Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Irányítástechnika és Informatika Tanszék

Bencsik Blanka

MÉLY NEURÁLIS HÁLÓZATOK HATÉKONY RITKÍTÁSA MODELL ALAPÚ MEGERŐSÍTÉSES TANULÁS ALKALMAZÁSÁVAL

Tudományos Diákköri Konferencia

Konzulens

Dr. Szemenyei Márton

BUDAPEST, 2021

Tartalom

Kivonat

A mély neurális hálók az elmúlt tíz évben kimagasló eredményt mutattak az objektumdetektálás területén, alkalmazásuknak azonban gátat szab a nagy számításikapacitás-igényük és méretük, mely a hatalmas paraméterszámukkal magyarázható. Különösen jelentős ez a probléma az erőforráskorlátos eszközök esetén (pl. mobiltelefonok, beágyazott rendszerek), ahol még a betanított modellek tárolása és futtatása is nehézséget okozhat. A probléma megoldására használatos a ritkítás (ún. pruning eljárás), melynek célja egy neurális háló modell paramétereinek lehető legnagyobb mértékű redukálása a pontosság romlásának elkerülése mellett.

Az eltávolítandó paraméterek kiválasztását jelentősen megnehezíti az, hogy a modell különböző rétegei nem egyformán érzékenyek a paraméterek törlésére. További nehezítő tényező, hogy a redukált modell pontosságromlását nem csak az éppen ritkítandó réteg érzékenysége befolyásolja, hanem az azt megelőző rétegek redukálásának mértéke is. A mély neurális háló modellekben ezen összefüggések feltérképezése a lehetséges variációk hatalmas száma miatt kézzel lehetetlen, csupán feltételezésekre támaszkodhatunk.

A probléma megközelíthető megerősítéses tanulás alkalmazásával, melynek során az ágens emberi beavatkozás nélkül igyekszik megtalálni az optimális eltávolítandó paraméterszámot a modell minden rétegéhez. A probléma ezen megközelítése jelenleg is nyitott téma az irodalomban, ám a létező megoldások legnagyobb hátulütője, hogy az ágens számára a két legfontosabb környezeti változást, a paramétersűrűséget és a pontosságromlást a modell tényleges ritkításával és a redukált modell validációs adatbázison való tesztelésével határozzák meg futási időben, ami nagymértékben lelassítja a tanítási folyamatot.

Munkám során egy olyan megerősítéses tanulás alapú rendszert valósítok meg, mely a YOLOv4 objektumdetektor optimális ritkítását teszi lehetővé a tanítási folyamat idejének csökkentése mellett. Az eddig létező megoldásokhoz képest a rendszert egy olyan neurális hálóval egészítem ki, mely a ritkítandó háló pontosságváltozását és paramétersűrűségét képes megbecsülni az addig törölt paraméterek mennyiségének és az adott réteghez választott redukáló tényező függvényében. Ezen állapotbecslő háló a környezet állapotának meghatározásához szükséges hosszadalmas műveletek szerepét veszi át, nagyságrendekkel növelve így a tanítás sebességét. Az állapotbecslő hálót önfelügyelt tanítás segítségével tanítom be, automatikusan generált adatok segítségével.

A megvalósított módszer eredményei a futási idő és a pontosság- és paramétersűrűség-változás tekintetében state-of the-art pruning módszerekkel kerülnek összevetésre.

Abstract

Deep Neural Networks (DNN) have achieved outstanding results in the field of object detection in the past decade. Unfortunately, their effectiveness is usually accompanied by billions of parameters, therefore huge computational capacity and memory are required to process them. These requirements make the deployment of DNNs challenging, especially in resource-constrained environments, such as mobile phones or embedded systems. One popular approach to overcome this issue is network pruning, which is accomplished by systematically removing parameters from an existing, accurate network. By doing so, a smaller network is produced while maintaining most of the initial accuracy.

The process of choosing the parameters to be pruned is quite demanding, since the different layers in the DNN are not equally sensitive for removing parameters from them. Moreover, the deterioration of the accuracy is not only determined by the sensitivity of a current layer, but also by the amount of removed parameters from all the previous layers. The number of possible variations of these dependencies are so large, they cannot be tried out manually.

This problem can be solved with reinforcement learning, where the agent tries to find the optimal number of parameters to be removed from each layer, without human interaction. This topic is currently an open issue in the literature, however, the main disadvantage of the existing solutions is that they determine the main environmental state variables – the deterioration of the accuracy and the sparsity – by pruning and testing the model on the validation dataset in runtime, which slows down the training procedure extremely.

During my work I implemented a reinforcement learning-based system which is able to prune the YOLOv4 object detector optimally, in addition to decreasing the training time. Compared to existing solutions, my system contains an additional neural network (so called state predictor network) which can predict the deterioration of the accuracy and the sparsity if the reduction coefficient for the current layer and the number of previously removed parameters are given. This network replaces the role of long procedures that were performed to determine environment state, making the training time significantly faster. The state predictor network is trained via self-supervised learning on automatically generated data.

The results of the presented solution are compared to state-of-the-art pruning methods observing runtime and deterioration of accuracy and sparsity ratio.

Bevezetés

A mély neurális hálók az informatika számos területén értek el kimagasló eredményeket az utóbbi években, azonban alkalmazásuk manapság is problémát jelent az erőforráskorlátos eszközökben – például mobiltelefonok, beágyazott rendszerek, stb. – méretük, valamint hely- és számításkapacitás-igényük miatt. Ezt a problémát célozza meg az úgynevezett mély neurális háló ritkítás (neural network pruning), melynek során egy betanított, pontos neurális háló modellből valamilyen struktúra szerint paramétereket (súlyokat) eltávolítva egy kisebb modellt kapunk, a pontosság minimális romlása mellett.

Természetesen ahhoz, hogy a pontosság csak kicsit csökkenjen, nem mindegy, hogy a modellből honnan, milyen szisztéma szerint és mennyi paramétert távolítunk el. Az elmúlt években számos kutatás jelent meg melyben különböző szabályrendszereket dolgoztak ki arra, hogy hogyan érdemes kiválasztani az eltávolítandó súlyokat, úgy, hogy minél jobb eredmény legyen elérhető. Ezen módszereknél már önmagában is hátrány, hogy teljes mértékben emberi beavatkozást igényelnek, de ebből adódóan további hiányosságot jelent, hogy így lehetetlen feltérképezni a lehetséges variációkat azok hatalmas száma miatt.

Az elmúlt 2-3 évben jelentek meg először olyan kísérletek, ahol a neurális háló ritkítás problémáját megerősítéses tanulással való automatizálással közelítik meg. Ezen kutatási terület még kezdeti fázisban van, főképp osztályozó hálókhoz születtek ilyen megoldások és összességében elmondható, hogy nincs egységes kiértékelési szempont definiálva. Az automatizált ritkítás feladata új kihívásokat állít a kutatók elé – miközben természetesen a hagyományos pruning kihívásai továbbra is fennálnak –, például, hogy milyen megerősítéses tanulási algoritmus (ágens) választása a legelőnyösebb, hogyan és milyen sűrűn érdemes jutalmazni az ágenst, vagy, hogy mi alkossa az állapotteret. A felsorolt problémákra több eredményes megoldás is publikálásra került, viszont az eddig létező megoldások közös hátránya az ágens hosszas tanítási ideje. A tanítás során ugyanis a legfontosabb környezeti változókat – a pruning utáni pontosságromlást és a ritkaságot – a ritkított modell validációs adatbázison való tesztelésével határozzák meg, ami nagymértékű lassulást visz a futási időbe.

Munkám során ezt a problématerületet célzom meg, és egy olyan automatikus ritkító rendszert tervezek és készítek a YOLOv4 objektumdetektorhoz, mely az eddig létező megoldásokhoz hasonló jóságú ritkítást valósít meg, az ágens tanítási idejének jelentős csökkentése mellett. Ennek elérése céljából az ágens számára a környezeti változókat a validálás helyett egy úgynevezett állapotbecslő háló segítségével határozom meg, melyet előzőleg automatikusan generált adatokon tanítok be. Az állapotbecslő háló alkalmazása továbbá lehetővé teszi, hogy az ágens tanításakor ne kelljen betölteni a teljes ritkítandó modellt, annak súlyait, csupán az architektúráját. Ennek eredményeképp a tanításhoz jóval kevesebb GPU memóriára van szükség, ami lehetővé teszi a multiágensű tanítást. Ez pedig stabilabb tanuláshoz és gyorsabb konvergenciához vezet. A probléma ezen megközelítése egyedi, legjobb tudomásom szerint az irodalomban eddig nincs rá

példa, ahogyan a YOLOv4 detektor megerősítéses tanulással automatizált ritkítására sem. A rendszer többi részét az irodalomban alkalmazott módszerek ihletésével állítom össze.

Az elkészült rendszer főbb eredményei, melyek a dolgozatban bemutatásra kerülnek a következők:

1. Sikeresen létrehoztam és önfelügyelt tanulással automatikusan generált adatokon betanítottam egy ún. állapotbecslő hálót, mely az automatikus pruning rendszer tanításakor a környezet szimulálására szolgál.
2. Az állapotbecslő háló alkalmazásával lehetőség nyílik a modell alapú, multiágensű megerősítéses tanulás alkalmazása a pruning rendszerben. Ennek eredményeképp az ágens tanítási ideje és a teljes fejlesztési idő a rendelkezésre álló GPU mellett nagyságrendekkel kisebb, mint a már létező automatikus ritkító módszerek alkalmazása esetén.
3. Az elkészült automatikus pruning rendszer felhasználásával előállt a YOLOv4 objektumdetektor olyan ritkított modellje, mely ritkaság és pontosságromlás szempontjából felülmúlja a saját, kézi beállításokkal ritkított változatokat.

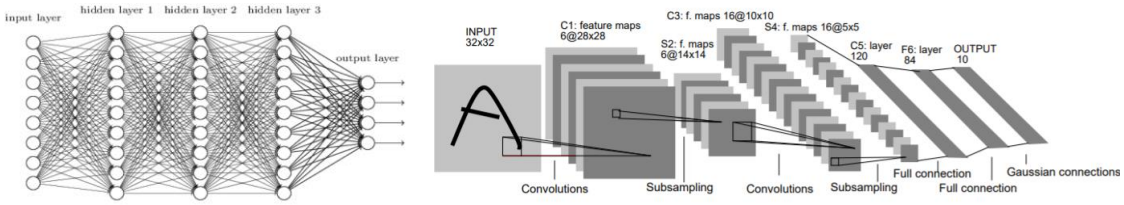
A dolgozatban először a konvolúciós neurális hálózatok, az objektumdetektorok, a pruning és a megerősítéses tanulás elméleti hátteréről olvashatunk. Ezek után egy irodalmi áttekintés következik, ahol mind a hagyományos, mind az automatikus pruning területén publikált megoldások módszereit és eredményeit foglalom össze, utóbbit valamivel részletesebben. Ezt követi a dolgozatban megoldandó probléma definiálása, majd a megvalósításhoz alkalmazott módszerek ismertetése. Végül az Eredmények fejezetben a fejlesztés során felmerült nehézségekről és azokra nyújtott megoldások eredményéről lesz szó, továbbá az elkészült automatikus ritkítást megvalósító rendszer értékelése és state-of the-art pruning módszerekkel való összevetése is itt olvasható. Végül a rendszer továbbfejlesztési lehetőségeire teszek javaslatokat.

1 Elméleti háttér

1.1 Konvolúciós neurális hálózatok

1.1.1 Konvolúciós neurális hálózatok felépítése

A mesterséges neurális hálózatok olyan, az emberi idegrendszer működése ihlette algoritmusok, melyek a bemeneti adat különböző tulajdonságainak felismerésére képesek. A mesterséges intelligencia ezen formája számos gépi tanulási feladat alapját képezi. A neurális hálók topológiájukat illetően többfélék lehetnek attól függően, hogy milyen rétegekből építjük fel őket. A két legjellemzőbb réteg a teljesen kapcsolt (lineáris) és a konvolúciós réteg. Teljesen kapcsolt rétegek kapcsolatot teremtenek az összes kimenő és bemenő neuron között, míg konvolúciós réteg alkalmazásával a kimenő neuronok csupán néhány, a lokális környezetükben lévő bemenő neuronnal állnak kapcsolatban [1] [2].



1.1. ábra. Teljesen kapcsolt [2] és konvolúciós neurális háló [3] architektúrák.

A konvolúciós rétegekből felépített neurális hálót konvolúciós neurális hálónak nevezzük. A képfeldolgozási problémák megoldásához ilyen típusú neurális hálózat alkalmazása a jellemző, azon előnyős tulajdonságai miatt, hogy – a teljesen kapcsolt hálóval ellentétben – jól ki tudja használni a képjellemzők térbeli struktúráját, ráadásul a rétegek közti kevesebb kapcsolat miatt kevesebb paraméterrel is rendelkezik [4]. A háló bemenetén a kép mint egy háromdimenziós tömb jelenik meg. Ebben a reprezentációban a képpontok felelnek meg a neuronoknak. Az első konvolúciós rétegben lévő szűrők állítják elő a kimeneti aktivációs tömböket (vagy kimeneti csatornákat). A kimenetként előállt csatornák a következő konvolúciós réteg bemenetét képezik.

A konvolúciós neurális háló tartalmazhat még különböző le- és felskálázó (ún. pooling) rétegeket is. Azt, hogy a konvolúciós neurális hálókban milyen sorrendben követik egymást az egyes konvolúciós, lineáris és pooling rétegek, milyen mély a háló és milyen méretűek az egyes rétegek, a háló architektúrájának nevezzük [2]. Általánosságban azonban elmondható, hogy a konvolúciós neurális hálók első rétegei valamilyen egyszerű képjellemzőt – például éleket, sarkokat, stb. – detektálnak, majd a következő rétegek ezek egyre komplexebb tulajdonságait érzékeli [2].

1.1.2 Neurális hálók tanítása

A neurális háló modell tanítása során azt szeretnénk, ha a neuronok súlyainak értékét úgy tudnánk változtatni, hogy a háló által generált kimenetek minél jobban közelítsék az előírt kimeneteket. Felügyelt tanulás során az elért pontosság úgy határozható meg, hogy minden bemenethez meghatározunk egy hibafüggvényt a két kimenet alapján. Erre a célra a leggyakrabban alkalmazott hibafüggvények az egyszerű négyzetes hiba (Mean Square Error - MSE), az úgynevezett Hinge hibafüggvény és a keresztentrópia (Cross Entropy Loss). A hibafüggvényeket legtöbbször ún. regularizációs büntetőtaggal (R) együtt szokták használni, mellyel csökkenthető a túl magabiztos modell, azaz a túltanulás (overfitting) jelensége. A két legegyszerűbb regularizációs tag a súlymátrix abszolút- és négyzetes hibája (L1 és L2). Az így előállt hibafüggvény:

$$L = \sum_{i}^{N} L_i + \lambda R(W) \qquad \textbf{1-1}$$

A megfelelő súlyok megtalálásához ezt a hibafüggvényt kell minimalizálni. Erre a célra használatos az iteratív gradiens módszer, mely azt a lehetőséget használja ki, hogy a hibafüggvény a súlyok szerint deriválható. Ezáltal minden lépésben úgy változtatja a súlymátrixot, hogy az a legmeredekebb lejtő felé lépést eredményezi a hibatérben, egészen addig, amíg el nem éri annak globális minimumát [5]. Ez súlymátrix változtatásának folyamata a következőképp írható le:

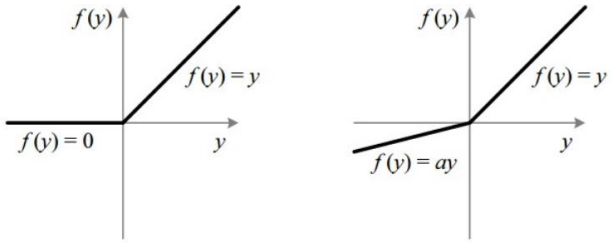
$$W_{k+1} = W_k - \alpha \frac{\partial \|L\|^2}{\partial W} \qquad \textbf{1-2}$$

ahol W a súlymátrix, L a hibafüggvény, α pedig a tanulási ráta (learning rate), azaz a lépésméret, mellyel a globális minimum felé haladunk [3]. A gradienst módszert jellemző valamilyen momentummal kiegészíteni, mellyel elkerülhető a lokális minimumokba való beragadás. Ezesetben a gradienst nem csupán lokális adatokból határozzuk meg, hanem valamekkora súllyal beleszámítjuk az előző időpontban meghatározott gradienst is [6]. Néhány elterjedten alkalmazott gradiens-alapú optimalizációs algoritmus például az SGD (Stochastic Gradient Descent), Adam, AdaGrad, RMSProp.

A mély neurális hálók tanítása esetén a hibafüggvény súlyok szerinti deriváltjának számítására használatos a hátraterjesztés (backpropatagation) algoritmus. Alapötlete, hogy a többrétegű neurális háló elképzelhető úgy, mint egy irányított számítási gráf, melynek csomópontjai a neuronmodellek [5]. A láncszabályt alkalmazva a gráfon hátrafelé haladva bármely csomópont bemenet és súly szerinti deriváltját meghatározhatjuk, hiszen a legutolsó csomópont deriváltja ismert: a háló kimenete nem más, mint maga a hibafüggvény, aminek saját maga szerinti deriváltja 1 [2] [7].

Az eddigiek ismeretében összefoglalható a konvolúciós neurális háló működése: a bemenetére érkező adatból a súlymátrixai felhasználásával transzformálva azt, valamilyen jellegzetes tulajdonságát kiemelő kimenetet állít elő. Ezen komplex eredmény eléréséhez nem elegendő csupán lineáris transzformációk sorozata – márpedig a konvolúció lineáris művelet. Ezért az architektúrába az egyes rétegek közé nemlineáris

függvényeket szokás beékelni. Ezek közül néhány sűrűn alkalmazott például a ReLu (Rectified Linear Unit), Leaky ReLu vagy az ELU (1.2. ábra).



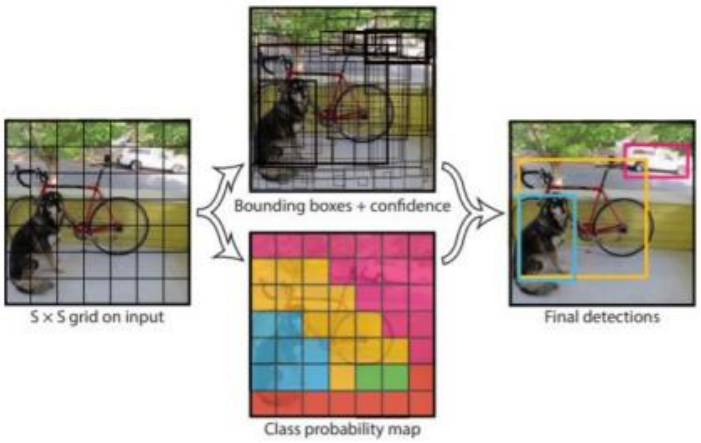
1.2. ábra. ReLu és Leaky ReLu aktivációs függvények [8].

1.2 YOLOv4 objektumdetektor

A neurális háló alapú objektumdetektorok két nagy csoportba oszthatók: régió alapú módszerek és ún. single-shot detektorok. Az első kategóriába tartozó detektorok a képeken először régiójavaslatokat keresnek, majd ezeken külön-külön végeznek objektumdetektálást konvolúciós neurális hálóval [9] [10]. A második kategória detektorai ezzel ellentétben egy lépésben, az egész képen egyidejűleg végzik el az objektumok osztályozását és detektálását is [11] [12] [13] [14] [15].

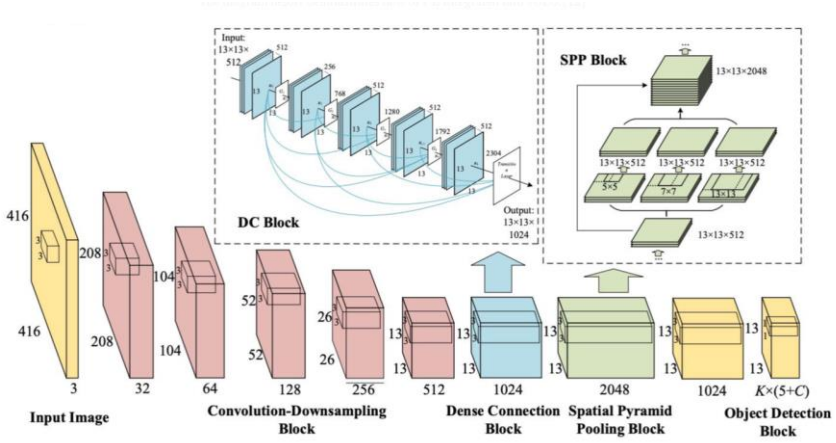
Az egyik népszerű, elsők közt megjelenő single shot típusú objektumdetektor a YOLO (You Only Look Once) [11] volt. Működésének alapelve (1.3. ábra), hogy a képet cellákra osztja egy SxS méretű rács segítségével. A háló minden cellához jósol B darab befoglaló téglalapot, melyet 5 paraméter határoz meg: x, y, w, h, c. Ezek közül (x, y) koordináták jelölik a jósolt befoglaló téglalap középpontját a cella bal felső sarkához viszonyítva, w és h pedig a téglalap szélességét és magasságát adják meg a teljes kép méreteihez viszonyítva. A c paraméter egy konfidenciaérték, ami azt mutatja, hogy mennyire biztos a modell abban, hogy az adott téglalap tartalmaz egy objektumot, illetve milyen pontosan illeszkedik az objektumra. Ezen kívül a háló jósol minden téglalaphoz még C darab paramétert, ahol C az osztályok száma, és az értékek azt mutatják, hogy melyik osztályba tartozik az adott téglalap által jósolt objektum. A jóslatok ekkor SxSx(B*5 + C) dimenziójú tenzorok formájában jelennek meg. A tanítás során a YOLO az egész képet vizsgálja, és a súlymátrix optimalizálásával egyidejűleg történik a detektáláshoz felhasznált paraméterek optimalizálása is [11].

A YOLO első verziója 2016-ban jelent meg, és azóta már több változata is publikálásra került. A YOLOv2-ben [12] (2017) a legjelentősebb változás az úgynevezett anchor tégalapok (anchor box) bevezetése volt. A cellákhoz jósolt téglalapok szélesség és magasság paramétereit itt már nem a teljes kép méretéhez viszonyítva adták meg, hanem az anchor téglalapok méretéhez képest. A YOLOv3 [13] architektúra 106 rétegből áll, egy jóval nagyobb, 53 rétegű Darknet gerinchálózatra épül. Több skálafaktor mellett végzi a detektálást, továbbá az előreterjesztés során időnként a néhány réteggel hátrébb lévő aktivációs térképeket összevonja, ami segít megőrizni a kis objektumok tulajdonságait is.



1.3. ábra. YOLO detektor működésének szemléltetése [11].

A negyedik verziója, a YOLOv4 [14] ma is széleskörűen alkalmazott single-shot detektor. A 2020-ban megjelent detektor korábbi verzióihoz képest speciális bővítményekkel lett kiegészítve, melyek lehetővé teszik a nagyobb érzékelési tartományt, javítják a figyelmi mechanizmust, és erősítik a képi jellemzők összevonásának képességét. A 160 rétegből álló architektúrája a CSPDarknet53 gerinchálózatra épül, ami egy ún. Spacial Pyramid Pooling blokk követ, melynek célja a jelentős kontextus jellemzők kiemelése (1.4. ábra). Már a korábbi verzióira is jellemző, hogy az előterjesztés során helyenként a néhány réteggel hátrébb lévő aktivációs térképeket összevonja, ami segít megőrizni a kis objektumok tulajdonságait is. A paraméterek összevonására az ún. PANet módszert alkalmazzák, és mindemellett egy új Mish nevű aktivációs függvényt használnak.



1.4. ábra. YOLOv4 detektor felépítése [16].

1.3 Ritkítás (pruning)

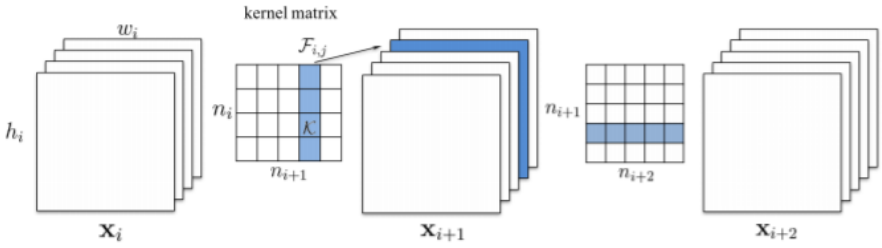
Attól függően, hogy egy mély neurális hálóból milyen struktúra szerint távolítunk el paramétereket (súlyokat), beszélhetünk nem strukturált és strukturált pruning eljárásról. Első esetben a konvolúciós rétegekből egyesével törlünk súlyokat, ezesetben kinullázzuk azokat. Bár ez valóban egy aktív neuronjaiban ritkább hálót eredményez, mivel az eredeti architektúra mérete nem változik, a számítási költségeket nem csökkenti

kellőképpen. Strukturált pruning esetében a súlyok egy összefüggő csoportját töröljük egyszerre, például szűrőket vagy akár teljes konvolúciós rétegeket. Itt a törlés már ténylegesen a szűrő vagy a réteg eltávolítását jelenti, ezáltal tulajdonképp egy új, kisebb architektúrát kapunk. Ezen módszer esetén a számítási kapacitás csökkenése nyilvánvaló, hiszen a paraméterek eltávolításával csökken az elvégzendő mátrixszorzás műveletek száma [17].

Jelölje n_i az i. konvolúciós réteg bemeneti csatornáinak számát, h_i és w_i pedig legyenek az adott réteg bemenet aktivációs tömbjeinek magassága és szélessége. Az i. konvolúciós réteg az $x_i \in \mathbb{R}^{n_i \times h_i \times w_i}$ bemeneti aktivációs tömböket $x_{i+1} \in \mathbb{R}^{n_{i+1} \times h_{i+1} \times w_{i+1}}$, kimeneti aktivációs tömbökké transzformálja, melyek a következő konvolúciós réteg bemeneteit képezik. Ehhez n_{i+1} darab háromdimenziós $\mathcal{F}_{i,j} \in \mathbb{R}^{n_1 \times k \times k}$ szűrővel kell végighaladni az n_i darab bemeneti csatornán, és minden szűrő egy kimeneti aktivációs térképet képez. Az imént említett háromdimenziós szűrőket n_i darab kétdimenziós $\mathcal{K} \in \mathbb{R}^{k \times k}$ szűrő alkotja. Ebből adódik, hogy egy konvolúciós réteg feldolgozása során az elvégzett műveletek száma:

$$n_{op} = n_{i+1} * n_i * k^2 * h_{i+1} * w_{i+1} \qquad \textbf{1-3}$$

Ha az x_i bemeneti csatornák közül eltávolítjuk a j-ediket, akkor a hozzá tartozó $\mathcal{F}_{i,j}$ háromdimenziós szűrő is törlődik, továbbá az $x_{i+1,j}$ kimeneti aktivációs térkép is, melyet a törölt szűrő hozott volna létre. Ez a lépés $n_i * k^2 * h_{i+1} * w_{i+1}$ darabbal csökkenti a műveletek számát. Mivel a kimeneti aktivációs térképek alkotják a következő konvolúciós réteg bemenetét, törlődik az a szűrő is, mely az eltávolított csatornán végezné a transzformációt az $i+1$. rétegben. Ezzel további $n_{i+2} * k^2 * h_{i+2} * w_{i+2}$ darabbal csökken az elvégzendő műveletek száma. Az i. konvolúciós réteg m darab csatornájának eltávolításával m/n_{i+1}-gyel csökken mind az $i.$, mind az $i+1$. réteg számítási költsége [17].



1.5. ábra. A strukturált pruning szemléltetése egy konvolúciós rétegen [17].

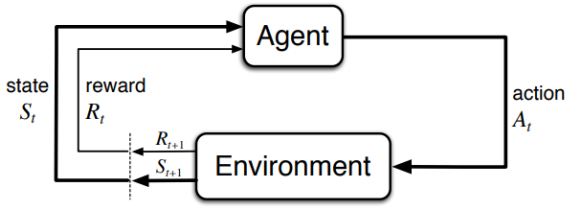
A ritkítás után az előállt modellt sokszor szokás néhány iteráció erejéig tanítani, hogy javuljon az eljárás során valamelyest leromlott pontossága. Ezt a lépést finomhangolásnak nevezzük. Fontos még megjegyezni, hogy nem ritka jelenség, hogy a ritkított modell pontossága felülmúlja az eredeti, kiindulási modell pontosságát. Ez azzal magyarázható, hogy a pruning tulajdonképp egy regularizációs eljárás, hiszen a súlyok törlésével csökken a modell magabiztossága, ezáltal az overfitting mértéke is [17].

1.4 Megerősítéses tanulás

1.4.1 Megerősítéses tanulási probléma

A tanuló algoritmusokat a tanító adatbázis milyenségéből adódóan több kategóriába sorolhatjuk. Az első nagy csoport a felügyelt tanítás (supervised learning), amikor a tanító adatok mellett rendelkezésünkre áll a hozzájuk tartozó címke is, ami az elvárt kimenetet határozza meg. Ilyenkor az algoritmus feladata a címkék minél pontosabb becslése; A címkék segítségével minden döntéshozás után a tanító algoritmus által jósolt kimenet összehasonlításra kerül az elvárttal, ezáltal az algoritmus alkalmazkodni tud a problémához, és idővel egyre pontosabban teljesít. Ezzel ellentétben, a felügyelet nélküli tanítás (unsupervised learning) során a kimenet egyáltalán nem ismert, a cél az adatok közti hasonlóságok és különbségek felismerése valamilyen belső, definiálatlan struktúra szerint. A két tanulási forma kombinációja a félig felügyelt tanulás (semi-supervised learning), ekkor a tanító adatok csak egy bizonyos része van felcímkézve. Megemlítendő továbbá az önfelügyelt tanulás (self-supervised learning), amikor az algoritmus a felügyelt tanuláshoz hasonlóan az adatbázis címkékkel van ellátva, ám azok generálása automatikusan történik [1].

Harmadik nagy csoport a megerősítéses tanulás (reinforcement learning), melynek során a cél lehet egy jó viselkedés, műveletsorozat vagy akár címke meghatározása. A tanuló algoritmus (ágens) egy külső környezettel interakcióban állva próbálja önállóan megmegoldani a feladatot. Ez a környezet különböző állapotokkal írható le, ezeket megfigyelve dönt az ágens minden lépésben az akcióról, melyet végrehajtva a környezet új állapotba kerül. A felügyelt tanulással ellentétben, itt nem tudja meg, hogy mi lett volna a helyes döntés, viszont a környezet bizonyos időközönként visszajelzést ad az ágensnek. Ezt jutalomnak nevezzük, és a feladat teljesítésének minősége származtatható belőle. Ennek következtében az ágens igyekszik az akciók megválasztásához egy olyan stratégiát kialakítani, melynek alkalmazásával a lehető legtöbb jutalmat kapja [18].



1.6. ábra. Az ágens és a környezet interakciója.

Az előző bekezdésben ismertetett probléma matematikailag a Markov döntési folyamattal adható meg, mely formálisan a következőképp írható le: $(S, \mathcal{A}, \mathcal{R}, \mathcal{P}, \gamma)$, ahol S az állapottér, $\mathcal{A}$ az akciótér, $\mathcal{R}(s, a)$ a jutalmak eloszlása egy adott állapot-akció párhoz, $\mathcal{P}(s, a)$: a következő állapot bekövetkezésének eloszlása, γ pedig a diszkont ráta, mellyel a korábbi jutalmakat súlyozzuk [18]. Az ágens döntési stratégiáját a policy adja meg, mely egy olyan függvény, ami minden állapothoz hozzárendel egy akciót:

$$\pi(a,s) = P(a_t = a|s_t = a) \quad \textbf{1-4}$$

Az ágens π stratégiát követve egy állapotból egy akció hatására a következő állapotba kerül, miközben egy bizonyos jutalmat kap. Ezt az állapot-akció-jutalom hármast trajektóriának nevezzük. Ezek ismeretében definiálható az értékfüggvény, mely azt adja meg, hogy π stratégiát követve, s állapotból indulva mi a várható teljes jutalom:

$$\pi(a,s) = P(a_t = a|s_t = a) \quad \textbf{1-5}$$

továbbá a Q-függvény, mely egy állapot-akció párhoz rendeli hozzá a jövőbeli teljes jutalom várható értékét:

$$Q^{\pi}(s,a) \;=\; E[\sum_{t\geq 0} \gamma^t r_t | s_0 = s, a_0 = a, \pi] \quad \textbf{1-6}$$

Attól függően, hogy az ágens milyen függvényeken keresztül keresi az optimális stratégiát különböző megerősítéses tanulási módokról beszélhetünk. A Q-Tanulás (Q-Learning) során az ágens az optimális Q-függvény megtalálására törekszik, majd ebből származtatja a stratégiát. Az optimális Q-függvény fontos tulajdonsága, hogy teljesíti az ún. Bellman-egyenletet, miszerint egy adott állapot-akció párból a szerezhető legnagyobb jutalom egyenlő a közvetlenül kapott jutalom és a következő állapotból elérhető legnagyobb jutalom összegével [19]. Ezt felhasználva definiálható a paraméterfrissírési-szabály és a költségfüggvény:

$$\hat{Q}(s,a,\theta) \;=\; E[r \;+\; \max_{a'} \hat{Q}(s',a',\theta)] \quad \textbf{1-7}$$

$$\Delta Q \;=\; r \;+\; \max_{a'} \hat{Q}(s',a',\theta_{i-1}) \;-\; Q(s,a,\theta_i) \quad \textbf{1-8}$$

$$L_i(\theta_i) \;=\; E[{\Delta\theta_i}^2] \quad \textbf{1-9}$$

$$\frac{\partial L_i(\theta_i)}{\partial \theta_i} = \mathrm{E}[\Delta Q \frac{\partial Q(s,a,\theta_i)}{\partial \theta_i}] \quad \textbf{1-10}$$

ahol θ jelöli a háló paramétereit, ΔQ a Bellman-egyenlet hibáját, L_i pedig a költségfüggvényt jelöli. Ezáltal a tanítás során minden iterációban frissül a Q-függvény, amíg az az optimális értékhez nem konvergál.

A másik csoportot a Policy Gradiens alapú módszerek alkotják, melyek a stratégiát közvetlenül, értékfüggvény közbeiktatása nélkül tanulják. A legegyszerűbb Policy Gradiens alapú algoritmus az ún. az ún. Monte Carlo Policy Gradient (REINFORCE). Ennek működése a következőképp értelmezhető: a háló adott állapot esetén valószínűségeket rendel az akciókhoz, majd a legnagyobb valószínűséggel rendelkezőt végrehajtja, amiért valamilyen jutalmat kap a környezettől. Ha a jutalom jó, akkor úgy módosul a háló, hogy legközelebb ezt az akciót nagyobb valószínűséggel hajtsa végre, amennyiben viszont a jutalom rossz, az adott akció végrehajtásának valószínűségét csökkenti. A tanítás végére így az akciók sorozata, tehát a stratégia az optimálishoz konvergál. A REINFORCE algoritmus esetén a költségfüggvény a következőképp definiálható:

$$J(\theta) = E[r(\tau)] = \oint_{\tau} r(\tau)\, p(\tau, \theta) \quad \textbf{1-11}$$

ahol θ a háló paramétereit τ a stratégia követése mellett kialakult trajektóriát, $r(\tau)$ a jutalomfüggvényt, $p(\tau, \theta)$ pedig egy akció meglépésének valószínűségét jelöli adott trajektória függvényében. A költségfüggvény θ szerinti deriváltja a háló kimenetének ($\pi_\theta(a_t|s_t)$stratégia) logaritmusának deriváltjával arányos, melynek elő előállítása a Monte-Carlo becslési módszerrel lehetséges [7]:

$$\nabla_\theta J(\theta) = \sum_{t \geq 0} r(\tau) \nabla_\theta log \pi_\theta(a_t|s_t) \quad \textbf{1-12}$$

1.4.2 Multiágensű Actor-Critic módszerek

A Policy Gradiens módszerek egy jellegzetes problémája, hogy a jutalmak minden esetben nemnegatív számok, így tulajdonképp minden visszajelzéskor megerősítjük a hálót a döntésében, csak éppen rossz döntéskor kisebb mértékben. Erre a problémára egy lehetséges megoldás lehet, ha akkor tekintünk egy jutalmat jónak, ha az nagyobb, mint az adott állapotból elérhető jutalom várható értéke. Ennek meghatározásához definiálható az ún. előnyfüggvény, mely az állapot-akció párhoz tartozó Q-függvény és az állapothoz tartozó értékfüggvény különbsége:

$$A^{\pi_\theta}(s, a) = Q^{\pi_\theta}(s, a) - V^{\pi_\theta}(s) \quad \textbf{1-13}$$

Ezáltal a Policy Gradiens költségfüggvényének deriváltja:

$$\nabla_\theta J(\theta) = \sum_{t \geq 0} (Q^{\pi_\theta}(s_t, a_t) - V^{\pi_\theta}(s_t)) \nabla_\theta log \pi_\theta(a_t|s_t) \quad \textbf{1-14}$$

Az ezen ötleten alapuló módszereket Actor-Criric módszereknek nevezzük, ahol a megnevezések két külön hálót jelölnek: az Actor háló az optimális stratégia előállításáért felelős, melyhez Policy Gradienst használ, míg a Critic háló feladata az előnyfüggvény előállítása Q-tanulás segítségével [18].

Az Actor-Critic módszerek rendkívül jól teljesítenek a megerősítéses tanulás feladatainak megoldásában. Számos különböző változatuk létezik, melyik közül a legjelentősebbek: Asynchronous Advantage Actor-Critic (A3C), A2C, Deep Deterministic Policy Gradient (DDPG), Proximal Policy Optimization (PPO) [20] [21] [22]. Ezek közül az A2C és a PPO alkalmazása valósul meg a fejlesztés során, ezért működési elve részletesebben is bemutatásra kerülnek, mivel azok szerves részét képezik a dolgozatnak.

Az eddig ismertetett három tanulási forma egymáshoz képest jelentős előnyökkel és hátrányokkal rendelkezik, közös azonban bennük, hogy egyszerre egy ágens lép interakcióba a környezettel. Egy 2016-ban megjelent keretrendszer lehetővé teszi több megerősítéses tanulási algoritmus hatékonyabb alkalmazását, egyszerre több ágenst is futtatva a környezet több példányán [20]. Ezen keretrendszert a 2.4. fejezetben röviden bemutatott Advantage Actor-Critic módszer köré építve kapjuk az ún. A2C algoritmust,

mely ma is széleskörűen alkalmazott eljárás. A párhuzamosan futó ágensek képesek a környezet különböző részeinek feltérképezésére, továbbá különböző keresési stratégiák alkalmazására. Egyes epizódokban a tanuló ágensek tapasztalatait összevonva kerül frissítésre a tanított modell, ami stabilabb tanítási folyamatot eredményez. Ennek következtében az addig erre a problémára megoldást nyújtó tapasztalat visszajátszás módszere [23] is elhagyható, mely csökkenti az algoritmus memória- és a számításikapacitás-igényét.

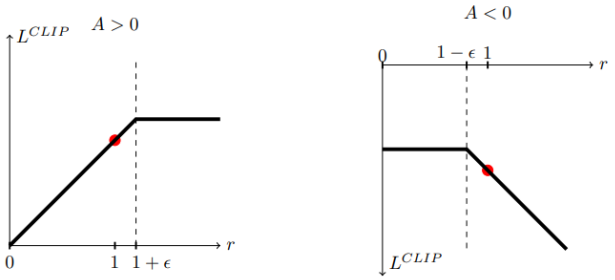
Egy másik elterjedt, stat-of-the-art algoritmus az ún. PPO (Proximal Policy Optimization Algorithm) [22]. Ez az A2C-hez hasonlóan több ágenst futtat egyszerre, újítása azonban az actor speciális költségfüggvényében rejlik, melynek alkalmazásával elkerülhető a túl magabiztos stratégia problémája. Ennek eléréséhez minden lépésben figyelembe veszi az előző lépésbeli stratégiát, és megakadályozza, hogy egy akció valószínűsége sokkal nagyobb legyen az adott lépésben, mint az előzőben volt. Jelölje r_t a régi és az új stratégia valószínűségi arányát:

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} \qquad \textbf{1-15}$$

Ismerve a megfigyelt akciók és állapotok egy sorozatát, r_t nagyobb, mint 1, ha az a_t akció valószínűsége nagyobb az új stratégia szerint, mint az utolsó paraméterfrissítés előtt. Ekkor a költségfüggvény a következőképp definiálható:

$$L^{CLIP}(\theta) = \hat{E}_t\left[min\left(r_t(\theta)\widehat{A_t}, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\widehat{A_t}\right)\right] \qquad \textbf{1-16}$$

ahol ϵ az ún. vágási tényező, értéke általában ϵ=0.2 körül mozog. Ennek értelmében a hiba megfelel r_t arány és az A_t előnyfüggvény szorzatának, abban az esetben, ha az r_t arány az 1 ϵ sugarú tartományába esik. Ellenkező esetben az szorzat értéke vágásra kerül.



1.7. ábra. Költségfüggvény vágásának szemléltetése [22].

Ez a költségfüggvény gyakorlatilag ugyanazt a viselkedést valósítja meg, mint a korábban publikált TRPO módszer [24]. A TRPO azonban rendkívül összetett algoritmus, a PPO sokkal egyszerűbb módon valósítja meg ugyanazt a működést, implementációja is egyszerűbb, és ezáltal sokkal elterjedtebben alkalmazott módszerré vált.

2 Publikált munkák

2.1 Hagyományos pruning eljárás

A pruning eljárás már az 1980-as években is ismert volt, alkalmazására azonban csak az elmúlt évtizedben nőtt meg az igény a mély neurális hálók elterjedésének köszönhetően. Ebben az időszakban számos publikáció és kísérlet jelent meg, melyekben a leghatékonyabb ritkítási stratégiát keresték nagyon mély konvolúciós neurális hálókhoz. Ezen módszerek mind arra törekedtek, hogy valamilyen gondosan szerkesztett szabályt definiáljanak az eltávolítandó paraméterek kiválasztására. Miután gyorsan kiderült, hogy a véletlenszerű ritkítást felülmúlja a nagyság alapú ritkítás, sok kísérletben azt vizsgálták, hogy mi az az optimális határérték, ami alatt már törölni érdemes a paramétereket [25]. Mások ettől a módszertől elrugaszkodva, sokkal komplexebb feltételeket szabtak, melyben nem csak az adott paraméter nagyságát vizsgálták, hanem azt is, hogy az egyes összeköttetések az architektúrában milyen befolyással vannak a háló teljes szerkezetére, miben járulnak hozzá annak pontosságához [26] [27].

A YOLO detektorok ritkítására is egyelőre csak kézzel szerkesztett, szabályszerűségekre alapozott módszerek léteznek, viszont a gondos tervezés következtében ezekkel nagyon jó eredményeket értek el. A 2020-ban megjelent YOLObile keretrendszer [28] a rétegek blokkos ritkításával a YOLOv4 architektúrában közel 93%-os ritkaságot ért el, míg a mAP mindössze 8.3%-ot romlott. Hasonlóan kiváló eredményt mutat a 2021-ben megjelent YOLO-Tight [29] módszer, mely a YOLOv3 architektúra ritkításához az ún. sparsity training-et alkalmazza, melynek során megmutatkoznak azok a paraméterek, melyek együttesen vannak jó hatással a háló pontosságára. Ezzel a megközelítéssel a paraméterek közel 90%-ának eltávolítása mellett a pontosság alig romlott. Szintén az idei évben publikálták a Micro-YOLO detektor ritkítását megvalósító keretrendszert, mely a paraméterek negyedét távolította el az architektúrából 3.1% mAP csökkenés mellett [30].

Azt, hogy a jobbnál jobb eredmények közül melyik a legeredményesebb, ma is nagyon nehéz megmondani, ami azzal magyarázható, hogy nincsenek egységes tesztelési körülmények definiálva a problémához [17]. Még a YOLO típusú detektorok esetén sem egységesek az alkalmazott adatbázisok és kiértékelési metrikák, általános esetben pedig két különböző architektúra ritkításának eredménye nem is hasonlítható össze érdemben. Egy tulajdonság azonban közös ezekben a módszerekben: minden esetben a feltétel meghatározása és a ritkított modell előállítása emberi beavatkozást igényel. Ez nem csak a módszerek lassúságát eredményezi, hanem jelentősen lekorlátozza a lehetőségek feltérképezését is. Erre nyúlt megoldást a következő fejezetben tárgyalt, megerősítéses tanulással automatizált pruning.

2.2 Megerősítéses tanulással megvalósított pruning

A megerősítéses tanulással megvalósított pruning témakörében az elmúlt 2-3 évben jelentek meg először publikációk, melyek túlnyomó részben osztályozó hálók

ritkításához lettek tervezve [31] [32] [33]. Az automatikus pruning megoldásakor a hagyományos ritkításon túlmenően további tervezési szempontokat kell figyelembe venni, melyek közül néhány legfontosabb: milyen megerősítéses tanulási algoritmus (ágens) választása a legelőnyösebb, hogyan és milyen sűrűn érdemes jutalmazni az ágenst, vagy, hogy mi alkossa az állapotteret. Az alábbi alfejezetekben azon a két state-of-the-art módszer kerül részletesen bemutatásra, melyek eredményes megoldásokat kínálnak az előbb felsorolt problémákra, és a leginkább hasonlítanak az általam készített automatikus pruning rendszerhez.

2.2.1 AMC

A 2018-ban a European Conference on Computer Vision konferencián megjelent AMC (AutoML for Model Compression) [32] nevet viselő eljárás volt az első megoldás, amely megerősítéses tanulással közelítette meg a pruning problémakörét. A módszer mind a szegmentálás, mind az objektumdetektálás területén több neurális háló modell ritkításánál is jobb paraméterredukció – pontosságromlás arányt ért el, mint amilyet az addigi kézi beállítással megvalósított redukálás során sikerült.

A redukálandó modellen strukturálatlan és strukturált ritkítást is végrehajtanak, utóbbi esetében a modellből egyes rétegeiből nem csak paramétereket, hanem teljes szűrőket távolítanak el. Az adott rétegből eltávolítandó paraméterek mennyiségét a réteg ritkaságával fejezik ki százalékban, a paraméterek kiválasztását pedig azok nagyságának vizsgálatával végzik (szűrők esetében az azt alkotó súlyok abszolút összegének vizsgálatával). A megerősítéses tanuláshoz a DDPG actor-critic algoritmus használják, melynek állapotterét 11 különböző paraméterrel írják le:

$$(t, n, c, h, w, stride, k, FLOPs[t], reduced, rest, a_{t-1}) \qquad \textbf{2-1}$$

ahol t a konvolúciós réteg indexe, a szűrő mérete *n* * *c* * *k* * *k*, a bemenet pedig *c* * *h* * *w* méretű. *FLOPs[t]* a *t*-edik réteghez tartozó lebegőpontos műveletek száma, *reduced* a modellben a *t*-edik rétegik terjedő, már pruning-olt rétegekhez tartozó FLOPs szám, *rest* pedig az utána következő rétegekhez tartozó. Végül a_{t-1} jelöli az előző lépésben választott akciót. Az akcióteret folytonos függvényként definiálják, az adott réteg/aktivációs tömb ritkasága százalékban kifejezve, folytonos függvényként. Az alkalmazott jutalomfüggvény:

$$R_{Param} = -Error * log(\#Param) \qquad \textbf{2-2}$$

ahol *Error* a pontosság romlását, *Param* pedig a törölt paraméterek számát jelöli. Az ágens csak a teljes modell ritkítása után, epizódonként egyszer kap jutalmat. A jutalom számításához a pontosságromlást az egyes iterációs lépésekben a ritkított háló validációs adatbázison való tesztelésével határozzák meg, finomhangolás nélkül.

A bemutatott megoldással ugyan jó eredményt értek el – Faster R-CNN detektor 50 %-os paramétercsökkentése mellett 0.1 % mAP javulás –, ám mivel az ágens mindig csak a teljes modell redukálása után kap visszajelzést, az algoritmus meglehetősen sok iteráció után konvergál.

2.2.2 PuRL

Ezt a hátrányt próbálták kiküszöbölni a 2020-ban az International Conference on Machine Learning konferencián bemutatott PuRL eljárással [31], melynek legfontosabb újítása az volt, hogy ún. sűrű jutalmat alkalmaztak, tehát az ágens nem csak a teljes modell redukálása után kapott visszajelzést, hanem minden egyes réteg ritkítása után. Ezzel jelentősen sikerült csökkenteni a konvergenciához szükséges iterációk számát. Az AMC-hez viszonyítva például 85 %-kal kevesebb iterációt használ.

Emellett a megoldás sok más tulajdonságában is különbözött az előző fejezetben bemutatott AMC-től. A PuRL nem strukturált pruning-ot használ, ami azt jelenti, hogy itt elsősorban a modell méretének csökkentése volt a szempont, mintsem a futási sebesség növelése. Az állapotteret 3 tulajdonsággal írják le:

$$s = (l, a, p) \tag{2-3}$$

ahol l a ritkítandó réteg indexe, a az addig redukált modell pontossága finomhangolás után, p pedig a törölt paraméterek százalékát jelöli. Az akcióteret hasonlóképp a rétegekhez választott ritkítási tényező (α) alkotja, mely tulajdonképp egy határérték, amit a réteg súlyainak szórásából határoznak meg, és az értéke alá eső nagyságú súlyok kerülnek eltávolításra:

$$Törölt\ Paraméterek_i(\alpha) = \{w \mid |w| < \alpha\sigma(w_i)\} \tag{2-4}$$

ahol $\sigma(w_i)$ az i. réteg súlyainak szórása. Az akciótér diszkrét, az α értékek a következő értékeket vehetik fel: $\alpha \in \{0.0, 0.1, 0.2, \dots, 2.2\}$.

A megerősítéses tanításhoz Deep Q-Learninget használnak. Továbbá, a sűrű jutalom bevezetése egy új, komplex jutalomfüggvény definiálását követelte meg:

$$R(s) = -\beta(max(1 - \frac{A(s)}{T_A}, 0) + max(1 - \frac{P(s)}{T_P}, 0)) \tag{2-5}$$

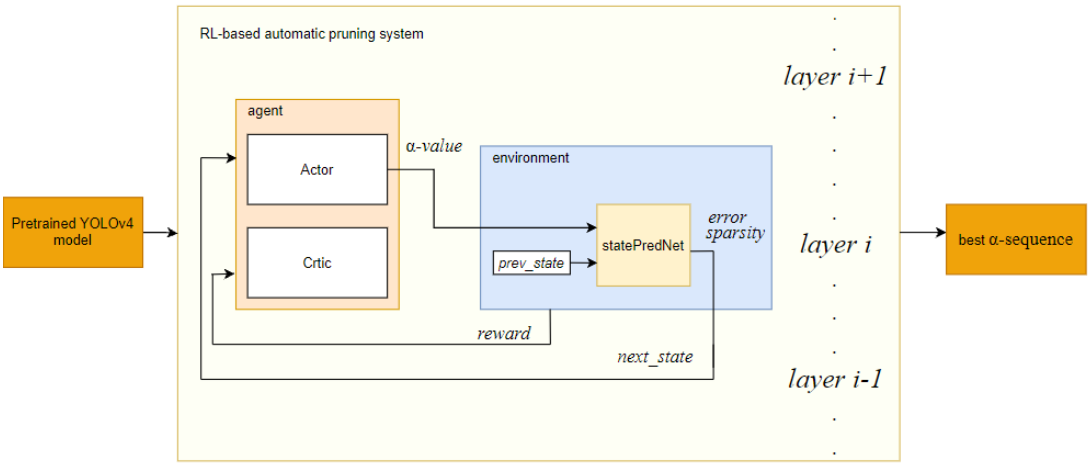
ahol *A(s)* and *P(s)* az *s* állapothoz tartozó újratanítás utáni pontosság és ritkaság arány, T_A and T_P az elérni kívánt pontosság és ritkaság arányt, β pedig egy skálafaktort jelöl.

Bár a PuRL jóval kevesebb iteráció alatt konvergál, mint az AMC, tényleges futási időt egyik publikációban sem tesznek közzé. Mindkét megközelítésben a háló pontosságának romlását validációval, sőt PuRL esetében egy kis adatbázison való finomhangolással határozzák meg futási időben. Ráadásul az utóbbi módszernél a sűrű jutalom miatt minden iterációban annyiszor végzik el ezeket a műveleteket, ahány réteg van a ritkítandó neurális hálóban. Az ismertetett módszerek ezért, bár kevés iterációs lépést, de rendkívül hosszadalmas tényleges tanítási időt igényelnek.

Az irodalomáttekintés után megállapítható, hogy a YOLO típusú detektorok ritkítására nagyon jó eredményt megvalósító hagyományos módszerek léteznek, melyek jelenleg jobb ritkasági arányt képesek elérni, mint a megerősítéses tanulással automatizált pruning módszerek. Viszont nem feledendő, az automatizált pruning előnye a hagyományossal szemben, miszerint az nem igényel emberi beavatkozást, alkalmazásának és fejlesztésének fontossága ezért nem elhanyagolható.

3 Probléma definiálása

Az előző fejezetben bemutatott megoldások tanulságait mérlegelve, a munkám célja egy olyan megerősítéses tanulás alapú ritkító rendszer tervezése és megvalósítása a YOLOv4 objektumdetektorhoz, mely a létező publikációkban ismertetettekhez hasonló jóságú ritkítást valósít meg a tanulás idejének jelentős mértékű csökkentése mellett. Meglátásom szerint az eddigi megoldások hosszas tanítási idejének legfőbb okozója az epizódonkénti – akár többszöri – validáció, ezért a rendszer ezen részét egy új módszerrel valósítom meg. Az egyes pruning lépések után a háló ritkaságát és pontosságát a tényleges validáció helyett egy ún. állapotbecslő hálóval határozom meg, melyet előzőleg automatikusan generált adatokon tanítok be. A rendszer többi részét, például a törölni kívánt paraméterek meghatározását, jutalomfüggvényt, a megerősítéses tanulási probléma állapot- és akcióterét vagy a tanulás típusát a már létező publikációkból merített ihletek alapján definiálom.



3.1. ábra. Az automatikus ritkítást megvalósító rendszer.

Az elkészült rendszer folyamatábráját az 3.1. ábra szemlélteti. Az ágens a betanított YOLOv4 háló kezdeti állapotát kapja meg bemenetként, majd a hálón rétegenként végighaladva minden réteghez választ egy ritkítási tényezőt (α). A háló ennek hatására új állapotba kerül, melyet a külső környezet az előtanított állapotbecslő háló segítségével határoz meg. A külső környezet ezután ezt az állapotot és az akció jóságát minősítő jutalmat visszajuttatja az ágenshez. A bemeneti háló összes rétegén végighaladva előáll egy α-szekvencia, mely felhasználásával ritkítva a bemeneti hálót egy kisebb, kevesebb paramétert tartalmazó háló áll elő.

Az ismertetett rendszer feladata tehát, hogy automatikusan, emberi beavatkozás nélkül találja meg azt az *n* hosszú α-szekvenciát, melynek felhasználásával a betanított YOLOv4 detektor rétegenkénti ritkítása a lehető legkevesebb paramétert tartalmazó hálót eredményezi a pontosságának lehető legkisebb romlása mellett. Az *n* a YOLOv4 detektor ritkítható rétegeinek számát jelenti, az α-szekvencia pedig – a megerősítéses tanulás értelmében – nem más, mint ágens által tanult stratégia.

4 Javasolt megoldások

Az alábbi fejezet a ritkító rendszer felépítéséhez szükséges elemek és folyamatok részletes ismertetését tartalmazza.

4.1 YOLOv4 detektor tanítása

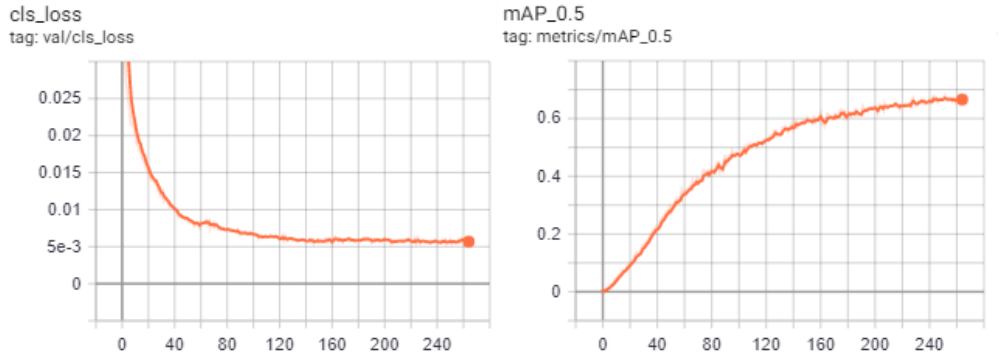
A rendszer megvalósításának első lépése a YOLOv4 detektor betanítása, melyhez tanító és validációs adatbázisra van szükség. Az említett detektor COCO adatbázison [34] betanított modellje publikusan elérhető és felhasználható ilyen célra. Tulajdonképp ez a modell is megfelelne a megvalósítani kívánt célra, ez azonban azt igényelné, hogy a ritkított modell validálása és finomhangolása is ezen adatbázis felhasználásával történjen. Mivel a COCO adatbázis rendkívül nagy, a gyorsabb tanítás céljából egy jóval kevesebb képet számláló adatbázisra, a KITTI-re esett a választás [35]. Az adatbázis közlekedésben alkalmazott detektorok tanítása céljából készült, 9 különböző kategóriát és kb. 7500 képet tartalmaz.

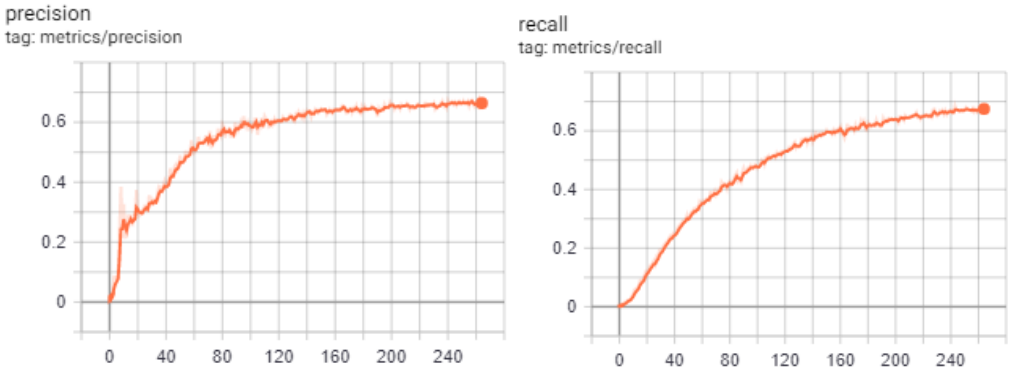
Az adatbázis képeit 70 % - 30 % arányban osztottam szét tanító és validációs adatokra. A YOLOv4 detektor megvalósításához és tanításához [36] implementációt használtam, a tanításhoz a batch méret kivételével az ebben definiált hiperparamétereket alkalmaztam, ezek közül a legfontosabbak a 4-1. táblázatban láthatók. A batch méretet az általam használt GPU memóriakapacitása miatt kellett csökkenteni.

epoch	**batch size**	**optimizer**	**learning rate scheduler**	**initial learning rate**	**weight decay**	**GPU**
260	16	Adam	LambdaLR [37]	1e-3	5e-4	NVIDIA TITAN X

4-1. táblázat. A tanításhoz használt fontosabb hiperparaméterek.

Az előtanított modell a KITTI adatbázison 68.5 %-os mAP értékre tanítottam be. A tanítás folyamatát a 4.1. ábra görbéi szemléltetik. Az így előállt modell képezi a ritkító rendszer bemenetét.





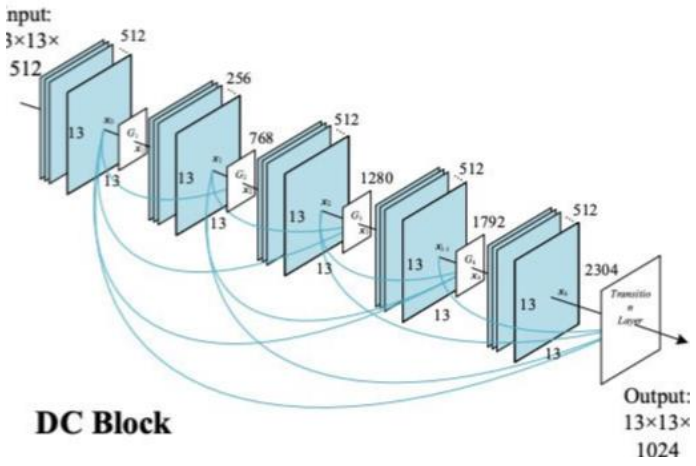
4.1. ábra. A YOLOv4 detektor tanításának validációs loss, mAP, precision és racall görbéi.

4.2 Pruning a YOLOv4 detektoron

Mivel a paraméterek csökkentése mellett a számítási kapacitás csökkentése is cél, a betanított YOLOv4 modell egyes konvolúciós rétegeiből teljes szűrőket távolítok el. A háló minden rétegéhez adott egy ún. ritkítási tényező (α), mely meghatározza, hogy adott rétegből hány szűrőt kell eltávolítani – erről a későbbiekben, az 4.4.2 fejezetben lesz szó részletesen, ám jelen fejezet megértéséhez ismerni szükséges, hogy α 23 különböző értéket vehet fel. Az alábbi fejezet a YOLOv4 architektúra ritkíthatóságát, felmerülő nehézségeket és a ritkítást végző algoritmust ismerteti.

4.2.1 Ritkítható rétegek a YOLOv4 architektúrában

A pruning végrehajtásakor az algoritmus az egymást követő konvolúciós rétegek bemeneti majd kimeneti csatornáiból felváltva távolít el szűrőket a választott α érték szerint, kezdve a 0. konvolúciós réteg kimeneti csatornáival. A feladatot azonban jelentősen megnehezíti a YOLOv4 detektor összetett architektúrája (4.2. ábra), ugyanis az számos előrecsatolást tartalmaz, ezáltal különös tekintettel kell lenni arra, hogy egy réteg változásakor a tőle függő rétegeket is annak függvényében kell módosítani.



4.2. ábra. Előrecsatolások a YOLOv4 architektúrában [16].

A YOLOv4 architektúrájában megtalálható rétegek típusai a következők:

• Convolutional: Egyszerű konvolúciós réteg.

• Maxpool: Egyszerű max pooling réteg.

• Upsample: Egyszerű upsample réteg.

• FeatureConcat: A megjelölt rétegek súlyait/csatornáit egymás után másolja. Az új réteg méretét a megjelölt rétegek méretének összege adja.

• WeightedFeatureFusion: A megjelölt rétegek súlyait/csatornáit összeadja. Az új réteg mérete a megjelölt rétegek közül a legkisebb méretű lesz.

• YoloLayer: Detekciós réteg. A bemenő aktivációs tömbből a detektor kimenetét, a detektált objektumok adatait állítja elő.

• ConvBeforeYolo: Egyszerű konvolúciós réteg, az eredeti architektúrában tulajdonképp nincs is megkülönböztetve, azonban a tervezés szempontjából fontos tulajdonsággal bír, így szükséges a megkülönböztetése.

Az modell ritkítását megvalósító algoritmus bemutatásához néhány segédparaméter ismertetése szükséges. A *dim* és *dim_next* paraméterek jelölik, hogy rendre az *i-edik* és *(i+1)-edik* rétegben a kimeneti vagy bemeneti csatornák kerülnek redukálásra: *dim = 0* esetén a kimeneti, *dim = 1* esetén a bemeneti csatornákból távolít el az algoritmus. Egyszerű konvolúciós rétegek esetén felváltva történik a ki- és bemeneti csatornák redukálása, így minden lépésben *dim_next = dim XOR 1*. Az *deleted_channels* paraméter a modellben található összes réteghez azon csatornák indexét tárolja, melyek törölve lettek. A *dims* tömb, *i-edik* réteghez a *dim_next* értéket tárolja. Ezek ismeretében az algoritmus működése a következő:

Réteg típusa	dim	Tennivaló
Convolutional	0	`deleted_channels += get_indexes(alpha)` `dim_next = dim XOR 1` *(új indexek)*
Convolutional	1	`deleted_channels += deleted_channels[-1]` `dim_next = dim XOR 1` *(indexek az előző réteg kimeneti csatornái szerint)*

FeatureConcat
Összevonandó rétegek:
layer_indicies
	0/1	`ii = []`
		`for j in layer_indicies`
		`  if dims[j] == 0`
		`  ii += indexes[j]`
		`deleted_channels += ii`
		`dim_next = 1`

WeightedFeatureFusion
Összeadandó rétegek:
layer_indicies
	0/1	`i = get the index of layer, where output channel number in MIN from layer_indicies`
		`deleted_channels += deleted_channels[i]`
		`dim_next = 1`

ConvBeforeYolo	1	`deleted_channels += deleted_channels[-1]` `dim_next = dontcare`

ConvBeforeYolo	0	`deleted_channels += []` *(nincs pruning)* `dim_next = dontcare`
YoloLayer	0/1	`deleted_channels += []` *(nincs pruning)* `dim_next = dims[-1]`

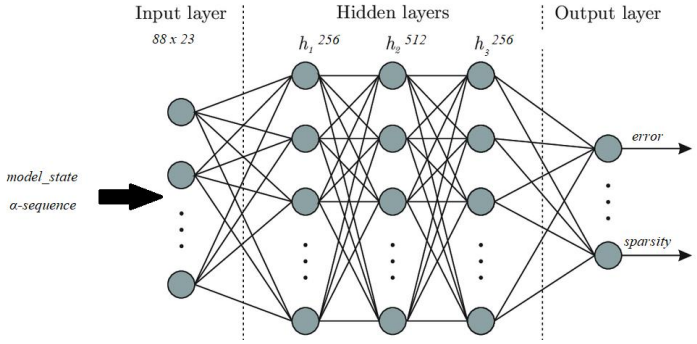
Maxpool, Upsamle	0/1	`deleted_channels += deleted_channels[-1]`
		`dim_next = 0`
		`dims[-1] = 0`

A fenti tervezési szempontok mellett az elkészült algoritmus a 160 rétegből álló architektúra 44 különböző konvolúciós rétegéhez választ α értéket – a továbbiakban az egyszerűség kedvéért ezeket nevezem „ritkítható rétegek”-nek. Mivel egy rétegből eltávolított kimeneti csatornák az azt követő rétegből is törlődnek, tulajdonképp ez azt jelenti, hogy a YOLOv4 architektúrában a ritkítható rétegek számának duplájából, azaz 88 rétegből törölhető paraméter.

4.3 Állapotbecslő neurális háló

4.3.1 Szerepe az ágens tanításában

Az állapotbecslő háló (*statePredNet)* a külső környezet feladatának egy részét veszi át: a modell állapotát (*model state)* és a modell rétegeihez rendelt α ritkítási tényezők sorozatát (*α-sequence*) bemenetként megkapva jósolja a háló pontosságromlását (*error*) és ritkaságát (*sparsity*) a tényleges pruning és validáció helyett (4.3. ábra4.3. ábra). Architektúráját tekintve 3 lineáris rétegből áll, rendre 256, 512 és 256 neuronnal, melyek között ReLU aktivációs függvény található.



4.3. ábra. Az állapotbecslő háló be- és kimeneteivel.

Az állapotbecslő háló automatikusan generált adatokon lett betanítva önfelügyelt tanítással. Mivel a tanító adatok nem fedik le az összes lehetséges esetet, ezért megtörténhet, hogy az ágens tanulása során előáll olyan α-szekvencia, melyre nem ad kellően pontos eredményt. Fontosnak tartom kihangsúlyozni, hogy az állapotbecslő hálónak nem is az a feladata, hogy tökéletes eredményt adjon minden bemenetére, csupán az, hogy a jellegzetes esetekre adjon helyes jóslatot, jó irányba terelve ezzel az ágens tanítását. Az ágens konvergenciája sokszor a nehezen állítható hiperparaméterektől is függ, egy-egy beállítás mellett legalább néhány 10 epizód szükséges ahhoz, hogy kiderüljön, konvergál-e a megoldás. Alkalmazásával az ágens nagyságrendekkel

gyorsabban közelít a jó stratégia felé, a jók közül a legjobbat viszont a tényleges ritkítás és validáció adta eredményből kell meghatározni.

4.3.2 Automatikus adatgenerálás

A 4.2.1 fejezetben bemutatott ritkító algoritmus felhasználásával egy előtanított modell redukálásakor tehát 44 különböző réteghez választható új α érték. Mivel α 23 különböző értéket vehet fel, ez 23 különböző érték választását teszi lehetővé rétegenként. Egy modell ritkítása során tehát 23^{44} számú lehetséges variáció létezik az α-szekvenciák összeállítására, ami egy 10^{59} -es nagyságrendű érték. Nyilvánvalóan ekkora számú minta generálása lehetetlen, azonban ez nem is szükséges, tekintve, hogy számos esetben alakul ki olyan állapot, amikor különböző α értékek választása nem eredményez jelentős pontosságváltozást. Például, ha a háló első, kis méretű rétegeit nagy α érték szerint redukáljuk, akkor már itt drasztikusan lecsökken a pontosság, és a későbbi rétegeknél már szinte mindegy milyen α-t választunk, a pontosság 0 lesz. Természetesen ezeket az eseteket is rögzíteni kell, de fontos észrevenni, hogy ezesetben kevés minta is elegendő lehet a háló megfelelő tanításához, míg például abban az esetben, amikor a modell jó pontosságot mutat még az utolsó rétegeknél is, melyek mérete nagy, így sok paraméter kisebb valószínűséggel okoz drasztikus pontosságcsökkenést, a különböző α értékek sokkal enyhébb változásokat fognak eredményezni, így ezesetben jóval nagyobb számú mintavétel szükséges.

Algorithm 1 Automatic data generation for training State Predictior Network

```
model ← pretrained YOLOv4 model
base_nParams ← calculate number of parameters in pretrained YOLOv4 model
base_mAP = test(model)
for layer_i in model do
    if layer_i can be pruned then
        α = random(0.0, 2.2)
        model, state_features = prune_network(model, layer_i, α)
        mAP = test(model)
        nParams ← calculate number of parameters in pruned model
        error = calc_error(base_mAP, mAP)
        sparsity = calc_sparsity(base_nParams, nParams)
        save(error, sparsity, state_features)
```

4.4. ábra. Automatikus adatgenerálás algoritmusa.

Az adatok generálása az 4.4. ábra. Automatikus adatgenerálás algoritmusa. látható algoritmus szerint történik: a betanított, ritkítandó modell betöltése után az algoritmus meghatározza a kezdeti mAP értéket és a modell paramétereinek számát, előbbit egy kisebb, 500 képből álló validációs adatbázison való teszteléssel valósítja meg. A háló rétegein végighaladva minden réteghez véletlenszerűen generál egy α értéket, majd ennek felhasználásával ritkítja a modellt. Ezek után az így kapott modell validálása következik, továbbá elmenti a ritkított modell azon tulajdonságait, melyek később az állapotbecslő háló bemenetét és az ágens állapotterét képezik. Az így generált adatok mellett néhány jellegzetes, kézzel szerkesztett eset is kerül a tanító adatbázisba, növelve így a lehetséges esetek lefedettségét.

4.4 Automatikus ritkítás megerősítéses tanulással

Jelen fejezetben az automatikus ritkítást megvalósító megerősítéses tanulást alkalmazó algoritmus (ágens) tervezési szempontjai kerülnek bemutatásra.

4.4.1 Állapottér

Az ágens bemenete minden *i*-edik réteg ritkításakor a ritkítandó háló állapota (s_i), mely minden ritkítható réteghez a következő 6 tulajdonságot hordozza:

$$s_i = (in\_channels, out\_channels, k, stride, pad, spars). \quad \textbf{4-1}$$

Az első 5 tulajdonság a konvolúciós rétegre vonatkozik: *in_channels* és *out_channels* a be- és kimeneti csatornák számát jelölik, *k* a szűrő méretét, *stride* a lépésméret, mellyel a szűrővel lépkedünk az aktivációs tömbökön, *pad* a rétegnél alkalmazott padding. A *spars* azt jelöli százalékosan, hogy adott rétegig a hálóból összesen mennyi paraméter lett eltávolítva.

Az állapot mérete, függetlenül attól, hogy éppen melyik réteg ritkítását végzi az ágens, mindig azonos: $nL * len(s_i)$, ahol *nL* a ritkítható rétegek száma, $len(s_i)$ pedig az állapotot leíró tulajdonságok száma. Azon rétegeknél, melyek még változatlanok, a felsorolt tulajdonságok 0 értékűek.

4.4.2 Akciótér

Az akciótér olyan α akciókból áll, melyek a ritkítás mértékét határozzák meg minden réteghez. Az α a rétegben lévő szűrők normájának szórásából származtatott határérték, mely a következőképp jelöli ki az eltávolítandó szűrőket:

$$Törölt\ szűrők_i(\alpha) = \{ch \mid |ch| < \alpha\sigma(ch_i)\} \quad \textbf{4-2}$$

ahol $\sigma(ch_i)$ az *i*. rétegben lévő szűrők normájának a szórása.

Az eltávolítás kritériumának ily módon történő meghatározását a PuRL publikáció [31] ihlette, ahol hasonlóképp járnak el, ám mivel ott egy rétegből nem strukturáltan törölnek súlyokat, a szűrők normája helyett a rétegben lévő súlyok szórását vizsgálják. Az akciótér hasonlóképp diszkrét, az α értékek a következő értékeket vehetik fel: $\alpha \in \{0.0, 0.1, 0.2, \dots, 2.2\}$.

4.4.3 Jutalomfüggvény

Rétegenkénti jutalom esetében egy jól működő jutalomfüggvény meghatározása nem triviális. Azt szeretnénk, ha a jutalom minden esetben megfelelően tanúskodna a megvalósított akció milyenségéről. Nagyvonalakban a következő az elvárás a jutalomfüggvénnyel szemben: ha az akció nagy ritkaságot és kis pontosságromlást eredményez, akkor nagyon jó jutalmat szeretnénk adni, kis ritkaság és nagy pontosságromlásnál nagyon rosszat, kis ritkaság és kis pontosságromlás esetén pedig valami közepesen rosszat. Ha csak a teljes modell ritkítása után kapna jutalmat az ágens, ez minden további nélkül teljesíthető lenne, azonban a rétegenkénti jutalom magával

vonzza azt a problémát, hogy a köztes rétegeknél előállhat olyan – adott rétegnél jónak számító – eredmény, mely az ismertetett elvárt működés szerint rossznak számít. Például, ha a modell elülső rétegeiből kevés paramétert törlünk, a pontosság nem romlik sokat, ellenben a ritkaság sem. Ez nem eredményez jó jutalmat, pedig ezek igazából jó döntések, a későbbi rétegeknél még nagy valószínűséggel elérhető a kívánt ritkaság. Az ágens ezért nagy valószínűséggel választ majd inkább olyan akciókat, melyek bár nagy pontosságromlást, de nagy ritkaságot is eredményeznek már az első rétegektől kezdve, ekkor azonban esély sincs arra, hogy az utolsó rétegekhez érve visszaálljon a pontosság.

A legtöbb publikációban, ahol csak a teljes modell ritkítása után kap jutalmat a háló, a jutalomfüggvényt a pontosságromlás és a törölt paraméterek valamilyen egyszerű kapcsolataként határozzák meg, például:

$$R_{Param} = -Error * log(\#Param) \quad \textbf{4-3}$$

Rétegenkénti, azaz sűrű jutalom esetében azonban ez nem megfelelő az előző bekezdésben ismertetett okok miatt. A PuRL publikációban [31] egy igen összetett jutalomfüggvényt definiálnak, mely kiválóan kompenzálja az összes felsorolt nehézséget.

$$R(s) = -\beta(max(1 - \frac{A(s)}{T_A}, 0) + max(1 - \frac{P(s)}{T_P}, 0)) \quad \textbf{4-4}$$

ahol *A(s)* és *P(s)* az *s* állapothoz tartozó újratanítás utáni teszt pontosság és ritkaság arány, T_A and T_P az elérni kívánt pontosság és ritkaság arányt, β pedig egy skálafaktort jelöl. A jutalomfüggvény ezzel azt is lehetővé teszi, hogy az ágens az elérni kívánt ritkaságra és pontosságromlásra optimalizálódjon.

Az automatikus ritkító rendszerrel olyan modell előállítása a cél, melynek kis pontosságromlása fontosabb szempont, mint a sok paraméter törlése. Ennek elérése céljából az ismertetett jutalomfüggvényt súlyozó tényezőkkel egészítem ki, melyekkel szabályozható, hogy melyik tulajdonság tévesztése járjon súlyosabb büntetéssel. Mindemellett, a publikációval ellentétben az általam megvalósított rendszerben nem az elérni kívánt pontosság, hanem a pontosságromlás (*error)* definiált. Ezek következtében az alkalmazott jutalomfüggvény a következő:

$$R_s(s) = -\beta(coeff_{err} * max(\frac{E(s) - T_E}{1 - T_E}, 0) + coeff_{spars} * max(1 - \frac{P(s)}{T_P}, 0)) \quad \textbf{4-5}$$

ahol *E(s)* és *P(s)* az *s* állapothoz tartozó pontosságromlás és ritkaság, T_E és T_P az elérni kívánt pontosság és ritkaság, $coeff_{err}$ és $coeff_{spars}$ az *error*-hoz és *sparsity*-hez tartozó súlyzó tényező, β pedig egy konstans skálafaktor. A ritkított a modell bekezdés elején ismertetett elvárt teljesítményének eléréséhez a jutalomfüggvényben a 4-2. táblázat. A jutalomfüggvény paramétereinek megválasztása. szerint választom meg a paramétereket.

T_E	T_P	$coeff_{err}$	$coeff_{spars}$	β
20%	60%	1.1	1.0	5

4-2. táblázat. A jutalomfüggvény paramétereinek megválasztása.

4.4.4 Algoritmus tervezése

Az előtanított állapotbecslő háló alkalmazásával az ágens tanításához csupán a ritkítandó háló állapotának betöltése szükséges, a teljes, betanított modell nem. Ez lehetővé teszi a nagyobb batch méret alkalmazását, azaz az 1.4.2 fejezetben ismertetett multiágensű tanítást. A feladat – hogy 10^37 számú variáció közül találjuk meg a lehető legjobb α-szekvenciát – gépi tanulás értelemben is nehéznek számít, így az actor és critic hálókat egy közepesen komplex architektúrából állítom össze. Az előbbi 3 lineáris rejtett réteget tartalmaz, rendre 512, 1024 és 256 neuronnal, míg utóbbi 2 rejtett rétegből áll, 256 és 512 neuronnal. Mindkét háló esetében a rejtett rétegek között ReLU aktivációs függvényt alkalmazok.

Algorithm 2 Advanced Actor-Critic algorithm for finding the best α sequences

```
bS ← batch_size as the number of agents
aS ← size of the action space
nL ← number of layers that can be pruned
α_sequence[bS, nL] ← []
episode ← 0

while episode ≤ max_episodes do
    model_arch ← load model architecture
    state ← load initial model state
    for each layer_i in model_arch do
        if layer_i can be pruned then
            distribution[bS, aS] = actorNet(state)
            q_value[bS, 1] = criticNet(state)
            action[bS] = sample from distribution

            error[bS], sparsity[bS] = statePredNet(action, state)
            state ← environment(error, sparsity)

            reward[bS] = reward_function(error, sparsity)
            α_sequence.append(action)
        end if
    end for

▷ check if the predicted action and sparsity are close enough to real values
    if validation episode then
        model ← load trained model
        pruned_model ← prune model by chosen α_sequence
        real_error, real_sparsity ← validate pruned_model on validartion
        dataset
        if real_error, real_sparsity are close enough to error, sparsity then
            continue training
        end if
    end if
end while

return best α_sequence[bS, nL]
```

4.5. ábra. Legjobb α-szekvenciát kereső megerősítéses tanulási algoritmus.

A 4.5. ábra ismerteti az így előállt automatikus ritkító algoritmus pszeudokódját. Az algoritmus betölti a ritkítandó modell kezdeti állapotát, majd másolatot készít arról minden ágens számára. Ezután a modell rétegein végighaladva, ha adott réteg ritkítható, az ágensek meghatározzák a hozzá tartozó α értéket, a critic háló jósolta akció-

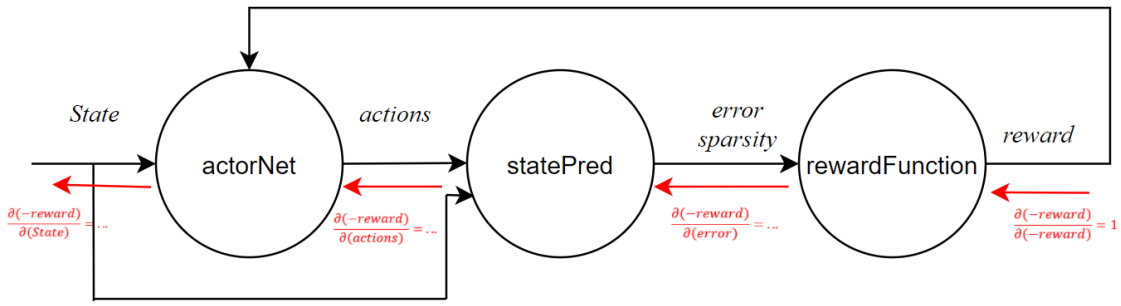
valószínűségieloszlás mintavételezésével. Az előtanított állapotbecslő háló az α értéket és a modell állapotát bemenetként megkapva meghatározza az α szerinti ritkítás utáni pontosságromlást és ritkaságot (*error, sparsity)*. A külső környezet ennek hatására új állapotba kerül, valamint jutalmat rendel az ágensekhez. Mindeközben a critic háló az optimális értékfüggvényt tanulja. Ahogy az 4.3.1 fejezetben említésre került, az állapotbecslő háló jóslatának helyessége nem garantálható minden esetben, ezért bizonyos epizódonként néhány α-szekvencia szerint ténylegesen megtörténik a betanított modell ritkítása, majd a validációs adatbázison való tesztelés. Amennyiben az így kapott *error* és *sparsity* értékek nem térnek el nagyban az állapotbecslő háló jósolta étékektől, a tanítás folytatódik.

A tanítás befejeztével az algoritmus *bS* darab méretű *nL* hosszú α-szekvenciát ad, ahol *bS* a batch méretet, azaz az ágensek száma, *nL* pedig a ritkítható rétegek száma. Az ágensek tanítása a batch-re átlagolt hiba szerint történik, de ne feledjük, hogy azok egymástól különböző stratégiát követnek. Ezért az eredményül kapott α-szekvenciák közül a kezdeti modell ritkításával és validálásával kerül kiválasztásra a legjobb, annak felhasználásával készül a végleges, ritkított modell, hasonlóan a pszeudokódban ismertetett tesztelés folyamatához.

4.4.5 Modell alapú tanítás

A megerősítéses tanulási problémák esetén fontos még tisztázni a modell alapú megerősítéses tanulás terminológiát, mely az ágens tanulási módjára vonatkozik. Pontos definiálása igencsak nehézkes, több szemlélet is létezik arra, hogy hogyan különböztethető meg a modell nélküli és modell alapú megerősítéses tanulás, mi teszi modell alapúvá az eljárást.

Egyik, könnyen definiálható alapvető különbség a két eset között az ágens és a környezet viszonyából származtatható: míg modell nélküli tanítás esetén az ágens közvetlenül a környezettel áll interakcióban, addig a modell alapú tanulásnál egy környezet szimulálására alkalmas modellel van kapcsolatban, melyet a környezeten tanítunk, így az ágens és a környezet kapcsolata közvetett. Egy másik, gyakran alkalmazott szemlélet szerint a modell alapúság meghatározható abból, hogy az ágens mi alapján választ akciókat az egyes lépésekben [38]. Az ágens célja, hogy olyan akciókat válasszon, melyek a lehető legnagyobb jutalmat eredményezik. Az egyszerűbb, modell nélküli esetben az ágens nem áll közvetlen kapcsolatban a jutalommal, csupán az adott szituációkban választott akciók valószínűségének növelésére törekszik. Az, hogy ez a jutalom növekedéséhez is vezet csak idővel lesz igaz, és a hatalmas számú mintavételezésnek köszönhető. A jutalom ezért a tanítás kezdetén rendkívül zajos lesz, és csak sok epizód – sok tapasztalat után konvergál. Modell alapú esetben ezzel szemben az ágens a jutalom függvényében képes tervezni, sok tapasztalat nélkül is olyan akciót választani, mely jó jutalom szerzéséhez vezet.



4.6. ábra. Számítási gráf az actor háló és a *reward* között.

Az általam tervezett megerősítéses tanulási rendszerben a modell alapú tulajdonság az állapotbecslő háló beiktatásával jelenik meg, hiszen annak szerepe a környezet szimulálása. Alkalmazásával ráadásul lehetőség nyílik a környezet modell paraméterek szerinti deriválására. Ezáltal az állapotbecslő háló összeköttetést biztosít az actor háló és a *reward* számítási gráfja között, hiszen a bemenetét – többek közt – az actor által választott akciók képezik, kimenete pedig az *error* és a *sparsity* melyek a *reward* számításához szükségesek. Ennek következtében a kapott *reward* deriválásával tehát közvetlenül meghatározható az a gradiens, mely azt az információt hordozza, hogy hogyan kell az actor háló súlyainak változni annak érdekében, hogy az általa választott akció minél jobb jutalmat eredményezzen (4.6. ábra). Ez pedig kevésbé zajos gradienshez, ezáltal stabilabb és nem mellesleg gyorsabb tanuláshoz vezet.

5 Eredmények

Az alábbi fejezetben az elvégzett tesztesetek eredményének, valamint a kiértékelési szempontok bemutatása következik. Továbbá a fejlesztés során felmerült nehézségek, megoldások és tanulságok is ismertetésre kerülnek.

5.1 Mérési szempontok és metrikák

Egy mély neurális háló ritkításának célja – függetlenül attól, hogy kézi módszerrel vagy automatizálva történik –, hogy egy kisebb modell álljon elő, kevesebb paraméterrel, mely gyorsabb futási időt igényel, mindemellett a pontossága a lehető legkevesebbet romoljon. Automatizált ritkítás esetén továbbra is ez a végső cél, viszont további szempont, hogy minél kevesebb időt vegyen igénybe az optimális megoldás megtalálása. Az eredmények kiértékeléséhez ezért ezen szempontok figyelembevételével kell meghatározni a metrikákat.

Mivel munkám során egy detektor háló ritkítását végzem, mindenekelőtt a detektorok teljesítményének méréséhez szükséges legfontosabb mértékek ismertetése szükséges:

- Precision [%]: Azt jelöli, hogy a háló által detektált összes objektum hány százalékát detektálta helyesen.
- mAP [%]: A validáció során adott osztályokra számított átlagos precizitás értéke az összes osztályra átlagolva. Legfontosabb mutatója a detektor jóságának.

Ezek ismeretében definiálhatóak a metrikák, melyeket az elkészült automatikus ritkítást megvalósító algoritmus kiértékeléséhez használtam:

- *mAP* [%]: Külön számítandó a kiindulási és a ritkított modellre.
- Pontosságromlás (*error*) [%]: Azt adja meg százalékosan, hogy a ritkított modell mAP értéke mennyit romlott a kiindulási modell mAP értékéhez képest. Ennek értelmében az *error* lehet 100 %-nál nagyobb értékű is.

$$error = \frac{mAP_{before}}{mAP_{after}} \qquad \textbf{5-1}$$

- Paraméterszám [millió db]: A modellben lévő paraméterek (súlyok) száma.
- Ritkaság (*sparsity)* [%]: Azt mutatja, hogy az eredeti modell paramétereinek hány százaléka lett eltávolítva.
- Lebegőpontos műveletek száma (Floating Point Operations – FLOPs) [db]: A neurális háló előterjesztése során elvégzett lebegőpontos számítási műveletek száma, mely a következőképp számítható:

$$n_{FLOPs} = \sum_{i=0}^{N} \frac{2 * w_i * h_i * k_{iw} * k_{ih} * ch_{IN} * ch_{OUT}}{stride^2} \qquad \textbf{5-2}$$

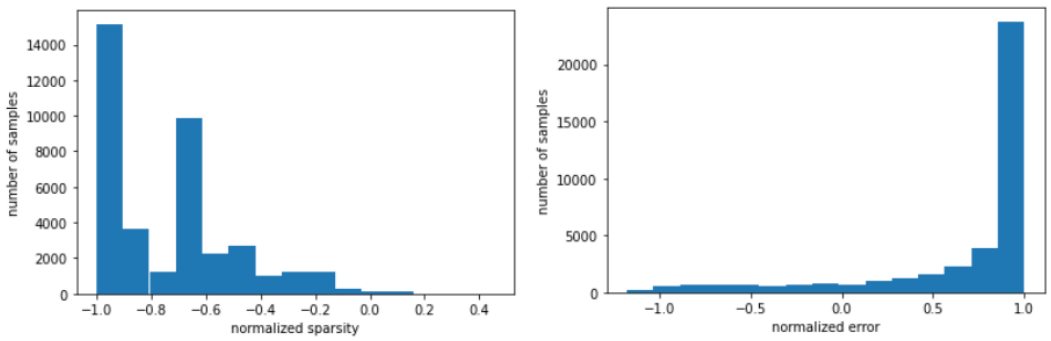
ahol wi és hi az i. réteg bemeneti aktivációs tömbjeinek szélessége és magassága, kiw és kih az adott réteghez tartozó szűrő szélessége és magassága, $chIN$ és $chOUT$ a réteg be- és kimeneti csatornáinak száma, $stride$ pedig a lépésszám, amellyel a szűrő végighalad az aktivációs tömbökön.

- Futási idő [FPS]: A háló előterjesztéséhez szükséges idő NVIDIA TITAN X GPU-n mérve, 1-es batch méret mellett. Az erre alkalmazott szabványos mértékegység a másodpercenkénti képkockák száma (Frame per Second – FPS), mely az imént ismertetett futási idő reciproka.

5.2 Kompromisszum az állapottér méretének szempontjából

A ritkítandó modell állapotát reprezentáló tömb mérete [*nFeatures* x *nL*], ahol *nL* a ritkítható rétegek számát jelöli a YOLOv4 architektúrában, *nFeatures* pedig a megfigyelt tulajdonságok száma. Ebben a formában azt, hogy adott lépésig hány réteg lett ritkítva a hálóban, a tömb feltöltöttsége mutatja. Kezdeti állapotban minden érték 0, az első réteg ritkítása után az első oszlop elemei a réteghez tartozó megfigyelt értékre cserélődnek, a többi réteget reprezentáló oszlop értéke továbbra is 0. Minden lépésben egy újabb oszlopba kerülnek új értékek, tehát a feltöltöttség akkor lesz teljes, ha az utolsó rétegen is megtörtént a pruning.

A modell állapota a rendszer két fontos részében kerül felhasználásra: az állapotbecslő háló és az ágens actor és critic hálójának bemenetét képezi. Az *nFeatures* mérete ezekben a különböző alkalmazásokban nem egyforma viselkedést eredményez, így az állapotteret nem ugyanolyan formában alkalmazom ezekben a részekben. A következő bekezdések a különböző viselkedés okát vizsgálják.



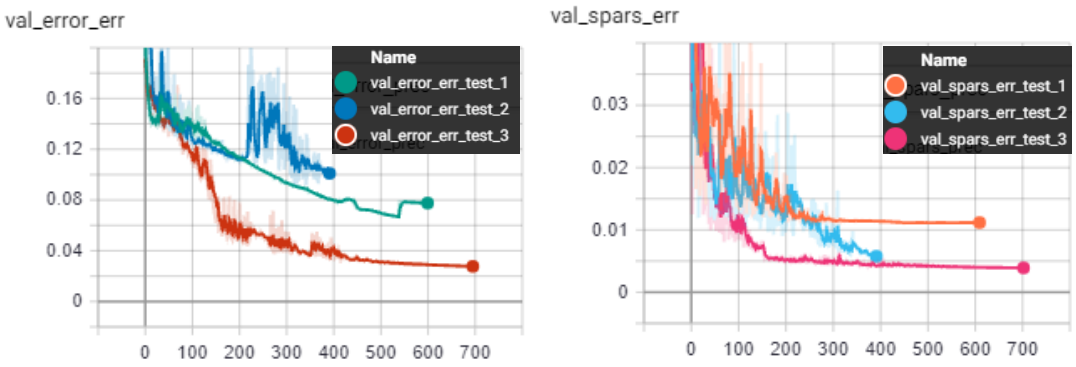
5.1. ábra. Generált adatokhoz tartozó címkék - *sparsity, error* - eloszlása. Értékük -1 és 1 közé normálva.

Az állapotbecslő háló tanításához generált tanító adatokhoz tartozó címkék – az *error* és *sparsity* eloszlása meglehetősen egyenetlen (5.1. ábra). A *sparsity* eloszlásában a kis értékek dominálnak. Ez azzal magyarázható, hogy az első néhány réteg mérete kicsi, így azokból még ha nagy számban is törlünk paramétert, az nem eredményez nagy ritkaságot a teljes háló méretéhez képest. Az *error* eloszlásában a nagy értékek vannak túlnyomó többségben, melynek oka, hogy egyetlen rossz ritkítási tényező választása is teljesen elronthatja a modell pontosságát, onnantól kezdve az összes többi rétegnél vett minta is teljes pontosságromlást mutat. Annak ellenére, hogy ezek az esetek ilyen

egyenetlen *error* eloszlást eredményeznek, szükség van a hozzájuk tartozó tanító adatokra, hiszen az állapotbecslő hálónak a rossz viselkedést eredményező eseteket is jól kell felismernie.

Az állapotbecslő háló tanítását már kezdetben megnehezítette a címkék egyenetlen eloszlása, ezért arra törekedtem, hogy a tanító adatok a lehető legkevesebb ilyen értéket tartalmazzák az adatösszetétel további rontásának elkerülése érdekében. Az 5.2. ábra tesztesetei mutatják az állapotbecslő háló tanításakor az *error* és *sparsity* hibájának alakulását három különböző bemenet mellett. Az első *tetst_1* esetben az α-szekvencia mellett az az ágens állapotterét alkotó tulajdonságok képezik a bemenetet: *(in_channels, out_channels, k, stride, pad, spars)*, az állapottömb feltöltése pedig az első bekezdésben ismertetett módon történik.

A *test_2* esetben szintén az említett 5 tulajdonság képezi a kiegészítő bemenetet, de a tömb feltöltése eltérő: a rétegspecifikus tulajdonságok, tehát a ki- és bemeneti csatornák száma, a szűrő mérete és a pad paraméter már a kiindulási állapotban betöltésre kerülnek a háló összes rétegéhez, nagyban csökkentve így a bemenetben lévő 0 értékek számát.



5.2. ábra. Az állapotbecslő háló tanításának szemléltetése az *error* és a *sparsity* vallidációs hibájának görbéjével. A hiba a tényleges eltérésre vonatkozik, tehát például *test_3* esetben az *error*-ra jósolt értékek átlagosan 2%-ban térnek el a valódi értéktől.

epoch	batch size	optimizer	loss function	learning rate scheduler	initial lr	final lr	weight decay	GPU
2400	2048	Adam	LogCosh	CosineAnnealingLR	1e-3	1e-5	1e-5	NVIDIA TITANX

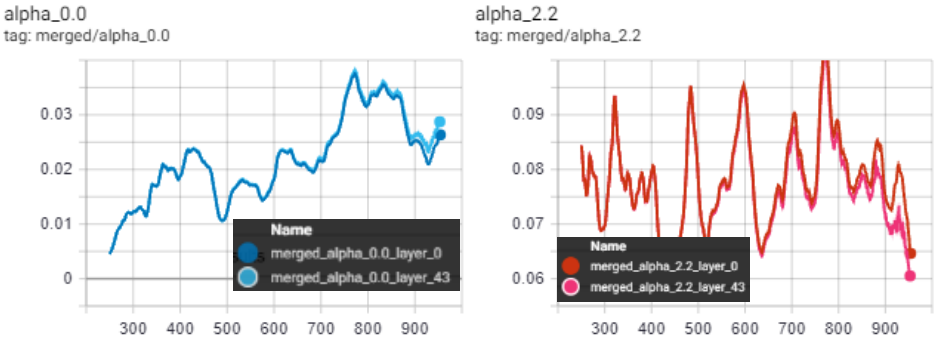
5-1. táblázat. Az állapotbecslő háló tanításához alkalmazott hiperparaméterek.

A *test_3* esetben az α-szekvencia mellett csupán a *spars*, tehát az adott rétegig elért ritkaság az állapotbecslő háló kiegészítő bemenete. Jól látható, hogy a három eset közül a test_3 adja a legjobb eredményt, tehát az állapotbecslő szempontjából előnyösebb, ha az állapottér kevés tulajdonságot foglal magába. A grafikonról továbbá az is leolvasható, hogy a *sparsity* tanulása nem okoz akkora nehézséget a hálónak, mely a címkék valamivel egyenletesebb eloszlásának tudható be. A *test*_3 tesztesetben a tanítás végleges eredménye, hogy az állapotbecslő háló 2 %-on belüli pontossággal képes

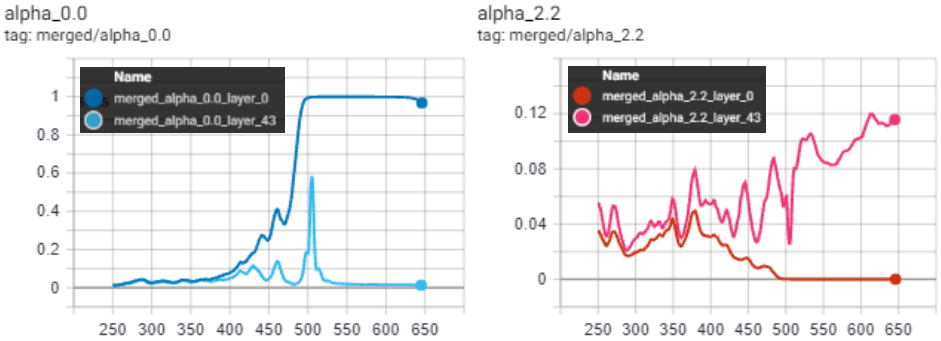
predikálni a pontosságromlást, és 1 % alatti eltéréssel a ritkaságot. Ezen eredmény eléréséhez a tanításhoz használt hiperparamétereket az 5-1. táblázat foglalja össze.

Az ágens az állapotteret illetően ezzel szemben másképp viselkedik. Ha bemenetként csupán a *spars* szekvenciát kapja meg, nem tudja kellőképp megkülönböztetni, hogy a háló architektúrában éppen melyik réteg ritkítása történik. Az 5.3. ábra vizsgálatával látható, hogy az akciók valószínűségi eloszlása egyáltalán nem függ attól, hogy éppen melyik réteghez jósolja azt az actor háló. Az α=0.0 akció választásának valószínűsége szinte ugyanakkora függetlenül attól, hogy az első vagy az utolsó réteg ritkítása történik, és ugyanez a jelenség figyelhető meg az α =2.2 akció valószínűségének megfigyelésekor is. Emiatt természetesen a stratégia sem javul, amit a *reward* stagnálása mutat (5.5. ábra).

A hatdimenziós *(in_channels, out_channels, k, stride, pad, spars)* állapot alkalmazása esetén azonban teljesen más eredmény tapasztalható. A tanítás előrehaladtával az mind az α=0.0, mind az α=2.2 akció választásának valószínűsége jól láthatóan egyre inkább eltér attól függően, hogy éppen az első vagy az utolsó réteg ritkítása történik (5.4. ábra). Ennek következtében megállapítható, hogy a látszólag új információt nem hordozó kiegészítő tulajdonságok az ágens megfelelő tanulásának szempontjából nagyon is fontosak. Az ágens tanításához ezért a nagyobb állapottér választása az ideális.



5.3. ábra. Az α=0.0 és α=2.2 akciók választásának valószínűsége az első és az utolsó réteg ritkításakor, egydimenziós állapottér alkalmazása mellett.

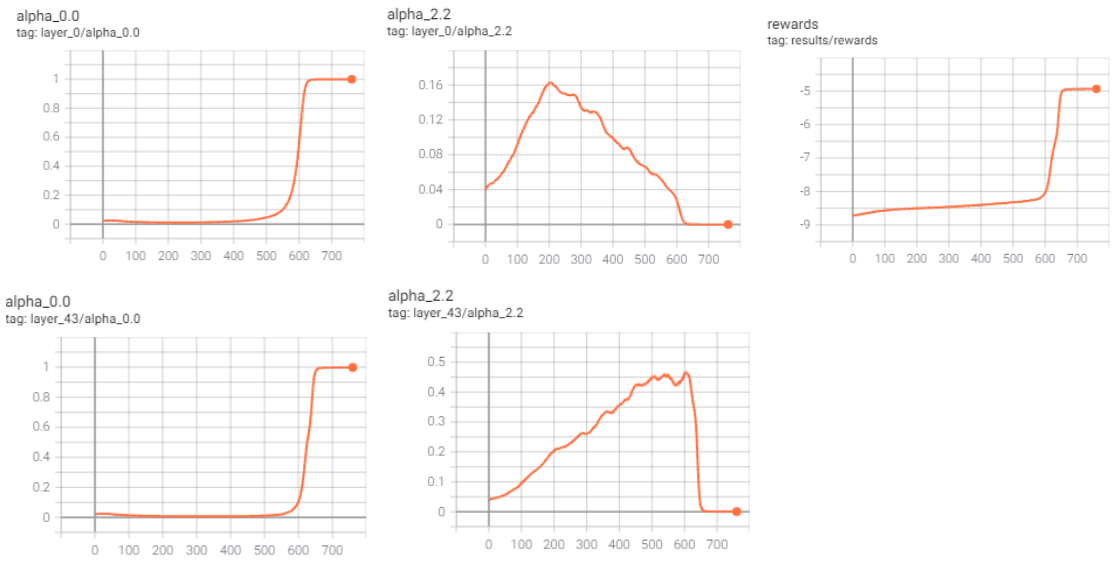


5.4. ábra. Az α=0.0 és α=2.2 akciók választásának valószínűsége az első és az utolsó réteg ritkításakor, hatdimenziós állapottér alkalmazása mellett.

5.5. ábra. Az utolsó réteg ritkítása után kapott jutalom batch-re átlagolva, egy- és hatdimenziós állapottér esetén.

5.3 Túl magabiztos stratégia

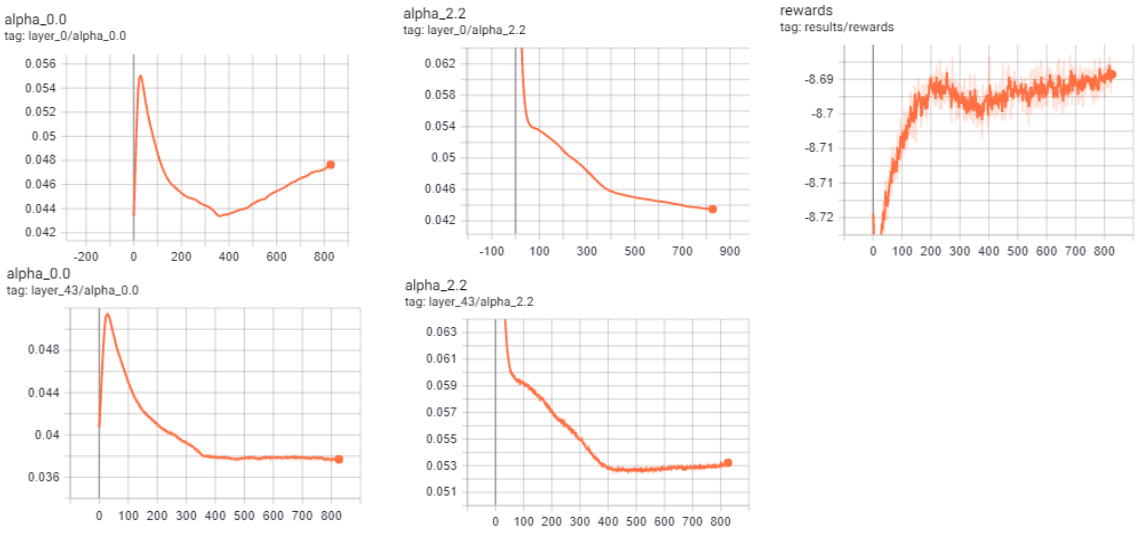
Az ágens tanításakor A2C algoritmus alkalmazása esetén hamar megmutatkozik a túl magabiztos stratégia problémája. A jelenséget legjobban az első és az utolsó rétegnél az α=0.0 és α=2.2 akciók választásának valószínűségi eloszlása szemlélteti. A jó stratégia nagyvonalakban azt jelenti, hogy az első réteghez kisebb α-t válasszon az ágens – tehát az α=0.0 valószínűsége nőjön az epizódok előrehaladtával, az α=2.2 valószínűsége pedig csökkenjen. Ezzel szemben az utolsó réteghez inkább nagyobb α-k választása jellemzi a jó stratégiát, így ott pont az ellenkező viselkedés az elvárt az akciók valószínűségének változását illetően.



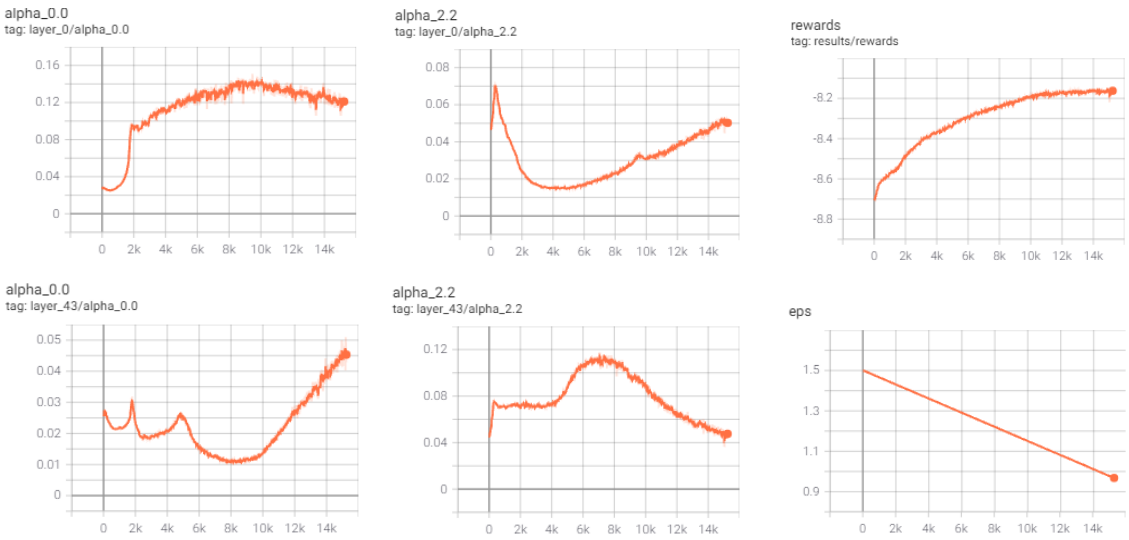
5.6. ábra. A2C algoritmus esetén az α=0.0 akció választásának valószínűsége az első és az utolsó réteghez (jobb felső és jobb alsó grafikon), az α=2.2 akció választásának valószínűsége az első és az utolsó réteghez (középső felső és középső alsó grafikon) és az utolsó réteg ritkítása után kapott jutalom batch-re átlagolva.

A tanítás kezdetén, nagyjából az 500. epizódig pontosan az elvárt viselkedés tapasztalható (1.1. ábra5.6. ábra). Ekkor azonban az α=0.0 akció választásának valószínűsége túlságosan megnő, néhány epizód után már minden réteghez ezt az α=0.0 értéket választja az ágens – ezt a jutalom értéke mutatja. Tulajdonképp jó, hogy az ágens megtanulja, hogy ha minden réteghez α=0.0 akciót választ, az jobb stratégiát eredményez, mint a random választás, viszont a problémát az jelenti, hogy ez a stratégia túl magabiztossá válik még azelőtt, hogy a további – várhatóan jobb – lehetőségeket is feltérképezné.

Pont ezt a viselkedést igyekszik megakadályozni a PPO algoritmus. Fontos észrevenni azonban, hogy míg a stratégia magabiztosságát csökkenti, addig az actor költségfüggvényének csonkolása miatt túl kicsi lépéseket tesz az optimum felé, ezzel lassítja a konvergenciát. A 5.7. ábra grafikonjai szemléltetik, hogy jó irányba fejlődik a stratégia, viszont rendkívül lassan, még akkor is, ha a vágási tényezőt kezdetben nagyobbra választom, és a tanítás előrehaladtával csökkentem azt (5.8. ábra). Több mint tízezer epizód alatt még a kritikus pontot sem közelíti meg a stratégia, nemhogy annál jobb irányba fejlődjön. Ennek következtében megállapítható, hogy a PPO alkalmazása jelen esetben több hátrányt von maga után, mint amennyi előnnyel rendelkezik.



5.7. ábra. PPO algoritmus, ϵ=0.2 esetén az α=0.0 akció választásának valószínűsége az első és az utolsó réteghez (jobb felső és jobb alsó grafikon), az α=2.2 akció választásának valószínűsége az első és az utolsó réteghez (középső felső és középső alsó grafion) és az utolsó réteg ritkítása után kapott jutalom batch-re átlagolva.



5.8. ábra. PPO algoritmus, ϵ csökkentése esetén az α=0.0 akció választásának valószínűsége az első és az utolsó réteghez (jobb felső és jobb alsó grafikon), az α=2.2 akció választásának valószínűsége az első és az utolsó réteghez (középső felső és középső alsó grafion) és az utolsó réteg ritkítása után kapott jutalom batch-re átlagolva.

5.4 Körültekintő hiperparaméter-választás eredménye

A kritikus ponton való túllendülést és a megfelelő konvergenciát az A2C algoritmus eredményezte, körültekintő hiperparaméterezéssel és a tanulási ráta megfelelő formázása mellett. A különböző paraméterezés mellett futtatott tesztesetek megfigyeléséből láthatóvá vált, hogy a tanítás elején előnyös a nagy tanulási ráta alkalmazása, ennek következtében az ágens gyorsabban megtanulja, hogy az kezdeti rétegekhez kis α érték választása az előnyös. Még mielőtt ez a stratégia túl magabiztossá válna (5.9. ábra, 250. epizód), több nagyságrenddel csökkentem a tanulási rátát, továbbá az actor költségfüggvényét nagyobb súlyban büntetem entrópiával. Az 5.9. ábra szemlélteti, hogy ennek következményeképp a *reward* görbe átlendül a kritikus -5-ös értéken, és egy nagyobb értékhez konvergál. Az actor és critic hálók hiperparaméterbeállításait az 5-2. táblázat tartalmazza.

	epoch	**batch size**	**optimizer**	**learning rate scheduler**	**initial lr**	**final lr**	**entropy coeff**	**GPU**
actor	**700**	4096	Adam	-	1e-3	5e-4 (ep.250)	5e-3 1e-2 (ep. 250)	NVIDIA TITANX
critic			Adam	-	1e-2	1e-2	-	

5-2. táblázat. Actor és critic hálók tanításához használt hiperparaméterek.

5.9. ábra. Az utolsó réteg ritkítása után kapott jutalom batch-re átlagolva. A 250. epizódtól kezdve (bal ábra) kisebb tanulási rátával lett folytatva az ágens tanítása (bal ábra).

Az 5-3. táblázat tartalmazza az ágens tanításának fontosabb fázisait. A 200. epizódnál az utolsó réteg ritkítása után kapott *reward* a teljes batch-re átlagolva még igencsak rossz. A batch-re átlagolt, állapotbecslő által jósolt *reward* és *sparsity* értékeket szemlélve az látható, hogy az ágensek olyan stratégiát követnek, mely bár nagy ritkaságot, 100 %-os pontosságromlást eredményeznek. Ezt a tényleges pruning és validálás során kapott *real_error* és *real_sparsity* értékek is megerősítik. A 450. epizódnál már valószínűbb az olyan akciók választása, melyek ugyan kisebb ritkaságot, de egyben kisebb pontosságromlást is eredményez. Innentől kezdve a ritkaság szempontjából javul a stratégia, nagyjából azonos pontosságromlás megtartása mellett.

epizód	átlagos reward	átlagos jósolt error [%]	hozzá tartozó jósolt sparsity [%]	legjobb valódi error [%]	hozzá tartozó valódi sparsity [%]	hozzá tartozó reward
200	-8.551	100	65	99	70	-4.96
450	-4.8	-70	17	0	20	-3.35
700	-4.32	-0.8	30	-10.5	49	-2.527

5-3. táblázat. Az ágens tanításának fontosabb fázisai.

A 700. epizódnál az állapotbecslő háló értékei azt mutatják, hogy az átlagosan 0.8 %-os pontosságjavulást és 30 %-os ritkaságot eredményeznek a stratégiák. A tényleges pruning és validálás során azonban előkerülnek olyan eredmények is a batch-ből, ahol ezek az értékek még ennél is jobbak: a legkimagaslóbb eredmény a 10.5 %-os pontosságjavulás a paraméterek 49 %-ának törlése mellett. Ezt az eredményt az 5.10. ábra kék színnel ábrázol α-szekvencia eredményezte. A kiindulási modell pontosságának javulása azzal magyarázható, hogy a pruning tulajdonképp egy regularizációs eljárás, hiszen a súlyok törlésével csökken a modell magabiztossága, ezáltal az overfitting mértéke is.

5.5 Eredmények értékelése

Jelen fejezetben az elért eredményeket két fő szempont szerint értékelem: először a saját automatikus pruning rendszerrel ritkított YOLOv4 modell pontosság és ritkaság paraméterei kerülnek összevetésre kézi beállítások mellett ritkított (handcrafted) modellekkel, valamit a state-of-the-art ritkító algoritmusok eredményeivel. Ezek után a ritkító algoritmust a megerősítéses tanulás időigényének szempontjából hasonlítom össze state-of-the-art megerősítéses tanuláson alapuló ritkító algoritmusokkal.

5.5.1 Ritkított modell teljesítménye

Az 5-4. táblázat tartalmazza a különböző módszerekkel ritkított YOLOv4 modellek validálása során kapott eredményeket. A kiindulási modell a 4.1 fejezetben ismertetett, a KITTI adatbázison 68.5 % mAP értékre betanított modell. A saját ritkító algoritmussal előállított modell jól láthatóan felülmúlja a kézi beállítások mellett ritkított modelleket: az alapmodellhez képest 49 %-kal kevesebb paramétert tartalmaz és mintegy 7.2 %-kal jobb pontosságot ért el. Ezzel szemben a ritkaság szempontjából legjobb kézzel ritkított modell (handcrafted1) csupán -39.3 %-kal kevesebb paramétert tartalmaz az eredeti modellhez képest, ezzel 4.7 %-kal nagyobb pontosságot ért el, míg a pontosság szempontjából legjobb kézzel ritkított modell (handcrafted2) mAP értéke 4.9 %-kal javult az eredetihez képest, viszont csak 31.9 %-kal kevesebb paramétert tartalmaz.

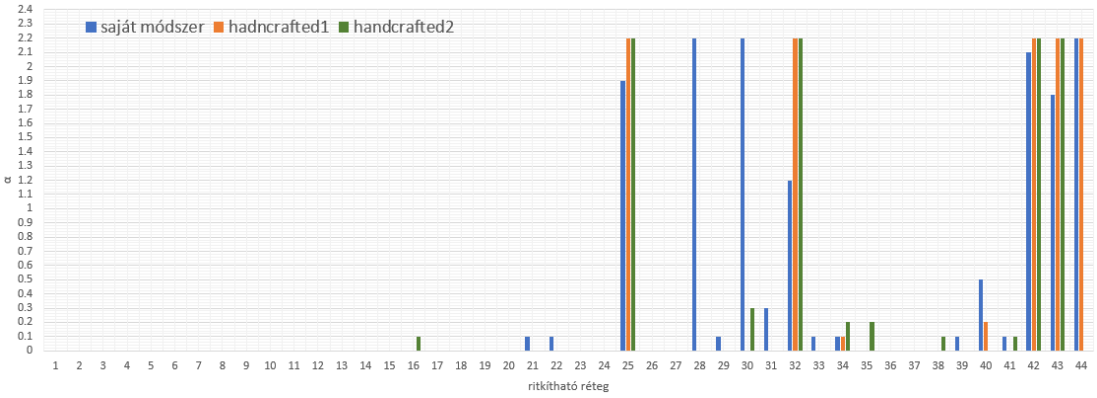
	mAP [%]	**paraméterszám [10^6 db]**	**méret [kB]**	**FLOPs [10^9 db]**	**futási idő [FPS]**
YOLOv4	68.5	63.9	250.5	60.54	53.76
Hadcrafted1	73.2 (+4.7%)	38.8 (-39.3%)	152.3	34.8	**61.88**
Handcrafted2	73.4 (+4.9%)	43.5 (-31.9%)	170.3	39.6	58.14
Saját módszer	**75.7 (+7.2%)**	**32.7 (-49%)**	**127.9**	**27.6**	58.82

5-4. táblázat. Különböző módszerekkel ritkított YOLOv4 modellek tulajdonságai.

A paraméterszám csökkenése minden módszer esetében a modell méretének csökkenésében és a futási idő javulásában is megmutatkozik. A kézzel szerkesztett modellek esetében az α értékek megválasztása azon feltételezésre támaszkodva történt, miszerint a háló hátsó rétegeiből érdemesebb több paramétert törölni, míg az elsőkből szinte semennyit. Az 5.10. ábra szemlélteti a rétegekhez választott α értékeket.

A YOLOv4 detektor ritkítását illetően state-of-the-art megoldásnak számít a YOLObile keretrendszer, melynek alkalmazásával a MS COCO adatbázison betanított eredeti modellből 93 % paraméter eltávolítása csupán 8.3 % mAP romlást eredményezett [28]. Hasonlóan kiváló eredményt mutat a YOLO-Tight módszer, mely a YOLOv3 architektúrát ritkítja közel 90 %-ig, a mAP 1 %-os romlása mellett [29]. Ezen

módszerekkel a saját pruning rendszerrel elért eredmény nem hasonlítható össze érdemben, hiszen vagy a kiindulási architektúrák különböznek (YOLO-Tight) vagy a modell betanítására használt adatbázis (YOLObile). Érezhető azonban, hogy mindkét esetben jóval nagyobb mennyiségű paramétert sikerült eltávolítani a kezdeti modellből, mint az általam megvalósított esetben.



5.10. ábra. Különböző pruning esetekhez választott α értékek.

Fontos azonban megjegyezni, hogy az ismertetett módszerek esetén a kisebb méretet és a gyorsabb futási időt priorizálták a pontosság enyhe romlásával szemben, míg esetemben a cél elsősorban a modell regularizálása volt. Ennek megfelelően, ahogy az 4.4.3 fejezetben is említettem a jutalomfüggvényt úgy formáltam, hogy a pontosságromlást nagyobb súllyal büntesse, mint az elért ritkaságot. A jutalomfüggvény más módon történő paraméterezése mellett nagy valószínűséggel elérhető lenne a nagy ritkaságot, de nagyobb pontosságromlást eredményező ritkított modell generálása is.

5.5.2 Algoritmus sebessége

Az ágens tanítási idejének szempontjából a saját ritkító algoritmust az AMC és PuRL módszerekkel hasonlítom össze. Az összehasonlíthatóságot illetően itt is megjelenik az probléma, hogy a három módszert nem ugyanazon kiértékelési szempontok szerint tesztelték: a saját módszer a YOLOv4 detektorhoz készült, a PuRL eljárást csak osztályozó neurális hálókra tesztelték, az AMC módszert pedig osztályozó hálók mellett a Faster R-CNN detektoron. Továbbá, ezen módszerek publikációiban csupán a konvergencia eléréséhez futtatott epizódok számát teszik közzé, az ehhez szükséges időt nem. Az említett módszereket ezért a következőképp alakítom összehasonlítható formára: mivel a saját megoldás legfontosabb különbsége a publikáltakhoz képest, hogy az ágens tanítása során a ritkított modell pontosságromlásának és ritkaságának meghatározásához validáció és finomhangolás helyett az állapotbecslő hálót alkalmazom, elvégzek olyan kísérleteket is, melyekben a saját rendszer ezen részét a publikált módszerekben alkalmazottra cserélem.

Az így született kísérletek eredménye az 5-5. táblázatban látható. Az AMC esetén a jutalom meghatározásához a ritkított modell validálása során kapott pontosság értékét használják a modell finomhangolása nélkül. Az eredeti módszer szerint egy epizódban csak egyszer – a teljes modell ritkítása után kap jutalmat az ágens, tehát epizódonként

egyszer végeznek validációt. A saját módszer esetén a sűrű jutalom miatt azonban ezt minden réteg ritkítása után el kell végezni, tehát epizódonként 44-szer. A PuRL eljárásban eredetileg is sűrű jutalmat alkalmaznak, de itt a validálás előtt 1 epizód erejéig újratanítják a ritkított modellt egy 1000 képből álló kis adatbázison. Ezen módszereket alkalmazva az NVIDIA TITAN X GPU-n mérve, 1 epizód futási ideje nagyságrendekkel nagyobb, mint a saját módszer esetén *batch_size=4096* batch méret mellett.

Saját módszer esetén konvergencia eléréséhez 700 epizódig kellett tanítani az ágenst. A megfelelő hiperparaméterek, állapottér-méret és jutalomfüggvény beállítások megtalálásához 90 tesztesetre volt szükség, tesztenként megközelítőleg 300 epizód futtatása kellett ahhoz, hogy láthatóvá váljon jó irányba halad-e a tanítás. 50 epizódonként a batch 10 véletlenszerűen kiválasztott mintájára validációt végez az algoritmus az állapotbecslő háló helyességének tesztelése végett. A teljes fejlesztési idő részét képezi az állapotbecslő háló tanítása és a tanításhoz szükséges adatok generálása is. Az automatikus adatgenerálás során 57000 adat keletkezett, ezek mindegyikéhez egy validáció futtatásának idejére volt szükség, összesen nagyjából 14 napra. Az állapotbecslő háló (SPN) tanítása 14 óráig tartott, a megfelelő paraméterezés megtalálásához 10 tesztre volt szükség. Ezek ismeretében a teljes fejlesztési idő a következő részidőkből áll elő:

$$t_{dev} = t_{data} + t_{SPN} + t_{tRL} + t_{fRL} \qquad \textbf{5-3}$$

ahol, t_{data} az adatgeneráláshoz szükséges idő, t_{SPN} az állapotbecslő háló betanításához szükséges idő, t_{tRL} az ágens konvergenciájához futtatni szükséges tesztesetek ideje és t_{fRL} a végleges modell tanítási ideje. Ezek egyenként a következőképp számíthatók:

$$t_{data} = num_{data} * t_{val}$$

$$t_{SPN} = test_{SPN} * t_{SPN\,train} \qquad \textbf{5-4}$$

$$t_{tRL} = (test_{RL} - 1) * (num_{avg\,ep} * t_{ep} + 10 * \frac{num_{avg\,ep}}{50} * t_{val})$$

$$t_{fRL} = num_{fin\,ep} * t_{ep} + 10 * \frac{num_{fin\,ep}}{50} * t_{val}$$

ahol num_{data} a generál adatok mennyisége, t_{val} a validációhoz szükséges idő, $test_{SNP}$ az állapotbecslő háló konvergenciájához szükséges tesztesetek száma, $t_{SPN\,train}$ az állapotbecslő háló tanítási ideje, $num_{fin\,ep}$ a végleges megoldásban az ágens konvergenciájához szükséges epizódok száma, $num_{avg\,ep}$ az epizódok száma, melyekből általánosan egy teszteset során következtetni lehet a konvergenciára, t_{ep} egy epizód futási ideje, $test_{RL}$ pedig megfelelő teljesítményt mutató modell megtalálásához szükséges tesztesetek száma. Ezek konkrét értékeinek felhasználásával a teljes fejlesztési idő:

t_{dev} = 57000*21s + 10*14*3600s + 89*(300*6.32s + 10*6*21s) + (700*6.32s + 10*14*21s) = 1989248 s = 23.023 nap

	Egy epizód futási ideje [s]	Epizód [db]	Ágens tanítási ideje (epizód * t_{ep}) [h]	Teljes fejlesztés ideje [nap]
AMC	21	400 [32]	2.33	**8.74**
AMC sűrű jutalommal	924 (= 21 * 44) = 15.4 m	400	102.67	385
PuRL	4879.6 (=110.9 * 44) = 1.35 h	**55** [31]	74.5	279.4
Saját módszer	**6.32**	700	**1.22**	23.023

5-5. táblázat. Megerősítéses tanulást alkalmazó ritkító módszerek összehasonlítása az ágens tanítási ideje és a teljes fejlesztési idő szempontjából.

Az AMC és PuRL módszereivel végzett kísérletek esetében egy epizód futási idejét mértem le, a teljes fejlesztési időt ebből és a publikációban közölt konvergenciához szükséges epizódok számából származtattam. A tesztek teljes futtatására idő hiányában nem került sor, hiszen azok a rendelkezésre álló GPU mellett az 5-5. táblázatban látható módon hónapokba telt volna. A teljes fejlesztési időre ezért csak megközelítőleges eredmények számolhatók. Feltéve, hogy a saját módszerhez hasonlóan ezekben az esetekben is 90 tesztre lett volna szükség az AMC sűrű jutalommal vett esetében a teljes fejlesztési idő 385 napot igényelt volna, a PuRL pedig 279.4 napot.

Az így kapott eredményekből jól láthatóvá válik az általam fejlesztett módszer legagyobb előnye: a leghosszabb időt a kezdeti adatgenerálás és az állapotbecslő háló tanítása veszi igénybe, ezek után viszont az ágens tanítása rendkívül gyorsan történik, sokkal több lehetőséget hagyva ezzel a fejlesztés érdemi részére: hiperparaméter-optimalizációra, jutalomfüggvény-formázásra és további, konkrétan az ágens tanításához köthető tesztekre. Ezzel ellentétben a már publikált módszerekben az egyes tesztek magukba foglalják a hosszas validációs időt is, nagymértékben korlátozva így az elvégezhető kísérletek számát, hiszen ahogyan láthattuk, egyes esetekben egyetlen teszt futtatása is hónapokba telhet egy közepes teljesítményű GPU alkalmazásával.

Az elkészült rendszer további erénye abban mutatkozik meg, hogy annak egyik része sem igényel nagy GPU memóriát, ellentétben például az újratanítást alkalmazó módszerrel. Ennek következtében a rendszer felhasználásával a megerősítéses tanulás alapú automatikus ritkítás területén történő fejlesztések elérhetővé válnak olyan kutatók számára is, akik nem rendelkeznek drága, nagyteljesítményű grafikus processzorokkal.

Összefoglaló

Munkám során a kezdeti terveknek megfelelően létrehoztam egy olyan megerősítéses tanulással automatizált ritkító rendszert, mely a KITTI adatbázison betanított YOLOv4 detektor optimális ritkítását teszi lehetővé. A fejlesztést megelőző irodalomkutatás során arra a következtetésre jutottam, hogy míg a state-of-the-art megoldások számos kiváló megoldást javasolnak az automatikus ritkítás feladatkörének különböző kihívásaira, közös hátrányuk, hogy az ágens számára a két legfontosabb környezeti paramétert, a paramétersűrűséget és a pontosságromlást a modell tényleges ritkításával és a redukált modell validációs adatbázison való tesztelésével határozzák meg futási időben, ami nagymértékben lelassítja a tanítási folyamatot.

Éppen ezért az elkészült rendszerben ezt a részt egy új, legjobb tudomásom szerint az irodalomban eddig nem ismertetett, modell alapú megerősítéses tanulási módszerrel valósítom meg: a környezeti változókat a validálás helyett egy úgynevezett állapotbecslő háló segítségével határozom meg, melyet előzőleg automatikusan generált adatokon tanítok be. Az ágens állapotterét 6 dimenziósra választom, sűrű jutalom módszerét alkalmazom, tehát az ágens minden réteg ritkítása után kap visszajelzést a környezettől. A jutalomfüggvényt úgy formálom, hogy a ritkított modell a lehető legkevesebbet veszítsen pontosságából, a nagy ritkaság csak másodlagos szempont. A megerősítéses tanítást az A2C algoritmussal valósítom meg, és mivel az állapotbecslő háló alkalmazása lehetővé teszi, hogy csupán a kezdeti modell állapotát kelljen betölteni, egyszerre több ágens futtatása is lehetővé válik, mely hozzájárul a tanítási folyamat stabilizálásához.

Az elkészült automatikus rendszer alkalmazásával egy olyan ritkított YOLOv4 modellt sikerült generálni, 50 %-kal kevesebb paramétert tartalmaz az eredetihez képest és 7.2 %-kal jobb mAP értéket ér el. Ezen eredmény mAP értékben 2.3 %-kal, ritkaságban pedig 17.1 %-kal felülmúlja a legjobb teljesítményű kézi beállítások mellett ritkított modellt. Egy epizód futási idejét tekintve felülmúlja a state-of-the-art módszereket: 772-szer gyorsabb, mint a PuRL módszer és 146.2-szer gyorsabb, mint az AMC. Fontos kiemelni azonban, hogy az említett módszerek érdemben nem hasonlíthatóak össze az eltérő architektúrák és tanító adatbázisok alkalmazása miatt. Teljes fejlesztési idő tekintetében az epizód futási idejéből származtatott eredményekről elmondható, hogy a rendelkezésre álló GPU mellett a saját módszerrel való fejlesztés hónapokkal kevesebb időt venne igénybe, minta azt a már publikált módszerek alkalmazásával végeznénk.

Összességében az elkészült módszer legnagyobb előnye abban mutatkozik meg, hogy a leghosszabb időt a kezdeti adatgenerálás veszi igénybe, ezek után viszont az ágens tanítása rendkívül gyorsan történik, sokkal több lehetőséget hagyva ezzel a fejlesztés érdemi részére, a megerősítéses tanulási algoritmus formázására. Mindemellett a rendszer egyik része sem igényel nagy GPU memóriát, ezáltal annak alkalmazásával a mesterséges intelligencia ezen kutatási területe elérhetővé válik olyan kutatók számára is, akik nem rendelkeznek nagyteljesítményű, drága GPU-val.

A jövőben megvalósítandó tervek részét képezi az ágens tanítása a jutalomfüggvény olyan paraméterezése mellett, mely egy nagyobb mértékben ritkított modell megtalálását teszi lehetővé. Továbbá, szeretném elvégezni az ágens tanítását COCO adatbázison betanított YOLOv4 detektort használatával, mellyel lehetővé válik a saját módszer érdemleges összehasonlítása a state-of-the-art YOLOv4-hez tervezett pruning módszerrel a ritkított modell teljesítményének szempontjából. Hasonlóképp az ágens tanítási idejének érdemi összehasonlíthatósága érdekében, a rendszert tesztelném olyan osztályozó architektúrákon, melyeket a state-of-the-art automatikus ritkító rendszerek is alkalmaznak. Végül, szeretném a rendszert kibővíteni úgy, hogy bármiféle előleges beavatkozás nélkül képes legyen bármilyen neurális háló architektúra automatikus ritkítására.

Irodalomjegyzék

[1] Ian Goodfellow, Yoshua Bengio, Aaron Courville, Deep Learning, MIT Press, 2016.

[2] M. A. Nielsen, "Neural Networks and Deep Learning," Determination Press, 2015. [Online]. Available: http://neuralnetworksanddeeplearning.com/index.html. [Accessed 25. 05. 2020.].

[3] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278-2324, 1998.

[4] Y. LeCun, K. Kavukcuoglu, and C. Farabet, „Convolutional networks and applications in vision.," *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*, pp. 253-256, 2010.

[5] Mitchell, Tom M., Machine Learning, McGraw-Hill Science/Engineering/Math, 1997.

[6] R. S. Sutton, "Two Problems with Backpropagation and Other Steepest-Descent Learning Procedures for Networks," *Proceedings of the Eighth Annual Conference of the Cognitive Science Society*, 1986.

[7] D. E. Rumelhart, G. E. Hinton, and R. J. Williams,, "Learning Representations by Back-propagating Errors," *Nature 323 (6088)*, pp. 533-536, 1986.

[8] Sagar Sharma, "Activation Functions in Neural Networks," 2017. [Online]. Available: https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6. [Accessed 26. 05. 2020.].

[9] R. B. Girshick, „Fast R-CNN.," *IEEE International Conference on Computer Vision (ICCV)*, pp. 1440-1448, 2015.

[10] S. Ren, K. He, R. Girshick and J. Sun, "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 39, no. 6, pp. 1137-1149, 2017.

[11] J. Redmon, S. Divvala, R. Girshick and A. Farhadi, „You Only Look Once: Unified, Real-Time Object Detection," *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 779-788, 2016.

[12] J. Redmon and A. Farhadi, „YOLO9000: Better, Faster, Stronger," *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 6517-6525, 2017.

[13] J. Redmon and A. Farhadi, „YOLOv3: An Incremental Improvement.,” 2018. [Online]. Available: https://arxiv.org/abs/1804.02767.

[14] A. Bochkovskiy, C.-Y. Wang and H.-Y. M. Liao, "YOLOv4: Optimal Speed and Accuracy of Object Detection," 2020. [Online]. Available: https://arxiv.org/abs/2004.10934v1. [Accessed 17 05 2021].

[15] Wei Liu and Dragomir Anguelov and Dumitru Erhan and Christian Szegedy and Scott Reed, „SSD: Single Shot MultiBox Detector,” 2016. [Online]. Available: https://arxiv.org/abs/1512.02325.

[16] Z. Huang és J. Wang, „DC-SPP-YOLO: Dense Connection and Spatial Pyramid Pooling Based YOLO for Object Detection,” 2019. [Online]. Available: https://arxiv.org/abs/1903.08589.

[17] D. Blalock, J. J. G. Ortiz, J. Frankle and J. Guttag, "What is the State of Neural Network Pruning?," 2020. [Online]. Available: https://arxiv.org/pdf/2003.03033.pdf. [Accessed 24 10 2021].

[18] R. S. Sutton and A. G. Barto, Reinforcement Learning: An Introduction, second ed., Cambridge, United States: MIT Press Ltd, 2018.

[19] A. P. Engelbrecht, Computational Intelligence: An introduction, Second edition szerk., John Wiley & Sons, Ltd, 2007, p. page 48.

[20] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver és K. Kavukcuoglu, „Asynchronous Methods for Deep Reinforcement Learning,” *International Conference on Machine Learning*, 2016.

[21] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver and D. Wierstra, "Continuous control with deep reinforcement learning," 2016. [Online]. Available: https://arxiv.org/abs/1509.02971. [Accessed 24. 10. 2021.].

[22] J. Schulman, F. Wolski, P. Dhariwal, A. Radford and O. Klimov, "Proximal Policy Optimization Algorithms," 2017. [Online]. Available: https://arxiv.org/abs/1707.06347. [Accessed 30. 10. 2021.].

[23] M. Riedmiller, „Method, Neural Fitted Q Iteration – First Experiences with a Data Efficient Neural Reinforcement Learning,” *European Conference on Machine Learning*, 2005.

[24] J. Schulman, S. Levine, P. Abbeel, M. Jorda és P. Moritz, „Trust Region Policy Optimization,” in *Proceedings of the 32nd International Conference on Machine Learning*, 2015.

[25] G. Li, C. Qian, C. Jiang, X. Lu és K. Tang, „Optimization based Layer-wise Magnitude-based Pruning for DNN Compression,” in *International Joint Conference on Artificial Intelligence*, 2018.

[26] Zhang, X. a. Zou, J. a. He, K. a. Sun és Jian, „Accelerating Very Deep Convolutional Networks for Classification and Detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence,* pp. 1943-1955, 2016.

[27] S. Han, J. Pool, J. Tran és W. J. Dally, „Learning both Weights and Connections for Efficient Neural Network," *Advances in Neural Information Processing Systems,* 2015.

[28] Y. Cai, H. Li, G. Yuan, W. Niu, Y. Li, X. Tang, B. Ren és Y. Wang, „YOLObile: Real-Time Object Detection on Mobile Devices via Compression-Compilation Co-Design," *Association for the Advancement of Artificial Intelligence,* 2021.

[29] W. Yan, T. Liu és Y. Fu, „YOLO-Tight: an Efficient Dynamic Compression Method for YOLO Object Detection Networks," in *13th International Conference on Machine Learning and Computing,* 2021.

[30] L. Hu és Y. Li, „Micro-YOLO: Exploring Efficient Methods to Compress CNN based Object Detection Model," *Proceedings of the 13th International Conference on Agents and Artificial Intelligence,* 2021.

[31] M. Gupta, S. Aravindan, A. Kalisz, V. Chandrasekhar és L. Jie, „Learning to Prune Deep Neural Networks via Reinforcement Learning," 2020. [Online]. Available: https://arxiv.org/abs/2007.04756.

[32] Y. He, J. Lin, Z. Liu, H. Wang, L.-J. Li és S. Han, „AMC: AutoML for Model Compression and Acceleration on Mobile Devices," *European Conference on Computer Vision,* 2019.

[33] J. Chen, S. Chen és S. J. Pan, „Storage Efficient and Dynamic Flexible Runtime Channel Pruning via Deep Reinforcement Learning," *Advances in Neural Information Processing Systems,* 2020.

[34] "COCO Dataset webpage," [Online]. Available: https://cocodataset.org/#home. [Accessed 24. 10. 2021.].

[35] „The KITTI Vision Benchmark Suite," [Online]. Available: http://www.cvlibs.net/datasets/kitti/.

[36] WongKinYiu, "ScaledYOLOv4," [Online]. Available: https://github.com/WongKinYiu/ScaledYOLOv4/tree/yolov4-csp. [Accessed 24. 10. 2021.].

[37] T. He, Z. Zhang, H. Zhang, Z. Zhang, J. Xie és M. Li, „Bag of Tricks for Image Classification with Convolutional Neural Networks," in *Conference on Computer Vision and Pattern Recognition,* 2018.

[38] T. M. Moerland, J. Broekens and C. M. Jonker, "Model-based Reinforcement Learning: A Survey," 2020. [Online]. Available: https://arxiv.org/abs/2006.16712. [Accessed 24. 10. 2020.].