text{иначе} \end{cases}$$

Односно, ε-похлепна политика је политика која са вероватноћом ε бира насумичну акцију, а са вероватноћом $1-\varepsilon$ бира ону акцију за коју мисли да доноси највећу вредност у задатом стању. Ова политика се користи како би агент истраживао и долазио у различита стања уместо да константно понавља исте акције. Истраживањем агент стиче веће знање о окружењу и добија прилику да открије боље акције од оних које је научио. Истраживање је веома битно зато што неке акције постају боље тек након што агент стекне одређено знање. На пример, у шаху, исти потез велемајстору може донети победу, а почетнику може донети пораз. Углавном се узима да вредност ε у почетку учења буде већа како би агент у почетку више истраживао. Кроз епизоде ова вредност се полако умањује и углавном се та вредност ограничава како никад не би достигла вредност 0, како агент никада не би престао да истражује.

Након што агент изабере и изврши акцију, он опази добијену награду и следеће стање и на основу њих врши ажурирање фунцкије вредности акције у стању за претходно извршену акцију. Ажурирање се врши помоћу формуле:

$$q(s,a) \leftarrow (1-\alpha)q(s,a) + \alpha[R + \gamma \max_a q(s',a)]$$

Овом формула се ажурира функција вредности акције у стању, за истражено стање упросечавањем вредности функције q за тренутно стање и извршену акцију са вредношћу добитка, односно збира тренутне награде и функције вредности стања у стању које је уследило.

Параметар α представња стопу учења, односно то је вредност која означава колико се вреднује старо знање, а колико ново. Удео старог знања, $q(s,a)$, у ажурирању вредности је једнак $(1-\alpha)$, док је удео знања стеченог из тренутне епизоде, $R+\gamma \max_a q(s',a)$, једнак α. Као и код вредности ε, углавном се узима да вредност α у почетку учења буде већа и да се полако смањује кроз епизоде. Ово омогућава агенту да у почетку учи доста брзо, а да касније полако прерађује своје знање. Ова вредност никад не достиже 0, јер у том случају не би уопште било учења.

Овај алгоритам има пар недостака. Први недостатак јесте тај да овај алгоритам није применљив на проблеме са бесконачним бројем могућих стања и акција или чак проблеме са великим бројем стања и акција због немогућности чувања података о сваком стању и акцији. Други недостатак представља то што да би агент знао како треба да се понаша у неком стању, он мора да је претходно био у таквом стању, самим тим, за неке проблеме потребно би било доста времена док се не истраже сва могућа стања.

Оба ова недостатка могу да се реше применом машинског учења. Односно, вредност функције $q(s,a)$ може се предвиђати коришћењем неуронске мреже. Пошто улази у неуронску мрежу могу бити континуални ово решава недостатак у проблемима са бесконачним бројем стања. Такође велика предност коришћења неуронске мреже јесте та што у већини случајева, слична стања и акције ће имати сличне вредности. Ово омогућава да мрежа стварно научи вредност акција у стањима чак и у ситуацијама у којима се пре није нашла, што значајно убрзава учење.

Битно је приметити да се алгоритам не мора много мењати уколико се користи неуронска мрежа. Формулу за ажурирање можемо написати на други начин:

$$q(s,a) \leftarrow q(s,a) + \alpha[R + \gamma \max_a q(s',a) - q(s,a)]$$

У овако написаној формули грешка која се прави се налази у делу који множи α, и самим тим параматар α има исту функцију као и стопа учења која се користи при тренирању неуронске мреже. Самим тим веома је лако променити алгоритам тако да се за предвиђање вредности функције стања и акције користи неуронска мрежа, која се тренира да минимизује грешку. Алгоритам *q-учења* који користи принципе машинског учења и неуронских мрежа назива се **дубоко *q-учење***.

Алгоритам дубоког *q-учења* може се унапредити уз прављење пар модификација.

Први начин унапређивања алгоритма представља имплементирање груписања или тзв. *Batching*-а. Коришћењем груписања тренирање се не извршава после сваке акције, већ се акције као и резултати акција чувају и групишу како би се у неком тренутку извршило ажурирање за све од њих. Ово може значајно убрзати време тренирања јер се тренирање мреже, односно алгоритам пропагације уназад може паралелизовати. Уз коришћење графичких картица ово убрзање се мери у хиљадама што омогућава решавање проблеме доста брже, односно омогућава решавање комплекснијих проблема који се иначе не би могли решити у разумном времену.

Други начин унапређивања представља тренирање са понављањем искуства или тзв. *Experience Replay*. Применом *Experience Replay*-а, у меморији се чувају стања и извршене акције, као и добијене награде и информација о стању које је уследило. Током процеса тренирања мреже из меморије се насумичним узроковањем узимају подаци о претходним стањима и извршеним акцијама и извршава се ажурирања над њима. Коришћење ове методе смањује шансе да ће се мрежа преобучити на новије епизоде, односно спречава агента да заборави вредности функције q у стањима у којима се дуго није нашао. Такође омогућава агенту да исправи грешку у својој процени тако што ће је поново сагледати касније са већим стеченим искуством.

И последње унапређење представља коришћење циљне мреже или тзв. *Target Network*. Пошто се за рачунање добитка који је нека акција донела користи процена неуронске мреже, односно за процену и тренирање неуронске мреже се користи сама та неуронска мрежа, може доћи до нестабилности при тренирању. Тачније, после сваког ажурирања мрежа ће бити промењена што ће утицати на сва следеће ажурирања. Како би се ово избегло, користиће се две мреже, једна тренутна и једна циљна мрежа. Тренутна мрежа представља мрежу која се тренира и која одлучује о акцијама, док циљна мрежа представља старију итерацију тренутне мреже и она се користи за израчунавање добитка у новом стању односно за рачунање циљне вредности која се предвиђа. После одређеног броја епизода, циљна мрежа се ажурира да буде једнака тренутној.

3 Окружење за игру Таблић

У овом поглављу је описано и имплементирано окружење за игру Таблић. Иако Таблић може да се игра са више играча, најчешће се игра са два играча, и описана правила као и имплементација су за игру са два играча.

3.1 Правила игре Таблић

3.1.1 Карте

Таблић се игра са стандардним шпилом од 52 карте без џокера. Краљ (K) има вредност 14, Дама (Q) 13, а Жандар (J) има вредност 12. Кец (A) може да има вредност 1 или 11, у зависности од процене играча. Све остале карте имају вредност која се на њима налази. Знак карте нема никакву улогу осим у случајевима "мале двојке" (♣2) и "велике десетке" (♦10), где се прави разлика у бодовању.

3.1.2 Дељење

Делилац меша карте и поставља четири карте на талон. Затим дели карте себи и противнику тако да оба играча на почетку имају по 6 карата у руци. Остатак шпила се привремено оставља са стране. Након што су оба играча одиграла све своје карте, делилац поново дели по 6 карата, и игра се наставља док се не поделе и одиграју све карте. Након одигране партије делилац се мења.

3.1.3 Игра

Играч који није делилац игра први, и онда се играчи смењују након сваке одигране карте.

Играч који је на потезу може да одигра једну карту из руке. Ако је вредност одигране карте једнака другој карти или збиру скупа карата на талону, играч може да однесе све такве карте или скупове. Свака карта може да припада само једном однесеном скупу.

На пример, ако на талону имамо карте 2, 3, 4, и 7, са картом 9 можемо однети скуп $(2 + 3 + 4)$ или скуп $(2 + 7)$ али не и оба ова скупа, јер карта 2 може да припада само једном од њих. Однесене карте, заједно са одиграном картом, играч ставља на своју гомилу.

Уколико вредност одигране карте није једнака вредности ниједне друге карте или скупу карата на талону, та карта остаје на талону, и у следећим потезима може бити однета.

Пример: Карте на талону су: A, 3, 6, 7, 8, Q.

- Уколико играч одигра 6 може да однесе само 6
- Уколико играч одигра 9 може да однесе $(A + 8) + (3 + 6)$
- Уколико играч одигра 10 може да однесе $7 + 3$

- Уколико играч одигра краља (14) може да однесе $A+6+7$, или $(A+Q)+(6+8)$, или $(A+3)+(6+8)$
- Уколико играч одигра 5 не може да однесе ниједну карту и та карта остаје на талону

Када играч одигра карту, није у обавези да однесе све што може. Он може да однесе само неке карте или скупове, или само да остави своју карту на талону.

Када се игра заврши, ако има карата које су остале на талону носи их онај играч који је последњи нешто носио. Када играч однесе све карте са талона својом картом, пише му се табла. Када играч на крају покупи карте које су остале на талону, због тога што је он последњи носио, он не добија таблу.

3.1.4 Бодовање

На крају партије, сваки играч се бодује у зависности од карата које је однео током игре на следећи начин:

- За сваки однети штих (10, J, Q, K, A) добија се по један бод.
- За "малу двојку" (♣2) добија се по један бод.
- За "велику десетку" (♦10) добија се по два бода.
- Играч који је однео највише карата у партији добија 3 бода, а уколико су оба играча однела исти број карата, онда нико не добија додатна 3 бода.
- За сваку "таблу" коју је играч однео добија се по један бод.

Пошто знак карте није битан, осим у случају мале двојке и велике десетке, у имплементацији је изостављено правило везано за њих како би свака од карата могла бити представљења помоћу само њене вредности. Односно у имплементацији мала двојка и велика десетка носе исти број бодова као и остале карте њихових вредности.

3.2 Имплементација игре Таблић

Игра Таблића имплементирана је у *Python* програмском језику унутар класе *Tablic*. Ова класа води рачуна о току игре, чувању стања игре као и одигравању потеза. Позивом конструктора се покреће нова партија Таблића. Конструктор има пар параметара. То су:

- *deck* - параметар којим се бира шпил са којим ће се партија одиграти. Уколико није специфирано користи се насумично промешан стандардан шпил од 52 карте.
- *hand_size* - параметар који означава број карата у руци сваког од играча након дељења. Подразумевана вредност је 6.
- *initial_table_size* - параметар који означава број карата на талону на почетку партије. Подразумевана вредност је 4.

3.2.1 Тренутно стање игре

Класа *Tablic* има неколико својстава и метода која служе да дају информацију о тренутном стању игре. То су:

- *current_player* - својство које означава који играч је тренутно на потезу
- *last_to_take* - својство које означава који играч је последњи носио
- *is_terminal* - својство које означава да ли је игра завршена
- *move_counter* - својство које означава број тренутног потеза
- *table* - својство које враћа низ карата које се тренутно налазе на талону
- *rewards* - својство које враћа број поена које су играчи освојили
- *get_taken(player)* - метода која враћа низ однетих карата за одређеног играча
- *get_hand(player)* - метода која враћа низ карата које се налазе у руци одређеног играча

3.2.2 Вектори стања и акција

Класа *Tablic* такође даје могућност приказивања тренутног стања у виду опсервационог вектора помоћу методе *get_observation_vector* чија је имплементација дата у листнгу 1.

```
def get_observation_vector(self, player):
    return np.concatenate((self._observation_hands[player],
                           self._observation_table,
                           self._observation_taken[player],
                           self._observation_taken[1-player],
                           [self._last_to_take, player]))
```

Листинг 1: Имплементација методе *get_observation_vector*.

Вектор опсервације представља други начин на који се тренутно стање које играч посматра може представити. Карте у играчевој руци, карте на талону и однесене карте за сваког од играча представљене су помоћу низова. Такође могу и да се представе помоћу вектора. Вектор има 13 чланова (за сваку од вредности карата), где сваки члан означава број одређене карте која се појављује у датом низу. Пар примера низова и вектора који њих описују су дати:

$$[A, 2, 2, 7, J, K] \rightarrow [1, 2, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1]$$
$$[5, 5, 5, 6, 6] \rightarrow [0, 0, 0, 0, 3, 2, 0, 0, 0, 0, 0, 0, 0]$$
$$[4, 10, Q, Q, Q, K] \rightarrow [0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 2, 1]$$

Опсервациони вектор тренутног стања се састоји од вектора руке играча, вектора талона, вектора однетих карата играча, вектора однетих карата противника,

и додатно садржи информацију о последњем носиоцу као и о тренутном играчу. Дужина вектора опсервације износи 54.

Осим ове методе класа такође садржи и методу *get_take_vector* која дату акцију претвара у векторски формат. Ова метода враћа вектор дужине 26, где првих 13 чланова означавају карте које играч жели да однесе а других 13 означавају карту коју играч жели да одигра. Имплементација ове методе је дата у листингу 2.

```
@classmethod
def get_take_vector(cls, played_card, take):
    take_vector = np.zeros(26)
    take_vector[Tablic.card_to_index(played_card) + 13] = 1
    for card in take:
        take_vector[Tablic.card_to_index(card)] += 1
    return take_vector
```

Листинг 2: Имплементација методе *get_take_vector*.

3.2.3 Дозвољени потези

Класа *Tablic* обезбеђује и класне методе *is_valid_take* и *get_valid_takes*. Метода *is_valid_take* проверава да ли је одређени потез валидан, док метода *get_valid_takes* враћа све валидне потезе за дату руку и талон. Потези су представљени паром вредности, где први члан представља одиграну карту док други члан представља низ карата које играч жели да однесе.

```
@classmethod
def get_valid_takes(cls, table, hand):
    all_takes = chain.from_iterable(set(combinations(table, r))
                                    for r in range(0, len(table)+1))
    for take in all_takes:
        for played_card in hand:
            if Tablic.is_valid_take(played_card, take):
                yield (played_card, take)
```

Листинг 3: Имплементација методе *get_valid_takes*.

Имплементација методе *get_valid_takes* је дата у листингу 3. Метода пролази кроз све могуће комбинације потеза, проверава њихову валидност и враћа у виду генератора оне потезе који су валидни. Провера валидности се врши помоћу функције *is_valid_take* чија је имплементација дата у листингу 4.

```
@classmethod
def is_valid_take(cls, played_card, take):
    if len(take) == 0: return True
    if played_card == 1: played_card = 11
    for take in cls._get_all_ace_combinations(take):
        left_to_take = sorted(take, reverse=True)
        while left_to_take:
            subset = cls._find_subset(played_card, left_to_take, [])
            if subset == None:
                break
            for element in subset:
                left_to_take.remove(element)
        if not left_to_take:
            return True
    return False
```

Листинг 4: Имплементација методе *is_valid_take*.

Уколико ниједна карта није однесена, онда је потез сигурно валидан. Затим уколико је одигран кец (A), постављамо вредност на 11, јер уколико је валидно носити са кецом вредности 1, онда је сигурно валидно носити и са кецом вредности 11, али обрнуто није случај. Затим метода пролази кроз све могуће комбинације кечева који су однети, и уколико ниједна од њих није валидна метода враћа негативан резултат. Провера валидности једне од комбинација се врши тако што се прво опадајуће сортира комбинација, а затим траже подскупови чија је вредност једнака вредности одигране карте. Када се нађе један подскуп, његови елементи се избацују из низа тренутне комбинације и овај поступак се понавља све док се низ не испразни што говори да је потез валидан, или док више није могуће наћи подскупове а низ није празан, и у том случају тренутна комбинација кечева није валидна и метода прелази на следећу.

```
@classmethod
def _get_all_ace_combinations(cls, take):
    take = list(take)
    yield take.copy()
    while 1 in take:
        take[take.index(1)] = 11
        yield take.copy()
```

Листинг 5: Имплементација методе *_get_all_ace_combinations*.

Метода *_get_all_ace_combinations* враћа све могуће комбинације кечева, то ради тако што замени једну вредност 1 са вредношћу 11, врати као генератор и затим понавља процес док не остане ниједна вредност 1 у низу. Њена имплементација је дата у листнигу 5.

```
@classmethod
def _find_subset(cls, size, elements, subset):
    if size == 0:
        return subset
    if size < 0 or len(elements) == 0:
        return None
    found_subset = cls._find_subset(size - elements[0], elements[1:],
                                    subset + [elements[0], ])
    if found_subset:
        return found_subset
    return cls._find_subset(size, elements[1:], subset)
```

Листинг 6: Имплементација методе *_find_subset*.

Имплементација методе *_find_subset* дата је у листингу 6. Ова метода налази подскуп величине *size* у низу *elements*. То ради помоћу рекурзивног алгоритма у којем прво провери да ли се први елемент налази у подскупу тако што проба да нађе подскуп величине $size - elements[0]$ у преосталим члановима, и уколико тај подскуп постоји враћа га заједно са првим елементом. У супротном први елемент се не налази у траженом подскупу и подскуп се тражи у преосталим елементима. Овај начин гарантује да ће се подскуп наћи уколико постоји, а ефикаснији је од обичног *brute-force* начина где се испробавају сви могући подскупови.

3.2.4 Играње потеза

```
def play_card(self, card, take):
    assert self.is_valid_take(card, take), "Invalid take"
    assert not self._is_terminal, "Game is over"
    if not take:
        self._remove_from_hand(card, self._current_player)
        self._add_to_table(card)
    else:
        self._remove_from_hand(card, self._current_player)
        self._add_to_taken(card, self._current_player)
        self._remove_from_table(take)
        if len(self._table) == 0:
            self._rewards[self._current_player] += 1
        self._add_to_taken(take, self._current_player)
        self._last_to_take = self._current_player
    self._current_player = 1 - self._current_player
    self._move_counter += 1
    self._update_game_state()
```

Листинг 7: Имплементација методе *play_card*.

Играње потеза се извршава помоћу методе *play_card(card, take)* којој се као аргументи прослеђују одиграна карта и низ карата које се односе у потезу. Имплементација ове метода дата је у листингу 7.

Прво, метода проверава да ли је потез валидан и да ли је партија завршена. Уколико потез није валидан или је партија завршена, метода враћа *AssertionError* са одговарајућом поруком. Затим уколико није дошло до грешке, обрађује се потез. Уколико играч не односи ниједну карту, онда се веома једноставно карта избацује из његове руке и додаје на талон. У случају када играч односи карте, карта се избацује из руке и додаје у играчеву гомилу однесених карата. Карте које се односе, се склањају са талона и такође додају у играчеву гомилу однесених карата, затим се ажурира информација о последњем играчу који је носио. Уколико је талон празан након потеза, играч добија један бод за однесену таблу. Затим, независно од тога да ли је играч носио карте или не, мења се тренутни играч и инкрементује редни број потеза. На крају се позива метода *_update_game_state* која води рачуна о поновном дељењу карата и окончавању игре чија је имплементација дата у листингу 8.

```
def _update_game_state(self):
    if (not self._hands[0] and not self._hands[1]):
        if self._deck_pointer >= len(self._deck):
            self._is_terminal = True
            self._add_to_taken(self._table, self._last_to_take)
            self._remove_from_table(self.table)
        else:
            self._deal_cards()
```

Листинг 8: Имплементација методе *_update_game_state*.

Метода *_update_game_state* води рачуна да играчи не остану без карата у руци. Уколико остану, а шпил није потрошен, дели карте играчима, а у случају када је шпил потрошен, окончава игру и преостале карте са талона ставља на гомилу играча који је последњи носио.

```
def _add_to_taken(self, cards, player):
    if not hasattr(cards, '__iter__'): cards = [cards, ]
    for card in cards:
        self._rewards[player] += self.card_to_tricks(card)
    if len(self._taken[player]) < 27
        and len(self._taken[player]) + len(cards) >= 27:
        self._rewards[player] += 3
    self._add_cards(cards, self._taken[player],
                self._observation_taken[player])
```

Листинг 9: Имплементација методе *_add_to_taken*.

У методи *_add_to_taken* датој у листингу 9 се осим додавања карата у гомилу однесених, такође извршава и додела бодова за однесене карте и додела 3 бода

када играч однесе своју 27. карту у партији.

3.2.5 Остале методе класе *Tablic*

Класа *Tablic* садржи још пар јавних метода које ће укратко бити описане. То су:

- *card_to_tricks(card)* - метода која враћа број бодова које карта доноси, односно 1 ако је карта штих, а 0 ако није
- *get_shuffled_deck()* - метода која враћа насумично промешан шпил карата
- *card_to_index(card)* - метода која враћа индекс који се односи на карту у вектору
- *index_to_card(index)* - метода која враћа карту на коју се односи индекс у вектору

У прилогу у датотеци *tablic.py* се може наћи потпуна имплементација ове класе као и свих њених метода.

3.3 Симулација партија

Класа *TablicArena* се користи за упоређивање два играча. Иницијализација класе се врши помоћу конструктора са два аргумента, који представљају два играча.

За одигравање једне партије користи се метода *start_and_play_game* чија се имплементација налази у листингу 10.

```
def start_and_play_game(self, deck=None, print_plays=False):
    game = Tablic(deck)
    while not game.is_terminal:
        card, take = self.player1.play_turn(game)
        if (print_plays): print(f"Playing {card} and taking {take}.")
        game.play_card(card, take)
        card, take = self.player2.play_turn(game)
        if (print_plays): print(f"Playing {card} and taking {take}.")
        game.play_card(card, take)
    return game.rewards
```

Листинг 10: Имплементација методе *start_and_play_game*.

Прво се иницијализује партија Таблића, а затим се играчи смењују играјући потезе док се партија не заврши. Када партија дође у завршно стање, метода враћа број бодова које су играчи освојили. Аргумент *print_plays* означава да ли ће се одиграни потези исписивати на стандардном излазу.

Пошто је Таблић игра која доста зависи од случајних елемената, начина на који је шпил промешан, једна партија не даје веродостојну слику о перформансама

играча. Метода *simulate_games* омогућава одигравање произвољног броја партија како би се умањнио удео случајности у резултатима. Њена имплементација је дата у листингу 11.

```
def simulate_games(self, num_of_games, seed=None):
    random.seed(seed)
    p1_wins, draws, p2_wins = (0, 0, 0)
    total_results = [0, 0]
    for game in range(1, num_of_games+1):
        deck = Tablic.get_shuffled_deck()
        battle_results = [0, 0]
        results = self.start_and_play_game(deck)
        battle_results[0] += results[0]
        battle_results[1] += results[1]
        self.switch_players()
        results = self.start_and_play_game(deck)
        battle_results[1] += results[0]
        battle_results[0] += results[1]
        self.switch_players()
        total_results[0] += battle_results[0]
        total_results[1] += battle_results[1]
        if (battle_results[0] > battle_results[1]): p1_wins += 1
        if (battle_results[0] == battle_results[1]): draws += 1
        if (battle_results[0] < battle_results[1]): p2_wins += 1
    return (p1_wins, draws, p2_wins, total_results[0], total_results[1])
```

Листинг 11: Имплементација методе *simulate_games*.

Метода игра *num_of_games* партија и враћа информације о броју победа сваког од играга, броју нерешених партија, као и укупном броју бодова који су оба играча освојили. Треба напоменути да се у овој методи под једном партијом подразумевају две стандардне партије Таблића, одигране са истим шпилом, где после прве партије играчи замене места (играч који је играо први сад игра други и обрнуто). С обзиром да је Таблић асиметрична игра, одигравањем исте партије са замењеним шпиловима представља најбољи начин упоређивања два играча, јер се оба играча надмећу у истим условима. Победа се пише оном играчу који је освојио више бодова у једном пару партија. Помоћу *seed* аргумента може да се постави иницијализација генератора случајних бројева тако да више позива ове методе дају исте резултате за исте играче. Ово олакшава упоређивање играча пошто се играчи могу поредити на истом скупу партија, и самим тим ће бити смањен утицај случајних елемената при оцењивању перформанси играча.

У прилогу у датотеци *tablic_arena.py* може се наћи потпуна имплементација класе *TablicArena*.

4 Играчи

У овом делу ће бити описани играчи за играње игре Таблић. Сваки од играча представљен је као класа која наслеђује класу *TablicPlayer*. Потребно је да сваки од играча имплементира методу *find_best_play*, која враћа потез који ће играч одиграти. Као и окружење играчи су имплементирани у *Python* програмском језику.

4.1 Похлепни играч

Једна од стратегија играње игре Таблић представља стратегија базирана на тзв. похлепног алгоритму. Похлепна стратегија представља начин играња у коме играч игра потез који му у том тренутку доноси највише бодова. Ова стратегије је уједно и прва стратегија коју нови играчи усвоје кад играју Таблић. Стратегија размишља краткорочно и самим тим не представља оптималан начин играња.

Похлепни играч имплементиран је помоћу класе *GreedyTablicPlayer*. Метода *find_best_play* враћа потез који играч хоће да одигра и њена имплементација је дата у листнгу 12.

```
@classmethod
def find_best_play(cls, game):
    hand = game.get_hand(game.current_player)
    return max(Tablic.get_valid_takes(game.table, hand),
               key=lambda x: cls.get_take_value(*x))
```

Листинг 12: Имплементација методе *find_best_play*.

Метода функционише тако што прође кроз све могуће потезе и враћа онај чија је вредност максимална. Вредност потеза зависи од функције *get_take_value* чија се имплементација налази у листнгу 13.

```
@classmethod
def get_take_value(cls, played_card, take):
    if len(take) == 0:
        return (0, 0)
    tricks = (Tablic.card_to_tricks(played_card)
              + sum(Tablic.card_to_tricks(card) for card in take))
    return (tricks, len(take) + 1)
```

Листинг 13: Имплементација методе *get_take_value*.

Вредност потеза је представљена помоћу пара вредности где прва вредност означава број штихова које потез односи а друга означава број карата које потез односи. Са оваквим начином рачунања вредности потеза, похлепни играч прво гледа да максимизује број штихова које односи, а затим и број карата.

4.2 Играч базиран на подржаном учењу

Сада ће бити описан играч базиран на подржаном учењу. Имплементација овог играча одрађена је помоћу 3 класе: *DQN*, *DQNAgent*, и *ReinforcedTablicPlayer*. За реализацију овог играча коришћена је *PyTorch* библиотека за машинско учење.

4.2.1 Класа *DQN*

Класа *DQN* представља неуронску мрежу која предвиђа вредност функције акције у стању $q(s, a)$. Имплементација ове класе је у целости дата у листингу 14.

```
class DQN(nn.Module):
    def __init__(self):
        super(DQN, self).__init__()
        input_size = 80
        output_size = 1
        self.neuralnet = nn.Sequential(
            nn.Linear(input_size, input_size*2),
            nn.ReLU(),
            nn.Linear(input_size*2, input_size*2),
            nn.ReLU(),
            nn.Linear(input_size*2, output_size))

    def forward(self, x):
        return self.neuralnet(x)
```

Листинг 14: Имплементација класе *DQN*

У конструктору се конструише неуронска мрежа. Величина улаза је 80, првих 26 улаза описују акцију дату вектором акције, а других 54 улаза описују стање дато опсервационим вектором. Неуронска мрежа се састоји из улазног слоја величине 80, два скривена слоја величине 160 са *ReLU* функцијом активације и излазног слоја величине 1. Помоћу методе *forward*, врши се пропагација унапред кроз мрежу и враћа се предвиђена вредност.

4.2.2 Класа *DQNAgent*

Класа *DQNAgent* је класа која представља агента и води рачуна о тренирању мреже као и о чувању претходних стања у којима се агент нашао.

```
def __init__(self, gamma=0, memory_len=7200,
             batch_size=512, learning_rate=0.00125):
        self.current = DQN().to(device)
        self.target = DQN().to(device)
        self.gamma = gamma
        self.memory = deque(maxlen = memory_len)
        self.batch_size = batch_size
```

```
        self.loss_fn = torch.nn.MSELoss(reduction='sum')
        self.optimizer = torch.optim.Adam(self.current.parameters(),
                                          lr=learning_rate)
```

Листинг 15: Имплементација конструктора класе *DQNAgent*

У конструктору ове класе, постављају се параметри који се користе при тренирању и то су фактор умањења, величина меморије, величина групе, и стопа учења. Такође се иницијализују неуронске мреже, функција губитка, као и оптимизатор. За губитак се користи *MSELoss (Mean-Squared-Error Loss)*, тј. губитак грешке квадрата. Као оптимизатор користи се *Adam (Adaptive Moment Estimation)* оптимизатор кога карактеристише стопа учења која се прилагођава током тренинга.

```
def backward(self):
    batch_len = min(self.batch_size, len(self.memory))
    batch = random.sample(self.memory, batch_len)
    batch_nn_input = torch.zeros([batch_len, 80]).to(device)
    for ind, (state_action, _, _) in enumerate(batch):
        batch_nn_input[ind] = state_action
    q_values = self.current.forward(batch_nn_input)
    with torch.no_grad():
        q_values_target = q_values.clone().detach()
        for ind, (_, reward, valid_actions) in enumerate(batch):
            if valid_actions == None:
                q_values_target[ind] = reward
            else:
                q_values_target[ind] = reward + self.gamma
                            * torch.max(self.target(valid_actions))
    loss = self.loss_fn(q_values, q_values_target)
    loss.backward()
    self.optimizer.step()
    self.optimizer.zero_grad()
```

Листинг 16: Имплементација методе *backward*

У методи *backward* се врши пропагација уназад помоћу које се мрежа ажурира. Прво се из меморија узима група случајним узроковањем и претвара у одговорајући формат. За изабрану групу се врши пропагација унапред, како би се добила предвиђања тренутне неуронске мреже. Након тога се израчунавају циљне вредности коришћењем награде и предвиђања циљне неуронске мреже. После тога се рачуна губитак за целу групу између предвиђене и циљне вредности. Позивом *backward* методе функције губитка као и *step* методе оптимизатора извршава се један корак градијентног спуста и врши се ажурирање мреже.

4.2.3 Класа *ReinforcedTablicPlayer*

Класа *ReinforcedTablicPlayer* представља класу која наслеђује класу *TablicPlayer* и имплементира интерфејс за играње Таблића коришћењем агента.

```
@classmethod
def take_to_state_action(cls, observation_vector, played_card, take):
    take_vector = Tablic.get_take_vector(played_card, take)
    result = np.concatenate((take_vector, observation_vector))
    return torch.from_numpy(result).type(torch.cuda.FloatTensor)
```

Листинг 17: Имплементација методе *take_to_state_action*

Метода *take_to_state_action* враћа тензор стања и акције који се доводи на улаз неуронске мреже. Метода креира вектор стања и акције спајањем вектора акције и опсервационог вектора, а затим тај вектор претвара у тензор. Њена имплементације је дата у листингу 17.

```
@classmethod
def get_valid_state_actions(cls, game):
    hand = game.get_hand(game.current_player)
    observation = game.get_observation_vector(game.current_player)
    valid_takes = list(Tablic.get_valid_takes(game.table, hand))
    valid_state_actions = \
      torch.zeros([len(valid_takes),80]).type(torch.cuda.FloatTensor)
    for ind, (played_card, take) in enumerate(valid_takes):
        valid_state_actions[ind] = \
          cls.take_to_state_action(observation, played_card, take)
    return valid_takes, valid_state_actions
```

Листинг 18: Имплементација методе *get_valid_state_actions*

Метода *get_valid_state_actions* враћа све валидне акције као и векторе стања и акције за сваку од њих и њена имплементација је дата у листингу 18.

```
def find_best_play_from_state_actions(self, takes, state_actions):
    with torch.no_grad():
        takes_value = self.agent.forward(state_actions)
    best_take_ind = torch.argmax(takes_value)
    return takes[best_take_ind]
```

Листинг 19: Имплементација методе *find_best_play_from_state_actions*

Метода *find_best_play_from_state_actions* дата у листингу 19 је метода која враћа најбољи потез користећи тензоре стања и акције. Метода врши пропагацију унапред кроз све тензоре стања и акције, и предвиђа њихове вредности. Затим одређује акцију, тј. потез за који је та вредност највећа, и враћа тај потез.

Остало је још да се имплементира метода *find_best_play* како би потребе базне класе *TablicPlayer* биле задовољене.

```
def find_best_play(self, game):
        return self.find_best_play_from_state_actions(
                        *self.get_valid_state_actions(game))
```

Листинг 20: Имплементација методе *find_best_play*

Након што су имплементиране претходне две методе, тривијално је имплементирани методу *find_best_play* и њена имплементација је дата у листингу 20.

```
def load_model(self, model_path):
    self.agent = torch.load(model_path)

def save_model(self, model_path):
    torch.save(self.agent, model_path)
```

Листинг 21: Имплементација метода *save_model* и *load_model*

Ова класа такође омогућава чување и учитавање агената помоћу метода *save_model* и *load_model* чије су имплементације дате у листингу 21. Ово омогућава лако чување тренираних модела и њихово касније учитавање и коришћење.

```
def get_random_play(self, game):
        all_valid_takes = Tablic.get_all_valid_takes(game.table,
                            game.get_hand(game.current_player))
        return random.choice(list(all_valid_takes))
```

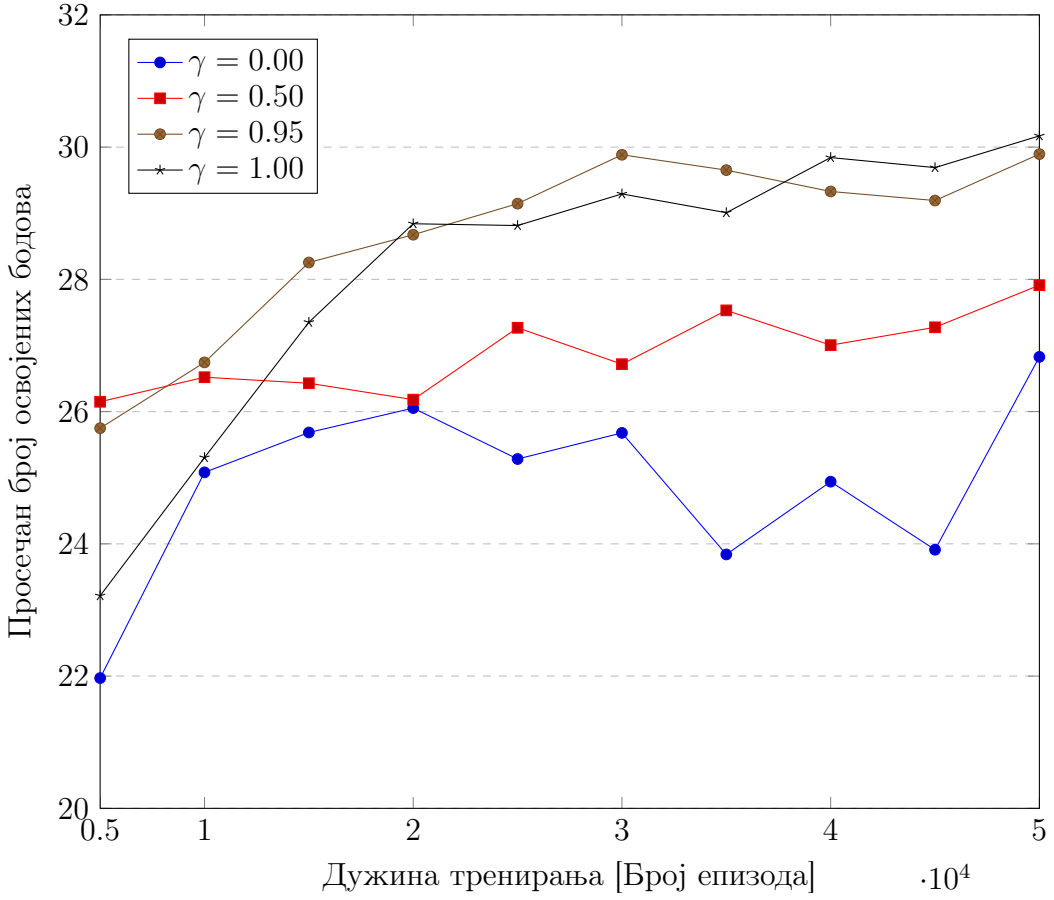
Листинг 22: Имплементација методе *get_random_play*

Ради лакшег имплементирања ε-похлепне политике која се користи током тренирања, имплементирана је метода *get_random_play* дата у листингу 22 која враћа случајно изабран потез.

За тренирање овог играча коришћен је алгоритам *q-учења* описан у алгоритму 1. Имплементација овог алгоритма се може наћи у прилогу у датотеци *tablic_training_grounds.py*.

5 Резултати

У овом делу ће бити приказани и анализирани резултати тренирања модела подржаним учењем. За потребу анализе тренирана су 7 играча базираних на подржаном учењу. Ови играчи ће најпре бити упоређени са похлепним играчем, а биће упоређени и међусобно.



Слика 2: График зависности броја освојених бодова од дужине тренирања

На слици 2 је приказан график који приказује број освојених бодова играча против похлепног играча у зависности од дужине тренирања. На графику су приказане вредности за четири различита модела, код којих је једина разлика избор вредности фактора умањења. На основу графика можемо приметити да се перформансе модела, са повећањем дужине тренирања углавном повећавају али то није увек случај. Током тренирања многи случајни елементи утичу на тренинг и често може да се деси да се агент преобучи у одређеним ситуацијама што понекад доведе до пада у перформансама.

Треба напоменути да перформансе ових модела доста зависе од многих случајних елемената. Иницијализација мреже, одигране партије и потези, па чак и сам процес тренирања имају доста случајних елемената. Самим тим на основу овог графика не можемо са великом сигурношћу рећи који фактор умањења је најбољи, али можемо закључити да су фактори умањења од 0.95 и 1.00 бољи

од преостала два. Али иако модел са фактором умањења 1.00 има мало боље перформансе, не можемо са сигурношћу рећи да је бољи избор од модела са фактором умањења 0.95 што додатно може да потврди чињеница да се ова два модела често смењују у томе који осваја више бодова како број епизода расте.

Играч	број победа	број нерешених	број пораза	број бодова играча	број бодова противника
GreedyTablicPlayer	0	1000	0	26.214	26.214
ReinforcedTablicPlayer $\gamma = 0.00$	535	32	433	26.828	25.341
ReinforcedTablicPlayer $\gamma = 0.50$	592	31	377	27.912	25.359
ReinforcedTablicPlayer $\gamma = 0.75$	620	33	347	28.617	24.703
ReinforcedTablicPlayer $\gamma = 0.85$	685	39	276	29.257	24.158
ReinforcedTablicPlayer $\gamma = 0.90$	665	32	303	29.141	24.124
ReinforcedTablicPlayer $\gamma = 0.95$	748	32	220	29.894	23.347
ReinforcedTablicPlayer $\gamma = 1.00$	732	38	230	30.170	23.232

Табела 1: Резултати играча против похлепног играча

У табели 1 приказани су резултати истренираних играча против похлепног играча. Сваки играч је одиграо 1000 партија против похлепног играча. Прво су приказани резутати похлепног играча, где се може приметити да су све партије нерешене. Пошто је похлепни играч потпуно детерминистички, ово је очекивано, јер ће два једнака играча увек одиграти партију на исти начин. Самим тим за похлепног играча метрика о броју победа и пораза нема никакву вредност, али број освојених бодова представља добру метрику за даљу анализу других играча.

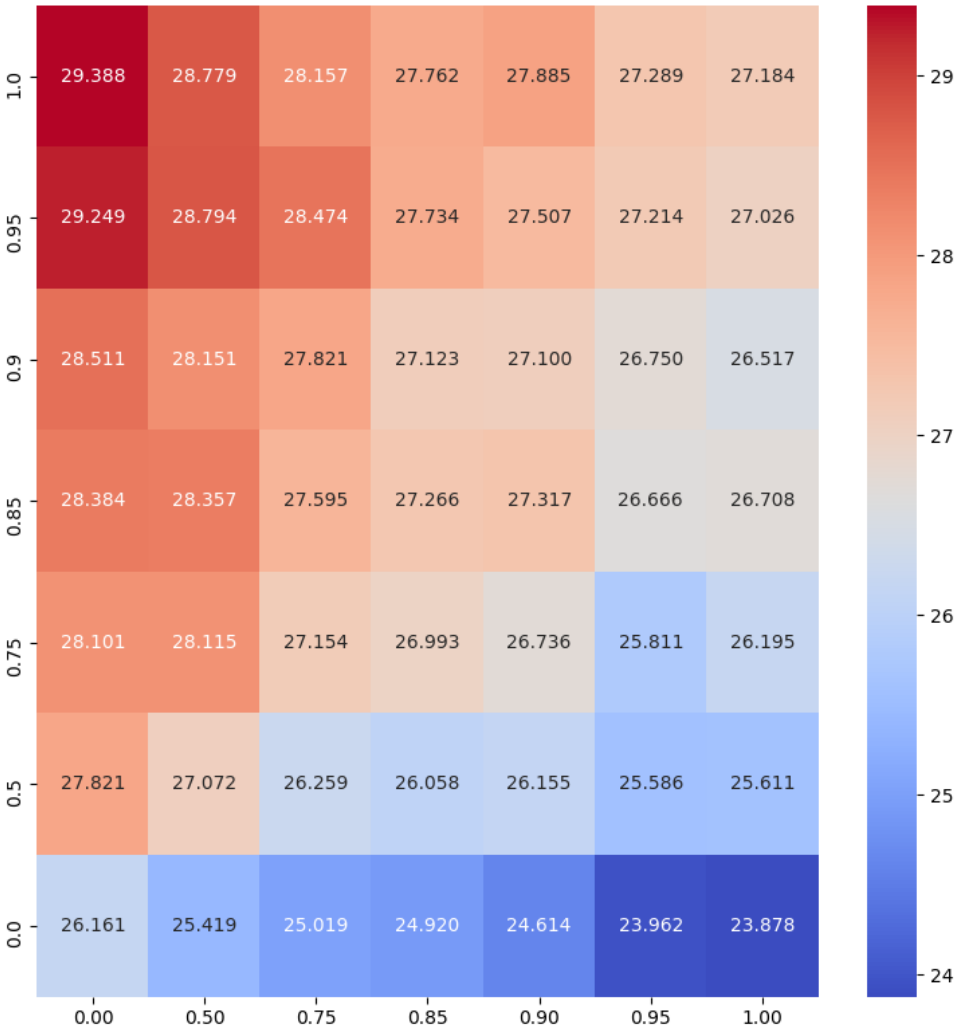
Сви играчи тренирани подржаним учењем приказани у табели су тренирани са истим параметрима, истом неуронском мрежом, и истим бројем епизода (50000). Једина разлика између њих представља вредност фактора умањења γ.

На основу ове табеле може се видети да сви играчи тренирани подржаним учењем имају боље перформансе од похлепног играча и да најбоље перформансе имају играчи са највећим фактором умањења. Интуитивно ово има смисла, с обзиром да играчи са већим фактором умањења више вреднују дугорочну награду и самим тим могу боље да науче како да максимизирају укупан добитак унутар једне партије.

Може се приметити да играч са фактором умањења $\gamma = 0.95$ односи више победа против похлепног играча, али мање бодова него играч код кога је фактор

умањења $\gamma = 1.00$. С обзиром да су играчи истренирани да максимизирају број освојених бодова а не број победа, можемо рећи да боље резултате има други играч.

Резултати који се налазе у табели 1 се односе на перформансе играча против похлепног играча. Због овога је потребно упоредити играче међусобно јер играч који је добар против похлепног играча не мора да буде добар и против других играча.



Слика 3: Графички приказ перформанси између играча

На слици 3 су графички приказани резултати остварени између свих играча. Редови означавају играча, а колоне означавају противника. Унутар сваке ћелије је приказан број бодова које је играч освојио против противника. Ћелије су обојене у зависности од броја освојених бодова где су ћелије са највећим бројем бодова обојене црвеном бојом а оне са најмањим плавом бојом.

Може се приметити да повећањем фактора умањења играча, играч осваја више бодова, а повећањем фактора умањења противинка, играч осваја мањи број

бодова. Самим тим може се закључити да играчи са већим фактором умањења играју боље од оних код којих је фактор умањења мањи.

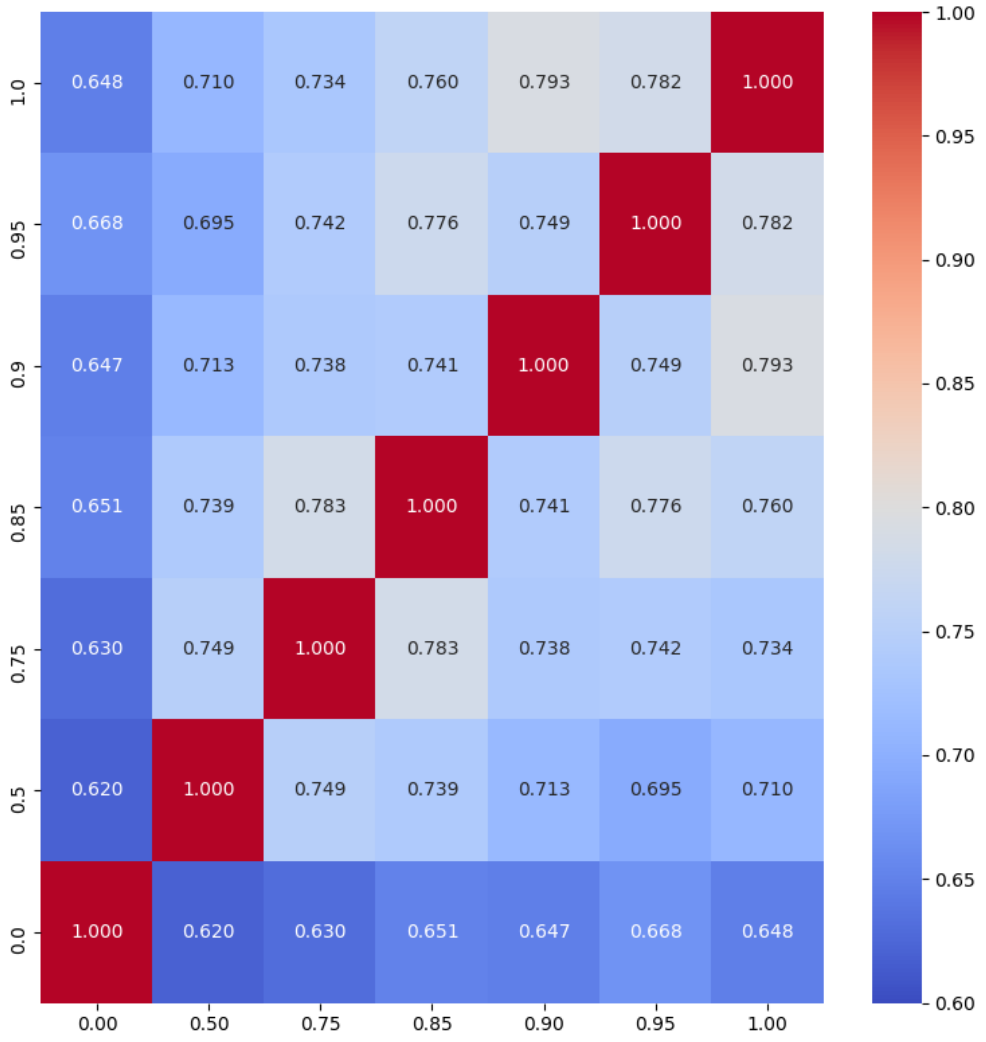
Играч	Проценат похлепности [штихови]	Проценат похлепности [штихови и карте]
ReinforcedTablicPlayer $\gamma = 0.00$	99.84326%	95.27962%
ReinforcedTablicPlayer $\gamma = 0.50$	99.82942%	91.86767%
ReinforcedTablicPlayer $\gamma = 0.75$	99.75830%	95.36051%
ReinforcedTablicPlayer $\gamma = 0.85$	99.65462%	96.80582%
ReinforcedTablicPlayer $\gamma = 0.90$	99.57421%	97.65380%
ReinforcedTablicPlayer $\gamma = 0.95$	99.17643%	98.46321%
ReinforcedTablicPlayer $\gamma = 1.00$	98.69075%	97.67203%

Табела 2: Проценат похлепности играча

У табели 2 су приказани проценти похлепности сваког од играча. Проценат похлепности представља метрику која говори о начину играња играча. Тачније представља проценат потеза који играч одигра који су похлепни. У табели се налазе две вредности процента похлепности, прва гледа да ли одигран потез односи највећи број штихова, а друга гледа да ли одигран потез односи највећи број штихова и карата.

Може се приметити да порастом фактора умањења похлепност која гледа само штихове опада, док похлепност која гледа штихове и карте расте. Овакво понашање је и логично с обзиром да је награда од однетих штихова тренутна, а награда од однетих карата долази са закашњењем, односно тек кад играч однесе 27-у карту.

Из ове табеле се такође може видети да сви играчи имају прилично велики проценат похлепости. Ради перспективе, проценат похлепности од 99% би значио да играч не игра похлепно сваки 100-ти потез, што је приближно једном у четири партије. Али иако су ови играчи веома похлепни, имају боље резултате него похлепни играч и разлог томе је објашњив. У једном потезу може бити више похлепних потеза, и ови играчи су бољи у бирању таквог од похлепног играча. То јест, ови играчи бирају похлепне потезе који су дугорочно бољи него потези које бира похлепни играч.



Слика 4: Графички приказ сличности између играча

На слици 4 приказана је сличност између играча. У свакој ћелији се налази вредност која представља колико често два играча одређена редом и колоном ћелије играју исте потезе. На основу овога може се приметити да иако сви играчи углавном играју похлепне потезе, постоје доста велике разлике између њих.

6 Закључак

Тренутно, подржано учење је у поређењу са надгледаним и ненадгледаним учењем најмање примењена и развијена парадигма машинског учења. Појавом *AlphaZero* програма, популарност подржаног учења је знатно порасла, поготово у домену игара. Подржано учење је најсличније начину учења људи и због тога алгоритми за подржано учење представљају једне од најгенералнијих алгоритама за учење. Самим тим подржано учење представља веома занимљиву тему у области вештачке интелигенције.

У овом раду су објашњени приници подржаног учења и касније примењени како би се реализовао модел за играње игре Таблић. Овај модел је релативно лако успео да надмаши похлепан алгоритам. И оно што је најбитније, то је успео без икаквог знања о томе како треба да се игра Таблић, већ је целокупно знање стекао само интеракцијом са окружењем. Показано је да за успех модела није потребно знање о начину на који модел треба да се понаша већ је потребно само направити добар модел окружења и алгоритам ће да одради своје.

У последњем делу је урађена анализа између различитих модела и показан је утицај фактора умањења на понашање алгоритма. Такође је показано да иако модели играју веома похлепно, да то чине са дугорочним добитком у плану и самим тим успевају да победе похлепни алгоритам.

Овај пројекат представља добру основу за даље унапређивање модела за игру Таблић. За даље унапређивање могућа унапређења су:

- Модификовање неуронске мреже, додавањем бар још једног слоја. Ово решење би успорило тренирање али би омогућило сложенија закључивања.
- Повећање броја епизода којим се модел тренира.
- Коришћење неког другог алгоритма за подржано учење уместо *q-учења*, на пример *Policy-gradient* алгоритма.

Такође овај пројекат даје добру основу и за имплементирање алгоритма *q-учења* на друге игре.

Литература

[1] Richard S. Sutton, Andrew G. Barto, *Reinforcement Learning: An Introduction, second edition*, MIT Press, Cambridge, MA, 2018

[2] Младен Николић, *Машинско учење*, Математички факултет, 2020

[3] *Tablić - pravila igre* (приступљено у октобру 2020. године): `http://www.igrajkarte.com/blog/post/tablic-pravila/`

[4] *Rules of card games: Tablić* (приступљено у октобру 2020. године): `https://www.pagat.com/fishing/tablic.html`

[5] *Python 3 documentation* (приступљено у октобру 2020. године): `https://docs.python.org/3/`

[6] *PyTorch documentation* (приступљено у октобру 2020. године): `https://pytorch.org/docs/stable/index.html`

[7] *NumPy Reference - Release 1.19.0* (приступљено у октобру 2020. године): `https://numpy.org/doc/stable/numpy-ref.pdf`

Прилог

[1] Изворни код пројекта: `https://github.com/nkili/TablicRL/tree/initial`