



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

Számítógépes Látórendszerök

JEGYZET

SZEMENYEI MÁRTON

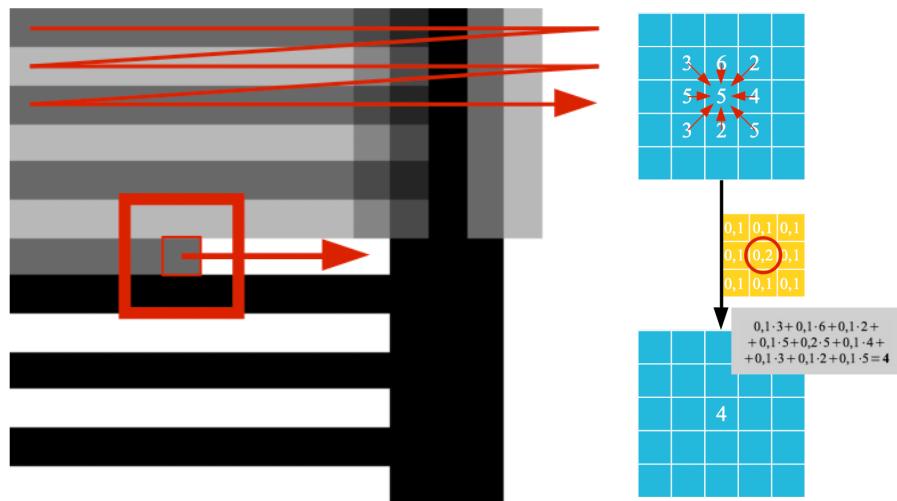
2020. május 7.

2.4. Konvolúciós szűrések

A képjavító algoritmusok családjának legfontosabb tagjai a különböző szűrő algoritmusok, amelyek a képi hibákat és zajokat hivatottak javítani. Ezek az eljárások konvolúciós szűrésen alapszanak. A képen egy kisméretű szűrőablakkal végighaladva, minden egyes pixelpozícióban az adott pixel új értéke a szűrőablak és a pixel lokális környezete között elvégzett konvolúció művelet eredménye lesz. A konvolúció műveletét az alábbi képlet adja meg:

$$(k \otimes I)(x, y) = \sum_{u=-n}^n \sum_{v=-n}^n k(u, v) * I(x-u, y-v) \quad (2.1)$$

Ahol $I(x, y)$ az y.-ik képsor x.-ik pixele. Mint a képletből is látható, a konvolúció művelete egyszerűen az adott környezetben lévő pixelknek a szűrőből vett súlyok alapján számított súlyozott összege. A gyakorlatban minden olyan szűrőt alkalmazunk, amelyeknél a súlyok összege egy, különben a képen világosabbá vagy sötétebbé tennénk. Fontos megjegyezni, hogy habár a konvolúció képlete alapján a képrészleten és a szűrőn ellentétes irányban kellene haladnunk, a gyakorlatban sokszor ezt mégsem így tesszük. Így a valóságban a keresztkorreláció műveletét számoljuk, de ezt mégis konvolúciót nevezünk, holott a kettő eredménye csak középpontosan szimmetrikus szűrők esetén egyezik meg.

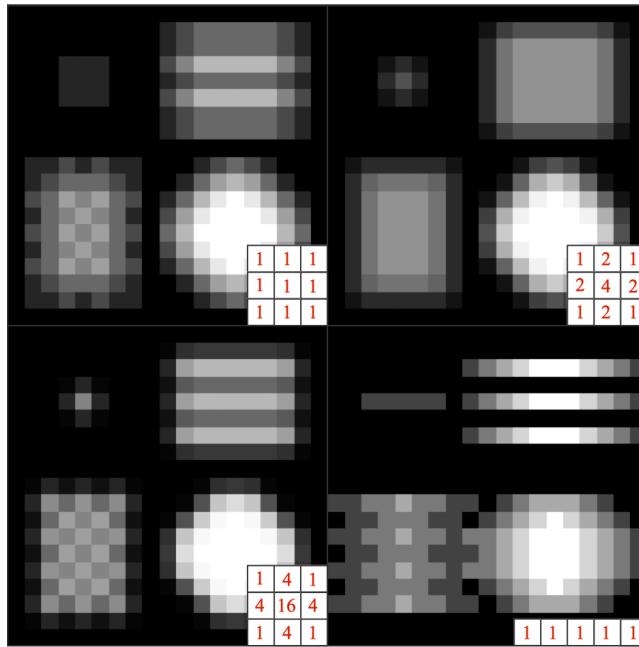


2.5. ábra. A konvolúciós szűrés elve (balra) és a konvolúció művelete egy adott pozícióban (jobbra).

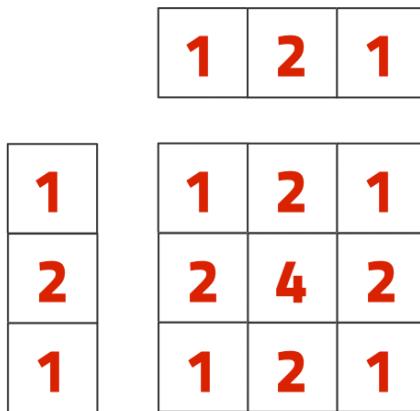
2.4.1. Lineáris szűrők

A zajszűrésre alkalmazott konvolúciós szűrőket simító szűrőknek nevezzük, amelyeknek legegyszerűbb változata az átlagoló szűrő. Valamivel kifinomultabb változat a Gauss-szűrő, amely a középponttól távolabb pixelket kisebb súlyval veszi figyelembe, mindezt egy Gauss-harang görbe alapján. A harangfelület szórásának állításával lehetőség van a simítás erősségeinek kézben tartására, sőt, ha az egyes irányokban más szórásértékeket adunk meg, akkor létrehozhatunk olyan Gauss-szűrőt, amely az egyik irányban sokkal drasztikusan simít, mint a másikban.

A simító szűrők egyik legalapvetőbb tulajdonsága, hogy a konvolúciós ablak minden eleme nem-negatív, az elemek összege pedig pontosan 1. Amennyiben a súlyok összege ettől eltér, a szűrő a simításon felül még világosítja, vagy éppenséggel sötéti a képet. A konvolúciós szűrők egyik jó tulajdonsága, hogy lineáris műveletek, így egymás utáni szűrések könnyedén összevonhatók. Ezen túl bizonyos szűrőablakok esetén lehetőség nyílik a szűrő szeparálására: ekkor a 2D szűrést, két egymás után 1D szűréssel helyettesíthetjük. Ebben az esetben a szűrő végrehajtásának számítása N^2 helyett $2 * N$ műveletbe kerül. Ezt azonban csak egy rangú szűrőmátrixok esetén tehetjük meg.



2.6. ábra. Néhány tipikus konvolúciós szűrő: Az átlagoló (bal felül), a Gauss szűrő két különböző szórás értékkel (jobb felül és bal alul), valamint egy vízszintes átlagolást végző szűrő (jobb alul).



2.7. ábra. Egy 2D szűrés fölbontva két 1D szűrőre.

A konvolúciós simító szűröknek két problémája van: az egyik, hogy az átlagolás után a zajokat ugyan hatékonyan eltüntetik, azonban a szűrés közben a kép egyes részleteit (főleg az éles váltásokat, éleket) is elmosák, ezzel homályossá téve a képet. Másrészt, mivel az összes ilyen szűrő valamilyen átlagolást végez, ezért az esetleges kiugró értékek (só- és borszaj) ezt az átlagot meg lehetősen el fogják tériteni. Ennek eredményeképpen a só és bors jellegű zajokat ezek a szűrők inkább csak elkenik, ahelyett hogy kiküszöbölnék.

2.4.2. Rang szűrők

Ezekre a problémákra adnak megoldást a rangszűrök. A rangszűrök szintén az adott pixel egy kis környezetét veszik figyelembe, azonban nem a konvolúció műveletét végzik el, hanem ehelyett a környezetben lévő pixeleket intenzitás szerint sorba rendezik, és a sorból egy értéket kiválasztva adnak új értéket az éppen vizsgált képpontnak. A rangszűrök közül a különböző feladatokra maximum, illetve minimum szűrőket szoktak használni, képszűrés esetén a mediánszűrők a legelterjedtebbek.

II. rész

Tanuló Látás

8. fejezet

Neurális hálózatok

8.1. Bevezetés

A számítógépes látás területének alapvető célja magas szintű információk kinyerése a képekből. Azonban a bevezető előadásban ismertetett problémák ez meglehetősen nehézzé tehetik. Emiatt adja magát a felvétés, hogy az emberi intelligencia képességeit próbáljuk meg valamilyen módon a számítógépes látás módszereibe beleültetni, ez által lehetővé téve a problémák megoldását. A mesterséges intelligencia tudományterülete hatalmas, számos lehetséges algoritmus áll rendelkezésünkre. Ezen algoritmusok jelentős része egzakt algoritmus, vagyis könnyen megfogalmazható utasítások és logai feltételek valamilyen sorozataként. Ezek az algoritmusok voltaképpen a készítő intelligenciáját aknázzák ki az intelligens működés eléréséhez. Az emberi látás működésének megértése híján ezek az algoritmusok nem segítenének megoldani a fent felsorolt problémákat.

A mesterséges intelligencia módszereinek létezik azonban egy másik csoportja, ezek az úgynevezett tanuló algoritmusok. A tanuló eljárások a probléma megoldására egy általános, paraméterező modellt nyújtanak, és a tanulás folyamata során egy tanító adathalmazt használnak fel arra, hogy ezeket a paramétereket olyan módon határozzák meg, hogy a kezdeti, általános modell az adott probléma megoldására specializálódjon. Ezen algoritmusok hatalmas előnye, hogy segítségükkel megoldhatunk olyan problémákat is, amelyek megoldásának módszerét magunk nem ismerjük, feltéve, hogy képesek vagyunk előállítani egy az algoritmus tanításához megfelelő adatbázist.

Fontos hátránya azonban a gépi tanulás módszereinek, hogy a tanítás végén kapott modell általában fekete doboz jellegű, vagyis a kapott modell megvizsgálásával nem feltétlenül jutunk közelebb a probléma megoldásának megértéséhez. A fekete doboz jelleg miatt azonban rendkívül nehéz megérteni az esetleges hibák, tévesztések okát, és jövőbeli elkerülésüknek a módját. Fontos még megjegyezni, hogy a gépi tanulás módszerei a paramétereket a tanulás során általában statisztikai módszerekkel, vagy numerikus optimalizálás segítségével határozzák meg, következésképp a helyes működésükre nem lehet garanciát mondani. Ezen hátrányok ellenére a számítógépes látás és általánosságban az érzékelés területén toronymagasan felülmúlják a hagyományos algoritmusok teljesítményét.

8.2. Tanuló algoritmusok felépítése

A gépi tanulás során egységesen egy tanuló algoritmus bemenetét x , kimenetét y , paramétereit pedig ϑ jelöli. Ezekkel a jelölésekkel egy tanuló algoritmus modellje megadható egy paraméterezeit függvény formájában:

$$\hat{y} = f(x, \vartheta) \tag{8.1}$$

Minden tanító algoritmushoz tartozik egy költségfüggvény (gyakran nevezik még hiba- vagy veszteségfüggvénynek), amely a tanuló algoritmus kimenetéhez hozzárendel egy hiba értéket, melynek

segítségével az algoritmus teljesítményét tudjuk értékelni. Ezen felül minden tanító algoritmusnak része egy vagy több optimalizálási módszer is, amelynek segítségével a hibafüggvényt minimalizálhatjuk a paraméterek változtatásával. A legtöbb tanító algoritmushoz tartoznak még úgynevezett hiperparaméterek is, melyek olyan paraméterek, amelyek a megoldás minőségét általában befolyásolják, azonban nem tudjuk őket az optimalizálási módszerrel meghatározni. Tipikusan magának az optimalizálási módszereknek, vagy a modell struktúrájának tulajdonságai ilyenek.

8.2.1. Tanulás típusai

A gépi tanulás módszereit számos fontos szempont szerint lehetséges csoportosítani, melyek közül az első az algoritmus kimenete szerinti csoportosítás. Regresszió esetén a tanuló rendszer kimenete egy folytonos szám. Amennyiben az algoritmus kimenete egy bináris változó, vagy egy véges halmazból származó egész szám, akkor pedig osztályozásról beszélhetünk. Az osztályozás alapeset a bináris osztályozás, mivel egy többértékű osztályozó rendszer előállítható több bináris osztályozó kompozíciójaként. Ez elképzelhető úgy, hogy minden osztályhoz tartozik egy bináris osztályozó, amely az adott osztályt minden mástól meg tudja különböztetni, és a végső osztályt az egyes osztályozók konfidenciája dönti el. Elképzelhető olyan rendszer is, ahol az egyes osztályozók két osztály között tudnak dönten, a végső osztályt pedig a sportbajnokságokhoz hasonló pontozási módszerrel döntik el.

Egy másik fontos csoportosítási elv a tanításhoz felhasznált tanító adatok milyensége. A gépi tanulás legegyszerűbb formája a felügyelt tanulás. Ebben az esetben a tanító adatok bemenet-elvárt kimenet párokban állnak rendelkezésre, vagyis minden bemenetre ismerjük a helyes választ, a tanuló algoritmustól pedig azt várjuk el, hogy ezeket minél nagyobb arányban, vagy minél pontosabban találja el. Előfordulhat, hogy a tanító adatbázis csak egy részéhez áll rendelkezésünkre az elvárt kimenet, ebben az esetben félleg felügyelt tanulásról beszélhetünk.

A felügyelt tanulásnak azonban jelentős korlátai vannak. Egyszerűen, a felcímkezett tanító adatbázisok előállítása rendkívül hosszadalmas és drága feladat. Másrészt a címkézés minősége alapvetően korlátozza a tanuló algoritmus minőségét. Végül pedig ahogy az egzakt algoritmusok használata csak akkor lehetséges, ha tudatos szinten értjük az adott probléma megoldását, a felügyelt tanuló algoritmusok pedig csak akkor használhatók, ha mi magunk meg tudjuk oldani az adott feladatot. Ebből következik, hogy egy felügyelt tanuló algoritmus sosem fog tudni olyan feladatokat megoldani, amit mi nem.

Létezik azonban felügyelet nélküli tanulás, amikor a tanító adathalmaz csak bemeneti értékekből áll, az elvárt kimenetet egyáltalán nem ismerjük. Ilyen adatbázisokat rendkívül olcsó előállítani, mivel a legtöbb esetben az új adatok beszerzése automatizálható. Ilyen esetekben azt várjuk el a tanuló algoritmustól, hogy képes legyen valamilyen belső struktúrát találni az adathalmazban, és ezáltal azt kompakt módon leírni. Az algoritmusok célja, hogy a bemeneten látott adatokat valamilyen kompakt modell segítségével magyarázzák, azoknak a belső struktúráját feltérképezzék. Ilyen algoritmusokra jó példák a korábbi fejezetben ismertetett klaszterezési eljárások (k-Means, MoG), amik voltaképpen az osztályozás felügyelet nélküli változatának tekinthetők. Az korábban röviden ismertetett TLS módszer is értelmezhető a lineáris regresszió felügyelet nélküli változatakat.

A gépi tanulás harmadik fő fajtája a megerősítéses tanulás, ami két fontos dolgban különbözik a másik két típustól. Egyszerűen a megerősítéses tanulás esetén szinte minden összefüggő döntések sorát kell az algoritmusnak meghozni, de a döntéssorozat helyességről jellemzően nem kap minden döntés után visszajelést. Másfelől a kapott visszajelzés során az algoritmus csak egy értékelést kap a döntések minőségéről, azt nem tudja meg, hogy a helyes döntés mi lett volna. A megerősítéses tanulási feladatokra tipikusan jó példák a különböző játékok (pl. sakk, go, számítógépes játékok) valamint a különböző járműirányítási feladatok.

8.2.2. Nehézségek

Első ránézésre a gépi tanulás könnyedén tűnhet egyfajta varázslatos módszernek, amivel a világ összes problémáját könnyedén meg lehet oldani. A gyakorlatban azonban ezeknek a módszereknek is bőségesen akadnak limitációik és csapdái, amikbe könnyedén bele lehet esni. A csapdák

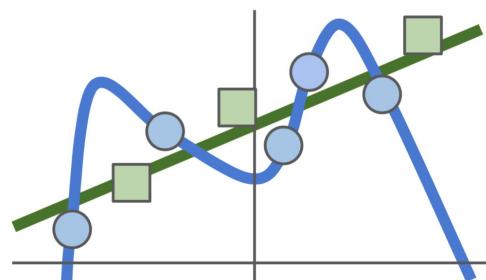
elkerülésének érdekében fontos minden emlékezni arra, hogy ezek az algoritmusok tanító adatokból dolgoznak, ami azt jelenti, hogy a világnak csak azt a szegletét tudják értelmezni, amit a tanító adatok lefednek. Ez azt jelenti, hogy a felhasznált tanító adatainknak minden lehetőséget le kell fednie, mivel nem tudjuk megjósolni, hogy az algoritmus hogyan fog még soha nem látott esetekben viselkedni.

Érdemes azt is észben tartani, hogy például a tanuló látórendszer tipikusan nem tudnak olyan összefüggéseket megtanulni, amelyhez a mozgás képessége, vagy más érzékszervek szükségesek. Ez persze így leírva triviálisnak tűnhet, de gondoljunk csak a szék fogalmára: „egy olyan tárgy, amire rá lehet ülni”. Ezt a definíciót pedig fel tudom használni a szék felismerésére, hiszen ránézésre általában el tudjuk dönteni, hogy valamire rá lehet-e ülni. Egy olyan algoritmus, ami viszont nem tud mozogni, csak összefüggéstelen képeket kap, sosem fogja az ülés fogalmát megalkotni.

Egy másik gyakori csapdája a gépi tanulásnak a tanuló eljárás komplexitásának kérdése. Alapvető emberi intuíció ugyanis az, hogy ha a tanuló algoritmus nem elég pontos, akkor, ha komplexebbé („okosabbá”) tessük, akkor a teljesítmény növelhető. Egy modell komplexitását számos módon lehet növelni: a bemeneti változók és a paraméterek számának növelése két nyilvánvaló módszer. Ezen felül a tanuló módszer hiperparaméterei is általában befolyásolják a komplexitást. A modell komplexitásának növelése azonban kételű fegyver: alapvetően a szituációtól függ, hogy ront, vagy javít-e a helyzetet.

Előfordulhat olyan eset, amikor az algoritmus nem elég komplex az adott feladat megoldásához. Ebben az esetben azt tapasztaljuk, hogy a tanító adatbázison elérte hiba meglehetősen nagy, és ha az algoritmust ezután olyan új adatokon teszteljük, amiket a tanítás során nem látott, akkor hasonlóan hagy hibát kapunk. Ezt a jelenséget alulillesztésnek (underfitting) hívjuk. Ebben az esetben a komplexitást növelte a tanítási és a tesztelési hiba is egyre csökken. Egy idő után azonban azt fogjuk tapasztalni, hogy a tanítási hiba csökkenése mellett a tesztelési hiba először stagnálni, majd növekedni kezd a komplexitás növelésével. Ezt a jelenséget hívjuk túlillesztésnek (overfitting).

A túltanulás oka az, hogy a tanításra használt adathalmaz nem teljesen tökéletes. Egyrészt véges, ami azt jelenti, hogy az algoritmus feladata, hogy megtanuljon általánosítani az adathalmaz segítségével. Másrészt mind a bemenetek, mind a kimenetek zajjal terheltek, így az algoritmus tanítási hibája tökéletes általánosítás esetén sem lesz nulla. Ez azt jelenti, hogy egy idő után az algoritmus már csak úgy tudja tovább csökkenteni a tanítási hibát, ha elkezdi egyesével memorizálni a tanító adatokra adandó helyes választ. Ennek következtében egy a problémát általánosságban megoldó algoritmus helyett egyre inkább egy asszociatív memóriára kezd hasonlítani. Ez azt jelenti, hogy a tanító adatbázisban nem szereplő bemenetekre egyre rosszabb válaszokat ad. Ráadásul minél komplexebb az algoritmus, annál könnyebben tud egy nagy adatbázist memorizálni.



8.1. ábra. Az overfitting egy dimenzió esetén. Látható, hogy a kék színű modell rátanul a tanító adatbázisban lévő zajra, így a tesztadatokon (zöld) rosszul teljesít.

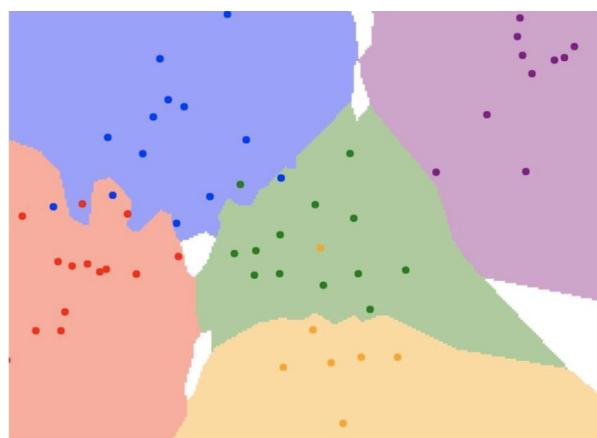
8.3. Képosztályozás

A számítógépes látás egyik legegyszerűbb formája a már korábban ismertetett osztályozás, vagyis amikor egy képhez egyetlen címkét rendelünk, amely a képen található objektum kategóriáját

kódolja. A gyakorlatban egy osztályozást végző számítógépeslátás-megoldás számos algoritmus egymás után történő végrehajtásából áll, amelyet algoritmikus csővezetéknek (pipeline) nevezünk. Ezek első lépése a képek készítése és digitalizálása, amelyet egy előfeldolgozó, képjavító (zajszűrések, intenzitástranszformációk) blokk követ. Ezt követően egy jellemző kiemelési fázis következik, amelynek célja, hogy a képen fellelhető információt a pixelintenzitások által meghatározott térből egy ennél nagyobb absztrakciós szinten létező képjellemzők által meghatározott térbe transzformálja. Ezeket a képjellemzőket úgy tervezük meg, hogy az általuk meghatározott térben könnyen elválaszthatók a feladat szempontjából releváns információkat a zavaró hatásoktól. Az utolsó lépés egy döntési fázis, amelyben az algoritmus a képjellemzők alapján címkét rendel az adott képhez.

8.3.1. Legközelebbi szomszéd

Érdemes a képjellemzők szükségességét egy szemléletes példán keresztül demonstrálni. Lehetséges ugyanis képosztályozást tisztán intenzitás vagy szín alapján végezni, legegyeszerűbben például a k legközelebbi szomszéd elnevezésű, vagyis a kNN (az angol k Nearest Neighbours kifejezésből) algoritmus segítségével. Ennek a végtagról egyszerű eljárásnak a lényege, hogy egy már ismert címkéjű képekből álló adatbázisban megkeresi az éppen osztályozandó képek k darab legközelebbi szomszédját. Ezek a szomszékok aztán többségi elven, szavazással döntik el az új kép címkéjét. A kNN a képek távolságát általában a két kép pixeleinek abszolút vagy négyzetes különbségeinek összegeként definiálja. A módszerben felhasznált k változó értékét a tervező szabadon választhatja.



8.2. ábra. A kNN algoritmus döntési területei $k = 3$ esetében.

A megoldás alapvető problémája, hogy az intenzitás és színértékek közti különbségek összege nincs összhangban a képek hasonlóságával, különösen nem a szemantikus osztályok közti különbséggel. Könnyen belátható, hogy a különböző megvilágítással vagy háttér előtt készült képek különbsége jelentős lesz a rajtuk szereplő objektum osztályától függetlenül. A helyzet különösen rossz olyan objektumok esetén, amelyek hajlamosak számos különböző színezetben előfordulni (például állatok, járművek, ember). E probléma miatt a színinformációt csak olyan esetekben használjuk képek osztályozására, amikor mind az objektumok kinézetét, mind a környezet vizuális tulajdonságait kézben tudjuk tartani. Ilyen szituációkra jó példák a különböző beltéri ipari alkalmazások (például alkatrész-felismerés) vagy a virtuális- és kiterjesztettvalóság-rendszerek.

8.3.2. Lineáris regresszió

Egy másik alapvető paraméterbecslő algoritmus a lineáris regresszió, vagy lineáris legkisebb négyzetes becslés (LS) módszere. A módszer alapelve, hogy a tanító adathalmazban lévő bemenetek és az előírt kimenet közötti összefüggést egy lineáris egyenettel közelítjük. Ez szemléletesen azt jelenti, hogy az adathalmazra egy olyan egyenest (több dimenzióban hipersíkot) próbálunk illeszteni, amely a lehető legkisebb négyzetes hibával közelíti az adatpontokat. A lineáris regresszió modellje az alábbi:



8.3. ábra. A fenti módosított képek négyzetes értelemben megegyező távolságra vannak az eredetitől (bal fent).

$$X\vartheta = Y \quad (8.2)$$

Ahol az X mátrix minden sora egy tanító adat, a ϑ vektor a hipersík paramétereit tartalmazza, Y pedig az elvárt kimenetek vektora. Az egyenlet hibáját minimalizáló megoldáshoz optimalizáló eljárás nem szükséges, ugyanis az optimális megoldás zárt alakban előállítható. Érdemes megjegyezni, hogy az X mátrix oszlopaiban nem csak különböző változók, hanem akár egyetlen változó különböző hatványai is szerepelhetnek. Ekkor nem egyenest, hanem valamilyen polinomot illesztünk az adathalmazra, mely esetben polinomiális regresszióról beszélhetünk.

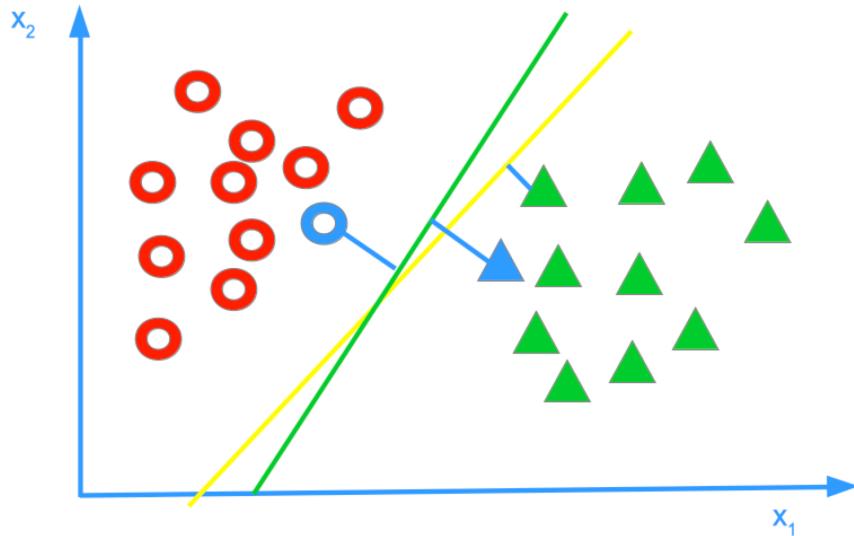
$$\begin{aligned} \|E\|^2 &= (Y - X\vartheta)^T(Y - X\vartheta) = Y^T Y - 2\vartheta^T X^T Y + \vartheta^T X^T X \vartheta \\ \frac{\partial \|E\|^2}{\partial \vartheta} &= -2X^T Y + 2X^T X \vartheta := 0 \\ \hat{\vartheta}_{LS} &= (X^T X)^{-1} X^T Y \end{aligned} \quad (8.3)$$

8.3.3. SVM

A lineáris megközelítést az osztályozás módszereire is ki lehet terjeszteni, ezt azonban úgy lehet elközpelní, hogy az egyes osztályokat egy egyenessel (hipersíkkal) próbáljuk meg egymástól elválasztani. Könnyen belátható azonban, hogy általában végtelen sok ilyen sík létezik, így valamelyen módon jó lenne ezek közül a legjobbat kiválasztani. Ezt elvégezhetjük úgy, hogy megpróbáljuk azt a hipersíket megtalálni, amelyik a legnagyobb biztonsággal választja el a két osztályt, azaz a síkhöz legközelebbi adatpont a lehető legmesszebb legyen az elválasztó hipersíktól. A síkhöz legközelebbi tanító adatot szupport vektornak, míg annak a síktól való távolságát résnek (margin) nevezzük.

A számítógépes látásban népszerű algoritmus az úgynevezett SVM (Support Vector Machine) algoritmus, amely az osztályokat maximális réssel elválasztó hipersíket keresi meg. Az SVM döntésfüggvénye az alábbi módon írható fel:

$$\hat{y}(x) = \sum_i^N \alpha_i y_i K(x_i, x) \quad (8.4)$$



8.4. ábra. A bináris osztályozási feladat: A két osztály, az elválasztó hipersík, a support vektorok és a margin.

Ahol N a tanító adatok száma, y_i és x_i az i-edik tanító adat be- és kimenete, α_i az i-edik tanító adathoz az SVM által megtanult súly, míg K egy kernel függvény. Ez a kernel függvény egy hasonlósági mérce, az éppen osztályozandó bemenet és a tanító adatok között. Ez tulajdonképpen azt jelenti, hogy minden tanító adat kimenete olyan mértékben befolyásolja a döntést, amennyire a két tanító adat hasonlít egymásra. Az ilyen jellegű módszereket általánosságban kernel módszereknek nevezünk, és bizonyos tekintetben a legközelebbi szomszéd módszer általánosításának tekinthetők.

Az SVM módszer egy fontos tulajdonsága, hogy a tanulás során előálló együtthatók csak a szupport vektorok esetén térnek el nullától, vagyis a fenti összeget elég csak erre a néhány vektorra elvégezni. Fontos még a kernel függvény megválasztásáról is beszélni, ugyanis ez alapvetően befolyásolja az SVM képességeit. Az alapértelmezett kernel függvény a lineáris kernel, ami a két bemeneti vektor skaláris szorzatát számolja ki. Lineáris kernel használata esetén az SVM a korábban bevezetett maximális résű elválasztó hipersíkot adja meg.

Érdemes azonban belátni, hogy nagyon sok valódi probléma esetén egyáltalán nem létezik olyan lineáris felület, ami elválasztaná a két osztályt. Erre egy kiváló példa a rendkívül egyszerű kizárt vagy probléma, de számos más szemléletes példa akad ilyen osztályozási feladatokra. Az SVM rendkívül hasznos tulajdonsága, hogy nemlineáris kernel függvények használata esetén képes az egyes osztályokat nemlineáris görbék segítségével is elválasztani, vagyis a módszer könnyedén kiterjeszthető. A két leggyakrabban használt nemlineáris kernel függvény a polinomiális, és az RBF (radiális bázisfüggvény).

$$\begin{aligned} K_{Lin}(x_1, x_2) &= x_1^T x_2 \\ K_{Poly}(x_1, x_2) &= (x_1^T x_2 + c)^k \\ K_{RBF}(x_1, x_2) &= e^{-\gamma ||x_1 - x_2||^2} \end{aligned} \tag{8.5}$$

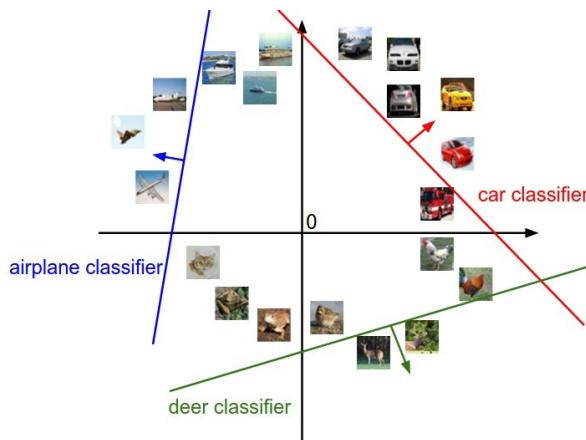
Ahol c , k és a módszer komplexitását kontrolláló paraméterek. A kernel függvények közös tulajdonsága, hogy minden szimmetrikus és pozitív szemidefinit függvények. Valóságban a lineáris kernelhez hasonlóan minden kernel függvény a két bemenete között egy skaláris szorzatot számol ki, azonban előtte a bemeneti vektorokon valamelyen nemlineáris transzformációt hajt végre (természetesen mindezt implicit módon). Ily módon az SVM tulajdonképpen minden kernel esetében hipersíkot illeszt, csak éppenséggel a paramétertér, amiben ezt teszi az eredeti paraméterek nemlineáris transzformációjából képződött, így az illesztett hipersík az eredeti téren valójában egy görbe lesz.

8.3.4. A Perceptron modell

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 rendezzi, 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:

$$s = Wx \quad (8.6)$$

Ahol x a bemenet, s az osztály jóság, W pedig a paraméterek, vagy súlyok mátrixa (ezt a jelölés-rendszeret a jelen fejezetben következetesen alkalmazzuk). Az osztályozás ilyen módját úgy lehet elköpzelni, 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).



8.5. ábra. A lineáris osztályozás esetében az egyes osztály jósági értékek a tér egy irányában lineárisan nőnek, miközött a többiben konstansak. A növekedés irányát a súlymátrix adott osztályhoz tartozó sorverkторa határozza meg.

8.4. A tanítás módszere

Miután definiáltuk a Perceptron algoritmus modelljét és elemezük annak működését, itt az ideje, hogy a modell helyes működéséhez szükséges W súlymátrix elemeinek meghatározásáról is szót ejtsünk. A módszer alapvető kérdése ugyanis, hogy hogyan lehet a súlyok értékét úgy meghatározni, hogy az osztályozás minél pontosabb legyen, valamint, hogy milyen költségfüggvény segítségével mérhető jól az algoritmus teljesítménye.

8.4.1. Hibafüggvények

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 végező 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.

Az egyik gyakran használt hiba az úgynevezett Hinge, vagy SVM hibafüggvény. Ennek a hibafüggvénynek az alapelve, hogy definiálunk egy 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:

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{corr} + \Delta) \quad (8.7)$$

Ahol s_j a j-edik, s_{corr} pedig a helyes osztály jóság értéke.



8.6. ábra. A Hinge költség szemléletesen.

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

$$H(p) = -\sum_i p_i \log p_i \quad (8.8)$$

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 keresztrentrópia mértéke. Persze minél inkább közelíti a q eloszlás a p-t, annál inkább csökken a keresztrentrópia. A két entrópiafajta különbségét KL divergenciának nevezzük, ami egy szigorúan nemnegatív függvény, amelyet gyakran használnak valószínűségi eloszlások hasonlósági mércéjének.

$$H(p, q) = -\sum_i p_i \log q_i \quad (8.9)$$

A keresztrentrópia 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:

$$q_{k,i} = q(y_k | x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}} \quad (8.10)$$

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

$$\begin{aligned} L_i &= H(p_i, q_i) = - \sum_k p_{k,i} \log q_{k,i} \\ L_i &= -\log q_{true,i} \end{aligned} \tag{8.11}$$

Ahol $p_{k,i}$ és $q_{k,i}$ az i-edik tanító adat k-adik osztályhoz tartozó előírt és becsült valószínűségei, $q_{corr,i}$ pedig a helyes osztály becsült valószínűsége. A keresztrömpa költségfüggvény egyik előnye, hogy nehezen értelmezhető „jóság” értékek helyett valószínűség 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.

8.4.2. Regularizáció

Mindkét hibafüggvénynek van azonban egy alapvető problémája. Könnyű ugyanis belátni, hogy minden költségfüggvény esetén, ha egyszerűen a modell aktuális súlymátrixát egy nagy számmal megszorozzuk, akkor a hibafüggvények értéke csökkeni 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:

$$\begin{aligned} L &= \sum_i^N L_i + \lambda R(W) \\ R_{L1}(W) &= \sum_k \sum_l |W_{k,l}| \\ R_{L2}(W) &= \sum_k \sum_l W_{k,l}^2 \\ R_{EL}(W) &= \sum_k \sum_l (\beta W_{k,l}^2 + |W_{k,l}|) \end{aligned} \tag{8.12}$$

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

8.5. Optimalizáció

Az előző előadás egyik legnagyobb lezáratlan kérdése, hogy miként lehet a korábban említett perceptron modell hibafüggvényét minimalizálni. Ebből kifolyólag az első témánk a hibafüggvények minimalizálására szolgáló optimalizálási módszerek tárgyalása. A perceptron súlyainak optimális értékére nem létezik zárt alakú megoldás, így iteratív optimalizálásra lesz szükség.

8.5.1. Gradiens-alapú optimalizálás

Szerencsére azonban a modell és a költségfüggvény is deriválható, így alkalmazhatunk gradiens alapú módszereket. Ha kiszámoljuk a hibafüggvény súlyok szerinti deriváltját (más szóval a súlyok

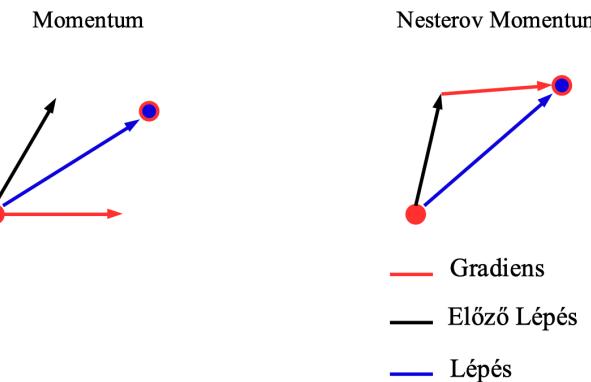
gradiensét), akkor megkapjuk azt, hogy hogyan kellene a súlyoknak megváltozni ahhoz, hogy a hibafüggvény a lehető leggyorsabban növekedjen. Ha azonban a súlyokat a gradienssel ellenkező irányba változtatjuk, az a leggyorsabb csökkenés iránya lesz. Ezt a lépést egyfajta „fordított hegymászként” ismételve egy idő után lokális minimumba jutunk.

Vegyük azonban észre, hogy mivel a teljes hibát szeretnénk minimalizálni, ezért minden egyes lépéskor ki kell 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 a 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:

$$W_{k+1} = W_k - \alpha \frac{\partial \|E\|^2}{\partial W} \quad (8.13)$$

Ahol 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 legrúró szikla képére. Más szóval élve a gradiens módszerhez egyfajta tehetszínűséget, 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.



8.7. ábra. A momentum módszer (bal) és a Nesterov-momentum (jobb). Ez utóbbi előbb lép a momentum irányba, majd ott számolja ki a gradiensest, ami valamivel stabilabb működést eredményez.

A sztochasztikus gradiens módszer egyik hiányossága, hogy a sokdimenziós paramétert minden irányát egyenértékűnek veszi. Előfordulhat azonban, hogy a költségfüggvény az egyik irányban meglehetősen meredek, így ebbe az irányba óvatosan érdemes haladni, mivel egy nagyobb ugrással könnyen átugorhatjuk a minimum helyét. Elközben egy másik irányban a költségfüggvény lapos, az optimum pedig meglehetősen messze található. Míg az első probléma a lépésméretet csökkentését igényelné, a második esetben pont növelni kellene, a kettőt egyszerre pedig nem lehet.

Erre a problémára ad megoldást az AdaGrad és az RMSProp módszer. Mindkettő alapelve az, hogy a gradiensek nagyságát megjegyzik az egyes irányokban, a lépésméretet pedig ezek inverzével skálázzák, így biztosítva az eltérő lépés méretet az egyes irányokban. Az optimalizáló algoritmusok közül az egyik leginkább elterjedt az Adam algoritmus, amely a momentum és a gradiens skálázás módszerét kombinálja.

8.5.2. Magasabb deriváltak

A gradiens módszernek van azonban két meglehetősen nagy hátránya: egyrészt a fix lépésméret következtében az optimum közelében a módszer oszcillálni kezd, hiszen nem tud pontosan az optimum pozícióba lépni. Ezen felül a módszer legtöbbször meglehetősen lassú, különösképp, ha az optimumtól messze lévő pontból indul.

Felmerülhet azonban bennünk, hogy amennyiben a minimalizálandó költségfüggvényt nem egy első-hanem egy másodrendű függvényt közelítenk, akkor ennek a minimumpontját analitikusan kiszámíthatjuk, és egyből bele is ugorhatunk. Ezen az elven működik a Newton-módszer, amely a fentiek alapján a függvényt az adott x_N pontban a másodfokú Taylor-polinomjával közelíti az alábbi módon:

$$f(W_k + \Delta W) \approx f(W_k) + f'(W_k)\Delta W + \frac{1}{2}f''(W_k)\Delta W^2 \quad (8.14)$$

Amit ΔW szerint deriválva és azt nullával egyenlővé téve megkaphatjuk a minimum helyet:

$$\begin{aligned} 0 &= f'(W_k) + f''(W_k)\Delta W \\ \Delta W &= -\frac{f'(W_k)}{f''(W_k)} \\ W_{k+1} &= W_k - \frac{f'(W_k)}{f''(W_k)} \end{aligned} \quad (8.15)$$

Természetesen, mivel a függvény valójában nem másodfokú, ezért a fenti képletet iterációs szabály-ként alkalmazzuk. Természetesen a Newton módszer többváltozós függvények esetében hasonló alakban működik, ekkor a függvény első deriváltja helyett a gradiens, a második deriváltja helyett pedig az ún. Hesse-mátrixot (a másodrendű parciális deriváltakat tartalmazó mátrix) használhatjuk:

$$W_{k+1} = W_k - H_W^{-1} \nabla_W f(W_k) \quad (8.16)$$

A módszernek azonban több problémája is van: egyrészt a Hesse-mátrix számolása gyakorta nehéz, ráadásul a mérete a változók számával négyzetesen nő, így egy több millió paraméterrel rendelkező gépi tanuló algoritmus esetében ez óriási lehet. Ezen felül további problémák adódhatnak, ha a Hesse-mátrix nem invertálható (vagyis valamelyik sajátértéke közel nulla). Ez abban az esetben fordulhat elő, ha a függvény valamelyik irányban lapos, vagy - sokkal valószínűbb - éppen nyeregpontban vagyunk.

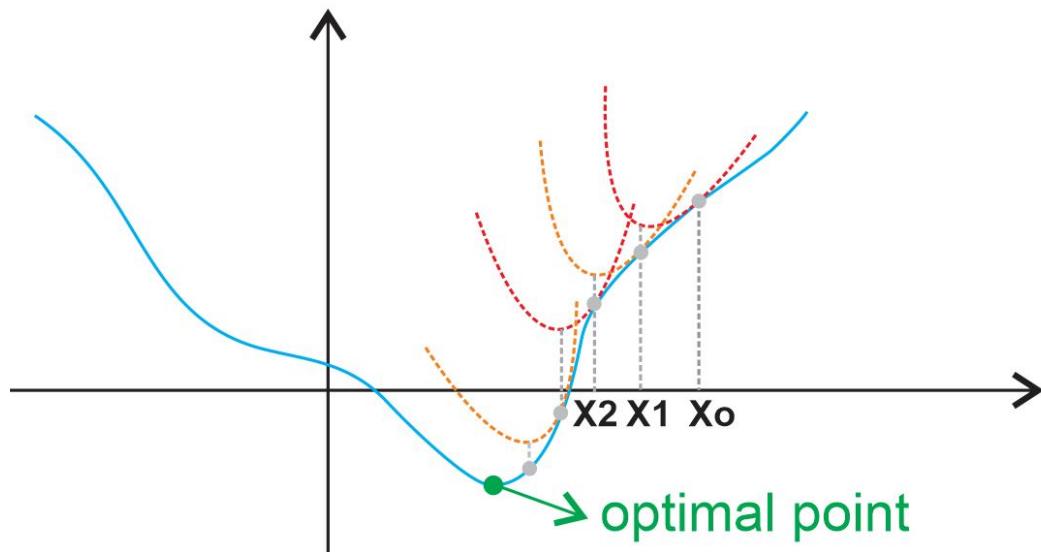
Ennek a problémának az elkerülésére használható a Levenberg-Marquardt algoritmus. Ez az eljárás az alábbi lépésszabályt alkalmazza:

$$W_{k+1} = W_k - [H_W + \lambda I]^{-1} \nabla_W f(W_k) \quad (8.17)$$

Azzal, hogy a Hesse-mátrixhoz az egységmátrix konstansszorosát hozzáadjuk biztosítjuk, hogy az invertálandó mátrix minden pozitív definit legyen. Szemléletesen ebből az adódik, hogy ha a Hesse-mátrix "kicsi", akkor az egységmátrix inverze dominál, és a módszer a gradiens-módszerré egyszerűsödik, míg ha a Hesse-mátrix "nagy", akkor a Newton-módszert használjuk. Ennek hatására a problémás területeken az algoritmus - ha lassan is - de biztosan áthaladhat.

8.5.3. Backpropagation

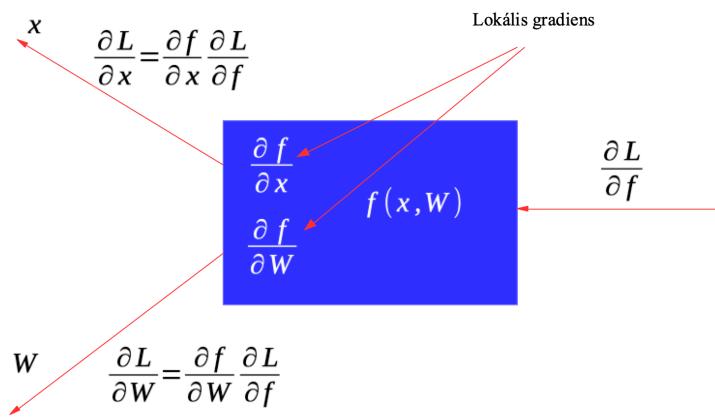
Amint azt már korábban említettük, az egyszerű lineáris modellek képességei erősen limitáltak, így a képi bemenetek esetében ritkán használjuk őket. Ismertetésük azonban szükséges volt, ugyanis



8.8. ábra. A Newton algoritmus elve.

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 súlyátrix, amelyet a korábban ismertetett költségfüggvények és optimalizálási módszerek segítségével határozhatunk meg.

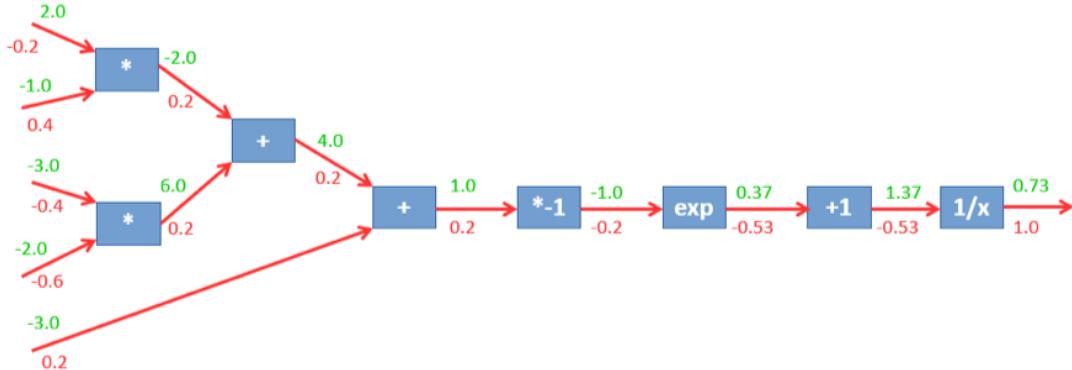
Az egyetlen kérdéses lépés az a hibafüggvény súlyok szerinti deriváltjának számítása. Egy neurális háló elképzelhető, mint egy számítási gráf, ahol a gráf egyes csomópontjai egyszerű, analitikusan deriválható függvényeket implementálnak. Amennyiben egy számítási gráfban ismerjük a bemeneteket, és az 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.



8.9. ábra. A láncszabály szemléltetése.

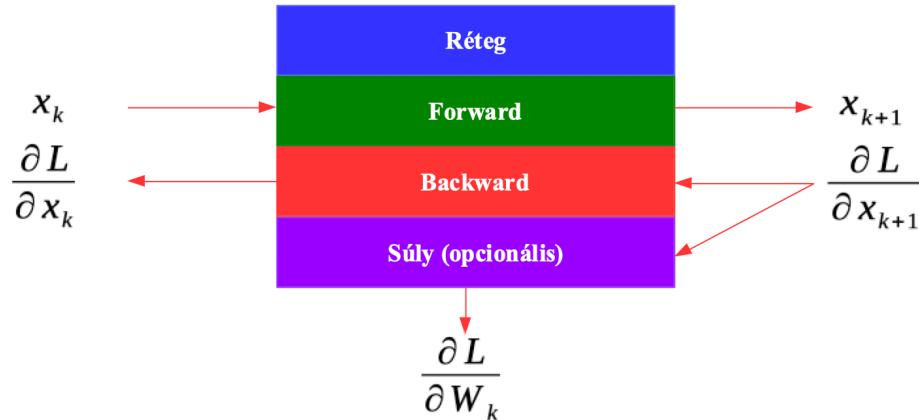
Ahol L a hibafüggvény, f és x az adott réteg ki- és bemenete, W pedig a réteg paraméterei. Ezzel a módszerrel a hálón hátrafele haladva az összes bemenet és súly gradiensét meg tudjuk határozni. Ezt a műveletet hátraterjesztésnek (backpropagation) nevezzük. Az egyetlen kérdés csupán, hogy hogyan kapjuk 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.



8.10. ábra. A backpropagation végrehajtása egy példa gráfon. Zölddel az aktivációk, pirossal a deriváltak szerepelnek.

Érdemes 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.



8.11. ábra. Egy réteg által biztosított, általános interfész, ami tartalmazza a tanításhoz és a predikcióhoz szükséges komponenseket is.

További Olvasnivaló

- [18] J. Heaton, “Ian Goodfellow, Yoshua Bengio, and Aaron Courville: Deep learning”, *Genetic Programming and Evolvable Machines*, vol. 19, no. 1-2, pp. 305–307, Oct. 2017. DOI: 10.1007/s10710-017-9314-z. [Online]. Available: <https://doi.org/10.1007%2Fs10710-017-9314-z>.
- [19] G. James, D. Witten, T. Hastie, and R. Tibshirani, *An Introduction to Statistical Learning*. Springer New York, 2013. DOI: 10.1007/978-1-4614-7138-7. [Online]. Available: <https://doi.org/10.1007%2F978-1-4614-7138-7>.
- [20] D. W. Marquardt, “An Algorithm for Least-Squares Estimation of Nonlinear Parameters”, *Journal of the Society for Industrial and Applied Mathematics*, vol. 11, no. 2, pp. 431–441, Jun. 1963. DOI: 10.1137/0111030. [Online]. Available: [https://doi.org/10.1137%2F0111030](https://doi.org/10.1137/0111030).

9. fejezet

Konvolúciós Neurális Hálók

9.1. Bevezetés

A korábbi előadásban megismertedtünk a gépi tanulás alapjaival, ezen belül is a lineáris osztályozás módszerével. Azt is beláttuk azonban, hogy ezek az algoritmusok nem elég komplexek ahhoz, hogy képek osztályozását jó minőségben elvégezzék. Ennek ellenére szerencsére ezek az egyszerű osztályozók felhasználhatók, mint egy nagyobb mesterséges intelligencia modell építőkövei. A mostani előadás témája az egyszerű lineáris modellekből épített mély neurális hálók lesznek.

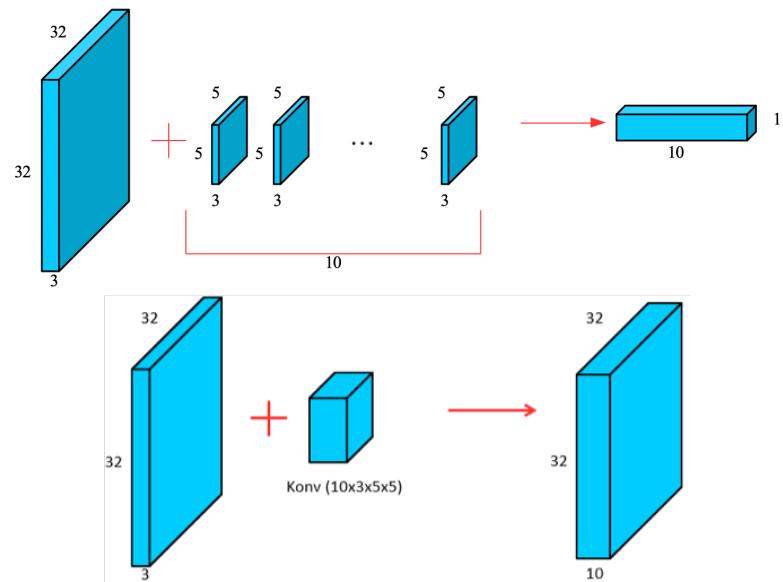
9.2. Konvolúciós neurális hálók

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 térbeli szélességét egyáltalán nem használja ki.

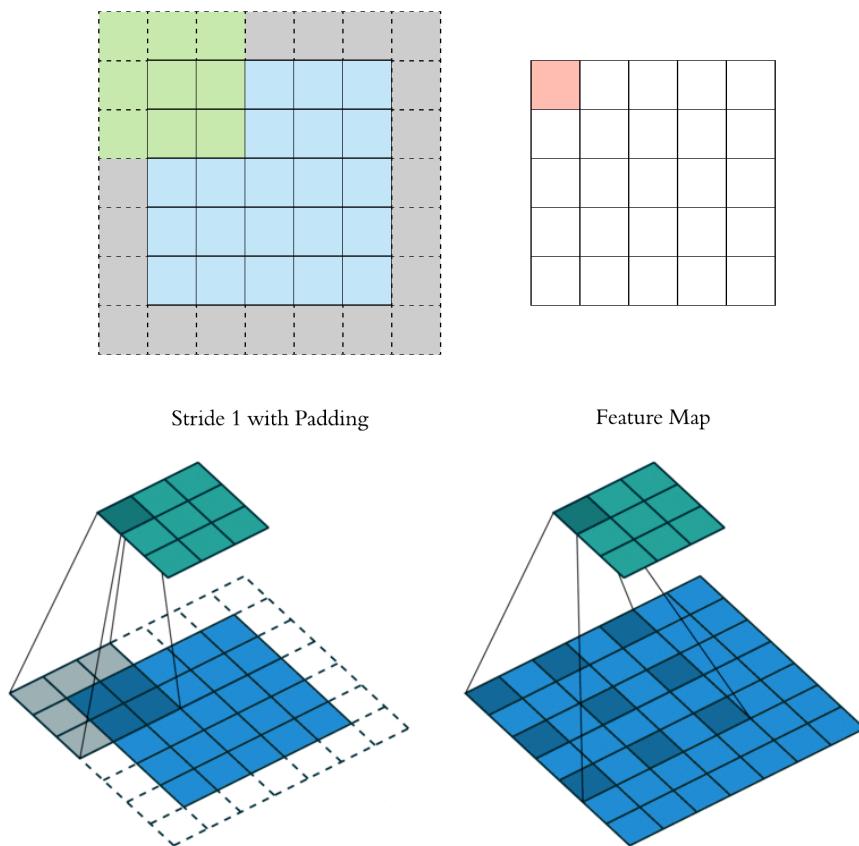
9.2.1. Konvolúciós réteg

Ezekre a problémákra ad megoldást a konvolúciós réteg, amely nevéről 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.

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. A dilatáció (18. ábra) értéke azt határozza meg, hogy a szűrőn egygel 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.



9.1. ábra. A konvolúció végrehajtása egy kép tömbön több szűrővel (felül), és az ezzel ekvivalens konvolúciós réteg (alul).

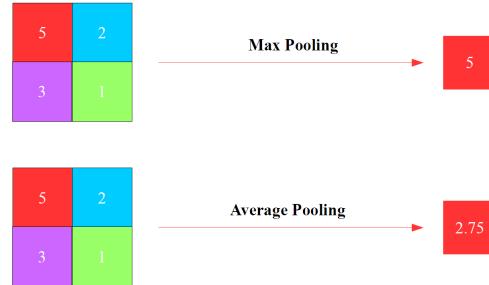


9.2. ábra. A padding (felül), a stride (alul bal) és a dilatáció (alul jobb) hatása a konvolúció műveletére.

9.2.2. Pooling

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

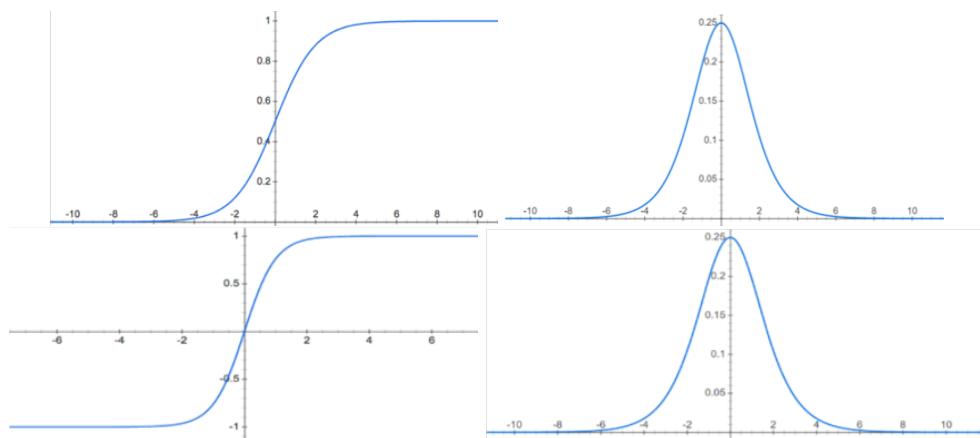
mérete hatalmas lesz. Éppen ezért néhány rétegenként érdemes az aktivációs tömb térbeli méreteit csökkenteni. Az egynél nagyobb stride paraméterrel rendelkező konvolúciós réteg mellett ezt még az úgynevezett pooling operáció segítségével is megtehetjük. A pooling szintén egy csúszóablakos művelet, amely az aktivációs tömb éppen lefedett részét egyetlen számmal helyettesíti. A két leggyakoribb eset, amikor ez az érték az ablak által lefedett értékek átlaga vagy maximuma.



9.3. ábra. A max (felül) és az átlag (alul) pooling.

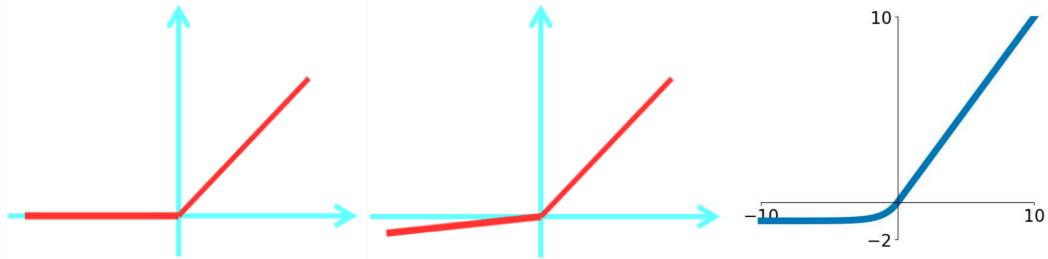
9.2.3. Aktivációk

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 sigmoid, 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 legtöbb deriváltat ki fogják nullázni. Ez a háló bemenethez közelírétegeinek a beragadásához és a tanulás meghívulásához vezet.



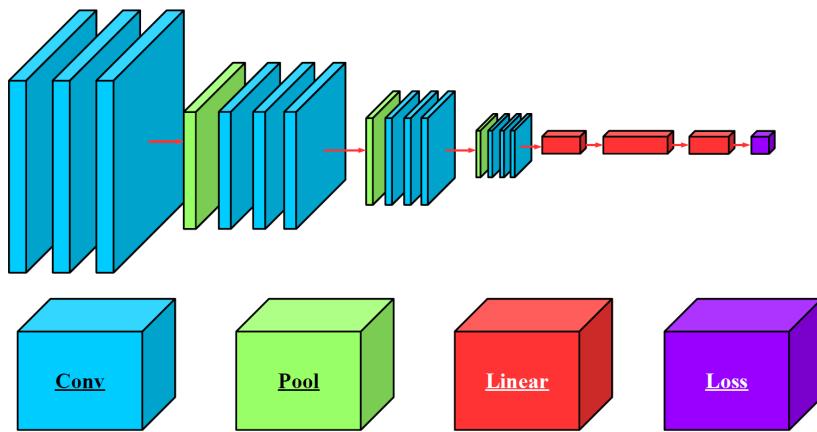
9.4. ábra. A sigmoid függvény és deriváltja (felül), valamint a tanh függvény és deriváltja (alul).

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. Létezik továbbá a ReLU egy olyan változata, amely a hagyományos verzióval ellentétben teljesen sima, így minden pontban deriválható. Ezt Elastic Linear Unit-nak (ELU) nevezzük.

9.5. ábra. A *ReLU* (bal), *Leaky ReLU* (közép) és az *ELU* (jobb) aktivációk.

9.3. Architektúrák

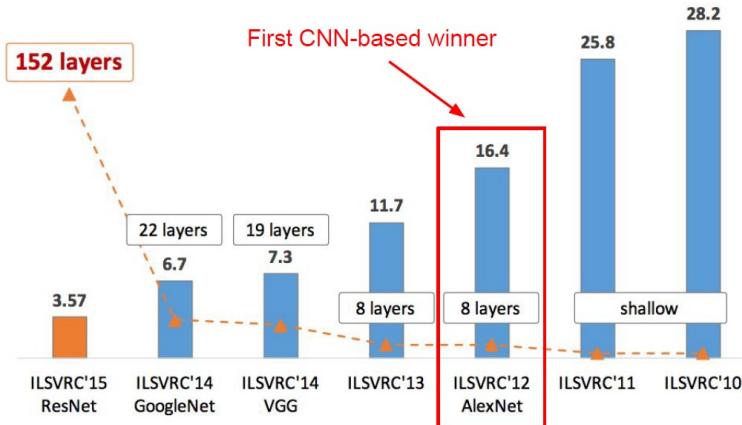
A jelen fejezetben bemutatott rétegeket tartalmazó neurális hálózatokat konvolúciós neurális hálózatoknak nevezzünk, melyeket első sorban a számítógépes látás területén alkalmaznak. A konvolúciós rétegek alkalmazása kifejezetten előnyös képi adatok esetén, mivel egy konvolúciós rétegnek a lineáriszhoz képest több nagyságrenddel kevesebb paramétere van. Egy konvolúciós neurális hálóban általában konvolúciós rétegek sorozata követi egymást (mindegyik kimenetén aktivációs függvénytel), 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.



9.6. ábra. Egy tipikus konvolúciós neurális háló.

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 kompozíciót detektálni képes rétegeket tartalmaz a háló végső részeiben. Ez a szemlélet meglehetősen hasonlít az emberi látás kompozíciós jellegére.

Bár a legelső sikeres konvolúciós hálózatok egy ehhez hasonló architektúrát valósítottak meg, ezeknek számos, fejlettebb változata is létezik. Ezen fejlesztéseknek általában két alapvető motivációja van: az egyik, hogy a mély neurális hálók numerikusan problémás konvergencia tulajdonságait valamilyen módon javítsuk. A másik, hogy olyan struktúrákat alkossunk, amelyek minél kevesebb szabad paraméter mellett minél komplexebb összefüggéseket meg tudnak tanulni, ennek segítségével ugyanis a túlillesztés jelensége csökkenthető. A jelen fejezetben ezen architektúrák mögött rejlik motivációt ismertetjük.



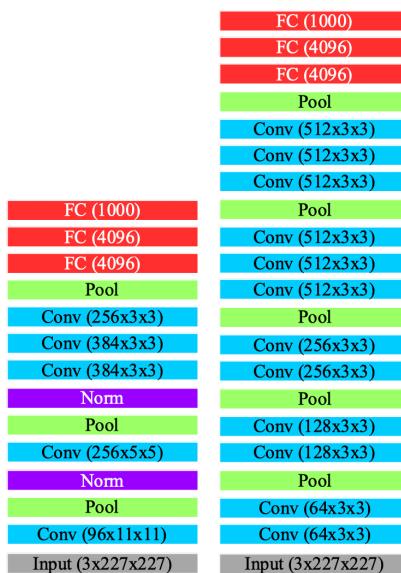
9.7. ábra. Az ImageNet klasszifikációs verseny nyertes hálói és azok rétegszáma.

9.3.1. AlexNet

Az egyik legelső sikeres konvolúciós háló architektúra az AlexNet volt, amely elsőként alkalmazott 10 körüli rétegszámot. Ennek a sikernek a legfőbb oka a ReLU aktivációs függvény és a normalizációs rétegek használata volt.

9.3.2. VGG

A VGG architektúra az egyik legnépszerűbb hagyományos neurális háló modell, melynek 16 és 19 rétegből álló változata is létezik. Az AlexNet-hez képest lényegesen szabályosabb architektúrája van, melyben kizártlag 3x3-as kernelméretű konvolúciós rétegeket alkalmaznak. E mögött a motiváció az, hogy három egymás mögé rakott 3x3 réteg ugyanakkora effektív látómezővel rendelkezik, mint egy darab 7x7, azonban kettővel több nemlineárítással és feleannyi paraméterrel rendelkezik. Ennek következtében összetettebb függvények megtanulása válik lehetővé, a kisebb paraméterszám viszont az overfitting valószínűségét csökkenti.



9.8. ábra. Az AlexNet (bal) és a VGG (jobb) modellek.

Az alábbi ábrán látható a VGG modell egyes rétegei számára szükséges memória mérete, és a réteg paramétereinek száma. Könnyen észrevehetjük, hogy a háló eleje a háló leginkább memóriaigényes része, paraméterek tekintetében viszont az utolsó rétegek szerepelnek igazán hangsúlyosan.

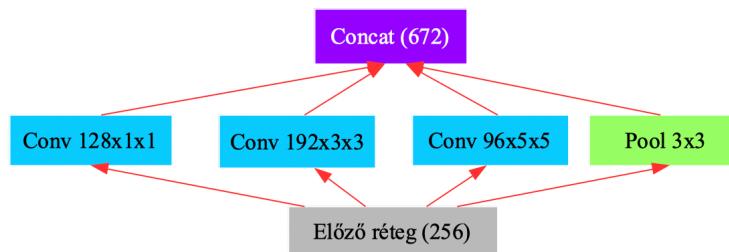
Érdemes megjegyezni, hogy a VGG a modern hálótípusokhoz képest rendkívül pazarló mind paraméterszám, minden memória használat tekintetében.

FC (1000)	1000	4M (4096x1000)
FC (4096)	4096	17M (4096x4096)
FC (4096)	4096	102M (7x7x512x4096)
Pool	25k (512x7x7)	0
Conv (512x3x3)	100k (512x14x14)	2.4M (512x512x3x3)
Conv (512x3x3)	100k (512x14x14)	2.4M (512x512x3x3)
Conv (512x3x3)	100k (512x14x14)	2.4M (512x512x3x3)
Pool	100k (512x14x14)	0
Conv (512x3x3)	400k (512x28x28)	2.4M (512x512x3x3)
Conv (512x3x3)	400k (512x28x28)	2.4M (512x512x3x3)
Conv (512x3x3)	400k (512x28x28)	1.2M (256x512x3x3)
Pool	200k (256x28x28)	0
Conv (256x3x3)	800k (256x56x56)	590k (256x256x3x3)
Conv (256x3x3)	800k (256x56x56)	294k (128x256x3x3)
Pool	400k (128x56x56)	0
Conv (128x3x3)	1.6M (128x112x112)	147,456 (128x128x3x3)
Conv (128x3x3)	1.6M (128x112x112)	73,728 (64x128x3x3)
Pool	800k (64x112x112)	0
Conv (64x3x3)	3.2M (64x224x224)	36k (64x64x3x3)
Conv (64x3x3)	3.2M (64x224x224)	1.7k (3x64x3x3)
Input (3x227x227)	Memória ~ 64M	Paraméterek ~ 140M

9.9. ábra. A VGG háló által használt paraméterek száma és memória mérete.

9.3.3. Inception

Paraméterek csökkentésére irányuló fejlesztésre jó példa az Inception névre hallgató réteg típus. Ennek a megoldásnak a lényege, hogy a konvolúciós rétegek nem csak sorban, hanem egymással párhuzamosan is létezhetnek. Ezek a párhuzamos rétegek általában különböző méretű konvolúciókat hajtanak végre, így egy olyan háló struktúrát eredményezve, amely lényegesen jobban tudja kezelni az objektumok skálájában fellépő variációkat.

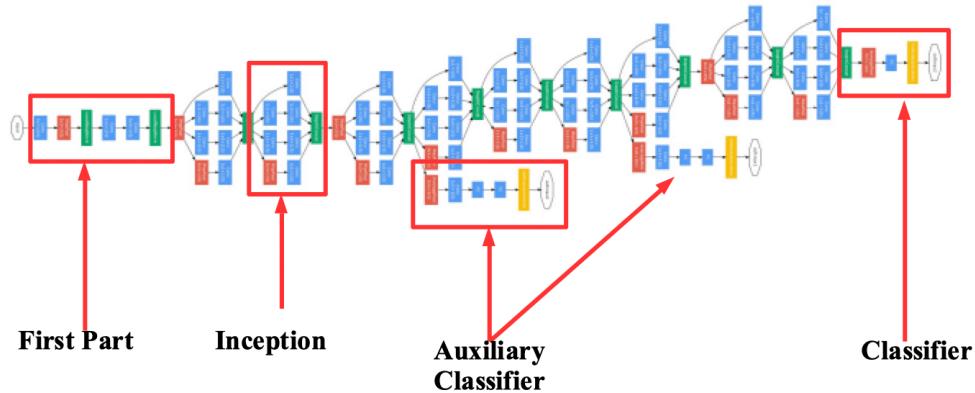


9.10. ábra. A naiv Inception réteg.

A GoogLeNet nevű hálózat számos Inception blokkból épül fel. Ezen felül a hálónak több alsóbb szintről nyíló segéd osztályozója van, melyeknek célja, hogy segítségével a konvergenciát a gradiensek alsóbb szintre történő bevezetésével.

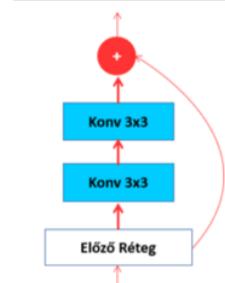
9.3.4. ResNet

A konvergencia javítására az úgynevezett reziduális blokk jó példa. Ez a réteg a konvolúció végrehajtása utáni kimenethez hozzáadja a bemenetet, így a rétegek tulajdonképpen az elvárt bemenet közti különbséget kell előállítania. Ennek a megoldásnak az a haszna, hogy az ilyen blokkokból álló neurális hálóban ezeken az összeadásokon keresztül a hiba deriváltja számára egy olyan út vezet vissza a háló előülső rétegeihez, ami mentén a derivált egy. Ennek következtében a



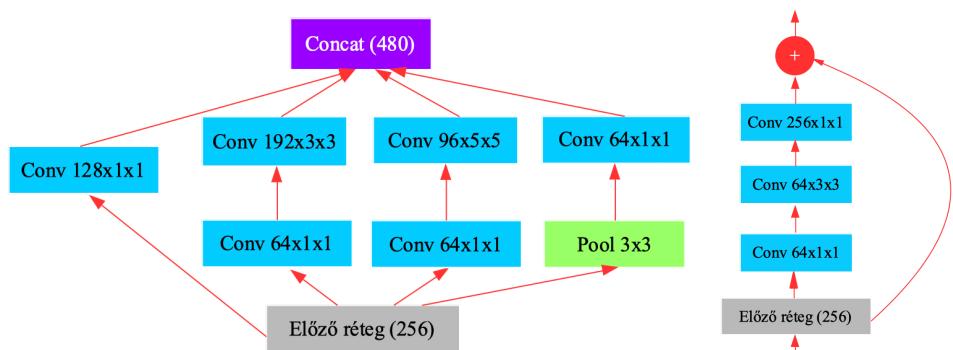
9.11. ábra. A GoogLeNet modell.

hátraterjesztés során végrehajtandó számtalan szorzásból eredő numerikus problémák orvosolhatók. A reziduális blokk bevezetésének hatására a konvolúciós hálók maximális mélysége a 30 réteg körüli értékről a 100-200 réteg nagyságrendjére növekedett.



9.12. ábra. A reziduális blokk.

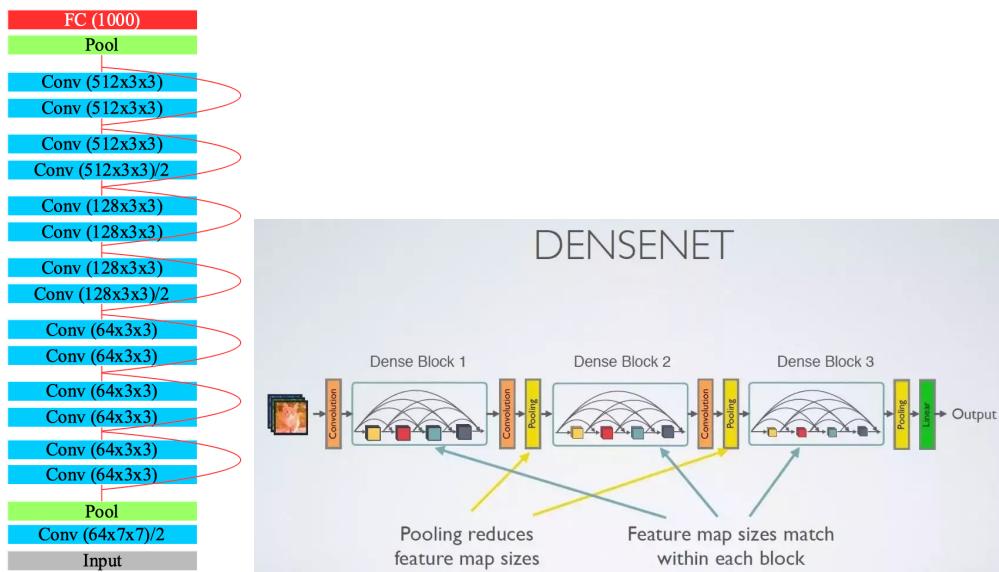
Érdemes még megemlíteni az úgynevezett tömörítő (bottleneck) rétegeket, amiket mind az Inception, mint a reziduális blokkok alkalmaznak. Ezek motivációja az, hogy a kétdimenziós konvolúciók elvégzése rendkívül drága, amennyiben az aktivációs térképek csatornáinak száma nagy. Éppen ezért ezekben a hálókban minden kétdimenziós konvolúciót megelőz egy 1x1-es konvolúciós réteg, amelyik a bemeneti aktivációs térkép csatornáinak számát annak töredékére csökkenti, vagyis tömöríti. Ezt követően a kétdimenziós (3x3, vagy 5x5) konvolúciót ezen a tömörített térképen végezzük el, ezt követően egy újabb 1x1-es konvolúciós réteg ez eredeti csatornaszámot visszaállítja. Ezzel a módszerrel mind az egyes rétegek paramétereinek száma, mind a réteg végrehajtásának ideje nagymértékben csökkenthető.



9.13. ábra. A Bottleneck megoldást alkalmazó Inception (bal) és reziduális (jobb) blokkok.

9.3.5. DenseNet

A DenseNet architektúra a ResNet továbbgondolt változata, melyben egy dense blokkon belül nem csak a közvetlenül egymás utáni rétegek között találhatók áthidaló kapcsolatok, hanem egy blokkon belül minden rétegpár között (innen a dense elnevezés). Ez az újítás lehetővé tette, hogy a ResNet modellel ekvivalens eredményeket érjenek el hozzávetőlegesen feleannyi számítás és paraméter árán.



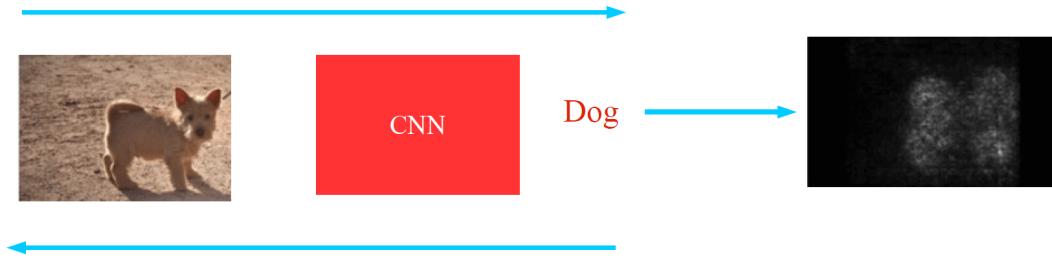
9.14. ábra. A ResNet (bal) és a DenseNet (jobb) modellek.

9.4. Vizualizáció

A korábbi előadásokon részletesen tárgyaltuk a különböző mély neurális hálók felépítésének, tanításának, validációjának és installálásának kérdéseit. Egy rendkívül fontos dologról azonban nem esett szó: ez pedig a hibakeresés problémája. Ez a neurális hálók esetében kifejezetten kritikus kérdés, hiszen a próbálkozás módszere rendkívül költséges, ugyanis egy nagyobb háló tanítása napokig, vagy akár hetekig is eltarthat. A probléma azonban az, hogy a neurális hálók egyfajta fekete dobozként viselkednek, vagyis nem nagyon értjük, hogy pontosan mit és miért csinálnak odabenn. Éppen ezért szükséges lehet a neurális hálók belső működésének valamilyen jellegű vizualizációjára, aminek segítségével betekintést nyerhetünk a belső állapotokba.

Érdemes észrevenni, hogy a gradiens számolás során alkalmazott hátraterjesztés művelete valójában általánosságban felhasználható a háló két tetszőleges pontja közti derivált számítására. Ez az észrevétel számos különböző módon felhasználható. Egyszerűt lehetővé teszi annak meghatározását, hogy melyik pixelek befolyásolják egy osztály jóság értékét egy adott képen. Ez felhasználható az objektumok szegmentálására, követésére, valamint arra is, hogy egy helytelen osztályozás esetén megtudjuk, hogy a képen melyik rész felelős a hibáért.

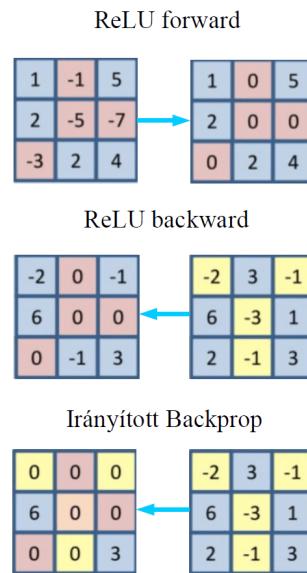
Ehhez semmi más nem kell tennünk, csak a kimeneten a hibafüggvény gradiense helyett a ki-választott osztály jóság kimeneti gradiensére egy értéket írunk elő, míg az összes többire nullát. Ezt követően végrehajtjuk a hátraterjesztés műveletét, azonban ebben az esetben nem a súlyok szerinti, hanem a bemeneti kép szerinti deriváltat határozzuk meg. Ennek az abszolút értékét véve, majd a csatorna dimenzió mentén maximumot véve előállíthatunk egy szürkeárnyalatos képet, ahol a pixelek intenzitása az adott jóság értékre kifejtett hatással arányos. Az így kapott képet saliency-nek nevezzük.



9.15. ábra. A saliency előállítása.

9.4.1. Guided backpropagation

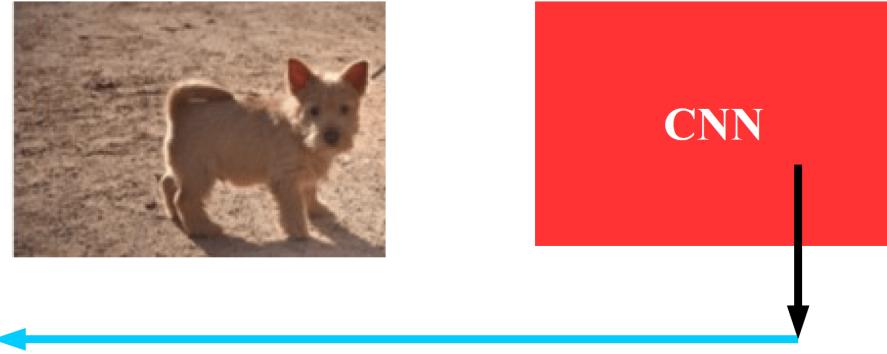
Ha szeretnénk ezt a módszert szegmentálásra vagy vizualizációra használni, akkor azonban szembesülünk egy apróbb problémával: a saliency ugyanis azokon a képrészleteken is nagy lesz, amik a kimenetet negatívan befolyásolták (vagyis "ahol nincs kutya" például). Ennek kiküszöbölésére a hátraterjesztés során el kellene nyomni azoknak a jellemzőknek a hatását, amelyek a vizsgálni kívánt osztály ellen hatnak. Ezt könnyedén megtehetjük, ugyanis könnyedén belátható, hogy ha egy adott jellemző a vizsgált végeredmény hatását csökkenti, akkor a hozzájuk tartozó derivált érték negatív lesz. Innentől kezdve csak annyi dolgunk van, hogy a hátraterjesztés során ezeket a gradienseket minden réteg esetén nullázzuk. Ezt a legegyszerűbb úgy megoldani, hogy a ReLU backward lépését úgy módosítjuk, hogy magán a gradiensekre is alkalmazzuk a ReLU függvényt. Ezt a műveletet hívjuk irányított, azaz guided backpropagationnek.



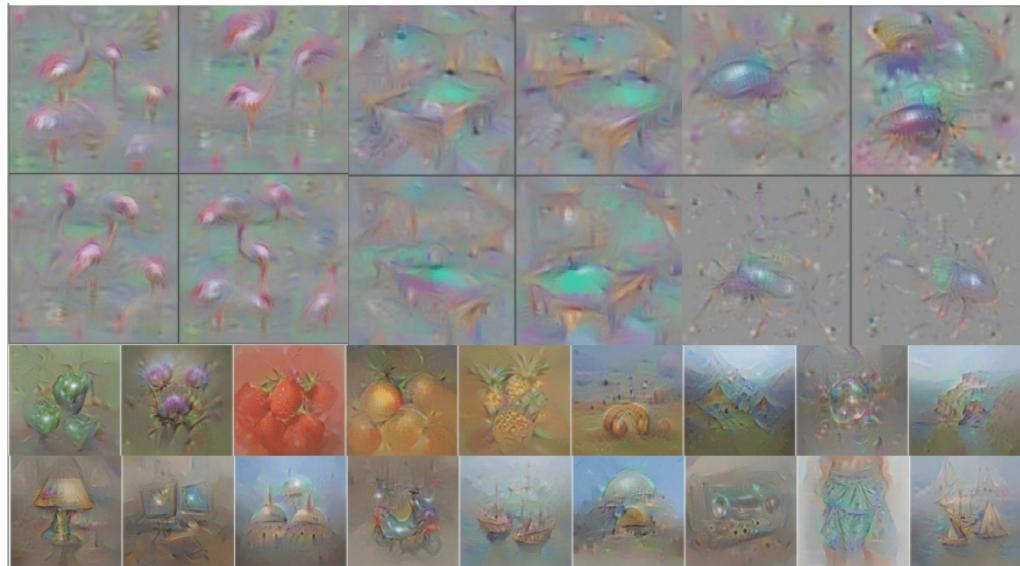
9.16. ábra. A guided backpropagation alkalmazása a ReLU nemlineárításon.

A jó minőségű saliency előállításán felül a guided backpropagation felhasználható tetszőleges neuronok vizualizációjára. Ekkor a célunk, hogy előállítsuk azt a képet, ami az adott neuront (értsd: konvolúciós szűrőt) maximálisan aktiválja. Ehhez először csupa nullával inicializáljuk a bemeneti képet, majd azt a hálóban a kiválasztott neuront tartalmazó rétegig előreküldjük. Ezt követően a réteg kimeneti gradienseit a korábban ismertetett módon előírjuk, és a elvégezzük a backpropagation műveletét egészen a képig. Ezután a kapott gradienseket hozzáadjuk a kép pixeleihez, az előző két lépést addig ismételve, amíg a kép számottevően változik.

Fontos megjegyezni, hogy a kapott képen számos javítást kell végeznünk, különben értelmezhetetlen lesz, valamint a pixel értékek folyamatosan nőni fognak és az algoritmus sosem fog leállni. Emiatt szükségszerű a képen valamilyen (általában L2) regularizációt végrehajtani, valamint rendszeresen simító szűrők segítségével zajt szűrni. Érdemes továbbá a backpropagation során kapott túlságosan kicsi pixel és gradiens értékeket kézzel nullázni.



9.17. ábra. A guided backpropagation elve.



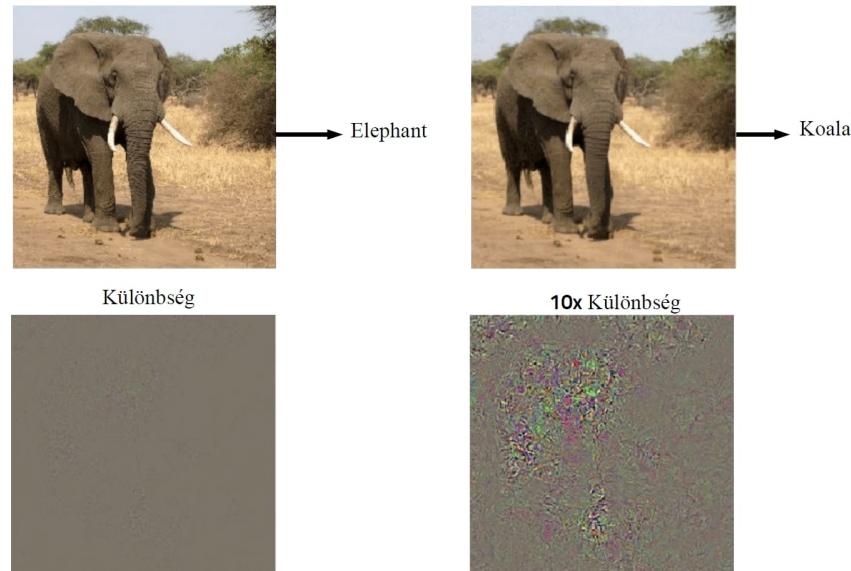
9.18. ábra. A guided backpropagation eredménye.

9.4.2. Ellenséges példák

A backpropagation algoritmus alkalmazásai során egy fontos felfedezés volt, hogy egy tipikus számítógépes látásban használt neurális háló esetén lehetséges helyesen osztályozott képeken olyan minimális, emberi szem által észrevehetetlen változásokat generálni, aminek hatására a neurális háló már tévesen fogja az adott képet osztályozni. Ezeket a képeket ellenséges példának nevezzük, és jelenlegi tudásunk szerint meglehetősen nehéz ellenük védekezni. A legjobb, amit tehetünk az, hogy a tanítás során magunk generálunk ilyen példákat, és ezekkel is tanítjuk a hálót. Persze az ember sem mentes ilyen hibáktól – az emberi látás ellenséges példáit illúzióknak nevezzük. Úgy tűnik azonban, hogy a neurális hálók illúziói jelentős mértékben különböznek az emberétől.

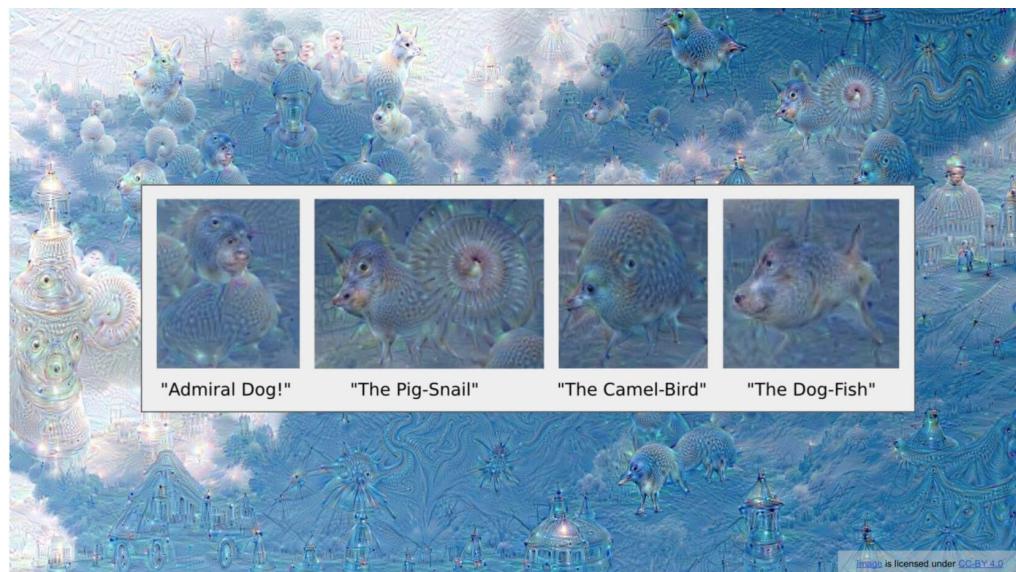
9.4.3. DeepDream

A guided backpropagationnek létezik még egy érdekes alkalmazása. Ez az alkalmazás annyiban különbözik a vizualizációtól, hogy a bemeneti kép nem üres, hanem egy tetszőleges valós kép. Ezt követően ezt ugyanúgy előreküldjük egy kiválasztott rétegig, azonban a réteg kimeneti gradiensét másiképp írjuk elő. A korábban alkalmazott one-hot vektor helyett a kimeneti gradienst egyszerűen egyelőré tesszük magával a réteg aktivációjával. Ez egyszerűen megfogalmazva annyit fog eredményezni, hogy azokat a jellemzőket, amiket a háló a képen erősen detektált felerősítjük, amiket pedig nem látott, tovább gyengítjük. Más szóval létrehozunk egyfajta pozitív visszacsatolást a kép és az adott réteg aktivációi között. Az eredmény egy meglehetősen kreatív, (ré)máloszterű kép.



9.19. ábra. Az ellenséges (*adversarial*) példák.

Bár ennek a módszernek a gyakorlati haszna elenyésző, bizonyította azonban, hogy konvolúciós neurális hálók rendelkezhetnek az emberihez hasonló asszociációs készségekkel.



9.20. ábra. A deep dream eredménye.

További Olvasnivaló

- [18] J. Heaton, “Ian Goodfellow, Yoshua Bengio, and Aaron Courville: Deep learning”, *Genetic Programming and Evolvable Machines*, vol. 19, no. 1-2, pp. 305–307, Oct. 2017. DOI: 10.1007/s10710-017-9314-z. [Online]. Available: <https://doi.org/10.1007%2Fs10710-017-9314-z>.
- [21] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, “Going deeper with convolutions”, in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, IEEE, Jun. 2015. DOI: 10.1109/cvpr.2015.7298594. [Online]. Available: <https://doi.org/10.1109%2Fcvpr.2015.7298594>.

- [22] K. He, X. Zhang, S. Ren, and J. Sun, “Deep Residual Learning for Image Recognition”, in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, IEEE, Jun. 2016. doi: 10.1109/cvpr.2016.90. [Online]. Available: <https://doi.org/10.1109/cvpr.2016.90>.
- [23] G. Huang, Z. Liu, L. V. D. Maaten, and K. Q. Weinberger, “Densely Connected Convolutional Networks”, in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, IEEE, Jul. 2017. doi: 10.1109/cvpr.2017.243. [Online]. Available: <https://doi.org/10.1109/cvpr.2017.243>.
- [24] L. A. Gatys, A. S. Ecker, and M. Bethge, *A Neural Algorithm of Artistic Style*, 2015. eprint: 1508.06576. [Online]. Available: <http://www.arxiv.org/abs/1508.06576>.
- [25] I. J. Goodfellow, J. Shlens, and C. Szegedy, *Explaining and Harnessing Adversarial Examples*, 2015. eprint: 1412.6572. [Online]. Available: <http://www.arxiv.org/abs/1412.6572>.

10. fejezet

Deep Learning a gyakorlatban

10.1. Bevezetés

A korábbi előadások 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 előadás 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 négy doleg nehezíti meg:

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

10.2. Konvergencia problémák

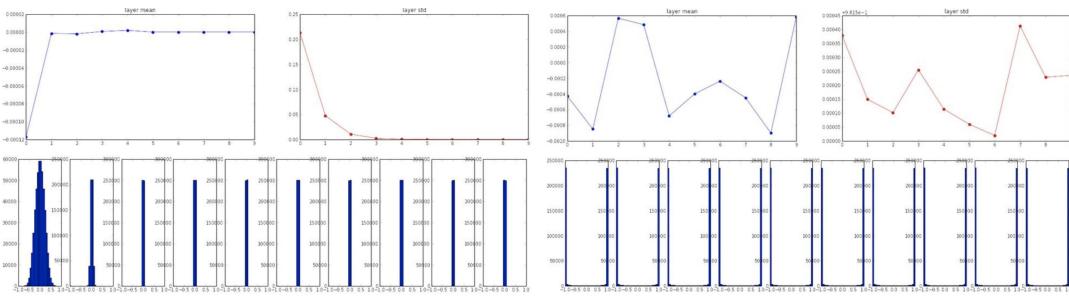
A korábbi fejezetben ismertetett gradiens módszer és backpropagation algoritmus a táblán és a tankönyvek (és ezen jegyzet) hasábjain minden esetben tökéletesen működik. A gyakorlatban azonban a lebegőpontos számábrázolás véges pontossága és tartománya miatt számos problémába ütközünk, amik az algoritmus konvergenciáját könnyedén meghiúsíthatják. Az esetek többségében elmondható, hogy a lebegőpontos aritmetika akkor működik igazán stabilan, ha a számaink eloszlása ”szép”, azaz nulla középértékű, és megközelítőleg egy szórású.

Ennek oka, hogy a backpropagation során valójában egy sor mátrixszorzást kell elvégeznünk. Könnyedén beláthatjuk, hogy amennyiben sok számot szorzunk össze, akkor amennyiben ezek a számok 1-nél kisebbek, egy idő után nullát kapunk, ha pedig egynél nagyobbak akkor végte lent. Ekkor a gradiens értékek (főleg a háló első rétegeinél, ahol a mátrixszorzat már meglehetősén hosszú) vagy kinullázódnak, vagy pedig elszállnak. Ezt hívjuk az eltűnő vagy a felrobbanó gradiensek problémájának. A szorzat csak akkor marad a véges tartományban, ha a számaink aránylag közel vannak egyhez.

Mátrixok szorzása esetében ez ugyanígy igaz, csak itt a mátrixok normájának kell egy körülönök lennie. A mátrixok esetében az egyik legelterjedtebb norma fajta az úgynevezett Frobenius norma, amely egyszerűen a mátrix elemeinek négyzetösszege. Amennyiben a mátrix elemeinek átlaga nulla, akkor ez megegyezik az elemek szórásnégyzetével.

10.2.1. Inicializáció

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 szándékosa hallgattunk, hogy hogyan kapjuk meg a kezdeti paraméter értékeit. A helyzet az, hogy a problémát helyesen megoldó paraméterekről a 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). Ha ugyanis 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 mindenhol közel nullák lesznek, melynek következtében a háló gradiensei is, így a háló a kezdeti értékekbe beragad.



10.1. ábra. A háló rétegeinek aktivációinak eloszlása kicsi véletlen (bal) és nagy véletlen (jobb) számokkal történő inicializálás esetén.

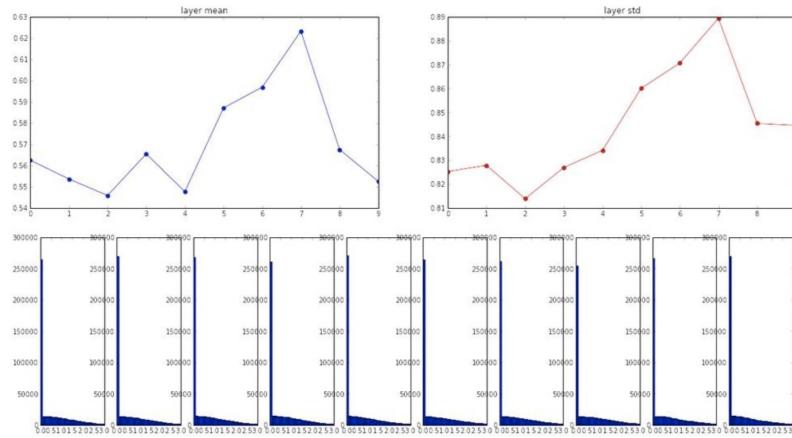
Ennek elkerülésére a háló súlyainak szórását nagy körültekintéssel kell megválasztani, hogy se túl kicsik, se túl nagyok ne legyenek. Ennek több módja is létezik, melyek közül a leginkább elterjedt választások a Xavier, illetve a He inicializációs formulák, amelyek a következőképp határozzák meg a háló egyes rétegeinek kezdeti súlyainak szórását:

$$\begin{aligned} \mu &= 0 \\ \sigma_{Xav} &= \frac{2}{n_i + n_o} \\ \sigma_{He} &= \frac{2}{n_i} \end{aligned} \tag{10.1}$$

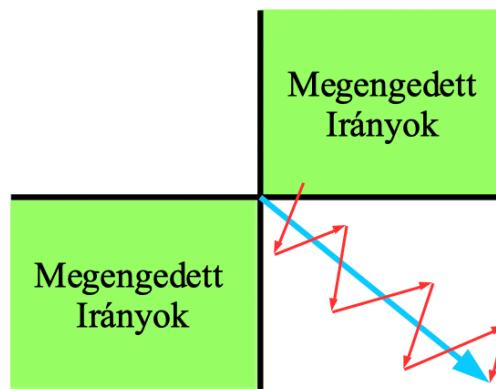
Ahol n_i és n_o az adott réteg be- és kimeneteinek számát jelöli. Érdemes megjegyezni, hogy ezekkel a választásokkal a háló aktivációi és gradiensei megközelítőleg normális eloszlásúak lennének, azonban a ReLU aktivációs függvény használata ezt valamelyest torzítja.

10.2.2. Adatnormalizálás

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 a gradiens lehetséges irányait korlátozzuk.



10.2. ábra. Az aktivációk eloszlása Xavier-inicializáció esetében.

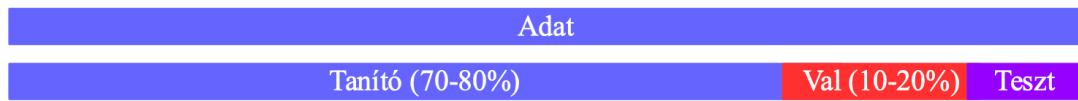


10.3. ábra. Csupa pozitív bemenet esetén a gradiens is csupa pozitív, vagy csupa negatív lesz, így csak "cikkcakkban" képes a kívánt irány felé haladni.

10.3. Validáció és regularizáció

A gépi tanulás alapjainál említettük, hogy a tanítás során felléphet az overfitting jelensége. Ennek észleléséhez két külön adathalmazt érdemes használnunk, egy tanító és egy validációs részt. Ezek lehetnek ugyanannak az adatbázisnak a véletlenszerűen kiválasztott részei (arányuk általában 80%-20% között mozog). A tanító szett segítségével végezzük el a tanítást, míg a validációs szett segítségével az overfitting jelenségét monitorozzuk, melynek alapján a hiperparaméterek hangolhatók. Fontos hangsúlyozni, hogy a validációs adatbázist **SOHA** nem használjuk tanításra.

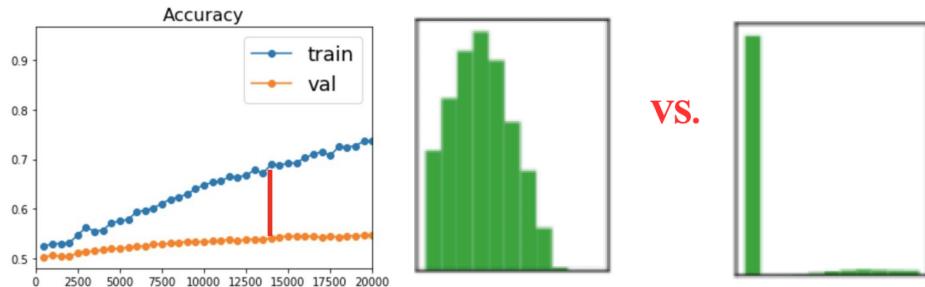
A gyakorlatban azonban két adatbázis nem elég. A tanító adathalmazt ugyanis a háló paramétereinek meghatározásához, míg a validációs adatbázist a háló hiperparamétereinek hangolásához használjuk. Így viszont nem tudjuk azt megmondani, hogy teljesen új adatokon milyen pontossággal teljesít majd a hálózat. Ehhez egy harmadik, a teljes adatmennyiséggel hozzávetőlegesen 10%-át kitervezett teszt adatbázist érdemes használni. A módszer megbízhatóságához elengedhetetlen, hogy ezt a három adatbázist teljesen elkülönítsük és csak egyetlen célra használjuk.



10.4. ábra. Az adatbázisok elosztása.

Érdemes megjegyezni, hogy a neurális hálók túlillesztésének jelensége szintén jellemző az egyes rétegek aktivációinak eloszlásával. A túlillesztés esetén ugyanis az történik, hogy a háló a be- és

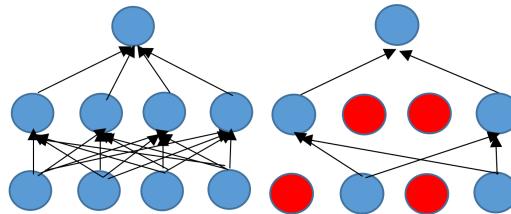
kimenetek közötti általános összefüggések helyett az egyes bemeneti tanítópéldákra adandó helyes választ kezdi el egyesével megtanulni. Ez tipikusan azt jelenti, hogy az egyes rétegekben minden egyes tanító adat esetén csak nagyon kevés aktiváció lesz maximális (amelyek épp az adott tanító példára emlékeznek), míg a többi aktiváció éppen az ellenkező véget 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 tárgyalta regularizációs módszerekre, azok pont a súlyok növekedését próbálták felezni.



10.5. ábra. Az overfitting jelenségéhez tartozó tanítási és validációs görbék (bal), valamint az aktivációk eloszlása normális (közép) és overfitting (jobb) esetben.

10.3.1. Dropout

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 előreterjesztés során az egyes rétegek aktivációinak bizonyos hányadát véletlenszerűen kinullázzuk, és a további rétegek aktivációit így számoljuk tovább (34. ábra). Könnyen belátható, hogy ez a módszer meglehetősen csökkenti a túllesztes 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.



10.6. ábra. A dropout alkalmazása.

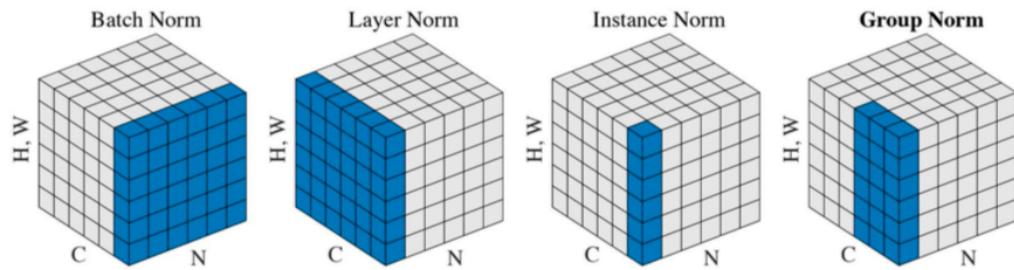
10.3.2. Batch Normalization

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 túlleszést mérsékeli az aktivációk eloszlásának normalizálásával, hanem a numerikus konvergenciát is javítja. A batch normalizálás képlete az alábbi:

$$\begin{aligned} x_{BN} &= \frac{x - \mu}{\sigma^2 + \epsilon} & \leftarrow & \text{vanilla} \\ x_{AF} &= \alpha x_{BN} + \beta & \leftarrow & \text{affin} \\ x_{DC} &= \Sigma^{-\frac{1}{2}}(x - \mu) & \leftarrow & \text{decorrelated} \end{aligned} \tag{10.2}$$

Ahol μ és σ/Σ az x bemenetek becsült átlaga és szórása/kovariancia mátrixa, 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 teljesítménynövedést mutat csak. A batch normalizálás előnyei az alábbi három pontban foglalhatók össze:

- Az aktivációk eloszlásának normalizálása csökkenti az overfitting mértékét.
- A normalizálás miatt az egyes rétegek eloszlásai és gradiensei megközelítőleg ugyanabban a nagyságrendben mozognak, ami segíti a konvergenciát.
- Mivel a gradiens módszer során minden réteg súlymátrixát egyszerre változtatjuk, az első kivételével az összes réteg bemeneteinek eloszlása megváltozik minden lépésben, aminek utána kell tanulni. A batch normalizálás pont ezt küszöböli ki, így lényegesen stabilabbá (és következőképp gyorsabbá) teszi a konvergenciát.



10.7. ábra. Érdemes megjegyezni, hogy a batch normalizációtól létezik még réteg (layer) normalizáció, ahol a teljes aktivációs tömböt normalizáljuk egyedenként külön. Egyed (instance) normalizáció esetén az egyes csatornákat külön-külön normalizáljuk a térfoglalás mentén. A kettő közötti kompromisszum a csoport normalizálás, ahol néhány csatornát egybe veszünk.

10.3.3. Adat Augmentáció

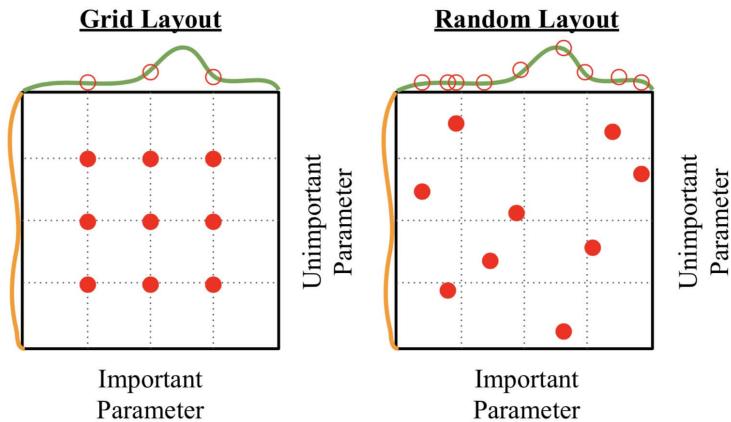
A túlillesztés jelenségének van még egy lehetséges elkerülési módja: gondolunk bele, hogy a túlillesztés esetén a neurális háló a tanító adatbázis elemeire egyesével tanulja meg a helyes választ. Nyilvánvaló, hogy minél több tanító adat áll rendelkezésre, annál nehezebb ezt megtenni. Éppen ezért a tanító adatok számának növelése szinte minden esetben segít a túlillesztés mértékének csökkenésében. 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.

10.4. Hiperparaméter optimalizáció

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 héttig 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.

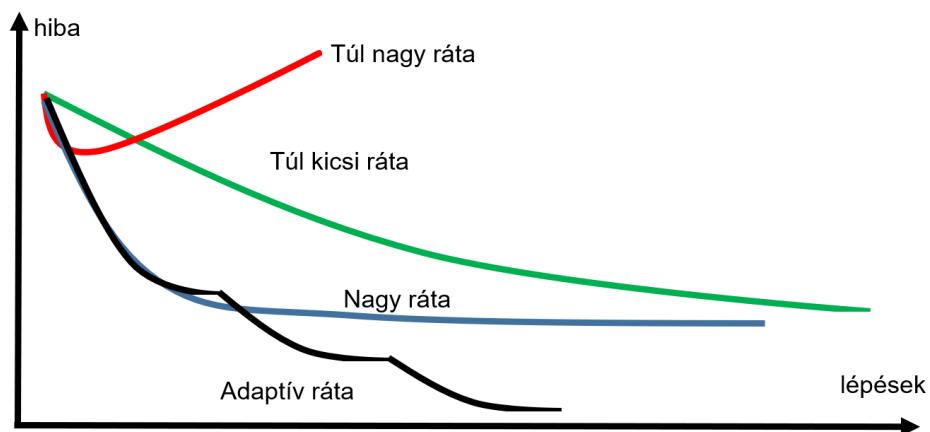
A teljes adatbázison történő tanítás során általában már csak néhány hiperparaméter értékét kell egy aránylag szűk tartományon belül meghatározni. Ekkor célszerű lehet ezeket a tartományokat egy egyenletes rácra osztva az egyes rácpontokban különböző tanításokat végezni, majd ezeket összehasonlítani. 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.



10.8. ábra. A hiperparaméterek optimalizációjának két lehetséges sémája.

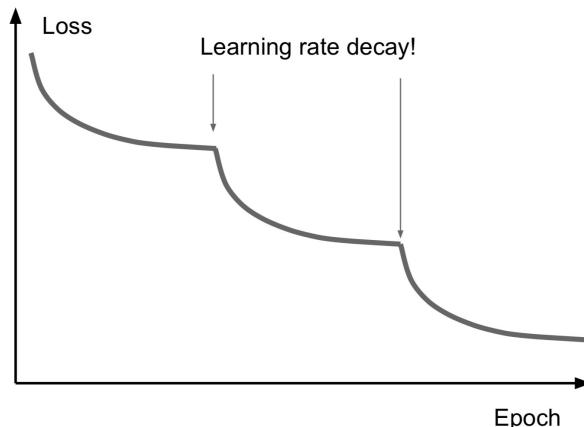
10.4.1. Tanulási ráta

A tanítás hiperparaméterei között külön tárgyalást igényel a gradiens módszer lépéseinak 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.



10.9. ábra. A különböző tanulási ráta választások hatása.

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 leggyorsabban 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 rátá állítására sok lehetséges módszer létezik, melyek közül az egyik leggyakoribb a ráta fix faktorral történő csökkentése bizonyos számú lépés után.



10.10. ábra. A tanulási rátá adaptív változtatása.

Lehetőség van természetesen a rátá 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 rátá „kilöki” a háló súlyait a lokális minimumból és a további tanulás során egy közel a jobb lokális minimum megtalálását teszi lehetővé.

10.5. Adatbázisok előállítása

A neurális hálók tanításának harmadik nehézsége a nagy számú címkézett tanítóadat előállítása. Ez a probléma kis mértékben csökkenthető a már korábban ismertetett adat augmentáció módszerével. További említésre méltó módszer az úgynevezett félre felügyelt tanulás, amikor az adatbázisnak csak egy kis részhalmaza címkézett, a maradék adat esetén azt várjuk el a hálótól, hogy a hasonló adatok hasonló címkét kapjanak.

10.5.1. Transfer learning

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 külböző 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 transfer tanulásnak nevezzük, és elterjedt megoldás a mély tanulás területén. A fent ismertetett érvelés annyira igaz, hogy számos esetben elegendő a hálók legutolsó, lineáris

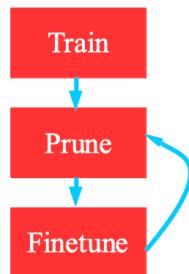
rétegét újra tanítani. Más esetekben szükség lehet az elülső hálórészek finomhangolására, azonban ehhez is nagyságrendekkel kevesebb adat elegendő lehet.

10.6. Installáció

A jelen fejezet utolsó témája a neurális hálók gyakorlati felhasználásra való felkészítésének (installálásának) kérdései. Neurális hálózatok telepítése esetén két probléma adódik: a neurális hálók ugyanis több millió paraméterrel rendelkeznek, vagyis egy mély háló mérete meglehetősen nagy. Ráadásul végrehajtásuk több milliárd műveletet igényel, így meglehetősen lassúak is. Ez nagymértékben megnehezíti a kisebb teljesítményű eszközökben való használatukat.

10.6.1. Pruning

A fenti problémákra adott megoldások közül az egyik legfontosabb a tisztítás művelete (pruning), amely során a háló súlyai közül kiválasztjuk a legkevésbé fontos néhány százalékot és ezeket töröljük. A súlyok rangsorolására számos módszer létezik, amelyek közül a legegyszerűbb egyszerűen a súlyok abszolút értékének használata. Ezt követően a törölt súlyok nulla értéken tartása mellett finomhangoljuk a hálót, majd ezt a két lépést többször megismétljük. Több kutatás is alátámasztja, hogy az iteratív tisztítás technikájával a háló súlyainak 90%-át törölni lehet csupán néhány százalékos teljesítménybeli veszteség mellett. Ez nyilvánvalóan tízszeres gyorsulást eredményez.



10.11. ábra. A Pruning módszere.

10.6.2. Weight sharing

Hasonlóan hatékony módszer a súlyok kvantálása, melynek lényege, hogy a súlyokat egy klaszterező algoritmus segítségével néhány csoportba szedjük, majd minden súlyt a klaszterek középértékével helyettesítünk. Ezt követően a klaszterközéppontok értékét finomhangoljuk, és a tisztításhoz hasonlóan ezeket a műveleteket is iteratívan végezzük. Egy átlagos neurális háló súlyait ilyen módon be lehet osztani 16 csoportba csupán néhány százalékos teljesítményvesztés mellett. Mivel azonban csak 16 különböző fajta súly van a hálóban, ezért egy súlyt elegendő 4 bit felhasználásával ábrázolni. Ez a szokásos 32 bites lebegőpontos számábrázoláshoz képest nyolcszoros tömörítést jelent.

10.6.3. Ensemble

Egy érdekes eljárás még a modell együttesek (ensemble) használata. Ebben az esetben általában több különböző felépítésű, és eltérő módon inicializált és tanított hálózat kimenetének összeállagolásával állítunk elő egy "konszenzus" becslést, amely a tapasztalatok alapján a legjobb háló kimenetét hozzávetőlegesen 2-3%-kal képes javítani. Az ilyen jellegű módszereket gyakran hívják szakértő rendszereknek is.



10.12. ábra. A Weight sharing módszere (bal) és a súlyok kvantálásának séma (jobb).

További Olvasnivaló

- [18] J. Heaton, “Ian Goodfellow, Yoshua Bengio, and Aaron Courville: Deep learning”, *Genetic Programming and Evolvable Machines*, vol. 19, no. 1-2, pp. 305–307, Oct. 2017. DOI: 10.1007/s10710-017-9314-z. [Online]. Available: <https://doi.org/10.1007%2Fs10710-017-9314-z>.
- [26] K. He, X. Zhang, S. Ren, and J. Sun, “Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification”, in *2015 IEEE International Conference on Computer Vision (ICCV)*, IEEE, Dec. 2015. DOI: 10.1109/iccv.2015.123. [Online]. Available: <https://doi.org/10.1109%2Ficcv.2015.123>.
- [27] S. Ioffe and C. Szegedy, *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*, 2015. eprint: 1502.03167. [Online]. Available: <http://www.arxiv.org/abs/1502.03167>.

11. fejezet

Detektálás és szegmentálás

11.1. Bevezetés

A tárgy bevezető előadásában a számítógépes látás több fontos feladatát is felsoroltuk, egyelőre azonban csak az osztályozás megvalósítását ismertettük. A jelenlegi előadásban az objektumdetektálás és a szegmentáció különböző fajtait vizsgáljuk.

11.2. Szemantikus szegmentálás

Az osztályozáshoz a legközelebb álló feladat a szemantikus szegmentálás, melynek során a kép összes pixelét kívánjuk osztályozni. Ez természetesen egy osztályozó neurális háló felhasználásával egy csúszóablakos eljárással elvégezhető, azonban ezt egy átlagos kép százezres, vagy milliós nagyságrendben lévő összes pixelre elvégezni rendkívül hosszú ideig tartana.

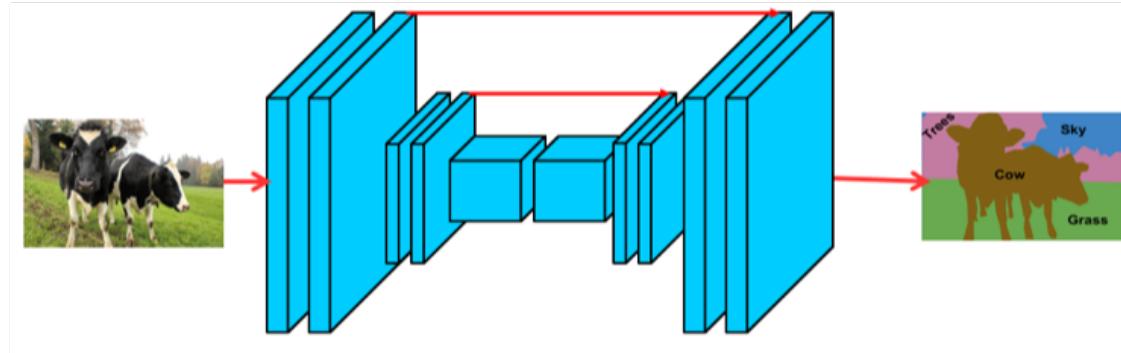


11.1. ábra. A szemantikus szegmentálás feladata.

11.2.1. Teljesen konvolúciós architektúra

Éppen ezért célszerű lenne a folyamatot párhuzamosítani úgy, hogy egyetlen neurális háló segítségével elvégezhető legyen. Erre a teljesen konvolúciós hálózatok (FCN – Fully Convolutional Network) alkalmasak melyek egyszerűen konvolúciós és aktivációs rétegek sorából állnak. A háló utolsó rétege is konvolúciós, kimeneti csatornáinak száma az osztályok számával egyezik meg, így a kimeneti aktivációs térkép elemei az egyes pixelek osztályozásának tekinthetők. Az architektúra problémája, hogy a kép teljes eredeti felbontásán elvégzett konvolúciós drágák, így érdemes lesklálázó operációkat is beiktatni. Ekkor viszont a kimeneti osztályozás is kisebb felbontású lesz, ami nem kívánatos.

A gyakorlatban használt FCN hálók egy fel- és leskálázó részből állnak, melyek többé-kevésbé egymás tükröképei. Ily módon az eredeti kép felbontásával megegyező méretű kimenetet kapunk, de a feldolgozás zömét alacsonyabb felbontáson végezzük, így a futási idő is elfogadható lesz. Érdemes még megjegyezni, hogy az FCN hálók általában tartalmaznak az azonos felbontású fel- és leskálázó részek között rövidzár összeköttetéseket, ami segíti a gradiensek áramlását, így a tanítás konvergenciáját. Az összeköttetések másik előnye, hogy a háló elején detektált alacsony szintű jellemzők segítenek az osztályok határvonalának minél pontosabb meghatározásában, amik a leskálázás során elvesznek.



11.2. ábra. Egy tipikus FCN architektúra.

Az FCN hálók teljesítménye számos további módszerrel javítható. Ezek közül az egyik legegyszerűbb a reziduális, vagy dense blokkok használata a háló leskálázó részében. Ezek használatával elérhető, hogy mély, nagy effektív látómezővel rendelkező hálót használunk szegmentálásra a konvergencia megnehezítése nélkül.

További lehetőség a dilatált (angol irodalomban néha atrous néven említett) konvolúciós szűrők használata. A dilatáció hatása, hogy a szűrő effektív látómezjet megnöveli azonos paraméterszám mellett. Így ugyanaz a háló nagyobb kontextust képes vizsgálni, így javítható a szegmentálás konziszenciája. Szintén hasonló javítást lehet elérni a térbeli piramis pooling (Spatial Pyramid Pooling) használatával. Ez a módszer a háló végén előálló aktivációs térképen több, különböző méretű pooling operációt végez el párhuzamosan, az így keletkező aktivációkat pedig konkatenálja, az osztályozást pedig az így keletkező jellemzők segítségével végzi el. Ennek a megoldásnak az előnye, hogy a több skálafaktor mellett előálló aktivációk együttes használatával a háló skálázékenységét csökkentjük. A piramis pooling műveletét is szokás dilatált módon végezni hasonló megfontolásokból.

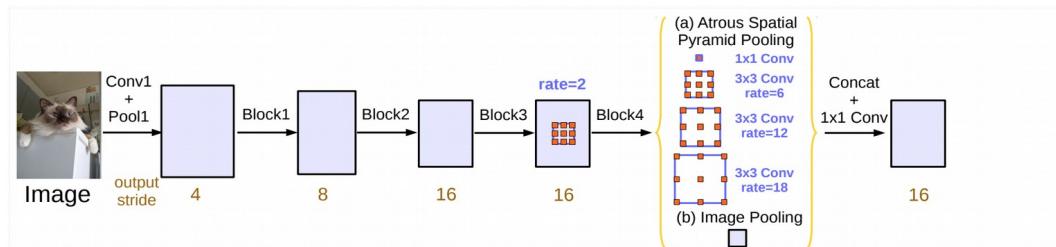
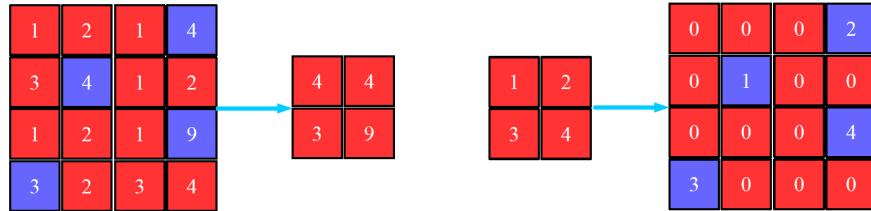


Figure 5. Parallel modules with atrous convolution (ASPP), augmented with image-level features.

11.3. ábra. Egy dilatált piramis poolingot használó architektúra.

11.2.2. Felskálázás módszerei

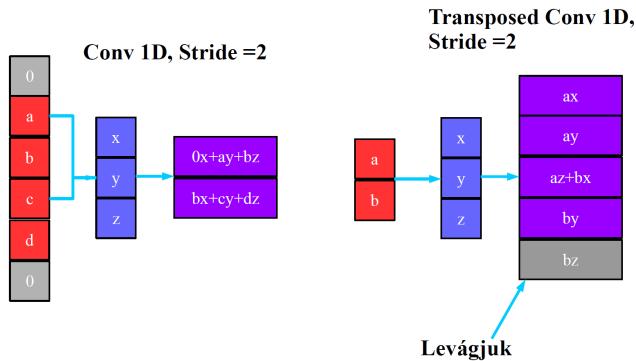
Az egyetlen megmaradó kérdés azonban az, hogy hogyan lehetséges a felskálázást konvolúciós neurális hálózatokban megvalósítani. Erre az egyik legegyszerűbb ötlet az úgynevezett unpooling operáció, melynek működése annyiból áll, hogy a leskálázó részben elvégzett maximum pooling



11.4. ábra. A max unpooling művelet.

során eltároljuk, hogy melyik pixel pozícióban volt a maximum érték, a felskálázás során pedig ebbe a pozícióba írjuk az alacsonyabb szint értékét, míg a többi pozíció nulla marad.

Szintén elterjedt megoldás a transzponált konvolúció, amely tulajdonképpen a stride-dal végzett konvolúció megfordítása. A módszer elnevezése onnan ered, hogy a konvolúció leírható, mint egy mátrixszal való szorzás, a transzponált konvolúció pedig ennek a mátrixnak a transzponáltjával történő szorzás. Ennek a módszernek nagy előnye, hogy a felskálázás tanulható, amely nagymértekben javítja a szegmentálás minőségét.



11.5. ábra. A traszponált konvolúció 1D-ben.

A traszponált konvolúció neve onnan ered, hogy a konvolúció leírható egy mátrixszorzással az alábbi módon:

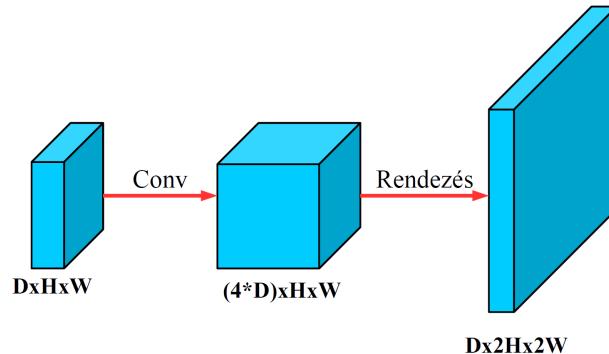
$$\begin{pmatrix} x & y & z & 0 & 0 & 0 \\ 0 & x & y & z & 0 & 0 \\ 0 & 0 & x & y & z & 0 \\ 0 & 0 & 0 & x & y & z \end{pmatrix} \begin{pmatrix} 0 \\ a \\ b \\ c \\ d \\ 0 \end{pmatrix} = \begin{pmatrix} ax \\ ay + bz \\ az + bx \\ by \\ cz + dy \\ dz \end{pmatrix} = Xa \quad (11.1)$$

A traszponált konvolúció akkor a korábbi mátrix transzponáltjával történő szorzásnak felel meg. Érdemes megjegyezni, hogy a mély tanulás területén a transzponált konvolúciót gyakorta szokás - helytelenül - dekonvolúciónak hívni. Ez a tévedés egy mátrix transzponáltjának és inverzének összekeverésével ekvivalens.

$$\begin{pmatrix} x & 0 & 0 & 0 \\ y & x & 0 & 0 \\ z & y & x & 0 \\ 0 & z & y & x \\ 0 & 0 & z & y \\ 0 & 0 & 0 & z \end{pmatrix} \begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix} = \begin{pmatrix} ax \\ ay + bx \\ az + by + cx \\ bz + cy + dx \\ cz + dy \\ dz \end{pmatrix} = X^T a \quad (11.2)$$

Létezik még egy harmadik elterjedt módszer is, ez a sűrű felskálázó konvolúció. Ennek lényege, hogy egy konvolúciós réteg segítségével az adott aktivációs térkép csatornáinak számát a négyszeresére

növeljük, majd az így kapott tömböt átrendezzük úgy, hogy az aktivációs térkép térbeli méretei az eredeti kétszeresei legyenek, csatornának száma pedig egyezzenek meg az eredetivel. Ez a módszer szintén tanuló felskálázás, viszont több paraméterrel rendelkezik, így képes komplexebb transzformációk megtanulására lassabb működés árán.



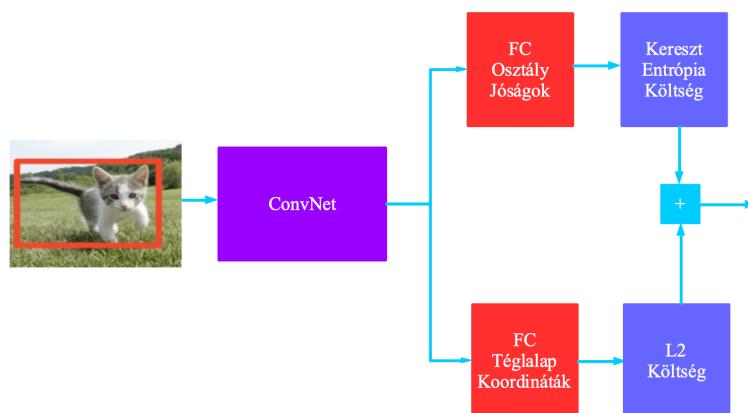
11.6. ábra. A sűrű felskálázó konvolúció (DUC).

11.3. Detektálás

A szemantikus szegmentáció tárgyalásának befejeztével a másik fontos tématerülettel, a detektálással foglalkozunk. Ennek során valamivel egyszerűbb a kinyert jellemző (általában befoglaló téglalapok), azonban megoldható az egyes objektumok különválasztása is.

11.3.1. Lokalizáció

Ennek az egyik legegyszerűbb változata a lokalizáció, amikor az osztályozás feladatát az adott osztályú objektum befoglaló téglalapjának meghatározásával egészítjük ki. Ez a feladat szintén könnyedén elvégezhető neurális hálók segítségével, hiszen nincs más dolgunk, mint egy osztályozó hálóhoz újabb négy kimenetet hozzáadni. Ezekre a kimenetekre előírjuk, hogy a befoglaló téglalap négy paraméterét minél pontosabban adja ki a neurális háló. Mivel ez egy regressziós probléma, ezért a téglalap paramétereinek pontosságát a négyzetes hiba költségfüggvénnyel célszerű mérni. A lokalizációs háló teljes hibája az osztályozás és a regresszió hibafüggvényeinek összege lesz.

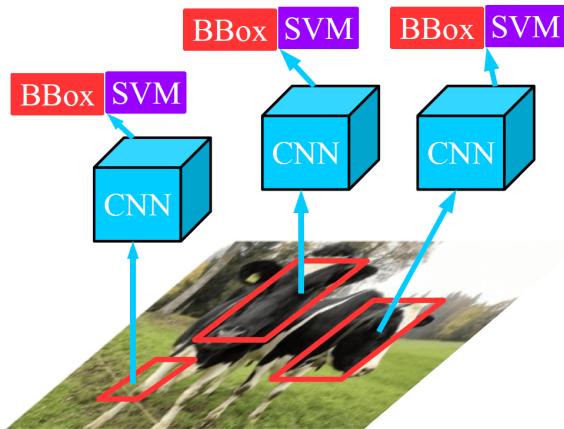


11.7. ábra. A lokalizációt megvalósító architektúra.

11.3.2. Régió-CNN

A detektálás feladata esetén azonban már lényegesen nehezebb dolgunk van. Ennek oka, hogy akárhány, akármilyen osztályú objektum előfordulhat, az architektúrát pedig ennek fényében kell megalkotnunk. Természetesen az objektumok számára egy durva felső becslést adhatunk, így készíthetnénk egy olyan konvolúciós hálót, aminek pontosan ennyi különböző osztályozó és téglalap becslő kimenete van. Ez azonban N maximális objektum és C osztály esetében $N*(C+4)$ kimenetet jelentene, ami rengeteg lehet tekintve, hogy N a néhány tucat, C pedig a százas, vagy az ezres nagyságrendben mozoghat.

Egy alternatív megoldást jelenthet, ha felhasználjuk a korábbi kötetben említett régiójavasló módszereket. Ezek az eljárások tulajdonképpen hagyományos szegmentálási módszerek, amelyek segítségével összefüggő régiójavaslatokat állíthatunk elő. Ezekben a régiójavaslatokon ezt követően egy lokalizációra használható konvolúciós neurális hálózatot futtatunk le egyesével. Ezt a módszert R-CNN (Region Convolutional Neural Net) néven ismerjük. Érdemes megjegyezni, hogy a felhasznált lokalizációs háló osztályozó kimenetének a releváns osztályokon felül tartalmaznia kell egy „egyik sem” kimenetet az objektumokat nem tartalmazó régiójavaslatok kiszűréséhez.



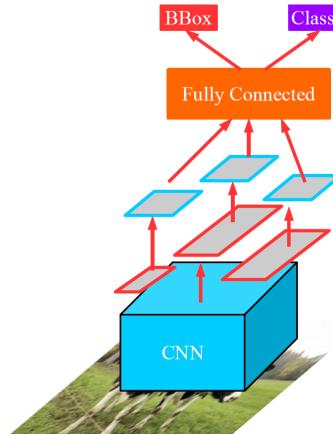
11.8. ábra. A R-CNN architektúra.

Az R-CNN módszer egyik legfontosabb hátránya, hogy az összes régiójavaslaton külön-külön futtatjuk le a neurális hálót, ami pazarlás. A módszer egy továbbfejlesztése, a FastR-CNN az egész képen futtat egy csak konvolúciós és leskálázó rétegekből álló hálót, majd az ez által elkészített aktivációs térképen keres régiójavaslatokat. Ezt követően ezeket a javaslatokat egy speciális pooling művelet segítségével azonos méretűre hozza (a hagyományos pooling műveletek egy adott faktorral skáláznak). Ezt követően a régió javaslatokon már csak egy kis méretű, csak lineáris rétegekből álló hálót futtat, amelyek az osztály és a téglalap becslését végzik (32. ábra). Ez a módszer az eredeti R-CNN módszerhez képest 10-20-szor gyorsabban működik.

A FastR-CNN működésének a leglassabb része a régiójavaslatok előállítása, ami a futási idő 90%-át teszi ki. Éppen ezért megalkották még egy továbbfejlesztett változatot, ami a régiójavaslatok előállítását is egy RPN (Region Proposal Net) nevű neurális háló segítségével végzi el. Ez a háló a kezdeti konvolúciós rész által előállított aktivációs térképből állít elő fix számú régió javaslatot, amelyek mindegyikét binárisan osztályozza (objektum/nem objektum). Erre azért van szükség, mert a fix számú régió kimenet miatt a háló fixen ennyi régiójavaslatot tesz. A fő detektáló háló tanítása mellett az RPN hálót is arra tanítjuk, hogy a régiók befoglaló téglalapját és „objektumszerrűségét” minél nagyobb pontossággal találja el. Ez a módszer a FastR-CNN megoldáshoz képest egy újabb tízszeres gyorsítást eredményez.

11.3.3. YOLO

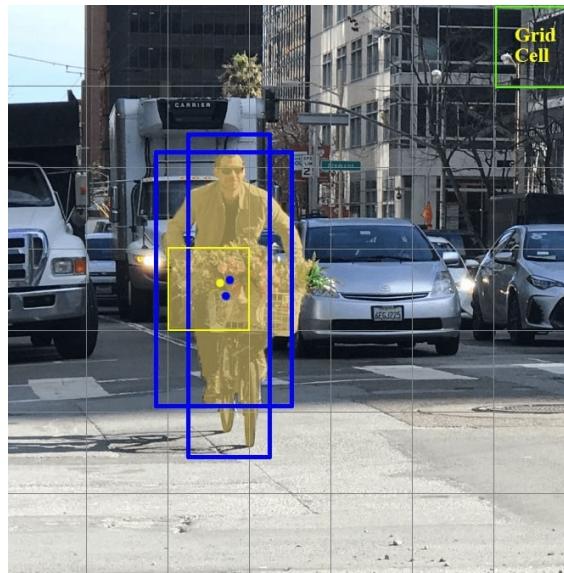
Fontos azonban tudni, hogy nem csak régiójavaslatok segítségével lehet hatékony objektumdetektálist végezni. Erre kitűnő példa a rendkívül népszerű YOLO (You Only Look Once) architektúra,



11.9. ábra. A Fast R-CNN architektúra.

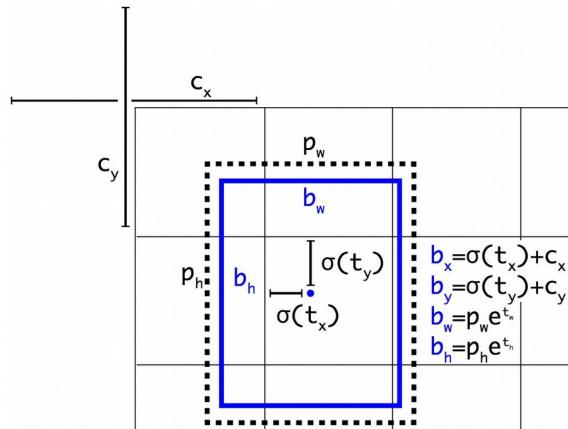
mely nem összetévesztendő a megegyező rövidítésű szállóigével. A YOLO megoldás alapvetően hasonlít a detektálás tárgyalásának elején felvettet sok külön lokalizáló kimenetet javasló megoldáshoz. A működése során a YOLO a képet először egy tisztán konvolucióból és leskálázásból álló hálón küldi végig, így előállítva a végső becslésekhez felhasznált aktivációs térképet.

A végső becsléshez a YOLO a képet egy NxN-es rács segítségével felosztja, és minden cellából B darab objektumjelölt téglalapot becsül. minden téglalaphoz tartozik egy C kimenetű osztályozó, valamint bináris osztályozó is, amely az adott téglalapba eső képrészlet „objektumszerűségét” adja meg. Így minden egyes cella esetén $B \times (5+C)$ kimenete van a hálónak, amelyet egy 1x1 méretű konvoluciós szűrővel állít elő. Érdemes megjegyezni, hogy a téglalapok pozícióját a YOLO cella bal felső sarkához képest becsüli meg, így minden objektum detektálásáért az a cella felelős, amelyikben az objektum középpontja található.



11.10. ábra. A YOLO modell által készített rács és becslések.

A befoglaló téglalap szélességét és magasságát a YOLO egy referencia téglalaphoz (ún. anchor box) képest becsli meg, amelyből összesen B darab van (minden becsléshez egy). Az egyes anchor boxok szélesség és magasság értékeit a tanító adatbázisban szereplő téglalapokon végzett B elemű klaszterezés segítségével határozzuk meg. Fontos még említeni, hogy a detektálás során a YOLO egy objektumot többször is megtalálhat, mely esetben a túlságosan hasonló alakú predikciók közül a legnagyobb konfidencia értékűt tartjuk meg, míg a többöt eldobjuk. Ezt a lépést nevezzük non-maximum suppression-nek.

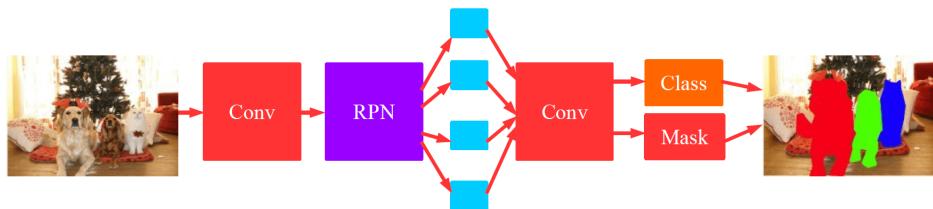


11.11. ábra. A YOLO téglalap becslési módja. A téglalap koordinátáit a rács bal felső sarkához képest, a méreteit pedig az anchor box-hoz viszonyítva becsüljük.

A YOLO-nak több változata is létezik, az anchor boxok használatát például a második verzió vezette be. A harmadik verzió a detektálást három különböző skálafaktor mellett is elvégzi, amihez a szegmentálás során megismert felskálázás és előrecsatolás trükkjeit alkalmazza. Ezzel a megoldással a YOLO teljesítménye jelentősen javul kis méretű objektumok pontos detekciója esetén. A YOLO legfőbb erénye a régió alapú detektálással szemben, hogy rendkívül gyors, így megfelelő hardver használata esetén valósidejű működésre is képes, különösképp a TinyYOLO névre hallgató változata.

11.3.4. Mask-RCNN

Az alfejezet végén érdemes még említeni a jelenlegi témakör utolsó feladatáról, amely az objektumszegmentálás volt. Ebben az esetben nem csupán szemantikus módon kívánjuk szegmentálni a képet, hanem az azonos osztályba tartozó egyes objektumokat is szeretnénk megkülönböztetni. Bár ez nyilvánvalóan a legnehezebb feladat az összes közül, az eddig ismertek alapján egy ilyen architektúra mégis könnyedén megérthető. Az RPN alapú objektumdetektálás során az egyes objektumokat tartalmazó képrészleteket ugyanis már előállítottuk, így a feladatunk csak annyiban különbözik, hogy a befoglaló téglalap helyett minden objektumhoz egy bináris maszkot kell előállítanunk. Ezt könnyedén megtehetjük a szemantikus szegmentálásból ismert felskálázó hálórész segítségével. Ezt az architektúrát Maszk R-CNN néven ismerik.



11.12. ábra. A Mask-RCNN architektúra.

További Olvasnivaló

- [18] J. Heaton, “Ian Goodfellow, Yoshua Bengio, and Aaron Courville: Deep learning”, *Genetic Programming and Evolvable Machines*, vol. 19, no. 1-2, pp. 305–307, Oct. 2017. DOI: 10.1007/s10710-017-9314-z. [Online]. Available: <https://doi.org/10.1007%2Fs10710-017-9314-z>.

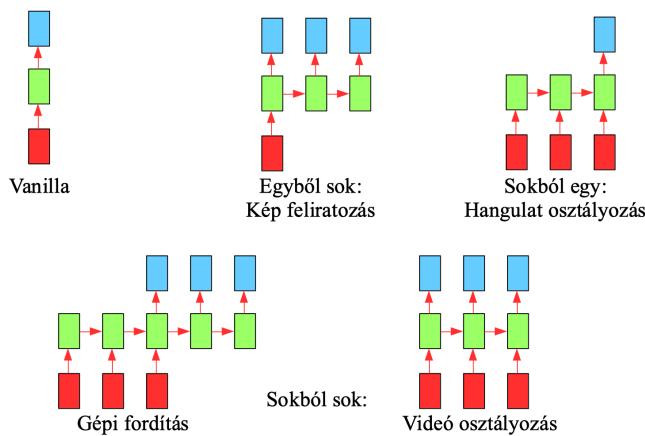
-
- [28] J. Long, E. Shelhamer, and T. Darrell, “Fully convolutional networks for semantic segmentation”, in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, IEEE, Jun. 2015. doi: 10.1109/cvpr.2015.7298965. [Online]. Available: <https://doi.org/10.1109%2Fcvpr.2015.7298965>.
 - [29] 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, Jun. 2017. doi: 10.1109/tpami.2016.2577031. [Online]. Available: <https://doi.org/10.1109%2Ftpami.2016.2577031>.
 - [30] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, “You Only Look Once: Unified, Real-Time Object Detection”, in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, IEEE, Jun. 2016. doi: 10.1109/cvpr.2016.91. [Online]. Available: <https://doi.org/10.1109%2Fcvpr.2016.91>.
 - [31] K. He, G. Gkioxari, P. Dollar, and R. Girshick, “Mask R-CNN”, in *2017 IEEE International Conference on Computer Vision (ICCV)*, IEEE, Oct. 2017. doi: 10.1109/iccv.2017.322. [Online]. Available: <https://doi.org/10.1109%2Ficcv.2017.322>.

12. fejezet

Visszacsatolt hálózatok

12.1. Bevezetés

Ez eddigi diszkusszió során olyan módszereket ismertünk meg, amelyek állóképeket egymástól független feldolgozására alkalmasak. Képsorozatok feldolgozásának azonban számos jelentős alkalmazása van, többek között a videók osztályozása, vagyis más néven az eseménydetektálás. Könnyen belátható, hogy ahogy bizonyos alkalmazások esetében szükség lehet a képen található objektumokat azonosítani, úgy még hasznosabb lehet egy videón lejátszódó eseményt vagy cselekményt felismerni. Egy valamelyest eltérő alkalmazás képek feliratázása, melynek során egy képhez nem egyetlen címkét, hanem egy egész mondatot rendelünk, így lényegesen komplexebb leírást tudunk adni. Ebben az esetben nem a háló bemenete, hanem annak kimenete értelmezhető sorozatként.



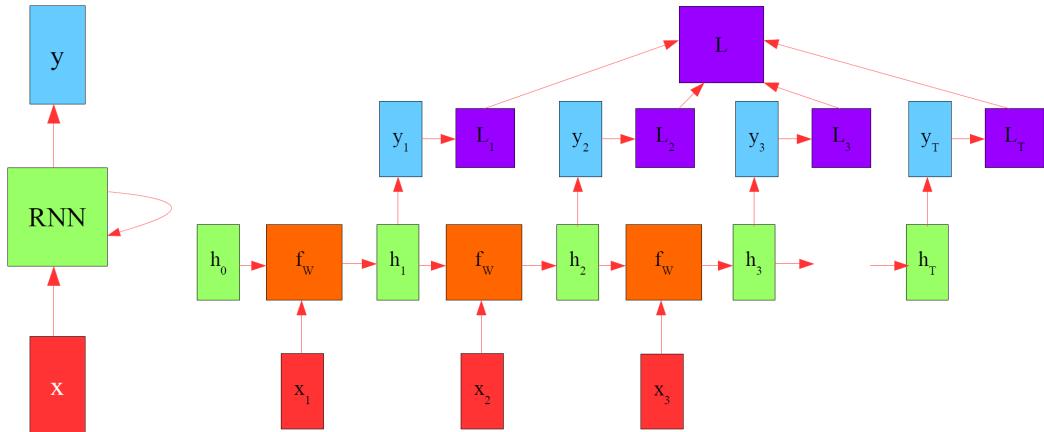
12.1. ábra. Különböző sorozatfeldolgozási feladatok.

12.2. Visszacsatolt neurális hálók

Könnyen belátható azonban, hogy az előrekesztő konvolúciós hálóknak memória eleme nincs, így nem igazán alkalmas időbeli sorozatok feldolgozására. Sorozatok hatékony feldolgozásához viszont egy olyan új háló struktúrára lesz szükségünk, amely valamilyen belső állappal is rendelkezik. Az ilyen hálózatokat visszacsatolt neurális hálózatoknak (RNN – Recurrent Neural Network) nevezzük. A visszacsatolt réteg működése során a belső állapotának aktuális értékét az aktuális bemenet és az egy lépéssel korábbi belső állapot értéke alapján számolja ki. A cella kimenete pedig a belső állapot imént kiszámolt aktuális értékétől függ. Egy RNN cella egyenlete az alábbi módon adódik:

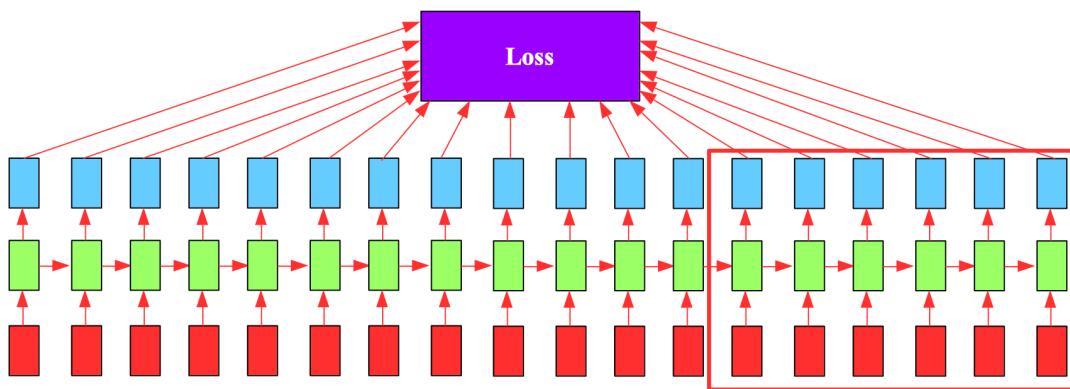
$$\begin{aligned} h_t &= \tanh(W_{hh}h_{t-1} + Wxhxt) \\ y_t &= \sigma(W_{hy}h_t) \end{aligned} \tag{12.1}$$

Ahol h a belső állapotot jelöli, t pedig az aktuális időpillanat. Könnyen belátható, hogy egy RNN cella tulajdonképpen három lineáris réteg és egy aktivációs függvény együtteseként adódik. Az új architektúra bevezetésével azonban felmerül az a kérdés, hogy hogyan lehet ebben a struktúrában a súlyok gradienseit meghatározni. Probléma ugyanis, hogy a backpropagation módszere visszacsatolt architektúrák esetén nem működik. Ez szerencsére azonban egy egyszerű trükkkel orvosolható: egy visszacsatolt háló ugyanis átalakítható egy hagyományos előrecsatolt hálóvá az időben történő kibontás műveletével. Ez azt jelenti, hogy az egyetlen RNN réteg különböző időpontokban felvett állapotára úgy tekintünk, mint egy hagyományos háló egymást követő rétegeire.



12.2. ábra. Egy RNN cella felépítése (bal) és időbeli kibontása (jobb).

Mivel egy RNN cellának minden időpontban van kimenete és hibája, ezért a kibontott háló minden rétegéhez fog tartozni egy-egy kimenet és hiba, melyeknek összege adja ki a teljes hibát. Innentől a már megismert backpropagation algoritmus minden további nélkül használható. Két fontos különbség adódik azonban a hagyományos előrecsatolt hálókhöz képest. Egyszerűen, ahogy haladunk előre az időben a kibontott háló mérete egyre növekszik, ami a tanítás folyamatának lassulásával jár. Ráadásul a helyes működéshez és tanításhoz nem szükséges a végtelenségig emlékezni a múltbeli bemenetekre. Éppen ezért a kibontás során a háló maximális méretét korlátozzuk és a legrégebbi réteget és bemenetet töröljük a kibontott hálóból.

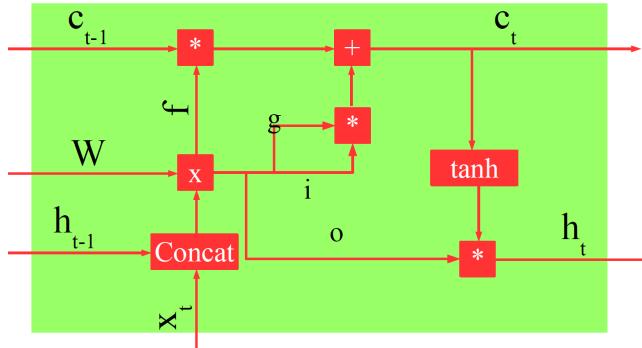


12.3. ábra. A Backpropagation through time (BPTT) algoritmus elve. Amikor a háló végén játunk, csak a piros kerettel jelölt részig végezzük el a visszaterjesztést, különben a probléma mérete a végtelenségig növekedne.

12.2.1. LSTM

A másik fontos különbség, hogy a kibontott sokrétegű háló esetén minden réteg súlymátrixa azonos (hiszen valójában egyetlen visszacsatolt rétegről van szó). Ez azt jelenti, hogy amikor a láncszabály segítségével a deriváltakat előállítjuk, akkor az egyes rétegek deriváltjainak egy hosszú szorzatát

kell kiszámolnunk. Ebben az esetben azonban a szorzat minden eleme azonos, vagyis valójában egy hatványról beszélhetünk. Az pedig könnyen beláthatjuk, hogy a gyakorlatban bármilyen szám vagy mátrix sokadik hatványa vagy nulla vagy végtelen, kivéve, ha az a szám pontosan 1. Ebből következik, hogy egy visszacsatolt cella gradiensei könnyedén eltűnnek, vagy „felrobbannak”, ami a tanítást ellehetetleníti.



12.4. ábra. Az LSTM cella felépítése.

Erre a problémára az egyetlen lehetséges megoldás az, ha olyan struktúrát alkotunk, ahol a belső állapot aktuális és egy lépéssel korábbi állapota közti derivált nagyjából egy. Pontosan ilyen architektúra az LTSM (Long Short-Term Memory) cella. A cella elnevezése onnan ered, hogy a készítői egy olyan rövidtávú memóriacellát kívántak alkotni, amely a gyakorlati használhatósághoz megfelelően hosszú ideig képes emlékezni az RNN cellával szemben. Míg az RNN cella három lineáris egységből állt, az LTSM négy aktivációs függvényt is tartalmazó egységből áll, melyeket kapuknak nevezünk.

Az LTSM cella működése során az egyes kapuk hatása nélkül a c cella állapot korábbi értéke változás nélkül átíródik az aktuális állapotba, így a kettő közötti derivált pontosan egy. A cella állapot értéke azonban még az egyes kapuk hatására módosulhat. Az első ilyen kapu a fejejtés kapu f , amely egy a cella állapottal azonos méretű vektor, melynek minden eleme nulla és egy között van a szigmoid nemlineárítás hatására. A cella állapot vektorát ezzel a vektorral elemenként megszorozva a cella állapot bizonyos részleteit elfelejtjük.

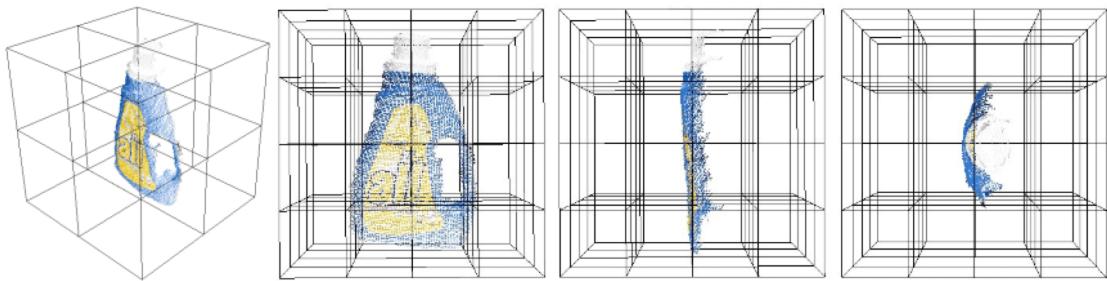
A következő kapu az úgynevezett főkapu g , amelynek feladata, hogy a bemenet aktuális értékéből és a cella kimenet korábbi értékéből kinyerje azokat a jellemzőket, amelyek a cella állapotban megjegyezhetők. Ezt követően a fejejtés kapuval analóg i bemeneti kapu vektorával a főkapu vektorát elemenként szorozzuk, ezáltal kiválasztva a megjegyezhető jellemzőkből a releváns részeket, majd ezt a cella állapothoz hozzáadjuk. A végső lépés a cella aktuális kimenetének előállítása, amire a cella állapotnak az o kimeneti kapu által szűrt értékeit adjuk ki.

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f * c_{t-1} + i * g$$

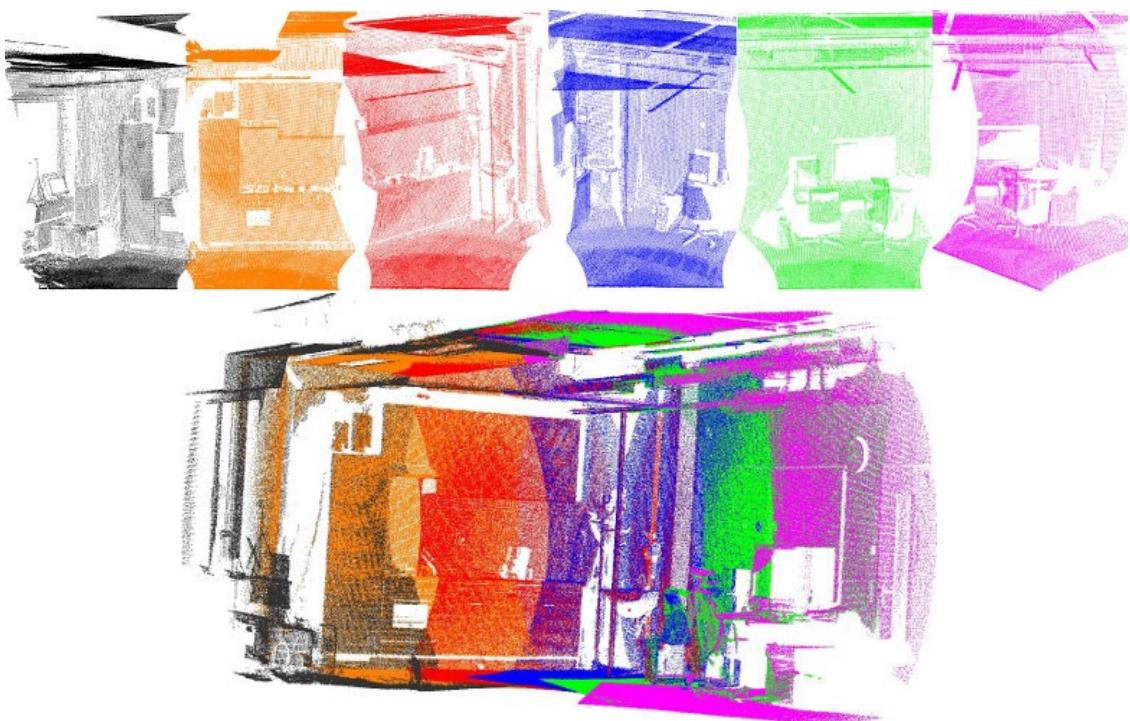
$$h_t = o * \tanh(c_t)$$
(12.2)

Ahol $*$ az elemenkénti szorzást jelöli. Érdemes megjegyezni, hogy az LTSM cellának számos apróságokban eltérő variációja létezik, valamint léteznek nagyobb eltérést mutató, de hasonló alapötletre épülő visszacsatolt cellák. Ilyen például az úgynevezett kapuzott visszacsatolt cella (GRU – Gated Recurrent Unit). Szintén érdemes észrevenni, hogy a gradiensek minél zavartalanabb hátrafele történő áramlásának elősegítése úgynevezett „rövidzár” kapcsolatok segítségével nem itt fordult elő először. Az előző fejezetben ismertetett reziduális blokk alapötlete ehhez rendkívül hasonló volt.



15.11. ábra. A globálisan leírandó objektum (bal) és a szóródási irányok alapján beforgatott változatai.

hogy ezeket a részleteket illesszük össze egyetlen a teljes teret leíró pontfelhővé. A regisztráció végrehajtásakor az átfedések detektálására gyakran alkalmaznak lokális jellemzőket.



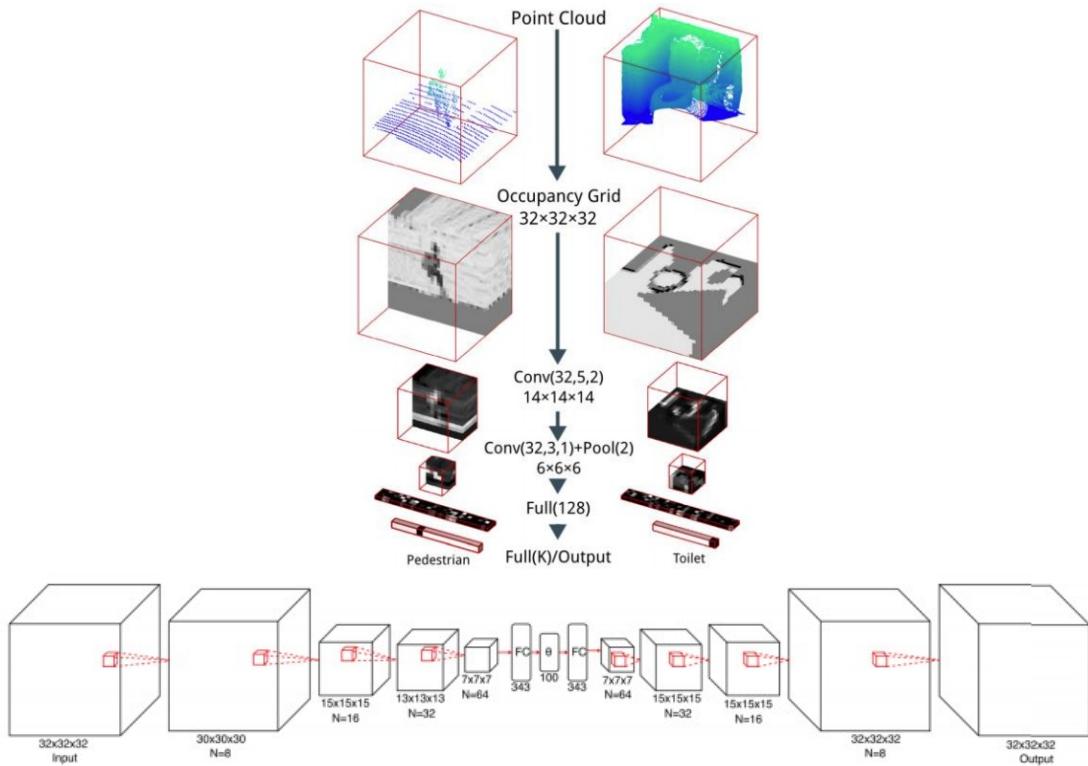
15.12. ábra. Pontfelhő részletek regisztráció előtt (felül) és után (alul).

15.6. 3D Deep Learning

A deep learning módszereinek alkalmazása alapvető nehézségekbe ütközik a 3D feldolgozás esetén, ami jelentősen megnehezíti ezen módszerek használatát.

15.6.1. Voxel hálók

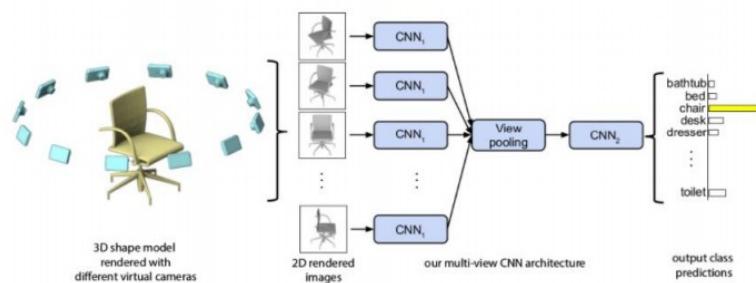
Adja ugyanis magát az ötlet, hogy használunk 2D konvolúciós hálók helyett 3D hálót voxelles adatokon. Azonban ahogy azt az előadás elején megbeszélünk, ez rendkívül problémás, mivel a tárolásnak nagy memóriaigénye van, ráadásul pazarló is. Ez a deep learning esetében különösen nagy probléma, mivel a mély neurális hálók már képek esetén is hatalmas memóriaigénytelivel rendelkeznek, ami az egyik legfőbb szűk keresztmetszetet jelenti a modern GPU-k alkalmazásánál. Ennek ellenére léteznek voxel neurális hálók, ezek azonban elég kicsi felbontáson működnek csak, így a teljesítményük is limitált.



15.13. ábra. Egy voxel alapú osztályozó (felül) és egy szegmentáló (alul) neurális háló.

15.6.2. Projekción alapuló hálók

Léteznek még neurális hálók, amik a 3D pontfelhőből véletlenszerűen generált nézőpontokból 2D vetületeket állítanak elő a vetítés egyenlete alapján, majd ezt egy hagyományos 2D konvoluciós háló segítségével osztályozzák. Ezzel visszavezethető a 3D osztályozás művelete 2D osztályozásra, azonban ez is jelentős lassulással járul, hiszen több képet kell végigfuttatni ugyanazon a hálón. Továbbá ezen módszerek igen nehezen terjeszthetők ki további problémák (detektálás, szegmentálás) megoldására.

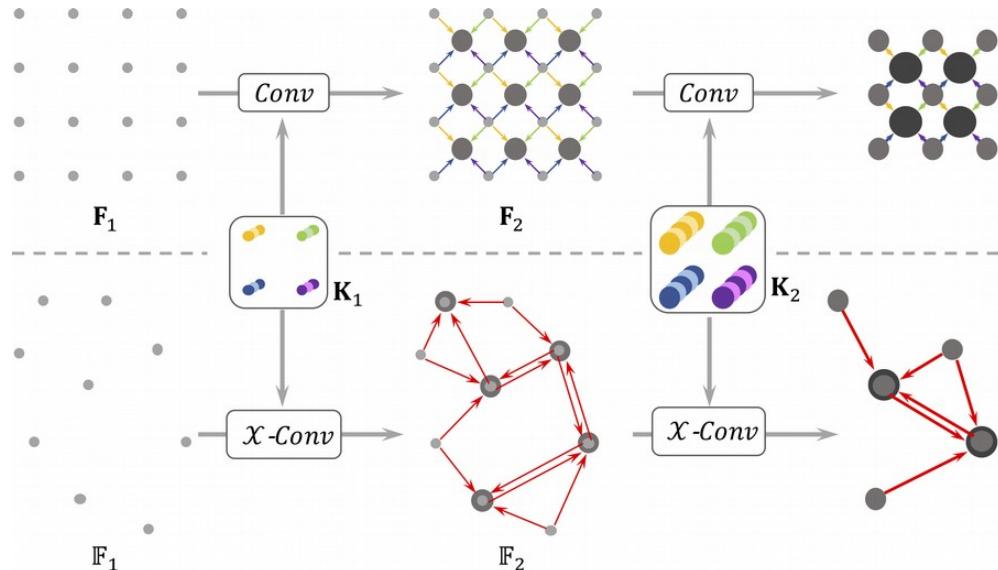


15.14. ábra. A Projekció alapú mély tanulás sémaja.

15.6.3. Pontfelhő hálók

Ha viszont megkíséreljük valamelyik kézenfekvőbb reprezentációs módszert alkalmazni, akkor hamar szembesülhetünk azzal, hogy ezek kevésbé foghatók meg jól konvoluciós hálók segítségével. Pontfelhők esetében például azzal a problémával szembesülünk, hogy a neurális hálók alapvetően rendezett adathalmazokon tudnak jól működni. Ha a pontfelhő felsorolásában megcserélek két pontot, akkor még minden ugyanazt a pontfelhőt írják le, egy hagyományos neurális háló viszont nem ugyanazt fogja végrehajtani.

Erre egy lehetséges megoldás, ha egy transzformáció segítségével a pontokat négyzetrácsos elrendezésbe hozzuk, majd ezt követően végezzük el a konvolúciót. Ez úgy képzelhető el, hogy minden pontnak megkeressük az egyes irányokba található legközelebbi szomszédját, majd ezeket tekintjük közvetlen szomszédnak a rácson. Az így elvégzett konvolúciót χ -Konvolúciónak nevezzük. A konvolúciót itt tovább módosíthatjuk úgy, hogy ne csak a szomszédos pontok/jellemzők értékeit, hanem azok távolságát is figyelembe vegye. Ezt a megoldást alkalmazza az úgynevezett Point-CNN architektúra.



15.15. ábra. A hagyományos konvolúció (felül) és a χ -konvolúció (alul).

További Olvasnivaló

- [1] R. Szeliski, *Computer Vision*. Springer London, 2011. doi: 10.1007/978-1-84882-935-0. [Online]. Available: <https://doi.org/10.1007%2F978-1-84882-935-0>.
- [38] J. L. Bentley, “Multidimensional binary search trees used for associative searching”, *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, Sep. 1975. doi: 10.1145/361002.361007. [Online]. Available: <https://doi.org/10.1145%2F361002.361007>.
- [39] R. Schnabel, R. Wahl, and R. Klein, “Efficient RANSAC for Point-Cloud Shape Detection”, *Computer Graphics Forum*, vol. 26, no. 2, pp. 214–226, Jun. 2007. doi: 10.1111/j.1467-8659.2007.01016.x. [Online]. Available: <https://doi.org/10.1111%2Fj.1467-8659.2007.01016.x>.
- [40] R. B. Rusu, N. Blodow, and M. Beetz, “Fast Point Feature Histograms (FPFH) for 3D registration”, in *2009 IEEE International Conference on Robotics and Automation*, IEEE, May 2009. doi: 10.1109/robot.2009.5152473. [Online]. Available: <https://doi.org/10.1109%2Frobot.2009.5152473>.
- [41] J. P. S. do Monte Lima and V. Teichrieb, “An Efficient Global Point Cloud Descriptor for Object Recognition and Pose Estimation”, in *2016 29th SIBGRAPI Conference on Graphics, Patterns and Images (SIBGRAPI)*, IEEE, Oct. 2016. doi: 10.1109/sibgrapi.2016.017. [Online]. Available: <https://doi.org/10.1109%2Fsibgrapi.2016.017>.
- [42] Y. Zhou and O. Tuzel, “VoxelNet: End-to-End Learning for Point Cloud Based 3D Object Detection”, in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, IEEE, Jun. 2018. doi: 10.1109/cvpr.2018.00472. [Online]. Available: <https://doi.org/10.1109%2Fcvpr.2018.00472>.

- [43] H. Su, S. Maji, E. Kalogerakis, and E. Learned-Miller, “Multi-view Convolutional Neural Networks for 3D Shape Recognition”, in *2015 IEEE International Conference on Computer Vision (ICCV)*, IEEE, Dec. 2015. DOI: 10.1109/iccv.2015.114. [Online]. Available: <https://doi.org/10.1109%2Ficcv.2015.114>.
- [44] Y. Li, R. Bu, M. Sun, W. Wu, X. Di, and B. Chen, *PointCNN: Convolution On X-Transformed Points*, 2018. eprint: 1801.07791. [Online]. Available: <http://www.arxiv.org/abs/1801.07791>.

IV. rész

Valósidejű Látás

16. fejezet

Hardverek

16.1. Bevezetés

Mivel a számítógépes látás során általában hatalmas adatmennyiség feldolgozását kell valós időben megvalósítani, ezért célszerű külön tárgyalni az egyes algoritmusok gyorsításának lehetőségeit. A folyamatosan növekvő processzorteljesítmény ugyan a fejlesztők oldalán áll, azonban általában több év szükséges a számottevő növekedés eléréséhez, amire a fejlesztés során nem lehet várni. Éppen ezért érdemes az algoritmusok gyorsításánál más megoldások felé tekinteni. Alapvetően a gyorsításnak három fő paradigmája létezik:

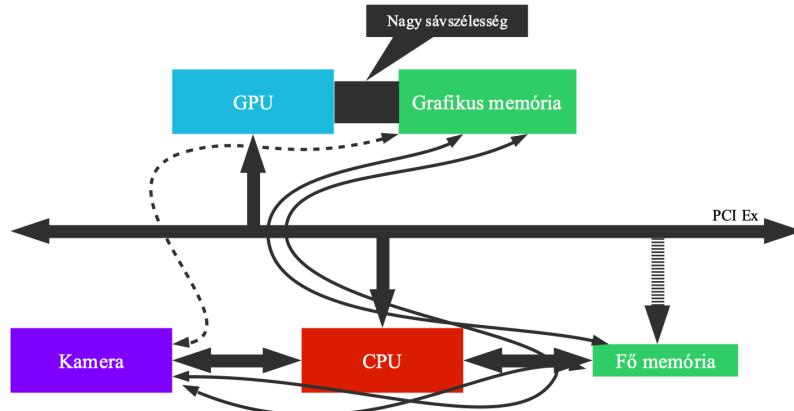
- **A feldolgozás szétosztása több processzorra:** ennek a módszernek a lényege, hogy a rendszerben több, független műveletvégző egység áll rendelkezésre, és emiatt lehetőség nyílik arra, hogy akár a feldolgozandó adatot, akár az elvégzendő műveleteket ezek között az egységek között szétosszuk.
- **Adatfolyam alapú feldolgozás:** ebben az esetben szintén nagy számú műveletvégző egység áll rendelkezésünkre, azonban ezek nem függetlenek, így a feladatokat nem tudjuk elosztani ezek között, az adatmennyiséget viszont igen.
- **Csővezeték technika:** a csővezeték technikát alkalmazó hardverek az egyes műveleteket részegységekre osztják, és azokat a sorban érkező adatokon párhuzamosan hajtják végre. Itt az egy egységgel előbb érkezett adat a feldolgozási lépésekben is egyetel előrébb jár. A csővezeték technika fontos tulajdonsága, hogy egyetlen adat feldolgozását nem gyorsítja meg a teljes feldolgozás átviteli sebességét viszont akár egy nagyságrenddel is növelheti.

16.2. Eszközök

Fontos megemlíteni, hogy a legtöbb speciális hardver eszköz a fent felsorolt paradigmák közül nem kizárolag egyet, hanem általában minden megvalósítja, csak éppen eltérő mértékben. A többmagos processzorok, számítógép hálózatok és szuperszámítógépek általában az első paradigmát valósítják meg. Tipikusan a Digitális Jelfeldolgozó Processzorok (DSP), a Celluláris Neurális Hálók (CNN), a grafikus processzorok (GPU), és a tenzor processzorok (TPU) tartoznak az adatfolyam feldolgozás területébe. A különböző programozható hardverek (FPGA) és alkalmazásspecifikus hardverek (ASIC) pedig a csővezetéktechnika által elért gyorsításra helyezik a hangsílyt.

A jelen előadás fókuszában a grafikus feldolgozó egységek (GPU-k) lesznek. Ezek tipikusan az általános célú processzoroknál kevésbé rugalmas eszközök, ugyanis számos feladat hardveresen huzalozva került bennük megoldásra, a teljesítményük akár több nagyságrenddel is jobb lehet. A feldolgozás sebességében ezek általában a programozható hardvereket és az ASIC megoldásokat alulmúlják, azonban ezeknek rugalmassága is kisebb.

GPU-kat minden esetben processzoros rendszerekben használunk. Itt a vezérő program futtatását minden esetben a CPU végzi, ez végzi a GPU vezérlését is. A kettő közti kommunikációt általában PCI-Express busz segítségével oldják meg, mivel ez egy meglehetősen nagy sávszélességű kommunikációs módszer. Ennek ellenére a két feldolgozó egység dedikált memóriaegységei közti adatmozgatás szokott lenni a grafikus feldolgozás egyik legfontosabb szűk keresztmetszete.



16.1. ábra. Egy GPU-t tartalmazó rendszer vázlata.

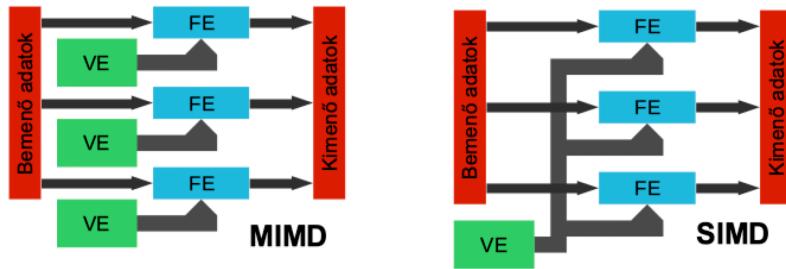
Nem egyértelmű talán első ránézésre, hogy miért is célszerű GPU-k alkalmazása a számítógépes látás feladataira, ugyanis ezek az eszközök első sorban grafikai feladatok elvégzésére lettek kialakítva. A számítógépes látás ugyan koncepcionálisan a grafika inverzének tekinthető, a két terült meglehetősen hasonlít az elvégzendő műveletek és a feldolgozandó adatok struktúráját illetően. Éppen ezért a GPU-k rendkívül jó támogatást nyújtanak a képfeldolgozás és a számítógépes látás műveleihez is.

16.2.1. Adatfolyam feldolgozás

A processzorok segítségével történő adatfeldolgozásnak három alapvető architektúrája létezik:

- **Single Instruction Single Data (SISD):** ez a feldolgozás lehető legegyszerűbb változata, ahol egyetlen feldolgozó egység végzi el az összes adat feldolgozását sorosan.
- **Multiple Instruction Multiple Data (MIMD):** ebben a megoldásban az adatokat több független vezérlőegységgel rendelkező feldolgozó egység végzi el, tipikusan a többmagos processzorok alkalmazzák ezt a megoldást.
- **Single Instruction Multiple Data (SIMD):** ebben a megoldásban szintén több feldolgozó egységünk van, azonban ezek nem függetlenek, hanem egyetlen közös vezérlő egységre csatlakoznak. Ennek következtében az egységek ugyanazokat műveleteket végeznek el, csak éppen eltérő adatokon. Az architektúra nagy előnye, hogy megegyező költség mellett lényegesen több műveletvégző egység elhelyezése megvalósítható. Ezen felül a SIMD architektúrák programozási paradigmája lényegesen egyszerűbb és kezelhetőbb, mint a MIMD esetben.

A jelen előadásban tárgyalt GPU-k és a később tárgyalandó TPU-k tipikusan a SIMD architektúrát alkalmazzák. A GPU-k esetében az adatfolyam feldolgozást fejlett memóriakezelés, nagy sávszélességű (4096 bit!) memóriák, és csővezeték feldolgozás is segíti. A vezérő egységek tipikusan kevésbé fejlettek, kevesebb "spekulációt" végeznek (feltételek jóslása stb.) Fontos eleme a GPU-knak, hogy a globális kártyamemorián kívül az egyes feldolgozó egységek saját regiszterekkel és fejlett cache rendszerrel is rendelkeznek. Érdemes még megjegyezni, hogy a modern processzorok szintén alkalmazzák ezt az elvet: a SWAR (SIMD Within A Register) megoldások lehetővé teszik, hogy egy 64 bites ALU-val rendelkező egység egyszerre 8 db char változóval végezzen műveletet például.



16.2. ábra. A MIMD (bal) és a SIMD (jobb) architektúrák.

A GPU-k támogatják továbbá a megegyező architektúrájú kártyák közötti multiprocesszor-jellegű feladat szétosztást is az NVLink technológia segítségével.

Az adatfolyam feldolgozás módszere tehát alapvetően akkor használható jól, amennyiben nagy mennyiségű adaton kell ugyanazt a műveletet elvégezni. Nem érdemes véletlenszerű feldolgozásokra, például szerverek, vagy adatbázisok kiszolgálására alkalmazni őket. Tipikusan könnyen megvalósítható művelet a Map (letérképezés), amikor egy tömb elemein kell függetlenül ugyanazt a műveletet végrehajtani. A redukció művelet során a Map függvényleképezés után az eredményeket egyetlen számba redukáljuk (általában összegzés segítségével). Hasonlóan gyakori a különböző adatszűrések, rendezés és keresés megvalósítása. Érdekes példák még a Gather-Scatter típusú műveletek. Ebben a két esetben úgy olvasunk(Gather)/írunk(Scatter) egy tömbbe, hogy egy másik, kisebb tömbből olvassuk ki az olvasás/írás indexeit.

16.3. GPU Architektúra

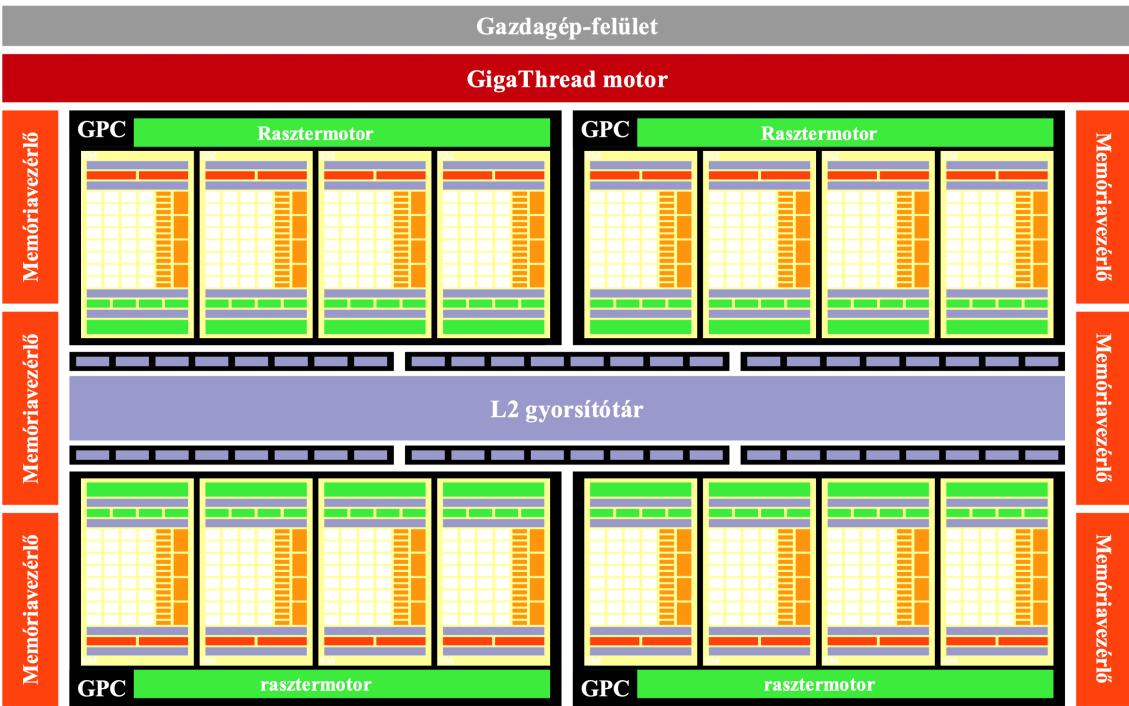
A GPU-k általános felépítése egy közös PCI interfészről és ütemező egységből áll. A globális memóriát több független memóriavezérlő is elérheti, ezek azonban egyetlen közös L2 cache rendszerrel dolgoznak. A GPU feldolgozóegységei klaszterekbe (GPC - Graphics Processing Cluster) rendeződnek, melyeknek saját riaszterizálója van, aminek elsősorban grafikus alkalmazásoknál van szerepe. A GPC-k általában 4-8 úgynevezett Streaming Multiprocessor-ból (SM) állnak, amelyben már konkrét feldolgozóegységek találhatók

16.3.1. Streaming Multiprocesszor

A Streaming Multiprocesszor a GPU legfontosabb részegysége, az ebbe tartozó magoknak ugyanis egyetlen vezérlő egysége van. Architektúrától függően egy SM-be 32/64 mag tartozik, melyeknek saját regiszterei vannak. Az SM része még továbbá néhány Special Function Unit (SFU), melyek különböző matematikai műveleteket (szögfüggvények, log, exp) képesek hardveresen megvalósítani. Az SM-nek külön dedikált memóriaolvasó/író úgynevezett Load/Store egységei is vannak, amelyek képesek a globális memóriából a szükséges adatokat a feldolgozártól függetlenül beolvasni. Ezekre azért lehet szükség, mert a globális memória válaszideje akár több száz gépi ciklus is lehet a rengeteg párhuzamos elérés következtében.

Fontos eleme még a Streaming Multiprocesszornak az úgynevezett osztott memória, amely egy olyan memória egység, amelyet az SM összes magja képes elérni (a regiszterek az egyes magokhoz tartoznak). Ezen felül az SM tartalmaz még különböző L1 gyorsítótárat, melyekből létezik külön az utasításokra és textúrákra szakosodott. Tartalmaz ezen felül külön textúra feldolgozó egységeket és grafikus céláramkörököt, melyeknek első sorban grafikai alkalmazások során van szerepe.

A GPU-k és az SM felépítése változott valamelyest az évek során. Maga a GPU leginkább csak a belepakolt SM-ek számában mutat érdemi eltérést, a Streaming Multiprocesszorok belső felépítése azonban néhány lényeges változásban ment keresztül. Ezek közül a legfontosabb, hogy a GPU-k ma már külön magtípusokat tartalmaznak az egyes számtípusok számára. A legfőbb feldolgozó egység 32 bites lebegőpontos számokon működik, azonban ugyanezek a magok a korábban ismertetett



16.3. ábra. Egy GPU általános felépítése.

SWAR elven képesek 16 bites half-precision számok feldolgozására is, amely mély neurális hálók esetén különösen hasznos, itt ugyanis a pontosság kevesebbet számít. Bizonyos architektúrák tartalmaznak 64 bites double egységeket, valamint egész (és fixpontos) aritmetikához használatos magokat is.

16.3.2. Tensor Core

Az egyik legfontosabb újítás ezekben az egységekben az úgynevezett TensorCore, amelyet a Volta és Turing architektúrájú GPU-kban találhatunk meg. Ezek az egységek mátrixszorzásra dedikált áramkörök, minden egyes tenzor mag a következő művelet hardveres megvalósítására képes:

$$C = C + A * B \quad (16.1)$$

Ahol az összes operandus egy 4x4 méretű lebegőpontos mátrix. A tenzor mag nagy előnye, hogy a GPU alapvetően vektor műveletekre lett kifejlesztve, így a mátrixokon végzett operációkat szoftveresen kellett összerakni. A tenzor magok segítségével azonban a mátrixszorzás (és következetében a különböző mély neurális hálók futtatása és tanítása) sebessége egy nagyságrenddel növekedett.

16.4. Programozási modell

Az előző előadás során tárgyaltuk a GPU-k fizikai felépítését, valamint a futási modelljüket, azonban a programozásuk módjáról nem beszélünk. Éppen ezért a jelenlegi előadás témája a GPU programozásának lehetősége lesz, ezek közül is a CUDA nyelvre fókuszálva.

A GPU programozási lehetőségeit két kategóriába oszthatjuk: ezek közül az első a különböző grafikus programozási nyelvek, a másik pedig az általános célú GPU (GPGPU) programozási nyelvek. Előbbiek közül a két legfontosabb megoldás az OpenGL és a Direct3D. Ezek első sorban grafikai alkalmazásokra kifejlesztett megoldások, azonban egyszerűbb képfeldolgozási feladatokra (szűrések, küszöbözős, színtér konverzió) rendkívül jól használhatók. Az OpenGL a legtöbb operációs



16.4. ábra. A 2010-es Fermi (bal) és a 2017-es Volta (jobb) architektúrák SM-jei.

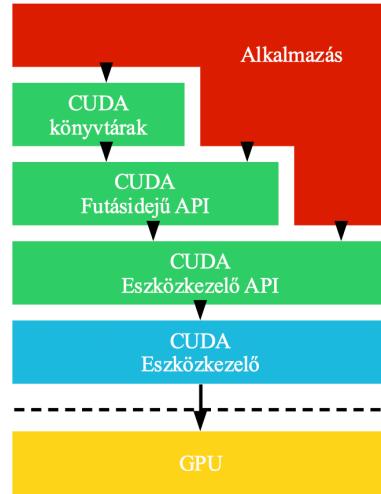
nyelven elérhető könyvtárként, és a GLSL nyelvet használja, amely a C++ kiegészítése. Hasonló ehhez a Direct3D által használt HLSL nyelv, azonban ez csak Windows operációs rendszer alatt használható, mivel a Microsoft fejleszti.

Jelen fejezetben sokkal érdekesebbek azonban számunkra a GPGPU programozási nyelvek, vagyis a CUDA és az OpenCL. A CUDA fejlesztését az NVIDIA végzi, ebből kifolyólag csak az ő kártyáikon érhető el. A CUDA egy homogén feldolgozó egységekre alapozó hardvert feltételez, ahol minden eszköz ugyanazt a műveletet végzi el. A CUDA nyelv alapvetően a C/C++ nyelvre épül, ehhez ad hozzá néhány kiegészítést. Egy CUDA programban az egy GPU mag által futtatott programrészletet (kernelt) kell megírni, ez kerül végrehajtásra minden magon. A CUDA-nak saját fordítója van, amely nvcc névre hallgat.

Az OpenCL (Open Computing Language) ezzel szemben egy nyílt forráskódú, minden GPU-n működő nyelv. Az OpenCL elméletben heterogén eszközökön is működhet, a gyakorlatban azonban ezt nem szokták használni. A CUDA-hoz hasonlóan a C/C++ nyelv kiegészítéseként készül és saját fordítóval rendelkezik, ez azonban GPU gyártónként különböző. A program elkészítésének filozófiája is megegyezik a CUDA-val. A másik lényeges különbség, hogy az OpenCL programok futásidőben is fordulhatnak.

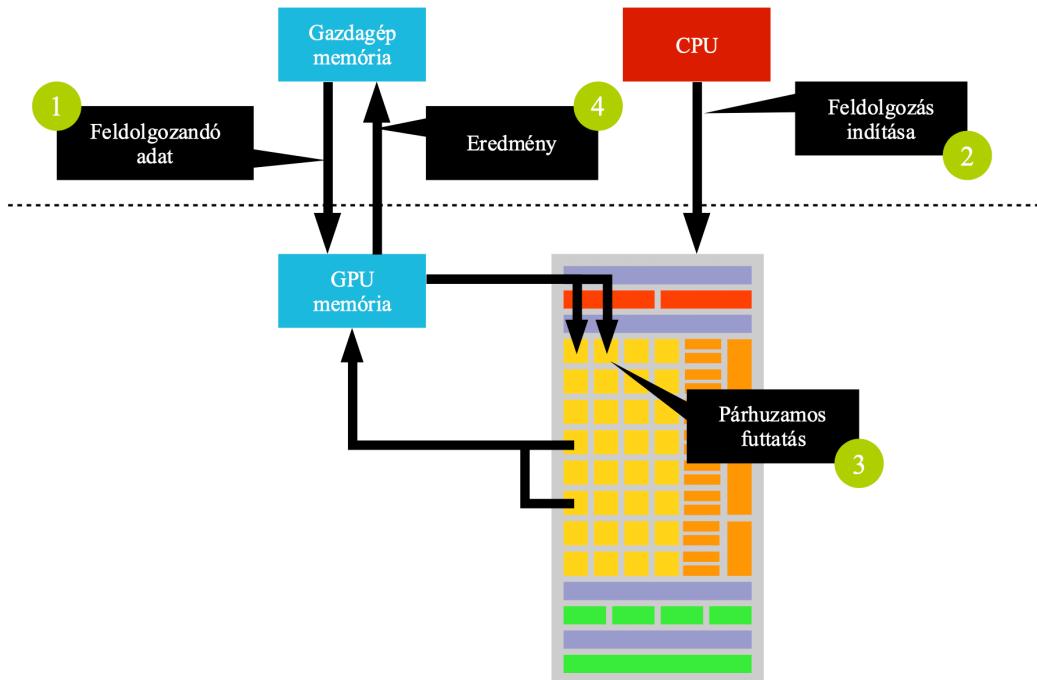
A CUDA nyelv egy több szintból álló architektúra. Ennek legalacsonyabb szintjén a CUDA eszközkezelő (driver) áll, amelyik a GPU-n történő számítás vezérléséért felel. Ehhez a driverhez létezik programozási felület (API), azonban ez egy rendkívül alacsony szintű hozzáférési lehetőség, amelyet csak a legritkább esetben használunk. Leggyakrabban a CUDA futásidőjű API-ját használjuk, valamint az arra épülő CUDA könyvtárakat. Fontos megjegyezni, hogy a driver és a futásidőjű API használata kölcsönösen kizártják egymást.

Ahhoz, hogy elkezdhetünk a GPU-n programozni, vizsgáljuk meg először, hogy az általunk írt program hogyan fut a GPU-n. Mint már említettük, a feldolgozást a CPU vezérli, és a feldolgozandó adat is a gazdagép memóriájában található. Első lépében az adatot át kell másolni a GPU



16.5. ábra. A CUDA szintjei.

saját memoriájába, majd a feldolgozást kell elindítani. Mivel a két egység közti adatmozgatás és kommunikáció költséges, ezért ezeket célszerű minimalizálni a sebességnövelés végett. Ezt követően a GPU elvégzi a párhuzamos program futtatását, és a végeredményt visszamásolhatjuk a gazdagép memoriájába.



16.6. ábra. A GPU program futásának módja.

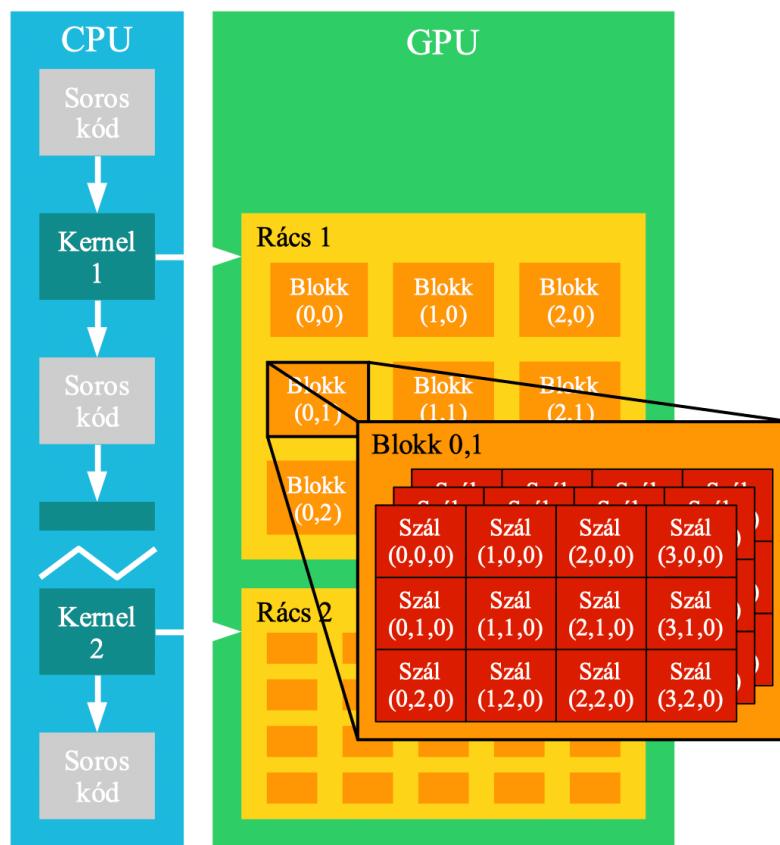
16.4.1. Futási modell

A GPU-n futó programot kernelnek hívjuk, melyből a legtöbb architektúra esetében egyszerre egyetlen futhat. A kernel program alapegyisége a szál, amely az egyetlen magon és egyetlen adategységen futó program. A szálakat blokkokba rendezzük, míg a blokkokat pedig egy rácsba (grid). A kétszintű rendezés oka az, hogy az ugyanabban a blokkban található szálak garantáltan ugyanazok az SM-en kerülnek végrehajtásra, addig a rácson található blokkok tetszőlegesen kerülnek

szétosztásra az SM-ek közt. Ez különösen fontos, ugyanis ennek következtében az egy blokkban található szálak ugyanazt a közös memóriát látják (hiszen ez fizikailag az SM-ben található), míg a különböző blokkok eltérő közös memóriát látnak.

További különbség ezen felül, hogy a blokkok mérete véges: maximum 1024 szálat tartalmazhatnak, míg a rácsban akármennyi blokk lehet. Ezen felül a blokkban a szálakat rendezhetjük egy, két és három dimenzióba, addig a rács maximum két dimenziós lehet. Ennek fizikai szempontból nincs jelentősége, a többdimenziós rendezés azonban nagymértékben megkönnyíti majd a programozás feladatát.

A futási modell utolsó fontos egysége a warp (fonat), amely az adott SM-en egyszerre futó 32/64 szálat jelképezi. Természetesen egy blokk több (maximum 32) fonatból állhat, de egyszerre csak 32/64 szál tud futni. Ez az egység azért fontos, ugyanis ha egy fonaton belül divergens kód adódik (vagyis egyes szálaknak más utasítást kellene végrehajtani), akkor a többi szál addig kénytelen várni, amíg ez a kódrészlet lefut. Éppen ezért a divergens kódot (pl. if-else) a fonaton belül érdemes kerülni.



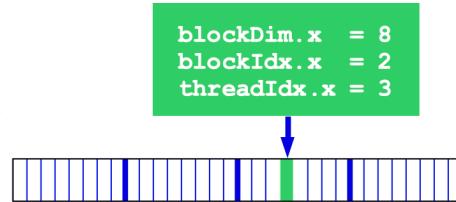
16.7. ábra. A CUDA futási modellje.

Rendkívül fontos speciális változók a kernel index változók, melyek az általunk írt kernelből elérhetők. Ezek az értékek megadják, hogy az adott szál éppen hányadik blokk hányadik szála. Ezek segítségével lehet általános képlet útján megadni, hogy az adott szál a feldolgozandó adattömb hányadik elemét dolgozza fel. Ehhez rendelkezésre állnak a rács és a blokk méretét jelző változók is.

```

1 dim3 gridDim; // rács dimenziója blokkokban (max 2D)
2 dim3 blockDim; // blokk dimenziója szálakban (max 3D)
3 dim3 blockIdx; // 32 blokk azonosítója a griden belül
4 dim3 threadIdx; // 64 szál azonosítója a blokkon belül
5 dim3 warpSize; // 32 fonat mérete (jelenleg mindig 32)
```

Egy tipikus példa ezen változók használatára az alábbi függvény, amely egy tömb minden elemét egy adott konstans értékre állítja.

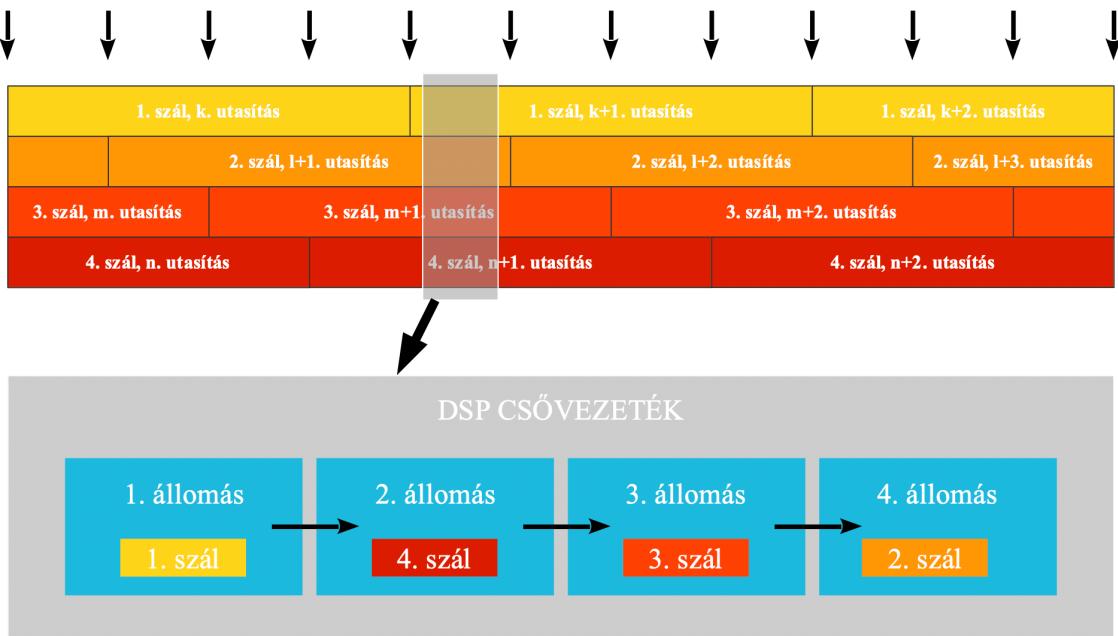


16.8. ábra. A kernel index számítása.

```

1 __global__ void assign( int* d_a, int value )
2 {
3     int idx = blockDim.x * blockIdx.x + threadIdx.x;
4     d_a[ idx ] = value;
5 }
```

Érdemes megjegyezni, hogy az SM-nek számos regisztere van, így a fonatok közti taszk váltást nulla többlet teher mellett képes elvégezni, sőt általában egy SM több blokkot is tud könnyedén futtatni. Fontos azt is tudni, hogy az egyes CUDA magok a DSP egységekhez hasonló csővezetéktechnikával rendelkeznek, vagyis képesek több fonatot is párhuzamosan futtatni így, hogy a csővezeték állomásai nem az ugyanannak a programnak az egymás után következő lépéseiit, hanem a különböző szálak lépéseit töltik be.



16.9. ábra. A csővezeték többszálás futásra való használata.

16.4.2. Kommunikáció

Fontos említeni még a GPU-n futó szálak közti kommunikáció megoldásáról is, ez ugyanis elengedhetetlen a hatékony programok írásához. Ezek közül az első a blokkon belüli szinkronizációs gát:

```

1 __shared__ int scratch[blocksize];
2 scratch[ threadIdx.x ] = value;
3 __syncthreads_()
4 leftValue = scratch[ threadIdx.x - 1 ]
5 }
```

Ezt a módszert első sorban akkor szoktuk használni, amikor a blokkon belül minden szál valami-lyen a többi szál számára releváns információt ír az osztott memóriába, ami a sorban következő művelethez már szükséges. Ilyenkor létfontosságú elérni, hogy a blokkon belül minden szál elérjen idáig mielőtt bármelyik másik szál továbbhaladhatna.

A korábban említett atomi memóriaműveletek szintén alkalmazhatók szálak közti kommunikáció megvalósítására, hiszen ezek garantáltan végrehajtónak. Tipikus példa erre például a skaláris szorzás implementációja. Ekkor minden szál kiszámolja az összeg egy tagját, majd ezt atomi memóriaműveletekkel ugyanahoz az osztott memóriában tárolt számhoz adják hozzá.

Fontos megjegyezni, hogy a CPU oldali program a kernel indítása után aszinkron módon tovább halad, ezért az eredmények felhasználása előtt szükséges lehet a szinkronizálást explicit módon elvégezni az alábbi módon:

1 `cudaThreadSynchronize()`

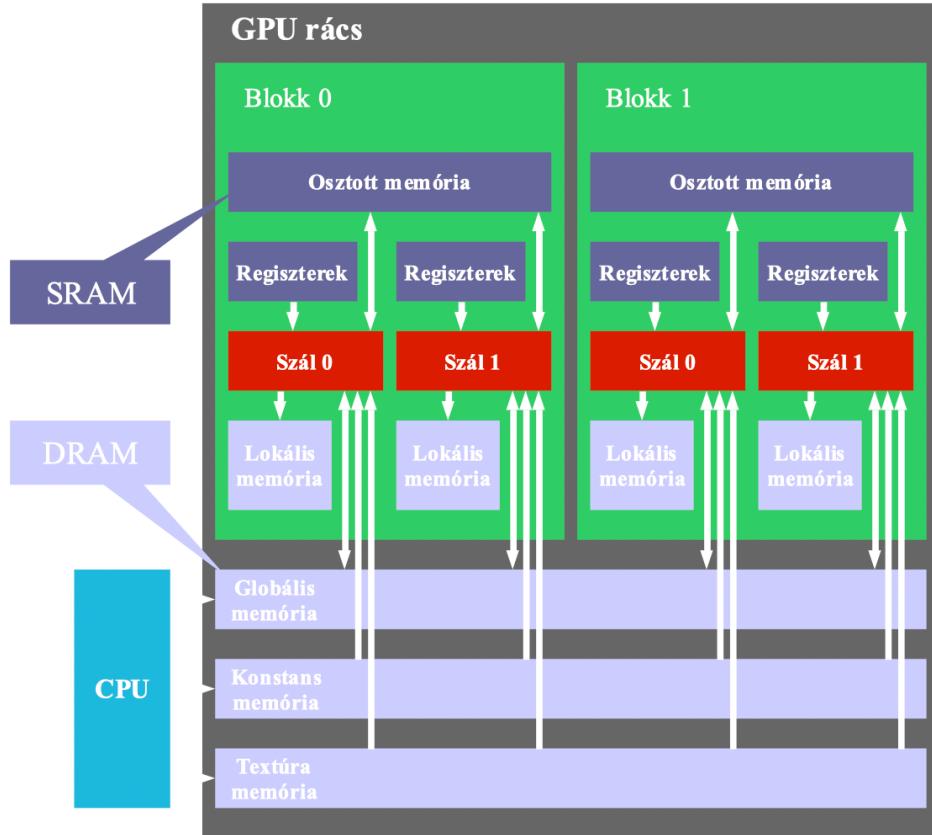
Ez a függvény az éppen futó kernel összes szálának befejezését megvárja. Fontos tudni, hogy újabb kernel indítása előtt ez automatikusan meghívódik, így akkor nem szükséges ezt explicit módon meghívni.

16.4.3. Memóriakezelés

Fontos még tárgyalni a GPU programozás során rendelkezésre álló memóriatípusokat és azok cél-szerű használatát. A GPU memória ugyanis a feldolgozás egyik legfontosabb szűk keresztmetszetét jelentheti, a számos memória opción pedig első ránézésre lehengerlő lehet. A GPU-k az alábbi memória típusokat különböztetik meg:

- **Regiszterek:** ezek szálanként külön privát memóriaterületek, amelyek fizikailag az SM-ben találhatók, és statikus RAM-ként vannak megvalósítva, vagyis a leggyorsabb memóriatípusról beszélhetünk. Alapesetben a kernelek lokális változói itt kerülnek eltárolásra.
 - **Lokális memória:** amennyiben a regiszterek elfogynak a további privát adat itt kerül tárolásra. A lokális memória azonban nevével ellentétben a nem az SM-ben található, ezért fizikailag messze van, ráadásul dinamikus RAM, vagyis meglehetősen lassú típusról van szó.
 - **Osztott memória:** az osztott memória szintén statikus RAM, ami az SM-en található, így rendkívül gyors. Ezt a memóriatípust azonban a blokkban lévő összes mag látja, ezért a bonyolultabb elérési és cachelési megoldások miatt valamivel lassabb (főleg írni). Ez a memóriatípus kiválóan használható magok közti kommunikáció megvalósítására, amennyiben elég a blokkon belül kommunikálni.
 - **Globális memória:** a globális memória a GPU kártyán külön chipként rendelkezésre álló dinamikus memória. Ez az a memóriatömb, amely a GPU kártyák adatlapján fel van tüntetve, mérete általában 2-16 GB. A legtöbb esetben rendkívül nagy sávszélességű, ennek ellenére az elérése rendkívül lassú tud lenni, amennyiben több ezer szál kíván egyszerre olvasni belőle.
 - **Konstans memória:** a konstans memória szintén a dinamikus memória, azonban csak olvasható (szoftveresen). Ennek előnye, hogy a többszásás működés miatt rendkívül bonyolult írás cache kezelés elhagyható, így a működése valamivel gyorsabb, mint a globális memóriáé.
 - **Textúra memória:** a textúra memória egy speciális konstans memória, azonban számos hasznos funkciót szolgáltat még számunkra. Ezek közül az egyik, hogy lehetővé teszi számunkra azt, hogy egy-, két- vagy háromdimenziós indexelést használunk, és az indexek memóriacímmé konvertálását hardveresen végzi. Ezen felül lehetőség van normalizált (vagyis 0 és 1 közötti) indexelés alkalmazására. Ennek külön előnye, hogy kérhetjük a textúra memóriától azt, hogy ha pont két pixel közé indexeltünk, akkor ezeket súlyozva átlagolja össze (konvolúciós szűrés), azonban ez is hardveresen történik.
- A textúra memória ráadásul képes 2 vagy 3 dimenziós asszociatív cache megvalósítására is. Ez azt jelenti, hogy ha egy érték bekerül a cache-be, akkor a textúra memória a környezetét

is automatikus beolvassa a cache-be. Ezt azonban nem a memóriacímek alapján, hanem a 2, vagy 3D környezet alapján teszi meg. Ezen felül lehetőség van a textúrából történő kiindexelésre is hiba nélkül. Ekkor több opció közül választhatunk, lehetséges például konstans értéket kérni, vagy akár ismétlődést is.



16.10. ábra. A CUDA memóriamodellje.

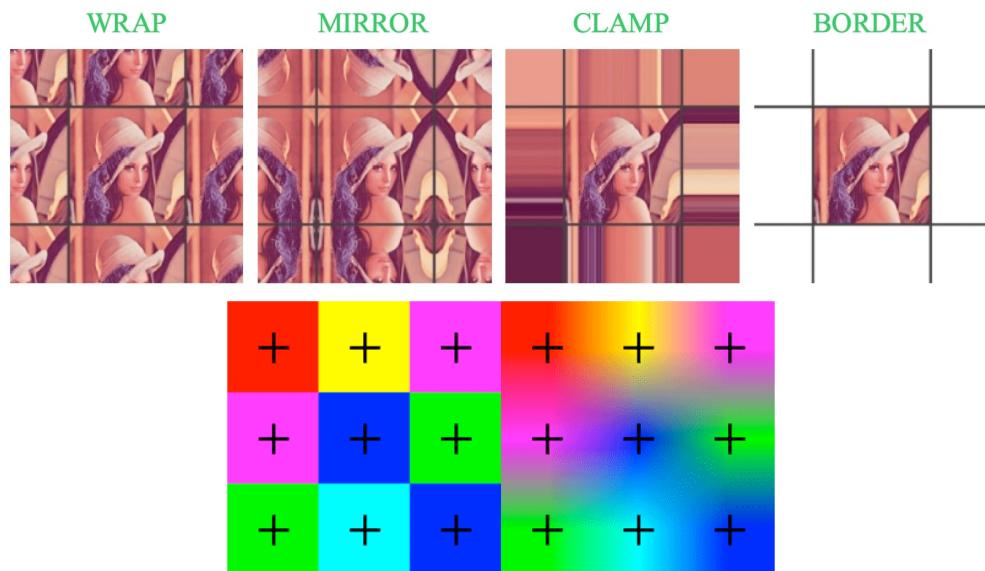
Érdemes még a közös memóriatípusok egyszerre írásáról röviden említést tenni. Az osztott és a globális memória ugyanis alapvetően másféleképpen működnek. Míg a globális memóriában több párhuzamos írás esetében minden írás garantáltan megtörténik (a sorrendjük ugyan nem garantált), addig az osztott memóriában csak egyetlen írás történik meg garantáltan.

16.4.4. Textúrák használata

Külön említést igényel a textúra memória használata, tekintve, hogy számos fontos beállítást/döntést meg kell hoznunk ezek használatához. Fontos tudni, hogy a textúra memóriát CPU oldalról kell definiálni, mivel GPU oldalról ez egy konstans memóriaként viselkedik. A textúra memóriának a következő beállításai lehetnek:

- **Adattípus:** a textúra egyetlen elemének típusa: ezek lehetne 1-4 dimenziójú vektorok, melyek egész, vagy lebegőpontos számok
- **Dimenzió:** a textúra (és a cachelés) dimenziója (1,2,3)
- **Olvasási mód:** ez a textúra indexelésének típusa: lehet egyszerű tömbindexelés, vagy normalizált, amely esetben a textúra két széle között egy [0,1] tartományba eső lebegőpontos értékkel címzünk.

- **Interpoláció:** beállítható különböző hardveres interpoláció a normalizált indexelés esetén: Ez lehet legközelebbi szomszéd, amely esetben nem történik szűrés. Választhatunk azonban bilineáris interpolációt is.
- **Határkezelés:** amennyiben kiindexelünk a tömbből választhatjuk az, hogy az olvasás egy előre meghatározott konstanssal, vagy a legközelebbi ténylegesen a tömbbe tartozó értékkel térjen vissza. Választhatjuk azt is azonban, hogy a textúrakezelő vegye úgy, mintha a tömb a határokon túl periodikusan ismétlődne, vagy ugyanezt tükröződő ismétlődéssel.



16.11. ábra. A különböző határkezelési (felül) és interpolációs (alul) opciók.

További Olvasnivaló

[45] *CUDA Programming*. Elsevier, 2013. doi: 10.1016/c2011-0-00029-7. [Online]. Available: <https://doi.org/10.1016%2Fc2011-0-00029-7>.

17. fejezet

VPU, TPU, FPGA

17.1. Bevezetés

A korábbi előadások során betekintést nyerhettünk a grafikus feldolgozóegységek működésébe és programozásába. A GPU-k azonban még mindig alapvetően általános célú eszközöknek tekinthetők, amelyekben bizonyos műveletek lettek hardveresen megvalósítva. A jelen előadás során megismerkedünk először a tensor feldolgozó egységekkel (TPU), melyek kifejezetten mátrixműveletek gyorsítására szolgáló egységek, valamint a programozható hardverek világával.

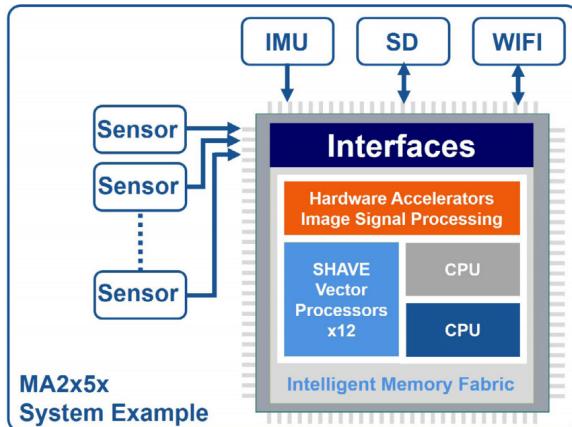
17.2. Vision Processing Unit

Fontos említést tenni a számítógépes látás esetében gyakorta használt Vision Processing Unit (VPU) eszközökről. Ezek általában alacsony fogyasztású néhány RISC architektúrájú maggal, valamint ezen felül számos további speciális párhuzamos feldolgozó egységgel rendelkező eszközök. Ezek a párhuzamos feldolgozók általában alacsony pontosságú lebegőpontos (16/32 bit), vagy fixpontos (8/16/32 bit) műveletek elvégzésére képesek, melyek működését előre huzalozott műveletvégzők is segítik. Ezek a feldolgozók tipikusan VLIW (Very Large Instruction Word) utasításkészlettel rendelkeznek, ami elősegíti a műveletek közti párhuzamosítást.

Ezen felül a VPU-k rendelkeznek lokális, a chipen elhelyezkedő memóriaterülettel is, ami a számítási részeredményeinek hatékonyabb kezelését teszi lehetővé. Végezetül a VPU-k általában számos, digitális kamerák által gyakran használt interfész is támogatnak. A fentiek alapján a VPU-k felépítése meglehetősen hasonlít az általánosabb célú DSP-re, ezek azonban tipikusan nagyobb pontosságú számítások elvégzésére alkalmasak.

17.3. Tensor Processing Unit

Fontos megérteni, hogy a GPU-k alapvetően vektoros műveletek elvégzésére lettek kifejlesztve (ez alól kivételt képez a tensor mag). Ezeknek alapvető tulajdonsága, hogy a bemeneteket egyszer használjuk fel egyetlen kimenet írásához. Mátrixműveletek esetén azonban az összes bemenetet több művelethez is fel kell használnunk, ráadásul a legtöbb kimeneti változónkat minden műveletvégzés esetében akkumulálnunk kell. Ehhez a GPU programozás során osztott memórián definiált atomi műveleteket kell alkalmaznunk, amik a kernel működését lassítanák. Mivel a mély neurális hálók tanítása és futtatása számos nagy méretű mátrix szorzást igényli, ezért célszerű lenne erre szakosodott hardvert készíteni.



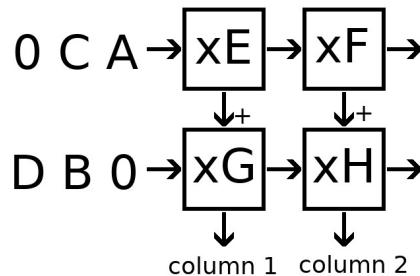
17.1. ábra. Egy tipikus VPU felépítése.

17.3.1. Systolic Array

Ehhez első körben vizsgáljuk meg a mátrixszorzást a legegyszerűbb, 2x2-es esetben. A képlet az alábbi:

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} * \begin{pmatrix} E & F \\ G & H \end{pmatrix} = \begin{pmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{pmatrix} \quad (17.1)$$

Ez szemléletesen megvalósítható négy szorzó és két összeadó egység segítségével az alábbi módon:



17.2. ábra. A mátrixszorzás elvégzése.

Itt a szorzó egységek fixen egy számmal szorozzák a bemenetükre sorban érkező adatokat. A futás végére a szorzat mátrix elemei a kimeneteken oszlopokként állnak elő. Itt érdemes megjegyezni, hogy ez a módszer természetesen a kimenetén néhány részeredményt is kiad, összesen azonban 4 ciklus alatt a mátrix összes eleme előáll.

Ezt a végrehajtási architektúrát szisztematikus tömbnek nevezzük. Itt a szisztematikus kifejezés a térbeli párhuzamosítás és a csővezeték-technika együttes intenzív alkalmazását jelenti. Érdemes észrevenni, hogy az alapból köbös műveleti igényű mátrixszorzás ezzel a módszerrel lineáris időben elvégezhető, ami kifejezetten nagy mátrixok esetében jelent hatalmas gyorsulást.

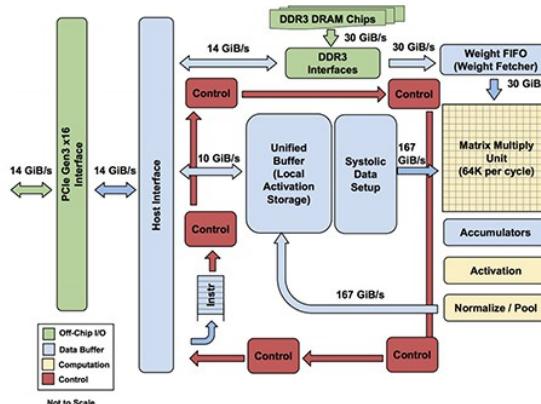
17.3.2. Felépítés

A TPU felépítésének központi eleme a mátrixszorzó egység, amely az imént ismertetett alapelveken működik. A mátrixszorzó egységek külön súly gyorsítótár egysége van, amely csak a neurális háló súlyainak betöltéséért és a mátrixszorzónak való átadásért felelős. A mátrixszorzó egységet a különböző aktivációs függvények huzalozott megvalósítását kínáló aktivációs egységek, és a különböző lesklálázások és normalizációk hardveres gyorsítását kínáló norm/pool egységek követik.



17.3. ábra. A szisztolikus mátrixszorzás (bal) és a TPU mátrixszorzó egységének felépítése (jobb).

Érdemes még kiemelni, hogy a normalizáló egységek kimenete és a bemeneti adatok/köztes aktivációk tárolásáért felelős egységes buffer között nagy sávszélességű buffer található. Ezt a buffert a mátrixszorzó egységgel egy szisztolikus előkészítő egység köti össze, melynek feladata, hogy az adatot a mátrixszorzáshoz megfelelő formátumba rendezze. Érdemes megjegyezni azt is, hogy míg a bemeneti adatot általában a gazdagép küldi a TPU számára, addig a súlyok lokálisan tárolódnak. Ennek oka, hogy az első generációs TPU alapvetően inferencia (háló futtatás) feladatokra lett kialakítva.



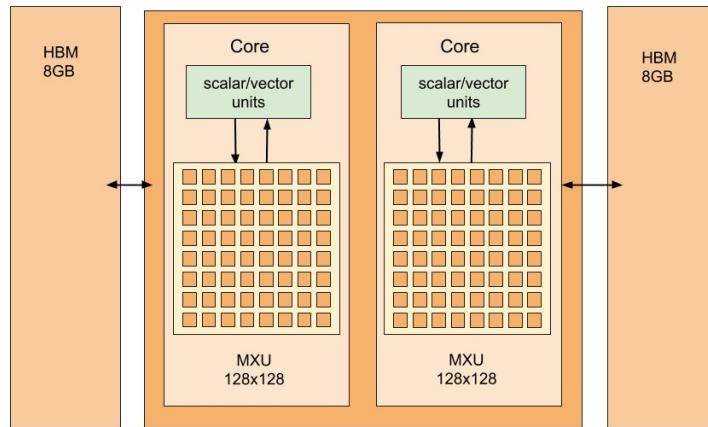
17.4. ábra. A TPU felépítése.

A második generációs TPU-k azonban már enyhén módosított architektúrával rendelkeznek. Megfigyelhető, hogy nagy mennyiségű lokális HBM (High Bandwidth Memory) memóriával rendelkeznek, valamint, hogy a korábbról ismert aktivációs, normalizációs és pooling egységek általános skalár/vektor feldolgozó egységekre lettek cserélve. Ez azért történt, mert a második generációs TPU-k már tanításra is alkalmazhatók.

17.4. Programozható hardverek

A GPU-k TPU-k Deep Learning feladatokra rendkívül jól használható egységek, azonban egyéb speciális látási algoritmusok nehezebben implementálhatók velük. Különösképp azok az algoritmusok nem gyorsíthatók ezeken az eszközökön, amelyek nagymértékben hagyatkoznak feltételes végrehajtásokra és elágazásokra. Ezekben az esetekben azonban létrehozhatjuk a saját feladatspecifikus hardverünket valamilyen programozható hardver segítségével.

Programozható hardvereknek számos fajtája létezik, kezdve a kombinációs hálózatok megvalósítására képes PLA (Programmable Logic Array) és PROM (Programmable Read-Only Memory) áramkörökkel. Léteznek sorrendi hálózatok megvalósítására képes eszközök is, ilyen például a



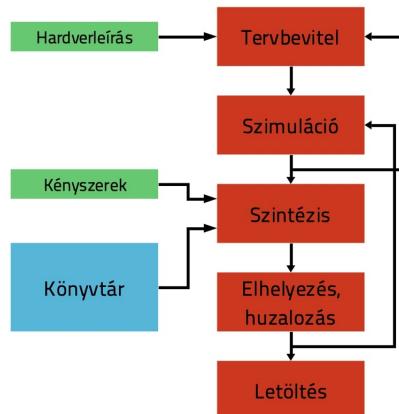
17.5. ábra. A második generációs TPU felépítése.

CPLD (Complex Programmable Logic Device). Bonyolultabb hardverek, algoritmusok megvalósítására azonban leggyakrabban FPGA (Field Prgrammable Gate Array) eszközöket szokás használni.

Az FPGA-k olyan speciális hardvereszközök, melyek segítségével elsősorban a csővezeték-technológia, kisebb részben a párhuzamos feldolgozás segítségével tudunk az algoritmusokon gyorsítani. Érdekes megjegyezni, hogy FPGA-kat gyakorta használunk hardvertervezés során, ugyanis az új IC-k készítése rendkívül költséges folyamat. Az elkészült és működő hardverterv esetében azonban lehetséges (és tömeges gyártás esetén célszerű is) a megtervezett hardvert ASIC (Application Specific IC) formájában is létrehozni.

Az FPGA tervezés során a hardvert kezdetben gyakran blokkvázlat szintjén hozzuk létre, majd ezt egy választott hardverleíró nyelv segítségével tudjuk lekódolni. Az FPGA áramkörökhez tartozik egy szintézis program is amely a leírt hardvert átkonvertálja az adott céleszköz egy - a hardvernek megfelelő - konfigurációjába. A tervezés során általában lehetőségünk van a leírt hardver szoftveres szimulációjára, ahol a tervezési hibák jó része már mutatkozik. Amennyiben a szimuláció sikeres, a következő lépés a hardver szintézis, melynek során a szintézis az általunk leírt hardvert egy könyvtár segítségével megvalósítható hardverre szintetizál.

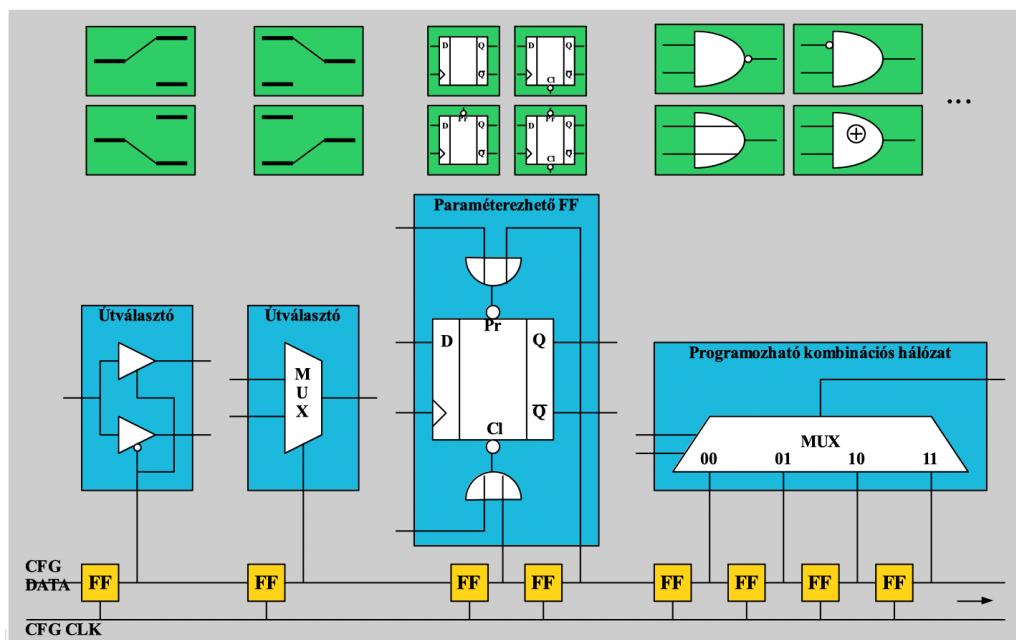
Ezt követően jön az elhelyezés és a huzalozás lépése, amely már a konkrét céleszköz paramétereit is igényli. Ennél a lépésnél kerülnek az egyes egységek az FPGA egyes blokkjaiba elhelyezésre, és itt határozható meg, hogy mi a hardver futásának maximális sebessége (ez az egyes kapcsolódó egységek közötti fizikai távolságtól függ). Az elhelyezés végén előáll a végleges konfiguráció, amely a hardverre már letölthető.



17.6. ábra. A FPGA tervezés folyamata.

17.4.1. Architektúra

Az FPGA architektúra tárgyalásának elején célszerű megérteni, hogy miként lehetséges egy hardvert programozhatóvá tenni. Az FPGA minden egyes programozható eleméhez tartozik legalább egy konfigurációs flip-flop, amelyek egy bites értéket képesek tárolni. Ezek sorasan vannak egy adatvezetékre felfűzve, az órajelet pedig egy külön a konfiguráláshoz dedikált generátortól kapják. A konfiguráció felöltésekor a flip-flopok órajelet kapnak, a konfigurációs értékeket pedig sorasan (a flip-flopok sorrendjével megegyező sorrendben) küldjük át az adatvezetéken. Ilyenkor a flip-flopok egyetlen hatalmas shift regiszterként viselkedve minden elemhez eljuttatják a saját konfigurációját. A konfiguráció befejeztével az órajelet kikapcsoljuk, így minden flip-flop konstans memóriaként tartja ezt a konfigurációs értékét, amelyet különböző hardverelemek bemenetként használnak fel. Az alábbiakban látható néhány példa ilyen elemekre:



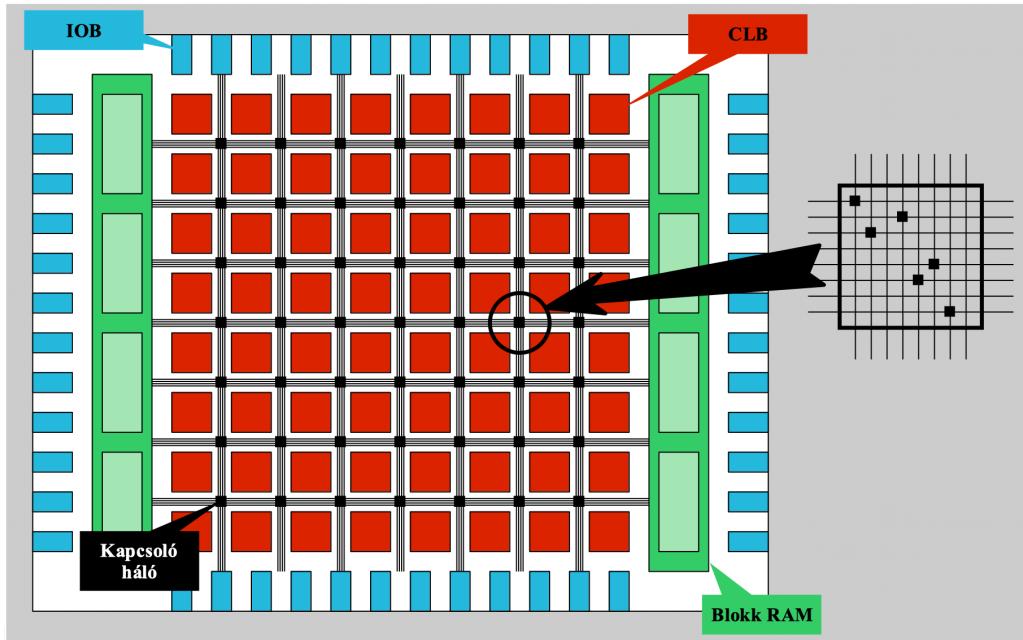
17.7. ábra. A programozható hardverek megoldása.

Egy teljes FPGA chip számos ilyen konfigurálható elemet tartalmaz, amelyek úgynevezett konfigurálható logikai blokkokba (CLB - Configurable Logic Block) szerveződnek. A CLB-k között egy programozható sínrendszer van, melynek segítségével az egyes CLB-k tetszőlegesen köthetők össze. Az FPGA-hoz tartoznak még úgynevezett blokk RAM egységek, amelyek on-chip memóriaként szolgálnak, valamint különböző IO blokkok, melyek segítségével az FPGA chip és a külvilág közti kommunikációs interfész valósítható meg.

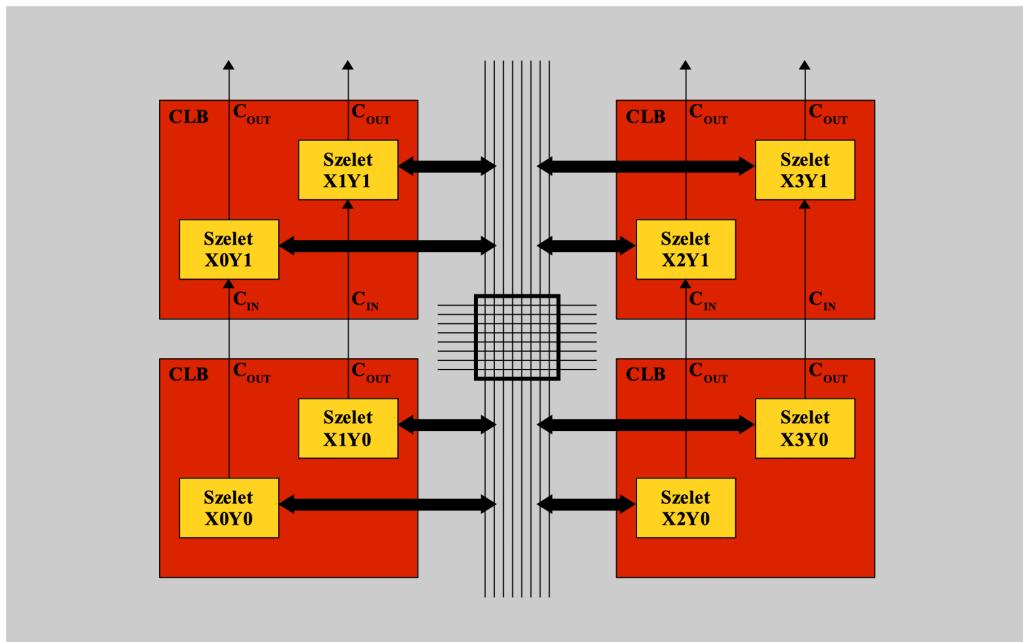
Egy CLB alapvetően két szeletből (Slice) áll, amely a konfigurálható hardverek alapegysége. Ezek a szeletek a szomszédos CLB-k szeleteivel közvetlen (a globális sínrendszerrel független) kapcsolatban is állnak, amely ennek a sínrendszernek a terhelését csökkenti nagy mértékben. A CLB-k közti összeköttetések ugyanis az FPGA tervezés egyik szűk keresztmetszete, így a gyakori, és egyszerűen megoldható kapcsolatokat érdemes külön, dedikáltan megvalósítani.

17.4.2. Szeletek

Minden szelet tartalmaz több look-up table áramkört, amelyek 1 bites értékeket tárolnak. Ezen felül minden szelet tartalmaz több, tetszőleges konfigurálható flip-flopot is. Ezen flip-flopok kezdeti értéke, szinkron/aszinkron viselkedése, órajelek engedélyezése, stb. tetszőlegesen megválasztható. minden szelethez tartozik továbbá néhány aritmetikai műveleteket kiegészítő eszköz: ilyen például az átvitelképző egység, vagy a számláló. Található még ezen felül számos multiplexer is a



17.8. ábra. Az FPGA általános felépítése.



17.9. ábra. A CLB-k általános felépítése.

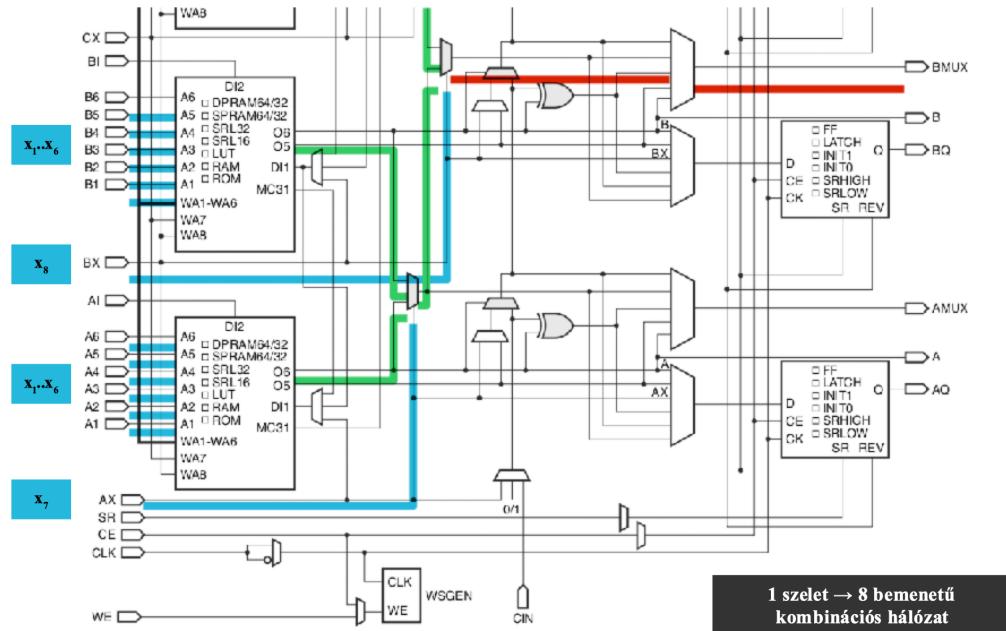
szeletekben, mellyel útvonalválasztás, kombinációs függvények kiegészítése, és egyéb speciális feladatok is végezhetők.

Fontos külön említeni a szeletekben található LUT egységek működéséről. Ezek az egységek alkotják ugyanis egy programozható szelet magját. LUT segítségével megvalósíthatunk egyszerű logikai függvényeket, melyek bemenetszáma 5-8 között változhat. Erre a bővítésre a slice multiplexerei használhatók. Természetesen egyetlen szeletben véges mennyiségű LUT található, így a több bemenetű logikai hálózatból kevesebb fér bele egy szeletbe. A kívánt működés eléréséhez csupán annyit kell tennünk, hogy a hálózat igazságábláját beletöljük a LUT memóriaelemeibe.

Érdemes még megemlíteni, hogy ezen felül a LUT használható RAM, vagy (virtuálisan) ROM elemként is. Ezek 1 bites memóriacellaként fognak viselkedni, azonban az elérésük lényegesen

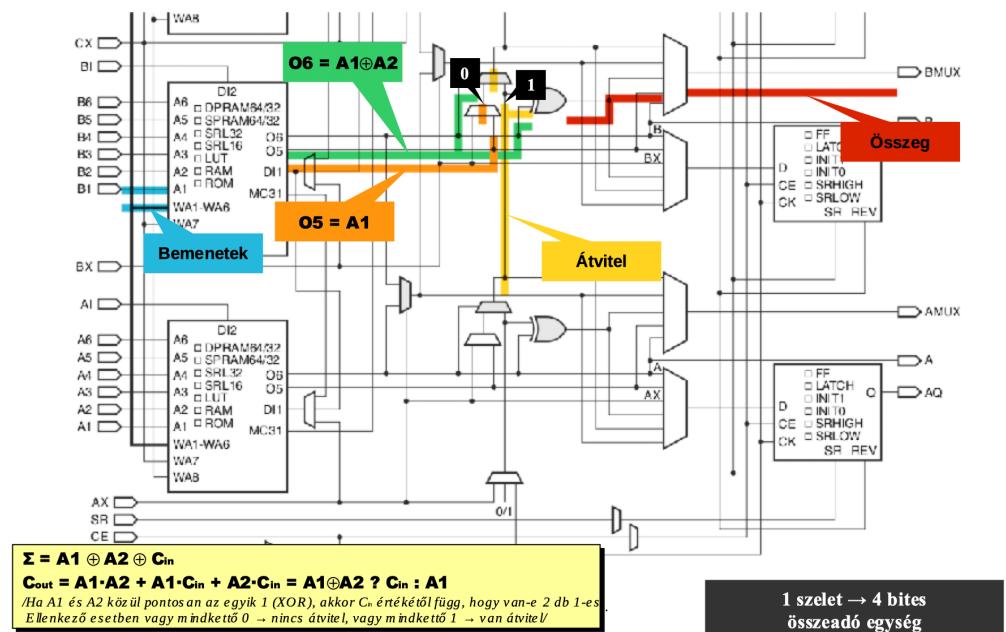
gyorsabb a blokk RAM-hoz képest. Felhasználhatók a LUT egységek még shift regiszterként is, amely hardverek esetén gyakran alkalmazott egység.

Vizsgálunk meg röviden néhány példát a szeletek használatára. Ezek közül a legegyszerűbb a kombinációs hálózat megvalósítása: itt a 8 bemenet közül hatot a LUT-ok bemenetére kapcsolunk, a másik kettő pedig a LUT-ok közül választó multiplexereket vezérli:



17.10. ábra. Egy 8 bemenetű kombinációs hálózat megvalósítása (megjegyzés: van még 2 db LUT a képen kívül).

Ennél valamivel bonyolultabb az összeadó megvalósítása. Itt a LUT-ot a XOR művelet megvalósítására használjuk fel, míg a carry logika megvalósítását a LUT mögött található kiegészítő áramközök segítségével végezzük el az alábbi módon:



17.11. ábra. Az összeadás megvalósítása.

17.4.3. Memória Opciók

Érdemes még említeni az FPGA-ban általában elérhető memória típusokról. Ezek közül kettőt már röviden tárgyalunk: a szeletekben lévő elosztott memóriát és a Blokk RAM-ot. Az elosztott memória tipikusan bites szervezésű és kis méretű: szeletenként 64-256 bit között változhat típusonként. Lokális elhelyezkedése miatt rendkívül nagy sebességű, és lehetőséget ad aszinkron módon történő olvasásra is.

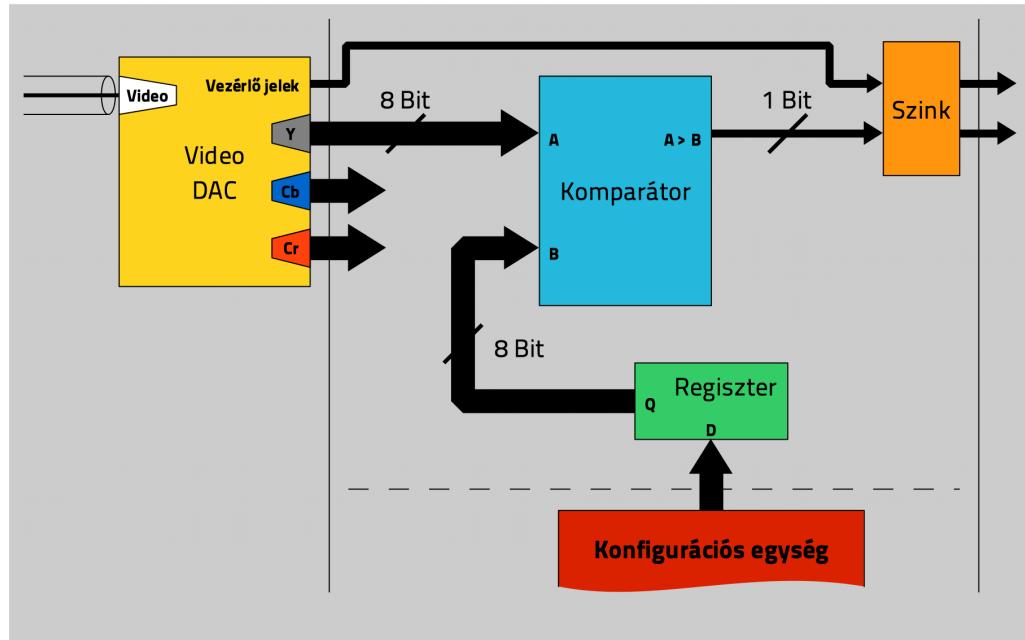
A blokk RAM az FPGA szilícium chipjén található, és közepes méretű (18-36kbit) RAM. Többféle bitszervezést is támogat (1,2,4,8,9,18,36 és 72), melyekbe a paritás bit is beleértendő. Fontos tulajdonsága, hogy az írás és az olvasás is szinkron módon történhet csak. Ez a memóriatípus használható FIFO üzemmódban is. Az FPGA kártyáknak a gyakorlatban szokott lenni egy külső memória chipjük is, ami már egy különálló DDR memória, és közösen használható az FPGA kártyát vezérző mikroprocesszorral. Ez a memória a három közül a legnagyobb, egyben a leglassabb is.

17.5. Példák

Végezetül nézzünk néhány példát képfeldolgozási műveletek hardveres megvalósítására.

17.5.1. Küszöböözés

A legegyszerűbb feladat a küszöböözés, melynek során a példában azt feltételezzük, hogy a pixelek közvetlenül a kamerából jönnek, sorosan, jelenleg YCbCr színtérben (ez kamerák esetében gyakori lehetőség).

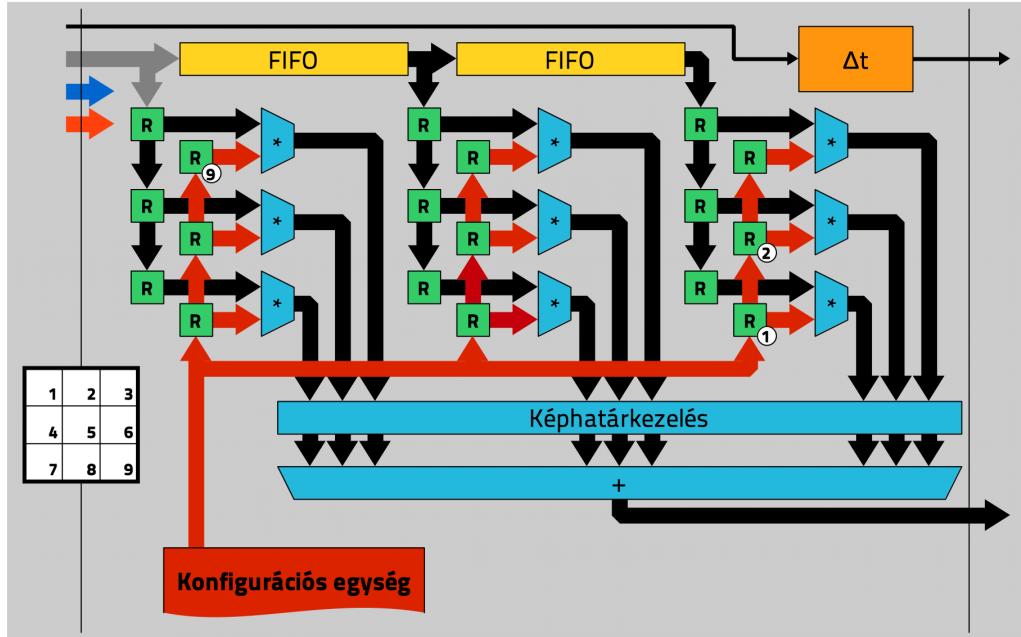


17.12. ábra. A küszöböözés blokkvázlatos megvalósítása.

17.5.2. Konvolúció

Konvolúció esetén már nehezebb a dolgunk, hiszen nem csak egyetlen pixelt kell felhasználnunk a művelet számításához, hanem az egy sorral előbb és később jövő képpontokat is.

Ebből kifolyólag a hardverünk nagy számú FIFO-t tartalmaz, amelyek a pixeleket 1-1 egész képsornival késleltetik. A szűrő súlyait a konfigurációs egységből kapják meg a szorzók, a pixelek sorrendjével fordított módon. Fontos megemlíteni, hogy a konvolúció megvalósításánál a képhatárkezelést is meg kell valósítani, vagyis a kép területéről kilogó helyekről érkezett szorzatokat el kell venni.



17.13. ábra. A konvolúció blokkvázlatos megvalósítása.

17.6. Magas szintű tervezés

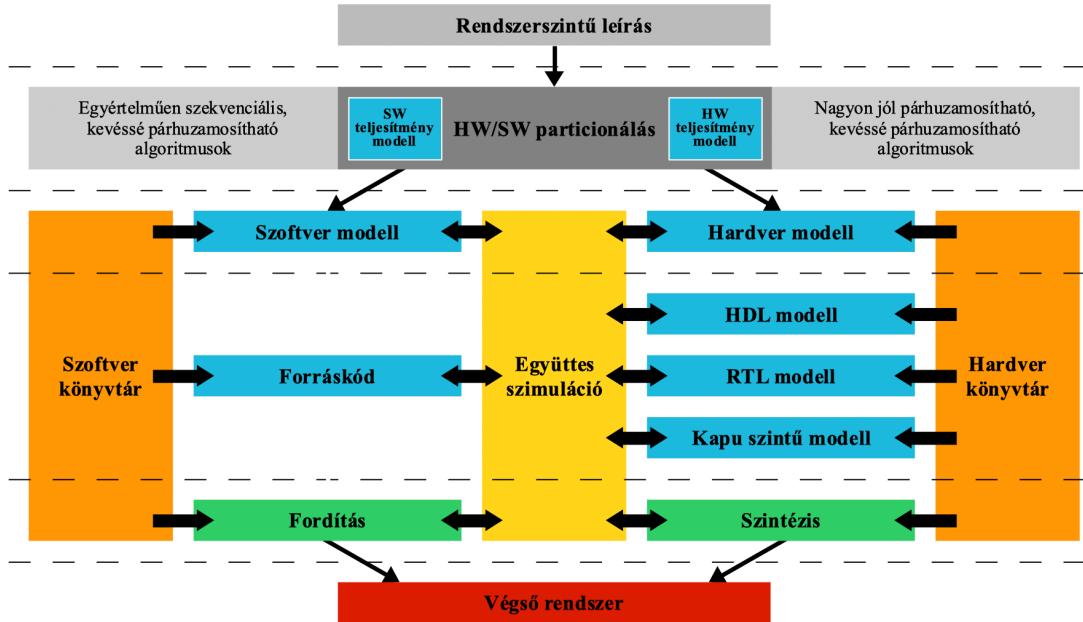
Az előadás utolsó részeként beszéljünk röviden az FPGA segítségével történő hardvertervezés néhány fontos kérdéséről. Fontos, hogy FPGA tervezés során a hardver és a szoftver tervezését együttesen, egymásra épülve érdemes elvégezni. Ehhez az elvégzendő feladatokat először elosztjuk a két eszköz között. Az egyértelműen szekvenciális algoritmusokat a CPU-n, míg a jobban párhuzamosítható és gyorsítható eljárásokat hardverből érdemes megvalósítani. A tervezés minden lépése során célszerű a hardver és szoftver rendszert közösen szimulálni.

17.6.1. Adatutak rendszere

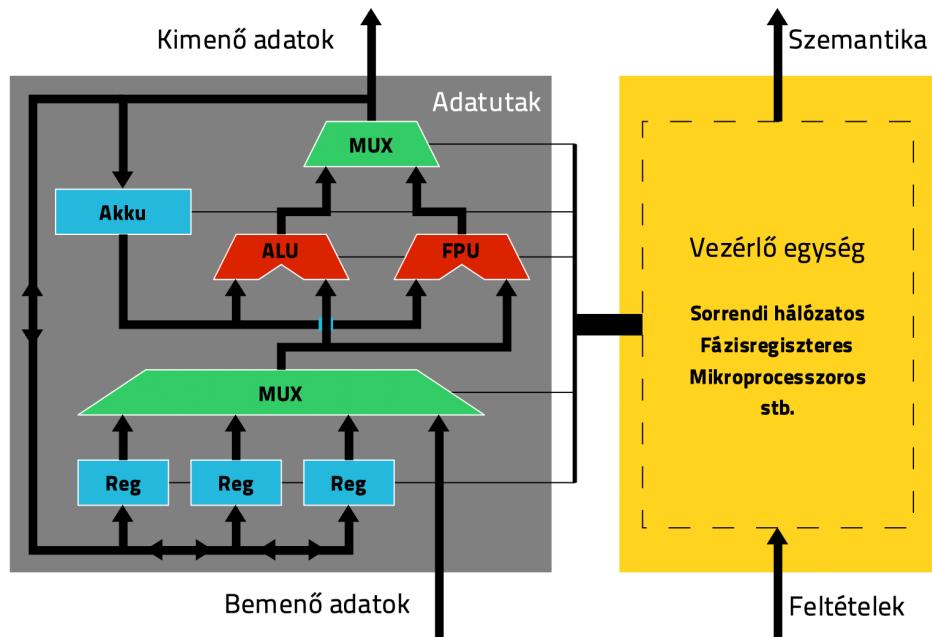
Az FPGA segítségével történő feldolgozás leírható úgynevezett adatút-rendszerként, amelyben az egyes adatelemek különböző funkcionális elemeken áthaladva kerülnek feldolgozásra. A feldolgozás menetét egy külön vezérlőegység felügyeli, amely a számára rendelkezésre álló feltételek alapján az egyes adatelemek konkrét útját változtathatja. Ez a vezérlő egység lehet egy egyszerű sorrendi hálózat, de akár egy komplett mikroprocesszoros rendszer is.

17.6.2. Csővezeték

Az adatutak rendszerének alapvető eleme az algoritmikus csővezeték. Ezen a csővezetéken a feldolgozandó adathalmaz sorosan, elemenként halad át, amely képek esetén azt jelenti, hogy az egyes műveleteket pixelenként végezzük sorosan. Ez azt eredményezi, hogy habár a teljes kép feldolgozása nem feltétlenül gyors (itt párhuzamosításról nem beszélünk), azonban a csővezetéket a pixelek soros továbbítási interfészébe beleépítve a feldolgozás csak minimális (az alábbi példa esetén például néhány sor és pixel nagyságrendű) késleltetést okoz.



17.14. ábra. A hardver és szoftver együtt tervezésének folyamata.



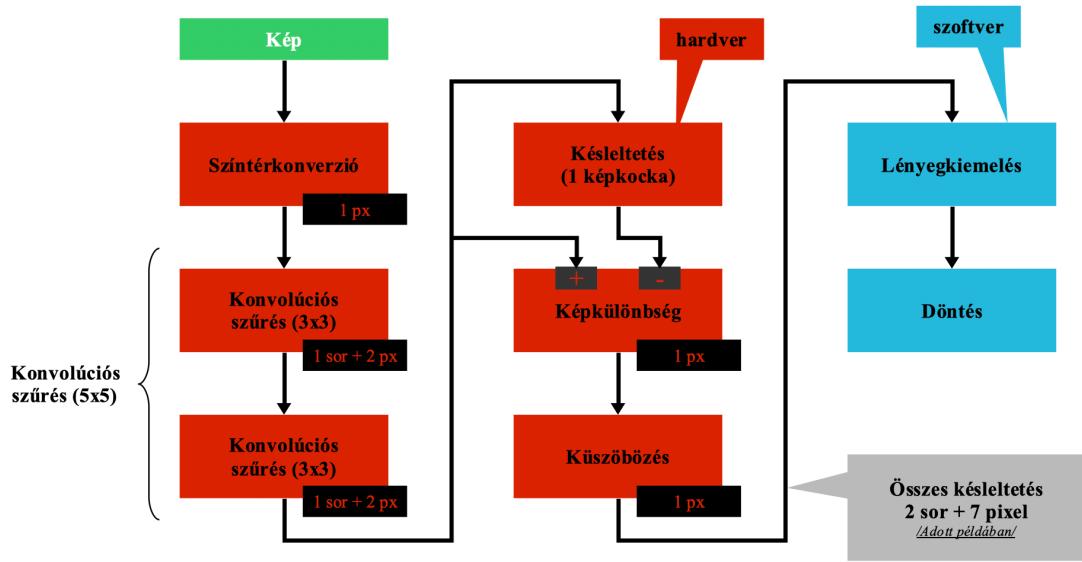
17.15. ábra. Az adatutak rendszere.

17.6.3. Újrakonfigurálás

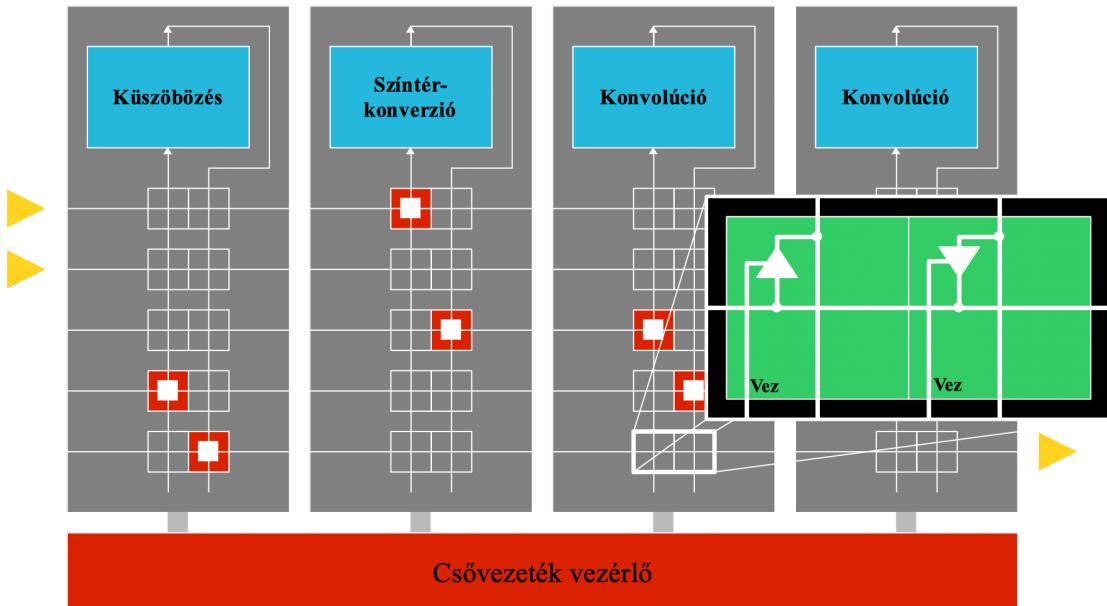
A csővezetékek fontos tulajdonsága lehet, hogy az egyes adatok útja változtatható legyen, vagyis programozható csővezetékünk adódjon. Ehhez az egyes algoritmikus blokkok közé egy konfigurálható buszrendszer kell elhelyezni.

Vezérelhető csővezetékből létezik az egyszerű megoldáson felül azonban úgynevezett szuperskalár csővezeték is. A szuperskalár feldolgozás azt jelenti, hogy a hardver egyszerre több, párhuzamos feldolgozási csővezetéket is használ és menedzsel. Ilyen csővezeték segítségével elvégezhető adat- vagy feladatszintű párhuzamosítás is az FPGA-n.

Létezik ezen felül olyan megoldás is, amelyben nem csak maga a csővezeték, hanem a teljes hardver



17.16. ábra. Egy küszöböözött képdifferenciát megvalósító hardveres csővezeték.



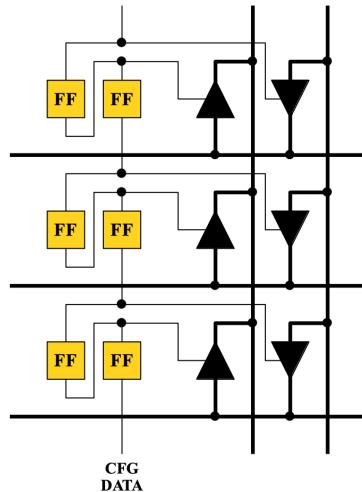
17.17. ábra. Az újrakonfigurálható csővezeték.

is újrakonfigurálható lesz. Erre első sorban lehet szükség, ha a feldolgozási feladat bizonysos események hatására változhat, vagy ha több feldolgozó egység között adaptívan szeretnénk a feladatokat kiosztani. Ez utóbbit megtehetjük a terhelés, vagy a fogyasztás optimalizálása miatt, vagy esetleg azért, hogy eltérő prioritású feladatokat más teljesítményű eszközökre is osztassunk (itt a prioritás változhat menet közben).

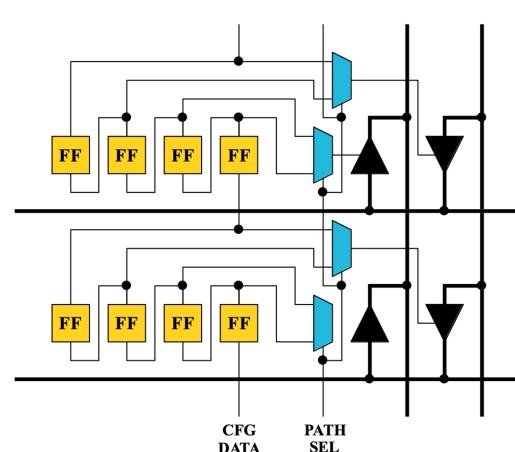
További Olvasnivaló

- [46] S. Kilts, *Advanced FPGA Design*. John Wiley & Sons, Inc., Jun. 2007. doi: 10.1002/9780470127896. [Online]. Available: <https://doi.org/10.1002%2F9780470127896>.

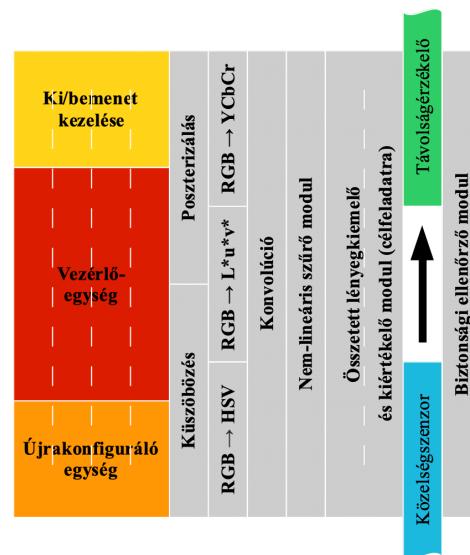
Egyeszerű csővezeték



Szuperskalár csővezeték



17.18. ábra. A hagyományos (bal) és a szuperskalár (jobb) csővezeték vezérlők.



17.19. ábra. Az újrakonfigurálható hardver.

Köszönetnyilvánítás

Köszönöm Reizinger Patriknak, hogy elkészítette a jegyzet angol nyelvű változatát.

További köszönet illeti Dr. Vajda Ferencet, aki a tárgy tematikáját kidolgozta, valamint a jegyzetben szereplő ábrák egy részét elkészítette.