EÖTVÖS LORÁND TUDOMÁNYEGYETEM

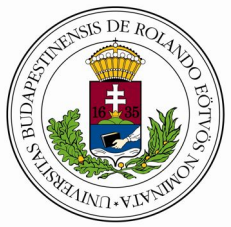
TERMÉSZETTUDOMÁNYI KAR

TETRIS JÁTÉK MEGOLDÁSA GÉPI TANULÁSSAL

SZAKDOLGOZAT

Hortobágyi Róbert
Matematika BSc
Matematikai elemző szakirány

Témavezető: Lukács András
Számítógéptudományi Tanszék



Budapest
2021

NYILATKOZAT

Név: Hortobágyi Róbert

ELTE Természettudományi Kar, szak: Matematika Bsc

NEPTUN azonosító: O3AZLS

Szakdolgozat címe:
Tetris játék megoldása gépi tanulással

A **szakdolgozat** szerzőjeként fegyelmi felelősségem tudatában kijelentem, hogy a dolgozatom önálló szellemi alkotásom, abban a hivatkozások és idézések standard szabályait következetesen alkalmaztam, mások által írt részeket a megfelelő idézés nélkül nem használtam fel.

Budapest, 2021.05.29.

a hallgató aláírása

Köszönetnyilvánítás

Szeretnék köszönetet mondani témavezetőmnek Lukács Andrásnak a szakmai hozzájárulásáért és konzultációinkba belefektetett óráiért. Ezenkívül szeretnék még köszönetet mondani családomnak a folytonos támogatásukért, amit nyújtottak életem során.

Tartalomjegyzék

1. fejezet

Bevezetés

Szakdolgozatom alapötletéül egy Youtube videó szolgált, névlegesen Code Bullet felhasználótól az *I created an A.I. to DESTROY Tetris*. A videó egy Tetris (1984) játék megépítéséről és annak automatizált játszásáról szól, viszont a mesterséges intelligenciát nem gépi tanuláson keresztül hozza létre, hanem egy formula alapján számolja ki a gép a legjobb lépést és az alapján cselekszik. A készítő is megemlíti, hogy nem fog gépi tanulással foglalkozni a videóban, annak ellenére, hogy ez korunk egyik fontos tudományága. Hiányzó előtudásom ellenére érdekelt, hogy meglehet-e oldani ezt a problémát gépi, azon belül mély tanulással és mivel a szakdolgozatra való felkészülés is jó motiváció arra, hogy új dolgokat tanuljak ezért ideálisnak találtam ennek keretében megismerkedni a témával.

A gépi tanulás tudománya olyan számítógép algoritmusokkal foglalkozik, melyek maguktól fejlődnek, csupán abból, hogy tapasztalatokat szereznek az adatfelhasználás során. Ezt úgy éri el az algoritmus, hogy rendszert keres az adatokban, és az erre felépített modellje segítségével hoz döntéseket, mindezt anélkül, hogy arra külön probléma specifikusan programozni kéne. Gépi tanulással ma már rengeteg helyen találkozhatunk a mindennapjainkban. A weboldalak által mutatott személyreszabott hírdetések, a telefon hangfelismerő rendszere és a spam e-mailek szűrése mind gépi tanulás alapján működnek.

Dolgozatomban a gépi tanulás egyik legizgalmasabb módszeréről írok, a megerősítéses tanulásról. A megerősítéses tanulás a felügyelt- és a felügyelet nélküli tanulás mellett a harmadik alapvető gépi tanulási feladat. Röviden egy ügynököt arra tanítunk, hogy viselkedésével egy ismeretlen környezetben minél több jutalmat tudjon szerezni. Ugyanebben a fejezetben írok még a megerősítéses tanulás egyik algoritmusáról a Q-tanulásról, amelyet Christopher Watkins fejlesztett ki 1989-ben és ennek alapkövéről a Markov döntési folyamatról. A rákövetkező fejezetben a mélytanulást mutatom be, amely az emberi agy mintájára épített gépi tanulási forma. Ez annyit jelent, hogy az adatokat az emberi agyban található neuronokhoz és neurális hálókhoz hasonló modellek dolgozzák fel, ez-

zel érve el tanulási eredményeket. Az utolsó fejezetben a Tetris játék ismertetése után leírom, hogy a korábban tárgyalt technikákkal, hogyan sikerült egy Tetrist játszó tanuló programot megvalósítani. A dolgozat végén a modellek tanulásával elért eredményeket is bemutatom. A dolgozatban felhasznált kódokat az alábbi GitHub linken lehet elérni:
`https://github.com/RobikaXD77/Tetris_jatek_Q_tanulas`

2. fejezet

Megerősítéses tanulás

2.1. Mi is az a megerősítéses tanulás?

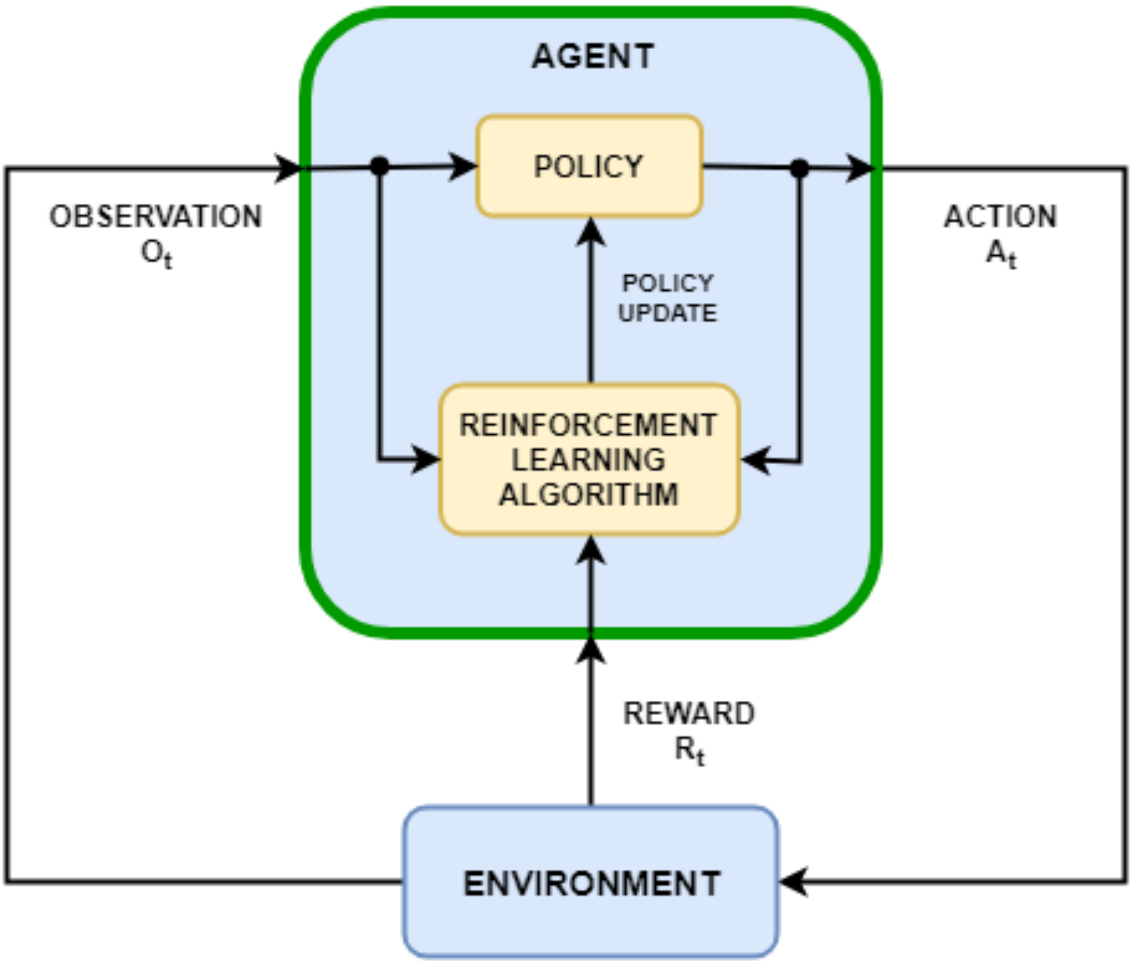
A megerősítéses tanulás mint ahogyan azt a bevezetésben is írtam egy gépi tanulási módszer, amely olyan tanulási problémákra specializálódik, melyekben a gépnek kell elsajátítania, hogy hogyan tudja maximalizálni a jutalmat (reward) amit cselekvéseiért (action) kap, viszont mindezt úgy, hogy nem definiáljuk, milyen lépéseket hajtson végre, hanem magától kell rájönnie és felfedezni, mely műveletek sikeresek vagy ellenkező esetben sikertelenek a jutalom maximalizálásának szempontjából.

Ahhoz, hogy a cselekvéseket tudjuk értelmezni, a gépnek interaktálnia kell valamivel. Ezt a valamit környezetnek (environment) nevezzük. A környezet bemutatására vegyünk egy mindennapi példát az autóvezetést. Mikor vezetünk az út, járókelők, közlekedési lámpák, más autósok, mind a környezet része, amelyre autóvezetési cselekvéseink (például a gyorsítás, kanyarodás, fékezés) hatással vannak. A jutalom ebben az esetben a szabályos cselekvésekkel írható le, mint például a fékezési távolság betartása, vagy a stop tábla előtti megállás. Negatív jutalmat is tudunk definiálni a gép számára, ezek példánkban a gyorshajtásnak, vagy a piros lámpán való áthajtásnak felelhetnek meg. Fontos megjegyezni, hogy a való életben ezek a környezeti tényezők nem konzisztensek. Példánknál maradva, nem minden autós vezet szabályosan, illetve a gyalogosok is leléphetnek az úttestre. Ezért mikor ilyen formában végzünk tanítást, az esetek nagy részében ideális környezetben dolgozunk, külső- és véletlen hatások nélkül annak érdekében, hogy elkerüljük a kiugró adatpontokat. A környezettel azonban nem bármilyen gép tud interaktálni. Az erre specializálódott gépeket nevezzük ügynöknek.

Az ügynök nem csak véletlenszerűen hat a környezetére, hanem céljai is vannak, amelyek a környezet állapotától (state) is függenek. Nyílvánvalóan ehhez az kell, hogy az ügynök valamilyen formában lássa a környezetet és folyamatosan figyelje azt. A cselekvé-

seinek is olyanoknak kell lennie, amelyek változtatnak a környezet állapotán. Ahhoz, hogy mindez értelmet nyerjen az ügynöknek cselekvéseiből tapasztalatokat (experience) kell szereznie és felhasználnia azokat, hogy javítson a későbbi cselekvésein, melyeket jutalom fomájában pontozunk, így elérve magát a tanulás folyamatát. Még fontos megjegyezni, hogy a gyakorlatban az ügynök és a környezet nem feltétlen különböző entitások. Előző példánknál maradva, az autó benzin szintje például lehet a környezetnek a része annak ellenére, hogy magához a személygépjárműhöz tartozik fizikailag.

Ahhoz hogy teljesen megértsük a megerősítéses tanulást, írnom kell még a matematikai aspektusairól, amelyekből számol és tanul az gép. Ezek közé tartozik a eljárás (policy), jutalom visszajelzés (reward signal), érték függvény (value function) és a környezeti modell.



2.1. ábra. Megerősítéses tanulás folyamata - [14]

Az eljárás írja le, hogy az ügynök milyen módon cselekedjen céljai eléréséhez, figyelembe véve a környezet állapotát. Ez a matematikai képlet, amely lehet egyszerűbb vagy összetettebb az alapja minden megerősítés tanulási ügynöknek és általánosságban szto-

chasztikus.

Az ügynök céljainka elérésére szolgál a jutalom visszajelzés, amely az ügynök minden lépése (step) után a már fent említett jutalmat küldi vissza az ügynöknek egy szám formájába azonnali visszajelzést biztosítva. A visszajelzésből tudja meg az ügynök, hogy mely lépései voltak jók vagy rosszak és ezek alapján próbálja meg maximalizálni a jutalmat tanulása során. Mivel ez a visszajelzés azonnali, ezért nem tud hosszútávú tanulást biztosítani , viszont erre találták ki az érték függvényt. Az érték függvény írja le, hogy egy adott környezeti állapotból, mennyi jutalomra számíthat az ügynök, ezzel megmutatva olyan eseteket ahol, lehet a jutalom nem a legjobb egy lépés után, viszont az azt követő állapotból összeségében, több jutalmat tud elérni. Vegyük az emberi egészséget példának. Ha egészségtelenül étkezünk, lehet jól érezzük magunkat mialatt eszünk viszont nem tesz jót a szervezetünken, ezért hosszútávon nem jó lépés. Ellenkező esetben ha az ember elmegy edzeni, elfárad és ezért kevésbé érzi jól magát, viszont hosszútávon egészséges életmódot folytat (ha tovább élünk, több jutalmat tudunk szerezni).

Az utolsó felhasznált tényező, a környezeti modell. Mint minden modell a környezeti modell is a valóság viselkedését utánozza, egy leegyszerűsített formában. Ezek a modellek ábárzolják, esetleg előre kiszámolják a környezet állapotait. [4]

2.2. Markov döntési folyamat

A fenti bevezető után, fontos, hogy a matematikai hátterét is megértsük a megerősítéses tanulásnak. A megerősítéses tanulás alapja a Markov döntési folyamat és a Markov tulajdonság. Ennek az alfejezetnek a megírásához a [3] cikket használtam fel.

2.1. Definíció (Markov-lánc). *Állapotok sorozatát akkor és csak akkor nevezzük Markov tulajdonságúnak, ha a valószínűsége annak, hogy a következő S_{t+1} állapotba lépünk csakis az azt megelőző S_t állapoton alapszik és nem pedig bármelyik korábbi $S_1, S_2, S_3 \dots S_t$ állapoton minden t esetén.*

Példaként vegyük a *Ki nevet a végén?* társasjátékot. Ebben a játékban bármelyik időpillanatban a következő lépés és ezzel a tábla következő állapota csak azon múlik, hogy az éppen soron lévő játékos hányast dob a kockával és milyen a jelenlegi tábla állapota. Nem számít, hogy több körrel ezelőtt, hogyan nézett ki a tábla, mivel ez nem befolyásolja az adott időpillanatban történő lépést. Egy kontraszt példát láthatunk kártyajátékoknál. A Black Jack játékban minden új felhúzott lapot befoláysolja az, hogy korábban milyen lapokat osztottunk ki. Mivel a következő állapotot az összes előtte lévő állapot befolyásolja, ezért a Black Jack nem Markov tulajdonságú.

2.2. Definíció (Markov döntési folyamat). *Markov döntési folyamatnak (MDF) nevezzük azokat a* (S, A, P, γ, R) *rendezett listákat ahol:*

- S *az állapotok véges halmaza*
- A *a lépések véges halmaza*
- P *az a mátrix, amely az állapotok közötti átmenetnek a valószínűségét tartalmazza* $P_{ss^{\emptyset}}(a) = P(S_t + 1 = s^{0})\mathbf{j}(S_t = s, A_t = a)$
- γ *2* $[0, 1]$ *az úgynevezett leértékelési változó.*
- $R : S \quad A$ *!* R *a jutalom függvény*

Markov döntési folyamatot megerősítéses tanulás környezetének modellezésére használjuk. MDF-ben a következő $S_t + 1$ állapotba lépés nem csak az előtte lévő S_t állapottól függ, hanem attól is, hogy milyen A_t lépést használt az ügynök az adott állapotban. Ezenkívül minden lépés-állapot pároshoz egy jutalom függvény van rendelve, amely a páros jóságát írja le. Maga a folyamat úgy működik, hogy kiindulunk egy s_0 állapotból és választunk a lépési lehetőségeink közül egy a_0 *2* A lépést. Ez átvisz minket a következő s_1 állapotba ami a $P_{s_0 s_1}(a_0)$ -ból választódik ki és ezt folyamatot folytatjuk a következő állapotokra is.

A jutalom egy visszajelzés a környezettől arra, hogy az adott lépésünk mennyire számított jó lépésnek. A gép feladata maximalizálni ezt az értéket. A jutalom függvénye a következő.

$$R_t = r_{t+1} + \gamma \quad r_{t+2} + \gamma^2 \quad r_{t+3} + \ldots,$$

ahol t egy adott időpillanatot jelent ahonnan elindulunk. Láthatjuk, hogy ez a függvény tartalmazza a γ leértékelési változót. Erre azért van szükségünk mert felmerülhet a probléma, ahol ez a szumma a végtelenhez tart, ha $\gamma = 1$. Ezenkívül a későbbi jutalmaknak a súlyozását szokták beállítani a paraméterrel.

Az eljárás egy olyan függvény, amely megadja, hogy milyen lépést kell az ügynöknek elvégeznie egy adott állapotban. ezt a függvényt $\pi(s, a)$-val jelöljük. A függvény visszaadja a valószínűségét annak, hogy a lépést választunk az s állapotban. Mivel ez egy valószínűségi eloszlás ezért $\sum_a \pi(s, a) = 1$. A megerősítéses tanulás egy optimális eljárást próbál keresni, viszont ha nem véletlenszerűen akarunk elindulni a tanuláson érdemes egy kezdeti eljárást is megadnunk. A leggyakrabban használt ilyen eljárás az epszilon-mohó algoritmus ami a következőképpen működik. Minden lépés előtt az ügynök $1 \quad \varepsilon$ valószínűséggel az általa tekintett legjobb lépést választja és ε valószínűséggel pedig egy véletlenszűren választott másik lépést tesz. Ez azért hasznos, mert új lépéseket tud

így felfedezni, amelyek jobbak lehetnek a régi legjobbnak ítélt lépésnél. Ezt a módszert nevezzük felderítés és kihasználásnak (exploration and exploitation).

Annak érdekében, hogy optimális eljárást találjunk a tanulás során valamilyen módon tudnunk kell mérni az állapotaink és lépéseink jóságát. Ehhez érték függvényeket definiálunk. Az állapot érték függvény megadja, hogy egy adott állapotból mi a várható értéke az összes jutalomnak amit kaphatunk belőle, ha követjük π eljárást.

$$V^{\pi}(s) = \mathsf{E}\left[\sum_{t=0}^{1} \gamma^t R_{t+1} \middle| s_t = s\right]$$

A lépés érték függvény hasonló az állapot érték függvényhez, csak itt figyelembe vesszük milyen lépést teszünk az adott állapotban és ennek tekintetében kapjuk vissza az összes jutalom várható értékét.

$$Q^{\pi}(s) = \mathsf{E}\left[\sum_{t=0}^{1} \gamma^t R_{t+1} \middle| s_t = s, a_t = a\right]$$

Maguk a tanulási algoritmusok erre a kettő érték függvényre alapszanak. Több ismert tanulási algoritmus is létezik a Monte Carlo és a SARSA is ilyenek. Mivel dolgozatomban Q-tanulást használok fel ezért ezt fogom részletesebben kifejteni.

2.3. Q-tanulás

A Q-tanulás egy off-policy megerősítéses tanulási algoritmus. Off-policy-nek vagy magyarul eljáráson kívülinek nevezzük azokat az algoritmusokat, amelyek az ügynök lépéseitől függetlenül tanulják meg az optimális eljárást. Mit is jelent ez? A Q-tanulási algoritmus a lépéseit a megadott kezdeti eljárás szerint választja (mi esetünkben ez az epszilon-mohó algoritmus), azonban a Q érték függvényt az alapján frissíti, hogy a következő állapotban melyik lépés adja a maximális Q értéket. Ezt úgy is vehetjük, hogy egy teljesen mohó eljárás alapján frissíti az értéket ($\epsilon = 0$). (Ebből következik, hogy az on-policy algoritmusok pedig a választott lépés értékével frissítik a Q értéket). Az algoritmus a [4] könyvből származik.

Algorithm 1: Q-tanulás algoritmus π π becslésére

Algoritmus paraméterei: lépés nagyság α 2 $(0, 1]$, kicsi $\epsilon > 0$;

$Q(s, a)$, inicializálása minden s 2 S^+, a 2 $A(s)$, önkényesen, kivéve

$Q(\text{terminal}, \ ) = 0$;

foreach *episode* **do**

  S inicializálása;

  foreach *step of episode* **do**

    Válasszunk ki egy A-t S-ből a Q-ból származtatott eljárás alapján (pl., ε-mohó);

    Lépjünk A szerint, figyeljük meg R, S^0;

    $Q(S, A)$ $Q(S, A) + \alpha[R + \gamma \max_a Q(S^0, a)$ $Q(S, A)]$;

    S S^0;

    Amíg S terminális

  end foreach

end foreach

Az algoritmus elején felállítjuk a Q táblázatot minden $Q(s, a)$ párra ami általában egy nullmátrix. Ezután az epizód során minden lépésben választ az ügynök egy A lépést epszilon-mohó algoritmus szerint. Megteszi a lépést majd a $Q(S, A)$ értéket az algoritmusban található függvény alapján frissíti. Ezután az új állapot a lépés utáni állapot lesz. Ezt addig folytatjuk amíg az epizód befejeződik. Az egyetlen változó amiről még nem írtam az α. Ez a változó a tanulási sebességet határozza meg, még pedig úgy, hogy az új információ amivel frissítjük a Q értéket, milyen sebességgel írja felül a régit. Minél közelebb van ez 1-hez annál inkább fogja preferálni az új információkat a tanulásból míg teoretikusan ha 0 lenne akkor nem fejlődne semmit az ügynök. Általában ez az érték 0.0001 α 0.1 .

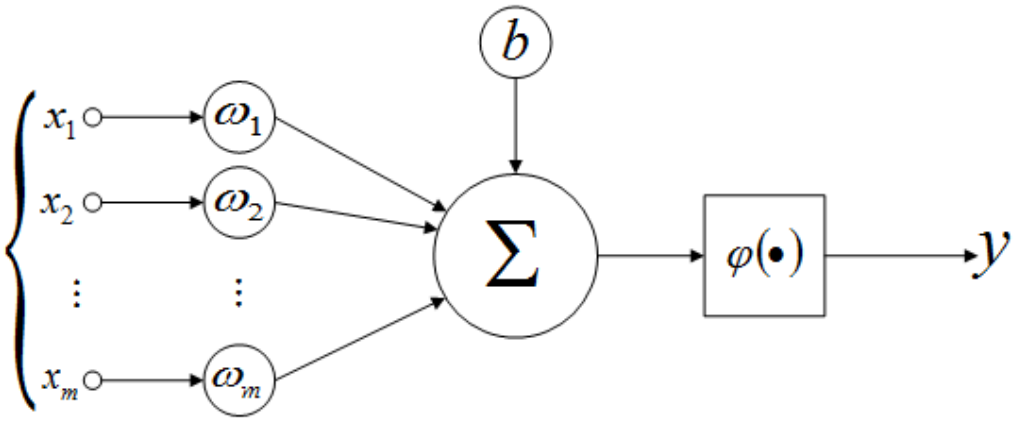
3. fejezet

Mélytanulás

A mélytanulás kezdetét 2010-re tesszük, hiszen ekkor sikerült az első mesterséges neurális hálózatokon alapuló gépi tanulást megvalósítani. A tanulás és a hálózat alapötlete az emberi agyban található neuronok alkotják, melyek elektromos- és kémiai jeleket küldenek egymásnak az agyban ezzel feldolgozva az oda érkező impulzusokat, amelyekre gépi tanulás szempontjából tekinthetünk úgy mint bemenő adatok. Ezeknek a neurális hálózatoknak az alapegyése a mesterséges neuronok. A neuron egy olyan több-bemenetű egy-kimenetű eszköz, amely a bemenetek és kimenetek között általában valamilyen nem lineáris leképezést valósít meg.

A 3.1 ábra kiválóan reprezentálja a neuron felépítést. $x_1, x_2, \ldots$ a bemenő adatokat míg, $w_1, w_2, \ldots$ a bemenő adatokhoz tartozó súlyokat jelöli. $\gamma(\ )$ az aktivációs függvény ami a leképezést valósítja meg, y pedig a kimenő adatokat jelöli. A b változót bias-nek nevezzük és arra szolgál, hogy az adatokat jobban tudjuk általánosítani, ezzel csökkentve a kiugró adatpontok hatását a tanulásra.

Fontosnak tartom, hogy írjak pár szót az aktivációs függvényekről. Mélytanulás során a neuron aktivációs függvényét a tanulás célja alapján választják. A kettő legelterjedtebb a szigmoid és a ReLU (Rectified Linear Unit) aktivációs függvények. A szigmoid függvényt olyan tanítások során alkalmazzák gyakran, ahol valamilyen valószínűséget szeretnének meghatározni kimeneti adatként. Ez azért lehetséges mert a kimeneti adatok a szigmoid függvény hatására 0 és 1 közötti értéket vesznek fel. Következőképpen írható fel.



3.1. ábra. Neuron felépítése - [15]

$$S(x) = \frac{1}{1+e^{\ x}} = \frac{e^x}{e^x+1} = 1 \quad S(\ \ x)$$

A ReLU aktivációs függvény manapaság legelterjettebb, mivel az összetetteb mély neurális hálozatokat jól kezeli és kiküszöböli a szigmoid függvénynek egy nagy hibáját az eltűnő gradiens problémát [16]. A probléma a tanulás során történik mivel a szigmoid függvény mindig valamilyen S alakot vesz fel, ezért a parciális deriváltak, amelyekkel az ilyen problémákban frissítjük a súlyok értékét eltűnően kicsik. Ez effektíve megakadályozza a súlyok változását, rossz esetben a tanulást is megállítja. A ReLU függvény a maximum x értéket adja vissza ha az érték pozitív, egyéb esetben nullát kapunk.

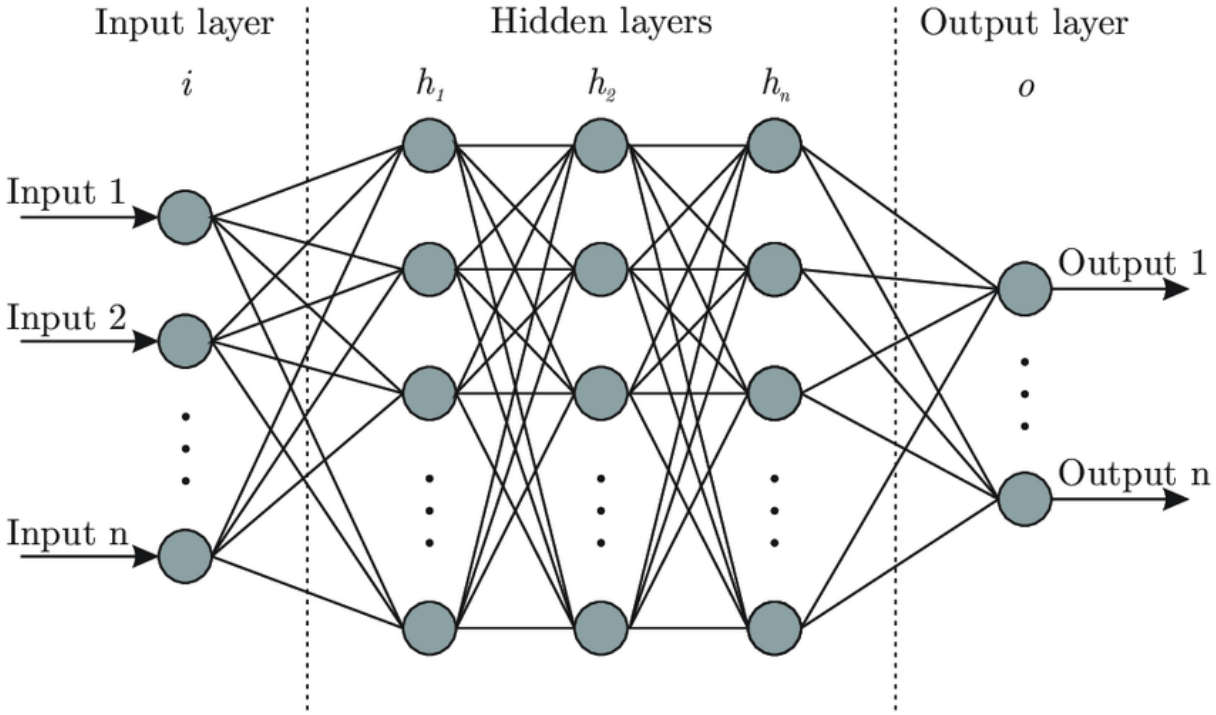
$$f(x) = max(0, x)$$

3.1. Neurális hálózatok

A mesterséges neurális hálózatokat sok mesterséges neuron összeköttetése alkotja, ezzel próbálva replikálni az emberi agyat. A következő építőeleme a neurális hálózatoknak a neuron után a réteg. Neuron rétegnek nevezzük azoknak a neuronoknak a csoportját, amelyek hasonlóak egymáshoz, de nem állnak egymással összeköttetésbe. A neurális hálózat elkészítése, több réteg összeköttetésével készül. Kettő fontos réteg, amelyet a dolgozatomban is használok a, sűrű- és konvolúciós réteg.

Sűrű rétegnek nevezzük azokat a neurális rétegeket, amelyeknek minden neuronjuk összeköttetésben áll minden szomszédos rétegben lévő neuronnal. Ezt a réteget használják

leggyakrabban gépi tanulásnál, mivel hatékonyabb más lineáris megoldásoknál.

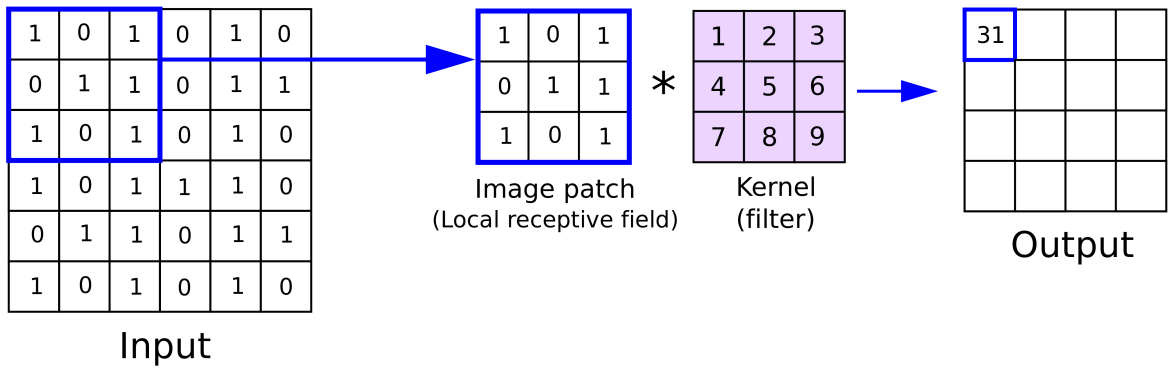


3.2. ábra. Sűrű háló - [17]

A konvolúciós réteg a dolgozatomban használt konvolúciós neurális hálózatok fő építőelemei. Ezeket a rétegeket kétdimenziós képek adatainak gyors és adatvesztés mentes feldolgozására fejlesztették ki. A következőképpen működnek.

- Bemenő adatként megkapják a kép RGB mátrix hármast (más mátrix is szóba jöhet).
- Egy kisebb méretű mátrix, úgy nevezett tapasz (patch) balról-jobbra végighalad a képen.
- Ezek a részmátrixok átmennek a réteg neuronjain (filter) ahol egy előre megírt, vagy tanult módon lineárisan leképezik a mátrixot egyetlen számmá egy mátrix szorzás segítségével.
- Ez ismétlődik addig amíg minden részmátrixot nem jár be a tapasz.
- A procedúra által kapott lekicsinyítet mátrixot kapjuk meg kimenetként.

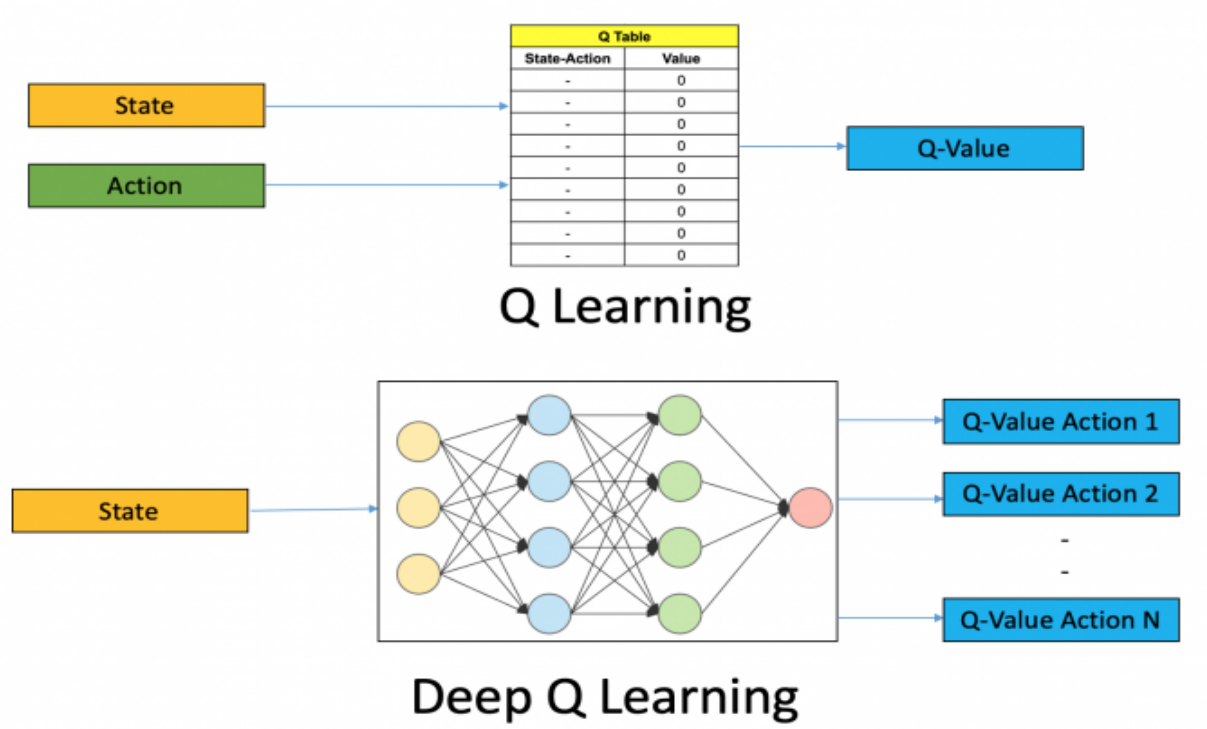
Konvolúcis rétegekben általában több filterrel dolgozunk. Ez a gyakorlatban annyit jelent, hogy mikor megkapjuk a kimenő mátrixot, valójában a filter számával megeggyező számú különböző mátrixot kapunk.



3.3. ábra. Konvolúciós réteg működése 1 filterrel - [13]

3.2. Mély Q-tanulás

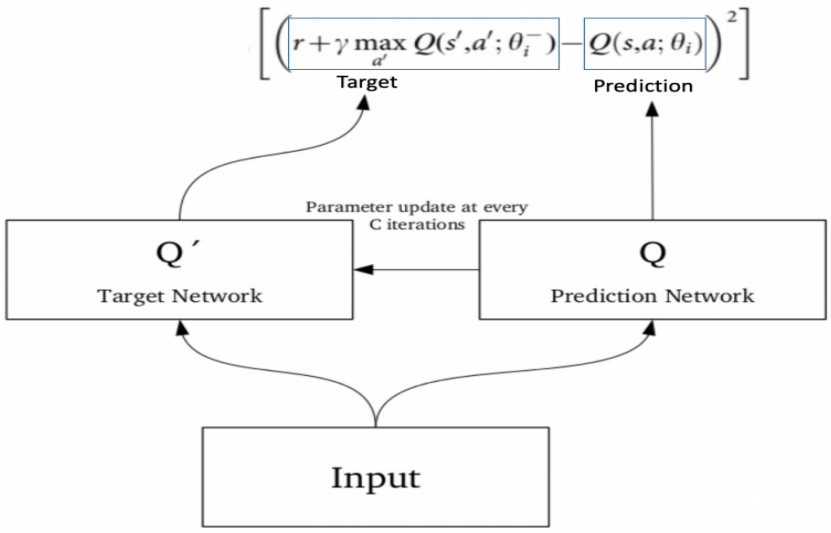
Sokakban az első kérdés ami felmerül a mély Q-tanulással (Deep Q-learning vagy röviden DQN) kapcsolatban, hogy ez miben jobb vagy másabb mint a sima Q-tanulás. Mikor a tradicionális Q-tanulást elkezdjük akkor felállítjuk a Q-táblát minden állapot-lépés párosra. Ez mindaddig nem jelent problémát amíg egyszerű a környezetünk amit reprezentálunk, hiszen addig a Q-tábla is egy kisebb mátrix csak. Azonban ha egy komplexebb környezettel találkozunk, mint például a dolgozatban szereplő Tetris játék, amely közel 2^{200} állapottal rendelkezik akkor azt már nem tudjuk egy Q-táblában tárolni, csak abszurd mennyiségű memória felhasználásával. Ennek megoldására fejlesztették ki a mély Q-tanulást a DeepMind cég kutatói. Mint ahogyan az 3.4-es ábrán is látszik, a Q-tábla helyett egy neurális hálózat fogja előrejelezni a Q-értékeket amiket használ az ügynök. Viszont a megoldás nem ilyen egyszerű.



3.4. ábra. Q-tanulás és Mély Q-tanulás felépítése - [5]

A problémát a mély megerősítéses tanulás okozza. Alapvetően a Q-tanulásban az elérni kívánt célponti Q-érték, amelyet a következő állapot-lépés páros alapján számolunk folyamatosan mozgásban van, hiszen mindig aszerint frissítjük, hogy éppen mi az aktuális maximum Q-érték az adott állapotban, ezzel javítva az eljárásunkat. Mélytanulásnál ez a célponti érték nem változik folyamatosan. Ahhoz, hogy ezt megoldjuk helyben kell hagyni egy időre a hálózatot a tanulás során. Ehhez egy második neurális hálózatot használunk fel, amely a másolata az eredetinek és előrejelzési hálózatnak nevezzük (emulator az algoritmusban, 3.5). Ez fogja előrejelezni az adott állapotban lévő Q-értékeket bizonyos ideig míg a célpont Q-érték hálózatot (amely a következő állapot-lépést számolja) helyben hagyjuk egy időre. Valamennyi lépés után (ez egy meghatározott konstans) pedig frissítjük a célpont hálózatot az előrejelzési hálózat szerint. Ezeknek az információknak a birtokában már értelmezni tudjuk magát az algoritmust a tanulás mögött.

A gradiens módszerről a [2] cikkben lehet bővebben olvasni.



3.5. ábra. Célponti és előrejelzési Q-hálózat működése - [5]

Algorithm 2: Mély Q-tanulás tapasztalati visszajátszással [2]

D visszajátszási memória inicializálása N kapacitással;

Q lépés-érték függvény inicializásála véletlen súlyokkal;

foreach *episode* = 1,M **do**

  $s_1 = x_1$ sorozat és $\phi_1 = \phi(s_1)$ előre feldolgozott sorozat inicializáláas;

  foreach t = 1,T **do**

    ϵ valószínűséggel válasszunk véletlen a_t lépést;

    egyébként pedig válasszuk $a_t = max_a Q\ (\phi(s_t), a; \Phi)$;

    Végezzük el az a_t lépést az emulatorban, vizsgáljuk meg az r_t jutalmat és x_{t+1} képet;

    Legyen $s_{t+1} = s_t, a_t, xt + 1$ és dolgozzuk fel előre $\phi_{t+1} = \phi(s_{t+1})$;

    Tároljuk el $\phi_t, a_t, r_t, \phi_{t+1}$ átmenetet D-ben;

    Válasszunk véletlenszerűen kis adatmennyiséget a D-ben található $\phi_j, a_j, r_j, \phi_{j+1}$ átmenetekből;

    Legyen $y_j = \begin{cases} \mathrm{r}_j & \text{ha } \phi_{j+1} \text{ terminális} \\ \mathrm{r}_j + \gamma max_{a^{\theta}}(Q(\phi_{j+1}, a^{\theta}; \Phi)) & \text{ha } \phi_{j+1} \text{ nem terminális} \end{cases}$

    Hajtsunk végbe gradiens süllyesztő lépést a $(y_j \quad Q(\phi_j, a_j; \Phi))^2$-re ;

    C lépés után $Q \ = Q$

  end foreach

end foreach

4. fejezet

Tanulás implementálása Tetris játékra

4.1. A Tetris játék

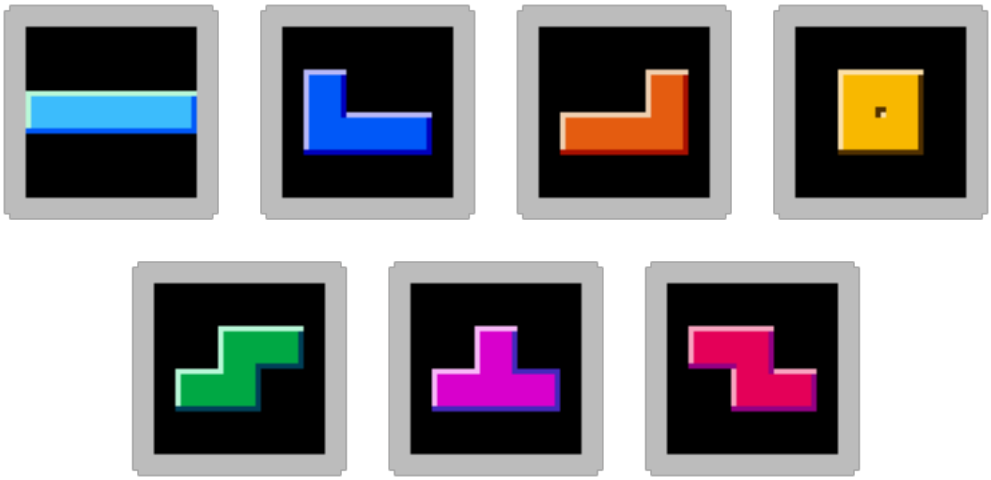
A Tetris ügyességi játékot az orosz származású Alexej Pajitnov készítte Electronika 60 gyártású számítógépekre 1984-ben. A játék angolul a tile-matching kategóriába tartozik, amit magyarul talán legjobban tömb kirakásnak fordíthatunk. Ezekben a játékokban általában színes tömböket, négyzeteket, golóykat kell összeilleszteni a játékosnak ezzel szerezve pontokat a játék során.

Tetrisben különböző négyzetekből összerakott alakzatokat kell összeillesztenie a játékosnak amiket tetrominoknak nevezünk. A játék során véletlenszerűen kiválasztott tetrominok kerülnek a táblára, amelyek egy adott sebességgel folyamatosan esnek lefele. A játékos ezeket az alakzatokat tudja forgatni, lefele (soft drop), balra vagy jobbra tolni illetve ledobni (hard drop), amíg le nem ér a tábla aljára. A játékos akkor kap pontokat, ha az így elhelyezett formák betelítenek egy vagy több sort. Ekkor a betelített sorok eltűnnek és a felettük lévő négyzetek lejjebb csúsznak. A játéknak akkor van vége, ha a legfelső sorba nem képes új tetrominot lehívni a gép, mert az fel van már töltve.

4.1.1. Tetris játékmenet

A alapjáték egy üres 20 10 -es négyzetrácsos táblán játszódik. Hétféle különböző tetromino közül választ véletlenszerűen egyet a gép és elhelyezi a legfelső sorba középre azt, amely ezután ereszkedni kezd. A játékos mindig látja az aktuálisan leeső elem utáni tetrominot, ezzel csökkentve a randomitás hatását a játékra. Modern játékvariánsokban a véletlenszerűség egy tarisznya alapos megoldáson alapszik, ami azt jelenti, hogy a különböző formákból egyet-egyet beletesz egy tarisznyába a gép és abból választ véletlenszerűen. Amint elfogynak belőle az elemek feltölt egy új tarisznyát, ezt ismételve a játék végéig. Ezt a változtatást a legtöbb 2001 után készült játék alkalmazza azért, hogy a játékot

statisztikai alapon is a végtelenségig lehessen játszani. A játékos 100 pontot kap minden eltűntetett sorért, 800 pontot egy tetris-ért ami 4 sor egyszerre való eltűntetését jelenti és 1200 pontot ha a tetris-t megelőzőleg, volt már egy tetris. Bizonyos ponthatároknál (játékonként különbözik) a tetromino gyorsabban kezd el lefele esni, ezzel nehezítve a játékot.



4.1. ábra. Tetrominok - [18]

4.1.2. Dolgozatban felhasznált variáns

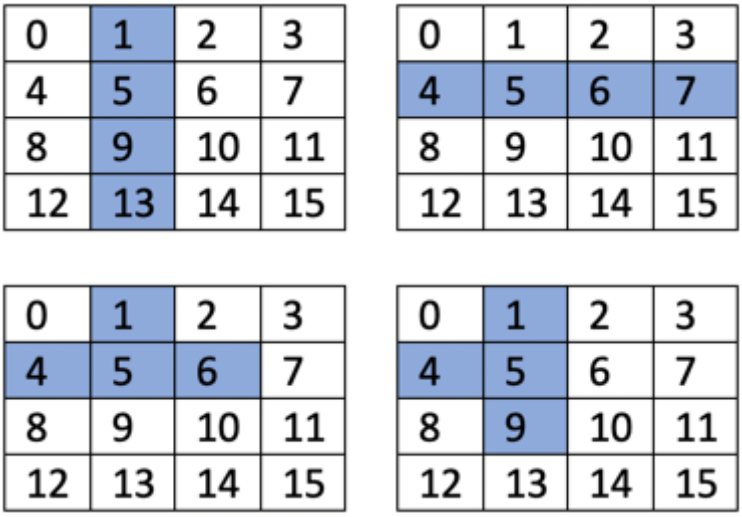
Dolgozatomban a Tetris játék egy módosított variánsát használom fel. Az alapjátékhoz hasonlóan egy 20 10 -es táblán történik a játék és a hétféle tetromino közül teljesen véletlenszerűen választ egyet a gép egyenletes eloszlás szerint. Amiben különbözik ez a variáns az eredetitől, hogy a játékos nem látja az aktuális elem utáni alakzatot, illetve az aktuálisan táblán lévő tetromino sem esik le automatikusan. Azért döntöttem az automatikus leesés megszűntetése mellett, mert az ügynök a döntéseit ezred másodperceken belül hozza meg, ezért ez a tulajdonság nem befolyásolja a tanulást.

4.2. Tetris programozása

Alapvetően egy útmutatót követtem a játék elkészítéséhez [10], és ezt alakítottam át úgy, hogy azt tanulási környezetként lehessen használni. A porgram python nyelven íródott és a következő főbb programcsomagokat használtam fel: Pygame 2.0, Tensorflow 2.3.0, Keras 2.4.3, gym 0.18, numpy 1.18.5.

4.2.1. Tetrominok

Az első fontos objektum amit le kell programozni azok a tetrominok. Több formaliag helyes megoldása van ennek kódolására, mivel ha megfigyelünk egy tetris játékot akkor láthatjuk, hogy az összes tetromino összes forgatása belefér egy 4 4-es mátrixba. Ezenkívül észrevehetjük, hogy az alakzatok forgatása középpontosan történik a tetromino egyik négyzete körül. Ezen információk tudatában tudjuk létrehozni azt a mátrixot, amely tartalmazza a tetrominok és azok forgatottjának alakját is. Továbbá a tetrominok rendelkeznek szín attribútummal is. Az eredeti játékban minden különböző formához különböző szín is tartozik (egyeneshez zöld, S alakhoz kék stb.). Ennek ellenére úgy döntöttem, hogy az egyszerűség és konzisztencia kedvéért elég ha az összes tetrominonak azonos a színe, mivel ez magát a játékot nem befolyásolja.



4.2. ábra. Tetrominok 4x4-es mátrixban - [10]

```
figures = [
    [[4, 5, 6, 7], [1, 5, 9, 13]],
    [[0, 1, 5, 6], [2, 6, 5, 9]],
    [[2, 3, 6, 5], [1, 5, 6, 10]],
    [[1, 2, 5, 9], [0, 4, 5, 6], [1, 5, 9, 8], [4, 5, 6, 10]],
    [[1, 2, 6, 10], [5, 6, 7, 9], [2, 6, 10, 11], [3, 5, 6, 7]],
    [[1, 4, 5, 6], [1, 4, 5, 9], [4, 5, 6, 9], [1, 5, 6, 9]],
    [[1, 2, 5, 6]]
]
```

4.3. ábra. Tetrominok listája kódban

4.2.2. Játéktábla és játékmechanizmusok

A játék kezdetekor a táblát egy 20 10 -es null mátrix reprezentálja, a pontszám nulla és megkapjuk az első random tetrominonkat a (3,0) pozícióba (tábla felső közepe). A tetromino mozgatását könnyen letudjuk programozni, hiszen csak a forma x koordinátáját kell 1 vagy 1 -el megváltoztatni mindig amikor balra vagy jobbra toljuk. Arra kell ügyelni, hogy ha a tábla szélén van akkor maradjon a régi koordináta ha megpróbálnánk átlépni a táblán túl. Forgatásnál annyi a dolgunk, hogy a tetromino figures listájában vesszük a lista következő elemét. Ezt úgy érjük el, hogy a lista hosszával leosztjuk maradékosan azt, hogy hányszor forgattuk meg eddig a tetrominot és ennek a maradéka határozza meg, hogy éppen melyik forgatásnál járunk. Ahhoz, hogy a rossz forgatásokat (szélének ütközik, beleütközik már lenn lévő blokkokba) elkerüljük definiálnunk kell a kódban az intersects funkciót. Az intersects funkciónak a lényege, hogy minden lépésben megnézzük, hogy a tetromino körüli 4 4-es mátrixban találhatóak-e falak, vagy már elhelyezett blokkok. ha igen akkor az intersection érteke True lesz ezzel indikálva, hogy nem lehet forgatni a blokkot legálisan. Ezenfelül definiálnunk kell a freeze funkciót ami a leérkező tetrominokat ahogyan a neve is sugallja megfagyasztja, helyben hagyja. Ez a funkció kétféle képpen következik be. Az aktuális tetromino ledobása úgy van kódolva , hogy addig növeljük 1-el a forma y koordinátáját amíg az intersection értéke nem True. Ha ez bekövetkezik akkor visszaveszünk egyet az y koordínátából, a forma megáll és átszínezzük (0-ból 1-es lesz) a táblát, ezzel vizualizálva, a tetromino megállását. A funkció hasonlóan működik az általános lefele esésre is. Az utolsó funkció amit definiálunk és a freeze funkcióban is megtalálható az a sorok törlése. A breaklines funkció minden freeze funkció után megnézi egy for ciklussal, hogy vannak-e betelített sorok. Ezeket kinullázza a táblán, majd minden blokkot eggyel lejjebb tesz, végül hozzáadja a megfelelő számot a pontszámunkhoz.

4.2.3. Vizualizálás és emberi játék

A játék alapjainak megírása után fontos, hogy azt vizualizálni is tudjuk, hiszen ez látványosabban tűnteti fel a tanítás eredményeit, és lehetőséget ad az emberi játékra is. A vizualizáláshoz a Pygame programcsomagot használtam fel. Ezt a programcsomagot azért hozták létre, hogy a fejlesztők egyszerűbben tudjanak játékokat írni python nyelvben az előre megírt funkciók segítségével. Miután inicializáljuk a programot kettő for ciklussal kitudjuk rajzolni a játékot. Az első for ciklus a táblát rajzolja le, míg a második az éppen aktuális tetrominot. Ezen kívűl címeket, kiírásokat is megtudunk adni az általunk tetszőleges betűtípussal és színnel. A játékban a pontszámot íratjuk ki a képernyőre mint szöveg.

Az emberi játékhoz kettő dologra van még szükségünk. Az éppen leeső tetromino automatikus leesésére, bizonyos időközönként és az emberi input megvalósítására. Szerencsére mindkettő funkció megtalálható a pygame programcsomagban. Az emberi inputhoz csak meg kell adnunk, hogy melyik billentyű lenyomás melyik művelethez tartozzon. A mi esetünkben az éppen leeső tetrominot balra, jobbra nyilakkal mozgatjuk, felfele nyíllal forgatjuk, lefele nyíllal visszük le és a space gomb megnyomásával dobjuk le. Ezenkívül az escape gombot tudjuk használni a játékból való kilépéshez. Az automatikus leesést a Pygame beépített órájával oldjuk meg, amely megfelelő időközönként eggyel lejjebb viszi az éppen leeső blokkot.

4.3. A gép tanítása

Ahhoz, hogy megerősítéses tanulást tudjunk végrehajtani szükségünk van a második fejezetben leírt elemek megvalósítására. Ehhez fogjuk felhasználni a feljebb említett python programcsomagokat. A Tensorflow és Keras csomagok segítségével tudjuk egyszerűen megalkotni a tanulási modellt és az előre felépített DQN ügynök segítségével elindítani a tanulást. Az OpenAI által fejlesztett Gym programcsomag pedig lehetővé teszi, hogy bármilyen játékot megfelelő paraméterezéssel környezetté alakítsunk.

4.3.1. A környezet

Ahhoz, hogy az ügynök interaktálni tudjon a környezettel, valamilyen módon reprezentálnunk kell azt. Ezek lesznek a bemenő adataink ami jelen esetünkben a 20 10-es tábla mátrixa ami nullákból és eggyesekből áll ahol a nullák az üres négyzeteket, míg az egyesek a betelített helyeket jelzik. Ezt úgy sikerült megvalósítani, hogy az éppen leeső tetrominot átkonvertáljuk egy 20 10-es mátrixxá és azt összeadjuk a tábla mátrixxával. Minden lépése után az ügynök a tábla frissített állapotát fogja visszakapni az új tetrominóval és az alapján hozza meg a következő lépését. A lépéseket egy step nevű funkcióval kell megadnunk. Mi esetünkben az ügynök egy lépését az teszi ki, hogy kiválasztja az aktuális tetrominot hányszor forgatja meg, mennyivel mozgatja balra vagy jobbra majd ezekután ledobja azt. Az ügynöknek még szüksége van egy módszerre, hogy újratudja indítani a játékot. Ez lesz a reset funkció ami kiüríti a táblát, a pontokat és az egyéb számolásokhoz szükséges változókat.

4.3.2. Pontozás

Az ügynök lépéseit a következő formula szerint pontozzuk.

$$(2 \quad 0.51 \quad \text{height}() \quad 0.36 \quad \text{holes}() \quad 0.18 \quad \text{bumpiness}())$$

Ennek a három funkciónak a magyar megfelelője sorrendben, magasság, lyukak és hepehupaság. A tábla magasságát a legmagasabban elhelyezkedő blokk határozza meg. Lyuknak nevezünk minden olyan üres mezőt a táblán ahol eggyel felettük blokkok találhatóak. A tábla hepehupaságát pedig úgy számoljuk ki, hogy a szomszédos oszlopok közötti magasságok különbségének abszolút értékét összeadjuk. A tulajdonságokhoz tartozó funkciók két lépés közötti különbségeket adják vissza abszolút értékben. A kettes konstans azért kell, hogy az ügynök lássa a pozitív pontok elérésének lehetőségét, mert a tanítás során észrevettem, hogy ha ez nem látszik akkor az ügynök minimalizálni próbálja a függvényt. Ezenkívűl az ügynök minden betöltött sorért kap 2 pontot. Ezt a függvényt a [11] cikkben alkalmazták és egy genetikai algoritmus segítségével finomították ezekre a számokra.

4.3.3. Alkalmazott neurális hálózatok

Több különböző neurális hálózati modellel dolgoztam a tanítás során, viszont csak ketféle lépési módszert alkalmaztam ezekre a modellekre. A kezdeti lépési módszer alapötlete, az emberi játékot próbálta szimulálni a gép számára. Minden lépése az ügynöknek csak egyetlen egy külön mozgatást végzett (balra, jobbra, forgatás, ledobás). Az első probléma ezzel a lépési módszerrel ott kezdődőtt, hogy az ügynök végtelen játékokba futott, ahol csak forgatta és balra jobbra vitte a leeső tetrominot, mivel nem sikerült megtanulnia, hogy a formák leejtésével tud pontokat szerezni, hiszen a környezet úgy lett megírva, hogy a tetrominok nem estek lefele maguktól. Ennek kiküszöbölésére született a második iteráció, ahol a forgatáson kivűl az ügynök minden lépése után eggyel lejjebb esett az aktuálisan lefelé eső tetromino. Ezzel lényegében egy nehezített Tetris variánst kellett a gépnek megtanulnia. A másik módszer a feljebb leírt csoportosított műveletekkel történt. Az ügynök 40 szám közül választ egyet és ez a szám megadja, hogy az ügynök hányszor szeretné forgatni (0, 1, 2 vagy 3) illetve hányszor szeretné balra vagy jobbra vinni a formát. Ez azért lehetséges mert minden tetromino minden forgatottja legfeljebb 5 lépésből a tábla szélére kerül. Természetesen ez azt eredményezi, hogy lesznek számok amik ugyanazt csinálják bizonyos tetrominoknál (legegyszerűbben a négyzet alakú tetrominonál látni ezt), de sajnos nem sikerült kitalálnom jobb megoldást a lépések reprezentálására.

A különböző neurális hálózatok kipróbálása után végül négy darab maradt, amelyre hosszabb tanítást hajtottam végre. Mindegyik modell egy permutáló réteggel kezdődik, amit az átláthatóság kedvéért használtam fel, hogy ténylegesen egy $20 \quad 10 \quad 1$-es mátrix

legyen a bemenetünk. Mindegyik modell konvolúciós rétegeket (Conv2D) és sűrű rétegeket használ fel (Dense). Ezenfelül mindegyik réteg ReLU (activation = 'relu') aktivációs függvényt használ. Az össze modellben az utolsó réteg egy sűrű réteg, amely annyi neuronnal rendelkezik mint amennyi különböző lépésre képes az ügynök. Az első modell a [1] cikkben találhato architektúra és információi alapján készült. Innentől kezdve ezt a modellt Stanford modellnek fogom nevezni és a következőképpen nézz ki.

```
def build_model(states, actions):
    model = Sequential()
    model.add(Permute((2, 3, 1), input_shape=(1, 20, 10)))
    model.add(tf.keras.layers.Conv2D(32, 3, activation = 'relu', padding = 'valid'))
    model.add(tf.keras.layers.Conv2D(32, 3, activation = 'relu', padding = 'valid'))
    model.add(tf.keras.layers.Conv2D(64, 3, activation = 'relu', padding = 'valid'))
    model.add(tf.keras.layers.Conv2D(64, (14,1), activation = 'relu', padding = 'valid'))
    model.add(tf.keras.layers.Conv2D(128, (1,3), activation = 'relu', padding = 'valid'))
    model.add(tf.keras.layers.Conv2D(128, 1, activation = 'relu', padding = 'valid'))

    model.add(Flatten())

    model.add(Dense(128, activation='relu'))
    model.add(Dense(512, activation='relu'))
    model.add(Dense(actions, activation='relu'))
    return model
```

4.4. ábra. Stanford modell

A kódot az alábbiak szerint értelmezzük. Minden sorral hozzáadunk egy új réteget a modellünkhöz. A konvolúciós rétegeknél az első változó a filterek számát, míg a második a tapasz méretét jelöli, ahol ha csak egy konstans áll, akkor egy négyzetes mátrixként értelmezzük azt. A sűrű rétegeknél az első változó pedig a neuronok számát adja meg. Ahhoz, hogy sűrű rétegeken keresztül tudjuk tanítani az adatainkat vektor formába kell tennünk a paramétereinket. Erre szolgál a Flatten() funkció. Az első modell ötlete azon alapszik, hogy a Tetris lényeges információit oszlopszerűen tartalmazza, ezt pedig a Conv2D(64, (14,1)) réteggel, próbáljuk kihasználni, ami az össze oszlopot 1-1 pixelre zsugorítja, ezzel gyorsítva a tanulás folyamatát.

További kettő modellt alkottam ezzel az ötlettel, azonban ezek lényegesebben leegyszerűsített verziói a Stanford modellnek, hiszen a bemenő adatok is sokkal egyszerűbbek az eredeti modellhez képest. A második modellben egy 10 1-es tapasz a konvolúciós rétegben úgy halad végig a mátrixon, hogy minden oszlopban tíz helyet kihagy, ezzel az outputja egy 2 10 mátrix lesz, ami a felső és alsó felét reprezentálja a táblának. Ezután egy 1 3 méretű tapasszal halad végig, amivel arra próbáltam ösztönözni a gépet, hogy a szomszédos oszlopok fontosságát is megtanulja. A harmadik modell ennél is egyszerűbb, itt az összes oszlopot 1-1 pixelre alakítja egy 20 1 tapasszal amit aztán sűrű rétegeken

küld át. Erre a két modellre Leegyszerűsített Stanford 1 és 2-ként fogok hivatkozni. Az utolsó modell témavezetőmmel ötletelve hoztam létre. Egy Conv2D(32, 10) réteg után az így kapott 11 1 kimenetelt egyetlen pixelre kicsinyítjük, és ezt küldjük át egy 64 neuronos sűrű rétegen. A továbbiakban erre Saját modellként hivatkozunk.

4.4. Eredmények

A fent leírt modellek mindegyikén 10 millió lépésszámú tanítást hajtottam végre (ami körüllbelül 50 órát jelent) egy vagy kettő lépési módszerrel, amit ezután 1000 játékra teszteltem. Mivel a játékban úgy szerzünk pontokat, hogy sorokat rakunk ki ezért ezt vehetjük a kiértékelés alapmércéjének. A lépés oszlopban láthatjuk, hogy melyik lépési módszerrel ment végbe a tanítás. A játék hosszát az alapján határoztam meg, hogy hány tetrominot sikerült a játék végéig leraknia az ügynöknek.

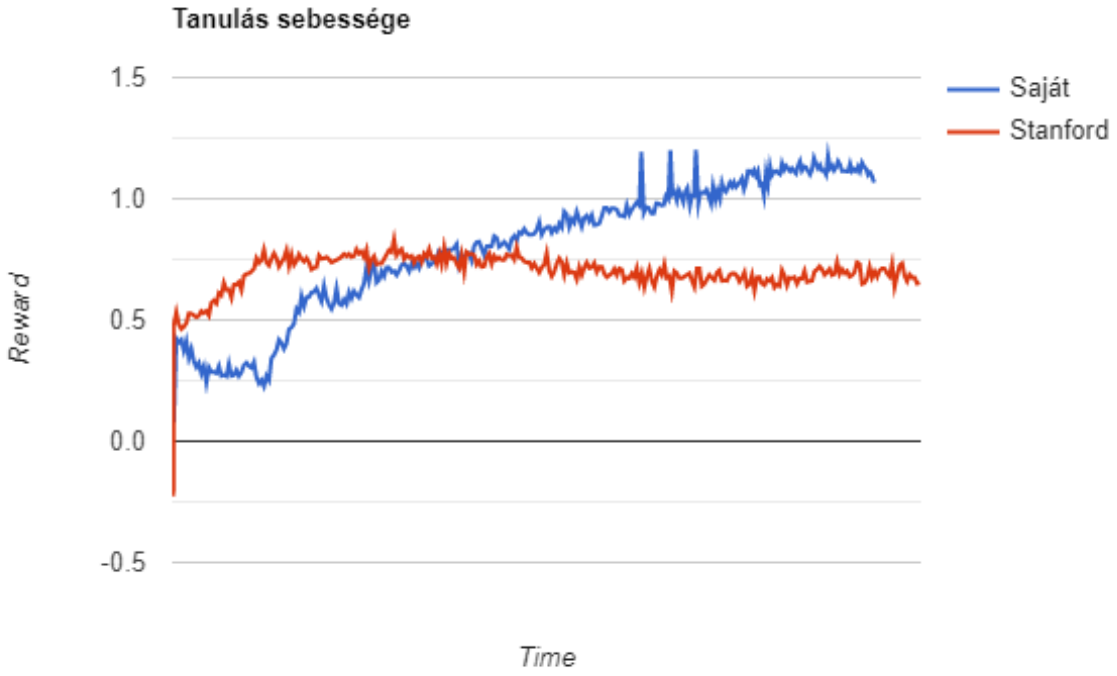
Modell	Lépés	Max törölt sorok	Átlag törölt sorok
Saját	Csoportosított	18	4.512
Stanford	Csoportosított	12	3.97
Leegysz. Stanford 1	Csoportosított	13	1.506
Leegysz. Stanford 2	Csoportosított	2	0.19
Saját	Elkülönített	0	0
Stanford	Elkülönített	0	0

Modell	Lépés	Leghosszabb játék	Átlag játékhossz
Saját	Csoportosított	76	42,974
Stanford	Csoportosított	69	37,409
Leegysz. Stanford 1	Csoportosított	48	18,846
Leegysz. Stanford 2	Csoportosított	38	28,273
Stanford	Elkülönített	11	7,534
Saját	Elkülönített	10	7,226

Az első fontos észrevétel a lépésmódszerek közötti különbség. Láthatjuk, hogy az elkülönített lépésmódszerrel az ügynök nem tanult a lépéseiből, ennek az oka pedig a jutalom függvény. Mivel a függvényt úgy implementáltuk, hogy mindig akkor kap jutalmat az ügynök, ha lerak egy blokkot, ezért rengeteg lépésnél nem tudja felmérni, hogy jónak számított-e az adott lépés vagy nem, hiszen 0 jutalmat kap értük. Számosítva körülbelül minden 10. lépésnél kapott bármilyen jutalmat is az ügynök, mivel ennyi lépés után

ért le a blokk a kódolás miatt. Szerencsére ezt a problémát megoldotta a csoportosított lépésmódszer, mert ezzel minden lépése után kapott már jutalmat.

A táblázatból egyértelműen látszik, hogy a legjobb játékot a Saját modellel tanított ügynök játszotta. Ez azért történt, mert a Stanford modell túl összetett a mi egyszerű bemeneti adatainkhoz, így feltételezhetően túltanulás fordult elő. Azonban a leegyszerűsített modell nem mindig hozz jobb eredményt. Ezt láthatjuk a leegyszerűsített Stanford 1 és 2 modelleknél is. Habár sikerült egy jobb játékot játszania Leegysz. Stanford 1-nek mint a sima Stanford-nak átlagosan 2,6-szor kevesebb sort tudott kirakni. Fontos megvizsgálni még a tanulás folyamatát.



A fenti grafikon megmutatja, a kettő legjobban teljesítő modell tanulási sebességét. A vízszintes tengely az idő múlását mutatja, míg a függőleges tengely adatpontjait úgy kapjuk, hogy 10000 lépésenként az addig lejátszott játékokban, kiátlagoljuk a lépésenként kapott jutalmat. Ideális esetben az ügynöknek közel 2 pontot kellene visszakapnia lépésenként a tökéletes játékhoz, hiszen a 4.3.2 alfejezetben található jutalom függvény akkor maximális, ha a negatív tényezők 0-val egyenlőek.

4.1. Megjegyzés. *Valójában akkor maximális a jutalom függvény, amikor 4 sort tűntet el egyszerre az ügynök, de ezt a lehetőségett most figyelmen kívűl hagyjuk.*

Láthatjuk, hogy a Stanford modell megközelítése ami a Tetris oszlopokként tárolt információján alapszik valóban a tanulás elején jobb eredményeket ért el, azonban a modell

komplexitása miatt elkezdett romlani hosszútávon. A Saját modell ezzel ellentétben, folyamatos javulást mutatott hosszútávon is annak ellenére, hogy az elején lassabban érjük el a javuló eredményeket.

4.4.1. Javítási lehetőségek

Minden gépi tanulás végén felmerül a kérdés, hogy lehet-e javítani az eredményeken és ez nálunk sincsen másképpen. Az első egyértelmű javítási lehetőség az időben hosszabb tanítás. Ha konvergens a modellünk akkor biztosan jobb eredményeket tudunk elérni a hosszabb tanítással. Egy másik lehetséges javítás a modellekben rejlik. Nehéz optimális modellt találni ilyen feladatokra, viszont lehetséges, hogy ha ötvöznénk a Stanford modell információ kezelési ötletét és a Saját modell egyszerűbb felépítését, akkor jobb eredményeket érhetnénk el.

Egy másik lehetőség a játék megváltoztatása. Erre nem túl sok opciónk van viszont kettő változtatás lehet javítana az eredményeken. Az első a tetrominok véletlenszerű generálása. A 4.1.1 alfejezetben leírt tarisznyás megoldás segítségével az első hét tetromino mindegyikéből 1-1 darab jönne, nem pedig véletlen mennyiségek a formákból. Ez csökkentené a a véletlen input hatását és a játék korai szakaszában lévő állapotok számát is. A másik javítási lehetőség a soron következő elem megjelenítése lenne. Ez valószínűleg csak a sokkal hosszabb távú tanulásnál nyújtana fejlődést mivel ezzel összeségében több állapottal rendelkezne a környezet, cserébe az ügynök kombinálni tudná az aktuális és a következő lépést.

Habár nem sikerült javítani a [1] cikkben található eredményeken, sikerült felépíteni egy Tetris játékot és egy arra épülő gépi tanulási hálózatot, amely képes játszani a játékkal.

Irodalomjegyzék

[1] Matt Stevens, Sabeek Pradhan. *Playing Tetris with Deep Reinforcement Learning* http://cs231n.stanford.edu/reports/2016/pdfs/121_Report.pdf

[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin Riedmiller. *Playing Atari with Deep Reinforcement Learning* https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf

[3] Xintian Han. *A Mathematical Introduction to Reinforcement Learning* https://cims.nyu.edu/~donev/Teaching/WrittenOral/Projects/XintianHan-WrittenAndOral.pdf

[4] Richard S. Sutton, Andrew G. Barto *Reinforcement Learning: An Introduction* A Bradford Book, The MIT Press, Cambridge, Massachusetts, London, England 2014, 2015

[5] Ankit Choudhary *A Hands-On Introduction to Deep Q-Learning using OpenAI Gym in Python* https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/

[6] *Deep Reinforcement Learning. Introduction. Deep Q Network (DQN) algorithm.* https://markus-x-buchholz.medium.com/deep-reinforcement-learning-introduction-deep-q-network-dqn-algorithm-fb74bf4d6862

[7] Tom M. Mitchell *Machine Learning* McGraw-Hill Science/Engineering/Math; 1997.

[8] Csaba Szepesvári *Algorithms for Reinforcement Learning* Morgan & Claypool Publishers 2009. (Last update: March 12, 2019.)

[9] Altrichter Márta, Horváth Gábor, Pataki Béla, Strausz György, Takács Gábor, Valyon József. *Neurális hálózatok* Hungarian Edition Panem Könyvkiadó Kft., Budapest 2006.

[10] Timur Bakibayev *How to write Tetris in Python* https://levelup.gitconnected.com/writing-tetris-in-python-2a16bddb5318

[11] Yiyuan Lee *Tetris AI – The (Near) Perfect Bot* https://codemyroad.wordpress.com/2013/04/14/tetris-ai-the-near-perfect-player/

[12] Ian Goodfellow,Yoshua Bengio, Aaron Courville *Deep Learning* The MIT Press, Cambridge, Massachusetts, London, England 2006 http://www.deeplearningbook.org

[13] Anh H. Reynolds *Convolutional Neural Networks (CNNs)* https://anhreynolds.com/blogs/cnn.html

[14] Reinforcement learning agents documentation https://www.mathworks.com/help/reinforcement-learning/ug/create-agents-for-reinforcement-learning.html

[15] https://www.researchgate.net/figure/Mathematical-model-of-artificial-neuron_fig1_320270458

[16] Chi-Feng Wang *The vanishing gradient problem* https://towardsdatascience.com/the-vanishing-gradient-problem-69bf08b15484

[17] SAUMYAB271 *Getting started with Deep Learning? Here's a quick guide explaining everything at a place!* https://www.analyticsvidhya.com/blog/2021/04/getting-started-with-deep-learning-heres-a-quick-guide-explaining-everything-at-a-place/

[18] Brandon Semilla *Tetromino* https://www.npmjs.com/package/tetromino