1 Introducere

În ultimul timp, rețelele neuronale artificiale sunt din ce în ce mai întâlnite în învățarea automată, însă acestea nu sunt nicidecum tehnici noi, ele fiind propuse imediat după Cel de-al Doilea Război Mondial. Mai exact, prima rețea neuronală a fost construită în anul 1948 și a încercat să propună un model matematic pentru modul în care funcționează neuronii biologici. Marile piedici pentru *deep learning*[1] în acea perioadă erau complexitatea computațională în procesul de antrenare și necesitatea unui nivel mare de date de antrenament pentru a obține o performanță bună. Astfel, rețelele neuronale și-au pierdut atractivitatea, fiind preferate alte metode de clasificare precum SVM (Support Vector Machines) sau clasificatori liniari. Odată cu creșterea în popularitate a internet-ului, a crescut și nivelul de date distribuite public, făcând tehnicile de deep learning viabile în contextul actual, cu performanțe chiar mai bune decât metodele clasice.

Domeniul *Computer Vision* [2] a cunoscut un adevărat progres în jurul anului 2012 atunci când Alex Krizhevsky, Ilya Sutskever și Geoffrey Hinton au construit ceea ce s-a numit AlexNet.[1] O rețea neuronală convoluțională [3] prin care au obținut o eroare de 15.3% în cadrul competiției ImageNet LSVRC-2010 care constă în clasificarea a 1.2 milioane de imagini de rezoluție înaltă într-una din 1000 de clase. Acest rezultat a fost cu 10.8% mai bun decăt precedentul, arătând că tehnicile de deep learning au un potențial enorm în problema recunoșterii de imagini. La data scrierii acestei lucrări, eroarea în cadrul competiției este $\approx 2.9\%$.

Prima rețea convoluțională a fost creată în anul 1998 de către Yann LeCun, numindu-se LeNet-5. [2] Scopul ei a fost clasificarea cifrelor scrise de mână, fiind inspirată de anumite descoperiri din biologie care s-au referit la faptul că, în cortexul vizual, există neuroni care răspund individual la regiuni mici dintr-un anumit stimul, creierul neprocesând o imagine ca un tot unitar.

[1] Familie de algoritmi de învățare automată ce au la bază rețelele neuronale

[2] Domeniu al inteligenței artificiale ce își propune înțelegerea imaginilor și a video-urilor de către calculator

[3] Rețea neuronală folosită în recunoașterea de imagini cu ajutorul operației de convoluție

În cadrul acestei lucrări vom studia folosința rețelelor neuronale convoluționale, a celor *obișnuite*, cât și a altor tehnici de învățare automată pentru a asigna independent unei instanțe, definită în problemă ca o mulțime de fotografii dintr-un reastaurant, fiecare dintre următoarele clase:

1. bun pentru prânz (good for lunch)
2. bun pentru cină (good for dinner)
3. acceptă rezervări (takes reservations)
4. are sejur în aer liber (outdoor seating)
5. este scump (restaurant is expensive)
6. oferă alcool (has alcohol)
7. are serviciu de masă (has table service)
8. atmosfera este rustică (ambience is classy)
9. bun pentru copii (good for kids)

Această problemă a fost propusă în anul 2015 de cei de la Yelp prin platforma Kaggle.[13] Motivația din spate a fost că răspunsurile pentru întrebările de mai sus reprezintă un factor important în sistemele lor de recomandări, însă utilizatorii nu le oferă foarte des. În acest caz, un model de învățare automată care primește ca date de intrare fotografii dintr-un restaurant și oferă o valoare din mulțimea $\{0, 1\}$ pentru fiecare clasă ar fi de folos.

Structura datelor este următoarea:

- 234842 de imagini de antrenament în format .jpg și .png.
- 237152 de imagini de test folosite pentru a determina scorul în cadrul competiției.

- ***train_photo_to_biz_ids.csv***: tabel ce asociază fiecare fotografie de antrenament la restaurantul din care provine folosind id-uri.

- ***train.csv***: tabel ce conține 2 coloane $\{business\_id, labels\}$ semnificând etichetarea unui restaurant (prin id-ul său) cu o submulțime din $\{1, 2, ..., 9\}$ ce reprezintă cele 9 clase ilustrate mai sus.

- ***test_photo_to_biz_ids.csv***: tabel ce asociază fiecare fotografie de test la restaurantul din care provine folosind id-uri.

În total sunt 1996 de restaurante în setul de antrenament și 10000, ce trebuie clasificate, în setul de testare.

2 Problema clasificării. Tipuri de clasificare

2.1 Învățare supervizată

În contextul învățării supervizate ne este pus la dispoziție un set de date format din perechi (x_i, y_i), x_i numindu-se *instanță* și y_i *etichetă*. Obiectivul unui algoritm de învățare automată este de a învăța o funcție care să modeleze cât mai bine relația dintre x_i și y_i ce reiese din date. După natura variabilei y, se pot defini 2 tipuri de probleme:

- Clasificare (variabila y este discretă)
- Regresie (variabila y este continuă)

În această lucrare vom aborda doar subiectul clasificării.

2.2 Clasificare binară

Problema clasificării binare ne cere ca pentru o instanța $x \in \mathbb{R}^p$ $p \in \mathbb{N}^*$ dată ca input și o clasă C, să returnăm o valoare $y \in \{0, 1\}$ astfel încât $y = 1$ dacă x face parte din clasa C, sau $y = 0$ în caz contrar. Mai formal, trebuie să gasim o funcție $h : \mathbb{R}^p \to \{0, 1\}$ definită astfel:

$$h(x) = \begin{cases} 1, & \text{x} \in \text{C} \\ 0, & \text{altfel.} \end{cases}$$

Această funcție se mai numește model sau ipoteză și scopul unui algoritm de învățare automată este de a oferii un model cât mai bun.
Un exemplu de clasificare binară este ca pentru o imagine să decidem dacă în aceasta se află sau nu o pisică. În acest caz, instanța x va fi un vector ce conține valoarea reală a fiecarui pixel și output-ul va fi $y = 1$ dacă în imagine apare o pisică sau $y = 0$ în caz contrar.

2.3 Clasificare cu clase multiple

La fel ca în cazul clasificării binare, instanța este $x \in \mathbb{R}^p$ $p \in \mathbb{N}^*$, însă, în loc de o singură clasă C, avem o mulțime $\{C_1, C_2, ..., C_k\}$ $k \in \mathbb{N}^*$ disjuncte, iar obiectivul este să găsim o

funcție $h : \mathbb{R}^p \to \{0, 1, 2, ..., k\}$ astfel încât:

$$h(x) = \begin{cases} 1, & \text{x} \in C_1 \\ 2, & \text{x} \in C_2 \\ \vdots & \vdots \\ \text{k}, & \text{x} \in C_k \\ 0, & \text{altfel.} \end{cases}$$

2.4 Clasificare cu etichete multiple

Pentru o instanță definită la fel ca mai sus $x \in \mathbb{R}^p$ $p \in \mathbb{N}^*$ și o mulțime $C = \{C_1, C_2, ..., C_k\}$ $k \in \mathbb{N}^*$ vom construi o ipoteză $h : \mathbb{R}^p \to \{0, 1\}^k$ astfel încât pentru o predicție $y = h(x)$ să avem:

$$y_i = \begin{cases} 1, & \text{x} \in C_i \\ 0, & \text{altfel.} \end{cases} \quad i = \overline{1, k}$$

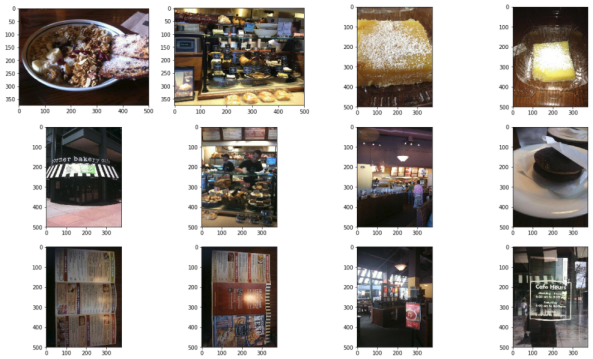
Apare des o confuzie între clasificarea cu etichete multiple și cea cu clase multiple. Diferența fundamentală este că în prima o instanță poate să aparțină mai multor clase, spre deosebire de cealaltă, în care o instanță este asignată unei singure clase. Totodată, clasificarea binară este un caz particular a clasificării cu etichete multiple atunci când $k = 1$. Detectarea obiectelor face parte din acest tip de clasificare, unde, pentru o imagine, vom marca prezența mai multor obiecte (clase) și nu a unui singur element.

2.5 Învățare multi-instanță

În acest caz, o instanță nu mai este reprezentată de un singur vector de numere reale, ci de o mulțime de astfel de vectori. Fie $X = \{x_1, x_2, ..., x_k\}$ $k \in \mathbb{N}^*$ $x_i \in \mathbb{R}^{p_i}$ $p_i \in \mathbb{N}^*$ o instanță. O ipoteză h determinată de un algoritm de învățare automată va clasifica întreaga mulțime X și nu fiecare componentă x_i independent. [3]

Problema abordată în această lucrare este una de învățare multi-instanță și de clasificare cu etichete multiple. Un restaurant este reprezentat de o mulțime de imagini și rezultatul

este un vector y cu 9 componente, fiecare marcând prezență sau absență celor 9 clase descrise în capitolul introductiv.



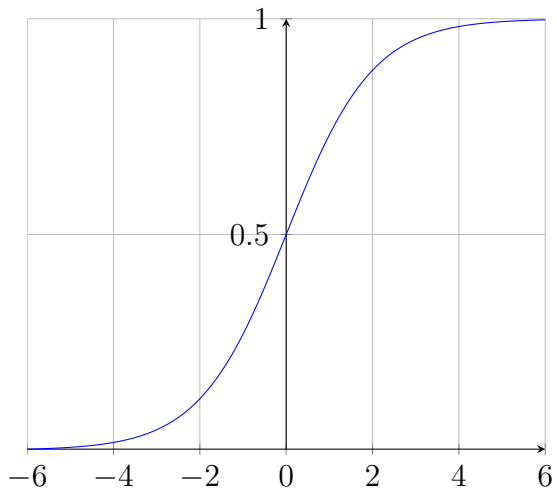
$$\longrightarrow \quad y = [1, 0, 0, 1, 1, 0, 1, 1, 0]$$

O instanță formată din 12 fotografii și vectorul asociat

3 Rețele neuronale

3.1 Regresie logistică

În cadrul acestui model vom calcula, pentru un anumit $x \in \mathbb{R}^{n_x}$, probabilitatea $P(y = 1|x)$. Parametrii modelului sunt un vector $w \in \mathbb{R}^{n_x}$ și un număr real $b \in \mathbb{R}$. Astfel, regresia logistică se definește ca $\hat{y} = \sigma(w^T x + b)$, în care funcția σ are următoarea expresie: $\sigma(z) = \frac{1}{1+e^{-z}}$. Această funcție se numește funcția logistică sau funcția sigmoid și are următorul grafic:



Astfel, în contextul clasificării binare, vom eticheta instanța x cu:

$$y_x = \begin{cases} 1, & \hat{y} \geq 0.5 \\ 0, & \hat{y} < 0.5 \end{cases}$$

Fiind dat un set de date cu m exemple $\{(x^{(1)}, y^{(1)})...(x^{(m)}, y^{(m)})\}$ dorim ca $\hat{y}^{(i)} \approx y^{(i)}$. Vom defini, pentru o predicție, funcția de eroare:

$$L(\hat{y}, y) = -(ylog(\hat{y}) + (1-y)log(1-\hat{y}))$$

și costul pentru întregul set de date este dat de:

$$J(w, b) = \frac{1}{m} \sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)})$$

Regresia logistică este, în fapt, o problemă de optimizare, în care încercăm sa găsim o configurație cât mai bună pentru parametrii w și b pentru a minimiza costul J pentru un

anumit set de date, iar în continuare vom prezenta câțiva algoritmi care ne vor ajuta în acest sens.

3.2 Metode de optimizare bazate pe gradient

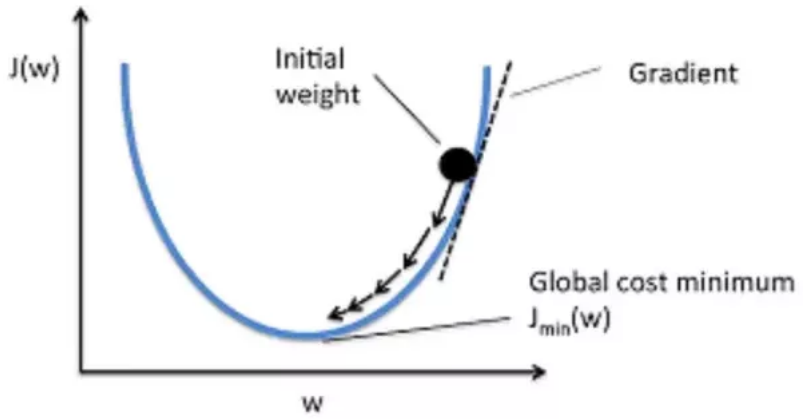
3.2.1 Metoda gradientului descendent

Aceasta este un algoritm de optimizare iterativ care aproximează minimul, de cele mai multe ori cel local, a unei funcții. În contextul funcției de cost J asociată regresiei logistice, algoritmul gradient descent are următorul pseudocod:

Algorithm 1 Gradient Descent

1: **while** stopping condition **do**
2: $w = w - \alpha \frac{\partial J(w,b)}{\partial w}$
3: $b = b - \alpha \frac{\partial J(w,b)}{\partial b}$

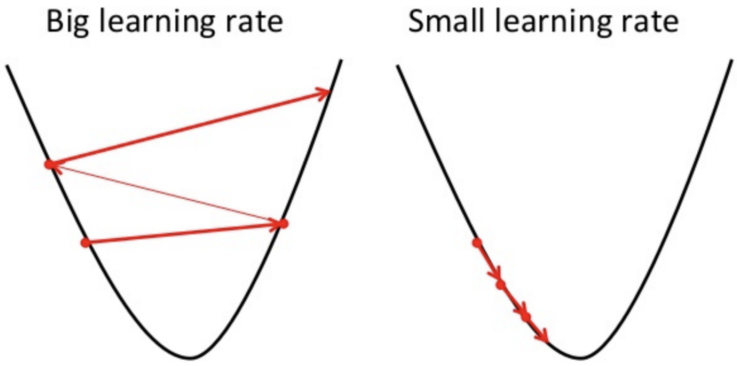
Condiția de oprire poate să fie efectuarea unui număr de iterații sau neîmbunătățirea rezultatului cu o valoare mai mare decât un ϵ fixat. Din punct de vedere grafic, pașii algoritmului arată astfel:



Funcția J din figură este o variantă simplificată a celei folosită în model, ignorând parametrul b

Există un așa numit hyper-parametru α (learning rate) care influențează mărimea pașilor efectuați atunci când algoritmul mișcă într-o direcție parametrii funcției ce trebuie minimizată. Acesta este fixat, rămâne constant pe tot parcursul execuției și valorile obișnuite sunt

de ordinul $\{0.01, 0.001, 0.0001, ...\}$. Totuși, dacă folosim o valoare mare, apare riscul de a face pași prea mari și să ratăm cu totul minimul, iar dacă folosim o valoare mică, dispare acest risc, însă convergență ar necesita mult mai mult timp din cauza pașilor foarte mici.

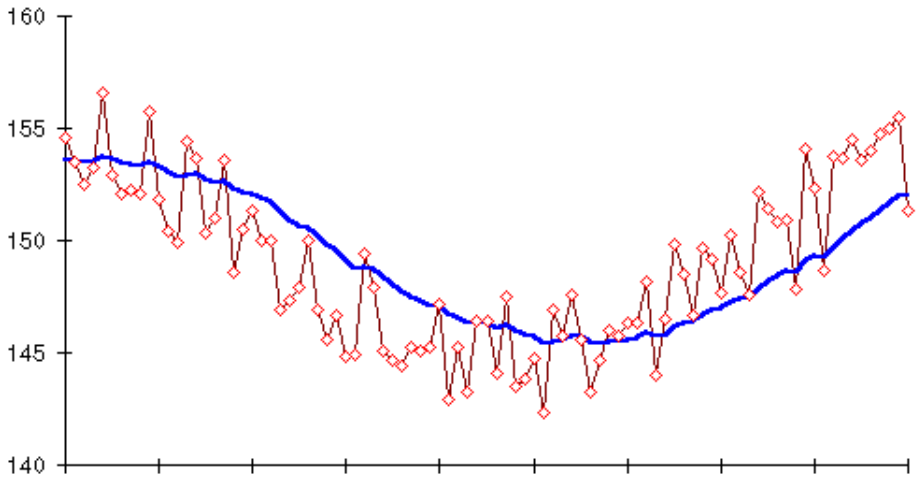


Problema de optimizare din cadrul regresiei logistice este una convexă iar algoritmul Gradient Descent face o treabă bună în a minimiza costul J, neavând inconvenienţa minimelor locale.

3.2.2 Momentum

În următoarele 3 puncte vom prezenta imbunătățiri aduse algoritmului inițial, ce au ca efect scăderea timpului necesar pentru convergență. Prima dintre aceste îmbunătățiri se numește *Momentum* și are la bază conceptul de medii ponderate exponențiale.

Fie un set de puncte într-un plan $\{p_0, p_2, ..., p_n\}$. Prin folosirea acestor medii, se poate aproxima eficient, un drum ce începe din p_0, se termină în p_n și trece prin toate cele n puncte.



Drumul complet de la p_0 la p_n (linia roșie) și drumul aproximat (linia albastră)

Linia albastră este dată de următoarea regulă:

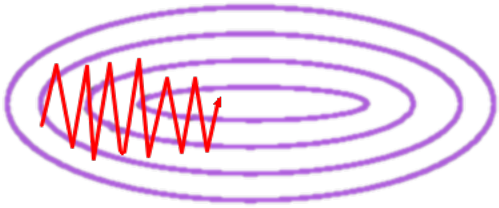
$$v_t = \begin{cases} 0, & t = 0 \\ \beta v_{t-1} + (1-\beta)p_t & t > 0 \end{cases} \qquad t = \overline{0, n}$$

Aceste idei se pot implementa în cadrul algoritmului original Gradient Descent pentru a grăbi convergență astfel:

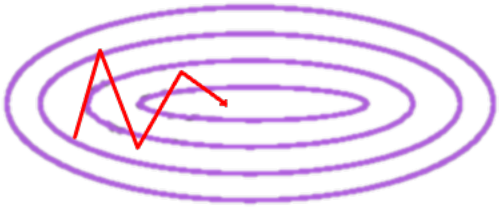
Algorithm 2 Gradient Descent with Momentum

$v_{dW} = 0$ $v_{db} = 0$
while stopping condition **do**
  $dW = \frac{\partial J(w,b)}{\partial w}$
  $db = \frac{\partial J(w,b)}{\partial b}$
  $v_{dW} = \beta v_{dW} + (1-\beta)dW$
  $v_{db} = \beta v_{db} + (1-\beta)db$
  $W = W - \alpha v_{dW}$
  $b = b - \alpha v_{db}$

În acestă versiune mai apare incă un hyper-parametru β, însă acesta are valoarea 0.9 și este rar modificat.



Pașii făcuți de algoritmul original



Pașii făcuți când se folosește Momentum

3.2.3 RMSProp

Algorithm 3 RMSProp

$s_{dW} = 0$ $s_{db} = 0$

while stopping condition **do**

$dW = \frac{\partial J(w,b)}{\partial w}$

$db = \frac{\partial J(w,b)}{\partial b}$

$s_{dW} = \beta s_{dW} + (1 - \beta) dW^2$

$s_{db} = \beta s_{db} + (1 - \beta) db^2$

$W = W - \alpha \frac{dW}{\sqrt{s_{dW} + \epsilon}}$

$b = b - \alpha \frac{db}{\sqrt{s_{db} + \epsilon}}$

RMSProp își propune să micșoreze oscilațiile în direcții divergente de locația minimului, făcute de Gradient Descent, prin modificarea cu valori foarte mici a parametrilor ce duc spre direcții neoptime. [11]

3.2.4 Adam

Adam combină Momentum și RMSProp într-un singur algoritm oferind, în majoritatea cazurilor, un rezultat mai bun decât cele 2. [10]

În acest caz, există 3 hyper-parametrii α, β_1, β_2. Dacă $\beta_1 = 0.9$ și $\beta_2 = 0.99$ sunt valori standardizate și foarte rar schimbate, α reprezintă cel mai important hyper-parametru și cel care are efectul net cel mai mare. Din această cauză, căutarea unei valori cât mai bune pentru acesta este un proces necesar pentru a obține o performanță bună.

Algorithm 4 Adam

$v_{dW} = 0\ v_{db} = 0\ s_{dW} = 0\ s_{db} = 0$

while stopping condition **do**

On iteration t

$dW = \frac{\partial J(w,b)}{\partial w}\ db = \frac{\partial J(w,b)}{\partial b}$

$v_{dW} = \beta_1 v_{dW} + (1-\beta_1)dW\ v_{db} = \beta_1 v_{db} + (1-\beta_1)db$

$s_{dW} = \beta_2 s_{dW} + (1-\beta_2)dW^2\ s_{db} = \beta_2 s_{db} + (1-\beta_2)db^2$

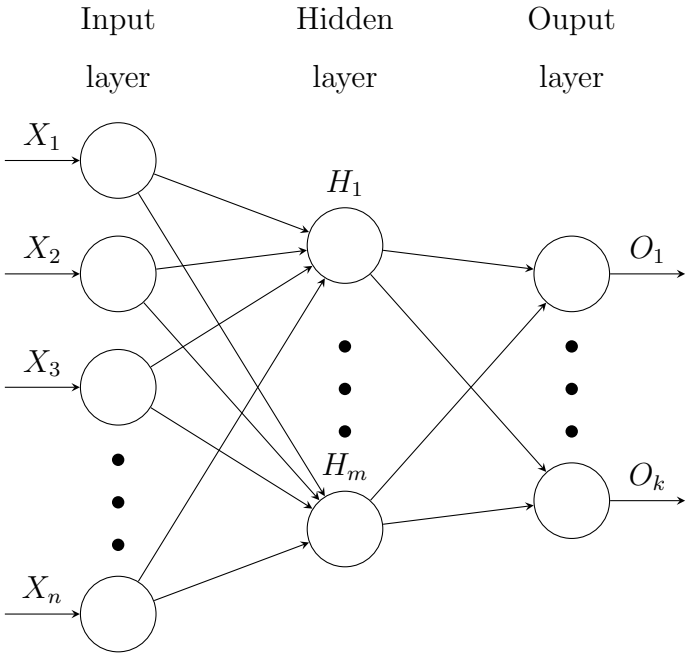
$v_{dW}^{corrected} = \frac{v_{dW}}{1-\beta_1^t}\ v_{db}^{corrected} = \frac{v_{db}}{1-\beta_1^t}$

$s_{dW}^{corrected} = \frac{s_{dW}}{1-\beta_2^t}\ s_{db}^{corrected} = \frac{s_{db}}{1-\beta_2^t}$

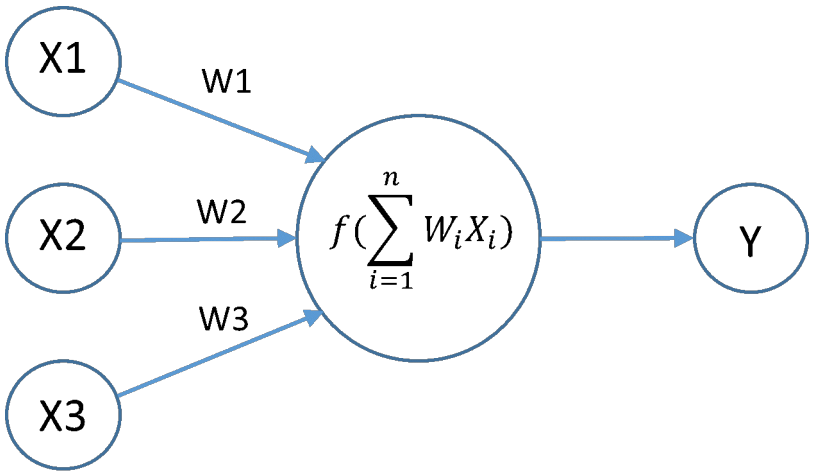
$W = W - \alpha \frac{v_{dW}^{corrected}}{\sqrt{s_{dW}^{corrected}} + \epsilon}\ b = b - \alpha \frac{v_{db}^{corrected}}{\sqrt{s_{db}^{corrected}} + \epsilon}$

3.3 Rețele neuronale

3.3.1 Structură și notații



Imaginea de mai sus reprezintă o rețea neuronală clasică și, la nivel înalt, are 3 componente principale. Un *input layer*, unul sau mai multe *hidden layers* și un *output layer*. Informația pornește de la input către output fiind procesată pe parcurs și la baza întregii construcții se află *neuronul*, reprezentat în figură prin cerc.



Un neuron

Fie $x = [x_1, x_2, x_3],\ w = [w_1, w_2, w_3] \quad x, w \in \mathbb{R}^3$ și $b \in \mathbb{R}$ (omis în figură). Operațiile care se fac în cadrul unui neuron sunt:

1. $z = w^T x + b$

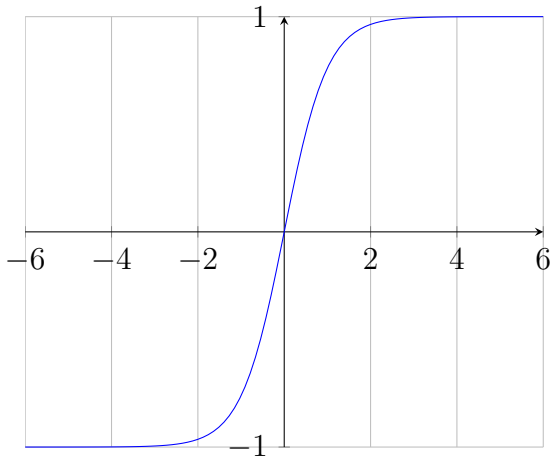
2. $y = f(z)$

În care vectorul w și numărul b sunt parametrii, x este un vector input și y se numește valoarea activării neuronului, f fiind funcția de activare.
Se observă că dacă înlocuim f cu funcția sigmoid $\sigma(z) = \dfrac{1}{1+e^{-z}}$ obținem fix modelul regresiei logistice, el putând fi interpretat ca o rețea neuronală cu un singur neuron.

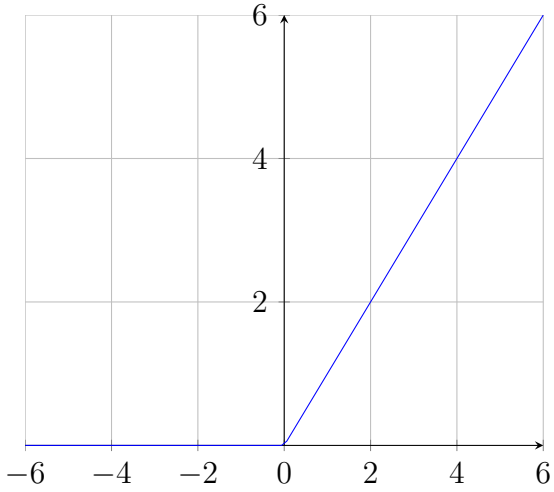
3.3.2 Funcții de activare

Am văzut cât este de simplu să obținem regresia logistică dacă înlocuim funcția de activare cu funcția sigmoid, însă există și alte opțiuni consacrate pe lângă aceasta.

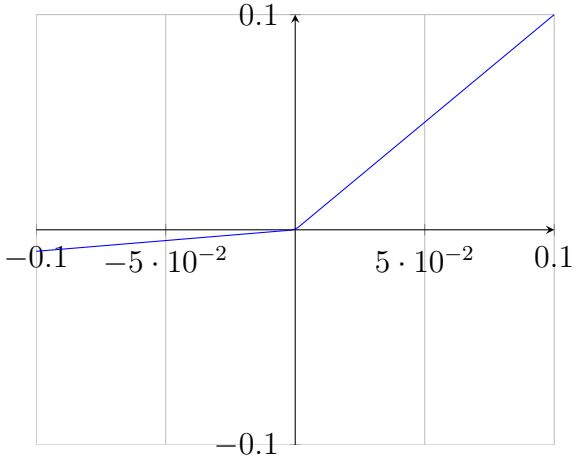
1. **Funcția tangentă hiperbolică (Hyperbolic tangent)**: $tanh(z) = \dfrac{e^x - e^{-x}}{e^x + e^{-x}}$



2. **Rectified Linear Unit (ReLU)**: $relu(z) = max(z, 0)$

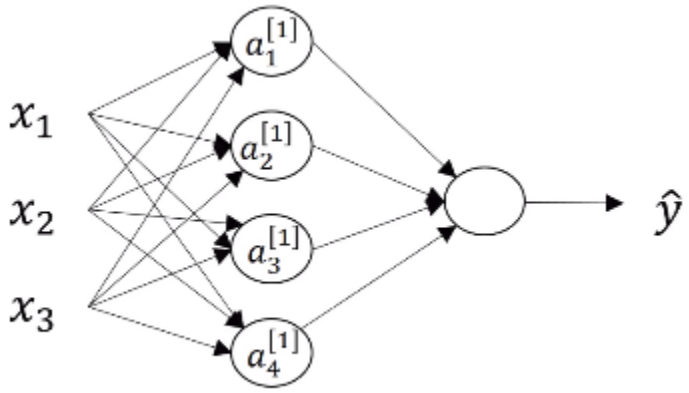


3. **Leaky ReLU**: $leakyReLU(z) = max(z, 0.01z)$



3.3.3 Propagarea înainte (Feed-Foward Propagation)

În această secțiune vom studia operațiile ce intervin când informația parcurge traseul *input-output*. Fie următoarea rețea neuronală:



Modelul nostru primește ca date de intrare un vector $x \in \mathbb{R}^3$, are un *hidden layer* cu 4 neuroni și va oferi ca *output* un număr real. Vom folosi notațiile:

- $x \in \mathbb{R}^3 \rightarrow$ vector de input.

$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

- $a_t^{[l]} \rightarrow$ Activarea neuronului t din layer-ul l.
- $A^{[l]} \rightarrow$ Vector ce conține activările tuturor neuronilor din layer-ul l.
- $W^{[1]} \in M_{4,3}(\mathbb{R}) \rightarrow$ Matrice ce conține parametrii w pentru fiecare neuron din primul layer pe câte o linie.

$$W^{[1]} = \begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \\ w_{31} & w_{32} & w_{33} & w_{34} \\ w_{41} & w_{42} & w_{43} & w_{44} \end{bmatrix}$$

- $W^{[2]} \in M_{1,4}(\mathbb{R}) \rightarrow$ Matrice ce conține parametrii w pentru fiecare neuron din ultimul layer pe câte o linie.

$$W^{[2]} = \begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} \end{bmatrix}$$

- $b^{[1]} \in \mathbb{R}^4 \quad b = [b_1, b_2, b_3, b_4] \rightarrow$ vector ce reține parametrii b pentru fiecare neuron din primul layer.
- $b^{[2]} \in \mathbb{R} \quad b = [b_1] \rightarrow$ vector ce reține parametrii b pentru fiecare neuron din ultimul layer.

Astfel operațiile ce se produc în propagarea $input - output$ sunt:

1. $A^{[0]} = x$ considerăm x valorile de activare a layer-ului de *input*.
2. $Z^{[1]} = W^{[1]}a^{[0]} + b^{[1]}$
3. $A^{[1]} = f(z^{[1]})$
4. $Z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$
5. $A^{[2]} = f(z^{[2]})$
6. $\hat{y} = A^{[2]}$

$A^{[2]}$ reprezintă valoarea finală, fiind activarea layer-ului de *output*, iar f este o funcție oarecare de activare. [12]

3.3.4 Funcția de cost

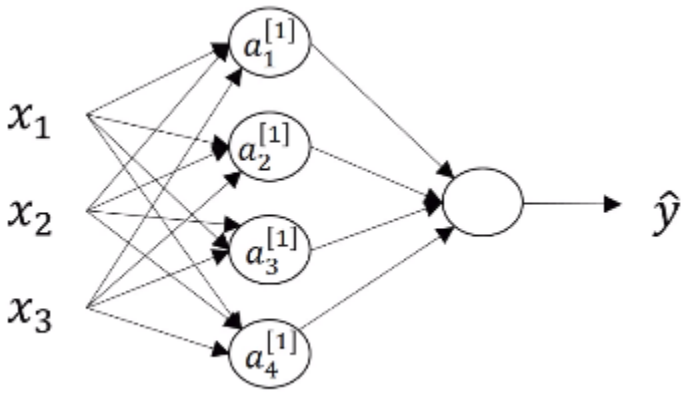
Am văzut cum regresia logistică este o rețea neuronală foarte simplă. Datorită acestui fapt, funcția de cost folosită atunci, în cadrul clasificării binare, poate fi utilizată pentru a antrena și rețele neuronale complexe.

$$L(\hat{y}, y) = -(ylog(\hat{y}) + (1-y)log(1-\hat{y}))$$

$$J(W, b) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)})$$

3.3.5 Propagarea înapoi (Backpropagation)

O problemă complicată în antrenarea rețelelor neuronale este propagarea erorii ce este produsă de activarea ultimului layer, cel de *output*, către neuronii aflați la începutul rețelei. Am prezentat modul în care informația parcurge traseul $input - output$, însă, pentru optimizarea parametrilor, trebuie sa parcurgem traseul opus, $output - input$, inversând toate operațiile făcute. Pentru aceasta, vom prezenta algoritmul *Backpropagation*. Fie aceeași rețea neuronală ca la *Feed-Foward Propagation* și folosind aceleași notații:



Operațiile efectuate de către *Backpropagation* sunt:

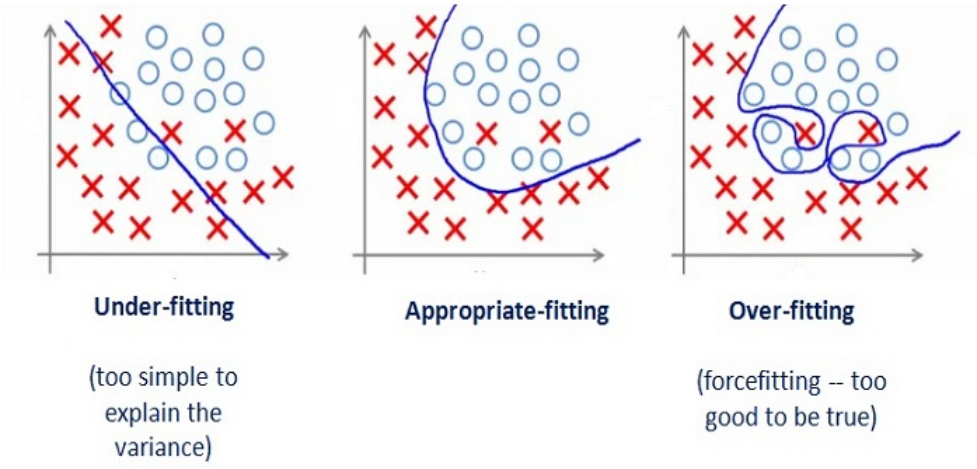
1. $A^{[2]} = \hat{y}$
2. $dZ^{[2]} = A^{[2]} - Y$
3. $dW^{[2]} = \frac{1}{m} dZ^{[2]} {A^{[2]}}^T$
4. $db^{[2]} = \frac{1}{m} np.sum(dZ^{[2]}, axis = 1, keepdims = True)$ [4]
5. $dZ^{[1]} = {W^{[2]}}^T dZ^{[2]} * f'(Z^{[1]})$
6. $dW^{[1]} = \frac{1}{m} dZ^{[1]} X^T$
7. $db^{[1]} = \frac{1}{m} np.sum(dZ^{[1]}, axis = 1, keepdims = True)$

[4] comandă *numpy* ce face suma compenentelor pentru fiecare linie (pentru un input de forma $[n, m]$ vom obține un output de forma $[n, 1]$)

După execuția acestui algoritm, avem calculați 4 vector $\{dW^{[2]}, db^{[2]}, dW^{[1]}, db^{[1]}\}$ pe care îi vom folosi pentru a modifica parametrii W, b a fiecărui layer, folosind unul din algoritmii de optimizare bazați pe gradient prezentați anterior. [12]

3.3.6 Regularizare L_2 și Dropout

În învățarea automată, o problemă frecventă este super-specializarea (overfitting) și, din păcate, aceasta este foarte întâlnită și în deep learning.



Pentru a ameliora acest efect nedorit, vom folosi ceea ce se numește *regularizare*, cele mai folosite 2 forme fiind:

1. **Regularizare L_2**:

   Vom aduce o mică modificare funcției de cost inițiale:

   $$J(W^{[l]}, b^{[l]}) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\sum_{l=1}^{L} \|W^{[l]}\|_F^2$$

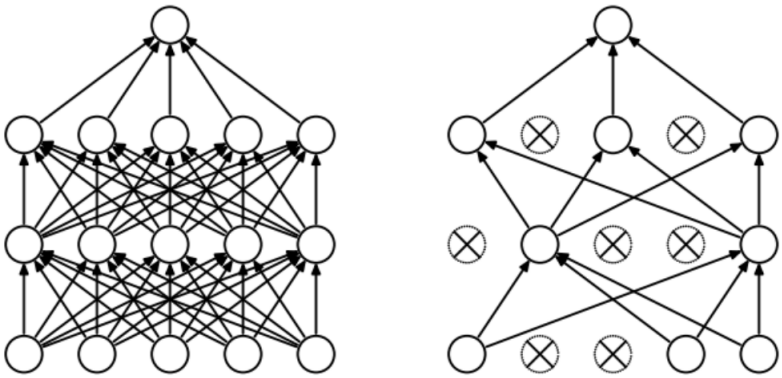
   $\lambda \rightarrow$ termen de regularizare

   $L \rightarrow$ Numărul de layere din rețea

   Pentru a minimiza această funcție J modificată, trebuie ca $W^{[l]} \approx 0$. Astfel, o parte din neuroni au activări mici, creându-se un model mai simplu, ce nu descrie perfect datele de antrenament, dar care are o capacitate de generalizare mai mare.

2. **Dropout**:

   Vom fixa un hyper-parametru $keep\_prob \in [0, 1]$. În fiecare iterație din procesul de antrenare, vom asocia aleatoriu fiecărui neuron o valoare în intervalul $[0, 1]$. Dacă acestă valoare depășește $keep\_prob$, nu vom lua în calcul neuronul respectiv în iterația curentă, setându-i temporar activarea pe 0.



(a) Standard Neural Net (b) After applying dropout.

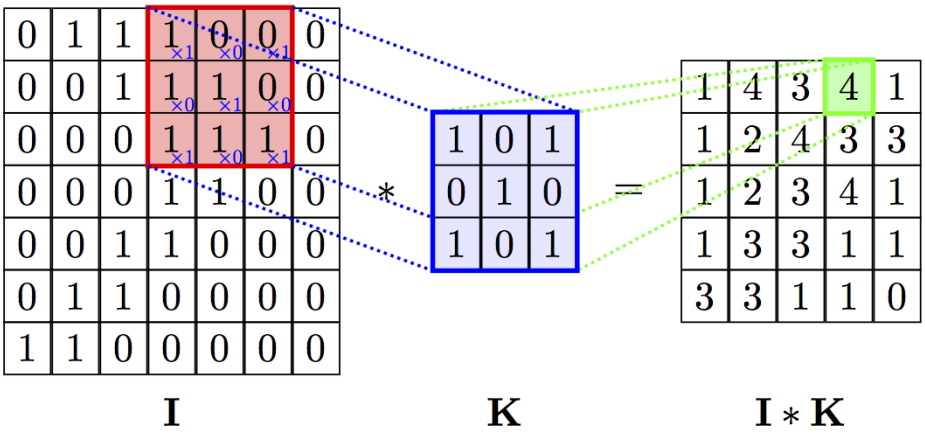
   Efectul acestei metode este de asemenea un model mai simplu care are capacitatea de generalizare mai mare decât a celui ce descrie perfect datele de antrenament.

4 Rețele neuronale convoluționale (CNNs)

Motivația din spatele acestor tipuri de rețele a apărut atunci când cercetătorii au încercat să aplice tehnicile clasice de deep learning în domeniul computer vision. Cel mai mare impediment a fost lipsa puterii computaționale. Spre exemplu, pentru o imagine RGB cu dimensiunile $32x32x3$ avem un vector de intrare cu 3072 de componente, iar dacă primul layer din rețea are 3500 de neuroni, prima matrice de parametri va avea forma $3500x3072$, adică peste 10 milioane de parametrii de optimizat numai pentru primul layer. Este evident faptul că, pentru ca deep learning să fie aplicat în recunoașterea imaginilor, ne trebuie o altă abordare.

4.1 Operația de convoluție

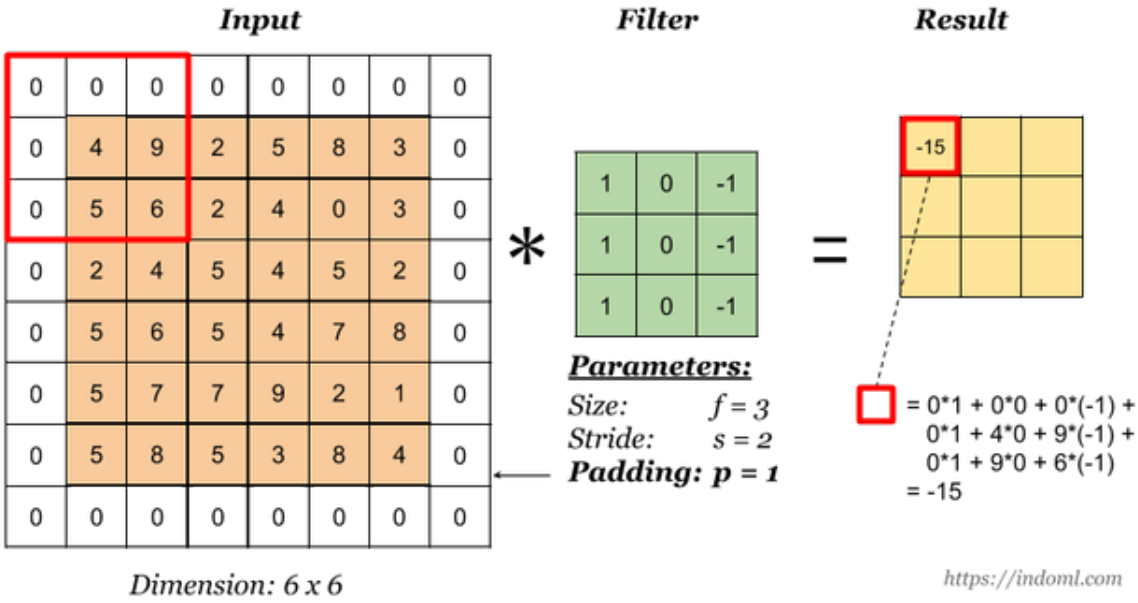
Fie 2 matrici, I matricea de pixeli corespunzătoare unei imagini și K o matrice $kernel(filter)$. Operația de convoluție având ca termeni I și K arată în felul următor:



Fie I de forma (n,n) și K de forma (f,f). Matricea rezultat va avea forma (n-f+1,n-f+1)

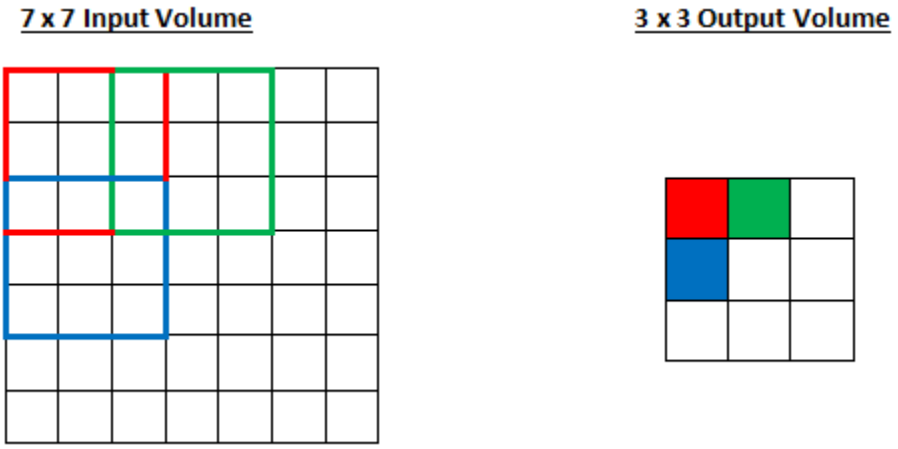
4.1.1 Padding și Striding

Padging-ul a venit ca o soluție pentru cazul în care filter-ul iese în afara imaginii în apropierea marginilor acesteia. Astfel se va completa matricea inițială cu zero-uri înaintea efectuării operației propriu-zise.



Această tehnică modifică regula dimensiunii output-ului. Pentru un padding de dimensiune p vom aveaun output de forma $(n+2p-f+1, n+2p-f+1)$. Observăm că operația de convoluție clasică (fără padding) micsorează imaginea, însă, dacă vrem să evitam acest lucru, putem alege un padding $p = \dfrac{f-1}{2}$.

Stridding-ul se referă la numărul de unități cu care vom mișca filter-ul pantru a scana întreaga imagine. Până acum am folosit un stride $s = 1$.

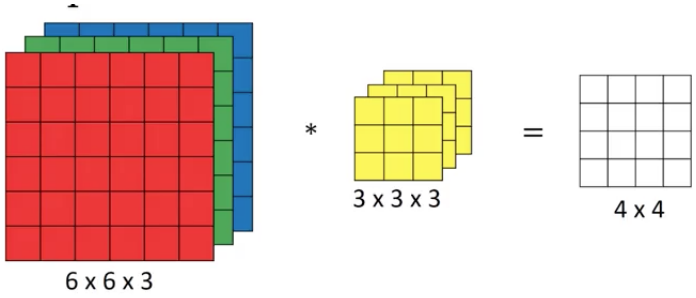


Operație de convoluție cu stride-ul $s = 2$

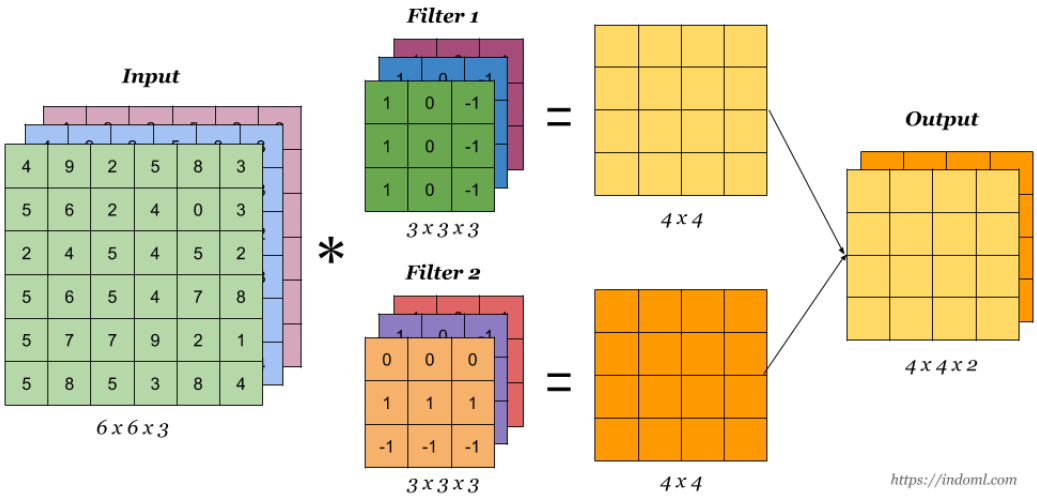
Scopul acestei tehnici este să reducă numărul de operații având ca compromis, o mică pierdere de informație. Astfel output-ul obținut va avea forma $\left(\left[\dfrac{n+2p-f}{s}\right]+1, \left[\dfrac{n+2p-f}{s}\right]+1\right)$.

4.1.2 Convoluție peste volume

Deocamdată, am vorbit despre operația de convoluție pe matrici de pixeli de forma (n, m) ce reprezintă imagini *greyscale*. în contextul imaginilor RGB, matricile lor au forma $(n, m, 3)$, iar pentru a putea efectua operația, vom copia filter-ul de 3 ori, formând o matrice de forma $(f, f, 3)$. Pentru fiecare channel, vom lua filter-ul si vom face calculele ca înainte, obtinând 3 numere pentru fiecare poziție din output. La final vom aduna aceste 3 numere pentru a calcula o componentă a output-ului, forma acestuia rămânând aceeași ca în cazul imaginilor cu un singur channel.



După ce am studiat ce este un filter și operația de convoluție, putem defini un layer într-o rețea neuronală convoluțională ca o multitudine de filtere și de operații de convoluție efectuate pe o matrice de pixeli. Aceste operații sunt independente, însă output-ul lor este combinat la final în următorul mod:

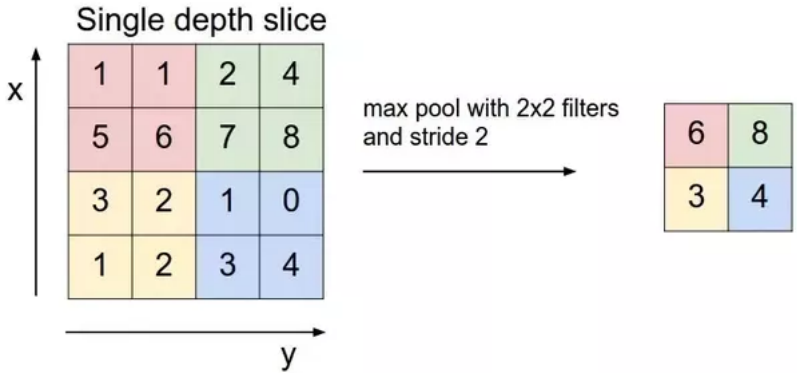


Un layer cu 2 filtere

Vom compara această abordare cu cea a rețelelor neuronale clasice. Având 2 filtere, vom lua cazul clasic a unui layer cu 2 neuroni obișnuiți. În contextul clasic, având un input de forma $(6, 6, 3)$ și 2 neuroni, matricea de parametrii va avea forma $(2, 6*6*3)$, în total 216 de parametrii. În cazul rețelelor convoluționale, avem 2 filtere de forma $(3, 3, 3)$, în total 54 de parametrii, făcând aceste rețele mult mai viabile pentru prelucrarea imaginilor de rezoluție înaltă.

4.1.3 Max pooling

Max pooling este un pas intermediar efectuat între layer-ele de convoluție, în special pentru a reduce dimensiunea vectorilor și complexitatea calculelor. Ca și în cazul operației de convoluție, max pooling are 3 hyperparametrii ($filterSize\, f, padding\, p, stride\, s$) și formula formei output-ului este aceeași $\left(\left[\frac{n+2p-f}{s}\right]+1, \left[\frac{n+2p-f}{s}\right]+1\right)$.



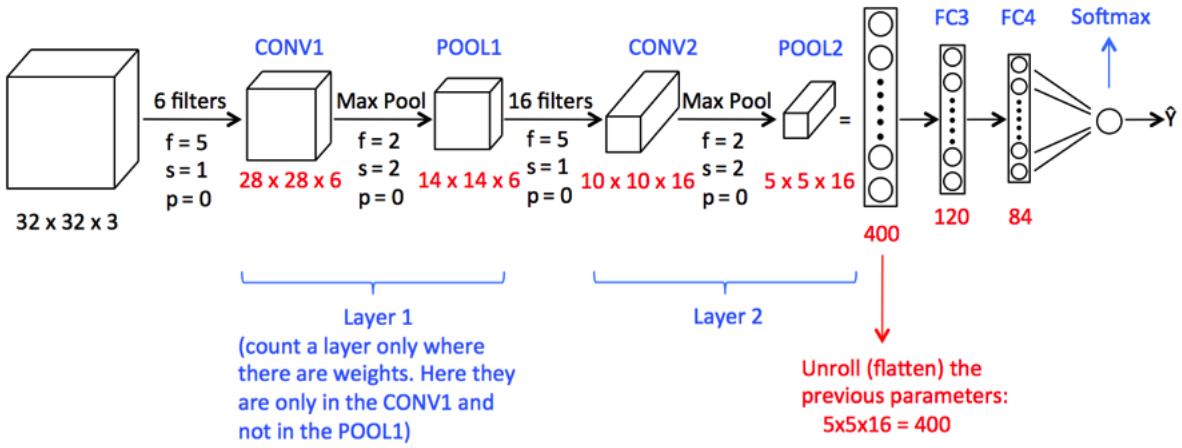
Este de observat că, spre deosebire de un layer obișnuit, cel de max pooling nu prezintă parametrii ce trebuie învățați, fiind o funcție fixată. În literatura de specialitate mai este întâlnit și *AveragePooling* în care nu luăm componenta maximă, ci facem media tuturor componentelor.

4.2 Arhitecturi celebre

O rețea neuronală convoluțională este o succesiune de layere de convoluție, intercalate de cele de max pooling. În practică este greu să găsim o configurație a acestor layere care să ofere cea mai bună performanță. Din această cauză, un punct bun de plecare este reprezentat de arhitecturile celebre consactare in literatura de specialitate.

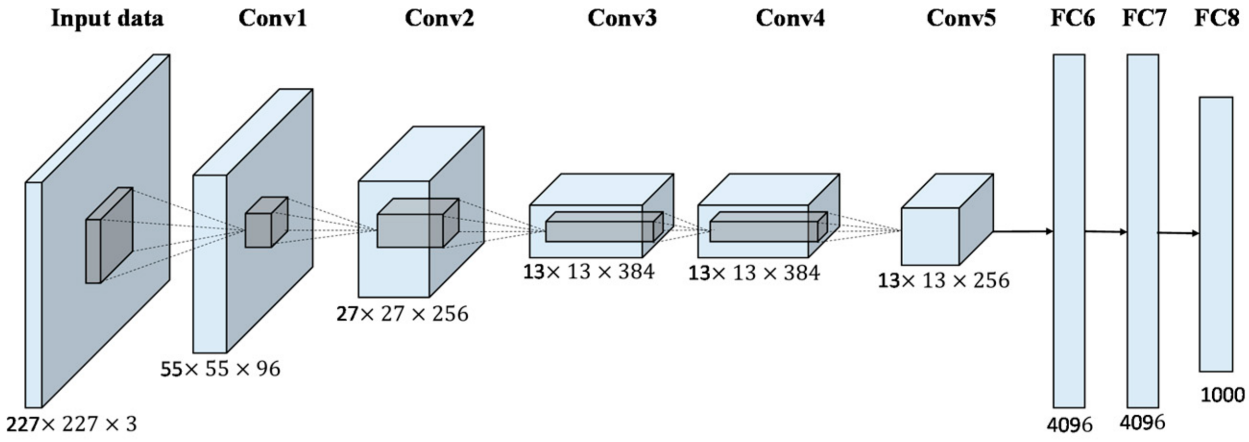
4.2.1 LeNet-5

LeNet-5 a fost prima rețea neuronală convoluțională cu rezultate notabile și folosită în afara unor experimente. Ea a fost folosită în anii 90 cu succes la recunoașterea cifrelor și a literelor de mână date ca input sub forma unei imagini RGB de rezoluție 32x32. Mai jos este arhitectura rețelei cu detalii despre hyperparametrii.



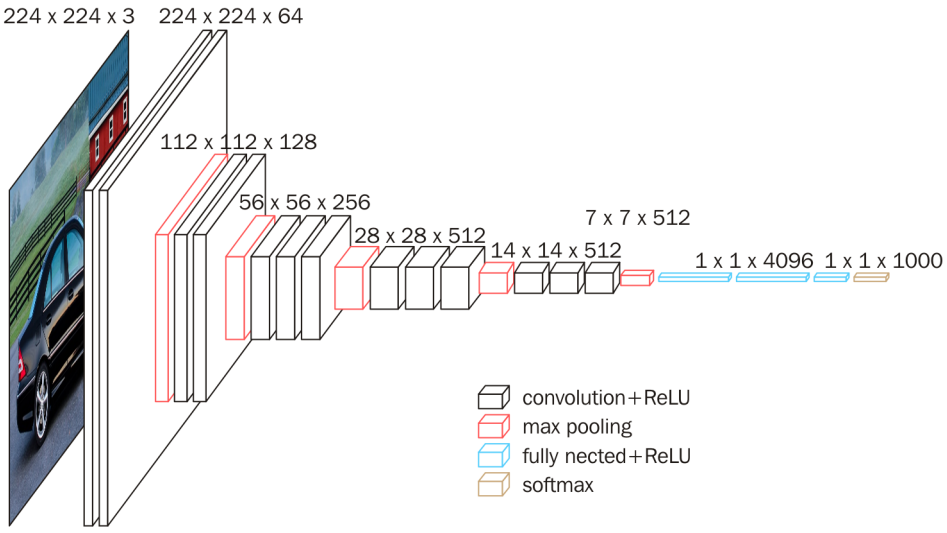
4.2.2 AlexNet

AlexNet este una dintre cele mai importante rețele neuronale convoluționale deoarece, în anul 2012, a reușit să aducă rata de eroare în cadrul competiției ImageNet LSVRC-2010 de la 25% la 16%, deschizând drumul tehnicilor de deep learning în domeniul Computer Vision. Arhitectura folosită este asemănătoare rețelei LeNet-5, folosind tot o secvență liniară de layere convoluționale intercalate cu cele de max pooling.



4.2.3 VGG-16

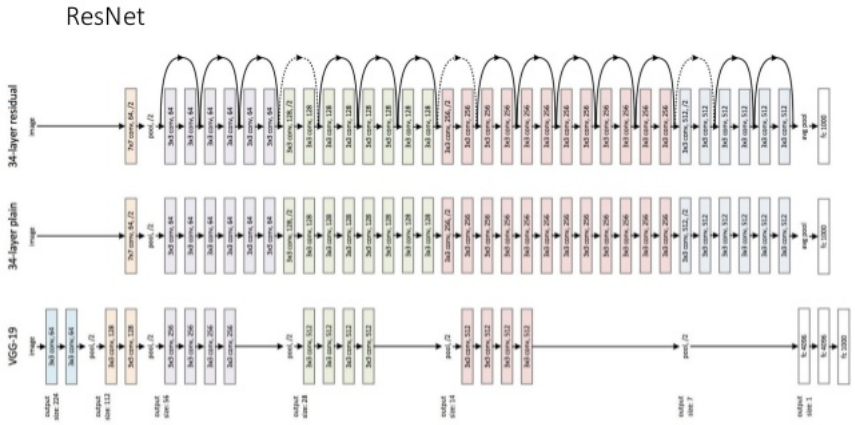
Această rețea a fost creată de *VisualGeometryGroup* din cadrul universității Oxford în anul 2014. VGG-16 a obținut rezultate foarte bune în cadrul competiției ImageNet LSVRC-2010, reușind să depășească performanța oferită de AlexNet. Rețeaua a ieșit în evidență prin numărul foarte mare de parametrii ce erau învățați ($\approx 100\,mil$), acesta fiind și principalul motiv pentru îmbunătățirile aduse.



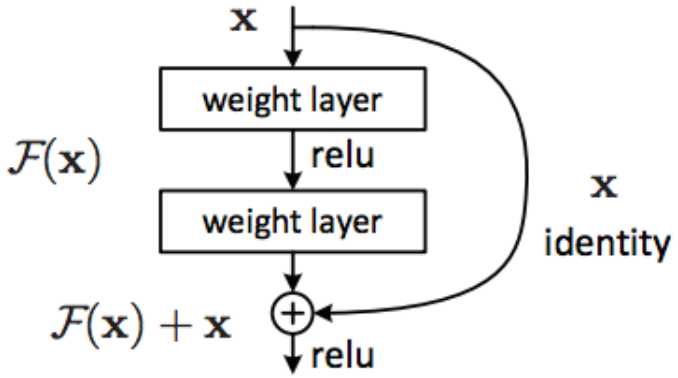
Arhitectura VGG-16

4.2.4 ResNet

Modelul ResNet a fost introdus în 2015 de cei de la Microsoft și ideea din spate este foarte simplă: *Deeper is better*. Văzând îmbunătățirile aduse din VGG-16 prin simplul fapt că avea foarte mulți parametrii, cercetătorii care au creat ResNet au dorit să creeze o rețea cât mai mare în adâncime cu speranță că performanța va crește. O mare problemă a fost complexitatea calculelor pentru antrenarea unei astfel de rețele, fiind necesar să recurgă la o tehnică numită *Residuallearningblock* pe care o vom descrie imediat. Asrfel, ResNet a putut fi antrenat, având diferite variante, în funcție de numărul de layere: ResNet-34, ResNet-50, ResNet-152.



ResNet-34 (sus) în comparație cu un CNN obișnuit și cu VGG-19 (o variantă extinsă a VGG-16)



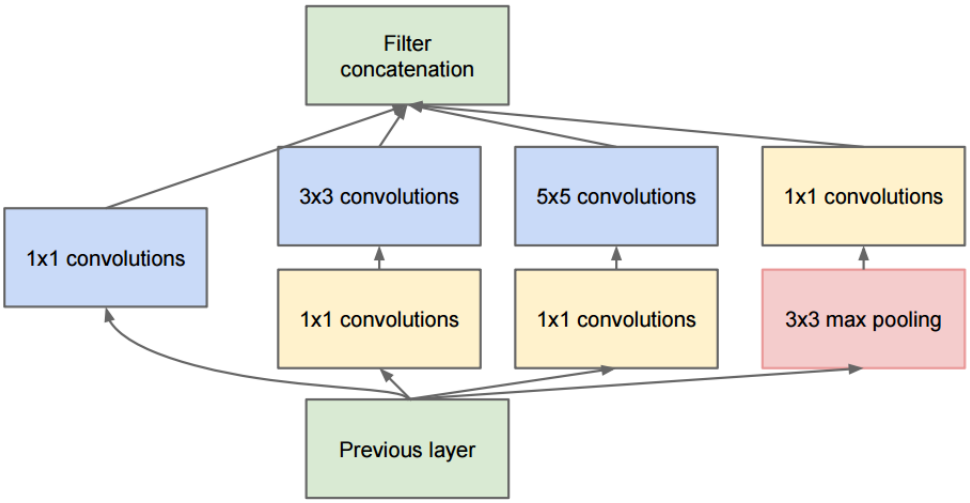
Residual Block

Acest bloc este, în mare parte, o construcție obișnuită dintr-un CNN, exceptând acea conecțiune (skip connection) ce sare cele 2 layere. La un nivel înalt, având o înșiruire de blocuri de acest tip, orice layer ce nu aduce informații utile, poate fi evitat, fiind foarte ușor pentru un algoritm de tipul *Gradient Descent* să învețe funcția identitate într-un astfel de context. Datorită acestui fapt, putem crea rețele neuronale convoluționale cu peste 150 de layere și procesul de antrenament să rămână viabil în continuare.

4.2.5 Inception

Dacă cei de la Microsoft au mers pe ideea crearii unei rețele cu foarte multe layere, în 2014, cei de la Google au propus o extindere a modelului clasic prin mărimea lățimii rețelei,

construind ceea ce se numește în literatura de specialitate *Inception Block*.

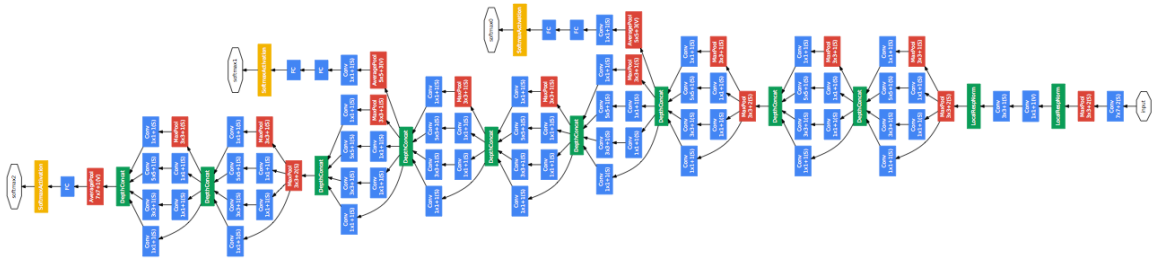


Inception Block

Principiul de bază este următorul:

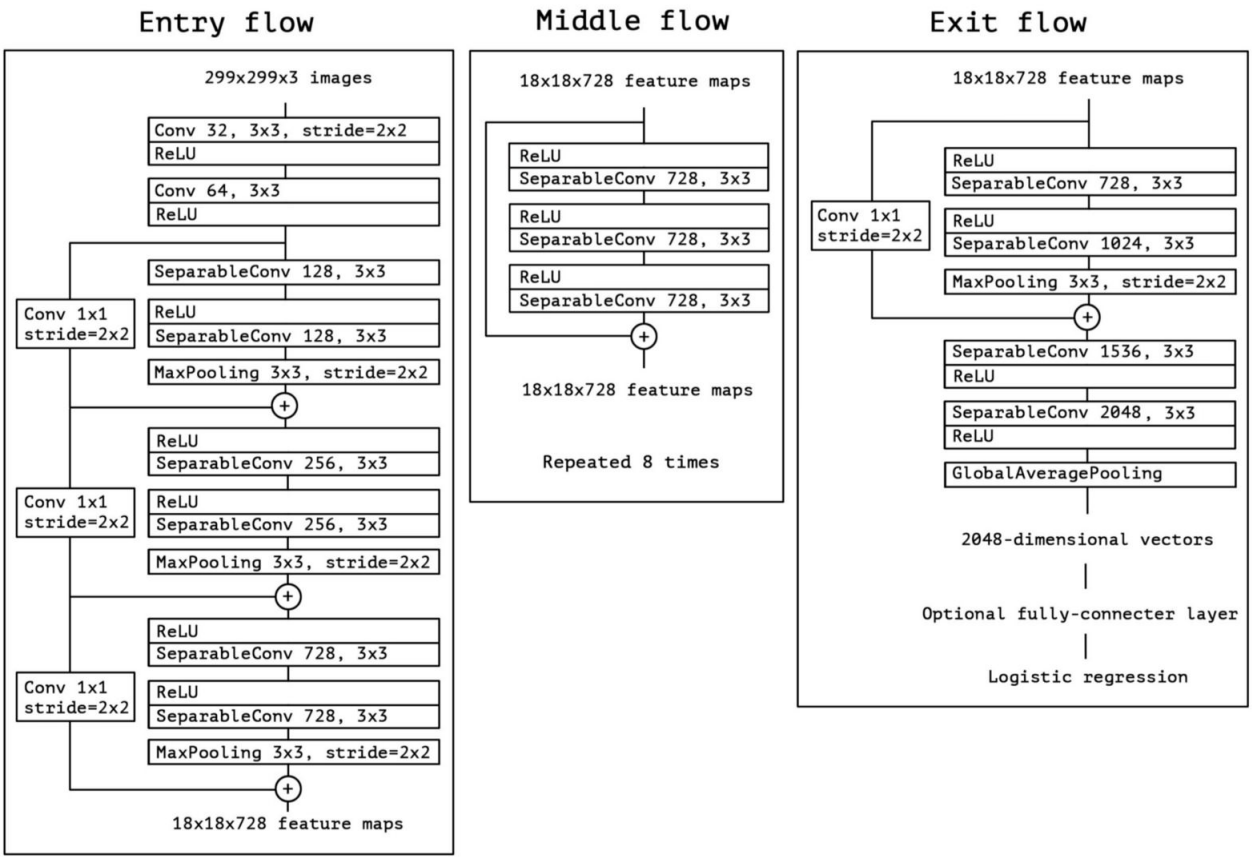
- La un moment dat în construcția arhitecturii putem să alegem dintr-o mulțime de operații (convolutie cu un filter de forma 1x1, cu un filter de forma 3x3, max pooling, etc...) și este dificil să stim anterior care ne aduce performanța cea mai bună. Într-un Inception Block se efectuează toate aceste operații și se concateneaza output-ul la final.

O secventă în cascadă de Inception Blocks reprezintă o rețea convoluțională de tip Inception.



Diferite modificări au fost aduse acestui model, cel din 2014 putând numele de Inception-V1, iar acum existând și Inception-V4.

4.2.6 Xception



4.3 Transfer Learning

Transfer learning se referă la faptul că nu tot ce a învățăt o rețea neuronală pentru o anumită problemă devine inutil când vrem să dăm aceleași rețele o nouă problemă. Neavând resurse coputaționale foarte mari, acestă tehnică este o componentă esențială în cadrul proiectului meu. Astfel, folosind un model deja antrenat (Xception) pe datele din competiția ImageNet LSVRC-2010, am extras vectori caracteristici pentru fiecare imagine din setul meu de date, nemaiantrenând modelul. Am avut succes deoarece imaginile din cadrul ImageNet, majoritatea fiind obiecte obișnuite din jurul nostru, nu diferă foarte mult de ceea ce se gasește într-o fotografie făcută într-un restaurant. Ar fi fost imposibil pentru mine să o iau de la 0, tinând cont că modelele descrise anterior au fost antrenate în cloud, chiar și timp de câteva săptămâni fără oprire.

Bibliografie

[1] Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton *ImageNet Classification with Deep Convolutional Neural Networks* 2012.

[2] Yann LeChun, Leon Bottou, Yoshua Bengio, Patrick Haffner *Gradient-Based Learning Applied to Document Recognition* 1998.

[3] Jaume Amores *Multiple Instance Classification: review, taxonomy and comparative study* Computer Vision Center, Computer Science Department, UAB, Spain 2013.

[4] Karen Simonyan, Andrew Zisserman *Very deep convolutional networks for large-scale image recognition* 2015.

[5] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun *Deep Residual Learning for Image Recognition*
Microsoft Research, 2015.

[6] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabinovich *Going deeper with convolutions* 2014.

[7] Francois Chollet *Xception: Deep Learning with Depthwise Separable Convolutions* 2017.

[8] Lars Hertel, Erhardt Barth, Thomas Käster, Thomas Martinetz *Deep Convolutional Neural Networks as Generic Feature Extractors*
Institute for Signal Processing, University of Luebeck, Germany
Institute for Neuro- and Bioinformatics, University of Luebeck, Germany.

[9] Ben Athiwaratkun, Keegan Kang *Feature Representation In Convolutional Neural Networks*
Department of Statistical Science Cornell University 2015.

[10] Diederik P. Kingma, Jimmy Lei Ba, *ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION*

University of Amsterdam, OpenAI 2017

University of Toronto 2017.

[11] Geoffrey Hinton *Neural Networks for Machine Learning: Lecture 6a Overview of mini-batch gradient descent*

[12] `https://www.coursera.org/learn/neural-networks-deep-learning`

[13] `https://www.kaggle.com/c/yelp-restaurant-photo-classification`