M07 – Objektumkövetés

Mérési útmutató

Irányítástechnika és képfeldolgozás laboratórium 1.

Szemenyei Márton, Reizinger Patrik

Irányítástechnika és Informatika Tanszék
2018

Tartalomjegyzék

1. Objektumkövetés konvolúciós hálók segítségével

A számítógépes látás a számítástudomány egyik legrohamosabban fejlődő területe, amelynek egyre több gyakorlati felhasználása létezik. Ez a tendencia nem meglepő, hiszen az ember az érzékszervei közül a szemre hagyatkozik a leginkább a napi feladatinak ellátásakor. Ebből következik, hogy ha a számítógépes látás algoritmusai képesek megközelíteni, vagy akár felülmúlni az emberi látás képességeit, akkor számos fontos feladatot leszünk képesek automatizálni. Könnyen belátható azonban az is, hogy a gyakorlatban ez egy rendkívül nehéz feladat, mivel általában olyan feladatokat tudunk könnyedén algoritmusok formájában leírni, ahol a feladat elvégzésének a menetét pontosan, tudatos szinten megértjük. A szemből érkező jelek feldolgozásának azonban a jelentős része tudatalatti szinten történik, így aligha tudjuk ezeket a folyamatokat könnyedén megérteni, és algoritmusok formájában lemásolni.

Ennek a problémának a megoldásához a mesterséges intelligencia, ezen belül is a gépi tanulás módszereihez kell nyúlnunk. A tanuló számítógépes látás legegyszerűbb formája az osztályozás, vagyis amikor egy képhez egyetlen címkét rendelünk, amely a képen található objektum kategóriáját kódolja. Bizonyos esetekben a címke mellé már egy az adott objektumot körbefogó téglalapot is rendelünk, ebben az esetben lokalizációról beszélhetünk. Adott esetben ennél magasabb szintű információt is szeretnénk kinyerni a képből: érdekelhet minket az, hogy melyek azok a pixelek az adott képen, amelyek az észlelt osztályhoz tartoznak. Amennyiben ezt az információt egy képsorozaton minden pixelre meghatározzuk, akkor követésről beszélhetünk.

1.1.Neurális hálózatok alapjai

A felügyelt tanulás alapvető módszereinek rövid bemutatása után a jelenlegi fejezettel kezdődően a mély tanulás alapú látórendszerek területének módszereit fogom részletesen bemutatni. A mély tanuló rendszerek alapeleme egy lineáris osztályozó algoritmus, amelynek számos elnevezése létezik. Gyakran szokás perceptron, illetve neuron elnevezéssel illetni, valamint – osztályozó jellege ellenére – logisztikus regresszió néven is ismert. Az algoritmus működésének lényege, hogy a kép pixeleit egyetlen vektorba rendezi, majd ezt a vektort egy súlymátrixszal szorozza meg, így egy kimeneti vektort állítva elő, aminek annyi eleme van, ahány osztály között döntenünk kell. Ennek a vektornak minden egyes eleme értelmezhető úgy, mint az egyik osztály „jósági” értéke, vagyis minél nagyobb, annál inkább tartozik a kép az adott osztályba. Formálisan a következőképp adható meg a perceptron modellje:

$$f(x, W) = Wx$$

Ahol x a bemenet, W pedig a paraméterek, vagy súlyok mátrixa. Az osztályozás ilyen módját úgy lehet elképzelni, hogy a W mátrix i-edik sora kijelöl egy olyan irányt a pixelek terében, amerre az i-edik osztály jósága nő. Ennek alapján az egyes osztályok közötti döntési határok egyenes szakaszokból tevődnek össze (bináris esetben egyetlen egyenes/hipersík). A módszer alapvető kérdése azonban, hogy hogyan lehet a súlyok értékét úgy meghatározni, hogy az osztályozás minél pontosabb legyen, valamint, hogy milyen költségfüggvény segítségével mérhető jól az algoritmus teljesítménye.

Első nekifutásra célszerű lehet az osztályozás minőségét a jól eltalált tanító adatok arányával jellemezni, ez azonban nem képes különbséget tenni egy azonos pontosságú, de eltérő bizonytalanságú osztályozás végző modell között. Éppen ezért a modell kimenete és az adott tanítóadathoz előírt kimenet között egészen új költségfüggvényeket fogunk definiálni, melyeknek az egész tanító adathalmazra vett átlaga megadja a teljes hiba mértékét. Célszerűnek tűnhet egyszerűen az elvárt és a becsült kimenet közti négyzetes hibát venni, amely regressziós problémák esetén a leggyakrabban használt hibafüggvény. A kimeneti érték numerikus közelítése viszont osztályozás esetében nem feltétlenül praktikus, és habár a négyzetes hiba ilyen esetekben is használható, mégis könnyedén lehet jobb hibafüggvényeket konstruálni.

1. ábra: Az SVM kötség geometriai szemléltetése.

Az egyik gyakran használt hiba az úgynevezett Hinge, vagy SVM hibafüggvény. Ennek a hibafüggvénynek az alapelve, hogy definiálunk egy Δ mennyiséget, amit résnek nevezünk, és ha a helyes osztály jósága legalább ezzel az értékkel nagyobb az összes többi jóságnál, akkor a hiba értéke 0. Ellenkező esetben a hiba értéke lineárisan nő. Ez a hibafüggvény felfogható egyfajta "biztonságos" elválasztást előíró kritériumként. Az SVM hiba formálisan a következő:

$$L_i = \sum_{j \neq corr} max\left(0, s_j - s_{corr} + \Delta\right), ahol\ s = f(x_i, W)$$

Létezik egy másik gyakorlatban elterjedt hibafüggvény, amelyik a geometriai szemlélet helyett inkább a valószínűségszámítás oldaláról közelíti meg a problémát. Ez a költségfüggvény az entrópia fogalmát használja fel. Az entrópia fogalma arra épül, hogy ha különböző valószínűséggel történő eseményeket szeretnénk elkódolni, akkor nem érdemes minden eseményre ugyanannyi bitet szánni, a valószínű eseményeket kevés, míg a valószínűtleneket sok biten érdemes ábrázolni, így az összes esemény közlésére elhasznált bitek mennyiségét minimalizálni lehet. Egy p valószínűséggel bekövetkező eseményt a p logaritmusának reciprokával megegyező számú biten érdemes kódolni. Ezt felhasználva az entrópia megadja az összes eseményre elhasznált bitek számának várható értékét:

$$H(p) = \sum_i p_i \frac{1}{\log p_i} = -\sum_i p_i \log p_i$$

Belátható azonban, hogy ha a p valószínűségi eloszlást nem ismerjük, hanem csak egy közelítő q eloszlást, akkor az optimálisnál csak nagyobb eredményt kapunk. Ezt a nagyobb értéket fejezi ki a keresztentrópia mértéke. Persze minél inkább közelíti a q eloszlás a p-t, annál inkább csökken a keresztentrópia. A két entrópiafajta különbségét KL divergenciának nevezzük, ami egy szigorúan nemnegatív függvény, amelyet gyakran használnak valószínűségi eloszlások hasonlósági mércéjének.

$$H(p,q) = \sum_i p_i \frac{1}{\log q_i} = -\sum_i p_i \log q_i$$

$$KL(p \parallel q) = H(p,q) - H(q)$$

A keresztentrópia felhasználható osztályozási hibafüggvényként az alábbi módon: első lépésként a modell kimeneti jóságait egy SoftMax nevű normalizáló függvény segítségével valószínűség jellegű értékekké konvertáljuk. Ez a függvény minden értéket a [0,1] tartományba transzformál úgy, hogy az értékek összege pontosa egy legyen. A SoftMax függvény az alábbi módon írható fel:

$$q_k = q(y_k \mid x) = \frac{e^{s_k}}{\sum_j e^{s_j}}, ahol\ s = f(x, W)$$

Innen a hibafüggvényt úgy definiáljuk, mint a címkék elvárt eloszlása, és a becsült q eloszlás közti keresztentrópia. Mivel a keresztentrópia akkor minimális, ha a két eloszlás megegyezik, ezért ennek a függvénynek a minimalizálásával a becsült valószínűségek az előírtakhoz fognak tartani. A címkék előírt eloszlását úgy konstruáljuk, hogy a helyes osztály elvárt valószínűségét 1-nek, míg az összes többiét nullának választjuk. Így a keresztentrópia hibafüggvény az alábbi alakra egyszerűsödik:

$$L_i = H(p_i, q_i) = -\sum_k p_{k,i} \log q_{k,i} = -log q_{corr,i}$$

A keresztentrópia költségfüggvény egyik előnye, hogy nehezen értelmezhető „jóság” értékek helyett valószínűég jellegű értékekkel dolgozik, így a modell kimenete könnyebben felhasználható. Hátránya az SVM hibával szemben, hogy a költség értéke sosem lesz nulla, vagyis az SVM hiba „takarékosabb”: megelégszik a biztonságos elválasztással, és hagyja, hogy a modell a megmaradt erőforrásait többi tanító adat helyes osztályozására fordítsa. A gyakorlatban a két költségfüggvény közti különbség azonban alig kimutatható.

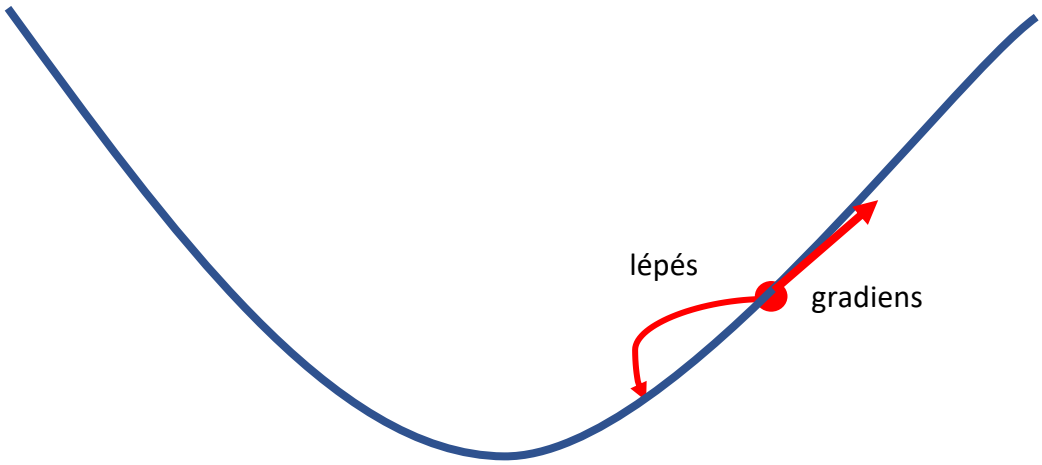
Mindkét hibafüggvénynek van azonban egy alapvető problémája. Könnyű ugyanis belátni, hogy mindkét költségfüggvény esetén, ha egyszerűen a modell aktuális súlymátrixát egy nagy számmal megszorozzuk, akkor a hibafüggvények értéke csökkenni fog, az osztályozás pontossága viszont változatlan marad, hiszen egyszerűen minden kimeneti jóság ugyanazzal a konstanssal szorzódik. Ennek következtében a súlymátrix normája minden határon túl növekedni fog, ami egyrészt numerikus problémákhoz, másrészt egy túl magabiztos modellhez fog vezetni.

Éppen ezért ezeket a hibafüggvényeket nem önállóan, hanem egy regularizációs büntetőtaggal együtt szoktuk használni, ami a súlymátrix normáját tartja kordában. Elterjedt megoldás a mátrixnak az L1, illetve az L2 normáját használni büntetőtagként. Létezik ezen felül még az úgynevezett elasztikus regularizáció, amikor a kétfajta norma súlyozott átlagát használják. A végső hibafüggvény a következőképp adódik:

$$L = \frac{1}{N}\sum_i L_i + \lambda R(W)$$

Ahol L_i az i-edik tanítóadatra számolt hiba, R a regularizációs tag, λ pedig a regularizáció relatív súlyát befolyásoló hiperparaméter. Érdemes megjegyezni, hogy a következő alfejezetben tárgyalt többrétegű hálók esetén a súlymátrix normájának növekedése túlillesztést okozhat, így ennek az elkerülése regularizáció.

A jelenlegi alfejezet utolsó kérdése az imént bemutatott hibafüggvények minimalizálására szolgáló optimalizálási módszerek tárgyalása. A perceptron súlyainak optimális értékére nem létezik zárt alakú megoldás, így iteratív optimalizálásra lesz szükség. Szerencsére a modell és a költségfüggvény is deriválható, így alkalmazhatunk gradiens alapú módszereket. Ha kiszámoljuk a hibafüggvény súlyok szerinti deriváltját (más szóval a súlyok gradiensét), akkor megkapjuk azt, hogy hogyan kellene a súlyoknak megváltozni ahhoz, hogy a hibafüggvény a lehető leggyorsabban növekedjen. Ha azonban a súlyokat a gradienssel ellenkező irányba változtatjuk, az a leggyorsabb csökkenés iránya lesz. Ezt a lépést egyfajta „fordított hegymászóként” ismételve egy idő után lokális minimumba jutunk.



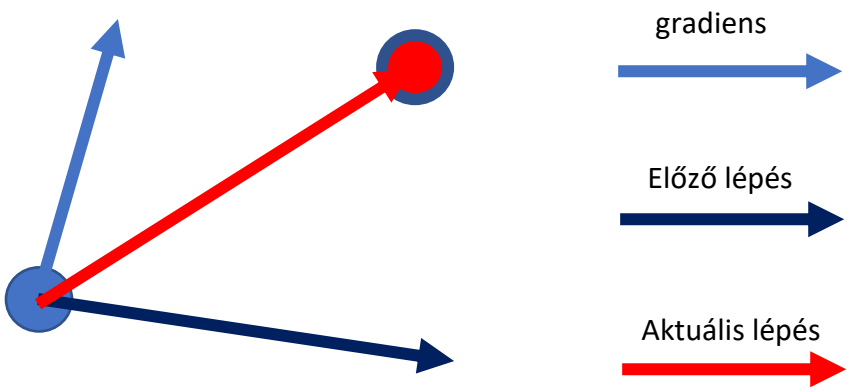
2. ábra: A gradiens módszer szemléltetése egy dimenzióban.

Vegyünk azonban észre, hogy mivel a teljes hibát szeretnénk minimalizálni, ezért minden egyes lépéskor ki kell az értékelni az egész tanító adatbázis hibáját. Mivel a gradiens módszer aránylag sok lépés után konvergál csak, azért ez nem praktikus. Éppen ezért a tanító adatbázist egyenlő méretű, véletlenszerűen kiválasztott részhalmazokra (minibatch-ekre) osztjuk, és minden egyes minibatch után végzünk egy lépést. Ezt a módszert sztochasztikus gradiens módszernek (SGD, Stochastic Gradient Descent) nevezzük. A minibatchek mérete általában kettő hatványa szokott lenni, implementációs okokból. Érezhető persze, hogy az egyetlen minibatch-re számolt gradiens nem egyezik teljesen meg az egész adatbázisra számolttal, de elég közel van ahhoz, hogy megközelítőleg a jó irányba haladjon a módszer. Mivel így egyetlen lépést sokkal olcsóbb végrehajtani, összességében jelentős gyorsulást érünk el. A gradiens módszer képlete a következő:

$$W_{k+1} = W_k - \frac{\alpha}{N_{MB}} \sum_{i}^{N_{MB}} \nabla L_i$$

Ahol α a tanulási ráta, ami egy olyan hiperparaméter, ami nagymértékben befolyásolja a tanítás sebességét és minőségét. Helyes megválasztásáról egy későbbi fejezetben beszélünk részletesen.

Fontos még észrevenni, hogy a gradiens módszer egyik hátránya, hogy könnyen beragadhat lokális minimumokba. Ennek megoldására az inverz hegymászó ötletét le kell cserélnünk a hegyről leguruló szikla képére. Más szóval élve a gradiens módszerhez egyfajta tehetetlenséget, momentumot veszünk hozzá. Ezt a gyakorlatban úgy tesszük, hogy az adott időpontban elvégzett lépés a negatív gradiens iránya és az eggyel korábbi időpillanatban tett lépés súlyozott átlagából tevődik össze. Ez a súly alkalmazástól függően általában a [0,1-0,9] tartományban mozog.



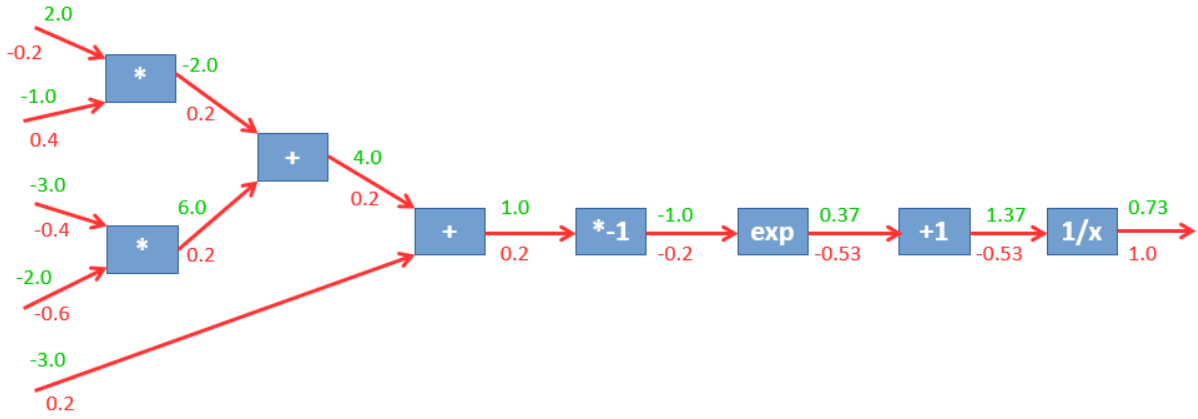
3. ábra: Az egyszerű momentum módszer szemléltetése.

Amint azt már korábban említettük, az egyszerű lineáris modellek képességei erősen limitáltak, így a képi bemenetek esetében ritkán használjuk őket. Ismertetésük azonban szükséges volt, ugyanis ezek az egyszerű lineáris modellek könnyedén összeépíthetők komplex nemlineáris tanuló eljárásokká. Amennyiben az előző alfejezetben ismertetett neuron modelleket egymás után csatoljuk, akkor egy többrétegű, előrecsatolt neurális hálózatot kapunk. A neurális hálózatoknak minden rétegéhez tartozik egy ismeretlen súlymátrix, amelyet a korábban ismertetett költségfüggvények és optimalizálási módszerek segítségével határozhatunk meg.

Az egyetlen kérdéses lépés a hibafüggvény súlyok szerinti deriváltjának számítása. Egy neurális háló elképzelhető, mint egy számítási gráf, ahol a gráf egyes csomópontjai egyszerű, analitikusan deriválható függvényeket implementálnak. Amennyiben egy számítási gráfban ismerjük a bemeneteket, és az egyes csomópontok által implementált függvényeket, akkor az összes csomópont kimenetét ki tudjuk számítani. Ezt nevezzük az előreterjesztés műveletének. Érdemes azonban észrevenni, hogy amennyiben ismerjük a csomópontok függvényeit, és ismerjük a számunkra érdekes mennyiség (ez esetben a hibafüggvény) deriváltját a csomópont kimenete szerint, akkor a deriválás láncszabályának segítségével könnyedén előállíthatjuk a csomópont bemenete és súlyai szerinti deriváltakat.

$$\frac{\partial L}{\partial x} = \frac{\partial f}{\partial x}\frac{\partial L}{\partial f}, \text{és } \frac{\partial L}{\partial W} = \frac{\partial f}{\partial W}\frac{\partial L}{\partial f}$$

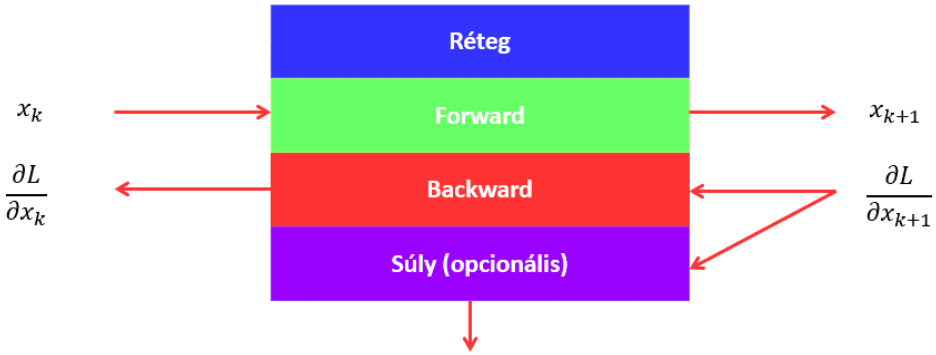
Ezzel a módszerrel a hálón hátrafele haladva az összes bemenet és súly gradiensét meg tudjuk határozni. Ezt a műveletet hátraterjesztésnek (backpropagation) nevezzük. Az egyetlen kérdés csupán, hogy hogyan kapjuk meg a hibafüggvény deriváltját a számítási gráf utolsó csomópontjának kimenete szerint. Vegyük észre azonban, hogy az előreterjesztés során legutolsó elvégzendő művelet pont a hibafüggvény kiszámítása, azaz a gráf utolsó pontjának a kimenete maga a hibafüggvény. Egy mennyiség saját maga szerinti deriváltja pedig triviálisan 1. Ily módon tehát minden rendelkezésre áll a háló gradienseinek számolására.



4. ábra: Példa számítási gráfra. Zold színnel az aktivációk, pirossal a kimenet adott érték szerinti deriváltja látható

1.2.Konvolúciós hálók

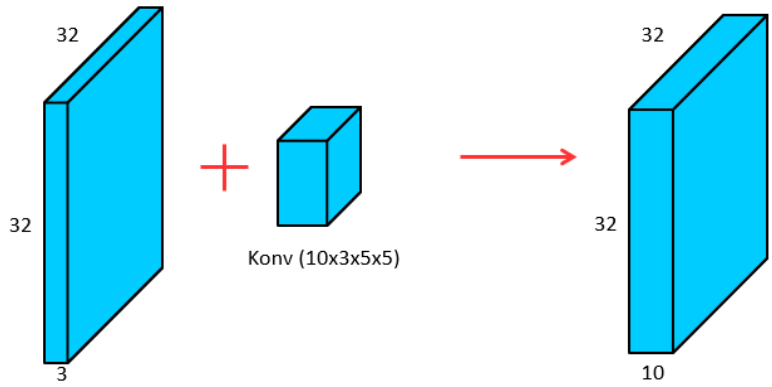
Az előző alfejezetben röviden ismertettük a többrétegű neurális hálók tanításának módját. Érdemes azonban észrevenni, hogy a neurális háló elemeinek nem feltétlenül kell a korábbi fejezetben megismert perceptron függvényeknek lenniük. A valóságban a neurális hálózatok számos különböző fajta rétegből állnak, melyeknek közös tulajdonságuk, hogy deriválható függvényeket valósítanak meg. Saját magunk is könnyedén készíthetünk új típusú rétegeket, egészen addig, amíg az előre- és hátraterjesztés részfeladatait elvégző függvényeket megvalósítjuk.



5. ábra: Általános réteg modell egy mély neurális hálóban.

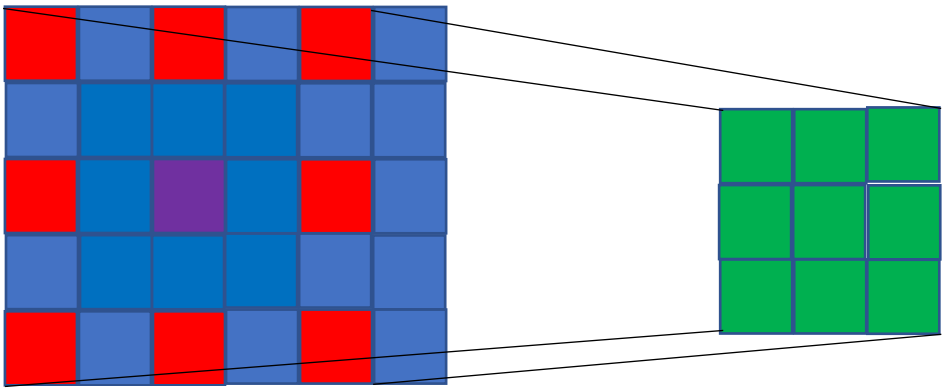
A korábban ismertetett lineáris neuron modell a számítógépes látásban használt hálókban ugyan előfordul, de tipikusan nem ilyen rétegekből épül fel a háló nagy része. Ennek oka, hogy a lineáris (más néven teljesen kapcsolt – fully connected) réteg minden bemenete és kimenete között létesít egy kapcsolatot, aminek következtében rengeteg paraméterrel rendelkezik, ami elősegíti a túlillesztés előfordulását, ráadásul a háló tárolását is megnehezíti. További hátránya, hogy a képek térbeliségét egyáltalán nem használja ki.

Ezekre a problémákra ad megoldást a konvolúciós réteg, amely nevéből adódóan a korábbi kötetben ismertetett konvolúciós szűrőkhöz hasonlóan működik. Egy konvolúciós réteg a bemeneti (általában 1-3 csatornás) képen N darab különböző konvolúciós szűrőt futtat végig, amelynek eredménye egy N csatornás szűrt kép. Az ezt követő konvolúciós rétegek már N csatornás bemeneti képpel dolgoznak. A használt szűrők mérete és a csatornák száma (más néven a réteg mélysége) tipikus hiperparaméterek. Érdemes megjegyezni, hogy a gyakorlatban a legtöbb esetben a kép térbeli méretének megőrzése érdekében a kép széleit 0-kal egészítjük ki.



6. ábra: Egy konvolúciós réteg struktúrája.

A konvolúciós rétegeknek még két fontos paramétere létezik: a stride, és a dilatáció. A konvolúciós szűrő egyes stride esetén minden pixelen végighalad, míg kettes stride esetén minden másodikon, és így tovább. Ezt a beállítást a kép térbeli méretének csökkentésére szokták használni. Az dilatáció értéke azt határozza meg, hogy a szűrőn eggyel arrébb található súlyt a képen hánnyal arrébb lévő pixellel szorozzuk. Egyes dilatáció értéke esetén a szűrő a megszokott módon viselkedik, egynél nagyobb érték esetén viszont egyre inkább „széthúzódik”. Ezt a beállítást arra szokták használni, hogy a konvolúciós szűrők által érzékelt képrészlet méretét növeljék.

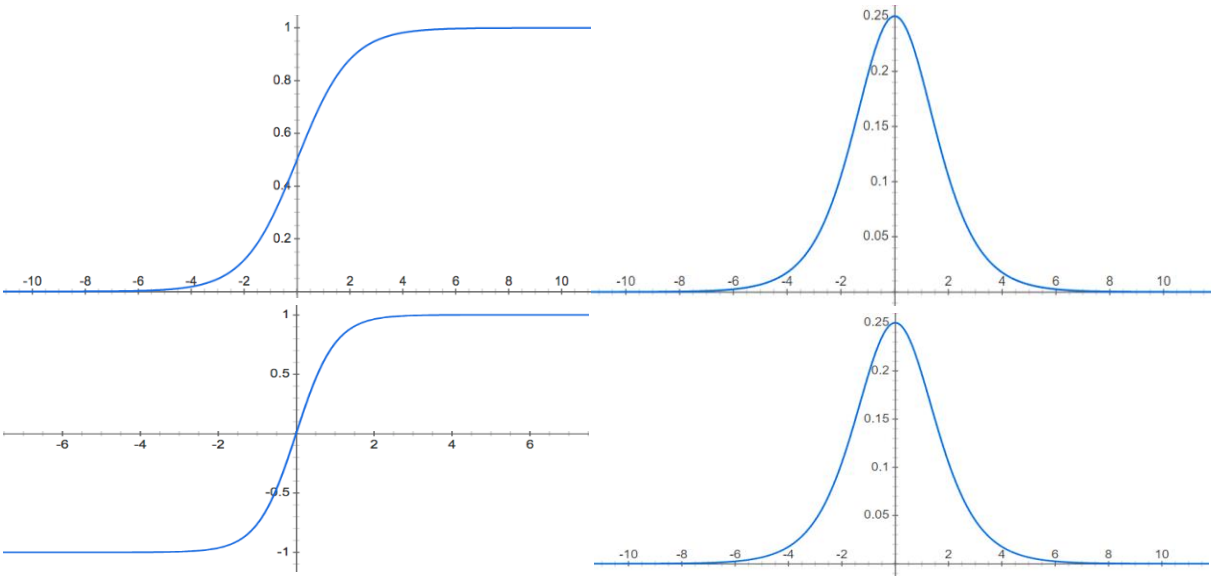


7. ábra: A dilatáció hatása: Egy 3x3-as szűrő 1 dilatáció melett a sötétkék, míg 2 dilatáció esetén a piros sznű pixelek értékeit használja.

Érdemes észrevenni, hogy amennyiben egymás után újabb és újabb konvolúciós rétegeket helyezünk el, egyre növekvő mélységgel, akkor egy idő után a rétegek kimenetén kapott aktivációs tömb mérete hatalmas lesz. Éppen ezért néhány rétegenként érdemes az aktivációs tömb térbeli méreteit csökkenteni. Az egynél nagyobb stride paraméterrel rendelkező konvolúciós réteg mellett ezt még az úgynevezett pooling operáció segítségével is megtehetjük. A pooling szintén egy csúszóablakos művelet, amely az aktivációs tömb éppen lefedett részét egyetlen számmal helyettesíti. A két leggyakoribb eset, amikor ez az érték az ablak által lefedett érékek átlaga vagy maximuma.

8. ábra: A pooling operátorok működése.

A többrétegű neurális hálók utolsó esszenciális rétege az aktivációs réteg, ami tipikusan minden konvolúciós és lineáris réteget követ. Ez a két réteg ugyanis lineáris műveleteket hajt végre, ezek kompozíciója is lineáris marad. Éppen ezért az egyes rétegek közé nemlineáris függvényeket ékelünk, be, amelyek az aktivációs tömb minden elemén függetlenül lefutnak. A hagyományos neurális hálók esetében népszerű választás volt a szigmoid, illetve a hiperbolikus tangens függvény. Ezen függvényeknek azonban közös hátránya, hogy az értelmezési tartományuk nagy részén a formájuk lapos, vagyis a deriváltjuk gyakorlatilag nulla. Ha sok ilyen deriváltat ékelünk a neurális hálóba, akkor a láncszabály értelmében előbb-utóbb a



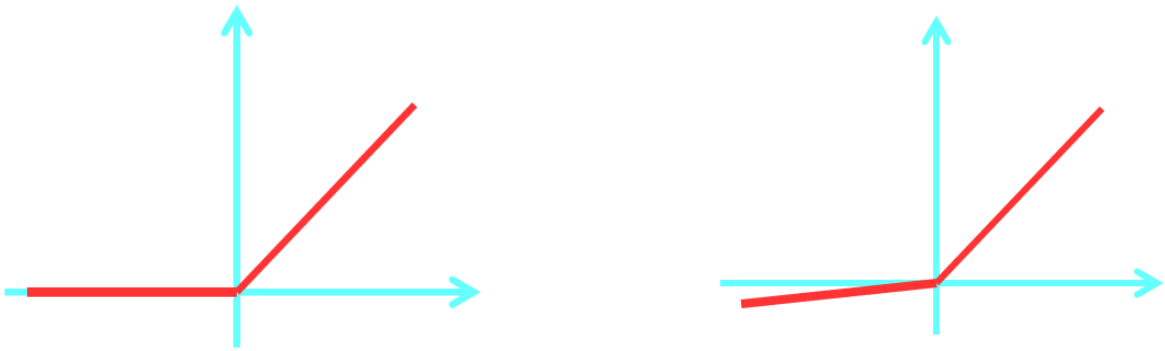
9. ábra: A szigmoid (felül) és a tanh (alul) aktivációk, és deriváltjaik a jobb oldalon

legtöbb deriváltat ki fogják nullázni. Ez a háló bemenethez közeli rétegeinek a beragadásához és a tanulás meghiúsulásához vezet.

A jelenlegi hálók esetén a legnépszerűbb aktivációs függvény a ReLU (Rectified Linear Unit), amely egy szakaszosan lineáris aktiváció, amely a negatív bemeneteket kinullázza, míg a pozitív tartományban nem fejt ki hatást. Ennek az aktivációnak a deriváltja a pozitív tartományban 1, a negatívban 0, így a deriváltakra kifejtett zavaró hatása lényegesen kisebb. Ennek ellenére ReLU használata esetén is előfordulhat az elülső rétegek beragadása, amelyre a paraméteres ReLU (PReLU), valamint a szivárgó (Leaky) ReLU adhat megoldást. Ezek annyiban különböznek az eredeti megoldástól, hogy a negatív tartományban nem nullázzák az aktivációkat, hanem egy egynél kisebb konstanssal szorozzák azokat. A szivárgó esetben ez a konstans egy hiperparaméter, míg a paraméteres esetben a gradiens módszer segítségével tanulható.

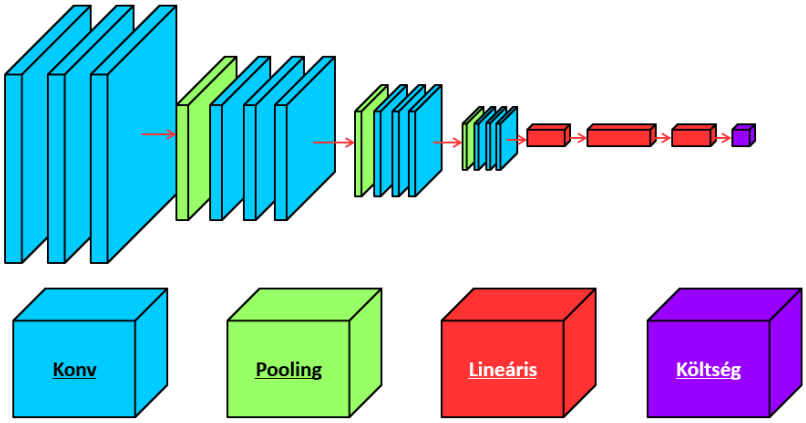
$$ReLU(x) = max(0, x) \qquad PReLU(x) = max(ax, x)$$

A jelen fejezetben bemutatott rétegeket tartalmazó neurális hálózatokat konvolúciós neurális hálózatoknak nevezzünk, melyeket első sorban a számítógépes látás területén alkalmaznak. A konvolúciós rétegek alkalmazása kifejezetten előnyös képi adatok esetén, mivel egy konvolúciós rétegnek a lineárishoz képest több nagyságrenddel kevesebb paramétere van. Egy konvolúciós neurális hálóban általában konvolúciós rétegek sorozata követi egymást (mindegyik kimenetén aktivációs függvénnyel), néhány konvolúciós rétegenként egy leskálázó réteg (konvolúció stride-dal, vagy pooling) közbe ékelésével. A háló végén tipikusan egy vagy több lineáris réteg állítja elő a végső kimenetet.



10. ábra: A ReLU (balra) és a PReLU (jobbra) aktivációk

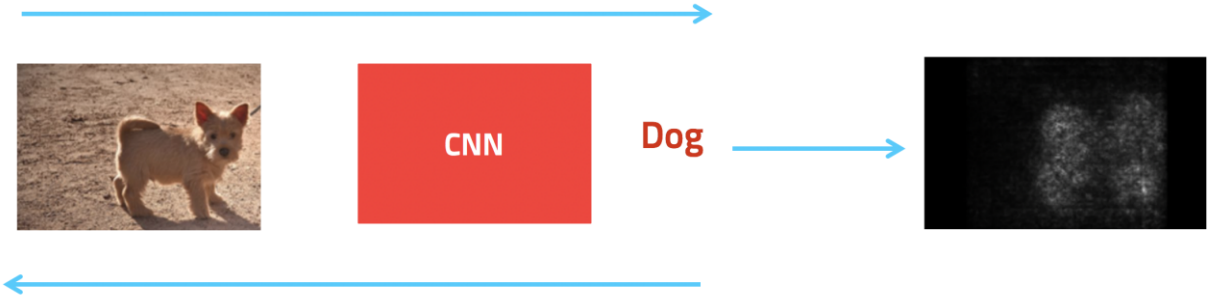
Érdemes elgondolkozni azon, hogy mit is csinál több egymással sorba kötött konvolúciós réteg. Egy konvolúció elképzelhető egy egyszerű jellemző detektorként, amely a bemeneteinek bizonyos kombinációira aktiválódik, míg másokra nem. Így az első konvolúciós réteg kimenetén kapott aktivációs térkép azt adja meg, hogy hol voltak olyan pixel kombinációk a képen, amik az egyes szűrőket aktiválták. A következő réteg bemenete azonban már ez az aktivációs térkép. Az ebben lévő szűrők tehát már nem a pixelek, hanem ezeknek az alacsonyabb szintű jellemzőknek bizonyos kombinációira aktiválódnak. Belátható ez alapján, hogy egy sokrétegű konvolúciós neurális háló kezdetben primitív képi jellemzők egyre bonyolultabb kompozícióit detektálni képes rétegeket tartalmaz a háló végső részeiben. Ez a szemlélet meglehetősen hasonlít az emberi látás kompozíciós jellegére.



11. ábra: Tipikus konvolúciós neurális háló felépítése.

1.3.Követés

Érdemes megjegyezni, hogy a backpropagation művelete valójában általánosságban felhasználható a háló két tetszőleges pontja közti derivált számítására. Ez azt jelenti, hogy nem csak a háló súlyai szerinti, hanem a bemeneti kép szerinti deriváltat is meghatározhatjuk. Ez lehetővé teszi annak meghatározását, hogy melyik pixelek befolyásolják egy osztály jóság értékét egy adott képen, ami felhasználható az objektumok szegmentálására, követésére, valamint arra is, hogy egy helytelen osztályozás esetén megtudjuk, hogy a képen melyik rész felelős a hibáért.



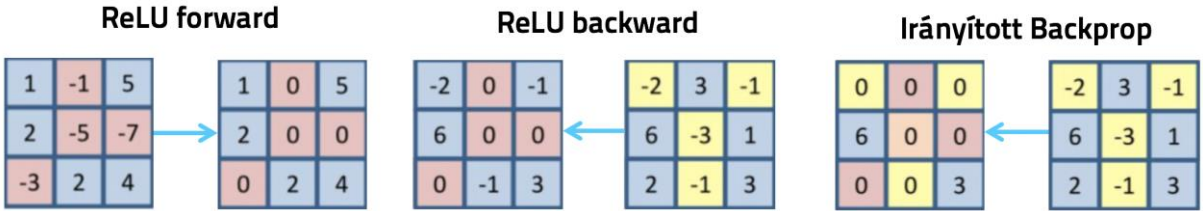
12. ábra: A salciency map készítésének elve.

Erre az első módszer az úgynevezett saliency (fontosság) térkép előállítása. Ehhez első lépésben az adott képet a neurális háló bemenetére adjuk, és kiválasztjuk a háló által legvalószínűbbnek vélt osztályt. Ezt követően az utolsó osztályozó réteg kimenetére előírjuk a gradiens vektor értékét, úgy, hogy minden eleme nulla legyen, kivéve a kiválasztott osztályhoz tartozót, amit egynek választunk. Ezt követően végrehajtjuk a backpropagation műveletét, melynek végén a legelső réteg bemenete (vagyis a kép) szerinti deriváltat is számítjuk. Az így kapott gradiens értékek abszolút értékét vesszük (hiszen minket csak az adott pixel hatásának nagysága érdekel) és a kép csatornái mentén a maximum értéket vesszük. Ezt küszöbözve előállítható egy bináris maszk.

A kép szerinti derivált minősége és a készített maszk pontossága azonban nagy mértékben javítható, ha az úgynevezett irányított (guided) backpropagation műveletét alkalmazzuk a hagyományos eljárás helyett. Ez az eredeti backpropagation egy módosított változata, amelyet

kifejezetten a kép szerinti derivált előállítására fejlesztettek ki. A módszer alapja egyszerű: a backpropagation műveletét ugyanúgy kell elvégezni, egyedül a ReLU nemlinearitások kezelése változik. Ez esetben először a ReLU backward műveletét végezzük el, vagyis a kimeneten kapott deriváltakat változatlanul átengedjük azokon a helyeken, ahol a ReLU bemenete pozitív volt, és kinullázzuk ott, ahol a bemenet negatív volt. Ezt követően az így kapott deriváltakra alkalmazzuk a ReLU függvényt, vagyis a negatív derivált értékeket is kinullázzuk.

A guided backpropagation követésen felül felhasználható arra, hogy megkapjuk azt a képet, ami a háló egy kiszemelt neuronját a lehető legerősebben aktiválja. Ezzel a vizualizációs módszerrel választ kaphatunk arra a kérdésre, hogy mit csinál egy adott neuron. Ilyen jellegű kérdések megválaszolása az gépi tanuló algoritmusok fekete doboz jellege miatt általában rendkívül nehéz.



13. ábra: A guided backpropagation művelete

1.4.Hasznos praktikák

A korábbi fejezetek során megismertünk a mély tanulás és a konvolúciós neurális hálók alapjait, valamint részletesen tárgyaltuk a sorozatok feldolgozására, valamint az osztályozásnál bonyolultabb látási feladatok elvégzésére létrehozott speciális struktúrákat. A mély tanulás azonban tipikusan azon területek közé tartozik, ahol a módszerek használata papíron rendkívül egyszerűnek tűnik, a gyakorlatban viszont számtalan nehézség adódik, melyek a módszerek használatát nehézzé teszik. A jelen alfejezet célja, hogy összeszedje azokat a gyakorlati megfontolásokat és praktikákat, amelyek nélkül rendkívül nehéz a való életben jól működő neurális hálókat létrehozni.

A neurális hálók tanítását alapvetően három dolog nehezíti meg:

1. A túlillesztés (overfitting) jelensége, amely a betanított modell való életben történő alkalmazhatóságát veszélyezteti
2. Numerikus problémák, melyek az optimalizáló algoritmus konvergenciáját hiúsítják meg
3. A hálók tanításához szükséges nagy mennyiségű címkézett adathalmaz előállításának problémája

A mély tanulásról szóló első alfejezetben bevezettük a gradiens módszert, ami a hibafüggvény deriváltjának segítségével iteratívan módosította a háló paramétereit a teljesítmény javításának érdekében. Arról viszont szándékosan hallgattunk, hogy hogyan kapjuk meg a kezdeti paraméter értékeket. A helyzet az, hogy a problémát helyesen megoldó paraméterekről kezdetben nem tudunk semmit, így nincs más választásunk, mint véletlenszerűen inicializálni őket. Az viszont egyáltalán nem mindegy, hogy milyen szórású véletlen számokkal végezzük ezt el (a nulla középérték nyilvánvaló módon adja magát). Ugyanis, ha a véletlen súlyok értéke

túl nagy, akkor a legtöbb aktiváció értéke is egyre nagyobb lesz, melynek következtében a gradiensek is rendkívül nagyok lesznek. Ez szemléletesen azt fogja eredményezni, hogy az optimalizálás során nem kis lépésekben fogjuk a hiba minimumát közelíteni, hanem hatalmas ugrásokkal fogunk a paramétertérben haladni, jó eséllyel teljesen átugorva a minimum helyét. Túl kicsi súlyok esetében néhány réteg után a háló aktivációi szinte minden közel nullák lesznek, melynek következtében a háló gradiensei is, így a háló a kezdeti értékekbe beragad.

Hasonló megfontolások miatt szükséges a neurális hálóknak bemenetként szolgáló képeken bizonyos transzformációk elvégzése. A korábbiak alapján belátható, hogy a jó numerikus konvergencia érdekében rendkívül fontos, hogy a bemenetre adott kép pixel értékeinek eloszlása megközelítőleg sztenderd normális legyen. Hiába inicializáljuk ugyanis jól a háló súlyait, ha a bemenet értékei túlságosan nagy számok, akkor hasonló problémába fogunk ütközni, mint a túl nagy súlyok esetén. Szintén fontos, hogy a pixelek átlaga a nulla közelében legyen, ugyanis csupa pozitív, vagy csupa negatív bemenet esetén numerikus problémák léphetnek fel.

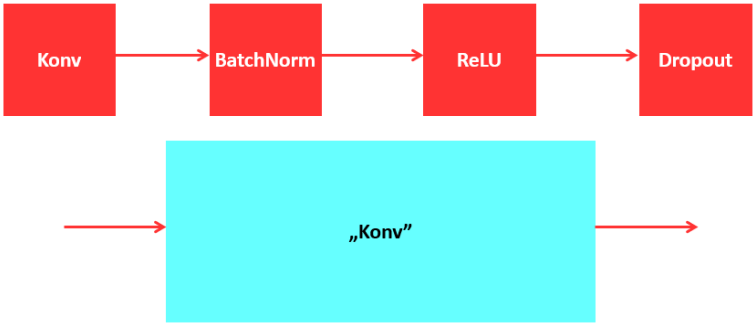
Érdemes megjegyezni, hogy a neurális hálók túlillesztésének jelensége szintén jellemezhető az egyes rétegek aktivációinak eloszlásával. A túlillesztés esetén ugyanis az történik, hogy a háló a be- és kimenetek közötti általános összefüggések helyett az egyes bemeneti tanítópéldákra adandó helyes választ kezdi el egyesével megtanulni. Ez tipikusan azt jelenti, hogy az egyes rétegekben minden egyes tanító adat esetén csak nagyon kevés aktiváció lesz maximális (amelyek épp az adott tanító példára emlékeznek), míg a többi aktiváció éppen az ellenkező véglet értékét veszi fel. Amint azt az imént beláttuk, ilyen „végletes” aktivációk akkor tudnak könnyedén előfordulni, ha az egyes réteges súlyai túlságosan nagyra nőnek. Ha visszaemlékezünk a korábban tárgyalt regularizációs módszerekre, azok pont a súlyok növekedését próbálták fékezni.

A nem kívánt aktivációk elkerülésére két további gyakran alkalmazott módszer létezik. Ezek közül az első a dropout nevű eljárás, melynek lényege, hogy a tanítás során minden egyes előreterjesztés során az egyes rétegek aktivációinak bizonyos hányadát véletlenszerűen kinullázzuk, és a további rétegek aktivációit így számoljuk tovább. Könnyen belátható, hogy ez a módszer meglehetősen csökkenti a túlillesztés mértékét, hiszen a hálót ezzel a módszerrel redundanciára kényszeríti. Fontos megjegyezni, hogy a tesztelés során a véletlenszerű törléseket már nem végezzük el, így viszont az egyes aktivációkat a dropout valószínűségének arányában skálázni kell.

A másik megoldás az úgynevezett batch normalizálás, aminek lényege, hogy az egyes rétegek után egy normalizáló műveletet végzünk el. Korábban említettük, hogy a tanítás során egyszerre nem egy képet, hanem egy úgynevezett minibatch-nek megfelelő (általában 32 többszöröse) képet értékelünk ki. A batch normalizálás műveletének lényege, hogy az egyes aktivációk átlagát és szórását a tanítás során folyamatosan számoljuk, és az egyes aktivációkat ennek segítségével normalizáljuk. Érdemes megjegyezni, hogy ez a művelet nem csak a túlillesztést mérsékeli az aktivációk eloszlásának normalizálásával, hanem a numerikus konvergenciát is javítja. A batch normalizálás képlete az alábbi:

$$x^1 = \frac{(x - \mu)}{\sigma} \qquad x_{out} = \alpha x^1 + \beta$$

Ahol μ és σ a becsült átlag és szórás, míg α és β tanult paraméterek. Érdemes megjegyezni, hogy modern neurális hálókban a batch normalizálás teljesen alapvető művelet, így általában minden konvolúciós réteget követ egy ilyen réteg. A batch normalizálás és a dropout akár együttesen is használható, bár a legtöbb kísérlet minimális teljesítménynövekedést mutat csak.



14. ábra: Egy általános „konvolúciós réteg” valójában ezen rétegek egymás után történő építéséből valósul meg. Batch normalizálás esetén a dropout alkalmazása opcionális

A túlillesztés jelenségének van még egy lehetséges elkerülési módja: gondoljunk bele, hogy a túlillesztés esetén a neurális háló a tanító adatbázis elemeire egyesével tanulja meg a helyes választ. Nyilvánvaló, hogy minél több tanító adat áll rendelkezésre, annál nehezebb ezt megtenni. Éppen ezért a tanító adatok számának növelése szinte minden esetben segít a túlillesztés mértékének csökkentésében. Tanító adatokat előállítani azonban rendkívül költséges, így ez a stratégia önmagában nem feltétlenül célravezető. Képek esetében azonban van lehetőségünk (részben) új tanítóadatok automatikus generálására, vagyis adat augmentációra. A módszer lényege, hogy tükrözés, véletlenszerű kivágás, forgatás, skálázás, intenzitástranszformációk segítségével mesterségesen növeljük az adatbázis méretét. Könnyen belátható, hogy ezek a műveletek a képek címkéjét nem befolyásolják, így büntetlenül elvégezhetők. Fontos megjegyezni, hogy a batch normalizálás, regularizáció és adat augmentáció módszereit együtt használjuk.

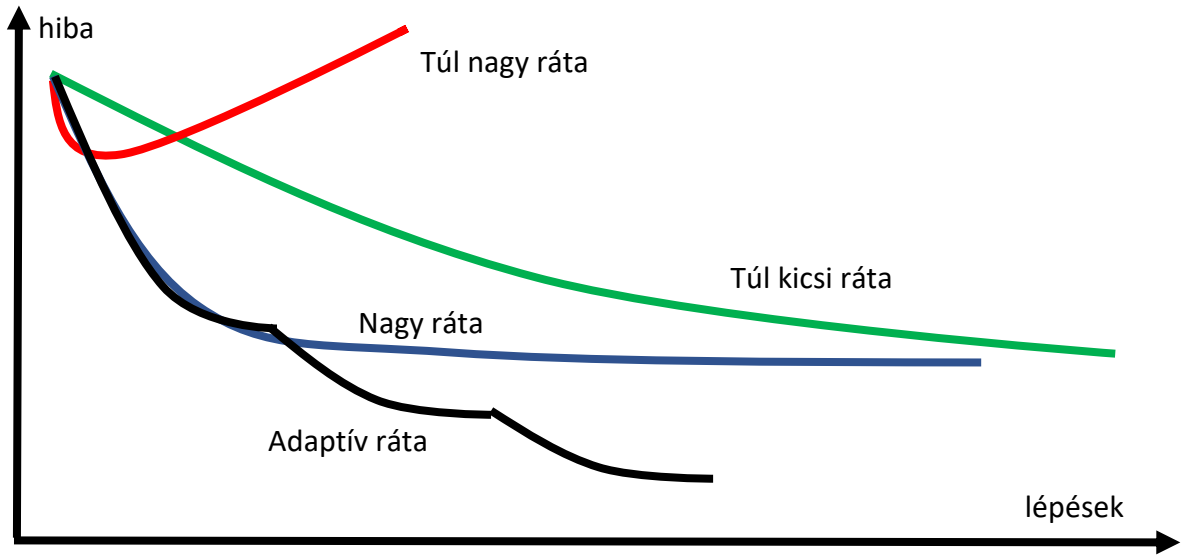
A neurális hálók tanításának további nehézsége, hogy egy tipikus tanítás során több tucat hiperparaméter is rendelkezésünkre állhat, melyek megfelelő értékének megválasztására nincs a próbálkozásnál jobb módszerünk. Egy neurális háló tanítása azonban meglehetősen sokáig tarthat (néhány órától akár néhány hétig is), így minden egyes próbálkozás rendkívül költséges. Ezért a tanítás kezdetén számos hiperparaméter megközelítőleg helyes értéke megválasztható úgy, ha a tanítást a teljes adatbázisnak csak egy kis részén végezzük el. Ez a módszer az esetleges programhibák felderítésében is segítségünkre lehet.

A teljes adatbázison történő tanítás során általában már csak néhány hiperparaméter értékét kell egy aránylag szűk tartományon belül meghatározni. Ekkor célszerű lehet ezeket a tartományokat egy egyenletes rácsra osztva az egyes rácspontokban különböző tanításokat végezni, majd ezeket összehasonlítani. Ennél azonban sokkal célszerűbb, ha az előbbi módszerrel megegyező mennyiségű véletlen hiperparaméter kombinációkkal végezzük a tanítást. Ekkor ugyanis minden hiperparaméter esetében nagyobb felbontáson mérjük az adott paraméter hatását. Ez különösen abban a gyakori esetben hasznos, amennyiben a

hiperparaméterek közül az egyik sokkal nagyobb mértékben befolyásolja a tanítás minőségét, mint a többi.

A tanítás hiperparaméterei között külön tárgyalást igényel a gradiens módszer lépéseinek nagyságát meghatározó tanulási ráta. A gradiens módszer során a hibafüggvény által képzett „völgy” legmélyebb pontjába szeretnénk a legmeredekebb csökkenés irányába tett lépések sorozatával eljutni. Amennyiben a lépések mérete túlságosan kicsi, akkor csak nagyon sok lépés után jutunk be a völgybe. Ha azonban a lépések mérete túl nagy, akkor ugyan gyorsan eljutunk a legmélyebb pont közelébe, az utolsó lépéssel azonban átlépünk a völgyön, és onnantól kezdve az idők végezetéig a völgy két oldala között fogunk „pattogni”. Sőt óriási lépésméret esetén előfordulhat, hogy akkorát ugrunk a völgy közepe felé, hogy annak ellenkező oldalán magasabb helyre lépünk, mint korábban voltunk. Ha ezt ismételgetjük, akkor minimalizálás helyett kimászunk a völgyből.

A gyakorlatban ezen megfontolások miatt nem egyetlen tanulási rátát szokás alkalmazni. E helyett a tanítás kezdetén a legnagyobb olyan tanulási rátával indítjuk az optimalizálást, amivel a hiba értéke stabil csökkenést mutat, így a lehető legyorsabban jutunk az optimum közelébe. Egy idő után a ráta értékét csökkentjük, így engedve, hogy a valódi optimum pozícióját minél jobban megközelítsük. Ez felfogható egyfajta durva optimalizálási és finomhangolási lépésként. A tanulási ráta állítására sok lehetséges módszer létezik, melyek közül az egyik leggyakoribb a ráta fix faktorral történő csökkentése bizonyos számú lépés után.



15. ábra: Különböző tanítási ráta választások hatása a tanítás konvergenciájára.

Lehetőség van természetesen a ráta adaptív állítására a tanulási sebesség folyamatos monitorozása által. Ekkor a rátát akkor csökkentjük egy fix faktorral, amennyiben a tanulás már bizonyos számú lépés óta nem tudta a korábbi legjobb eredményét meghaladni. Szintén hatásos módszer a koszinuszos lágyítás alkalmazása, ami a tanulási rátát egy maximum és minimum érték között egy koszinusz függvény első fél periódusának megfelelően állítja. A koszinusz lágyítás alkalmazásakor a félperiódus befejezésekor tovább lehet folytatni a tanítást a maximális tanulási értéket használva és újabb lágyítást végezni. Ennek értelme, hogy a hirtelen

megnövelt tanulási ráta „kilöki” a háló súlyait a lokális minimumból és a további tanulás során egy közeli jobb lokális minimum megtalálását teszi lehetővé.

A neurális hálók tanításának harmadik nehézsége a nagy számú címkézett tanítóadat előállítása. A mély tanulás területének egyik legjelentősebb áttörése ezt a problémát orvosolja. Beláttuk ugyanis, hogy a konvolúciós neurális hálók első konvolúciós és leskálázó rétegekből álló része különböző képi jellemzők detektálását végzi el. Ahogy előrefele haladunk a háló rétegei között úgy ezek a jellemzők egyre komplexebbé, egyre feladatspecifikusabbá válnak. Ebből következik azonban, hogy ha már van egy valamilyen feladatra betanított hálózatunk, akkor annak elülső rétegei felhasználhatók egy másik, hasonló feladat elvégzésére. Így elegendő az új feladathoz csak a háló utolsó rétegeit újra tanítani, amihez lényegesen kevesebb adat elegendő, hiszen kevesebb szabad paramétert tartalmaznak, mint az egész háló. Ezt a technikát transzfer tanulásnak nevezzük, és elterjedt megoldás a mély tanulás területén. A fent ismertetett érvelés annyira igaz, hogy számos esetben elegendő a hálók legutolsó, lineáris rétegét újra tanítani. Más esetekben szükség lehet az elülső hálórészek finomhangolására, azonban ehhez is nagyságrendekkel kevesebb adat elegendő lehet.

2. A mérés környezete

A mérés során a *PyCharm* elnevezésű IDE áll rendelkezésre, amely rendkívül sokoldalú szolgáltatásokkal könnyíti meg a szoftverfejlesztést, például konfigurálható automatikus formázási lehetőségek állnak rendelkezésünkre. További részletekért érdemes lehet a JetBrains ide vonatkozó weboldalát felkeresni (https://www.jetbrains.com/help/pycharm/quick-start-guide.html). Függvények, objektumok esetében a ***Ctrl+P*** billentyűkombináció pop-up segítségként szolgálva mutatja nekünk a paramétereket.

A fejezet elsődleges célja egyrészt a Python nyelv, másrészt pedig a PyTorch nevű, mesterséges intelligencia-keretrendszer rövid összefoglalása a mérést megkönnyítendő.

2.1. A Python nyelv

A Python programnyelv egy ***interpretált szkriptnyelv*** (a saját bytekódját hajtja végre), mely alapvetően az *objektumorientált* paradigma által vezérelve került kialakításra. Jelenleg kétféle, kismértékben eltérő főverziója érhető el, ezek közül a mérésen a 3-as verziót használjuk majd.

Az OOP szemlélet olyannyira központi szerepet tölt be Pythonban, hogy minden változó objektumnak tekinthető – ami azt jelenti, hogy az integer és float típusok is objektumok. A nyelv számára jelentős hátrány adatintenzív számítási feladatok elvégzésekor ennek ténye, ugyanis arról beszélünk, hogy a Python ***nem rendelkezik natív számtípussal.*** Nem véletlen, hogy a hatékony modulok alacsonyabb szintű nyelveken nyugszanak, így a következő alfejezetben ismertetett PyTorch is.

További nem szokványos jellemzője a nyelvnek, hogy ***nem erősen típusos***, a változónevekhez *futási időben rendelődik hozzá* a referált objektum (azaz egy ***változónév*** igazából egy referens, vagyis adott ***példányra mutató referenciát*** tartalmaz). Habár Pythonban a referenciák teljesen úgy viselkednek, mint a változók (vagyis semmilyen szintaktikai kiegészítésre nincs szükség, még argumentumok átadása esetében sem, mint ahogy azt C++-ban láttuk). *(A félreértések elkerülése végett fontos szem előtt tartani, hogy amennyiben a segédletben a továbbiakban változó szerepel, akkor is igazából referencia van a háttérben, ezért hangsúlyozott általában, hogy nem a változó, hanem a változónév tartalmazza az objektumra mutató referenciát.)*

Vagyis lehetőség van arra, hogy egy referenshez (változónévhez) a programban különböző típusú objektumokat rendeljünk hozzá – ez teljesen logikusnak tűnik, ha belegondolunk abba, hogy a „Pythonban minden objektum” korábbi kijelentés lényegében azt sejteti, hogy minden az objektum ősosztály leszármazottja. A referenciák kezelése referenciaszámláló segítségével történik – a koncepció analóg a például C++-ban megtalálható *shared_ptr* típus esetén használttal.

Pythonban nincsen továbbá pointer sem, az ***argumentumok*** átadása ***referencia*** szerint történik. Itt azonban meg kell különböztetnünk az objektumokat az alapján, hogy módosíthatóak-e. A következőket érdemes alaposan átgondolni, különben hibás működésű programot kaphatunk.

Kétféle objektumtípus létezik, ***módosítható*** *(mutable),* ill. ***nem módosítható*** *(immutable).* Nem módosíthatóak többek között az egyszerű adattípusok *(POD, plain old data),* mint az egész vagy lebegőpontos számok, valamint a sztringek. Habár a gondolatmenet e megfontolás mögött elsőre furcsának tűnik, a következőképpen magyarázható: minden adott szám vagy szöveg egyedi, vagyis ha pl. két változóhoz hozzárendeljük az 5 értéket, akkor a kettő tartalma egymással teljesen azonos, ha az egyiket meg szeretnénk változtatni, akkor az már egy másik objektum lesz: nem egy tagváltozót módosítunk, hanem magát az objektumot teljes mértékben és kizárólagosan azonosító elemet. Talán érdemes a fordított programnyelvekből egy szemléletes példát átgondolni: a fordító ugyanolyan értékeket helyettesít be a gépi kódba (érdeklődők számára: a Pythonnak esetében is elérhető egyfajta disassembly a *dis* modul segítségével).

Módosítható lényegében minden egyéb típus, így a Pythonba beépített konténer jellegű típusok, mint *list, dict, tuple* (ezek különlegessége, hogy nem csak egy típust képesek egyidejűleg magukba foglalni, hanem bármilyen objektumot), de a saját osztályok példányai is ide tartoznak. Ebben az esetben a módosítás nem eredményezi új objektum létrehozását.

Ezek a különbségek ***objektumok másolása*** esetében is jelentkezik. Fontos különbség, hogy Pythonban alapvetően a hozzárendelés *(assignment)* igazából C++-szemszögből inkább a *copy constructor* hívásának feleltethető meg. ***Mutable*** esetben az eredeti objektumhoz tartozó referenciszámláló kerül megnövelésre, ***módosítás*** esetén pedig ***új objektum*** jön létre, értelemszerűen ugyanarra az értéket tartalmazó változók egyikének megváltoztatása nem hat ki a többire. ***Immutable*** esetben azonban nem ilyen egyszerű a helyzet: mivel alapvetően referenciákat tartalmaznak a változónevek, amelyek módosítható objektumokat referálnak, így ugyanarra a példányra mutató ***referenciák bármelyikének módosítása*** változást eredményez a referált egy darab objektum esetében, vagyis ***bármelyik változóval*** hivatkozunk rá, a ***változás*** mindegyik esetben ***látható*** lesz számunkra. Az ilyen jellegű másolást shallow copy-nak szokás nevezni, melynek párja a *deepcopy* (elérhető a *copy* modulban), ami Pythonban immutable esetben is új példányt hoz létre, így az új objektum független lesz a többitől.

Paraméterek átadása hozzárendeléssel történik, ez azonban nagy objektumok esetében sem okoz komolyabb problémát, ugyanis a ***referenciák*** kerülnek csak ***másolásra***. Ez a láthatóságra a következőképpen van hatással: ha a paraméter ***immutable*** és a ***függvénytörzsben módosításra*** kerül, akkor lényegében a függvény scope-jában egy ideiglenes objektum kerül létrehozásra, a függvényből ***visszatérve*** a **változó** megtartja ***eredeti értékét***. ***Immutable*** esetben, mivel a *referencia* kerül *másolásra*, az objektumok másolásánál láttuk, hogy az általuk referált objektumok *módosítása érvényes*, *bármelyik referenciájával hivatkozunk* is rá, így a függvény ***visszatérését*** követően az ***objektum*** már ***módosult*** értékével használható.

Rendkívül hasznos tulajdonság, hogy a Python gyakorlatilag ***bármennyi visszatérési értéket*** támogat.

A nyelvi koncepciók ismertetése után a következőkben a szintaktikai részletek kerülnek összefoglalásra.

Pythonban a ***programkód tagolása indentálással*** történik, vagyis a kódblokkokat egy tabulátorral beljebb kell kezdeni (ha valamilyen okból üres függvényt, ciklust, stb, kívánunk írni, akkor is kell egy indentált blokk, ezt egy sorban, a *pass* utasítással valósíthatjuk meg, ami nem hajt végre semmilyen műveletet).

Modulok betöltésére az *import* utasítással van lehetőségünk, mégpedig *kétféle* módon: importálhatjuk a teljes modult, ekkor a modul minden osztálya/függvénye a *modul neve után írt „.” (pont)* operátorral érhető el, ha *from*-ot használunk, lehetőségünk van csak egyes elemeket betölteni, ekkor *a modul nevét nem kell* az importált elem neve elé *kiírni*.

```
import module
from module import MyClass, my_func
```

Függvények a következő módon hozhatók létre:

```
def func(x):
    x +=1
    print("x = ", x)
```

A *def* kulcsszó után a függvény neve, majd a paraméterlista kerül megadásra, azt követően pedig az indentált függvénytörzs következik.

Osztályok esetében sem bonyolult a konstrukció:

```
class my_class(object):
    def __init__(self):
        self.x = 5
```

Pythonban a ***konstruktort*** az ***__init__*** (2-2 aláhúzással) rutin testesíti meg, mint látható, a tagfüggvények is majdnem teljesen megegyeznek az általános függvényekkel, azzal a különbséggel, hogy az ***első argumentum*** mindenképpen az ***adott példányra*** vonatkozik (mint ahogy a *this* C++-ban) – ezt konvenció szerint *self*-nek szoktuk nevezni.

Öröklés esetén nincs más teendőnk, mint az *object* osztály helyett megadni az általunk választott ősosztályt, majd a konstruktorban meghívni az ősosztály konstruktorát a ***super, általánosan az ősosztályra*** használható objektum segítségével.

```
class base_class(object):
    def __init__(self):
        print("I am Groot")

class inherited_class(base_class):
    def __init__(self):
        super().__init__()
        print("I am inherited")
```

További példaprogramok és kódrészletek találhatók az alábbi címen: https://gist.github.com/search?utf8=%E2%9C%93&q=user%3Arpatrik96&ref=searchresults

Aki esetleg mélyebben érdeklődik a Python nyelv iránt, annak érdemes lehet felkeresnie a következő TMIT SmartLab blogjának következő bejegyzéseit (angolul) :

https://medium.com/@SmartLabAI/python-under-the-hood-tips-and-tricks-from-a-c-programmers-perspective-01-b5f96895663

https://medium.com/@SmartLabAI/b52675c7c0af

2.2.PyTorch

A PyTorch a Facebook Research által fejlesztett Deep Learning keretrendszer, melyet a mérés során használni fogunk, így a következőkben röviden áttekintésre kerülnek a fontosabb koncepciók. A PyTorch egyik előnye a TensorFlow-val, Keras-szal szemben, hogy a hálózatot leíró gráf dinamikusan kerül létrehozásra, nincs fordítási szakasza a tanításnak – ez a kevesebb lépés mellett akár azt is lehetővé teszi számunkra, hogy a háló rétegeit tanítás közben változtassuk (mert az egyik hiperparaméterünk lehet a lineáris rétegek száma).

A PyTorch alapja az akár GPU-n is futtatható ***tenzor*** típus, mely egy pratikus tagfüggvényekkel felvértezett, ***multidimenzionális tömböt megvalósító osztály*** – tehát felettébb alkalmas például képek tárolására is.

Tenzorokon alapul a rétegeket/hálózatokat megvalósító *Module* osztály, melyből minden saját hálózatunknak örökölnie kell, így kerülnek regisztrálásra az előre-, ill. visszaterjesztést megvalósító függvényeink. Szerencsére nem kell a hibavisszaterjesztést kézzel megírni, a PyTorvh beépített ***automatikus differenciálásra*** képes modulja (*autograd*) megteszi ezt helyettünk.

Fontos megjegyezni, hogy az automatikus differenciálás *nem szimbolikus differenciálás* (azaz nem az a célja, hogy paraméteresen felírja egy bonyolult összefüggés deriváltfüggvényét) és *nem is numerikus differenciálás* (tehát az sem cél, hogy iteratívan közelítsünk egy értéket, mert a kvantizálás/kerekítés jelentős eltérésekben nyilvánulhat meg). Automatikus differenciálás során egy ***számítási gráf*** segítségével, a ***láncszabályt alkalmazva*** vagyunk képesek, viszonylag jól skálázhatóan, parciális deriváltakat kiszámítani, ami feltétlenül szükséges gradiens-alapú tanulóalgoritmusok esetében.

A PyTorch *optim* nevű almodulja tartalmazza a különböző, a tanítás során használt algoritmusokat, ezek közül nekünk most csak az SGD-re *(stochastic gradient descent)* lesz szükségünk. Az SGD objektum számára megadható a modell, a tanulási ráta (learning rate), a momentum, ill. van lehetőség normán alapuló regularizáció konfigurálására is.

Ahhoz, hogy tanítani tudjunk egy hálózatot, a következő struktúrájú programot kell megvalósítanunk és iteratívan meghívnunk. A következőkben a PyTorch egyik kiegészítésének (Ignite) forráskódjából látható egy részlet, ami tömören magába foglalja a szükséges lépéseket – a mérésen ugyanezen lépések segítségével, de egy kicsit más struktúrában fogjuk a tanítást végezni.

```
    def _update(batch):
```

```
        model.train()
        optimizer.zero_grad()
        x, y = _prepare_batch(batch, device=device)
        y_pred = model(x)
        loss = loss_fn(y_pred, y)
        loss.backward()
        optimizer.step()
        return loss.item()
```

Az alábbi hivatalos dokumentációban megtalálható példa tanulmányozása világosan mutatja, hogy a lépések megegyeznek (ebben már a validáció, ill. az adatok betöltése is megtalálható).

https://github.com/pytorch/examples/blob/master/mnist/main.py

Részleteiben a következő történik: tudatnunk kell a programmal, hogy most ***tanítjuk a hálót*** (hogy pl. a dropout csak ilyen esetben kerüljön aktiválásra). Ezt követően a ***gradiensek*** előző értékét kell ***nullláznunk*** (ugyanis megtehetnénk, hogy valamihez még felhasználjuk, ezért nem történik automatikusan). Ahhoz, hogy a következő előreterjesztést véghez vihessük, ***elő kell készíteni*** a következő ***minibatchet*** (ezt a kódban egy saját függvény, _prepare_batch jelöli). Az ***előreterjesztés*** végrehajtása felettébb egyszerű, magának a ***modellnek adjuk át*** argumentumként a ***minibatchet***. A ***hiba kiszámítása*** a következő lépés, ezt az általunk konfigurált hibafüggvény segítségével tehetjük meg, majd a hiba *backward* tagfüggvényét meghívva végrehajtjuk a ***hibavisszaterjesztést***. Nagyon fontos az utolsó lépés az ***optimalizáló algoritmus*** (SGD) ***lépésének*** végrehajtása, így kerülnek a ***súlyok frissítésre***.

Számos gyakran használt adatbázis elérhető a *torchvision* modulból, ezek előkészítésére, ill. adat augmentációra is használható transzformációk is rendelkezésünkre állnak. Ezen felül van lehetőség *DataLoader* objektumok segítségével (*torch.utils.data* modul) az adatok betöltését automatizálni. Továbbá lehetőségünk van például a tanulási ráta automatikus megváltoztatására adott szabályrendszer szerint.

Hivatalos dokumentáció: https://pytorch.org/docs/stable/index.html

Példák: https://pytorch.org/tutorials/index.html

2.3. OpenCV

Az OpenCV egy nyílt forráskódú számítógépes látás algoritmusokat tartalmazó függvénykönyvtár. Az OpenCV elsődleges nyelve a C++, azonban elérhetőek hozzá hivatalos wrapperek többek között Java és Python nyelven. Az OpenCV rengeteg hivatalosan támogatott algoritmust tartalmaz, melyen felül a külön letölthető Contrib modulban harmadik felek által kifejlesztett további funkciók is elérhetők. Az OpenCV mérésben használt verziójának dokumentációja elérhető itt: https://docs.opencv.org/3.4.2/index.html

3. Mérési feladatok

A mérés folyamán az alábbi feladatokat kell elvégezni:

1. Készítsen egy egyszerű néhány rétegből álló neurális háló architektúrát.
2. Készítsen eljárást egyszerű neurális háló tanítására a CIFAR 10 adatbázison. Vizsgálja meg az egyes hiperparaméter választások hatását!
3. Egy választott hiperparaméter kombináció segítségével végezzen tanítást a CIFAR 10 adatbázison egy mély, DenseNet architektúrájú hálóval.
4. Implementáljon követő algoritmust egy előre tanított neurális hálózattal, a guided backpropagation módszer felhasználásával.

1. táblázat: A készítendő háló rétegei

Típus	**Csatornák száma**	**Méret**	**Stride**
Conv	planes	3	1
Conv	planes*2	3	2
Conv	planes*2	3	1
Conv	planes*4	5	4
Conv	planes*4	3	1
Conv	planes*8	5	4
Osztályozó	10	1	1

2. táblázat: A vizsgálandó hiperparaméter kombinációk

Planes	**LR**	**Momentum**	**Weight Decay**	**Batch Size**
8	0.1	0.9	1e-4	128
8	0.01	0.9	1e-4	128
8	0.1	0	1e-4	128
8	0.1	0.9	1e-4	32
16	0.1	0.9	1e-4	128
16	0.1	0.9	0	128

3. táblázat: A densenet tanításhoz használandó hiperparaméterek:

Epoch szám	100
Scheduler lépésszám	25
LR	0.1
Momentum	0.9
Weight Decay	1e-4

4. Hasznos kódrészletek

Különböző rétegek létrehozása PyTorch környezeteben

```
self.conv = nn.Conv2d(inplanes,planes,size,padding=size//2,stride=stride)
self.bn = nn.BatchNorm2d(planes)
```

Rétegek meghívása

```
out = F.relu(self.bn(self.conv(x)))
```

Térbeli dimenziók eldobása az osztályozó réteg után

```
out = out.squeeze()
```

Adat augmentációs függvények

```
transforms.RandomCrop(32, padding=4),
transforms.RandomHorizontalFlip(),
transforms.ColorJitter(brightness=0.25, contrast=0.25, saturation=0.25,
hue=0.2),
transforms.ToTensor(),
transforms.Normalize((0.49139968, 0.48215827, 0.44653124),
                     (0.24703233, 0.24348505, 0.26158768))
```

Különböző augmentációs módszerek összefűzése

```
transform = transforms.Compose( <list of transforms> )
```

Adatbázis létrehozása

```
trainSet = torchvision.datasets.CIFAR10(root=root, download=True,
                                        train=True/False,
transform=transform)
```

Adat betöltő létrehozása

```
trainLoader = torch.utils.data.DataLoader(trainSet, batch_size=128,
shuffle=True, num_workers=2)
```

Költségfüggvény létrehozása

```
criterion = nn.CrossEntropyLoss()
criterion = nn.MultiLabelMarginLoss()
```

Optimalizáló módszer létrehozás

```
optimizer = optim.SGD(net.parameters(), lr=0.1, momentum=0.9,
nesterov=True, weight_decay=1e-4)
```

Tanulási ráta ütemező létrehozása

```
scheduler = lr_scheduler.StepLR(optimizer,10)
```

Neurális háló módjainak állítása

```
net.train()
net.eval()
```

Progress bar készítése

```
# Create progress bar
bar = progressbar.ProgressBar(0, len(trainLoader), redirect_stdout=False)

# Inside the loop:
bar.update(i)

# Upon finishing:
bar.finish()
```

Tipikus tanításra használt epoch

```
# Epoch loop
for i, data in enumerate(trainLoader, 0):

    # get the inputs
    inputs, labels = data

    # Convert to cuda conditionally
    if haveCuda:
        inputs, labels = inputs.cuda(), labels.cuda()


    # zero the parameter gradients
    optimizer.zero_grad()

    # forward + backward + optimize
    outputs = net(inputs)
    loss = criterion(outputs, labels)
    loss.backward()
    optimizer.step()
```

Osztályozási pontosság számítása az adott minibatch esetén

```
# Compute cumulative loss
running_loss += loss.item()

# Get class with max probability
_, predicted = torch.max(outputs, 1)

# Update number of total and correctly classified images
total += labels.size(0)
correct += predicted.eq(labels).sum().item()
```

Tanulási ráta ütemező léptetése

```
Scheduler.step()
```

Modell mentése

```
Torch.save(net, root + '/model.pth')
```

OpenCV kép PyTorch tenzorrá történő konvertálása

```
# BGR to RGB
img2 = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

# Convert to PIL image
pil_im = Image.fromarray(img2)

# Apply transform function, unsqueeze and convert to cuda
imageT = transform(pil_im).unsqueeze(0).cuda()
```

Gradiens számítása az adott változóra

```
imageT.requires_grad_(True)
```

Guided backpropagation meghívása

```
guided_grads = GBP.generate_gradients(imageT, classInd)
```

Gradiensek abszolút értékének számítása és csatornánként történő összegzése

```
gradImg = np.sum(np.abs(guided_grads), axis=0)
```

Tenzor konvertálása OpenCV képpé

```
gradImg = grad2OpenCV(gradImg)
```

Küszöbözés

```
_,binImage = cv2.threshold(gradImg,threshVal,255,cv2.THRESH_BINARY)
```

Kontúrok keresése

```
_, contours, _ =
cv2.findContours(binImage,cv2.RETR_EXTERNAL,cv2.CHAIN_APPROX_SIMPLE)
```

Kontúr területének számolása

```
area = cv2.contourArea(cont)
```

Kontúr rajzolása

```
cv2.drawContours(contImage,contours,index,255,-1)
```

5. Ellenőrző kérdések

1. Ismertesse röviden a Hinge és a Kereszt-entrópia költségfüggvényeket! Mi az előnyük/hátrányuk?
2. Hogyan működik a gradiens módszer? Milyen fontos kiegészítései vannak?
3. Mire való a backpropagation? Hogyan működik?
4. Ismertesse egy konvolúciós háló és gyakori rétegei felépítését!
5. Mi az a saliency? Hogyan állítható elő? Mi a guided backpropagation? Mi értelme van?
6. Milyen módszerekkel kerülhető el az overfitting? Ismertesse ezeket egy-egy mondatban!
7. Milyen módszerekkel javítható egy neurális háló konvergenciája? Ismertesse ezeket egy-egy mondatban!
8. Ismertesse röviden a PyTorch könyvtárat! Milyen alapvető adattípusai, moduljai léteznek? Milyen szolgáltatásokat nyújt neurális hálók számára?
9. Készítsen egy egyszerű szkriptet Python nyelven:
   9.1. Töltse be a *torch* könyvtár *nn* nevű modulját!
   9.2. Definiáljon egy új, *MyNet* nevű osztályt, melynek ősosztálya az *nn.Module*
   9.3. Az osztály konstruktorában hozzon létre egy *mult* nevű **tag**változót, értéke legyen 5.
   9.4. Definiáljon egy *forward* nevű tagfüggvényt, melynek bemeneti paramétere *x*, és a *mult* nevű tagváltozóval szorozza meg, az eredményt pedig adja vissza.
10. Hogyan néz ki egyetlen tanulási ciklus PyTorch-ban? (Pszeudó kód elég)