TFG – Arquitectura de computadors i sistemes operatius

FITA#05: Disseny de l'arquitectura de xarxes neuronals

Gestió del projecte a Github:

Branca del repositori:
https://github.com/UOC-Assignments/uoc.tfg.jbericat/tree/FITA%2305

Dashboard de seguiment de les tasques associades a la fita:
https://github.com/UOC-Assignments/uoc.tfg.jbericat/projects/5

Estudiant: Jordi Bericat Ruz

Professor col·laborador: Daniel Rivas Barragan

Semestre: Tardor 2021/22 (Aula 1)

Versió: ESBORRANY_v9

Índex

5 - Disseny de l'arquitectura de xarxes neuronals 1
5.1 - Tasques d'investigació i recerca 1
5.1.1 - Repàs cronològic dels models de Deep Learning més destacats i estudi comparatiu de les seves característiques d'alguns d'ells 2
5.1.1.1 – AlexNet 9
5.1.1.2 – DenseNet121 11
5.2 - Estructuració de l'algorisme de Deep Learning 13
5.2.1 - Preparació de les dades per a l'entrenament del model 13
5.2.1.1 - Pre-processament de les imatges del dataset 13
5.2.1.2 - Divisió dels dataset en grups 18
5.2.2 – Definició d'un model base (baseline model) 19
5.2.2.1 – Versió 1.0 21
5.2.2.2 – Versió 2.0 21
5.2.2.3 – Versió 3.0 21
5.2.3 – Càlcul d'hyperparàmetres 22
5.2.3.1 – Versió 1.0 24
5.2.3.2 – Versió 2.0 25
5.2.3.3 – Versió 3.0 27
5.2.4 – Implementació de l'estructura del model CNN amb el framework pyTorch 30
5.2.4.1 – Versió 1.0 31
5.2.4.2 – Versió 2.0 32
5.2.4.3 – Versió 3.0 33
5.2.5 – Definició de la funció de pèrdua (loss function) 34
5.3 – *Wrapping-Up*: Implementació d'un algorisme d'entrenament i generació de gràfiques i estadístiques 35
5.4 – ANNEX OPCIONAL: Possibles millores i optimitzacions del model 37
5.4.1 - Dropout regularization 37

5 - Disseny de l'arquitectura de xarxes neuronals

5.1 - Tasques d'investigació i recerca

Donat que la implementació d'un model CNN específic per a resoldre el problema proposat a la PdC d'aquest projecte no és una tasca gens trivial i que a més sobrepassa en complexitat els objectius marcats per al TFG, es decideix fer recerca de diferents arquitectures i metodologies d'entrenament i implementació ja existents per a avaluar si ens poden ser d'utilitat i així evitar haver de dissenyar una xarxa neuronal "from scratch". En aquest sentit, s'ha trobat que els capítols 4 i 5 de la referència bibliogràfica [LLIBRE DL] ens poden servir de guió per a, d'una banda, preparar les dades (*datasets* imatges) per a entrenar model, i de l'altra, per a fer un estudi de la evolució de les xarxes neuronals des de les primeres implementacions fins l'actualitat. Aquest estudi ens ajudarà posteriorment a avaluar i seleccionar una arquitectura ja existent per al seu ús en la PdC.

A més, s'ha complementat la informació obtinguda de la font anterior amb les que ofereixen els següents enllaços:

- https://docs.microsoft.com/en-us/windows/ai/windows-ml/tutorials/pytorch-train-model
- https://deeplizard.com/learn/video/IKOHHItzukk
- https://towardsdatascience.com/convolutional-neural-networks-explained-9cc5188c4939
- https://www.pyimagesearch.com/2021/05/14/convolutional-neural-networks-cnns-and-layer-types/
- https://setosa.io/ev/image-kernels/

5.1.1 - Repàs cronològic dels models de Deep Learning més destacats i estudi comparatiu de les seves característiques d'alguns d'ells

En primer lloc es procedeix a realitzar un estudi comparatiu superficial (taula 5.X.X.X) de les característiques d'aquelles arquitectures de xarxes neuronals profundes més conegudes fins a dia d'avui[1], per tal d'escollir aquella que s'ajusti millor a les dimensions del problema que volem resoldre i a als recursos computacionals dels que es disposa[2] (estació de desenvolupament descrita a la secció 1.x.x.x). Com a criteris per a realitzar l'estudi comparatiu es tindran en compte 3 factors; d'una banda, la precisió assolida pel model mesurada en l'escala "Top-1". De l'altra, la complexitat computacional en termes d'instruccions per segons (en G-FLOPS) requerides per a entrenar el model, i finalment, la quantitat de nodes (paràmetres) de cada arquitectura. La figura 5.x.x.x cataloga segons aquests criteris les DCNN més destacades fins a 2018:

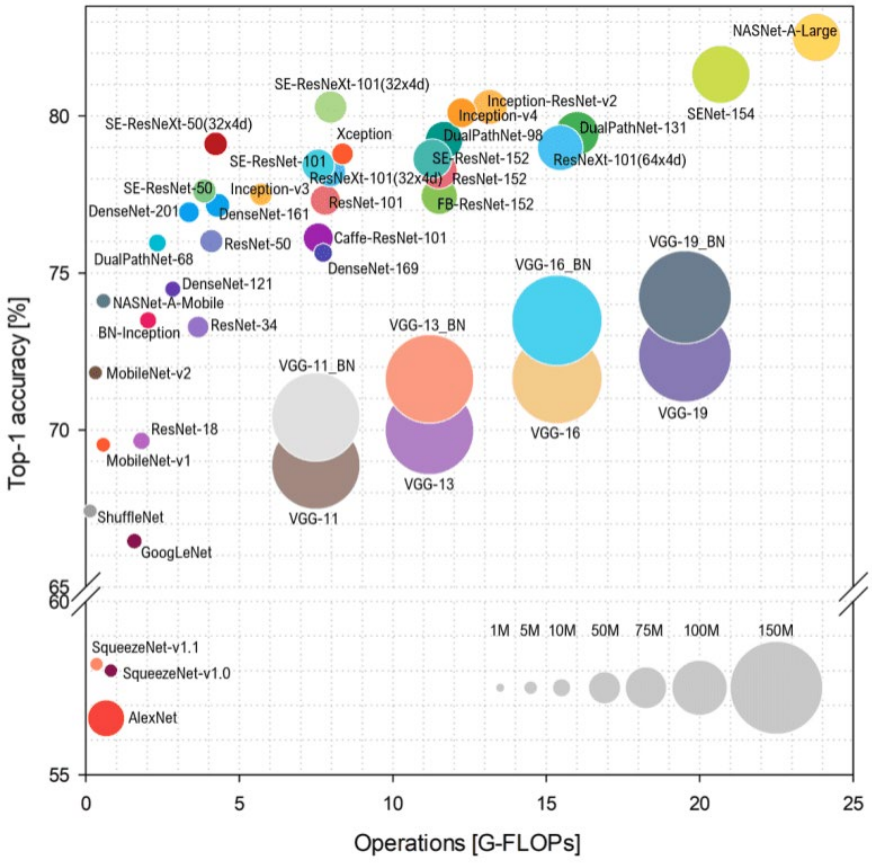


Figura 5.x.x.x – Visió general d'arquitectures DCNN fins a 2018 – Font: https://theaisummer.com/cnn-architectures/

[1] *Referències bibliogràfiques: Capítol 5 Llibre DL, pàgs: 199, https://theaisummer.com/cnn-architectures/*

[2] *Tanmateix, per a l'entrenament del model també es podria fer ús de plataformes tercers com per exemple "Google AI platform" → https://cloud.google.com/ai-platform/training/docs/using-gpus*

Arquitectura DCNN	Característiques principals	Numero de Paràmetres	Anàlisi de rendiment (Rati Top-5)	Complexitat temporal (G-FLOPs)
LeNET	L'arquitectura està composta per 5 capes, 3 de convolucionals que s'encarreguen de la extracció de característiques de les imatges, i dues de completes que implementen la classificació.	61K	Presenta un bon rendiment (fins a un 99%) amb datasets d'**imatges en escala de grisos** i aplicades a **problemes de classificació que no superin les 10 classes**[3].	Baixa
AlexNET	Tot i oferir una arquitectura semblant a LeNET, AlexNET implementa noves característiques i una major quantitat de capes convolucionals (5)[4] que li proporcionen una **capacitat d'aprenentatge major** que la primera en termes de complexitat de les característiques que pot arribar a reconèixer[5].	60M	En problemes de classificació, el rendiment que ofereix AlexNet mesurat en escala Top-5[6] és del 15,3%, valor que supera la precisió d'encert d'altres models anteriors al seu desenvolupament, pels volts de l'any 2012.	Baixa
VGGNet16	Comparada amb les arquitectures vistes fins al moment, VGGNet destaca per la seva arquitectura uniforme i quantitat mes reduïda	138M	Si utilitzem el ratio Top-5 amb el que s'ha mesurat el rendiment de l'arquitectura AlexNet, podem destacar	Alta

[3] *Ref. P203 llibre DL → "LeNet performance on the MNIST dataset: When you train LeNet-5 on the MNIST dataset, you will get above 99% accuracy"*

[4] *Ref. P205 llibre DL → "AlexNet consists of five convolutional layers, some of which are followed by max-pooling layers, and three fully connected layers with a final 1000-way softmax"*

[5] *Ref. P204 llibre DL → "AlexNet has about 60 million parameters and 650,000 neurons, which gives it a larger learning capacity to understand more complex features. This allowed AlexNet to achieve remarkable performance"*

[6] *El ratis Top-1 I Top-5 són utilitzats de manera habitual per a descriure la precisió d'un algorisme de classificació, de manera que el rati Top-1 representa la proporció de vegades en funció del temps que el classificador no ha proporcionat la classe que s'esperava, mentre que el rati Top-5 indica el percentatge vegades que la resposta correcta no estava entre les 5 primeres propostes de predicció del model. Refs: P211 Llibre DL*

	de paràmetres per capa[7], que la fan més fàcil d'entendre i gestionar[8]. Tot i aquesta uniformitat, els components que formen l'arquitectura són els mateixos que els vistos a les anteriors, però en aquest cas trobem una major profunditat de la xarxa: més capes convolucionals (13) i de completes (3), fet que fa augmentar significativament el nombre de paràmetres totals del model.		una **important millora del rendiment** de VGG16 respecte del primer, ja que utilitzant el mateix dataset d'imatges aconsegueix una ratio del 8.1%.	
GoogLeNet	Aquesta arquitectura es caracteritza per ser de les que ofereix una major profunditat amb 22 capes convolucionals. Tanmateix, tot i estar basada en les arquitectures vistes anteriorment (AlexNet i VGGNet), la quantitat de paràmetres que requereix es molt inferior.	13M	Seguint el mateix criteri que amb les arquitectures anteriors, podem dir que GoogLeNet és la DCNN que ha aconseguit un millor rati Top-5 fins la data del seu desenvolupament, assolint valors molt baixos de fins el 6.67% que denota un rendiment del model aproximat al que ofereix el cervell humà (pel que fa a resoldre problemes de classificació).	Baixa
ResNet	L'arquitectura de xarxes neuronals residuals desenvolupada per Microsoft *ResNet* es caracteritza per introduir una tècnica de normalització per lots que li permet	Entre 5M i 20M, depenent de la	ResNet és capaç d'assolir un ratio Top-5 mínim de 5,25% en les mateixes condicions que les altres arquitectures vistes.	Moderada

[7] ***Ref. P214 llibre DL →*** *"VGG16 yields ~138 million parameters; VGG19, which is a deeper version of VGGNet, has more than 144 million parameters. VGG16 is more commonly used because it performs almost as well as VGG19 but with fewer parameters. VGGNet, has more than 144 million parameters. VGG16 is more commonly used because it performs almost as well as VGG19 but with fewer parameters."*

[8] ***P210 llibre DL →*** *"Both LeNet and AlexNet have many hyperparameters to tune. The authors of those networks had to go through many experiments to set the kernel size, strides, and padding for each layer, which makes the networks harder to understand and manage. VGGNet (explained next) solves this problem with a very simple, uniform architecture".*

	augmentar la profunditat de la xarxa (fins a 152 capes a les versions més profundes) tot reduint a l'hora la seva complexitat computacional i augmentant el seu rendiment[9].	implementació		
DenseNet121/264	Treballs posteriors al desenvolupament de ResNet[10] han sigut capaços de demostrar que habilitant la interconnexió de les diferents capes del model entre elles (especialment cap aquelles més properes a la entrada i sortida de l'arquitectura) és possible augmentar la profunditat de la xarxa (des de 121 fins a 201) i en conseqüència la precisió de les prediccions sense que això impliqui un increment de la quantitat de paràmetres i de la complexitat, com és el cas de DenseNet	Entre 5M i 50M, depenent de la implementació	ResNet és capaç d'assolir un ratio Top-5 mínim d'entre 5.29% i el 7.71%, depenent de la implementació[11].	Moderada

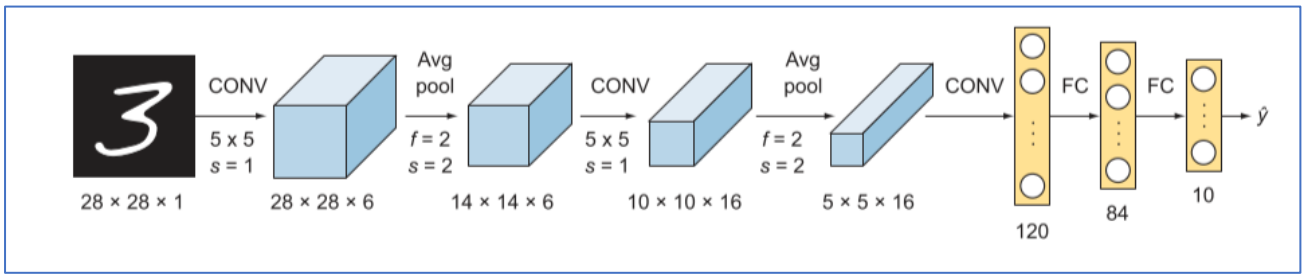
Taula 5.X.X.X: Comparativa d'algunes de les arquitectures DCNN avançades més destacades desenvolupades des de 1998 fins a 2018

[9] ***Refs. P230 llibre DL*** → *"The Residual Neural Network (ResNet) was developed in 2015 by a group from the Microsoft Research team.5 They introduced a novel residual module architecture with skip connections. The network also features heavy batch normalization for the hidden layers. This technique allowed the team to train very deep neural networks with 50, 101, and 152 weight layers while still having lower complexity than smaller networks like VGGNet (19 layers). ResNet was able to achieve a top-5 error rate of 3.57% in the ILSVRC 2015 competition, which beat the performance of all prior ConvNets"*

[10] *https://arxiv.org/abs/1608.06993* → *"Recent work has shown that convolutional networks can be substantially deeper, more accurate, and efficient to train if they contain shorter connections between layers close to the input and those close to the output."*

[11] *https://towardsdatascience.com/review-densenet-image-classification-b6631a8ef803*

A continuació es mostra el diagrama de cadascuna de les arquitectures estudiades a la taula 5.x.x.x. Cal destacar que amb l'ajuda d'aquests, a simple vista és possible arribar a fer-se una idea de la profunditat (i per tant de la precisió) de cada model:



Imatge obtinguda de [LLIBRE DL, Pàg.]

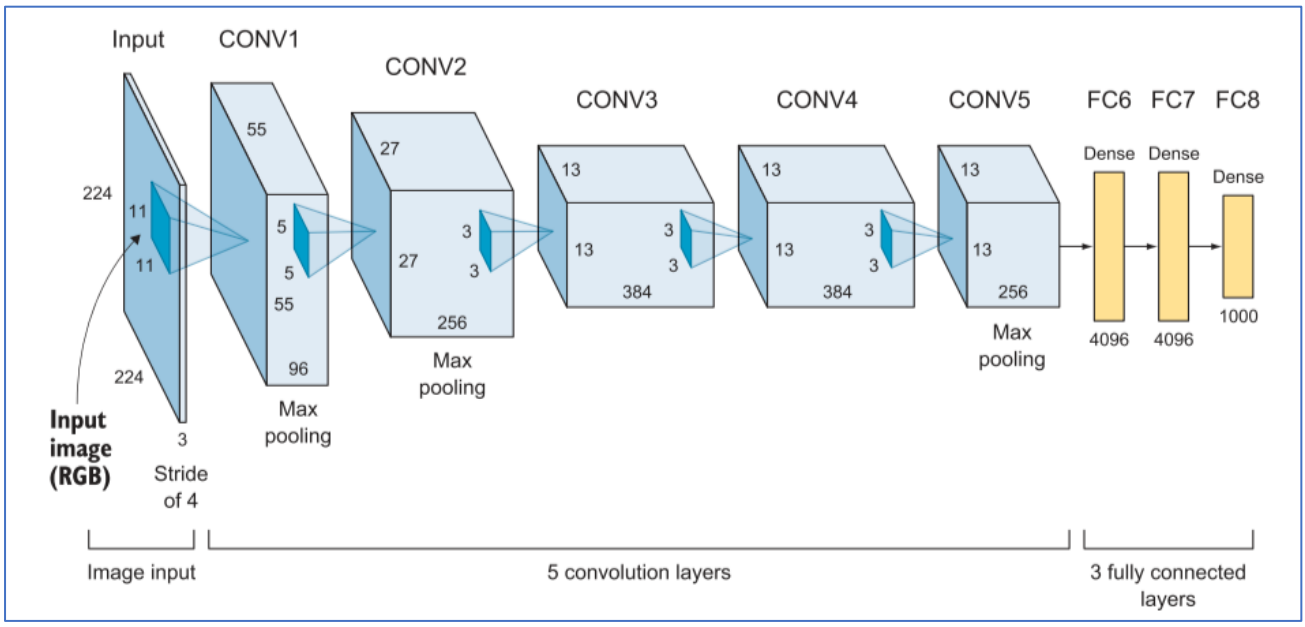


Figura 5.xxx – Arquitectura AlexNet – Font: [LLIBRE DL, Pàg.]

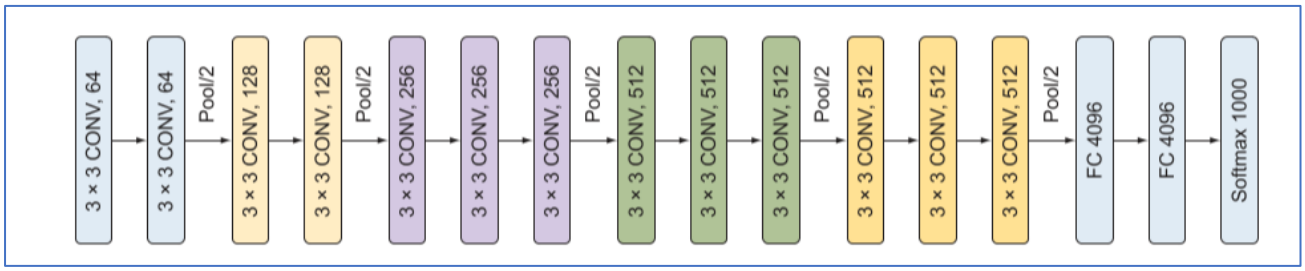


Figura 5.xxx – Arquitectura VGGNet16 – Font: [LLIBRE DL, Pàg.]

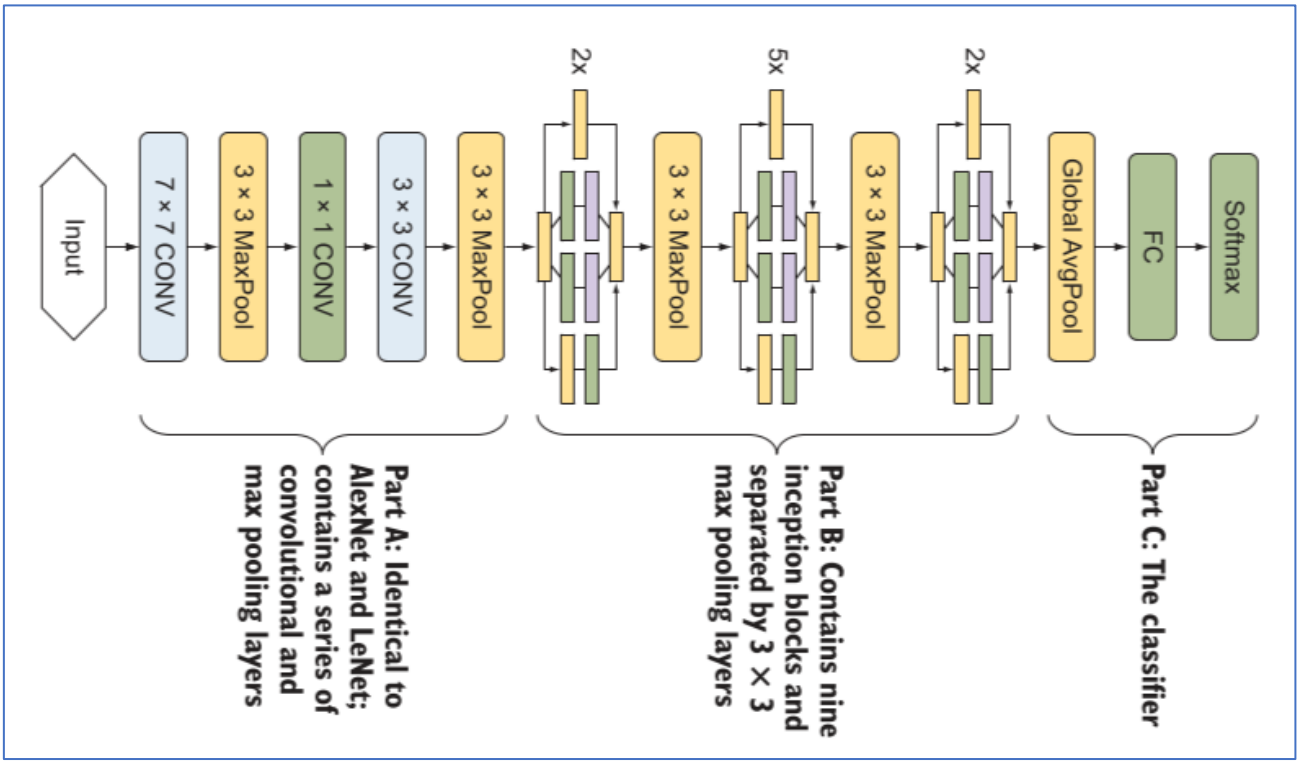


Figura 5.xxx – Arquitectura GoogLeNet – Font: [LLIBRE DL, Pàg.]

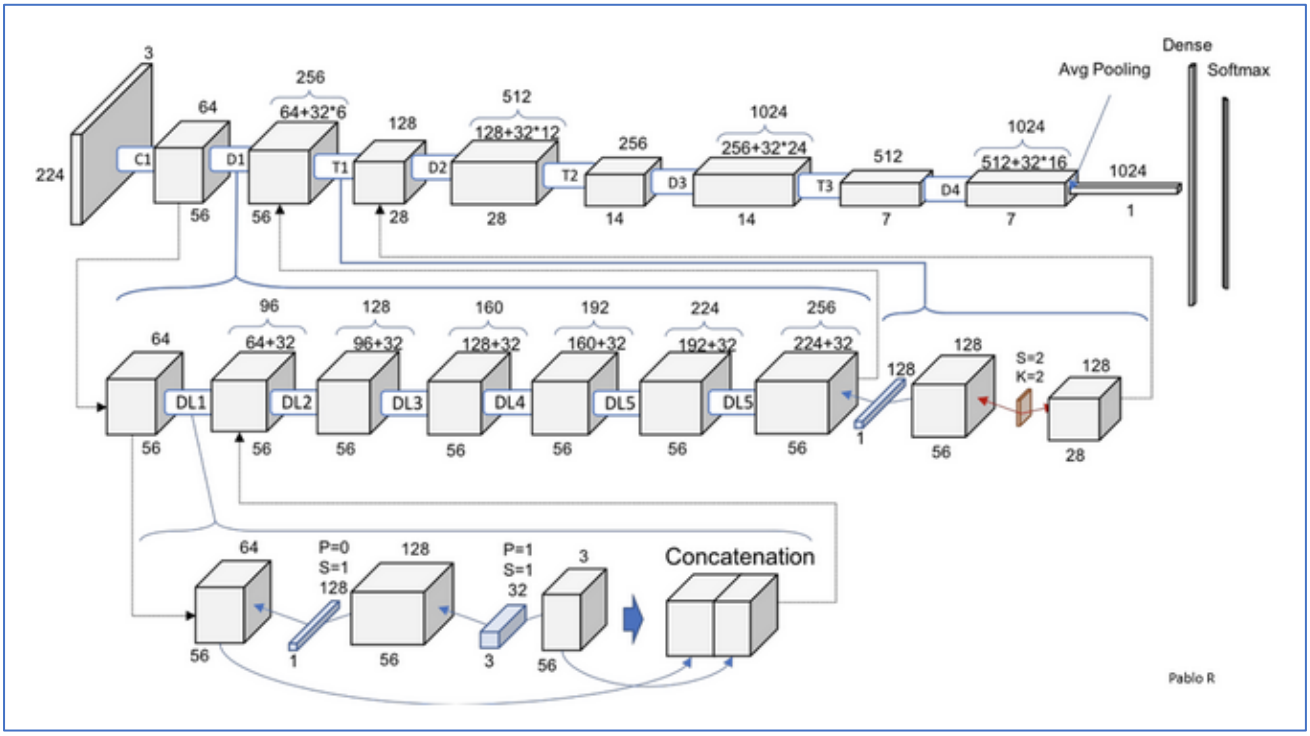


Figura 5.xxx – Arquitectura DenseNet - Font: Pablo R.

Un cop vistes les principals arquitectures dels darrers anys a la taula 5.x.x.x, seguidament es procedeix a fer un breu estudi de les característiques més destacades d'algunes d'elles. Això ens ajudarà més endavant a entendre la finalitat d'afegir algunes capes en particular, quan implementem l'arquitectura escollida (veure secció 5.3). Per a fer l'estudi, de partida es descarten aquelles que presenten una complexitat d'implementació molt elevada (GoogLeNet i ResNet queden descartades en aquest sentit). D'altra banda, degut a la seva obsolescència (data del 1998), també es descarta LeNet. Així doncs, passarem a realitzar un breu estudi de dos dels tres models restants (AlexNet i DenseNet). Pel que fa AlexNet, tot i ser una model obsolet, aquest s'inclou en l'estudi per qüestions didàctiques, ja que és un bon punt de partida per a entendre el funcionament d'arquitectures més avançades, com és el cas de DenseNet. D'aquesta manera també podrem realitzar un anàlisi comparatiu de com ha sigut la evolució de les DCNN durant la darrera dècada. En aquest sentit, més endavant (a la secció 6) estudiarem el nivell de complexitat de configuració dels hyper-paràmetres vs. complexitat computacional necessària per a processar els dataset vs. rendiment obtingut, tot fent ús de les mateixes mètriques que es definiran a la mateixa secció.

5.1.1.1 – AlexNet

El model AlexNet, amb 60 milions de paràmetres i al voltant de 650.000 neurones és una de les arquitectures que en el moment del seu desenvolupament va causar un impacte més significatiu, degut a que va permetre demostrar que la utilització de xarxes neuronals per a resoldre problemes de classificació d'una certa complexitat aplicats a visió per computador[12]. Concretament, AlexNet està capacitat per a classificar una imatge d'entrada d'entre fins a 1000 classes diferents.

Arquitectura:

AlexNet implementa una arquitectura de xarxa neuronal formada per 5 capes convolucionals per a la extracció de característiques (mitjançant una sèrie de Kernels de mida determinada, en funció de la capa) i 3 de completes, que tenen com a sortida la funció de classificació softmax. D'altra banda, AlexNet també incorpora un seguit de capes de "*pooling* màxim d'encavalcament" (Overlapping Max Pooling) que ajuden a reduir lleugerament el coeficients Top-1 i Top-5 de predicció.

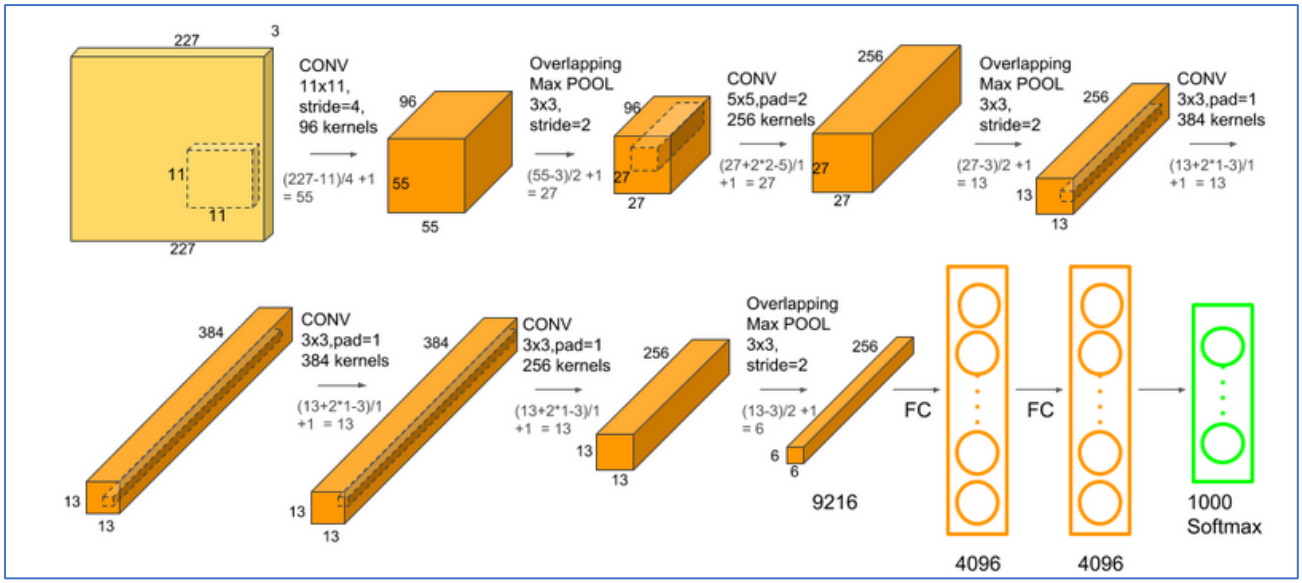


Figura 5.x.x.x *– Arquitectura del model AlexNet – Font: https://learnopencv.com/understanding-alexnet/*

[12] *https://learnopencv.com/understanding-alexnet/ → [...] with the publication of the paper, "ImageNet Classification with Deep Convolutional Neural Networks" by Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton, he and a handful of researchers were proven right. It was a seismic shift that broke the Richter scale! The paper forged a new landscape in Computer Vision by demolishing old ideas in one masterful stroke.*

Característiques destacades:

Un dels trets diferencials del model AlexNet respecte dels seus predecessors l'any 2012 i que va permetre accelerar l'entrenament de xarxes relativament profundes va ser la utilització d'una funció d'activació[13] *ReLu* (*Rectified Linear unit*), que permet una convergència de la xarxa molt més ràpida que altres funcions d'activació utilitzades amb anterioritat, com *tanh* o *sigmoid*. Podem representar la funció d'activació ReLu de la següent manera[14]:

$$f(x) = \max(0, x)$$

On es defineix que f(x) tindrà com a sortida el major valor entre 0 i x. Ho podem veure clar si ho expressem com una funció definida a trossos:

$$f(x) = \begin{cases} 0, si\ x < 0 \\ x, si\ x \geq 0 \end{cases}$$

Observem que la sortida (imatge de x) serà igual a zero (FALSE) sempre i quan la entrada (x) sigui un valor negatiu. En cas contrari, tindrem que la sortida serà la mateixa entrada, és a dir "x". Si ens fixem en la gràfica de les funcions ReLu i tanh podem veure que a la segona, el pendent de la recta per a valors en l'interval (-inf, -2) U (2, +inf) és molt pròxim a zero, factor que pot decelerar el procés de descens gradual (*gradient descent*) per l'arquitectura de capes.

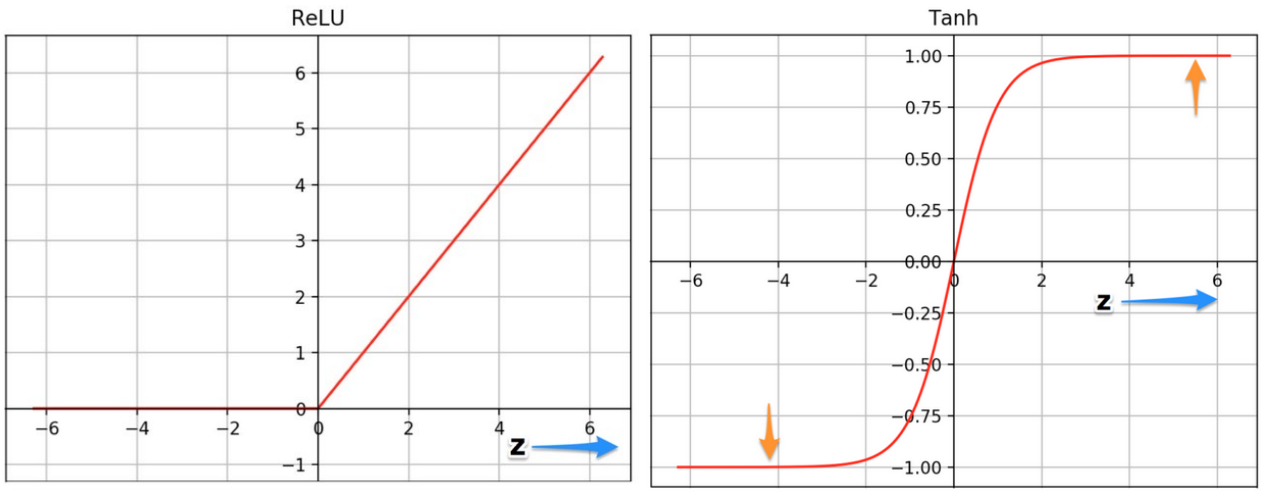


Figura 5.xxx - Imatges obtingudes de https://learnopencv.com/understanding-alexnet/

[13] *Recordem que les funcions d'activació s'encarreguen de determinar si s'ha reconegut o no una característica (resultat binari) DESENVOLUPAR I CERCAR REFS.*

[14] https://deepai.org/machine-learning-glossary-and-terms/relu

Altres característiques rellevants del model:

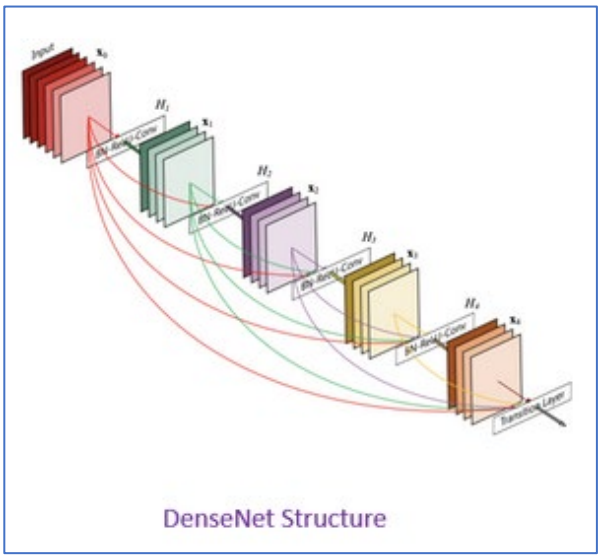
Hi ha una sèrie de factors que hem de tenir en consideració a l'hora d'adaptar les imatges del dataset d'aquesta PdC i poder utilitzar-lo per a l'entrenament del model AlexNet:

a) **Mida** → Les imatges del dataset han de ser convertides a mida 256x256 mitjançant la funció de pre-processament corresponent (veure secció 5.2.3.1)

b) **Color** → Les imatges han de ser en RGB (3 canals), però si s'alimenta el model amb imatges en escala de grisos (com es el cas), aleshores la mateixa implementació de l'arquitectura s'encarrega de replicar el canal únic de l'escala de grisos a tres matrius exactament iguals.

5.1.1.2 – DenseNet121[15]

El desenvolupament de DenseNet data del 2018 (5ª revisió)[16] i va estar motivat per les limitacions d'escalabilitat d'altres xarxes de la mateixa generació com ResNet, amb la que s'efectua una degradació progressiva de la informació degut al llarg recorregut existent entre la capa d'entrada i la de sortida (fins a 150 capes intermèdies o ocultes). Per a aconseguir aquest increment de profunditat, DenseNet utilitza la concatenació de dades a la sortida de cada capa amb la de totes les capes següents, assolint la forma d'un graf amb un total de $\frac{L\cdot(L+1)}{2}$ arestes i L nodes, que representen les capes del model (5 de convolucionals i de "pool", 3 de transició, una de classificació i 2 "blocs densos"). La figura 5.xxx es mostra una proposta simplificada de representació de l'arquitectura:



Tanmateix, la utilització de la concatenació de dades només es possible si les dimensions dels mapes de característiques[17] són les mateixes per a tot el model, fet pel qual DenseNet implementa

[15] *Referències bibliogràfiques i snippets de codi: https://www.pluralsight.com/guides/introduction-to-densenet-with-tensorflow*

[16] *https://arxiv.org/abs/1608.06993*

[17] FER RECERCA AVIAM A QUE ES REFEREIX AIXÒ

la segmentació de capes en blocs amb filtres (Kernels) de mides diferents entre capes que pertanyen a diferents blocs, però de la mateixa mida per a capes d'un mateix bloc. El diagrama de la figura 5.xxx mostra de manera més detallada l'arquitectura de blocs del model DenseNet:

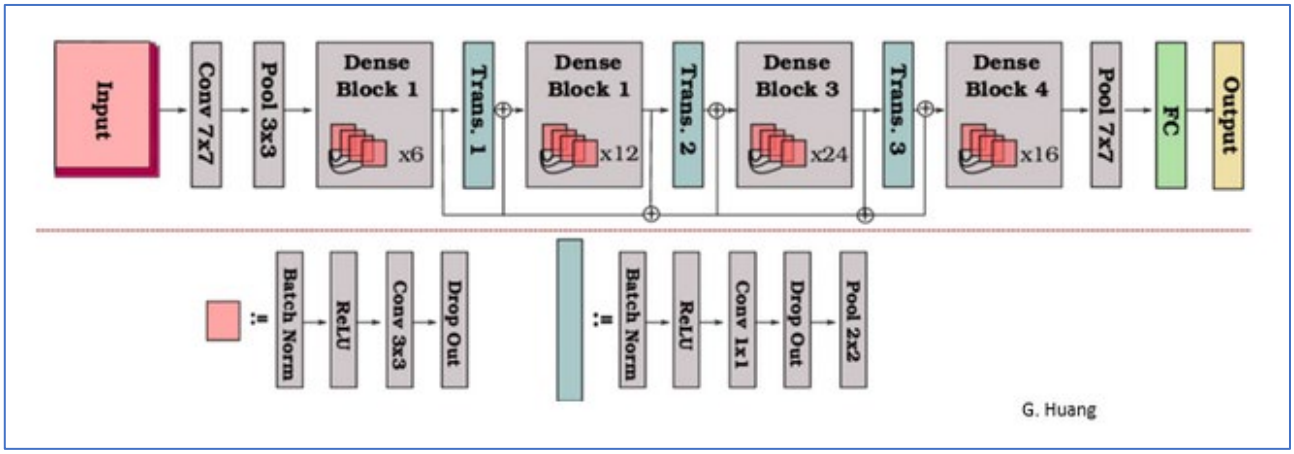


Figura 5.x.x.x (Font: G. Huang, Z. Liu and L. van der Maaten, "Densely Connected Convolutional Networks," 2018)

Pel que fa les aplicacions del model, s'ha pogut demostrar que la característica diferencial de **reusabilitat de les característiques (features) de les dades** que aquesta arquitectura ofereix possibilita la seva aplicació en un ventall ampli d'àmbits que va més enllà de la classificació d'imatges[18], com es pot veure a la figura de la esquerra.

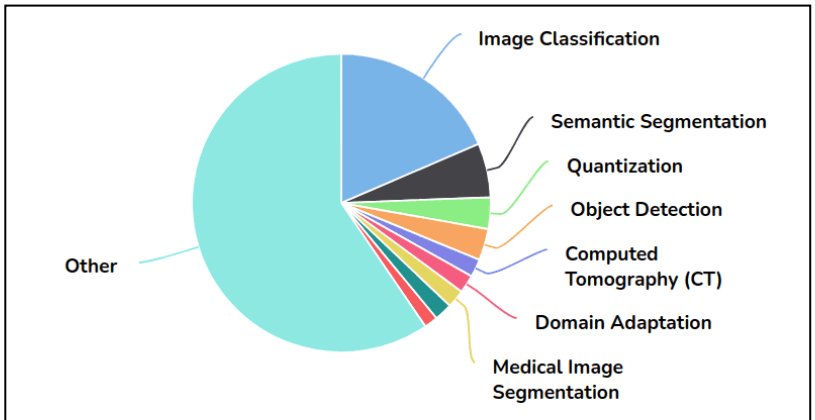


***Figura 5.x.x.x** – Àmbits d'aplicació del model DenseNet*
Font: https://paperswithcode.com/method/densenet

[18] *Font: https://theaisummer.com/cnn-architectures/ → "Even though DenseNet was proposed for image classification, it has been used in various applications in domains where feature reusability is more crucial (i.e. segmentation and medical imaging application)."*

5.2 - Estructuració de l'algorisme de Deep Learning

5.2.1 - Preparació de les dades per a l'entrenament del model

Abans de procedir a realitzar l'entrenament del model escollit a la secció anterior (5.2.2 – Disseny d'un model base), s'ha de:

a) Pre-processar els dataset d'imatges "en cru o *raw"* que s'han generat com a resultat de les accions realitzades a la secció 4 d'aquest document
b) Dividir els mateixos dataset en 3 grups: Entrenament, validació i explotació

Tot seguit es detalla en què consisteixen les accions de preparació anterior, així com la manera com s'aplicaran a les dades de què disposem per a la PdC.

5.2.1.1 - Pre-processament de les imatges del dataset

El pre-processament i "neteja" de les dades que alimentaran el model de xarxes neuronals es un pas previ prescindible quan es tracta d'entrenar models DL, però important si es tracta de refinar al màxim la seva velocitat d'aprenentatge i rendiment. Per tant aplicarem les estratègies més habituals de preprocessament a les imatges dels dataset de la PdC.

a) Imatges en color vs. escala de grisos

Coneixent que per a representar una imatge en color RGB amb una estructura de dades calen tres matrius (una per a cada color base), mentre que una imatge en escala de grisos la podem representar amb una sola matriu, aleshores podem concloure que utilitzant imatges en escala de grisos "estalviarem" complexitat computacional a l'hora d'entrenar el model, ja que la quantitat de paràmetres que haurem de configurar serà molt menor (això és, els pesos de les d'arestes del graf que formen els nodes de les diferents capes ocultes de la xarxa neuronal profunda). En el cas d'aquesta PdC, com que ja estem utilitzant imatges que simulen visió IR tèrmica (veure secció 3.x.x.x) i aquestes estan convertides al B/N per a aconseguir una simulació més fidel, aleshores no haurem de prendre cap acció i simplement considerarem que les imatges ja han sigut pre-processades en aquest sentit. Cal notar que la utilització d'imatges en escala de grisos sempre serà convenient si es poden reconèixer les seves característiques utilitzant la visió humana, per tant en el cas dels dataset d'aquesta PdC ens podrem beneficiar de l'estalvi en complexitat.

b) Redimensionament de les imatges

Quan estem entrenant xarxes convolucionals profundes (DCNN), igual com passa amb les xarxes de perceptrons multicapa (MLP), hem de tenir cura de que totes les imatges dels dataset tinguin la mateixa mida, ja que la capa d'entrada (input Layer) de la xarxa neuronal sempre contindrà tants nodes com píxels conté la imatge. En el nostre cas, estem utilitzant imatges amb una resolució relativament alta (512x512), fet pel qual s'haurà de preprocessar cada imatge abans de ser utilitzada per a entrenar el model.

Aquesta tasca es pot efectuar, o bé 1) durant la preparació prèvia de les dades que es realitza sempre abans d'alimentar el model, o bé 2) en el moment de carregar el dataset al model:

1) **Durant la preparació de les dades**:

   podem realitzar la implementació mitjançant el framework *pyTorch* per a Python:

```
transformations = transforms.Compose([
    transforms.ToTensor(),
    # Normalizing the images __________
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
    # We need square images to feed the model (the raw dataset has 640x512 size images)
    transforms.RandomResizedCrop(512),
    # Now we just resize into any of the common input layer sizes (32×32, 64×64, 96×96,
224×224, 227×227, and 229×229)
    transforms.Resize(DATASET_IMG_SIZE)
    ])
```

Snippet 5.x.x.x *– Pre-procés de la mida de les imatges (Implementació AlexNet) - Font: https://learnopencv.com/understanding-alexnet/*

2) **Durant l'entrenament del model**:

   En aquest cas, la estratègia consisteix en afegeix una nova capa anomenada *Lambda layer* per davant de la capa d'entrada o *input Layer*, la qual s'encarregarà de canviar la mida de cada imatge en temps real a en el moment d'entrenar el model (cal notar que aquesta estratègia implica un augment del temps d'entrenament per a cada *epoch*):

```
# Preprocessing: Adding lambda layer that scales up the data to the correct size

model.add(K.layers.Lambda(lambda x:K.backend.resize_images(x,
height_factor=7,width_factor=7,data_format='channels_last')))
```

Snippet 5.x.x.x *– Pre-procés de la mida de les imatges (Implementació DenseNet) - Font: https://bouzouitina-hamdi.medium.com/transfer-learning-with-keras-using-densenet121-fffc6bb0c233*

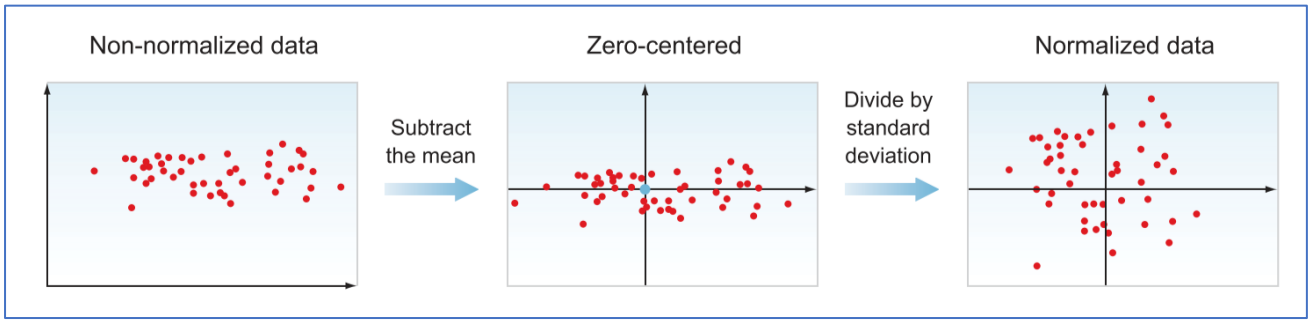
c) Estandardització de les dades

La estandardització o normalització de dades en termes estadístics permet, donat un conjunt de mostres (en el cas que ens ocupa, entenem com a mostres els píxels d'una imatge que podem representar en una variable amb valors $x_1,x_2,..,x_n$), localitzar amb la menor complexitat computacional temporal possible una dada en relació amb la seva mitjana, així com comparar els valors de les observacions de conjunts diferents, o dit d'una altra manera, comparar el valor dels píxels corresponents a imatges diferents[19].

Per a calcular el valor estandarditzat o *Z-Score* d'una observació (píxel) donada la variable $x = x_1,x_2,..,x_n$, amb mitjana $\bar{x}$ i desviació típica s_x, només cal restar la mitjana al valor de la observació i dividir-lo per la desviació típica, tal que:

$$x_i \xrightarrow{\text{Estandarditzar}} z_i = \frac{x_i - \bar{x}}{s_x}$$

D'aquesta manera, si estandarditzem les dades obtindrem com a resultat que el valor de tots els píxels de cada imatge es trobarà en el rang [0,1] de valors, que a efectes pràctics ens permetrà accelerar el procés d'aprenentatge de la xarxa neuronal[20]. La següent figura mostra la distribució uniforme de les dades aconseguida un cop realitzat el procés d'estandardització:



Imatge obtinguda de [Ref. Llibre DL], pàg. 155

---

[19] *Referències bibliogràfiques: Pàgs. 44-46, materials de l'assignatura "Estadística", mòdul "Estadística descriptiva: Introducció a l'anàlisi de dades", Àngel J. Gil Estallo (UOC)*

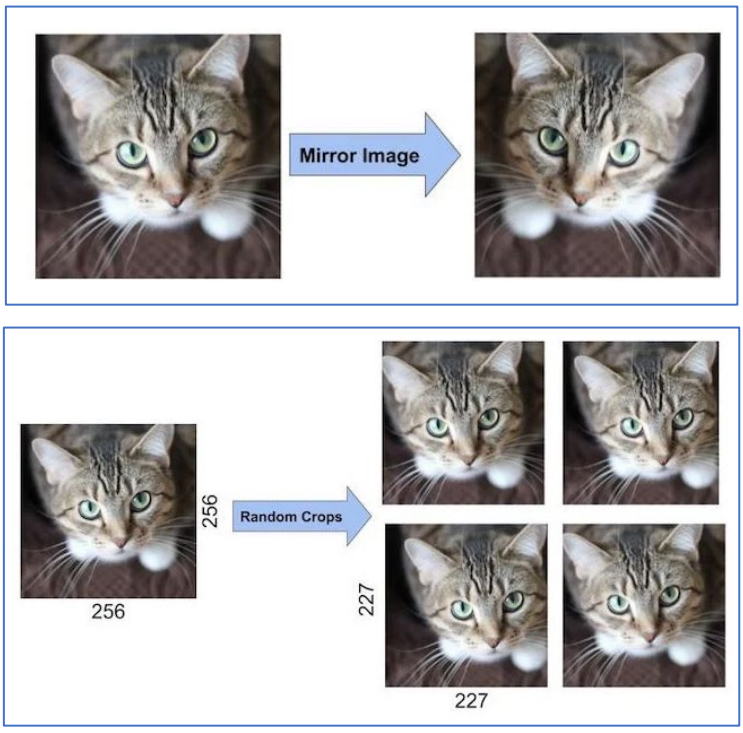
[20] *P. 154 Llibre DL → "Although not required, it is preferred to normalize the pixel values to the range of 0 to 1 to boost learning performance and make the network converge faster."*

Implementació:

Podem implementar la normalització dels dataset de la PdC de manera conjunta per lots mitjançant la llibreria de funcions d'estandarització que incorpora pyTorch (veure snippet 5.x.x.x, línia subratllada en verd).

d) Augmentació de dades (data augmentation)

Es tracta d'un conjunt de tècniques que permet ampliar la quantitat d'imatges dels dataset (i per tant incrementar el rati d'aprenentatge del model) simplement aplicant-hi certes modificacions per a generar diferents variacions. A més, està demostrat que ampliant el catàleg d'imatges amb còpies lleugerament modificades s'augmenta la precisió de les prediccions realitzades pel model i es redueix la probabilitat de que es produeixi *overfitting*[21], és a dir, que s'acabi "sobre-entrenant" el model fent que aquest perdi la seva capacitat de generalització amb imatges que no formin part del dataset d'entrenament. Alguns exemples de tècniques utilitzades per a entrenar alguns models coneguts (p.e. AlexNet) són el "mirroring", o el "random cropping", tal i com es mostra a continuació:



Imatges obtingudes de https://learnopencv.com/understanding-alexnet/

[21] https://www.unite.ai/what-is-overfitting/

Cal fer notar que existeixen altres tècniques semblants, com la rotació d'imatges, l'apropament o "zoom", etc, i que a més és possible utilitzar combinacions de diferents tècniques per a ampliar encara més els dataset d'imatges.

Per a aplicar les tècniques esmentades d'augmentació d'imatges als dataset d'aquesta PdC per exemple podem utilitzar la el mòdul `transforms` que proporciona el framework `pyTorch` , o bé la llibreria `Augmentor`[22] per a `Python`. Escollirem la primera opció degut a que permet realitzar l'augmentació de la col·lecció en el moment de preparar les imatges[23].

Implementació:

```
augmentations = transforms.Compose([
transforms.ToTensor(),
# Normalizing the images ____________
transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
transforms.RandomHorizontalFlip(p=0.5), # Augmentation techique: Horizontal Mirroring
# We need square images to feed the model (the raw dataset has 640x512 size images)
# DEBUG - UNCOMMENT NEXT LINE FOR v4 DATASET
#transforms.RandomResizedCrop(512),
# Now we just resize into any of the common input layer sizes (32×32, 64×64, 96×96, 224×224,
227×227, and 229×229)
transforms.Resize(DATASET_IMG_SIZE)
])

# 1.2 - Create an instance for training.
train_data = datasets.ImageFolder(root=TRAIN_DATA_DIR, transform=transformations) +
             datasets.ImageFolder(root=TRAIN_DATA_DIR, transform=augmentations)

# [...]

# 1.4 - Create an instance for testing.
test_data = datasets.ImageFolder(root=TEST_DATA_DIR, transform=transformations) +
            datasets.ImageFolder(root=TEST_DATA_DIR, transform=augmentations)
```

Snippet 5.x.x.x – Font: https://programming-review.com/pytorch/data-augmentation

A l'snippet 5.x.x.x observem com es pot duplicar la quantitat d'imatges del dataset d'entrenament i de testeig, tot reaprofitant la definició de transformacions establerta més amunt en ambdós casos (mòdul `transforms`).

[22] https://augmentor.readthedocs.io/en/master/

[23] *Pel que fa les les tasques de de pre-processament, normalment és habitual realitzar-les o bé a la entrada del model com en el cas del redimensionament, o bé en etapes intermèdies, com passa amb la estandardització*

5.2.1.2 - Divisió dels dataset en grups

Un cop s'hagin aplicat les diferents tècniques de pre-processament d'imatges als dataset per tal d'adaptar-los als requeriments de l'arquitectura CNN, el que haurem de fer és dividir cada dataset en tres grups o subconjunts diferents;

```
a) Grup d'entrenament (training dataset)      → 75% - 80%
b) Grup de validació (validation dataset)     → 15% - 20%
c) Grup de proves (test dataset)              → 15% - 20%
```

Figura 5.X.X.X - *Divisió dels dataset i la seva proporció – Font: [LLIBRE DL P152]*

De manera que utilitzarem el subconjunt d'entrenament **a)** per a entrenar el model tot ajustant el pes de les arestes que connecten les capes ocultes (*hidden layers*) de la CNN; el subconjunt de validació **b)** per a contrastar el rendiment del model **durant l'entrenament** (de manera periòdica, cada certes *epoch o* iteracions d'entrenament) contra un subconjunt de mostres "desconegut" per al model per a comprovar el nivell de generalització del coneixement adquirit pel model durant la etapa d'entrenament i realitzar els ajustos corresponents als pesos de les arestes per a millorar el seu rendiment; i finalment, el subconjunt de test **c)** ens servirà per a avaluar el rendiment final del model **un cop completat l'entrenament**.

Per tal de preparar les dades de cara a alimentar el model i tenint en compte el dataset utilitzat en aquesta PdC, la implementació que caldrà fer amb pyTorch serà com segueix:

```
# 1.3 - Create a loader for the training set which will read the data within batch size and
put into memory.
train_loader = DataLoader(train_data, batch_size=batch_size, shuffle=True, num_workers=0)
print("The number of images in a training set is: ", len(train_loader)*batch_size)

# 1.5 - Create a loader for the test set which will read the data within batch size and put
into memory. Notice that each shuffle is set to false for the test loader.
test_loader = DataLoader(test_data, batch_size=batch_size, shuffle=False, num_workers=0)
print("The number of images in a test set is: ", len(test_loader)*batch_size)
```

Snippet 5.x.x.x **–**

Pel que fa la proporció de mostres dels dataset que s'assignen a cada subconjunt tindrem en compte que, per norma general, si el dataset està comprés per una quantitat relativament gran de mostres (de l'ordre dels milions) aleshores amb un 1% de les mostres per als subconjunts de validació i d'explotació ja n'hi haurà prou, de manera que reservarem tota la resta per a efectuar l'entrenament i així aconseguir una major precisió de les prediccions del model. En el cas de la PdC

que estem desenvolupant però, els dataset contindran una quantitat d'imatges molt inferior (de l'ordre dels centenars o milers), de manera que els ratis habitualment utilitzats per a l'entrenament de models ML seran més convenients (veure figura 5.x.x.x).

5.2.2 – Definició d'un model base (baseline model)

Un cop hem preparat el dataset que *alimentarà* el model durant el seu entrenament i testeig, podem procedir tant a definir com a implementar la xarxa neuronal convolucional que ens permetrà realitzar la classificació de les imatges. En aquest sentit, s'implementaran diferents versions del model CNN de manera que es podran comparar els resultats obtinguts en termes de precisió en les prediccions, fet que ens permetrà escollir la millor d'elles per a la implementació de la PdC.

De manera general, una xarxa neuronal conté cinc tipus diferents de capes: Capa convolucional o *conv,* capa d'activació, capa d'estandardització o *batchNorm*, capa de pooling o *maxPool* i capa de sortida o *linear.* Tot seguit es donen alguns detalls als respecte de la finalitat de cada capa del model, tot i que de manera superficial:

- **Capa *Conv*** → Es tracta de la capa principal de la xarxa CNN i té la finalitat de detectar característiques concretes dins de cada imatge (p.e. formes). A cada capa convolucional o `conv` se li assigna un nombre de **kernels** que són els que estabiliran la mida de les característiques detectades. En el cas que ens ocupa, establirem un kernel de mida relativament elevada a la capa d'entrada (9x9 o bé 11x11) per tal de detectar les característiques més grans possibles (tot i que així reduïm la quantitat de característiques detectables). A la secció 5.2.3 es donen més detalls sobre la seva funcionalitat.

- **Capa *ReLu*[24]** →La funció d'activació ReLu (*Rectified Linear Function*) s'utilitza de manera generalitzada a les capes ocultes (hidden) de la majoria de models CNN i proporciona l'habilitat de desactivar aquells perceptrons de la xarxa que no assoleixin un valor llindar (zero en el cas de `ReLu`) en el moment de calcular el pes de l'aresta que el connecta amb la altres capes del model (el qual té forma de graf), evitant així la formació de grafs complets entre capes o *Fully Connected Layers*. Dit amb altres paraules, amb l'aplicació de la funció d'activació dotem el model de no linealitat, el que a la pràctica permet implementar la classificació: per exemple, si un perceptró del model està encarregat de detectar una característica concreta d'una imatge de classe 1, però al realitzar els càlculs del pes de l'aresta que connecta el perceptró amb l'anterior s'obté un nombre igual o inferior a zero, aleshores tindrem que la característica detectada no correspondrà a una imatge de classe #1 i en conseqüència la sortida de la funció `ReLu` serà zero i el perceptró restarà desactivat (per això s'anomena funció d'activació). A la secció 5.1.1.1 es donen més detalls sobre les característiques de la funció `ReLu`.

[24] *Font: The Role of Activation Function in CNN [Wang Hao, Lou Yaqin] https://conferences.computer.org/ictapub/pdfs/ITCA2020-6EliKprXTS23UiQ2usLpR0/114100a429/114100a429.pdf*

- **Capa *BatchNorm*** → De la mateixa manera com es realitza la estandardització de les dades abans d'alimentar el model (veure secció 5.2.1.1, apartat c), també cal realitzar-la entre la sortida de cada capa i l'entrada de la següent. Això ho aconseguim situant una capa d'estandardització després de cadascuna de les capes convolucionals `conv`.

- **Capa *MaxPool*** → La inclusió d'una capa `MaxPool` de manera estratègica dividint el model en dos parts, dota l'arquitectura de la capacitat de detectar característiques independentment de la seva posició dins la imatge.

- **Capa *Linear*** → La capa `linear` és la darrera capa del model i té la finalitat de calcular els índexs de probabilitat de predicció (scores) de cadascuna de les classes, de manera que la etiqueta que representi la classe de la qual s'hagin detectat més característiques tindrà un *score* major que la resta d'etiquetes i per tant es correspondrà amb la predicció realitzada pel model. Aquesta capa ha d'estar present sempre com a capa de sortida de l'arquitectura.

La figura 5.x.x.x mostra de manera conceptual com el model és capaç, després d'haver analitzat una quantitat determinada d'imatges, de descartar totes aquelles característiques d'una imatge que no es corresponen amb cap de les classes representades per l'etiquetat de les dades, i a més, de determinar a quina classe correspon el conjunt de característiques detectades (és a dir, d'efectuar una predicció):

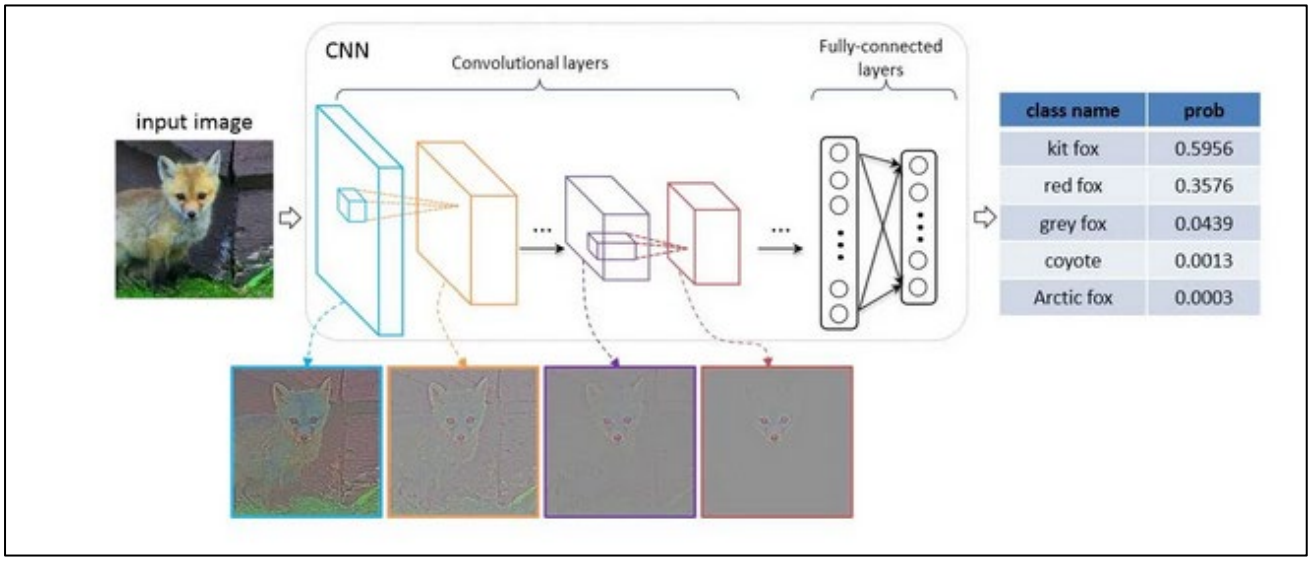


Figura 5.x.x.x *– Disseny conceptual del funcionament d'una xarxa CNN. Font: https://viso.ai/deep-learning/vgg-very-deep-convolutional-networks/*

A continuació es realitzarà la proposta de tres versions del model, cadascuna amb diferents graus de complexitat (en termes de quantitat de paràmetres configurables):

5.2.2.1 – Versió 1.0

En primer lloc, es proposa la següent arquitectura de 3 capes per tal de començar a experimentar amb el disseny de xarxes neuronals, sense aplicar capes d'estandardització ni de pooling. D'aquesta manera serà més senzill realitzar en càlcul de el paràmetre `out-channel` de la capa linear de sortida del model (La implementació corresponent a aquest model es realitza a la secció 5.2.4.1):

Figura 5.x.x.x – Estructura de capes del model CNN – Versió 1

5.2.2.2 – Versió 2.0

S'ha trobat que el següent curs d'introducció al desenvolupament de xarxes neuronals amb `pyTorch` està prou bé documentat i l'algorisme que proposa a part de presentar una complexitat moderada d'implementació, també està orientat a la classificació d'imatges en color, que també ens pot servir per a classificar imatges en B/N:

- https://deeplizard.com/learn/video/IKOHHItzukk

En aquest cas, també farem ús del paquet `torch.nn` de `Python` per a implementar el següent model de xarxa CNN (veure secció 5.2.4.2), aquest cop però, estructurat en 9 capes:

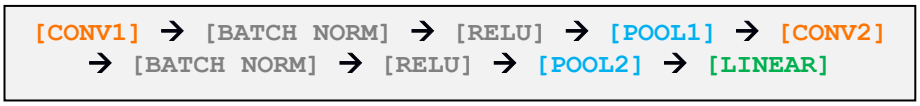


Figura 5.x.x.x – Estructura de capes del model CNN – Versió 2

5.2.2.3 – Versió 3.0

Finalment, tenim que Microsoft també ofereix a la seva extensa plataforma de documentació una proposta interesant (a nivell didàctic / acadèmic) d'arquitectura CNN:

https://docs.microsoft.com/en-us/windows/ai/windows-ml/tutorials/pytorch-train-model

En aquest cas, la complexitat del model és lleugerament més alta comparada amb l'arquitectura proposada a la versió 2.0. Concretament, construirem el següent model de xarxa CNN estructurat en 14 capes (veure secció 5.2.4.3):

```
[CONV1] → [BATCH NORM] → [RELU] → [CONV2] →
[BATCH NORM] → [RELU] → [POOL] → [CONV3] → [BATCH NORM] →
[RELU] → [CONV3] → [BATCH NORM] → [RELU] → [LINEAR]
```

Figura 5.x.x.x – Estructura de capes del model CNN – Versió 3

5.2.3 – Càlcul d'hyperparàmetres[25]

La capa convolucional és la encarregada de dur a terme la major càrrega computacional del model. Concretament, la seva tasca és la de realitzar el producte entre dues matrius, definides d'una banda pel conjunt de paràmetres configurables (mida del kernel), i de l'altra, la porció o *subset* de la imatge restringida per la mida camp receptiu (que depèn de la mida del kernel), com il·lustra la figura 5.x.x.x:

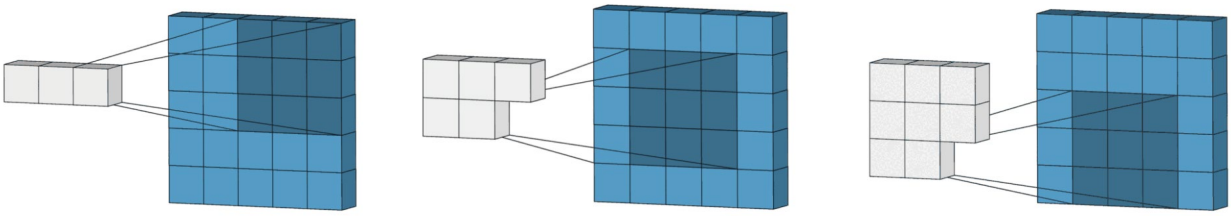
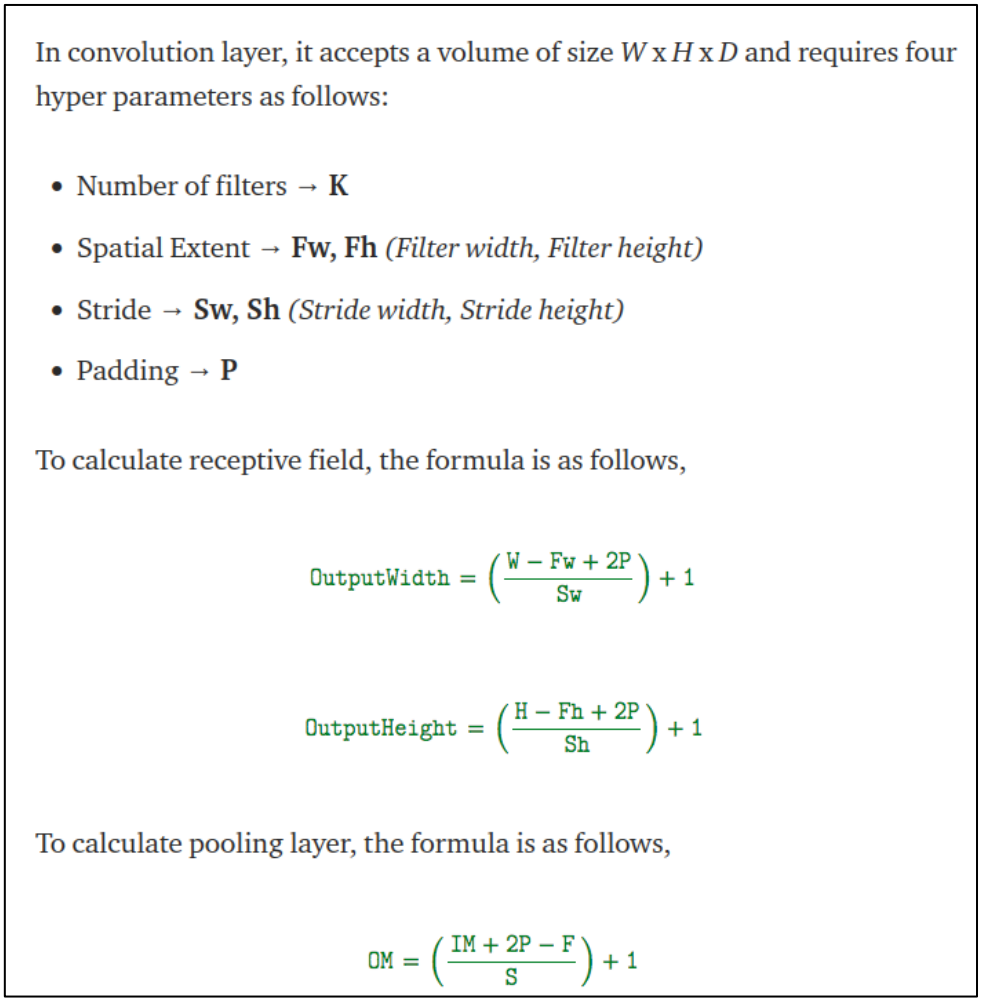


Figura 5.x.x.x – Ilustració d'alguns passos en una operació de convolució.
*Font: https://miro.medium.com/max/2340/1*Fw-ehcNBR9byHtho-Rxbtw.gif*

Dels càlculs realitzats anteriorment s'obté una imatge que representa aquest camp receptiu, de la qual s'han de conèixer de partida les mides, ja que aquestes determinaran la mida de la entrada o in-channel de la capa següent de l'arquitectura durant el procés de *forward-pass*. Per a calcular la mida de sortida de cada capa, primer caldrà establir una sèrie de paràmetres coneguda com *híper-paràmetres* de manera totalment aleatòria, tot fent servir una estratègia d'assaig-error per tal de trobar aquella combinació que el doni millor rendiment (accuracy) donades les característiques de les imatges del dataset construït.

[25] *La informació per a desenvolupar els continguts s'ha aconseguit del següent enllaç web: https://towardsdatascience.com/convolutional-neural-networks-explained-9cc5188c4939*

La imatge següent resumeix d'una banda els *híper-paràmetres* que cal establir prèviament, i de l'altra, els càlculs que s'ha realitzar per a calcular la mida de la sortida de les capes convolucionals i de pooling, respectivament:



In convolution layer, it accepts a volume of size W x H x D and requires four hyper parameters as follows:

- Number of filters → **K**
- Spatial Extent → **Fw, Fh** *(Filter width, Filter height)*
- Stride → **Sw, Sh** *(Stride width, Stride height)*
- Padding → **P**

To calculate receptive field, the formula is as follows,

$$\text{OutputWidth} = \left(\frac{\text{W} - \text{Fw} + 2\text{P}}{\text{Sw}}\right) + 1$$

$$\text{OutputHeight} = \left(\frac{\text{H} - \text{Fh} + 2\text{P}}{\text{Sh}}\right) + 1$$

To calculate pooling layer, the formula is as follows,

$$\text{OM} = \left(\frac{\text{IM} + 2\text{P} - \text{F}}{\text{S}}\right) + 1$$

Figura 5.x.x.x – Hiper-paràmetres i càlcul dels volums de sortida de les capes convolucionals i de pooling. Font: https://towardsdatascience.com/convolutional-neural-networks-explained-9cc5188c4939

Un altre aspecte que caldrà tenir en compte és que haurem d'adaptar els arguments d'alguns dels paràmetres de la primera capa de l'arquitectura per tal que la capa d'entrada processi imatges d'un sol canal (whitescale) en comptes de 3 canals (RGB), tal i com s'ha definit al generar el dataset d'imatges a la secció 4 d'aquest document.

Seguidament es realitzen els càlculs necessaris per tal d'implementar els models de xarxes neuronals definits a la secció 5.2.2[26].

[26] *Per a facilitar els càlculs, s'ha dissenyat una plantilla amb la calculadora online http://calcme.com i que es pot trobar al repositori del projecte* `uoc.tfg.jbericat/src/CNN/hyper-parameter-calculator.wiris`

5.2.3.1 – Versió 1.0

CONV 1
Mida de la entrada (W_1 x H_1 x D_1) → 229 x 229 x 1
• Requereix quatre hyper-paràmetres: ○ Nombre de kernels: k = 16 ○ Mida / extensió de cada kernel: F = 5 ○ Mida del pas (stride): S = 1 ○ Farciment de zeros (padding): P = 2
• Volum de sortida → W_2 x H_2 x D_2 ○ $W_2 = \frac{w_1 - F + 2P}{S} + 1 = \frac{229-5+2*2}{1} + 1 = 229$ ○ $H_2 = \frac{H_1 - F + 2P}{S} + 1 = \frac{229-5+2*2}{1} + 1 = 229$ ○ $D_2 = k = 16$
Sortida de CONV 1 (W_2 x H_2 x D_2) → 229x229x16

CONV 2
Mida de la entrada (W_1 x H_1 x D_1) → 229 x 229 x 16
• Requereix quatre hyper-paràmetres: ○ Nombre de kernels: k = 16 ○ Mida / extensió de cada kernel: F = 5 ○ Mida del pas (stride): S = 1 ○ Farciment de zeros (padding): P = 2
• Volum de sortida → W_2 x H_2 x D_2 ○ $W_2 = \frac{w_1 - F + 2P}{S} + 1 = \frac{229-5+2*2}{1} + 1 = 229$ ○ $H_2 = \frac{H_1 - F + 2P}{S} + 1 = \frac{229-5+2*2}{1} + 1 = 229$ ○ $D_2 = k = 16$
Sortida de CONV 1 (W_2 x H_2 x D_2) → 229x229x16

Capa completament enllaçada (FC Layer o *linear*)
Mida de la entrada (W_2 x H_2 x D_2) → 229x229x16
Mida de la sortida (nombre de classes) → 3

5.2.3.2 – Versió 2.0

CONV 1
Mida de la entrada (W_1 x H_1 x D_1) → 229 x 229 x 1
• Requereix quatre hyper-paràmetres: ○ Nombre de kernels: k = 16 ○ Mida / extensió de cada kernel: F = 5 ○ Mida del pas (stride): S = 1 ○ Farciment de zeros (padding): P = 2
• Volum de sortida → W_2 x H_2 x D_2 ○ $W_2 = \frac{w_1 - F + 2P}{S} + 1 = \frac{229-5+2*2}{1} + 1 = 229$ ○ $H_2 = \frac{H_1 - F + 2P}{S} + 1 = \frac{229-5+2*2}{1} + 1 = 229$ ○ $D_2 = k = 16$
Sortida de CONV 1 (W_2 x H_2 x D_2) → 229x229x16

POOL 1
Mida de la entrada (W_2 x H_2 x D_2) → 229 x 229 x 16
• Requereix dos hyper-paràmetres: ○ Mida / extensió de cada kernel: F = 2 ○ Mida del pas (stride): S = 1
• Volum de sortida → W_3 x H_3 x D_2 ○ $W_3 = \frac{w_3 - F}{S} + 1 = \frac{229-2}{1} + 1 = 228$ ○ $H_3 = \frac{H_3 - F}{S} + 1 = \frac{229-2}{1} + 1 = 228$
Sortida de POOL 1 (W_3 x H_3 x D_2) → 228 x 228 x 16

CONV 2

Mida de la entrada (W_3 x H_3 x D_2) → 228x228x16

- Requereix quatre hyper-paràmetres:
  - Nombre de kernels: k = 32
  - Mida / extensió de cada kernel: F = 5
  - Mida del pas (stride): S = 1
  - Farciment de zeros (padding): P = 2

- Volum de sortida → W_4 x H_4 x D_3
  - $W_4 = \frac{W_3 - F + 2P}{S} + 1 = \frac{228 - 5 + 2*2}{1} + 1 = 228$
  - $H_4 = \frac{H_3 - F + 2P}{S} + 1 = \frac{229 - 5 + 2*2}{1} + 1 = 228$
  - $D_3 = k = 32$

Sortida de CONV 2 (W_4 x H_4 x D_3) → 228x228x32

POOL 2

Mida de la entrada (W_4 x H_4 x D_3) → 228 x 228 x 32

- Requereix dos hyper-paràmetres:
  - Mida / extensió de cada kernel: F = 2
  - Mida del pas (stride): S = 2

- Volum de sortida → W_5 x H_5 x D_3
  - $W_5 = \frac{W_3 - F}{S} + 1 = \frac{228 - 2}{2} + 1 = 114$
  - $H_5 = \frac{H_3 - F}{S} + 1 = \frac{228 - 2}{2} + 1 = 114$

Sortida de POOL 1 (W_3 x H_3 x D_2) → 114x114x32

Capa completament enllaçada (FC Layer o *linear*)

Mida de la entrada (W_4 x H_4 x D_3) → 114 x 114 x 32

Mida de la sortida (nombre de classes) → 3

5.2.3.3 – Versió 3.0

CONV 1
Mida de la entrada (W_1 x H_1 x D_1) → 229 x 229 x 1
• Requereix quatre hyper-paràmetres: ○ Nombre de kernels: k = 16 ○ Mida / extensió de cada kernel: F = 5 ○ Mida del pas (stride): S = 1 ○ Farciment de zeros (padding): P = 2
• Volum de sortida → W_2 x H_2 x D_2 ○ $W_2 = \frac{w_1 - F + 2P}{S} + 1 = \frac{229-5+2*2}{1} + 1 = 229$ ○ $H_2 = \frac{H_1 - F + 2P}{S} + 1 = \frac{229-5+2*2}{1} + 1 = 229$ ○ $D_2 = k = 16$
Sortida de CONV 1 (W_2 x H_2 x D_2) → 229 x 229 x 16

CONV 2
Mida de la entrada (W_2 x H_2 x D_2) → 229 x 229 x 16
• Requereix quatre hyper-paràmetres: ○ Nombre de kernels: k = 16 ○ Mida / extensió de cada kernel: F = 5 ○ Mida del pas (stride): S = 1 ○ Farciment de zeros (padding): P = 2
• Volum de sortida → W_2 x H_2 x D_2 ○ $W_2 = \frac{w_1 - F + 2P}{S} + 1 = \frac{229-5+2*2}{1} + 1 = 229$ ○ $H_2 = \frac{H_1 - F + 2P}{S} + 1 = \frac{229-5+2*2}{1} + 1 = 229$ ○ $D_2 = k = 16$
Sortida de CONV 1 (W_3 x H_3 x D_3) → 229 x 229 x 16

POOL 1

Mida de la entrada (W_3 x H_3 x D_3) → 229 x 229 x 16

- Requereix dos hyper-paràmetres:
  - Mida / extensió de cada kernel: F = 2
  - Mida del pas (stride): S = 1

- Volum de sortida → W_5 x H_5 x D_3
  - $W_5 = \frac{w_3 - F}{S} + 1 = \frac{229-2}{1} + 1 = 228$
  - $H_5 = \frac{H_3 - F}{S} + 1 = \frac{229-2}{1} + 1 = 228$

Sortida de POOL 1 (W_4 x H_4 x D_3) → 228 x 228 x 16

CONV 3

Mida de la entrada (W_4 x H_4 x D_3) → 228 x 228 x 16

- Requereix quatre hyper-paràmetres:
  - Nombre de kernels: k = 32
  - Mida / extensió de cada kernel: F = 5
  - Mida del pas (stride): S = 1
  - Farciment de zeros (padding): P = 2

- Volum de sortida → W_2 x H_2 x D_2
  - $W_2 = \frac{w_1 - F + 2P}{S} + 1 = \frac{228-5+2*2}{1} + 1 = 228$
  - $H_2 = \frac{H_1 - F + 2P}{S} + 1 = \frac{228-5+2*2}{1} + 1 = 228$
  - $D_2 = k = 32$

Sortida de CONV 1 (W_5 x H_5 x D_4) → 228 x 228 x 32

CONV 4
Mida de la entrada 1 (W_5 x H_5 x D_4) → 228 x 228 x 32
• Requereix quatre hyper-paràmetres: ○ Nombre de kernels: k = 32 ○ Mida / extensió de cada kernel: F = 5 ○ Mida del pas (stride): S = 1 ○ Farciment de zeros (padding): P = 2
• Volum de sortida → W_2 x H_2 x D_2 ○ $W_2 = \frac{w_1 - F + 2P}{S} + 1 = \frac{228-5+2*2}{1} + 1 = 228$ ○ $H_2 = \frac{H_1 - F + 2P}{S} + 1 = \frac{228-5+2*2}{1} + 1 = 228$ ○ $D_2 = k = 32$
Sortida de CONV 1 (W_6 x H_6 x D_5) → 228 x 228 x 32

Capa completament enllaçada (FC Layer o *linear*)
Mida de la entrada (W_6 x H_6 x D_5) → 228 x 228 x 32
Mida de la sortida (nombre de classes) → 3

5.2.4 – Implementació de l'estructura del model CNN amb el framework pyTorch

Ara que hem definit l'estructura de la xarxa convolucional, així com establert uns híper-paràmetres de referència i realitzat els càlculs apropiats pel que respecta la mida dels volums d'entrada i sortida de cadascuna de les capes, ja podem procedir a implementar tant la **topologia del model** com la funció **d'avançament o *forward function***, que a part de permetre definir la manera com es realitzarà la operació de *forward-pass* entre capes (que resultarà en les prediccions finals), també implementa la propagació de resultats "cap endarrera" per tal d'actualitzar els pesos o *weights* del model (valors del paràmetres de la NN). En aquest sentit, pel que respecta la **funció de propagació cap al darrera o *gradient back-propagation function,*** tenim que el mateix framework `pytorch` per a `python` la defineix de manera automàtica mitjançant el mòdul `torch.autograd` i per tant no ens caldrà implementar-la. De manera més precisa, podem dir que la funció de propagació cap al darrera té com a finalitat re-configurar els paràmetres de cada capa del model (els pesos de les arestes) al fer el càlculs de probabilitat cada cop que les característiques d'una imatge són analitzades, acció que es realitza de manera recurrent per a cadascuna de les imatges processades.

5.2.4.1 – Versió 1.0

```
##########################################################################################################
##########################################################################################################
####                                                                                                  ####
####                          1. CNN MODEL VERSION #1 (3 layers architecture)                         ####
####                                                                                                  ####
##########################################################################################################
##########################################################################################################

# 2.1 - First we define a pretty simple, 2-layer convolution neural network in order to guess the
# best hyper-parameter configuration possible - That is, the one that works with the given data
# (229 x 229 x 1 images) as well as provides with best accuracy results:

class Network_v1(nn.Module):

    def __init__(self):
        # By inheriting from thorch.nn's Network() class, we are able to use it's extended features
in our
        # own class definition (such as printing the model architecure) wehen invoking the class
constructor
        super(Network_v1, self).__init__()

        # Next we set the convolutional, pass-through (batch norm, pooling) and output layers
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=16, kernel_size=5, stride=1, padding=2)
        self.conv2 = nn.Conv2d(in_channels=16, out_channels=16, kernel_size=5, stride=1, padding=2)
        self.fc1 = nn.Linear(229*229*16, 3)

# 2.2 - Implement the forward function:
    def forward(self, input):
        output = F.relu(self.conv1(input))
        output = F.relu(self.conv2(output))
        output = output.view(-1, 229*229*16 )
        output = self.fc1(output)

        return output
```

Snippet 5.x.x.x *– Implementació del model CNN v1.0 amb el framework pyTorch per a Python – CNN_models.py*

5.2.4.2 – Versió 2.0

```
#################################################################################################
#################################################################################################
####                                                                                         ####
####                          2. CNN MODEL VERSION #2 (9 Layers architecture)                ####
####                                                                                         ####
#################################################################################################
#################################################################################################

class Network_v2(nn.Module):

    def __init__(self):

        super(Network_v2, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=16, kernel_size=11, stride=2, padding=2)
        self.bn1 = nn.BatchNorm2d(16)
        self.pool1 = nn.MaxPool2d(2,1)
        self.conv2 = nn.Conv2d(in_channels=16, out_channels=32, kernel_size=5, stride=3, padding=2)
        self.bn2 = nn.BatchNorm2d(32)
        self.pool2 = nn.MaxPool2d(2,2)
        self.fc1 = nn.Linear(114*114*32 , 3)

# 2.2 - Implement the forward function:
    def forward(self, input):

        output = F.relu(self.bn1(self.conv1(input)))
        output = self.pool1(output)
        output = F.relu(self.bn2(self.conv2(output)))
        output = self.pool2(output)
        output = output.view( -1, 114*114*32)
        output = self.fc1(output)

        return output
```

Snippet 5.x.x.x *– Implementació del model CNN v2.0 amb el framework pyTorch per a Python -* `CNN_models.py`

5.2.4.3 – Versió 3.0

```
#################################################################################################
#################################################################################################
####                                                                                         ####
####                          3. CNN MODEL VERSION #3 (14 layers architecture)               ####
####                                                                                         ####
#################################################################################################
#################################################################################################

# See this project report's (section 5.2.3.3) for further information in regards to the calculations
# made to set the output-channel size (linear layer)

# Define a convolution neural network
class Network_v3(nn.Module):
    def __init__(self):
        super(Network_v3, self).__init__()

        self.conv1 = nn.Conv2d(in_channels=1, out_channels=16, kernel_size=9, stride=2, padding=2)
        self.bn1 = nn.BatchNorm2d(16)
        self.conv2 = nn.Conv2d(in_channels=16, out_channels=16, kernel_size=5, stride=2, padding=2)
        self.bn2 = nn.BatchNorm2d(16)
        self.pool = nn.MaxPool2d(2,1)
        self.conv3 = nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3, stride=1, padding=2)
        self.bn3 = nn.BatchNorm2d(32)
        self.conv4 = nn.Conv2d(in_channels=32, out_channels=32, kernel_size=5, stride=1, padding=2)
        self.bn4 = nn.BatchNorm2d(32)
        self.fc1 = nn.Linear(32*58*58, 3)

    def forward(self, input):
        output = F.relu(self.bn1(self.conv1(input)))
        output = F.relu(self.bn2(self.conv2(output)))
        output = self.pool(output)
        output = F.relu(self.bn3(self.conv3(output)))
        output = F.relu(self.bn4(self.conv4(output)))
        output = output.view(-1, 32*58*58)
        output = self.fc1(output)

        return output
```

Snippet 5.x.x.x *– Implementació del model CNN amb el framework pyTorch per a Python - CNN_models.py*

5.2.5 – Definició de la funció de pèrdua (loss function)[27]

La definició de la funció de pèrdua o *loss function* és un mecanisme indispensable la implementació del qual ens servirà per a estimar un coeficient que ens permetrà dur un control del grau d'optimització del model rere cada iteració o *epoch* d'entrenament. Cal observar que no s'ha de confondre el valor de pèrdua o *loss* amb la precisió del model, la qual es calcula utilitzant el dataset de testeig i ens proporciona el percentatge d'encert en les prediccions. Es donaran més detalls al respecte de la mesura i interpretació de la pèrdua i el rendiment durant el desenvolupament de la secció 6 (entrenament del model).

En el cas que ens ocupa; per al desenvolupament de l'algorisme d'entrenament que es presentarà a la secció següent s'ha utilitzat una de les diferents funcions de pèrdua que proporciona el framework `pytorch` específic per a ML i DL, i que ja hem utilitzat per a implementar l'arquitectura de la CNN. Concretament, es farà ús de la funció de pèrdua específica per a problemes de classificació anomenada `Classification Cross-Entropy`, així com de la funció d'optimització `Adam Optimizer`:

```
################################################################################################
################################################################################################
####                                                                                        ####
####                                4. DEFINE THE LOSS FUNCTION                             ####
####                                                                                        ####
################################################################################################
################################################################################################

# Define a loss function
from torch.optim import Adam

# Define the loss function with Classification Cross-Entropy loss and an optimizer with Adam
optimizer
loss_fn = nn.CrossEntropyLoss()
optimizer = Adam(model.parameters(), lr=LEARNING_RATE, weight_decay=0.0001)
```

***Snippet 5.x.x.x** – Implementació de la funció de pèrdua o loss-function mitjançant el framework pyTorch per a Python – Fitxer: /src/CNN/pytorch-training.py*

Observem que caldrà especificar el valor del paràmetre lr (rati d'aprenentatge o learning rate) durant la implementació de l'algorisme d'entrenament (habitualment el seu valor és de 10^n amb n = { -1, -2, -3, ... , -n }.

[27] *Font: https://docs.microsoft.com/en-us/windows/ai/windows-ml/tutorials/pytorch-train-model#define-a-loss-function*

5.3 – *Wrapping-Up*: Implementació d'un algorisme d'entrenament i generació de gràfiques i estadístiques

Un cop establerts els paràmetres inicials amb els quals crearem l'arquitectura de xarxes neuronals que utilitzarem en aquesta PdC així com establerta la seva definició i implementació, podem procedir a implementar l'algorisme principal que utilitzarem per a entrenar el model construït. Per a tal efecte, de partida utilitzarem una algorisme ja dissenyat, la implementació de la qual es pot trobar al següent enllaç:

https://docs.microsoft.com/en-us/windows/ai/windows-ml/tutorials/pytorch-train-model

Tanmateix, un cop començada la implementació es troba que la proposta d'algorisme anterior té algunes funcionalitats limitades o bé defectuoses, tal que:

- No implementa la generació de les gràfiques de pèrdua i rendiment mitjançant llibreries com per exemple `matplotlib`.
- No realitza la divisió "en calent" del subset d'entrenament i de validació, així com tampoc implementa la validació del model pròpiament dita durant l'entrenament.
- No estableix una parametrització d'alguns valors de configuració d'entrenament (p.e. nombre de canals d'entrada, nombre de classes o *output-features,* el *learning-rate*, entre d'altres).
- El resum de sortida proporciona dades en un format difícil d'interpretar i que dona poques dades.
- No permet escollir entre diferents arquitectures de CNN.
- Presenta "bugs" que no permeten la utilització de tècniques computació d'alt rendiment (GPU) per a l'entrenament del model.

Per tant, es decideix fer una revisió modificada de l'algorisme tot incorporant codi i funcionalitats d'altres implementacions, especialment de la següent:

https://www.pyimagesearch.com/2021/07/19/pytorch-training-your-first-convolutional-neural-network-cnn/

A grans trets, la estructura de l'algorisme resultant es correspon amb la següent, un cop realitzades les modificacions per a pal·liar les carències esmentades:

1. Importar les dependències
2. Preparar les dades per a l'entrenament
    i. Dividir el dataset d'entrenament i validació en dos subsets mitjançant la funció `torch.utils.data.random_split` que proporciona el framework `pytorch`
    ii. Importar els subsets d'entrenament i validació del dispositiu d'emmagatzematge (arxiu) mitjançant la classe `torchvision.datasets.ImageFolder` de `pytorch`
    iii. Importar el subset de test del dispositiu d'emmagatzematge (arxiu)
    iv. Aplicar transformacions i tècniques d'ampliació als subsets d'entrenament, validació i testeig mitjançant el mòdul `torchvision.transforms` de `pytorch`
    v. Aplicar estandardització a les imatges (*Normalization*) mitjançant el mòdul `torchvision.transforms` de `pytorch`
    vi. Carregar els subsets d'entrenament, validació i testeig a la memòria principal de la CPU mitjançant la classe `torch.utils.Dataloader` de `pytorch`
    vii. Codificar les etiquetes (*labels*) representades pel dataset mitjançant una estructura de dades (llista)
3. Importació de l'arquitectura CNN escollida des de la classe `Network()` definida al fitxer extern `CNN_models.py`
4. Entrenament del model mitjançant la funció `train()`
    a. Entrenament amb *gradient-back-propagation* activat mitjançant la classe `torch.no_grad()` de `pytorch`
    b. testeig del rendiment al final de cada epoch per tal de poder generar la gràfica amb la corba d'aprenentatge (*accuracy progression*)
    c. generació de les corbes de pèrdua (*loss-rate*) i d'aprenentatge (*accuracy progression*) mitjançant el framework `matpyplot` de `python`
5. Creació de la gràfica d'aprenenatge contra el dataset de test
6. Testeig final contra una mostra petita del dataset mitjançant la funció `test_batch()` per tal d'efectuar una prova de predicció.
7. Presentació de resultats i empaquetament dels arxius resultants al paquet "tar" corresponent → `/bin/CNN/cnn-training-%TIMESTAMP%.tar.gz`
    a. arxiu `trained-model.pth` amb el model resultant rere l'entrenament
    b. Arxiu `trained-model.info` amb les estadístiques d'entrenament del model
    c. arxiu `model-loss-curves.png` amb gràfiques de les funcions de pèrdua d'entrenament i validació
    d. Arxiu `model-accuracy.png` amb la gràfica de la corba d'aprenentatge

La implementació final de l'algorisme d'entrenament es pot trobar al següent arxiu del repositori d'aquest projecte:

https://github.com/UOC-Assignments/uoc.tfg.jbericat/blob/master/src/CNN/pytorch_training.py

Cal observar que els comentaris originals del codi s'han adaptat i ampliat en gran mesura per tal de documentar tot el procés d'implementació de la millor manera possible.

5.4 – ANNEX OPCIONAL: Possibles millores i optimitzacions del model

5.4.1 - Dropout regularization

https://learnopencv.com/understanding-alexnet/