# Rendszámtábla megtalálása képen

Kis Ádám

March 2018

## 1. Feladatleírás

A feladat továbbra is rendszámtáblák pozíciójának megtalálása egy képen. A feladatot nagyban nehezíti, hogy a képek különböző országban, különböző kamerákkal, különböző felbontással készültek, némely kép fekete-fehér, de a többség színes. Ezeket már az előző féléves beadandómban taglaltam, itt nem térnék ki rá bővebben.

## 2. Példahalmaz növelése

A mélytanulásos módszerek egyik nagy gyengesége jelenleg, hogy a betanításhoz nagy példahalmazra van szükség, azonban a képek felcímkézése nagyon drága, mert nem (jól) autómatizálható. A feladathoz eredetileg 499 db példát kaptam. Összehasonlításképpen, az ImageNet adatbázis, amin a mélytanulási módszereket össze szokták hasonlítani több mint 14 millió képet tartalmaz [3], bár kategóriánként nagy az eltérés, ezertől százezerig. Hosszabb keresgélés után az Ade 20k adathalmazt találtam, ami könnyen hozzáférhető (nem kutatóknak is)[1]. Ez szegmentálásos feladatra lett kitalálva: minden képponthoz meg van adva, hogy milyen tárgyhoz tartozik (ház, autó...). Van egy nagy előnye, amit jelenleg nem használok ki, hogy tárgyak részeit is külön kategorizálja: egy képpont tartozhat egyszerre egy autóhoz és azon belül egy rendszámtáblához, vagy egy házhoz és azon belül egy ablakhoz.

Ezzel együtt a tanító halmazom 1258 darabra bővült, az ellenörző halmazom 536-ra, ez tőbb, mint háromszor annyi, mint előző félévben. Ennek azonban megvan az az ára, hogy ezt a példahalmazt nem kifejezetten erre a feladatra találták ki, ezért a rendszámok legtöbbször nem leolvashatóak és a felvételek szemszöge sem egyezik meg az ellenörző kamerák tipikus szemszögével: a képek fejmagasságból lettek készítve, míg az ellenörző kamerák tipikusan 4-5 méter magasságban vannak (1. ábra). Ez jelentheti azt, hogy a háló nem lesz képes más szemszögből készült képeken felismerni a rendszámtáblákat, de azt is, hogy a két példahalmaz egyesítésével rá lesz kényszerítve, hogy elvonatkoztasson a nézőponttól. Másik probléma, hogy a rendszámok sokszor (bár nem mindig) kivehetetlenek. Ez az előzőhöz hasonlóan segíthet, hogy más jelekből ismerje fel a rendszámtáblát, például az autó pozícióját felhasználva, de mind a két probléma a felé mutat, hogy a betanulás hosszabb és nehezebb lesz, ha egyáltalán (adott erőforrások mellett), lehetséges. Ha a cél az eredeti példahalmazon való jó eredmény elérése, akkor ezeken betanítva a hálónak jobb eredményt kellene produkálnia, mert arra motiválja, hogy ne csak részletek alapján ismerje fel, hanem legalább egy primitív modellt építsen fel az autóról és annak valószínű pozícióiról. A betűk felismerése önmagában hibát hordoz: lehetséges, hogy a betűk egy utcatáblán vannak (bevallom egy ilyennel sem találkoztam, de a lehetőség fennáll), de az eredeti példahalmazban vannak arab rendszámok is, amik tovább komplikálják a betűk felismerését. Egy további hiba, hogy az új adathalmazban nincsenek motorosok, míg az eredetiben igen.

## 3. Eltűnő/felrobbanó gradiens

A mélytanulásban egy általános probléma, hogy a visszaterjesztés során a gradiens egyre kisebb vagy nagyobb lesz. Ennek a végeredménye, hogy az alsőbb rétegekben a gradiens értéke vagy túl nagy lesz, vagy elenyészően kicsi.

(a) Kép az eredeti példahalmazból

(b) ADE 20k dataset egy eleme

1. ábra. Az eredeti példahalmazban a kép tipikusan felülről készült (szögek eltérőek lehetnek), az újban azonban fejmagasságból. Másik probléma, hogy a (b) képen a rendszám alig látszik és nem is leolvasható.


2. ábra. Egy nehéz példa: a rendszám árnyalata alig tér el az környezetétől (színesben egy kicsit jobban), a betűk kézzel lettek felfestve és az arab ábct használja

Tegyük fel, hogy a háló $n$ rétegből áll. És legyen $b < 1$ felső korlátja annak, hogy a hiba hogyan változik a visszaterjesztés során, azaz minden rétegben megszorzódik egy ennél kisebb számmal. Ha ez a $b$ kellően kicsi, akkor az első rétegbeli gradiensekre igaz lesz, hogy $||\nabla w_1|| < ab^n$, ahol $w_1$ az első rétegbeli neuronok bemeneteihez tartozó súlyvektor, $a$ pedig a hiba a legfelső rétegen. Azaz a hibára vett partiális deriváltak elenyészőek lesznek, akár számítási hibán belüliek is, azaz a gradiens (vagy ellentetje) hozzáadása a súlyhoz semmilyen változást nem fog okozni. Ekkor az alsó rétegek nagyon lassan, vagy akár egyáltalán nem tanulnak. Ha ezt magas *learning rate*-tel akarjuk ellensúlyozni, akkor a felsőbb rétegek súlyai túl nagyokat ugrálhatnak, amíg az alsók alig mozognak, . Ennek ellentéte a felrobbanó gradient, amikor $b > 1$ és ez alsó korlátja a visszaterjesztett hibának. Ez általában gyorsan ahhoz vezet, hogy az alsó réteg súlyai közül sok értéke *NaN* lesz és a háló nem ad értelmezhető kimenetet.

Ennek elkerülésére több megoldás létezik. A ReLU aktivációs függvény sikerét sokat pont annak tulajdonítják, hogy elkerüli a *szigmoid szaturáció* problémáját [10]. Ez azonban csak az aktivációs függvényből adódó eltűnő gradiens problémát oldja meg, de az adódhat a súlyok értékéből is. Ezt azzal próbáltam ellensúlyozni, hogy magasabb *learning rate*-et rendeltem az első réteghez, mert annak gradiense feltűnően alacsonyabb volt a többinél.

# 4. Neuron kimenetének normalizálása

LeCunn megmutatta, hogy a tanulás sebességében segít, hogy ha a neuron kimenetének varianciája 1 körül van[11]. Ennek oka eredetileg a szigmoid szaturáció volt, de ReLU aktivációs függvény esetében is segít. Ennek elérésére 2 megoldást alkalmaztam:

- Glorot (vagy Xavier) inícializáció [7]
- Batch normalization [5]

## 4.1. Glorot inícializáció

A cél úgy inícializálni, hogy a varianciája a neuronok bemenetének 1 legyen. Mivel két 1 szórású, 0 várható értékű, független változó szorzatának 1 a várható értéke:

$$\mathrm{Var}(XY) = \mathrm{E}(X^2)\mathrm{E}(Y^2) - [\mathrm{E}(X)]^2[\mathrm{E}(Y)]^2 = 1 - 0 = 1 \tag{1}$$

és két független eloszlású változó varianciája egyszerűen a varianciájuk összege, ezért ha induláskor a neuronok kimenetét függetlennek vesszük, akkor egy adott neuron egy bemenetének varianciája:

$$\mathrm{Var}(x) = n\mathrm{Var}(f) \tag{2}$$

Ahol $n$ a bemenetek száma, $\mathrm{Var}(f)$ az előző réteg kimenetének varianciája (itt felteszem, hogy mindre azonos, ezért nem fontos, hogy $f$ melyik alatta lévő neuront jelöli). $f$ itt egy leaky ReLU kimenete:

$$f(x) = \begin{cases} x & \text{ha } x > 0 \\ 0.1x & \text{különben} \end{cases} \tag{3}$$

Azaz ha $\mathrm{E}[x] = 0$, akkor

$$\mathrm{Var}(f) = \frac{\mathrm{Var}(x) + 0.1\mathrm{Var}(x)}{2} \tag{4}$$

Ezt szeretnénk, ha 1 lenne, azaz:

$$\frac{\mathrm{Var}(x) + 0.1\mathrm{Var}(x)}{2} = 1$$

$$\mathrm{Var}(x) = \frac{2}{1.1}$$

De $\mathrm{Var}(x) = \mathrm{Var}(\sum w_i x_i)$. Felételezve, hogy $x_i$ varianciája 1, átlaga 0, és feltételezve hogy $w_i$-k átlaga 0, varianciája 1, továbbá függetlenek:

$$\mathrm{Var}(w_i) = \frac{2}{1.1n}$$

Ahol $n$ az $x$ bemeneteinek száma. Az eredeti publikációnak megfelelően normál eloszlást választottam [7]:

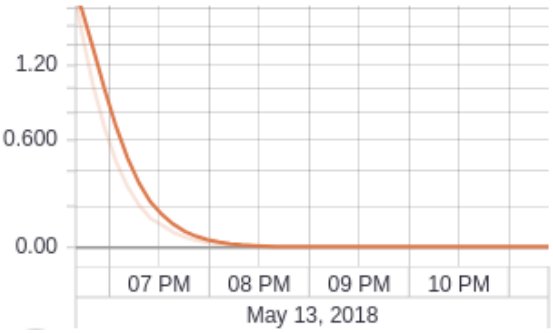
$$w_i \sim \mathcal{N}(0, \frac{2}{1.1n})$$

## 4.2. Ortogonális inícializáció

Nyilvánvaló, hogy ha két neuron kimenete lineárisan összefüggő, akkor az egyiket feleslegesen számoltuk ki: ha $f_1$ az egyik kimenet, $f_2$ a másik és $f_2 = \lambda f_1$, akkor a rákövetkező rétegben $w_1 f_1 + w_2 f_2 = (w_1 + \lambda w_2) f_1$. Emiatt segít, hogy ha a súlyokat ortogonálisra inícializáljuk, amennyire csak lehetséges. Mivel a konvolúció (vagy pontosabban kereszt korreláció) gyakorlatilag skaláris szorzatok sorozata, a súlyok mátrix alakja megtévesztő: valójában inkább vektorokként érdemes kezelni őket, azaz egy 3x3 konvolúciós mátrix felfogható egy 9 elemből álló vektorként. Nyilván ekkor pontosan 9 ortogonális vektort képezhetünk, de ez csak a mátrix (vagyis inkább vektor) méretétől függ. Ezzel a módszerrel valamennyire továbbfejleszthető az eredeti Glorot inícializáció. Legyen $n$ db konvolúciós kernel egy rétegben, ezek mérete legyen $m \times m$-es. Ekkor az algoritmus:

- Vegyünk $n$ db $m^2$ méretű vektort, aminek eloszlása $\mathcal{N}(0, 1)$
- Ortogonalizáljuk SVD algoritmussal
- Szorozzuk meg a kapott mátrixot $\frac{2}{1.1n}$-nel

Amennyiben $m^2 \geq n$, a kapott súlyok páronként ortogonálisak lesznek és varianciájuk $\frac{2}{1.1n}$ lesz, ami megfelel a fentebbi, Glorot inícializáció során támasztott követelménnyel. Tesztjeim során azt kaptam, hogy ez jelentősen javítja a konvergenciát a tanulás kezdeti szakaszában.

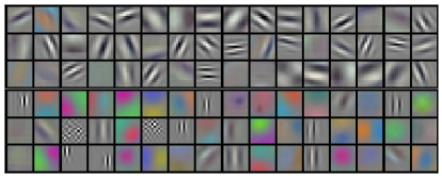



3. ábra. Veszteségfüggvény kimenete. Idővel a súlyok vesztenek ortogonalitásukból, de a tanítás elején segít: alig több, mint 1 óra alatt ezredére csökkent a veszteség.

Ennél vannak komplexebb módszerek is, mint az LSUV inícializáció [13], amelyeket érdemes a jövőben megvizsgálni. Dmytro Mishkin github-on található benchmark-ja szerint valamivel jobban konvergál [2].

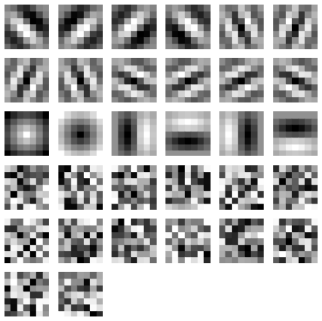
## 4.3. Gábor wavelet

Egy érdekes eredménye az AlexNet publikációnak[10], hogy az első réteg konvolúciós kernele a tanítás végére Gábor-wavelet-hez hasonlóvá vált:



4. ábra. A publikációban szereplő első réteg konvolúciós kerneleinek vizualizációja. Egyes elemei nagyon hasonlítanak a Gábor wavelet-ekre.

A Gábor wavelet-ek használata a képfeldolgozásban nem újkeletű [12], és macskák vizuális kérgének bizonyos része is jól közelíthető vele. Kipróbáltam, hogy az alsó réteget úgy inícializálom, hogy ezeket is tartalmazza, meg pár sima élkeresőt illetve kört. Az élkeresőhöz is Gauss ablakot raktam, mert jól reprezentálja mi a fontos a képfeldolgozás során: az érdekel elsősorban minket, hogy egy pont környezetében él van-e, de attól távolodva a pontok világosságértéke egyre kevésbé fontos, így kevésbé szabályos éleket is meg tud találni.



5. ábra. Kezdeti réteg inícializációja: Az előre definiált kernelek még ki vannak egészítve megfelelő számú véletlengenerálttal.

A kísérleteim eredménye azonban, hogy ezek nem segítettek. A hiba, hogy ehhez viszonylag nagy méretű kernelekre van szükség, ráadásul a legalsó rétegben, ahol a *max pooling* rétegek még nem húzták össze a képet, azaz itt a konvolúció számítási igénye még nagyon magas. Ha a kép $700 \times 700$ képpontból áll, akkor a $n \times n$ kernellel való ú.n. SAME konvolúció $700 \times 700 \times n \times n$ szorzást igényel és $700 \times 700 \times (n \times n - 1)$ összeadást. Ez nagyon drága, ráadásul nem túl rugalmas: nem tudja pl. egykönnyen lecserélni a kernel bal alsó sarkát. Ha ehelyett mondjuk hozáadunk plusz egy réteget, az már képes erre. Ennek következményeként jobb $3 \times 3$-as kerneleket használni az alsó rétegben és a felszabaduló kapacitást több réteg hozzáadására költeni. Fast YOLO [14] is inkább ezt a megoldást használja.

# 5. Batch normalization

A batch normalization[9] egy réteg a neuronhálóban, aminek célja, hogy fenntartsa az 1 körüli varianciát és 0 várható értéket a rajta áthaladó tenzorban, egy dimenzió kivételével - ez utóbbi garantálja, hogy a beérkező adat neurononként legyen normalizálva. Ahogy a neve mutatja, az egész batch-en keresztül optimalizál és legjobban nagy batch size esetén működik - itt ez csak 10, ami limitálja a hatékonyságát. Az algoritmus a következő:

Legyen $x$ a egy darab neuron bemenete.

- $x$-t normalizálja
- a kimenet: $x' = \alpha x + \beta$, ahol $\alpha$ és $\beta$ tanulható paraméterek

$\alpha$ kezdeti értéke 1, $\beta$ kezdeti értéke 0. Az $\alpha$ paraméter oka, hogy ha *szigmoid* vagy *tanh* aktivációs függvényt akarunk használni, akkor a $[-1, 1]$ intervallumon ezek közel lineárisak. Mivel a lineáris aktivációs függvénnyel ellátott neurális hálók nagyon korlátozottak, ezért ez gyakorlatban nem praktikus. ReLU esetén ez a paraméter befagyasztható 1-re. A $\beta$ paraméter értelme pedig, hogy ha a bemenő adat átlagát 0-ra állítjuk, akkor ha a neuron még is 0-tól különböző átlagú bemenetet szeretne (vagyis a hiba visszaterjesztése ebbe az irányba hat), akkor az a végtelenségig fogja növelni az előző réteg *eltolás (bias)* paraméterét - ennek az ő bemenetére semmilyen hatása nem lesz, de elronthatja más neuronok bemenetét vele.

A batch normalizálást közvetlenül a ReLU előtt vagy után érdemes rakni. Christian Szegedy eredetileg a ReLU előtt használta, később áttért arra, hogy utána[5], de a kérdés messze nem eldöntött. Jelen architektúrában utána jobban működik.

# 6. YOLO architektúra

A YOLO architektúra[14] újdonsága a kiemeneti réteg és a veszteségfüggvény. Ezt a felépítést úgy nevezett "object recognition and localization" feladatra találták ki, azaz fel kell ismernie az objektumot és meg kell adnia hozzá egy keretet (bounding box-ot). Ezt leszámítva a háló annyiban tér el az AlexNet-től[10], hogy tartalmaz $1 \times 1$-es konvolúciós réteget, melyet már az Inception is sikeresen használt [4].

## 6.1. Kimeneti réteg

Ha egy képen több objektumot akarunk felismerni, akkor van egy olyan probléma, hogy milyen sorrendben adjuk meg a címkéket. Ha ugyanis a háló más sorrendben találja meg az objektumokat, mint azt megadtuk, akkor a veszteségfüggvény értéke akkor is magas lesz, ha az emberi szemmel az eredmény szinte tökéletes. Az háló kimenetét minden permutációban összehasonlítani a címkékkel nagyon drága művelet, gyakorlatban kivitelezhetetlen. Erre lehet egy megoldás pl. lexiografikusan rendezni a tárgyak pozícióját. Ez akkor okozhat gondoz, ha egy koordinátájuk közel van. A YOLO megoldása erre, hogy a képet felosztja $n \times n$ cellára. Minden cella egy kimeneti vektort ad:

$[x, y, w, h, p, c_1, c_2, ...]$

ahol $(x, y) \in ([0, 1] \times [0, 1])$ a befoglaló téglalap középpontja, $w$ és $h$ a szélessége és magassága, $p$ a *probability*, azaz egy 0 és 1 közötti szám, ami azt reprezentálja, hogy a háló mennyire biztos abban, hogy a cellában van egy objektum. $c_n$ az objektum tipusokhoz rendelt *probability*. Jelen esetben egyetlen objektumot akarok felismerni: egy rendszámtáblát, így az én implementációmban csak az $c_1$ van jelen.

$(x, y)$ az adott cellán belüli pozíciót határozza meg: $(1, 1)$ a cella obb alsó sarkát, $(0, 0)$ a bal felső sarkát jelenti. Megjegyzem, hogy gyakorlatban semmi sem kényszeríti ki, hogy $x$ és $y$ 0 és 1 közötti szám legyen

(nincs pl. egy szigmoiddal beszorítva), de a hálónak adott címkékben mindig az lesz. Gyakorlatban erre gyorsan rájön a háló és tartja magát ehhez, akkor is amikor teljesen hibás kimenetet produkál.

## 6.2. Hibafüggvény

A YOLO hibafüggvénye összetett:

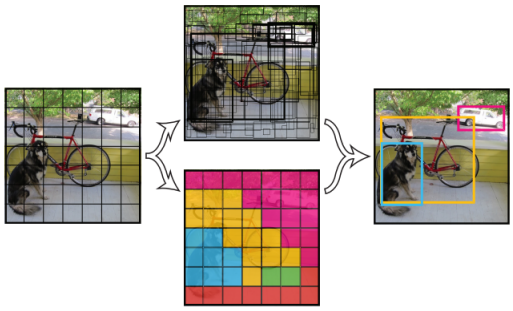
$$\begin{aligned}\text{loss} &= \lambda_{coord}\sum_{i=0}^{S^2}\sum_{j=0}^{B}1_{i,j}^{\text{obj}}(x_i-\hat{x}_i)^2+(y_i-\hat{y}_i)^2\\ &+\lambda_{coord}\sum_{i=0}^{S^2}\sum_{j=0}^{B}1_{i,j}^{\text{obj}}\left((\sqrt{w_i}-\sqrt{\hat{w}_i})^2+(\sqrt{h_i}-\sqrt{\hat{h}_i})^2\right)\\ &+\sum_{i=0}^{S^2}\sum_{j=0}^{B}1_{i,j}^{\text{obj}}(c_i-\hat{c}_i)^2+\lambda_{noobj}\sum_{i=0}^{S^2}\sum_{j=0}^{B}1_{i,j}^{\text{noobj}}(c_i-\hat{c}_i)^2\\ &+\sum_{i=0}^{S^2}1_{i,j}^{\text{obj}}\sum_{j=0}^{B}(p_i(c)-\hat{p}_i(c))^2 \quad (5)\end{aligned}$$

Ahol $1_i^{\text{obj}}$ azt jelöli, hogy az objektum az adott cellában van-e, $1_{i,j}^{\text{obj}}$ pedig hogy az $i$. cella $j$. prediktora felelős az objektum felsimeréséért. Az egyszerűség kedvéért én cellánként egy prediktort használtam, aminek következményeképp egy cellában mindössze egy befoglaló téglalapot találhat a háló maximum. Ez a gyakorlatban a következőket jelenti:

- Ha a tárgy középpontja az nem az adott cellában van, akkor az arra vonatkozó hibafüggvény csak az $1_{i,j}^{\text{noobj}}$ taggal megszorzott rész lesz. Ha a cellába belelóg az $i$. kategóiájú objektum, akkor $\hat{c}_i = 1$, ha nem akkor 0.

- Ha a tárgy középpontja az adott cellában van, akkor a hibafüggvény a középpont, a szélesség és magasság, objektum típus, illetve az valószínűség hibájából fog állni.

Gyakorlatban tehát leginkább azon a cellán keresztül terjeszti vissza a hibát, amiben az bekeretezendő tárgy középpontja található.



6. ábra. A YOLO két fajta kimenete: befoglaló téglalapok és cellában lévő objektum osztálya.

## 6.3. Non-max suppression

Amikor egy háló több befoglaló téglalapot is ad kimenetnek, akkor gyakran előfordul, hogy egy objektumot többszörösen ismer fel. Ilyenkor az érvényesnek tekintett kimenetet a *non-max suppression* algoritmussal választjuk ki. Legyen $p_i(c)$ az adott $i$ cellához és $c$ osztályhoz tartozó *valószínűség* kimenet. Legyen a nyílt halmaz a befoglaló téglalapok (cellák kiementeinek) halmaza, a zárt halmaz pedig a kimenet. Ekkor az algoritmus a következő lépésekből áll:

1. Ha $p_i(c) < \alpha$, akkor a befoglaló téglalapot eldobjuk

2. Egyébként legyen $i_c = \max_i(p_i(c))$, ezt vegyük ki a nyílt halmazból és tegyük át a zárt halmazba

3. Vegyük az $\text{IOU}_{i,j}$ azaz *intersection over union* értékét az összes $j(j \neq i)$ azonos osztályhoz tartozó befoglaló téglalappal

4. Ha ez az $\text{IOU}_{i,j} > \beta$, akkor dobjuk el a $j$. befoglaló téglalapot

5. ismételjük, a 2. ponttól, amíg a nyílt halmaz üres nem lesz

Az IOU pedig a két befoglaló téglalap metszetének és uniójának hányadosa (a 0-val való osztás elkerülése érdekében a metszet uniójához hozzá kell adni egy $\epsilon$ számot). $\alpha$ és $\beta$ szabad paraméterek, jellemzően $\alpha = 0.9$, $\beta = 0.5$ körüli értékek.

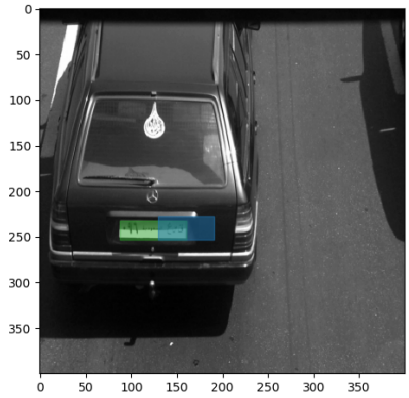
## 7. Optimalizáció

A YOLO nagy hátránya, hogy a bonyolult hibafüggvény és a cellák miatt a gradiens visszaterjesztése lassú. Gyakorlatban tensorflow környezet használva kb. 3-4x lassabb a betanulás, mint amikor a veszteségfüggvény csak a cellákat nem használó háló kimenete a lexiografikusan rendezett befoglaló téglalapoktól való $L_2$ távolsága.

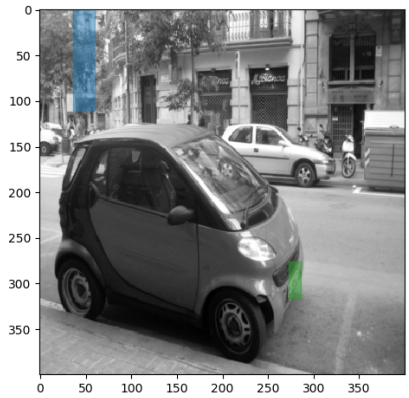
Ennek ellensúlyozására két lépést próbáltam:

1. DataSet használata az előző *input pipeline* helyett
2. $3 \times 3$ max_pool rétegek használata
3. háló rétegeinek csökkentése vagy karcsúsítása

Ebből az 1. semmilyen gyorsulást nem hozott: nem az input pipeline volt a legszűkebb keresztmetszet. A 2. megoldás kiértékelése folyamatban van, a 3. megoldás azonban egyértelműen működik: a hibafüggvényen nem látszott meg az egy rétegben lévő neuronok drasztikus csökkentése sem. Az eredeti YOLO hálózatban több kategóriát (akár ezres nagyságrendben) kellett felismernie. Itt csak rendszámokat kell, így logikus, hogy nem kell annyi *feature*-t kiszednie a képből. Túlzásba nem érdemes esni: a rendszámtábla megtalálásában ideális esetben a környezet is segít (autó pozíciója és iránya, út helyzete), némely esetekben pedig semmilyen más lehetőség nincs.



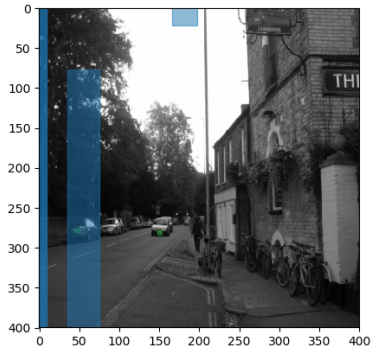

7. ábra. Kb. tízből egyszer ilyen eredményt ad, de inkább kevesebbszer. Itt $\alpha = 0.95$




8. ábra. Egy átlagos eredmény. Itt $\alpha = 0.95$

## 8. Eredmények

Ennek a papírnak az elkészültéig a háló betanulása nem fejeződött be. Gyakorlatban a fentieken kívül még a hyperparaméterek beállításával is foglalkoznom kellett. A tanulást egy Geforce 1070 videókártyán 16GB ram memóriával végeztem, ezzel gyakorlatban minimum 3-4 nap nem inkább több, mire ellenőrizni tudtam, hogy egy adott konfigurációval hogyan halad a háló betanulása. Mire lejutottam a többé-kevésbé végleges verzióhoz, már nem állt elegendő idő a számomra, hogy a betanulással is végezzen, ani minimum 2 hét, de inkább több.




9. ábra. Tízből 2-3x liyen eredményt produkál. Itt $\alpha = 0.95$

Nem véletlen, hogy 7. ábrán viszonylag jól találja meg még a betanítás ilyen korai szakaszában is: a rendszámtábla jól kivehető, árnyalatban erősen eltér a kép többi részétől. Jelenleg az eredmény nagyon változó: van pár meglepően jó, de bőven többségben vannak azok, ahol még semmilyen értékelhető kimenetet nem produkál, azaz pl a rendszámok számát sem találja el. A korábban is említett hálóm - amelyik a lexiografikusan rendezett befoglaló téglalapokat kellett kiadnia - ebben meglepően jó volt: a koordinátáknak általában nem sok köze volt annak, ahol a képen a rendszámtábla volt, de a számukat nagyon jól eltalálta - az objektumok felismerése már egész jól ment, azok lokalizálása azonban nem.

# 9. További irány

Az előző félév végi célokból csak párat sikerült kipipálnom:

- YOLO leimplementálása
- befoglaló téglalap keresése poligon helyett
- további adathalmaz szerzése
- batch normalizáció
- súlyok inícializálása
- alsó réteg éldetektorokra való inícializációja

A hátralévő elemek:

- RCNN: lassab betanítani és futtatni, de jobb eredményt ad - egy próbát megér
- példa generálás: a képeket lehet torzítani, zajjal ellátni, kivágni belőle darabokat
- képek előfeldolgozása: pl. HOG-ot (histogram of gradient-et) is beadni a hálónak, vagy éldetektor futtatása: ez annyiban tér el attól, hogy ha az alsó réteget éldetektorokra inícializálom, hogy nem kell minden alkalommal újra és újra megcsinálni, elég egszer - mivel az első kovolúciók a legdrágábbak, ez gyakorlatban meggyorsíthatja a betanítást
- vonal pásztázó: függőleges és/vagy vízszintes pásztázó algoritmusként használni a hálót: egyszerre csak egy sort (oszlopot) kap bemenetnek, és az alapján kell eldöntenie, hogy abban a sorban van-e rendszám. Ez gyorsabb lenne és ezáltal jobban tanítható korlátozott számítási kapacitás esetén. Ehhez hasonló megoldás 1xn és nx1-es konvolúciós kernelek használata az első rétegben - ugyanaz, csak tanítható, de minden képre újra és újra ki kell számolni
- előtanítás hasonló példahalmazon: a házszámok elolvasása pl. elég hasonló probléma. Ez egy bevett módszer (Karpathy említi a Stanford-on tartott CS231n kurzusán) kevés rendelkezésre álló példa esetén. A dolog hátránya hogy azonos architektúrát kell használni az eredeti megoldással - ezeket azonban nem az én lehetőségeimre szabták. Nem hiszem, hogy először ezzel fogom folytatni a munkát, de a lehetőséget nem zárom ki: lehet kísérletezni pl. azzal, hogy csak az alsó pár réteget veszem át.
- be lehet tanítani a háló egy darabját járművek felismerésére, majd ezt be lehet kötni mint input egy másik háló számára. Ezt használja pl [6] (bár a 2 hivatkozással ez nem tűnik túl... népszerű publikációnak, de más helyen is találkoztam ezzel a megoldással). Személyes véleményem hogy ez "overkill" - persze hogy több, teljes hálót (amelyek több ezer kategóia közül ismernek fel objektumokat) egymás után kötve jó eredményt lehet hozni, de az erőforrásigénye hatalmas a konvencionálisabb, mélytanulást nem használó módszerekhez képest, amelyek nem teljesítenek jelentősen rosszabbul a gyakorlatban (vagy is pl. fix kameraállás mellett).

Amiket jelenleg nem tartok reális megoldásnak:

- capsule network[15]: jelenlegi erőforrásaimmal nem tűnik reális alternatívának és semmilyen benchmark-on nem szerepel. Az a benyomásom, hogy ez inkább a jövő zenéje: az ötlet jó, talán egyenesen forradalmi, de a hardware-eknek fejlődniük kell, hogy ki lehessen használni és a dynamic routing-ot is nehéz hatékonyan implementálni. Az igazság, hogy ebbe nem ástam bele magamat még túlságosan, ezért nem tartom elképzelhetetlennek.
- neural turing machine: különösen jól teljesít kevés példából tanulás esetén[8], de rekurrens, ami lassítja a betanulást és valószínűleg, ha csak nem tudok plusz erőforrást találni, nem reális

- hyperparameter algoritmikus beállítása: amíg 1 hétig fut egy próbatanítás, addig ezeket kézzel "megérzés alapon" állítgatni nem a legszükebb keresztmetszet a feladat hatékony megoldásában

# Hivatkozások


[1] Ade20k database. http://groups.csail.mit.edu/vision/datasets/ADE20K/. Accessed: 2018-05-10.

[2] Caffenet batch normalization benchmark. https://github.com/ducha-aiki/caffenet-benchmark/blob/master/batchnorm.md. Accessed: 2018-05-10.

[3] Imagenet database. http://www.image-net.org/. Accessed: 2018-05-10.

[4] Md Zahangir Alom, Mahmudul Hasan, Chris Yakopcic, and Tarek M Taha. Inception recurrent convolutional neural network for object recognition. *arXiv preprint arXiv:1704.07709*, 2017.

[5] François Chollet. Batch normalization in keras. https://github.com/keras-team/keras/issues/1802issuecomment-187966878. Accessed: 2018-05-10.

[6] C Gerber and M Chung. Number plate detection with a multi-convolutional neural network approach with optical character recognition for mobile devices. 12:100–108, 03 2016.

[7] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.

[8] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

[9] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[10] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[11] Yann LeCun, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller. Efficient backprop. In *Neural networks: Tricks of the trade*, pages 9–50. Springer, 1998.

[12] Tai Sing Lee. Image representation using 2d gabor wavelets. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 18(10):959–971, Oct 1996.

[13] Dmytro Mishkin and Jiri Matas. All you need is a good init. *arXiv preprint arXiv:1511.06422*, 2015.

[14] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 779–788, 2016.

[15] Sara Sabour, Nicholas Frosst, and Geoffrey E Hinton. Dynamic routing between capsules. In *Advances in Neural Information Processing Systems*, pages 3857–3867, 2017.