MESTERSÉGES NEURONHÁLÓK ÉS ALKALMAZÁSAIK

Készítette: Dr. Tóth László

A kapcsolódó gyakorlati anyag itt található:
http://www.inf.u-szeged.hu/~groszt/index.php?id=DLmaterials
(Készítette: Dr. Grósz Tamás)

A tananyag az EFOP-3.5.1-16-2017-00004 pályázat támogatásával készült.

SZÉCHENYI 2020
MAGYARORSZÁG KORMÁNYA
Európai Unió
Európai Szociális Alap
BEFEKTETÉS A JÖVŐBE

GÉPI TANULÁSI ALAPFOGALMAK

A tananyag az EFOP-3.5.1-16-2017-00004 pályázat támogatásával készült.

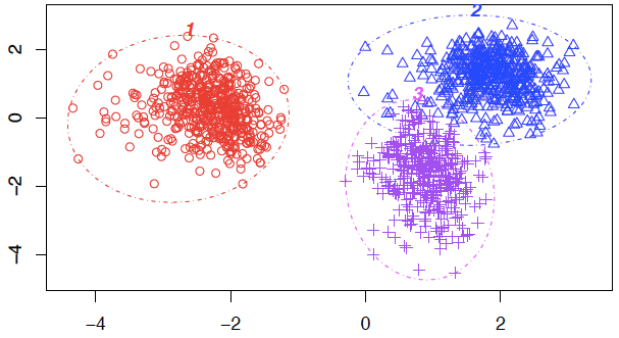
A gépi tanulás célja

- Cél: Olyan programok létrehozása, amelyek a működésük során szerzett tapasztalatok segítségével képesek javítani a saját hatékonyságukon
- Tanulóalgoritmus: Olyan algoritmusok, amelyek képesek szabályosságok, összefüggések megtalálására tanítópéldák egy halmaza alapján
  - 1. megj: A lényeg nem a konkrét tanulópéldák megtanulása, hanem a helyes általánosítás a tanulás során nem látott példákra is!
  - 2. megj: a feltételezett összefüggést „hipotézisnek” fogjuk hívni, ugyanis sosem lehetünk biztosak benne, hogy működni fognak a nem látott esetekre is
  - 3. megj: további példákat kapva a hipotézist javítjuk, az új példák alapján

Gépi tanulási feladatok típusai

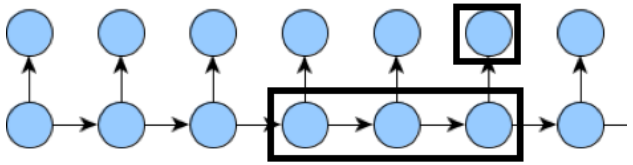
- Felügyelt tanulás: minden tanítópéldához meg van adva, hogy milyen választ várunk a géptől
  - Tipikus feladat: osztályozás
  - Példa: karakterfelismerés
  - Input: betűket ábrázoló képek

  - Output: (ezt kell a gépnek kitalálnia): milyen betű van a képen
  - Felügyelt tanítás esetén a tanítópéldákhoz (betűk képei) az elvárt választ is megadjuk (milyen betű van a képen)
- Felügyelet nélküli tanulás: az osztályokat is automatikusan kell megtalálnia a gépnek, pl. valamilyen hasonlóság alapján
  - Tipikus feladat: klaszterezés

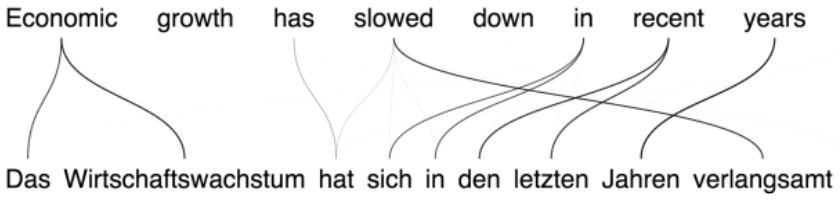


Feladattípusok (2)

- A neuronhálókat eredetileg az osztályozási feladat sémájára találták ki, azaz egy (fix méretű) inputra egy fix méretű outputot adnak
- Egyéb, fontos, speciális feladattípusok is léteznek, amelyek nem illenek ebbe a klasszikus sémába, pl:
- Időbeli folyamatok modellezése
  - Az aktuális kimenet a korábbi bemenetektől is függ(het), pl. videó felismerése, beszéd-felismerés, tőzsdei árfolyamok modellezése



- Sorozat→sorozat leképezés, pl. gépi fordítás
  - A sorozat hossza változó
  - Input-output hossza eltérhet
  - Nincsenek egyértelmű szó-szó párok, sorrend is keveredhet



Feladattípusok (3)

- Megerősítéses tanulás (reinforcement learning)
  - Pl. mesterséges „élőlények” létrehozása
  - Interakcióban van a környezetével, tapasztalatokat gyűjt, azokra reagál
  - Az osztályozással szemben itt nincs egyértelmű tanítócímke minden egyes eseményhez, csak egy hosszú távú cél (minél tovább életben maradni)
  - Kézzelfoghatóbb példa: sakk (vagy go) játszásanak megtanulása
  - Cél: a parti megnyerése, az egyes lépések nem egyértelműen minősíthetők jónak vagy rossznak
- Összegzés: a neuronhálós modellt alapvetően az osztályozási feladat megoldására találták ki, de újabban próbálják alkalmazni más jellegű, bonyolultabb feladatok megoldására is (önmagában, vagy más algoritmusokkal kombinálva)

Az osztályozási feladat

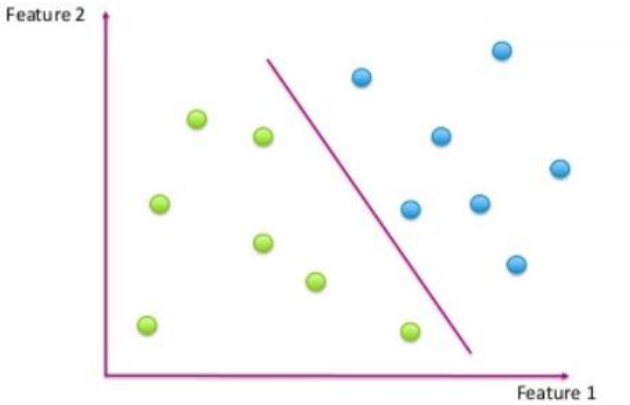
- A leggyakoribb gépi tanulási feladat
- Cél: objektum-példányok besorolása előre adott $c_1,\ldots,c_M$ osztályok valamelyikébe
- Input: valamilyen mérési adatokból álló (fix méretű) vektor
  - Jellemzővektor, feature vector, attributumvektor
- Tanítópéldák halmaza: jellemzővektorokból és hozzájuk tartozó osztálycímkékből álló párosok egy halmaza
  - Példa: Influenzás-e a beteg?

Feature vector			Osztálycímke (I/N)
Láz	Ízületi_fájdalom	Köhögés	Influenzás
38,2	Van	Nincs	Igen
36,7	Van	Száraz	Nem
41,2	Nincs	Nyákos	Igen
38,5	Van	Száraz	Igen
37,2	Nincs	Nincs	Nem

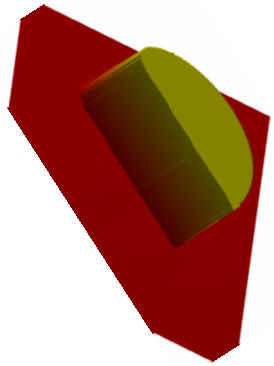
tanító-példányok

A jellemzőtér

- Ha a jellemzővektorunk N komponensből áll, akkor a tanítópéldáink egy N-dimenziós tér pontjaiként jeleníthetők meg
  - Példa:
    - két jellemző –> 2 tengely (x_1, x_2)
    - Osztálycímke (c): színekkel jelölve

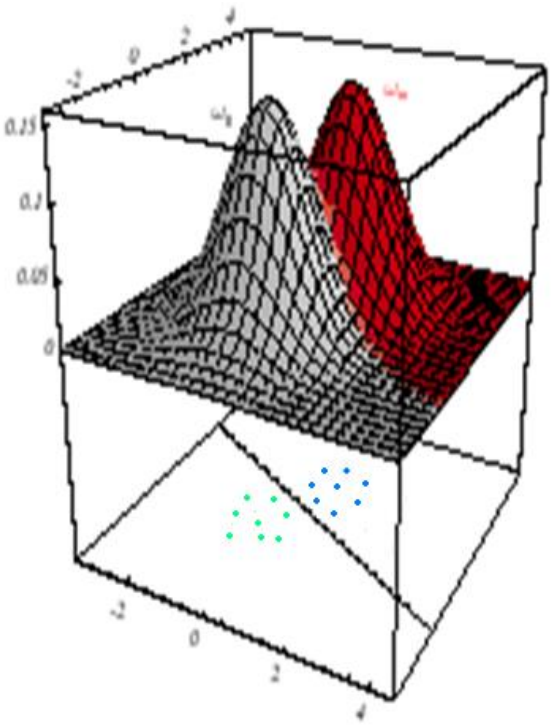


- A tanulási feladat tehát: becslést adni az (x1,x2)→c függvényre a tanítópéldák alapján
- M osztály esetén ez ekvivalens M db. (x1,x2)→{0,1} karakterisztikus függvény megtanulásával
  - A karakterisztikus függvény szakadásos, folytonos jellemzőtér felett nehezen reprezentálható
  - Kétféle módon tudjuk megoldani, hogy folytonos modellel tudjuk reprezentálni a fenti szakadásos függvényt



A döntési felület reprezentálása

- Direkt (geometriai) szemlélet: közvetlen módon a határoló felületeket reprezentáljuk
  - Valamilyen egyszerű módon, pl. egyenesekkel
- Indirekt (döntéselméleti) szemlélet:
  - Minden osztályhoz rendelünk egy függvényt, amely megmondja, hogy a tér egyes pontjai mennyire tartoznak az osztályhoz (tkp. olyan, mint a karakterisztikus fgv., de ez folytonos is lehet)
  - A diszkriminánsfüggvényekből úgy kapunk osztály-címkét, hogy a tér adott pontját az ott *maximális* értéket adó függvény osztályához soroljuk
  - Az osztályok határait indirekt módon a diszkriminánsfüggvények metszési görbéi határozzák meg



- A neuronhálók működése mindkét módon interpretálható!

A statisztikai alakfelismerés alapjai

- Az osztályozási feladat statisztikai alapú megközelítése
  - A gyakorlatban a gépi tanulás egyik legnépszerűbb irányzata
    - Szilárd matematikai háttér (valószínűségszámítás/statisztika)
    - Gyakorlatban is implementálható/használható megoldásokat ad
- Lényege a Bayes döntési szabály, amely matematikai formalizmust rak a döntéselméleti szemlélet mögé
  - Osztályozandó objektumok → **x** jellemzővektor
  - Feladat: az objektumok $\{c_1,..,c_M\}$ osztályok valamelyikébe sorolása
  - Cél: a téves besorolások számának minimalizálása hosszú távon
  - Bayes döntési szabály: adott **x** vektor esetén azt az *i* osztályt kell választani, amelyre $P(c_i|\mathbf{x})$ maximális
  - Azaz a döntéselméleti módszer optimális megoldást ad, ha diszkriminánsfüggvényként $P(c_i|\mathbf{x})$-et használjuk!
- Innentől alakfelismerés = $P(c_i|\mathbf{x})$ minél pontosabb becslése a tanítópéldák alapján!

Statisztikai alakfelismerés és neuronhálók

- A neuronháló működését értelmezhetjük az egyszerűbb „geometriai” szemlélet alapján
  - Azaz tekinthetjük úgy, hogy a háló az egyes osztályok közötti határvonalat igyekszik megkeresni
  - Ez az egyszerűbb, szemléletesebb magyarázat
- A neuronháló statisztikai alakfelismerési modellként is értelmezhető
  - Bebizonyítható, hogy bizonyos feltételek teljesülése esetén a neuronháló kimeneteit értelmezhetjük úgy is, mint az egyes osztályok $P(c_i|\mathbf{x})$ valószínűségére adott becslések
  - Ezek pedig megfelelnek a korábban döntéselméleti módszert által elvárt osztályonkénti diszkriminánsfüggvényeknek.
  - Ez a bonyolultabb értelmezés, viszont matematikailag sokkal hasznosabb (lehetővé teszi a modell matematikai értelmezését, elemzését…)
- A továbbiakban mindkét értelmezést megnézzük majd részletesebben

Kiértékelés

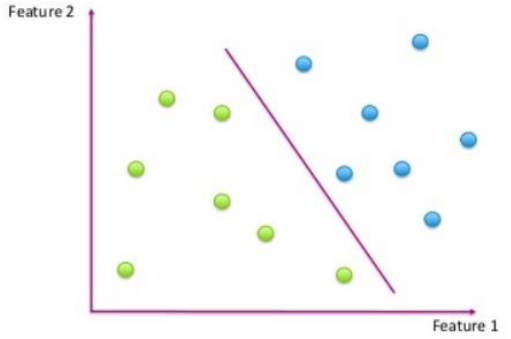
- A tanítás eredménye: a tanítópéldák alapján a tanítóalgoritmus készít egy modellt, azaz egy hipotézist a (x1,x2)→c függvényre
  - Ez a tér tetszőleges (x1, x2) pontjára meg tudja tippelni, hogy az melyik osztályba esik
- Honnan tudjuk, hogy ez a modell mennyire jó vagy rossz?
  - Megmérhetnénk, hogy a tanítópéldák hány százalékát osztályozza helyesen
- Fő célunk azonban nem a tanítópéldák címkéjének hibátlan megtanulása, hanem a tanítás során nem látott példákra való általánosítás!
  - A tanítópéldákon mért pontosság becsapós: kellően rugalmas modell jól be tudja tanulni a tanítópéldákat, azaz kicsi hibát ad rajtuk, mégis lehet, hogy rosszul általánosít a tanulás során látottaktól kicsit eltérő példákra

Kiértékelés (2)

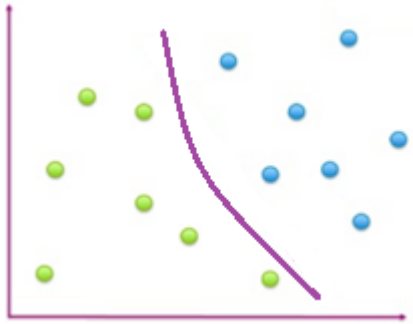
- Hogyan tudunk becslést adni az általánosító képességre?
  - A tanítóadatokat nem használhatjuk a fentiek miatt
  - Viszont mindenképp címkézett adat kell a tippelt és a valódi címkék összehasonlításához
- Megoldás: a tanítópéldáink egy részét félretesszük, nem használjuk fel a tanítás során
  - Az lesz az ún. teszthalmaz
- A tanítás+kiértékelés lépései tehát:
  - A tanító és –tesztadatok szétválasztása
  - Háló betanítása a tanító halmazon
  - A háló kiértékelése a teszthalmazon → tippelt címkék
  - A tippelt és a valódi címkék összehasonlítása a teszthalmazon
  - Osztályozási hibaszázalék = elrontott címkék száma/összes tesztpélda száma

A gépi tanulási modell paraméterei és metaparaméterei

- Minden gépi tanulási algoritmusnak vannak beállítandó paraméterei
  - Tekintsük a korábbi példát, ahol egy egyenessel igyekeztünk elválasztani két osztályt
  - A modell paraméterei ekkor az egyenes együtthatói
  - Ezeket optimalizáljuk a tanítás során, ezzel tudjuk az egyenes optimális irányba és pozícióba igazítani



- És mik azok a meta-paraméterek (más szóval hiperparaméterek)?
  - Általánosítsuk a fenti modellt úgy, hogy egyenes helyett polinomot használjon
  - A polinom fokszáma lesz az algoritmus metaparamétere
  - A paraméterekkel általában a modell adatokra való illeszkedését, a meta-paraméterekkel pedig a modell flexibilitását, reprezentációs erejét tudjuk állítani



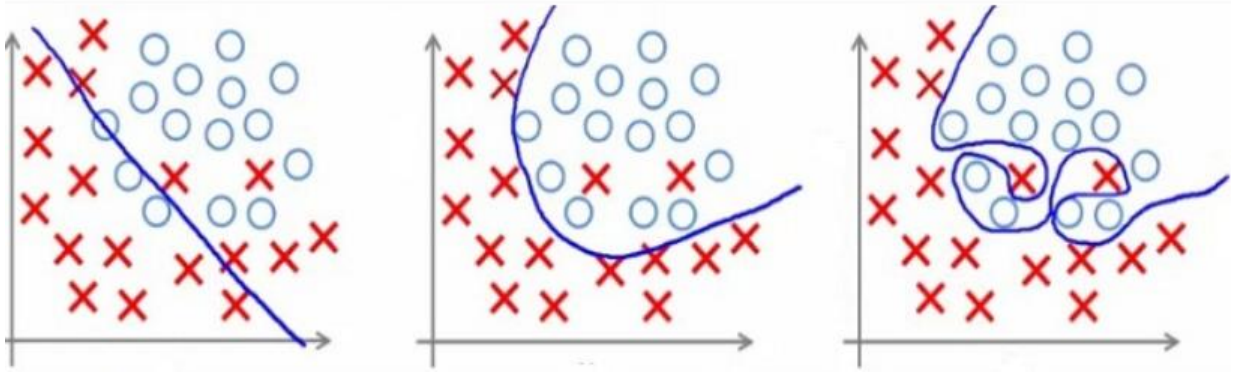
  - A paramétereket a tanítóalgoritmus hangolja be, a metaparamétereket általában mi választjuk meg kézileg, valamilyen heurisztika alapján
  - Neuronhálók esetén meta-paraméter: neuronok száma, rétegek száma,…

A gépi tanulási modell paraméterei és metaparaméterei

- Különböző paraméterértékek mellett kicsit különböző modelleket kapunk
  - Hogy tudjuk megtalálni az optimális meta-paramétereket?
- A tanítóhalmazból lecsípünk egy kisebb validációs (vagy development) példahalmazt
  - A példákat tehát train-dev-test halmazokra osztjuk
  - A tanítást a train halmazon többször elvégezzük, különböző metaparaméter-értékek mellett
  - A kapott modelleket kiértékeljük a development halmazon
  - Végül a development halmazon legjobbnak bizonyuló modellt értékeljük csak ki a teszthalmazon

A metaparaméterek szerepe

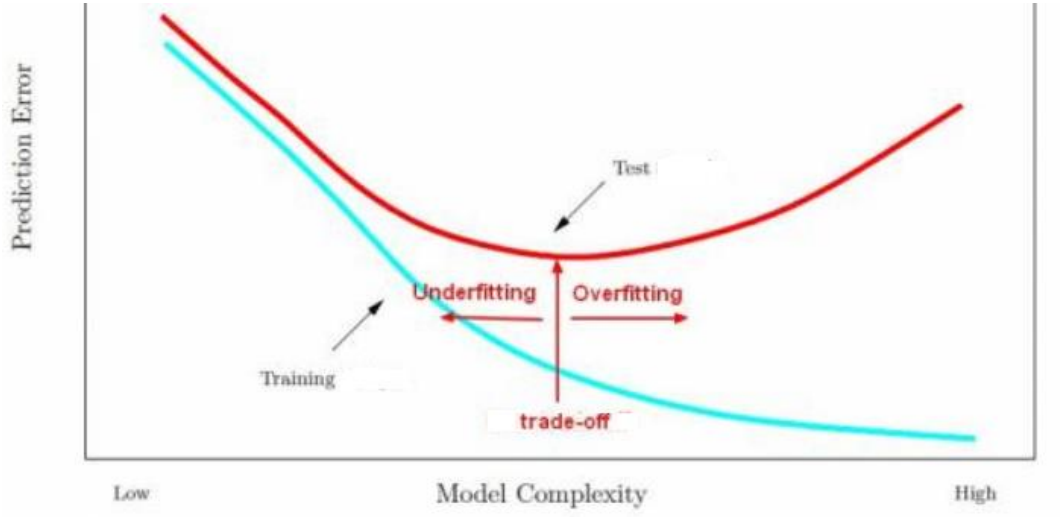
- Neuronháló esetén a paraméterek: a háló súlyai
- A fő metaparaméterek pedig: neuronok száma, rétegek száma,…
  - Ezekkel a háló méretét állítjuk
  - Nagyobb háló bonyolultabb döntési felületeket képes kialakítani



- Különböző metaparaméter-értékek mellett különböző modelleket kapunk
  - Hogy tudjuk megtalálni az optimális meta-paramétereket?
  - Használjunk nagyon rugalmas modellt?
  - Sajnos nem feltétlenül jó, mert a tanulópéldák félrevezetőek is lehetnek (véges mintát vettünk egy végtelen eloszlásból), vagy pl. hibásak (zajosak)

Általánosítás és túltanulás

- Végtelen jellemzőtér, véges számú tanító példa → sosem lehetünk biztosak benne, hogy a tanult modell jól általánosít a tanulás során nem látott esetekre
- A modell méretének növelésével a modell rugalmassága nő, egyre pontosabb lesz a tanítópéldákon
  - Viszont a tesztpéldákon egy ponton túl romlani fog a teljesítménye
  - Ezt hívják túltanulásnak (overfitting): a modell az általános tulajdonságok után már az aktuális véges mintahalmaz egyedi furcsaságait kezdi megtanulni



Optimális modellméret megtalálása

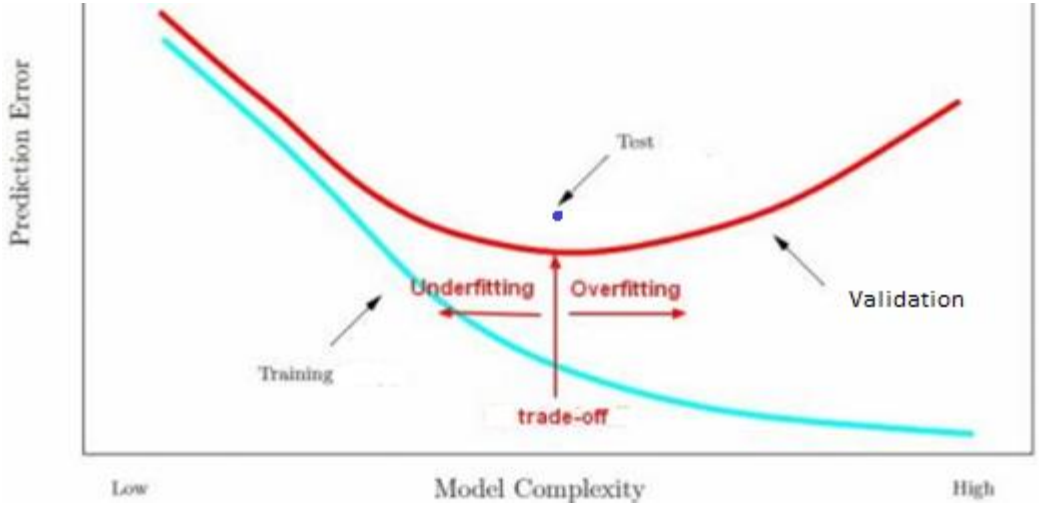
- Az optimális modellméret feladatonként nagyon eltérő lehet
  - A feladat jellege mellett függ a példaszámtól, jellemzők számától,…
  - Bonyolult elméleti háttere van: „No free lunch” tétel
- Gépi tanulási tapasztalatot igényel a belövése
  - bár egyre inkább vannak automatikus megoldások erre is…
- Néhány „hüvelykszabály”:
  - „There is no data like more data” - Minél több tanítópéldánk van, annál nagyobb hálózatot építhetünk a túltanulás veszélye nélkül
  - Érdemes kis jellemzőszámra törekedni („curse of dimensionality”)
    - Az (adatmennyiséghez képest) nagyon sok jellemző esetén szintén nő a túltanulás veszélye
  - Kevés adat és/vagy nagy jellemzőszám esetén érdemes regularizációs technikákat használni (ld. később)
  - Megjegyzés: a „deep learning”-nek csak nagyon nagy adatmennyiség mellett van igazán értelme…

A metaparaméterek belövése a gyakorlatban

- Különböző paraméterértékek mellett kicsit különböző modelleket kapunk
  - Hogy tudjuk megtalálni az optimális metaparamétereket?
  - Le kellene mérnünk a modell pontosságát különböző metaparaméter-értékek mellett egy teszthalmazon
  - A végső teszthalmaznak függetlennek kell lennie, mind a paraméterek, mind a metaparaméterek optimalizálása során használt adatoktól
- A tanítóhalmazból lecsípünk egy kisebb validációs (más szóval development) példahalmazt
  - A példákat tehát train-dev-test halmazokra osztjuk
  - A tanítást a train halmazon többször elvégezzük, különböző metaparaméter-értékek mellett
  - A kapott modelleket kiértékeljük a development halmazon
  - Végül a development halmazon legjobbnak bizonyuló modellt értékeljük csak ki a teszthalmazon

A metaparaméterek belövése a gyakorlatban

- Ábrával szemléltetve:
  - A tanítást a train halmazon többször elvégezzük, különböző metaparaméter-értékek mellett – megkapjuk a kék görbét
  - A kapott modelleket kiértékeljük a validációs halmazon – piros görbe
  - Végül a development halmazon legjobbnak bizonyuló modellt értékeljük csak ki a teszthalmazon – kék pötty



KÖSZÖNÖM A FIGYELMET!

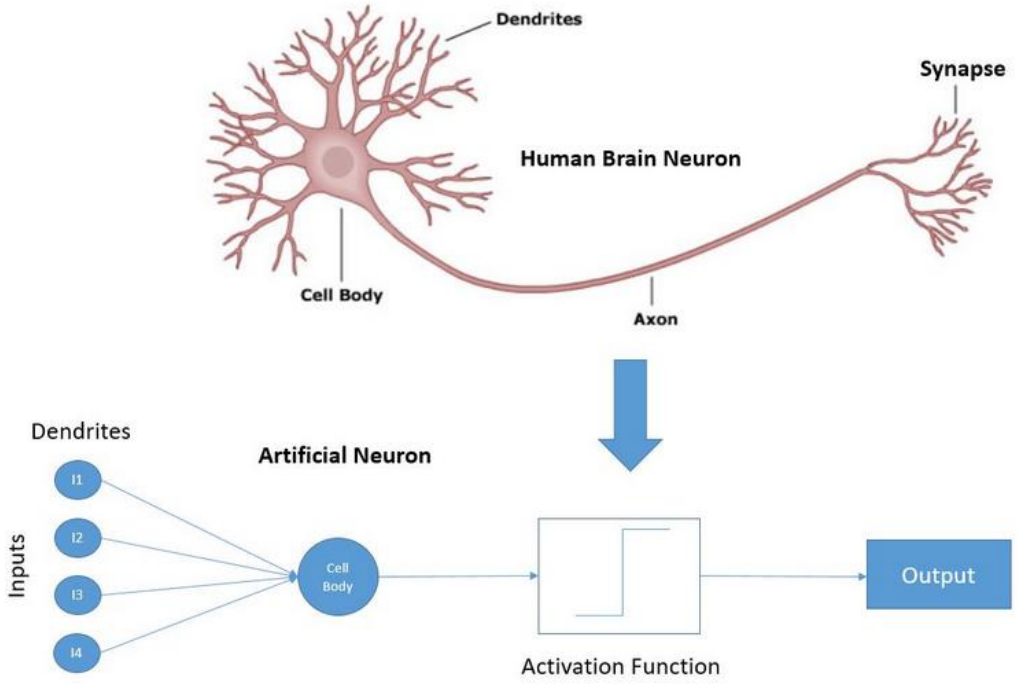
A tananyag az EFOP-3.5.1-16-2017-00004 pályázat támogatásával készült.

A PERCEPTRON NEURÁLIS MODELL ÉS TANÍTÁSA

A tananyag az EFOP-3.5.1-16-2017-00004 pályázat támogatásával készült.

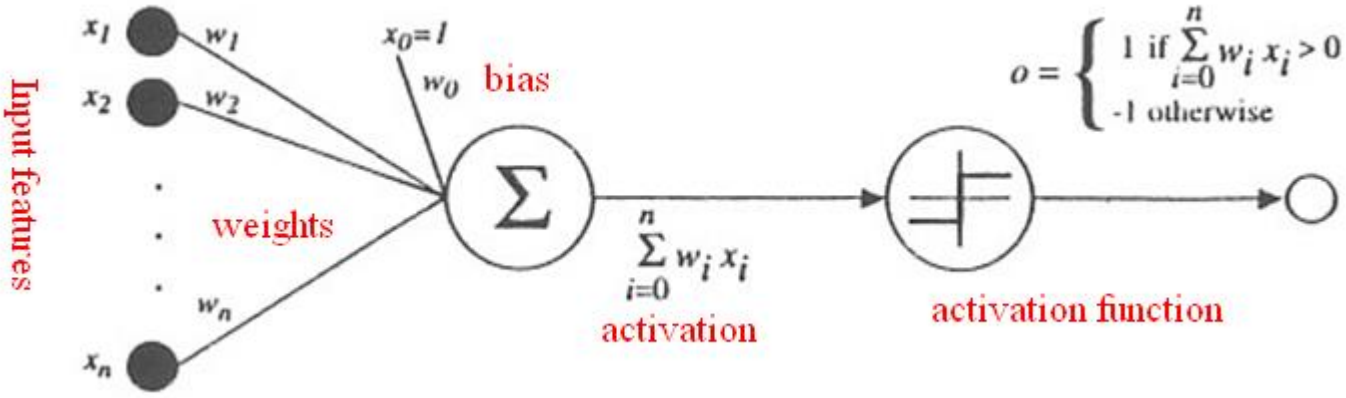
A „perceptron” neurális modell

- A legismertebb neurális modell a “perceptron” modell
- Rosenblatt találta fel 1957-ben
- A medell biológiai neuronokhoz való hasonlósága mellett érvelt
  - Sok input – “dendritek”
  - Egy output – “axon”
  - Ha az ingerek (súlyozott) összege elér egy küszöböt, a neuron tüzel, különben nem – a neuron kimenete bináris (az eredeti modellben), erre szolgál az aktivációs függvény



A perceptron matematikai modellje

- Az inputok az $x_1 \ldots x_n$ jellemzőknek felelnek meg
- Az egyes kapcsolatok erősségét a $w_1,..,w_n$ súlyok modellezik
- Ezek mellett van egy w_0 "bias" paraméter
- A neuront érő ingerek összességét aktivációnak hívjuk: $a = \sum_{1}^{n} w_i x_i + w_0$
- Egyszerűsítő jelöléssel, egy fix $x_0=1$ inputot felvéve: $\sum_{i=0}^{n} w_i x_i$
- Az *a* aktivációból az *o(a)* outputot az aktivációs függvény állítja elő
  - Ez eredeti modellben ez egy egyszerű küszöbölést végez:



- A modell tanulandó paraméterei a $w_0,..,w_n$ súlyok

A perceptron tanítása

- Rosenblatt nem csak feltalálta a perceptron modellt, de egy egyszerű tanítómódszert is megadott
  - Ez “perceptron tanulási szabály” néven ismert
- Két osztályt akarunk elválasztani
  - Az egyik osztály pontjain -1, a másikon 1 értékű kimenetet várunk
- A perceptron tanulási szabály
  - Iteratív algoritmus (végigmegy a tanítópéldákon, akár többször is)
  - Ha egy mintát helyesen osztályoz a rendszer, nem csinál semmit
  - Ha a minta osztályozása hibás, módosítja a súlyokat
  - Bizonyítható, hogy ha a két osztály lineárisan elválasztható, akkor véges lépésben jó megoldást talál
  - Azonban semmit sem garantál a lineárisan nem szétválasztható esetre
    - Emiatt inkább majd más módszereket fogunk preferálni

A perceptron tanulási szabály

- A Rosenblatt által javasolt tanítóalgoritmus:

inicializálás: véletlen súlyok választása
iteráció:
- engedjük be az egyes tanítópéldákat a perceptronba
- módosítsuk a súlyokat az alábbi szabály szerint:

$w_i = w_i + \Delta w_i \quad \Delta w_i = \eta(t-o)x_i$, ahol t a várt, o pedig a kapott output;

η : kis pozitív konstans (tanulási faktor)

az iterációt addig folytassuk, amíg szükségesnek látjuk (a már tanított mintákat akár újra felhasználva!!)

- Bizonyítjuk, hogy a módosítás után az aktiváció jó irányban változik (az adott példára):

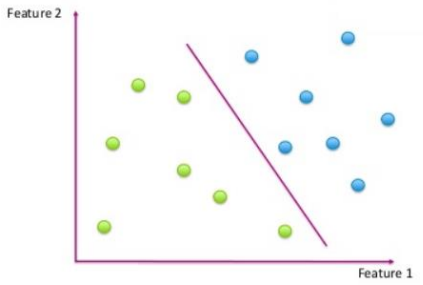
$$a_{uj} = \sum_{i=0}^{n}(w_i + \Delta w_i)x_i = \sum_{i=0}^{n} w_i x_i + \sum_{i=0}^{n} \Delta w_i x_i = a + \eta(t-o)\sum_{i=0}^{n} x_i^2$$

- Az aktiváció változásának előjelét (t-o) dönti el
  - Ha t=o, nincs változás
  - Ha t<o, az aktiváció csökken
  - Ha t>o, az aktiváció nő

A neuron kimenetének értelmezése

- Az $\sum_{i=0}^{n} x_i w_i = 0$ egy egyenes (több dimenzióban: hipersík) egyenlete
  - A neuron a súlyok által meghatározott hipersík mentén 0 aktivációs értéket ad
  - a hipersík egyik oldalán negatív, másik oldalán pozitív aktivációs értékeket ad
  - Így az aktivációs függvény az egyenes egyik oldalán -1-et, másik oldalán 1-et ad

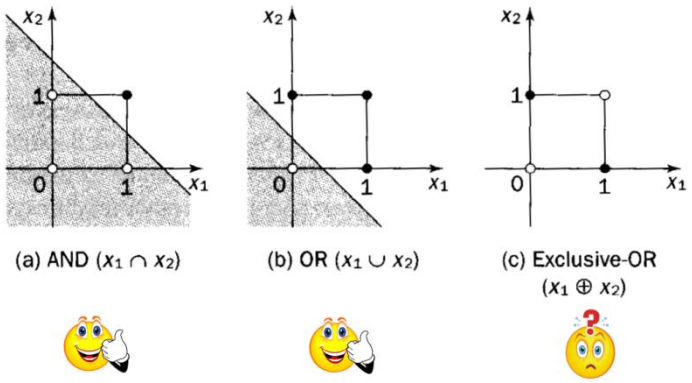
  - Azaz a perceptron a teret két résztérre képes osztani egy hipersík mentén
  - Vagyis két osztály pontjait csak akkor tudja elválasztani, ha azok egy hipersík két oldalára esnek



    - És a perceptron tanulási szabály is csak ebben az esetben működik (konvergál)

Egyetlen neuron reprezentációs ereje

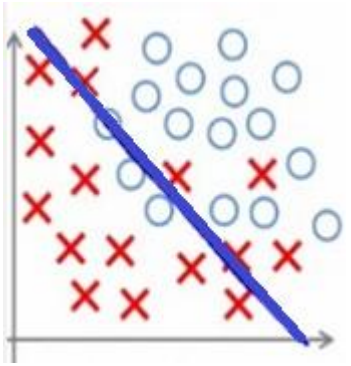
- Az egyenessel való elválasztás elég szerény reprezentációs erőt (tanulási képességet) jelent a gyakorlatban
  - Sajnos elég kevés valós feladat oldható meg ilyen egyszerűen…
  - Például az ÉS ill. VAGY művelet megtanulható, XOR nem:



  - Erre Minsky és Papert mutatott rá 1969-ben
  - Emiatt egész az 1980-as éveig hanyagolták a neuron-modell kutatását (ez volt az első „AI winter”)

Tanítás egy hibafüggvény optimalizálásával

- Tökéletes osztályozást egyetlen neuronnal sajnos csak akkor tudunk garantálni, ha a két osztály lineárisan elválasztható
  - És a perceptron tanulási szabály is csak ekkor működik
- Minket azért egy kis hibával való osztályozás és érdekelne…
  - Azaz amikor csak kevés pont esik a rossz oldalra…
  - Hogyan tudjuk a legkisebb hibájú lineáris osztályozást megtalálni?
  - Ehhez formalizálnunk kell, hogy mit értünk „kis hibán”



- Ehhez a módszercsaládhoz valahogy számszerűsítenünk kell az osztályozó hibáját egy függvény definiálásával
  - E függvényt többféleképp hívják: hibafüggvény, költségfüggvény, célfüggvény …
  - A függvény változói persze a neuron súlyai lesznek

A mean squared error (MSE) hibafüggvény

- A legegyszerűbb hibafüggvény az MSE hibafüggvény
  - Jelölje *t* a neuron kimenetén elvárt értéket („target”)
  - Jelölje *o* a neuron kimenetén kapott értéket („output”)
  - Ekkor *t* és *o* négyzetes eltérése $(t-o)^2$
- Természetesen a fenti hibaérték az össze tanítópéldára nézve érdekel bennünket, ezért vesszük a hibák átlagát a D tanító példahalmaz összes példája fölött
  - Ez lesz az átlagos négyzetes hiba (mean squared error, MSE):

$$E(\mathbf{w}) = \frac{1}{N} \sum_{d \in D} (t_d - o_d)^2$$

  - Ne feledjük, hogy a hibafüggvény változói a neuron súlyai, hiszen az o output értéke a **w** súlyvektortól és az **x** inputvektortól függ (de az utóbbi adott, tehát csak az előbbin tudunk változtatni…)

Tanítás a hibafüggvény optimalizálásával

- Az optimális súlyértékeket a hibafüggvény optimalizálásával (jelen esetben minimalizálásával) fogjuk megtalálni
  - Ezzel a gépi tanulási feladat egy sokváltozós optimalizálási feladatba megy át
  - Vegyük észre, hogy a módszer nem csak a neuronhálók esetében működhet, hanem annál sokkal általánosabb
  - A legegyszerűbb esetekben a globális optimum zárt képlettel megadható
  - Bonyolultabb esetekben iteratív algoritmusokat fogunk használni
  - A legnehezebb esetekben ezek az iteratív algoritmusok is csak lokális optimumhelyet fognak garantálni, nem globálisat
- A hibafüggvény optimalizálásán alapuló módszer akkor is működik, amikor a perceptron tanulási szabály nem
- És kiterjeszthető lesz egyetlen neuronról neuronhálókra is
  - Ezért a továbbiakban ezt fogjuk használni, nem a perceptron szabályt

A “gradient descent” algoritmus

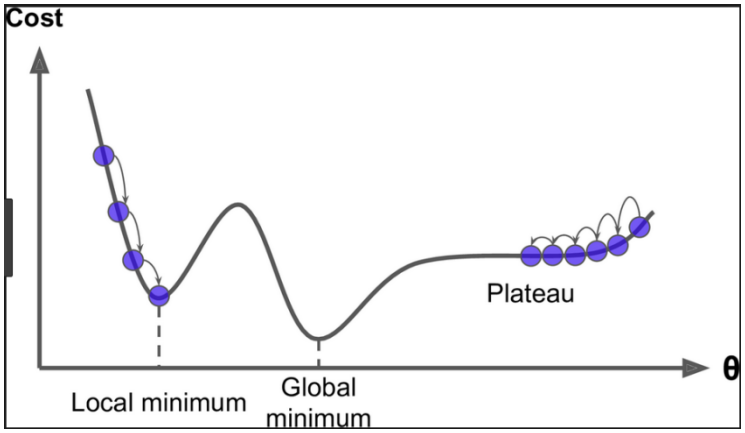
- A legegyszerűbb általános optimalizáló algoritmus sokváltozós függvényekre
- Tegyük fel hogy minimalizálni akarjuk az $E(\mathbf{w})$ sokváltozós függvényt
- A függvény deriváltja (sokváltozós esetben gradiensvektora):

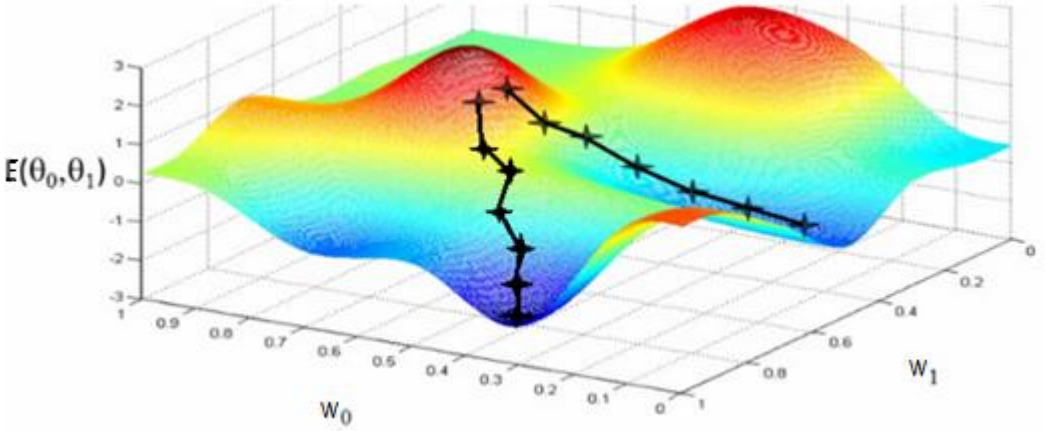
$$\nabla E(\mathbf{w}) = \left[\frac{\partial E}{\partial w_0}, \ldots, \frac{\partial E}{\partial w_n} \right]$$

  - Tudjuk, hogy a minimumpontban ez nulla
  - Tehát ki kellene számolnunk a gradienst, majd nullával egyenlővé tennünk
  - Sajnos sok gyakorlati esetben ennek megoldást túl nehéz formálisan levezetni
- A gradient descent algoritmus ennél egyszerűbb:
  - iteratívan csökkenti a célfüggvény értékét
  - mindig a legmeredekebb csökkenés irányába lép
  - amely irányt a gradiensvektor segítségével határoz meg

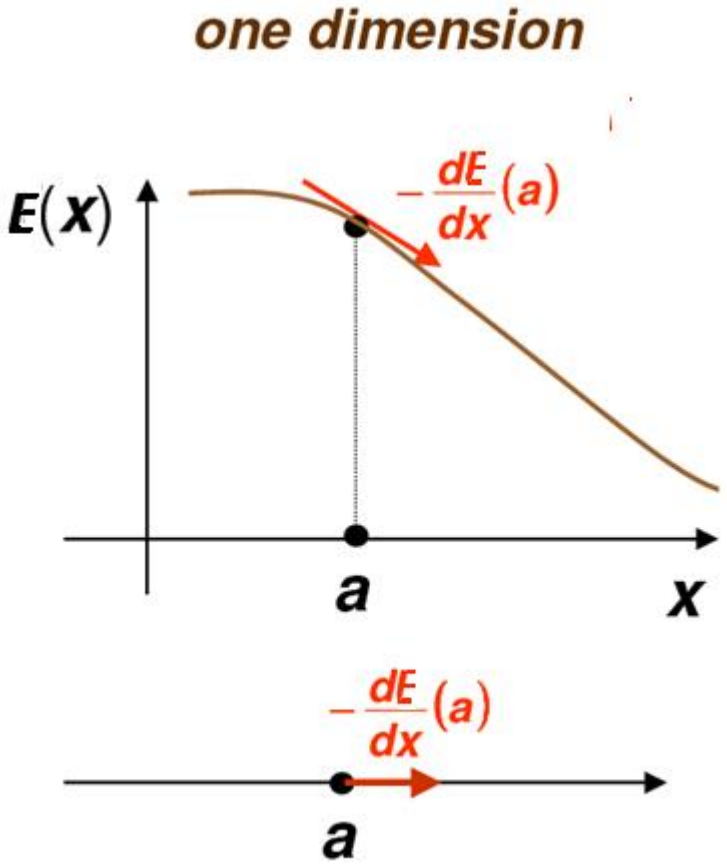
A gradient descent algoritmus (2)

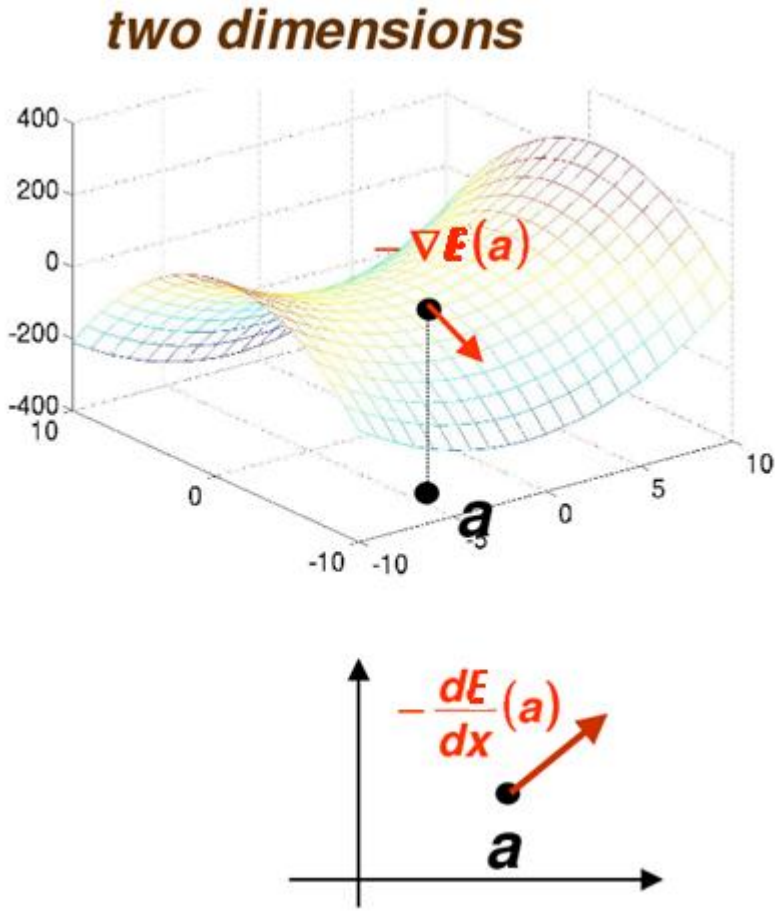
- A „hegymászó” algoritmus változata folytonos függvények esetére
  - A derivált alapján fogjuk eldönteni, hogy merre lépjünk a hibafelületen („gradiens módszer”) – gradiens-alapú csökkentés elve: a gradiensvektorral ellentétes irányba fogunk lépni
  - Véletlenszerű inicializálásból indulunk
  - A lépésközt heurisztikusan állítjuk be („learn rate”) –ld. később
  - Iteratív (addig lépkedünk, amíg el nem akadunk)
  - Csak lokális optimum megtalálását garantálja





A gradient descent szemléltetése





Gradient descent – az iteratív algoritmus

Gradient Descent for minimizing any function $J(\boldsymbol{x})$

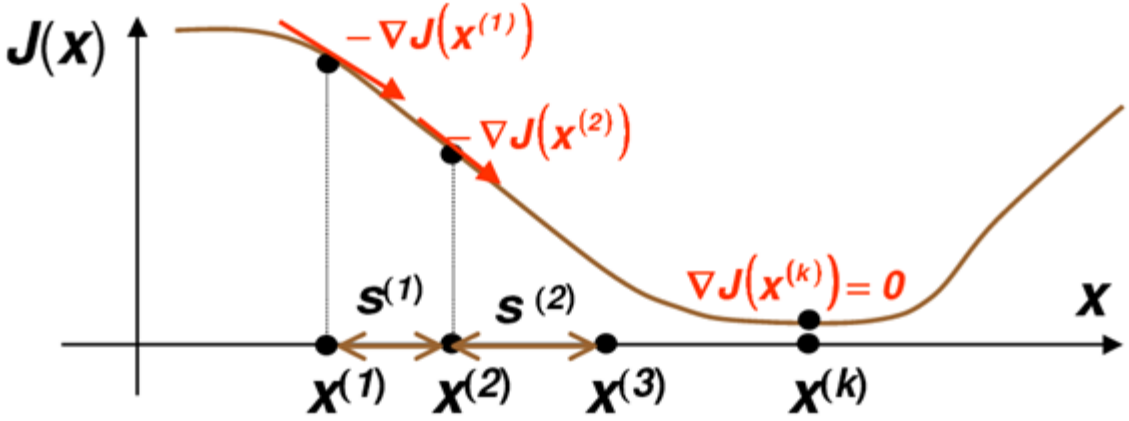
set $k = 1$ *and* $\boldsymbol{x}^{(1)}$ to some initial guess for the weight vector

while $\eta^{(k)} \left| \nabla J\left(\boldsymbol{x}^{(k)}\right) \right| > \varepsilon$

choose learning rate $\eta^{(k)}$

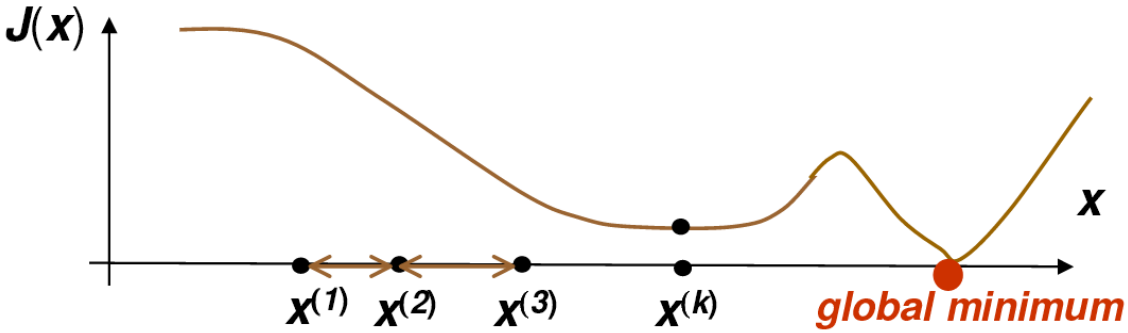
$$\boldsymbol{x}^{(k+1)} = \mathbf{x}^{(k)} - \eta^{(k)} \nabla J(\boldsymbol{x})$$ (*update rule*)

$$k = k + 1$$

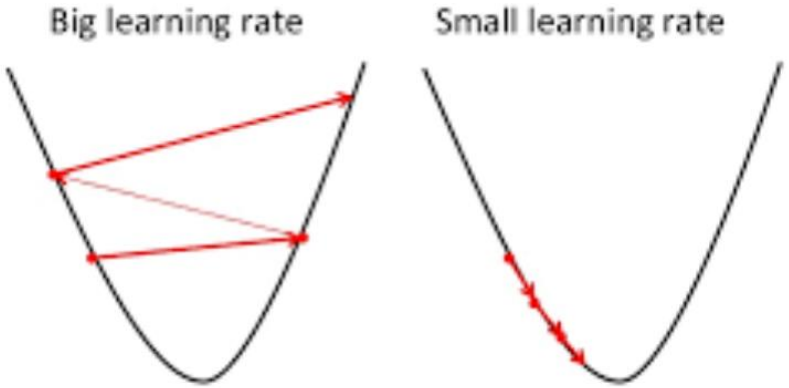


A gradient descent gyenge pontjai

- A gradient descent elakadhat lokális optimumban

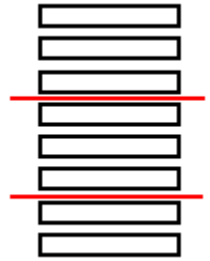


- Ha a tanulási ráta túl kicsi, az algoritmus nagyon lassan konvergál
- Ha a tanulási ráta túl nagy, nem találjuk meg az optimumot
  - A későbbiekben látunk majd heurisztikákat a tanulási ráta beállítására



A gradient descent változatai

- Az aktuális modell (paraméter-értékek) hibáját általában a hiba-függvény összes tanítópontban felvett értékének összegeként definiáljuk
  - Ezt hívjuk az algoritmus off-line avagy „batch” verziójának
- Azonban használhatjuk az algoritmus on-line vagy semi-batch verzióját is – ilyenkor az adatokat batch-ekre bontjuk
  - Ilyenkor a hibát egyetlen példán (on-line), vagy a példák egy részén (batch) számítjuk ki, majd frissítjük a paramétereket
  - Ez a változat a sztochasztikus gradient descent (SGD):
    - Mivel a hibát minden iterációban a példák más-más részhalmazán számoljuk, minden lépésben az eredeti hibafüggvény kicsit más verzióját optimalizáljuk (mivel az az *összes* példa fölött van definiálva)
    - Ez egy kis véletlenszerűséget visz az eljárásba
    - De ez a gyakorlatban nem árt, sőt inkább segíti a lokális optimumok elkerülését, és gyorsabb konvergenciát is eredményez

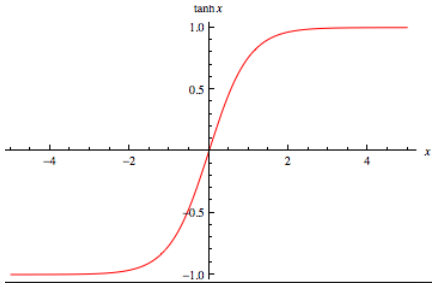


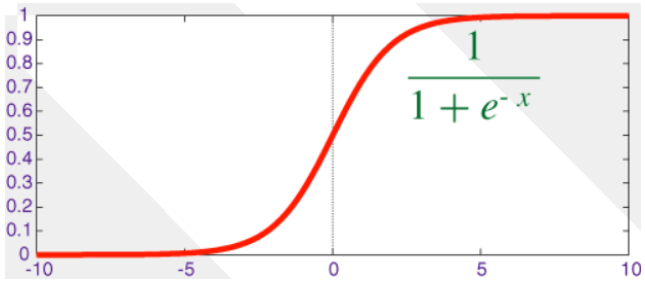
A sigmoid aktivációs függvény

- A gradient descent tanításhoz még egy problémát meg kell oldanunk:
  - Csak deriváltható függvényekkel működik, csakhogy az aktivációs függvényünk lépcsős, így nem deriválható:

$$o = \begin{cases} 1 \text{ if } \sum_{i=0}^{n} w_i x_i > 0 \\ -1 \text{ otherwise} \end{cases}$$

  - Ezért lecseréljük a *tanh* függvényre, ami közelítőleg hasonló alakú:





  - A gyakorlatban inkább a sigmoid függvény terjedt el (jobb oldalon), ami ugyanolyan alakú, de értékkészlete (-1,1) helyett (0,1)
    - Mindkettő deriválható, így már nincs akadálya a gradiens alapú tanításnak

A sigmoid aktivációs függvény (2)

- Az MSE hibafüggvény tetszőleges valós *t* célértékek és o kimeneti értékek mellett értelmezve van, nem csak a -1, 1 értékek esetére
- A lépcsős aktivációs függvény lecserélésével a neuron kimenete nem csak a -1, 1 értékek egyike lehet, hanem a (0,1) intervallumból bármely érték
- Tehát – elvileg – célértéknek is megadhatunk a (0,1) intervallumból tetszőleges számot
- Folytonos célértékek esetén osztályozás helyett regressziós feladatról beszélünk
- Mivel a sigmoid aktivációs függvény a kimeneti értéket a (0,1) intervallumra korlátozza, regressziós feladat esetében inkább lineáris aktivációs függvényt használunk ($f(x)=x$)

Neuron tanítása gradient descent algoritmussal

- Egyetlen (sigmoid aktivációs) neuron MSE hibafüggvénye:

$$E(\mathbf{w}) = \frac{1}{N}\sum_{d\in D}(t_d - o_d)^2 \quad \text{- hibafüggvény}$$

$$o_d = 1/(1+e^{-a_d}) \quad \text{- aktivációs függvény}$$

$$a_d = \sum_i w_i x_i \quad \text{- lineáris aktiváció}$$

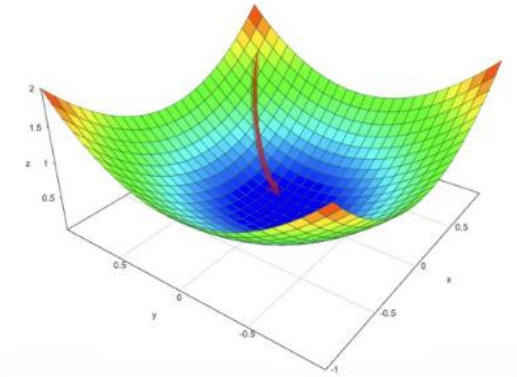
- Ezeket egymásba ágyazva kapjuk meg, hogyan függ E(w) egy konkrét w_j-től
- Deriválás: ezek deriváltjai láncszabállyal összerakva

$$\frac{\partial E}{\partial w_i} = \frac{1}{N}\sum_{d\in D} 2(t_d - o_d)o_d(1-o_d)x_{id}$$

- Súlyok frissítése minden lépésben: $w_i = w_i - \eta\frac{\partial E}{\partial w_i}$
  - Ahol η egy kis pozitív kontans („learn rate”)

Neuron tanítása gradient descent algoritmussal (2)

- Lineáris aktivációs függvény esetén az MSE hibafelület kvadratikus, egyetlen globális optimummal
  - A gradient descent algoritmus garantáltan megtalálja a globális optimumot (ha a learning rate fokozatos csökkentésére odafigyelünk, lásd később)



- Sigmoid aktivációs függvény esetén a hibafelület konvexitása nem feltétlenül áll fenn, tehát elvi garancia csak lokális optimum megtalálására van, de a gyakorlati tapasztalatok szerint az algoritmus ekkor is jó eredményeket ad

KÖSZÖNÖM A FIGYELMET!

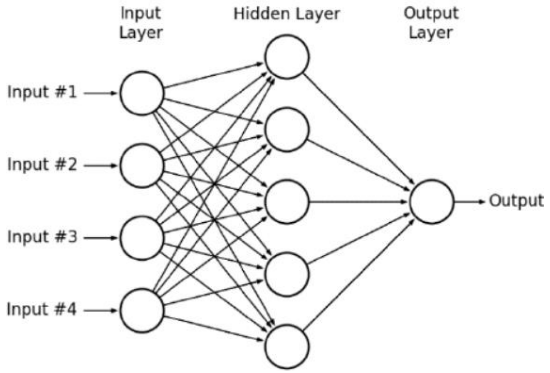
A tananyag az EFOP-3.5.1-16-2017-00004 pályázat támogatásával készült.

NEURONHÁLÓK ÉS TANÍTÁSUK A BACKPROPAGATION ALGORITMUSSAL

A tananyag az EFOP-3.5.1-16-2017-00004 pályázat támogatásával készült.

Neuron helyett neuronháló

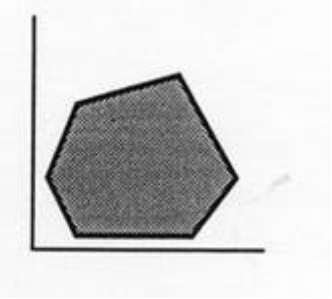
- Neuron reprezentációs erejének növelése: építsünk hálózatot!
- Klasszikus struktúra: Multilayer feedforward network
  - Minden réteg az alatta lévő rétegtől kapja az inputját
  - A rétegek között „full connection”: minden neuron minden neuronnal
  - Csak előremutató kapcsolatok



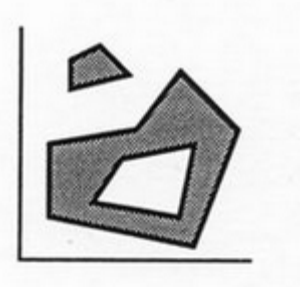
- Viszonylag ritkán szokás, de:
  - Fully connected helyett ritka („sparse”) hálót is lehet csinálni
  - Rétegek kihagyása szintén viszonylag könnyen megoldható
- Visszacsatolt (rekurrens) struktúra is lehetséges, azt viszont jóval bonyolultabb tanítani (ld. recurrent neural networks - később)
  - Ezek főleg időbeli sorozatok modellezésében hasznosak

A neuronháló reprezentációs ereje

- Konvex térrész körbekerítése: 1 rejtett réteg elég
  - A rejtett réteg egyes neuronjai reprezentálnak 1-1 határoló egyenest
  - A kimeneti neuron ÉS műveletet végez: ott jelez, ahol a rejtett neuronok mindegyike jelez



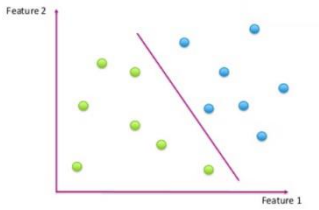
- Tetszőleges (akár non-konvex, nem összefüggő) térrész megtanulása: 2 rejtett réteg elég
  - Az első rejtett réteg neuronjai egyeneseket húznak
  - A második réteg neuronjai ÉS művelettel konvex térrészeket jelölnek ki
  - A kimeneti neuron VAGY művelettel ezeket non-konvex, nem összefüggő térrészeké is össze tudja kapcsolni



- Elvben 2 rejtett réteggel minden gyakorlati tanulási feladat megoldható
  - A tetszőleges pontossághoz végtelen sok neuron, végtelen sok tanító adat és tökéletes (globális optimumot adó) tanító algoritmus kellene…

Többosztályos tanulás

- Egyetlen neuron 2 osztályt tud csak elválasztani
  - Neuronháló esetén viszont már szóba jön a többosztályos tanulás is



- Többosztályos osztályozási feladat megoldása neuronhálóval
  - Minden c_i osztályhoz egy kimeneti neuront rendelünk
    - Azt várjuk a hálótól, hogy adott példa esetén a helyes osztályhoz rendelt kimeneten egyet adjon, a többin nullát
  - Tanító példák címkéje: 1-of-M kódolású (vagy „one-hot-encoded”) vektor
    - M méretű vektor (osztályok száma), benne egyetlen 1-es:

0 0 1 0 0 0
↑
correct class

  - Tanítási célfüggvény: maradhat az MSE error
    - Csak összegezni kell az összes kimenetre: $E(\mathbf{w}) = \frac{1}{N} \sum_{d \in D} \sum_{k \in outputs} (t_{kd} - o_{kd})^2$
  - Hogyan soroljuk be az aktuális példát?
    - Arra tanítjuk a hálót, hogy a helyes osztályra 1-et adjon, máshol 0-t
    - Valójában - ha a hiba nem 0 - a sigmoid aktivációs fgv. 0-1 közti értékeket ad vissza
    - Válasszuk a maximális értéket adó kimeneti neuron osztályát!

Többosztályos tanulás a CE hibafüggvénnyel

- Egy korábbi ábrán szemléltettük, hogy a neuronháló hogyan képes osztályokat elszeparálni (geometriai szemlélet)
- Most azt látjuk, hogy többosztályos háló esetén egy-egy kimenetünk lesz minden c_i osztályhoz
  - És ezek közül a maximális értéket adóra tippelünk
  - Ezt pontosan megfelel a döntéselméleti szemlélet döntési sémájának!
- De értelmezhetjük-e a háló kimeneteit $P(c_i|\mathbf{x})$ valószínűségi becslésként?
- Igen, így a többosztályos háló a döntéselméleti szemléletnek is meg tud felelni, de ehhez az alábbiak kellenek még:
  - Módosítani kell a kimeneti neuronok aktivációs függvényét, hogy a kimenetek összege 1 legyen
  - Módosítani fogjuk a hibafüggvényt is: az MSE hiba helyett az ún. keresztentrópia (cross entropy, CE) hibafüggvényt fogjuk használni

Többosztályos tanulás a CE hibafüggvénnyel (2)

- A kimeneti neuronok értékét $P(c_i|\mathbf{x})$ becslésként akarjuk értelmezni
- Ezek együtt egy diszkrét valószínűségi eloszlást adnak meg
  - Az egyes kimenetek értékkészlete [0,1], összegük 1 kell legyen
  - Előbbit teljesíti a sigmoid függvény, de az utóbbit nem
- A kimeneti neuronokon a sigmoid helyett a softmax aktivációs függvényt fogjuk alkalmazni

$$\sigma(\mathbf{a})_j = \frac{e^{\mathbf{a}_j}}{\sum_{k=1}^{M} e^{\mathbf{a}_k}} \quad \text{for } j = 1, \ldots, M$$

  - A kimenetek értéke így garantáltan (0,1) közé esik, és az összegük 1

Többosztályos tanulás a CE hibafüggvénnyel (3)

- A keresztentrópia hibafüggvény:

$$E(\mathbf{w}) = -\frac{1}{N}\sum_{d\in D}\sum_{k\in outputs} t_{kd}\ln(o_{kd})$$

  - A keresztentrópiát diszkrét eloszlások eltérésének mérésére találták ki
  - Vegyük észre, hogy 1-of-N kódolás esetén t értéke csak 0 vagy 1 lehet, így a célfüggvény az alábbi módon egyszerűsödik:

$$E(\mathbf{w}) = -\frac{1}{N}\sum_{d\in D}\ln(o_{correct,d})$$

  - Ez akkor lesz minimális, ha a helyes osztályhoz tartozó kimenet minél inkább 1-hez közelít
  - Mivel a softmax függvény révén a kimenetek össze vannak kötve, ez csak úgy lehetséges, ha az összes többi osztályhoz tartozó kimenet értéke 0-hoz közelít

Kapcsolat a statisztikai alakfelismeréssel

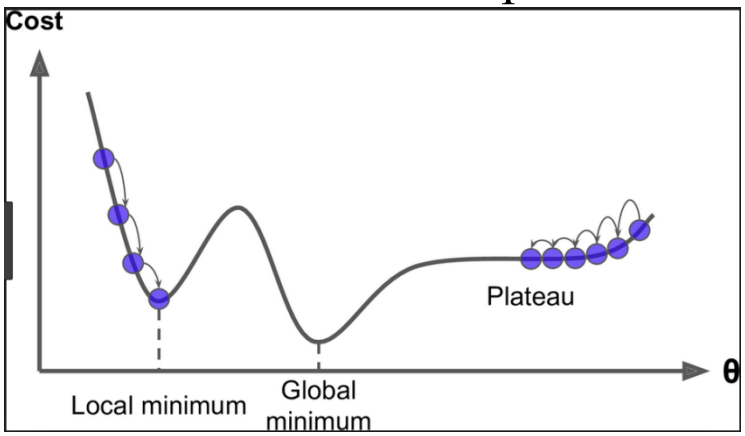
- A fenti módosításokkal a háló kimenetei értelmezhetők $P(c_i|\mathbf{x})$ –re adott becslésként, de vajon a tanulás során előálló értékeknek tényleg van közük $P(c_i|\mathbf{x})$ –hez?
- Bebizonyítható, hogy a korábban leírtaknak megfelelően felépített és tanított háló kimenetei tanítás során $P(c_i|\mathbf{x})$ valószínűségekhez tartanak!
  - A gyakorlatban nem szabad elfelejteni, hogy ez csak közelítés, a tökéletes modellezéshez végtelen nagy háló, végtelen sok tanító adat és globális optimumot garantáló tanítóalgoritmus kellene
- A statisztikai alakfelismeréssel, illetve a Bayes döntési szabállyal való összekapcsolás bizonyos alkalmazási területeken jelentősen megnövelte a neuronhálók iránti bizalmat

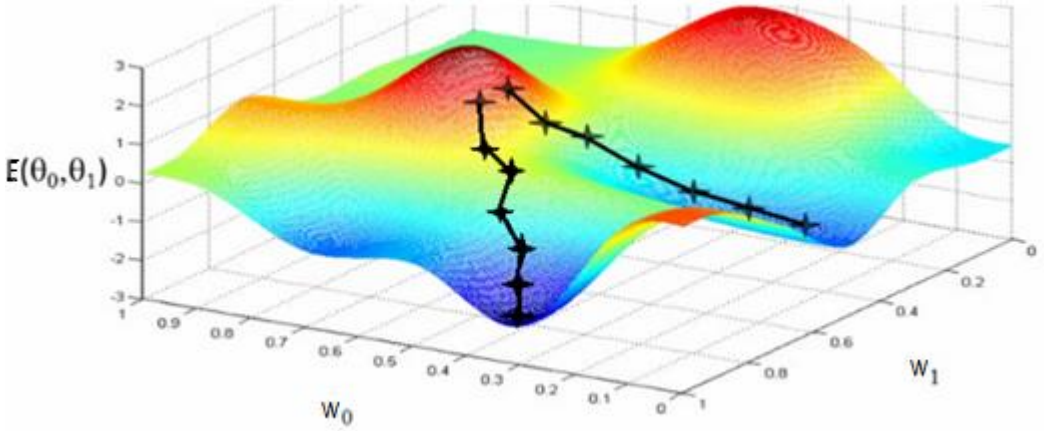
A neuronháló tanítása

- A hibafüggvények által definiáltuk, hogy hogyan mérjük a háló hibáját egy adott mintahalmazon
- A neuronháló úgy tekinthető, mint egyetlen nagy függvény, amelynek változói a súlyok
- A tanítás során a súlyokat akarjuk úgy beállítani, hogy a tanító példahalmazon a háló minél kisebb hibát adjon
- Ez egy sokváltozós optimalizálási feladat
  - Sokféle optimalizáló algoritmust lehetne használni a tanuláshoz
  - A legegyszerűbb és leggyakrabban használt megoldás a korábban már látott gradient descent algoritmus
  - Neuronhálók esetében ez hiba-visszaterjesztéses (backpropagation) algoritmus néven lett közismert

A backpropagation algoritmus

- A „hegymászó” ill. „gradent descent” algoritmus változata neuronhálók esetére
  - A derivált alapján fogjuk eldönteni, hogy merre lépjünk a hibafelületen („gradiens módszer”) – legmeredekebb csökkentés elve: a gradiensvektorral ellentétes irányba fogunk lépni
  - Véletlenszerű inicializálásból indulunk
  - A lépésközt heurisztikusan állítjuk be („learn rate”) –ld. később
  - Iteratív (addig lépkedünk, amíg el nem akadunk)
  - Csak lokális optimum megtalálását garantálja





A backpropagation algoritmus egyetlen neuron esetére (emlékeztető)

- Az MSE hibafüggvény egyetlen neuronra:

$$E(\mathbf{w}) = \frac{1}{N}\sum_{d\in D}(t_d - o_d)^2 \quad \text{- hibafüggvény}$$

$$o_d = 1/(1+e^{-a_d}) \quad \text{- aktivációs függvény}$$

$$a_d = \sum_i w_i x_i \quad \text{- lineáris aktiváció}$$

- Ezeket egymásba ágyazva kapjuk meg, hogyan függ E(w) egy konkrét w_j-től
- Deriválás: ezek deriváltjai láncszabállyal összerakva

$$\frac{\partial E}{\partial w_i} = \frac{1}{N}\sum_{d\in D} 2(t_d - o_d)o_d(1-o_d)x_{id}$$

- Súlyok frissítése minden lépésben: $w_i = w_i - \eta\frac{\partial E}{\partial w_i}$
  - Ahol η egy kis pozitív kontans („learn rate”)

A backpropagation algoritmus általánosan

- Általános eset: több kimenő neuron, többrétegű hálózat

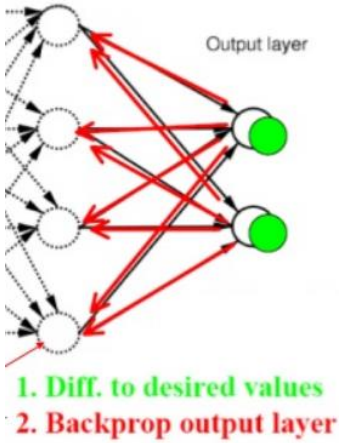
$$E(\mathbf{w}) = \frac{1}{2N} \sum_{d \in D} \sum_{k \in outputs} (t_{kd} - o_{kd})^2$$

  - Ugyanúgy a láncszabályt kell alkalmazni, csak több lépésen keresztül (a felső neuronok bemenete nem x, hanem az alatta levő réteg kimenete)
    - *(az egyszerűség kedvéért a D-re összegzést kihagyjuk a levezetésből)*
    - A deriváltat (röviden: „hibát”) először levezetjük a kimeneti rétegre
    - Majd rétegről rétegre terjesztjük vissza (backpropagation!) az alsóbb rétegekre

- a k. kimenőegység hibája: $\delta_k = o_k(1-o_k)(t_k - o_k)$
- a h. rejtett egység hibája (rétegenként visszafele):

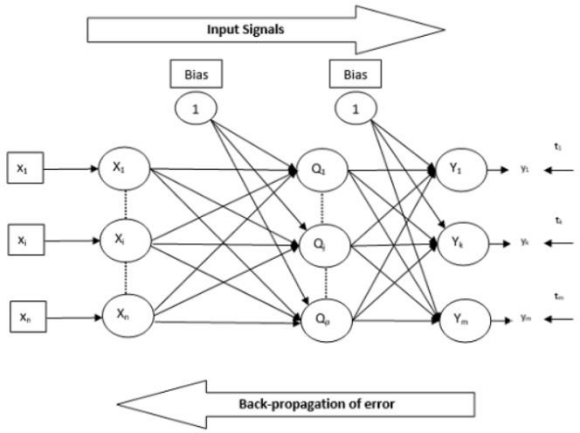
$$\delta_h = o_h(1-o_h) \sum_{k \in Uses(h)} w_{kh} \delta_k$$

    - Értelmezés: adott rejtett neuron hibája az ő kimenetét felhasználó neuronok hibájának súlyozott összegével arányos
      - Ahol a súlyok megegyeznek az adott kapcsolat súlyával



A backpropagation algoritmus - Összegzés

- Összegezve, a backpropagation tanítás fő lépései:
  - Inputtól az output felé haladva kiszámoljuk az egyes neuronok kimeneteit a D tanítópéldákon
    - *A hibafüggvényhez kell tudnunk a kimeneteket!*
  - Az outputtól az input felé haladva kiszámoljuk az egyes neuronok hibáit
  - Frissítjük a súlyokat: $w_{ji} = w_{ji} + \eta \delta_j x_{ji}$



- Megjegyzés: a hiba-visszaterjesztés megengedi a ritka (“sparse”) hálózatot, vagy a rétegek közötti ugrást tartalmazó hálózatot is. Az egyetlen lényeges megkötés a hálózati struktúrára, hogy ne legyen visszacsatolás (kör)
  - Ha nincs, akkor topológiai rendezés sorrendjében lehet frissíteni
  - Ha van, akkor visszacsatolt (rekurrens) hálózatot kapunk – ezek tanítása jóval komplikáltabb

Sztochasztikus backpropagation

- A hibafüggvényeink a hibát az összes tanítópéldára átlagolják:

$$E(\mathbf{w}) = \frac{1}{2N} \sum_{d \in D} \sum_{k \in outputs} (t_{kd} - o_{kd})^2$$

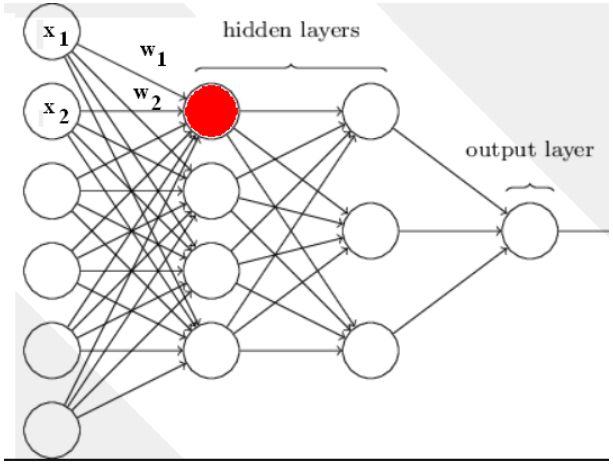
  - Ennek kiértékeléséhez az összes példát össze kell várnunk. Ez kizárja pl. az online tanulást
- On-line backpropagation: a hibafüggvényből kihagyjuk a D-re összegzést. Ehelyett minden egyes beérkező példán elvégezzük a kiértékelést, majd rögtön a hibaszámolást és súlyfrissítést is
- Semi-batch backpropagation: nem egy, de nem is az összes példa alapján frissítünk, hanem egy blokknyi („batch”) adat alapján
  - Tulajdonképpen minden frissítéskor kicsit más hibafüggvényt optimalizálunk! (az adott batch-ből számoltat)
  - Ez nem ront, hanem javít: kis véletlenszerűséget visz a rendszerbe – csökkenti a lokális optimumban való elakadás esélyét
    - Innen a név: „sztochasztikus” gradient descent (SGD)

Semi-batch backpropagation

- A gyakorlatban mindig ezt használjuk
- A batch alapú tanítás előnyei:
  - Csökken a lokális optimumban elakadás kockázata
  - Gyorsabb konvergencia
  - Gyorsabb végrehajtás (nem kell az összes adatot feldolgozni egyszerre)
  - Egyszerűbb implementáció (egy batch-nyi adat befér a memóriába)
  - Lehetővé teszi a (szemi) online tanítást
- Tipikus batch-méret: 10-100-1000 példa
- Mint a teljes adathalmazon tanításnál, itt is általában többször végigmegyünk az összes adaton
  - Egy „tanítási kör” az adatokon angolul: egy “training epoch”

Backpropagation GPU-n

- Miért hatékonyabb a neuronhálókat GPU-n tanítani?
  - Egy neuron aktivációjának kiszámítása: $a = \sum_{i=0}^{n} w_i x_i$
    =vektor-vektor szorzás
  - De az adott réteg neuronjainak aktivációját párhuzamosan is kiszámolhatjuk!
    =mátrix-vektor szorzás
  - De ugyanezt párhuzamosan elvégezhetjük egy batch-nyi input vektorra is!
    =mátrix-mátrix szorzás



  - A GPU-k a szorzatmátrix egyes celláinak értékét párhuzamosan tudják számolni – 30-40-szer gyorsabb, mint ha egyetlen CPU-n végeznénk

Tanítási tippek és trükkök

- A backpropagation algoritmus megadja a matematikai alapot a neuronhálók betanításához
  - Viszont csak lokális optimum megtalálását garantálja, így jó a tanítási eredmény elérése sajnos különféle gyakorlati trükkökön is múlik
  - Ezek többnyire inkább csak heurisztikák, nem precízen megalapozott elvek
- A legfontosabb gyakorlati fortélyokat mutatjuk be a továbbiakban
  - Adatok normalizálása, randomizálása
  - Súlyok inicializálása
  - A learning rate hangolása

Adatok randomizálása

- Semi-batch tanításnál sokat segíthet, ha az adatvektorokat véletlenszerű sorrendbe rakjuk
  - Nélkülözhetetlen, ha pl. a példák osztálycímkék szerint nincsenek jól összekeverve (pl. először az 1. osztály példái, aztán a 2. osztály, stb.)
  - Akkor is segít, ha az osztálycímkék ugyan keverednek, de valami más szempont szerint nem véletlenszerű a példák sorrendje (pl. beszédfelismerés: a felvételek beszélők szerint sorban jönnek)
  - Ugyanis mindig az aktuális batch hibafüggvényére optimalizálunk, így az újabb adatok nagyobb hangsúlyt kapnak, mint a régiek – a rendszer rátanul a későbbi adatok speciális tulajdonságára, a korábbiakat elfelejti
- Az adatokon többször végig kell menni → elvileg érdemes minden epoch előtt újra randomizálni az adatokat
  - Sajnos a randomizálás nem egyszerű: ha a memóriában csináljuk, akkor lassul az adatelérés, ha fájlban, akkor még macerásabb.

Adatok normalizálása (standardizálása)

- Tanítás előtt érdemes az egyes jellemzők értéktartományát egységes skálára hozni

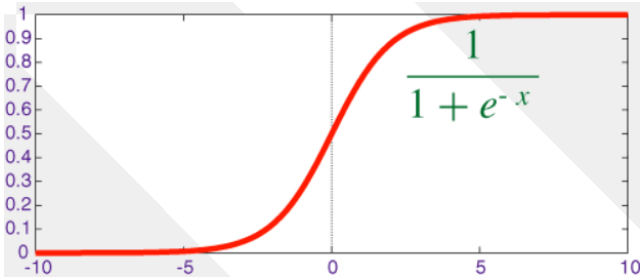
Láz	Ízületi_fájdalom	Köhögés	Influenzás
38,2	Van	Nincs	Igen
36,7	Van	Száraz	Nem
41,2	Nincs	Nyákos	Igen
38,5	Van	Száraz	Igen
37,2	Nincs	Nincs	Nem

  - min=-1, max=1: nem annyira jó ötlet, egyetlen kilógó adat („outlier”) hazavághatja
  - Szokásos megoldás: az egyes jellemzők átlaga 0, szórása 1 legyen
- Miért segít az egységes skála?
  - A súlyokat egységes tartományban inicializáljuk. Ha a jellemzők más-más skálára esnének, egyes jellemzők elnyomnának másokat az aktivációban:

$$a = \sum_{i=0}^{n} w_i x_i$$

Adatok normalizálása (2)

- Miért 0 környékére normalizálunk?
  - Idézzük fel a sigmoid aktivációs fügvényt:



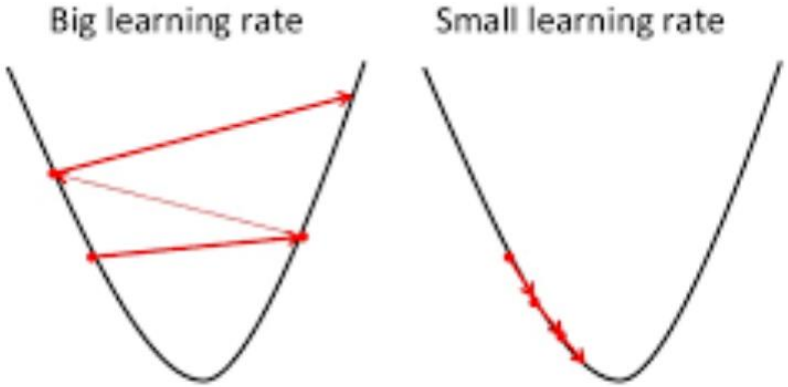
- Ha $a = \sum_{i=0}^{n} w_i x_i$ túl nagy van túl kicsi, akkor a sigmoid „lapos” részére esik → itt a derivált lényegében 0, a rendszer nem fog tanulni („elhaló gradiens” problémája)
  - Erre még visszatérünk az újabb aktivációs függvények, illetve a regularizációs módszerek bemutatásakor

Súlyok inicializálása

- A súlyokat kis random értékekkel inicializáljuk
  - Pl. [-0.1, 0.1] közötti véletlenszámokkal
- Különösen mély (sok rejtett réteget tartalmazó) hálók esetén fontos, hogy hogyan inicializálunk
  - A jó inicializálás segíti a hiba visszaterjesztését a legalsó rétegig
  - Különböző stratégiák léteznek az inicializálásra, ez a használt aktivációs függvénytől is függhet
    - Pl: a súlyok legyenek Gauss-eloszlású véletlenszámok 0 várható értékkel és 1/inputszám szórással
    - A magyarázat ugyanaz, mint az input normalizálásánál: szeretnénk az $a = \sum_{i=0}^{n} w_i x_i$ szorzatot egységesen ugyanabban a tartományban tartani

A learning rate beállítása

- A learning rate értékét általában tapasztalat alapján lőjük be
  - Bár vannak automatikus optimalizálási próbálkozások is
  - Túl nagy learn rate: nincs tanulás, ugrálás a hibafelületen
  - Túl kicsi learn rate: lassú tanulás, könnyebb elakadás lokális optimumban



  - Szokásos általános heurisztika:
    - Indítsunk olyan nagy learn rate-tel, amekkorával csak lehet (van tanulás)
    - Egy idő után (ha már nem csökken a hiba) kezdjük el csökkenteni (pl. felezgetni)

A learning rate beállítása (2)

- A túltanulás elkerülése érdekében érdemes a hiba alkaulását nem csak a tanítóadatokon, hanem egy független validációs halmazon is figyelni
- A learning rate frissítése validációs példahalmaz segítségével
  - A tanítás előtt tegyük féle az adatok egy részét → validációs halmaz
  - Valamennyi tanítási lépés (pl. 1 epoch) után értékeljük ki a hibát a validációs halmazon
  - Ha az előző kiértékeléshez képest a hiba csökkent: újabb iteráció ha nőtt, vagy csökkenése kisebb, mint egy küszöb → learning rate felezése
  - Teljes leállás: ha a learning rate lement egy küszöb alá

KÖSZÖNÖM A FIGYELMET!

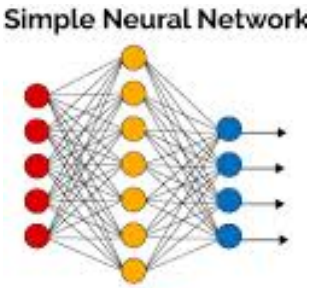
A tananyag az EFOP-3.5.1-16-2017-00004 pályázat támogatásával készült.

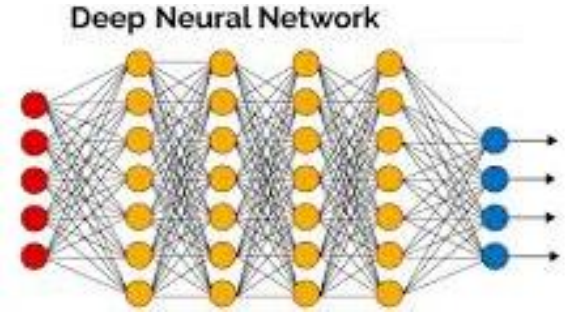
MÉLY NEURONHÁLÓK

A tananyag az EFOP-3.5.1-16-2017-00004 pályázat támogatásával készült.

Hagyományos és mély neuronhálók

- Miben különbözik a hagyományos és a mély neuronháló?
  - Strukturálisan annyi a különbség, hogy jóval több rejtett réteg van (1-2 helyett 5-10, de újabban akár 100-150 is)
- Egyszerűnek hangzik - miért csak most??
  - A mély hálók tanításához új algoritmusok kellettek
    - Legelső ilyen: DBN-előtanítás, 2006
  - A mély háló előnyei igazából csak sok tanító adat esetén mutatkoznak meg
    - Ez se volt meg a 80-as években
  - A mély háló tanítása számításigényes
    - Erre megoldás a GPU használata

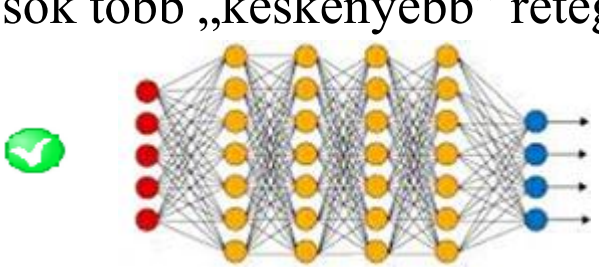


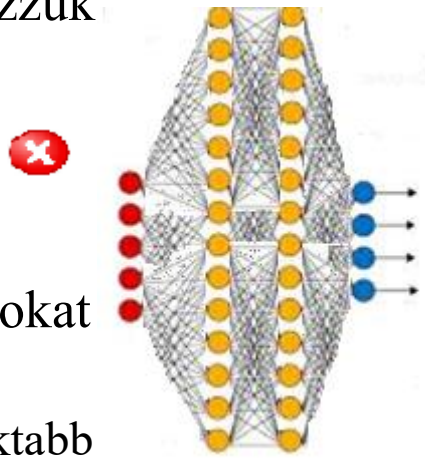


- A mély hálók jelenlegi sikeréhez az új algoritmusok, a sok tanító adat és a számítási kapacitás szerencsés együttállása kellett

Miért hatékonyabb a mély neuronháló?

- Korábban láttunk egy elvi bizonyítást arra, hogy két rejtett réteggel minden feladat megoldható
  - Azonban ez csak akkor igaz, ha végtelen nagy neuronhálónk, végtelen sok tanító adatunk és globális optimumot adó tanító algoritmusunk van
- Adott véges neuronszám mellett hatékonyabb, ha a neuronokat 1-2 „széles” réteg helyett sok több „keskenyebb” rétegbe rendezzük





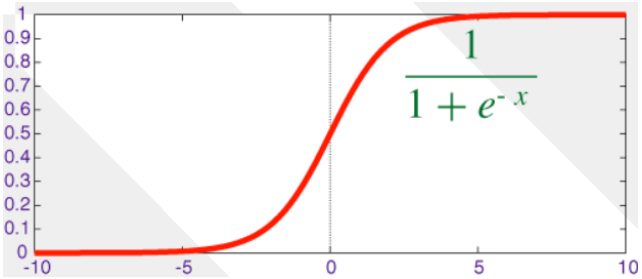
- Így a háló hierarchikusan tudja feldolgozni az adatokat
  - Képi alakfelismerési feladatokon jól látszik, hogy a magasabb rétegek egyre komplexebb, egyre absztraktabb fogalmakat tanulnak meg
    - Pont, él, → szem, orr→ arc→…

A mély háló tanítása

- A mély hálók tanítása sajnos nehezebb, mint a hagyományos hálóé
- Ennek fő oka a backpropagation algoritmus működési elve, illetve a sigmoid aktivációs függvény alakja
  - A backpropagation algoritmus a kimeneti rétegtől terjeszti vissza a hibát
  - Minél több réteget megyünk visszafele, annál nagyobb az esélye, hogy a gradiens „eltűnik”,
    - Ez az ún „vanishing gradient” effektus
  - A gyakorlatban ez azt eredményezi, hogy a mély háló egyre mélyebben levő rétegei egyre kevésbé tanulnak
  - Azaz hiába adunk újabb rétegeket a hálóhoz, az eredmények nem javulnak (sőt esetleg romlanak is)

Miért nehéz a mély háló tanítása?

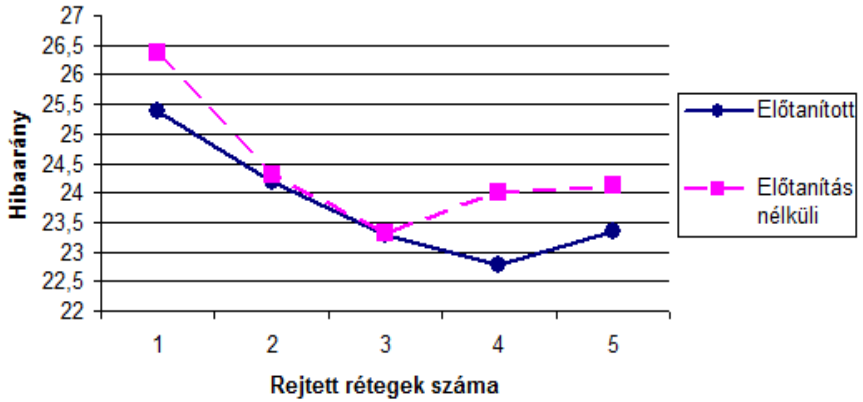
- Nézzük meg a sigmoid aktivációs függvény hatását:



  - A sigmoid függvény bemenete az aktiváció: $a = \sum_{i=0}^{n} w_i x_i$
  - Ha az aktiváció értéke nagyon nagy vagy kicsi, a bemenet a sigmoid „lapos” részeire esik
  - Itt a derivált nullához közeli, „eltűnik”, így a tanulás se fog működni
  - Ez ellen próbáltunk védekezni a súlyok inicializálása ill. az input normalizálása során is
    - A rejtett rétegek értékeinek „kordában tartására” azonban ez már kevés
    - Minél több az rejtett réteg, annál nehezebb a háló tanítása

Mély háló tanítása - szemléltetés

- Az ábra egy beszédfelismerési feladaton kapott eredményeket mutatja
  - Függőleges tengely: felismerési hiba
  - Vízszintes tengely: rejtett rétegek száma a hálóban



  - A lila görbe mutatja ez eredményeket backpropagation tanítás esetén
    - A rétegek hozzáadásával egyre kisebb a javulás, 4-5 rétegnél már romlás van
  - A kék görbe az egyik legkorábban javasolt megoldással (előtanítás) kapott eredményeket mutatja

Megoldások a mély neuronháló tanítására

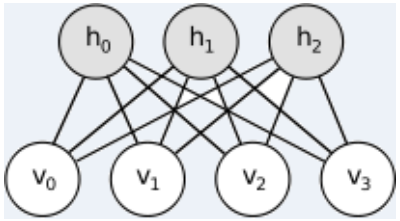
- A mély hálók hatékony tanításához módosítanunk kell a tanító algoritmust és/vagy az aktivációs függvényt
- A legismertebb módosítási lehetőségek az alábbiak
  - Előtanítás címkézetlen adatokkal (DBN-előtanítás kontrasztív divergencia (CD) hibafügvénnyel)
    - Ez volt az első ötlet, elég lassú és matematikailag bonyolult
  - A hálózat rétegről-rétegre való felépítése és tanítása
    - Jóval egyszerűbb, csak backpropagation kell hozzá
  - Újfajta aktivációs függvények használata
    - A legegyszerűbb megoldás, először ezt fogjuk megnézni
- A mély hálók tanításában további trükkök is segíthetnek (batch normalization, highway network, stb.) – ezekről később…

DBN-előtanítás

- Történetileg ez volt a legelső mélyháló-tanító algoritmus (2006)
- Egy előtanítási és egy finomhangolási lépésből áll, ahol a finomhangolás megfelel a már ismert backpropagation tanításnak
- Előtanítási lépés:
  - Konstruál egy ún. „deep belief” hálót
  - Ezt tanítja a backpropagationtól teljesen eltérő algoritmussal
- Tanítási lépés:
  - A betanított DBN hálót átkonvertálja hagyományos (sigmoidos) neuronhálóvá
  - Ezt már a backpropagation algoritmussal tanítja
  - Ezt a lépést „finomhangolásnak (fine-tuning)” nevezi

Korlátos Boltzmann-gépek

- A deep belief hálóhoz először a korlátos Boltzmann-gépet (RBM) kell megismernünk
  - Ez hasonlít a neuronháló egy rétegpárjára
  - De valós helyett bináris kimeneti értékeket ad
  - V: „visible” réteg: ez lesz egyszerre az input és az output réteg
  - H: „hidden” réteg: a rejtett réteg
  - A visible és hidden rétegek kapcsolata nem determinisztikus, hanem sztochasztikus
  - Ettől eltekintve a képlet eléggé hasonlít a standard neuronokéra (sigmoid aktivációs függvénnyel)
  - Vegyük észre, hogy standard neuronokkal szemben a V és H rétegek szerepe szimmetrikus, mindkét irányban tudnak menni az adatok

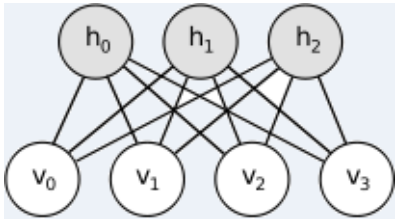


$$P(h_j = 1|v) = \sigma\left(b_j + \sum_{i=1}^{m} w_{i,j} v_i\right)$$

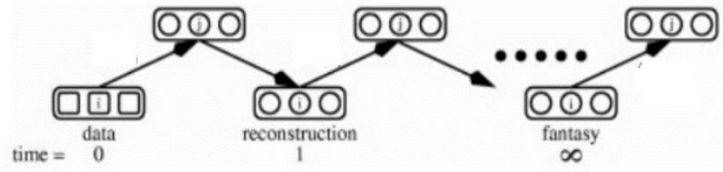
$$P(v_i = 1|h) = \sigma\left(a_i + \sum_{j=1}^{n} w_{i,j} h_j\right)$$

RBM-ek tanítása

- Az RBM-ek tanítására a kontrasztív divergencia (CD) algoritmust használhatjuk
  - Ez egy felügyelet nélküli módszer (nincsenek osztálycímkék)
  - Ugyanúgy iteratív algoritmus, mint a backpropagation
  - Cél az input rekonstruálása a rejtett réteg alapján (Hasonlít a Maximum Likelihood célfüggvényhez)



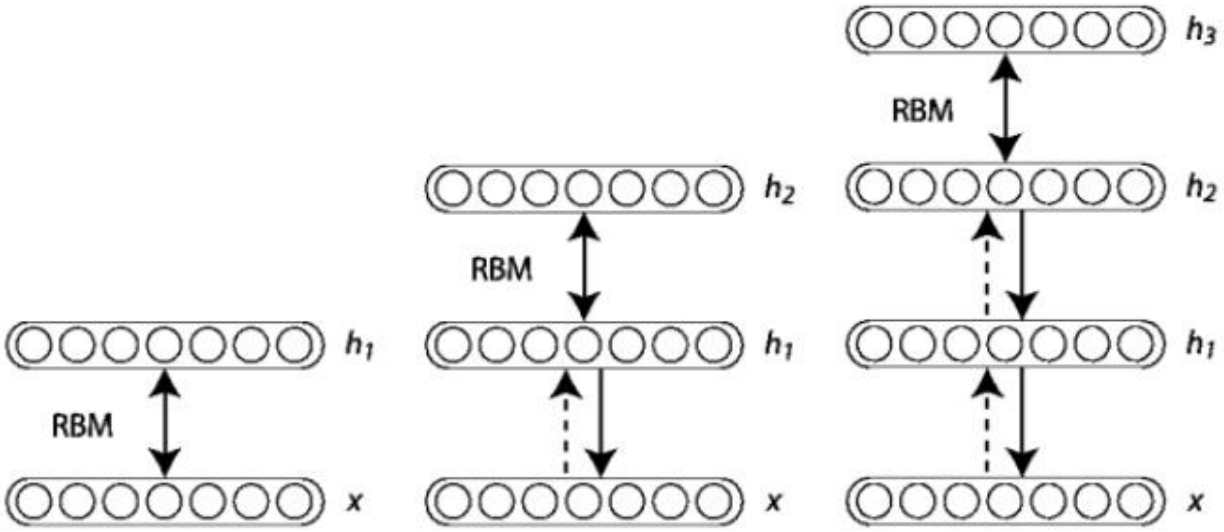
- Az algoritmus az eredeti inputból elindulva felváltva frissíti a rejtett, majd a visible réteget
  - Cél, hogy a kapott rekonstrukció minél jobban hasonlítson az eredeti inputhoz



  - Az ideális reprezentáció megtalálásához végtelen sok iteráció kellene
  - A gyakorlatban csak egyetlen iterációval közelítjük ezt, majd továbblépünk a következő adatvektorra
  - A fentieket iterálva áll elő a CD algoritmus

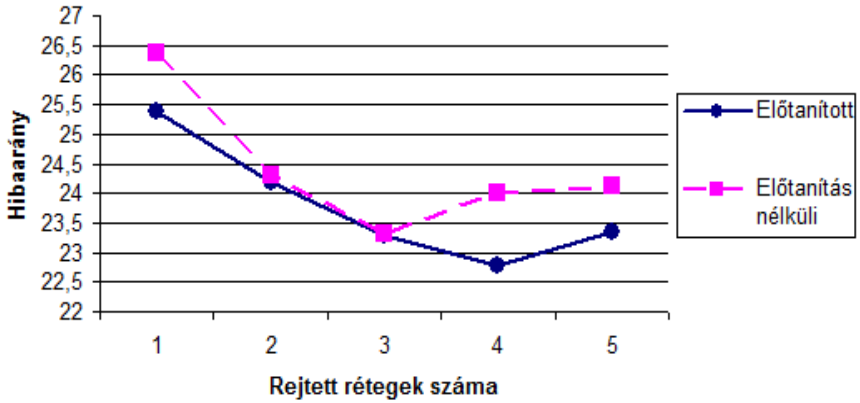
Deep Belief Network (DBN)

- A DBM-et RBM-ek egymásra pakolásával kapjuk meg
- A hálót nem rögtön egyben tanítja, hanem rétegenként építi fel
  - 1 rejtett réteg → tanítás
  - Újabb rejtett réteg hozzáadása → tanítás
  - …



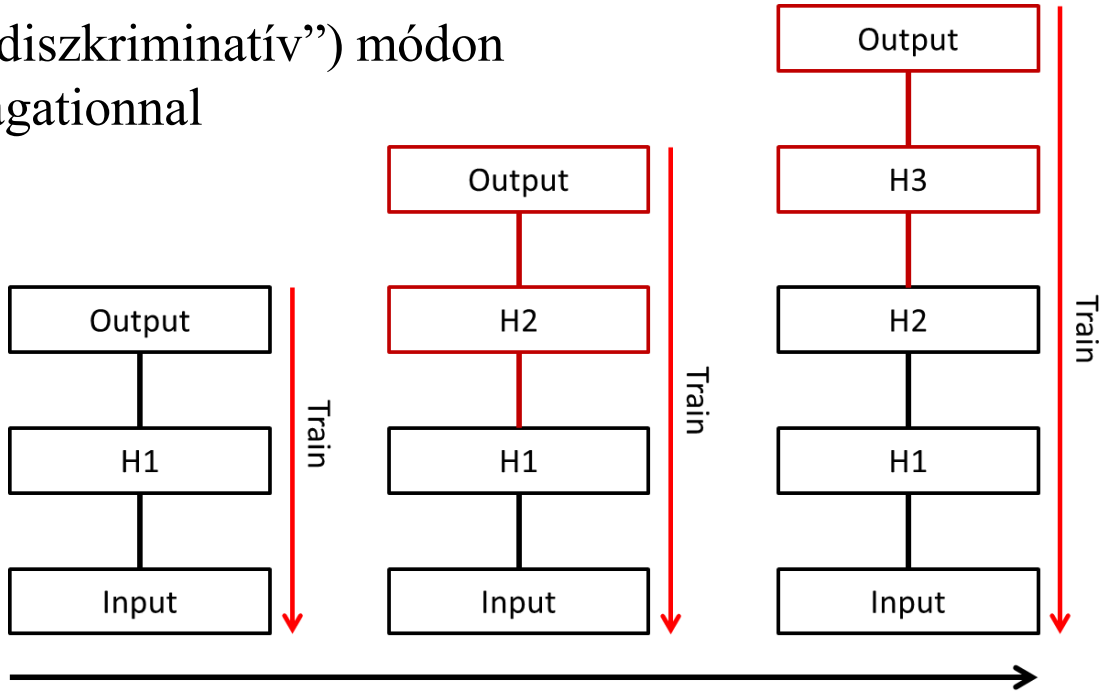
Konvertálás mély neuronhálóvá

- A DBN előtanítás után az RBM neuronjait hagyományos neuronokká konvertáljuk (nyilván a súlyok megtartásával)
- A tetejére rárakunk egy softmax réteget annyi neuronnal, ahány osztálycímkénk lesz
- És innentől elindíthatjuk a hagyományos, felügyelt backpropagation tanítást a címkézett példáinkat használva
- Az előtanítás szerepe tehát lényegében a súlyok inicializálása
- A random incializálásnál jobb eredményeket kapunk előtanítással (ld. korábbi példafeladat)



Diszkriminatív előtanítás

- Ez a módszer is a mély háló nehéz taníthatóságát igyekszik orvosolni
- Alapötletét a DBN-tanításból veszi: a hálót nem rögtön egyben tanítja, hanem a rétegeket egyenként adja hozzá
  - De: felügyelt („diszkriminatív”) módon tanít, backpropagationnal

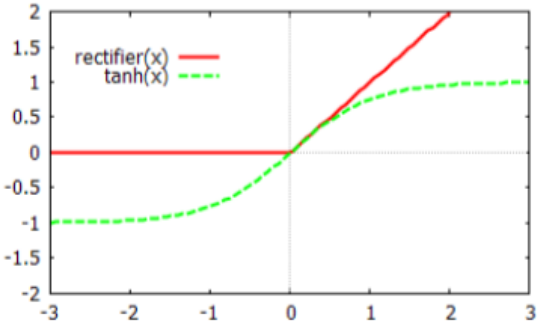


Diszkriminatív előtanítás (2)

- A módszer előnye: nagyon egyszerű implementálni, nem kell új tanítóalgoritmus, csak a backpropagation
- Hátrány: megnövekedett tanítási idő
  - Bár a tapasztalatok szerint az egyes rétegek hozzáadása után nem kell teljes konvergenciáig tanítani, elég pár iteráció („előtanítás”)
  - Teljes tanítást csak az utolsó rejtett réteg hozzáadása után végzünk
- A tapasztalatok szerint az „előtanítás” hatására a súlyok jobb kezdőértéket vesznek fel, mint ha véletlenszerűen incializálnánk őket
  - Ezért az utolsó, „teljes” tanítási lépés kisebb eséllyel akad el lokális optimumban

A rectifier aktivációs függvény

- Ez a módszer nem a tanításba nyúl bele, hanem lecseréli az aktivációs függvényt
- Az alábbi ábra a rectifier aktivációs függvényt mutatja
  - Képlete: $\phi(z) = max(z, 0)$
- Összehasonlítva a tanh aktivációs függvénnyel:



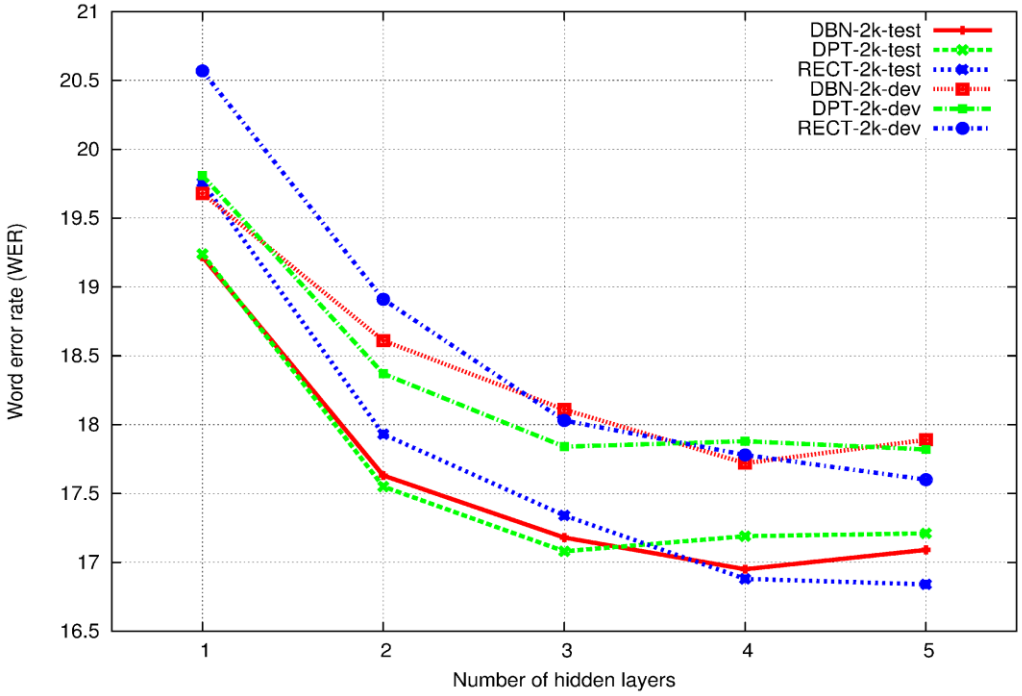
- Pozitív inputra
  - Megj: Az aktivációs függvénynek muszáj nemlineárisnak lenni!
- Pozitív inputra a deriváltja mindig 1, sehol sem tűnik el
- Negatív inputra a kimenet és a derivált is 0

A rectifier aktivációs függvény

- Előnye: pozitív inputra sosem 0 a derivált
- Hátrányok:
  - Negatív inputra viszont mindig 0
    - A gyakorlatban úgy tűnik, hogy ez nem okoz nagy problémát
  - Nem szimmetrikus (csak nemnegatív outputot ad)
    - A gyakorlatban ez se nagy probléma
  - Nem korlátos a kimenete, azaz végtelen nagy kimenetet is adhat
    - Vannak trükkök ennek a kezelésére, például a súlyok nagyságának időnkénti normalizálása

A három módszer összehasonlítása

- Kísérleti összehasonlítás beszédfelismerési adatokon a dev-test halmazokon
  - DBN: DBN-előtanítás, DPT: diszkriminatív előtanítás, RECT: rectifier háló



A tanítási idők összehasonlítása

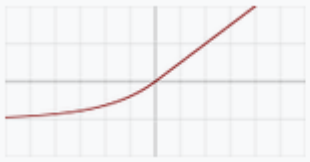
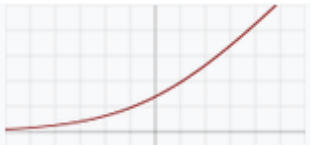
- A háromféle tanítási módszer futásideje a korábbi feladatra (öt rejtett réteg esetén):

Training method	Pre-training time	Fine-tuning training time
DBN pre-training	48 hours	14 hours
Discr. pre-training	9 hours	11 hours
Rectifier network	0 hours	14.5 hours

- Míg a három módszerrel kapott eredmények elég hasonlóak voltak, a rectifier háló tanítása sokkal kisebb időigényű, mivel nem kell előtanítani
- A „rectified linear” (ReLU) neuronokra épülő háló jelenleg a de facto standard a mély tanulásban

Még újabb aktivációs függvények

- Vannak javaslatok még újabb aktivációs függvényekre
  - Ezek a korábban említett hátrányokat próbálják korrigálni
  - Lásd még: https://en.wikipedia.org/wiki/Activation_function
- Példák: ELU, SELU, Softplus…

Exponential linear unit (ELU)[14]		$f(\alpha, x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$
SoftPlus[18]		$f(x) = \ln(1 + e^x)$

- Ezekkel néha kicsit jobb eredmények jönnek ki, mint a rectifier függvénnyel, de általános áttörést egyik sem hozott eddig

KÖSZÖNÖM A FIGYELMET!

A tananyag az EFOP-3.5.1-16-2017-00004 pályázat támogatásával készült.

REGULARIZÁCIÓ

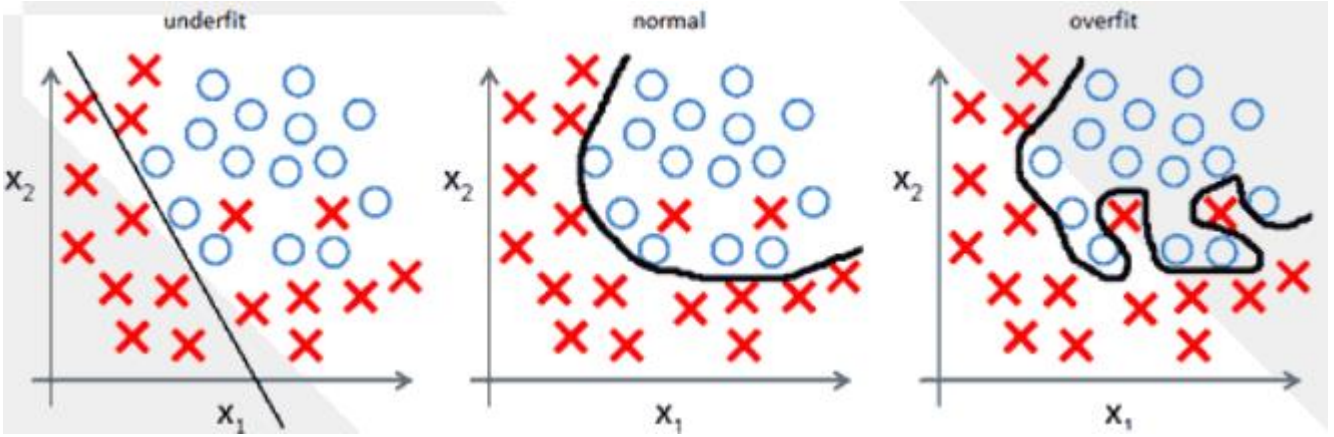
A tananyag az EFOP-3.5.1-16-2017-00004 pályázat támogatásával készült.

Mi az a túltanulás?

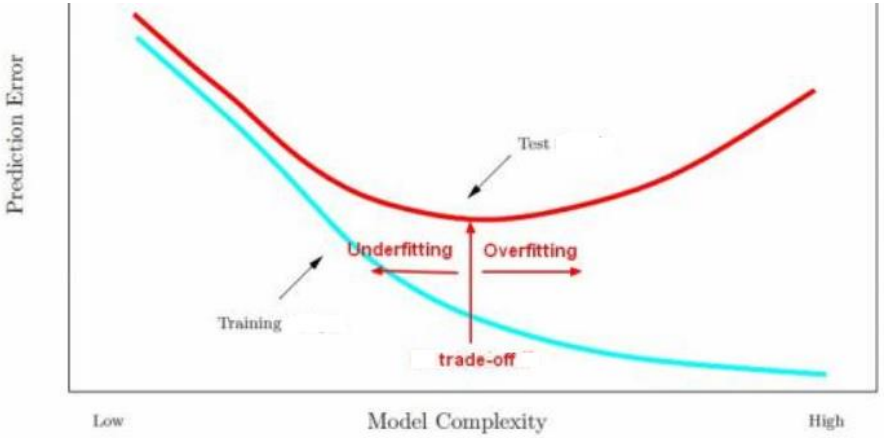
- Példa: leszáll a földön egy ufó, azzal a feladattal, hogy térképezze fel, milyenek az emberek
  - Lát 10 embert, és ebből levonja a következtetést: az embereknek két lábuk van, két kezük, egy fejük, és barna a szemük
  - Míg az eső három valóban igaz, a negyedik tulajdonság csak az adott 10 emberre teljesül → túltanulás
- Túltanulás (overfitting): a modell az általános tulajdonságok mellett az aktuális véges mintahalmaz egyedi furcsaságait is megtanulni
  - Nyilván csökkenthető az esélye a példaszám növelésével, de végtelen tér esetén véges számú példa sosem lesz tökéletesen reprezentatív
  - A gépi tanulási modell rugalmasságának növelésével nő a túltanulás esélye (a modell jobban tud az apró részletekre figyelni)
  - A túltanulás csökkentése miatt kulcsfontosságú, hogy a modellt a tanítóhalmaztól független teszthalmazon is kiértékeljük

A túltanulás jellemzői

- A modell méretének (komplexitásának) növelésével a modell rugalmassága nő, így egyre pontosabb lesz a tanítópéldákon



- Viszont a tesztpéldákon egy ponton túl romlani fog a teljesítménye
- Ezért lehet egy független teszt (vagy validációs) halmazon való kiértékeléssel megkeresni az optimális modell-komplexitást



A túltanulás neuronhálóknál

- Neuronhálók esetén a modell rugalmasságát alapvetően a neuronok és a rétegek számának növelésével növelhetjük
- Emellett backpropagation tanítás esetén a túltanulás jellemzően a tanítás vége felé jelentkezik

- Ezért a túltanulás ellen a legalapvetőbb eszközök:
  - A tanítás korai leállítása
  - A kis paraméterszámra való törekvés

„Early stopping”

- Ezt legkönnyebben egy validációs példahalmaz segítségével tudjuk megvalósítani
- Erről már beszéltünk a learning rate állításának ismertetésekor
  - Kicsit most frissítjük
- A learning rate frissítése validációs példahalmaz segítségével
  - A tanítás előtt tegyük féle az adatok egy részét → validációs halmaz
  - **Valamennyi tanítási lépés előtt mentsük el az aktuális súlyokat**
  - (pl. 1 epoch) után értékeljük ki a hibát a validációs halmazon
  - Ha az előző kiértékeléshez képest a hibacsökkenés kisebb, mint egy küszöb → learning rate felezése, új iteráció
  - **Ha hiba nőtt volna, visszaállítjuk az előző súlyokat**
  - Teljes leállás: ha a learning rate lement egy küszöb alá **vagy a validációs hiba több lépés során is nőtt**

Regularizáció

- A kis paraméterszámra való törekvés csökkenti a túltanulás esélyét
- Azonban a tanulás hatékonyságát is ronthatja
  - Adott paraméterszámú modellből nem tudjuk kihozni a maximumot, mivel az optimalizáló algoritmusaink nem garantálnak globális optimumot
- Ezért a gyakorlatban jobb stratégia paraméterszámban „túltervezni” a modellt, viszont kiegészíteni a hibafüggvényt egy plusz taggal, például:

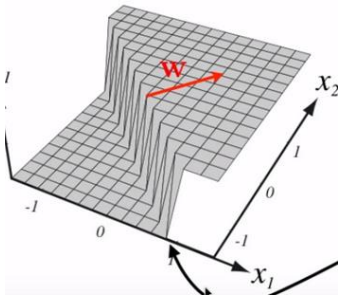
$$E = \frac{1}{2} * \sum (t_k - o_k)^2 + \frac{\lambda}{2} * \sum {w_i}^2$$

négyzetes hiba — regularizációs hiba

- Ez a regularizáció - a λ súllyal a regularizáció erőssége szabályozható
- A hibafüggvény regularizációs tagjának feladata, hogy az optimalizálás során az egyforma hibájú, de különböző alakzatot leíró modellek közül az optimalizáló az „egyszerűbbet”, kevésbé „túltanultat” preferálja
  - Ennek gépi tanulási modellenként más-más lehet az értelmezése, most megnézzük neuronhálók esetére

Neuronhálók regularizációja

- Neuronhálók esetén a döntési felület sok-sok hipersíkból áll össze, közvetlen módon nem tudjuk befolyásolni a simaságát
- Nézzük, hogy mit tehetünk pl. egyetlen neuron esetén
  - Az aktivációk egy hipersíkra esnek: $a = \sum_{i=0}^{n} w_i x_i$
  - A döntési felület egy hipersík, ahol $0 = \sum_{i=0}^{n} w_i x_i$
  - A sigmoid függvény a hipersík két végét 0-hoz és 1-hez „görbíti”



- Ha súlyokat megszorozzuk egy nagy konstanssal, akkor a döntési felület ugyanott marad, de a hipersík meredekebb lesz, így a sigmoid meredekebben tart 0-hoz ill. 1-hez
  - Azaz nagyobb súlyok esetén a neuron „határozottabb” döntést hoz, amit a tanulás elején nem szeretnénk (túltanulás)
  - Korábban megbeszéltük, hogy a tanításra nézve is rossz (a derivált kicsi lesz), ha a $\sum_{i=0}^{n} w_i x_i$ érték nagy
  - Ezért inicializáltuk a súlyokat kis értékekkel, ezért normalizáltuk a jellemzőket 1 szórásúra

Weight decay

- A legegyszerűbb regularizációs módszer a hálót a kis súlyok preferálására készteti – ez a weight decay
- Képlettel (itt most négyzetes hiba esetére, de az alap hibafüggvény más is lehet):

$$E = \underbrace{\frac{1}{2} * \sum (t_k - o_k)^2}_{\text{plain error}} + \underbrace{\frac{\lambda}{2} * \sum w_i^2}_{\text{L2 weight penalty}}$$

- Miért hívják weight decay-nek?
  - Emlékezzünk, hogy a backprop szabály szerint $w_i = w_i - \eta \frac{\partial E}{\partial w_i}$
  - A fenti regularizációs hibafüggvénnyel $w_i = w_i - \eta\lambda w_i = (1 - \eta\lambda) w_i$
  - Azaz a súlyokat meg kell szorozni egy egynél picit kisebb konstanssal
  - Ennek hatására a súlyok - legalábbis amelyeket a négyzetes hiba nem akar mindenáron nagy értékűnek megtartani - a nulla felé tartanak
- Hívják L2 regularizációnak is (a súlyvektor 2-es normája a regularizációs tag)

Sparse neuronhálók

- A kettes helyett az egyes normát is használhatjuk – ez lesz az L1 avagy „Lasso” regularizáció

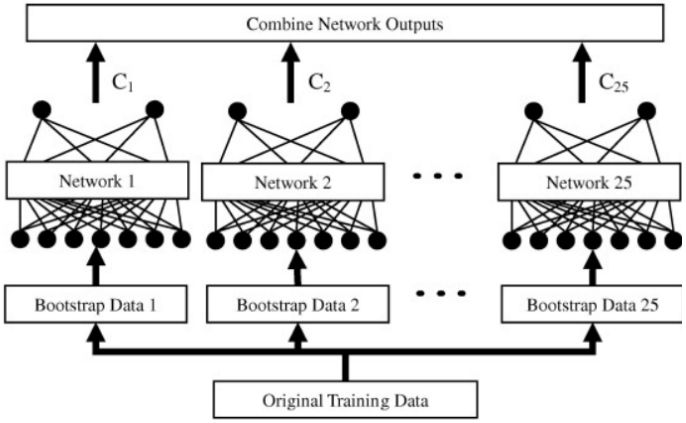
$$E = \frac{1}{2} * \sum (t_k - o_k)^2 + \lambda * \sum |w_i|$$

squared error L1 weight penalty

- Ilyenkor a súlyokat nem egy egynél kisebb konstanssal szorozzuk, hanem egy kis konstanst hozzáadunk-kivonunk (előjeltől függően)
  - Ez jóval aggresszívebben regularizál, nem csupán nulla felé nyomja a súlyokat, hanem ki is tudja nullázni
- Nem csak a súlyokra, hanem a neuronok kimenetére is használhatjuk, így a kevésbé fontos neuronokat is kinullázás felé nyomjuk (pl. ReLU neuronokkal, mi ténylegesen is tud 0 kimenetet adni)
- Ezzel a hálót „fully connected” helyett ritka (sparse) hálóstruktúra felé tudjuk irányítani (élek vagy teljes neuronok eltüntetése)
  - Így tkp. visszajutunk az eredeti túltanulás elleni felvetéshez (kevés paraméter), csak egyfajta „soft” változatban

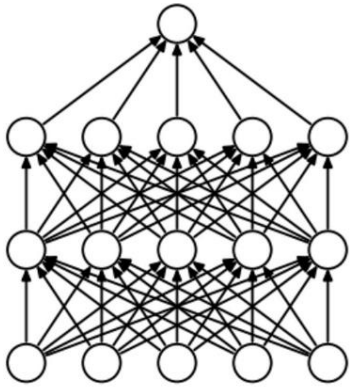
Ensemble neural networks

- Ha nem nagyon nagy a tanító adatbázis, akkor megtehetjük, hogy a hálót sokszor betanítjuk
  - A véletlen inicializálás és az adatok random bejárása miatt kicsit eltérő súlyokat fogunk kapni
  - De azt is megtehetjük, hogy az adatok valamilyen véletlenszerű (vagy egyéb, ravaszabb módon kiválasztott…) részhalmazain tanítunk
- A kapott modellek kimeneteit végül kiátlagoljuk, így egy „ensemble” modellt kapunk
  - Ez az átlagolás is stabilizálja a modellt, hiszen az áltagolás miatt kevésbé lesz érzékeny az egyes részhalmazok megválasztásából eredő esetleges egyedi furcsaságokra

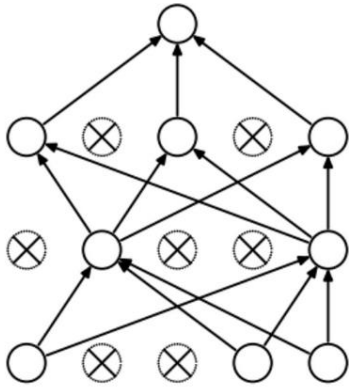


Dropout

- A tanítás során az egyes neuronokat bizonyos valószínűséggel (pl. 0.2) „kiejtjük”, kinullázzuk (az adott input vektor esetén, a következő példánál már „élni” fog!)
- Ez a neuronokat arra kényszeríti, hogy önállóan tanuljanak, kevésbé támaszkodjanak a szomszédokra
- Miért regularizál?
  - Megmutatható, hogy olyan hatása van, mint ha a háló össze részhálóját, azaz 2^N hálót tanítanánk párhuzamosan, és ezeket átlagolnánk



(a) Standard Neural Net



(b) After applying dropout.

- Technikai problémák
  - Lassítja a tanulást, jóval (5-10x) többször kell végigmenni az adatokon
  - Egy adott neuronnak kevesebb inputja lesz → a megmaradt inputokat felszorozzuk egy konstanssal, hogy az összeg hasonló skálára essen

N-szeres keresztvalidáció

- Ezt a módszert akkor szoktuk használni, ha tényleg nagyon kevés tanítópéldánk van
  - Luxus lenne validációs célra „elherdálni” akár a példák 10% is.
- A példáinkat felosztjuk N egyenlő részre (mondjuk N=10)
- 9 részen tanítunk, a tizediken validálunk
  - És ezt megismételjük 10-szer, mindig másik halmazt kihagyva validálásra
  - Ily módon az összes adaton tudunk tanítani, de azért a független adaton való validálás is megvolt
  - Validálási pontosság: a 10 validálási pontosság átlaga
- De hogyan lesz egy végleges modellünk a 10-ből?
  - Átlagolhatjuk a modellek kimeneteit (ld. ensemble model)
  - Esetleg csinálhatunk egy végleges tanítást (validálás nélkül!) az összes adaton a legjobbnak bizonyult meta-paraméterekkel
- Extrém eset: „leave-one-out” – egyetlen példát hagyunk ki validálásra

Data augmentation

- Ez nem regularizációs módszer, de a túltanulás esélyét csökkenti
- Tudjuk, hogy a túltanulás fő oka a kevés tanítópélda
- Hogyan tudunk a meglevőkből mesterségesen további tanítópéldákat kreálni?
- A példákon kisebb transzformációkat alkalmazunk, olyanokat, amelyek az osztálycímkét nem befolyásolják
  - Pl. karakterfelismerés: eltolás, kicsinyítés-nagyítás, kisebb fokú elforgatás
  - Pl. beszédfelismerés: pár százaléknyi lassítás-gyorsítás, háttérzaj hozzáadása
- Ezekkel csökkenthetjük a háló érzékenységét ezekre a transzformációkra, így nőni fog az általánosító képessége

Noise injection

- Pici (általában normál eloszlású) zajt adunk az inputhoz, vagy akár az egyes neuronok kimenetéhez is
  - Tulajdonképpen a data augmentation egy speciális, „buta” esete (amikor mondjuk nem tudjuk, hogy hogyan érdemes az adatokat transzformálni)
  - A tapasztalatok szerint ez is segít a túltanulás ellen, főleg amikor nagyon kevés a tanító adat
  - De olyan változatok is vannak, ahol a hibafüggvény deriváltjához, sőt akár a tanítócímkékhez adnak zajt

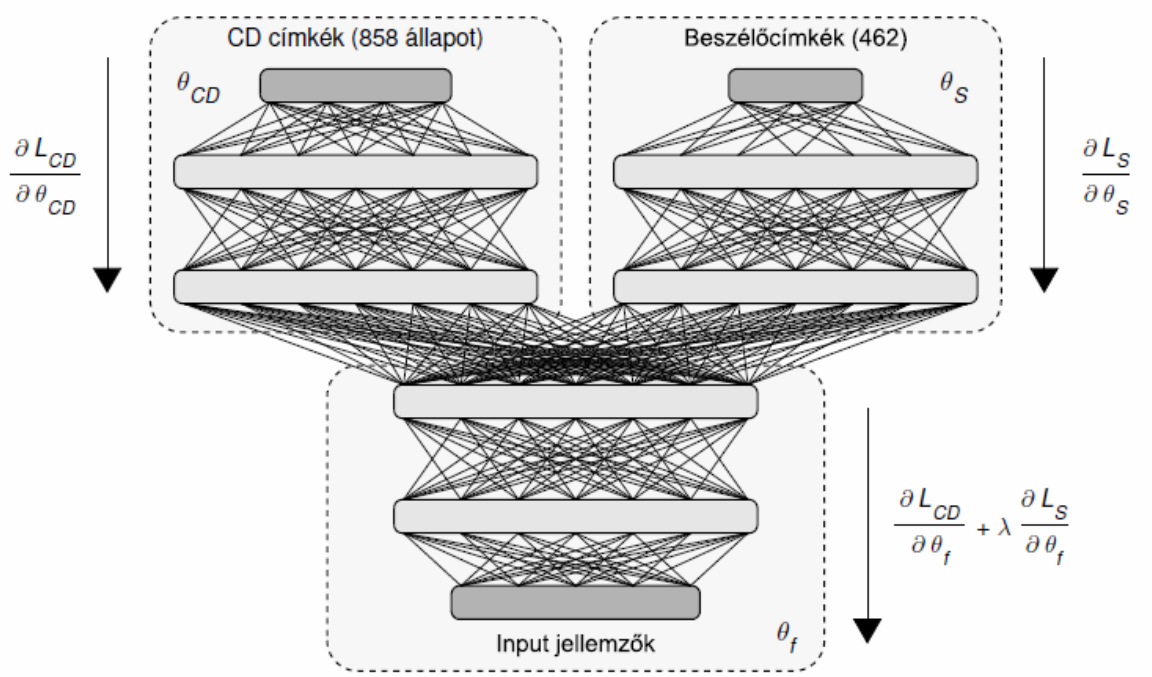
  - https://machinelearningmastery.com/train-neural-networks-with-noise-to-reduce-overfitting/

Multi-taszk tanítás

- A neuronhálónak párhuzamosan több (általában 2) tanulandó feladatot adunk – ehhez speciális hálózati struktúra kell
- Alapvetően arra találták ki, hogy két feladatot egyszerre oldjanak meg, de:
  - Megfigyelték, hogy gyakran mindkét feladaton jobb eredményt kapnak, mint ha külön-külön hálót tanítanának rájuk
  - Azért, mert az „egyben” tanulásnak regularizáló hatása van
- Példa beszédfelismerésből (ld. következő ábra): a főfeladat a beszédhangok felismerése, a másodlagos feladat a beszélő személyének felismerése

Multi-taszk hálóstruktúra

- A két feladat párhuzamos megoldásához speciális hálózati struktúra kell:
  - Közös input réteg és alsó rejtett rétegek
  - Feljebb feladat-specifikus rétegek
  - Két kimenő réteg
  - Két hibafüggvényt minimalizálunk
  - Hibavisszaterjesztés során a két ágból érkező hibát súlyozottan összegezzük (λ paraméter)



Multi-taszk tanítás

- Nyilván akkor van értelme, ha a két feladat hasonlít, de nem ugyanaz
- λ mellett egy további paraméter az is, hogy milyen mélységben ágaztassuk el a hálót
  - Ésszerűnek tűnik, hogy minél eltérőbbek a feladatok, annál korábbi rétegbe kell helyezni az elágazást (kevesebb közös és több feladatspecifikus réteg kell)
  - De ezt jelenleg csak kísérleti úton lehet belőni
- Mire jó a multi-taszk tanítás (amellett, hogy 2 feladatot oldunk meg egyszerre)?
  - A hálónak a közös rétegekben olyan reprezentációt kell találnia, ami mindkét feladat megoldását segíti
  - Ennek regularizáló hatása van (kisebb túltanulás, jobb általánosítás)
  - Ezért gyakran az eredmények is jobbak mindkét feladaton, mint ha külön-külön oldanánk meg őket

Ellenséges (adversarial) multi-taszk tanítás

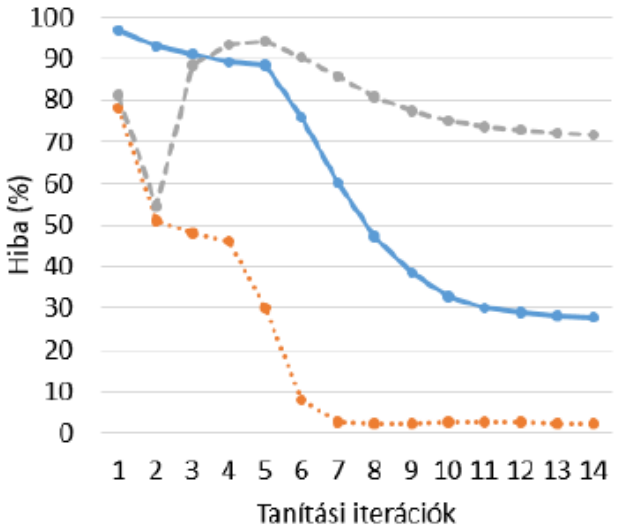
- Egy kis csavart visz a multi-taszk modellbe
- A másodlagos feladat hibáját minimalizálás helyett maximalizáljuk
- Technikai kivitelezés: λ értékét negatívnak választjuk, a másodlagos ág hibáját továbbra is minimalizáljuk
  - A feladatspecifikus rétegek továbbra is minimalizálásra törekednek
  - Az előjelváltás a legfelső közös rétegben fejti ki a hatását
  - Csak egy előjelet kell módosítani a kódban
  - λ abszolút értékét fokozatosan, c lépésben érdemes növelni
  - Azaz a k-adik iterációban

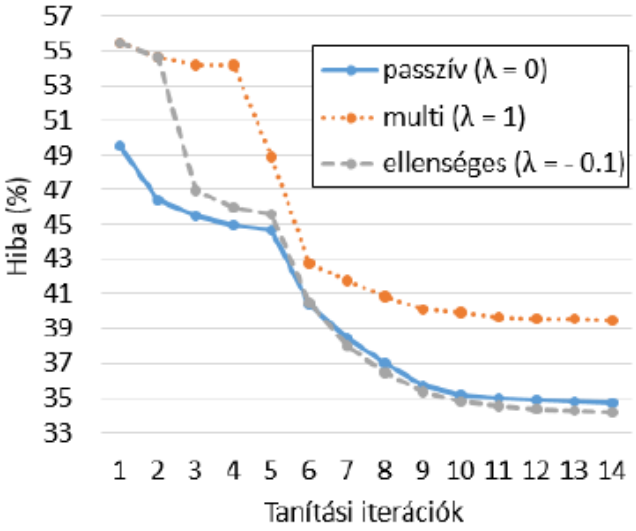
$$\lambda_k = min(\frac{k}{c}, 1) \cdot \lambda$$

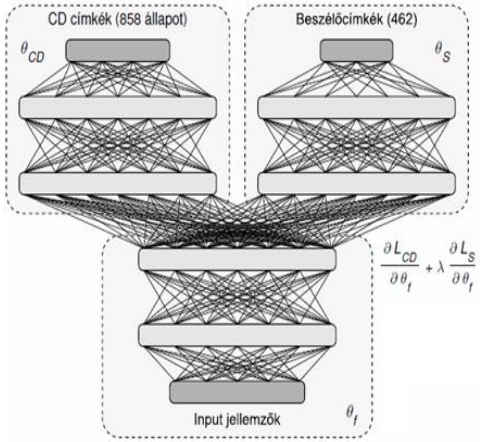
- Hatása: a háló olyan közös reprezentációt keres, ami alapján a másodlagos feladatot minél kevésbé lehet megoldani
  - Az előző példában beszélőfüggetlen reprezentációt fog keresni

A tanítás szemléltetése a példán

- Bal oldal: A másodlagos (beszélőfelismerési) ág a tanítóhalmazon
- Jobb oldal: a fő feladat hibája a fejlesztési halmazon







- „passzív” tanulás: a másodlagos ág tanul, de nem szólhat bele a közös rétegekben kialakuló reprezentációba (a főág „egyfeladatosan” tanul)
- Eredmény: a multi-taszk ront, az ellenséges multi-taszk kicsit javít az elsődleges (beszédfelismerési) feladat pontosságán

KÖSZÖNÖM A FIGYELMET!

A tananyag az EFOP-3.5.1-16-2017-00004 pályázat támogatásával készült.

TOVÁBBI TANÍTÁSI MÓDSZEREK

A tananyag az EFOP-3.5.1-16-2017-00004 pályázat támogatásával készült.

SZÉCHENYI 2020
MAGYARORSZÁG KORMÁNYA
Európai Unió
Európai Szociális Alap
BEFEKTETÉS A JÖVŐBE

Backpropagation és alternatívái

- Messze a legnépszerűbb, legelterjedtebb tanító algoritmus
  - Egyszerű alapelv, könnyű implementálni
  - Csak a derivált kell hozzá
  - Batch módban is használható, így nem muszáj az összes tanító adatot egyszerre betölteni, manipulálni
- Az alap backpropagation algoritmusnak van több finomítása
  - Alapvetően a derivált és a learning rate számítását próbálják finomítani
  - Régebben: Momentum módszer, Quickprop, resilient backprop, …
  - Újabban: Adagrad, Adadelta, Adam, …
- Fejlettebb optimalizálási technikák – mit várunk tőlük?
  - Globális optimum – ez nem fog menni…
  - Jobb lokális optimum
  - Gyorsabb konvergencia
  - Kevesebb kézileg állítandó meta-paraméter (pl. learning rate és felezése)

A momentum módszer

- Emlékezzünk vissza, hogy hogyan változtattuk a súlyokat:

$$\Delta w_{ij} = \eta * \frac{\partial E}{\partial w_{ij}}$$

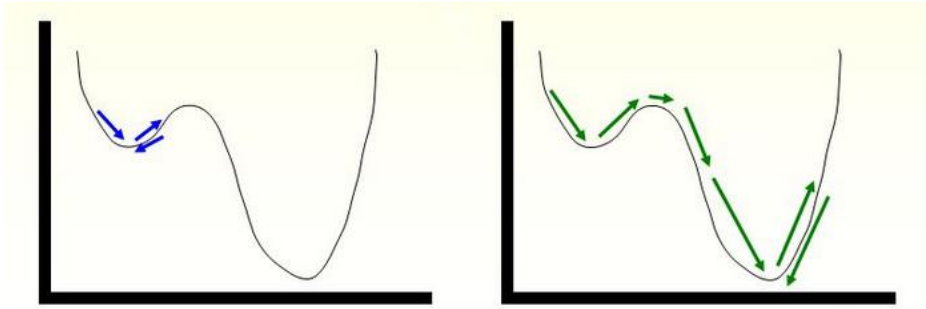
weight increment — learning rate — weight gradient

- Most ezt kiegészítjük egy „momentum” taggal:
  - Ahol a „momentum faktor” egy 0 és 1 közötti érték

$$\Delta w_{ij} = \eta * \frac{\partial E}{\partial w_{ij}} + (\gamma * \Delta w_{ij}^{t-1})$$

momentum factor — weight increment, previous iteration

- Ennek az a hatása, hogy időben elsimítja a Δ értéket, „lendületet” ad neki
- Átsegíthet kisebb lokális optimumokon



Adagrad

- Minden egyes paraméterhez *külön* learning rate
- A learning rate-et leosztja az eddigi deriváltak négyzetes átlagával
  - A nagyobb deriváltakat produkáló paraméterekhez kisebb learning rate
  - A kisebb deriváltúakhoz nagyobb learning rate fog tartozni
- Előnyök:
  - Kiegyensúlyozza az egyes paraméterek Δ értékének nagyságát
  - Nem kell menet közben állítani a learning rate-et, magát szabályozza
- Hátrány:
  - Mivel az összes korábbi deriváltat összeadja, a learning rate túl gyorsan elfogy
  - Gyakori tapasztalat, hogy lassabban konvergál, illetve kicsit rosszabb eredményt ad, mint egy jól beállított backprop momentummal

Adadelta, Adam, …

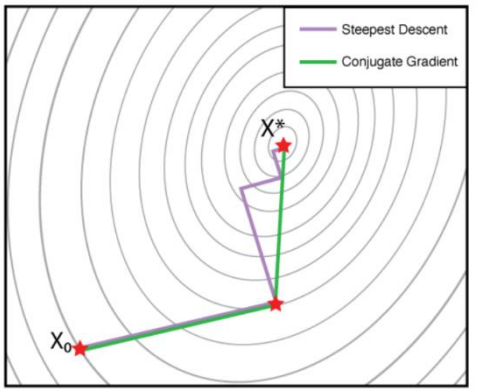
- Adadelta: az összegzésnél csak a k legutóbbi deltát, vagy az összeset, de exponenciálisan csökkenő súllyal veszi figyelembe
  - Ezzel kiküszöböli az Adagrad túl gyorsan elfogyó learning rate-jét
- Adam:
  - Szintén futó átlagot számol, de nem csak az átlagot, hanem a szórást is nézi
- Vannak még újabb finomítások is:
  - AdaMax, Nadam, Eve, …

Kifinomultabb optimalizálási módszerek

- Newton-módszer és rokonai
  - Kell hozzá a második derivált is → sokváltozós esetben a Hesse-mátrix
    - n változó estén ez *n*x*n*-es
    - Neuronhálók esetén ez túl nagy
    - Én invertálni is kellene minden iterációs lépésben…

$$\mathbf{H} = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}.$$

- Kvázi Newton-módszerek – csak közelítik a Hesse-mátrixot
  - Pl. L-BFGS
- Konjugált gradiens módszerek – szintén elkerülik a Hesse-mátrix kiszámítását és invertálását
  - Adott irányban line search + mindig az előző irányokra merőlegesen lép
  - n-dimenzós kvadratikus felület optimumát n lépésben megtalálja
  - Nem kvadratikus felület – a fentit kell ismételgetni

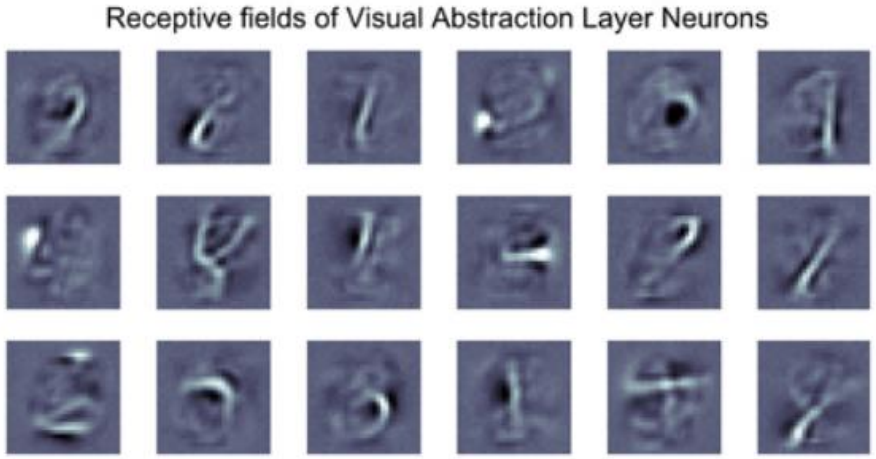


Kifinomultabb optimalizálási módszerek

- Összehasonlító kísérletekben általában a backpropagation hasonlóan, vagy csak kicsit rosszabbul teljesít
  - Érdemes lehet a két módszercsaládot kombinálni, pl. a tanulás elején a hiba gyorsan csökkenthető backpropagationnal, a végén pedig be lehet vetni a konjugált gradienst, vagy a kvázi-Newton módszereket
- Általános vélemény jelenleg:
  - Annyival bonyolultabbak matematikailag és implementációs szempontból, hogy nem éri meg az a kis javulás/gyorsulás, mit bizonyos feladatokban adnak

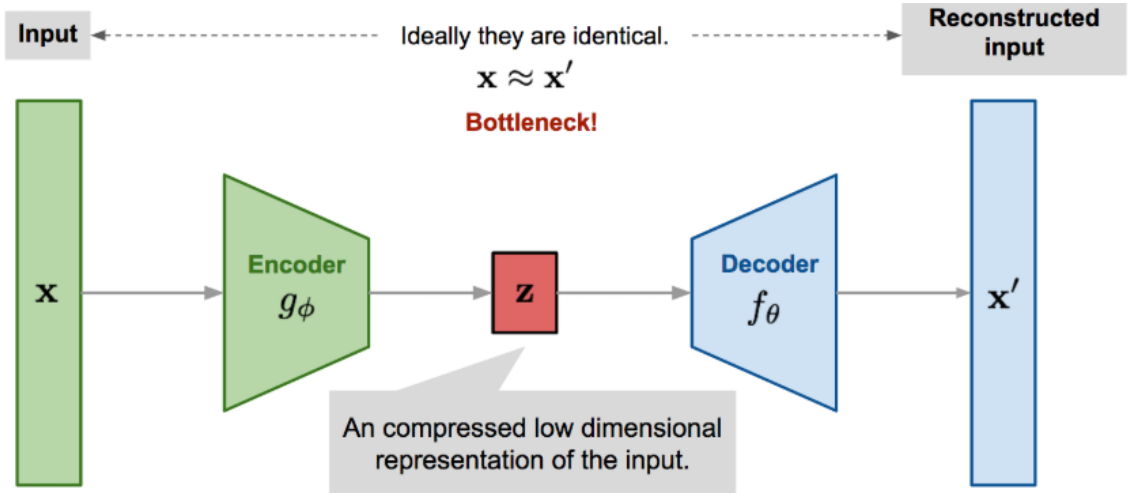
DBN-ek jelenlegi alkalmazása

- A DBN-ek CD-kritériummal való tanítása *felügyelet nélküli* tanítás
  - Továbbra is lehet értelme a DBN-előtanításnak, ha rengeteg címkézetlen adatunk van, és csak kevés címkézett
- Az RBM két rétege köz szimmetrikus a kapcsolat
  - Ezért a rejtett rétegből vissza lehet generálni az inputot
  - Könnyű vizualizálni, hogy milyen rejtett reprezentációt alakított ki



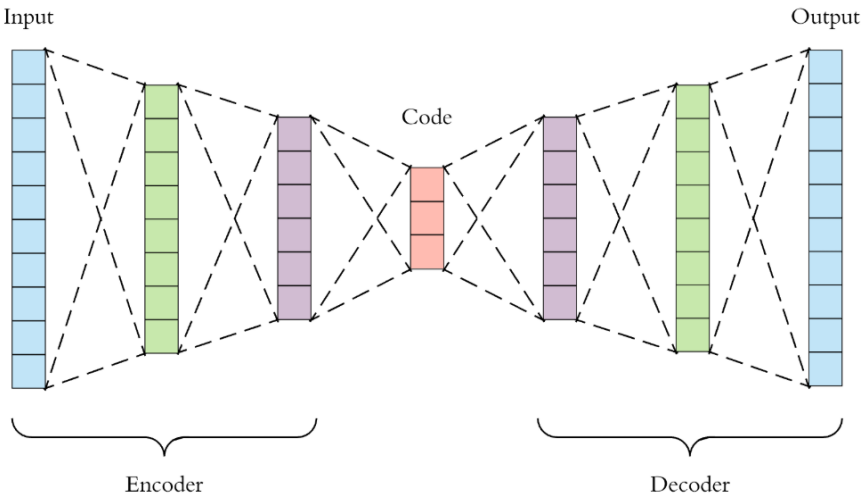
Mély hálók felügyelet nélküli tanítása

- A standard mély hálókat is taníthatjuk felügyelet (értsd: osztályokat megadó tanító címkék) nélkül
  - A leggyakoribb ilyen hálóstruktúra az ún. autoencoder háló, amikor a megtanulandó címkék megegyeznek az inputtal
  - A háló az inputját igyekszik rekonstruálni az outputon
  - Az autoencoder háló tipikusan homokóra alakú (az egyre kisebb rétegekkel próbáljuk rákényszeríteni a tömörítésre)
  - a középső, legkisebb réteget hívják „bottleneck” rétegnek



Autoencoder hálózatok

- Mire jó az autoencoder háló?
  - A tanulás során rákényszerítjük a hálót, hogy találjon egy tömör reprezentációt, amiből az input minél pontosabban rekonstruálható
  - Ezért használhatjuk jellemzőtér-redukcióra (a decoder részt eldobjuk, az eredetei jellemzők helyett a bottleneck réteg kimenetét használjuk)
  - De felügyelet nélküli előtanításra is jó (ha sok címkézetlen adatunk van, és kevés címkézett, betanítjuk vele az autoencodert, majd a decoder részt eldobjuk, és új kimeneti réteg(ek)et felvéve innen kezdjük a felügyelt tanítást)

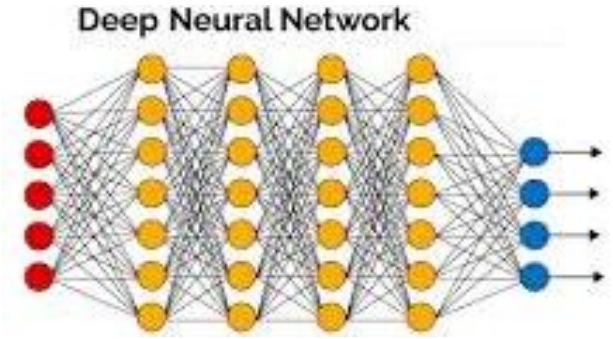


Batch normalizálás

- Azt mondtuk, hogy az input jellemzők normalizálása 0 átlagra és 1 szórásra segíti a tanulást

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

(Ez tkp. egy skálázás és egy eltolás)



- Mély háló esetén a k. réteg inputja a k-1. réteg outputja
- De akkor a k-adik réteg is könnyebben tanulna, a k. réteg kimenete 0 átlagú és 1 szórású lenne – *ehhez tudni kéne a skálázás és eltolás értékét*
- Megtehetnénk, hogy először csak az első réteget tanítjuk be, aztán kiszámolnánk a kimenet várható értékét és szórását, normalizálnánk, majd hozzáadnánk a hálóhoz egy új réteget, és így tovább…
  - De ez így lassú és nehézkes

Batch normalizálás 2

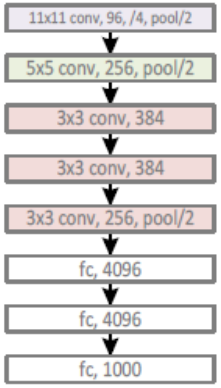
- Hatékonyabb megoldás, ha a normalizálást „online” végezzük, azaz mindig csak az adott batch-re
- De eközben a megfelelő skálázó és eltoló paramétereket megtanuljuk
  - Ugyanúgy tanulandó paraméternek tekintjük őket, mint a háló súlyait
  - Ezekre is kiterjesztjük a backpropagation algoritmust, és a tanulás során megtanuljuk
  - A tesztelés során már ezeket a tanult értékeket haszáljuk
- A batch normalizálás jótékony hatásai:
  - Gyorsabb konvergencia
  - Nagyobb kiindulási learning rate-et lehet használni
  - Mély hálók jobb eredménnyel taníthatók
  - A háló kevésbé lesz érzékeny a súlyok inicializálására

Nagyon mély neuronhálók

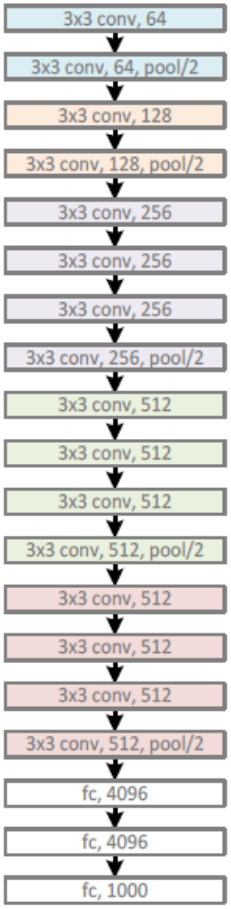
- A „mély háló” fogalma nincsen pontosan definiálva, de általában 5-10 rejtett réteggel rendelkező hálót jelent
- Újabban azonban megjelentek a „nagyon mély” (very deep) hálók is, tipikusan képfeldolgozási feladatokra
  - Ez sincs pontosan definiálva, de már 50-150 rejtett réteges hálókat is látni
- A nagyon mély hálók tanításánál már a korábban mutatott trükkök sem segítenek a hiba visszaterjesztésében
  - ReLU neuronok, ügyes inicializálás, batch normalization,…
- A mélyen levő rétegek hatékony tanításához új trükkök kellenek

Nagyon mély neuronhálók - példák

AlexNet, 8 layers
(ILSVRC 2012)



VGG, 19 layers
(ILSVRC 2014)



GoogleNet, 22 layers
(ILSVRC 2014)

ILSVRC (Large Scale Visual Recognition Challenge)

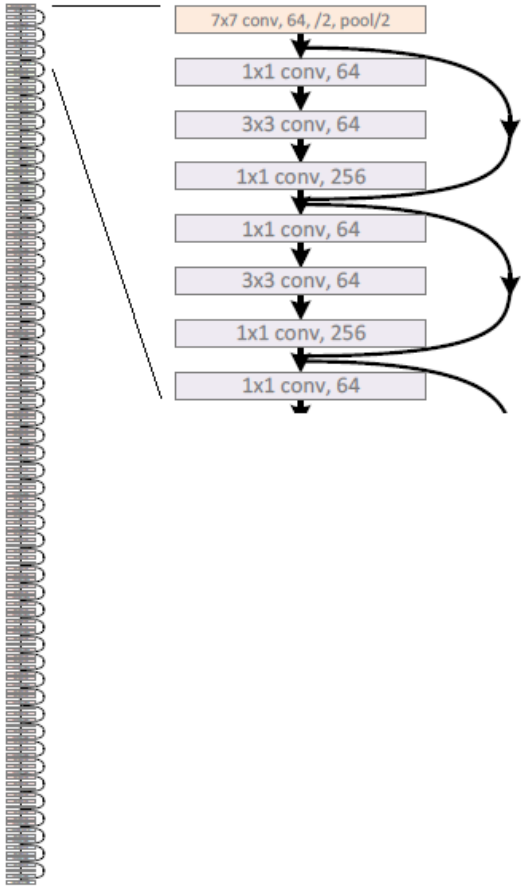
(slide credit: Jian Sun, MSR)

Nagyon mély neuronhálók - példák

AlexNet, 8 layers
(ILSVRC 2012)

VGG, 19 layers
(ILSVRC 2014)

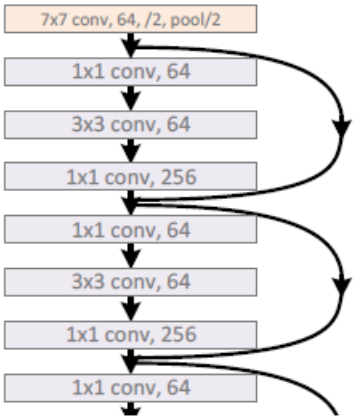
ResNet, 152 layers
(ILSVRC 2015)



ILSVRC (Large Scale Visual Recognition Challenge)

Nagyon mély hálók extra kapcsolatokkal

- A mély (távoli) rétegekbe visszaterjesztéskor a hiba sok rétegen megy át
  - Minden rétegben kicsit módosul – egyre nő a kockázata, hogy elvész vagy „felrobban”
- A javasolt megoldásokban közös, hogy új kapcsolatokkal egészítik ki a hálót, amelyek mentén a hiba kevesebb transzformáción átesve csatolódik vissza
- Néhány gyakori elnevezés ezekre a kapcsolatokra, hálókra:
  - „linear pass-through” vagy „jump” vagy „skip” connection
- És az ilyen hálók nevei:
  - „Highway network”
  - „Residual network”
  - „Linearly augmented network”

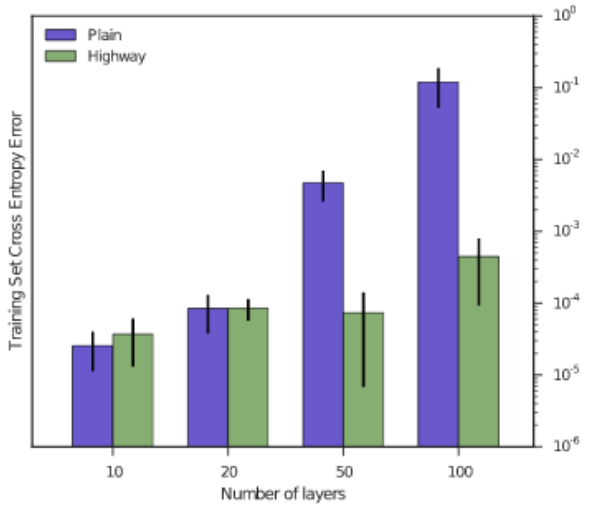


Highway Network

- Az új kapcsolatok egy ”information highway-t” alkotnak, ahol az információ transzformáció nélkül áramolhat
- Alapötlet: a rekurrens hálóknál is sok rétegen át kellett visszaterjeszteni a hibát, az LSTM kapuzós megoldása segített ebben
  - Hagyományos réteg kimenete: $\mathbf{y} = H(\mathbf{x}, \mathbf{W_H})$
  - Highway háló egy rétege: $\mathbf{y} = H(\mathbf{x}, \mathbf{W_H}) \cdot T(\mathbf{x}, \mathbf{W_T}) + \mathbf{x} \cdot (1 - T(\mathbf{x}, \mathbf{W_T}))$
    - Ahol T a „transfer gate”: $T(\mathbf{x}) = \sigma({\mathbf{W_T}}^T \mathbf{x} + \mathbf{b_T})$
    - T így 0 és 1 közötti értékeket vehet fel
    - A kimenet *T*=0 ill. *T*=1 esetén:

$$\mathbf{y} = \begin{cases} \mathbf{x}, & \text{if } T(\mathbf{x}, \mathbf{W_T}) = \mathbf{0}, \\ H(\mathbf{x}, \mathbf{W_H}), & \text{if } T(\mathbf{x}, \mathbf{W_T}) = \mathbf{1}. \end{cases}$$

    - Azaz T szabályozza a transzformálatlan x és a transzformált H(x) arányát
    - Példa: beszédfelismerési hiba



Linearly Augmented Network

- A kapuzásnál egyszerűbb megoldást használ
- A hagyományos háló egy rétegének kimenete
  - (korábbi jelölésünkhöz képest V egy extra lineáris trafó):

$$f_i(x) = V_i \phi(U_i x + b_i)$$

- A lineárisan kiegészített réteg viszont ezt fogja számolni:

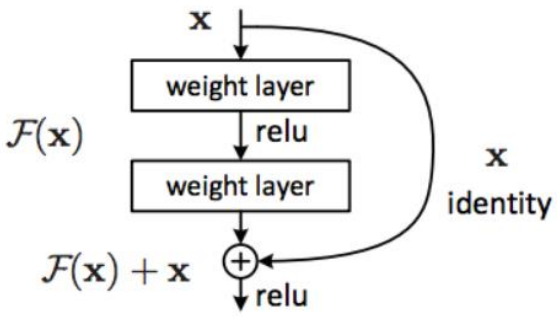
$$f_i(x) = V_i \phi(U_i x + b_i) + T_i x$$

  - Azaz az output egy hagyományos nemlineáris és egy lineáris transzformáció összegeként áll elő
  - T nem feltétlenül teljes transzformációs mátrixot jelent, diagonális mátrixszal, illetve rögzített (nem tanítható) egység-mátrixszal is jó eredményeket kaptak
  - Példa: beszédfelismerési eredmények

Num of H.Layers	Layers Size	# Params	PER %
3	1024X256	2.9M	**22.39**
12	512X256	3.8M	21.8
48	256X128	3.4M	**21.7**

Residual network

- Abból indul ki, hogy a kapuzás fölösleges túlbonyolítás
- Sőt, még a lineáris transzformáció sem kell
- Az extra kapcsolaton az input transzformáció nélkül jut el az outputig („identity mapping”)
- Ha a korábbi rétegek már megoldották a feladatot, az identitás leképezés elég, F(x)-nek már nincs mit tanulni
- Amit az eddigi rétegek még nem tanultak meg, azt a maradék („residual”) hibát kell visszacsatolni és megtanulni a nemlineáris F(x) leképezés segítségével



KÖSZÖNÖM A FIGYELMET!

A tananyag az EFOP-3.5.1-16-2017-00004 pályázat támogatásával készült.

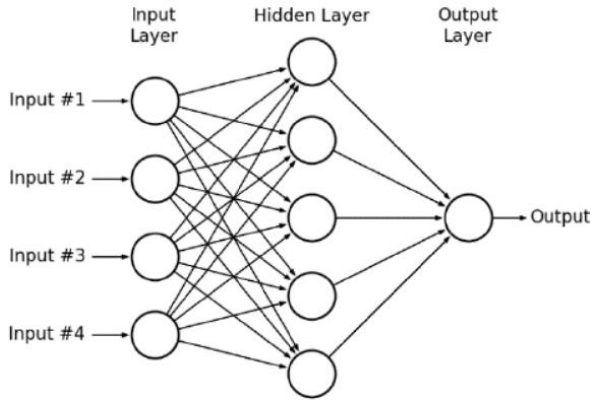
KONVOLÚCIÓS NEURONHÁLÓK

A tananyag az EFOP-3.5.1-16-2017-00004 pályázat támogatásával készült.

SZÉCHENYI 2020
MAGYARORSZÁG KORMÁNYA
Európai Unió
Európai Szociális Alap
BEFEKTETÉS A JÖVŐBE

1. motiváció

- A klasszikus neuronháló struktúra a „fully connected” háló
  - Két réteg között minden neuron kapcsolódik minden neuronnal
  - Az egyes inputoknak teljesen egyforma szerepük van
  - Ha az inputokat véletlenszerűen permutáljuk (de minden egyes tanító vektort ugyanúgy!), akkor a háló ugyanolyan eredményesen fog tanulni

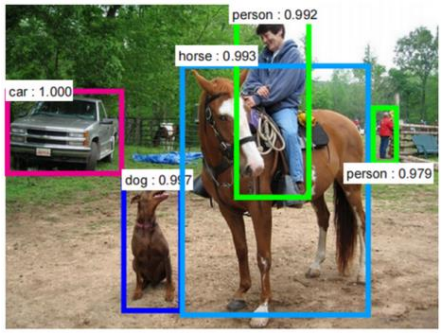


- Nagyon sok feladatnál a jellemzők sorrendjének tényleg nincs szerepe
  - Pl. korábbi feladat: (láz, ízületi_fájdalom, köhögés)→influenza
  - Nyilván pont annyira tanulható, mint (ízületi_fájdalom, köhögés, láz)→influenza
- Vannak azonban feladatok, ahol a jellemzők elrendezése fontos

2. motiváció

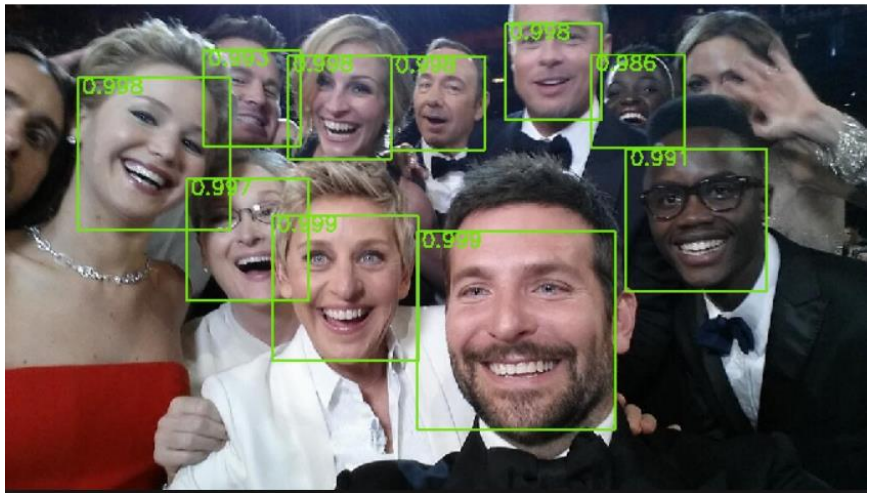
- Vannak azonban feladatok, ahol a jellemzők elrendezése, egymáshoz való viszonya (topológiája) fontos
  - Pl. képi alakfelismerés: Nem ugyanazt látjuk, ha a pixeleket összekeverjük
  - Egy „fully connected” háló viszont ugyanúgy fog tanulni a kétféle képtípuson

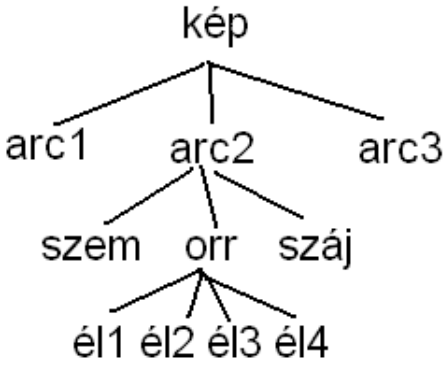
- A pixelek elrendezésében az agyunk számára fontos információ van, amit viszont a fully connected háló nem használ ki!
  - A pixelek együtt alkotnak képet
  - A közeli pixelek kapcsolata jellemzően szorosabb (együtt alkotnak objektumokat)



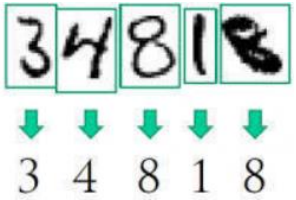
3. motiváció

- A képek jellegzetesen hierarchikusan épülnek föl
  - Egyszerűbb, lokális építőelemektől a nagyobb, összetettebb objektumok felé haladva





  - A korai neuronhálós alakfelismerési kísérletek az ilyen összetett képek felismerését eleve reménytelennek tartották
    - Csak egyszerűbb feladatokkal próbálkoztak, pl. karakterfelismerés
  - Most már összetett, „valódi” képekkel is kísérleteznek
    - Ezek felismerésében a konvolúciós háló jelentős javulásokat hozott



4. motiváció

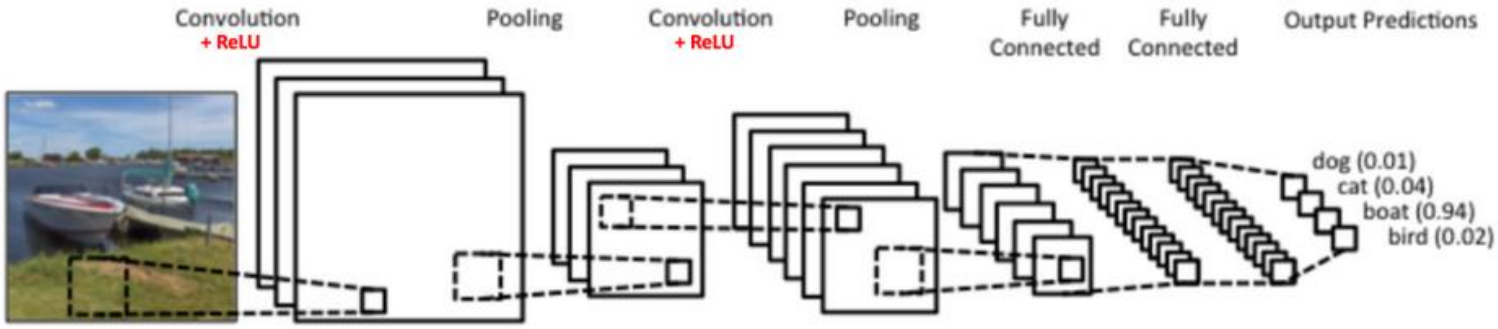
- Vannak arra utaló kutatások, hogy az emberi agy is hierarchikusan dolgozza fel a képeket
  - A vizuális agykéregben bizonyos neuronok a képen látható egyszerű objektumok, pl. különböző irányban álló élek látványára „tüzelnek”
  - Vagy erre utal pl. az ún. „Thatcher illúzió” – a fejjel lefele álló kép esetén az agyunk megállapítja, hogy a száj, orr, szem rendben van és a helyén van (együtt kiadnak egy arcot), de nem veszi észre a finom részletekben rejlő hibát

Konvolúciós neuronhálók

- A fenti motivációkhoz igazodva a konvolúciós neuronháló
  - Az inputot hierarchikusan dolgozza fel
    - ebből adódóan szükségszerűen mély lesz, azaz sok rétegből fog állni
  - Az egy egyes rétegek inputja lokális, azaz a kép kis részleteire koncentrál
    - De fölfele haladva egyre nagyobb részeket fed le
  - A magasabb rétegek egyre nagyobb képrészeket fednek le, de ezzel egyidejűleg egyre rosszabb felbontásúak (a finom részletek egyre kevésbé számítanak)
  - Kevésbé lesz érzékeny az objektumok pontos pozíciójára (ebben segít majd a konvolúció)
- A konvolúciós mély hálók fő alkalmazási területe a alakfelismerés, de más olyan feladatban is szóba jöhet, ahol az input kép, vagy ahhoz hasonlóan hierarchikus felépítésű (pl. beszédfelismerésben is használják)

A konvolúciós neuronhálók építőelemei

- Egy tipikus konvolúciós háló felépítése
  - https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/



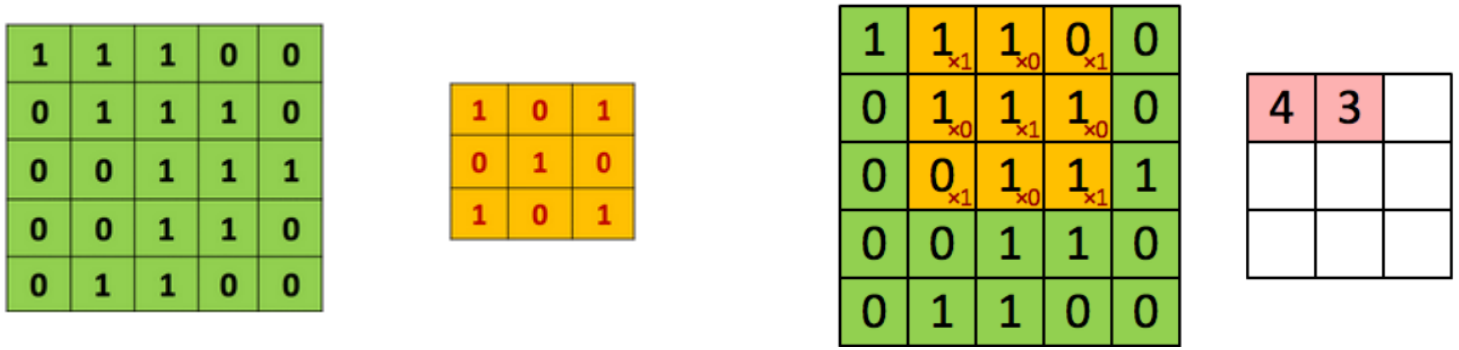
  - Konvolúciós neuronok (szűrők)
  - Pooling (egyesítés) művelet
  - A fentiek többször ismételve
  - A háló tetején általában néhány „fully connected” réteggel, és persze a szokásos kimenő réteggel
  - A tanítás szokványosan a backpropagation algoritmussal történik (ebben a pooling lépés okoz némi bonyodalmat)

A konvolúciós lépés

- A konvolúciós neuronok pontosan ugyanolyan neuronok, mint amilyenekről eddig beszéltük. A fő különbségek:
  - Lokalitás: a neuronok a bemenő képnek egyszerre csak egy kis részletét dolgozzák fel
  - Konvolúció: ugyanazt a neuront a kép több különböző pozícióján is kiértékeljük (mivel ezek a kiértékelések ugyanazokat a súlyokat használják, szokás ezt „weight sharing” tulajdonságként is emlegetni)
- A neuronok által végzett művelet nagyon hasonlít a képfeldolgozásból ismert szűrők hatására, ezért gyakran szűrőnek (filter) is hívjuk ezeket a neuronokat
  - De most a kimeneten valamilyen nemlineáris aktivációs függvényt is alkalmazni fogunk
  - Valamint az együtthatókat (súlyokat) nem kézzel állítjuk be, hanem tanuljuk

A konvolúciós lépés (2)

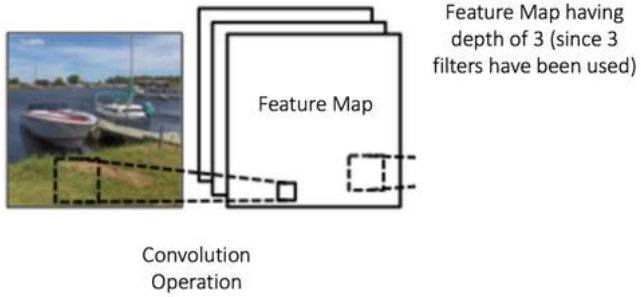
- Konvolúciós neuron működésének szemléltetése:
  - Input kép, szűrőmátrix (neuron súlymátrixa), és az eredmény:



  - A kiértékelést minden pozíción elvégezhetjük, ekkor a fenti példához hasonlóan egy mátrixot kapunk
    - De a bejárt tartományt esetleg korlátozhatjuk is
  - Stride: a szűrő lépésköze. Ha pl. 2, az azt jelenti, hogy minden második pozíciót átugrunk
  - Zero padding: A szűrő szélső pixelekre való illesztéséhez szükség lehet a kép kitoldására (jellemzően 0 értékekkel) – ez tervezési kérdés

A konvolúciós lépés (3)

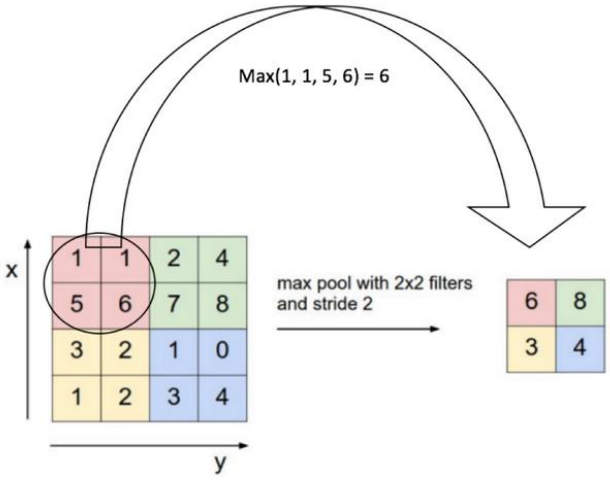
- A konvolúciós szűrőktől avagy neuronoktól azt várjuk, hogy valamilyen többé-kevésbé absztrakt fogalmat tanuljanak meg (pl. egy él detektálása, vagy egy orr detektálása)
  - Ehhez általában egyetlen neuron nem elég, tehát párhuzamosan több neuront tanítunk ugyanazon az input pozíción. Pl. három neuron esetén 3 kimeneti mátrixot kapunk, ez lesz kimenet „mélysége (depth)”



  - A kimenetre újabb rétegeket fogunk tanítani, tehát úgy tekintjük, hogy az adott réteg jellemzőket nyert ki a háló többi rétege számára
    - Ezért hívják a kimenetet az ábrán „feature map”-nek

Az egyesítési (pooling) lépés

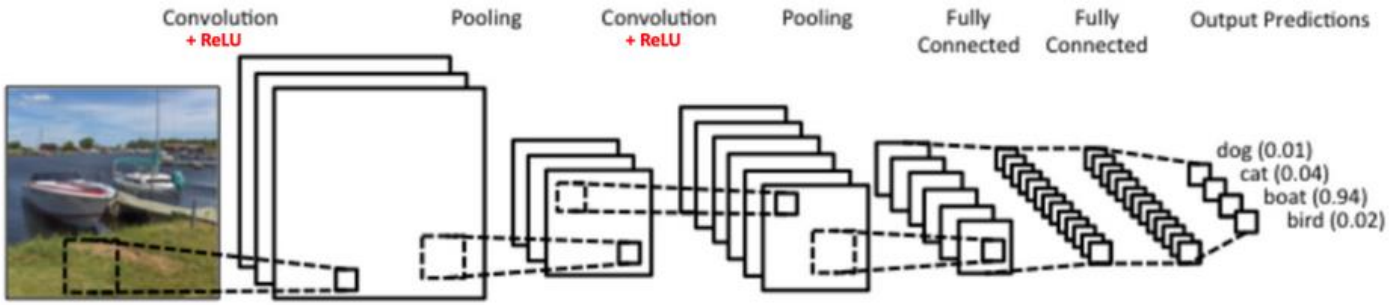
- Magasabbra menve egyre kevésbé akarjuk a finom részleteket megőrizni
  - Pl. ha az „orr-detektáló” jelzett, hogy talált egy orrot, akkor annak a pontos pozíciójára már nem lesz szükségünk magasabb rétegekben
- Ennek megfelelő műveletet végez a pooling:
  - Egy kis környezetben a kimenő értékeket egyesíti egyetlen értékké
    - Pl. a maximumot véve
- Előnyei:
  - Eltolás-invariancia a pooling régión belül (bárhol jelzett a háló, a max. ugyanakkora lesz)



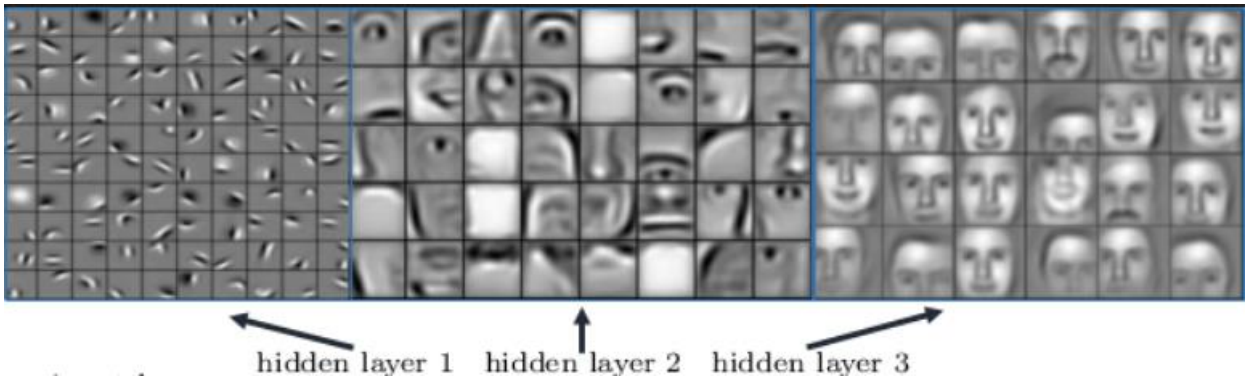
  - Fokozatosan csökken a kimenő mátrix mérete (downsampling) – a paramétercsökkentés mindig hasznos!
  - Följebb lépve ugyanakkora szűrővel egyre nagyobb területet fedünk le egyre rosszabb felbontással – hierarchikus feldolgozás

Hierarchikus feldolgozás

- A fenti konvolúció+pooling lépésekből sokat pakolunk egymásra



- Azt várjuk, hogy az egyre feljebb eső rétegek egyre absztraktabb fogalmakat tanulnak meg, nyernek ki
- Ez többé-kevésbé így is van (példa egy arcfelismerő hálóból):



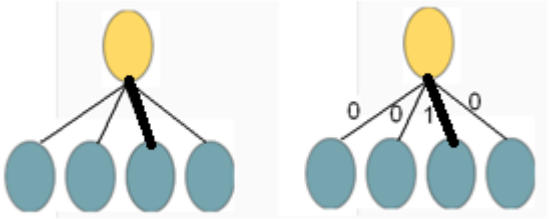
- Végül a végső osztályozást teljesen kapcsolt rétegekkel végezzük el

Összegzés

- A konvolúciós háló fő tulajdonságai és előnyei, hátrányai
  - Konvolúció előnyei: teljes input helyett lokális feldolgozás, több pozíción ugyanazokkal a súlyokkal - kevesebb paraméter, eltolás-invariancia
    - Hátrány: nagyobb műveletigény, mint a teljesen kapcsolt hálóknál
  - Pooling előnyei: kevesebb paraméter, eltolás-invariancia, hierarchikusan egyre kisebb felbontású, de egyre nagyobb térrészt lefedő input
    - hátrány: a rész-objektumok pontos pozíciója elvész, néha pedig fontos lenne
  - Hierarchikus feldolgozás előnye: az összetett képek hierarchikusan épülnek fel, ésszerű az elemzést is így végezni
    - Hátrány: szükségszerűen mély hálót kapunk, a nagyon mély hálók tanítása problematikus lehet

Technikai megjegyzés

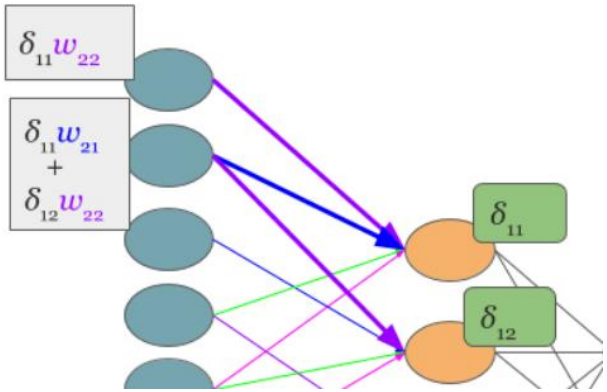
- Konvolúciós hálók esetén a derivált kiszámolása némileg bonyolultabb, mint egy sima előrecsatolt hálóban. A két bonyodalmat okozó lépés a konvolúció, illetve a pooling művelete.
- A derivált átvitele a pooling műveleten viszonylag egyszerű:
  - Pl. Max pooling esetén jelölje a maximumot adó inputot a vastag vonal:



  - Ez esetben a kimentet a 3. input adja, a másik 3 neuron hozzájárulása 0
  - Ezért a derivált értéke a 3. neuron esetén 1 lesz, a többi esetén 0
  - Vagyis a hibát a „győztes” útvonalon terjesztjük csak vissza, a többi útvonalon nem

Technikai megjegyzés (2)

- A derivált visszavezetése a konvolúció műveletén jóval bonyolultabb: Egy input többször is felhasználásra kerül
  - Nem csupán a különböző neuronok által
  - Hanem ugyanazt azt inputot ugyanaz a neuron is többször felhasználja, különböző súlyokkal (ld. ábra, 1D-konvolúcióra)



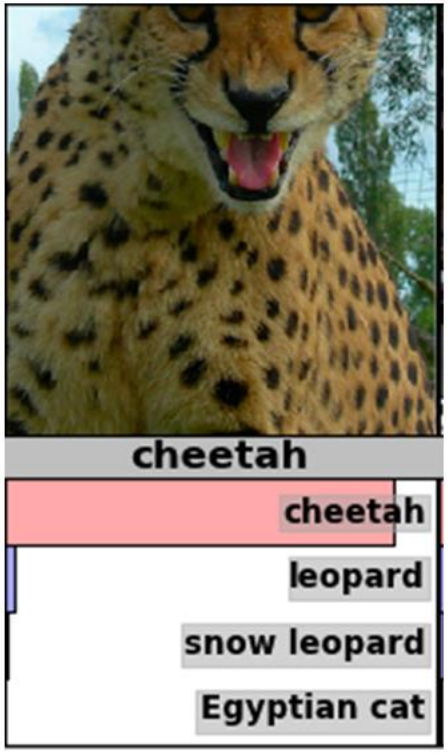
  - Az adott neuron hibája ezen hibaértékek összege lesz
- Hosszadalmas levezetés után azt kapjuk, hogy a hiba-visszaterjesztésnél is egy konvolúciós műveletet kell elvégezni
- További részletek itt:
  - https://grzegorzgwardys.wordpress.com/2016/04/22/8/

2. Megjegyzés

- A háló struktúrájának speciális megválasztásával külső tudást viszünk a rendszerbe. Például a konvolúció+pooling műveletekkel *explicit* módon visszük be azt a tudást, hogy az eltolás nem változtatja meg egy objektum azonosságát.
  - Hátrány: pontos tudással kell rendelkeznünk arra nézve, hogy hogyan kell a modellt megfelelően módosítani
- Ugyanezt a tudást *implicit* módon is be lehetne vinni, például a data augmentation technikával (rengeteg mesterséges példát generálni, ahol ugyanaz az objektum különböző pozíciókon szerepel, de kép címkéje mégis ugyanaz)
  - Hátrány: Jelentősen megnő a tanítópéldák száma és a tanítási idő
- A tanítóadatok mennyiségének növekedésével elvileg egyre csökkeni fog a különbség a speciális hálók (CNN) és az egyszerűbb hálók között

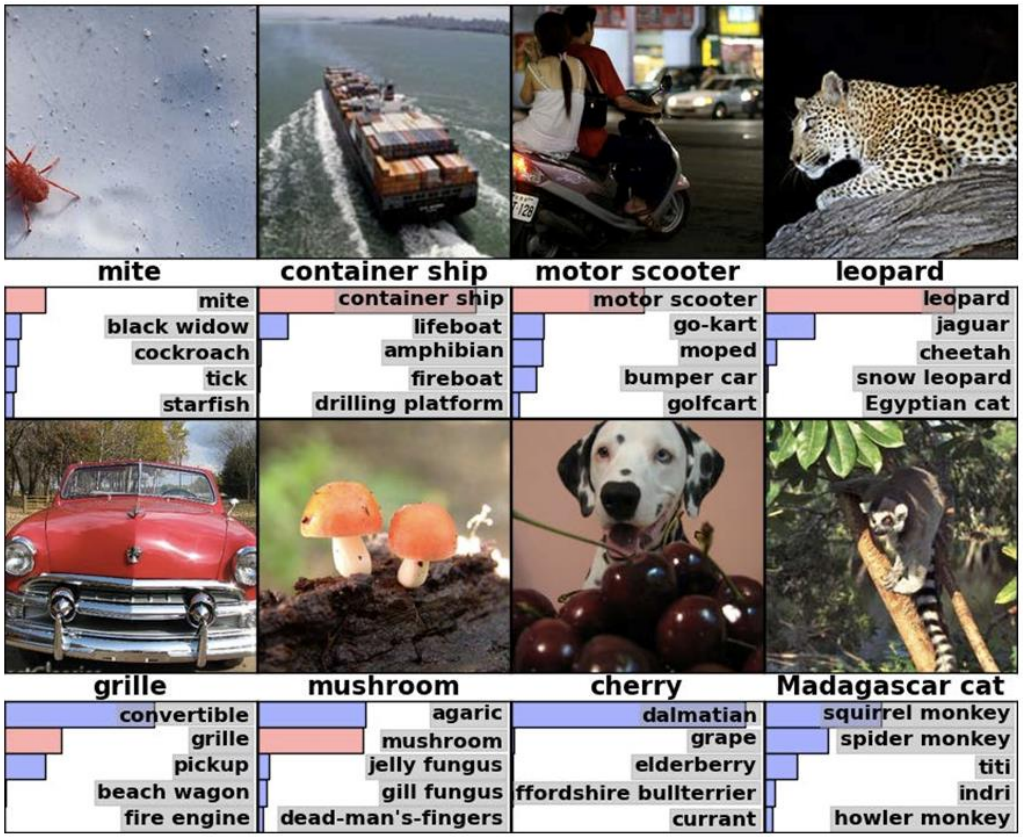
Felismerési példa

- Komplex képfelismerési feladat: többféle objektum is lehet a képen
  - ImageNet adatbázis: 1,2 millió kép, 1000 felismerendő alakzat (címke)
  - Az algoritmus 5-öt tippelhet, helyesnek tekintjük a választ, ha a helyes címke ezek között van



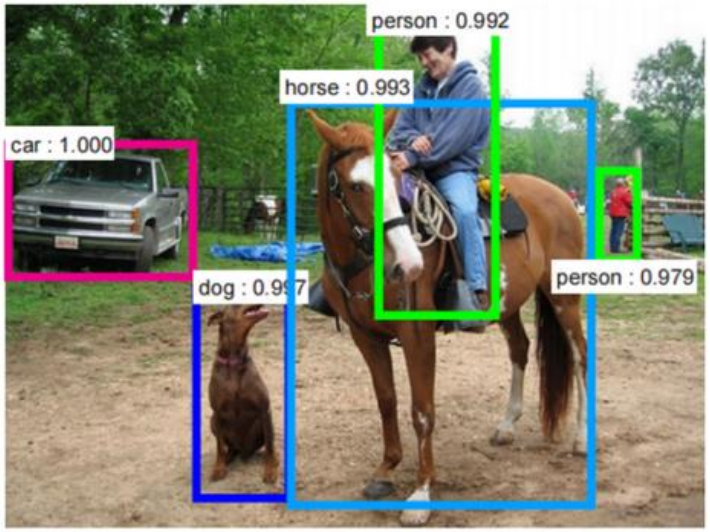
Felismerési példa (2)

- Konvolúciós hálók előtti legjobb eredmény: 26% hiba
- Első konv. háló (2012): 16% -os hiba
- jelenleg a legjobb eredmény: a hiba <5%



Objektum-detektálás

- Nagyon sok gyakorlati alkalmazásban nem csak egyetlen objektum lehet a képen, hanem több is (pl. önvezető autók)
- Ilyenkor a kimenet nem egyetlen címke, hanem:
  - Meg kell találni a képen levő objektum(ok) pozícióját
  - És azonosítani kell őket
- Párhuzamosan kell elvégezni az objektumok osztályozását és detektálását
- Erre is konvolúciós hálókat használnak, de speciális módosításokkal



KÖSZÖNÖM A FIGYELMET!

A tananyag az EFOP-3.5.1-16-2017-00004 pályázat támogatásával készült.

VISSZACSATOLT (REKURRENS) NEURONHÁLÓK

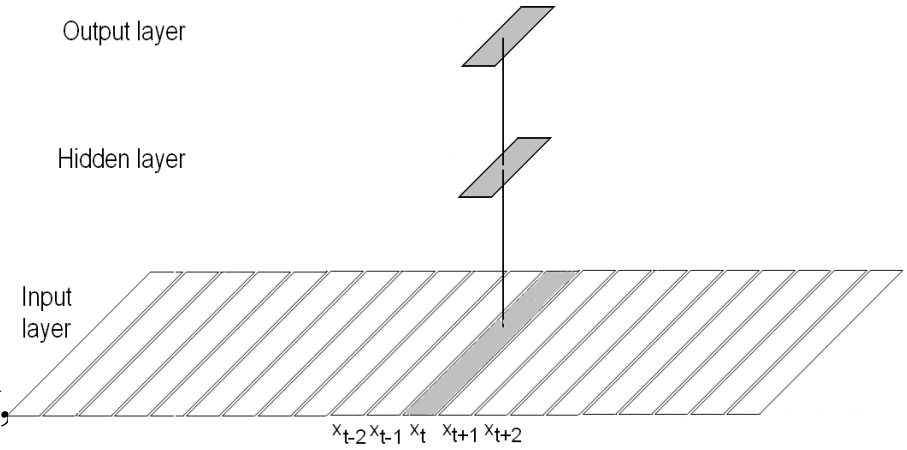
A tananyag az EFOP-3.5.1-16-2017-00004 pályázat támogatásával készült.

Időbeli sorozatok modellezése

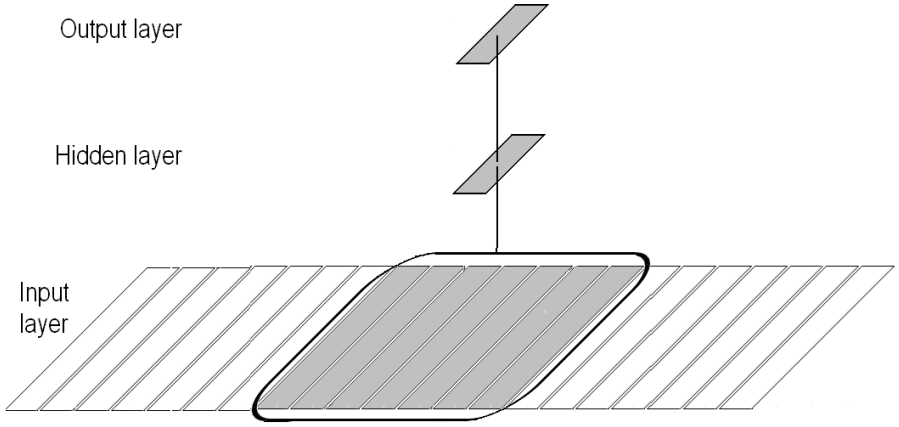
- Eddig feltételeztük, hogy az egymás után következő példák függetlenek egymástól
  - Ez a legtöbb alakfelismerési feladat esetén igaz is
    - Pl. képi alakfelismerés
- Vannak azonban feladatok, ahol a minták egymásutánisága fontos plusz információt hordoz
  - Tipikusan időbeli sorozatok modellezésénél fordul elő
  - Pl.: beszédfelismerés, nyelvi feldolgozás, kézírás-felismerés, videók elemzése, árfolyambecslés,…
- Kérdés, hogy hogyan tudjuk úgy módosítani a hálót, hogy a szomszédokat is figyelembe vegye
  - Előrecsatolt háló több szomszédos inputon: Time-delay neural network
  - Visszacsatolt hálózat: recurrent neural network
  - Visszacsatolt háló hosszú távú memóriával: Long short-term memory network

Szomszédos inputok figyelembe vétele

- Az eredeti (egyetlen input vektort feldolgozó) hálónk így nézett ki (a vonal teljes kapcsolatot jelez):

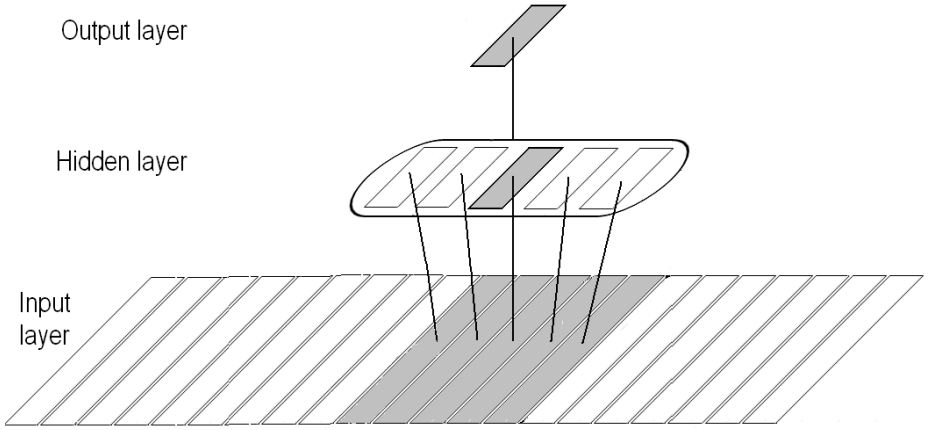


- Ezt egyszerűen módosíthatjuk úgy, hogy az input több szomszédos input vektorból álljon:
  - Előny: semmi módosítást nem igényel sem a hálózatban, sem a tanításában
  - Hátrány 1: az input mérete nagyon megnő
  - Hátrány 2: továbbra is véges környezetet néz



Time-Delay Neural Network

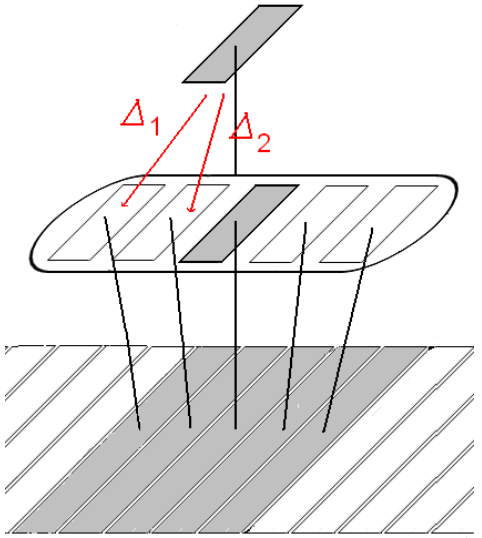
- Több szomszédos inputot is fel akarunk dolgozni
  - De a különböző input vektorokon ugyanazt a feldolgozást végezzük el (legalábbis alacsony szinten)
  - A feldolgozás eredményét csak magasabb szinten egyesítjük
- A rejtett réteg 5 blokkja ugyanazokat a súlyokat használja →"Weight sharing"



- Előny: a feldolgozott input vektorok számának növelésével a rejtett réteg neuronjainak száma nem nő
- Ha a rejtett réteg kicsi, akkor a fölső réteg inputszáma sem nő gyorsan
- Természetesen az alsó és felső feldolgozó rész is állhat több rétegből is

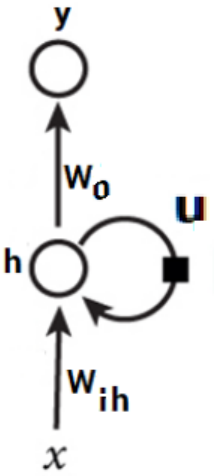
Time-Delay Neural Network 2

- A TDNN előrecsatolt háló
  - A backpropagation tanítás ugyanúgy használható
  - A „weight sharing” okoz némi technikai nehézséget
- Kiértékelés („forward pass”):
  - Ugyanúgy, mint eddig, csak ugyanazt a súlymátrixot többször is felhasználjuk
- Tanítás („backward pass”):
  - a különböző útvonalak mentén kapott delta értékek ugyanazokhoz a súlyokhoz tartoznak, ezért össze kell adni őket, és a súlyokat ezzel módosítani
- A TDNN közeli rokona a konvolúciós hálónak (ld. később)



Visszacsatolt (rekurrens) háló (RNN)

- Valódi visszacsatolást viszünk a hálózatba
- Azaz egy adott pillanatban az input nem pusztán az input vektorból áll, hanem az eggyel korábbi kimenetet is tartalmazza
  - De a kimentet helyett inkább a rejtett réteget szokás visszacsatolni
  - Ezzel memóriával látjuk el a hálózatot, „emlékezni fog” az eggyel korábbi rejtett állapotára is

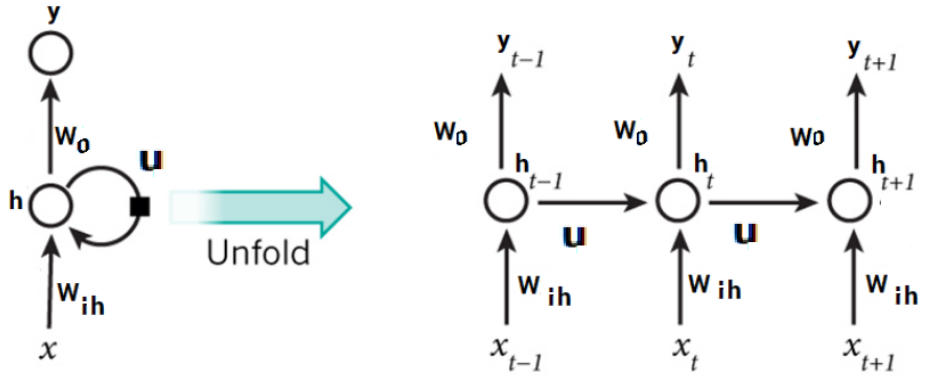


$$\boldsymbol{h}_t = \sigma(\boldsymbol{W}_{ih} \cdot \boldsymbol{x}_t + \boldsymbol{U} \cdot \boldsymbol{h}_{t-1} + \boldsymbol{b})$$
$$\hat{\boldsymbol{y}}_t = \sigma(\boldsymbol{W}_o \cdot \boldsymbol{h}_t)$$

$\boldsymbol{x}_t$	Input (feature) vector at time t
$\hat{\boldsymbol{y}}_t$	Network output vector at time t
$\boldsymbol{h}_t$	Network internal (hidden) states vector at time t
$\boldsymbol{W}_{ih}$	Weight matrix from input to hidden
$\boldsymbol{W}_o$	Weight matrix from hidden to output
$\boldsymbol{U}$	Weight matrix from hidden to hidden
$\boldsymbol{b}$	Bias parameter vector

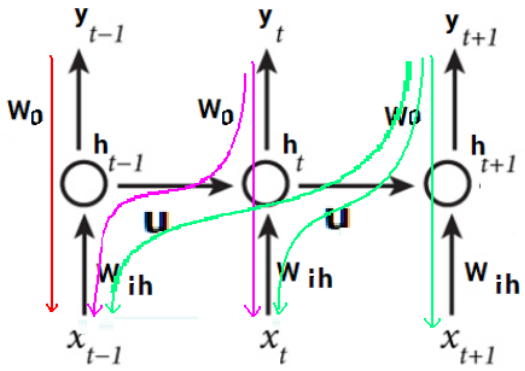
Backpropagation through time

- Hogyan lehet az RNN-t kiértékelni?
  - Az időben előre, azaz balról jobbra, vektorról vektorra haladva
  - Nem tudunk átugrani vektorokat
  - A legelső lépésnél valahogy inicializálni kell h_{t-1}-et
- Hogyan lehet az RNN-t betanítani?
  - Elvileg minden lépésben kell az előző h_{t-1} → végtelen rekurzió
  - A gyakorlatban azonban minden tanító adat véges
    - Vagy mesterségesen is elvághatjuk…
  - Így a hálózatot időben „kiteríthetjük” (unfolding)
  - És taníthatjuk back-propagation-nel



Backpropagation through time

- Mik a „backpropagation through time” nehézségei?
  - A különböző pozíción levő „másolatokhoz” ugyanazok a súlyok tartoznak!
    - A Δ-kat össze kell gyűjteni, ugyanúgy mint „weight sharing” esetén láttuk



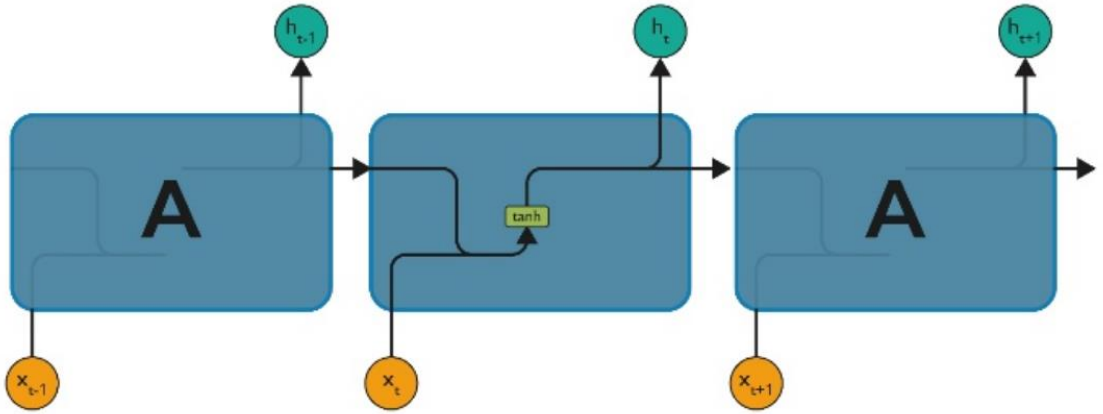
  - Hosszú visszacsatolási útvonalak vannak
    - Elvileg az adott pillanatban adott kimenetet az összes korábbi bemenet befolyásolja
    - Gyakorlatilag azonban a hosszú útvonalak mentén a gradiensek elveszhetnek vagy „felrobbanhatnak”
    - Az RNN tanítása során instabilitási problémák jelentkezhetnek
    - Ha sikerül betanítani, akkor sem tud igazán hosszú távú összefüggéseket megtanulni
    - Tkp. ugyanaz a probléma, mint mély hálók tanításánál!

Long short-term memory (LSTM) háló

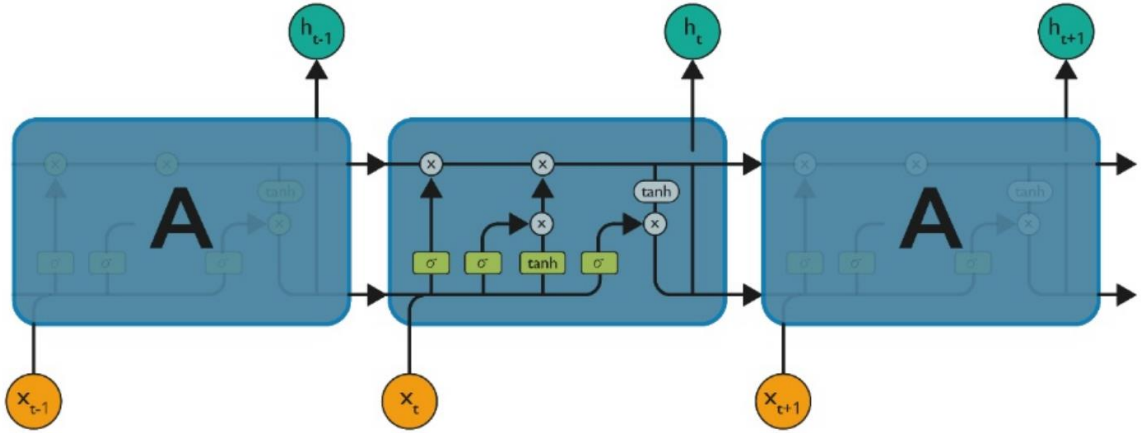
- Tanulhatóvá szeretnénk tenni, hogy az egyes visszacsatolt neuronok mely korábbi inputokat felejtsenek el, és melyekre emlékezzenek
  - Ezzel a hosszú távú emlékezés problémáját is megoldanánk
- Egy külön belső állapotot hozunk létre, amely memóriaként fog működni, és kevésbé lesz érzékeny a backpropagation lépésekre
- Az információ több párhuzamos útvonalon fog áramlani
  - Külön „kapu” fogja szabályozni a memória törlését („forget gate”)
  - Azt, hogy az adott inputból mire kell emlékezni („input gate”)
  - És hogy hogyan álljon össze a kimenet („output gate”)
- Az így kapott modell lesz az LSTM neuron, ilyenekre fogjuk lecserélni a korábbi visszacsatolt neuronokat

RNN vs. LSTM neuron

- RNN (tanh aktivációval):

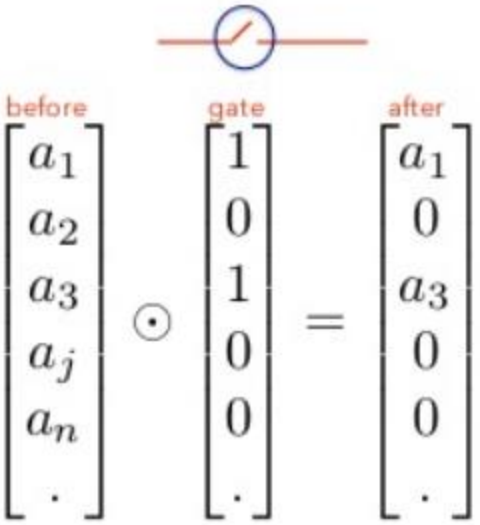


- LSTM:



A kapuk működése

- A kapuk súlyai komponensenként szorozzák meg a bemenet értékeit
- A kapuk súlyait 0 és 1 közt tartjuk szigmoid függvénnyel
- Az optimális súlyértékeket tanulással határozzuk meg
- Példa 0 és 1 kapu-súlyokkal:

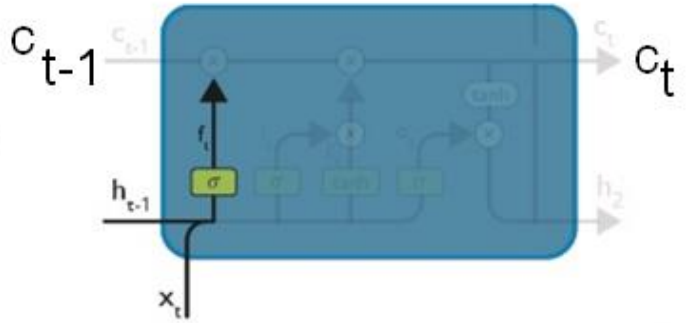


$$\begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_j \\ a_n \\ . \end{bmatrix} \odot \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ . \end{bmatrix} = \begin{bmatrix} a_1 \\ 0 \\ a_3 \\ 0 \\ 0 \\ . \end{bmatrix}$$

LSTM neuron

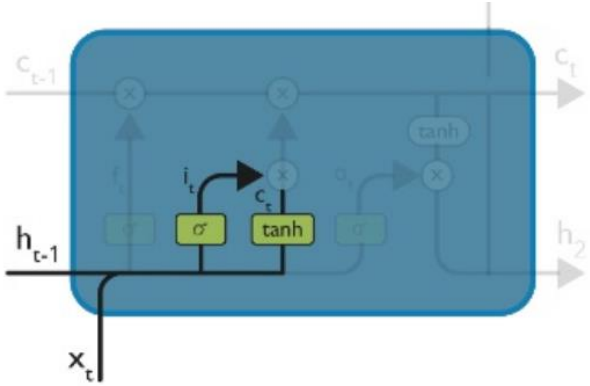
- Forget gate: mely komponenseket kell elfelejteni a C memóriából

$w_f = Weight$
$h_{t-1} = Output\ from\ the\ previous\ time\ stamp$
$x_t = New\ input$
$b_f = Bias$



$$f_t = \sigma(w_f[h_{t-1}, x_t] + b_f)$$

- Input gate: mely input komponenseket kell eltárolni C-be



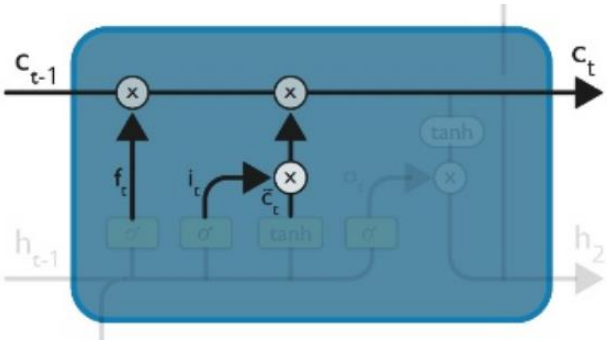
$$i_t = \sigma(w_i[h_{t-1}, x_t] + b_i)$$

$$\tilde{c}_t = tanh(w_c[h_{t-1}, x_t] + b_c)$$

In the next step, we'll combine these two to update the state.

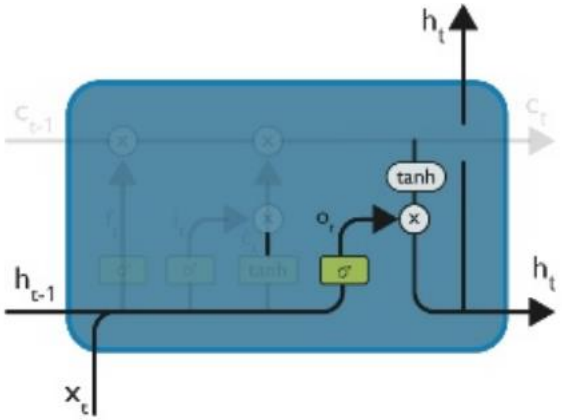
LSTM neuron

- A cell state („memória”) frissítése a forget gate és az input gate segítségével:



$$c_t = f_t * c_{t-1} + i_t^* \tilde{c}_t$$

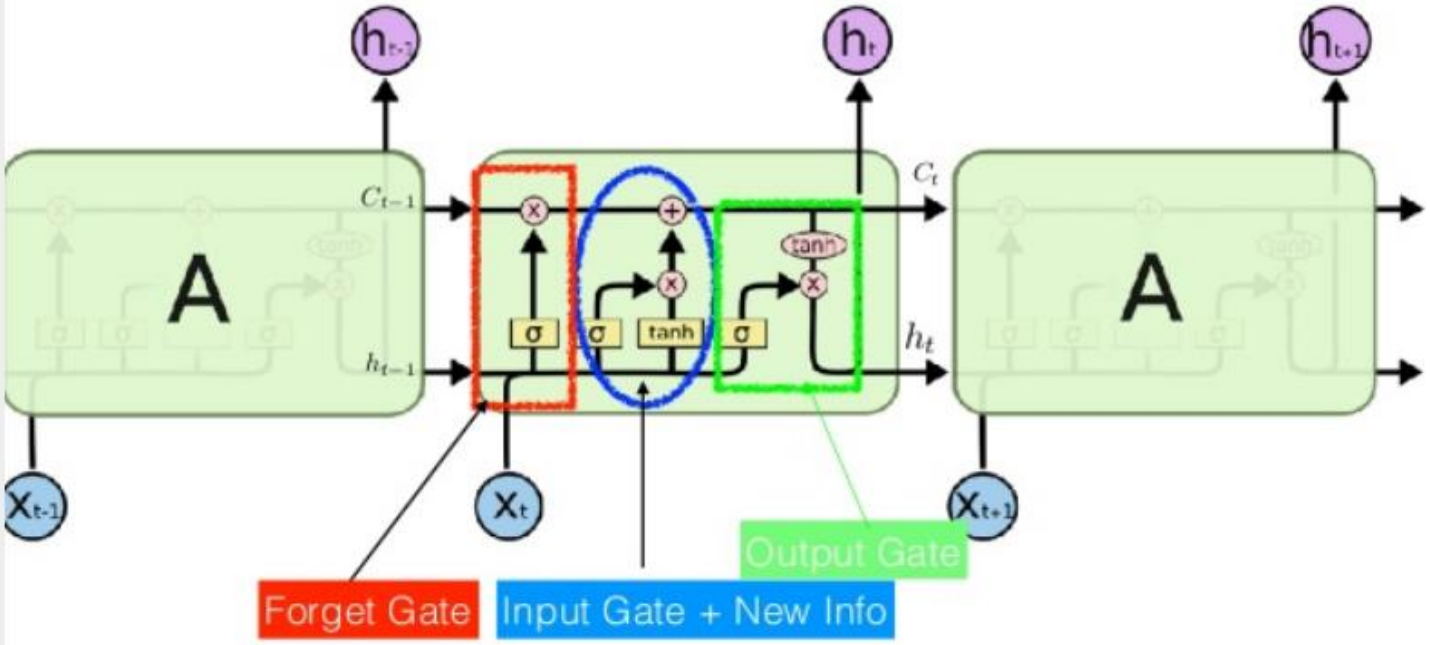
- A rejtett állapot kimeneti értékének kiszámolása:



$$o_t = \sigma(w_o[h_{t-1}, x_t] + b_o)$$

$$h_t = o_t * \tanh(c_t)$$

LSTM összegezve



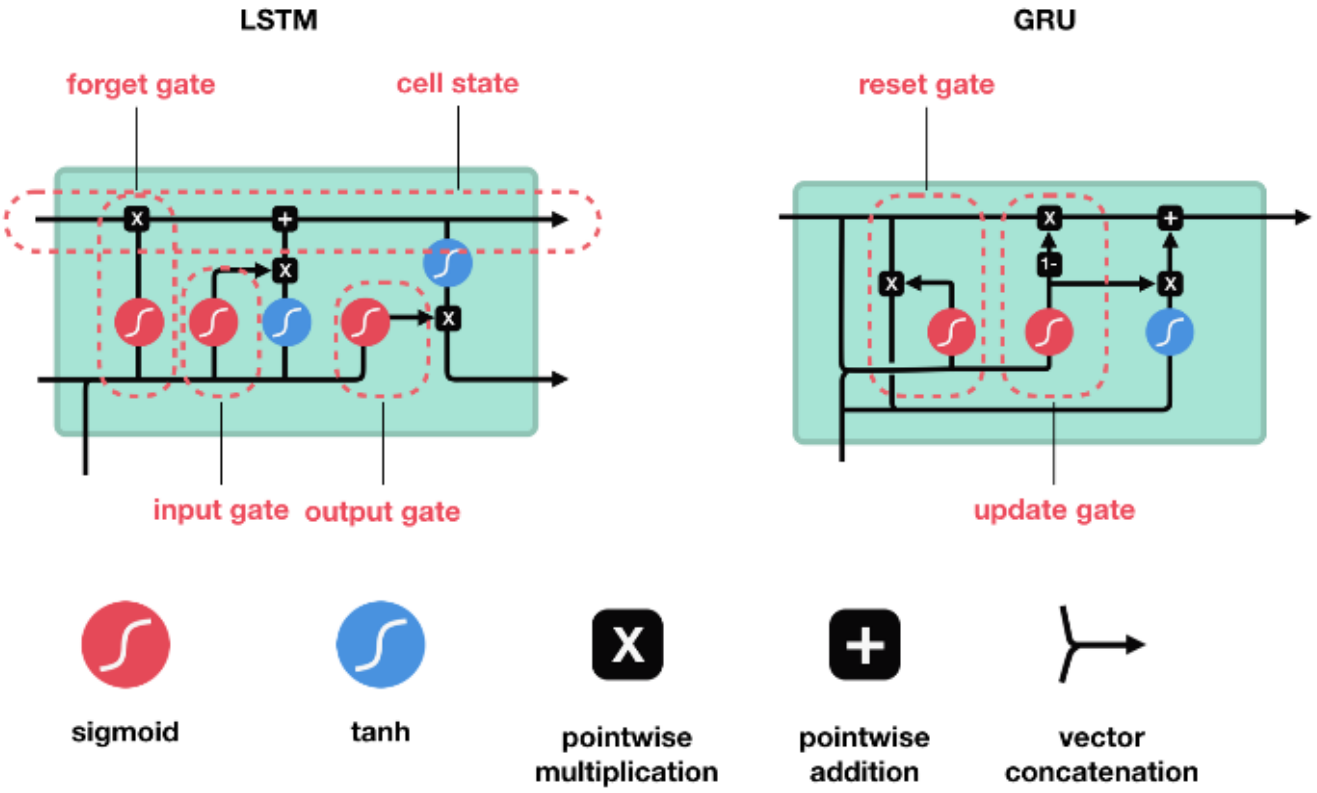
$$f_t = \sigma_g(W_f x_t + U_f h_{t-1} + b_f) \quad \textit{forget gate}$$
$$i_t = \sigma_g(W_i x_t + U_i h_{t-1} + b_i) \quad \textit{input gate}$$
$$o_t = \sigma_g(W_o x_t + U_o h_{t-1} + b_o) \quad \textit{output gate}$$
$$c_t = f_t \circ c_{t-1} + i_t \circ \sigma_c(W_c x_t + U_c h_{t-1} + b_c) \quad \textit{New cell memory}$$
$$h_t = o_t \circ \sigma_h(c_t) \quad \textit{New Hidden}$$

LSTM hálózat

- LSTM cellákból ugyanúgy építhetünk hálózatot, mint a sima visszacsatolt neuronokból
  - A betanítás persze bonyolultabb és lassabb lesz
  - De a legtöbb feladaton tényleg jobb eredményeket ad
- LSTM variánsai:
  - Vannak még több útvonalat tartalmazó változatok
    - LSTM „peephole” kapcsolatokkal
  - Vagy épp ellenkezőleg, egyszerűsített változatok
    - Pl. GRU – gated recurrent unit (ld. következő 2 dia)
  - Jelenleg az egyik legaktívabb kutatási terület

- https://www.slideshare.net/LarryGuo2/chapter-10-170505-l
- https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21

LSTM vs. GRU

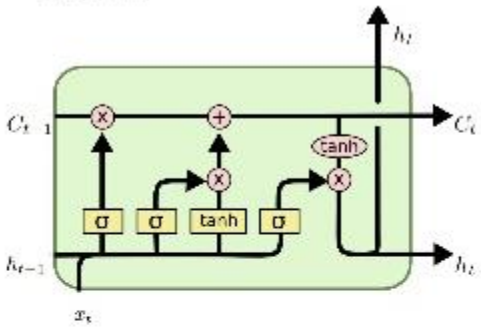


- A GRU kevesebb kaput használ, de a tapasztalatok szerint könnyebben, gyorsabban tanítható, és hasonló pontosságra képes, mint az LSTM

LSTM vs. GRU

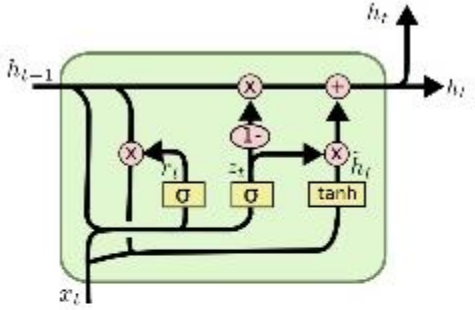
- A kevesebb kapu miatt egyetlen GRU egység kevesebb egyenlettel írható le, mint az LSTM

- LSTM



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] + b_f\right)$$
$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] + b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$
$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$
$$o_t = \sigma\left(W_o [h_{t-1}, x_t] + b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

- GRU



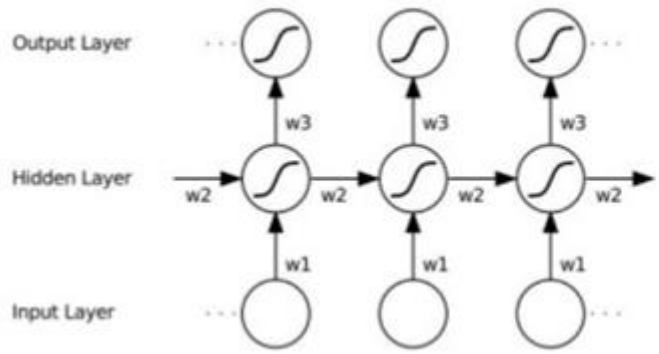
(Biases are omitted.)

$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

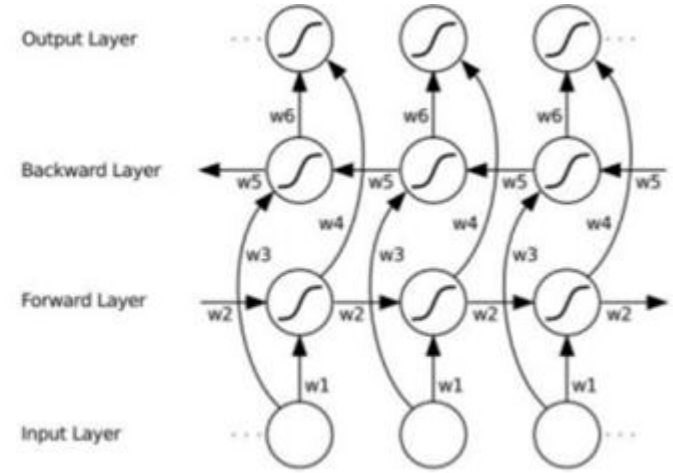
Kétirányú (bidirectional) rekurrens háló

- A jelet mindkét irányban feldolgozzuk
  - Mindkét irányhoz tartozni fog 1-1 rejtett réteg (egymástól függetlenek!)
  - A kimeneti réteg felhasználja mindkét rejtett réteget
    - Előny: az előző és a későbbi kontextust is figyelembe veszi
    - Hátrány: valós időben nem megy

RNN:

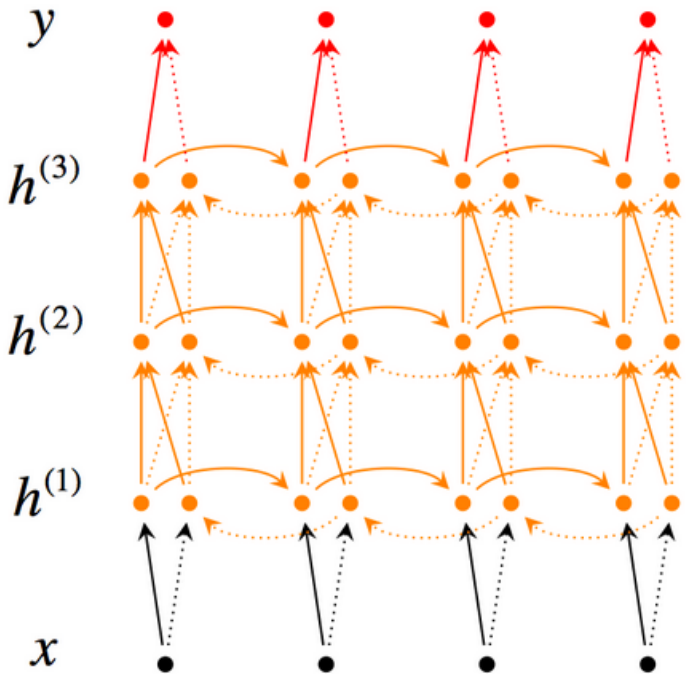


BRNN:



Mély rekurrens hálók

- Természetesen rekurrens hálónál is lehet több réteget egymásra pakolni
  - Akár kétirányút is
  - De kevésbé (illetve kevesebb réteget) szokás, mint előrecsatolt rétegekkel
    - A rekurrens hálónak eleve nagyobb a reprezentációs ereje
    - A tanítása viszont lassabb és bonyodalmasabb



$$\overrightarrow{h}_t^{(i)} = f(\overrightarrow{W}^{(i)} h_t^{(i-1)} + \overrightarrow{V}^{(i)} \overrightarrow{h}_{t-1}^{(i)} + \overrightarrow{b}^{(i)})$$

$$\overleftarrow{h}_t^{(i)} = f(\overleftarrow{W}^{(i)} h_t^{(i-1)} + \overleftarrow{V}^{(i)} \overleftarrow{h}_{t+1}^{(i)} + \overleftarrow{b}^{(i)})$$

$$y_t = g(U[\overrightarrow{h}_t^{(L)}; \overleftarrow{h}_t^{(L)}] + c)$$

KÖSZÖNÖM A FIGYELMET!

A tananyag az EFOP-3.5.1-16-2017-00004 pályázat támogatásával készült.

OBJEKTUM-DETEKTÁLÁS

A tananyag az EFOP-3.5.1-16-2017-00004 pályázat támogatásával készült.

SZÉCHENYI 2020
MAGYARORSZÁG KORMÁNYA
Európai Unió
Európai Szociális Alap
BEFEKTETÉS A JÖVŐBE

Osztályozás+lokalizálás

- Képfeldolgozási alkalmazásról lesz szó, mint a konvolúciós neuronhálóknál
- Ott feltettük, hogy minden képhez egyetlen címke tartozik
  - Azaz a képen egyetlen objektum látható, és az az egész képet betölti
  - Ezért a feladatot osztályozási feladatként tudtuk megfogalmazni
- Ha egyetlen objektumunk van, de nem tölti be a képet, akkor egy osztályozási és egy lokalizálási feladatot kell egyszerre megoldanunk
  - Meg kell találni az objektum címkéjét is pontos pozícióját is

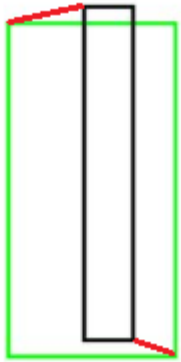
Osztályozás

Osztályozás + lokalizálás

Osztályozás+lokalizálás

- Az objektum pozícióját általában egy befoglaló téglalappal adjuk meg
- De bizonyos esetekben pixel-pontos meghatározásra is szükség lehet
  - Ezt szegmentálási feladatnak hívják, meghatározandó az objektum maszkja

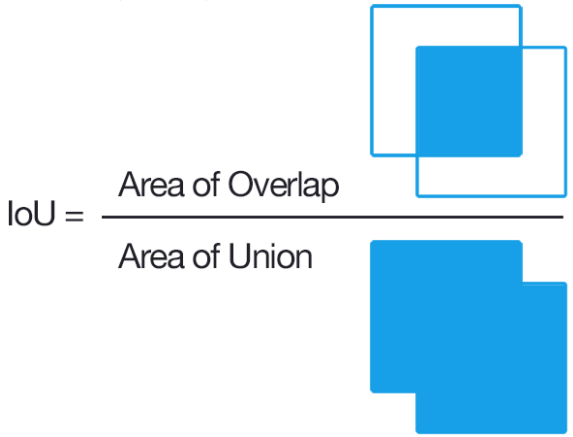
- Az osztályozás+lokalizálás egyidejűleg megoldható egy multi-taszk hálóval
- Az egyik feladat az objektum felismerése (osztályozási feladat)
- A másik feladat a befoglaló téglalap pontos helyének megtalálása
  - Ez megadható pl. a bal felső és a jobb alsó sarok 2-2 koordinátájával
  - A hiba mérhető a valódi és a tippelt téglalap koordinátáinak négyzetes eltérésével (a képen a piros szakaszok összhossza)
  - A hiba egy valós szám, tehát a feladat regresszió

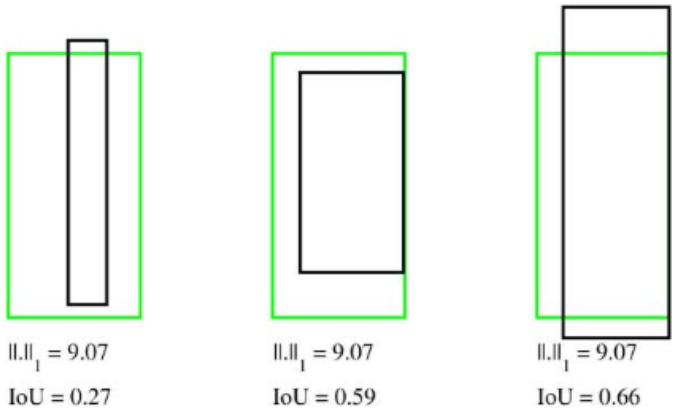


- A multi-taszk feladat össz-hibája az osztályozási és a regressziós feladat hibájának súlyozott összegeként definiálható:
  - Hiba = osztályozási_hiba + $\alpha \cdot$ regressziós_hiba

Osztályozás+lokalizálás kiértékelése

- Az osztálycímke diszkrét érték, vagy eltaláltuk, vagy nem
- A befoglaló téglalap valóditól való eltérése viszont folytonos érték
- A pontossága mérésére az intersection over union (IoU) értéket szokás használni
- Általában akkor tekintjük helyesnek a detektálást, ha a címke helyes, és IoU≥0,5
- Megjegyzés: a koordináták négyzetes hibája és az IoU nem pontosan ugyanazt méri, pl:

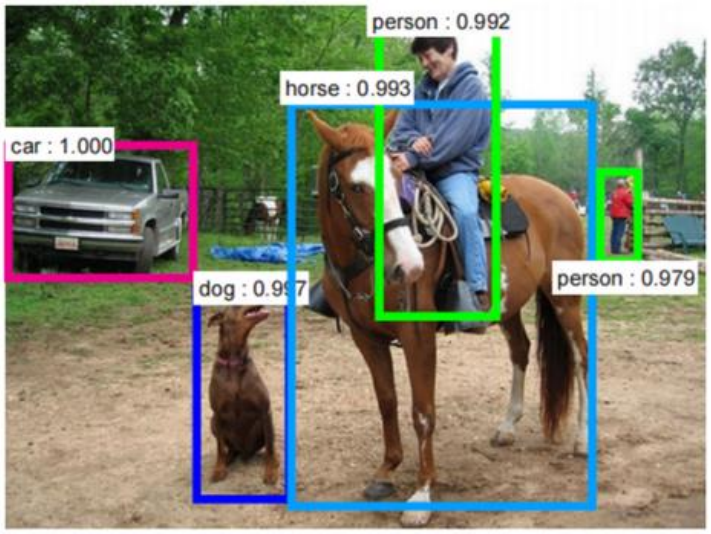




- Az IoU precízebb, pl. nem függ az objektum méretétől
- Viszont a tanításhoz nem jó, mert ha az átfedés 0, akkor mindig 0 lesz, és a derivált is 0

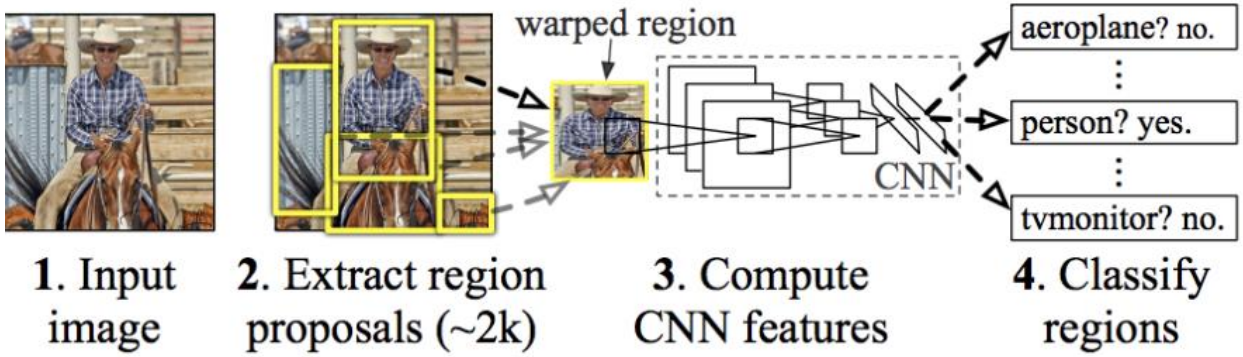
Objektum-detektálás

- Nagyon sok gyakorlati alkalmazásban nem csak egyetlen objektum lehet a képen, hanem több is (pl. önvezető autók)
- Ilyenkor a kimenet nem egyetlen címke, hanem:
  - Meg kell találni a képen levő összes objektum pozícióját
  - És azonosítani kell őket
- Párhuzamosan kell elvégezni az objektumok osztályozását és detektálását
- Probléma: Az összes lehetséges pozíciót és ablakméretet végig kellene próbálni → túl sok lehetőség
- Ráadásul sok alkalmazásban ennek valós időben kellene működnie
- És lehetőleg szerényebb képességű célhardveren, nem szuper képességű szerveren (pl. önvezető autók)



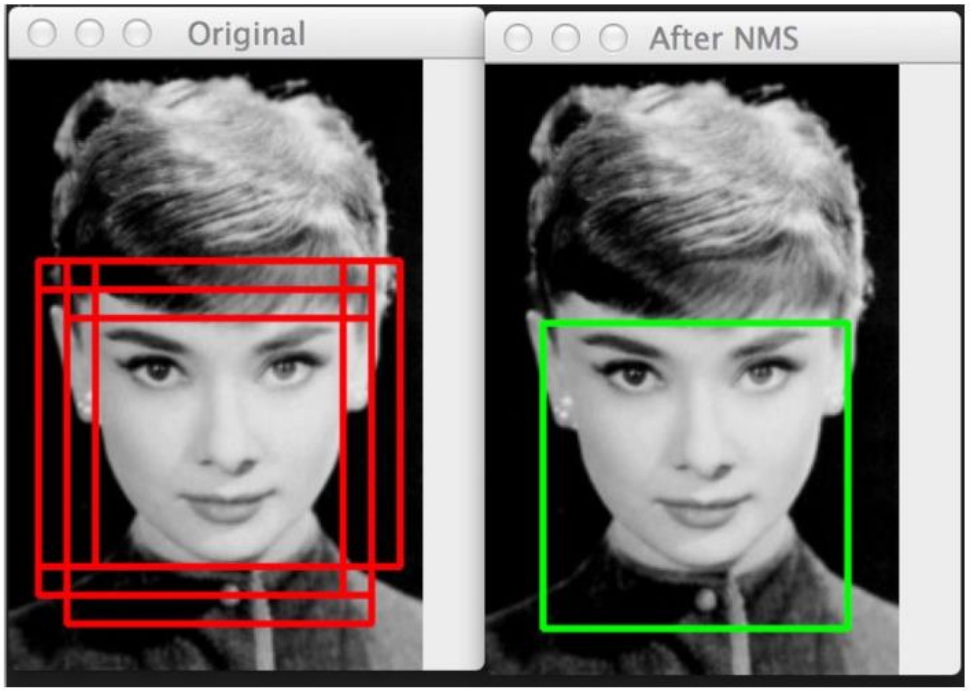
R-CNN

- A régiójavaslaton (region proposal) alapuló módszerek esetén a lehetséges detekciós ablakok (régiók) számát valamilyen algoritmussal leszűkítjük.
- A korai módszereknél a kiértékelésre javasolt régiókat egy külső, neuronhálótól független algoritmus szállítja
- Az R-CNN (2014) esetén ez egy hierarchikus képszegmentáló-klaszterező algoritmus, amely képeként kb. 2000 régiójavaslatot állít elő
- A javasolt régiókat egységes méretűre/arányúra konvertáljuk
- Ezeket egy CNN segítségével feldolgozzuk (jellemzőkinyerés)
- Az osztályozást egy lineáris SVM-mel végezzük
- A régiók utólagos pontosítása is lehetséges (ld. lokalizációs feladat)



Non-Maximum Suppression

- A detektált régiók között nagyon nagy átfedés lehetséges. Utólagosan meg lehet szűrni a régiókat a non-maximum suppression módszerrel. Ennek lényege, hogy a bizonyos küszöbértéknél magasabb arányban átfedő régiók közül csak a legnagyobb osztályozási valószínűséget adót tartjuk meg.



https://www.pyimagesearch.com/wp-content/uploads/2014/10/hog_object_detection_nms.jpg

Fast R-CNN

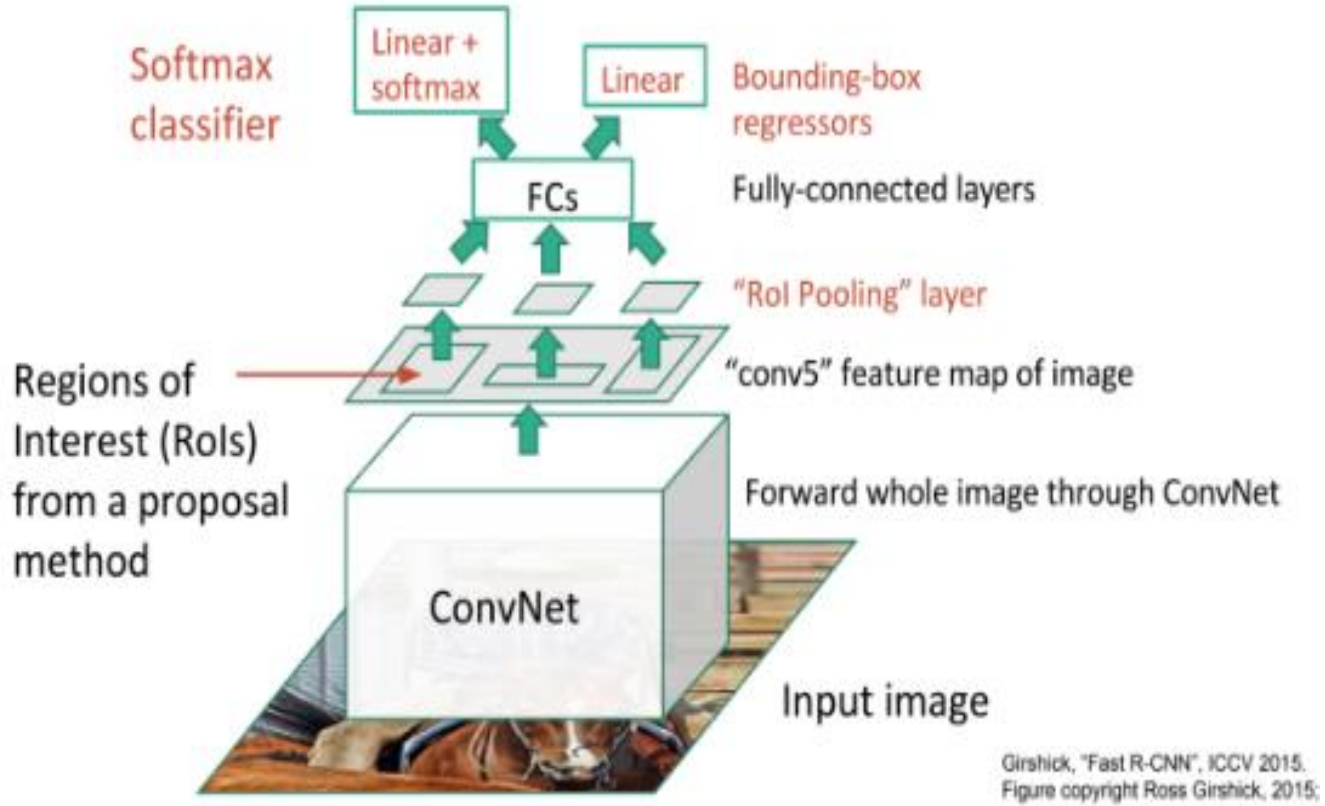
- Az R-CNN volt az egyik legelső algoritmus, amelyik neuronhálót használt objektum-detektálásra, és nyilván sok gyenge pontja van:
- 3 különböző tanulóalgoritmusból áll össze, ezek külön-külön vannak tanítva: CNN (jellemzőkinyerés), SVM (osztályozás), ANN (regresszió)
- Egy negyedik algoritmust használ a régiók megtalálására
- Lassú, mert egyetlen képen 2000-szer kell lefuttatni a CNN jellemzőkinyerést (minden javasolt régión külön-külön)
- A Fast R-CNN modellben két módosítást javasoltak. Az egyik a három tanuló egyesítése egyetlen tanulómodellé. Ehhez csak az SVM-et kellett lecserélni egy fully connected osztályozó rétegre, valamint a három részmodellt egyben tanítani a korábban látott multi-taszk módon
- A másik módosítás célja az volt, hogy ne kelljen a CNN részt minden régióra külön-külön kiszámolni

Region of interest pooling

- Emlékezzünk, hogy a CNN eleve lokalizáltan dolgozza fel a kép különböző részeit.
- Ezért nem a kivágott régiókon futtatjuk le a teljes CNN-t (külön-külön)
- Hanem a CNN-t ráeresztjük a *teljes* képre *egyszer*
- Majd a kapott konvolúciós jellemzőkből vágjuk ki az egyes régiókat leíró részeket
- Mivel a régiók nem egyforma méretűek, valahogy egységes méretűre kell őket hozni az osztályozó és a lokalizáló számára
- Ezt végzi el a „region of interest pooling”
  - A túl nagy régiókat egységnyi részekre bontjuk, és a részekre belül pooling-gal egyesítünk
  - A túl kicsi régiókat felnagyítjuk

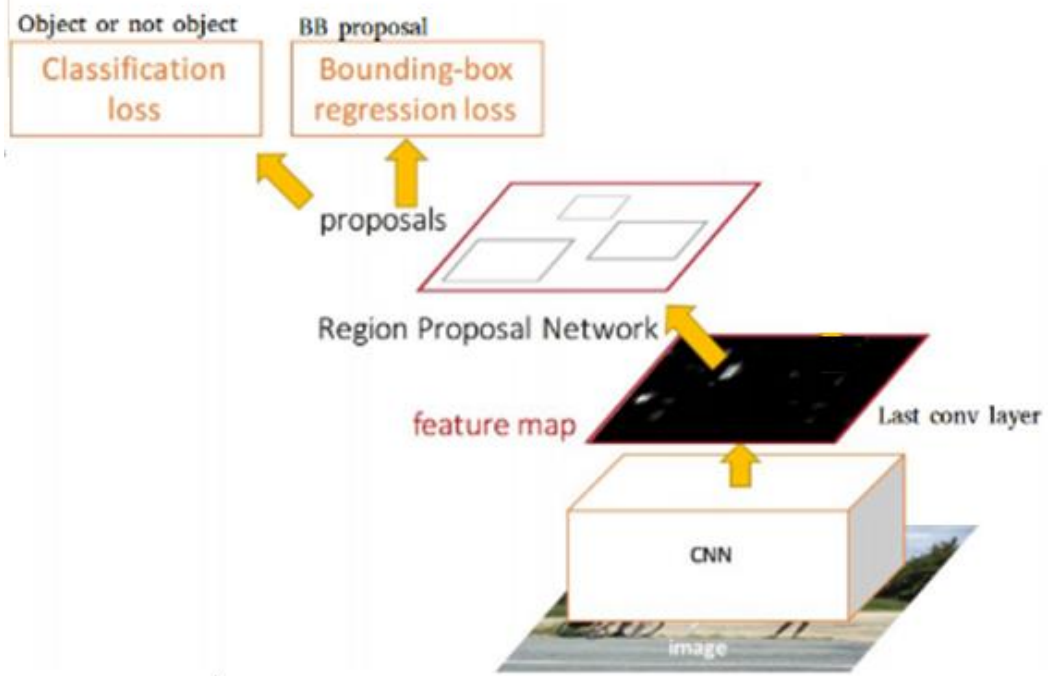
Fast R-CNN

- Összefoglaló ábra:



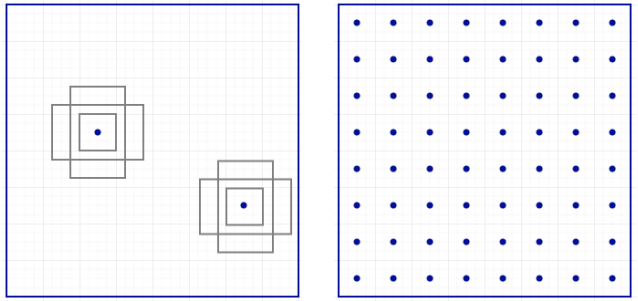
Faster R-CNN

- A fast R-CNN annyival gyorsabb az R-CNN-nél, hogy a futási időt itt már a régiójavasló algoritmus dominálja
- A faster R-CNN (2016) megpróbálja a régiójavaslatokat is egy neuronháló segítségével előállítani (Region Proposal Network)
- Ehhez a konvolúciós rétegek kimenetét használja inputként, nem az eredeti képet
- Az RPN két dolgot becsül:
  - A régió objektum-e
  - Ha igen, mik a pontos koordinátái



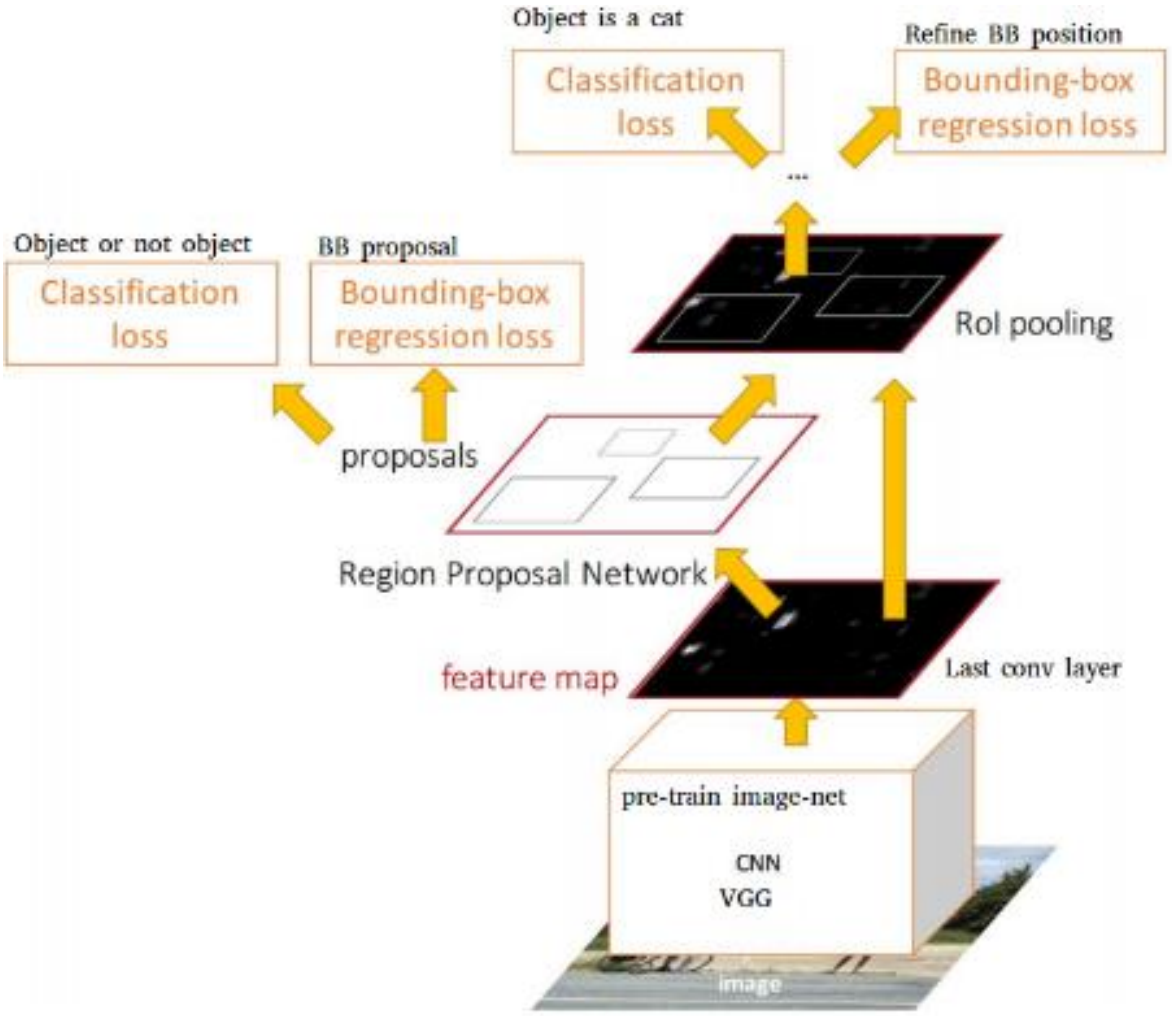
Az RPN működése

- Az RPN nem vizsgál meg minden lehetséges régiót, hanem leszűkíti a keresést bizonyos „anchor box”-okra
  - Az eredeti cikkben csak 9 lehetséges ablak-alakot vizsgál
  - És ezeket 16-16 pixelenként helyezi el
  - Ezzel együtt is tízezres nagyságrendű lehetséges régió marad



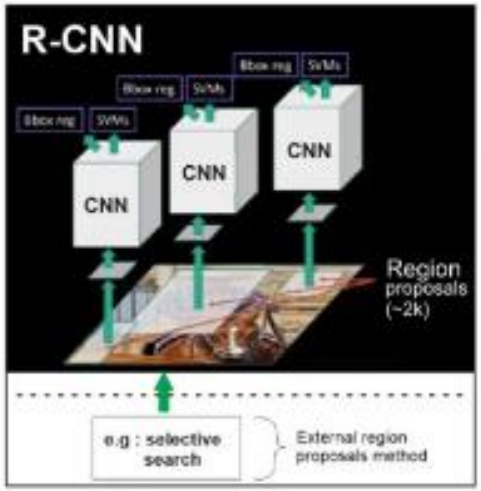
- Az RPN minden egyes anchor box-ra megmondja, hogy
  - Van/nincs benne objektum (2-osztályos osztályozás)
  - Ha van benne objektum, pontosítja a doboz koordinátáit (regresszió)
- Az objektum-tartalmazási valószínűség szerint ki lehet válogatni a legjobb régiókat, illetve non-maximum suppression-nel is szűrhetők
- A megmaradt régiók mennek tovább osztályozásra+regresszióra (a háló főágán)

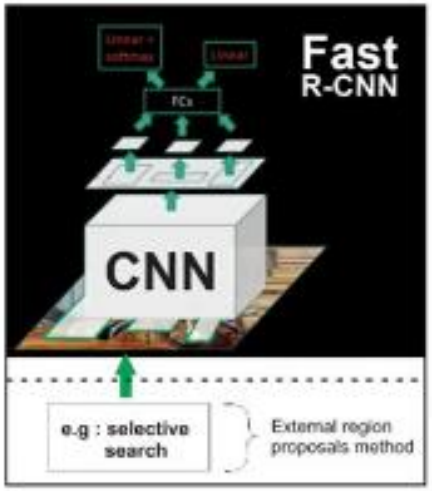
Faster R-CNN - összegzés

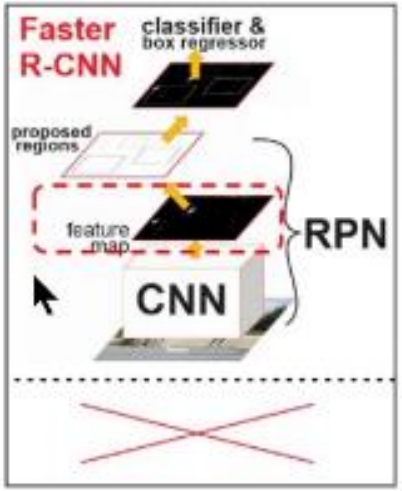


Összehasonlítás

- Az R-CNN különböző változatainak sebessége és pontossága





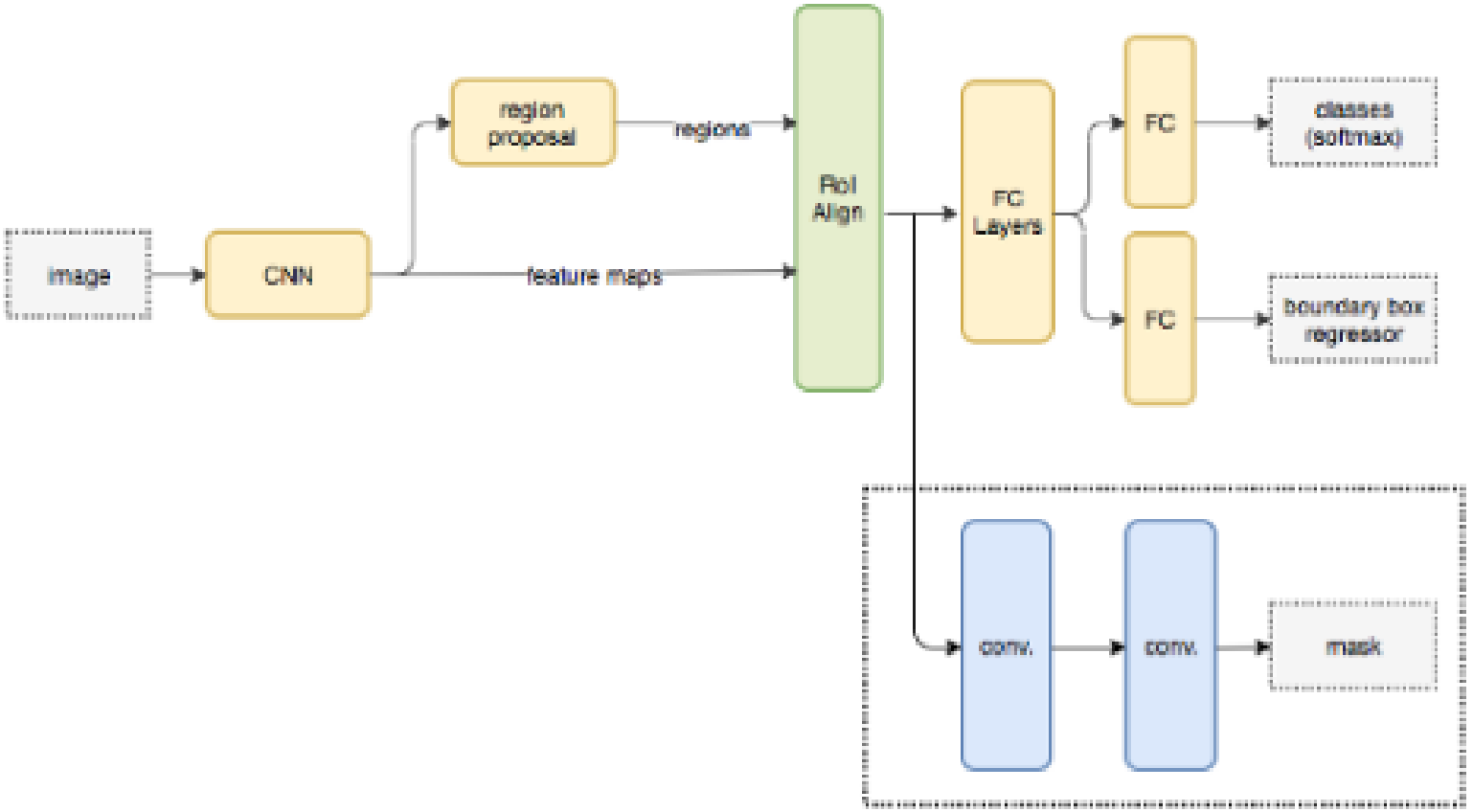


	R-CNN	Fast R-CNN	Faster R-CNN
Test time per image	50 seconds	2 seconds	0.2 seconds
Speed-up	1x	25x	250x
mAP (VOC 2007)	66.0%	66.9%	66.9%

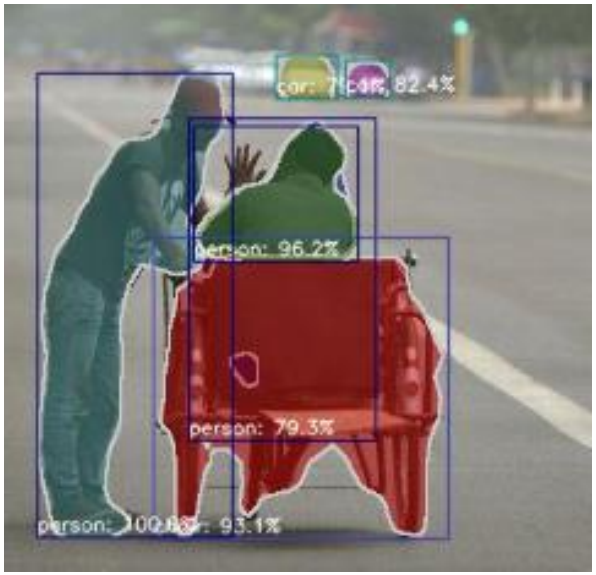
* Standford lecture notes on CNN by Fei Fei Li and Andrej Karpathy

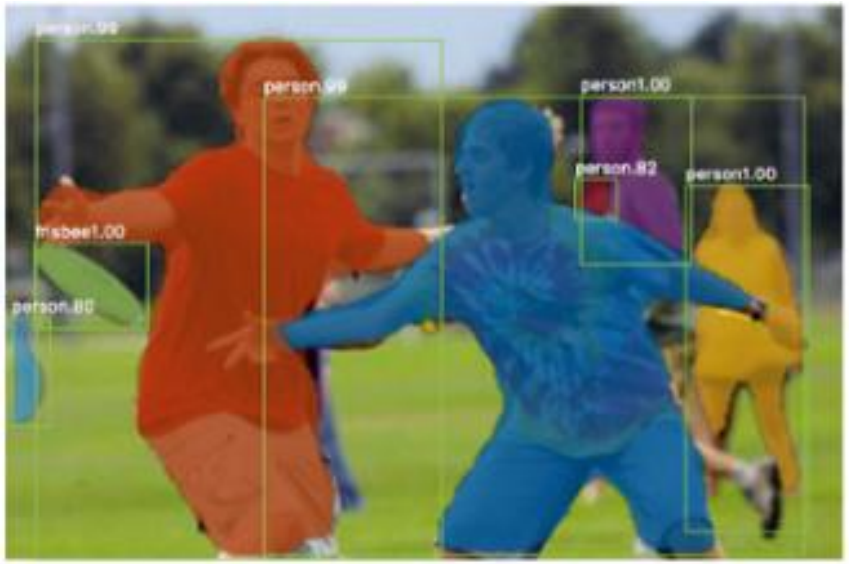
Mask R-CNN

- A faster R-CNN modellt kiegészíthetjük egy újabb ággal, ami egyúttal a maszkokat is megjósolja



Mask R-CNN példa





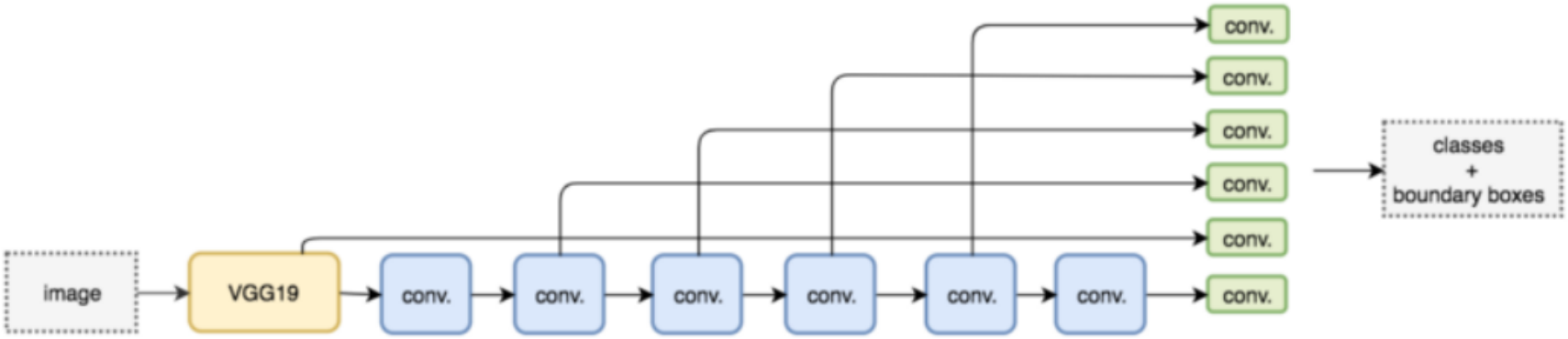
- Alkalmazás: pl. testtartás felismerése (pose recognition)

Single-shot detektorok

- A faster R-CNN esetén az RPN ág és a főág eléggé hasonló feladatot végeznek. Nem lehetne ezeket összevonni?
- A single-shot detektorok kihagyják a régiójavaslatok készítését, az osztályozási+regressziós feladatot közvetlenül, egy lépésben („single-shot”) próbálják megoldani.
- Ezen módszerek közül a két legismertebb a YOLO és az SSD.
- Az R-CNN „anchor”-jaihoz hasonló régiókból indulnak ki.
- Egy konvolúciós háló által adott konvolúciós jellemzőkön dolgoznak.
- További konvolúciós rétegekkel próbálják az osztályozási+ regressziós feladatot egyszerre megoldani

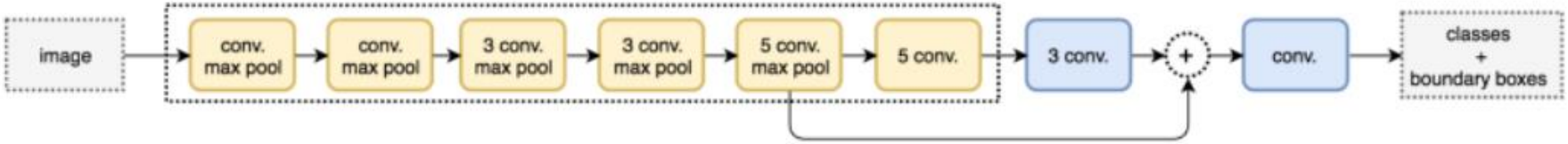
Single-shot detector (SSD)

- A kevés vizsgált ablakméret miatt hajlamosak a túl nagy vagy túl kicsi objektumokat nem észrevenni.
- Az SSD ezért a (különböző felbontású!) konvolúciós rétegek mindegyikének kimenetén elvégzi a detekciót

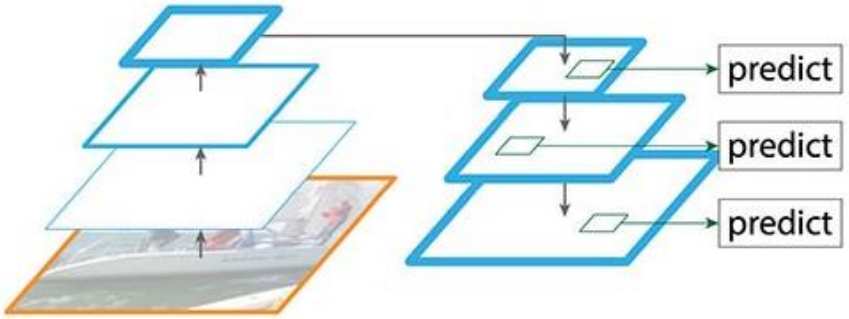


YOLO („you look only once”)

- A YOLO módszer pedig a konvolúciós feature map-ek össze-konkatenálásával állít elő inputot a detekció számára
- Így csak egyszer kell elvégezni a detekciót, de az eredmény pontatlanabb lesz

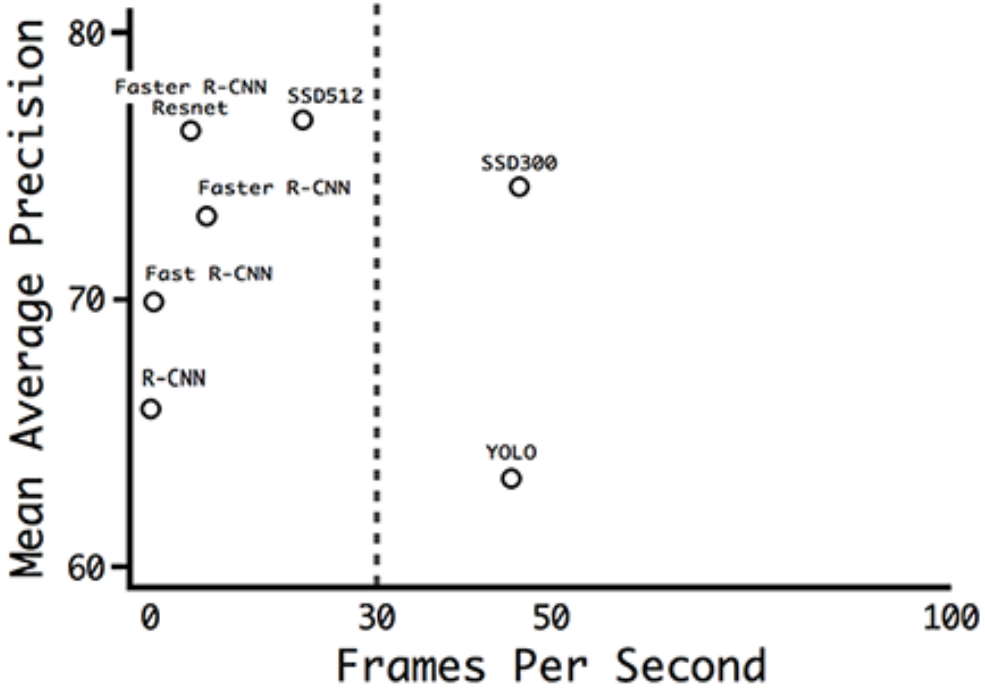


- Megjegyzés: az újabb, „feature-pyramid” alapú módszerek is igyekeznek a konvolúciós rétegek mindegyikét használni, a skála-függetlenség növelése céljából



Sebesség vs. pontosság

- A YOLO gyors, de pontatlan,
- az R-CNN változatai lassabbak, de pontosabbak
- az SSD mindkettőben jó



(SSD512 és SSD300 esetén a szám a képek felbontását jelöli)

KÖSZÖNÖM A FIGYELMET!

A tananyag az EFOP-3.5.1-16-2017-00004 pályázat támogatásával készült.

MEGERŐSÍTÉSES TANULÁS, MÉLY Q-HÁLÓK

A tananyag az EFOP-3.5.1-16-2017-00004 pályázat támogatásával készült.

SZÉCHENYI 2020
MAGYARORSZÁG KORMÁNYA
Európai Unió
Európai Szociális Alap
BEFEKTETÉS A JÖVŐBE

Megerősítéses tanulás

- A megerősítéses tanulás (reinforcement learning) egy speciális tanulási feladattípus
  - kilóg a hagyományos felügyelt/felügyeletlen gépi tanulási feladatok közül
  - Saját módszertana és irodalma van
- A gépnek nem egyetlen döntést, hanem döntések sorozatát kell meghozni
  - Az elemi döntések helyességére nem kap visszajelzést, csak a cselekvéssorozat végén kap jutalmat/büntetést
  - A formális modell bevezethet „jutalmat” az egyes lépésekért, de a lényeg ekkor sem az elemi lépések, hanem a lépéssorozat helyessége
- A fő cél tehát egy stratégia („policy”) tanulása
  - A gépnek próbálkoznia kell, hogy megtalálja a helyes lépéssorozatot
  - Járt út vagy járatlan?
  - Melyik cselekvés jó, milyen környezetben?
  - A múltbeli tudást hogyan tudja felhasználni?
- Alkalmazási példák: sakk, Atari játék, Go, Starcraft …

A megerősítéses tanulás változatai

- **Reflexszerű ágens**

Olyan stratégiát tanul, amely közvetlenül képezi le az állapotokat cselekvésekre, nem modellez hasznosságot. Az egyszerű reflexszerű csak az aktuális állapot alapján dönt, nem veszi figyelembe a korábbi állapotokat.

- **Hasznosságalapú ágens**

Az állapotokra alapozott hasznosságfüggvényt tanul, és az alapján választja ki azokat a cselekvéseit, amelyekkel maximálja az elérhető hasznosság értékét. Tehát állapotokhoz, vagy állapotok sorozatához rendeli a hasznosságot.

- **Q-tanuló**

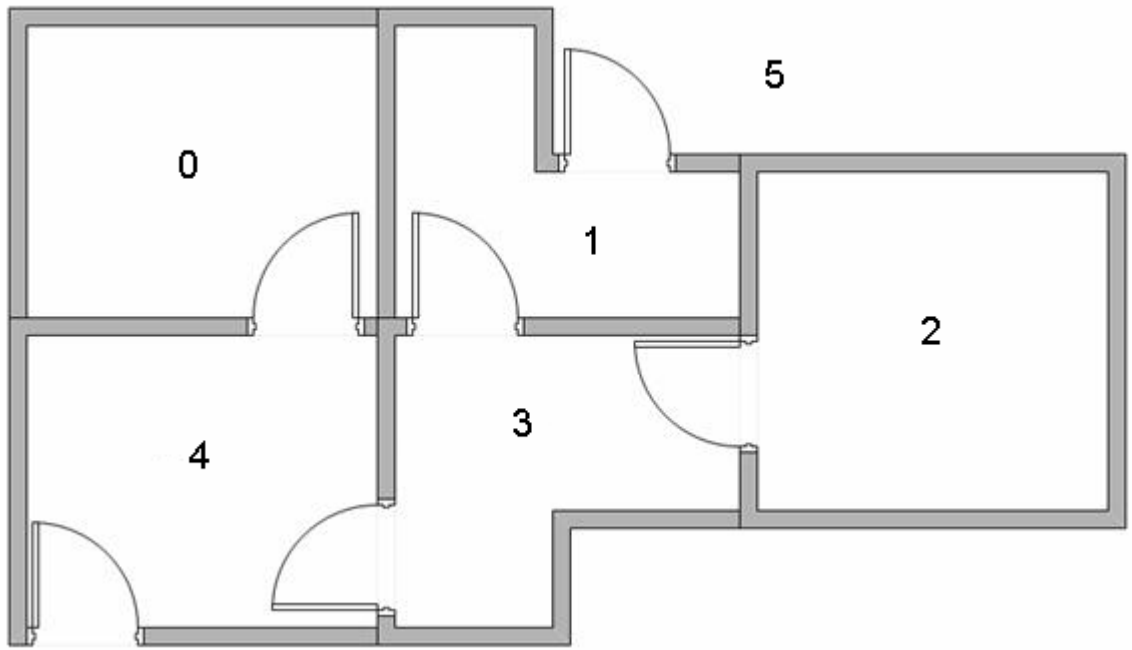
Egy függvényt – Q-függvényt (quality) – tanul, amely a várható hasznosságát becsüli meg egy adott helyzetben egy adott cselekvésnek. Tehát nem állapotokhoz, hanem állapot+cselekvés párokhoz rendel hasznosságot.

Q-tanulás

- Az egyik legegyszerűbb megerősítéses tanulási algoritmus
- Feltesszük, hogy a modellnek állapotai (states) vannak, és minden állapotban vannak lehetséges elemi cselekvések (action)
- Minden állapothoz és cselekvéshez tartozik egy jutalom (reward). Ez jutalmazhatja az elemi lépéseket is, de lehet, hogy csak a kívánt végállapot elérése esetén jutalmaz
- A Q-learning olyan stratégiára törekszik, amely az összes lépésre kapott jutalmak összegét (várható értékét) igyekszik maximalizálni
- A tanulás során összegyűjtött tudást egy ún. Q-táblában tároljuk
- A tanuláshoz nyilván próbálkozni kell lépéssorozatokkal. Ez történhet véletlenszerűen (exploring), vagy az eddig megtanult stratégiát követve (exploiting). Az exploiting/exploring lépések aránya is fontos lehet a módszer konvergenciájához (de ebbe itt nem megyünk bele)

Példa Q-tanulásra

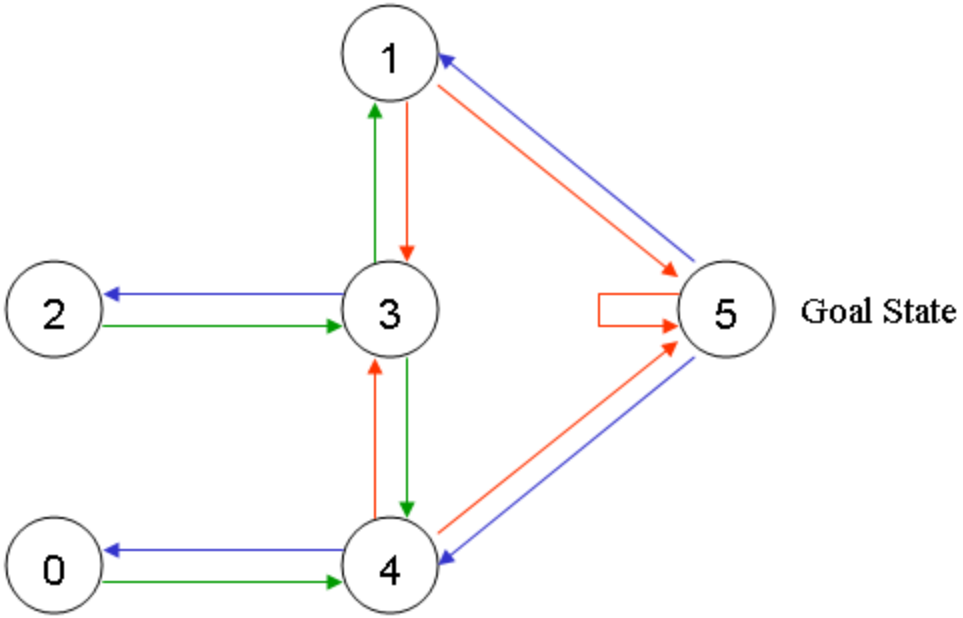
- Feladat: a házból kijutni minél gyorsabban (bármelyik szobából indulva)



Forrás: http://mnemstudio.org/path-finding-q-learning-tutorial.htm

A feladat reprezentálása

- A szobákat állapotoknak, a cselekvések éleknek megfeleltetve a feladatot gráfként rajzolhatjuk fel



- Minden állapot-cselekvés párhoz tartozni fog egy jutalom
- Az 5-ös csúcsba mutató élekhez 100-as jutalmat rendelünk (eléri a célt)
- A többi élhez 0-át (nem tudjuk a cselekvés hasznosságát)
- Az 5-5 hurokél (technikai okból fog kelleni) jutalma szintén 100 lesz

A Q és R-táblák

- A kezdeti R jutalommátrix tehát (minden állapot-cselekvés párra):
  - -1 kódolja a hiányzó éleket

$$R = \begin{array}{c} \\ \text{State} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} \multicolumn{6}{c}{\text{Action}} \\ 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} -1 & -1 & -1 & -1 & 0 & -1 \\ -1 & -1 & -1 & 0 & -1 & 100 \\ -1 & -1 & -1 & 0 & -1 & -1 \\ -1 & 0 & 0 & -1 & 0 & -1 \\ 0 & -1 & -1 & 0 & -1 & 100 \\ -1 & 0 & -1 & -1 & 0 & 100 \end{bmatrix} \end{array}$$

$$Q = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

- Ezen felül létrehozunk egy Q táblát is, amelyben az adott állapotban adott akcióhoz tartozó becsült hasznosságot fogjuk tárolni.
  - Kezdetben Q minden eleme 0, (a tanulás célja Q helyes értékeinek megtalálása)

A Q-tanulási algoritmus

Válasszunk egy alfa értéket (0 és 1 közt), valamint töltsük ki a jutalmak R mátrixát.

Inicializáljuk a Q mátrixot nullákkal.

Minden tanulási epizódra:

Válasszunk kezdőállapotot véletlenszerűen.

Do While nem értünk a célállapotba

Az adott állapotban lehetséges akciók közül válasszunk véletlenszerűen.

Jelölje „next state” azt az állapotot, ahova a választott akció vinne

Keressünk meg Q maximumát a next state állapotban a ott lehetséges akciókra nézve

Frissítsük Q aktuális állapothoz és akcióhoz tartozó értékét:

Q(state, action) = R(state, action)+Alfa*Max[Q(next state, all actions)]

Menjünk át a „next state” állapotba.

End Do

End For

A Q-tanulási algoritmus

- Tanulási epizód: amíg egy véletlenszerű állapotból eljutunk a célállapotba; a sikeres tanuláshoz sok tanulási epizód kell
- Az algoritmus lényege az update szabály:
  - Q(state, action) = R(state, action) + Alfa* Max[Q(next state, all actions)]
- Az alfa „learning rate” hatása
  - 0 körüli érték: az aktuális jutalom a fontosabb, mint a hosszú távon várható jutalom (ezt becsüli Q)
  - 1 közeli érték: az aktuális jutalom helyett inkább a hosszú távú haszonra fókuszálunk
- Betanulás után a kiértékeléshez ugyanezt azalgoritmus használhatjuk, de
  - A Q mátrixot rögzítjük (nincs frissítés)
  - Az új állapotot nem véletlenszerűen választjuk, hanem a maximális Q érték alapján (exploring helyett exploiting-ot végzünk)

Példa tanulási epizódra - 1

- Alfa = 0.8, az 1. szobából kezdünk, R adott, Q üres

$$R = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} \multicolumn{6}{c}{\text{Action}} \\ 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} -1 & -1 & -1 & -1 & 0 & -1 \\ -1 & -1 & -1 & 0 & -1 & 100 \\ -1 & -1 & -1 & 0 & -1 & -1 \\ -1 & 0 & 0 & -1 & 0 & -1 \\ 0 & -1 & -1 & 0 & -1 & 100 \\ -1 & 0 & -1 & -1 & 0 & 100 \end{bmatrix} \end{array} \quad \text{State}$$

$$Q = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

- R alapján az 1. szobából a 3. és az 5. szobába lehet menni. Tegyük fel, hogy a véletlenszerű választás az 5. szobába visz minket. Ekkor
  - Q(1, 5) = R(1, 5) + 0.8 * Max[Q(5, 1), Q(5, 4), Q(5, 5)] = 100 + 0.8 * 0 = 100
- Az új Q:

$$Q = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 100 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

Példa tanulási epizódra - 2

- Alfa=0.8, a 3. szobából kezdünk, majd véletlenül az 1. szobába, onnan pedig az 5. szobába mentünk

$$R = \begin{array}{c} \text{State} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \text{Action} \\ \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} -1 & -1 & -1 & -1 & 0 & -1 \\ -1 & -1 & -1 & 0 & -1 & 100 \\ -1 & -1 & -1 & 0 & -1 & -1 \\ -1 & 0 & 0 & -1 & 0 & -1 \\ 0 & -1 & -1 & 0 & -1 & 100 \\ -1 & 0 & -1 & -1 & 0 & 100 \end{bmatrix} \end{array}$$

$$Q = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 100 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

- Az első, majd a második lépésre:
  - Q(3, 1) = R(3, 1) + 0.8 * Max[Q(1, 3), Q(1, 5)] = 0 + 0.8 * Max(0, 100) = 80
  - Q(1, 5) = R(1, 5) + 0.8 * Max[Q(5,1), Q(5,1), Q(5, 5)] = 100 + 0.8 * 0 = 100
- Az új Q:

$$Q = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 100 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 80 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

Q-tanulási példák

- Jó sok próbálkozás (tanulási epizód) után ezt kapjuk:

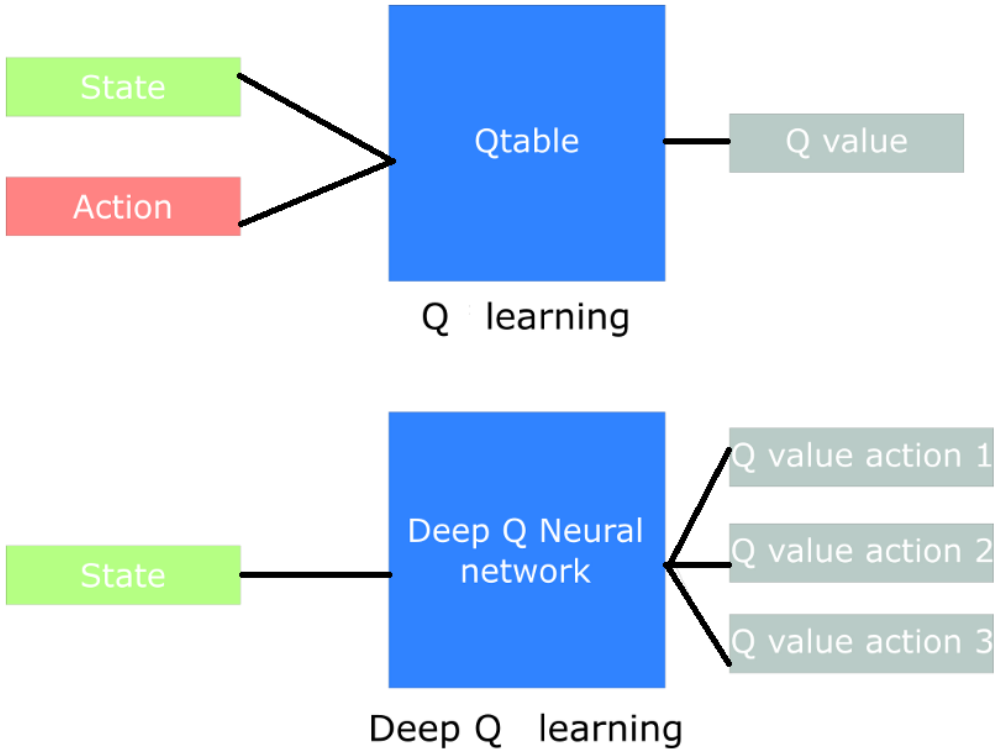
$$Q = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 400 & 0 \\ 0 & 0 & 0 & 320 & 0 & 500 \\ 0 & 0 & 0 & 320 & 0 & 0 \\ 0 & 400 & 256 & 0 & 400 & 0 \\ 320 & 0 & 0 & 320 & 0 & 500 \\ 0 & 400 & 0 & 0 & 400 & 500 \end{bmatrix} \end{array}$$

- Ezt lenormalizálhatjuk, hogy a legnagyobb érték 100 legyen:

$$Q = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 80 & 0 \\ 0 & 0 & 0 & 64 & 0 & 100 \\ 0 & 0 & 0 & 64 & 0 & 0 \\ 0 & 80 & 51 & 0 & 80 & 0 \\ 64 & 0 & 0 & 64 & 0 & 100 \\ 0 & 80 & 0 & 0 & 80 & 100 \end{bmatrix} \end{array}$$

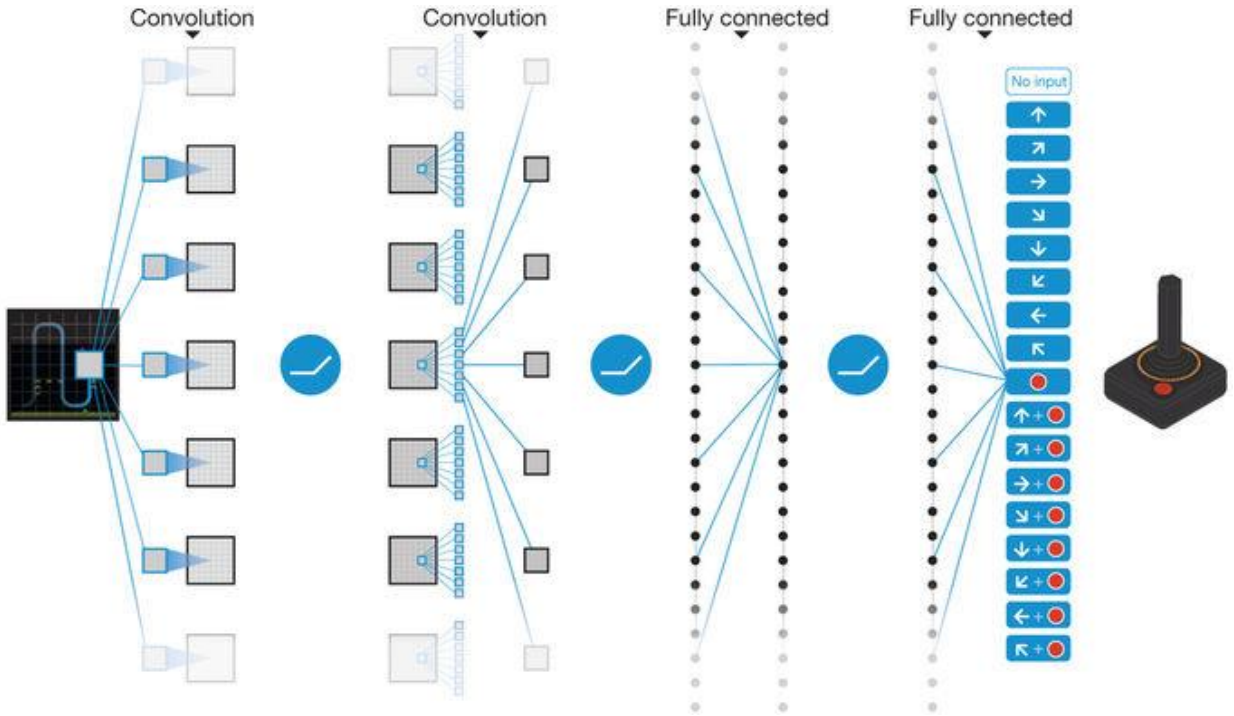
Mély Q-tanulás

- A Q hasznosságot egy egyszerű táblázat helyett egy neuronháló fogja tanulni
- A háló inputja az aktuális állapot, ez egyes kimenő neuronjai pedig a lehetséges akcióknak felelnek meg, azok hasznosságát becsülik meg:



Mély Q-tanulás

- Például ezzel a módszerrel tanították meg a gépet videójátékokat játszani
  - Input: az aktuális képőrnyőkép
  - Output: a lehetséges joystick-mozdulatok hasznossága
  - Háló: egy mély konvolúciós neuronháló



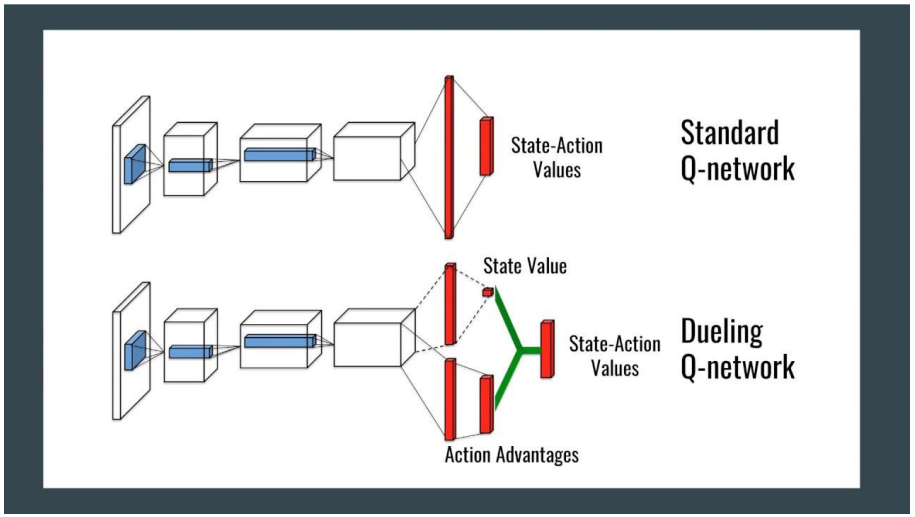
- Forrás: Human-level control through deep reinforcement learning (Nature)

Double Q-learning

- A Deep Q-learning egyik továbbfejlesztése
- Motiváció: a tanulás elején még semmit sem tudunk (a táblázatos modellnél a Q-tábla teljesen üres)
  - A Max[Q(next state, all actions)] függvény a tanulás elején félrevezető értékeket ad, ezért a tanulás nehezen indul be
- Egy helyett 2 neuronhálót tanítunk
  - Az első háló azt tanulja, hogy melyik állapotot érdemes választani (a maximális értéket adó helyett)
  - A második háló pedig az ehhez tartozó Q értéket becsüli
- A tapasztalatok szerint stabilabb tanulást, gyorsabb konvergenciát eredményez

Dueling Q-learning

- A deep Q-learning egy másik továbbfejlesztése
- Szintén 2 hálóra szedi szét az eredeti hálót, de kicsit más koncepció alapján
  - Az egyik háló azt becsüli, hogy mennyire jó egy adott állapotban lenni (state value), a másik pedig, hogy melyik akciót mennyire érdemes választani (state action)
  - Bár külön becsüli ezeket, később egy magasabb szinten összekombinálja

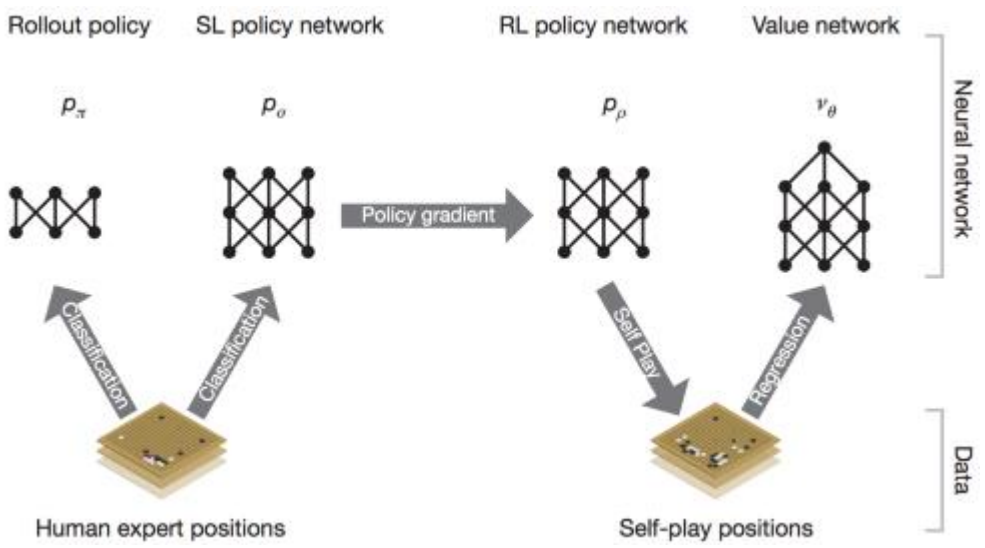


- Miért hatékonyabb ez a szétbontott becslés?
  - Bizonyos állapotok magukban is „jók”, bármit is lépünk (pl. nincs ellenség a közelben), illetve „rosszak” (mindegy, mit lépünk, meghalunk)
  - Az ilyen esetek kezelését könnyebben meg tudja tanulni a dueling háló

Alpha Go

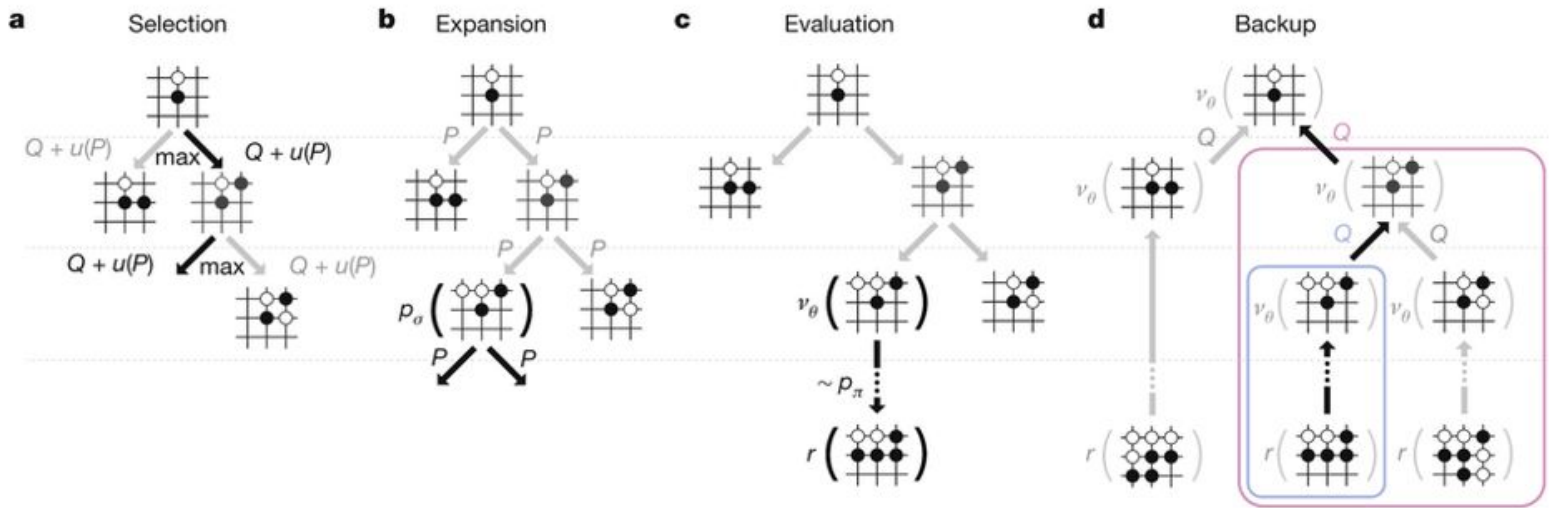
- A Google fejlesztése
- 2015-ben legyőzte a go világbajnokot
- Két mély hálót tartalmaz:
  - policy network: megbecsüli, hogy mit érdemes lépni
  - value network: megbecsüli a lépés hasznosságát
  - (mint az action és value hálózatok az előző dián)

- Először felügyelt módon (SL), 30 millió emberi lépésen betanítottak két hálót (bal oldali ábra)
- Ezután innen indítva, self-play módban tanították be a reinforce-ment learning (RL) policy és value hálóit



Alpha Go

- A hosszú távú stratégia megtalálásához a neuronhálók kombinálva vannak egy fabejáró keresőalgoritmussal
- E célra a Monte Carlo Tree Search (MCTS) nevű algoritmust használták:



a, Each simulation traverses the tree by selecting the edge with maximum action value Q, plus a bonus $u(P)$ that depends on a stored prior probability P for that edge. **b**, The leaf node may be expanded; the new node is processed once by the policy network p_σ and the output probabilities are stored as prior probabilities P for each action. **c**, At the end of a simulation, the leaf node is evaluated in two ways: using the value network v_θ; and by running a rollout to the end of the game with the fast rollout policy p_π, then computing the winner with function r. **d**, Action values Q are updated to track the mean value of all evaluations $r(\cdot)$ and $v_\theta(\cdot)$ in the subtree below that action. (Mastering the game of Go with deep neural networks and tree search)

AlphaGo Zero

- Hasonló a „sima” Alpha Go-hoz, de már nem igényel ember által lejátszott partikon való „előtanítást” a betanuláshoz
- Önmaga ellen játszva tanul
- Módosítások az Alpha Go-hoz képest:
  - Egyszerűsítettek a tábla reprezentációján
  - A policy és value network egyesítve lett (dueling)
  - Konvolúciós háló helyett konvolúciós+reziduális háló
  - Egyszerűsített fakeresés az egyesített háló segítségével (nincs „rollout”)
- Az Alpha Go olyan új értelmes lépési stratégiákat is képes volt felfedezni, amelyeket az emberi játékosok nem is ismertek korábban

KÖSZÖNÖM A FIGYELMET!

A tananyag az EFOP-3.5.1-16-2017-00004 pályázat támogatásával készült.