Meglécz Máté

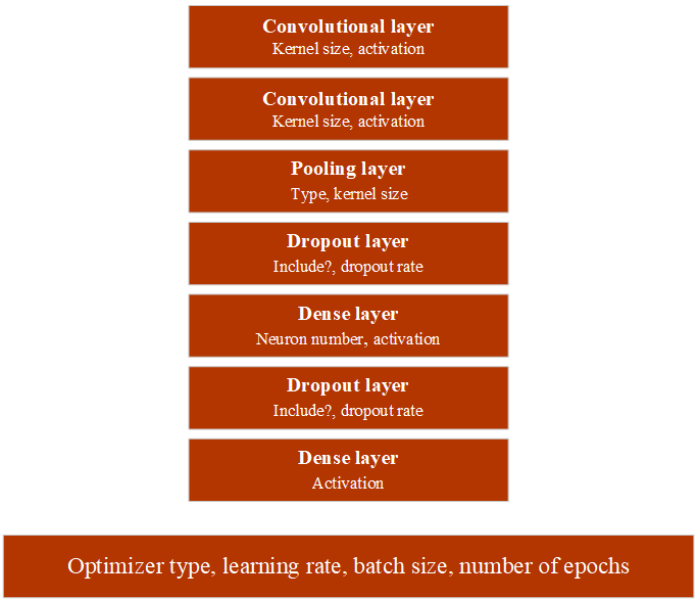
CNN architektúra optimalizálás genetikus algoritmus segítségével

Neurális hálózatok házi feladat

Házi feladatként egy adaptív genetikus algoritmust készítettem egy konvolúciós neurális háló architektúrájának optimalizálására. A feladat megoldásához Jupyter notebook-ot használtam. Tanítóhalmazként a CIFAR10 adathalmazt alkalmaztam, ami, 10 osztályból, osztályonként 6000 32x32-es színes képből áll. A háló felépítéséhez az alábbi demo-t használtam segítségül: http://home.mit.bme.hu/~hadhazi/Oktatas/NN18/dem3/html_demo/CIFAR-10Demo.html. Célnak azt tűztem ki, hogy a genetikus algoritmus által megtalált optima jobb legyen (kisebb loss), mint a demoban szereplő háló.

Optimalizálandó architektúra

A neurális hálózat felépítéséhez a Tensorflow Keras magas-szintű neurális hálózat könyvtárat alkalmaztam.



A háló 2 konvolúciós layer-el indul, amik 2 paraméterrel rendelkeznek: kernel méret és aktivációs függvény. Ezek után következik egy pooling layer, ami fajtája és kernel mérete szerint optimalizálható. A következő szinten egy opcionális dropout layer található, dropout rate paraméterrel. Majd egy dense layer jön neuron szám és aktivációs függvénnyel. A következő szintén egy opcionális dropout layer. Végül az utolsó szint egy dense layer aktivációs függvény paraméterrel, a neuron számának 10-nek kell lennie a 10 osztály miatt. A szintek paraméterei mellett még az optimalizáló fajtája, a learning rate-je, a batch mérete és a tanítási epoch-ok száma kerül optimalizálásra.

A paraméterekhez tartozó tartományokat igyekeztem úgy megválasztani, hogy ne legyen a paraméter tér mérete túl nagy, mert nagyon le tudta volna csökkenteni az optimum felé konvergálás sebességét. Ezért is érdemes jól megfontoltan megválasztani a tartományokat, például az előre tudható, hogy az utolsó dense layer várhatóan Softmax függvénnyel fog rendelkezni, viszont a demonstráció kedvéért meghagytam az algoritmusnak a lehetőséget, hogy nem ideális opciók is előfordulhassanak. Az optimalizálandó paraméterek és a hozzájuk tartozó tartományok a fenti táblázatban találhatóak.

Parameters	Ranges
Kernel size	1x1, 3x3, 5x5, 7x7, 9x9
Pooling layer types	Max Pooling, Average Pooling
Neuron numbers	1-300
Number of training epochs	1-20
Learning rate	0.0001 - 0.2
Batch sizes	16, 32, 64
Optimizer types	Adam, SGD
Activation function types	ReLU, Softmax
Include dropout	True/False
Dropout rate	0-1

Genetikus algoritmus

A genetikus algoritmus felépítéséhez *Ivan Gridin, Learning Genetic Algorithms with Python: Empower the performance of Machine Learning and AI models with the capabilities of a powerful search algorithm* című könyvét használtam segítségül. A genetikus algoritmus a természetes evolúció által inspirált heurisztikus algoritmus. Jellemzően optimalizálási és keresési feladatokra alkalmazzák.

Az algoritmus egyedeit, a különböző paraméter értékek szerint felépített architektúrák képzik. Ezek a paraméter értékek képzik az egyedek kromoszómáit. Az egyes generációban jelenlevő egyedek összeségét populációnak nevezzük.

Az egyedek „jóságát" a teszt adathalmazon végzett kiértékelések loss értéke szerint vizsgáljuk, minél kisebb egy neurális háló loss értéke annál jobb az adott architektúra. Ez azt jelenti, hogy ahhoz, hogy egy egyednek meg tudjuk határozni a loss értékét, be kell tanítani a tanító adathalmazon, majd ki kell értékelni a teszt adathalmazon. A tanítási és kiértékelési folyamat jellemzően a legnagyobb számítási szükséglettel rendelkező része az algoritmusnak, ezen feladat esetén is elmondható ez, hiszen a neurális hálózatok tanítása számításigényes feladat, ezért is volt fontos például, hogy esetemben minél kisebb architektúrát optimalizáljak, illetve mint korábban is látható volt, hogy pl. a tanítási epoch-ok száma felülről erősen korlátozva legyen.

Először felépítettem a genetikus algoritmus alapját, ami egy fitness kalkulációból, szülő egyedek kiválasztásából, keresztezésből, mutációból és a következő generációra fentmaradó populáció kiválasztásából áll.

Fitness függvény

A feladathoz egy egyszerű ranking fitness megoldást választottam. Ez a megoldás sorba rendezi a loss-ok alapján az egyedeket, majd egy adott tartományon beül egyenletesen kioszt egy fitness értéket az egyedeknek. Esetemben a legrosszabb egyed 0 fitness értékkel rendelkezik, a legjobb pedig 2-vel.

Szülő egyedek választása (Selection)

A szülő egyedek kiválasztására sztohasztikus univerzális mintavételezést alkalmaztam, majd az így kiválasztott egyedeket szülőpárokba állítottam, figyelve rá, hogy magával ne állítódjon párba egy egyed sem és hogy ne szerepeljen kétszer ugyanaz a pár. A sztohasztikus univerzális mintavételezés (SUS) minimális szórást és nulla torzítást ad, egyenletes tartományokra választja ki az egyedeket, így esélyt adva a rosszabb egyedek számára is, hogy szülő legyen.

Keresztezés (Crossover)

A szülőpárokban levő egyedek keresztezésére egyenletes keresztezést használok. Így minden kromoszóma hely keresztezési ponttá válhat. Implementációmban minden paraméterre 50% esély van, hogy keresztezés történik az adott ponton. A keresztezés közbenső rekombinációval történik az alábbi képlet szerint, amennyiben egy egész számokból álló tartományra történik rekombináció az eredmény kerekítve lesz. Amennyiben a paraméter enum értékekből áll a két szülő felcseréli az értékeit. Minden keresztezésből 2 új gyerek egyed keletkezik.

$$\sigma = P_1 + \mathbf{a}. * (P_2 - P_1)$$

Mutáció (Mutation)

A mutációra a paraméter tér jobb felderítésének érdekében van szükség. A mutáció során a populáció minden egyedéből valamekkora eséllyel keletkezik egy új egyed. A keresztezéshez hasonlóan a mutáció során minden paraméter 50% eséllyel mutálódik, amennyiben egy szám halmaz képzi a paraméter tartományát, gauss eloszlás szerint generálódik az új paraméter, figyelembe véve, hogy a tartomány határaiból ne csússzon ki az új érték. Enum értékek esetén véletlenszerűen új értéket kap.

Következő populáció kiválasztás (Reinsertation)

A keresztezés során új egyedekkel bővült a populáció, viszont a generációkban kötött a populációk mérete, ezért szükség van generációról-generációra a fentmaradó egyedek kiválasztására. Erre alkalmazok egy elit stratégiát, ami a következő populáció valahány százalékát az előző populáció legjobb egyedeinek tartja fent. Ezen egyedek kiválasztása után a maradék egyed a régi egyedek, gyerek egyedek és mutáció során létrejövő új architektúrákból kerül ki a legjobbak kiválasztásval.

Az algoritmus paraméterei

A genetikus algoritmushoz tartoznak paraméterek, amik alapján szabályozható az optima felé konvergálás sebessége, illetve a globális optima megtalálásának az esélye. Az algoritmusnak szüksége van arra, hogy hány generációra szeretnénk futtatni, mekkorák legyenek a populációk méretei, hány százaléknyi gyerek elem generálódjon, mekkora eséllyel történjen mutáció és hogy a populáció mekkora százaléka legyen elit. Általánosan, ha a konvergálás sebességét szeretnénk növelni akkor csökkentjük a populáció méretét és csökkentjük a mutációk esélyét, viszont így jellemzően az algoritmus egy lokális optima felé fog konvergálni. Amennyiben a lehető legjobb megoldást szeretnénk kapni, növelni kell a populáció méretét és növelni kell a mutáció esélyét, hogy jobban fel tudjuk deríteni a paraméter teret. Viszont ez így a konvergálás lelassulását eredményezi.

Mivel a feladatom elég számításigényes, így elég hosszú futás idővel rendelkezik kis generáció számra és kis populáció méretre is, nem engedhettem meg, hogy többszöri futtatással találjam meg az algoritmus optimális paramétereit.

Adaptív genetikus algoritmus

A korábban felsorolt problémák miatt döntöttem az előzőleg implementált algoritmus adaptivitással való kibővítése mellett.

Az adaptív genetikus algoritmus célja, hogy futás közben generációról-generációra dinamikusan állítsa az algoritmus értékeit, hogy minél gyorsabban lehessen megtalálni a lehető legjobb eredményt. Ezt úgy valósítja meg, hogy generációról-generációra figyeli a populációban levő legjobb egyedet, amiennyiben ez az egyed javuló tendenciában van, akkor az egyedek konvergálnak, tehát csökkenteni lehet a gyerekek, mutációk számát illetve a populáció méretét, így gyorsítva a konvergálást, viszont ha a populációk nem javulnak, akkor az algoritmus feltehetőleg beragadt egy lokális optima-ba, így növelni kell a populáció méretét új random egyedekkel és növelni kell a felderítés valószínűségét, tehát növelni kell a gyerekek számát és a mutációk esélyét. Amennyiben a populációk hosszú ideje nem javulnak, az azt jelenti, hogy lehetséges, hogy megtaláltuk a globális optima-t tehát az algoritmust felesleges tovább futtatni, ezért kilép a ciklusból.

Javulás vizsgálat

Annak az eldöntése, hogy a populációk javuló tendenciában vannak-e, az előző előre meghatározott mennyiségű generáció legjobb egyedeinek az átlag loss értékéhez való eltérés vizsgálatával történik. Amennyiben a javulás, ehhez az átlaghoz képest egy bizonyos % alatt van úgy vesszük, hogy már nem történik javulás.

A teljes adaptív genetikus algoritmus pszeudokódja

```
WHILE current_generation < min_generation OR
        ( current_generation < max_generation AND best result not improving):
        IF best result not imporving:
                Increase mutation rate
                Increase crossover rate
                Increase population size
        ELSE:
                Decrease mutation rate
                Decrease crossover rate
                Decrease population size

        Calculate fitnesses
        Select parents
        Crossover
        Mutate
        Select new population
```

Eredmények

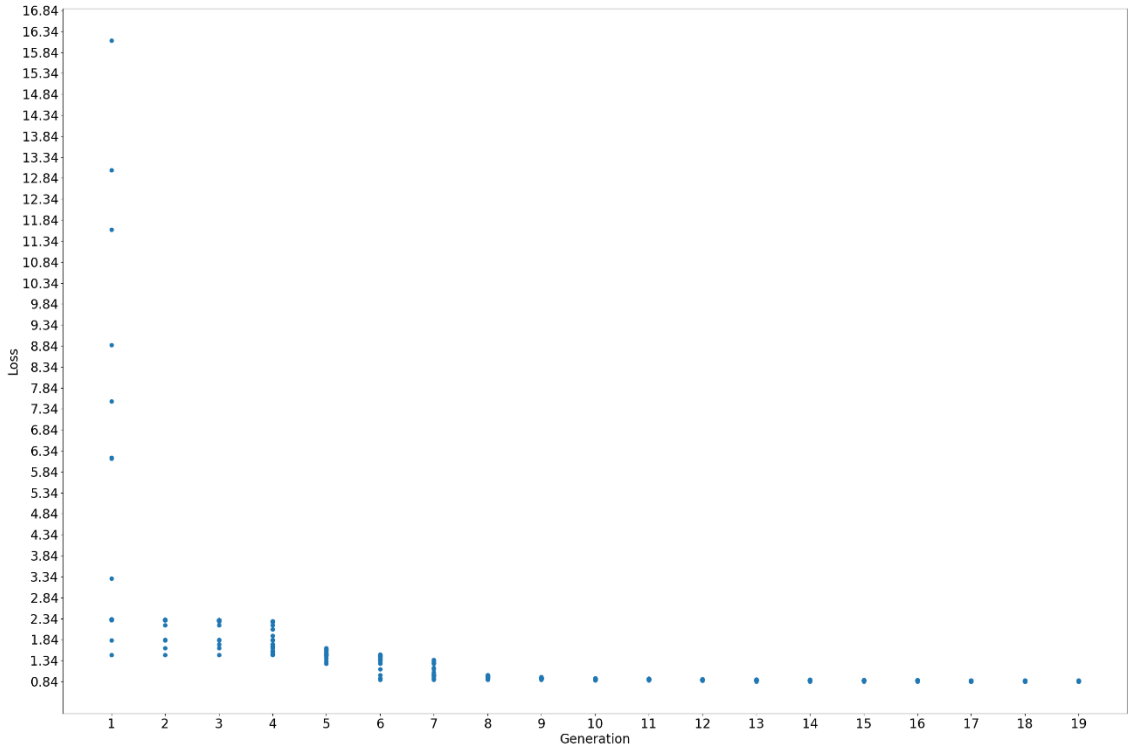
A számítási kapacitás területén sajnos elég erős megkötéseim voltak, az eredeti tervem az volt, hogy a programot Google Colab-al fogom futtatni, azonban fejlesztés közben realizáltam, hogy Magyarországon nincs lehetőség előfizetés vásárlására, hogy ne zárjon ki a session-ből futás közben. Így kénytelen voltam saját hardwaren futtatni, ami így megötszörözte az egyes epoch-ok futási idejét (Google Colab: ~10s, saját hardware: ~50s). Próbáltam párhuzamosítani a tanításokat 2 job-ra, ami talán minimális gyorsítást eredményezett.

Az alább vizsgált futtatás esetén a genetikus algoritmus paraméterei:

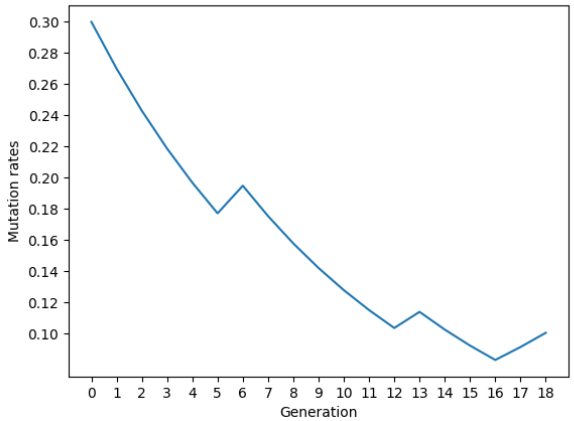
Ezek mellett, a ciklus feltételeként használt improvement vizsgálatnál az utóbbi 7 generációra tekint vissza, míg az if-else-ben az utóbbi 5-re.

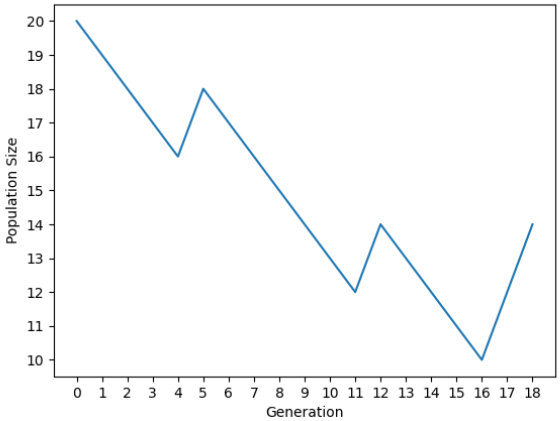
Parameter	Value
Max generations	50
Min generations	10
Max population size	50
Min population size	6
Start population size	20
Start mutation rate	30%
Min mutation rate	0%
Start crossover rate	0.8%
Min crossover rate	0.1%
Elit rate	10%
Improvement thershold	1%

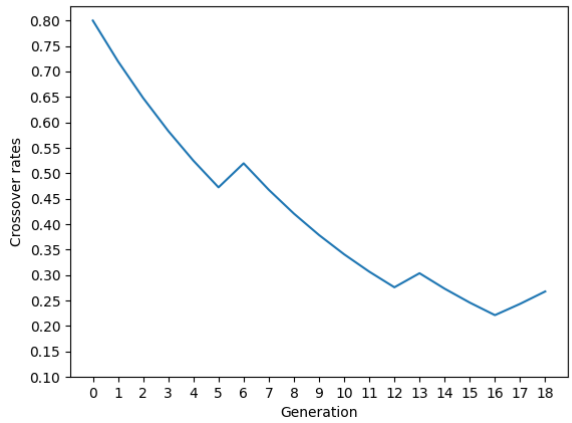
Az algoritmus ezekkel a beállításokkal 19 generációt futott. A generációkban a loss értékek eloszlása az alábbi ábrán látható. Látszik hogy a populációban az értékek egyre közelebb vannak egymáshoz és hogy az utolsó generációkban már nem látható jelentős javulás.



A következő diagramokon a populáció méretének, a mutáció valószínűségének és a gyerekek arányának a változása látható, ami az adaptivitásnak az eredménye:



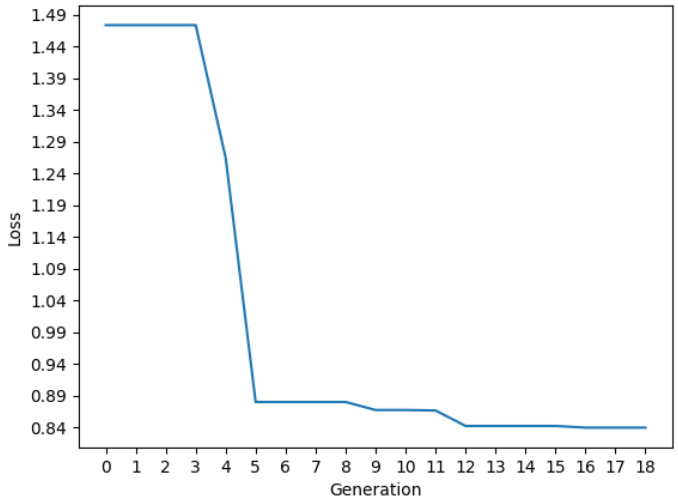




A mutation és a crossover értéken is 10%-ot növeltem/csökkentettem, ami feltehetőleg szerencsésebb lett volna növelésnél nagyobb értéknek választani vagy fix értékekkel növelni/csökkenteni, mert így nagyon lassan kezdett el növekedni az új egyedek száma, amikor szükség lett rájuk.

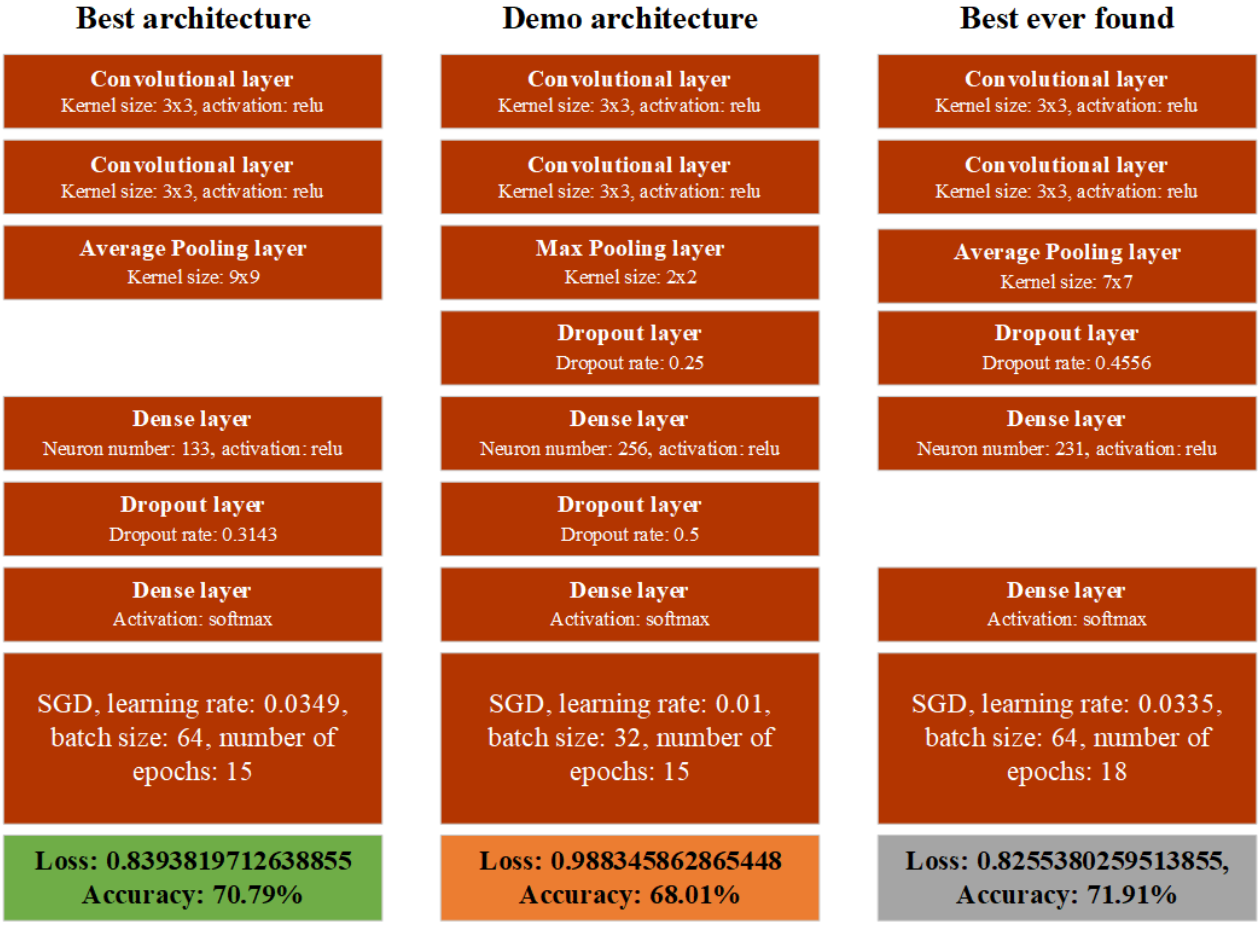
Emellett úgy gondolom, hogy a leállási kritérium visszatekintését is lehetett volna még növelni jobb eredmény reményében. Illetve az 1%-nál kisebb változást is még javulásnak tekinteni.

A legjobb loss eredmények alakulása generációként:



Az algoritmus végeredménye

A lenti ábrán az algoritmus által megtalált legjobb architektúra (bal) látható összehasonlítva a demo-ban látható architektúrával (közép). Látható, hogy az algoritmus sikeresen talált egy jobb architektúrát, mint a demo architektúrája. Azonban biztosan kijelenthetem, hogy ez az egyed sem a legjobb, mivel egy korábbi futtatásból volt jobb eredményem, ez az eredmény látható a jobb oldalon.



Értékelés

Feltehetően a korábban megfigyelt hibák orvosolásával, illetve nagyobb számítási kapacitással sikerülhetett volna megtalálni a globális optimumot, azonban úgy értékelem, hogy ez is egy elfogadható eredmény. Az itt bemutatott futtatás 14 órát vett igénybe egy AMD RX590 típusú videokártyán, jobb hardware-el lehetőség lett volna növelni a populáció méretét, ezzel jobban lefedve a paraméter teret.

Egyéb felhasznált irodalom

A korábban felsoroltak mellett inspirációként használtam: **https://arxiv.org/abs/2006.12703 : Efficient Hyperparameter Optimization in Deep Learning Using a Variable Length Genetic Algorithm**