

Budapesti Műszaki és Gazdaságtudományi Egyetem

Villamosmérnöki és Informatikai Kar

Szilágyi Krisztián Gergely

Autonóm jármű tanítása szimulációs környezetben

Önvezető szimulátor fejlesztése megerősítéses tanulással

Konzulens

Dr. Szemenyei Márton

BUDAPEST, 2021

Tartalomjegyzék

[Összefoglaló 3](#_Toc89299298)

[Abstract 4](#_Toc89299299)

[1 Bevezetés 5](#_Toc89299300)

[2 Irodalmi áttekintés 6](#_Toc89299301)

[2.1 Mély tanulás 6](#_Toc89299302)

[2.1.1 Csoportosítás 6](#_Toc89299303)

[2.1.2 Neurális hálózatok 8](#_Toc89299304)

[2.1.3 Konvolúciós neurális hálózatok 13](#_Toc89299305)

[2.1.4 Visszacsatolt neurális hálózatok 15](#_Toc89299306)

[2.2 Megerősítéses tanulás 18](#_Toc89299307)

[2.2.1 Q-tanulás és policy gradiens módszerek 20](#_Toc89299308)

[2.2.2 Actor-Critic 23](#_Toc89299309)

[2.2.3 Proximal Policy Optimization 24](#_Toc89299310)

[2.3 Imitációs tanulás 25](#_Toc89299311)

[2.3.1 Behavioural Cloning 26](#_Toc89299312)

[2.3.2 Direct Policy Learning 26](#_Toc89299313)

[2.3.3 Inverse Reinforcement Learning 27](#_Toc89299314)

[2.4 Attention 28](#_Toc89299315)

[2.4.1 Self-Attention 28](#_Toc89299316)

[2.4.2 Pozíció kódolás 32](#_Toc89299317)

[2.5 RAdam 34](#_Toc89299318)

[3 Specifikáció, tervezés 37](#_Toc89299319)

[3.1 Specifikáció 37](#_Toc89299320)

[3.2 Fejlesztői eszközök 38](#_Toc89299321)

[3.2.1 Colaboratory 38](#_Toc89299322)

[3.2.2 PyTorch 38](#_Toc89299323)

[3.2.3 PyBullet 39](#_Toc89299324)

[3.3 Tervezés 39](#_Toc89299325)

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.

# Bevezetés

# Irodalmi áttekintés

Ebben a fejezetben 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 csak szűkebb körben ismertek.

## Mély tanulás

A gépi tanulás (*Machine Learning*) az egyik út a sok lehetséges közül a mesterséges intelligencia felé. Ebben az eljárásban felvonultatott 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.

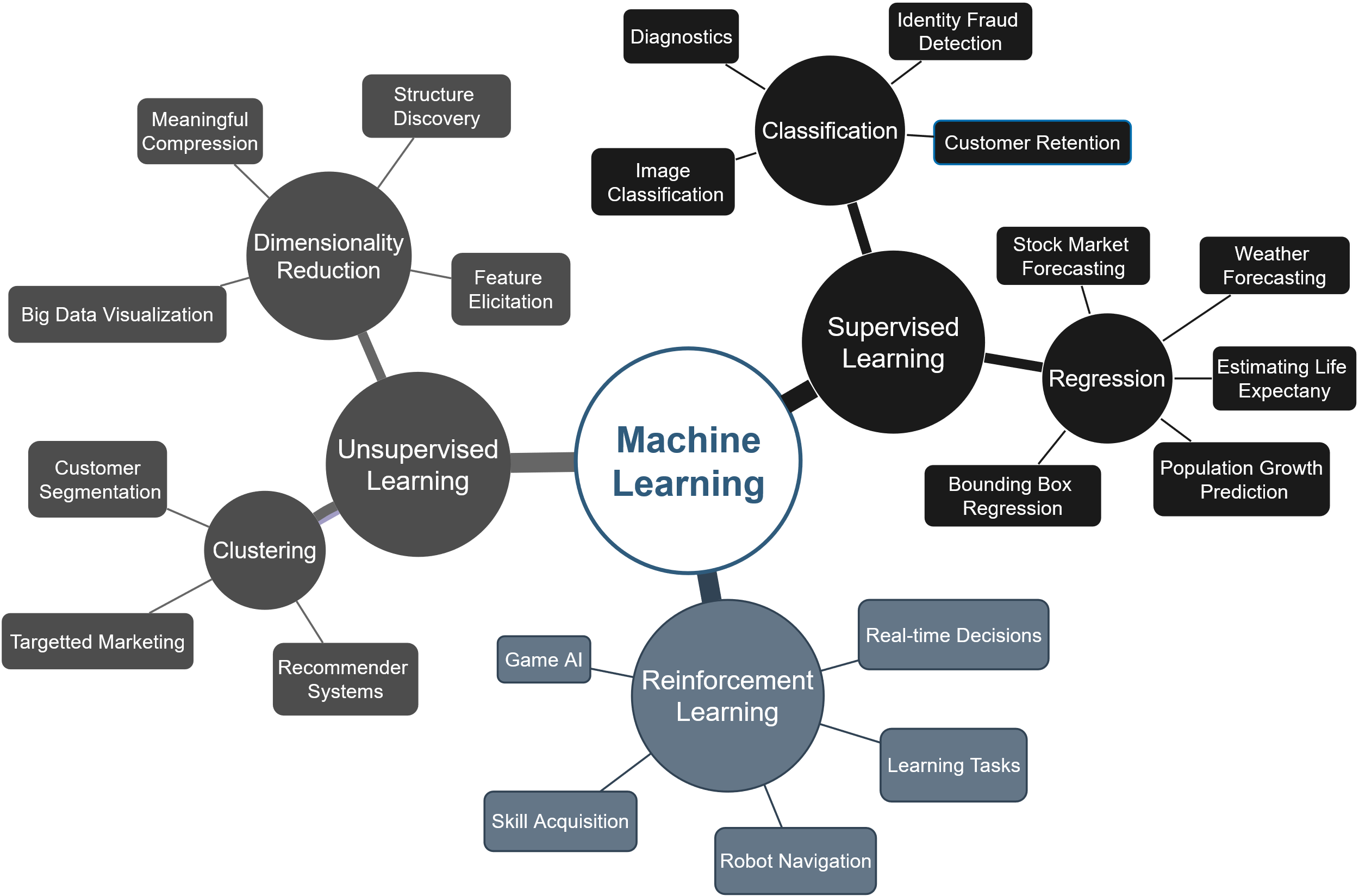
A mély tanulás (*Deep Learning*) a reprezentáció tanulásnak egyik alkalmazása, míg a reprezentáció tanulás a gépi tanulásos eljárások egyik részhalmaza. A reprezentáció tanulásnál a belső reprezentációkat, jellemzőket nem kézzel kell beállítani, hanem képes ezeket megtanulni az algoritmus. Mély tanulásnál ez a folyamat több rétegű, egyre összetettebb 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, melyek kifejtése előtt érdemes még részletezni a gépi tanulás lehetséges csoportosításait és tipikus alkalmazási területeit.

### 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.

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.



2.1. ábra A gépi tanulás egy csoportosítása

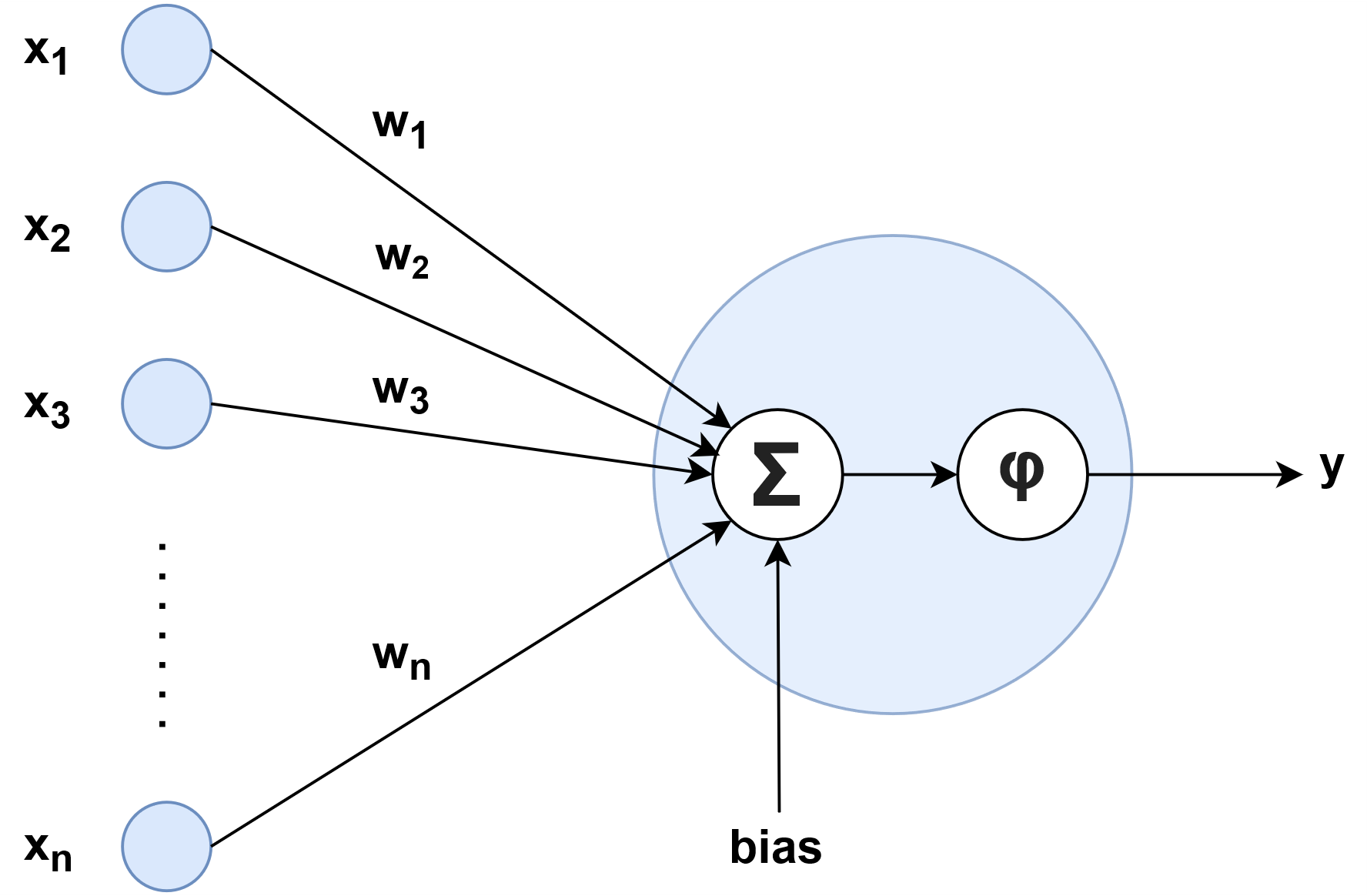
### Neurális hálózatok

A neurális hálózat egy mély tanulást 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 a 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.

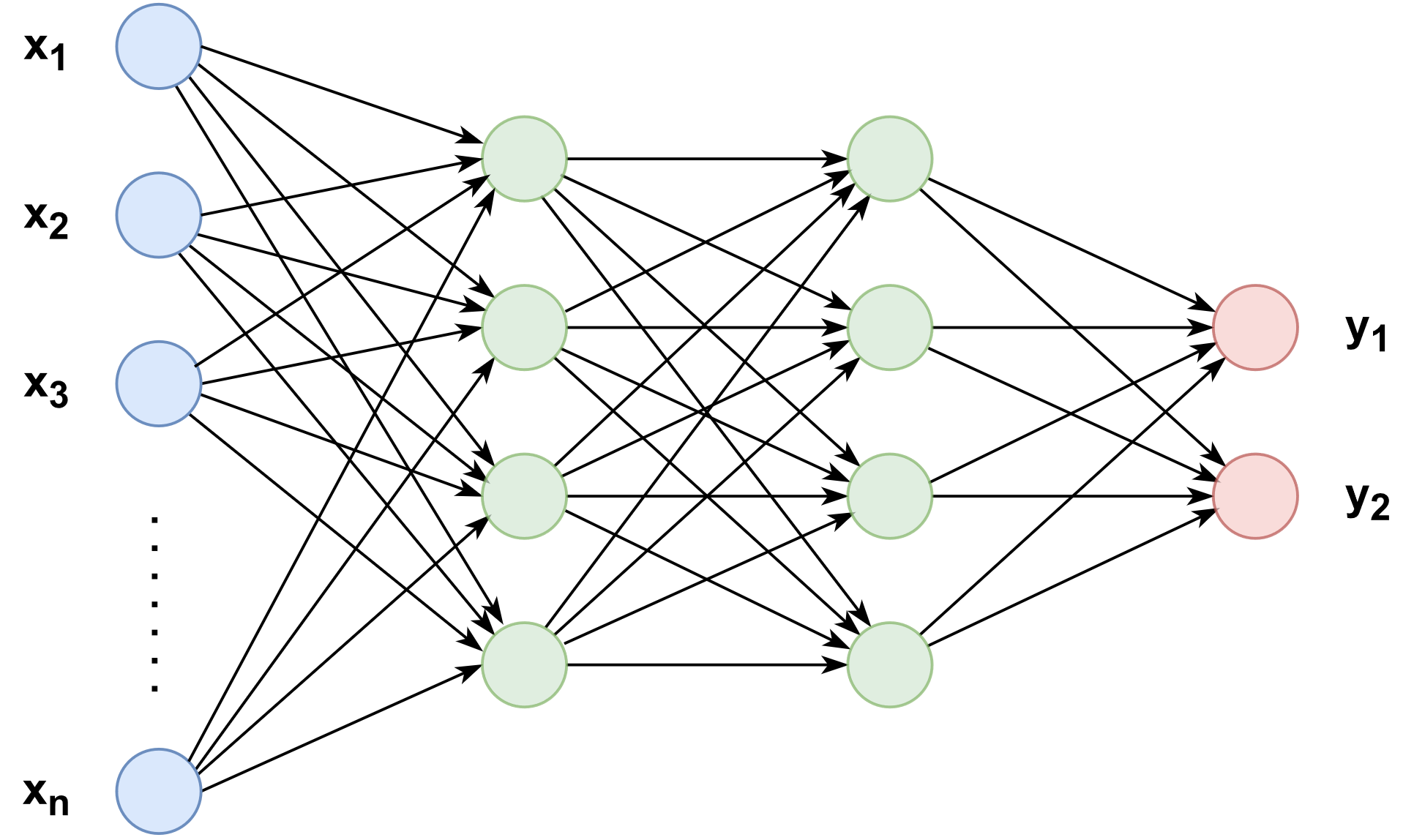
A perceptron az a struktúra, mely egy neuronból és az előtte lévő rétegbeli neuronokkal való kapcsolatából áll (lásd **2.2. ábra**). Ez a modul a legelemibb algoritmus a neurális hálózatban. Igazából ez volt a legelső lineáris osztályozó (az osztályozás feltétele a kimenet előjele). Vegyük az előző réteg 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 2.1 egyenlet), amely azt reprezentálja, hogy mennyire aktiválódik (tüzel) a neuron. Fontos, hogy 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 lenne képes tetszőleges bemenetre tetszőleges függvényt illeszteni.

|  |  |  |
| --- | --- | --- |
|  |  | (2.1) |



2.2. ábra Perceptron

Az MLP, azaz a *multilayer* *perceptron* (a legismertebb neurális hálózat modell) perceptronok összeségéből épül fel, az említett rétegekbe rendezve. A **2.3. ábra** a kék neuronok alkotják a bemeneti réteget, a zölddel színezettek a rejtett rétegeket és végül a piros neuronok a kimeneti réteget. Rendkívül egyszerű a felépítése, csak előre-csatolt rétegekből épül fel. A bemeneti neuronok nem perceptronok, nincs nem-lineáris aktivációjuk, csak a bemeneti vektort reprezentálják.



2.3. ábra Egy egyszerű multilayer perceptron felépítése

A neurális hálók leglényegesebb metódusa, a hiba-visszaterjesztési algoritmus (*backpropagation*). Ennek segítségével számolható a veszteség gradiens, mely felhasználásával valósítható meg a hálózat tanítása. Tanuláskor 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 nézve.

Azonban 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 (véges időn belül) megtalálják a globális optimumot. Hátrányuk, hogy az egyáltalán nem garantált, hogy ez emberi mércével 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 a 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.

Ennél 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) paraméter terének dimenziója. Például amíg egy egyszerűbb hálónak 10-20 hiperparamétere van, addig tipikusan több millió paramétere. Egy ilyen neuroevolúciós 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, KPI 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 hibafelületen a kiszámolt gradiens irányába, a szerinte megfelelő irányban változtatva a háló paramétereit.

Az optimalizáló a veszteség gradiens kiszámításához használja fel a hiba-visszaterjesztési algoritmust. Az elnevezés onnan ered, hogy a módszer a kimenettől visszafelé rétegenként számolja ki a veszteség deriváltjait (az adott paraméterek szerint) egészen a háló a bemenetéig. Így végül megkapja, hogy milyen mértékben függ a kimenet (a veszteség) a bemenettől vagy a paraméterektől. 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 a tanítás pontatlan lenne. Ha viszont csak a teljes adatsor után frissítenénk a paramétereket, akkor a hálózat nagyon lassan konvergálna egy optimális állapotba. Éppen ezért az adatokat csoportosítják, úgynevezett batch-ekbe (gyakran *mini-batch*-nek is nevezik) rendezik, és ezt az adatcsomagot adják be a hálónak, így kisebb a veszteség varianciája, de reális időn belül konvergálna a tanítás. Először az egész adathalmazt felbontjuk azonos méretű (tipikusan 4 hatványai, ami 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 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 reziduális hálókkal, a regularizációs eljárásokkal, az *over-fitting* és további hasonló jelenségekkel a megerősítéses tanulás esetében.

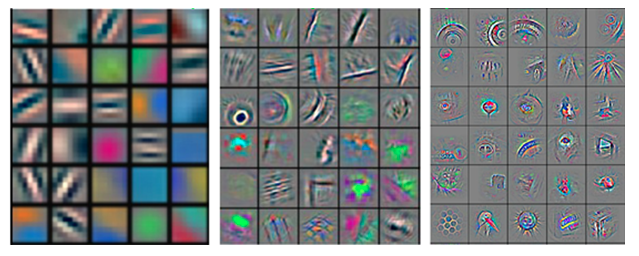
### Konvolúciós neurális hálózatok

Egy háló bemenete többnyire nem csak egy skalár vagy egy független elemekből álló vektor, hanem egy tenzor, melynek elemei nem függetlenek egymástól (például egy kép). Ekkor, ha egy egyszerű lineáris réteg lenne a bemeneti réteg, lemondanánk a térbeli információkról, mivel az egymás melletti pixeleket, amelyek egyébként összefüggenek, egymástól függetlenül dolgoznánk fel őket, az összes pixelt ugyanúgy kezelné a háló. Erre megoldást nyújtanak a konvolúciós neurális hálózatok (CNN), ahol nem lesz minden neuron hatással mindegyikre, hanem csak bizonyos (lokális) neuronokra. A ritkább kapcsolat következménye a konvolúciós rétegek másik nagy előnye, hogy nagyságrendekkel kevesebb paraméterre van szükség. Például egy színes full HD kép feldolgozása esetén az első réteg , azaz kb. 6 millió neuronból kell, hogy álljon. Ha a következő réteg csak 1000 neuronból is áll, már akkor 6 milliárd súlyról beszélünk. A sok paraméter hátránya a lassabb tanításon kívül a nehezebb tárolás és az over-fitting valószínűségének a növekedése.

A réteg úgy működik, mint egy konvolúciós szűrő (igazából, mint egy keresztkorrelációs szűrő), vagyis egy (többnyire) négyzetes ablakkal, avagy kernellel végrehajtunk egy konvolúciót a kép egy azonos méretű tartományán (2.2 egyenlet). A számítás eredménye egy skalár, melyet az új képen azon a koordinátán helyezünk el, mely a szűrőablak közepe (általában páratlan elemű a kernel az egyértelmű számolás miatt). A kernelt végig léptetve a képen megkapjuk az szűrt képet.

|  |  |  |
| --- | --- | --- |
|  |  | (2.2) |

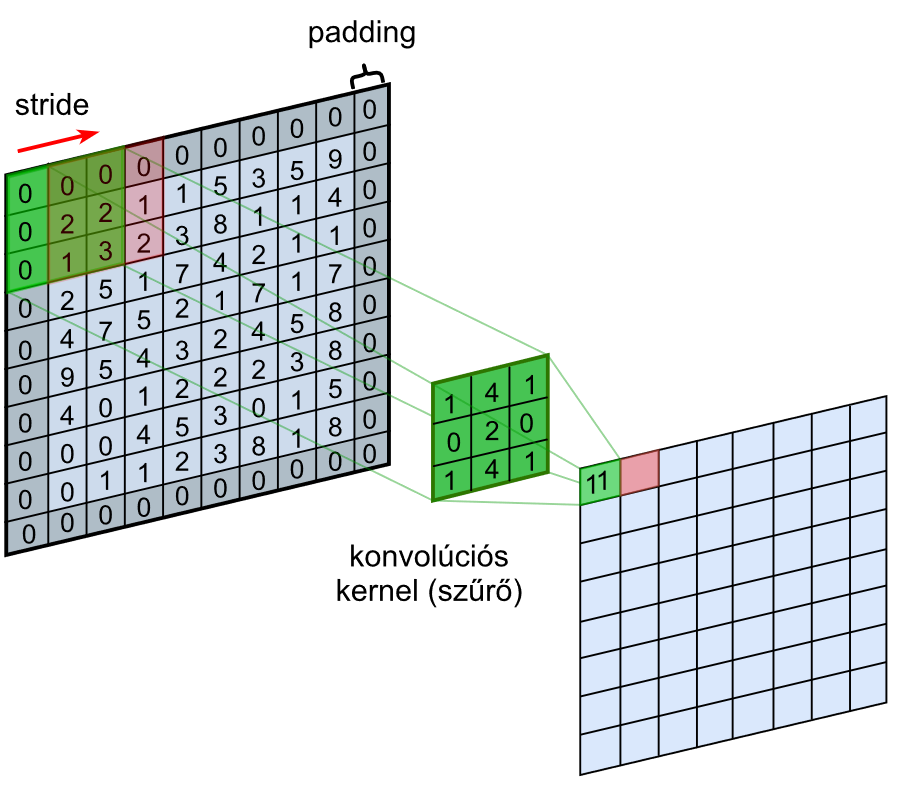
A réteg egy jellemző detektort valósít, minden egyes jellemzőt egy adott szűrővel képes detektálni. Ezeket a szűrőket tanulja meg a háló, melyek az egyes jellemzőknél a hozzá tartozó szűrés után maximális lesz az aktiváció. Például egy arcfelismerő háló tanítása során kialakul egy olyan szűrő, ami ott ad maximum aktivációt, ahol talál egy szemet. Korábban már említtetettem, hogy az első réteg a kép pixelei közti alacsony szintű összefüggéseit tanulja meg, például élek, sarkok, görbeszakaszok. A második réteg ennek az aktivációs térképét kapja bemenetként, itt tehát ezek a képjellemzők közti összefüggéseit tanulja meg. Így tovább haladva egyre komplexebb jellemzőket lesz képes figyelembe venni (lásd **2.4. ábra**). Az előbbi példát nézve egy későbbi rétegben a szűrők már a szempárakat, szájakat, füleket képesek detektálni. Az azt következő réteg meg ezek alapján megtalálja az arcot, mert megtanulta, hogy általában ezek milyen elrendezésben szoktak lenni egymáshoz képest. A fenti számolós példára visszatérve így, ha kell 1000 szűrő, amelyek például -esek, akkor csak , tehát csak 140 ezer súly van a több, mint 6 milliárd helyett.



2.4. ábra Jellemzőkre rátanult szűrők

Az egyik legfontosabb hiperparamétere a konvolúciós rétegnek a mélysége, mely a szűrők számát jelenti (). Így a bemeneti csatornás tenzor csatornás lesz a szűrés után. Fontos hiperparaméterek még a szűrő kernelek méretei, a keretezés vastagsága (*padding*), a kernel léptetésének mértéke (*stride*) és a dilatáció. Ezekkel beállíthatjuk, hogy hogyan változzanak a kép (tenzor) térbeli dimenziói, vagyis a szélessége és magassága. Tipikusan úgy állítják be ezeket, hogy ne változzanak a térbeli dimenziók, de elterjedt módszer még a leskálázó (*strided*) konvolúció is. Ezenkívül a csatorna szám is változtatható egy -es konvolúció esetén. Ilyet valósítanak meg a tömörítő (*bottleneck*) rétegek, melyek egyszerűen csak lecsökkentik a csatornaszámot, majd a számításigényesebb műveletek elvégzése után visszatérnek az eredeti csatornaszámhoz, a térbeli méreteket meghagyva.

Fontos megemlíteni még, hogy a konvolúció lineáris művelet, így továbbra is szükség van a szűrés után nem-lineáris aktivációra. Tipikusan ReLU-t (*Rectified Linear Unit*), vagy annak különböző variánsait szokták alkalmazni. Ez a függvény egészen egyszerűen a kimenet minden negatív elemét nullába állítja, míg a pozitív értékeket változatlanul hagyja: .

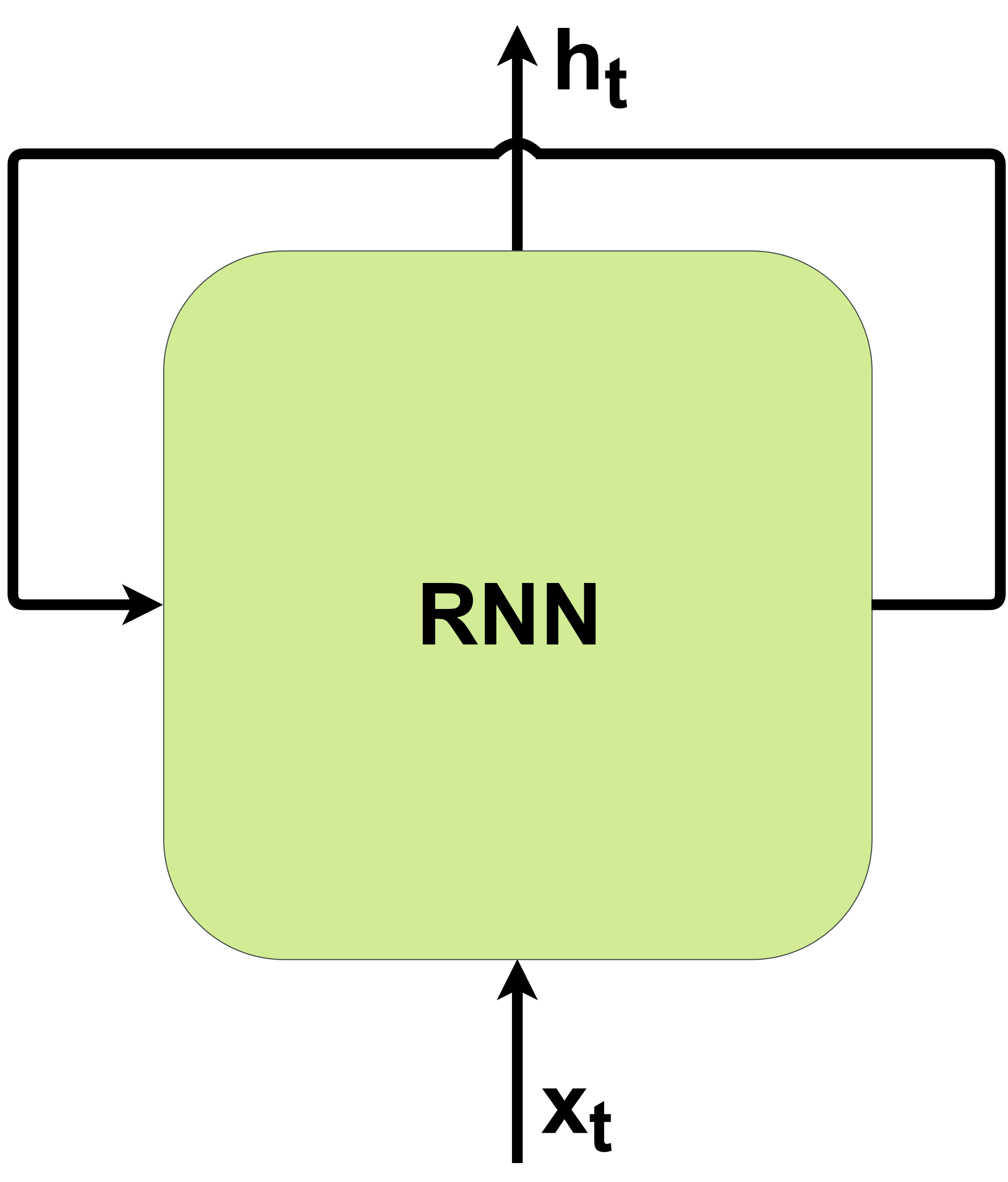


2.5. ábra Konvolúció, 1-es léptetéssel és nullákkal való keretezéssel

A leskálázásra alkalmazott népszerű eljárás még a *pooling*, de ez nem tanulható skálázás. Rendkívül egyszerű a működése: ez is egy ablakot léptet végig a tenzoron és az ablakban található elemeket helyettesíti egy darab számmal egy adott függvény szerint, így csökkentve a térbeli méreteket. Ez a függvény tipikusan az elemek maximuma vagy az átlaga. Pooling esetén a paraméter szám csökkentése a legfontosabb cél, a lényeg, hogy elhagyunk felesleges információkat. Például, osztályozás esetén, elveszíthetjük az információt arról, hogy pontosan hol detektáltunk egy adott jellemzőt a képen. Nem kell tudnunk, hogy pontosan hol található egy kocsi kereke a képen, elég, ha tudjuk, hogy egy autó van a képen. Ezáltal helyzet-invariáns, azaz robusztusabb lett a háló. Ezenkívül így a háló toleránsabb lesz a zajokra és a nagy varianciákra.

### Visszacsatolt neurális hálózatok

A képek hatékony feldolgozása után a következő kérdés lehet az, hogy miként tudunk feldolgozni képsorozatokat. Az előre csatolt hálóknak nincsen memóriaeleme, ezért elvesznek a képek közti változások, vagyis az időbeli információk. Szükség lenne tehát egy olyan háló struktúrára, amely képes modellezni az időt, tehát legyen belső állapota (memóriája). Az ilyen típusú feladatokra találták ki a visszacsatolt neurális hálózatot (RNN), mely képes hatékonyan feldolgozni a bemenetére érkező szekvenciát, melyből általában szintén valamiféle szekvenciát állít elő (*sequence-to-sequence* modellek).



2.6. ábra Standard visszacsatolt NN modell

Tanításkor az ún. BPTT (Backpropagation Through Time) algoritmust kell alkalmazni, amely a hiba-visszaterjesztés visszacsatolt hálókra vett adaptációja. A BPTT során „kiterítjük” időben a hálót, így keletkezik darab példány. Ezek egymás másolatai, minden egyes példány ugyanazokkal a tanulható parméterekkel rendelkeznek. Minden példány esetében számított hibákat ezen a kiterített hálózaton keresztül terjesztjük vissza egészen a legelső bemenetig, a réteg vesztesége ezeknek az összege. A nem fix hosszúságú bemeneteknél előjön az a probléma, hogy ahogy nő a bementi szekvencia hossza, úgy egyre nagyobb lesz a réteg, amelyen számolni kell a BPTT-t. 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. A visszacsatolt neurális hálózatok gyakori felhasználási területe a videóanalízis és a nyelvi fordítók.

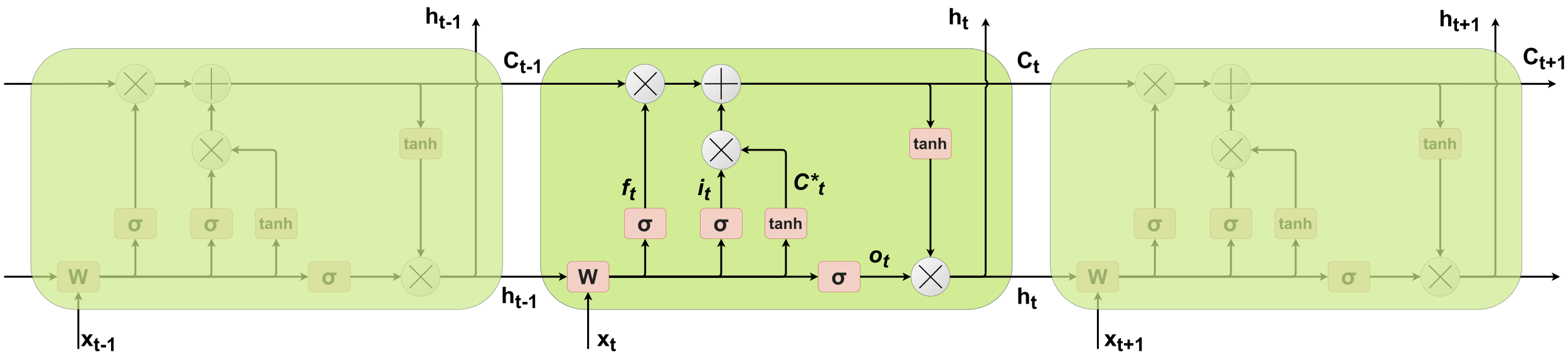
Többféle visszacsatolt neurális hálózat struktúra is napvilágot látott, két elterjedt típusa az LSTM (*Long Short-Term Memory*) és a GRU (*Gated Recurrent Unit*). Mindkettő komplexebb architektúrájú, mint a standard RNN réteg, melynél csak a rejtett állapotot csatoljuk vissza. Így képesek arra, hogy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

Továbbá az egyik legfontosabb tulajdonságuk, hogy kiküszöbölik az eltűnő gradiens és a felrobbanó gradiens (*vanishing* és *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 bemenethez közeli rétegek paraméterei 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, nem lesz képes konvergálni egy optimális állapot felé. 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 (), egy bemeneti (), egy kimeneti () kapu és egy update () rétegből. Adott idő-lépésben a bemenet, a kimenet és a cella állapota. Az alábbi első négy egyenletben látható a kapuk leírása (2.3-6 egyenlet), mind a négy a cella előző időpontbeli kimenetétől és az aktuális bemenettől függ. A **2.7. ábra** -vel jelölt blokk jelzi a két vektor konkatenációját és az adott súlymátrixokkal és *bias* értékkel való affin transzformációt. Az első három kapu esetében sigmoid (2.9 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 (2.7 egyenlet). Végezetül az így kapott cella aktuális állapotából számítjuk ki az aktuális kimenetet. A kimeneti kapu állítja be, hogy milyen mértékben teszi ezt.

|  |  |  |
| --- | --- | --- |
|  |  | (2.3) |
|  |  | (2.4) |
|  |  | (2.5) |
|  |  | (2.6) |
|  |  | (2.7) |
|  |  | (2.8) |
|  |  | (2.9) |

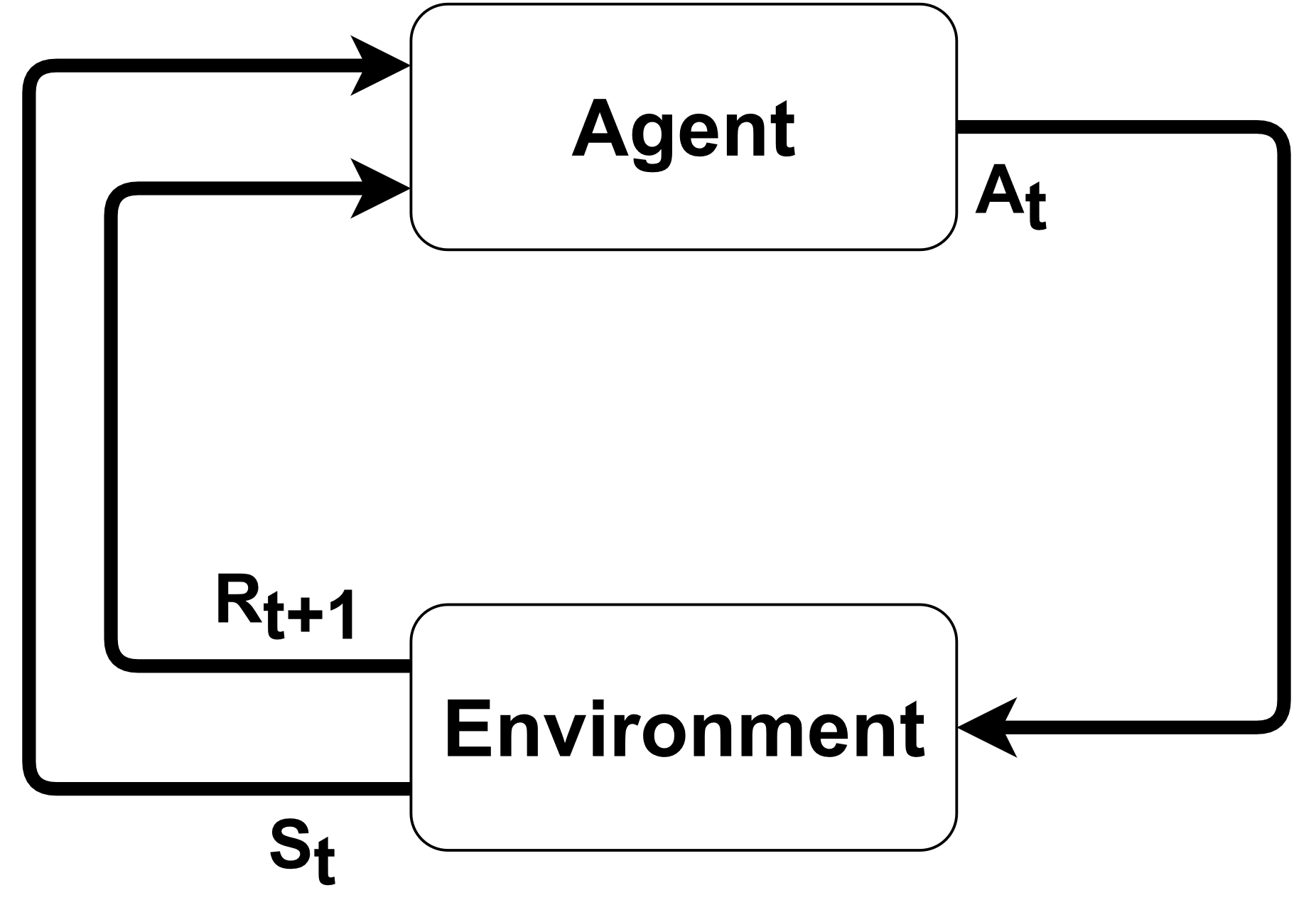


2.7. ábra LSTM cella időben kibontott modellje

Mint említettem, az LSTM képes kezelni az eltűnő gradienseket, ehhez az kell, hogy a és a közötti derivált 1 körül legyen, mert ekkor a derivált sokadik (összes idő-lépésnyi) hatványa sem válik zérussá és nem növekszik minden határon túl. Látható, hogy a szerinti deriváltja csak -től függ, így ez akár lehet egy is (0 és 1 közötti értéket vehet fel a *sigmoid* miatt).

## Megerősítéses tanulás

A megerősítés tanulás során egy úgynevezett ágens interakcióba lép 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, a jutalom jellemzően annál nagyobb. 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.



2.8. 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 (2.11 egyenlet). A jövőbeli diszkontált jutalom (2.10 egyenlet) a jövőbeli jutalmak összege, exponenciálisan súlyozva a diszkont rátával (). 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:

|  |  |  |
| --- | --- | --- |
|  |  | (2.10) |
|  |  | (2.11) |
|  |  | (2.12) |

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 () változó segítségével:

|  |  |  |
| --- | --- | --- |
|  |  | (2.13) |

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 az állapottér 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 vektor (*tuple*) a jelenlegi állapotból, akcióból, az akcióra kapott jutalomból és a következő állapotból áll: . 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.

### Q-tanulás és policy 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:

|  |  |  |
| --- | --- | --- |
|  |  | (2.14) |

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:

|  |  |  |
| --- | --- | --- |
|  |  | (2.15) |

Ne felejtsük el, hogy a jutalom 1-től indexelőik, tehát a nulladik akcióra a jutalom (lásd **2.8. ábra**). Az iteráció felírásához kell még egy összefüggés: az optimális *V* függvény ugyanis felírható az optimális *Q* függvény felhasználásával. Mivel az állapot-érték függvény lényegében a várható értéke az akció-érték függvénynek, ezért könnyen adódik az alábbi formula:

|  |  |  |
| --- | --- | --- |
|  |  | (2.16) |

Az 2.15 és 2.16 egyenletet összerakva kapjuk, hogy

|  |  |  |
| --- | --- | --- |
|  |  | (2.17) |

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

|  |  |  |
| --- | --- | --- |
|  |  | (2.18) |

A *Q* függvény megtanulása viszont rendkívül nehéz feladat lehet a nagy számú, vagy nem diszkrét esetben 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. policy gradiens 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 policy gradiens, azaz a költségfüggvény háló paraméterei szerinti deriváltja a levezetés után a következőképpen néz ki:

|  |  |  |
| --- | --- | --- |
|  |  | (2.19) |
|  |  | (2.20) |

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 figyelmen kívül marad 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 lassan, 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:

|  |  |  |
| --- | --- | --- |
|  |  | (2.21) |

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:

|  |  |  |
| --- | --- | --- |
|  |  | (2.22) |
|  |  | (2.23) |

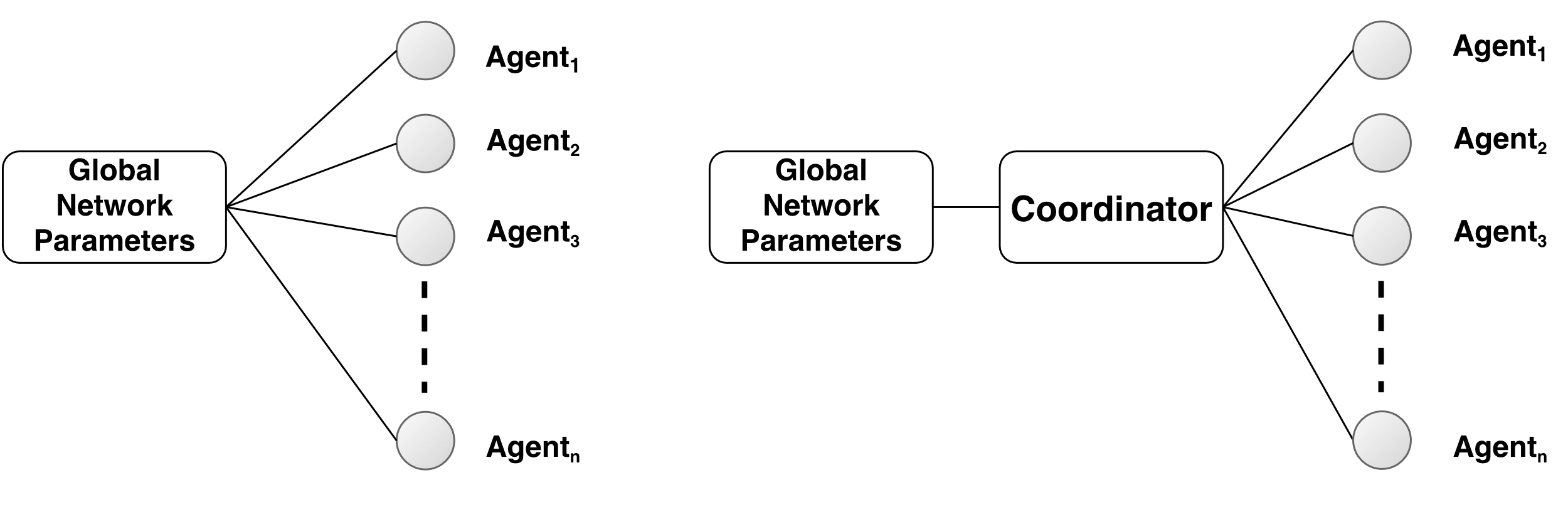
### Actor-Critic

A következő említendő policy 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 2.15 és a 2.22 egyenleteket felhasználva kapjuk meg így az előny függvényt számunkra hasznos alakját:

|  |  |  |
| --- | --- | --- |
|  |  | (2.24) |

E két algoritmus lényege, hogy tanítás alatt több ágens hajt végre akciókat több párhuzamosan futó környezetben, függetlenül egymástól (lásd **2.9. 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.



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

### Proximal Policy Optimization

A PPO lényegében ugyanolyan struktúrájú policy gradiens módszer, mint az A2C, az egyetlen lényeges eltérés az Actor veszteségfüggvényében van. Hasonlóan a TRPO algoritmushoz, a lényege, hogy stabilabbá teszi a tanulást, azáltal, hogy korlátozza a stratégia változásának mértékét. Ehhez azt a megszorítást használja, hogy az új és a régi stratégia KL-divergenciáját korlátozza egy általunk meghatározott érték alá. Ezzel megakadályozva, hogy nagy mértékben eldivergáljon a stratégia.

A PPO egyszerűsít ezen a feltételen: veszi a két stratégia hányadosát (2.25 egyenlet), majd a hányadost beszorítja egy 1 körüli, szűk intervallumba (2.26 egyenlet, ahol egy általunk meghatározott hiperparméter). Végezetül veszi a hányados és a clippelt hányados minimumát és ezzel a taggal súlyozzuk az előny függvényt. A Critic részén a veszteségnél nincs változás.

|  |  |  |
| --- | --- | --- |
|  |  | (2.25) |
|  |  | (2.26) |
|  |  | (2.27) |

A PPO az alkalmazott változtatások miatt ugyan lassabban tanul, nincs meredek felfutása, viszont stabilabb is, melyet az A2C esetén nehéz elérni. A2C esetén lehet például azt tenni, hogy egy elfogadhatónak tűnő állapotból (*checkpoint*) újraindítjuk a tanítást, de már kisebb lépésközzel. Ütemezővel ezt sajnos nem lehet megvalósítani, mert nem lehet kitalálni, hogy mikor érdemes változtatni a lépésközön.

Hátránya, hogy diszkrét akciótér és ritka, nagy jutalmak esetén könnyedén megakadhat szuboptimális állapotban, amelyre már vannak megoldások.

## Imitációs tanulás

Az olyan környezetek esetében, ahol ritka a jutalom gyakorisága komoly kihívás lehet megtanulni az optimális stratégiát megerősítéses tanulás segítségével. Ekkor módosítani kell a jutalom függvényt, hogy gyakrabban lássuk el az ágenst jutalommal. Általánosságban igaz, hogy kézzel létrehozni egy jutalom függvényt, mely a megfelelő viselkedésre sarkallja az ágenst rendkívül komplex feladat.

Az imitációs tanulás során a megerősítéses tanulást problémát (optimális stratégia megtalálása, 2.30 egyenlet) módosítjuk. Nem kell manuálisan implementálni egy jutalom függvényt, amelyből tanulna az ágens. Hanem egy úgynevezett „szakértő” (*expert*) demonstrációkat prezentál az ágensnek, aki ezeket megpróbálja leutánozni, megtanulni. Tehát imitációs tanulás során adottak az expert trajektóriái, ahol az akciók az optimális stratégiát követik. Van, amikor tanítási időben szeretnénk hozzá férni ezekhez, van, hogy csak előre felvett adatbázisra lesz szükségünk. Az alábbiakban bemutatok három gyakori imitációs tanulás metódust, amelyek a veszteségfüggvényben és a tanulás folyamatában különböznek.

### Behavioural Cloning

Továbbiakban BC, a legegyszerűbb módszer, mely lényegében a megerősítéses tanulás probléma lecsökkentése felügyelt tanítássá. Példákat készítünk a környezetben, hogy mi magunk mit tennénk és ezeket tanítjuk meg az ágensnek. A bemenetet független és azonos eloszlású állapot-akció pároknak feltételezzük. A veszteség az elvárt optimális stratégia szerinti akció és az ágens által meghozott akció közt kerül kiszámításra.

A módszernek szemmelláthatóan van egy nagy hátránya. Nagy, komplex környezetekben a demonstrálás kivitelezhetetlen, de kisebb környezetekben is nehéz lefedni minden lehetséges eshetőséget. Általánosítani nehezen fog tudni az ágens, így, ha egy pályát nem fedtünk le, arra nemigen fog tudni rátanulni. Ennek oka, hogy felügyelt esetben a bemeneteket függetlennek vesszük, mely megerősítés tanulás esetében hibás feltevés, hiszen egy állapot az előző állapotból következik. Így az egyes állapotokban előforduló hibák a pálya során összeadódnak és kritikus állapotba kerül és abba benn is ragad az ágens. Ezért csak egyszerű környezetek esetében használható ez az eljárás.

### Direct Policy Learning

A DPL, vagy más néven IIL (*Interactive Imitation Learning*), a BC módszernél már egy szofisztikáltabb megoldás. Ez is felügyelt tanításnak tekinthető, de itt nem mi vagyunk a szakértők, hanem egy interaktív demonstrátor, amelytől a demonstrációkat lekérhetjük tanítás közben (futás időben). A metódus a következő: egy kezdeti stratégiából indulunk ki, és minden iterációban futtatjuk a jelenlegi stratégiánkat (*roll out*). Így kapunk egy trajektóriát, melyet átadunk a demonstrátornak és minden állapotra összevetjük, hogy ő mit tett volna az adott esetben. Ezt kiértékelve felügyelt tanítást alkalmazva frissítjük az ágens stratégiáját és kezdődik elölről a folyamat egészen az optimális állapotig. Látható, hogy a BC lényegében a DPL speciális esete egy lépésre nézve.

Ahhoz, hogy a módszer effektíven működjön, fontos, hogy emlékezzen az ágens a korábbi hibáira, így tanítási időben meg kell őriznünk az összes előző tanító adatot. Legelterjedtebb két algoritmus erre a feladatra a *Data* és a *Policy Aggregation*. Az előbbi esetében minden iterációban az aktuális stratégiát az összes addigi tanító adaton tanítjuk. Az utóbbi esetében csak az aktuális tanító adaton tanítunk, viszont ezután összekombináljuk a jelenlegit az előző iterációbeli stratégiával, a következőképpen:

|  |  |  |
| --- | --- | --- |
|  |  | (2.28) |

Mindkettő módszer bizonyítottan konvergál. A DPL módszer kijavítja a BC gyengeségeit, egyetlen komoly nehézsége a megfelelő demonstrátor megvalósítása.

### Inverse Reinforcement Learning

Az IRL lényege, hogy nem közvetlenül a stratégiát tanulja meg az ágens, hanem a jutalom függvényt egy demonstrátor segítségével, majd ezt alkalmazva végül megtalálja az optimális stratégiát megerősítéses tanulással. Tehát az nem ismert (az MDP felírásából).

A módszer a következő: Összegyűjtjük az expert demonstrációit, amelyet optimálisnak tekintünk, és megbecsüljük a parametrizált jutalom függvényt, amely előidézhette az expert stratégiáját. Tehát először frissítjük a jutalom függvény paramétereit, ezzel a jutalommal folytatunk egy megerősítéses tanulást, megpróbálva megtalálni az optimális stratégiát (2.30 egyenlet). Majd ezt összehasonlítjuk az expert stratégiájával és ezt egészen addig ismételjük, amíg a kettő nem kerül számunkra megfelelően közel egymáshoz.

Két fő megközelítés létezik, a *model-free* és a *model-given* eset. Amikor adott a modell, vagyis ismert a környezeti állapot-átmenet dinamika (), akkor a jutalom függvény lineáris és minden iterációban a teljes RL problémát meg kell oldanunk:

|  |  |  |
| --- | --- | --- |
|  |  | (2.29) |

Mivel láthatóan ez nagyon időigényes lenne ez csak akkor hasznos módszer, ha kicsi az állapotterünk.

A szabad-modell változat általánosabban közelíti meg a problémát. Ezesetben a jutalom függvény komplexebb, amelyet leggyakrabban egy neurális hálózattal modelleznek. Ekkor nem ismert a eloszlás, de feltételezzük, hogy hozzáférünk a környezethez. Az állapottér itt már tekinthető nagynak vagy folytonosnak. Ezért nem a teljes, hanem csak egy lépésnyi megerősítés tanítást végzünk el.

Egyik esetben sincs egyértelmű jutalom függvény, több függvény segítségével is eljuthatunk az optimális (expert) stratégiához. Ennek feloldásához a maximum entrópia tételt (E.T. Jaynes 1957) kell alkalmazni, vagyis mindig a legnagyobb entrópiájú pályaeloszlást () kell választanunk.

|  |  |  |
| --- | --- | --- |
|  |  | (2.30) |
|  |  | (2.31) |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | közvetlen stratégia tanulás | jutalom tanulás | hozzáférés a környezethez | interaktív demonstrátor | előre felvett demonstrációk |
| BC | Pipa egyszínű kitöltéssel | Bezárás egyszínű kitöltéssel | Bezárás egyszínű kitöltéssel | Bezárás egyszínű kitöltéssel | Pipa egyszínű kitöltéssel |
| DPL | Pipa egyszínű kitöltéssel | Bezárás egyszínű kitöltéssel | Pipa egyszínű kitöltéssel | Pipa egyszínű kitöltéssel | *opcionális* |
| IRL | Bezárás egyszínű kitöltéssel | Pipa egyszínű kitöltéssel | Pipa egyszínű kitöltéssel | Bezárás egyszínű kitöltéssel | Pipa egyszínű kitöltéssel |

2.1. táblázat Imitációs tanulás eljárások összehasonlítása

## 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 lép interackióba. 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.

### 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*).

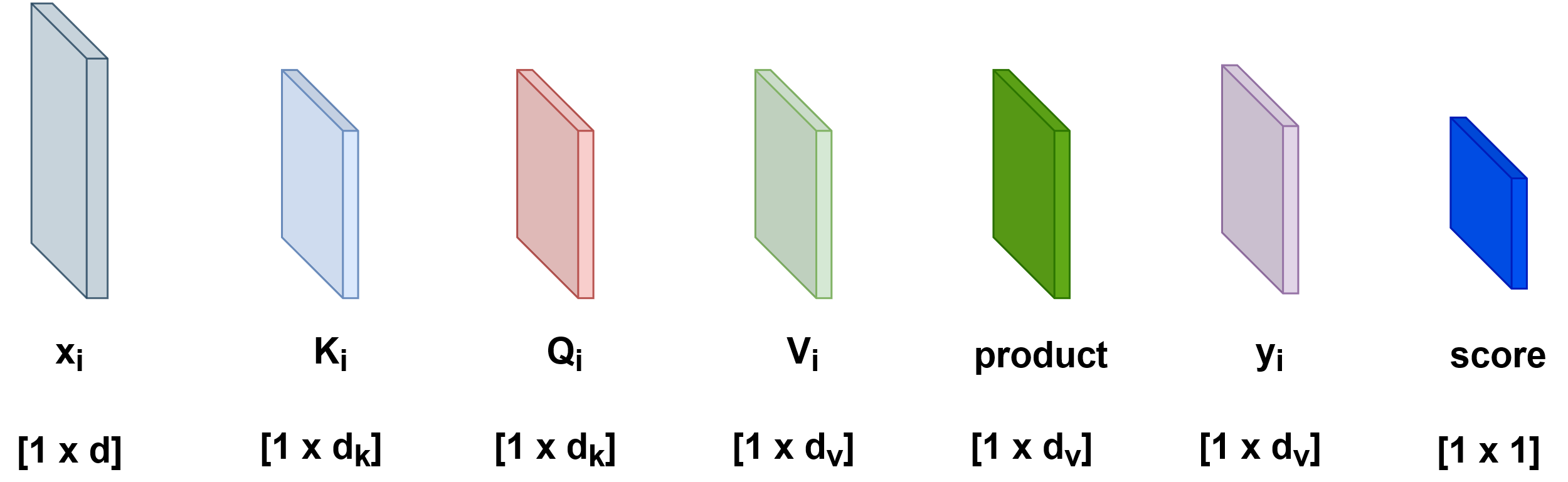
Minden bemeneti vektornak három belső reprezentációja van: egy *key* (**k**), egy *query* (**q**) és egy *value* (**v**) vektor. A *key* vektor identifikálja az elemet, leírja, hogy az adott elem mit tartalmaz. A *query* vektor leírja, hogy mire szeretnénk figyelni, mit keresünk az adott bemenetben. 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 () 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:

|  |  |  |
| --- | --- | --- |
|  |  | (2.28) |
|  |  | (2.29) |

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 **2.11. ábra** érdemes nyomon követni.

Ahhoz, hogy megkapjuk a fent említett belső reprezentációs vektorokat, a bemeneti méretű vektort egy az adott reprezentációhoz tartozó súlymátrixxal (, ,), például a kulcs-mátrixxal szorzunk. Ezeknek a méretei a **2.10. ábra** alapján könnyedén megadhatóak: a **W**K és mátrixnak azonos méretűnek kell lenniük: , míg a egy méretű mátrix lesz (megegyezhet értékével).



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

A bemeneti vektorokat egy méretű mátrixba rendezve a súlymátrixokkal 3 mátrix szorzással megkaphatjuk a reprezentációk , és mátrixait (*N* a szekvencia hossza, azaz a bemeneti vektorok száma). Ezután kiszámoljuk a bemeneti szekvencia első eleméhez (vektorhoz) tartozó figyelem pontot. A **2.10. ábra** látható, hogy ennek az eredménye egy skalár. Vegyük észre, hogy az első bementhez tartozó kimeneti vektor számításához csak az első bemenethez tartozó *query*vektorát kell felhasználni, mivel az első elemhez tartozó figyelem pontokat a vektor határozza meg. Tehát a  méretű vektort szorozzuk az összes *key* reprezentációt tartalmazó -es mátrixxal. Az így kapott 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:

|  |  |  |
| --- | --- | --- |
|  |  | (2.30) |

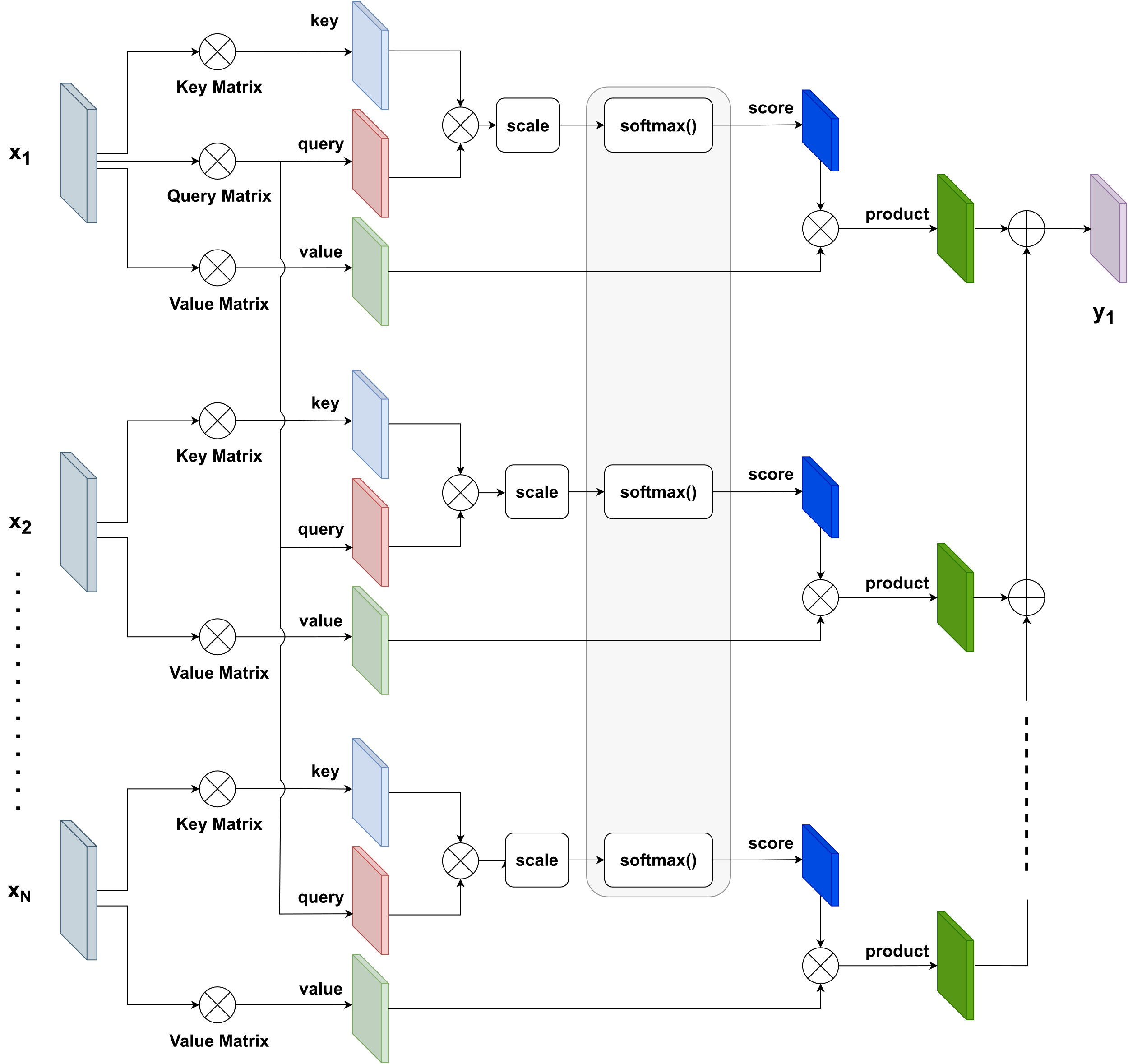
A visszaskálázásra azért van szükség, mert a szorzat, vagyis a logit-ok varianciája szorosukra nőnek, és ez telítésbe viheti az *softmax*-ot. Ezért visszaskálázzuk a szorzat eredményét, hogy a szórás ne változzon.

A következő lépésben a *value* reprezentációkat súlyozzuk a kiszámolt figyelem pontokkal, majd ezeket az méretű súlyozott *value* vektorokat összeadjuk elemenként. Az így megkapott vektor az első bemeneti elemhez tartozó reprezentációja. Láthatjuk, hogy a kimenet méretét a mátrix oszlopainak számával határozhatjuk meg. Legvégül ezt a lépés sorozatot megismételjük a szekvencia minden elemére és így megkapjuk a kimeneti vektorokból álló méretű mátrixot.

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

|  |  |  |
| --- | --- | --- |
|  |  | (2.31) |

Mint említettem, a Self-Attention lényegében egy egyszerű előre csatolt neurális háló. A , , és súlymátrixok a háló paraméterei, vagyis tanításkor ezeknek a mátrixoknak az elemeit frissítjük a hiba-visszaterjesztés során. Az 2.31-es egyenlet pedig felfogható a réteg aktivációs függvényeként. A három súlymátrixot kis értékekkel szokás inicializálni, például Gauss-eloszlással.



2.11. 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 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 mátrixaiból konkatenációval képzett mátrixot jobbról szorozva kapjuk a Multi-Head méretű kimeneti mátrixot bemenetre.

### 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.

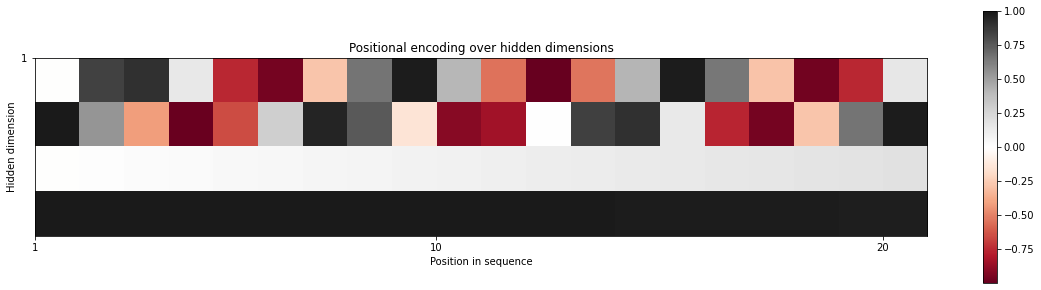
De miként válasszuk meg a kódolás metódusát? Ugyanis egyáltalán nem mindegy, hogy milyen módszert használunk, egy rossz módszer könnyedén lehet kontraproduktív. Néhány példát bemutatok, hogy mikre érdemes figyelni, mielőtt rátérnék arra a módszerre, amit alkalmaztam is. Először is, ha mindegyik bemenethez 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: . 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 minden 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:

|  |  |  |
| --- | --- | --- |
|  |  | (2.32) |

Az egyenletben szereplő jelöli a pozíciót, a rejtett dimenzió méretét, az az adott rejtett dimenzió indexe. Látható, hogy 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 hullámhosszú szinusz- és koszinuszgörbe, míg alatta a hullámhosszú görbék. Vegyük észre, hogy nincs két egyformán kódolt pozíció, azaz két egyforma oszlop.



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

## RAdam

A Rectified Adam egy módosított Adam (Adaptive Moment Estimation [11]), ami egy iteratív, negatív gradiens alapú optimalizáló algoritmus, ez egy 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 , mely nem más, mint a tanulási ráta vagy lépéshossz (szokták -val vagy -val is jelölni), a és , 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 algoritmus megalkotóinak ajánlásai alapján ezeket a következő módon szokás beállítani: , , és . Ha megnézzük az Adam PyTorch-os implementációját, alapértelmezett 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.

|  |  |  |
| --- | --- | --- |
|  |  | (2.33) |
|  |  | (2.34) |
|  |  | (2.35) |

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:

|  |  |  |
| --- | --- | --- |
|  |  | (2.36) |
|  |  | (2.37) |
|  |  | (2.38) |

A probléma az Adammal, hogy kezdetben nagy a lépésköz varianciája, melyet jó lenne csökkenteni a gyorsabb konvergencia érdekében. Erre az egyik módszer a *warmup*, 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 és ezzel stabilabb tanulást eredményezve. A *rectified* megoldás ezzel szemben úgy oldja meg ezt a problémát, hogy először kiszámoljuk az egyszerű mozgó átlag közelítésének (SMA) a maximum hosszát, melyet -val jelölünk. Majd ezt felhasználva minden iterációban kiszámoljuk az közelített SMA hosszát ( és ha ez átlép egy általunk megválasztott küszöböt, akkor változtatunk a tanulási rátán, pontosabban beszorzunk egy ún. *variance* *rectification* (2.42 egyenlet) taggal. Egyéb esetben csak alfa konstans együtthatóval súlyozzuk az első momentumot (2.43 egyenlet).

|  |  |  |
| --- | --- | --- |
|  |  | (2.39) |
|  |  | (2.40) |
|  |  | (2.41) |
|  |  | (2.42) |
|  |  | (2.43) |

# Specifikáció, tervezés

## 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.

## Fejlesztői eszközök

### 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].

### 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.

### 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.

## 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.