

Soproni Egyetem

Simonyi Károly Műszaki, Faanyagtudományi és Művészeti Kar

Informatikai és Gazdasági Intézet

Adataalapú döntéshozatal gépi tanulás eszközökkel tőzsdei adatokon

Témavezető:

Dr. Pödör Zoltán

egyetemi docens

A szakdolgozat készítője:

Csanaki Richárd

IV.évf. gazdaságinformatikus BSc hallgató

Tartalomjegyzék

Tartalomjegyzék.....	2
Ábrajegyzék.....	3
Absztrakt	7
Abstract.....	8
1. Bevezetés	9
2. Motiváció, célkitűzések.....	11
2.1 A probléma bemutatása	12
2.2 Motiváció és célkitűzés	13
3. Irodalmi áttekintés, a gépi tanulás elméleti háttere	14
3.1 A gépi tanulás elméleti háttere	16
3.2 Gépi tanulás megközelítések	17
3.3 Visszacsatolásos tanulás	18
3.4 További elméleti megfontolások	27
4. Feladat megoldása	28
4.1 Használt eszközök	28
4.2 Adatok összegyűjtése, előkészítése elemzésre	31
4.3 Modell architektúra kialakítása és agent funkciók implementálása	36
4.4 Modell betanítás és a betanított modell mentése	44
4.5 Betanított modell tesztelése, eredmények összefoglalása	51
5. Összefoglalás, kitekintés	54
Irodalomjegyzék, hivatkozások.....	56

Ábrajegyzék

1. Ábra: Extended BI process (forrás: saját)	11
2. Ábra: Dueling Deep Q network felépítése (1) [27]	20
3. Ábra: Dueling Deep Q network felépítése (2) [28]	20
4. Ábra: Deep neural network felépítése [29]	21
5. Ábra: Egy neuron bemenetei és kimenete [30]	22
6. Ábra: Gradient descent módszer vizualizálása [31]	23
7. Ábra: Fully connected layer [32]	24
8. ábra: CNN dimenzió csökkentés [33]	25
9. ábra: Konvolúció művelete mindkét irányban [34]	25
10. ábra: Recurrent réteg az irányított gráfos visszacsatolás művelettel [32]	26
11. ábra: LSTM sejt felépítése [37]	27
12. ábra: Google Colab felület "magic" paranccsal (forrás: saját)	29
13. ábra: Az adatbegyűjtő modul fő ciklusa (forrás: saját)	31
14. ábra: Adatkombináció (forrás: saját)	32
15. ábra: Részlet a kombinált CSV fájlból (forrás: saját)	33
16. ábra: Egyedi és kombinált forrás adatok a felhőben (forrás: saját)	33
17. ábra: Kiválaszthatjuk, hogy kombinált adathalmazzal dolgozunk vagy pedig egyedi részzel (forrás: saját)	34
18. ábra: Az adatok kezeléséhez szükséges könyvtárak importálása (forrás: saját)	34
19. ábra: Adathalmaz betöltése, felszeletelése train és test halmazokra (forrás: saját)	35
20. ábra: Adatok normalizálása (forrás: saját)	35
21. ábra: A szükséges könyvtárak importálása (forrás: saját)	36
22. ábra: Adam optimalizációs algoritmus összehasonlítása más algoritmusokkal [44]	37
23. ábra: A modell architektúra definiálása (forrás: saját)	37
24. ábra: Alapértelmezett call függvény felülírása és custom advantage metódus (forrás: saját)	38
25. ábra: ReplayBuffer osztály részlete (forrás: saját)	39
26. ábra: Az Agent osztály konstruktorának paraméterei (forrás: saját)	40
27. ábra: Agent osztály konstruktor, részlet (forrás: saját)	40
28. ábra: store_transition metódus (forrás: saját)	41
29. ábra: choose_action metódus részlete (forrás: saját)	41

30. ábra: choose_action függvény train ágának modell viselkedés leírója (forrás: saját) ...	41
31. ábra: learn metódus (forrás: saját)	42
32. ábra: Reward függvény definíciója (forrás: saját)	43
33. ábra: save_model és load_model metódusok (forrás: saját)	43
34. ábra: Hiperparaméter konfiguráció [18]	44
35. ábra: Agent objektum létrehozása (forrás: saját)	44
36. ábra: Helper változók és a transaction DataFrame (forrás: saját)	44
37. ábra: Konvencionális batch-ek feldolgozása a fő ciklusban (forrás: saját)	45
38. ábra: LSTM által érthető háromdimenziós adatszerkezet [45]	46
39. ábra: Sell művelet részlet kódban (forrás: saját)	47
40. ábra: Sell művelet előzőleg talált eladási művelet függvényében (forrás: saját)	48
41. ábra: Tranzakció dictionary hozzáadása DataFrame-hez (forrás: saját)	48
42. ábra: A betanítási fő ciklus vége (forrás: saját)	49
43. ábra: A betanított modell elmentése (forrás: saját)	50
44. ábra: SavedModel formátumban lementett neurális háló (forrás: saját)	50
45. ábra: Betanított modell betöltése memóriába (forrás: saját)	51
46. ábra: Tesztelési agent létrehozása (forrás: saját)	51
47. ábra: A különböző megközelítésekkel betanított modellek eredményei (forrás: saját)	52

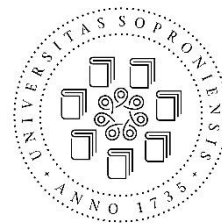


SOPRONI EGYETEM

Simonyi Károly Műszaki, Faanyagtudományi és
Művészeti Kar

H-9401 Sopron, Bajcsy-Zs. u. 4. Pf.: 132.

Tel: +36 (99) 518-101 Fax: +36 (99) 518-259



NYILATKOZAT

Alulírott **Csanaki Richárd** (neptun kód: **KZEADR**) jelen nyilatkozat aláírásával kijelentem, hogy az **Adatalapú döntéshozatal gépi tanulás eszközökkel tőzsdei adatokon** című

szakdolgozat/diplomamunka

(a továbbiakban: dolgozat) **önálló munkám**, a dolgozat készítése során betartottam a szerzői jogról szóló 1999. évi LXXVI. tv. szabályait, különösen a hivatkozások és idézések tekintetében.

Hivatkozások és idézések szabályai:

Az 1999. évi LXXVI. tv. a szerzői jogról 34. § (1) és 36. § (1) első két mondata.)

Kijelentem továbbá, hogy a dolgozat készítése során az önálló munka kitétel tekintetében a konzulens, illetve a feladatot kiadó oktatót **nem tévesztettem meg.**

Jelen nyilatkozat aláírásával tudomásul veszem, hogy amennyiben bizonyítható, hogy a dolgozatot **nem magam készítettem**, vagy a dolgozattal kapcsolatban szerzői jogsértés ténye merül fel, a Soproni Egyetem **megtagadja a dolgozat befogadását és ellenem fegyelmi eljárást indíthat.**

A dolgozat befogadásának megtagadása és a fegyelmi eljárás indítása nem érinti a szerzői jogsértés miatti egyéb (polgári jogi, szabálysértési jogi, büntetőjogi) jogkövetkezményeket.

Sopron, 2020. november 22.

.....
hallgató

Soproni Egyetem, Simonyi Károly Műszaki, Faanyagtudományi és Művészeti Kar
Informatikai és Gazdasági Intézet
9400 Sopron, Bajcsy-Zs. u. 4.

SZAKDOLGOZAT FELADAT

Szakdolgozatot készítő neve:	Csanaki Richárd , gazdaságinformatikus BSc hallgató
A szakdolgozatot készítő Neptun kódja:	KZEADR
Szakdolgozat címe:	Adataalapú döntéshozatal gépi tanulás eszközökkel tőzsdei adatokon
Intézeti konzulens(ek):	Dr. Pödör Zoltán , egyetemi docens
A dolgozat kódja	SKK-INGA-7-2020-SZ

Elvégzendő feladatok

1. Szakirodalmi feldolgozás: gépi tanulás, mesterséges intelligencia lehetőségek tőzsdei adatokon
2. Adatgyűjtés: részvény és opciós ügyletek árfolyam adatainak összegyűjtése
3. Gépi tanulás modell architektúrájának kialakítása
4. Bemeneti adatok előkészítése, előfeldolgozás
5. Modellalkotás tanuló-validáló halmazokkal, modell finomítása
6. Modell validálás, eredmények feldolgozása
7. Modell alkalmazhatóságának vizsgálata más domain területeken

Beadási határidő: 2020. december 04. 12:00

Sopron, 2020. 09. 20.


Prof. Dr. Magoss Endre
dékán


Dr. Bednárk Éva
intézetigazgató



Absztrakt

A Szakdolgozatban bemutatjuk a gépi tanulás rövid elméleti hátterét és részletesen foglalkozunk a tudományág egy részterületével, a visszacsatolós tanulással. Ennek során egy agent-et bízunk meg azzal, hogy egy Markov láncként formalizált környezetben hozzon döntéseket, miközben egy előre definiált, domain specifikus jutalom függvényt hosszú távon maximalizál. A jutalom függvény kimeneteivel tudjuk értékelni az ún. Q-függvényt, amely a visszacsatolós tanulásban használatos policy (π) függvény egy fajtája. Paraméterei a vizsgált környezet aktuális állapota és az aktuálisan végrehajtandó lépés, így definiálva az agent számára, hogy adott szituációban, állapotban, milyen lépést tegyen. A Q-függvényeket használó Q-learning módszer alapvetően egy modell nélküli tanulási megközelítés, azonban a dolgozatban ennek egy továbbfejlesztett verziójával dolgozunk, amelyben kettő neurális háló párhuzamosan tanul egymástól, a kettő modell közötti kommunikációt biztosító agent segítségével. A módszert egy gyakorlati feladatban implementáltuk, ahol két hat rétegű Long Short-Term Memory architektúrájú neurális háló tanul egymástól azért, hogy meghatározzanak egy optimális értékpapír befektetési stratégiát a bemeneti, 2 attribútumból (zárási árfolyam, napi kereskedett volumen) álló adathalmaz alapján. A gyakorlati feladatban több megközelítést is kipróbáltunk: egyéni részvényekkel való kereskedés, index fund részvényekkel való kereskedés és egy szektorba tartozó részvényekkel történő kereskedés azért, hogy egy modellt tudjunk alkalmazni több részvénnel lefolytatott ügyletekhez. Az egyéni részvények esetén stabil profitrátát értünk el, míg azon adathalmazokon, amelyek több részvényt tartalmaztak, volatilis hozamokat tapasztaltunk.

Abstract

In the thesis we present the theoretical background of machine learning and dive deep into the field of reinforcement learning during which an agent is instructed to carry out decisions in an environment formalized as a Markov decision process in order to maximize a predefined, domain specific reward function. The outputs of this reward function serve as a type of assessment of a certain Q-function which is a twist on the classic policy (π) function commonly utilized in reinforcement learning applications, has multiple parameters including the current state of the examined environment and the designated next action that ultimately defines the behavior of the agent in a given situation. The Q-learning process that enables Q-functions is fundamentally a model-free approach, nevertheless in the thesis we develop an improved version of this as we prepare two neural networks that simultaneously learn from each other through a line of communication established by the agent. We then apply this method to a practical problem where architecturally two six-layered Long Short-Term Memory neural networks learn from each other to define an optimal securities investment strategy based on an input dataset containing two attributes, the daily close prices and daily trading volumes. In the practical implementation we experiment with numerous trading strategies such as working with individual papers including index fund stocks and finally a curated list of tickers belonging to a specific sector in order to develop a model that generalizes well across various use cases involving more than one stock. With individual stocks we achieved a stable rate of profit whereas with the datasets combining multiple stocks we experienced volatile results.

1. Bevezetés

Napjaink legértékesebb erőforrása már nem az olaj, hanem az adat [1]. Digitalizált világunkban minden tevékenységünk adatot generál (például lokációs, kép, hang, szöveg, egészségügyi, termelékenységi adatok, stb.) és rendelkezünk azzal az infrastrukturális háttérrel, hogy mindezen adatokat hosszútávon le is tároljuk. Az adatok tárolása önmagában még nem termel semmiféle hasznot, azonban tárolt adataink potenciálisan értékes információt rejthetnek. Információt életvitelünkről, egészségi állapotunkról, történelmünkről, vállalatunk működéséről, folyamataink hatékonyságáról. Ahhoz, hogy ebből az adattengerből mindezen információkat ki tudjuk nyerni, olyan alapvetően informatikai megközelítésekre, módszertanokra van szükségünk, melyek segítenek minket célunk elérésében. Az így megszerzett információk újdonságokkal, nem várt, nem triviális aspektusokkal szolgálhatnak a számunkra, melyek segítségével például folyamataink költségeit csökkenthetjük, hatékonyságukat pedig növelhetjük.

Ez az adattenger elárasztotta a pénzügyi világ egykor izolált szigeteit, a tőzsdéket is. Ezek a kereskedelmi csomópontok mára magas fokon digitalizáltak, amit kiválóan szemléltet az a megfigyelés, hogy a tőzsdei kereskedés volumenének legalább 80 százalékát számítógépek által vezérelt algoritmikusok végzik az európai és amerikai piacokon [2]. Specializált kereskedő gépek tranzakciók milliárdjait hajtják végre minden nap, a világ összes fejlett piacán, emberi beavatkozás nélkül, automatizált módon. A számítógépek által vezérelt piaci ökoszisztéma két csoportra bontható. Az első kategória a koncepcionálisan egyszerű algoritmikus kereskedési megoldásokat tartalmazza, a második pedig a mesterséges intelligencia alapú megközelítések gyűjteménye. Az algoritmikus kereskedés a számítógépeknek az emberhez viszonyított felsőbbrendű számítási kapacitásait és sebességét használja ki a tőzsdei tranzakciók lebonyolításához. Itt fontos tényező továbbá a tőzsdétől való távolság, ugyanis nem mindegy, hogy az akár ezredmásodperceken belül változó árfolyam információnak mekkora utat kell megtennie, hogy elérje azt a számítógépet, amely az aktuális ár függvényében, egy ember által előre megalkotott algoritmus szerint veszi fel vagy semmisíti meg adott ügylethez kapcsolódó pozícióját. Ez az ember által létrehozott és implementált algoritmus időben önmagától nem változik, felülírásához emberi beavatkozás szükséges. Az ilyen algoritmusok a való életben a konvencionális számítógépes programoknak felelnek meg.

Az algoritmikus kereskedés mellett jelentek meg a gépi tanulásra épülő rendszerek, melyek fő differenciáló tulajdonsága az emberi beavatkozás nélküli tanulás és döntéshozatali képesség. A manapság használt mesterséges intelligencia modellek matematikai, elméleti háttere már az 1950-es években létezett [3], azonban elterjedésüket alapvetően gátolta a kor számítási kapacitásainak hiányos mivolta és az ebből adódó kísérleti, tapasztalati háttér szűkössége. Ez napjainkban már egyáltalán nem jelent problémát, az olcsó és hatékony számítási komponensek elterjedésével elérkezett a mesterséges intelligencia demokratizálódásának kora [4]. Népszerű könyvtárak tömegei, nyílt forráskódú kódbázisok, ingyenes kurzusok, fejlesztő felületek és az egész szakterület alapját képező akadémiai, tudományos kutatások állnak rendelkezésünkre. Természetesen ez az információ áradat egyfajta újszerű problémát is megteremtett: a hiteles és megbízható források megtalálása

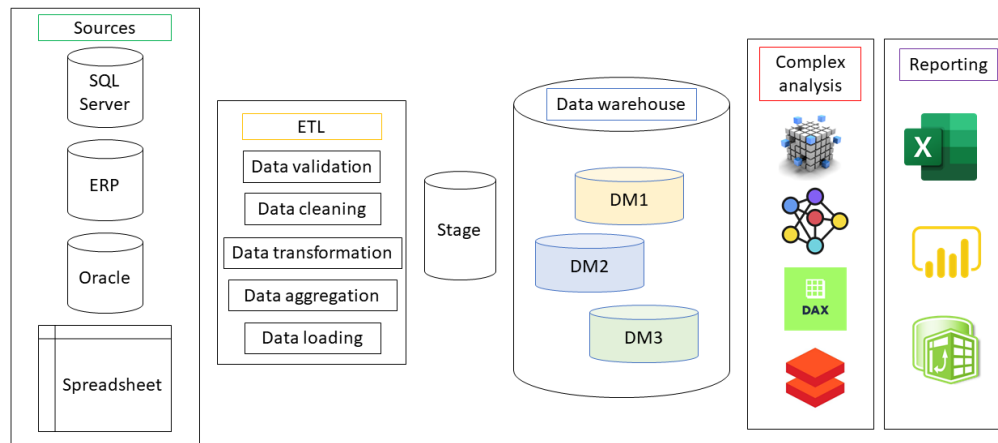
nehézkessé vált. Egy jó megoldást jelenthet a szakmában elismert kutatók publikációinak követése és az ezekben található referenciák felkeresése. Így kialakíthatunk magunknak egy hírszerzési és tájékoztató pipeline-t, amely automatizált módon lát el minket a tudományág legfrissebb fejleményeivel, híreivel. Erre pedig nagy szükségünk van, ugyanis forradalmian új gépi tanulás modell architektúrák napi szinten bukkannak fel, mindenféle kritikai átvilágítás nélkül. Ez leginkább az Euró-atlanti tudományos közösséget érinti, amely hatalmas nyomás alatt van a szcénát jelenleg uraló kínai és orosz kutatók által [5].

A mesterséges intelligencia megkerülhetetlenül kritikus témakör, amely már ma technológiai és erkölcsi dilemmák garmadáját hozta a világra. A következő évtizedeket a gépi tanulás alapú algoritmusok által automatizált munkahelyek átalakulása, a hadiipari alkalmazási lehetőségek és a technológiában rejlő potenciál vagy akár az ezekkel való visszaélés fogja meghatározni [6]. A dolgozatban bemutatjuk a gépi tanulás elméleti hátterét és megoldunk egy gyakorlati problémát, amely már régóta foglalkoztatott. Ezen gyakorlati példán keresztül megismerhetjük egy konvencionális machine learning pipeline (folyamat orientált teljes gépi tanulás megoldás) létrehozásának és használatának folyamatait, kezdve az adat összegyűjtéstől és előfeldolgozástól a modell kialakításán keresztül a betanított modell teszteléséig, értékeléséig. Egy gépi tanulás pipeline kialakításának menete mára sztenderdnek mondható, még akkor is, hogyha ezen folyamat nem lokális, hanem felhős környezetben kerül implementációra. A neurális hálókkal kapcsolatos tapasztalatok azt mutatják, hogy érdekes módon a pipeline kialakításának talán legnehezebb része nem is a modell létrehozása vagy a modell architektúrális jellegének meghatározása, hanem mindazon input adatok összegyűjtése, felosztása és előfeldolgozása, amelyek segítségével a modellt betaníthatjuk. Az adattengerek korában nincs hiány forrás adatokban, azonban ezek közül kiszűrni a ténylegesen használható és megbízható adathalmazokat nem egyértelmű.

A Szakdolgozat célja a gépi tanulás megoldások elméleti hátterének, kialakításának, előnyeinek, hátrányainak, valamint jelentőségének bemutatása, továbbá egy, a mesterséges intelligencia területén megoldott feladat megoldási, megvalósítási lépéseinek prezentálása. Az elméleti fázisban részletesen tárgyaljuk a gépi tanulás főbb területeit, különös hangsúlyt helyezve a visszacsatolós tanulásra és a mögötte lévő matematikai formalizálásokra. A visszacsatolós tanulást kombináljuk az alapvetően modell nélküli Q-learning módszertannal, amelynek egy továbbfejlesztett verzióját valósítjuk meg a gyakorlati részben, ahol párhuzamosan két neurális háló tanul egymástól egy általunk létrehozott agent segítségével. A megoldást egy pénzügyi környezetben helyeztük el, ahol egy optimális befektetési stratégia kialakítása a cél. Több megközelítéssel is kísérleteztünk, egyéni részvényeket és komplett szektorok papírjait is vizsgáltuk. Az egyéni részvények esetén a teszt adathalmazokon stabil hozamokat értünk el, míg a kombinált, szektoros módszerrel volatilis eredményeket tapasztaltunk.

2. Motiváció, célkitűzések

A mesterséges intelligencia, mint terület jól illeszkedik az eddigi Önálló laboratórium dolgozataim sorába, melyek a Business Intelligence (BI) folyamat egyes fázisait elemezték [7, 8]. A gépi tanulás vagy ezen folyamat legvégén, az elemzési részen helyezkedik el vagy pedig a Business Intelligence folyamat mintájára egy új modellt hozunk létre, amely leegyszerűsíti a vizsgálatainkat úgy, hogy kihagyjuk az adattárház építésének lépését. Ez egy felettebb összetett folyamat, így szerettük volna megérteni az elméleti hátterét és megismerni mindazon technológiákat, amelyek segítségével a gyakorlatban is meglehet valósítani egy-egy ilyen megoldást. Mind az egyetemi-intézeti projektekben, mind pedig a munkahelyemen találkoztam már teljes BI megoldásokkal, sőt, felhő alapú machine learning pipeline-okkal is. Ezen projektek során szerezhettem tapasztalatot erdészeti, meteorológiai adatok összegyűjtésében, kezelésében, illetve a professzionális szcéna révén nyerhettem bepillantást az adattudományok nagyvállalati környezetben való felhasználásába. Volt lehetőségem gépi tanulás alapú kézírás-, és hangfelismerési, illetve computer vision technológiákat alkalmazó mikrokifejezés detektáló feladatokon dolgozni. A megszerzett tapasztaltokat pedig egy számomra kedves és engem régóta foglalkoztató területen alkalmaztam: a „pénz és tőzsde csodavilágában” [19].



1. Ábra: Extended BI process (forrás: saját)

Az 1. ábrán láthatjuk, ahogy megérkeztünk a BI folyamat egyik utolsó fázisához, az összegyűjtött és előfeldolgozott adatokon történő komplex analízis lefuttatásához. Ezen elemzéseknek több sztenderd módja van: adatmodell kialakítása (OLAP kocka vagy tabuláris modell), az adatok inputként való felhasználása gépi tanulás algoritmusokban, modellekben és mindezeknek a felhő alapú implementációja (pl. Azure Data Factory és Databricks kombinációja). A kapott eredményeket pedig a Reporting lépésben az ügyfél számára érthető formában prezentáljuk.

Az értékpapír piac azért is vonzó terület, mert egy rendkívül kompetitív és veszélyes szubkultúra otthona. A tőzsdézésben részt vevő befektetők, spekulánsok, magánbefektetők, kockázati befektetők 90%-a elveszíti az összes befektetett pénzét [9]. Továbbá tekintve a manapság „biztonságos” befektetési lehetőségeket, szomorú következtetéseket vonhatunk le. A jegybanki alapkamat a szakdolgozat írásakor 0,60% [10], az állampapírok hozamai átlagosan 1,10-6,00 százalék között mozognak [11], mindemellett az éves maginfláció 4,7% és az élelmiszer árak az előző évhez képest 7,9 százalékkal nőttek [12]. Ezeket az adatokat és egyéb kimutatásokat összetéve (például azt, hogy 1970 óta radikálisan csökkent a középosztály részesedése az aggregált javakból [13]) azt a képet kapjuk, hogy az átlagos, egy-csatornás jövedelemből élők kezéből kifolyik a pénz és a középosztály egyre gyorsuló ütemben elszegényedik. Érdekes megfigyelés, hogy pont a technológiai fejlődés lehet az egyik fő katalizátora a társadalmi egyenlőtlenségek kialakulásának. Tehát a tőzsdén futnak össze a modern társadalmaink stabilitását fenyegető globális problémák, előidézve egy olyan hihetetlenül komplex környezetet, amelyben az egyszeri befektető szinte csak veszíteni tud. Azonban ez a káosz egy potenciális létra, csak meg kell szerezniünk azokat a kompetenciákat, amivel megtudjuk mászni.

2.1 A probléma bemutatása

A dolgozatom célja egy teljes machine learning pipeline készítése tőzsdei kapcsolódással, melynek komponensei a következők:

- Tőzsdei árfolyamindex adat begyűjtő és feldolgozó modul,
- Neurális háló modellek,
- A neurális háló modellek közötti kommunikációt és memóriakezelést lebonyolító „agent”,
- Betanítási, „train” adathalmaz betöltő modul (batch-ek kialakítása),
- Modell betanítási logika, train fő ciklus,
- Portfólió és likviditás menedzsment logika,
- Betanított modell és végrehajtott tranzakciók lementése,
- Teszt adathalmaz betöltő modul,
- Rugalmas modell tesztelési logika, teszt fő ciklus,
- Tesztelési eredményeket letároló modul.

Ez a fajta pipeline architektúra egy összetettebb variációja a klasszikus gépi tanulás folyamatoknak. A megnövekedett komplexitás nagy részéért maga a modell betanításának módszere a felelős. A 3. „Irodalmi áttekintés” szekcióban tárgyalt publikációk hatására nem egy sztenderd egy modelles betanítási folyamatot választottunk, hanem egy robusztusabb, futás közben párhuzamosan egyszerre kettő modellt tanító módszert. Az architektúra megnevezése: „Dueling Deep Q Reinforcement Learning” (DDQRL), azaz egymással versengő Q-függvény alapú visszacsatolásos mélytanulás modell. Ennek a bonyolult felépítésnek az összes fogalmával az elméleti elemzésben fogunk részletesen foglalkozni. Innen nézve megvan az a szemlélet, amely mentén betaníthatjuk azt a modellt, ami nem

más, mint egy zárt kimeneti eseményterű függvényhalmaz. Azonban a betanítási módszer mellett meg kell határoznunk egy modell architektúrát is: azt, hogy a betanítási fázis legelején egymással megegyező neurális hálók milyen rétegekből épüljenek fel. Szintén a feldolgozott publikációk nyomán esett a választás az ún. Long Short-Term Memory (LSTM) rétegekre, amelyek felettébb sikeresnek bizonyultak idősoros adatok elemzésénél [18].

2.2 Motiváció és célkitűzés

A dolgozat célja egy olyan tőzsdei adatok elemzésén alapuló intelligens környezet kialakítása, mely a bemeneti adatok alapján automatikus döntéseket hoz tőzsdei papírok eladása-vétele tekintetében a nyereség maximalizálása mellett. A teljes megvalósítandó folyamat első lépése a forrás adatok beszerzése és preprocessálása úgy, hogy az összeállítandó modellek bemeneteként fel tudjuk azokat használni. A következő lépés a modelleket és ezek funkcióit reprezentáló osztály létrehozása, mely tartalmazza maguknak a modelleknek a réteges felépítését és azokat a metódusokat, amelyekkel input adatokat tudunk nekik megadni, a betanított modellt le-, és betudjuk tölteni, illetve amellyel megkaphatjuk a modell kimeneteit. A modell architekturális tervezésekor megtudjuk határozni a kimenet értékkészletét és formátumát. Itt definiálunk egy ún. jutalom függvényt („reward function”), amelynek szerepe kritikus a nem felügyelt tanulási megoldásokban. Létre kell hoznunk azt az agent logikát, amelynek segítségével az egymás mellett párhuzamosan futó, tanuló modellek tudnak egymással kommunikálni és a közös memóriát, valamint kezelésének logikáját, amelyből mindketten dolgoznak. Ezután kezdhetünk el gondolkodni a betanítás menetén. Ehhez az kell, hogy a potenciálisan több ezer soros adathalmazunkat felbontsuk batch-ekre és ilyen egyenlő méretű adagokban találjuk a modelleknek. Itt léphetünk bele a betanító fő ciklusba, amelynek fontos komponense a portfólió és likviditás menedzser modul. Ezt majd két részre bontjuk: az első rész foglalkozni az adathalmaz legnagyobb részével, a második rész pedig a train adathalmaz legvégével. Miután implementáltuk a train logikát, az eredményeinket le is kell tárolni: a betanított modellt és a végrehajtott tranzakciókat, amelyek segítségével értékelni tudjuk a modellünk teljesítményét. Következő lépésként megalkotjuk a tesztelési keretrendszert, amelynek központi komponensei a teszt adat betöltő modul és a tesztelési fő ciklus, amelynek fontos tulajdonsága, hogy itt már a modell nem tanul semmit, hanem egyszerűen a betanítási fázisban kialakult modellünket hívjuk meg. A futás végén itt is le kell menteni a kapott eredményeket, a végrehajtott tranzakciókat.

A Szakdolgozat célja ezen gépi tanulás folyamat megtervezése, létrehozása, tesztelése és a mindezekhez szükséges technológiák felfedezése. A feladat megoldásának tárgyalása előtt részletesen elemezzük a felhasznált mesterséges intelligencia komponensek mögötti elméleti hátteret és a fejlesztés során feldolgozott, felhasznált publikációkat.

3. Irodalmi áttekintés, a gépi tanulás elméleti háttere

A bevezetőben említett információ áradat az általam kutatott szűkebb területet elkerülte. Természetesen nem volt hiány részvény árfolyam regresszióval foglalkozó kódbázisokban, azonban attól függetlenül, hogy mennyire vonzó témakör az árfolyam regresszió, meglehetősen pontatlan és egyáltalán nem robosztus eredményekkel találkoztunk. Továbbá nem feltétlenül segítség egy befektető számára, hogyha tudja egy adott részvény jövőbeni árfolyamát. Ezen ismeret függvényében meg is kell tudni hoznia a döntést, hogy jelen pillanatban mit tegyen a portfóliójával, azaz a lehetséges cselekedetek közül mit válasszon: vegyen a részvényből, adja el jelenleg tartott pozícióit vagy pedig ne tegyen semmit. A döntés felelőssége pedig nagy teher és egy hozzám hasonlóan tapasztalatlan befektető mi alapján hozhatna jó döntést? Hozhatna döntést például mindazon tudáshalmaz alapján, ami jelenleg az algoritmikus tőzsdei kereskedés elméleti gerincét adja de ezek a módszerek leginkább arbitrázs ügyletekhez készültek és nem kecsegtetnek nagy haszonnal – cserébe az ügyletnek rendelt rizikó minimális. A szakirodalom alapján meghatároztuk, hogy hogyan lehet az egyes vállalatok negyedéves pénzügyi jelentéseit értelmezni, a publikált számadatok fényében a kereskedelmi forgalomban lévő részvényeiket alul-, esetleg felülértékeltnek tekinteni [14]. Főkönyvi számlák rekordjainak egymáshoz viszonyított értékei, arányszámait alapján lehet következtetni a vállalati erőforrások felhasználásának hatékonyságára és a menedzsment sikerességére. Azonban minden egyes ilyen kalkuláció végén ott volt a gondolat, hogy egyszerűen innen kívülről nem ismerhetjük eléggé a vállalat belső működését ahhoz, hogy messzemenő következtetéseket tudjunk levonni a jövőbeli sikerességét illetően.

Így tettünk egy lépést hátrafele és definiáltuk a szándékainkat: egy olyan rendszer létrehozását szeretnénk elérni, amely néhány bemeneti attribútum segítségével képes lesz adott időpillanatban eldönteni, hogy a kiszemelt részvényt megvegye vagy ha már a tulajdonunkban van, akkor eladja, esetleg ne tegyen semmit, egyszerűen várjon a következő döntési helyzet elérékezésére. Innen látszik, hogy a rendszer kimenete egy három elemből álló zárt eseménytér, amely már elegendő támpontot nyújtott az algoritmikus kereskedés alternatíváit felfedező keresésben. Tipikusan nem regressziós megoldások elemzése során találtunk rá a nem felügyelt gépi tanulás megközelítések között a visszacsatolásos tanulásra (reinforcement learning). Ezen belül is több modell architektúra létezik, azonban a források mind arra utaltak, hogy egy bizonyos Q-learning módszer az, amely a legnagyobb hasznot érte el a teszt adathalmazokon [15, 16, 17, 18]. Ennek megfelelően az alább röviden bemutatott publikációkat dolgoztuk fel és a segítségükkel fedeztük fel a gépi tanulás elméleti hátterét.

Deep Q-trading [15] és An Application of Deep Reinforcement Learning to Algorithmic Trading [16]

Az első feldolgozott tudományos publikáció, amely ezzel a területtel foglalkozott a *Deep Q-trading* című papír, kínai kutatók tollából [6]. Itt ismerkedhettünk meg a Q-learning elméleti hátterének bevezetésével, az alapfogalmak definícióival és hogy ez a megközelítés hogyan függ össze a nem felügyelt (unsupervised) gépi tanulással. A papír kitért azon hiperparaméterekre is, amelyek a modell betanítása során állandóak maradnak. Említésre

kerül egy bizonyos kódrendszer és annak szerzője is, aki a papír mögötti programot megírta, azonban ez sehol nem volt fellelhető, így tovább kellett lépni, hogyha szerettem volna programban is implementált Q-függvényt látni és nem csak matematikai szimbólumokkal tarkított általános definíciókat. Az *An Application of Deep Reinforcement Learning to Algorithmic Trading* című papír is egy tisztán matematikai, elméleti magyarázatot közölt arról (függetlenül a címtől), hogy a Q-learning miért és hogyan lehet jó választás zárt eseményterű tőzsdei alkalmazások esetén, azonban itt sem találtunk használható kódrendszert.

Algorithmic Financial Trading with Deep Convolutional Neural Networks: Time Series to Image Conversion Approach [17]

A keresés során sokszor tévedtünk mellékvágányra és így találtunk nagyon absztrakt elképzeléseket is. Ez a papír azt a koncepciót járja körbe, hogy hogyan lehet egy egyébként digitális képfeldolgozásra kifejlesztett modell architektúrát (convolutional neural network, CNN) felhasználni idősoros tőzsdei árfolyam adatok elemzésére. Azzal a reménnyel olvastam végig, hogy hátha sikerült egy, a Q-learning-nél is hatékonyabb, esetleg egyszerűbb megoldást találni, ugyanis ilyen modellekkel már foglalkoztunk korábban, a klasszikus MNIST kézírás adathalmaz elemzése során. Tekintve a teszt adathalmazokon elért 10-12 százalékos éves hozamokat és az előző papírt, amely több tíz százalékos éves hozamokról számolt be, ez a megközelítés zsákutcának bizonyult.

Deep Reinforcement Learning for Trading [18]

Habár létező kódrendszerre ebből a papírból sem mutatott referencia, a gyakorlati munkát ezen publikáció alapján végeztem. Ez a papír konkrétan bemutatta a felhasznált modell architektúrákat, a modellekhez beállított hiperparamétereket és meghatározta a domain specifikus reward függvényt is. Az Oxfordi Egyetem kutatói által készített átfogó tanulmány a Q-learning területének minden számunkra fontos aspektusára kitér. Az irodalmi áttekintést követi a kereskedési folyamat Markov-láncként való formalizálása és a kritikus elméleti fogalmak definiálása. Ezen elméleti háttérrel a „Gépi tanulás elméleti háttérének bemutatása” szekcióban bővebben ismertetjük. A papír átfogó mű, rengeteg más stratégiát is bemutat és emellett össze is vet a DDQL módszerrel. Tárgyalja továbbá azt is, hogy miért érdemes kettő modellt párhuzamosan tanítani, szemben a konvencionális egy modelles megközelítésekkel: a tanulási folyamat stabilizálása miatt. Részletesen kitér a létrehozandó modellek réteges felépítésére, hiperparamétereinek beállítására és ami számomra a legnagyobb segítséget jelentette: definiál egy reward függvényt. A papír elolvasása előtt sehol nem találtunk arra vonatkozóan információt, hogy pénzügyi adatokkal dolgozó visszacsatolós modellek esetében, hogyan kell kinéznie egy reward függvénynek, így ennek megtalálása hatalmas előrelépést tett lehetővé. A paraméter beállítások és reward függvény meghatározása után az egyes módszerek lefuttatásának eredményeit olvashatjuk, amelyekből kiemelkedik a Q-learning teljesítménye. Ezek után már egyértelmű volt, hogy a DDQL megközelítést fogjuk implementálni.

A pénz és a tőzsde csodavilága [19]

Ez az 1990-ben megjelent könyv volt az első tőzsdés olvasmányom, ami szakmai publikációnak tekinthető, attól függetlenül, hogy lehet közelebb áll a romantikus mémoire műfajhoz. A szerző - aki büszkén vállalja, hogy spekuláns - az értékpapír piacokat saját munkahelyeként mutatja be és osztja meg az itt megszerzett tapasztalatait és az egyes lebonyolított ügyletekhez kapcsolódó anekdotáit. Ezekből a kis történetekből vezeti le a tőzsdei kereskedés szabályait és mutatja be az értékpapír piacok belső működését. A kidolgozott szabályok egyike így szól: „ne akarjunk régi árfolyamokat figyelembe venni!” A csokornyakkendős Kostolany természetesen nem sejtette, hogy napjaink legnagyobb Wall Street-i bankjai hisztórikus adatok alapján működő mesterséges intelligencia modellekkel fogják támogatni befektetési döntéseiket...

3.1 A gépi tanulás elméleti háttere

A gépi tanulás a mesterséges intelligencia tudományának egyik alfaja. Amikor mesterséges intelligenciáról beszélünk, akkor a nem élő eszközök, gépek által demonstrált intelligenciára utalunk [20]. Az intelligencia pedig nem más, mint az önálló probléma megoldó képesség. Ezzel a képességgel egyes élőlények magasabb mértékben rendelkeznek, mint egyéb társaik és napjainkig nem ismertünk olyan élettelen entitást, amely bármiféle szintű intelligenciát mutatott volna. Azonban az emberi intelligencia komponensekre való bontásával úgy tűnik, hogy tettünk egy lépést abba az irányba, hogy az élettelen, tárgyi eszközöket is felruházzuk az önálló döntéshozás képességével. Ahhoz, hogy egy eszköz erre képes legyen, rendelkeznie kell néhány alapvető tulajdonsággal: tudjon tartósan adatot tárolni, ezen adatokat képes legyen feldolgozni és a feldolgozás módszereit képes legyen önmagától alakítani a feldolgozás eredményének függvényében. Ez egy leegyszerűsített követelmény rendszer és észrevehetjük azt, hogy nem szerepel benne az öntudat, mint kritérium. Ennek az az oka, hogy még egyszerűen nem tudjuk azt, hogy mi a szerepe az öntudatnak az intelligencia meglétében és annak mértékében. Pontosan emiatt hatalmas hiba a mesterséges intelligencia fejlődése során felmerülő problémák és etikai megfontolások mellőzése: nem kell egy gépnek öntudatra ébrednie ahhoz, hogy veszélyt jelentsen a környezetére. Pusztán az önmaga által meghatározott, ironikus esetben az ember által meghatározott céljának kell konfliktusba kerülnie a humán érdekekkel (pl. erőforrásokhoz való hozzáférés) és máris egy olyan szituációban találja magát az emberiség, amelyben egy, az ő kollektív képességeit nagyságrendekkel meghaladó entitással áll szemben. Hogyha ezen entitás csak kis mértékben is de hordoz humán jellemzőket, akkor ez egy messze nem ideális állapot az emberi túlélés szempontjából. Gondoljunk csak azokra a történelmi precedensekre, amikor egy fejlettebb, intelligensebb civilizáció találkozott elmaradottabb, fejletlenebb embercsoportokkal.

A gépi tanulás definíciója

Ahhoz, hogy a gépi tanulás területével szakmailag konzisztensen tudjunk foglalkozni, definiálnunk kell. Általános definíció: a gépi tanulás az informatikai rendszerek egy olyan képességgel való felruházását jelenti, amelynek segítségével külső beavatkozás és explicit programozás nélkül, az adott rendszer automatikusan képes tanulni és fejlődni. Ezen rendszerek további kritikus tulajdonsága a tanuláshoz felhasznált adatokhoz való hozzáférés [20].

3.2 Gépi tanulás megközelítések

A gépi tanulás módszertanában több megközelítés létezik azért, mert a technológia minden egyes alkalmazása sok rétvű és egymástól merőben eltérő területeken történik. Természetesen erre nem lenne szükség akkor, hogyha sikerült volna kifejleszteni az mesterséges általános intelligencia (AGI – artificial general intelligence) képességével rendelkező rendszereket, amelyek elméletben univerzálisan bármilyen probléma körben bevezethetők lennének.

Felügyelt tanulás

A felügyelt tanulási modellek esetében a gépi tanulási megoldás egy olyan matematikai modell kialakítását kíséri meg, amelyben mind a bemeneti, mind pedig a kimeneti adatok ismertek. Ezt a fajta adathalmazt más elnevezéssel címkézett, „labeled” adathalmaznak nevezzük: a bemeneti adatok mellett ismerjük a kívánt kimeneti adatokat. Ez azért fontos, mert így van mihez egyszerűen viszonyítanunk a modell output-jait. Egyfajta iteratív függvény optimalizációs folyamaton végig érve kapjuk meg azt a matematikai modellt, amelynek segítségével megjósolhatjuk a tanulási folyamatban nem részt vevő bemeneti adatokhoz a kimeneti értékeket [21]. A modell által felállított kimeneti értékeket ekkor összevethetjük a tényleges értékekkel, ezzel meghatározva a modell jóságát. A felügyelt tanulás tipikus alkalmazásai: osztályozás és regressziós feladatok.

Nem felügyelt tanulás

A nem felügyelt tanulási modellek esetében csakis a bemeneti adatok állnak a gépi tanulás rendszer rendelkezésére. Az ilyen fajta modellek szabályszerűségeket, közös tulajdonságokat keresnek a bemeneti adathalmazban, valamiféle olyan relációt, amellyel a tanulási folyamat előtt sem a rendszer, sem pedig a fejlesztő nem rendelkezett. Ezen algoritmusok jóságának meghatározása nehézkes, azonban van egy hatalmas előnyük: sok esetben olyan kapcsolatokat fedeznek fel a meghatározott bemeneti adathalmazban, ami könnyen elkerülheti az emberi figyelmet. A módszer egyik legígéretesebb kutatási területe a gépi tanulás alapú számítógépes látás (computer vision). A computer vision technológiák mára emberfeletti pontossággal képesek detektálni rákos elváltozásokat bemeneti orvosi felvételek alapján [22]. Valamint érdemes megemlíteni a hatalmas előrelépéseket az emberi arcfelismerésben: az egész kínai társadalmat megfigyelő és osztályozó társadalmi kreditrendszer is a több, mint 200 millió kamerából álló hálózat által előállított és computer vision algoritmusok által feldolgozott képi adatokra épül. Felmerülhet, hogy ez vajon miért

a nem felügyelt tanulás területéhez tartozik. Azért, mert a vizsgált emberi arcon detektált arc karakterisztikákat definiáló landmark pontok elhelyezését egy olyan modell hajtja végre melynek kimenetét nem tudjuk standardizált módon ellenőrizni [23]. A nem felügyelt tanulás tipikus alkalmazásai: klaszterezés, sűrűség függvény analízis, anomália detektálás.

3.3 Visszacsatolós tanulás

A visszacsatolós tanulás (reinforcement learning, RL) a legkomplexebb machine learning terület, amivel eddig foglalkoztam és erre épül a dolgozatban bemutatott feladat, így ennek az elméleti háttérével részletesebben foglalkozunk. Az elv a következő: egy vagy több számítógépes ágenst (agent) megbízunk azzal, hogy egy adott területen belül hajtson végre akciókat. Ez absztrahálva úgy hangzik, hogy a környezetet, amiben az agent munkálkodik, markovi döntési folyamatként formalizáljuk és az agent számára definiálunk egy eseményteret, amely azon tevékenységeket tartalmazza, amelyek közül az agent a meghozott döntéseinek megfelelően választhat. Ezen eseménytér lehet diszkrét (egyedi opciók) vagy folytonos (valószínűségi százalék). A döntéseket egy bizonyos jutalom függvény (reward function) alapján hozza meg az agent, amit szintén a fejlesztő definiál. Ebben a reward függvényben reprezentálódnak az agent által felfedezendő terület fontos aspektusai, azaz mik azok az attribútumok, amelyeket vizsgálva jutalmazhatjuk vagy büntethetjük az agent-et. Fontos dimenzió az idő: a fejlesztő által megadott periódusonként kerül interakcióba az agent a környezettel. Ezen interakciók során, a reward függvény értelmében választ egy akciót a megadott eseménytérből. Ezt a folyamatot a tanulási fázis során ideális esetben magas minőségű adatokon sokszor megcsinálja és az eredmény az ún. irányelv (policy) lesz. A folyamat formalizálva úgy írható le, hogy veszünk egy S halmazt, amiben az egyes környezeti és agent állapotok vannak, veszünk egy A halmazt, mely tartalmazza az agent akcióit és ezek kombinációjából létrejön a π policy függvény, melynek paraméterei a pillanatnyi állapot és a végrehajtandó akció.

$$\pi : A \times S \rightarrow [0, 1]$$

Mi is pontosan a policy?

$$\pi(a, s) = \Pr(a_t = a \mid s_t = s)$$

A policy egy függvény, amely megadja annak a valószínűségét, hogy az agent végrehajtja a akciót, s állapotban. Amit még érdemes definiálnunk ahhoz, hogy tudjuk követni a következő lépéseket, az az állapot-érték (state-value) függvény és annak az egyes komponensei.

$$V_{\pi}(s) = \mathbb{E}[R] = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s\right],$$

A state-value függvény adja meg a reward (R) várható értékét az adott állapot függvényében, hogyha követjük a π policy függvényt.

$$R = \sum_{t=0}^{\infty} \gamma^t r_t,$$

A reward pedig az egyes időpillanatokban elérhető jutalmak összege, „súlyozva” a gamma diszkont rátával. A gamma diszkont ráta szerepe az, hogy a segítségével szabályozni tudjuk a múltbeli jutalmak hatását a jövőben elérhető jutalmakra. Továbbá fontos megemlíteni egy másik hiperparamétert, amelyet a modell betanítása előtt tudunk megadni. Az epsilon egy szám, amely kifejezi annak a valószínűségét, hogy adott állapotban véletlenszerűen választunk akciót vagy pedig $1 - \varepsilon$ valószínűséggel hagyjuk a modellt dönteni.

$$0 < \varepsilon < 1$$

Q-learning, Deep Q-learning és Double (Dueling) Deep Q-learning

A Q-learning a visszacsatolásos tanuláshoz egy alapvetően modell nélküli megközelítése. Segítségével részben véletlen döntésekkel, részben tanult módon megtalálható az az optimális policy függvény, amellyel maximalizálni tudjuk a reward várható értékét bármely állapotban [24]. A policy függvényt ez esetben Q -val jelöljük. Az idő előrehaladtával a megoldás választ egy akciót az éppen adott állapot függvényében és ezt a megtett akciót visszacsatolva a fejlesztő által definiált reward függvénybe frissül a Q -függvény.

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \underbrace{\left(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \right)}_{\text{new value (temporal difference target)}}$$

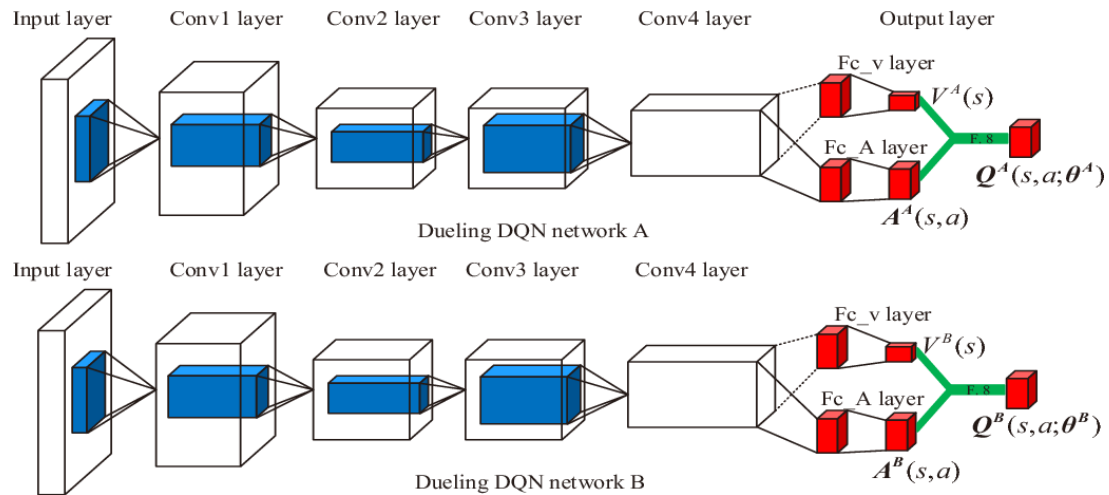
A fenti képletben láthatunk néhány eddig nem ismert komponenst: α a tanulási ráta (learning rate), a $\max Q$ elem pedig a jövőbeli akciók megtételéhez kötött maximális reward. A tanulási rátával adhatjuk meg, hogy a rendszer által újonnan látott információ milyen mértékben írja felül az eddig ismerteket. Ezt a paramétert a gyakorlatban konstansként kezeljük, értéke pedig $0 < \alpha < 1$ [25]. A $\max Q$ függvénnyel kapjuk meg a $t + 1$ időpillanatban elérhető maximális reward várható értékét. Továbbá a visszacsatolásos tanulás részénél említett γ diszkont ráta itt egy másik megvilágításba kerül, segítségével szabályozhatjuk a jövőbeli jutalmak fontosságát. Ezen komponensek kombinációjából kapjuk meg a Q -függvény frissített értékét.

A Deep Q-learning a bemutatott Q-learning módszer mélytanulás (deep learning) modellel felszerelt változata, amely a Google DeepMind Atari játékokat vizsgáló rendszerében került először implementálásra [26]. Itt egy deep learning modell kimeneteit meghatározó Q -függvényt próbálunk előállítani, azonban érdekes módon a betanítási fázisban ez a megközelítés volatilis eredményeket produkál azért, mert ennek az egy modellnek kell két funkciót ellátnia: a jövőbeni optimális lépések kalkulációit elvégezni és magukat az akciókat kiválasztani [18]. A betanítási volatilitás ellensúlyozására jött létre a Double Deep Q-learning vagy Dueling Deep Q-learning megközelítés. Itt egyszerre két modell „tanul” párhuzamosan: az egyik modellt használjuk a jövőbeli lépések értékelésére (Q^A), a másikat (Q^B) pedig a jövőbeli akciók kiválasztására.

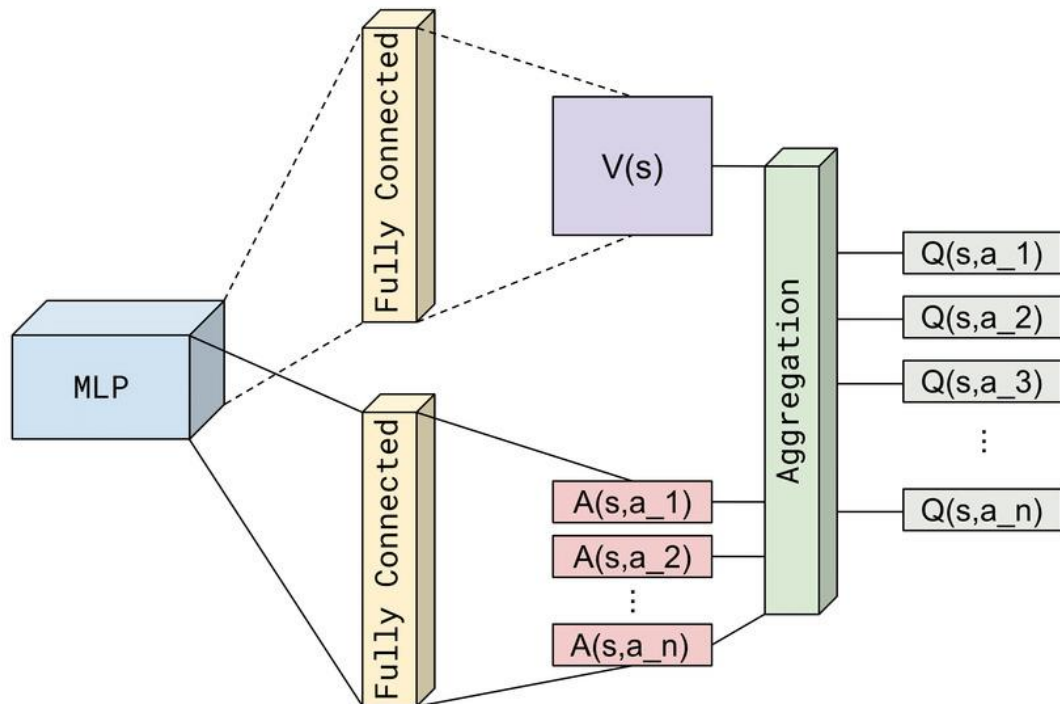
$$Q_{t+1}^A(s_t, a_t) = Q_t^A(s_t, a_t) + \alpha_t(s_t, a_t) \left(r_t + \gamma Q_t^B \left(s_{t+1}, \arg \max_a Q_t^A(s_{t+1}, a) \right) - Q_t^A(s_t, a_t) \right)$$

$$Q_{t+1}^B(s_t, a_t) = Q_t^B(s_t, a_t) + \alpha_t(s_t, a_t) \left(r_t + \gamma Q_t^A \left(s_{t+1}, \arg \max_a Q_t^B(s_{t+1}, a) \right) - Q_t^B(s_t, a_t) \right).$$

Észrevehetjük, hogy a Q^A és Q^B is felhasználja az egymás által kiszámolt Q-függvény értékeket, ami a gyakorlatban annyit jelent, hogy futás közben a két modell egymással kommunikál a visszacsatolós tanulás bemutatásával foglalkozó részben említett agent segítségével. Ennek az agent-nek a tulajdonságait és a memóriakezelését kialakítani nem egyértelmű, abszolút probléma specifikus.



2. Ábra: Dueling Deep Q network felépítése (1) [27]

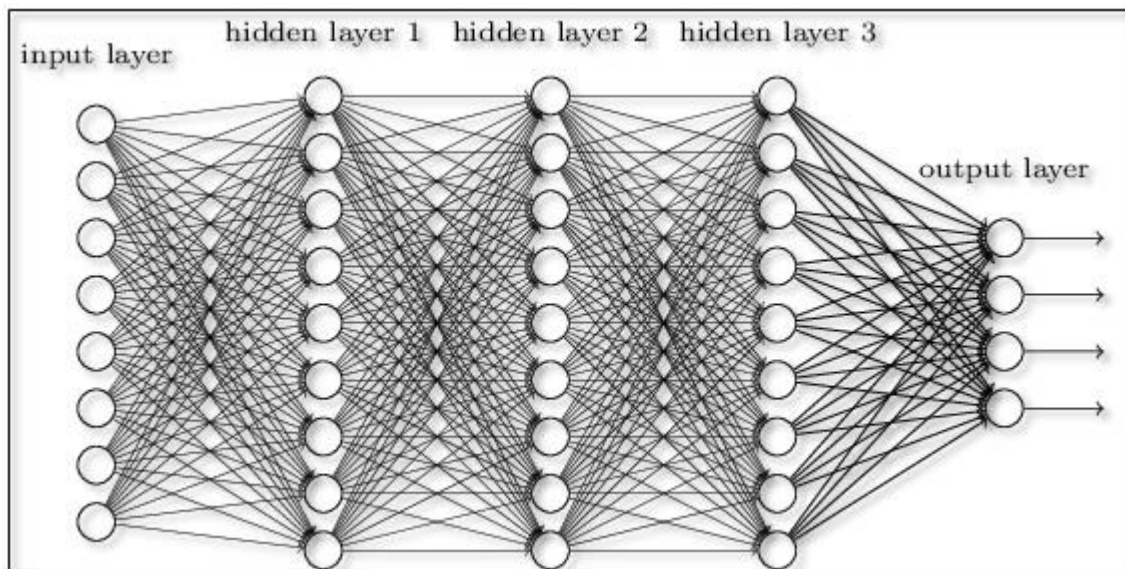


3. Ábra: Dueling Deep Q network felépítése (2) [28]

A 2. és 3. ábrán látható a Dueling Deep Q módszer felépítése, működése: két különböző modell egymással párhuzamos betanítása és egy aggregációs modulon keresztül az eredményeik kombinációja, az aktuális Q-függvény frissítése. A 3. ábrán megjelenik egy újabb elterjedt elnevezési konvenció, a Q^A modellt V, azaz value modellnek, Q^B -t pedig A-nak, azaz advantage modellnek is nevezhetjük. A futás során az advantage modell hozza meg a döntéseket a definiált eseménytérén belül.

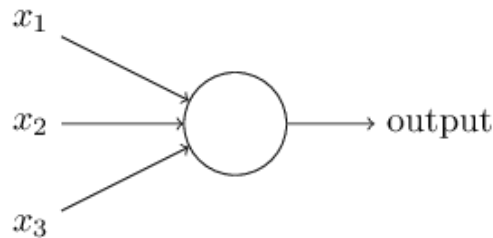
Mesterséges neurális háló (artificial neural network) modellek

Az előzőekben részletesen tárgyaltuk a machine learning pipeline betanítási folyamatának módszertani hátterét, most pedig áttérünk a módszertanon belül működő mesterséges intelligencia modell architektúrákra, azaz hogyan épülnek fel és hogyan operálnak azok a modellek, amelyeket a Dueling Deep Q-learning során egymás mellett párhuzamosan tanítunk. A mesterséges neurális hálók az emberi agy modellezett reprezentációi. A fő alkotóelemek a neuronok, perceptronok (idegsejtek) és az őket összekötő kapcsolódások, a szinapszisok. Ezeken a szinapszisokon keresztül folynak az elektromos impulzusok, a számítógépes verzió esetében pedig az adatok. A szinapszisokat éleknek is szokták nevezni. Két neuront összekötő szinapszishoz tartozik egy súly (weight), amely a tanulási folyamat során változik és a két neuron közötti kapcsolat relevanciáját hivatott kifejezni. A neuronok általában rétegekbe (layer) rendeződnek és akkor beszélünk mélytanulásról (deep neural network), hogyha a bemeneti és kimeneti rétegek között van még legalább kettő, ún. rejtett (hidden) layer. Az élek súlyain kívül a neuronokhoz is tartozik egy, a betanítás során változó tulajdonság: az aktivációs szint. Egy neuron akkor aktiválódik (az adatfolyam akkor választja a rajta keresztül vezető utat), hogyha a felé érkező adatfolyamból számolt aktivációs szint megüt egy előre definiált szintet. Ezt a szintet az aktivációs függvény (activation function) számolja a modell. Konvencionális esetben az elektromos impulzusok, adatfolyamok az egyes rétegek neuronja között közlekednek egy meghatározott irányban és végül a kimeneti rétegben aggregálódnak.



4. Ábra: Deep neural network felépítése [29]

A 4. ábrán láthatjuk a deep neurális hálózat építőköveit: a körökkel jelölt neuronokat, a neuronok közötti kapcsolatokat definiáló éleket, a rétegekbe rendezett neuronokat, a bemeneti és kimeneti réteget, valamint e kettő között elhelyezett rejtett rétegeket. És mindez hogyan működik, hogyan képes tanulni? Először is vesszük a bemeneti réteget, ahova megérkezik a betanítási adathalmaz. Minden neuronnak, így a bementi réteg neuronjainak is van egy vagy több bemenete (attól függően, hogy hol helyezkedik el a modellben) és egy kimenete. Gondolhatunk úgy is a neuronokra, mint függvény csomópontokra. Azért van szükségünk több neuronra, mert minden egyes neuron a bementi adat egy adott tulajdonságát vizsgálja. Hogyha vesszük a klasszikus kép osztályozási feladatot, amelynek célja az, hogy bementi képekről megállapítsa, hogy az adott kép kutyát vagy macskát jelenít meg, akkor a modellünk egyes rétegei ennek a képnek a különböző aspektusait vizsgálják annyi szempontból, amennyi neuront a rétegekhez definiáltunk. Az első rejtett réteg például foglalkozhat a kép általános tulajdonságaival, fényerősség, elmosódás, hasonlók. Továbbmenve a bemeneti kép részleteinek vizsgálatával, a további rétegek foglalkozhatnak a kutya vagy macska mátrix mappelt (konvolúció művelet) fejével, fülével, nyelvével és egyéb testrészeivel.



5. Ábra: Egy neuron bemenetei és kimenete [30]

Közelítve a neurális hálót felépítő egységekre, a neuronokra (5. ábra): a neuronba tartó kapcsolatok, élek fontosságát tudjuk beállítani a súlyokkal. Egy neuron akkor sül el (fire), amikor a súlyok és bemenetek súlyozott szorzatösszege átlép egy előre meghatározott értéket (threshold). A betanítási fázis végén ki fog alakulni minden potenciális bemenetre az az optimális út, amit a bemeneti adat megtesz addig, amíg át nem alakul a neurális háló kimenetévé.

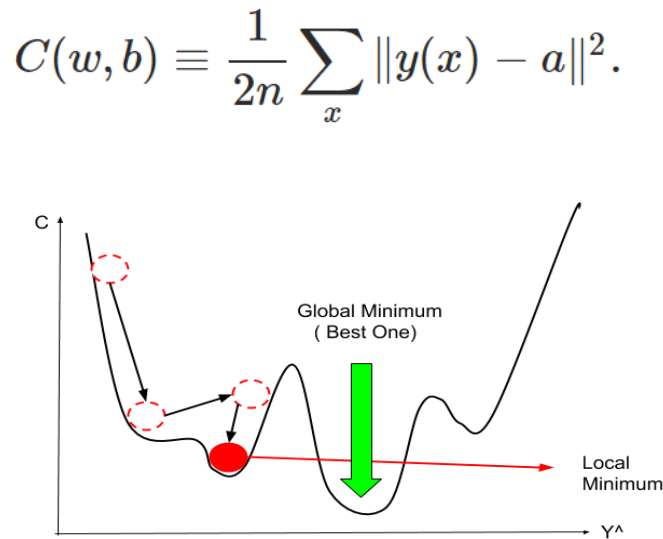
$$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{threshold} \end{cases}$$

A fenti képleten láthatjuk egy neuron elsülésének, aktiválódásának menetét és feltételeit. A betanítási folyamat során kritikus szerepet játszanak még a modell által nem tanulható paraméterek, a hiperparaméterek vagy elfogultság paraméterek (bias). A hiperparaméter a betanítási folyamat során állandó és a súlyok változása, neuronok aktivitása sem befolyásolja.

$$x_1w_1 + x_2w_2 + \dots + x_nw_n + bias$$

A hiperparaméterek egy tipikus alkalmazása az, amikor egy adott tulajdonságnak, neuronnak szeretnénk a tanulásban betöltött szerepét expliciten növelni vagy csökkenteni. A hiperparamétereknek azonban ezen kívül is még rengeteg alkalmazási lehetősége van: például a Q-learning részben említett tanulási ráta is egy hiperparaméter.

Ahhoz, hogy megállapíthassuk a modell teljesítményét, „jóságát”, találnunk kell egy olyan módszert, amellyel egységesen tudjuk értékelni az előállított kimeneteket. Ez az ún. költség függvény (cost function). C -vel jelöljük a függvényt, a paraméterek pedig w , a modellben lévő súlyok halmaza és b , a hiperparaméterek összessége; n a bemenetek száma, a a modell kimenete, amit összevetünk a tényleges, helyes eredménnyel (feltételezve, hogy ezt ismerjük, tehát felügyelt tanulásról van szó). A betanítás célja a cost függvény értékének minimalizálása. Erre is van egy módszer: gradient descent (magyarul a cost függvény globális minimumának megtalálása). Azaz mely súly és hiperparaméterek kombinációjából érhető el a legkevesebb eltérés a tényleges eredményektől.



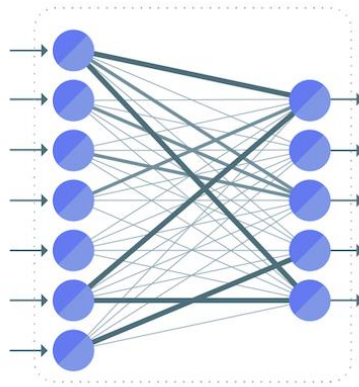
6. Ábra: Gradient descent módszer vizualizálása [31]

A 6. ábrán láthatjuk a gradient descent módszer nagy kihívását: a betanítás közben a súlyok és hiperparaméterek beállítása ne úgy sikerüljön, hogy tévesen gondolja a modell azt, hogy az adott paraméter konfigurációval megtalálta a meghatározott cost függvény globális minimumát, miközben egyszerűen egy lokális minimumot elérő kombinációt sikerült kiszámolni. Az eddigiekben tárgyaltuk mi történik a bemeneti adatokkal, amint „végigpattognak” a neurális háló rétegein, neuronjain és válnak kimenetté. A tanulás

folyamata pedig nem más, mint azon súlyok és egyéb paraméterek megtalálása, amelyek a meghatározott cost függvény értékét minimalizálják. A manapság használt megközelítés az ún. *backpropagation*, amelynek segítségével gyorsan kitudja számolni a modell, hogy az egyes súlyok és egyéb paraméterek megváltoztatása milyen hatással van a cost függvényre.

Neurális háló modell réteg architektúrák [32]

Röviden bemutatjuk a leggyakrabban használt modell réteg architektúrákat. Ez azért fontos, mert ezeken keresztül definiálhatjuk maguknak a neuronoknak a fajtáját és az egymással kialakított kapcsolataikat.



7. Ábra: Fully connected layer [32]

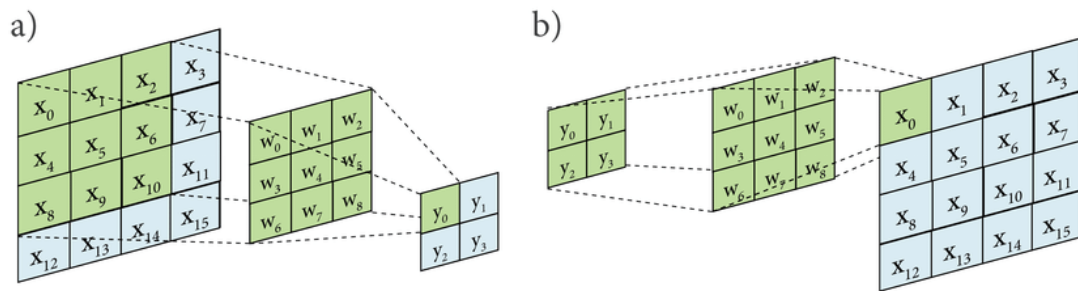
A *fully connected layer* esetében (7. ábra) a réteg minden neuronja csatlakozik az előző réteg minden neuronjához. Ez az a fajta réteg, amellyel a gépi tanulással foglalkozók először megismerkednek, ugyanis az összes alapvető fogalmat meglehet az őket tartalmazó példákon ismerni. A gyakori felhasználási módjai tipikusan a kimeneti rétegek. Komplex modelleket csak ezzel a fajta réteggel nem érdemes létrehozni, mert a sok kapcsolat számontartása, alakítása erőforrásigényes (ami egyébként bármelyik gépi tanulás architektúráról elmondható) és így szűk az alkalmazási területek spektruma.

A *convolutional layer*-ekkel tipikusan a digitális képelemző modellek esetében találkozunk, bár, ha visszaemlékszünk az Irodalmi áttekintésre, akkor láttunk tőzsdei kereskedési alkalmazási lehetőséget is. Ezen réteg meghatározó képessége az úgynevezett konvolúció műveletének végrehajtása, amely a digitális képfeldolgozás egyik leghatékonyabb formája. A gépi tanulás egy nehéz kihívása, hogy hogyan tudjuk a bemeneti adathalmazunkat olyan formátumra hozni, amely a standard modellekkel kompatibilis – ez különösen érdekes kérdés olyan összetett adatok esetén, mint a képek. Az intuitív megközelítés röviden így néz ki: a modell a skálázott bemeneti képet (vizsgálat céljától függően) a konvolúció műveletének segítségével a lehető legfeldolgozhatóbb formátumra hozza, azaz a nem lényeges képi jeleket eldobja (dimenzió csökkentés), létrehoz a 8. ábrán látható képhez hasonlót. A convolutional layer-eket tartalmazó modelleket konvolúciós neurális hálóknak (convolutional neural network - CNN) nevezzük.



8. ábra: CNN dimenzió csökkentés [33]

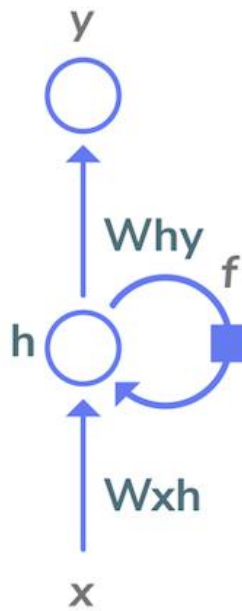
Innentől kezdődik a konvolúció művelete: a pixeleket egyenlővé tesszük egy standard skálán szereplő számokkal (pl. fekete pixel: 1, fehér pixel: 0), így megkapjuk a kép mátrix reprezentációját, amely pontosan az az adatformátum, amellyel a neurális hálók működnek. Azonban ez a mátrix még egységesített méretű képek esetén is nagy lehet – erre jelent megoldás a konvolúció művelete.



9. ábra: Konvolúció művelete mindkét irányban [34]

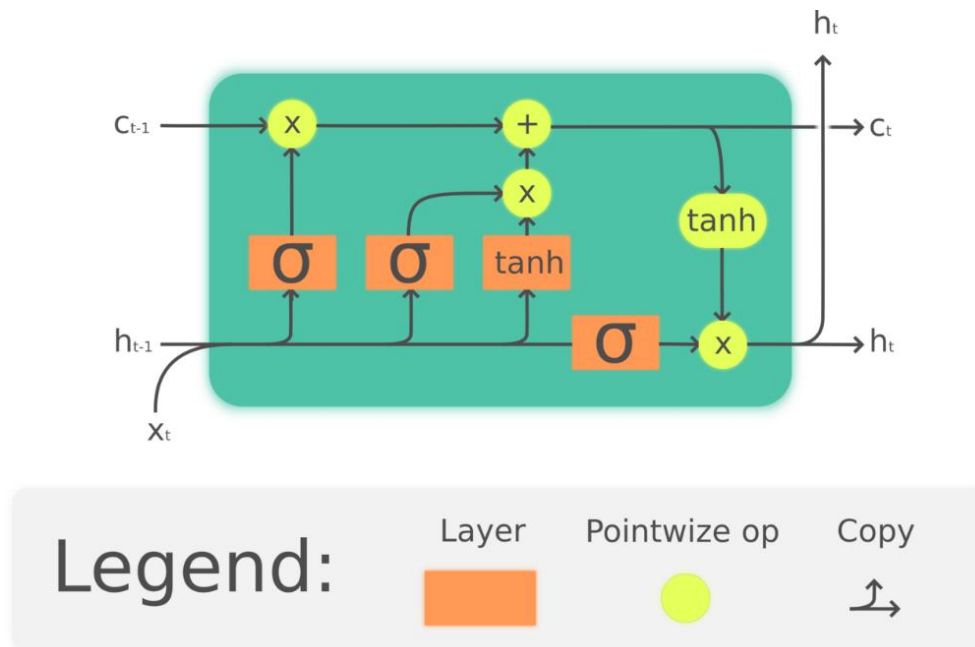
A konvolúció műveletével egy mátrix méretét csökkenthetjük úgy, hogy vesszük az eredeti mátrixot és egy állandó kernellel csúszó ablak (sliding window) jelleggel végig szorozzuk. Ahogy a 9. ábra a) képen látjuk a 4x4-es mátrixból kaptunk egy 2x2-eset, amelyből a b) képen látható módszerrel visszanyerhetjük az eredeti képet. A módszer szemléltetése azért fontos és érdekes, mert érdemes megfigyelni, hogy mekkora hangsúly van a machine learning pipeline-ban az adatok előkészítésén, előfeldolgozásán. A jó minőségű és helyesen előfeldolgozott adat a machine learning pipeline lelke.

A *recurrent layer* a kivétel az eddig tárgyalt architektúrák között. Erre a rétegre épülnek a visszacsatolós tanulás modellek. A konvencionális egyirányú *feedforward* modellekből alakultak ki, azonban ellentétben az előddel, a recurrent réteg neuronjai nem csak „előrébb” képesek irányítani az adatfolyamot, hanem a múlt adatfolyamainak egy részét vagy a modell kimenetét újra fel tudják dolgozni, azaz egyfajta memória képességgel rendelkeznek. Ebből a rekurzív képességből következik, hogy felhasználásuk során változó méretű adatcsomagokat is fel tudnak dolgozni, azaz a réteg neuronjainak memóriatartalma időben változhat. Ez alkalmassá teszi őket arra, hogy hatékonyan lehessen velük feldolgozni idősoros jellegű adathalmazokat.



10. ábra: Recurrent réteg az irányított gráfos visszacsatolás művelettel [32]

A 10. ábrán láthatjuk a recurrent réteget, mint irányított gráfot. A „visszacsatolás” kifejezés a *recurrent neurális hálónál* és a *reinforcement tanulás* tárgyalásakor is felmerült. Ez azért van, mert az angol szakirodalomban külön definiált fogalmakra nincsen magyar megfelelő, továbbá az elméleti elvet osztja a két terület: akár egy modell kimenete, akár egy agent által előidézett akció és annak eredménye is ugyanazt a logikát követi, visszakerül a tanulási folyamatba, mint többletinformáció, amelyet felhasználhatunk a jövőbeni kimenetek vagy meglépendő akciók kalkulálásában. A különbséget úgy érdemes éreztetni, hogy amíg a recurrent réteg egy ténylegesen leprogramozott függvénysorozatot takar, addig a reinforcement learning egy módszertan. A recurrent neurális háló architektúrájának egy meghatározó típusa a Long Short-Term Memory (LSTM). Ez lesz az a réteg típus, amit a gyakorlati feladatban is (részben) használunk, így érdemes felmérni a mögötte lévő elméletet. Minden igaz rá, amit a recurrent rétegnél említettünk, tipikusan az, hogy széles körben alkalmazzák idősoros adatok elemzésénél, továbbá a segítségével mélyebben megismerhetjük egy recurrent réteg neuronjainak belső felépítését. Egy klasszikus LSTM neuron egy sejtből, egy bemeneti és kimeneti kapuból, valamint egy felejtő (forget) kapuból áll [35]. A neuron sejtje a fejlesztő által meghatározott ideig képes a memóriájában tárolt információ megtartására, a kapuk pedig az adatáramlás irányítását végzik szigmoid aktiválódási függvények segítségével. A 11. ábrán látható az LSTM sejt réteges felépítése. A C-vel jelölt réteg egy lineáris függvény által definiált egyszerű adattovábbító egység, amelybe vagy beleirányítják az adatfolyamot a kapuk, vagy pedig kiolvasnak belőle. Az LSTM egységből több típus is kifejlődött [36]: „peephole” LSTM, Gated Recurrent Unit (GRU), Depth Gated Units, stb.



11. ábra: LSTM sejt felépítése [37]

3.4 További elméleti megfontolások

A gépi tanulás elméleti hátterének bemutatása során foglalkoztunk a különböző tanulási megközelítésekkel (felügyelt, nem felügyelt) és részletesen vettük a visszacsatolásos (reinforcement learning) módszert, azon belül is az egyes modell nélküli és modellel kiegészített Q-learning változatokat. Ezeket azért elemeztük mélyebben, mert erre épül a gyakorlati szekcióban bemutatott feladatom. A Q-learning megközelítéseket követte a mesterséges neurális hálók felépítésének felfedezése. Fontos megjegyezni, hogy ez a részletesebb bemutató is pusztán csak a mesterséges intelligencia területének felszínéig jutott: végtelen további alkalmazási lehetőség, modell architektúra és egyéb kutatási téma áll az előtt, aki belevág a gépi tanulás elméleti hátterének feldolgozásába. Azonban mára szinte ezen elméleti megfontolások egyikével sem kell tisztában lenni ahhoz, hogy gépi tanulás alkalmazásokat tudjunk fejleszteni. Számtalan API és előre megírt könyvtár gondoskodik arról, hogy a „barrier to entry” történelmi alacsony szintre került, ami hatalmas szerepet játszott abban, hogy ilyen széles körben és gyorsan elterjedt a gépi tanulás alkalmazása.

4. Feladat megoldása

A tervezési és fejlesztési lépések bemutatását először is a felhasznált eszközök bemutatásával kezdjük, amelyekre bárki biztosan tud támaszkodni, ha machine learning pipeline-t szeretne létrehozni, hiszen mind ingyenesen elérhető és/vagy nyílt forráskódú szoftverekről, szolgáltatásokról van szó. Említést teszünk majd a választás okairól és az egyéb alternatívákról is, ami azért fontos, mert rengeteg ezekhez hasonló funkcionalitású más termék is elérhető és aki először vág bele a gépi tanulás vizsgálatába, könnyen lehet, hogy nehezen látja át a jelenleg releváns termékkínálatot.

4.1 Használt eszközök

Python

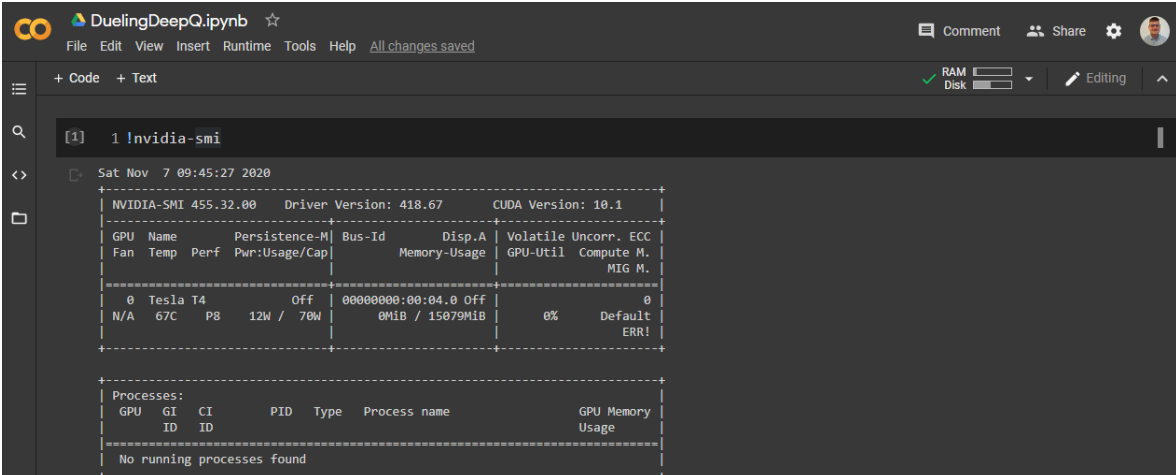
A Data Science területen az R mellett legelterjedtebb interpretált, high-level programozási nyelv. Komparatíván nagyon lassú, ugyanazon funkcionalitással rendelkező C-ben megírt kód ezerszer gyorsabban fut le, mint a Python-ban írt program. Ettől függetlenül rendkívül elterjedt, sikerének titka az, hogy rengeteg előre megírt C/C++ könyvtárat tudunk a segítségével a projektjeinkben felhasználni és így a fejlesztésre szánt időt a töredékére csökkenteni. Emiatt szokták a Python-t egyszerűen C/C++ API-nak nevezni [38].

Ebben a szekcióban érdemes tárgyalnunk azokat az előre megírt könyvtárakat, amelyeket felhasználtam a fejlesztés során. A NumPy egy többdimenziós tömb és mátrix manipulációs könyvtár, amelynek segítségével olyan formátumra hozhatjuk a többdimenziós bemeneti adathalmazainkat, amelyet képes egy gépi tanulás modell feldolgozni. A Pandas könyvtár révén van lehetőségünk több, memóriában tárolt high-performance adatszerkezettel dolgozni (DataFrame, Series), melyek megkönnyítik az adatösszegyűjtés és előfeldolgozás feladatait, továbbá a segéd metódusai adják a feladatban létrehozott portfólió menedzsment logika alapját. A scikit-learn könyvtár ismét nagy segítségünkre volt, ez esetben azonban nem a modell felépítésben, hanem a forrásadatok előfeldolgozásának normalizációs részénél. A Tensorflow (vagy TensorFlow) egy, a Google által fejlesztett „end-to-end nyílt forráskódú gépi tanulás platform” [39], amelynek segítségével létrehoztuk a gépi tanulás modell neurális háló architektúráját. Itt mindenképpen meg kell említeni, hogy a Tensorflow segítségével a fejlesztő az egyes modellek minden aspektusát képes állítani, azonban ilyen nagy felbontású személyre szabhatóságra itt nem volt szükség. Így jön a képbe a Keras könyvtár, ami a Tensorflow egy high-level API-ja, amivel könnyedén lehet modell architektúrákat definiálni, optimalizációs algoritmusokat a modellhez rendelni és a Tensorflow nagyjából minden szolgáltatását egyszerűbben kihasználni. A Tensorflow-n kívül sok más machine learning könyvtár létezik, például a Facebook által fejlesztett PyTorch. Azért választottuk a Tensorflow-t, mert ennek a szintaktikájával, szubkultúrájával voltam ismerős. Ezen kívül amúgy könnyen választhattuk volna a PyTorch-ot, ugyanis funkcionalitásban pontosan ugyanazt tudja, mint a Tensorflow. A PyDrive könyvtárat használtuk a Google Drive-al való kommunikációra, autentikációra, ugyanis az egész megoldás a felhőben fut, ez részletesen a Google Colab-ot bemutató részben kerül kifejtésre. Azért esett a választásom a PyDrive könyvtárra, mert konzisztens volt azzal az elképzeléssel, hogy hogyan lehet egy felhős szolgáltatás funkcióit kódból meghívni, azon

tapasztalatok alapján, amit a Microsoft Azure felhőjében szereztünk. Azonban később kiderült, hogy van egy egyszerűbb, grafikus megközelítés is, amivel össze lehet kötni egy Colab notebook-ot a Drive fiókkal, de így, hogy minden megvan kódszinten, könnyebben hordozhatóvá vált a megoldás. A PyDrive és google.colab könyvtárak kombinációjával a jegyzetfüzet minden funkcióját kitudtuk használni. A nyers adatokat a Yahoo Finance egy nem hivatalos Python API-jával szereztük be, a yahoo_fin könyvtárral.

Google Colaboratory

A Colaboratory (Colab) a Google ingyenes IPython notebook webes felülete. Mivel a Python egy interpretált programozási nyelv, így soronként is lehet Python kódot futtatni, ebben és a könnyebb dokumentálásban segíthet egy IPython jegyzetfüzet környezet. Ami kivételesen jó opcióvá teszi a Colab-ot, az az előre konfigurált machine learning környezete és ingyenesen használható GPU kínálata. 12 órán keresztül a rendszer ingyenesen a rendelkezésünkre állít egy Nvidia Tesla K80 videokártyát, amely az egyik legmagasabb minőségű gépi tanuláshoz használható GPU. További előnyei a felhőbeli integrációja, természetesen főleg a Google Drive szolgáltatással de egyszerűen tudunk például a GitHub kódbázisokkal is kommunikálni. Véleményem szerint ebben a szolgáltatásban manifesztálódik a felhőalapú fejlesztés legnagyobb ajándéka, a környezet virtualizáció. Amíg a saját számítógépemen napok mentek el arra, hogy egymással együttműködővé tudjam konfigurálni a saját GPU-mat és a felhasznált open source könyvtárakat (itt a GPU-ra optimalizált Tensorflow-val voltak a legnagyobb gondok), addig a Colab-on annyit kellett tennem, hogy bejelentkeztem a Google fiókomba és összekötöttem a szolgáltatást a GitHub profilommal, hogy megoldhassam a projekt verziókövetését. A létrehozott könyvtárak annak a Google fióknak a Drive alkalmazásában kerülnek lementésre, amellyel bejelentkeztünk a Colab-ra. A Colab környezetben rendelkezésünkre állnak egyedi „magic” parancsok (felkiáltó jellel kezdődő parancsok), amelyekkel felfedezhetjük a környezet infrastrukturális hátterét, személyre szabhatjuk azt, közvetlenül a Colab jegyzetfüzetből indíthatunk el Python könyvtár letöltést és még rengeteg egyéb funkciót.



```
[1] 1 !nvidia-smi
```

```
Sat Nov 7 09:45:27 2020
```

NVIDIA-SMI 455.32.00		Driver Version: 418.67		CUDA Version: 10.1	
GPU	Name	Persistence-M	Bus-Id	Disp.A	Volatile Uncorr. ECC
Fan	Temp	Perf	Pwr:Usage/Cap	Memory-Usage	GPU-Util Compute M. MIG M.
0	Tesla T4	Off	00000000:00:04:0	Off	0
N/A	67C	P8	12W / 70W	0MiB / 15079MiB	0% Default ERR!

Processes:						
GPU	GI	CI	PID	Type	Process name	GPU Memory Usage
ID	ID	ID				
No running processes found						

12. ábra: Google Colab felület "magic" parancssal (forrás: saját)

A 12. ábrán látható egy magic parancs futtatásának módja és eredménye: ebben az esetben a jegyzetfüzet mögötti GPU konfigurációt kérdeztem le.

GitHub

A Microsoft által 2018-ban felvásárolt GitHub egy, a git verziókövető rendszerre épülő kollaboratív felhőszolgáltatás. Habár a Google Colab is a „felhőben dolgozik” és automatikus mentéseket készít a fejlesztés során az eszközölt változtatásokról, ezen felül nem lehet használni verziókövetésre, azaz az egyes változatok közötti különbségek számontartására. Viszont szépen van integrálva a GitHub szolgáltatással, így magából a fejlesztői felületből elérhető az a repository (akár privát kódbázis), amelyben a kódot vezetjük [40].

Yahoo Finance

A Yahoo Finance az egykor szebb időket is megélt Yahoo! cég pénzügyi adat és hírszolgáltatása, amelynek adat lekérdező komponense mára elvileg nem támogatott, azonban még tudományos publikációk is hivatkoznak rá, mint adatforrás. Attól függetlenül, hogy a szolgáltatás hitelessége megkérdőjelezhető, a minősége egyáltalán nem. Rengeteg funkcióval és szűrési feltétellel juthatunk hozzá pénzügyi jellegű adatokhoz ingyen és még a honlapot sem kell felkeresnünk, mert több Python könyvtár segítségével használhatjuk a szolgáltatás API-ját [41].

Ezekén kívül nem használtunk semmilyen más szolgáltatást, technológiát. Maga a tény, hogy a publikációkban, papírokban felvázolt elméleti modellek és módszerek viszonylag gyorsan összerakhatók egy Python programban, futtathatók és tesztelhetők, megmutatja, hogy mennyire szoros a kooperáció a gépi tanulás elméleti hátterét kutatók és a szoftveres implementációkat készítő fejlesztők között.

4.2 Adatok összegyűjtése, előkészítése elemzésre

A machine learning pipeline az Irodalmi áttekintésben említett *Deep Reinforcement Learning for Trading* című papír alapján készült. Ebben a papírban az Oxfordi Egyetem kutatói határidős ügyletekkel kereskednek, amelyeknek, habár potenciális hozama sokszorosa az egyszerű részvényekkel való kereskedésnek, a vállalt rizikó is hasonló nagyságrendben viszonyul a konzervatív megközelítéshez. Továbbá az a megfontolás is vezérelt, hogy sokkal egyszerűbb historikus részvény árfolyamokhoz hozzáférni, valamint az egyszerű tény, hogy többet foglalkoztunk ezekkel a pénzügyi instrumentumokkal, mint az egyes opciós, határidős szerződésekkel – a jó befektető nem nyúl olyan üzletbe, amelyhez nem ért, ugyanis akkor az ügyletet nem befektetésnek hívják, hanem szerencsejátéknak. Ennek megfelelően próbálkoztunk először a korábban már használt Alpha Vantage [42] szolgáltatással, azonban a nagy mennyiségű historikus lekérdezések eredményének vizsgálatakor sok inkonzisztenciát és pontatlanságot találtunk az adatsorokban, így egy másik szolgáltatást kellett keresni. A végső választás a Yahoo Finance lett, amelyről először manuálisan, aztán pedig egy Python API-n keresztül szereztük be az adatokat. Az adatgyűjtésnek, a modell kialakításának, a betanításnak és a tesztelési módszernek is megvoltak a maga evolúciós lépései. Kezdetben a modellt egyetlen részvényen tanítottuk be és meglepő volt, hogy pár ezer sornyi adaton órákig rágódott a modell. Aztán kiderült, hogy egyszerűen végtelen ciklusba került a betanítási logika. Azonban ebben a részben úgy gondoltuk, hogy az eddigi legösszetettebb adatbegyűjtési megoldásomat mutatjuk be, amelynek alapötlete az, hogy vesszük az amerikai értékpapír piac egy szektorának 64 befolyásos vállalatának nap végi (close) részvényárfolyamait 2010-től napjainkig. A kiválasztás szempontját az S&P 500 tőzsdeindexben szereplő vállalatok adták a piaci kapitalizáció mértéke alapján. Az adatbegyűjtő modul nem a felhőben fut de semmi akadálya nem lenne a felhős implementációnak. Így viszont van egy további lépés, az összegyűjtött CSV fájlok manuális feltöltése a Google Drive felületre.

```
15 # loop through top_list and save them as individual csv-s
16 for i, ticker in enumerate(tech_list):
17     print('Working with ' + str(ticker) + ', the ' + str(i) + '. stock on the list.')
18     data = get_data(str(ticker), start_date='06/29/2010', end_date='10/06/2020', index_as_date=False, interval='1d')
19     print(data.head())
20
21     # preprocessing
22     data = data.rename(columns={'Date': 'date', 'Close': 'close', 'Volume': 'volume'})
23     data = data.drop(columns=['open', 'high', 'low', 'adjclose'])
24
25     # add id column
26     data['id'] = [item for item in range(0, len(data))]
27     print(data.head())
28
29     file_path = str(current_path) + '/../data/tech/' + str(ticker) + '_Yahoo.csv'
30
31     # write the data to file
32     data.to_csv(file_path, index=False)
```

13. ábra: Az adatbegyűjtő modul fő ciklusa (forrás: saját)

A 13. ábrán láthatjuk az adatbegyűjtő kód fő ciklusát. Itt végig futunk azon a 64 céget tartalmazó listán, melyek megfelelnek a fentebb tárgyalt kritériumoknak, küldünk egy kérést a cég nevével a Yahoo Finance szolgáltatásnak a nem hivatalos Python API-n keresztül és a request eredményét megkapjuk Pandas DataFrame formátumban, amelyet egy kicsit szerkesztünk úgy, hogy az oszlopok közül azokat, amelyekre a betanítás során nem lesz

szükség, eldobjuk, és a megmaradt attribútumokat pedig átnevezzük azért, hogy konzisztensek legyenek a Pythonban használatos *snake_case* elnevezési konvencióval. Ezután a DataFrame-hez hozzáadunk egy ID oszlopot, amelynek az a szerepe, hogy a memóriába betöltött adatszerkezetet valami alapján tudjuk rendezni, majd pedig CSV fájlba íratjuk. Amikor végig értünk minden cégen, akkor lépünk tovább a következő programba, mely naponként, majd ABC sorrendnek megfelelően kombinálja az egyedi CSV fájlokat azért, hogy egy viszonylag átlátható és könnyen hivatkozható adatformát hozzunk létre.

```
17 for i, data_file in enumerate(data_files):
18     # current dataframe
19     current_df = pd.read_csv(str(data_file))
20
21     # sort current_df
22     current_df = current_df.sort_values(by=['id'], ascending=True)
23
24     # attach current_df to data vertically
25     data = pd.concat([data, current_df], axis=0)
26
27     # sort df
28     data = data.sort_values(['date', 'ticker'], ascending=(True, True))
29
30     # write df to file
31     combined_path = str(current_path) + '/../data/tech/tech_combined.csv'
32
33     data.to_csv(combined_path, index=False)
34
35     print('Writing dataframe to file succeeded.')
```

14. ábra: Adatkombináció (forrás: saját)
















A beolvasott 64 CSV fájlt a 14. ábrán látható módon kombináljuk: létrehozunk egy üres DataFrame-t, amelyet feltöltünk a 64 fájlból beolvasott adatokkal, majd sorba rendezzük őket először az ID, aztán a date és ticker oszlopok alapján. A ticker nem más, mint a részvény szimbóluma, rövidítése, például az Apple Inc. részvényei AAPL néven keringenek a NASDAQ tőzsdén.

1	date	close	volume	ticker	id
2	6/29/2010	9.148929	1133344800	AAPL	0
3	6/29/2010	38.65	6533100	ACN	0
4	6/29/2010	26.9	22062500	ADBE	0
5	6/29/2010	28.24	4053200	ADI	0
6	6/29/2010	35.46093	9470800	ADP	0
7	6/29/2010	24.58	6242700	ADSK	0
8	6/29/2010	41.04	8389200	AKAM	0
9	6/29/2010	12.32	29442500	AMAT	0
10	6/29/2010	7.48	43861400	AMD	0
11	6/29/2010	24.26	11290400	AMX	0
12	6/29/2010	108.61	12866300	AMZN	0
13	6/29/2010	41.22	553600	ANSS	0
14	6/29/2010	20.02	1952800	APH	0
15	6/29/2010	10.6	10879400	ATVI	0
16	6/29/2010	20.97	1327400	AVGO	0
17	6/29/2010	19.02	1566500	BR	0
18	6/29/2010	25.89	120100	CHN	0
19	6/29/2010	35.5	2700	CHTR	0
20	6/29/2010	8.84	61922600	CMCSA	0
21	6/29/2010	21.3525	15199200	CRM	0

15. ábra: Részlet a kombinált CSV fájlból (forrás: saját)

A 15. ábrán látható a szépen strukturált adatszerkezetű fájl (tech_combined.csv), amely már készen áll arra, hogy feltöltsük a Google Drive felületre.

My Drive > Colab Notebooks > data ▾

Name ↑	Owner	Last modified	File size
 AAPL_tech_Yahoo.csv 	me	Oct 11, 2020 me	125 KB
 AAPL_Yahoo.csv 	me	Aug 26, 2020 me	260 KB
 AMD_tech_Yahoo.csv	me	Oct 10, 2020 me	120 KB
 AMD_Yahoo.csv 	me	Sep 30, 2020 me	179 KB
 AMZN_Yahoo.csv 	me	Sep 30, 2020 me	201 KB
 ATVI_Yahoo.csv	me	Oct 10, 2020 me	120 KB
 combined.csv 	me	Oct 7, 2020 me	7 MB
 MSFT_tech_Yahoo.csv	me	Oct 10, 2020 me	122 KB
 tech_combined.csv 	me	Oct 10, 2020 me	7 MB

16. ábra: Egyedi és kombinált forrás adatok a felhőben (forrás: saját)

A 16. ábrán láthatóak az eddig elkészített és vizsgált adathalmazok. A rendszer élesztésekor az adatbegyűjtő és előfeldolgozó modul automatizálni kell majd de felhős környezetben ez

könnyen megoldható. A kombinált fájl mérete kb. 165500 sor. A következő lépés az, hogy importáljuk ezt vagy ezeket a fájlokat a Google Colab felületen, azaz töltsük be memóriába, hogy elérhető legyen a betanítási és tesztelési modulokban. Ahhoz, hogy a megoldás dinamikus legyen, vannak a megoldásban olyan részek, amelyeket a program konfigurálására tudunk felhasználni.

```
1 # whether i want to use combined dataset or not
2 combined = True
3
4 if combined:
5     ticker = 'tech_combined'
6     data_index = 115840
7
8     # limit - minus_value: az adott ticker indexének dinamikus megtalálása
9     idx = 0 # AAPL
10    batch_size_basic = 64
11    minus_value = batch_size_basic - idx
12 else:
13    ticker = 'AMD'
14    data_index = 3968
15    minus_value = 1
```

17. ábra: Kiválaszthatjuk, hogy kombinált adathalmazzal dolgozunk vagy pedig egyedi részvénnnyel (forrás: saját)

A 17. ábrán láthatjuk azt a logikát, amelyben megadhatjuk, hogy milyen jellegű adathalmazon szeretnénk dolgozni, kombinálton vagy egyedi részvényen. Az egyedi részvény kiválasztása inkább csak a kezdeti fázisokban volt releváns, amikor az volt a cél, hogy az egész kódbázis egy koherens, működő megoldássá álljon össze. Ezután a program komplexitásának növekedésével, a szerepe egyre inkább háttérbe szorult. Ettől függetlenül érdekes módon máig az egyedi részvényeken betanított modellek teljesítettek a legjobban!

```
1 # main_keras_dueling_dqn_lstm.py
2 import pandas as pd
3 import numpy as np
4 from sklearn import preprocessing
5 import math
6 from statistics import median, mean
7
8 # connect to Drive
9 !pip install -U -q PyDrive
10 from pydrive.auth import GoogleAuth
11 from pydrive.drive import GoogleDrive
12 from google.colab import auth
13 from oauth2client.client import GoogleCredentials
14
15 auth.authenticate_user()
16 gauth = GoogleAuth()
17 gauth.credentials = GoogleCredentials.get_application_default()
18 drive = GoogleDrive(gauth)
```

18. ábra: Az adatok kezeléséhez szükséges könyvtárak importálása (forrás: saját)

A 18. ábrán látható módon töltjük be azokat a könyvtárakat, melyekre szükségünk lesz az adatmanipulációs lépések, illetve a Google Drive-val való autentikáció során.

```

43 data_link = 'f'
44 id = '1|'
45 print(id)
46
47 downloaded = drive.CreateFile({'id':id})
48
49 if combined:
50     print('...combined data chosen...')
51     data_file = 'tech_combined.csv'
52     downloaded.GetContentFile(data_file)
53
54     # iterate csv file and getting train batches
55     data = pd.read_csv(data_file)
56
57     data_train = pd.DataFrame(data[:data_index])
58     data_train = data_train.sort_values(by=['date', 'ticker'], ascending=(True, True))
59 else:
60     print('...conventional individual stock data chosen...')
61     data_file = ticker + '_Yahoo.csv'
62     downloaded.GetContentFile(data_file)
63
64     # iterate csv file and getting train batches
65     data = pd.read_csv(data_file)
66
67     # data[:1792] for VOO
68     data_train = pd.DataFrame(data[:data_index])
69     data_train = data_train.sort_values(by=['id'], ascending=True)
70

```

19. ábra: Adathalmaz betöltése, felseleltelése train és test halmazokra (forrás: saját)

```

71 # normalize train data by adding normalized columns
72 data_normalizer = preprocessing.MinMaxScaler()
73
74 close_array = np.array(data_train['close'])
75 close_resaped = close_array.reshape(-1, 1)
76
77 data_train['close_normalized'] = data_normalizer.fit_transform(close_resaped)
78
79 volume_array = np.array(data_train['volume'])
80 volume_resaped = volume_array.reshape(-1, 1)
81
82 data_train['volume_normalized'] = data_normalizer.fit_transform(volume_resaped)
83
84 # adding helper columns for usage in reward function - only using the normalized values
85 data_train['r_t'] = data_train['close_normalized'].diff().fillna(0)
86 data_train['sigma_t'] = data_train['r_t'].ewm(span=60).std().fillna(0)

```

20. ábra: Adatok normalizálása (forrás: saját)

A 19. ábrán egy konvencionális 70% train, 30% teszt felosztást elvégző kódrészletet látunk és a memóriába betöltött adatok rendezését. Ez a fajta megközelítés kritikus a megoldás egésze során, ugyanis a memóriában tárolt adatok nem feltétlenül maradnak abban az állapotban, amelyben szeretnénk, így mindig különös figyelmet kell arra fordítani, hogy rendezett halmazokkal dolgozzunk. A 20. ábrán pedig az egyik legfontosabb lépést teljesítjük: normalizáljuk a bemeneti adatainkat azért, hogy ne „tanuljon félre” a modell pusztán a bemeneti adathalmazban megtalálható lényegtelen különbségek miatt. Például attól, hogy amíg az Amazon cég részvényei 3000 dollár fölött vannak, addig a stock split utáni Apple részvény pusztán 118 dolláron áll, így az Amazon részvény árfolyama bemeneti

adatként sokkal nagyobb mértékben befolyásolja a tanuló algoritmusokat, mint az Apple részvénye, attól függetlenül, hogy a valóságban az Apple-nek nagyobb a piaci kapitalizációja, mint az Amazonnak. Ennek megfelelően azonos skálára kell hozni ezeket az árfolyamokat, melynek módját a 20. ábrán szemléltetjük. A normalizáláshoz a scikit-learn könyvtár `MinMaxScaler()` metódusát használtam, aztán a normalizált adatok bementként szolgálták további fontos oszlopok létrehozásához, amelyeknek a később definiált reward függvényben lesz szerepük.

$$x_{scaled} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

Az adatgyűjtő és előfeldolgozó szekcióban tárgyaltuk a megoldás alapját képező adatok összegyűjtésének és előfeldolgozásának folyamatát. Sikertől jó minőségű, konzisztens adathalmazokat kialakítani, amelyeket átküldve a normalizáló függvényen most már felhasználhatunk a modell(ek) betanításához és teszteléséhez.

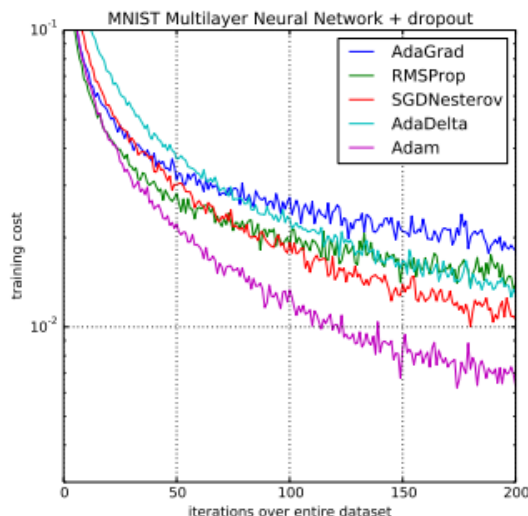
4.3 Modell architektúra kialakítása és agent funkciók implementálása

Elérkeztünk a machine learning pipeline következő fázisához, a modell felépítésének és viselkedésének kialakításához. A modell architektúráját a papír [18] alapján csináltuk, bár pontosan nem voltak meghatározva a rétegek, csak a rétegek fajtája és az activation függvények. Továbbá a modell példányok közötti kommunikációt lebonyolító agent funkciók pedig Phil Tabor kiváló YouTube tutorial-ja [43] alapján készültek.

```
1 # dueling_dqn_lstm.py
2 import tensorflow as tf
3 import tensorflow.keras as keras
4 from tensorflow.keras.optimizers import Adam
5 from tensorflow.keras.models import load_model
6 import numpy as np
```

21. ábra: A szükséges könyvtárak importálása (forrás: saját)

A 21. ábrán láthatjuk a modell kialakításához szükséges könyvtárak importálását. Használjuk majd a Tensorflow könyvtárat is közvetlenül, azonban a felépítés definiálásakor a Keras API-ra támaszkodunk. A NumPy mátrix könyvtár azért kell, mert ez az a formátum (numpy mátrix), amelybe át kell konvertálnunk a Pandas DataFrame-be betöltött forrás adatokat ahhoz, hogy a Tensorflow modell fel tudja őket dolgozni. Egyéb utility funkciókat is megoldunk a Keras könyvtárral, optimalizáló algoritmust és modell betöltési képességet rendelünk a programhoz. Az *Adam* optimalizáló algoritmust az elméleti bevezetőben említett sztochasztikus gradient descent helyett használjuk azért, mert a segítségével gyorsabban elérjük azt az állapotot, melyben optimális eredményt érünk el az adott neurális háló paraméterek beállításával.



22. ábra: Adam optimalizációs algoritmus összehasonlítása más algoritmusokkal [44]

A 22. ábrán láthatjuk azt, hogy az Adam optimalizáló algoritmussal kevesebb kört kell futnunk (technikai megfogalmazásban: kevesebb epoch-ra van szükség) az adathalmazunkon ahhoz, hogy megtaláljuk a neurális háló modellünk optimális súly-, és paraméter kombinációját.

```

8 class DuelingDeepQNetwork(keras.Model):
9     # fc = fully connected, fc_dims: number of units, neurons
10    def __init__(self, n_actions, fc1_dims, fc2_dims):
11        super(DuelingDeepQNetwork, self).__init__()
12
13        keras.backend.set_floatx('float64')
14        self.leaky_relu1 = keras.layers.LeakyReLU()
15        self.lstm1 = keras.layers.LSTM(units=64, input_shape=(64,2), return_sequences=True, dtype='float64', activation=None)
16        self.leaky_relu2 = keras.layers.LeakyReLU()
17        self.lstm2 = keras.layers.LSTM(units=64, input_shape=(64,2), return_sequences=True, dtype='float64', activation=None)
18        self.V = keras.layers.Dense(1, activation=None)
19        self.A = keras.layers.Dense(n_actions, activation=None)

```

23. ábra: A modell architektúra definiálása (forrás: saját)

A 23. ábrán egy egyáltalán nem konvencionális modell felépítési megközelítést láthatunk. A megszokott mód az, hogy létrehozunk Keras-ban egy `keras.Model` objektumot, amit valamilyen fajtának meghatározunk (pl. `Sequential`) és ehhez a `Model` objektumhoz hozzáadjuk az egyes rétegeket. Itt egy lépéssel hátrébb kezdünk, ugyanis egy teljesen egyedi modellt hozunk létre, amely nem szerepel a Keras könyvtárban. Ezért a definiált osztályunkat (`DuelingDeepQNetwork`) csak származtatjuk a `keras.Model` osztályból, amely jelezni fogja a Keras API-nak, hogy ez egy egyedileg meghatározott, nem konvencionális modell architektúra lesz. Erre azért van szükség, hogy a későbbiekben felül tudjuk írni, személyre tudjuk szabni a `keras.Model` osztály beépített függvényeit. Rátérve a tényleges modell architektúrára, a modellünk összesen 6 rétegből épül fel és kb. 50400 állítható paraméterrel rendelkezik. A rétegek között megtaláljuk a Leaky ReLU aktiváló függvény réteget is, amelyet elsőre gondolhatnánk, hogy az LSTM réteg paramétereiben kellene megadni, mint `activation` függvény de a Keras az előbbi konvenciót használja. A két LSTM réteg előtt külön-külön vesszük fel a Leaky ReLU `activation` függvényeket.

$$f(x) = \begin{cases} x & \text{if } x > 0, \\ 0.01x & \text{otherwise.} \end{cases}$$

A fenti képleten látható, hogy attól függetlenül, hogy ilyen furcsa neve van, a Leaky ReLU egy egyszerű aktiváló függvény. A Leaky ReLU réteg után jön az LSTM réteg, mely a recurrent learning funkciókkal látja el a modellt. A réteg létrehozásakor definiáljuk a tulajdonságait: 64 neuronból álljon, a bemeneti adatokat 64x2 alakú mátrixban várja, amik float64 típusúak. Miután a bemeneti adatok átértek rajta, tudjuk definiálni, hogy visszaadja-e az előállított kimeneteket, ezt True-ra állítottuk, ugyanis szükségünk van a réteg kimeneteire ahhoz, hogy tovább tudjuk őket passzolni a későbbi rétegeknek. Az utolsó kettő Dense réteg (Dense azért, mert minden neuron kapcsolódik az előző réteg minden neuronjához) a két DDQN modell objektum különböző szerepei miatt szükséges, amelyeket extenzíven tárgyaltunk az elméleti bevezetőben: az advantage modell „A” rétege adja ki a zárt eseményterű eredményhalmazunkat.

```

23     def call(self, state, training=True):
24         x = self.leaky_relu1(state)
25         x = self.lstm1(x)
26         x = self.leaky_relu2(x)
27         x = self.lstm2(x)
28
29         # define behaviour during training
30         if training == True:
31             V = self.V(x)
32             A = self.A(x)
33
34             Q = (V + (A - tf.math.reduce_mean(A, axis=1, keepdims=True)))
35
36             return Q
37
38         # during testing i only need the advantage model
39         else:
40             A = self.A(x)
41
42             return A
43
44     @tf.function
45     def advantage(self, state):
46         x = self.leaky_relu1(state)
47         x = self.lstm1(x)
48         x = self.leaky_relu2(x)
49         x = self.lstm2(x)
50         A = self.A(x)
51
52         return A

```

24. ábra: Alapértelmezett call függvény felülírása és custom advantage metódus (forrás: saját)

A 24. ábrán látható a modell objektum meghívásának két módja, az alapértelmezett `call` függvény felülírása (`override`) és egy egyedi `advantage` függvény. Ezeknek a felhasználásáról majd az `Agent` osztály elemzésekor tudunk többet mondani, egyelőre annyit érdemes megemlíteni, hogy a kód egy lineáris lefutást definiál, a bemeneti adathalmazt végig futtatjuk a modell egyes rétegein és az utolsó réteg kimenetét pedig visszaadjuk. A következő szekcióban hozzuk létre a `ReplayBuffer` osztályt. Ez teljes mértékben utility funkciókat lát el, a Q-learning elméleti hiányosságait hivatott javítani, különösképpen a DDQN azon tulajdonságát, hogy nagy mértékben függ a modell súly-, és paraméter kombinációja a bemeneti adatok függvényében, ezért szinte minden új bemenetre egy radikálisan más eredményt ad a modell, így rendkívül volatilisá válik és nem tud megtanulni semmit. A Q-learning ezen elméleti hiányosságát hivatott maga a kettős modell használata és a `ReplayBuffer` osztály által definiált átmeneti memóriatárolás megoldani.

```
54 class ReplayBuffer():
55     def __init__(self, max_size, input_shape):
56         self.mem_size = max_size
57         self.mem_cntr = 0
58
59         self.state_memory = np.zeros((self.mem_size, *input_shape),
60                                     dtype=np.float64)
61         self.new_state_memory = np.zeros((self.mem_size, *input_shape),
62                                         dtype=np.float64)
63         self.action_memory = np.zeros(self.mem_size, dtype=np.int64)
64         self.reward_memory = np.zeros(self.mem_size, dtype=np.float64)
65         self.terminal_memory = np.zeros(self.mem_size, dtype=np.bool)
66
67     def store_transition(self, state, action, reward, state_, done):
68         index = self.mem_cntr % self.mem_size
69         self.state_memory[index] = state
70         self.new_state_memory[index] = state_
71         self.action_memory[index] = action
72         self.reward_memory[index] = reward
73         self.terminal_memory[index] = done
74
75         self.mem_cntr += 1
```

25. ábra: `ReplayBuffer` osztály részlete (forrás: saját)

A 25. ábrán láthatunk egy részletet a `ReplayBuffer` osztály működéséből. Itt tudjuk megadni a modellek futása közben az agent által menedzselte memória kezelését szabályozni, az `advantage` modellnek átadott `reward`, a választott akció és az aktuális állapot (`state`) letárolásával. Ez a memória az általunk definiált periódusonként törlődik. Most következik az `Agent` osztály, amely felhasználja a `ReplayBuffer`-t, mint belső memória kezelő és irányítja a két modell futás alatti kommunikációját, valamint definiál olyan segéd metódusokat, amely segítségével például letudjuk menteni a betanított modellt.

```

89 class Agent():
90     def __init__(self, lr, gamma, n_actions, epsilon, batch_size, input_dims, epsilon_dec=1e-3, eps_end=0.01,
91                 mem_size=10000, fname='model', fc1_dims=128, fc2_dims=128, replace=100, testing=False, model=None):

```

26. ábra: Az Agent osztály konstruktorának paraméterei (forrás: saját)

Az Agent osztály paraméterei szerteágazóak (26. ábra), ugyanis az osztály széleskörű, komplex funkcionalitással rendelkezik. Az elméletben említett összes Q-learning paramétert itt tudjuk megadni, többek között a learning rate-t, a gammát, az epszilont, a batch méretet (mekkora szeletekre vágjuk fel a bemeneti adathalmazt), a bemeneti adathalmaz dimenzióit, a memória méretet (amelynek megtelésekor az agent törli a tartalmat), teszt modellt irányít-e az agent, stb.

```

124         self.action_space = np.array([-1,0,1])
125         self.gamma = gamma
126         self.epsilon = epsilon
127         self.eps_dec = epsilon_dec
128         self.eps_min = eps_end
129         self.fname = fname
130         self.replace = replace
131         self.batch_size = batch_size
132
133         self.learn_step_counter = 0
134         self.memory = ReplayBuffer(mem_size, input_dims)
135         self.q_eval = DuelingDeepQNetwork(n_actions, fc1_dims, fc2_dims)
136         self.q_next = DuelingDeepQNetwork(n_actions, fc1_dims, fc2_dims)
137
138         # set learning rate and optimizer
139         self.q_eval.compile(optimizer=Adam(learning_rate=lr),
140                             loss='mean_squared_error')
141         # just a formality, won't optimize network
142         self.q_next.compile(optimizer=Adam(learning_rate=lr),
143                             loss='mean_squared_error')
144
145         # keeping track of chosen actions
146         self.chosen_actions = []
147
148         print('Training agent created')

```

27. ábra: Agent osztály konstruktor, részlet (forrás: saját)

A 27. ábrán látható, hogy egy Agent objektumnak beállítjuk a ReplayBuffer memóriáját, létrehozuk a két modellt - `q_eval`, amely a döntéseket hozza adott állapot függvényében és `q_next`, amely a lehetséges döntéseket kalkulálja – és beállítjuk hozzájuk a paraméter listában megadott learning rate-t és az Adam optimalizáló algoritmust, költség függvénynek pedig az MSE módszert választjuk. A konstruktort két részre bontottam, a 27. ábrán a tanítási ág látható, a teszt részben az az eltérés, hogy itt az agent-et már a lementett, betanított modellel kell létrehoznunk, így itt nem új `DuelingDeepQNetwork` objektumot hozunk létre, hanem a paraméter listában átadott `model` objektumot vesszük fel. A konstruktort követi a `store_transition` metódus, amelynek segítségével eltudjuk tárolni a modell által hozott aktuális döntést és annak körülményeit.


```

151 def store_transition(self, state, action, reward, new_state, done):
152     self.memory.store_transition(state, action, reward, new_state, done)

```

28. ábra: store_transition metódus (forrás: saját)

A 28. ábrán látható a sok utility metódus közül az egyik, a store_transition, amellyel az agent eltudja tárolni a q_eval modell által hozott döntéseket és a modell által látott körülményeket. A következő metódus az agent objektum lelke, ezt a metódust hívjuk meg akkor, amikor akciót szeretnénk választani.

```

154 def choose_action(self, observation, testing=False):
155     if testing == False:
156         if np.random.random() < self.epsilon:
157             action = np.random.choice(self.action_space)
158             # converting to native Python type
159             action = action.item()
160             print('actionE:', action)
161
162             action_dicti = {'Epsilon': action}
163             self.chosen_actions.append(action_dicti)

```

29. ábra: choose_action metódus részlete (forrás: saját)

A choose_action metódusban/függvényben (29. ábra) döntünk arról, hogy a modell hozzon-e döntést vagy pedig random választunk a [-1,0,1] zárt eseménytérből. A random választást az epsilon függvény írja le és ebben a metódusban is szétválasztjuk a betanítási és tesztelési fázis közben használt funkcionalitásokat.

```

164     else:
165         # state = np.array([observation])
166         state = observation
167         actions = self.q_eval.advantage(state)
168         action = tf.math.argmax(actions, axis=1).numpy()[0]
169
170         # get index of max value inside ndarray
171         max_idx = np.where(action == np.amax(action))
172
173         action_idx = 0
174
175         # max_idx[0] is a 'list' (1D ndarray)
176
177         if len(max_idx[0]) > 1:
178             action_idx = int(np.random.choice(max_idx[0], 1))
179         else:
180             action_idx = int(max_idx[0])
181
182         if action_idx == 0:
183             action = -1
184         elif action_idx == 1:
185             action = 0
186         elif action_idx == 2:
187             action = 1
188         print('actionM:', action)
189
190         action_dicti = {'Model': action}
191         self.chosen_actions.append(action_dicti)

```

30. ábra: choose_action függvény train ágának modell viselkedés leírója (forrás: saját)

A 30. ábrán látjuk a modell általi döntéshozatal logikáját. A modell megkapja az adott állapot batch-ét, azaz a bementi batch-et végigfuttatjuk a `q_eval` döntéshozó modellen és vesszük a kimenet releváns részét (Tensorflow natív `math.argmax` függvénnyel) és kapunk egy három elemű listát. Ezen három elemű rendezett lista elemei az eseménytérhez mappelt döntési valószínűségek (pl. `[23,40,56]` \rightarrow `[-1,0,1]`). Ebből a listából kiválasztjuk a legnagyobb szám indexét (azaz a legnagyobb valószínűséggel optimális hozamot elérő döntés valószínűségét a reward függvényében) és ha ez a pozíció a nulladik, akkor az akció -1 (eladás), ha egyes, akkor az akció 0 (megtart), ha pedig kettes, akkor az akció 1 (vétel) és ezt visszaadjuk.

```

229 def learn(self):
230     if self.memory.mem_cntr < self.batch_size:
231         return
232
233     if self.learn_step_counter % self.replace == 0:
234         self.q_next.set_weights(self.q_eval.get_weights())
235
236     states, actions, rewards, states_, dones = self.memory.sample_buffer(self.batch_size)
237
238     q_pred = self.q_eval(states)
239     q_next = tf.math.reduce_max(self.q_next(states_), axis=1, keepdims=True).numpy()
240     q_target = np.copy(q_pred)
241
242     for idx, terminal in enumerate(dones):
243         if terminal:
244             q_next[idx] = 0.0
245             q_target[idx, actions[idx]] = rewards[idx] + self.gamma*q_next[idx]
246
247     self.q_eval.train_on_batch(x=states, y=q_target) # states
248
249     self.epsilon = self.epsilon - self.eps_dec if self.epsilon > self.eps_min else self.eps_min
250
251     self.learn_step_counter += 1

```

31. ábra: learn metódus (forrás: saját)

A `learn` metódusban (31. ábra) definiált viselkedés szerint tanul a modell. A döntéshozó (tanuló) `q_eval` modell paraméter értékeit átadjuk a `q_next` modellnek, ha megtelt a `ReplayBuffer` memória és átállítjuk, paramétereiket frissítjük, azaz tanítjuk a modelleket úgy, hogy a `q_next` modellel kiszámoljuk a közelítendő optimális Q-függvény értékeket és visszacsatolva a `q_eval` függvénybe frissítjük annak súlyait, paramétereit. Itt találkozunk újra a *backpropagation* módszerrel, amelynek segítségével egy optimális kimenet ismeretében (amit a `q_next` modell Q-függvény és reward függvény kalkulációinak jóvoltából ismerünk) frissíthetjük úgy a modell szerkezetet (a `q_eval` modell architektúráját), hogy azon neuronokat összekötő élek kerüljenek hangsúlyos pozícióba, amelyek bejárásával elérjük a kiszámolt optimális kimenetet, illetve azon élek pedig degradálódnak, amelyek bejárása és összekötött neuronjaik aktiválódása nem vezet el az optimális kimenethez. Azaz a modell megtanulja, hogy a bemeneti adat mely aspektusai azok, amelyek szignifikánsan befolyásolják a kimenetet és hogy hogyan kell ezen aspektusokat feldolgozó aktiválódási függvényeket és élsúlyokat alakítani ahhoz, hogy megkapjuk azt a modellt, amely képes az adott bement függvényében közelíteni a meghatározott Q-függvényt. Még egyszerűbben: a modell úgy alakítja magát, hogy a

bemeneti állapotnak megfelelően a legtöbb hozamot generálja a meghozott döntésével. Most elérkeztünk az ominózus reward függvény programbeli definiálásához. Ez az a függvény, amely domain specifikusan képes értékelni egy döntést, esetünkben elmondani, hogy adott akcióval milyen jutalomra (következményekre) számíthat a modell, hogyha egy lépést meg akar tenni. Ezt a függvényt meghatározni bonyolult és minden alkalmazási területen egyedi.

```
253 def reward_function(self, a_t1, sigma_tgt, sigma_t1, r_t, bp, p_t1, sigma_t2, a_t2):
254     R_t = a_t1 * (sigma_tgt/sigma_t1) * r_t - bp * p_t1 * abs((sigma_tgt/sigma_t1) * a_t1 - (sigma_tgt/sigma_t2) * a_t2)
```

32. ábra: Reward függvény definíciója (forrás: saját)

A 32. ábrán látjuk a reward függvény Python-i implementációját. A kalkulációban olyan paramétereket használunk, amelyeket többek között a bementi adatok előkészítésénél állítottunk elő.

$$R_t = \mu \left[A_{t-1} \frac{\sigma_{tgt}}{\sigma_{t-1}} r_t - bp p_{t-1} \left| \frac{\sigma_{tgt}}{\sigma_{t-1}} A_{t-1} - \frac{\sigma_{tgt}}{\sigma_{t-2}} A_{t-2} \right| \right]$$

A fenti képlet pedig a tényleges, matematikai definíció. Értékpapírok (vagy a papírban használt határidős ügyletek) hozamát és az ezzel kapcsolatosan meghozott modell döntéseket ezzel a függvénnyel tudjuk értékelni. A σ_{tgt} a target (cél) volatilitás, a σ_t pedig az adott időpont béli volatilitás, amelyet az exponenciálisan súlyozott mozgó szóródás 60 napra csúszóablakosan r_t -re (normalizált zárasi ár additív profitja) számolunk. A_t az adott időpontbeli akció, bp a jutalék diszkont ráta (az ügylet értékének mekkora százaléka megy el jutalékra), p_t pedig az adott időpontbeli price, azaz esetünkben a zárasi árfolyam. Ezen komponensek kombinációjából tevődik össze az R_t , amelyet próbálunk maximalizálni, minden egyes lépéssel.

```
258 def save_model(self):
259     # define full path for model to save to
260     self.q_eval.save(self.fname, save_format='tf', overwrite=True)
261
262 def load_model(self):
263     self.q_eval = load_model(self.fname)
264     return self.q_eval
```

33. ábra: save_model és load_model metódusok (forrás: saját)

A 33. ábrán látjuk az Agent osztály további funkcióinak metódusait, amelyekkel a betanított modellt tesztelési és későbbi felhasználás céljából lementhetjük (`save_model`) és ezt a lementett modellt ismét betölthetjük memóriába (`load_model`). A `save_model` esetében fontos megjegyezni a lementett modell formátumát. A standard a H5 formátum de a Tensorflow 2.0 megjelenésével elterjedt a Tensorflow saját (proprietary) formátuma, a SavedModel protokoll. A `load_model` esetében pedig csak a `q_eval` döntéshozó modellt kell előhívunk.

4.4 Modell betanítás és a betanított modell mentése

A modell betanításához néhány további funkcionalitásról is kell gondoskodni. Először is példányosítani kell az `Agent` osztályt (35. ábra), hogy legyen egy, a modellek betanításához használható agent. Ez az agent több, a papírban meghatározott hiperparaméter értékekkel rendelkezik (34. ábra).

Table 1: Values of hyperparameters for different RL algorithms.

Model	α_{critic}	α_{actor}	Optimiser	Batch size	γ	bp	Memory size	τ
DQN	0.0001	-	Adam	64	0.3	0.0020	5000	1000
PG	-	0.0001	Adam	-	0.3	0.0020	-	-
A2C	0.001	0.0001	Adam	128	0.3	0.0020	-	-

34. ábra: Hiperparaméter konfiguráció [18]

```
3 model_name = '/content/gdrive/My Drive/Colab Notebooks/saved_files/model_tech_combined'
4 print(model_name)
5
6 agent = Agent(lr=0.0001, gamma=0.3, n_actions=3, epsilon=1, batch_size=64, input_dims=(64,2),
7             epsilon_dec=1e-3, eps_end=0.01, mem_size=5000, fname=model_name,
8             fc1_dims=64, fc2_dims=64, replace=1000)
```

35. ábra: Agent objektum létrehozása (forrás: saját)

A 35. ábrán látjuk a betanításhoz használt `Agent` objektumot. Az objektum paraméterezése konzisztens a papírban meghatározottakkal. Amire eddig még nem tértünk ki, az a batch méret: 64-es részekre bontjuk a bementi adathalmazt, melynek további két oszlopa (close – zárási árfolyam, volume – kereskedett volumen) fog alkotni egy batch-et. A betanítási (és tesztelési) folyamat kritikus komponense a portfólió menedzsment vagy tranzakciókezelés (36. ábra).

```
1 eps_history = []
2
3 # constants
4 sigma_tgt = 0.03
5 reward = 0
6 L = len(data_train)
7
8 print('L:', L)
9
10 done = False
11 round = 0
12
13 # actual price: close * (1-bp)
14 bp = 0.001
15
16 transactions = pd.DataFrame(columns=['action', 'close', 'value', 'cost', 'qty', 'crt_blc', 'round'])
17
18 # budget
19 current_balance = 100000
```

36. ábra: Helper változók és a transaction DataFrame (forrás: saját)

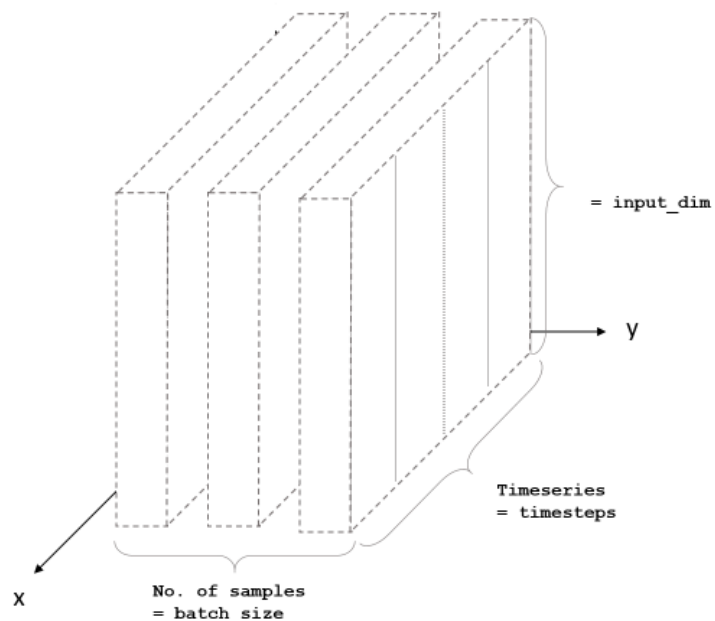
Minden egyes batch-nél végrehajtunk egy random vagy a modell által meghatározott döntést. Ezen döntés részletei egy `dictionary` adatszerkezetben vannak letárolva, amit

a lépés megtétele után beleillesztünk a `transactions` DataFrame-be. Ennek a DataFrame-nek az a szerepe, hogy minden, a betanítás során végrehajtott tranzakcióról tudjuk annak pontos körülményeit: milyen döntés született (eladás, megtartás, vétel), milyen áron, mi az ügylet értéke, mekkora az ügylet költsége (tranzakciós jutalék diszkontálásához), mennyi részvénnel kereskedtünk, az ügyletet követően mennyi likvid pénzünk van és hogy mindezek melyik kereskedési körhöz, lépéshez tartoztak. A portfólió menedzsmentről a fő ciklus tárgyalásakor fogunk többet olvasni. A fő ciklus feladata a bemeneti adathalmaz batch-enkénti feldolgozása, eljuttatása a modellbe, modell ezáltal betanítása, a tranzakciók monitorozása és a betanítási folyamat végén, a kész modell elmentése. A fő ciklusban két esetet vizsgálunk: az adathalmaz elejétől vett adatokat és a végét. Azért kell felbontanunk ezt is további két részre, mert az utolsónak feldolgozandó batch már nem felel meg a tanulás követelményeinek (nem tudunk egy következő batch-et kijelölni, amelyet megadhatunk a Q-függvény kalkulációban) így egy minimális mértékben máshogy kellett implementálni, mint az adathalmaz többi részét feldolgozó logikát.

```
218 # a limit belefér az adathalmazban itt folytathatjuk a batch-et, done = False
219 else:
220     batch_raw = data_train[batch_start:limit]
221
222     # filter
223     batch = batch_raw[['close_normalized', 'volume_normalized']]
224
225     # convert it to numpy array, then it becomes the observation
226     observation = np.array(batch)
227
228     observation_3d = observation.reshape(1, 64, 2)
229
230     # berakni a modellbe, várni az outputot
231     action = agent.choose_action(observation_3d)
232     print('action in learning:', action)
233
234     close_median = batch_raw['close'].median()
```

37. ábra: Konvencionális batch-ek feldolgozása a fő ciklusban (forrás: saját)

Az 37. ábrán látjuk a konvencionális batch-ek feldolgozásának első lépéseit. Kijelöljük az input halmazból az első 64-es batch-et, ebből kiemeljük a modell számára értelmezhető oszlopokat (normalizált zárasi árfolyam és normalizált volumen értékek), átalakítjuk háromdimenziós NumPy mátrix formátumba, azért, hogy az LSTM réteg tudja értelmezni (38. ábra) és odaadjuk a döntés választó logikának (`choose_action`), amelyből kiszadjuk az aktuális lépést.



38. ábra: LSTM által érthető háromdimenziós adatszerkezet [45]

Az aktuális lépés a `[-1,0,1]` lista egyike lehet, ezért további 3 elágazást definiálunk. Eladási művelet esetén minden jelenleg nálunk lévő részvényt eladunk. Hogyan tudjuk megtalálni a jelenleg birtokolt részvények számát? Mivel minden egyes előző tranzakciót követtük, így ezen „lista” (DataFrame) azon elemeit számoljuk, amelyek a legutóbbi eladási művelet után lettek létrehozva: ezek kizárólag vételi és megtartási műveletek lehetnek, így az a feladatunk, hogy a megvett mennyiségeket összeszámoljuk és az éppen aktuális zárási árfolyamon véglegesítsük az ügyletet.

```

# sell
if action == -1:
    # done = True
    transactions = transactions.sort_values(by=['round'], ascending=True)
    last_sell_index = transactions.loc[transactions['action'] == -1].last_valid_index()

    close = 0
    value = 0
    cost = 0
    qty = 0

    if last_sell_index == None:
        valid_records = transactions.loc[transactions['action'] == 1]

        if valid_records.empty:
            #close = data_train['close'].iloc[limit-minus_value]
            #print(data_train['ticker'].iloc[limit-minus_value])
            close = close_median
            value = 0
            cost = 0
            qty = 0
        else:
            # close = (valid_records['close'] * valid_records['qty']).sum()
            # qty = valid_records['qty'].sum()
            #close = data_train['close'].iloc[limit-minus_value]
            #print(data_train['ticker'].iloc[limit-minus_value])
            close = close_median
            qty = valid_records['qty'].sum()
            value = qty * close
            cost = value * bp

            current_balance = current_balance + (value - cost)

```

39. ábra: Sell művelet részlet kódban (forrás: saját)

Az 39. és 40. ábrákon láthatjuk az eladási (sell) művelet logikáját. Először is rendezzük a `transactions` DataFrame-t azért, mert csak rendezett adatszerkezetben tudunk relatív pozíciókat konzisztensen vizsgálni. Aztán megkeressük a pandas könyvtár `loc` függvényének segítségével az utolsó eladási műveleti tranzakció indexét, ahonnan majd vizsgáljuk a DataFrame további részeit. Előkészítünk néhány segédváltozót, amelyekkel feltölthetjük majd a tranzakciós dictionary-t, amelyet pedig majd hozzáadunk a `transactions` DataFrame-hez. Az eladási művelet végén a likvid pénzünk mértéke az ügyletnek a tranzakciós jutalékkal diszkontált értékével lesz több. Természetesen előfordulhat, hogy nem volt még sell művelet, ekkor minden eddigi vételi műveletet figyelembe kell vennünk.

```

else:
    valid_interval_data = transactions.iloc[last_sell_index:len(transactions)]

    # filter valid_interval_data where action == 1
    valid_records = valid_interval_data.loc[valid_interval_data['action'] == 1]

    if valid_records.empty:
        close = 0
        value = 0
        cost = 0
        qty = 0
    else:
        # close = (valid_records['close'] * valid_records['qty']).sum()
        # qty = valid_records['qty'].sum()
        #close = data_train['close'].iloc[limit-minus_value]
        #print(data_train['ticker'].iloc[limit-minus_value])
        close = close_median
        qty = valid_records['qty'].sum()
        value = qty * close
        cost = value * bp

    current_balance = current_balance + (value - cost)

```

40. ábra: Sell művelet előzőleg talált eladási művelet függvényében (forrás: saját)

```

dicti = {'action': action, 'close': close, 'value': value, 'cost': cost, 'qty': qty, 'crt_blc': current_balance, 'round': round}

# convert dict to pandas Series
s = pd.Series(dicti)

# add Series to transactions df - df = df.append
transactions = transactions.append(s, ignore_index=True)

```

41. ábra: Tranzakció dictionary hozzáadása DataFrame-hez (forrás: saját)

Az 41. ábrán látható a sell művelet paramétereit tartalmazó dictionary hozzáadása a transactions DataFrame-hez. A pandas DataFrame-hez nem lehet natív Python dictionary-t hozzáadni, azt először Series objektummá kell konvertálni. A következő műveletek kezelése hasonló. Megtartás (0) műveletnél is megkeressük az utolsó valid sell művelet rekord indexét és az ezután előforduló vételi műveleteket összesítjük, így megkapva a jelenleg nálunk lévő részvények darabszámát és aktuális árfolyamát. Itt is előfordulhat, hogy nem találunk valid sell indexet, ekkor az összes eddigi tranzakciót vizsgáljuk. A vételi műveletnél kell egy kicsit bővíteni a logikán, ugyanis nem mindegy, hogy a likvid pénzünk mekkora részét költjük el, egyáltalán mekkora részét költhetjük el. Egyelőre azt a logikát valósítottam meg, hogy a készpénzünk ötödével megegyező részvényt pakettbe tudunk befektetni egy vételi művelet során. Ekkor ezt az elköltött pénzt levonjuk a likvid tőke értékéből, viszont a tulajdoni hányadunk az adott vállalatban növekszik. Avagy a modell által választott vagy véletlenszerűen kiválasztott művelet végrehajtása után kiszámoljuk a reward függvény értékét. Azon paraméterek nagy részét, amelyeket letároltunk a DataFrame-ben, illetve azokat, amelyeket az adatok előfeldolgozásánál kiszámoltunk, behelyezzük a reward függvénybe és így megkapjuk az aktuális műveletünk matematikai értelmét, a lépés függvényében kiszámolt jutalmat. Ezután létrehozuk azokat

a további segédváltozókat, amelyeket el kell tárolnunk az agent memóriájában azért, hogy a modellt tudjuk tanítani, azaz backpropagation módszerrel a súlyok és egyéb paraméterek halmazát optimalizálni. Ha ezzel is megvagyunk, akkor léphetünk tovább a következő batch-re és a fő ciklus újra elindul előlről (42. ábra).

```
done = False

# false new observation_ for memory management
false_batch_start = batch_start + 64
false_batch_end = batch_end + 64

false_limit = min(false_batch_end, L)

false_batch_raw = data_train[false_batch_start:false_limit]

false_batch = false_batch_raw[['close_normalized', 'volume_normalized']]

observation_ = np.array(false_batch)
observation_3d_ = observation_.reshape(1, 64, 2)

# store transition
agent.store_transition(observation_3d, action, reward, observation_3d_, done)

# run state through model
agent.learn()

# reset false_batch
false_batch_raw = false_batch_raw.drop(false_batch_raw.index, inplace=True)
false_batch_start = 0
false_batch_end = 0

# at the end get new batch (state), ez az eltolásos módszer
batch_start += 64
batch_end += 64

# record epsilon value
eps_history.append(agent.epsilon)
round = round + 1
```

42. ábra: A betanítási fő ciklus vége (forrás: saját)

```

2 from google.colab import drive
3 drive.mount('/content/gdrive')
4
5 agent.save_model()
6
7 transactions_file = '/content/gdrive/My Drive/Colab Notebooks/saved_files/transactions_' + ticker + '.csv'
8 print('transactions_file:', transactions_file)
9
10 # save transactions and profit to file
11 transactions.to_csv(transactions_file, index=False)
12
13 # add current_balance to file
14 print(current_balance)
15
16 with open(transactions_file, 'a') as trans_file:
17     text = '\n' + 'Current balance:' + str(current_balance)
18     trans_file.write(text)
19 print('Write OK')

```

43. ábra: A betanított modell elmentése (forrás: saját)

A betanítási folyamat végén csak néhány feladatunk maradt: elmenteni a betanított modellt és kiírni a `transactions` DataFrame tartalmát fájlba. A modellt `SavedModel` formátumban egyszerűen feltudjuk tölteni a megadott Google Drive mappába. Ahhoz, hogy gyorsan megismerjük az egész megoldás lényegét, hogy mennyi pénzünk lett a futás végén, ezt is beleírjuk a tranzakciós fájlba. A `SavedModel` formátum használatára rá vagyunk kényszerítve. Az ehhez hasonlóan egyedi származtatott `keras.Model` modell objektumokat nem lehet H5 formátumban lementeni.

My Drive > Colab Notebooks > saved_files > model_tech_combined ▾

Name ↑	Owner	Last modified	File size
 assets	me	Oct 11, 2020 me	—
 variables	me	Oct 11, 2020 me	—
 saved_model.pb	me	Oct 11, 2020 me	751 KB

44. ábra: `SavedModel` formátumban lementett neurális háló (forrás: saját)

Érdekes módon az általam ismert machine learning könyvtárak mindegyike bináris formában tárolja a lementett neurális háló modelleket, nem pedig egyszerűen egy konfigurációs fájlként (pl. JSON). Ezekben a bináris fájlokban benne van a modell minden tulajdonsága, paramétere, rétegei azért, hogy tesztelésnél vagy pedig éles környezetben való beindításkor ne kelljen egy adott modellt mindig újra betanítani.

4.5 Betanított modell tesztelése, eredmények összefoglalása

A betanított modellt a teszt adathalmazon teszteljük, amely teljes mértékben olyan adatokat tartalmaz, amelyeket eddig még nem látott a modell. Ebben a fázisban már nem tanul a modell, sőt már nincsenek is random választások. Egyszerűen a modell kimeneteit használjuk az összes teszteset batch-en. A futtatási logika ettől függetlenül teljes mértékben megegyezik a betanítási procedúrával: megadjuk a batch-et a modellnek, ő kiad egy általa optimálisnak vélt lépést, ezt végrehajtjuk és a tranzakció paramétereit memóriában vezetjük, majd, amikor végigértünk a teszt adathalmaz egészén, akkor a tranzakciókat fájlba kiírjuk. A folyamat egyik érdekessége a betanított modell betöltése. Ekkor megvan a teszteléshez szükséges modell objektumunk, amelyet átadva a tesztelési agent-nek tudunk futtatni (45. és 46. ábra).

```
1 # mount drive
2 from google.colab import drive
3 drive.mount('/content/gdrive')
4
5 # load model
6 model = agent.load_model()
7 model.summary()
```

45. ábra: Betanított modell betöltése memóriába (forrás: saját)

```
1 # creating testing agent
2 agent_test = Agent(lr=0.0001, gamma=0.3, n_actions=3, epsilon=1, batch_size=64, input_dims=(64,2),
3                  epsilon_dec=1e-3, eps_end=0.01, mem_size=5000, fname=model_name,
4                  fc1_dims=64, fc2_dims=64, replace=1000, testing=True, model=model)
```

46. ábra: Tesztelési agent létrehozása (forrás: saját)

Ha visszagondolunk az Agent osztály konstruktorára, akkor azon belül megkülönböztettünk train és test agent-et, a tesztelő esetben pedig a paraméterként átadott modellt töltjük be. A következő lépés a teszt adatok betöltése. Mivel már megvan az az index, ameddig néztük a betanítási adatokat, most egyszerűen nem addig, hanem onnantól vesszük az adatokat az eredeti bemeneti adathalmazból. Itt is létrehozuk a normalizált oszlopokat, a reward függvény segéd oszlopait, stb. Innentől kezdődik a tesztelési fő ciklus, amely logikájában ugyanaz, mint a betanításai, azt leszámítva, hogy itt a modell kimeneteit nem vezetjük az agent memóriájában, hiszen nem fog tanulni a modell. És mivel nem tanul a modell, így a tesztelési agent test ágában definiált logika mentén választunk akciót: minden lépést a döntéshozó (q_{eval}) modellre bízunk. Végigérünk az összes batch-en és a tranzakciós adatokat kiírjuk fájlba.

Megközelítések és eredmények

A projekt első szakaszaiban individuális részvényekre koncentráltunk, aztán megpróbáltuk más részvényekre is általánosítani a megoldást. Azt szeretnénk volna, hogyha létrejön egy általános modell, amelybe bármilyen részvényt vagy részvény csomagot belerakva, képes megütni a 10 százalékos lélektani hozam értéket. Ebben az esetben magasfokú naivitásról beszélhetünk, egy praktikusán kettő attribútumból álló bementi adathalmazzal ez természetesen nem megoldható. Ettől függetlenül próbálkoztunk tovább index fund részvényekkel (olyan értékpapírok, melyek követik a tőzsde indexek árfolyamát, pl. az S&P 500 indexét), valamint specifikus szektoronkénti elemzésekkel és szektorokat nem figyelembe vevő elemzéssel is. A szektoros elemzés mögötti megfontolás az volt, hogyha nem is tudjuk leírni az egész piacot, akkor legalább annak egy összefüggő, összetartozó részét viszont már talán mégis sikerülni fog. Ebben az esetben vettük a S&P 500 tőzsde indexet kiadó 64 legnagyobb informatikai-technológiai vállalatot (Information Technology, Communications Services, E-commerce szektorokból) és az adataikat összegyűrve hoztuk létre a projekt során használt legnagyobb, kb. 165500 soros adathalmazt, amelyet részletesen tárgyaltunk az adatok előfeldolgozását bemutató részben. Megvan 64 cég historikus részvény árfolyama, mi alapján döntjük el, hogy adott tranzakció esetében melyiket vesszük meg, melyiket adjuk el? Ezen kérdés megoldásának módszertanát egyszerűen közelítettük meg: azt a részvényt adjuk el vagy vesszük meg egy adott lépésben, amely a batch medián értékével megegyezik. Abban az esetben, amikor nem vettük figyelembe a szektorokat, hanem egyszerűen vettük az S&P 500 tőzsde index 64 legnagyobb cégét (piaci kapitalizáció alapján), akkor nem medián értékkel számoltunk, hanem az Apple részvényeivel kereskedtünk. Adtunk a modelleknek 100000 dollárnyi büdzt és rájuk bízunk ezen összeg befektetését. Az eredmények a teszt adathalmazokon futtatott betanított modellek teljesítményeit tükrözik. A tapasztalatokat az alábbi táblázatban foglaltuk össze.

Combined	Model	Ticker	Holding quantity	Close	Current balance	Value of Portfolio
No	AAPL_model	AAPL	947	97.19	357696	449734.93
No	VOO_model	AAPL	63	499	127527	158964
No	VOO_model	AMD	84	81.77	27542	34410.68
No	VOO_model	AMZN	0	3144.88	13596	13596
Yes	Combined_model	RDSA	0	24.24	62463.21	62463.21
Yes	Tech_Combined_model	AAPL	726	125.01	31245.62	122002.88
Yes	Tech_Combined_model	AMD	0	83.8	97554	97554
Yes	Tech_Combined_model	MSFT	0	226.58	99204.42	99204.42
Yes	Tech_Combined_model	TSLA	149	447.75	116829.72	183544.47

47. ábra: A különböző megközelítésekkel betanított modellek eredményei (forrás: saját)

Az eredmények nagy teljesítménybeli szórást mutatnak. Egyéni részvényes módban az Apple papírokra nagyon jól teljesített, index fund módban az Amazon-on veszített, ami abszolút meglepő, hiszen az Amazon árfolyama kis túlzással 10 éve nem esett, az pedig várható volt, hogy olaj cég révén, a Shell papírokra veszteséget fog elérni, tekintve az olaj árak zuhanását. Az eddig legalaposabban átgondolt módszerrel, a technológiai szektor részvényein történő teszteléskor, az Apple-el ismét nyertünk, AMD-vel és Microsoft-al minimálisan veszítettünk és hatalmas meglepetésre a Tesla-val pedig nagyot nyertünk. Ez azért érdekes, mert a Tesla volt talán az utóbbi évtized legmegbízhatóbb,

legvolatilisabb részvénye. Ezek a tesztelési megközelítések egyelőre nem mutatnak átfogó képet a modell teljesítményéről, több szektor még több részvényén kell betanítani a modellt és az ár választást is biztosan lehet tovább hangolni. Továbbá fontos megemlíteni, hogy ez a modell architektúra a papír szerint határidős ügyletek szerződéseire lett optimalizálva, nem egyszerű részvények kereskedésére. Azonban azok komplex pénzügyi instrumentumok és még a működésük hátterét is át kell látni ahhoz, hogy konzekvensen belehessen őket integrálni egy ilyen machine learning megoldásba.

5. Összefoglalás, kitekintés

A szakdolgozat keretein belül megismerkedtünk a gépi tanulásnak, mint a BI folyamat egy integrális részét képező állomásának elméleti hátterével, a különböző területeivel, részletesen ismertettük a visszacsatolós tanulás és azon belül a Q-learning matematikai formalizációit és fogalmi környezetét. Ismertettük a machine learning pipeline állomásait, az adatbetöltéstől és preprocesszálastól kezdve a modell alkotáson keresztül, a modell értékeléséig és mentéséig. Az egyes gépi tanulás megközelítések közül a felügyelt, nem felügyelt és a visszacsatolós tanulással foglalkoztunk. A visszacsatolós módszer volt az, amely során egy agent-et megbíztunk azzal, hogy a számára kialakított környezetben hozzon döntéseket, melyeket a terület specifikus reward függvénnyel értékeltünk. A DDQN kapcsán az agent-hez definiáltunk továbbá kettő neurális háló közötti kapcsolattartás képességének szükségességét is. Megemlítettük, hogy a gépi tanulás széles spektrumú feladatköröket lát el és mivel egyelőre nem létezik olyan megközelítés, amely jól tudna általánosítani az életben fellelhető összes komplex feladat között, így minden egyes bevetési területen specifikus architektúrákat és módszereket kell alkalmaznunk (domain specifikus alkalmazás). Ez azért van, mert még kutatások ezrei folynak annak kapcsán, hogy ezeket a különböző architektúrájú és működésű neurális hálókat, egyszerű felügyelt tanulási matematikai modelleket vagy esetleg a már létező összetettebb megoldásokat hogyan lehetne integrálni. Ez a problémakör tökéletesen leírja az emberi idegrendszerben tapasztaltakat: testünkben milliárdnyi neuron képes elektromos impulzusok generálására, azonban egy végtag irányítása teljesen más mechanizmusokkal jár, mint egy belső szerv monitorozása. Erre az evolúció az embernek megoldásként a legnagyobb térfogatú agyat adta, mely az emberi idegrendszer központja és az általunk egyelőre ismert legösszetettebb organizmus. Innen nézve láthatjuk, hogy nem egyértelmű ezeknek a különböző területekre optimalizált megoldásoknak az integrációja. Felhívtuk a figyelmet arra, hogy nem kell egy élettelen objektumnak öntudatra ébredni ahhoz, hogy akarattal és intelligenciával rendelkezzen. Az intelligencia nem más, mint problémák megoldásának képessége. Ehhez koncepcionálisan egyszerű komponensek szükségesek: tárolási, számítási és végrehajtási kapacitások, melyek mindegyikével már most rendelkeznek a számítógépeink. Éppen ezért követünk el nagy hibát akkor, hogyha nem készülünk fel arra a napra, amikor az élettelen tárgyak intelligenciája meghaladja a kollektív emberi képességeket. Ezt a napot azonban már egy most folyamatban lévő átmeneti időszak előzi meg, amelynek során munkahelyek sokasága szűnik meg a digitalizáció és automatizáció révén.

Az elméleti bemutató után pedig végigvettük egy teljes machine learning pipeline gyakorlatban is megvalósított tervezését és felépítését. A cél egy olyan megoldás elkészítése volt, amely képes néhány attribútummal rendelkező (napi zárási árfolyam és napi kereskedett volumen) részvény adathalmaz alapján olyan, egy zárt eseményterű eredményhalmazból választott lépéseket tenni, amely hosszú távon maximalizálja az adott részvényekkel történő kereskedésünk közben akkumulált hasznót. A megoldáshoz az adatokat összegyűjtő és feldolgozó modul mellett kettő neurális hálót használtunk fel, melyek egymással párhuzamosan futottak úgy, hogy egyikük feladata volt egy bizonyos Q-függvény optimális értékének meghatározása, míg a másiknak az elé prezentált állapotok függvényében történő lépések meghozatala. Erre azért volt szükség, mert ha mindezen

feladatok elvégzésére egyetlen neurális hálót használtunk volna, akkor a modell nem tudott volna a limitált méretű adathalmazainkon szinte semmit megtanulni. Ezt a rendszert több jellegű adaton tanítottuk be, volt, hogy individuális (egyéni vállalat) papírokon és megpróbáltunk először az egész piacot, majd pedig a piac egy-egy szektorát általánosan leíró adathalmazokat felhasználni. Az eredmények meglepően sokszínűek lettek, elmondhatjuk, hogy a legnagyobb sikereket az egyéni részvényeken történő betanítás és tesztelés során értük el és az biztos, hogy további munkát kell befektetni a több részvény közötti általánosítást megvalósító módszerekbe. A technológiai cégeken vagy sokat nyertünk vagy keveset veszítettünk, az olaj cégen pedig a vártak megfelelően sokat veszítettünk. A következő lépések tisztán kirajzolódtak a tesztelési fázis során: több attribútumot is vizsgálni kell, egy olyan komplex rendszert, mint a tőzsdei kereskedelem, nem lehet pusztán kettő attribútum segítségével hatékonyan modellezni. Így egy ún. *feature engineering* modul fejlesztése jó következő lépés lehet, amelynek segítségével megtalálhatjuk azokat a további attribútumokat, amelyek szignifikánsan befolyásolják egy részvény árfolyam ingadozását. Természetesen nem hagyhatjuk ki azt a továbblépési lehetőséget sem, hogy teljesen más pénzügyi instrumentumokat vizsgálunk. Manapság a részvényekkel történő kereskedelem konzervatív technikának minősül és nem kecsegtet hatalmas haszonnal. Befektetésünk megsokszorozásához más ügyleteket is vizsgálnunk kell: határidős ügylet szerződéseket (futures) és opciós ügyleteket (options trading). Felmerülhet az is, hogy abszolút új technológiai megközelítéseket vizsgálunk, más modell architektúrákkal. A jelenleg elterjedt alternatív megoldások a GAN (Generative Adversarial Networks) és AC (Actor Critic) hálózatok, amelyek úgy gondolom, hogy további lehetőségeket jelenthetnek.

Ezen megoldás elkészítésével egy régi célom elérésének irányába tettem egy nagy lépést. A munka során hatalmas segítségemre volt konzulensem, Dr. Pödör Zoltán, akinek szeretném megköszönni a mindig jó hangulatú konzultációkat, az útmutatást és a folyamatos, szakszerű támogatást.

Szombathely, 2020. november 12.

Irodalomjegyzék, hivatkozások

- [1] The world's most valuable resource is no longer oil, but data: <https://www.economist.com/leaders/2017/05/06/the-worlds-most-valuable-resource-is-no-longer-oil-but-data>
- [2] Alessandro Bigiotti, Alfredo Navarra - Optimizing Automated Trading Systems: https://link.springer.com/chapter/10.1007%2F978-3-030-02351-5_30 (Utolsó megtekintés: 2020.11.17)
- [3] Keith D. - A Brief History of Machine Learning: <https://www.dataversity.net/a-brief-history-of-machine-learning/#>
- [4] Democratization of AI: <https://towardsdatascience.com/democratization-of-ai-de155f0616b5>
- [5] China Is Leading in Artificial Intelligence: <https://www.inc.com/magazine/201809/amy-webb/china-artificial-intelligence.html>
- [6] Artificial Intelligence - examples of ethical dilemmas: <https://en.unesco.org/artificial-intelligence/ethics/cases>
- [7] Csanaki Richárd, Dr. Pödör Zoltán - ETL folyamatok megismerése és megvalósítása, Önálló laboratórium 1.
- [8] Csanaki Richárd, Dr. Pödör Zoltán - Különböző környezetekben működő adattárházak közötti adat áttöltés tesztelése, Önálló laboratórium 2.
- [9] 90% People Lose Money In Stocks: <https://bit.ly/3dK4lCW>
- [10] Alapkammat: <https://www.mnb.hu/>
- [11] Állampapír hozamok: <https://www.allampapir.hu/allampapirok/>
- [12] Maginfláció: <https://infostart.hu/gazdasag/2020/09/09/ksh-39-szazalek-volt-az-inflacio>
- [13] Középosztály részesedésének csökkenése: <https://www.pewsocialtrends.org/2020/01/09/trends-in-income-and-wealth-inequality/>
- [14] William H. Pike - Why Stocks Go Up and Down: <https://www.amazon.com/Why-Stocks-Go-Up-Down/dp/0989298205>
- [15] Yang Wang, Dong Wang, Shiyue Zhang, Yang Feng, Shiyao Li, Qiang Zhou - Deep Q-trading
- [16] Thibaut Théate, Damien Ernst - An Application of Deep Reinforcement Learning to Algorithmic Trading
- [17] Omer Berat Sezer, Ahmet Murat Ozbayoglu - Algorithmic Financial Trading with Deep Convolutional Neural Networks: Time Series to Image Conversion Approach
- [18] Zihao Zhang, Stefan Zohren, and Stephen Roberts - Deep Reinforcement Learning for Trading

- [19] A pénz és tőzsde csodavilága (1990) – André Kostolany
- [20] Machine Learning – Tom Mitchell: <http://www.cs.cmu.edu/~tom/mlbook.html>
- [21] Mohri, Mehryar; Rostamizadeh, Afshin; Talwalkar, Ameet - Foundations of Machine Learning
- [22] A.I. Took a Test to Detect Lung Cancer. It Got an A.: <https://www.nytimes.com/2019/05/20/health/cancer-artificial-intelligence-ct-scans.html>
- [23] How to Evaluate Unsupervised Learning Models: <https://towardsdatascience.com/how-to-evaluate-unsupervised-learning-models-3aa85bd98aa2>
- [24] Reward függvény maximalizálása: <http://users.isr.ist.utl.pt/~mtjspaen/readingGroup/ProofQlearning.pdf>
- [25] Alfa értékének beállítása: <http://incompleteideas.net/sutton/book/ebook/the-book.html>
- [26] Volodymyr Mnih, Koray Kavukcuoglu - Google DeepMind Atari: <https://patentimages.storage.googleapis.com/71/91/4a/c5cf4ffa56f705/US20150100530A1.pdf>
- [27] DQN felépítés (1): <https://www.semanticscholar.org/paper/V-D-D3QN%3A-the-Variant-of-Double-Deep-Q-Learning-Huang-Wei/e6a1640c03c50a55ef3e00a0592dbb0851fe33bb>
- [28] DQN felépítés (2): https://www.researchgate.net/profile/Yagna_Patel2/publication/329642599/figure/fig7/AS:703631024197632@1544770117737/Dueling-Deep-Q-Network-Framework.ppm
- [29] Deep Neural Network: <https://www.houseofbots.com/images/news/1442/cover.png>
- [30] Egy neuron bemenete, kimenetei: <http://neuralnetworksanddeeplearning.com/chap1.html>
- [31] Gradient descent kihívása: <https://www.mltut.com/stochastic-gradient-descent-a-super-easy-complete-guide/>
- [32] Four Common Types of Neural Network Layers: <https://towardsdatascience.com/four-common-types-of-neural-network-layers-c0d3bb2a966c>
- [33] Donald J. Trump elnök: <https://www.forbes.com/sites/danalexander/2020/09/24/trump-tower-has-collected-16-million-from-presidents-campaign-including-38000-last-month/?sh=1e8e876164ea>
- [34] Konvolúció művelete: https://www.researchgate.net/figure/Example-of-a-discrete-convolution-a-and-equivalent-transposed-convolution-operation-b_fig2_321719286
- [35] LSTM neuron felépítése: https://www.researchgate.net/publication/13853244_Long_Short-term_Memory
- [36] LSTM fajták: <https://colah.github.io/posts/2015-08-Understanding-LSTMs/>

- [37] LSTM sejt felépítése: https://en.wikipedia.org/wiki/Long_short-term_memory#/media/File:The_LSTM_cell.png
- [38] Dave Kuhlman - A Python Book: https://web.archive.org/web/20120623165941/http://cutter.rexx.com/~dkuhlman/python_book_01.html
- [39] TensorFlow: <https://www.tensorflow.org/>
- [40] GitHub: <https://techcrunch.com/2018/06/04/microsoft-has-acquired-github-for-7-5b-in-microsoft-stock/>
- [41] Yahoo Finance: <https://www.nytimes.com/2015/02/19/business/ex-fortune-editor-joins-yahoo-finance.html>
- [42] Alpha Vantage: <https://www.alphavantage.co/>
- [43] Dueling Deep Q Learning with Tensorflow 2 & Keras - Phil Tabor: <https://youtu.be/CoePrz751lg>
- [44] Adam optimalizáló algoritmus: <https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/>
- [45] LSTM bemeneti formátuma: <https://mc.ai/understanding-input-and-output-shape-in-lstm-keras/>