Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Automatizálási és Alkalmazott Informatikai Tanszék

Szilágyi Krisztián Gergely

AUTONÓM JÁRMŰ TANÍTÁSA SZIMULÁCIÓS KÖRNYEZETBEN

Önvezető szimulátor fejlesztése megerősítéses tanulással

KONZULENS

Dr. Szemenyei Márton

BUDAPEST, 2021

Tartalomjegyzék

Összefoglaló

A diploma munkám egy az Irányítástechnika és Informatika Tanszéken megtalálható HPI Trophy Flux Buggy távirányítós versenyautó autonóm járművé alakítása. A rendszer bemenete az autóra szerelt RGB-D kamerából szerzett információk, a döntéshozatal utáni kimenete pedig az autó irányításához szükséges jel magasszintű reprezentációja. Az autonóm járművek biztonságos és gazdaságos fejlesztése megköveteli a szimulált környezetek alkalmazását. Így az algoritmusok biztonságos keretek közt tesztelhetők.

Lényegében a feladat egy megerősítéses tanulással betanított neurális hálózat [1] alapú software létrehozása. A tanításhoz létre kell hozni egy a feladat elvégzéséhez alkalmas szimulált környezetet. A szimulált ágensnek implementálni szükséges olyan funkciókat, mint például a sávkövetés/tartás, álló és mozgó akadályok detektálása, kikerülése, sőt akár a jelzőtáblák és közlekedési lámpák figyelembevétele.

Ez egy több féléves projekt, így tagoltam a cél eléréséhez vezető utat négy fázisra a négy félév szerint: potenciálisan alkalmazható technológiák megismerése és egy kezdetleges architektúra megtervezése, egyszerű szimulációs környezet kialakítása. Majd a kezdeti architektúra tesztelése a környezetben, első tanítások alapján iteratívan az architektúra finomítása, módosítása, és a környezet finomítása. A végleges komplex környezetben az elkészült, kiforrott algoritmus tanítása, legvégül pedig integráció a célhardware-re és valós környezet béli tesztek és finomítások elvégzése. A projekt megvalósításán egyébként egy több fős csapat dolgozik. A hardvert fejlesztők feladata a szenzorok és a feldolgozó egység kiválasztása, integrálása, míg az én feladatom megalkotni azt a szoftvert, mely autonóm járművet varázsol a távirányítós autóból.

Abstract

My thesis is the conversion of an HPI Trophy Flux Buggy remote controlled racing car into an autonomous vehicle, which is the property of the Department of Control Engineering and Informatics. The input of the system is the information obtained from the RGB-D camera mounted on the car, and the post-decision output is a high-level representation of the signal needed to control the car. The safe and economical development of autonomous vehicles requires the use of simulated environments. Thus, the algorithms can be tested in a secure framework.

In essence, the task is to create a software based on neural networks [1] taught through reinforcement learning. For training, a simulated environment suitable for the task must be created. The simulated agent should have functions such as lane keeping, detecting and avoiding stationary and moving obstacles, and even recognize signs and traffic lights.

This is a long-term project, so I divided the path to achieve the goal into four phases according to the four semesters: searching for potentially applicable technologies and designing a rudimentary architecture, also creating a simple simulation environment. Then testing the initial architecture in the environment, based on first trainings, iteratively refining the architecture of the model, and refining the environment. Teaching the completed, mature algorithm in the final, complex environment, and finally integration with the target hardware and performing real-world tests and refinements. Incidentally, a team of several people is working on the development of the project. It is the job of the hardware developers to select and integrate the sensors and the processing unit, while it is my part of the job to create the software that turns the the remote controlled car into an autonomous vehicle.

1 Irodalmi áttekintés

Ebben a fejezetben a teljesség igénye nélkül bemutatom, hogy mik azok a kifejezések, eljárások, melyek mindenképpen szükségesek a dolgozatban bemutatott megoldások és eredmények megértéséhez. A legfontosabb részeket részletesebben prezentálom, de nyilvánvalóan túl nagy ez a tématerület, hogy túlságosan elmélyedhessünk a részletekben.

Az elején nagyvonalakban bemutatom, hogy mit érdemes tudni a mély tanulásról, azután végig megyek a dolgozatban felhasznált háló struktúrákon. Végezetül néhány speciális algoritmus is bemutatásra kerül, melyek kevésbé ismertek.

1.1 Deep Learning

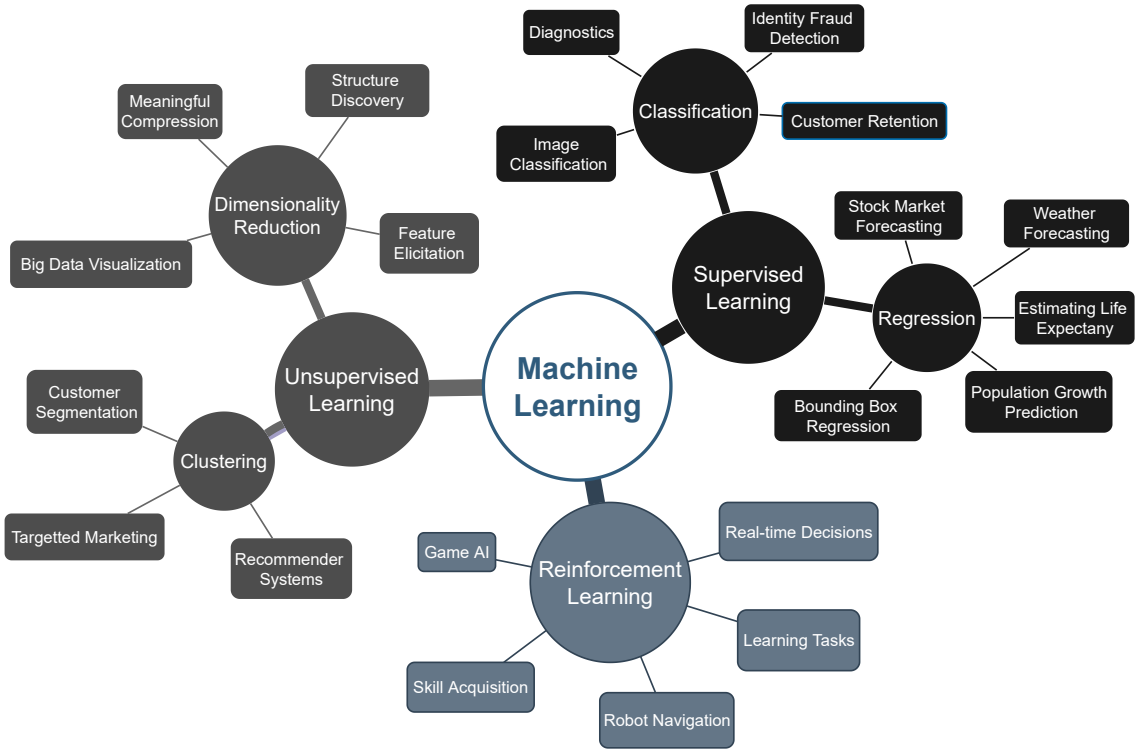
A gépi tanulás (Machine Learning) az egyik út a sok közül a mesterséges intelligencia felé. Ennél az eljárásnál használt algoritmusok analizálják az adatokat, tanulnak az adatokból, majd meghatároznak vagy megjósolnak új adat pontokat. Szemben egy tradicionális algoritmussal, amelynél előre meg van írva, milyen helyzetben mit kell csinálnia (feltételekre épülő struktúra), ehelyett helyzeteket prezentálunk az algoritmusnak, amelyekre megtanul jól reagálni.

A mély tanulás (Deep Learning) a reprezentáció tanulásnak egyik alkalmazása, míg a reprezentáció tanulás a gépi tanulásos eljárások egyik részhalmaza. A reprezentáció tanulásnál a belső reprezentációkat, jellemzőket nem kézzel kell beállítani, hanem megtanulja az algoritmus. Mély tanulásnál ez a folyamat több rétegű, egyre pontosabb belső jellemzőket (*feature*) alkot a bemenetből. Kezdetben a bemenet egyszerűbb jellemzőire tanul rá, majd rétegről rétegre egyre komplexebb, absztraktabb tulajdonságokat képes felismerni az algoritmus. Például egy képfelismerő algoritmus esetén az első rétegek megtanulják felismerni az éleket, sarkokat a képen, míg az utolsó rétegek már felismernek szemeket, szájakat vagy autó kerekeket stb. A mély tanulás jellemző eszközei a mély neurális hálók, melyekről később szó lesz részletesebben.

1.1.1 Csoportosítás

A gépi tanulás módszereit többféleképpen lehet csoportosítani, tanulási eljárás alapján három felé szokták osztani: létezik felügyelt (supervised), felügyelet nélküli

(unsupervised) és megerősítéses (reinforcement) tanítás. Ezek más és más típusú problémákhoz nyújtanak hatékony segítséget. Osztályozáshoz, azaz adatok csoportosításához, valamint regresszióhoz felügyelt tanítást érdemes használni. Az osztályozás esetén a lehetséges kimenetek diszkrét értékek, például egy bináris osztályozónál a bemenet jó/nem jó, 0 vagy 1. Regressziónál folytonos kimenetet kapunk, például objektumdetektálásnál bounding box illesztésénél az objektum köré a kimenetek a téglalap leírásához szükséges 2-2 koordináta értékek. A felügyelt jelző ezesetben azt jelenti, hogy miután a gép jósolt egy eredményt, mi megmondjuk neki, hogy mi lenne a helyes eredmény, amiből tud tanulni. Tehát a bemeneti adatok címkézettek, a gép adat-címke párokat kap tanuláskor, ellenben a felügyelet nélküli tanításnál. Ezt az utóbbi módszert főleg klaszterezéshez [2], struktúraminták felismeréséhez használják. Az utolsó említett eljárás, a megerősítéses tanulás esetében a rendszer egy dinamikus környezettől kap pozitív vagy negatív visszacsatolást a meghozott döntései után. Többnyire jutalom- vagy büntetőpontokat kap, miközben próbálja elérni a célját, például, hogy minél messzebbre jusson egy autóval a versenypályán. Ezt a koncepciót főleg játékok MI-jének fejlesztéséhez, robotok, autók navigációjához használják. Ebben a dolgozatban ezt az eljárást alkalmaztam, ezért a későbbiekben ezt fogom csak részletezni.



1.1. ábra A gépi tanulás egy csoportosítása

A gépi tanulás megvalósítására többféle eljárás létezik. A felügyelt tanításnál főleg az SVM, azaz szupport-vektor gép [3] és a neurális hálózat alkalmazása terjedt el. A megerősítéses tanuláshoz alkotott modellemet neurális hálóval valósítottam meg, így a továbbiakban csak ezt a módszert fogom kifejteni. A neurális hálózatok szintén tovább bonthatók több típusra különböző szempontok alapján. Más architektúra effektív képfelismerésnél, más videók analízisére és megint más természetes képek generálására.

1.1.2 Neurális hálózatok

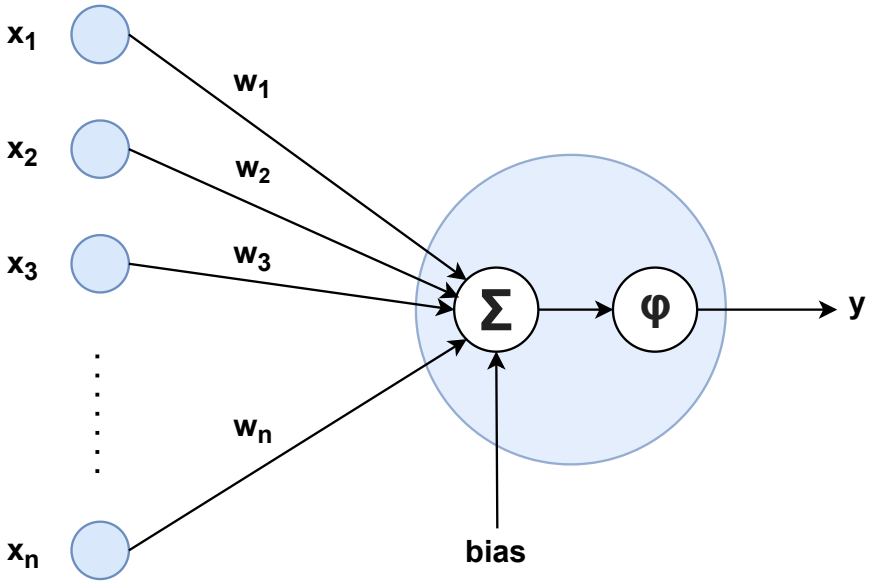
A neurális hálózat egy mély tanulást (Deep Learning) megvalósító soft-computing módszer, melynek tömören a funkciója, hogy a bemenetén kapott adatból képez egy belső reprezentációt, majd ez alapján hoz meg egy döntést. Ennek reprezentációnak a jobb és jobb megalkotását tanulja meg a háló, hogy minél pontosabb döntést hozzon. Elnevezését onnan kapta, hogy mesterséges neuronokból és ezek közti kapcsolatokból épül fel, ezzel imitálva az emberi agy felépítését. A neuronokat rétegekbe rendezik, melyek egy hálózaton belül három fő csoportra oszthatók: bemeneti, kimeneti és rejtett rétegekre. Ha a rejtett rétegek száma nagy, akkor hívjuk a hálót mély neurális hálózatnak (DNN).

A továbbiakban bemutatok néhány definíciót, melyeket szükséges kicsit részletezni ahhoz, hogy érthetők legyenek a későbbi fogalmak az olvasó számára, valamint, hogy például mikre szükséges figyelni egy neurális hálózat megalkotásakor, tanításakor. Viszont ennél részletesebben nem célom kifejteni a témakört, mert nem ez a dolgozat témája.

A perceptron az a struktúra, mely egy neuronból és az előtte lévő rétegbeli neuronokkal való kapcsolatából áll (lásd **1.2. ábra**). Ez a modul a legelemibb algoritmus a neurális hálózatban. Igazából ez volt a legelső lineáris osztályozó (az osztályozás feltétele a kimenet előjele). Vegyük az előző réteg neuronjainak a kimeneteit súlyozva, melyet összegezve egy konstanst eltolással (*bias*) megkapjuk a perceptron bemenetét. Egyszerűbben fogalmazva ezzel egy affin transzformációt hajtottunk végre. Ezt követi az aktivációs függvény, amely megadja a perceptron kimenetét (bemenete a kapott összeg, kimenete egy valós szám, lásd 1.1 egyenlet). Azt reprezentálja, hogy mennyire aktiválódik (tüzel) a neuron. Ez egy nem-lineáris függvény, célja, hogy nem-linearitást vigyen a rendszerbe, hiszen enélkül csak lineáris műveletekből épülne fel a hálózat és így nem lenne univerzális approximátor, azaz nem tudna tetszőleges bemenetre tetszőleges

függvényt illeszteni. Az MLP, azaz a *multilayer perceptron* (a legismertebb neurális hálózat modell) perceptronok összeségéből épülnek fel, az említett rétegekbe rendezve.

$$y = \varphi\left(\sum_{i=1}^{n} w_i \cdot x_i + b\right) \tag{1.1}$$



1.2. ábra Perceptron

A neurális hálók leglényegesebb metódusa, a hiba-visszaterjesztési algoritmus (*backpropagation*), ez valósítja meg a tanulás képességét. Ekkor írja felül a háló a megtanulható paramétereit: például a súlyokat és eltolásokat egy MLP esetében. Tehát ekkor frissül a háló az újonnan megszerzett „tudásával". Például megfigyeléses tanításkor az általa megjósolt eredményt összehasonlítja a valós (*ground-truth*) értékkel, azaz az elvárt eredménnyel, majd ebből számolunk egy veszteséget/költséget. A veszteségfüggvénnyel azt határozhatjuk meg, hogy ezt miképp tegye, például a veszteség a két érték különbségének L2 normája (vagy más szóval az euklideszi távolságuk) legyen. A tanítás célja ennek a veszteségnek a minimalizálása, vagyis hogy így a háló kimenete egyre közelebb kerüljön a valós értékhez a lehető legtöbb bemeneti adatra.

Viszont már a lokális minimum elérése sem egy könnyen elérhető cél, a globális minimum megtalálása pedig egy rendkívül nehéz feladat. Az utóbbi feladat elvégzésére szokás alkalmazni a genetikus algoritmusokat, amelyek bizonyítottan előbb vagy utóbb megtalálják a globális optimumot. Hátrányuk, hogy az egyáltalán nem garantált, hogy ez belátható időn belül sikerül, valamint ezek az algoritmusok ezen tulajdonságából adódóan

igen pazarlóak. Ezeknél gyorsabb konvergenciát mutatnak a gradiens alapú módszerek, cserébe viszont a globális optimumot nem valószínű, hogy képesek megtalálni. Tipikusan a költség/veszteség minimalizálása miatt a neurális hálózatok esetében a negatív gradiens alapú optimalizálást érdemes használni.

A globális optimum megkereséséhez a hálózat hiperparaméterein kell változtatni. Hiperparamétereknek hívjuk azokat a változókat, melyek a hálózat struktúráját definiálják, a tanítás paramétereit állítják stb. Ilyenek például a neuronok és rétegek száma, a veszteségfüggvény típusa, a nem-linearitások típusa, tanulási ráta (*learning rate*), epochok száma és még megannyi más. Az optimum megtalálását legegyszerűbben kézi próbálkozásokkal lehet elérni, bizonyos előismeretek alapján lehet sejteni (valamint ökölszabályok követését is érdemes figyelembe venni), hogy bizonyos paramétereket milyen nagyságrendben érdemes megválasztani stb. Szofisztikáltabb megoldás az evolúciós algoritmusok használata, tehát ismét előkerülnek a genetikus algoritmusok. Ebben az esetben már érdemesebb felhasználni azt a képességüket, hogy képesek megtalálni a globális optimumot. Ugyanis a hiperparaméterek száma sok-sok nagyságrenddel kisebb, mint a hálózat (tanulható) paramétereinek dimenziója. Például amíg egy egyszerűbb hálónak 10-20 hiperparamétere van, addig tipikusan több millió paramétere. Egy ilyen megoldásra példa a NEAT algoritmus, ahol a populáció egyedei, melyek szaporodnak, mutálódnak, elpusztulnak, azok egy-egy neurális hálózatok, különböző génekkel (hiperparaméterekkel, topológiákkal). A globális optimumnál található egyed(ek), azaz amikor például legkisebb az általánosítási hiba (az új adatokon mért veszteség) vagy legpontosabb a kimenet (egy általunk bevezetett mérőszám alapján), lesz a választott hiperparaméterű hálónk. Egy másik nem ennyire bonyolult megoldás a Bayes optimalizáció, mely egy iteratív eljárás függvények optimalizálására, leginkább a kimenet maximalizálására. Így használhatjuk a háló tanításakor a pontosság maximalizáláshoz, miközben iteratívan keresi az algoritmus az optimális hiperparaméter beállítást. Ez is egy globális optimumot megtaláló metódus.

Tanítás közben az ún. optimalizáló feladata, hogy minimalizálja a veszteséget a súlyok és eltolások frissítésével. Mint említettem többnyire negatív gradiens alapú algoritmusokat szokás használni. Ezek mind a költségfüggvény háló paraméterei szerinti deriváltjait számítják, majd egy konstans vagy adaptívan változó lépéshosszal lépnek a hipersíkon a kiszámolt gradiens irányába, a szerinte megfelelő irányban változtatva a háló paramétereit.

Az optimalizáló a veszteség gradiens kiszámításához használja fel a hiba-visszaterjesztési algoritmust. Hiba-visszaterjesztésnek azért hívjuk, mert a kimenettől visszafelé rétegenként számolja ki a veszteség deriváltjait (az adott paraméterek szerint) egészen a háló a bemenetéig. Így végül megkapva, hogy milyen mértékben függ a kimenet (a veszteség) a bemenettől. Egyszerűbben fogalmazva: Az optimalizáló a kimeneti réteg aktivációján változtatni csak úgy tud, hogy vagy változtat a kimeneti réteg súlyain és az eltoláson, vagy az előző réteg aktivációján (lásd 1.1 egyenlet). De az előző réteg aktivációján csak úgy tud változtatni, hogy változtatja az adott réteg súlyait és eltolását vagy az azt megelőző réteg aktivációját és így tovább. Egészen a bemeneti rétegig láncszabállyal képezi a veszteség gradienseit.

A neurális hálózatok tanítását szakaszokra osztják fel, melyek mérete a tanító adathalmaz függvényében változik. Ugyanis, ha a háló tanulható paramétereit minden egyes adat után írnánk felül, akkor az egy rettentő lassú és számításigényes folyamat lenne. Éppen ezért az adatokat csoportosítják, úgynevezett batch-ekbe (gyakran *mini-batch*-nek is nevezik) rendezik, és ezt az adatcsomagot adják be a hálónak egyetlen adat helyett. Először az egész adathalmazt felbontjuk azonos méretű (tipikusan 4 hatványai, ami állítólag a GPU-k felépítése miatt alakult így) batchekre. Miután egy batch adatra kiszámolta a háló a kimeneteit és a veszteséget, frissíti a paramétereit. Egy teljes ciklust, amikor az összes batch végig ment a hálón - vagyis a háló találkozott már az összes tanításra szánt adattal - hívjuk egy epochnak. Tehát, ha 10 epoch-ot szeretnénk futtatni tanításkor, és a 100 képből álló adathalmazunkat felbontjuk 20 méretű batchekre, akkor minden adattal pontosan tízszer találkozott a neurális háló és összesen ötvenszer frissítette a paramétereit.

Neurális hálózat nem csak perceptronokból épülhet fel, később látni fogjuk, hogy bármiből lehet neurális hálózatot építeni, mely eleget tesz azon kritériumoknak, hogy több rétegű, képes a bemenetből kiszámolni egy kimenetet (ez evidens) és képes legyen a hiba-visszaterjesztésre (tehát minden elemének differenciálhatónak kell lennie). Néhány ilyen fontos struktúra például a konvolúciós hálók (CNN – Convolutional Neural Network) és a visszacsatolt hálók (RNN – Recurrent Neural Network). A dolgozatom specifikus témája miatt ennél több neurális hálózatokkal kapcsolatos témát nem érdemes érinteni, mert nem lényeges foglalkozni például a regularizációs eljárásokkal, az *over-fitting* és más jelenségekkel a megerősítéses tanulás esetében.

1.1.3 LSTM

A képek hatékony feldolgozása után a következő kérdés lehet az, hogy miként tudunk feldolgozni képsorozatokat? Az előre csatolt hálóknak mivel nincsen memóriaeleme, elvész a képek közti változások információja, vagyis az időbeli információk. Kellene egy olyan háló struktúra, amely képes modellezni az időt, tehát legyen belső állapota (memóriája). Az ilyen típusú feladatokra találták ki a visszacsatolt neurális hálózatot (RNN), melyek képesek hatékonyan feldolgozni a bemeneten érkező szekvenciát, és általában abból szintén valamiféle szekvenciát állítanak elő (sequence-to-sequence modellek). Nem fix hosszú bemeneteknél előjön az a probléma, hogy ahogy nő a bementi szekvencia hossza, úgy egyre nagyobb lesz a réteg, amelyen számolni kell a hiba-visszaterjesztést. A számításigény csökkentésének érdekében a hiba-visszaterjesztés fix N hosszú lépésig megy vissza, azaz nem szükséges, hogy a végtelenségig „visszaemlékezzen” a háló. Majd látni fogjuk, hogy a mi esetünkben nem kell figyelni a változó hosszúságú bemenetekre, mindig ugyanannyi képet (megfigyelést) dolgozunk fel. Gyakori felhasználási területük a videóanalízis és a nyelvi fordítók.

Többféle visszacsatolt neurális hálózat struktúra is napvilágot látott, két elterjedt típusa az LSTM (Long Short-Term Memory) és a GRU (Gated Recurrent Unit). Mindkettő komplexebb architektúrájú, mint az egyszerű RNN réteg, így képesek arra, hogy sok idő-lépésen is át memorizálhassák az állapotokat. Így mindkettő megoldják a hosszútávú függőségek okozta problémákat, szemben az RNN rétegekkel.

Továbbá az egyik legfontosabb tulajdonságuk, hogy kiküszöbölik az eltűnő gradiens és a felrobbanó gradiens (vanishing and exploding gradient problem) jelenségét, míg ezek komoly gondot jelentenek az alap RNN rétegnél. Az előbbi jelenség lényege, hogy a neurális hálózat tanítása közben a hiba-visszaterjesztésnél a gradiensek a bemeneti réteg felé haladva fokozatosan eltűnnek. Ez akkor fordulhat elő, ha a gradiens kisebb lesz egynél és a láncszabály következtében az egymást követő hatványozások, vagy szintén kis értékekkel való szorzások miatt lényegében egy idő után már zérusok lesznek a deriváltak. Emiatt a bemeneti réteg felé a súlyok nem lesznek frissítve és lényegében megszűnik a tanulás folyamata, mivel beragad ebbe az állapotba a háló (ha a háló első fele nem tanul, akkor lényegében a háló sem tanul). A felrobbanó gradiens ennek az ellentétje, akkor fordul elő, ha a gradiens sokkal nagyobb lett egynél és emiatt a hiba-visszaterjesztésnél elszállnak a súlyok, instabil lesz a hálózat. Előre csatolt hálóknál egyik legismertebb megoldás ezekre a reziduális hálók (lásd ResNet), visszacsatolt hálóknál

pedig az LSTM és a GRU cellák. A továbbiakban az előbbit fogom részletezni, egyébként csak minimális eltérés van a két cella működése között.

Az LSTM cella tulajdonképpen négy (az RNN csak három) lineáris neurális háló „rétegből" (más néven kapukból) áll: egy felejtő (f_t), egy bemeneti (i_t), egy kimeneti (o_t) kapu és egy update ($\tilde{C}_t$) rétegből. Adott t idő-lépésben x_t a bemenet, h_t a kimenet és C_t a cella állapota. Az alábbi első négy egyenletben látható a kapuk leírása (1.2-5 egyenlet), mind a négy a cella előző időpontbeli kimenetétől és az aktuális bemenettől függ. A **1.3. ábra** *W*-vel jelölt blokk jelzi a két vektor konkatenációját és az adott súlymátrixokkal és *bias* értékkel való affin transzformációt. Az első három kapu esetében *sigmoid* (1.8 egyenlet) aktivációs függvényt, míg az update kapunál *tanh* nemlinearitást használunk. A cella aktuális állapotát két komponensből kapjuk meg: a felejtő kapuval beállítjuk, hogy mennyit tartson meg, vagy más szóval mennyit „felejtsen el" az előző időpontbeli értékéből, míg a bemeneti kapuval beállítjuk, hogy mennyit frissítsünk a belső állapoton (1.6 egyenlet). Végezetül az így kapott cella aktuális állapotából számítjuk ki az aktuális kimenetet. A kimeneti kapu állítja be, hogy milyen mértékben teszi ezt.

$$f_t = \sigma(W_f[x_t, h_{t-1}] + b_f) \tag{1.2}$$

$$i_t = \sigma(W_i[x_t, h_{t-1}] + b_i) \tag{1.3}$$

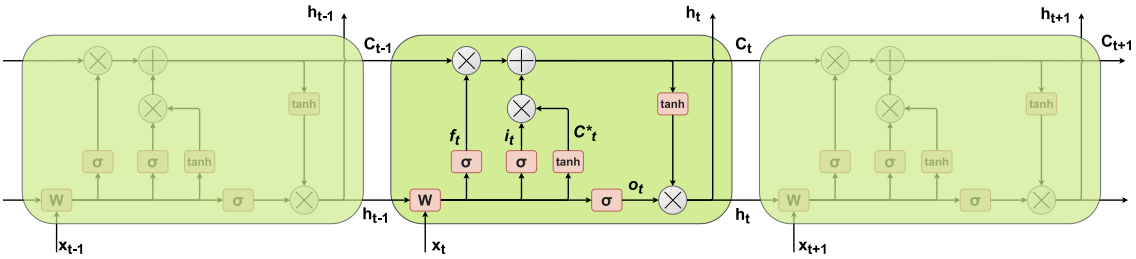
$$o_t = \sigma(W_o[x_t, h_{t-1}] + b_o) \tag{1.4}$$

$$\tilde{C}_t = \tanh(W_c[x_t, h_{t-1}] + b_c) \tag{1.5}$$

$$C_t = f_t C_{t-1} + i_t \tilde{C}_t \tag{1.6}$$

$$h_t = o_t \tanh(C_t) \tag{1.7}$$

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{1.8}$$

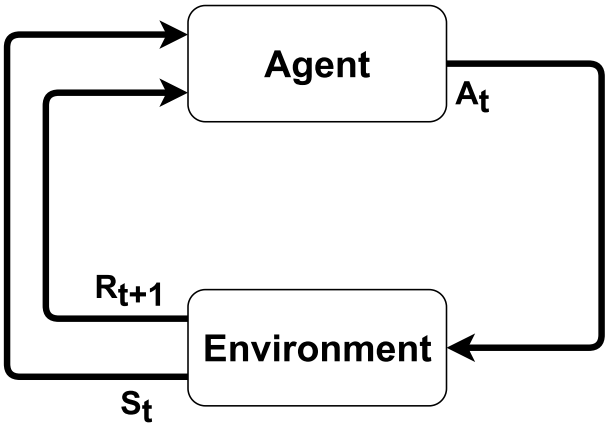


1.3. ábra LSTM cella

Mint említettem, az LSTM képes kezelni az eltűnő gradienseket, ehhez az kell, hogy a C_t és a C_{t-1} közötti derivált 1 körül legyen, mert ekkor a derivált sokadik (összes idő-lépésnyi) hatványa nem száll el egyik irányba sem. Látható, hogy a C_t C_{t-1} szerinti deriváltja csak f_t-től függ, így ez akár lehet egy is (0 és 1 közötti értéket vehet fel a *sigmoid* miatt).

1.2 Reinforcement Learning

A megerősítés tanulás során egy úgynevezett ágens interaktál egy környezettel: különböző akciókat hajt végre környezetben, a környezet adott időpontbeli állapotától függően és ezért valamekkora jutalmat kap. Minél közelebb került az ágens a célfeladat elvégzéséhez, annál többet. Minden egyes akció után a környezet egy új állapotba kerül, majd jutalmazzuk/büntetjük az akciót. Tanításkor egy epizódnak nevezzük azt a ciklust, melynek a végén a környezet visszaáll a kezdeti állapotára és kezdődik elölről a folyamat. Hasonló ehhez a felügyelt tanulásnál használt epoch fogalma.



1.4. ábra MDP

Az ágens legfőbb tulajdonsága a stratégia (policy), mely egy olyan függvény, ami minden állapothoz hozzárendel egy akciót. A sztochasztikus stratégiát, mely az

állapotokhoz egy valószínűségi eloszlást rendel π-vel jelölünk, míg a determinisztikus stratégiát μ-vel szokás. A megerősítéses tanulás célja, hogy megtaláljuk az optimális stratégiát, vagyis azt a stratégiát, ami maximalizálja a teljes jutalom várható értékét.

A π sztochasztikus stratégia szerinti érték (állapot-érték) függvény véve az *s* helyen megmutatja, hogyha ezt a stratégiát követjük, mennyi az *s* állapot értéke, azaz mennyi a jövőbeli diszkontált jutalom várható értéke (1.10 egyenlet). A jövőbeli diszkontált jutalom (1.9 egyenlet) a jövőbeli jutalmak összege, exponenciálisan súlyozva a diszkont rátával ($0 < \gamma \leq 1$). Az állapot-érték függvényhez hasonlóan definiálhatunk akció-érték függvényt, mely egy állapot-akció párhoz rendeli a jövőbeli diszkontált jutalom várható értékét, adott π stratégia mellett:

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{k+t+1} \tag{1.9}$$

$$V^{\pi}(s) = \mathbb{E}_{a\sim\pi}[G_t | s_t = s] \tag{1.10}$$

$$Q^{\pi}(s,a) = \mathbb{E}_{a\sim\pi}[G_t | s_t = s,\ a_t = a] \tag{1.11}$$

Két hasznos eljárást érdemes megemlíteni, mielőtt rátérnék a tanuló algoritmusokra. Az első az epsilon greedy stratégia [4]. Megerősítéses tanulás esetén kérdés, hogy milyen taktikát válasszon az ágens, inkább felfedezze a környezetet (exploration), vagy inkább a már felfedezett trajektóriát folytatva kiaknázza a lehetőségeket (exploitation). Felfedezés esetén nem a legvalószínűbb akciót választjuk, hanem véletlenszerűen mintavételezünk az akciók közül. Célja, hogy olyan állapotba is eljuthasson az ágens, melyben még nem volt, ne korlátozza be magát a lehetséges állapotok egy szűk halmazába. Kiaknázáskor a cél, az, hogy maximalizálja az elérhető jutalmat, más szóval azt az akciót hozza meg, melyre a legnagyobb a diszkontált jutalom várható értéke. Minden epizód elején eldöntjük, hogy melyik legyen a prioritás, az epszilon ($0 \leq \varepsilon \leq 1$) változó segítségével:

$$a_t = \begin{cases} sample(\pi), & \varepsilon \\ \arg\max_{a\in A} Q^{\pi}(s,a), & 1-\varepsilon \end{cases} \tag{1.12}$$

A probléma az, hogy kiaknázáskor a jutalom nem biztos, hogy a lehető legnagyobb (lokális maximumot találunk meg). Jó eljárás lehet az, ha az epszilon 1 közeli értékről az

epizódok során folyamatosan csökken, tehát először hagyjuk az ágenst felfedezni az első epizódokban, utána viszont ösztönözzük, hogy aknázza ki a legjobb trajektóriákat.

A másik említendő eljárás, amely szintén megoldja azt, hogy ne ragadhasson be az ágens állapotok egy szűk halmazába a tapasztalat visszajátszás (*experience replay*). A véletlenszerűséget úgy garantálja, hogy minden idő-lépésben az akció után kimentjük az ágens tapasztalatát a memóriába (*replay memory*). A tapasztalat a jelenlegi állapotból, akcióból, az akcióra kapott jutalomból és a következő állapotból áll: $e_t = (s_t, a_t, r_{t+1}, s_{t+1})$. Majd véletlenszerűen kiválasztunk egy batch-et a memóriában tárolt tapasztalatok közül, ezekből az állapotokat küldjük végig a hálón és ezután számoljuk ki a költséget. Tehát végül véletlen mintákon végeztük el a tanítást (akár csak a véletlenszerű batch-ek esetében megfigyeléses tanulásnál), bár ehhez az kellett, hogy kétszer értékeljük ki a hálót minden egyes idő-lépésben.

1.2.1 Q-tanulás és stratégia gradiens módszerek

Q-tanulásnak [5] nevezzük azt az iterációs algoritmust, mely egy véletlenszerűen inicializált *Q* függvényből iteratívan előállítja az optimális *Q* függvényt, azaz az összes stratégia közül a maximális akció-érték függvény:

$$\hat{Q}^{\pi}(s,a) = \max_{\pi} Q^{\pi}(s,a) \tag{1.13}$$

Ebből pedig már meghatározható az optimális stratégia, hiszen az optimális stratégiánk az optimális *Q* függvény szerinti legjobb akció meglépése.

Az iteráció alapja a Bellman-egyenlet, mely azt fejezi ki, hogy egy adott állapot-akció párból a lehető legnagyobb jutalom megegyezik a közvetlenül kapott jutalom és a következő állapotból elérhető legnagyobb jutalom összegével:

$$Q(s_t, a_t) = \mathbb{E}[r_{t+1} + \gamma V^{\pi}(s_{t+1})] \tag{1.14}$$

Ne felejtsük el, hogy a jutalom 1-től indexelőik, tehát a nulladik akcióra a jutalom r_1 (lásd **1.4. ábra**). Az iteráció felírásához kell még egy összefüggés: az optimális *V* függvény milyen alakban írható fel az optimális *Q* függvényében. Mivel az állapot-érték függvény lényegében a várható értéke az akció-érték függvénynek, ezért könnyen adódik az alábbi formula:

$$\hat{V}^{\pi}(s) = \max_{a} \hat{Q}^{\pi}(s,a) \tag{1.15}$$

Az 1.14 és 1.15 egyenletet összerakva kapjuk, hogy

$$\hat{Q}(s_t,a_t) = \mathbb{E}\left[r_{t+1} + \gamma \hat{V}^{\pi}(s_{t+1})\right] = \mathbb{E}\left[r_{t+1} + \gamma \max_{a_{t+1}} \hat{Q}^{\pi}(s_{t+1},a_{t+1})\right] \tag{1.16}$$

A második egyenlőség csakis az optimális stratégiát követve teljesül. Ezt felhasználva írhatjuk fel a Bellman-szabályt, amely a Q-tanulás iterációs szabálya lesz:

$$\hat{Q}_{k+1}(s_t,a_t) = \mathbb{E}\left[r_{t+1} + \gamma \max_{a_{t+1}} \hat{Q}_{k}^{\pi}(s_{t+1},a_{t+1})\right] \tag{1.17}$$

A *Q* függvény megtanulása viszont rendkívül nehéz feladat lehet a nagy számú, vagy akár végtelen lehetséges állapottal rendelkező környezetek esetében, miközben a stratégia egy viszonylag egyszerű függvény. Emiatt célszerűbbnek tűnik, ha közvetlenül a stratégiát próbálnánk meg megtanulni, az akció-érték függvény helyett. Ez a céljuk az ún. stratégia gradiens (policy gradient) módszereknek. A legegyszerűbb ilyen a *REINFORCE* algoritmus [6], más néven a Monte-Carlo policy gradient. Az utóbbi elnevezést onnan kapta, hogy a várható értéket Monte-Carlo módszerrel becsüljük, azaz, véletlen mintavételezéssel a várható értéket az átlaggal közelítjük (így tulajdonképpen az állapot-érték függvény az átlagos diszkontált jutalom). A stratégia gradiens, azaz a költségfüggvény háló paraméterei szerinti deriváltja egy hosszadalmas levezetés után a következőképpen néz ki:

$$J(\theta) = \mathbb{E}(r(\tau)) = \int_{\tau} r(\tau) p(\tau;\theta) \tag{1.18}$$

$$\nabla_{\theta} J(\theta) = \sum_{t=0}^{T-1} \nabla_{\theta} log \pi_{\theta}(s_t,a_t)\, G_t \tag{1.19}$$

Az egyenlet egyszerűen kifejtve: Az algoritmus lényege, hogy ha egy akcióra nagy jutalmat kapott, akkor megerősítjük a döntésében (a gradiens irányába lépünk), tehát úgy módosítjuk a háló paramétereit, hogy legközelebb nagyobb valószínűséggel hajtsa végre ezt az akciót. Ellenkező esetben ellenezzük a döntését, így csökkentjük az adott akció valószínűségét (a gradienssel ellenkező irányba lépünk). Látható, hogy a kapott képletben

a gradiensre nem az összes jutalom, hanem csak a jövőbeli diszkontált jutalmak vannak hatással. Ennek oka, az úgynevezett credit-assignment probléma csökkentése. Lényege, hogyha a teljes trajektória jutalmát néznék, akkor nem tudnánk megmondani, hogy a sok-sok akció közül melyek voltak igazából a jó vagy a rossz döntések. Így az akció-sorozatban el tud tűnni egy-egy nagyon rossz lépés, ha ettől még átlagosan jó a jutalom és fordítva. A következménye pedig az, hogy instabillá válik az algoritmus, zajos lesz a gradiens becslés és nehezen fog konvergálni. Ezért szűrjük le a számításba jöhető jutalmakat, úgy, hogy egyrészt az adott akciónál csak akció után kapott jutalmak számítsanak (jövőbeli), másrészt érdemes figyelni arra, hogy jövőben távoli jutalmak kevésbé, míg a közelebbi jutalmak nagyobb mértékben számítsanak (diszkontált). Ha belegondolunk az ember is így működik.

Szintén egy megoldandó probléma az is, hogy miként állítsuk be a jutalmazás mértékét. Ha a legtöbb esetben nemnegatív értékek a jutalmak, akkor a hálót tulajdonképpen nem is büntetjük egy rossz döntésnél, inkább csak kevésbé erősítjük meg a döntésében. Ezért célszerű lenne kiszámolni egy alap értéket, például a véletlenszerű stratégia által elérhető jutalmat (ami nem feltétlenül nulla), melyet kivonunk az aktuális jutalomból. Ez a *baseline* fogalma, mint egy offset, eltoljuk a nulla átmenetet. Tehát ennél az alapértéknél jobb teljesítményt jutalmazzuk, a rosszabbat büntetjük. A következőképpen módosul a stratégia gradiens alakja:

$$\nabla_\theta J(\theta) = \sum_{t=0}^{T-1} \nabla_\theta log\pi_\theta(s_t, a_t)\left(G_t - b(s_t)\right) \tag{1.20}$$

A *baseline* a meghatározására léteznek különböző, jól bevált módszerek. Például ahelyett, hogy konstans értékűnek választanánk, érdemesebb lenne adaptívnak beállítani. Például válasszuk meg úgy, hogy akkor jó a jutalom, ha az nagyobb, mint az adott állapotból elérhető jutalom várható értéke, azaz az érték függvénynél. Ezt a különbséget hívjuk előny függvénynek (*advantage*), mely tulajdonképpen a Q és V függvények különbsége:

$$A^\pi(s_t, a_t) = Q^\pi(s_t, a_t) - V^\pi(s_t) \tag{1.21}$$

$$\nabla_\theta J(\theta) = \sum_{t=0}^{T-1} \nabla_\theta log\pi_\theta(s_t, a_t)\, A^\pi(s_t, a_t) \tag{1.22}$$

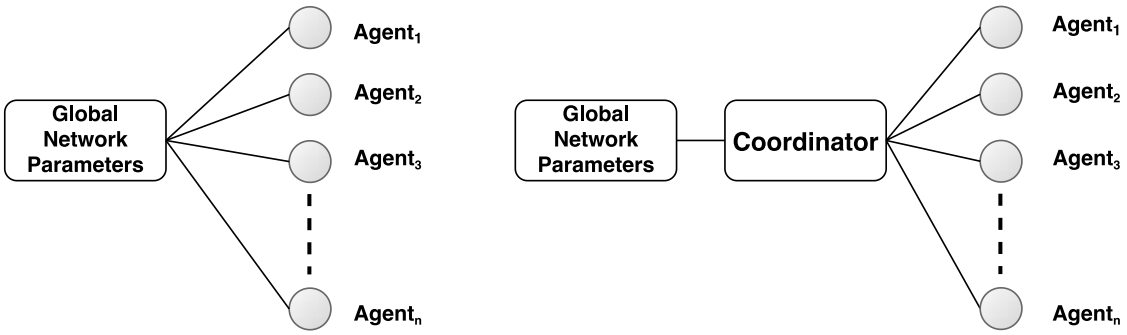
1.2.2 Actor-Critic

A következő említendő stratégia gradiens módszer az ún. Actor-Critic [7]. Az ilyen funkciót ellátó neurális hálóknak két „fejük" van, azaz a háló egy pontján ketté válnak a rétegek. Van egy *Actor* fej, mely θ paraméterekkel rendelkezik és *REINFORCE* módszerrel tanulja az optimális stratégiát abba az irányba, amelybe a *Critic* fej javasolja. A *Critic* fej viszont Q-tanulás segítségével az *A* előny függvényt próbálja meg előállítani *w* paraméterekkel. Pontosabban előtte algoritmustól függően az állapot-érték függvényt (*V*) vagy az akció-érték függvényt (*Q*) állítja elő. Az utóbbi módszert szokták Q Actor-Critic-nek nevezni.

További két fontos változata létezik ennek a módszernek: az A2C (Advantage Actor-Critic [8]) és az A3C (Asynchronous A2C [9]). Ezeknél a *Critic* fej az állapot-érték függvényt állítja elő, melyből megkaphatjuk az előny függvényt a Bellman-egyenlet segítségével. A 1.14 és a 1.21 egyenleteket felhasználva kapjuk meg így az előny függvényt számunkra hasznos alakját:

$$A^\pi(s_t, a_t) = r_{t+1} + \gamma V^\pi(s_{t+1}) - V^\pi(s_t) \tag{1.23}$$

E két algoritmus lényege, hogy tanítás alatt több ágens hajt végre akciókat több párhuzamosan futó környezetben, függetlenül egymástól (lásd **1.5. ábra**). Egyik előnyük, hogy így könnyebb felfedezni a környezetet, így nincs szükség sem az epsilon greedy stratégiára, sem a tapasztalat visszajátszásra. Az A3C nagy hátránya, hogy a globális háló paramétereket aszinkron módon használják, így előfordulhat az az inkonzisztenciát okozó eset, hogy az ágensek különböző stratégia verziót használnak éppen, ezért a paraméter frissítés nem lesz optimális. Ennek kiküszöbölésére az A2C bevezet egy koordinátort, mely szinkronizálja a szálakat. Megvárja, míg minden párhuzamosan futó ágens befejezte a feladatát (véget ért az epizódjuk, mivel vagy sikeres lett feladat, vagy mert például lejárt az idő). Csak ezután történik meg a frissítés, ezzel megoldva a problémát, hogy így minden epizódot mindegyik ágens ugyanazzal a stratégia verzióval kezdi. Mérések alapján az A2C gyorsabb konvergenciához vezet.



1.5. ábra Bal oldalt az A3C, jobb oldalt az A2C működése látható

1.3 Attention

2019-ben nagy népszerűségnek örvendtek az ún. Transformer típusú neurális hálózatok [10]. Ezeknél a *seq-to-seq* felépítésű, főleg nyelvi fordításra használt hálóknál alkalmazott eljárások az Encoder-Decoder Attention és a Self-Attention. Az utóbbi esetében N hosszú szekvencia elemei interaktálnak egymással (self) és „kitalálja" a háló, hogy melyikükre figyeljen a legjobban (attention). Ezzel szemben az Encoder-Decoder Attention metódusban a bemenet a cél kimenettel interaktál. A dolgozatban a Multi-Head Attention módszert alkalmazom az ágensemben, aminek az alapja a Self-Attention, így csak ennek a működését fejtem ki a továbbiakban.

1.3.1 Self-Attention

A figyelem mechanizmusa tömören annyit jelent, hogy egy bemeneti szekvencia elemeit nem egyenlő arányban veszi figyelembe egy algoritmus (például egy neurális háló), hanem úgymond ráfokuszál a bemenet egyes elemeire, míg a többi elemet nagyjából figyelmen kívül hagyja. Lényegében dinamikusan súlyozza a bemenetét: a súlyozott átlag súlyait a bemeneti elemek tulajdonságai alapján (kulcs - *key*) és az alapján állítja, hogy mire szeretnénk, hogy figyeljen a háló (lekérdezés - *query*). Minden bemeneti vektornak három belső reprezentációja van: egy *key* (**k**), egy *query* (**q**) és egy *value* (**v**) vektor. A *key* vektor identifikálja az elemet, leírja, hogy az adott elem mit tartalmaz. A *query* vektor leírja, hogy mire szeretnénk figyelni, mit keresünk az adott bemenetben. A *value* vektor a bemenet értékét adja, meg, e mentén fogjuk súlyozni az elemeket. Így már részletesebben is megérthetjük, mi a lényegi különbség a kétfajta említett figyelem mechanizmus között. A Self-Attention esetében mind a 3 reprezentáció vektor a bemenettől származik, míg az Encoder-Decoder Attention esetében csak a *key* és *value* származik a bemenettől (enkóder), míg a *query* vektor a cél kimenettől (dekóder).

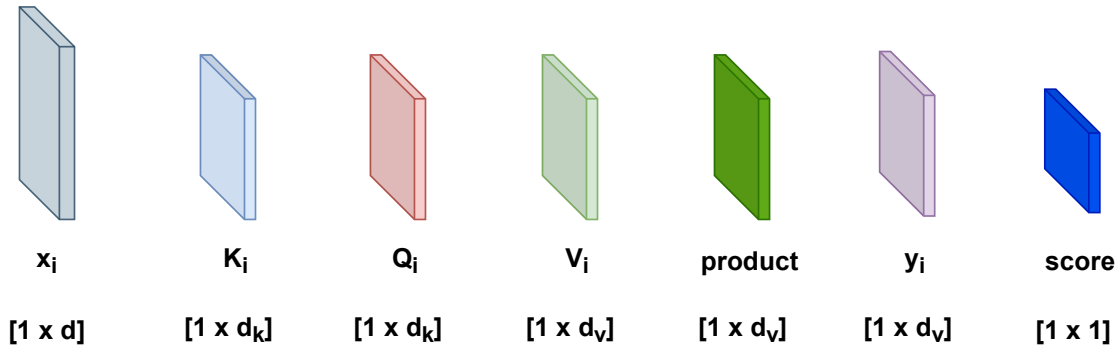
Fontos komponense még az algoritmusnak maga a pontozás beállítása, azaz, hogy miszerint számítsa ki a figyelem pontokat, melyekkel kialakítja a megfelelő súlyozást. Mint említtetettem ezt a *key* és *query* alpján teszi, tehát a pontozó függvény (f_{score}) bemenete a szekvencia adott elemére az elem kulcs vektora és a *query* vektor. A függvény lehet bármilyen metódus, akár egy neurális hálózat is, de a Self-Attention esetén skaláris szorzatot alkalmaznak. Az összes elemre kiszámolt figyelem pontokra számol ezután egy úgynevezett *softmax* függvényt (később fejtem ki), így megkapva a súlyokat, melyekkel súlyozva az elemek értékét (*value* vektorok) számoljuk ki az eredményt:

$$s_i = \frac{\exp\left(f_{score}(key_i, query)\right)}{\sum_j \exp\left(f_{score}\left(key_j, query\right)\right)} \tag{1.24}$$

$$y = \sum_i s_i \cdot value_i \tag{1.25}$$

Látni fogjuk, hogy ez az algoritmus lényegében egy egyszerű előre csatolt neurális hálót valósít meg. A továbbiakban a fenti metódust fejtem ki részletesebben, melyet a **1.7. ábra** érdemes nyomon követni.

Ahhoz, hogy megkapjuk a fent említett belső reprezentációs vektorokat, a bemeneti $\boldsymbol{x}$ $1 \times d$ méretű vektort egy az adott reprezentációhoz tartozó súlymátrixxal ($\mathbf{W}^{\boldsymbol{K}}$, $\mathbf{W}^{\boldsymbol{Q}}$, $\mathbf{W}^{\boldsymbol{V}}$), például a $\mathbf{W}^{\boldsymbol{K}}$ kulcs-mátrixxal szorzunk. Ezeknek a méretei a **1.6. ábra** alapján könnyedén megadhatóak: a $\mathbf{W}^{\mathrm{K}}$ és $\mathbf{W}^{\boldsymbol{Q}}$ mátrixnak azonos méretűnek kell lenniük: $d \times d_k$, míg a $\mathbf{W}^{\boldsymbol{V}}$ egy $d \times d_v$ méretű mátrix lesz (d_v megegyezhet d_k értékével).



1.6. ábra A Self-Attention algoritmusban használt vektorok méretei, a helytakarékosság miatt elforgatva (transzponálva) vannak ábrázolva a téglatestek.

A bemeneti vektorokat egy $\boldsymbol{X}$ $N \times d$ méretű mátrixba rendezve a súlymátrixokkal 3 mátrix szorzással megkaphatjuk a reprezentációk $\boldsymbol{K}$, $\boldsymbol{Q}$ és $\boldsymbol{V}$ mátrixait (N a szekvencia

hossza, azaz a bemeneti vektorok száma). Ezután kiszámoljuk a bemeneti szekvencia első eleméhez ($\boldsymbol{x_1}$ vektorhoz) tartozó figyelem pontot. A **1.6. ábra** látható, hogy ennek az eredménye egy skalár. Vegyük észre, hogy az első bementhez tartozó kimeneti $\boldsymbol{y_1}$ vektor számításához csak az első bemenethez tartozó $\boldsymbol{q_1}$ *query* vektorát kell felhasználni, mivel az első elemhez tartozó figyelem pontokat a $\boldsymbol{q_1}$ vektor határozza meg. Tehát a $\boldsymbol{q_1}$ $1 \times d_k$ méretű vektort szorozzuk az összes key reprezentációt tartalmazó $\boldsymbol{K^T}$ $d_k \times N$-es mátrixxal. Az így kapott $1 \times N$ méretű figyelem pont vektort még vissza kell skálázni a rejtett dimenzió méretének a gyökével. Ezután a szorzatra, azaz a vektor elemeire számolunk egy *softmax* függvényt, mely a bemenetére adott vektor elemeit 0 és 1 közötti elemekre képezi, úgy, hogy az elemek összege 1 legyen (*logits*). Azaz, mintha vennénk a figyelem pontok egy valószínűségi eloszlását:

$$softmax(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}} \tag{1.26}$$

A vissza skálázásra azért van szükség, mert a szorzat, vagyis a logit-ok varianciája d_k szorosukra nőnek, és ez telítésbe viheti az *softmax*-ot. Ezért visszaskálázzuk a szorzat eredményét, hogy a szórás ne változzon.

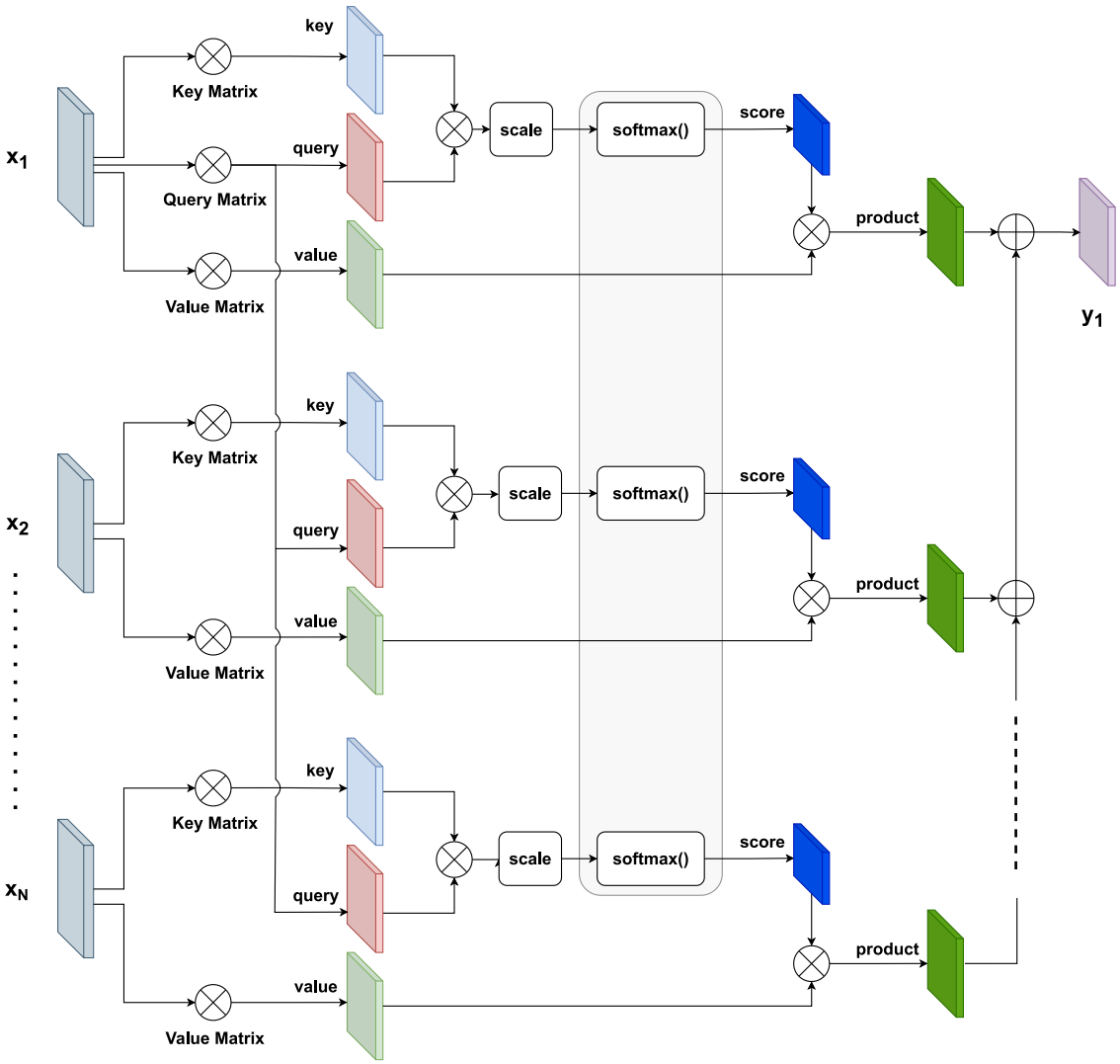
A következő lépésben a *value* reprezentációkat súlyozzuk a kiszámolt figyelem pontokkal, majd ezeket az $1 \times d_v$ méretű súlyozott *value* vektorokat összeadjuk elemenként. Az így megkapott vektor az első bemeneti elemhez tartozó $\boldsymbol{y_1}$ reprezentációja. Láthatjuk, hogy a kimenet méretét a $\mathbf{W^V}$ mátrix oszlopainak számával határozhatjuk meg. Legvégül ezt a lépés sorozatot megismételjük a szekvencia minden elemére és így megkapjuk a kimeneti vektorokból álló $\boldsymbol{Y}$ $N \times d_v$ méretű mátrixot.

Vegyük észre, hogy az itt bemutatott műveletek tulajdonképpen mátrixszorzatok. Vagyis a fenti metódus tömörebben a következő formulát valósítja meg:

$$\boldsymbol{Y} = Self - Attention(\boldsymbol{K}, \boldsymbol{Q}, \boldsymbol{V}) = softmax\left(\frac{\boldsymbol{Q}\boldsymbol{K}^T}{\sqrt{d_k}}\right)\boldsymbol{V} \tag{1.27}$$

Mint említettem, a Self-Attention lényegében egy egyszerű előre csatolt neurális háló. A $\mathbf{W^K}$, $\mathbf{W^Q}$, és $\mathbf{W^V}$ súlymátrixok a háló paraméterei, vagyis tanításkor ezeknek a mátrixoknak az elemeit frissítjük a hiba-visszaterjesztés során. Az 1.27-es egyenlet pedig

felfogható a réteg aktivációs függvényeként. A három súlymátrixot kis értékekkel szokás inicializálni, például Gauss-eloszlással.



1.7. ábra A Self-Attention működése az első bemeneti elemre.

A Multi-Head Attention innen már csak egy lépésre van: igazából az nem más, mint több Self-Attention blokk párhuzamos működése. A Multi-Head előnye, hogy így az egyes Self-Attention blokkok a szekvencia különböző jellegeire (*feature*) is képesek lesznek rátanulni. Ehhez az kell, hogy különböző súlymátrix hármasaink legyenek, tehát a mindegyik fejhez tartozó 3-3 súlymátrixot véletlenszerűen inicializáljuk. A már említett mátrixokon kívül szükség van még egy $\mathbf{W}^{\boldsymbol{O}}$ kimeneti súlymátrixra is, melynek elemeit szintén tanításkor frissíti a háló. Ez a mátrix arra szolgál, hogy a h-fejű Multi-Head Attention h darab kimeneti $\boldsymbol{Y_i}$ mátrixaiból konkatenációval képzett $N \times h \cdot d_v$ mátrixot jobbról szorozva kapjuk a Multi-Head $N \times d_o$ méretű $\boldsymbol{Z}$ kimeneti mátrixot $\boldsymbol{X}$ bemenetre.

1.3.2 Pozíció kódolás

A Transformer hálók egyik tulajdonsága, mely egyszerre lehet előny is és hátrány is különböző alkalmazások esetén, hogy mivel nem szekvenciálisan, hanem egyszerre dolgozza fel a bemenetet, ezért invariáns a szekvencia elemeinek sorrendjére. Tehát nem kódolja az elemek pozícióit, hanem mintegy halmazként kezeli csak őket. Például ez tipikusan a fordítóknál vagy más nyelvi feldolgozásnál hátrányt jelenthet, ha a kimenet független a bemeneti elemek sorrendjétől, hiszen egy mondat jelentése többnyire függ a szavak sorrendjétől. Ilyen esetekben érdemes kódolni a pozíciókat és ezt az információt is átadni az *attention* rétegnek.

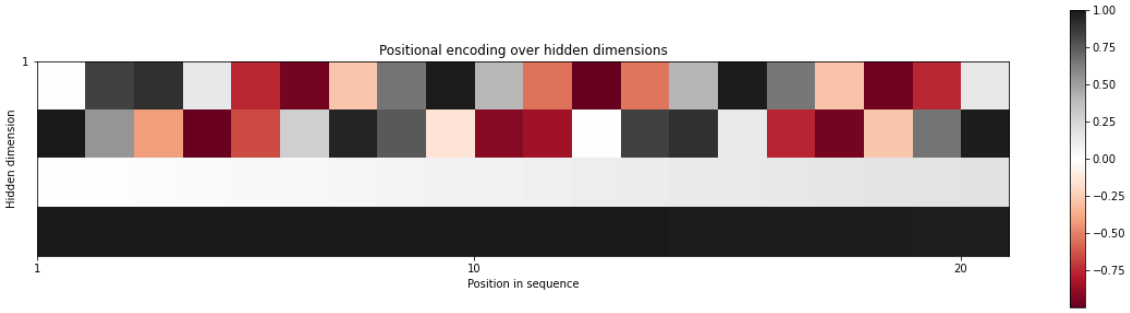
De miként válasszuk meg a kódolás metódusát? Ugyanis egyáltalán nem mindegy, hogy milyen módszert használunk, egy rossz módszer könnyedén lehet kontraproduktív. Néhány példát bemutatok, hogy mikre érdemes figyelni, mielőtt rátérnék arra a módszerre, amit alkalmaztam is. Először is, ha mindegyik bemenethez hozzáadok egy egész számot, például a pozícióját a sorozatban, akkor egy nagyon hosszú sorozatnál az utolsó vektorokhoz már akár egy több nagyságrenddel nagyobb számot adunk hozzá. Ez eltéríti a figyelem pontokat a *softmax* miatt. Egy másik ötlet lehet, mondjuk, ha 0 és 1 közé skálázott, a valós számok tengelyén egymástól egyenlő távolságra lévő pontokat adunk a vektorokhoz. Így nincs nagyságrendbeli változás, viszont, ha különböző hosszúságú szekvenciák lehetnek a bemenetek, akkor ugyanahhoz a pozícióhoz különböző hossznál különböző kódot rendelünk adunk. Emiatt a háló nem lesz képes megtanulni a kódolt pozíciókat, ezért, ha ezt szeretnénk használni, akkor garantálni kell, hogy a bemenetek fix hosszúságúak legyenek. Tehát a két példából láthatjuk, hogy a kódolási technikánál, amit alkalmaznánk egy adott pozíciónak mindig legyen ugyanannyi a kódja, a szekvencia hosszától függetlenül, a kód véges intervallumon belül vegyen fel értéket és különbözzön minden pozícióra.

Egy ismertebb megoldás, mely megfelelő kódolást állít elő, trigonometrikus függvényeket használ, különböző hullámhosszokkal. Először kódoljuk a pozíciókat a szinuszgörbe értékei szerint: $PE = \sin(p)$. Így a kód független a szekvencia hosszától, véges intervallumon belül vehet csak fel értéket, viszont nem egyediek a kódok, ismétlődni fognak a szinusz periodikussága miatt. Ekkor használjuk fel azt, hogy a bemeneti vektorok elemeit is külön kódolhatjuk és így nem csak a pozíciók mentén, hanem a rejtett dimenziók mentén is kódolunk. Minden egyes rejtett dimenzió mentén ugyanúgy teljesülni kell, a korábban említett tulajdonságoknak, így adódik a megoldás,

hogy mindegy dimenzió mentén használjuk a szinuszgörbét, viszont különböző frekvenciával. Ha jól választjuk meg a frekvencia változtatását, akkor garantálhatjuk, hogy két különböző pozíció kódolása különböző is lesz, mivel, ha egy dimenzió mentén azonos is lenne a szinuszgörbén felvett érték, egy másik dimenzió mentén már biztosan nem lesz az. Ez a kódoló eljárás az alábbi módon néz ki:

$$PE(p,i) = \begin{cases} \sin\left(\frac{p}{10000^{i/d}}\right), & i \equiv 0 \; mod \; 2 \\ \cos\left(\frac{p}{10000^{(i-1)/d}}\right), & i \equiv 1 \; mod \; 2 \end{cases} \tag{1.28}$$

Az egyenletben szereplő p jelöli a pozíciót, d a rejtett dimenzió méretét, az i az adott rejtett dimenzió indexe. Látható, hogy p mentén szinuszosan változnak az értékek, míg a rejtett dimenzió mentén a szinuszok hullámhossza exponenciálisan nő. A páros dimenziók mentén szinuszgörbének veszi a pontjait, míg a páratlan indexű dimenzióknál a koszinusz függvényt értékeli ki. Az alábbi képen látható az ábrázolt kódolás 20 elemű bementre nézve, melyben a vektorok rejtett dimenziója 4. Az első két dimenzió mentén megfigyelhető a 2π hullámhosszú szinusz- és koszinuszgörbe, míg alatta a $2\pi\sqrt{10000}$ hullámhosszú görbék. Vegyük észre, hogy nincs két egyformán kódolt pozíció, azaz két egyforma oszlop.



1.8. ábra A kód előállítása 20 elemű szekvenciára, az elemek rejtett dimenzióinak a száma 4

1.4 RAdam

A Rectified Adam egy módosított Adam (Adaptive Moment Estimation [11]), ami egy iteratív, negatív gradiens alapú optimalizáló algoritmus, ez state-of-the-art optimalizáló eljárás. Azonban mielőtt rátérnék, hogy miért jobb a Rectified Adam, előbb nézzük meg, hogy működik az egyszerű Adam.

Az Adam két ismert algoritmus, az RMSProp és az AdaGrad jó tulajdonságait ötvözi. Célja a nevéből is adódóan az adaptív tanulási sebesség, akárcsak az RMSPropnál viszont itt a gradiens négyzetek összegzésén kívül a gradienseket is összegezzük, és kijavítja az AdaGrad nagy hátrányát: az időben végül majdnem nullára csökkenő tanulási sebességet. A függvény paraméterei, az α, mely nem más, mint a tanulási ráta vagy lépéshossz (szokták η-val vagy λ-val is jelölni), a β_1 és β_2, melyek a gradiensek első (átlag) és második momentumának (középnélküli varianciája) exponenciális felejtési rátája. Ez utóbbiak 1 körüli értékek, míg az α egy nagyon kicsi szám. Ezeken kívül szükség van még az epszilonra (ε), mely a numerikus stabilitást biztosítja, azaz, hogy a nevező értéke sose lehessen nulla. Az Adam-et kiötlők ajánlásai alapján ezeket a következő módon szokás beállítani: $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ és $\varepsilon = 10^{-8}$. Ha megnézzük az Adam PyTorch-os implementációját, default paraméterként ugyanezeket az értékeket fogjuk látni. Ezeket a paramétereket felhasználva tudjuk kiszámolni az első és második momentumot. Természetesen szükségünk van még a gradiens vektorra, melyet megkapunk a költségfüggvény θ szerinti deriváltjából, ahol a θ a háló paraméterei. A *t* alsó index az időlépést, azaz az időbeli iterációt jelöli.

$$g_t = \nabla_\theta J(\theta_t) \tag{1.29}$$

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \tag{1.30}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t \odot g_t \tag{1.31}$$

Egy további korrekciót kell még alkalmazni, ha netán a gradiensek átlaga és a gradiens négyzetek átlaga kezdetben nagyon kis értékűek lennének, akkor van rá esély, hogy beragadnak ilyen kis értéken. Ezért korrigálunk a bétákkal („bias-corrected” mozgó átlag és mozgó második momentum), így a kezdeti értékek ($t = 0$) a gradiensek és gradiens négyzetek lesznek (Hadamard/elemenkénti szorzatuk), így az egyenletek a következőképpen alakulnak:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \tag{1.32}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{1.33}$$

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t \tag{1.34}$$

A probléma az Adammal, hogy kezdetben nagy a lépésköz varianciája, melyet jó lenne csökkenteni a gyorsabb konvergencia érdekében. Erre az egyik módszer a *warmup* (AdamW), azaz, hogy a tanulási ráta nem egy konstans, vagy egy csökkenő érték (*learning rate decay*), hanem egy bizonyos T ideig kezdetben egy nagyon kicsi értékről növeljük az alfát, ezzel csökkentve a varianciát. A *rectified* ezzel szemben úgy oldja meg ezt a problémát, hogy először kiszámoljuk az egyszerű mozgó átlag közelítésének (SMA) a maximum hosszát, melyet ρ_∞ -val jelölünk. Majd ezt felhasználva minden iterációban kiszámoljuk az közelített SMA hosszát (ρ_t) és ha ez átlép egy általunk megválasztott küszöböt, akkor változtatunk a tanulási rátán, pontosabban beszorzunk egy ún. *variance rectification* (1.38 egyenlet) taggal. Egyéb esetben csak alfa konstans együtthatóval súlyozzuk az első momentumot (1.40 egyenlet).

$$\rho_\infty = \frac{2}{1 - \beta_2} - 1 \tag{1.35}$$

$$\rho_t = \rho_\infty - \frac{2t\beta_2^t}{1 - \beta_2^t} \tag{1.36}$$

$$l_t = \frac{1}{\sqrt{\hat{v}_t}} \tag{1.37}$$

$$r_t = \sqrt{\frac{(\rho_t - 4)(\rho_t - 2)\rho_\infty}{(\rho_\infty - 4)(\rho_\infty - 2)\rho_t}} \tag{1.38}$$

$$\theta_{t+1} = \theta_t - \alpha r_t l_t \hat{m}_t \tag{1.39}$$

$$\theta_{t+1} = \theta_t - \alpha \hat{m}_t \tag{1.40}$$

2 Specifikáció, tervezés

2.1 Specifikáció

Alapvetően a feladatom megalkotni egy rendszert, amely képes az ágens autonóm viselkedését megvalósítani. Ehhez én megerősítéses tanulást alkalmazok, amelyben az ágens modelljét Deep Learnig módszerrel valósítom meg. Ez a módszer a stratégia gradiens és a Q-learning eljárások ötvözésén alapuló Actor-Critic neurális hálózat. Viszont tanítás szempontjából az egyik legjobb ilyen algoritmus az A2C, mely gyorsabb konvergenciához és stabilabb tanuláshoz vezet, mint az A3C, így ezt alkalmazom.

Második sarkalatos pontja a dolgozatnak a szimulációs környezet kialakítása. Itt figyelni kell, hogy olyan környezetet, könyvtárat használjunk, melyben könnyedén lehessen implementálni a számunkra fontos funkciókat. Az ágens képes legyen benne tanulni, valamint módosítható legyen a környezet felépítése, létre lehessen hozni benne a szükséges objektumokat. Az ágens bemenetére tudjunk képet (RGB-D) adni, vagy akár más szenzorok jelét is, ezenkívül pontos fizikával rendelkezzen. Fontos még, hogy tudjunk renderelni, hogy ellenőrizni tudjuk az ágens akcióit, a környezet állapotait. Az ágens modelljét meg tudjuk tervezni, például, hogy hasonlítson a távirányítós autónkra.

A tanítás szempontjából nem hanyagolható el a megfelelő hardverek jelenléte a számításigény végett. A megerősítéses tanulás az egyszerűbb osztályozásnál is nagyságrendekkel több tanítási ciklust vár el, így szükségünk van az erős GPU/TPU-ra, ha nem szeretnénk hetekig tanítani. A fejlesztés során rengeteg ellenőrző tanítást kell végezni, amely lassíthatja a fejlesztést, ha egy-egy eredményre órákat kell várni. Továbbá a fejlesztéshez szükség van a számunkra megfelelő függvénykönyvtár és fejlesztői környezet kiválasztására is.

Legvégül tesztelni kell az implementált algoritmus pontosságát és hatékonyságát. Alapvetően cél az, hogy a szimulációban (de igazából a célhardveren is) effektíven futna a szoftver és pontos, gyors döntéseket tudna hozni az ágens. Valósidejűség szempontjából mérni kell, hogy a háló kiértékelése (*inference*) mennyi időt vesz igénybe. Egy önvezető autónál a cél nyilvánvalóan az lenne, hogy a döntései gyorsabbak lennének, mint az emberi döntések, amelyek néhány 100 ms-ok. A kamera 30 FPS-s, azaz 33 ms-omonként következnek az új állapotok, melyekre kellene reagálnia az ágensnek. Tehát a legjobb lenne ez alá vinni a futásidőt.

2.2 Fejlesztői eszközök

2.2.1 Colaboratory

A Colaboratory (a továbbiakban Colab) a Google ingyenes Jupyter jegyzetkezelő környezete [12] [13]. Egyszerűen használható Python kódok futtatására. A felhő alapú szolgáltatás mögött egy Linux rendszer áll, amelyre tölthetünk fel-le adatokat, futtathatjuk a kódunkat, akár GPU-n, sőt TPU-n (Tensor Processing Unit) is.

Legnagyobb előnye a Colab-nak, hogy a hardware erőforrásai nagyságrendekkel erősebbek, nagyobb számítási kapacitással rendelkeznek, mint egy átlagos otthoni PC vagy laptop konfigurációja. Így ingyenes elérhető az NVIDIA Tesla K80 videókártyákat tartalmazó gépei (fizetős verziónál akár T4-et és P100-at is használhatnánk). Ezek a GPU-k sokkal nagyobb teljesítményűek és több memóriával is rendelkeznek, így ideálisabb ezeket a hardware-es gyorsítókat használni tanításnál. Ez által a kódok CUDA futtatása is nagyságrendekkel gyorsabb Colab-ban [14].

Későbbiekben kifejtem, hogy egyelőre miért kell mellőznem a használatát a fejlesztés során. Viszont a későbbi tesztelések során már nem kell megkerülnünk, például így biztosan sor fog kerül a használatára. Addig is lokálisan végzem a tanításokat, a JetBrains Python fejlesztőikörnyezetét, a PyCharm IDE-t használom [15].

2.2.2 PyTorch

A PyTorch egy Python [16] alapú nyílt-forráskódú tudományos könyvtár gépi tanulásos számításokhoz [17]. Vannak más hasonló könyvtárak, mint például a Keras vagy a TensorFlow, mindegyik másban jobb vagy rosszabb a másiknál. A Keras leginkább kezdőknek hasznos, mivel egyszerűen tanulható, cserébe nagyon korlátozott (specifikusan csak neurális hálózatok fejlesztésére találták ki) és lassú. Ezzel ellentétben a PyTorch alacsonyabb szintű, ezért nehezebb is kezelni, de sokkal gyorsabb, főleg nagy adathalmazokra, és több mindenre lehet felhasználni. A TensorFlow a PyTorch-nál is alacsonyabb szintű, így még nehezebb ügyesen kezelni.

Én a PyTorch mellettem döntöttem, mivel a feladathoz hozzá tartozik, hogy bele kell tudni avatkozni a háló működésébe alacsony szinten is már. Ezenkívül rengeteg hasznos dokumentum található meg hozzá, leírások, példamunkák stb. Az sem elhanyagolandó szempont, hogy az algoritmusok, különböző komponensek melyeket felhasználunk a projektben többnyire szintén PyTorch felhasználásával készültek. Egy

ilyen könyvtár leghasznosabb tulajdonsága, hogy könnyeddé teszi a többdimenziós tömbökön, vagyis a tenzorokon végzett műveletek számítását GPU segítségével, továbbá rengeteg olyan függvény és osztály van implementálva, melyeket fel szoktak használni gépi tanulás alkalmazások fejlesztésénél. A számunkra legfontosabb könyvtára a *torch.nn*, ebben speciálisan neurális hálók fejlesztését megkönnyítő osztályok és függvények állnak a rendelkezésünkre. Hatalmas mértékben gyorsítja a hálók fejlesztését, ráadásul átláthatóbb és hordozhatóbb kódunk lesz, ha ezeket az alapfüggvényeket és osztályokat alkalmazzuk. Ebben találhatóak meg a konvolúciós, lineáris és más egyéb, például visszacsatolt rétegeket megvalósító osztályok, aktivációs függvények, költségfüggvények. Valamint a Multi-Head Attention függvény implementációja is.

2.2.3 PyBullet

A PyBullet [18] egy olyan Python csomag, melyben különböző szimulációs környezetekben végezhető gépi tanulásos implementációk találhatóak, ezenkívül modellek és környezetek sora is a rendelkezésünkre áll, alapja az ingyenesen elérhető Bullet fizikai motor. Többnyire megerősítéses tanulásnál használják az effajta környezeteket, ebben szimulálhatóak az ágens akciói, és az ebben szimulált környezet állapotaira reagál az ágens.

A rendelkezésünkre áll néhány, a fejlesztők által elkészített környezet, melyeken viszonylag könnyedén tudunk változtatni, a saját feladatunkra szabni. A környezetek támogatnak folytonos és diszkrét akciókat. További előnyei még, hogy egy részletes útmutató érhető el hozzá, valamint egyre többen használják, így folyamatosan fejlesztik is. Egy komolyabb hátránya van: Sok funkció nincs még implementálva, így jónéhány függvénnyel találkoztam, mely igazából még üres.

2.3 Tervezés

Legelőször az önvezető algoritmust kellett megtervezni, kezdetben érdemes volt keresni egy jól megírt és könnyedén használható implementációt. Első körben a Stable Baselines kódjait használtam fel, de itt két akadályba is ütköztem. A kisebb gond az volt, hogy a TensorFlow függvénykönyvtár felhasználásával íródott a modell, mellyel összeegyeztethetőségi problémák voltak a PyTorch miatt. Ennél nagyobb gondot okozott az, hogy rengeteg absztrakt függvényt kellett volna implementálni, erre pedig akkor nem terveztünk időt szánni. A Stable Baselines az OpenAI Baselines függvénykönyvtárán

alapuló megerősítéses tanulást könnyítő függvények implementációit tartalmazza, a kezdőbb fejlesztők számára ajánlják, mivel a Baselines-nál stabilabb működést biztosít. Ennek a része a *SubprocVecEnv* osztály, mellyel könnyedén vektorizálhatjuk a környezeteinket. De mivel a Baselines nem PyTorch, hanem TensorFlow segítségével íródott, továbbra is elég sok gond volt az összeegyeztetéseknél, de végül sikerült a hibákat kiküszöbölni. Végül egy diáktársam kódját használtam fel, melyen elvégeztem a szükséges módosításokat, hogy kompatibilis legyen a környezettel, valamint teljesen átdolgoztam az Actor-Critic struktúráját.

A projekt során több környezettel is próbálkoztam. Először az OpenAI SafetyGym környezetével, melyről sajnos kiderült, hogy a MuJoCo fizikai motort használja, amelyre ingyen csak diákként lehet licenszt szerezni, viszont maximum csak egy évre és egy adott gépre lehet kérvényezni. Ez kizárja annak a lehetőségét, hogy a Colab-on igénybe vehessük a SafetyGym-et a motor miatt, valamint a jövőbeli fejleszthetőséget kockáztatjuk meg azzal, ha nem kapunk később licenszt, vagy csak az eredeti árán, mely a dolgozat írásakor 500€. Így másik motor és környezet után kellett nézni, míg végül a Bullet alapú PyBullet-re esett a választás.

A PyBullet szimulációs környezete teljesen ingyenes mindenkinek, így Colabon is lehetne elvileg futtatni. Azonban itt több problémába is ütköztem. Ha szeretnénk debuggolni az ágenst, vagyis, szeretnénk nyomon követni az ágens akcióit a környezetben, akkor igen hasznos funkció lenne, ha meg tudnánk jeleníteni a környezetet. Így az egy fontos szempont volt, hogy a Colabon meg tudjuk ezt valósítani. Sajnálatos módon, mint kiderült ez egy nagyon nehezen megvalósítható funkció. Egyszerűbb környezetekkel, mint például az Atari ezt sikerült elérni oly módon, hogy egy függvény videót készít a környezet állapotairól, az elvégzett akciókról és a futás végén ezt kimenthetjük, visszanézhetjük. De a PyBullet egy SDK-ban (Software Development Kit) fut, melyet a Colab-bal nehéz megnyitni és sajnos nem lehet videóra rögzíteni sem. Így a Colab-ról egyelőre le kell mondanunk, amíg nem lesz tökéletes a szimulátor működése és nem lesz kész hozzá a környezet, hogy mellőzni lehessen a szimuláció renderelését.

3 Megvalósítás

Ebben a fejezetben bemutatom, hogy a második fejezetben felvonultatott ismereteket, algoritmusokat miként alkalmaztam a gyakorlatban a feladatom elvégzéséhez. Előbb magát megerősítéses tanulást megvalósító modellen megyek végig részletesen, majd az általam használt szimulációs környezet felépítéséről és módosításairól írok. Végezetül az implementált jutalmazó függvényeket is bemutatom egyesével.

3.1 Architektúra

Előbb a teljes architektúrát részletesen bemutatom, célom végig vezetni az olvasót a főbb építőelemein.

3.1.1 Felépítése

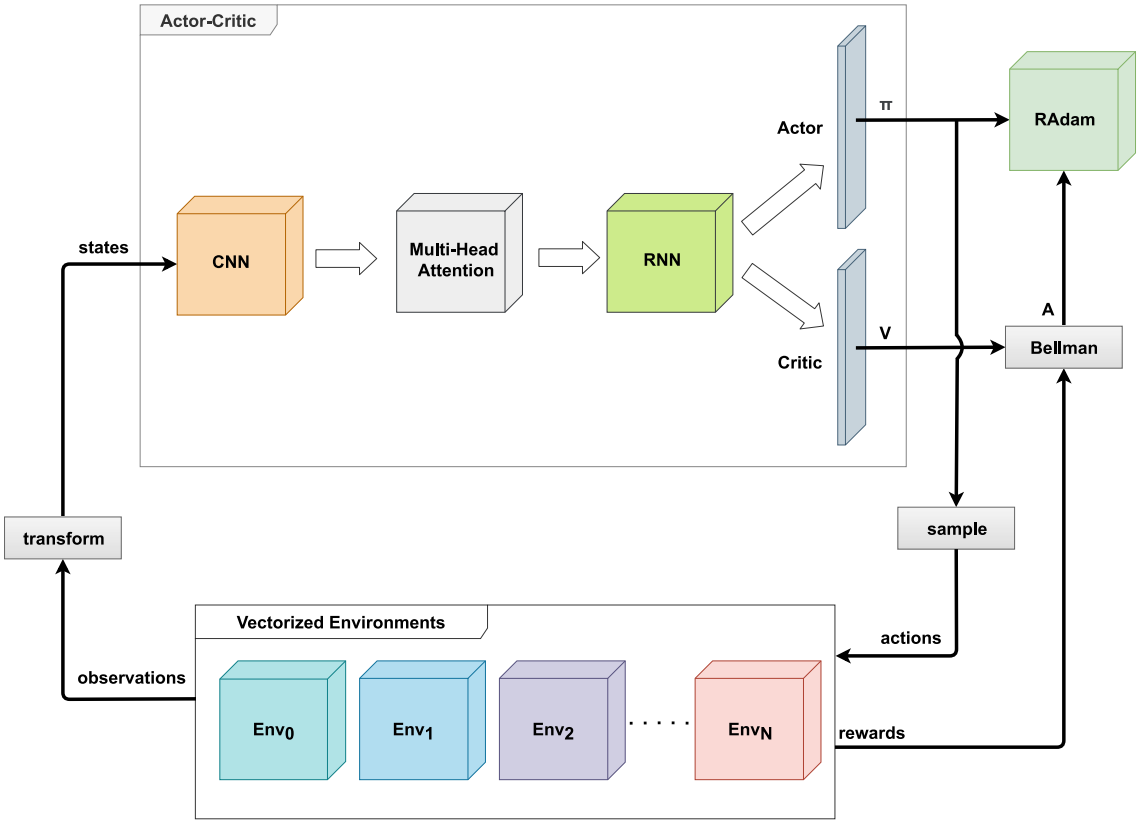
Az általam választott, megerősítéses tanulást megvalósító implementáció az A2C stratégia gradiens metódus lett. Az architektúra több kisebb logikai komponensre bontható: A rendszer magja az Actor-Critic algoritmus, melyet egy kétfejű mély neurális hálózattal valósítok meg. Az ágens bármilyen algoritmusból állhat, de érdemes olyat választani, melynek a paraméterei tanulhatóak és végig hátra tud menni a gradiens a modell bemenetéig. Az Actor fej kimenete a stratégia, vagyis az akciók eloszlása, melyből mintavételezünk akciókat, melyeket végrehajt az ágens az adott környezetekben. A Critic fej kimenete a szokásos állapot-érték függvény.

Az ágens bemenete a környezetekből érkező megfigyelésekből képzett állapotok tenzora. Mivel A2C-t használunk, így logikus több környezetet futtatnunk egyszerre, határt csak a hardware erőforrásaink szabhatnak, én párhuzamosan maximum négyet tudtam futtatni. A vektorizált környezetek egyik kimenete a megfigyelések, melyek RGB-D képek csomagja, melyet transzformálni kell, mielőtt átadom az Actor-Critic hálónak. Ennek az oka, hogy a több párhuzamosan futó környezet egyszerre több képet is visszaad (később*: roll-out* fogalma). Mivel a modellben bemenetén megtalálható CNN struktúra NCHW formátumú bemenetet vár (batch méret $\times$ csatorna szám $\times$ kép magasság $\times$ kép szélesség), ezért mint egy batch-be össze kell csomagolni az összes képet, így a batch mérete környezetek száma $\times$ framek szám lesz.

A környezetek másik kimenete az aktuális jutalom, melyet a Critic fej kimenetén kapott állapot-értékkel együtt felhasználva a Bellman-egyenlettel előállítom az előny függvényt (2.15 egyenlet). Ez lényegében egy diszkontálást jelent, de ezenkívül itt egy kisebb trükköt is kell alkalmazni. Az előny mínusz egyszeresét kell venni (vagyis az állapot-értékből kell kivonni az akció-érték függvényt), mivel a jutalmat maximalizálni szeretnénk, ezért az optimalizáláskor nem minimumkeresést hajtunk végre, hanem a globális maximumot szeretnénk megtalálni.

A háló kimenetén egy RAdam optimalizáló algoritmus található, mely a háló jelenlegi paramétereit és a veszteségfüggvény gradienseit (stratégia gradiens) felhasználva számolja ki a háló új paramétereit a hiba-visszaterjesztés segítségével. Ehhez csak a hálótól (pontosabban az Actor-tól) kapott stratégiára (és annak entrópiájára) és az előny függvényre van szüksége, mivel ezeket használom fel a veszteség (költség) kiszámításához.

Az alábbi képen látható a felépített A2C struktúra, melyet érdemes összevetni az alap **1.4. ábra** látható működéssel.

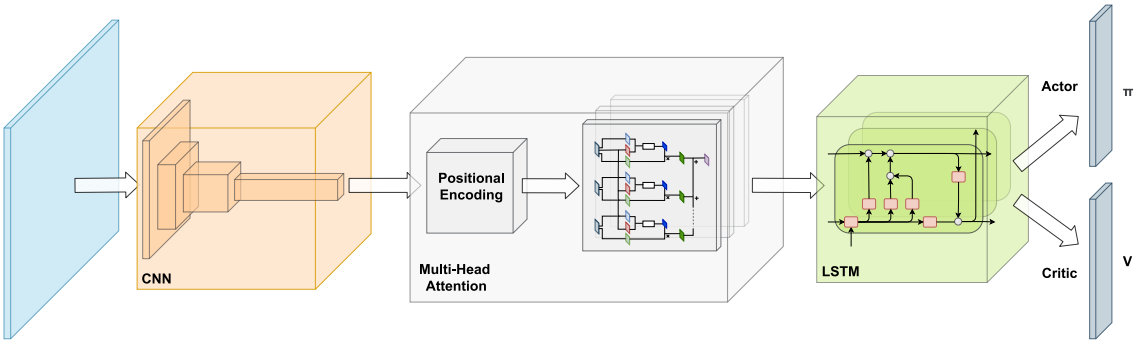


3.1. ábra A2C architektúra

3.1.2 Ágens

Az architektúra lelke, vagyis maga az ágens az Actor-Critic neurális hálózat. Ez a módszer alkalmas az optimális stratégia megtanulására.

A hálót alapvetően két egységre lehet bontani: Az első fele a bemeneten kapott állapotot kódolja, egy belső reprezentációt rendel hozzá (*feataure* enkóder), míg a második fele igazából a két fejet takarja, amely ebből kódolt jellegekből előállítja az optimális stratégiát és az állapot-érték függvényt. Az utóbbi komponens általában egyszerű, teljesen-összecsatolt osztályozó rétegeket tartalmaznak. Diszkrét esetben az Actor fej egy N_A (a lehetséges akciók száma) neuronokból osztályozó, míg a Critic fej egyetlen skalárt ad meg, az állapot-értéket. Ez a része a hálónak viszonylag egyértelműen adódik, de egyáltalán nem evidens, hogy a kódoló része hogyan épül fel.

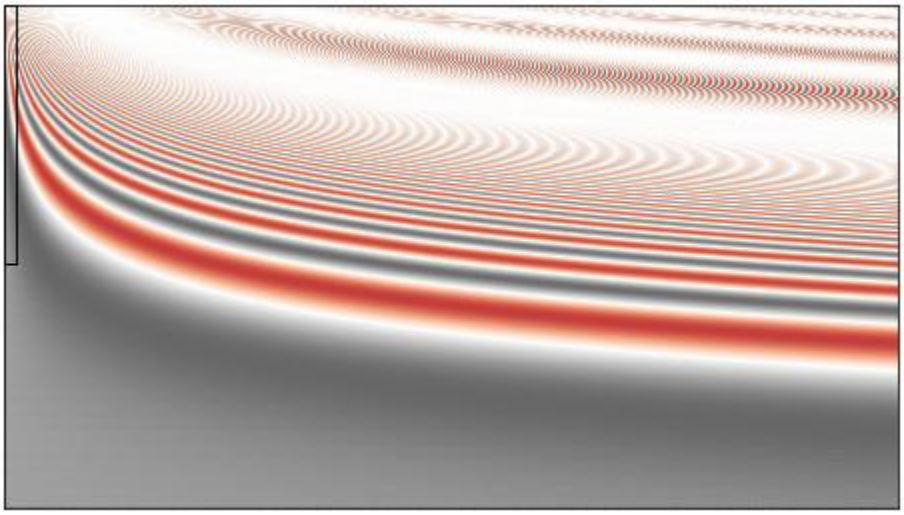


3.2. ábra Actor-Critic modell

A modell, azaz a kódoló első komponense jelen esetben egy CNN hálózat, mely a bemenetén kapott NCHW formátumú képek tenzorait egy H hosszú vektorba képezi le. A dimenziócsökkentést négy darab egymást követő kétdimenziós leskálázó (*strided*) konvolúcióval [19] érem el, tehát *pooling* réteget nem alkalmazok. A konvolúciós rétegek aktivációs függvényeiként *LeakyReLU* nemlinearitást alkalmazok, mely hasonló a *ReLU* függvényhez, de a negatív tartományban nem nullát rendel a bemenethez, hanem egy kicsi negatív számot. Végül kicsomagolom a képeket, hogy ismét külön kezelhessük a különböző környezetektől kapott frameket. Ekkor a reprezentáció tenzorja $N_E \times N_F \times$ H méretű, ahol az első dimenzió a környezetek száma, a második a framek száma.

A következő két blokk opcionális, hogy különböző struktúrákat össze tudjak hasonlítani a tanítások során. Az első kikapcsolható blokk a Multi-Head Attention réteg [20], mely előtt még alkalmazzuk a korábban említett pozíció kódolási technikát. A bemeneti szekvencia hossza a képek száma (N_F), hiszen a képek pozícióit szeretnénk

kódolni, a környezetek mentén azonos lesz a kódolás (nyilvánvalóan nem szeretnénk különbséget tenni a független környezetek között). A rejtett dimenziók száma H, mely egy-két nagyságrenddel nagyobb, mint a képek száma. A **3.3. ábra** látható egy példa, melyen ábrázoltan, hogy hogyan alakulnak a kódok egy olyan beállításnál, mely egy maximum 1920 hosszú és 1080 rejtett dimenziójú szekvenciát képes kódolni. Az így kapott Full HD képen már jobban ki lehet venni az exponenciálisan nővő hullámhosszokat, a **1.8. ábra** szemben. A balfelső sarokba rajzolt téglalap jelzi az általam választott dimenziók esetén a kódolási beállítást, ahol $H = 512$ és a szekvencia hossza, azaz a képek száma: $N_F = 5$. Az arányok nem teljesen stimmelnek, de így jobban látszódik a téglalap.



3.3. ábra Pozíció kódolás, feltüntetve az általam használt kódok halmazát

A Multi-Head Attention megvalósításához a PyTorch implementációját használtam fel. A tanításokat egy két fejű rétegen végeztem el, a dimenziókat úgy állítottam be, hogy a bemenet dimenzió ne változzanak.

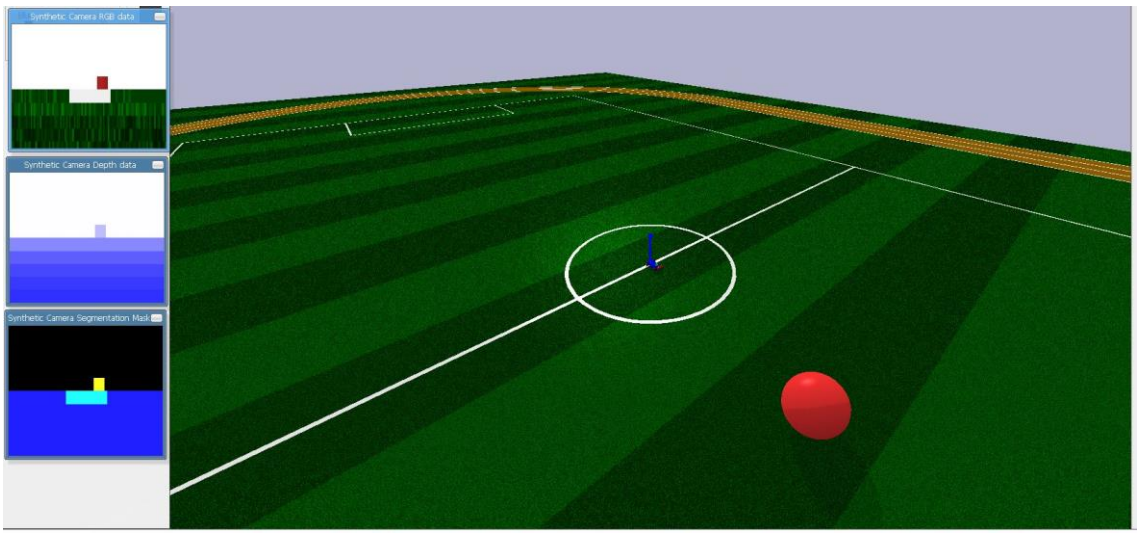
Az Attention blokkot követi opcionálisan egy LSTM cella (Long Short-Term Memory [21]). Az opcionalítás oly módon van beállítva, hogy a Multi-Head Attention után mindenképp az LSTM réteg következzen. Tehát összesen három módban futtatható az ágens: az enkóder csak a CNN blokkból áll, vagy a CNN és LSTM blokkokból áll, vagy az összes blokk be van kapcsolva. Mivel az LSTM és emiatt az enkóder kimenetén lévő lineáris osztályozók is egy a környezetenként kódolt *feature* vektort vár (az LSTM felépítéséből adódóan nem változtat a dimenziókon), ezért még egy transzformációra szükség van. Mivel a képek menti változás információját is a vektorba kell tömöríteni ezért Attention használata esetén a képek mentén átlagolok, míg a többi esetben

egyszerűen csak kiválasztom az utolsó kapott frame reprezentációját és azt adom bemenetként az LSTM cellának vagy már közvetlenül a lineáris rétegeknek. Így az enkóder kimenete egy $N_E \times$ H méretű mátrix lesz.

3.2 Szimulációs környezet

A célhardveren tanítani, illetve folyton tesztelni a hálót egyrészt lassú lenne, mert felesleges overhead-et okozna az integráció, hiszen mindig jelen kéne lenni a laborban. Másrészt igencsak költséges is lehet, hiszen, egy komolyabb hiba vagy rossz döntés miatt kárt tehet magában és a környezetében is a versenyautó. Ezért célszerű a szoftver működését egy jó fizikai motorral szimulált részletes környezetben tanítani és tesztelni, ahol ezek a hátrányok természetesen nem jelentkeznek.

A kitűzött feladat elvégzéséhez a PyBullet fejlesztői kettő hasonló környezetet építettek már ki. Mindkettőben az ágens egy kis távirányítós versenyautó, ennek kell eljutnia egy nagy üres pályán (eredetileg egy focipályán) a pálya közepétől egy a pályán véletlenszerűen elhelyezett labdához. A két környezet abban különbözik leginkább, hogy míg az egyiknél a megfigyelés csak a labda pozíciója a kamera képén (x, y), addig a másiknál a teljes RGB-D kamera kimenete. Az utóbbi környezetet fejlesztettem tovább, a környezet teljesen új modellekből épül fel, kijavítottam és módosítottam szükséges függvényeket, valamint új funkciókat hoztam létre és teljes új jutalmazó függvényt is írtam, melyet később részletesen bemutatom.



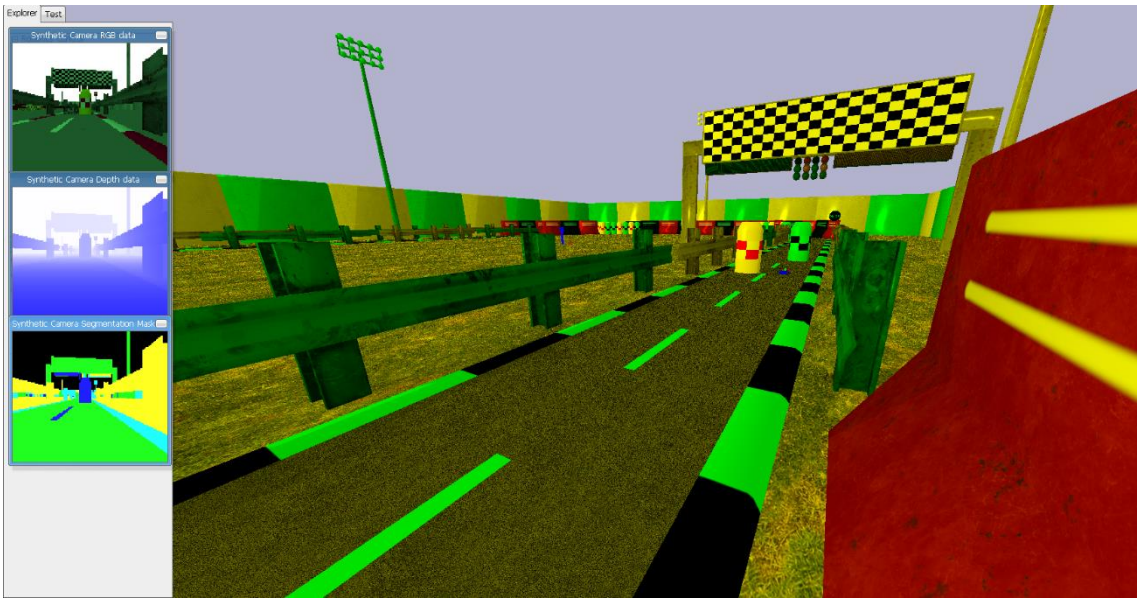
3.4. ábra Az eredeti PyBullet-es környezet, bal oldalt a kamera képe látható (felülről lefelé: RGB, mélység, szegmentált)

3.2.1 Objektumok beolvasása

A PyBullet egyik legnagyobb hátránya, hogy meglehetősen limitáltan tudja beolvasni az objektumokat és elhelyezni a környezetben. Alapvetően két fájl típust tud kezelni, az URDF és SDF fájlokat, ezek mind XML alapúak. Az előbbi a bonyolultabb objektumokat írja le, mozgatható egységekkel, csuklókkal stb. Tipikusan a robotok leírását szolgálja, alapvetően ezek lesznek az ágensek a szimulációban, míg az utóbbi az egyszerűbb tereptárgyak jellemzőit írja le. Ezek olyan tulajdonságokat tartalmaznak, mint például a tömeg, súrlódás, pozíció és orientáció, szín, anyagjellemzők stb. A PyBullet-nél csak kevés ilyen előre elkészített modell volt számomra hasznos, így csak a versenyautó URDF modelljét használtam fel. A többi objektumot kezdetben kézzel próbáltam elkészíteni, vagyis XML fájlokat írtam, melyekbe ingyenesen beszerzett OBJ fájlokat linkeltem. De rengeteg időt vett volna igénybe az össze tereptárgyat ily módon elkészíteni, ezért alternatív megoldások után néztem. Végül egy nemrégiben implementált osztály lett a segítségemre, melyben található egy olyan metódus, mely WORLD típusú XML fájlokat képes feldolgozni. Ez tömören egy tereptárgyakat (SDF) összefoglaló XML, melyben minden objektum megjelenik. Ezeket, valamint a korábban említett két fájl típust is a Gazebo 3D robot szimulátor szoftver kimenetei. Így lehetőség nyílt arra, hogy vagy tervezzek magamnak egy egész pályát, melyet egy az egyben be tud már olvasni a PyBullet, vagy egyszerűen keresek egy Gazebo-ban tervezett kész pályát. Végül találtam is egy számomra szimpatikus pályát, melyet szándékosan a PyBullet-hez készítettek el. Sok tekintetben hasonlít az elképzeléseinkre, és egyelőre tökéletesen megfelel a szimulációhoz. Ha mégis kellene később, akkor lehet változtatni rajta. Sok féle objektum található ezen a versenypályán: különböző falak és korlátok, lámpák, sávok, bóják stb. Egyetlen komolyabb hátránya, hogy maga a versenypálya egy fix objektum, és nem több részből áll, ezért jelenleg fixen futópálya alakú a versenypálya.

A felesleges tereptárgyak törlése és egyéb módosítások után még elhelyeztem két álló objektumot, valamint egy mozgó objektumot is magán a versenypályán. A versenyautót a pályán véletlenszerűen helyezem el az epizódok elején. A többi pálya elemet még fix pozícióval generálódik a környezetbe, a jövőben ezeket is el lehetne helyezni véletlenszerűen akár. Ezenkívül kiemelendő objektum még a célvonal, valamint egy közlekedési lámpa is, melyeket felhasználunk a jutalom számításához. A pályán található még egy Stop tábla is, de ezt és egyéb táblákat jelenleg még nem használunk fel az autó tanításához.

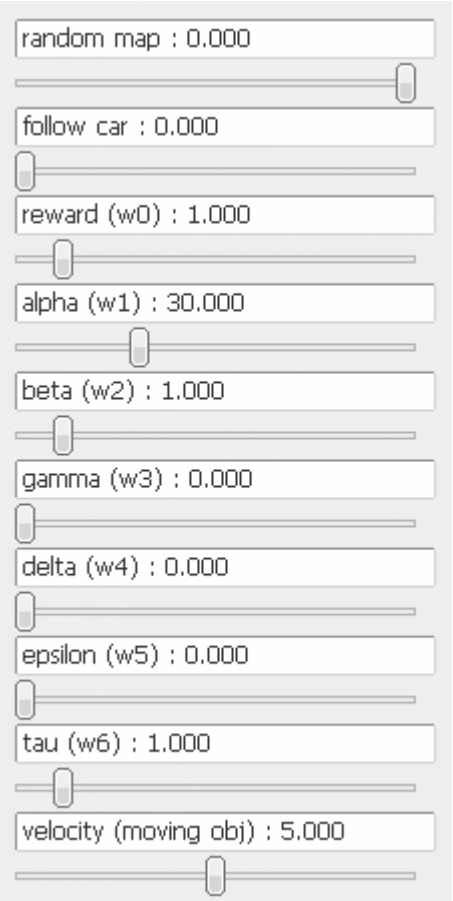
Itt megemlíteném, a PyBullet legnehezebben javítható bug-ját, a WORLD és SDF fájlok beolvasása nem tökéletes, valamiért elrontja az anyagjellemzőket. Ez abban mutatkozik meg, hogy az objektumokat felváltva sárga vagy zöld színnel „maszkolja”. Ezt sajnos nem tudtam sehogy sem javítani. Az alábbi képen az elkészült versenypálya látható.



3.5. ábra Az elkészült pálya, a hibás színkezeléssel együtt

3.2.2 UI

A PyBullet egyik hasznos tulajdonsága, hogy a kezelőfelületén gombokat és csúszkákat helyezhetünk el, melyekkel bármilyen paramétert állíthatjuk. Sajnos a gomb implementálása kissé bug-osra sikerült, így a gomb helyett is csúszka jelenik meg (például a **3.6. ábra** látható felső kettő csúszka igazándiból gombok). A felső csúszka arra szolgál, hogy szeretnénk-e azt, hogy az epizódok kezdetén véletlenszerű pályát hozzon létre a környezet vagy sem. Ezt a funkciót jelenlég még nem implementáltam. A második gombot, ha bekapcsoljuk, akkor kocsi fölött fixálja a GUI ablakát, így könnyedén tudjuk követni a kocsit a pályán. Kikapcsolva ezt a funkciót szabadon nézelődhetünk a szimulált környezetben. Az ez alatt megtalálható 7 csúszka a később említésre kerülő jutalmak súlyait állítja, melyet így szimuláció közben is hangolhatunk folyamatosan. A legalsó csúszkával a mozgó objektum sebességét változtathatjuk bármikor a szimuláció során.

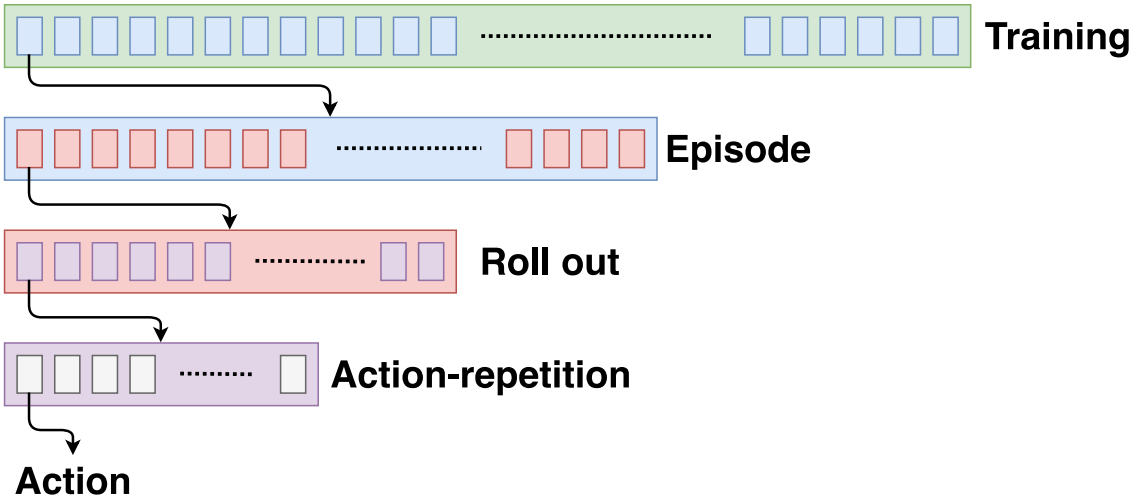


3.6. ábra Gombok és csúszkák a GUI felületén

3.3 Tanítás

A tanítás folyamata igencsak hosszadalmas lenne, ha minden állapotban egyetlen akciót vennénk csak figyelembe a költség számolásánál, ezért érdemes több akciót megvárni a modell paramétereinek frissítéséhez. Ennek a folyamatnak a megnevezése az ún. *roll-out,* mellyel megadhatjuk, hogy hány lépést várjunk meg, mielőtt kiszámoljuk költségfüggvény eredményét. Ez felfogható úgy, mint a megfigyeléses tanulásnál alkalmazott batch fogalma, ahol szintén sokat lassítana a tanításon, ha az adatokat egyesével adnánk a hálónak és emiatt adatonként kellene frissítenie a paramétereit. A roll-out másik előnye, hogy így nem szükséges a teljes epizód trajektóriáját eltárolni a memóriában, ami az A2C esetében igen nagy probléma lenne, mivel több ágenst is futtatunk egyszerre, ráadásul komplexebb feladatnál nagyon hosszadalmasok lehetnek az epizódok. De, ezesetben ugye nincs meg még a végső jutalom, tehát nem tudjuk a jutalmakat diszkontálni, ezért egy trükköt kell alkalmazni: a roll-out utolsó állapotára a Critic fej által számolt érték, vagyis az utolsó állapotban predikált jutalmat diszkontáljuk. Szokásosan kis számú akciót szokás megvárni, például öt-hatot.

Egy tanítást - mint már sokszor említettem - epizódokra bontunk (epoch), az epizód végén újraindítjuk a környezetet és ismét a nulladik állapotból indul az ágens. Egy epizód roll-out-okból áll (batch), viszont egy roll-out-ba is összefoghatunk több azonos akciót. Ennek a célja jelen esetben az, hogy összegyűjtsünk több megfigyelést is (framet) a környezetből és így a hálónak egyszerre több információt adunk át, amelyre akár számolni is tudja a figyelmet. Ezenkívül kevésbé lassítjuk így a szimulációt, mivel a többedik akció után értékeljük csak ki az állapotot.



3.7. ábra

Nézzünk egy példát: egy tanítást 100 epizódra végezzük el és 1 epizód 80 roll-out-ból, 1 roll-out 5 akció-ismétlésből álljon, valamint 1 akciót ötször ismételjünk meg a szimulációban. Ekkor a tanítás során az ágens 200 ezer akciót végez, ha 4 környezetet párhuzamosítunk, akkor máris 800 ezer akcióról beszélünk. A modell paraméterei 8000 ezerszer frissültek.

A háló kimentését a jutalom alapján végzem, a legjobban jutalmazott modell paramétereit mentem ki. A modell viszonylag kicsi, néhány Mbyte helyet foglal, így nem kellett a háló tömörítésével foglalkoznom (lásd *prune* vagy *weight sharing*). A tanítás számításigényén látszik, hogy még ez a nem túl komplex modell esetén is a fenti példát véve 100 epizód 12 órát vesz igénybe, renderelés nélkül majdnem 10 órát.

3.4 Jutalom függvény

A másik legfontosabb feladat a környezet elkészítése mellett a jutalmazó algoritmus megírása, mely eldönti, hogy az adott akcióra mekkora jutalmat ad. Ez kulcsfontosságú lépés, hiszen itt sok elvi hibát lehet ejteni, könnyű beleesni a kobra

effektus esetébe, azaz, hogy azt hisszük egy megoldási javaslat tényleg megoldja a problémát/feladatot, de igazából csak még inkább rontunk rajta. Ezért sokat kell tesztelni, nehogy az ágensünk furcsa vagy haszontalan dolgot tanuljon meg: Például ne vágja le az utat egy kanyar helyett, vagy ne tolatva jusson el a célba stb. Összesen hat féle jutalmazást implementáltam, melyeknél az ágens az önvezetés különböző aspektusait sajátíthatja el. További jutalmazásokra azonban szükség lehet, például jelenleg a közlekedési táblákat nem veszi figyelembe.

Az alábbi hat jutalomból csak az első kettő (α és β) tekinthető folytonosnak, a többi diszkrét. A diszkrét jutalmak, habár jó eredményre visznek minket, lassabb konvergenciát okoznak, mivel az ágens ritkábban kap visszacsatolást a döntései után. A végső jutalmat a hat részeredmény súlyozott összegéből kapjuk meg:

$$R = w_R\left(w_\alpha\alpha + w_\beta\beta + w_\gamma\gamma + w_\delta\delta + w_\varepsilon\varepsilon + w_\tau\tau\right) \tag{3.1}$$

3.4.1 Alfa

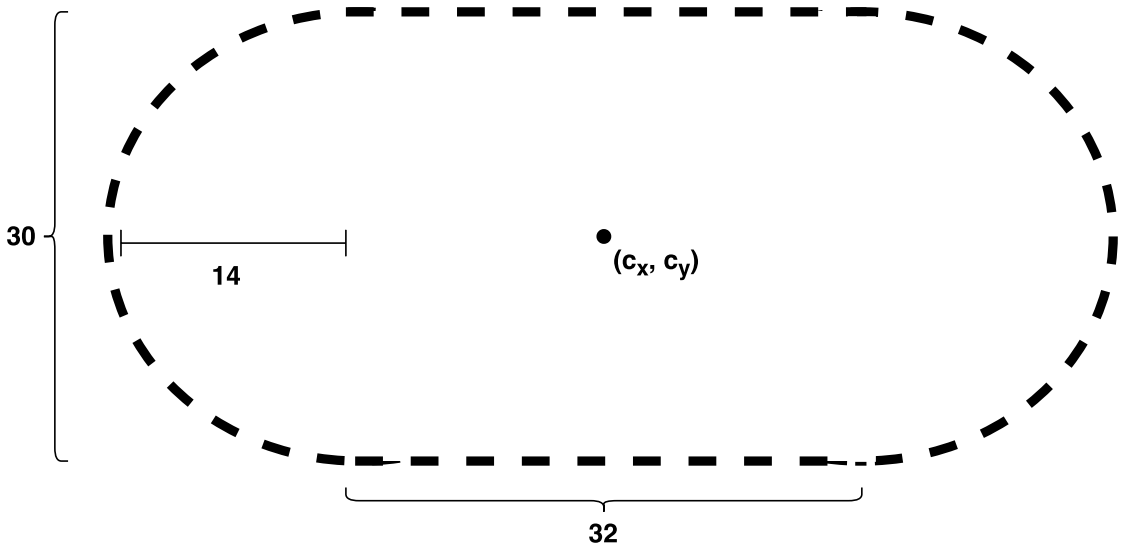
Ez az egyik legegyszerűbb jutalmazó algoritmus mind közül. Célja, hogy eljuttassa az ágenst a kijelölt célba (időtől függetlenül) egyenesvonalban. Folytonos az alfa jutalmazás, tehát minden akció-ismétlés után kiértékeljük a függvényt. A jutalom annak a függvényében pozitív, hogy az előző állapothoz képest közelebb jutott-e a célhoz vagy sem. Ha nőtt a távolság, azaz a céltól elfelé mozdult, akkor negatív a jutalom:

$$\alpha_t = dist_{t-1} - dist_t \tag{3.2}$$

Ezenkívül még egy funkciót ellát az alfa jutalmazás. A tesztelések során kiderült, hogy nagyon gyakran elakad az ágens a pálya szélén, akár már az epizód elején. Mivel egyébként is hosszadalmas a tanítás, ezért bevezettem egy számlálót, mellyel nyomon követjük, hogy hány akción keresztül nem került lényegesen arrébb az ágens az előző pozíciójához képest. Ha ez elérte a küszöbértéket, az azt jelenti, hogy a kocsi elakadt és ekkor újraindul a környezet és kezdődik a következő epizód. Mivel szeretnénk elkerülni, hogy a kocsi fennakadjon (és erre a később részletezett tau jutalom kevés), ezért egy nagy negatív értékre állítjuk az alfát.

3.4.2 Béta

A béta viszont a legkomplexebb algoritmus az implementáltak közül. Ez az algoritmus a sávtartásért felel, azaz a célja, hogy az ágensünk ne menjen át a szembe sávba és ne mehessen le a pályáról sem. Nehézsége abból adódott, hogy a sáv egy fix objektum a környezetben, melynek egyetlen attribútuma van, a középpontjának koordinátái (c_x és c_y). Ennyire kevés információval körülményesebb a távolságokat számolni. Mivel nem találtam hasznos leírást arról, hogy mekkorák a pálya paraméterei, így lemértem és függvényt illesztettem rá. Úgy kezelem a szaggatott vonalból képzett pályát, mintha egy téglalapból állna, melynek két oldalán egy-egy félkör található. A téglalap oldalai egész számokra jöttek ki, így pontosnak gondolom a mérést, de a félkörök nem pontosan félkörök, a tetejük kissé nyomott, így csak becsülöm a sugarat (**3.8. ábra**). Ezeket az adatokat felhasználva a szimuláció koordináta-rendszerében már viszonylag pontosan lehet becsülni az autó helyzetét a sávokon belül.

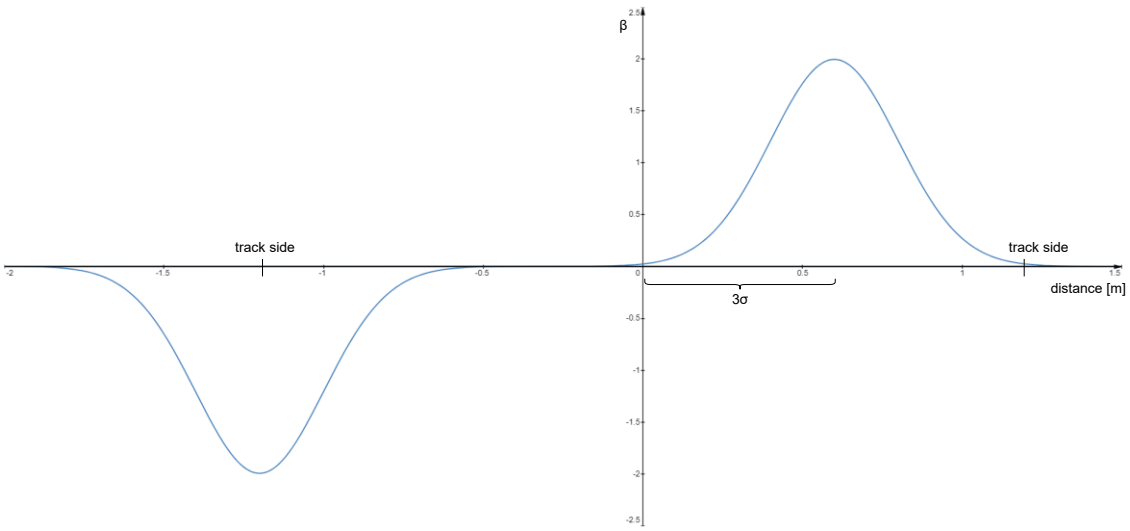


3.8. ábra A szaggatott vonal méretei

Miután megbecsültem a pozíciót, már csak a pontozás mértékét kell beállítani. Ehhez Gauss-görbét használok, melynek a maximuma a jobb oldali sáv közepére van állítva, míg a koordináta-rendszer origója a szaggatott sávon helyezkedik el (**3.9. ábra**). A görbe szélességét úgy állítottam be, hogy a 3σ távolság a sáv széleire essen. Így lényegében ezen az intervallumon kívül balra, vagyis a szembe sávban nincs büntetés az úttest széléig, mivel a Gauss-görbe közel nulla értéket vesz fel. Azonban az úttest széleitől kifelé súlyosan büntetünk, innentől egy nagy abszolút értékű negatív számra (pl.: -100) állítom a bétát. Viszont a szembe sávban haladást is büntetni kellene, tehát arra a sávra

egy az abszcisszára tükrözött, ugyanakkora szórású Gauss-görbét illesztettem. A középpontját, azaz minimumát a pálya szélén éri el. Önkényesen választottam meg az áttérés helyét is az egyik görbéről a másikra, úgy, hogy az átmenetben minimális ugrás legyen:

$$\beta = \begin{cases} -100, & x \geq 6\sigma \\ \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}, & -\sigma < x < 6\sigma \\ -\frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x+2\mu}{\sigma}\right)^2}, & -6\sigma < x \leq -\sigma \\ -100, & x \leq -6\sigma \end{cases} \tag{3.3}$$



3.9. ábra Gauss-görbe elhelyezkedése a pályán

3.4.3 Gamma

A következő jutalmazás célja, hogy a kocsi megálljon a piros lámpánál. Az algoritmus állapotgép-szerűen működik, bizonyos időközönként vált a lámpa piros és zöld között (sárgával egyelőre még nem szükséges foglalkozni). A váltás grafikusan még nem jelenik meg a környezetben, az objektum változtatásokat megvalósítani a PyBullet-ben nem triviális.

Ha zöld a lámpa a gamma értéke nulla. Ha piros, akkor megvizsgáljuk az ágens távolságát a lámpától (a síkban), és ha 2 egységen (méter) belül van, akkor a cél, hogy álljon meg a kocsi, vagyis csökkentse le a sebességét nullára. Ezért a gamma értékét a sebesség függvényében választom meg, jelen eseteben legyen egyszerűen a sebesség

mínusz egyszerese. Ha a kocsi már fél méterre is megközelítette a lámpát, miközben az piros, akkor a gamma szintén egy nagy abszolútértékű negatív konstans értéket vesz fel:

$$\gamma = \begin{cases} 0, & d > 2 \\ -\sqrt{{v_x}^2 + {v_y}^2}, & 2 \geq d \geq 0.5 \\ -100, & d < 0.5 \end{cases} \tag{3.4}$$

3.4.4 Delta

Biztonsági szempontból az egyik legfontosabb képesség, hogy az ágens ki tudja kerülni az útjába kerülő objektumokat. A delta jutalom, annál nagyobb, minél kevésbé közelít meg az ágens egy álló objektumot. Egyelőre a legközelebbi objektum távolságát vetjük össze az ágens távolságával. Ezt a későbbiekben érdemes lehet még tovább fejleszteni, például, hogy csak arra az objektumra figyeljen, amerre halad, vagy akár több objektum távolságát is figyelembe vegye egyszerre.

A függvény alakja ennél is egy x-tengelyre tükrözött Gauss-görbe. A görbe a kocsi és a legközelebbi álló objektum közti egyenesre illeszkedik, mégpedig oly módon, hogy a görbe középpontja egybe esik az objektummal. A cél, hogy az ágens legalább 1 méter sugarú körön kívül kerülje el az objektumot, így 1 méteren belül büntetünk, azon kívül nincs pedig jutalmazás. A kitevőben lévő 7-es szorzót, azaz a görbe szélességét azért választottam ennyinek, mert így a w_δ értékét 0 és 100 között változtatva is kicsi lesz az ugrás $d = 1$ méternél:

$$\delta = \begin{cases} -e^{-7d^2}, & d < 1 \\ 0, & d \geq 1 \end{cases} \tag{3.5}$$

3.4.5 Epszilon

Nemcsak az álló, hanem a mozgó járműveket is célszerű lenne elkerülnie az ágensnek, erre szolgál az epszilon algoritmus. A mozgó objektum a GUI-ban állítható konstans sebességgel mozog két fix pont között oda és vissza. Jelenleg egy ilyen objektum található a pályán, mely az úttest két széle között ingázik a célvonalnál. Ha az ágens túlságosan megközelíti, például 1 méterre, akkor negatív jutalmat adunk, ugyanúgy, mint a delta esetében:

$$\varepsilon = \begin{cases} -e^{-7d^2}, & d < 1 \\ 0, & d \geq 1 \end{cases} \tag{3.6}$$

3.4.6 Tau

Még egy fontos tényezőre nem figyel az ágens: az időre. Szükséges bevezetni egy olyan jutalmazó függvényt is, mely arra sarkallja az ágenst, hogy ne csak egyszerűen eljusson a célba, hanem ezt minél gyorsabban tegye meg. Logikailag a függvényt úgy érdemes felépíteni, hogy legyen pozitív a jutalom, ha az epizód befejezte előtt eljutott a kocsi a célba és legyen nagy negatív a jutalom, ha nem. Minél hamarabb fejezi be az epizódot, annál nagyobb a jutalom. Látható, hogy nehézkes tesztelni a függvény hatását, hiszen epizódonként csak egyszer fut le az algoritmus. A függvény alakját én hiperbolikusnak választottam meg (lásd az alábbi képletet, ahol *T* egy epizód teljes ideje). Fontos megjegyezni, hogy ezek igazából nem idő dimenziójú mértékek, hanem a pontosság kedvéért akciók számában vannak mérve. Ezért nem is fordulhat elő, hogy egy nagyon kicsi számmal való osztás miatt elszállna a jutalom, hiszen a legelső jutalom számolása egy roll-out mennyiségű akció után történik, ezért a numerikus stabilitásra nem kell figyelni (valamint nem is reális, hogy az ágens elér a célba az első akciók alatt).

$$\tau = \begin{cases} \dfrac{T}{\tau}, & \tau \leq T \\ -100, & t > T \end{cases} \tag{3.7}$$

4 Tesztelés

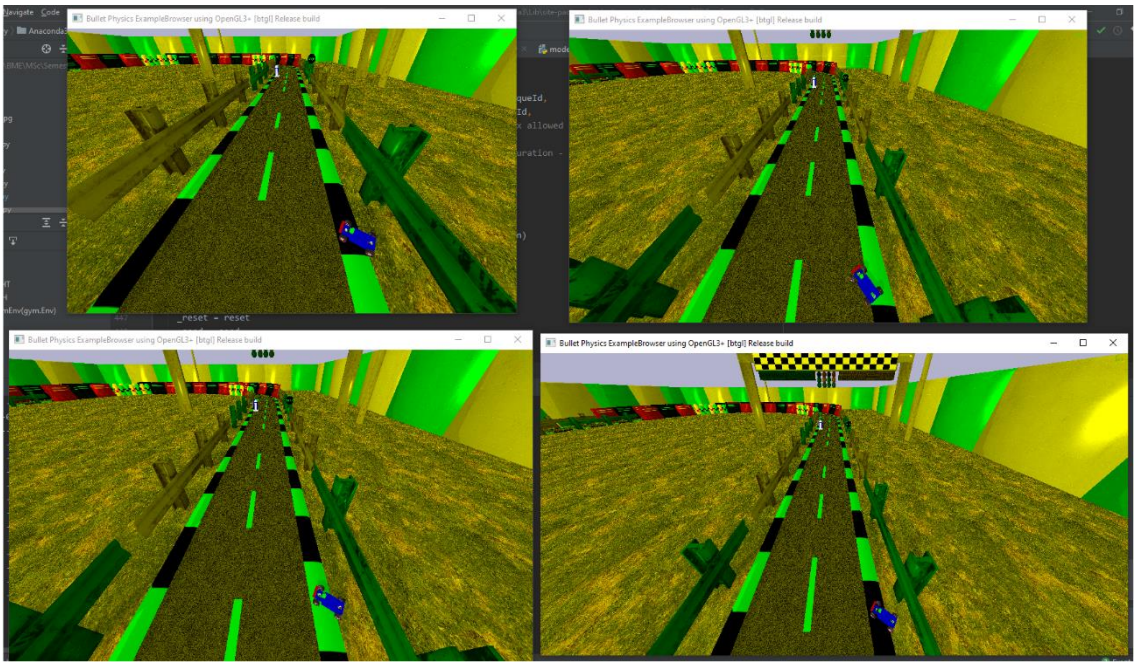
A tesztelés ezesetben az algoritmus tanulásának ellenőrzését és a környezet tesztelését jelenti. A legtöbb itt prezentált eredmény esetében a tanítás közben 4 ágens futtattam a 4 független környezetben. Mint említettem a hardver limitációk miatt igen hosszadalmas egy-egy tanítás. A gyorsítás érdekében le kellett mondani a renderelésről, így a legtöbbször nehéz volt ellenőrizni, hogy valójában mit is csinálnak az ágensek, csak néhány adatból lehet következtetni az ágensek pályáira. A

Az különböző adatokat, mint például a jutalmakat, állapot-értéket, az akciók eloszlását Tensorboard [22] segítségével ábrázoltam. Mindegyik képen a vízszintes tengely a roll-out-ok száma, százzal leskálázva. Egy epizód maximum 80 roll-out-ból állhat, így, ha nincs az epizód közben megszakítás, akkor a diagramokon 0.08-asával követik egymást az epizódok, és 10 epizód kb. egy órát vesznek igénybe (tehát a diagramokon szereplő számot $5/4$-el szorozva kapjuk meg a tanítás idejét).

4.1. ábra Az akció-érték alakulása 1 súlyok esetén.

A legelső méréseknél a jutalmak súlyai mind egységnyire voltak állítva, de az eredményekből látszódott, hogy egy idő után elszaporodtak a nagy negatív értékek (**4.1. ábra**), melyekből arra következtettem, hogy a kocsi folyton elhagyja a versenypályát.

Ellenőrzésképpen bekapcsolva a renderelést meg is bizonyosodtam, hogy az ágens rátanult egy olyan műveletsorozatra, melynél elindul hátrafelé, majd jobbra elhagyja a pályát és fennakad a rázókövön (**4.2. ábra**).

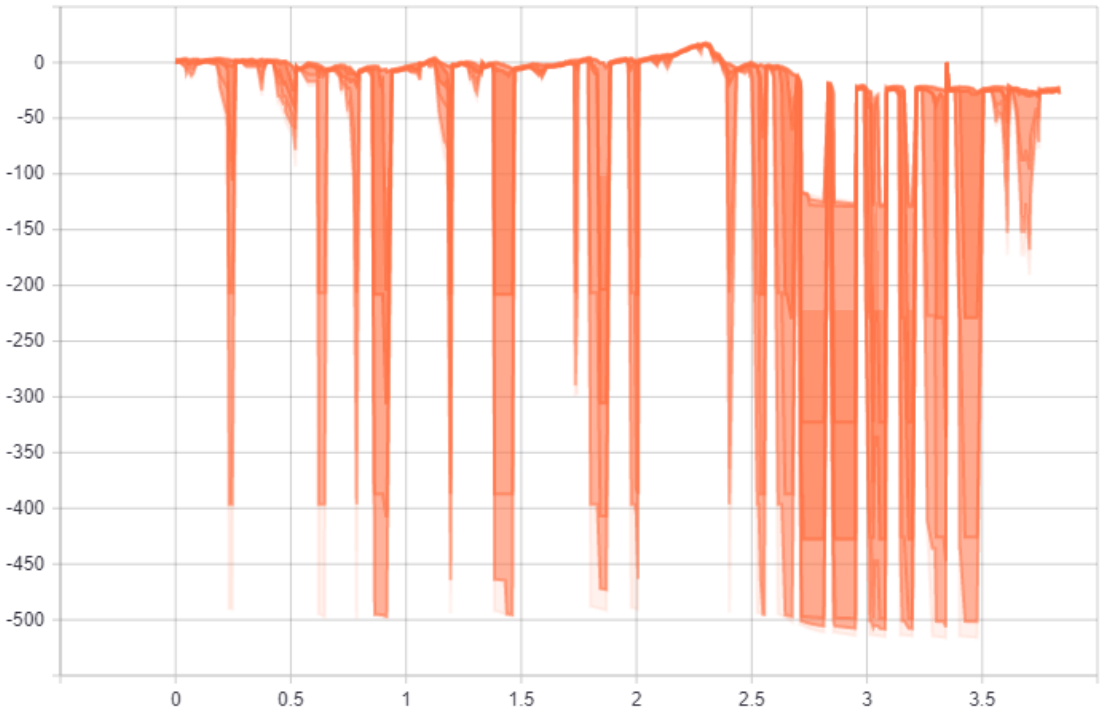


4.2. ábra A hibás betanulás

Először arra gyanakodtam, hogy a hiba abból ered, hogy a kocsi túl közel kerül az egyik álló objektumhoz, és rátanult, arra, hogy minél messzebb kerüljön attól, akár tolatással is. Viszont az objektumot eltávolítva is jelentkezett a hibás működés. Végül a jutalom algoritmusokat elemezve rájöttem, hogy közel sem esik egy nagyságrendbe az alfa a többivel, így esélye sincs megtanulni az ágensnek, hogy a cél felé kéne haladnia. Ezt javítva a w_α-át 30-ra állítva már elértem azt, hogyha a sávközepén tolat a kocsi, akkor is negatív a jutalom, míg eddig ezért pozitív jutalmat kapott. Az **4.3. ábra** látható a tanítás eredményét ezen paraméterek esetén. Látható, hogy kevesebbszer hagyta el a kocsi a pályát, de így is rátanult egy hasonló műveletre.

A javítás érdekében először csak alfát kapcsoltam be a jutalmazáshoz, minden más jutalom nem játszott szerepet a tanításban. Ekkor egy idő után az akció-érték már konvergált egy pozitív értékhez (15-25 körül), ellenőrzésképpen rendereltem a környezetet és csak egy ágenst futtattam. Megfigyeltem, ahogy az ágens egészen rátanult arra, hogy előrefelé haladjon, a maximális értéke a jutalomnak 2.5 körül volt, ebből kiszámolható, hogy a maximális sebessége nagyjából 13 cm/s. Viszont itt is rátanult arra, hogy egy idő után elhagyja a pályát és a pálya szélén próbáljon a cél felé haladni,

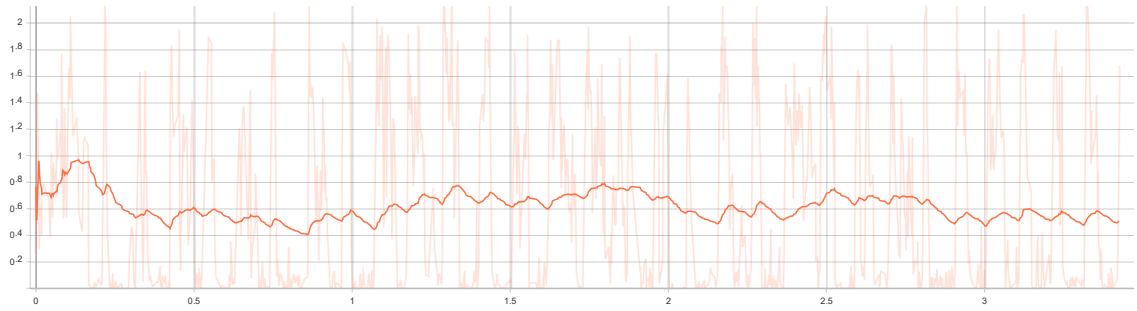
amelyet néhány óra után el is ért. Pont a pályaelhagyás büntetésére lett kitalálva a béta jutalom, így azt is visszavezetve a jutalmazásba, folytattam a tanításokat. Ekkor ismét előjött az az eset, hogy egyből lekanyarodik a pályáról és fennakad a pálya szélén az ágens. ennek a hiba hatására többször is javítottam a béta függvényén, mire az előző fejezetben lévő állapotát elérte. Valamint ezen a ponton vezettem be azt a funkciót, hogy ha elakad az ágens, akkor induljon újra a környezet és kezdődjen a következő epizód. Ez sokat javított a tanítások eredményein, de az ominózus jelenség továbbra is előjött.



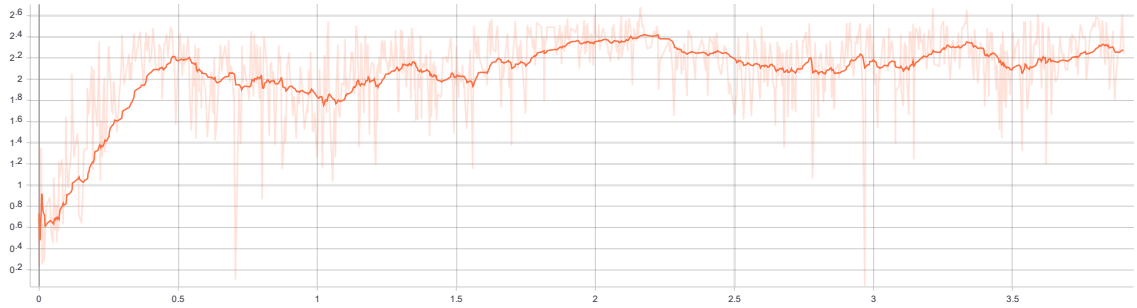
4.3. ábra Az akció-érték még a súlyok javítása után is negatív értékre konvergál.

Ezekután a bétát újra kikapcsolva tanítottam tovább a modellt. Az alfa mellett a tau jutalmat is gyakran használtam, mely sajnos jellegéből adódóan nagyon lassan konvergálhat. Legtöbbször csak 30-40 epizódig tanítottam, de néhány száz epizódot kéne megvárni inkább. A sok tesztelés közben még egy hiba kibukott, ezek a görbék az eredeti implementációban jutalom néven lettek ábrázolva, de mint jelzem, ezek valójában az akció-érték görbék, nem a roll-out-onkénti „nyers” jutalmak, amelyeket közvetlenül a környezettől kapnak az ágensek. Ettől még használhatóak, de a valódi jutalmakat az első mérésekről éppen ezért nem tudom itt bemutatni. Az alábbi két képen az igazi alfa jutalmak láthatóak, mindkettő tanítás nagyjából ugyanannyi ideig futott (epizódban már nincs értelme mérni a rendszeres elakadások miatti újraindítások miatt). Az előbbi képen nem volt bekapcsolva az elakadás figyelése, így jól láthatóan sokkal többször mozgott a

jutalom a nulla érték körül, és így a jutalom átlaga is sokkal kisebb lett, míg bekapcsolva a funkció szépen látszódik a konvergencia, és hogy nagyjából 2.5 körül lehet az alfa maximuma.



4.4. ábra Az alfa jutalom, nincs újraindítás elakadás esetén

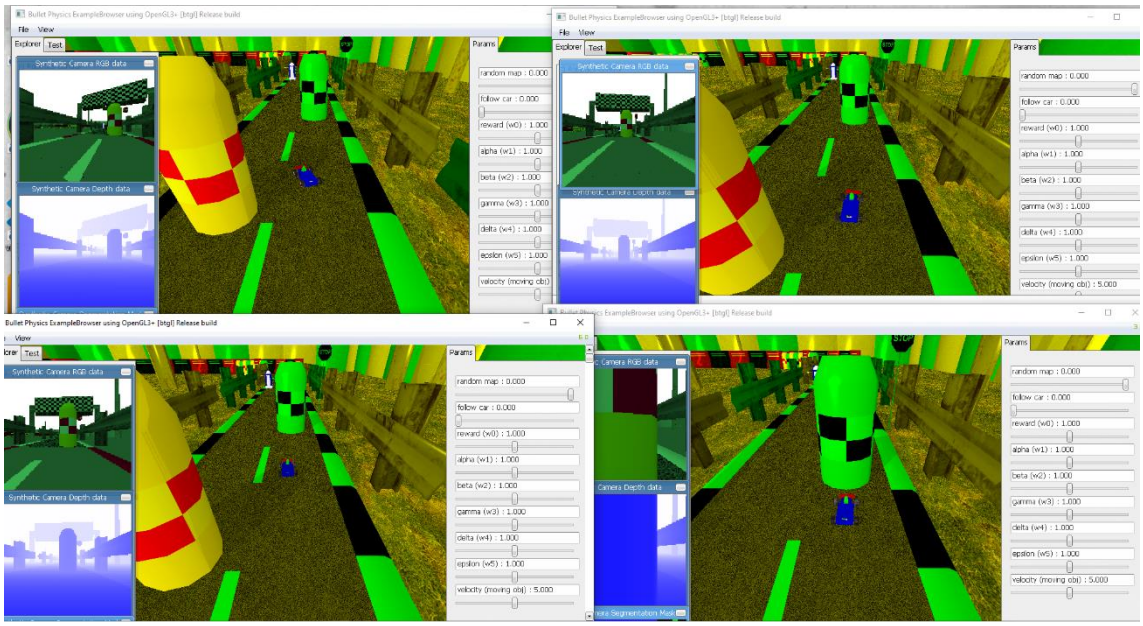


4.5. ábra Az alfa jutalom, az elakadások miatti újraindításokkal együtt

Ezeken kívül még hasonló jutalmazások mellett teszteltem a különböző modell konfigurációkat. Az alap konfiguráció esetén, tehát amikor csak a konvolúciós blokkot használom a konvergencia simább, mint abban az esetben, mikor minden be van kapcsolva. Ugyanakkor kisebb értékre konvergál, bár ezek a tanítások csak rövidebb ideig futottak. A legjobb eredményt az Attention blokk kikapcsolásával értem el, mely a legszebb konvergenciához vezetett és a nagyobb értékhez konvergált a többihez képest.

5 Összefoglaló

Az önvezető funkciót ellátó ágenst struktúráját sikerült véglegesíteni. A hiperparaméterek hangolásához viszont szükséges még további tanításokat végezni. Ezenkívül elkészítettem azokat a jutalom függvényeket, melyek már elegendőnek kellene lenniük ahhoz, hogy egy komplexebb pályán megtanuljon az ágens végig menni. A jutalmazások súlyait, azaz, hogy egymáshoz képest mennyire vegyük figyelembe a tanításkor, még hangolni kell. Ehhez a jövőben használhatunk például Bayes optimalizációt, melyben a jutalom függvényt szeretnénk maximalizálni, a súlyok függvényében.



5.1. ábra Négy ágens tanul egyszerre.

A pályával kapcsolatban még sok mindent kellene javítani. Az egyik legfontosabb az önvezetés robusztussága szempontjából, hogy nem teljesen véletlenszerűen generált a pálya. Ez a randomizáció azt a célt szolgálja, hogy a kiskocsi vezetni tanuljon meg, és ne egy pályát magoljon be. Ennek megoldásában a legnagyobb akadályt az úttest fogja jelenteni, mely egyetlen objektumból áll, így valószínűleg új objektumokat kell majd szerkeszteni. Későbbiekben a táblafelismerés képességével is fontos lenne növelni az ágens funkcióit, ezért táblákat is el kell majd helyezni a környezetben. Ezenkívül a magasságbeli változtatásokat sem tudjuk még megtanítani a kocsinak, érdemes lehet lejtőket és emelkedőket is betervezni a versenypálya bizonyos szakaszaiba. Útjelzések és egyéb közlekedést irányító jelzéseket is érdemes lenne megtanítani az ágensnek.

6 Irodalomjegyzék

[1] M. A. Nielsen, Neural Networks and Deep Learning, szerk., %1. kötet, , : Determination Press, 2015, p. .

[2] . . Siuly, Y. . Li és P. . Wen, „Clustering technique-based least square support vector machine for EEG signal classification,” *Computer Methods and Programs in Biomedicine,* %1. kötet104, %1. szám3, pp. 358-372, 2011.

[3] Y. . Lin, Y. . Lee és G. . Wahba, „Support Vector Machines for Classification in Nonstandard Situations,” *Machine Learning,* %1. kötet46, %1. szám1, pp. 191-202, 2002.

[4] V. C. Raykar és P. . Agrawal, „Sequential crowdsourced labeling as an epsilon-greedy exploration in a Markov Decision Process,” , 2014. [Online]. Available: http://proceedings.mlr.press/v33/raykar14.pdf. [Hozzáférés dátuma: 31 5 2020].

[5] L. . Baird, „Residual algorithms: Reinforcement learning with function approximation,” *ICML,* %1. kötet, %1. szám, p. 30–37, 1995.

[6] „Reinforcement Learning / Successes of Reinforcement Learning,” , . [Online]. Available: http://umichrl.pbworks.com/Successes-of-Reinforcement-Learning/. [Hozzáférés dátuma: 31 5 2020].

[7] J. . Peters, S. . Vijayakumar és S. . Schaal, „Natural actor-critic,” *Lecture Notes in Computer Science,* %1. kötet, %1. szám, pp. 280-291, 2005.

[8] S. . Li, S. . Bing és S. . Yang, „Distributional Advantage Actor-Critic.,” *arXiv: Learning,* %1. kötet, %1. szám, p. , 2018.

[9] A. . Juliani, „Simple Reinforcement Learning with Tensorflow Part 8: Asynchronous Actor-Critic Agents (A3C),” , . [Online]. Available: https://medium.com/emergent-future/simple-reinforcement-learning-with-

tensorflow-part-8-asynchronous-actor-critic-agents-a3c-c88f72a5e9f2. [Hozzáférés dátuma: 31 5 2020].

[10] J. . Vig és Y. . Belinkov, „Analyzing the Structure of Attention in a Transformer Language Model," *arXiv: Computation and Language,* %1. kötet, %1. szám, p. , 2019.

[11] S. . Bock, J. . Goppold és M. . Weiß, „An improvement of the convergence proof of the ADAM-Optimizer.," *arXiv: Learning,* %1. kötet, %1. szám, p. , 2018.

[12] „Welcome to Colaboratory," , . [Online]. Available: https://colab.research.google.com. [Hozzáférés dátuma: 22 5 2019].

[13] „Project Jupyter," , . [Online]. Available: https://jupyter.org/. [Hozzáférés dátuma: 22 5 2019].

[14] „Parallel Programming and Computing Platform," , . [Online]. Available: http://www.nvidia.com/object/cuda_home_new.html. [Hozzáférés dátuma: 22 5 2019].

[15] „Download PyCharm," , . [Online]. Available: https://www.jetbrains.com/pycharm/download/. [Hozzáférés dátuma: 22 5 2019].

[16] „About Python," , . [Online]. Available: https://www.python.org/about. [Hozzáférés dátuma: 22 5 2019].

[17] N. . Ketkar, „Introduction to PyTorch," , 2017. [Online]. Available: https://link.springer.com/chapter/10.1007/978-1-4842-2766-4_12. [Hozzáférés dátuma: 22 5 2019].

[18] „GitHub: bulletphysics/bullet3 releases," , . [Online]. Available: https://github.com/bulletphysics/bullet3/releases. [Hozzáférés dátuma: 31 5 2020].

[19] M. D. Zeiler és R. Fergus, „Visualizing and Understanding Convolutional Networks," 2014.. [Online]. Available: https://cs.nyu.edu/~fergus/papers/zeilerECCV2014.pdf.

[20] J. . Wang, X. . Peng és Y. . Qiao, „Cascade multi-head attention networks for action recognition,” *Computer Vision and Image Understanding,* %1. kötet, %1. szám, p. 102898, 2020.

[21] S. Hochreiter és J. Schmidhuber, „Long Short-Term Memory,” *Neural Computation,* %1. kötet9, %1. szám8, p. 1735–1780, 1997.

[22] TensorFlow, „TensorFlow,” 2018.. [Online]. Available: https://www.tensorflow.org/tensorboard. [Hozzáférés dátuma: 2020.].

[23] H. . Robbins és D. O. Siegmund, Optimizing Methods in Statistics, szerk., %1. kötet, J. S. Rustagi, Szerk., , : Academic Press, 1971, p. .