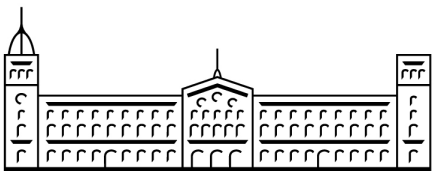
Treball de fi de Grau

Estudi de les xarxes neuronals convolucionals profundes mitjançant Caffe



Axel Brando Guillaumes

GRAU D'ENGINYERIA INFORMÀTICA

Facultat de Matemàtiques

Universitat de Barcelona

Director: Jordi Vitrià Marca

Departament de Matemàtica Aplicada i Anàlisi. UB

Barcelona, 23 de gener de 2014

Abstract

The deep learning techniques and the use of GPUs have made neural networks the leading option for solving some computational problems and it has been shown to produce the state-of-the-art results in many fields like computer vision, automatic speech recognition, natural language processing, and audio recognition.

This final grade dissertation is divided into three parts: first, we will discuss theoretical concepts that allow us to understand how neural networks work. Second, we will focus in deep convolutional neural network to understand and learn how to build such networks that Caffe Framework use. Finally, the third part will be a compilation of all learned skills to make that a neural network will classify correctly the data set MNIST, and then we will change Caffe Framework files so it can read images with more channels and we will watch the results obtained. At last, we will match and improve the public state-of-the-art classification system of the data set Food-101.

After all the work, our goals will be achieved: we will modify Caffe Framework and we will check that in the case of the MNIST we will improves the classification rate. But above all, the result that we want to emphasize is the to release of classification system for the Food-101 that will improve the accuracy from 56.40% to 74.6834% and finally we will propose ideas for improving this classification in the future.

Índex

Capítol 1

Introducció, antecedents, estat actual i objectius

Recentment, amb les tècniques d'aprenentatge profund, les xarxes neuronals han tornat a ser punteres i han renovat l'interès pel camp anomenat aprenentatge automàtic dins de la intel·ligència artificial. Gràcies a l'ús de les GPUs i les noves tècniques, les xarxes neuronals han aconseguit fer que els processos d'aprenentatge convergeixin i donin solucions a problemes que milloren l'estat de l'art actual en molts camps com la visió per computador, el reconeixement automàtic de la parla, el processament de llenguatge natural i, fins i tot, el reconeixement de so.

1.1 Alguns antecedents

Tal com diu Piccinini [2004], l'any 1943 existia una comunitat de biofísics i filòsofs realitzant avenços matemàtics en les xarxes neuronals. Cal destacar que, aquell any, Warren McCulloch i Walter Pitts van presentar un article (McCulloch and Pitts [1943]) on plantejaven un model lògic i computacional per tal d'entendre l'activitat neuronal i mental. Aquesta contribució portava una generalització de la noció d'autòmat finit i es pot considerar una de les primeres teories modernes del funcionament computacional de la ment i del cervell.

Des del laboratori aeronàutic de Cornell, Frank Rosenblatt va idear entre els

anys 1957 amb Rosenblantt [1957] i 1958 amb Rosenblatt [1958] el perceptró, un tipus de neurona articial amb la intenció de reconèixer molts tipus de patrons. Però Minsky and Papert [1969] van demostrar que era impossible que un tipus de xarxa construïda amb perceptrons aprengués una funció del tipus XOR i, per tant, es va extendre la idea que les xarxes neuronals no eren una metodologia tan genèrica per resoldre problemes com s'havia promès. Amb la mateixa força que van aparèixer van quedar arraconades en poc temps. La solució a aquest problema va venir quan es van considerar xarxes de perceptrons amb múltiples capes.

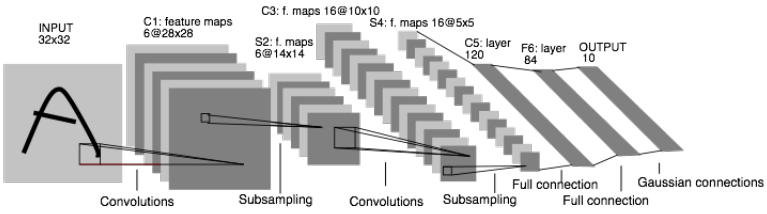


Figura 1.1: Model de capes convolucionals de LeNet del 1989 per tal d'identificar nombres escrits a mà. Imatge extreta de Yangqing Jia [2014].

L'any 1980, a partir de l'article Fukushima [1980], s'introdueix un nou tipus de xarxa neuronal artificial anomenada xarxa neuronal convolucional que s'inspirava en els processos biològics fins aleshores coneguts. El nou tipus de xarxes estaven basades en estructures de perceptrons de múltiples capes. Posteriorment, el seu disseny va ser millorat per LeCun et al. [1998], generalitzat per Behnke [2003] i simplificat per Simard et al. [2003].

Però no des del primer moment les xarxes neuronals han estat tema principal d'estudi. Sobretot entre els anys 1990 i 2000 l'ús de les xarxes neuronals en molts camps va disminuir en favor d'altres models com les support vector machines (SVMs) o els Random Forest. Això va ser degut a que durant els anys anteriors, entre els 1980 i 1990, els investigadors van intentar usar l'*stochastic gradient descent* i *backpropagation* per entrenar xarxes que contenien més d'una capa oculta. Desafortunadament, amb excepció d'algunes arquitectures molt concretes, això no va tenir bons resultats ja que l'aprenentatge de les xarxes era massa lent.

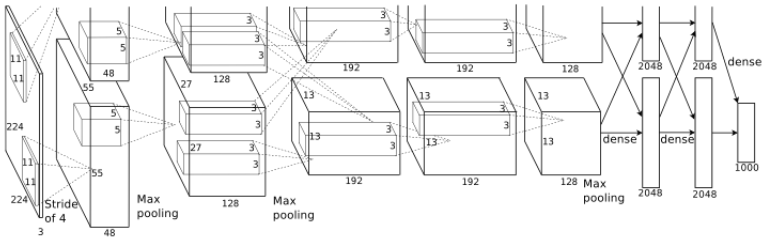


Figura 1.2: Model de capes convolucionals de ILSVRC12 amb moltes més dades, usant GPU, regularització. Imatge extreta de Yangqing Jia [2014]

Tot i això, cap a l'any 2006 es van desenvolupar una serie de tècniques que van permetre que xarxes neuronals més complexes que les de només una capa oculta aprenguessin. Aquestes tècniques continuaven basant-se en l'*stochastic gradient descent* i *backpropagation* però introduïen noves idees relacionades amb la regularització. Posteriorment, l'any 2007, Geoffrey E. Hinton publica un article, Hinton [2007], on es mostra com les limitacions de l'aprenentatge basat amb el *backpropagation* són superades fent ús de *multilayers neural networks* tractades cada capa com una *restricted Boltzmann machine* i, posteriorment, usant *backpropagation* pel *Fine-tuning*.

Entre totes les tècniques, s'ha de destacar els últims avenços en l'ús de les Unitats de Procés Gràfic (GPUs) que han ajudat molt a la renovació de l'interès pel *deep learning*, transformant temps d'aprenentatge de setmanes a dies i fent viables aquests aprenentatges.

1.2 L'estat actual de l'aprenentatge automàtic

Com podem veure a la Figura 1.3, actualment hi han molts mètodes d'aprenentatge automàtic. Des de supervisats (i que, per tant, esperem obtenir unes classes concretes) com les xarxes neuronals convolucionals, recurrents, les SVM, etc. com no supervisats (i que s'espera obtenir agrupacions de manera que després es pugui trobar relacions interessants) com les Deep Belief Networks, les Restricted Boltzmann machine, Sparse Coding,etc.

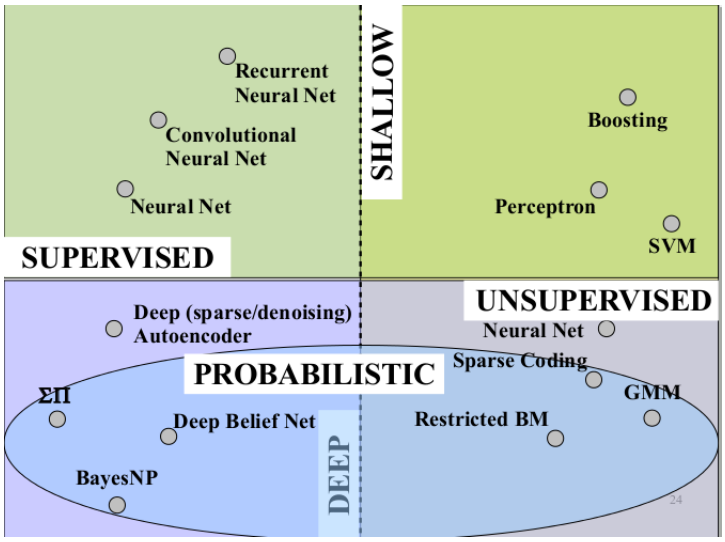


Figura 1.3: Alguns mètodes d'aprenentatge automàtic més usats. Extret de Ranzato [2014]

L'ús d'un mètode o altre és restringeix al problema que s'intenta resoldre i, per tant, quan ens trobem davant un problema del que coneixem les diferents classes que podem obtenir i quan el nombre de dades és gran, les xarxes neuronals profundes són una opció a tenir en compte.

Però, què són exactament les xarxes neuronals profundes? Per tal d'entendre aquest concepte entendrem primer el de xarxa neuronal simple i com funciona aquesta per, posteriorment, poder veure què passa quan afegim més capes.

1.3 Els objectius del treball

En aquest treball ens plantejarem els següents objectius: primerament, desenvoluparem la teoria necessària per tal d'entendre què són les xarxes neuronals profundes i com funcionen. Seguidament, introduirem un tipus d'arquitectura d'aprenentatge profund anomenada xarxes neuronals convolucionals profundes, que ens permetran entendre com funciona i com usar el *Framework* Caffe. Posteriorment, aprofundirem en l'estudi d'aquest *Framework* per tal de resoldre dos problemes concrets. Primer, entrenarem una xarxa neuronal per tal d'aprendre

a classificar una base de dades de dígits escrits a mà que ja té una bona ràtio de classificació. I segon, aplicarem algunes tècniques, que explicarem durant el treball, i modificarem el funcionament del *Framework* per tal d'intentar millorar l'estat de l'art actual en un problema més complexe.

Capítol 2

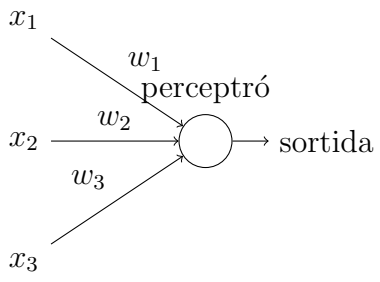
Xarxes neuronals

Per tal d'entendre que és una xarxa neuronal començarem explicant un tipus de neurona aritifical anomenada perceptró.

2.1 Perceptrons

A partir del treball fet prèviament per Warren McCulloch i Walter Pitts, Rosenblatt [1958] desenvolupa una primera teoria sobre el que serà el perceptró entenent-lo com un model per tal d'imitar la capacitat del cervell de reconèixer i discriminar.

El funcionament d'un perceptró és el següent.



Obté unes entrades binaries $x_1, x_2, x_3, \ldots$ i produeix a la sortida una única sortida binaria. Rosenblatt va introduir el concepte de pesos w_j, uns nombres reals associat a cada una de les entrades que expressaven la importància de cada

una respecte la sortida. Per tal de saber el valor binari de la sortida, aquesta es regia segons si la suma del producte de pesos per la seva respectiva entrada era més gran o no que un cert llindar establert, i.e.

$$\text{sortida} = \begin{cases} 0 & \text{Si } \sum_j w_j x_j \leq \text{llindar} \\ 1 & \text{Si } \sum_j w_j x_j > \text{llindar} \end{cases}$$

Però, tal i com diu Nielsen [2015], en la majoria de llibres s'agrupen tots els pesos i valors en un vector $\mathbf{w}$ i $\mathbf{x}$ respectivament i consideren el producte escalar $\mathbf{w} \cdot \mathbf{x} \equiv \sum_j w_j x_j$. Per altra banda, no es parla de llindar sinó que el llindar es mou a l'altra banda de l'inequació, reemplaçant aquest valor pel que es coneix com el biax del perceptró, θ, és a dir, $\theta \equiv -\text{llindar}$. El funcionament del perceptró es reescriu com:

$$\text{sortida} = \begin{cases} 0 & \text{Si } \mathbf{w} \cdot \mathbf{x} + \theta \leq 0 \\ 1 & \text{Si } \mathbf{w} \cdot \mathbf{x} + \theta > 0 \end{cases} \tag{2.1}$$

En aquest cas, podem pensar amb el biaix com en una mesura del fàcil que és aconseguir que s'activi la sortida del perceptró (o un 1).

L'objectiu serà desenvolupar algoritmes que modifiquin de forma automàtica els pesos i biaixos de la xarxa neuronal de forma que aquesta obtingui el resultat que ens interessa donada una entrada concreta. Aquests algoritmes respondran a estímuls exteriors sense la intervenció directe del programador.

2.2 Neurones sigmoidees

El problema que ens trobem amb els perceptrons, tal i com podem observar a la Figura 2.1, és que petits canvis en els pesos o biaxos poden produir grans canvis a la sortida, ja que la sortida només serà 1 o 0 depenent si $\mathbf{w} \cdot \mathbf{x} + \theta$ és positiu o no. Introduïm ara un nou tipus de neurona artificial, molt similar al perceptró però que té l'objectiu de resoldre aquest problema. Aquest canvi permetrà que una xarxa formada per neurones sigmoidees pugui aprendre.

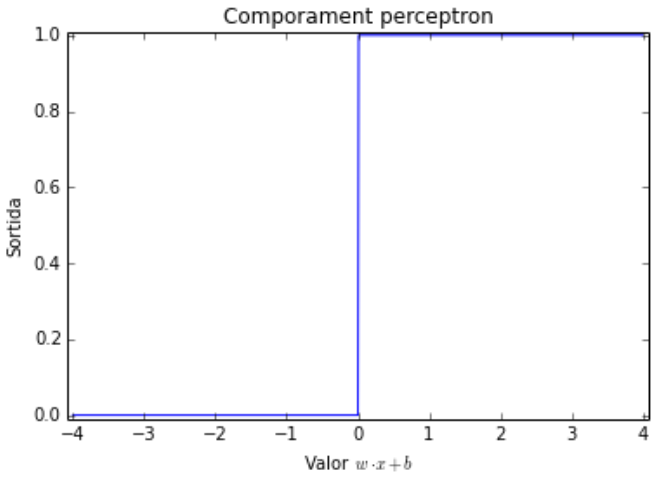


Figura 2.1: Representació funció perceptró

Com en el cas del perceptró, les neurones sigmoidees tindran unes entrades $x_1, x_2, x_3, \ldots$ i produiran una sortida, aquest cop però no serà un valor binari sinó que podrà prendre valors reals entre 0 i 1. Sigui $z \equiv \mathbf{w} \cdot \mathbf{x} + \theta$, seguint la notació del perceptró, la sortida serà la funció sigmoidea σ i la definirem com

$$\sigma(z) \equiv \frac{1}{1 + e^{-z}} \in [0, 1] \tag{2.2}$$

El motiu d'usar aquesta funció en front d'altres és pels avantatges algebraics que ens atorga, com veurem en apartats posteriors. A més, com en el cas de la funció d'activació que regeix els perceptrons, la funció sigmoidea compleix que si

- Suposem que z és un nombre molt gran positiu, aleshores $e^{-z} \approx 0$ i, per tant, la sortida $\sigma(z) \approx 1$.
- Suposem, per altra banda, que z és un nombre negatiu molt gran, aleshores $e^{-z} \to +\infty$ i, per tant, la sortida $\sigma(z) \to 0$.

A la Figura 2.2 representem la funció sigmoidea, podem veure que a petits canvis en pesos i biaixos els canvis són suaus en comparació amb els de la funció del perceptró, i.e.

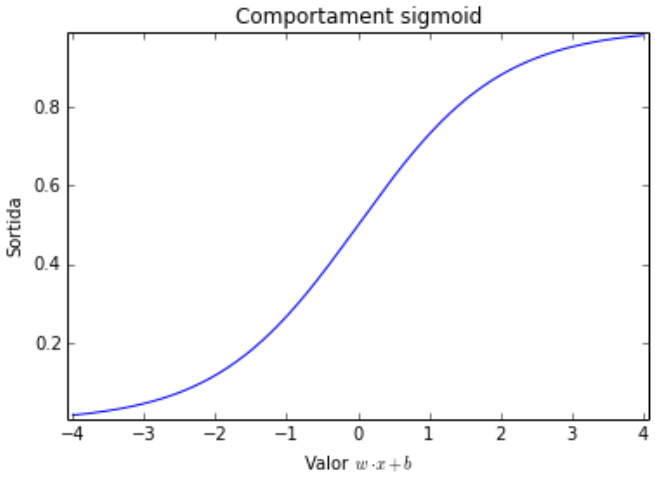


Figura 2.2: Representació funció sigmoidea

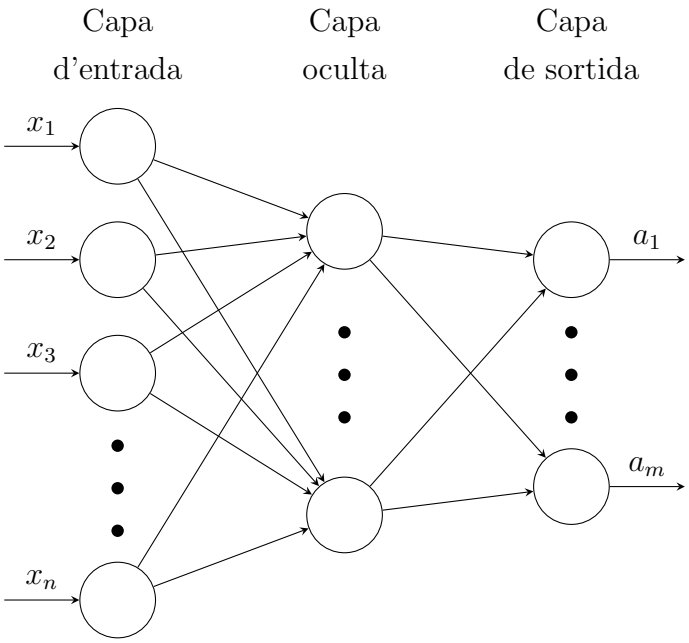
$$\triangle \text{ sortida} \approx \sum_j \frac{\partial \text{ sortida}}{\partial w_j} \triangle w_j + \frac{\partial \text{ sortida}}{\partial \theta} \triangle \theta$$

I en aquest cas, ens interessa molt veure que la variació de la sortida es comportarà de forma lineal als canvis dels pesos i biaixos ja que això ens ajudarà a identificar més fàcilment com canviar aquests valors per obtenir la sortida desitjada. A més, les neurones sigmoidees en contraposició dels perceptrons, contemplaran situacions en les quals es tenen algunes evidències de que una cosa sigui, sense haver d'afirmar que ho és o no, ja que podrem adjudicar com a sortida qualsevol valor real de 0 a 1.

2.3 Arquitectura d'una xarxa neuronal

Abans de continuar explicant com realitzar aquest aprenentatge, introduïm uns conceptes bàsics per tal d'entendre les parts d'una xarxa neuronal de forma genèrica.

2.3.1 Disseny xarxa neuronal



Una xarxa neuronal està dividida en capes. Aquestes capes són agrupacions de neurones situades a la mateixa columna.

- La capa situada més a la esquerra és la capa d'entrada. Aquesta capa és la primera i obtindrà la informació de l'exterior.
- La capa més a la dreta és la capa de sortida. Aquesta capa pot tenir una o diverses neurones; en pot tenir tantes com en nombre de classes es vulguin classificar la informació que ens introduiran. Per exemple, en el cas de la classificació dels nombres escrits a mà a partir de la base de dades MNIST (Yann LeCun and Burges [1998]) el nombre de neurones de sortida podrien ser tantes com dígits hi han al nostre sistema numèric, és a dir, 10.
- Les capes ocultes correspondran a totes les capes intermèdies. En el cas de la xarxa neuronal anterior, tenim només una capa oculta però n'hi poden haver tantes com es vulguin. L'interessant d'aquestes capes serà com es realitzaran les connexions de les neurones entre una capa i la següent, de forma que quan s'aprenguin els pesos i biaixos doni bons resultats al final la classificació.

Tot i que el disseny de la capa d'entrada i sortida és normalment senzill, realitzar el disseny de les capes ocultes i connexions entre elles actualment és un art ja que, per ara, no és tenen grans regles per tal de tenir criteris fiables per a la construcció. Molts investigadors dissenyen heurístiques per a ajudar a aquest procés; per exemple, algunes heurístiques serveixen per a determinar fins a quantes capes ocultes introduir tenint en compte l'increment de temps d'entrenament de la xarxa que comportarà. Com a exemples tenim els articles Islam and Murase [2001] que introdueixen nous algoritmes com el CNNDA (cascade neural network design algorithm) per tal de dissenyar xarxes neuronals o Leung et al. [2003], Tsai et al. [2006] que intenten donar resposta sobre quins paràmetres escollir més adequats per tal d'entrenar la xarxa usant en aquest cas algoritmes genètics.

De la mateixa manera que en el sistema neurològic, no totes les xarxes neuronals compleixen que la sortida d'una capa és entrada de la següent. Aquest tipus de xarxes se les anomena en anglès *feedforward neural networks*, això vol dir que la informació sempre avança i mai es retroalimenta enrere, per tant, no hi ha bucles i la funció σ mai dependrà de la sortida. L'altre tipus de model de xarxa neuronal que contindrà bucles s'anomena **xarxes neuronals recurrents** i ens permetrà que una neurona s'activi per una duració limitada de temps abans d'inactivar-se; això podria provocar que altres neurones s'activéssin i produir així una cascada d'activacions. En aquest cas, els bucles no seran un problema perquè la sortida només afectarà a l'entrada passat un cert temps.

En aquest treball ens centrarem amb les *feedforward neural networks* perquè actualment els algoritmes d'aprenentatge generalment són més potents que respecte les *recurrent* [1]. Tot i això, el funcionament del nostre cervell és més semblant a les *recurrent*; per això no es pot descartar que en un futur aquestes es necessitin per solucionar problemes.

[1] Si el lector hi està interessat, es recomana la lectura de l'article Sundermeyer et al. [2013] on es fa una comparació dels dos models tractant el problema del reconeixement de veu.

2.3.2 L'aprenentatge mitjançant *gradient descent*

Una vegada tenim el disseny de la xarxa neuronal necessitem saber com realitzar el procés d'aprenentatge. Per tal de realitzar-ho, necessitarem dividir el nostre conjunt d'informació a classificar (el *data set*), en una part que es destinarà a l'entrenament (el *training set*) i una altra que serà per comprovar com de bé la nostre xarxa classifica (el *validation* i *test set*) .

Donades les entrades del *training set* en un vector $\mathbf{x}$, si $\mathbf{a}$ és el vector sortida després de passar per la xarxa i $\mathbf{y}(\mathbf{x})$ són els valors esperats, definirem una **funció de cost**[1] que ens permeti calcular l'error entre el valor obtingut i esperat per tal d'aproximar els pesos i biaixos més adequats. En aquest cas, usem la funció de cost quadràtic (també anomenat error quadràtic):

$$C(w,b) = \frac{1}{2}\sum_x \|\mathbf{y}(\mathbf{x}) - \mathbf{a}\|^2 \tag{2.3}$$

On w serà la col·lecció de pesos de tota la xarxa i θ els biaixos respectius. Aquesta funció tindrà una forma suau i això ens permetrà entendre com els petits canvis de pesos i biaixos influeixen. Si, per exemple, consideréssim una funció de cost que calculés el nombre d'ítems classificats de forma correcte, no passaria el mateix ja que, generalment, petits canvis en els pesos i biaixos no canviarien el nombre d'ítems correctament classificats i tornaríem a tenir un cas similar a quan comparàvem la forma de la funció dels perceptrons envers la funció sigmoidea.

L'objectiu, doncs, serà minimitzar aquesta funció de cost C i per això usarem l'algoritme *gradient descent* per a trobar els valors dels pesos w i biaixos θ que corresponguin a un mínim local. A partir de Nielsen [2015], recordem que l'algoritme consistia en, donada una possible solució $v = \begin{pmatrix} v_1 \\ v_2 \end{pmatrix}$ (imaginem que estem en dos dimensions, però és extensible a n), obteníem el gradient de la funció en aquest punt $\nabla C \equiv \begin{pmatrix} \frac{\partial C}{\partial v_1} \\ \frac{\partial C}{\partial v_2} \end{pmatrix}$ i ens movíem en la direcció del gradient, negativament, de forma repetida. És a dir, el pas de l'iteració k-éssima a la $k+1$ seria

[1] La funció de cost també és anomenada *LOSS function* o *objective function*. Durant el treball usarem les dues primeres terminologies.

$$v^{k+1} = v^k - \eta \cdot \nabla C$$

On η és un coeficient fixat que representa la taxa d'aprenentatge o *learning rate*, en anglès. Per a trobar-lo seleccionàvem un valor el suficientment petit com perquè $\nabla C \leq 0$, però a la vegada el suficientment gran com perquè l'algoritme no fos massa lent trobant el mínim.

Tot i això, per tal d'agilitzar el procés d'aprenentatge, normalment s'usa l'algoritme de *stochastic gradient descent* que consistirà en agafar una mostra aleatòria de mida definida $x_1, \ldots, x_m$ del conjunt d'aprenentatge (anomenada *mini-batch*) i calcular la mitjana dels gradients corresponents per aquest subconjunt suposant que $\sum_{k=1}^{m} \frac{\nabla C_k}{m} \approx \nabla C$. Posteriorment, el procès és el mateix que l'explicat anteriorment. De fet, el procediment conegut per aprenentatge *online*, *on-line* o incremental és quan $m = 1$ i, de fet, és semblant al que usem els humans.

Ens falta saber calcular el gradient de la funció de cost. Per tal de fer-ho usarem l'algoritme anomenat *backpropagation*.

2.4 L'algoritme de *backpropagation*

Tot i ser un algoritme introduït als anys 70, no és fins el 1986 quan es publica l'article Rumelhart et al. [1988] que el *backpropagation* (abreviació de 'propagació cap a enrere dels errors' en anglès) s'usa per solucionar problemes fins aquell moment sense solució.

En termes generals, el *backpropagation* és un algoritme dividit en dues fases. Primerament, es propagarà una entrada del conjunt d'entrenament des de la primera capa de la xarxa neuronal fins arribar a la sortida, es compararà amb la sortida desitjada i es calcularà l'error obtingut de la funció de cost C respecte tots els pesos $\frac{\partial C}{\partial w}$ i biaixos $\frac{\partial C}{\partial \theta}$ de la xarxa. Seguidament, es propagarà enrere els canvis que s'hagin de realitzar, partint de l'última capa, a cada una de les capes anteriors distribuint aquests canvis de forma proporcional a la col·laboració que hagin tingut cada una de les neurones per obtenir la sortida.

Per tal d'explicar més concretament l'algoritme, primer suposarem que la

funció de cost C es pot escriure com a suma de funcions de cost per cada una de les entrades $C = \sum_x C_x$ i, segon, que el cost pot ser escrit com una funció a partir de les sortides de la xarxa neuronal.

Sigui a_j^l la activació de la j-éssima neurona a la l-éssima capa, tenim que

$$a_j^l = \sigma(\sum_k w_{jk}^l a_k^{l-1} + \theta_j^l) = \sigma(\mathbf{w}^l \mathbf{a}^{l-1} + \theta^l)$$

on la suma respecte k és sobre totes les neurones de la capa $(l-1)$-éssima i la següent igualtat s'obté si considerem la notació vectorial compacta.

L'algoritme de *backpropagation* està basat en quatre equacions fonamentals. Aquestes usen el *producte matricial de Hadamard* (o de *Schur*) $A \circ B$ que correspon al producte de cada un dels elements a una posició concreta d'una matriu A amb el seu respectiu element a la mateixa posició de la segona matriu B de mateixes dimensions.

Sigui L l'última capa i sigui δ_j^l l'error de la neurona j a la capa l, $\delta_j^l \equiv \frac{\partial C}{\partial z_j^l}$.

- La primera equació fa referència a l'error en la capa de sortida, δ^L. Tenim,

$$\delta_j^L = \frac{\partial C}{\partial z_j^L} = \sum_k \frac{\partial C}{\partial a_k^L} \frac{\partial a_k^L}{\partial z_j^L}$$

  Aplicant la definició de l'error i usant la regla de la cadena, reexpressem les derivades en funció de l'activació de sortida. Com que $a_j^L = \sigma(z_j^L)$, el segon terme $\frac{\partial a_k^L}{\partial z_j^L} = \sigma'(\partial z_j^L)$. Obtenim que,

$$\delta_j^L = \sum_k \frac{\partial C}{\partial a_k^L} \sigma'(\partial z_j^L) = \nabla_a C \circ \sigma'(\partial \mathbf{z}^L) \tag{2.4}$$

  Hem escrit l'equació en forma matricial, on $\nabla_a C$ és el vector que conté les derivades parcials $\frac{\partial C}{\partial a_j^L}$ com a components.

- La segona equació serà per a calcular l'error δ^l en funció de l'error de la següent capa δ^{l+1}. Per trobar-la haurem de reescriure $\delta_j^l = \frac{\partial C}{\partial z_j^l}$ en termes de $\delta_k^{l+1} = \frac{\partial C}{\partial z_k^l}$. Per a fer-ho usarem la regla de la cadena,

$$\delta_j^l = \frac{\partial C}{\partial z_j^l} = \sum_k \frac{\partial C}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_j^l} = \sum_k \delta_k^{l+1} \frac{\partial z_k^{l+1}}{\partial z_j^l}$$

Com $z_k^{l+1} = \sum_j w_{kj}^{l+1} a_j^l = \sum_j w_{kj}^{l+1} \sigma(z_j^l)$, si calculem i substituïm la derivada respecte z_j^l obtenim

$$\delta_j^l = \sum_k \delta_k^{l+1} w_{kj}^{l+1} \sigma'(z_j^l) = ((\mathbf{w}^{l+1})^T \boldsymbol{\delta}^{l+1}) \circ \sigma'(\mathbf{z}^l) \tag{2.5}$$

On a l'última igualtat ho hem expressat en forma vectorial denotant $(w^{l+1})^T$ com el vector de pesos transposat de la capa $(l+1)$.

- La tercera equació ens permetrà relacionar la ràtio de canvi de cost respecte qualsevol biaix de la xarxa neuronal. Com $\frac{\partial z_j^l}{\partial \theta_j^l} = 1$, es té que,

$$\frac{\partial C}{\partial \theta_j^l} = \frac{\partial C}{\partial z_j^l} \frac{\partial z_j^l}{\partial \theta_j^l} = \frac{\partial C}{\partial z_j^l} = \delta_j^l \tag{2.6}$$

- Finalment, la quarta equació ens permetrà calcular el quocient entre el canvi de cost respecte qualsevol canvi a la xarxa. Aquesta serà,

$$\frac{\partial C}{\partial w_{jk}^l} = \frac{\partial C}{\partial z_j^l} \frac{\partial z_j^l}{\partial w_{jk}^l} = \delta_j^l a_k^{l-1} \tag{2.7}$$

ja que $\frac{\partial z_j^l}{\partial w_{jk}^l} = \frac{\partial}{\partial w_{jk}^l}(\sum_i w_{ji}^l a_i^{l-1}) = a_k^{l-1}$

Per tant, el procés de *backpropagation* ens dóna una forma de calcular el gradient de la funció de cost. L'algoritme és el que segueix:

- A partir de l'entrada generem la corresponent activació $\mathbf{a}^1$.
- Fem el procés de *Feedforward*: Per cada $l = 2, 3, \ldots, L$ calculem $\mathbf{z}^l = \mathbf{w}^l \mathbf{a}^{l-1} + \boldsymbol{\theta}^l$ i $\mathbf{a}^l = \sigma(\mathbf{z}^l)$.
- Calculem el vector error de la sortida $\boldsymbol{\delta}^l = \delta_a C \circ \sigma'(\mathbf{z}^L)$.

- Fem *backprogagation* de l'error: Per cada $l = L-1, L-2, \ldots, 2$ calculem $\boldsymbol{\delta}^l = ((\mathbf{w}^{l+1})^T \boldsymbol{\delta}^{l+1}) \circ \sigma'(\mathbf{z}^l)$.
- Com a sortida donem el gradient de la funció de cost a partir de $\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j$ i $\frac{\partial C}{\partial \theta^l_j} = \delta^l_j$.

Pel que fa a la rapidesa, podem observar que el cost computacional de realitzar el pas de *backward* és similar al de *forward* i, per tant, el cost total de l'algoritme és el mateix que realitzar dos passos *forward* a tota la xarxa. Tot i això, quan aquest algoritme s'usa en xarxes neuronals profundes s'han de fer servir altres tècniques per tal de fer aquests càlculs viables.

2.5 Tècniques a per millorar l'aprenentatge de les xarxes neuronals

Tanmateix, amb els conceptes explicats fins ara no en tenim suficient per tal que les xarxes profundes convergeixin. Cal que introduïm noves eines per a fer-ho possible.

2.5.1 La funció de cost *cross-entropy*

Primer de tot, analitzem la funció escollida de cost quadràtic 2.3 que la podem escriure com

$$C(w,b) = \frac{1}{2}(\mathbf{y}(\mathbf{x}) - \mathbf{a})^2$$

Recordant que $\mathbf{a} = \sigma(z)$ on $z = \mathbf{w} \cdot \mathbf{x} + \theta$. Si derivem respecte el pes i el biaix usant la regla de la cadena obtenim

$$\frac{\partial C}{\partial w} = (\sigma(z) - \mathbf{y})\sigma'(z)\mathbf{x} = \sigma(z)\sigma'(z) \tag{2.8}$$

$$\frac{\partial C}{\partial \theta} = (\sigma(z) - \mathbf{y})\sigma'(z) = \sigma(z)\sigma'(z) \tag{2.9}$$

Ara calculem què és $\sigma'(z) = (\frac{1}{1+e^{-z}})' = \frac{e^{-z}}{(1+e^{-z})^2} = \frac{1}{1+e^{-z}} \cdot (\frac{1+e^{-z}}{1+e^{-z}} - \frac{1}{1+e^{-z}}) = \sigma(z)(1-\sigma(z))$.

Tenint present la forma de la funció sigmoidea (Figura 2.2) podem observar que quan el valor d'aquesta sigui proper a 0 o 1, $\sigma'(z)$ tindrà un valor petit i, per 2.8 i 2.9, $\frac{\partial C}{\partial w}$ i $\frac{\partial C}{\partial \theta}$ seran petites, i això vol dir que l'aprenentatge serà lent. Podríem millorar la nostra funció de cost si en trobéssim una altra que evités aquest problema.

Si considerem com a funció de cost la funció d'entropia creuada o *cross-entropy* en anglès,

$$C = -\frac{1}{n}\sum_x [\mathbf{y}(\mathbf{x}) \ln \sigma(z) + (1 - \mathbf{y}(\mathbf{x})) \ln(1 - \sigma(z))] \tag{2.10}$$

On n és el nombre total d'ítems del conjunt d'entrenament, el sumatori és sobre tots els ítems i y és la sortida esperada donada una entrada x. Podem interpretar la *cross-entropy* com a funció de cost, ja que sempre $C > 0$ perquè els logaritmes s'apliquen sobre valors entre 0 i 1 i tenim un signe negatiu a l'inici de l'equació. Per altra banda, la sortida de la *cross-entropy* és propera a zero quan, donada una entrada x, el valor esperat y és proper al calculat $\sigma(z)$.

Veiem què passa si calculem com varia la nova funció de cost respecte el pes i el biaix,

$$\frac{\partial C}{\partial w_j} = -\frac{1}{n}\sum_x (\frac{\mathbf{y}}{\sigma(z)} - \frac{1-\mathbf{y}}{1-\sigma(z)})\frac{\partial \sigma}{\partial w_j} = \frac{1}{n}\sum_x (\frac{\sigma(z)-\mathbf{y}}{\sigma(z)(1-\sigma(z))})\sigma'(z)x_j = \frac{1}{n}\sum_x x_j(\sigma(z)-\mathbf{y}) \tag{2.11}$$

$$\frac{\partial C}{\partial \theta} = -\frac{1}{n}\sum_x (\frac{\mathbf{y}}{\sigma(z)} - \frac{1-\mathbf{y}}{1-\sigma(z)})\frac{\partial \sigma}{\partial \theta} = \frac{1}{n}\sum_x (\frac{\sigma(z)-\mathbf{y}}{\sigma(z)(1-\sigma(z))})\sigma'(z) = \frac{1}{n}\sum_x (\sigma(z)-\mathbf{y}) \tag{2.12}$$

Si comparem l'equació 2.8 amb la 2.11 (resp.2.9 amb 2.12) el terme $\sigma'(z)$ no existeix i, per tant, evitem l'aprenentatge lent per a aquest terme.

2.5.2 El problema de l'*overfitting* i l'ús de tècniques de regularització

Un dels problemes més coneguts durant l'entrenament d'una xarxa neuronal és el *overfitting*. Aquest apareix quan l'error en usar el conjunt d'entrenament és petit però quan verifiquem amb noves dades és gran. Això pot ser perquè la nostra xarxa ha començat a memoritzar la classificació del conjunt d'entrenament en comptes d'aprendre una generalització.

Una forma eficaç d'evitar aquest problema és aconseguir que el nombre de paràmetres de la xarxa sigui molt més petit que els ítems del conjunt d'entrenament. Per tant, augmentar la mida del conjunt d'entrenament seria una possible solució. Molts cops, però, això no és possible per dificultats d'obtenir dades, per això haurem de tenir tècniques per detectar i prevenir l'*overfitting*.

Per tal de detectar-lo, tot i que es tracta només d'un signe d'*overfitting*, es pot fer un seguiment de la precisió d'encert i, si veiem que no augmenta, parar d'entrenar. Per això, podríem reservar un subconjunt de les nostres dades, anomenat *validation set* (diferent del *test set*) que dedicaríem a comprovar que aquest efecte no s'està produint. Aquesta estratègia s'anomena *early stopping*. Cal fer notar que aquest nou subconjunt seria bo que fos disjunt del *test set*, perquè ens servís per a escollir uns hiperparàmetres que poguéssim tenir la certesa que són bons comprovant-ho posteriorment amb el *test set*. Aquest procediment de separar aquests dos subconjunts de verificació s'acostuma a anomenar el mètode de *hold out*.

Les tècniques per tal de reduir l'efecte d'*overfitting* s'anomenen tècniques de regularització. Hi han molts tipus de tècniques; tot i això, nosaltres només n'explicarem breument algunes.

Començarem amb l'anomenada **regularització L2** o *weight decay*. La idea d'aquesta tècnica és incorporar un nou terme, anomenat **terme de regularització**, a la nostra funció de cost. Seguint la notació del treball,

$$C = -\frac{1}{n}\sum_j [y_j \ln a_j^L + (1-y_j)\ln(1-a_j^L)] + \frac{\lambda}{2n}\sum_w w^2 \tag{2.13}$$

On el segon terme sumant de 2.13 correspon a la suma del tots els pesos de la xarxa escalat per un factor que conté $\lambda > 0$ que s'anomena **paràmetre de regularització** i n que continua sent la mida del nostre *training set*.

L'efecte d'aquesta tècnica és que la xarxa prefereixi aprendre de pesos petits, i els pesos grans només seran permesos si realitzen una millora considerable a la part original de la nova funció de cost. A més, podem modificar el valor de λ i, si és gran (resp. petit) fer que prefereixi pesos petits (resp. minimitzar la funció de cost original). D'aquesta manera, considerar més important els pesos petits dificulta l'aprenentatge de soroll local que puguin tenir les dades, en contraposició a una xarxa que tingui pesos grans que pot estar subjecte a canvis molt més abruptes per petits canvis en l'entrada.

Una pregunta que podem fer és per què aquesta tècnica no conté el biaix. La resposta empírica, segons Nielsen [2015], és que introduir-lo no canvia massa el resultat ja que podem tenir un biaix molt gran i això no comporta els mateixos efectes negatius que quan els pesos són grans.

Continuem ara amb una nova tècnica similar a l'anterior anomenada **regularització L1** . En aquest cas, incorporarem un altre tipus de sumand.

$$C = -\frac{1}{n}\sum_j [y_j \ln a_j^L + (1-y_j)\ln(1-a_j^L)] + \frac{\lambda}{n}\sum_w |w| \tag{2.14}$$

Amb aquest darrer factor sumand de 2.14, tornem a potenciar que s'escullin pesos petits enfront grans. La diferència respecte la tècnica anterior és que, donat que ara es considera linealment el valor del pes w, això produeix que, d'acord amb Nielsen [2015], la regularització L1 tendeixi a concentrar els pesos de la xarxa en un nombre petit de connexions importants i les altres tendeixin a 0. Efecte que s'acostuma a definir com que la sortida és de tipus *sparse*[1].

Ara presentem una nova tècnica de regularització que no modifica la funció de cost com les dues anteriors sinó que modificarà la xarxa. Aquesta tècnica s'anomena *dropout* i, a partir de la conferència de Hinton [2013], podem dir que

[1]Provinent de l'efecte de *sparsity* de les matrius que vol dir que només algunes de les posicions no estan a zero.

consisteix en ometre aleatòriament la meitat de les neurones de cada una de les capes ocultes, cada vegada que entrem un nou ítem a la xarxa neuronal. Aquest procés l'aniríem repetint per a cada un dels nous ítems que passin per la xarxa i, d'aquesta manera, aniríem trobant uns pesos i biaixos, més robusts vers el soroll. Una vegada acabat aquest procés, per tal de compensar que hem estat aprenent sempre amb la meitat de les neurones ocultes, reduirem a la meitat tots els pesos resultants. Per a més informació, es recomana la lectura de l'article Hinton et al. [2012] on es presenten diferents exemples de com la tècnica del *dropout* dóna bons resultats i ajuda a aprofundir sobre el tema.

2.5.3 Dos tipus més de models de neurona artificial

A més dels dos tipus de neurones explicats anteriorment, també són molt coneguts dos tipus més que consisteixen en substituir la funció d'activació:

La primera es tracta d'usar com a funció d'activació la tangent hiperbòlica,

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{2.15}$$

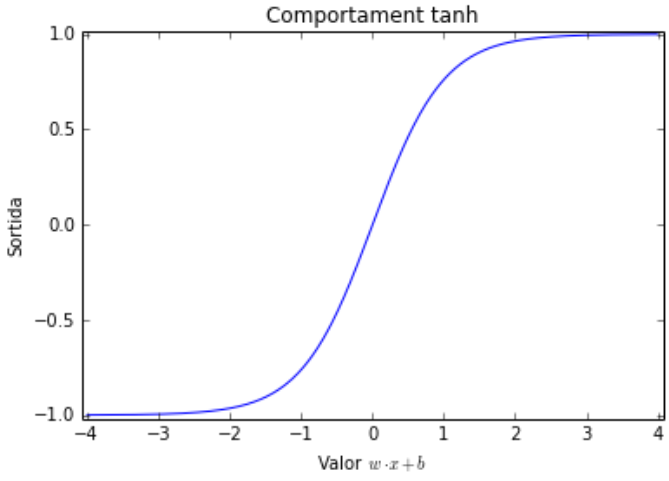


Figura 2.3: Representació funció *tanh*

Aquest tipus de neurones els anomenarem neurones tangent hiperbòliques,

tanh. Si dibuixem la forma de la funció (Figura 2.3) podem observar que pren valors entre $[-1, 1]$, la qual cosa comporta que si es vol normalitzar la sortida s'haurà de fer de forma diferent que en el cas de les sigmoidees.

Finalment, introduim una nova variant de model de neurona artificial anomenada *rectified linear unit* (o abreviat $ReLU$). En aquest cas, la funció d'activació és

$$r(z) = max(0, z) \tag{2.16}$$

On $z = \mathbf{w} \cdot \mathbf{x} + \theta$. Com podem observar a la Figura 2.4, la $ReLU$ el que fa és posar a 0 tots els valors de z que siguin no positius. Tot i tenir uns inconvenients clars com la no diferenciabilitat ni linealitat en el zero, en articles com Glorot et al. [2011] s'ha trobat que aquest tipus de neurones són un model igual o millor, fins i tot, que el de les neurones *tanh* i, computacionalment, només estem fent comparació, suma i multiplicació.

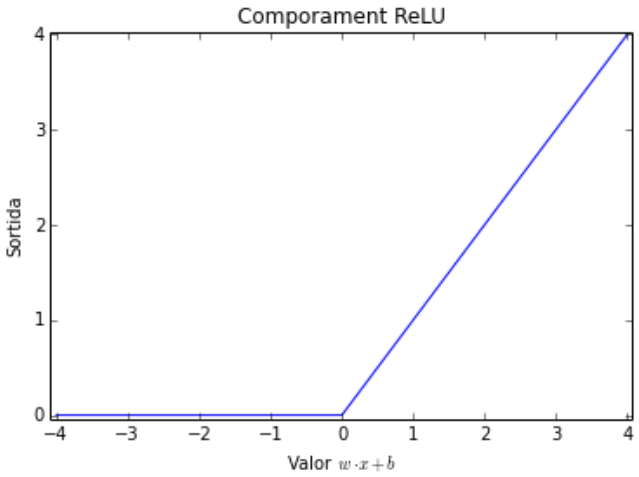


Figura 2.4: Representació funció $ReLU$

2.6 Xarxes Neuronals Profundes

Com havíem vist a 1.3, dins de l'aprenentatge màquina profund, actualment trobem genèricament dos tipus de xarxes neuronals, una les *Convolutional Neural Networks* (*CNN*) i les altres les *Deep Belief Networks* (*DBN*). Nosaltres ens centrarem amb les *CNN* ja que, tal i com diuen Chen et al. [2014] i Arel et al. [2010], estan dissenyades per a tractar dades de dues dimensions, com imatges o vídeos i, a més, el framework Caffe Jia et al. [2014] que usarem, està pensat per a arquitectures convolucionals.

2.6.1 Xarxes neuronals convolucionals

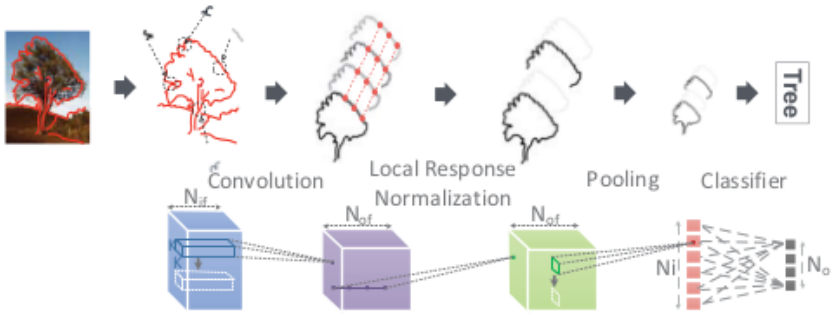


Figura 2.5: Els quatre tipus de capes d'una CNN. Imatge extreta de Chen et al. [2014]

Segons Arel et al. [2010], les *CNN* són un tipus de família de xarxes neuronals amb múltiples capes. De fet, com esmenta Chen et al. [2014], aquestes es poden interpretar com múltiples instàncies de 4 tipus de capes (vegis Figura 2.5): *convolutional layer* (*CONV*), *pooling layer* (*POOL*), *local response normalization layer* (*LRN*) i *classifier layer* (*CLASS*), o també anomenades *fully connected layer* o producte interior, del que ara n'explicarem el seu funcionament.

Tot i això, la combinació d'aquestes capes no és suficient per a obtenir bons resultats i calen d'altres tècniques, com s'explica a Ranzato [2014]. En aquest treball n'hem introduït algunes, com per exemple les regularitzacions L1, L2, el

dropout, l'ús de neurones *ReLU*.

Anem a parlar dels quatre tipus de capes bàsics (a partir de Chen et al. [2014]):

- *convolutional layer*: Intuïtivament, una capa convolucional implementa un conjunt de filtres per tal d'identificar elements característics donada una entrada. Si aquesta entrada és visual, un filtre tindrà una mida fixada de $K_x \times K_y$ formant l'anomenat nucli o *kernel* en anglès. Els coeficients d'aquest *kernel* seran apresos i formen la capa de pesos sinàptics. Aquesta capa és desplaçada N_I cops a través de l'entrada, donant com a molt N_O *feature maps* que seran les sortides.

  La fórmula per a obtenir el valor de la posició (i, j) d'una *features maps* f_o és:

  $$\text{sortida}_{f_o}(i,j) = \sum_{n=0}^{N_I}\sum_{x=0}^{K_x}\sum_{y=0}^{K_y} w_{f_n,f_o}(x,y) \cdot \text{ entrada}_{f_n}(i+x, j+y)$$

  On $\text{entrada}_f(i,j)$ (resp. $\text{sortida}_f(i,j)$) representa l'entrada (resp. sortida) de la posició (i,j) de la *feature map* f. I siguin f_n,f_o les *feature map* d'entrada i sortida respectivament, aleshores $w_{f_n,f_o}(x,y)$ és el pes sinàptic de la posició (x,y) del *kernel* en aquestes *feature map*. Donat que la capa d'entrada pot contenir múltiples *feature map*, el *kernel* és normalment de tres dimensions $K_x \times K_y \times N_I$. Pel que fa a les *CNN*, el *kernel* comparteix els valors per a totes les neurones amb la mateixa *feature map* de sortida.

- *Pooling layer*: Aquesta capa calcula el màxim o mitjana sobre tots els punts veïns donada una posició, i.e.

  $$\text{sortida}_f(i,j) = \max_{0\le x\le K_x, 0\le y\le K_y} \text{entrada}_f(i+x, j+y)$$

  D'aquesta manera reduïm les dimensions de la capa d'entrada i permetem que només les característiques importants es conservin i siguin identificades a la següent capa convolucional. Aquesta capa no té paràmetres per a aprendre.

- *Local Response Normalization layer* (LRN): Aquesta capa realitza una competició entre les mateixes posicions de les diferents *feature maps*. Com diu Krizhevsky et al. [2012], el seu efecte és similar al de la inhibició lateral que realitzen les neurones biològiques (tal com podem observar a la Figura 2.5). El càlcul és el següent:

$$\text{sortida}_{f_o}(i,j) = \frac{\text{entrada}_{f_o}(i,j)}{(c + \alpha \sum_{k=max(0,o-\frac{n}{2})}^{min(N-1,o+\frac{n}{2})} [\text{entrada}_{f_k}(i,j)]^2)^\beta}$$

  On la suma es mou en les n *feature map* al voltant, N és el nombre total de *kernels* a la capa on ens trobem i c, n, α i β són hiper-paràmetres que són constants que es fixen prèviament.

- *Fully connected layer*: Aquesta capa és una capa totalment connectada enllaçant totes les seves N_i entrades amb les N_o sortides (vegis Figura 2.5). L'objectiu d'aquesta capa és correlacionar les diferents característiques extretes dels altres tipus de capa anterior. La fórmula seria:

$$\text{sortida}(j) = a(\sum_{i=0}^{N_i} w_{ij} \cdot \text{entrada}(j))$$

  On $a()$ és la funció d'activació decidida (sigmoidea, *tanh*, $ReLU$, etc.).

Capítol 3

Desenvolupament de Xarxes Neuronals Profundes

Actualment, tal i com es diu a Yangqing Jia [2014] hi han diferents *frameworks* per tal de construir i entrenar models profunds. Entre els existents podríem destacar:

- Torch7: Dóna un suport flexible i ràpid per tal de consturir algoritmes d'aprenentatge màquina. Desenvolupat a la *New York University* (NYU).
- Theano/Pylearn2: Llibreries de python d'aprenentatge màquina amb la possibilitat d'usar GPU. Desenvolupat a la Universitat de Montreal.
- Cuda-convnet2: Implementació molt ràpida amb ús de GPUs i multi-GPU paral·lelitzades usant la llibreria C++/CUDA. Desenvolupat per Alex Krizhevsky.
- Caffe: *framework* desenvolupat actualment pel Centre de Visió i Aprenentatge de Berkeley (BVLC) i per comunitat, i originalment desenvolupat per Yangqing Jia.

Tots els anteriors *frameworks* compleixen amb les característiques de ser implementacions codi obert i, depenent de l'experiment que es va a tractar, en

podríem escollir un o altre. En el nostre cas, escollirem el *framework* Caffe perquè ens dóna els avantatges de ser molt ràpid (es calcula que pot processar 40 milions d'imatges per dia amb una GPU NVIDA K40, que equival a 5ms/imatge en el entrenament i 2ms/imatge en el test), a més de tenir una arquitectura clara, com podrem observar a continuació, i una gran comunitat activa darrere, tan desenvolupant-lo com donant-li ús.

Anem a explicar el procés d'instal·lació d'aquest *framework* i prosseguirem amb la comprensió de com usar-lo.

3.1 Instal·lar Caffe i les diferents dependències

Per tal de realitzar la instal·lació vam usar la guia d'instal·lació oficial de Caffe. Tal i com diuen, si la instal·lació es realitza sobre una distribució Ubuntu 14.04, Ubuntu 12.04 o bé sobre OS X 10.9, OS X 10.8 aquesta no hauria de tenir cap problema. Tot i això, en el nostre cas vam realitzar-la sobre una distribució de 64-bits *Debian GNU/Linux 7 (wheezy)* sense una GPU prou bona com per a usar amb Caffe. La idea seria que aquesta instal·lació permetés realitzar el procés posterior de testeig i visualització dels resultats, i que la part d'entrenament que requeria GPU s'usés en una altra màquina més potent.

El procés d'instal·lació no va ser tan ràpid com es descrivia a la web i es van haver de fer alguns processos extres. Els primeres problemes van venir per a poder aconseguir instal·lar les llibreries bàsiques sense l'ús de GPU, però els problemes més llargs van ser posteriorment per la versió *Wheezy* de Debian que no podia instal·lar algunes llibreries de bones a primeres, sense una actualització a una versió posterior de Debian. Es va considerar oportú recollir els passos realitzats i escriure'ls en un blog (Figura 3.1) per tal de facilitar futures instal·lacions i assegurar que és possible realitzar la instal·lació en aquesta versió de Linux. A més, el blog podia servir en un futur per a recopilar el treball fet.

Resumidament i seguint el que està escrit en el blog, Caffe té dues dependències bàsiques: la llibreria de CUDA i els programes de càlcul matricial BLAS que, en el nostre cas, vam instal·lar ATLAS. Posteriorment, ens interessava poder tractar les dades des d'un Notebook d'IPython a partir del *Python wrapper* de Caffe i,

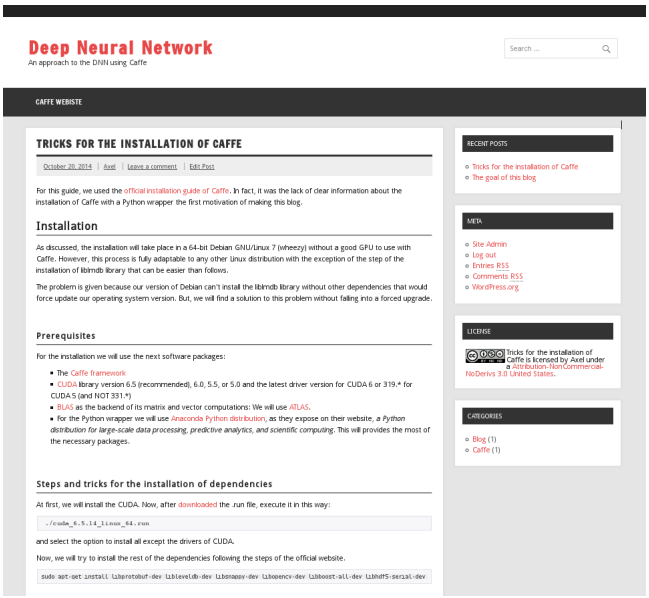


Figura 3.1: Captura de pantalla del blog escrit per ajudar a la instal·lació de Caffe durant el treball.

per tant, vam instal·lar la distribució Anaconda Python per tal de tenir la majoria de paquets necessaris.

Quan ens vam trobar amb problemes de dependències o errors per falta de paquets, en els casos de la *liblmdb*, els *gflags* i el *glog* vam acabar descarregant exteriorment els paquets i instal·lant-ho. I per tal d'evitar posteriorment un problema amb la llibreria *liblmdb.so*, vam realitzar la compilació de Caffe amb un mètode alternatiu usant CMake, tot i estar en fase de proves, que ens va permetre evitar aquell error.

3.2 L'arquitectura de Caffe

Caffe ens permetrà definir una xarxa neuronal capa per capa des de les dades que entrarem, fins una capa final que serà on calcularem la funció de cost. Les dades i derivades transiten a través de la xarxa amb el procés explicat anteriorment de *fordward* i *backpropagation* i Caffe s'encarrega de comunicar i manipular aquesta informació amb el que s'anomenen *Blobs*. Un *Blob* és un contenidor de

les dades, derivades i paràmetres actuals que estan sent processats i transmesos per Caffe i que, a la vegada, permet la sincronització entre la CPU i la GPU. Més concretament, aquests *blobs* són unes col·leccions 4 dimensionals que guarden ordenadament el *batch* (N) (que anomenen Num) que és el nombre d'imatges o filtres que s'entraran juntes, el canal (K) (*Channel*) que és el nombre de capes que té la imatge (una imatge RGB tindria 3 capes), l'alçada (H) i l'amplada (W) de la imatge.

Per tal de definir un model d'arquitectura concret, Caffe usa un fitxer del tipus *protocol buffer* amb extensió *prototxt*. Les diferents capes que podem definir en aquest fitxer són genèricament:

- Les quatre bàsiques de les xarxes neuronals convolucionals explicades en l'anterior capítol (*convolutional layer*, *pooling layer*, *Local Response Normalization layer* i *Fully connected layer*)
- La *Loss layer* que serà una capa que ajudarà a l'aprenentatge minimitzant la funció de cost que pot ser:
  - L'error quadràtic (2.3) o també anomenada norma Euclidiana (EUCLIDEAN_LOSS)
  - La *Sigmoid Cross-Entropy* (SIGMOID_CROSS_ENTROPY_LOSS) (2.2 i 2.10)
  - La *Softmax* (SOFTMAX_LOSS) que li diuen a una capa *softmax* (que és una generalització de la funció logística que té la forma de la funció sigmoidea 2.1) on seguidament a la capa *softmax* és col·loca una *multinomial logistic loss* i això provoca que hi hagi millor estabilitat en el gradient.
  - O d'altres tipus explicats en la secció on parlen de les capes de la guia oficial de Caffe.
- Les capes d'*Activation/Neuron* que, en general, són operadors element per element agafant un *blob* d'entrada i produint un altre *blob* de mateixes dimensions. Aquest tipus d'operador pot ser la funció ReLU (2.16), la sigmoidea (2.2), la tangent hiperbòlica (2.15), el valor absolut o d'altres.

Per exemple, anem a crear la xarxa convolucional de la Figura 1.2. El fitxer que definiria les capes seria el train_val.prototxt (seguiu l'hipervincle per veure el fitxer complet de l'exemple) i en ell trobaríem les següents capes:

- Una *convolucional* amb un *kernel* de 11×11 i fent un salt (*stride*) de 4 i amb 96 sortides.
- Una d'*Activation/Neuron* tipus ReLU
- Una *pooling* agafant el màxim d'un *kernel* de 3×3 i amb *stride* de 2.
- Una LRN, seguint la notació del treball, amb $c = 5, \alpha = 0.0001, \beta = 0.75$.

Ara vindrien les capes intermèdies que serien una combinació de les anteriors amb els hiperparàmetres modificats, i passaríem a les últimes quatre capes que serien:

- Una capa que realitzaria un *dropout* amb un ratio del 50%, tal i com vam definir en l'anterior capítol del treball.
- Una capa *fully connected* que tindrà 1.000 sortides que són les classes que anem a classificar.
- Una capa, anomenada *accuracy*, que servirà pel test que ens calcularà el valor d'encert.
- Una capa que ens determinarà el tipus de funció de cost que usarem. En aquest cas una *softmax*.

Al principi del fitxer tenim dues capes amb nom *data*[1]. Aquestes capes ens marcaran el *dataset* d'entrenament i test respectivament. Caffe accepta les dades en els formats estàndard d'imatges, en HDF5 o més eficientment usant el tipus de base de dades indexada LMDB o LevelDB. En el cas de l'exemple, el tipus de format de les dades és LMDB, això voldrà dir que, abans, haurem de realitzar un procediment d'enpaquetament de les dades amb un codi similar al fitxer d'intèrpret d'ordres create_imagenet.sh. Una vegada generada la LMDB,

[1]Per a més informació vegis la web oficial on es parla d'aquest tipus de capa

si volem que s'homogeneïtzin les dades amb la mitja, com és el cas del nostre exemple, crearem aquesta mitja en format *binaryproto* amb un codi similar a make_imagenet_mean.sh.

Una vegada escrit el model de la xarxa neuronal caldria especificar els paràmetres que tindrà la xarxa i com s'aniran actualitzant per optimitzar el valor de la funció de cost. Per tal de fer aquesta tasca Caffe defineix un fitxer anomenat Solver[1]. En aquest fitxer trobarem paràmetres com el nombre d'iteracions a realitzar (*max_iter*), cada quantes iteracions realitzar un test, el tipus de política d'aprenentatge, el *momentum* (paràmetre molt important tal i com diu Sutskever et al. [2013]) i altres paràmetres, com cada quan realitzar una copia de seguretat dels valors de la xarxa neuronal (*snapshot*) o si els càlculs es volen realitzar sobre CPU o GPU.

Si continuem amb l'exemple, el contingut del Solver.prototxt seria:

```
net: "models/bvlc_reference_caffenet/train_val.prototxt"
test_iter: 1000
test_interval: 1000
base_lr: 0.01
lr_policy: "step"
gamma: 0.1
stepsize: 100000
display: 20
max_iter: 450000
momentum: 0.9
weight_decay: 0.0005
snapshot: 10000
snapshot_prefix:
    "models/bvlc_reference_caffenet/caffenet_train"
solver_mode: GPU
```

On podem observar, en particular, que cada 20 iteracions mostrarem l'estat de la funció de cost, que el nombre d'iteracions que realitzarà l'entrenament mitjançant GPU seran 450.000, que realitzarem un test cada 1.000 iteracions

[1]Vegis la pàgina web oficial per a més informació detallada del fitxer

i que el *learning rate* el situarem, al principi, en 0.01.

Després de configurar aquests fitxers hem d'entrenar la nostra xarxa neuronal. Si seguim amb l'exemple, l'ordre seria:

```
./build/tools/caffe train \
    --solver=models/bvlc_reference_caffenet/solver.prototxt
```

On estem cridant a l'executable Caffe dient-li que volem entrenar i indicant a on es troba la direcció del nostre fitxer *solver*. Posteriorment, el *solver.prototxt* ja indicarà on es troba el *train_val.prototxt* per tal de carregar les capes de la xarxa neuronal. Aquesta execució generarà, quan s'hagi indicat en el valor *snapshot* del solver, els fitxers *caffenet_train_xxxx.caffemodel* i *caffenet_train_xxxx.solverstate* on *xxxx* serà el nombre d'iteracions realitzades en el moment de realitzar la copia de seguretat. El fitxer amb format *solverstate* ens permetrà reprendre l'entrenament en aquell punt carregant de nou els pesos i l'estat mitjançant la comanda

```
./build/tools/caffe train \
--solver=models/bvlc_reference_caffenet/solver.prototxt \
--snapshot=models/bvlc_reference_caffenet/
caffenet_train_xxxx.solverstate
```

Pel que fa al fitxer amb format *caffemodel*, aquest serà el que usarem per posteriorment provar la xarxa entrenada mitjançant el *Python Wrapper* tal i com veurem més endavant. També aquest fitxer ens permetrà realitzar un procés que s'anomena diu *Fine-tuning* i que consisteix en, donats dos problemes a resoldre, si volem usar la mateixa arquitectura de xarxa neuronal, entrenem primer la xarxa neuronal per tal de solucionar el primer problema i, un cop tenim uns pesos que ens serveixen per a classificar el primer problema, usem aquests pesos per inicialitzar els pesos d'una nova xarxa neuronal que serà idèntica a la primera però amb els pesos de les 'últimes capes' reiniciats. Si donem molta més prioritat d'aprenentatge a aquestes 'últimes capes' s'ha trobat que, per a certs casos, la classificació pel segon problema continua sent molt bona, com podrem observar en el treball més endavant. Per a fer això mitjançant Caffe ho podem aconseguir posant el *blobs_lr* de les 'últimes capes' molt més gran en les demés i canviant el

nom d'aquestes 'últimes capes' de manera que Caffe només carregarà els pesos de les capes que conservaran el nom. Una vegada modificat el fitxer *train_val.prototxt* hauríem d'executar la següent ordre

```
caffe train -solver models/.../solver.prototxt
-weights models/primer_problema/caffenet_train_xxxx.caffemodel
```

Aquest tipus de procediment s'ha començat a usar quan existeixen models de xarxes neuronals per a problemes com la ImageNet, que han estat entrenades amb milions d'imatges amb l'ús de GPUs molt potents i s'ha comprovat que, si s'usen els valors obtinguts d'aquests models per a nous problemes, hi han cops que els resultats són molt bons i ajuden a trobar una nova bona classificació.

Finalment, Caffe defineix un fitxer que s'anomena *deploy.prototxt* on s'especifica l'esquelet de les capes usades per tal que, posteriorment, es pugui interpretar el *caffemodel*. En el cas del nostre exemple, el deploy es pot consultar en aquest hipervincle.

3.3 Interfícies de treball

Una vegada tenim la xarxa neuronal entrenada, Caffe proporciona una interficie per línia de comandes, una per Python (pycaffe) i una per MATLAB per a la visualització i per al desenvolupament posterior. En el nostre cas, ens hem centrat en la de Python. En particular, aquesta transformarà els *blobs* en *numpy ndarrays*, cosa que farà més fàcil l'ús i visualització d'aquests.

Per tal d'importar el mòdul de *pycaffe* i els seus scripts, la primera comanda que hem d'introduir en el nostre Notebook seria:

```
caffe_root = '/directori/principal/caffe/'
sys.path.insert(0, caffe_root + 'python')
import caffe
```

Seguidament si volguéssim carregar la xarxa entrenada hauríem de fer

```
MODEL_FILE = 'models/.../deploy.prototxt'
PRETRAINED = 'models/.../caffenet_train.caffemodel'
```

```
net = caffe.Classifier(MODEL_FILE, PRETRAINED,
channel_swap=(2,1,0), raw_scale=255, image_dims=(227, 227))
net.set_mode_cpu()
```

On *net* serà un objecte que contindrà la xarxa carregada. Observem que en aquest exemple s'està considerant que estem usant imatges de tres canals (com per exemple RGB) i els hem invertit per tal que a la visualització surtin tal i com esperem (ja que Caffe inverteix els 3 canals internament) mitjançant el paràmetre *channel_swap*. Seguidament, el paràmetre *raw_scale* ens permetrà modificar l'escala de tons que en principi Python tracta les imatges en $[0, 1]$ i, d'aquesta manera, queda en $[0, 255]$. I el paràmetre *image_dims* ens indica la mida d'escalament o retall de les imatges d'entrada.

I amb la funció *set_mode_cpu()* fem que quan, posteriorment, realitzem una predicció d'una nova entrada no usem GPU sinó només CPU.

Donada una imatge, la carregarem mitjançant la funció *load_image* i, posteriorment, farem la predicció com prossegueix

```
IMAGE_FILE = 'dataset/test/image.jpg'
input_image = caffe.io.load_image(IMAGE_FILE)
prediction = net.predict([input_image],oversample=False)
```

Observem que, en aquest cas, a l'hora de fer la predicció hem desactivat l'*oversample* i, per tant, hi haurà més soroll.

I amb això obtenim un *numpy.ndarray* anomenat *prediction*, on a la primera posició podem obtenir els valors de la predicció realitzada. Per tant, si volguéssim mostrar-ho podríem escriure

```
print 'prediction␣shape:', prediction[0].shape
plt.plot(prediction[0])
print 'predicted␣class:', prediction[0].argmax()
```

On imprimiríem el nombre de classes, pintaríem la gràfica del valor que associa a cada classe i imprimiríem el número de la classe que obté el valor més alt.

Capítol 4

Definició d'una Xarxa Neuronal Profunda per a un problema concret

Per tal d'aprendre com construir una xarxa neuronal i aprendre el funcionament mitjançant Caffe, primer anem a resoldre un problema que actualment té solucions que donen bons resultats i que no és molt costós computacionalment, de manera que puguem fer proves ràpides. Aquest primer problema serà resoldre la classificació dels nombres escrits a mà de la base de dades MNIST (Yann LeCun and Burges [1998]).

Posteriorment, ens plantejarem un problema més gran i complex. L'article Bossard et al. [2014] ens motivarà a plantejar-nos tres reptes:

- Primer de tot, aconseguir obtenir el resultat d'encert al que han arribat els autors.
- Seguidament, a partir de veure que, potser, en el cas del *dataset* de la Food-101, podria millorar si afegíssim una nova banda que contingués informació de textura del menjar, intentar modificar Caffe per tal que accepti aquesta nova informació i mirar quins resultats s'obtenen.
- Finalment, intentar pujar el nivell de reconeixement del *dataset* Food-101.

4.1 Resolució d'un problema petit: la MNIST

La MNIST és una base de dades de dígits escrits a mà que conté 60.000 imatges d'entrenament i 10.000 pel test. Aquestes imatges van ser obtingudes a partir del NIST i, posteriorment, van ser normalitzades i centrades en una mida fixa. És una bona base de dades per començar a fer proves ja que les imatges són petites (28×28) i, per tant, el cost computacional serà molt menor. Totes les proves que es van realitzar amb la MNIST i el temps que esmentarem posteriorment són en l'ordinador on vam instal·lar Caffe i, per tant, sense GPU.

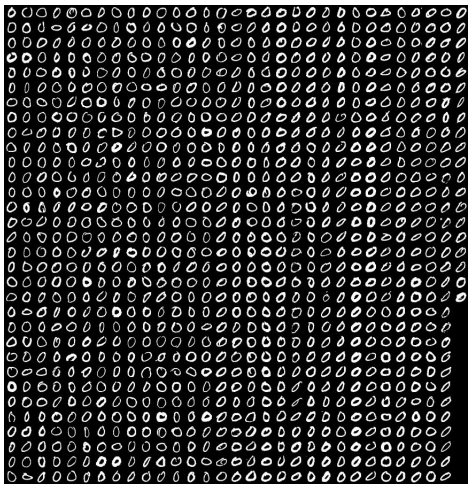


Figura 4.1: Part del test del *dataset* del número 0. Imatge extreta de l'ISY

El que primer vam haver de fer va ser obtenir les dades amb el mateix format que teníem el nostre problema complex posterior, és a dir, amb imatges individuals i separades les classes per carpetes. Si anem a la web oficial de la MNIST veurem que el format que ens proporcionen no és el que busquem, així que el que vam fer va ser descarregar-nos les imatges des de l'*Institutionen för systemteknik* on teníem agrupades cada una de les imatges de cada classe juntes, tal i com s'observa a la Figura 4.1.

Una vegada descarregades les imatges, vam tallar les imatges que contenien tots els nombres escrits d'una classe en les diferents imatges dels nombres i les vam col·locar dins de les seves carpetes respectives a partir d'un pe-

tit programa en Python que es pot trobar a les primeres cel·les del Notebook *TFG_TUTORIAL_MNIST.ipynb*. Una vegada realitzat aquest procediment, vam netejar les imatges que no contenien res ja que, com podem observar a la Figura 4.1, les col·lumnes no estaven totes completament plenes.

Seguidament havíem de generar dos fitxers anomenats *train.txt* i *val.txt* que contindrien, separat per salts de línia, la ruta relativa des del directori arrel de la MNIST fins a cada una de les imatges del conjunt d'entrenament (resp. de test), un espai en blanc, i el número de la classe (o categoria) que els hi correspon a cada una.

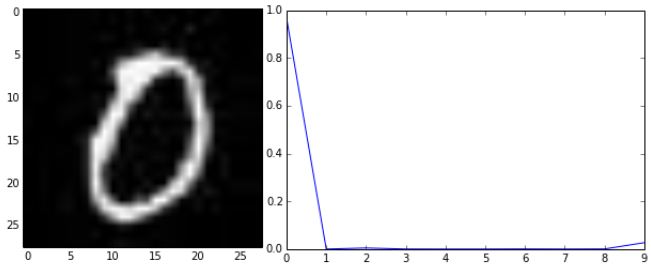


Figura 4.2: Exemple de predicció donada la imatge del zero. El codi es pot trobar al Notebook *TFG_TUTORIAL_MNIST.ipynb*

I, en principi, ja podríem haver generat les corresponents *lmdb* però restava un detall i era que les imatges que nosaltres passàvem eren d'una canal i la implementació de Caffe quan generàvem la lmdb sense especificar-li quants canals tenia la nostra imatge el que feia era copiar la informació del primer canal als dos canals posteriors i generar una imatge amb tres canals iguals. Un cop mirada la implementació de Caffe vam poder observar que hi havia a la classe convert_imageset.cpp un paràmetre booleà anomenat *gray* que ens permetia especificar a Caffe quan les imatges que estàvem passant eren en escala de grisos i, en cas que estigués activat, a la funció *ReadImageToDatum* del fitxer io.cpp Caffe ho tractava com una imatge d'un canal. Això, ens va permetre començar a conèixer quines serien les funcions que posteriorment hauríem de modificar de Caffe per tal que aquesta pogués acceptar no només imatges de 1 o 3 canals sinó qualsevol valor que ens interessés.

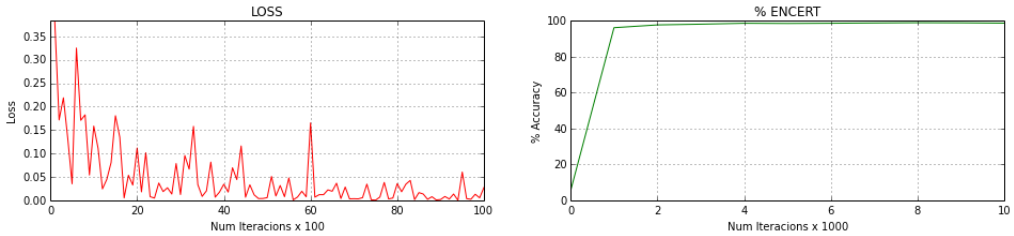


Figura 4.3: Encert i *LOSS* per la MNIST Grayscale usant l'arquitectura LeNet. El codi es pot trobar al Notebook *TFG_TUTORIAL_MNIST.ipynb*

Primer vam provar d'entrenar una xarxa amb les mateixes capes que la LeNet amb 10.000 iteracions però realitzant una copia de seguretat cada 2.500 tal i com podem observar a la carpeta adjunta al treball en el fitxer *MNIST/Grayscale/-lenet_solver.prototxt*. El resultat després de dues hores va ser el de la Figura 4.3. On vam poder aconseguir un encert del 0.9898 i un resultat de la funció *LOSS*, o funció que també l'havíem anomenat funció de cost, que acaba als 0.0299233 (valor petit i, per tant, un bon resultat).

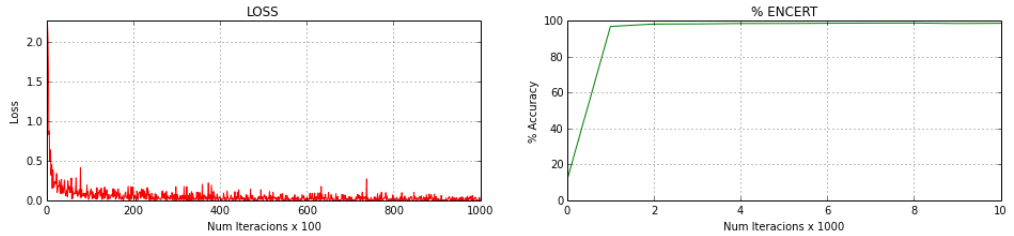


Figura 4.4: Encert i *LOSS* per la MNIST Color usant l'arquitectura LeNet. El codi es pot trobar al Notebook *TFG_TUTORIAL_MNIST.ipynb*

Un cop feta aquesta execució, vam verificar què passaria si realitzàvem el mateix procediment però sense el paràmetre *gray* (o el que és el mateix, amb el paràmetre *gray=false*) i, per tant, generant a partir de la imatge d'un canal els dos següents copiant el contingut. El resultat va ser el de la Figura 4.4. Podem observar que en aquest cas la funció *LOSS* comença molt més amunt que en l'anterior, però acaba assolint un valor similar de 0.0260812 i l'encert en aquest cas puja una mica més i es situa al 0.9919. Aquest procés d'entrenament va durar

dues hores, tal i com podem llegir en el fitxer de la sortida per consola.

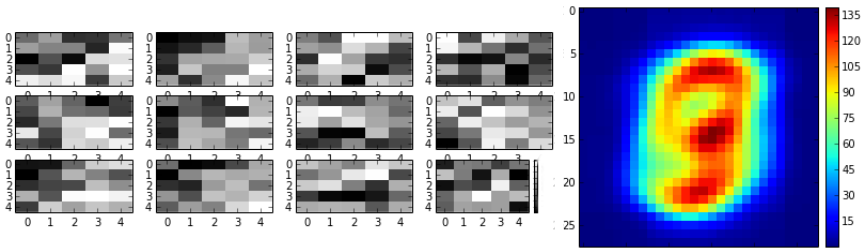


Figura 4.5: Filtres obtinguts a la primera capa convolucional i matriu de confusió d'un canal del classificador. El codi es pot trobar al Notebook *TFG_TUTORIAL_MNIST.ipynb*

En aquell moment es va veure necessari modificar el codi de la Caffe per tal de fer que aquesta pogués interpretar n-bandes. Com que la idea seria en un futur tenir una imatge de tres canals i inserir un canal més per tal de posar la informació que ens pogués donar un filtre passat per la imatge, vam considerar oportú implementar l'extensió de la Caffe amb n-bandes de la següent manera: l'usuari indicaria la direcció de la imatge, si és o no en blanc i negre usant el paràmetre ja existent *gray*, i amb un nou paràmetre indicaria el nombre de canals extres que es voldrien afegir a la imatge original. Aquests nous canals, vindrien donats amb imatges que s'anomenarien igual que la imatge original, una barra baixa i un número identificador del canal extra. Per exemple, si tenim la imatge *imatge3.jpg*, les n capes extres seran fitxers d'una capa anomenats *imatge3.jpg_0*,...,*imatge3.jpg_n* situats al mateix directori. La implementació, a més, estava pensada perquè si en un futur es necessitava, les capes extres poguessin ser de 3 canals modificant molt poc el codi. Una altra manera més intuïtiva d'haver solucionat aquest problema hagués estat crear anteriorment imatges de n capes i llegir-les directament, però es va considerar oportú poder visualitzar les capes extres de manera ràpida i, per tant, es va escollir aquesta solució.

Els fitxers a modificar van ser *caffe-master/tools/convert_imageset.cpp* on vam afegir un paràmetre més anomenat *channels* que indicava el nombre de canals extres que s'anaven a carregar i, posteriorment, es va modificar la crida de la funció *ReadImageToDatum* de *caffe-master/src/caffe/util/io.cpp* afegint un nou

paràmetre *channel* que era el mateix valor de *channels* però en enter. Com la lectura dels píxels la realitzàvem en aquesta funció *ReadImageToDatum*, va caldre afegir una part que realitzés un procediment extra de llegir les següents imatges en cas que *channel* fos més gran que 0 i actualitzar correctament els valors de quants canals hi havia en total segons el *datum*. Paral·lelament, també es van haver de modificar els fitxers equivalents de la interfície en Python, és a dir, els fitxers *caffe-master/python/caffe/io.py*, concretament, la funció *load_image* per tal que carregués els n canals posteriors. Totes aquestes modificacions les podem veure en el codi en els fitxers adjuntats al treball.

Una vegada realitzades aquestes modificacions ja podíem fer la primera prova que seria carregar les imatges dels nombres escrits a mà com a imatges d'escala de grisos i afegir dues capes extres que fossin la mateixa imatge[1]. D'aquesta manera, obtindríem una situació similar a quan dèiem que ho carregués com a imatge de tres canals (no posant el *gray* a *true*) i triplicàvem la mateixa informació. El resultat va ser aconseguir un encert molt proper al cas igual, amb 0.9905 i un valor de la funció *LOSS* que acaba en 0.0293949 en dues hores. La gràfica seria la Figura 4.6.

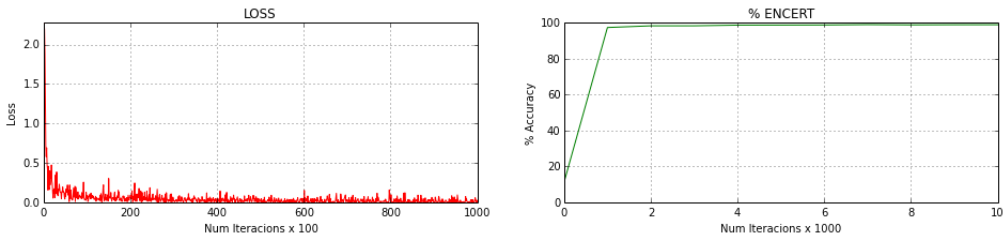


Figura 4.6: Encert i *LOSS* per la MNIST 3BandesIgual usant l'arquitectura LeNet. El codi es pot trobar al Notebook *TFG_TUTORIAL_MNIST.ipynb*

Finalment, per a experimentar la possibilitat d'afegir capes i superar el nombre de 3 canals en total, es va considerar el cas de llegir la imatge com escala de grisos i afegir 3 capes més que fossin altres imatges de la mateixa categoria[2]. D'aquesta

[1] Aquesta prova la vam anomenar 3BandesIguals i es pot trobar en els fitxers adjunts al treball

[2] Seguint la nomenclatura, aquesta prova la vam anomenar 4BandesDiferents i es pot trobar en els fitxers adjunts al treball

manera, un nombre escrit a mà no seria un nombre sinó la unió de 4 capes on totes contenen el nombre en qüestió. Pel cas de la MNIST podia tenir sentit ja que no eren imatges que continguessin molt soroll i així va ser (tot i que no té cap utilitat el resultat ja que per fer el test també es consideraven imatges de 4 canals on cada canal contenia el mateix número escrit diferent). Els resultats van ser d'un encert del 0.9944 i una *LOSS* del 0.0285009 després de dues hores quarantacinc minuts i la gràfica de l'increment d'encert/*LOSS* respecte les iteracions va ser la Figura 4.7[1].

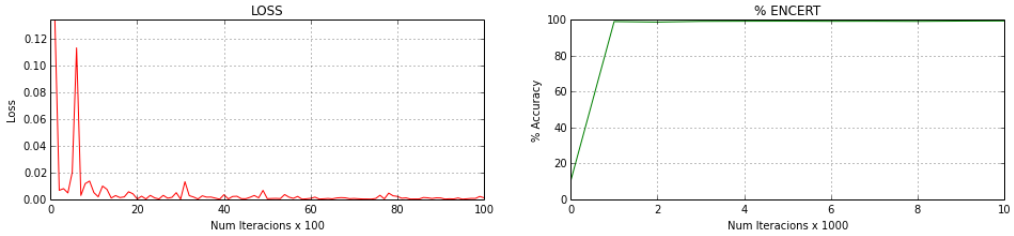


Figura 4.7: Encert i *LOSS* per la MNIST 4BandesDiferents usant l'arquitectura LeNet. El codi es pot trobar al Notebook *TFG_TUTORIAL_MNIST.ipynb*.

L'ús de la GPU en aquest cas no va ser necessari, com podem veure. Els entrenaments eren processos que es podien dur a terme en poques hores i, per tant, eren viables. Tot i això, aquest avantatge va acabar quan es va tractar el problema de la base de dades Food-101 (Bossard et al. [2014]).

4.2 Anàlisi d'un problema més complex: la Food-101

La base de dades Food-101 és un recull de 101.000 imatges de 101 classes diferents de plats. Aquesta informació, tal i com expliquen en la mateixa web oficial, conté soroll i hi ha errors de mal etiquetatge i les imatges no estan escalades a la mateixa mesura (tot i haver un màxim de llargada de 512 píxels). Tot i això, és

[1]Curiosament, tenim un pic en la funció de cost que en casos anteriors no havíem experimentat, tot i que, l'encert continua augmentant constant. Això pot ser degut a que, casualment, no s'hagués entrat un tipus concret de classe fins aquell *mini-bacth* on hi ha el pic.

Figura 4.8: Exemples de dins de la Food-101, imatge extreta de la web oficial.

una bona base de dades per a començar a classificar, i els resultats fins el moment de classificació continuen sent prou baixos.

Els autors de l'article Bossard et al. [2014] havien realitzat un estudi sobre la utilització d'un mètode usant *Random Forests* i havien arribat a un encert del 50.76%, però els mateixos autors esmentaven que si usaven l'arquitectura de la *ImageNet*, en 450.000 iteracions una CNN es situava amb un resultat superior de 56.40% en sis dies mitjançant una GPU NVIDIA Tesla K20X. Aquest va ser el primer objectiu d'aquesta part: aconseguir el mateix resultat que havien aconseguit els autors de l'article.

Donat que abans de llançar la xarxa gran a aprendre voliem realitzar proves en l'ordinador sense GPU potent, es van retallar les imatges de la Food-101 formant la Food-101-crop50 i, posteriorment, vam generar els fitxers *train.txt*, *val.txt* com en el cas de la MNIST i un fitxer *cat.txt* que ens indicaria per cada classe de plat el seu nombre identificador de categoria.

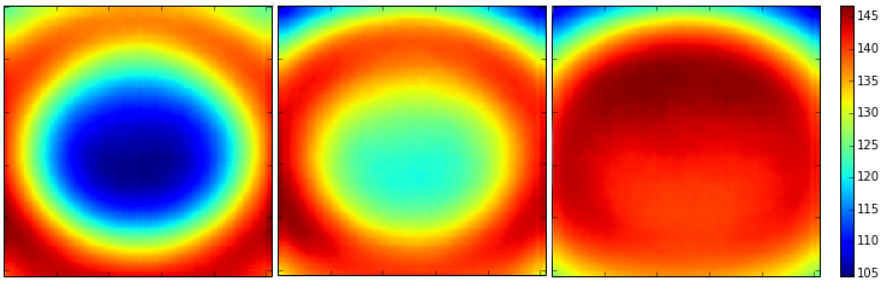


Figura 4.9: Les tres matrius de confusió per a cada canal R, G, B de la Food-101 a partir de la mitja. Observem que es marca la zona on la majoria d'imatges tenen el plat. El codi es pot trobar a *TFG_TUTORIAL_Food-101.ipynb*.

Posteriorment es va provar de fer un subconjunts de la Food-101 que consistia en acabar reduint els fitxers *train.txt*, *val.txt* agafant menys mostres de cada categoria o bé inclús reduir el nombre de classes també a detectar. Tanmateix, tots els casos només van servir per comprovar que els paràmetres no donaven errors quan entrenàvem, però no van servir per a fer proves a petita escala que servís per quan s'entrenés amb un ordinador amb GPU. No va ser fins que no es va executar la Food-101 amb 450.000 iteracions i seguint l'arquitectura i valors de la *ImageNet* que no vam començar a tenir resultats interessants. Vam preparar els fitxers tal i com es pot observar en el Notebook *TFG_TUTORIAL_Food-101.ipynb* i una vegada executat, el resultat obtingut va ser l'esperat. Per tant, l'objectiu primer va ser assolit. Els fitxers de configuració es trobaran dins la carpeta *Food-101/Food-101-ImageNet*. un exemple de predicció seria la Figura 4.10.

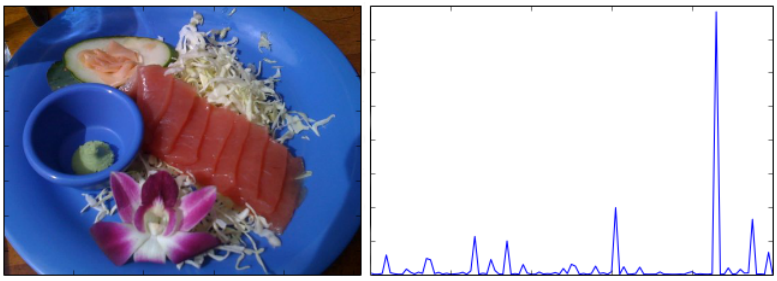


Figura 4.10: Predicció correcte de la classe *sashimi* mitjançant la Food-101-ImageNet. El codi es pot trobar a *TFG_TUTORIAL_Food-101.ipynb*.

A partir d'aquest punt es va començar a realitzar dos intents d'afrontar el problema per tal de tenir més possibilitats de trobar alguna solució que acabés pujant el nivell de reconeixement.

Per una banda, vam començar a preparar les dades per tal d'afegir una nova capa que contingués informació que permetés destacar la textura del menjar i incloure aquesta informació en la quarta capa. Dels filtres a escollir, es va considerar que aplicar un filtre d'entropia sobre les imatges podia ser una bona solució. La col·lecció d'algoritmes en Python de processament d'imatges scikit-image ens va permetre realitzar aquesta tasca mitjançant la seva funció entropy. El resultat va ser el de la Figura 4.11 i el codi es pot trobar en el Notebook *TFG_TUTORIAL_Food-101.ipynb* i les imatges es generarien a *Food-101/Food-*

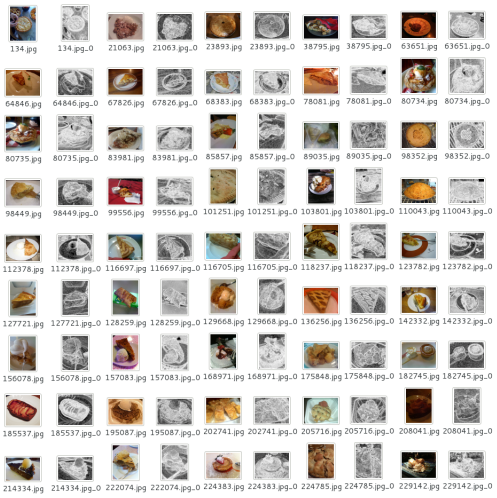


Figura 4.11: Diverses imatges i la seva corresponent capa d'entropia de la classe *apple_pie* de la Food-101.

101-BandaEntropy/images.

Per altra banda, vam recórrer a una tècnica explicada al final del capítol anterior anomenada *Fine-tunning*. Primer de tot, vam preparar els fitxers per a realitzar un *Fine-tuning* usant com a model base el bvlc_reference_caffenet.caffemodel; és a dir, sobre la *BVLC CaffeNet*. Els fitxers es poden trobar dins el directori *Food-101/Food-101-Fine-Tuning-BVLC/*. Una vegada tot estava llest, es va llençar l'execució que va durar unes trenta-vuit hores i quart tal i com es pot comprovar en la sortida de la consola de l'entrenament. El resultat va ser el de la Figura 4.12:

On el valor màxim d'encert va ser un 55.414% que continuava superant el 50.76% que havien tingut els autors de l'article Bossard et al. [2014] mitjançant un *Random Forest*, però no arribava a superar el 56.40% de la CNN entrenada des de 0 amb l'arquitectura de la *ImageNet*.

Una vegada obtinguts aquests resultats, vam passar a comprovar si en cas d'afegir una nova capa que seria el filtre d'entropia i entrenar la xarxa amb l'arquitectura de la *ImageNet* però modificant l'entrada per tal que fos de 4 bandes, donava bons resultats. Després de diversos intents, i atesa la seva complexitat, encara no ens ha estat possible trobar la combinació de paràmetres per tal que

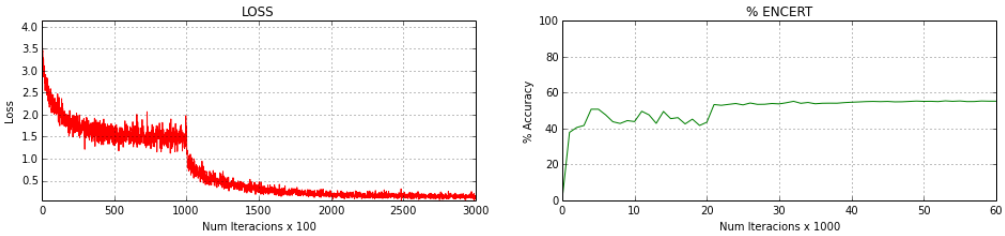


Figura 4.12: Encert i *LOSS* per la Food-101 *Fine-tuning* sobre *BVLC CaffeNet* usant l'arquitectura ImageNet. El codi es pot trobar al Notebook *TFG_TUTORIAL_Food-101.ipynb*

aquesta idea pugui resultar una bona solució. Tot i això, aquesta via també va ser aparcada degut a que es va trobar una solució que sí que va donar molts bons resultats.

Des del departament d'enginyeria de la Universitat d'Oxford un equip va realitzar una nova arquitectura per tal de resoldre l'ImageNet ILSVRC-2014 (Simonyan and Zisserman [2014]) i, segons exposen a la web oficial es van assegurar el primer i el segon lloc mitjançant una xarxa neuronal convolucional que van anomenar *Very Deep ConvNet*. Aquestes dues xarxes que van plantejar contenien 16 i 19 capes i eren considerablement molt més grans que la *BVLC CaffeNet* amb la qual abans havíem realitzat el *Fine-tuning* i semblava que estava donant molt bons resultats. Tot això ens va motivar a provar de realitzar un *Fine-tuning* usant aquesta nova arquitectura.

El primer problema va estar que només teníem el 19_layers_deploy.prototxt però no teníem les dades extres que ens aporta el *solver.prototxt* i el *train_val.prototxt*, així que vam haver de partir dels valors que usàvem en la *ImageNet* i, posteriorment, anar modificant-los buscant un millor resultat.

El segon problema que ens vam trobar era de recursos. Si executàvem sense reduir el nombre de *batch_size* de les capes de tipus *data* ràpidament obteníem un *segmentation fault*. És per això que la solució va ser reduir dels 256 als 8 les dimensions del *batch* tan del test com de l'entrenament i, aleshores, sí que vam poder executar el *Fine-tuning* i el resultat ja va ser molt bo, però encara vam pujar una mica més amb la següent configuració en el *solver.prototxt*

```
net: "train_val.prototxt"
test_iter: 1000
test_interval: 1000
base_lr: 0.0001
lr_policy: "step"
gamma: 0.1
stepsize: 80000
display: 20
max_iter: 400000
momentum: 0.9
weight_decay: 0.0005
snapshot: 10000
snapshot_prefix: "/.../verydeep/snapshot"
solver_mode: GPU
```

I la sortida després de 3 dies, 7 hores i 32 minuts d'execució va ser la següent[1]:

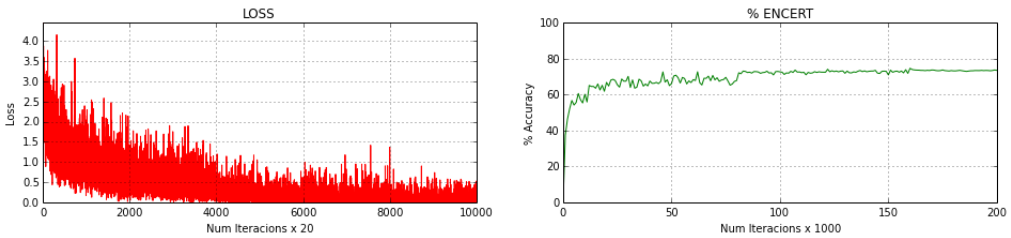


Figura 4.13: Encert i *LOSS* per la Food-101 *Fine-tuning* sobre *VGG_ILSVRC_19_layers* usant la seva arquitectura. El codi es pot trobar al Notebook *TFG_TUTORIAL_Food-101.ipynb*

Per tant, amb això hem complert amb tots els objectius marcats, ja que amb aquest resultat pugem el nivell de reconeixement del *dataset* de la Food-101 del 56.40% a un 74.6834% i podem considerar que aquest resultat és l'estat de l'art per a aquest problema fet públic[2]. De fet, per a comprovar si aquests errors eren

[1]Com podem observar, l'execució la mostrem fins les 200.000 iteracions ja que portàvem massa iteracions sense millorar l'*accuracy* i estàvem patint *overfitting*.

[2]La companyia MetaMind assegura, sense donar gaires detalls de com ho han realitzat, que ells han obtingut un classificador que assoleix el 79%. Per a més informació vegis la pàgina

molt menors si miràvem les n posicions següents en les anteriors proves, havíem realitzat un codi per a mirar quin tant per cent d'encert realitzaria si pogués dir-ne n. Si aquest test el passem amb la *snapshot* que conservem de l'iteració 200.000, tot i tenir una mica menys del resultat màxim d'encert (situat a les 160.000 iteracions), obtenim la següent sortida:

```
Encert amb 1 oportunitat: 68.0332739156 %
Encert amb 2 oportunitats: 78.0352545058 %
Encert amb 3 oportunitats: 82.788671024 %
Encert amb 4 oportunitats: 85.6110120816 %
Encert amb 5 oportunitats: 87.5816993464 %
```

Per tant, ens podríem plantejar, ja amb aquest model, realitzar una aplicació per a dispositius portàtils que permetés, donada una imatge realitzada amb una fotografia del plat que l'usuari va a menjar, proposar-li n opcions del plat més possible i deixant aquest marge d'error perquè l'usuari decidís de quin plat es tracta. Una vegada se sabés el plat, una idea podria ser calcular aproximadament el nombre de calories que l'usuari ingerirà o altra informació de tipus nutricional, efectes terapèutics, procedència, composició d'ingredients, forma de cocció, vinculació amb una determinada cultura, variacions més conegudes, etc. que pugui ser interessant.

web.

Capítol 5

Resultats, conclusions i línies de continuació

Aquest treball ha tingut tres parts importants. La primera, una introducció i comprensió sobre molts conceptes nous referents a un mètode d'aprenentatge automàtic artificial basat en el model neuronal: les xarxes neuronals. Tot per aprofundir en una branca de les NN molt motivant i esperançadora anomenada la xarxa neuronal profunda, i per després centrar-nos en les xarxes neuronals convolucionals per tal de passar a la segona part. La segona part, ha estat un anàlisi de com poder transformar els conceptes apresos en execucions mitjançant el *Framework* Caffe. I finalment, la tercera part ha estat una recopil·lació de tot allò aprés per tal de repetir bons sistemes de classificació actuals, realitzar algunes modificacions del *Framework* Caffe perquè acceptés nova informació i proposar una millora en algun problema complex en concret.

Podem considerar que els objectius plantejats han estat complerts ja que hem modificat el *Framework* Caffe de manera satisfactòria i, per exemple, per al cas de la MNIST el resultats milloraven. Però sobretot, el resultat a remarcar seria que ara tenim un sistema de classificació del 74.6834% per a la Food-101 quan en l'article Bossard et al. [2014] havien arribat al 56.40%.

En definitiva, hem obtingut un resultat apreciable. Per a posar la marca més amunt, i veient com funciona la xarxa amb aquest *dataset*, podríem plantejar tres nivells de possibles millores: jugar amb els paràmetres, fer un petit canvi d'arquitectura, o plantejar el problema des del punt de vista de les xarxes recurrents.

Les línies concretes a seguir podrien ser:

- Incrementar el *batch_size* i re-entrenar amb *Fine-tuning* (cosa que requeriria més espai de GPU).
- Incrementar la base de dades d'imatges Food-101.
- Proseguir amb la idea d'inserir més capes d'informació a partir de filtres aplicats a la imatge i, a la vegada, trobar uns paràmetres adequats per a la xarxa.
- Com acabem de veure amb la Very-Deep, sembla que la profunditat de la xarxa té interès en l'aprenentatge de la classificació. Per tant, podríem realitzar el procés de *Fine-tuning* sobre altres models més grans com la BVLC_GoogleNet, tot i que, tornaríem a requerir molt més espai de GPU.
- Realitzar canvis en l'arquitectura com la NIN (Network in a Network) que consisteix en substituir algunes capes convolucionals per petites MLP (capes de perceptrons de múltiples capes), ja que hi han equips, com Lin et al. [2013], que han aconseguit igualar resultats amb menys capes. És a dir, aquest punt consistiria en explorar variants de les capes convolucionals.
- Estudiar i aprofundir amb les xarxes neuronals recurrents, ja que se sap que en la visió humana les xarxes neuronals són recurrents i en aquest treball sempre hem estat plantejant xarxes del tipus *feed-fordward*. Tot i això, avui dia encara no està clar com construir aquest tipus de xarxes i que siguin eficients. Per tant, una de les recerques que es podria plantejar seria què pot aportar una xarxa recurrent sobre problemes d'aquest tipus.

References

Itamar Arel, Derek C Rose, and Thomas P Karnowski. Deep machine learning-a new frontier in artificial intelligence research [research frontier]. *Computational Intelligence Magazine, IEEE*, 5(4):13–18, 2010. 22

Sven Behnke. *Hierarchical neural networks for image interpretation*, volume 2766. Springer, 2003. 2

Lukas Bossard, Matthieu Guillaumin, and Luc Van Gool. Food-101 – mining discriminative components with random forests. In *European Conference on Computer Vision*, 2014. 34, 40, 41, 43, 47

Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, et al. Dadiannao: A machine-learning supercomputer. *Proceedings of the 47th IEEE/ACM International Symposium on Microarchitecture, (MICRO'14)*, 2014. 22, 23

Kunihiko Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological cybernetics*, 36(4):193–202, 1980. 2

Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier networks. In *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics. JMLR W&CP Volume*, volume 15, pages 315–323, 2011. 21

Geoffrey E Hinton. Learning multiple layers of representation. *Trends in cognitive sciences*, 11(10):428–434, 2007. 3

Geoffrey E Hinton. Dropout: A simple and effective way to improve neural networks. 2013. 19

Geoffrey E Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012. 20

Md Monirul Islam and Kazuyuki Murase. A new algorithm to design compact two-hidden-layer artificial neural networks. *Neural Networks*, 14(9):1265–1278, 2001. 11

Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014. 22

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. 24

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11): 2278–2324, 1998. 2

F.H.F. Leung, H.K. Lam, S.H. Ling, and P.K.S. Tam. Tuning of the structure and parameters of a neural network using an improved genetic algorithm. *Neural Networks, IEEE Transactions on*, 14(1):79–88, Jan 2003. ISSN 1045-9227. doi: 10.1109/TNN.2002.804317. 11

Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. *CoRR*, abs/1312.4400, 2013. URL http://arxiv.org/abs/1312.4400. 48

Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943. 1

Marvin Minsky and Seymour Papert. Perceptron: an introduction to computational geometry. *The MIT Press, Cambridge, expanded edition*, 19:88, 1969. 2

Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015. 7, 12, 19

Gualtiero Piccinini. The first computational theory of mind and brain: a close look at mcculloch and pitts's "logical calculus of ideas immanent in nervous activity". *Synthese*, 141(2):175–215, 2004. 1

Marc'Aurelio Ranzato. Deep learning for object category recognition. 2014. URL http://cvgl.stanford.edu/teaching/cs231a_winter1314/lectures/lecture_guest_ranzato.pdf. 4, 22

Frank Rosenblantt. The perceptron, a perceiving and recognizing automation. 1957. 2

Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958. 2, 6

David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Cognitive modeling*, 1988. 13

Patrice Y Simard, Dave Steinkraus, and John C Platt. Best practices for convolutional neural networks applied to visual document analysis. In *2013 12th International Conference on Document Analysis and Recognition*, volume 2, pages 958–958. IEEE Computer Society, 2003. 2

K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014. 44

M. Sundermeyer, I. Oparin, J.-L. Gauvain, B. Freiberg, R. Schluter, and H. Ney. Comparison of feedforward and recurrent neural network language models. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 8430–8434, May 2013. doi: 10.1109/ICASSP.2013.6639310. 11

Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pages 1139–1147, 2013. 30

Jinn-Tsong Tsai, Jyh-Horng Chou, and Tung-Kuan Liu. Tuning the structure and parameters of a neural network by using hybrid taguchi-genetic algorithm. *Neural Networks, IEEE Transactions on*, 17(1):69–80, Jan 2006. ISSN 1045-9227. doi: 10.1109/TNN.2005.860885. 11

Evan Shelhamer Jeff Donahue Yangqing Jia, Ross Girshick. Diy deep learning for vision: a hands-on tutorial with caffe. 2014. URL https://docs.google.com/presentation/d/1UeKXVgRvvxg9OUdh_UiC5G71UMscNPlvArsWER41PsU/preview. 2, 3, 25

Corinna Cortes Yann LeCun and Christopher J.C. Burges. The mnist database of handwritten digits. 1998. URL http://yann.lecun.com/exdb/mnist/. 10, 34