

Xarxes neuronals convolucionals amb Keras

Iloc: [Institut d'Ensenyaments a Distància de les Illes Balears](#)
Curs: Programació d'intel·ligència artificial
Llibre: Xarxes neuronals convolucionals amb Keras

Imprès per: Carlos Sanchez Recio
Data: dilluns, 24 de febrer 2025, 20:22

Taula de continguts

1. Introducció a les xarxes neuronals convolucionals

2. Components de les CNN

2.1. Convolució

2.2. Pooling

3. Implementació d'una CNN bàsica en Keras

3.1. Arquitectura d'una CNN

3.2. Definició del model

3.3. Configuració, entrenament i evaluació

4. Hiperparàmetres

4.1. Mida i nombre de filtres

4.2. Padding

4.3. Stride

5. Aplicació al conjunt de dades Fashion-MNIST

5.1. Model bàsic

5.2. Capes i optimitzadors

5.3. Capes de Dropout i BatchNormalization

5.4. Disminució de la taxa d'aprenentatge

6. Xarxes neuronals preentrenades

6.1. API funcional de keras

6.2. Conjunt de dades CIFAR-10

6.3. ResNet50

6.4. VGG19

7. Xarxes neuronals convolucionals amb keras

7.1. Classificació d'imatges

7.2. Segmentació d'imatges

7.3. Detecció d'objectes

1. Introducció a les xarxes neuronals convolucionals

Després d'haver introduït les xarxes neuronals al lliurament quart de Sistemes d'Aprenentatge Automàtic ja estam en condicions d'aprofundir en arquitectures més avançades. En aquest capítol treballarem les xarxes neuronals convolucionals, que s'apliquen a la visió per computador.

Presentarem un exemple que seguirem passa a pas per entendre els conceptes implicats en aquest tipus de xarxes. Programarem una xarxa neuronal convolucional per resoldre el mateix problema del reconeixement de díigits MNIST, però ara amb una arquitectura específica per al reconeixement d'imatges. Al lliurament anterior de Sistemes d'Aprenentatge Automàtic varem usar una arquitectura general de xarxes neuronals que no aprofitava l'estructura bidimensional de les dades d'imatge. Per això els resultats que obteníem no eren tan bons com els que obtindrem usant xarxes convolucionals.

Després de l'exemple de reconeixement de díigits MNIST, reprendrem també l'exemple de Fashion-MNIST, amb imatges de productes de roba.

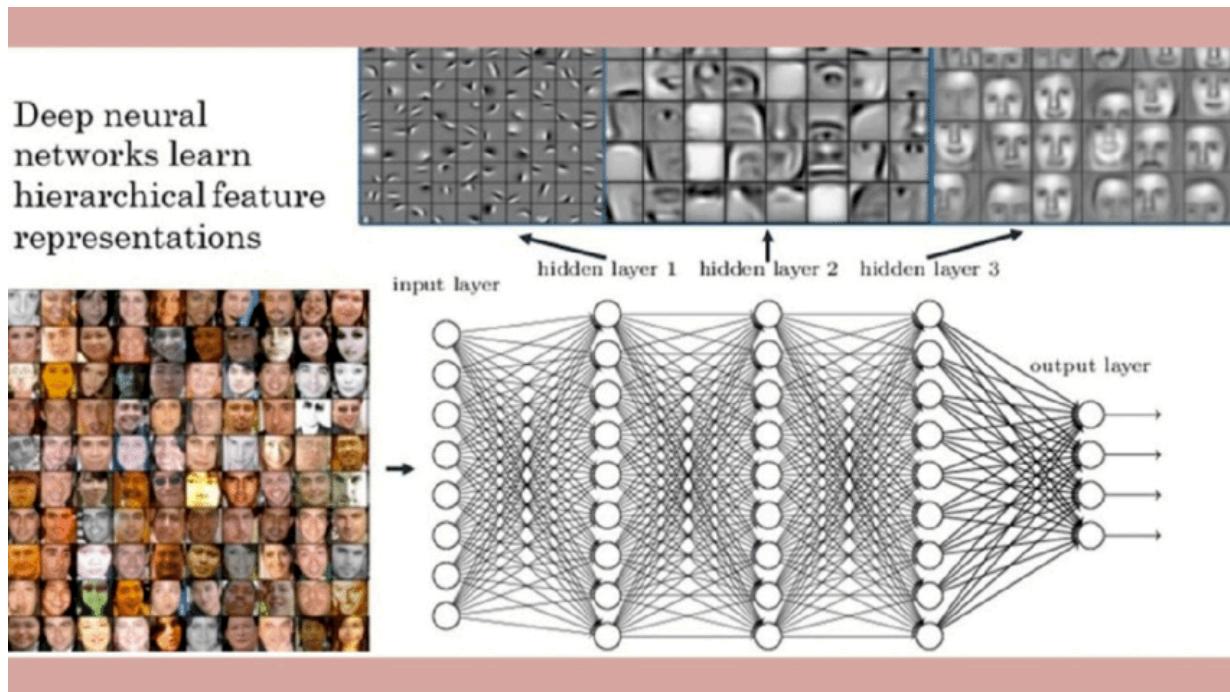
Les **xarxes neuronals convolucionals** (*Convolutional Neural Networks* en anglès, amb les sigles **CNN** o **ConvNets**) són un cas concret de xarxes neuronals que ja es varen introduir a finals de la dècada de 1990, però que en els darrers anys s'han popularitzat enormement, ja que aconsegueixen resultats imporessionants en el reconeixement d'imatges. Això produeix un gran impacte en l'àrea de visió per computadora.

Com les xarxes neuronals en general, les xarxes neuronals convolucionals també tenen capes de neurones amb paràmetres en forma de pesos i biaixos que es poden aprendre mitjançant els algorismes de descens de gradient i retropropagació. Ara bé, una característica diferencial de les xarxes convolucionals és que aprofiten la informació que les dades d'entrada són imatges. Això ens permetrà codificar unes determinades propietats en l'arquitectura per reconèixer elements particulars a les imatges.

Per fer-nos una idea que com funcionen les CNN, pensem com nosaltres reconeixem visualment els objectes. Per exemple, una cara la reconeixem perquè té orelles, ulls, un nas, cabells, etc. Per decidir que una imatge és una cara, de qualsevol forma comprovam que té una llista de característiques que anam marcant. De vegades, una cara no tendrà una orela perquè està tapada pels cabells, els ulls poden estar tapats per ulleres, la boca pot mostrar les dents o no, però en qualsevol cas es compleixen els requeriments de l'objecte *cara*. En aquest cas, podem veure una xarxa neuronal convolucional com un classificador que prediu la probabilitat que una imatge concreta sigui una cara o no.

Ara bé, per poder identificar la cara, hem de saber quina forma tenen les parts. Hem de poder identificar línies, vores, textures, formes semblants a les de les parts de la cara. Aquesta és la funció de les capes internes de la xarxa convolucional.

A més, s'ha de tenir en compte la posició relativa i la mida dels elements de la cara. La cara ha de tenir els seus elements i, a més, amb una disposició relativa concreta. Podem veure la relació entre les diferents capes de la xarxa convolucional profunda i les característiques que s'aprenen en cada nivell a la imatge següent.



Imatge: Yann LeCun et al. (2015)

2. Components de les CNN

Ara que tenim una primera visió intuïtiva de com funcionen les xarxes neuronals convolucionals, entrarem a distingir els dues operacions principals que realitzen: **convolució i pooling**.

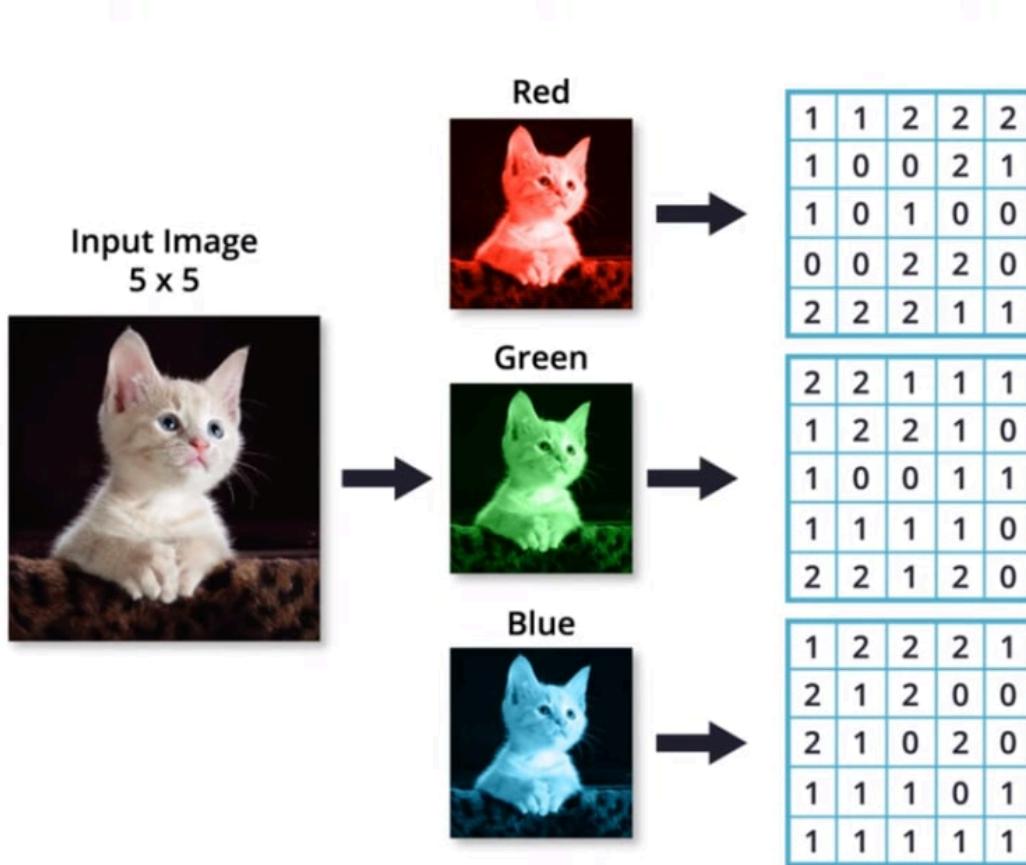
2.1. Convolució

Al llurament quart de sistemes d'aprenentatge vàrem introduir les xarxes densament connectades. Allà s'aprenien patrons globals en l'espai d'entrada. En canvi, a les xarxes convolucionals s'aprenen patrons locals dins la imatge, en petites finestres de dues dimensions.

L'objectiu principal d'una capa convolucional és detectar característiques o trets visuals de les imatges, com ara arestes, línies o gotes de color. Això és molt interessant perquè una vegada s'ha après una característica en un punt concret de la imatge, la pot reconèixer després en qualsevol altre punt de la imatge. En canvi, una xarxa neuronal densament connectada ha d'aprendre el patró novament en cada punt de la imatge.

Una altra característica important és que les capes convolucionals poden aprendre jerarquies espacials de patrons, preservant relacions espacials. Per exemple, una primera capa convolucional pot aprendre elements bàsics com arestes, i una segona capa pot aprendre patrons composts formats per elements bàsics que venen de la capa anterior. D'aquesta forma, les xarxes neuronals convolucionals poden aprendre d'una forma eficient conceptes visuals cada vegada més complexos i abstractes.

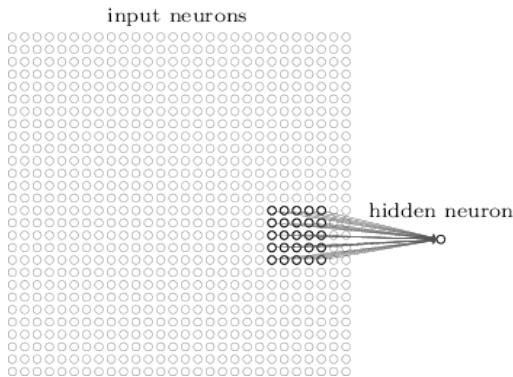
En general, les capes convolucionals operen sobre tensors 3D, anomenats mapes de característiques (*feature maps* en anglès), amb dos eixos espacials d'alçada i amplada (*height* i *width*), a més d'un eix de canal (*channel*), també anomenat profunditat (*depth*). En una imatge de color RGB, la dimensió de l'eix *depth* és 3, ja que la imatge té tres canals: vermell, verd i blau (*red*, *green* i *blue*). En canvi, per a una imatge en blanc i negre, com és el cas dels dígits MNIST, la dimensió de l'eix *depth* és 1 (nivell de gris).



Una imatge en color es descompon en tres canals RGB. Imatge: <https://dev.to/sandeepbalachandran/machine-learning-convolution-with-color-images-2p41>

En el cas de MNIST, com a entrada de la nostra xarxa neuronal podem pensar en un espai de neurones de dues dimensions 28x28, que transformarem en un tensor 3D (*height*=28, *width*=28, *depth*=1), encara que la tercera dimensió sigui simplement de mida 1. Una primera capa de neurones ocultes connectades a les neurones de la capa d'entrada realitzaran les operacions convolucionals. Però no es connecten totes les neurones d'entrada amb

totes les neurones del primer nivell de neurones oclutes, sinó que aquesta connexió es realitza per petites zones localitzades de l'espai. A la imatge següent veim que es connecten 5x5 neurones de la capa d'entrada a una sola neurona de la següent capa.



Aquesta finestra de 5x5 va recorrent totes les posicions de la imatge 28x28. Si analitzam amb detall aquest exemple, veurem que una entrada de 28x28 i una finestra de 5x5 ens defineixen un espai a la primera capa oculta de 24x24, ja que la finestra només es pot desplaçar 23 posicions cap a la dreta i cap avall abans de xocar amb l'extrem inferior dret de la imatge. Aquest 24 s'obté com a $28+5-1 = 28-5+1$.

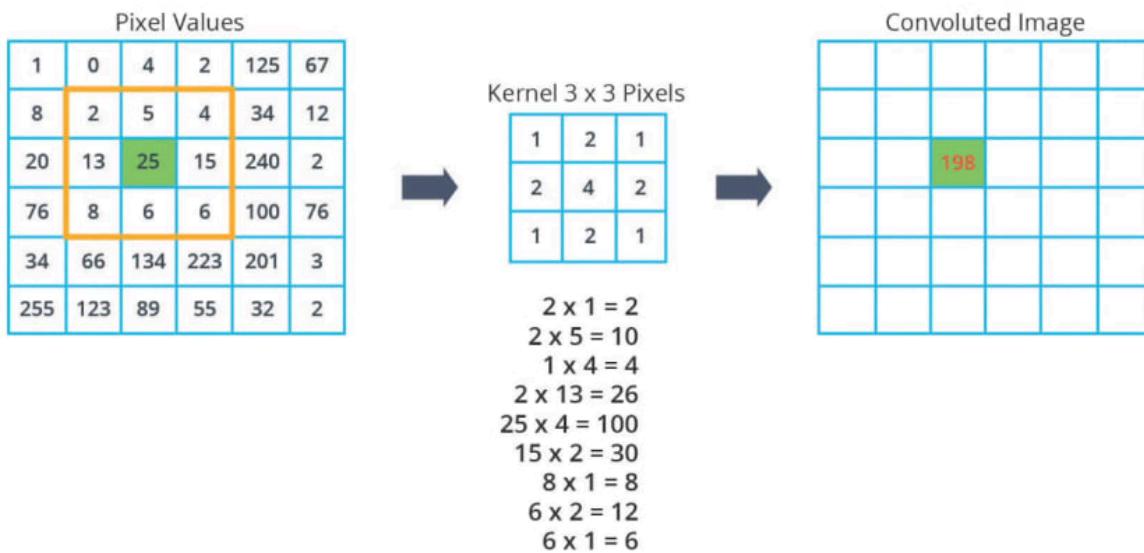
Hem suposat que la finestra avança 1 pixel amb cada pas. Ara bé, es poden fer servir altres longituds de pas, anomenat en anglès *stride*. També és habitual afegir zeros a la vora de la imatge per mantenir la mida de la imatge en comptes d'anar-la reduint en cada capa. Aquest farciment s'anomena *padding* en anglès.

Per tant, a l'hora de connectar cada neurona de la capa oculta amb les seves 25 neurones corresponents de la capa d'entrada, farem servir una matriu de pesos W , en aquest cas de mida 5x5, més un biaix b .

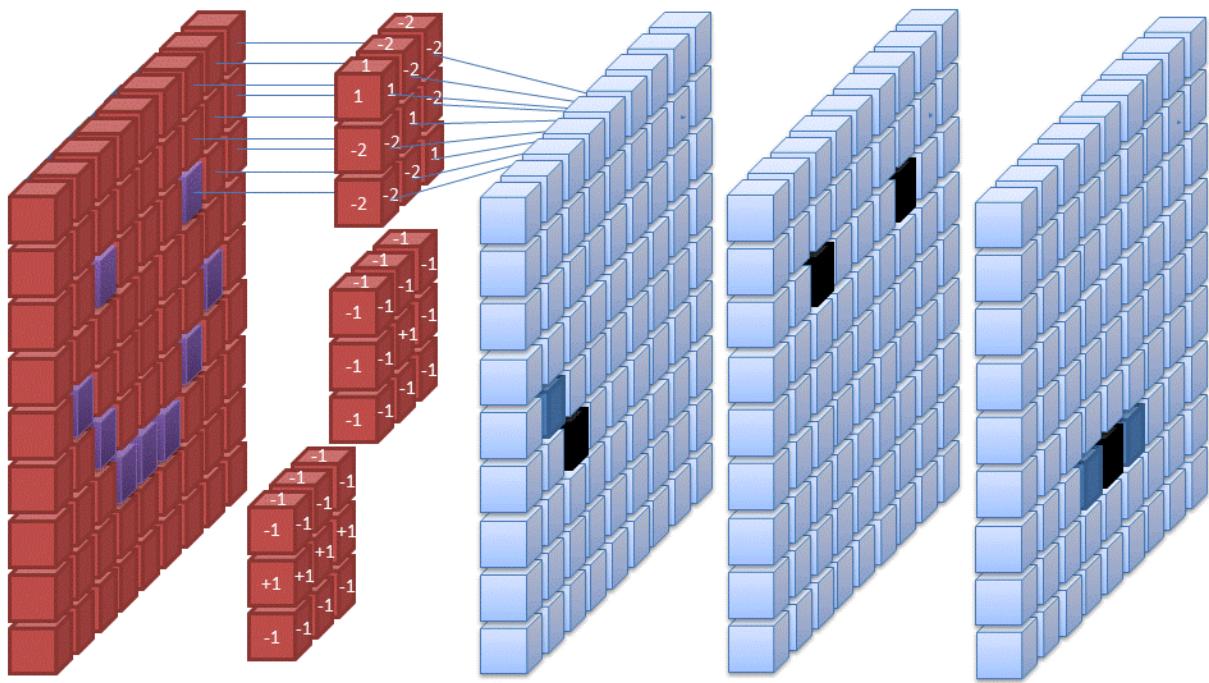
La característica clau de les CNN és que s'usen els mateixos paràmetres W i b per a totes les neurones d'una mateixa capa. Això permet trobar les formes visuals d'interès en qualsevol punt de la imatge, al mateix temps que es redueix molt el nombre de paràmetres. Si la connexió fos completa, entre aquestes dues capes tendríem $5 \times 5 \times 24 \times 24 = 14400$ pesos, mentre que en tenim només $5 \times 5 = 25$ si es tracta d'una xarxa convolucional.

Podem veure exemples de l'aplicació de diversos *kernels* o filtres sobre una mateixa imatge a la documentació del programa d'edició de gràfics GIMP, a l'adreça <https://docs.gimp.org/2.6/en/plug-in-convmatrix.html>. En aquests exemples, els filtres són predefinits manualment d'acord amb les característiques que es volen trobar. En xarxes neuronals, en canvi, els coeficients dels filtres es determinen automàticament en el procés d'aprenentatge a partir del conjunt de dades d'entrada, per maximitzar els resultats de classificació.

A continuació, vegem amb un exemple com es calcula el resultat d'aplicar un filtre sobre una determinada regió d'una imatge. Aquí només hi ha la matriu de pesos W , falta el biaix b .



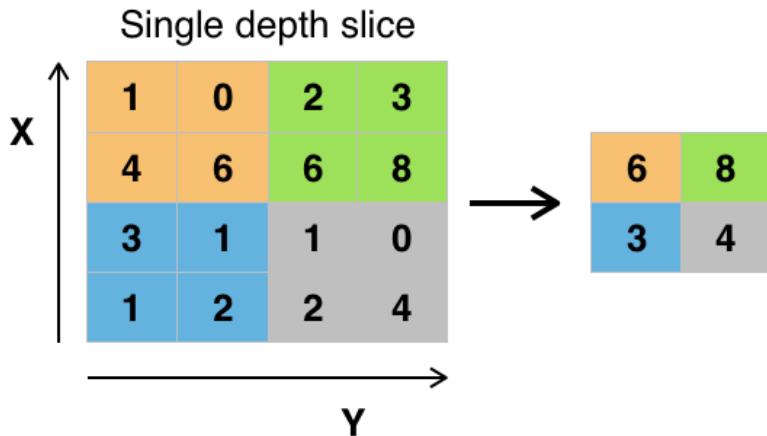
Per acabar, vegem com amb diferents filtres es poden obtenir diferents formes bàsiques, ja siguin punts aïllats, línies horitzontals o línies inclinades.



2.2. Pooling

A més de les capes convolucionals de l'apartat anterior, a les CNN fan servir unes capes de *pooling* (que podem traduir com **agrupació**), que se solen aplicar just després de les capes convolucionals. Podem entendre que la funció d'aquestes capes de *pooling* és **simplificar i condensar** la informació.

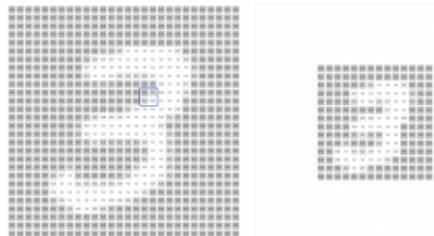
Per exemple, a la il·lustració següent es resumeix la informació de quatre punts contigus en un sol punt, mitjançant una finestra de *pooling* de 2x2.



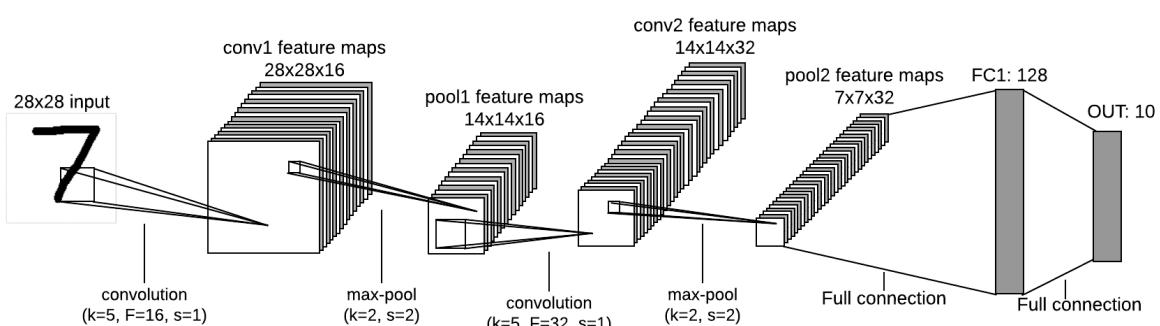
Hi ha diverses formes de condensar la informació, una de les més habituals és l'anomenada **max-pooling**, i és la que mostra la imatge anterior. Observem que a la imatge condensada, després d'haver aplicat el *pooling*, s'han seleccionat els valors màxims de les regions 2x2 on s'ha aplicat el procediment. Observem també que una capa de *pooling* 2x2 aplicada sobre imatges de 24x24 les deixarà en imatges 12x12.

Una alternativa a max-pooling és **average-pooling**, en què en lloc del màxim es calcula el valor promig de la regió enfinestrada. En general, però, **max-pooling** sol funcionar molt bé.

És interessant observar que la transformació de *pooling* manté la relació espacial. Vegem-ho a la imatge següent, obtinguda de <https://androidkt.com/explain-pooling-layers-max-pooling-average-pooling-global-average-pooling-and-global-max-pooling/>



Finalment vegem la concatenació de capes convolucionals i de *pooling* en un sistema complet. Observem que la mida de les sortides de la primera capa convolucional no és 24x24, sinó 28x28. Això s'aconsegueix aplicant padding, que consisteix en enrevoltar la imatge de zeros per aconseguir que la mida no varii.



La imatge ve de <https://www.easy-tensorflow.com/tf-tutorials/convolutional-neural-nets-cnns/cnn1?view=article&id=108:cnn>

3. Implementació d'una CNN bàsica en Keras

Ja estam en condicions d'implementar usant Keras una primera xarxa convolucional. Heurem d'especificar els paràmetres de convolució i *pooling*. Usarem un *stride 1* (pas amb què es desplaça la finestra) i un *padding* de 0, sense farciment. Ja veurem a la següent secció com modificar aquests paràmetres. Aplicarem *max-pooling* amb una finestra de mida 2x2.

3.1. Arquitectura d'una CNN

Implementem la nostra primera xarxa neuronal convolucional, que consistirà en una convolució seguida de *max-pooling*. Especificarem 32 filtres, una finestra de mida 5x5 per a la convolució i una finestra de mida 2x2 per al *max-pooling*. Triarem com a funció d'activació lineal rectificada ReLU. Les imatges són en escala de grisos (un canal), quadrades de 28 píxels de costat. Per tant, el tensor d'entrada és de mida (28, 28, 1). El codi Keras és el següent.

```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Conv2D
from tensorflow.keras.layers import MaxPooling2D

model = Sequential()
model.add(Conv2D(32,(5,5), activation = 'relu', input_shape=(28,28,1)))
model.add(MaxPooling2D((2,2)))
```

Amb el mètode **summary()** podem conèixer els detalls sobre el nombre de paràmetres de cada capa i sobre el nombre i la forma dels filtres de cada capa.

El nombre de paràmetres de la capa **conv2D** és 832. Això ve de considerar els 5x5 pesos més el biaix, per a cadascun dels 32 filtres: $32 \times (5 \times 5 + 1) = 832$. Tots els paràmetres són entrenables, tant els pesos com els biaixos. En canvi, la capa max-pooling no usa paràmetres, ja que simplement obté el màxim de cada regió. Per això el nombre de paràmetres indicats pel resum és 0.

3.2. Definició del model

Per aconseguir una xarxa neuronal profunda (*deep*) apilarem diverses capes. El següent grup de capes tendrà 64 filtres de mida 5x5 a la capa convolucional seguits d'una capa de **max-pooling** 2x2. En aquesta capa convolucional ja no fa falta que especifiquem la mida de l'entrada, perquè Keras la dedueix automàticament. El codi és el següent.

```
model = Sequential()
model.add(Conv2D(32, (5,5), activation='relu', input_shape=(28,28,1)))
model.add(MaxPooling2D((2,2)))
model.add(Conv2D(64,(5,5),activation='relu'))
model.add(MaxPooling2D((2,2)))
model.summary()
```

En aquest cas, obtenim que la mida de la sortida de la segona capa de convolució és 8x8. Això ve d'aplicar una finestra 5x5 amb *stride* 1 a la sortida de la capa anterior, que era de 12x12: $12 - 5 + 1 = 8$. El nombre de paràmetres és 51264 perquè la segona capa té 64 filtres, amb 801 paràmetres cada un: 1 és el biaix i llavors hi ha la matriu W de 5x5 per a cada una de les 32 entrades. És a dir, el càlcul és $64((5 \times 5 \times 32) + 1) = 51264$.

Les dimensions *width* i *height* tendeixen a fer-se més petites a mesura que avançam cap a les capes ocultes de la xarxa. El nombre de filtres el determina el primer argument de la capa **Conv2D**.

El pas següent, ara que tenim 64 filtres de 4x4, és afegir una capa densament connectada, que servirà per alimentar una capa final de **softmax** per fer la classificació.

```
model.add(layers.Dense(10,activation='softmax'))
```

Abans, però, hem d'ajustar les dimensions dels tensors. Tenim que a la sortida de la segona capa de **max-pooling** hi ha un tensor **3D**. L'entrada de la capa **softmax** és un tensor **1D**. Per tant, cal adaptar aquestes dimensions, mitjançant una capa intermèdia d'aplanament, **Flatten**. La sortida (4, 4, 64) passarà a un vector (1024) abans d'aplicar la **softmax**.

El codi complet és el següent.

```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Conv2D
from tensorflow.keras.layers import MaxPooling2D
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Flatten

model = Sequential()
model.add(Conv2D(32,(5,5), activation='relu', input_shape=(28,28,1)))
model.add(MaxPooling2D((2,2)))
model.add(Conv2D(64,(5,5),activation='relu'))
model.add(MaxPooling2D((2,2)))
model.add(Flatten())
model.add(Dense(10,activation='softmax'))

model.summary()
```

En aquest cas, el nombre de paràmetres de la capa softmax és 10250, és a dir $10 \times 1024 + 10$.

3.3. Configuració, entrenament i evaluació

Després d'haver definit el model de la xarxa neuronal convolucional, estam en condicions d'entrenar el model, ajustar els paràmetres de totes les capes de la xarxa convolucional.

Al codi següent, destaquem que el canvi de forma de les imatges té en compte que la capa convolucional pren com a entrada tensors 3D. Aquest canvi de forma el realitzarem amb el mètode **reshape()**.

```
from tensorflow.keras.utils import to_categorical

mnist = tf.keras.datasets.mnist
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()

train_images = train_images.reshape((60000, 28, 28, 1))
train_images = train_images.astype('float32') / 255

test_images = test_images.reshape((10000, 28, 28, 1))
test_images = test_images.astype('float32') / 255

train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)

model.compile ( loss='categorical_crossentropy',
                optimizer = 'sgd',
                metrics = ['accuracy'])

model.fit(train_images, train_labels,
          batch_size = 100,
          epochs=5,
          verbose=1)

test_loss, test_acc = model.evaluate(test_images, test_labels)

print('Test accuracy' , test_acc)
```

Amb aquest codi obtenim una precisió del 97%.

4. Hiperparàmetres

Hi ha quatre hiperparàmetres principals a les capes convolucionals de les CNN. Són els següents.

- Mida de la finestra del filtre
- Nombre de filtres
- Mida del pas d'avanc (stride)
- Farciment (padding)

Vegem-los en detall en els apartats següents.

4.1. Mida i nombre de filtres

La **mida de la finestra** (*window_height x window_width*) que conté informació dels píxels propers sol ésser de 3x3 o bé de 5x5. Observem que és un nombre senar perquè estigui centrada en un píxel.

El **nombre de filtres**, que indica quantes característiques volem calcular (*output_depth*) sol ésser 32 o 64.

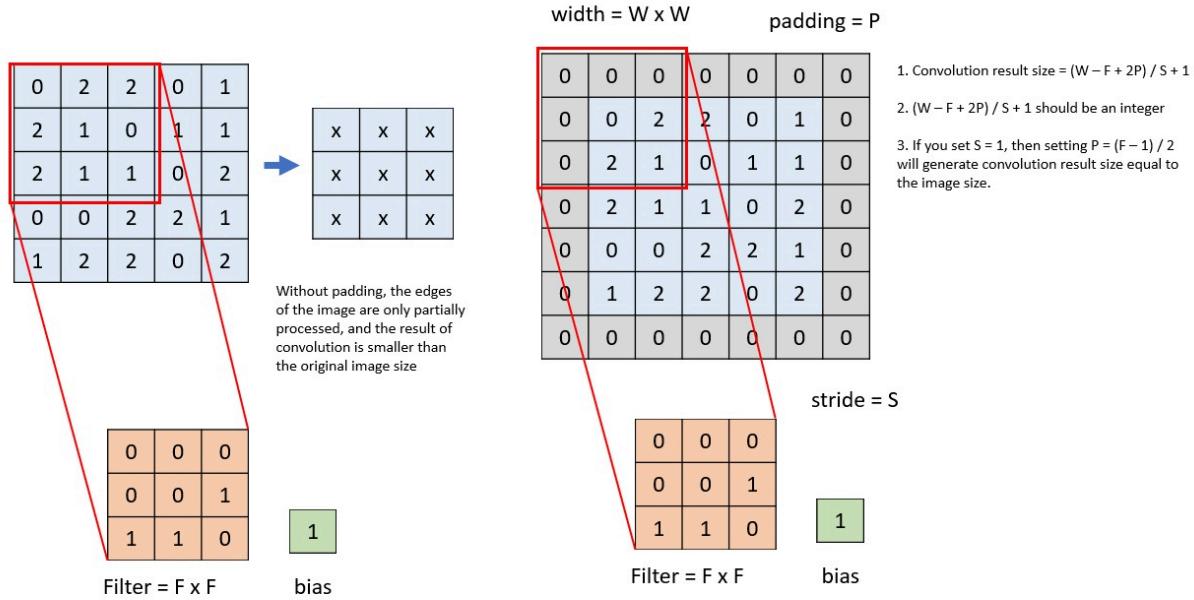
A les capes Conv2D de Keras aquests hiperparàmetres són els arguments que passam en l'ordre estableert.

```
Conv2D ( output_depth, (window_height, window_width))
```

Podem consultar més paràmetres i exemples d'ús a la [documentació de la capa convolucional 2D de Keras](#).

4.2. Padding

Vegem la diferència entre usar *padding* o no usar-ne amb el següent exemple, extret del blog de James D. McCaffrey (<https://jamesmccaffrey.wordpress.com/2018/05/30/convolution-image-size-filter-size-padding-and-stride/>)



Considerem la imatge de mida 5×5 de l'esquerra, i el filtre de mida 3×3 . Amb un pas *stride* igual a 1, podem situar el filtre en un total de 9 posicions sobre la imatge, cosa que correspon a una imatge resultat de mida 3×3 .

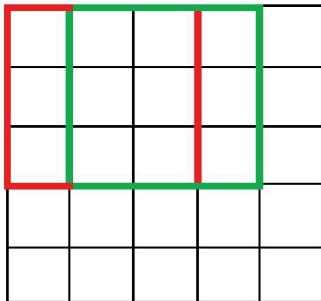
En canvi, a la dreta, veim que afegint un marge de zeros al voltant de la imatge, podem aconseguir una imatge resultat de la mateixa mida que la imatge d'entrada, 5×5 .

La forma d'indicar com volem usar el padding en Keras, és especificant l'argument **padding='valid'**, si volem el resultat de l'esquerra, o bé **padding='same'**, si volem mantenir la mida de la imatge.

4.3. Stride

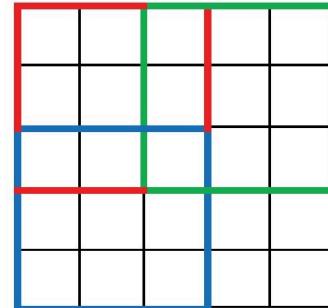
El paràmetre *stride* regula el pas d'avanç del filtre o *kernel* per sobre de la imatge que es processa. Vegem en aquest exemple com amb diferents valors de *stride* la mida de la imatge resultant varia.

**Convolution
with Stride=1**



Output

**Convolution
with Stride=2**



Output

Imatge: <https://www.analyticsvidhya.com/blog/2022/03/basics-of-cnn-in-deep-learning/>

Habitualment la reducció de la informació es realitza en les capes de **pooling**. Per això el valor per defecte de l'argument és **strides=(1,1)**, que correspon a avançar la finestra píxel a píxel, sense botar-ne cap.

5. Aplicació al conjunt de dades Fashion-MNIST

En aquest apartat tornarem a utilitzar el conjunt de dades Fashion MNIST que ja empràrem al lliurament anterior de Sistemes d'Aprenentatge Automàtic. Ara això ens servirà per a dues coses: reforçar els conceptes de xarxes neuronals anteriors i introduir un parell d'idees noves: dropout, batch normalization i la disminució de la taxa d'aprenentatge.

5.1. Model bàsic

Comencem carregant les dades i preparant-les per entrar a la xarxa neuronal.

```
fashion_mnist = keras.datasets.fashion_mnist
(train_images, train_labels), (test_images, test_labels) =
fashion_mnist.load_data()

class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat', 'Sandal', 'Shirt',
'Sneaker', 'Bag', 'Ankle_boot']

train_images = train_images.reshape((60000,28,28,1))
train_images = train_images.astype('float32') / 255

test_images = test_images.reshape((10000, 28, 28, 1))
test_images = test_images.astype('float32') / 255
```

La xarxa neuronal convolucional espera un tensor 3D, per això hem hagut de fer **reshape** del tensor 2D afegint-hi una dimensió.

Com a punt de partida, considerem la mateixa xarxa en què hem classificat els dígits MNIST a la secció anterior.

```
model = Sequential()
model.add(Conv2D(32, (5,5), activation = 'relu', input_shape = (28,28,1)))
model.add(MaxPooling2D((2,2)))
model.add(Conv2D(64, (5,5), activation = 'relu'))
model.add(MaxPooling2D((2,2)))
model.add(Flatten())
model.add(Dense(10, activation='softmax'))

model.compile(optimizer='sgd',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.fit(train_images, train_labels, epochs=5)

test_loss, test_acc = model.evaluate(test_images, test_labels)

print('Test accuracy:', test_acc)
```

La precisió que aconsegueix aquesta xarxa convolucional supera el 85%, mentre que la xarxa densament connectada del lliurament anterior no arribava al 80%. Això mostra l'avantatge de les xarxes convolucionals per a la classificació d'imatges respecte de les xarxes densament connectades.

5.2. Capes i optimitzadors

Podem intentar millorar els resultats modificant els hiperparàmetres. Una primera opció és afegir més capes a la xarxa. Per exemple, considerem el codi següent.

```
model = Sequential()

model.add(Conv2D(64,(7,7), activation='relu', padding='same', input_shape=(28,28,1)))
model.add(MaxPooling2D(2,2))
model.add(Conv2D(128,(3,3), activation='relu', padding='same'))
model.add(MaxPooling2D(2,2))
model.add(Flatten())
model.add(Dense(64,activation='relu'))
model.add(Dense(10,activation='softmax'))
```

En aquest model hem afegit el doble de neurones a les capes convolucionals i hem afegit una capa densa abans del classificador final. També il·lustrem l'ús del paràmetre **padding**.

Canviem també l'optimitzador. En lloc d'SGD usarem Adam.

```
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])

model.fit(train_images, train_labels, epochs=5)

test_loss, test_acc = model.evaluate(test_images, test_labels)

print('\nTest accuracy:', test_acc)
```

Amb aquests canvis aconseguim una millora notable de la precisió, que ara passa del 90%. Els hiperparàmetres juguen un paper molt important a l'hora de trobar un model que s'ajusti bé a les dades.

5.3. Capes de Dropout i BatchNormalization

A continuació introduirem dos tipus de capes noves, **Dropout** i **BatchNormalization**. Comencem observant el seu ordre en el codi següent.

```
from tensorflow.keras.layers import Dropout, BatchNormalization

model = Sequential()
model.add(Conv2D(filters=32, kernel_size=(3,3),
                 activation='relu', strides=1, padding='same',
                 input_shape=(28,28,1)))
model.add(BatchNormalization())

model.add(Conv2D(filters=32, kernel_size=(3,3),
                 activation='relu', strides=1, padding='same'))
model.add(BatchNormalization())
model.add(Dropout(0.25))

model.add(Conv2D(filters=64, kernel_size=(3,3),
                 activation='relu', strides=1, padding='same'))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Dropout(0.25))

model.add(Conv2D(filters=128, kernel_size=(3,3),
                 activation='relu', strides=1, padding='same'))
model.add(BatchNormalization())
model.add(Dropout(0.25))

model.add(Flatten())
model.add(Dense(512, activation='relu'))
model.add(BatchNormalization())
model.add(Dropout(0.5))
model.add(Dense(128, activation='relu'))
model.add(BatchNormalization())
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))
```

La capa [BatchNormalization](#) usa una idea introduïda el 2015, que consisteix a normalitzar les entrades de la capa de forma que tenguin aproximadament una mitjana igual a $\langle 0 \rangle$ i una desviació típica $\langle 1 \rangle$. Això és anàleg a com s'estandarditzen les entrades a les xarxes.

La capa Dropout aplica una de les tècniques més usades per ajudar a evitar el sobreajust dels models. Consisteix a ignorar una part de les neurones durant la fase d'entrenament, de forma aleatòria.

```
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
model.fit(train_images, train_labels, epochs=5)

test_loss, test_acc = model.evaluate(test_images, test_labels)
print('\nTest accuracy:', test_acc)
```

Amb aquesta arquitectura de xarxa i els hiperparàmetres indicats, es pot aconseguir una precisió superior al $\langle 92 \rangle\%$.

5.4. Disminució de la taxa d'aprenentatge

Un altre hiperparàmetre important és la taxa d'aprenentatge.

Per ajustar-la dinàmica, Keras disposa del callback [LearningRateScheduler](#), que pren la funció de reducció del pas d'aprenentatge com a argument i en retorna el valor actualitzat per usar a l'optimitzador en cada epoch. Al següent codi veim com es pot especificar.

```
optimizer = tf.keras.optimizers.Adam(lr=0.001)

model.compile(optimizer=optimizer,
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

reduce_lr = tf.keras.callbacks.LearningRateScheduler
            (lambda x: 1e-3 * 0.9 **x)

model.fit(train_images, train_labels, epochs=30,
          callbacks=[reduce_lr])

test_loss, test_acc = model.evaluate(test_images, test_labels)

print('\nTest accuracy:', test_acc)
```

Amb aquest canvi, arribam a una precisió de més del \((94)\)%.

Però, què és l'argument callbacks? Un callback és una eina per personalitzar el comportament d'un model de Keras durant l'entrenament, l'avaluació o la inferència. A més de LearningRateScheduler, Keras ofereix diversos [callbacks](#) diferents.

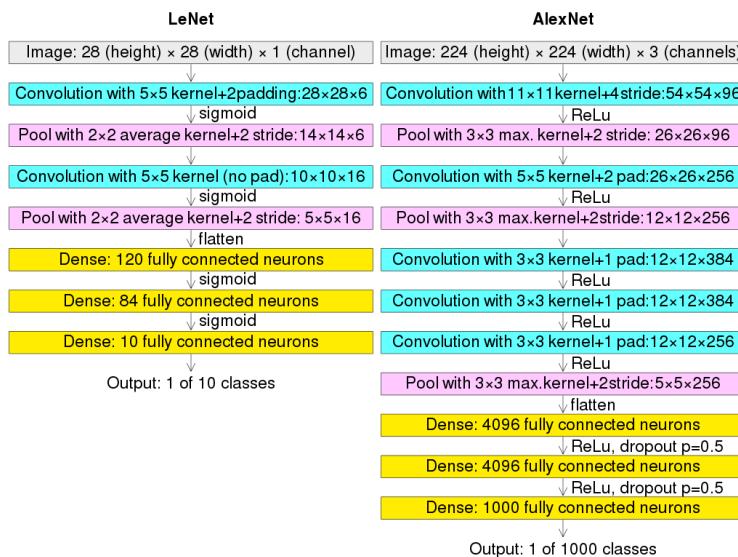
6. Xarxes neuronals preentrenades

Arquitectures amb nom propi

La competició **ImageNet** ha estat un gran impuls en la recerca i desenvolupament d'arquitectures de xarxes neuronals.

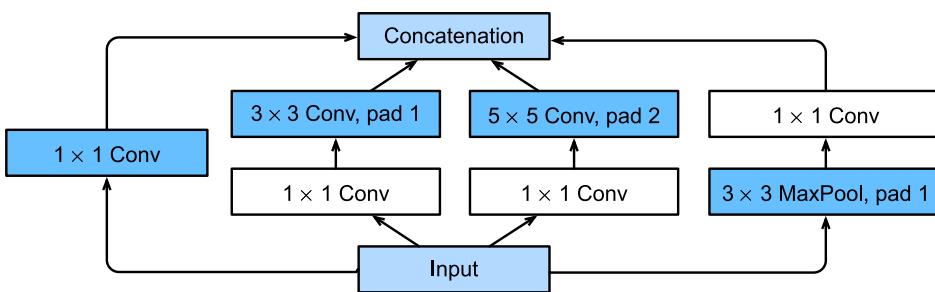
Algunes de les propostes guanyadores d'aquesta competició han esdevingut arquitectures de referència i són conegudes pel seu nom: **AlexNet**, **GoogLeNet**, **VGG** o **ResNet**.

L'arquitectura **AlexNet** guanyà la competició l'any 2012 gràcies a l'ús de GPU. L'arquitectura de la xarxa usada és semblant a LeNet-5, però molt més grossa: 60 milions de paràmetres. Va ser la primera proposta que empilava capes convolucionals directament una damunt l'altra, en comptes de posar una capa de *pooling* entre dues capes convolucionals. L'arquitectura està formada per 5 capes convolucionals, 3 capes d'agrupament (*pooling*) i 3 de denses.

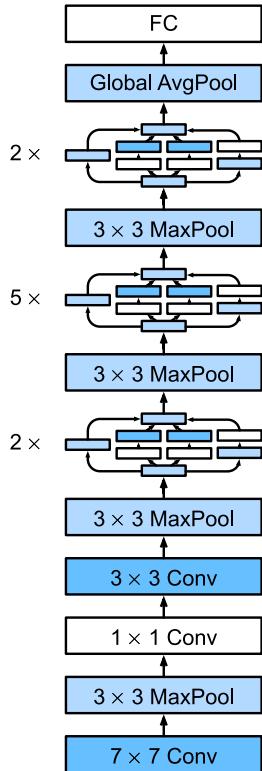


Imatge: <https://en.wikipedia.org/wiki/AlexNet>

L'arquitectura **GoogLeNet** ve d'un equip de recerca de Google Research i guanyà la competició el 2014. El seu èxit es basa en la profunditat molt més gran que les CNN anteriors. Això és possible gràcies a les subxarxes anomenades **inception**, que permeten usar els paràmetres de forma molt més eficient. Tot i que té moltes més capes, en realitat GoogLeNet té molts més pocs paràmetres que AlexNet (devers 6 milions, comparats amb els 60 milions d'AlexNet).



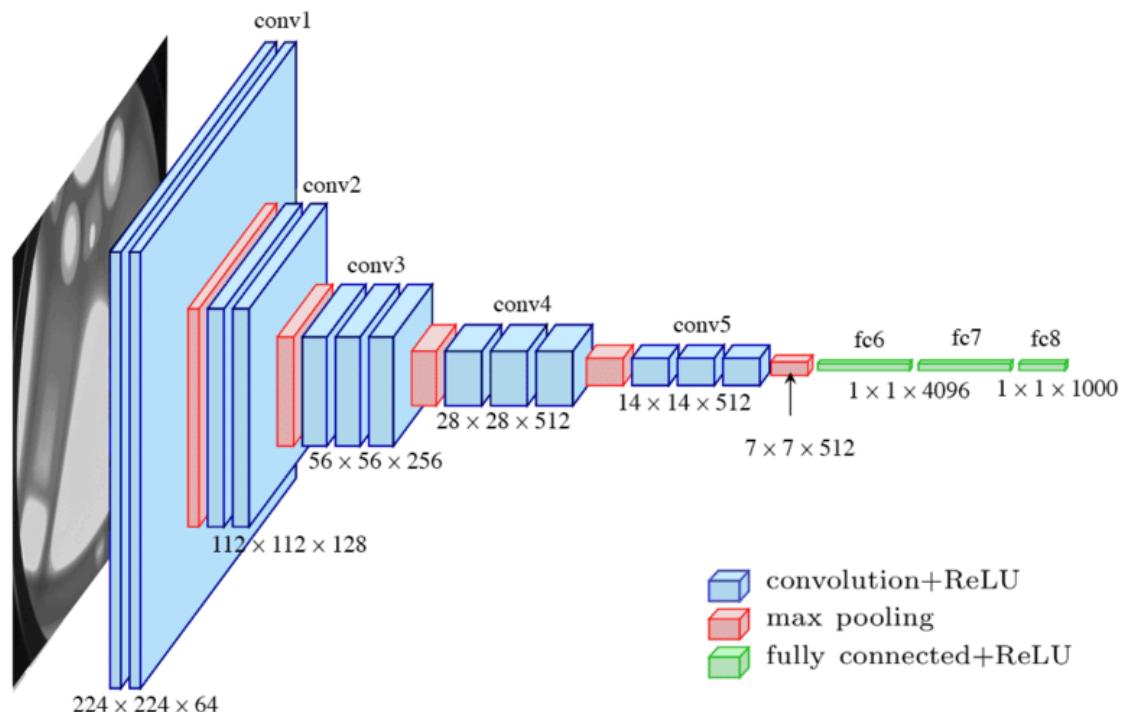
Bloc *inception*



Arquitectura GoogLeNet

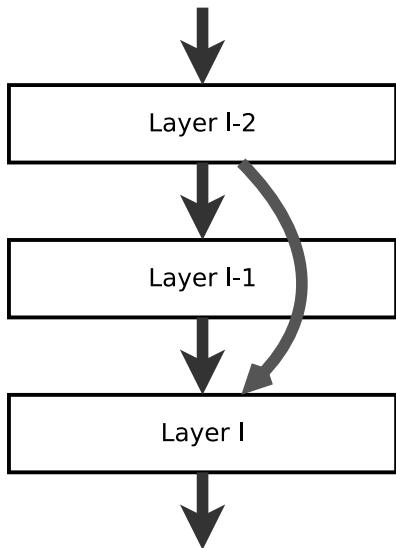
Imatges: https://classic.d2l.ai/chapter_convolutional-modern/googlenet.html

L'any 2014 mateix, el segon lloc de la competició el va aconseguir **VGGNet**, una xarxa d'un grup de recerca de la Universitat d'Oxford. Tenia una arquitectura molt simple i clàssica, amb 2 o 3 capes convolucionals i una capa de pooling, després novament 2 o 3 capes convolucionals i una capa de *pooling*, fins a un total de 16 o 19 capes convolucionals (conegeudes com a VGG16 i VGG19, respectivament), més una part densa final amb 2 capes ocultes i la capa de sortida. Fa servir molts de filtres de mida 3x3.



Arquitectura VGGNet16. Imatge: Siddhesh Bangar a <https://medium.com/@siddheshb008/vgg-net-architecture-explained-71179310050f>

L'any 2015 guanyà la competició l'arquitectura **ResNet**, d'un grup de Microsoft. Era una arquitectura molt profunda amb 152 capes. Aquesta arquitectura segueix la tendència general de més capes però més pocs paràmetres. La clau per poder entrenar una xarxa tan profunda és a les anomenades *skip connections*, que fan que el senyal que alimenta una capa s'afegeixi també a una capa situada una mica més endavant. S'han fet populars les variants amb 34, 50 i 101 capes.



Il·lustració de les *skip connections* a les **ResNet**. Imatge: https://en.wikipedia.org/wiki/Residual_neural_network

Aquestes són quatre de les més populars. Se'n poden consultar més a l'article [Recent Advances in Convolutional Neural Networks](#).

6.1. API funcional de keras

En el quadern de Colab següent hi ha un seguit d'exemples d'arquitectures avançades de xarxes neuronals extretes del capítol 19 del llibre Python Deep Learning de Jordi Torres.

<https://colab.research.google.com/drive/1xlyifCCbsffSjyy-UmGen4QIAWg3gTha?usp=sharing>

6.2. Conjunt de dades CIFAR-10

Als següents apartats illustrarem com usar xarxes pre-entrenades, aplicant-les al conjunt de dades [CIFAR-10](#), proporcionat pel Canadian Institute For Advanced Research. Aquest dataset té 60000 imatges en color de 32x32 píxels classificades en 10 classes, amb 6000 imatges per classe. Hi ha 50000 imatges d'entrenament i 10000 de prova.

Podem carregar el conjunt de dades amb el codi següent, extret del quadern de Colab [Xarxes neuronals pre-entrenades](#).

```
from tensorflow.keras.datasets.cifar10 import load_data
import matplotlib.pyplot as plt

(train_images, train_labels), (test_images, test_labels) = load_data()
train_images, test_images = train_images/255.0, test_images/255.0
```

A continuació, podem inspeccionar les dades així.

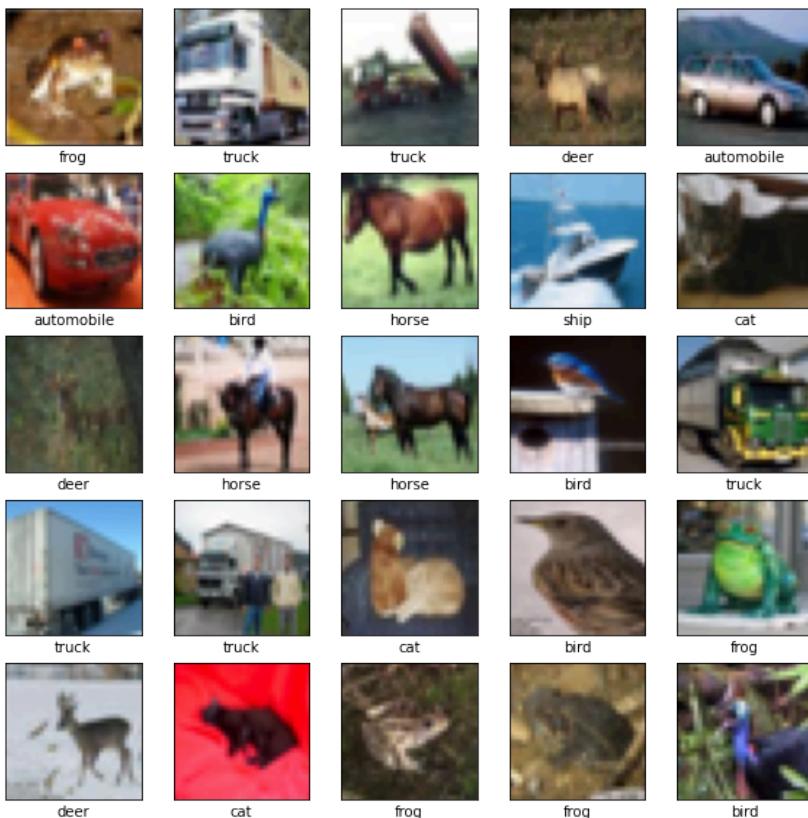
```
class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',
'ship', 'truck']

plt.figure(figsize=(10,10))

for i in range(25):
    plt.subplot(5,5,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(train_images[i], cmap=plt.cm.binary)

    plt.xlabel(class_names[train_labels[i][0]])
plt.show()
```

I obtendrem el resultat següent.



Codi i imatges: Quadern de Colab [Xarxes neuronals pre-entrenades](#).

6.3. ResNet50

Començam l'ús de xarxes pre-entrenades amb ResNet50. És una xarxa que té més de 25 milions de paràmetres distribuïts en moltíssimes capes. Feim servir la versió optimitzada ResNet50V2 que ofereix Keras. En el primer codi d'exemple, usam la xarxa sense els pesos entrenats sobre ImageNet.

```
import tensorflow as tf

modelresnet50v2 = tf.keras.applications.ResNet50V2(include_top=True, weights=None,
input_shape=(32,32,3), classes=10)

opt = tf.keras.optimizers.SGD(0.002)

modelresnet50v2.compile(
    loss='sparse_categorical_crossentropy',
    optimizer = opt,
    metrics = ['accuracy']
)

history = modelresnet50v2.fit(train_images, train_labels, epochs=10, validation_data=
(test_images, test_labels))
```

La precisió obtinguda amb les dades de prova és del 57%

Aquest segon codi d'exemple usa els pesos pre-entrenats sobre ImageNet.

```
modelresnet50v2pre = tf.keras.Sequential()

modelresnet50v2pre.add(tf.keras.applications.ResNet50V2(
    include_top=False,
    weights='imagenet',
    pooling='avg',
    input_shape=(32,32,3)))

modelresnet50v2pre.add(tf.keras.layers.Dense(10,activation='softmax'))
opt = tf.keras.optimizers.SGD(0.002)
modelresnet50v2pre.compile(loss='sparse_categorical_crossentropy',
                           optimizer = opt,
                           metrics = ['accuracy'])

history = modelresnet50v2pre.fit(train_images, train_labels, epochs=10, validation_data=
(test_images, test_labels))
```

El resultat millora fins al 73%.

6.4. VGG19

Amb la xarxa més avançada VGG19, pre-entrenada sobre ImageNet, s'obté una precisió superior al 80%. El codi per aconseguir-ho és el següent.

```
model = tf.keras.Sequential()
model.add(tf.keras.applications.VGG19(include_top=False,
                                       weights='imagenet', pooling='avg', input_shape=(32,32,3)))

model.add(tf.keras.layers.Dense(10, activation='softmax'))

opt = tf.keras.optimizers.SGD(0.002)

model.compile(loss='sparse_categorical_crossentropy', optimizer=opt, metrics=['accuracy'])

history=model.fit(train_images, train_labels, epochs=10, validation_data=
(test_images,test_labels))
```

7. Xarxes neuronals convolucionals amb keras

Les aplicacions de la llibreria **keras** a problemes de [visió per computador](#) s'agrupen en una diversitat d'àrees. Donarem un exemple complet de les tres següents.

- Classificació d'imatges
- Segmentació d'imatges
- Detecció d'objectes

7.1. Classificació d'imatges

L'exemple que incloem de la secció de classificació d'imatges es refereix al conjunt de dades MNIST de dígits manuscrit, resolt amb xarxes convolucionals.

El codi està extret de l'adreça https://keras.io/examples/vision/mnist_convnet/

Per començar, importam les llibreries necessàries.

Setup

```
import numpy as np
import keras
from keras import layers
```

A continuació, preparam les dades. Observam que les imatges tenen unes dimensions de 28x28 píxels.

Prepare the data

```
# Model / data parameters
num_classes = 10
input_shape = (28, 28, 1)

# Load the data and split it between train and test sets
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()

# Scale images to the [0, 1] range
x_train = x_train.astype("float32") / 255
x_test = x_test.astype("float32") / 255
# Make sure images have shape (28, 28, 1)
x_train = np.expand_dims(x_train, -1)
x_test = np.expand_dims(x_test, -1)
print("x_train shape:", x_train.shape)
print(x_train.shape[0], "train samples")
print(x_test.shape[0], "test samples")

# convert class vectors to binary class matrices
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)
```

```
x_train shape: (60000, 28, 28, 1)
60000 train samples
10000 test samples
```

Ja podem construir el model, seqüenciant les diverses capes de neurones artificials. A les capes interiors, utilitzam la funció d'activació ReLU, mentre que a la capa final l'activació és softmax.

Build the model

```
model = keras.Sequential(
    [
        keras.Input(shape=input_shape),
        layers.Conv2D(32, kernel_size=(3, 3), activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Flatten(),
        layers.Dropout(0.5),
        layers.Dense(num_classes, activation="softmax"),
    ]
)

model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 26, 26, 32)	320
max_pooling2d (MaxPooling2D)	(None, 13, 13, 32)	0
conv2d_1 (Conv2D)	(None, 11, 11, 64)	18,496
max_pooling2d_1 (MaxPooling2D)	(None, 5, 5, 64)	0
flatten (Flatten)	(None, 1600)	0
dropout (Dropout)	(None, 1600)	0
dense (Dense)	(None, 10)	16,010

Total params: 34,826 (136.04 KB)

Trainable params: 34,826 (136.04 KB)

Non-trainable params: 0 (0.00 B)

Una vegada hem configurat la xarxa neuronal, la podem entrenar amb quinze iteracions.

Train the model

```
batch_size = 128
epochs = 15

model.compile(loss="categorical_crossentropy", optimizer="adam", metrics=["accuracy"])

model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs, validation_split=0.1)
```

```
Epoch 1/15
422/422 —————— 7s 9ms/step - accuracy: 0.7668 - loss: 0.7644 - val_accuracy: 0.98
Epoch 2/15
422/422 —————— 1s 2ms/step - accuracy: 0.9627 - loss: 0.1237 - val_accuracy: 0.98
Epoch 3/15
422/422 —————— 1s 2ms/step - accuracy: 0.9732 - loss: 0.0898 - val_accuracy: 0.98
Epoch 4/15
422/422 —————— 1s 2ms/step - accuracy: 0.9761 - loss: 0.0763 - val_accuracy: 0.98
Epoch 5/15
422/422 —————— 1s 2ms/step - accuracy: 0.9795 - loss: 0.0647 - val_accuracy: 0.98
Epoch 6/15
422/422 —————— 1s 2ms/step - accuracy: 0.9824 - loss: 0.0580 - val_accuracy: 0.99
Epoch 7/15
422/422 —————— 1s 2ms/step - accuracy: 0.9828 - loss: 0.0537 - val_accuracy: 0.98
Epoch 8/15
422/422 —————— 1s 2ms/step - accuracy: 0.9838 - loss: 0.0503 - val_accuracy: 0.99
Epoch 9/15
422/422 —————— 1s 2ms/step - accuracy: 0.9861 - loss: 0.0451 - val_accuracy: 0.99
Epoch 10/15
422/422 —————— 1s 2ms/step - accuracy: 0.9866 - loss: 0.0427 - val_accuracy: 0.99
Epoch 11/15
422/422 —————— 1s 2ms/step - accuracy: 0.9871 - loss: 0.0389 - val_accuracy: 0.99
Epoch 12/15
422/422 —————— 1s 2ms/step - accuracy: 0.9885 - loss: 0.0371 - val_accuracy: 0.99
Epoch 13/15
422/422 —————— 1s 2ms/step - accuracy: 0.9901 - loss: 0.0332 - val_accuracy: 0.99
Epoch 14/15
422/422 —————— 1s 2ms/step - accuracy: 0.9885 - loss: 0.0340 - val_accuracy: 0.99
Epoch 15/15
422/422 —————— 1s 2ms/step - accuracy: 0.9891 - loss: 0.0326 - val_accuracy: 0.99

<keras.src.callbacks.history.History at 0x7f8497818af0>
```

Finalment, avaluem l'exactitud (accuracy) del model, superior al 99%.

Evaluate the trained model

```
score = model.evaluate(x_test, y_test, verbose=0)
print("Test loss:", score[0])
print("Test accuracy:", score[1])
```

```
Test loss: 0.02499214932322502
Test accuracy: 0.9919000267982483
```

7.2. Segmentació d'imatges

A les aplicacions de segmentació estam interessats a distingir el contorn de les figures, quins píxels de la imatge formen part d'un determinat objecte i quins no. A la frontera entre les dues grans zones, hi haurà uns punts intermedis que separaran les dues regions.

Aquest exemple és a l'adreça https://keras.io/examples/vision/oxford_pets_image_segmentation/

Per començar, descarregem les dades del conjunt pets del grup de visió de la Universitat d'Oxford.

```
!!wget https://www.robots.ox.ac.uk/~vgg/data/pets/data/images.tar.gz
!!wget https://www.robots.ox.ac.uk/~vgg/data/pets/data/annotations.tar.gz
!
!curl -O https://thor.robots.ox.ac.uk/datasets/pets/images.tar.gz
!curl -O https://thor.robots.ox.ac.uk/datasets/pets/annotations.tar.gz
!
!tar -xf images.tar.gz
!tar -xf annotations.tar.gz
```

Seguidament, les organitzam als seus directoris.

```
import os

input_dir = "images/"
target_dir = "annotations/trimaps/"
img_size = (160, 160)
num_classes = 3
batch_size = 32

input_img_paths = sorted(
    [
        os.path.join(input_dir, fname)
        for fname in os.listdir(input_dir)
        if fname.endswith(".jpg")
    ]
)
target_img_paths = sorted(
    [
        os.path.join(target_dir, fname)
        for fname in os.listdir(target_dir)
        if fname.endswith(".png") and not fname.startswith(".")
    ]
)

print("Number of samples:", len(input_img_paths))

for input_path, target_path in zip(input_img_paths[:10], target_img_paths[:10]):
    print(input_path, "|", target_path)
```

```
Number of samples: 7390
images/Abyssinian_1.jpg | annotations/trimaps/Abyssinian_1.png
images/Abyssinian_10.jpg | annotations/trimaps/Abyssinian_10.png
images/Abyssinian_100.jpg | annotations/trimaps/Abyssinian_100.png
images/Abyssinian_101.jpg | annotations/trimaps/Abyssinian_101.png
images/Abyssinian_102.jpg | annotations/trimaps/Abyssinian_102.png
images/Abyssinian_103.jpg | annotations/trimaps/Abyssinian_103.png
images/Abyssinian_104.jpg | annotations/trimaps/Abyssinian_104.png
images/Abyssinian_105.jpg | annotations/trimaps/Abyssinian_105.png
images/Abyssinian_106.jpg | annotations/trimaps/Abyssinian_106.png
images/Abyssinian_107.jpg | annotations/trimaps/Abyssinian_107.png
```

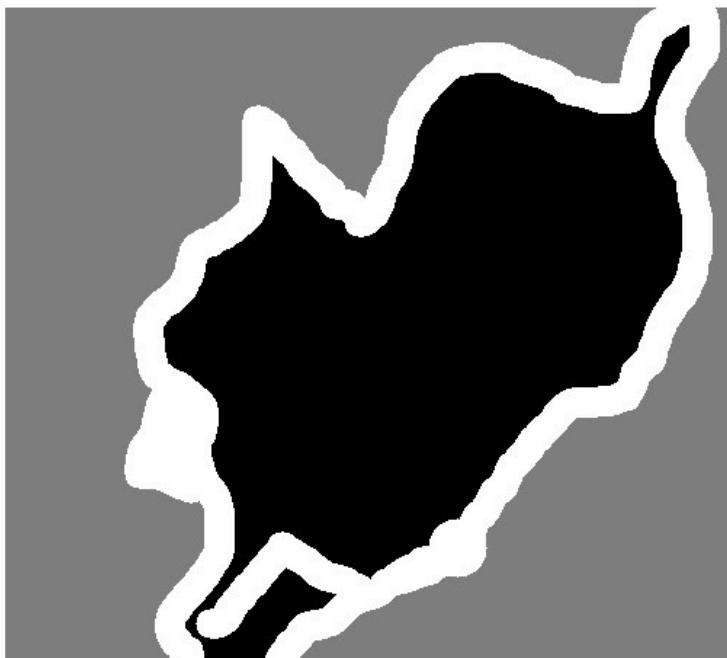
Amb el codi següent exploram les màscares de les imatges del conjunt de dades.

```
from IPython.display import Image, display
from keras.utils import load_img
from PIL import ImageOps

# Display input image #7
display(Image(filename=input_img_paths[9]))

# Display auto-contrast version of corresponding target (per-pixel categories)
img = ImageOps.autocontrast(load_img(target_img_paths[9]))
display(img)
```

Vegem una imatge i la seva màscara.



Seguidament, preparam el conjunt de dades per carregar-lo i vectoritzar-lo.

```

import keras
import numpy as np
from tensorflow import data as tf_data
from tensorflow import image as tf_image
from tensorflow import io as tf_io

def get_dataset(
    batch_size,
    img_size,
    input_img_paths,
    target_img_paths,
    max_dataset_len=None,
):
    """Returns a TF Dataset."""

    def load_img_masks(input_img_path, target_img_path):
        input_img = tf_io.read_file(input_img_path)
        input_img = tf_io.decode_png(input_img, channels=3)
        input_img = tf_image.resize(input_img, img_size)
        input_img = tf_image.convert_image_dtype(input_img, "float32")

        target_img = tf_io.read_file(target_img_path)
        target_img = tf_io.decode_png(target_img, channels=1)
        target_img = tf_image.resize(target_img, img_size, method="nearest")
        target_img = tf_image.convert_image_dtype(target_img, "uint8")

        # Ground truth labels are 1, 2, 3. Subtract one to make them 0, 1, 2:
        target_img -= 1
        return input_img, target_img

    # For faster debugging, limit the size of data
    if max_dataset_len:
        input_img_paths = input_img_paths[:max_dataset_len]
        target_img_paths = target_img_paths[:max_dataset_len]
    dataset = tf_data.Dataset.from_tensor_slices((input_img_paths, target_img_paths))
    dataset = dataset.map(load_img_masks, num_parallel_calls=tf_data.AUTOTUNE)
    return dataset.batch(batch_size)

```

Definim el model de la xarxa neuronal, que serà del tipus **U-Net Xception**.

```
from keras import layers

def get_model(img_size, num_classes):
    inputs = keras.Input(shape=img_size + (3,))

    ### [First half of the network: downsampling inputs] ###

    # Entry block
    x = layers.Conv2D(32, 3, strides=2, padding="same")(inputs)
    x = layers.BatchNormalization()(x)
    x = layers.Activation("relu")(x)

    previous_block_activation = x # Set aside residual

    # Blocks 1, 2, 3 are identical apart from the feature depth.
    for filters in [64, 128, 256]:
        x = layers.Activation("relu")(x)
        x = layers.SeparableConv2D(filters, 3, padding="same")(x)
        x = layers.BatchNormalization()(x)

        x = layers.Activation("relu")(x)
        x = layers.SeparableConv2D(filters, 3, padding="same")(x)
        x = layers.BatchNormalization()(x)

        x = layers.MaxPooling2D(3, strides=2, padding="same")(x)

        # Project residual
        residual = layers.Conv2D(filters, 1, strides=2, padding="same")(
            previous_block_activation
        )
        x = layers.add([x, residual]) # Add back residual
        previous_block_activation = x # Set aside next residual

    ### [Second half of the network: upsampling inputs] ###
```

```
### [Second half of the network: upsampling inputs] ###

for filters in [256, 128, 64, 32]:
    x = layers.Activation("relu")(x)
    x = layers.Conv2DTranspose(filters, 3, padding="same")(x)
    x = layers.BatchNormalization()(x)

    x = layers.Activation("relu")(x)
    x = layers.Conv2DTranspose(filters, 3, padding="same")(x)
    x = layers.BatchNormalization()(x)

    x = layers.UpSampling2D(2)(x)

# Project residual
residual = layers.UpSampling2D(2)(previous_block_activation)
residual = layers.Conv2D(filters, 1, padding="same")(residual)
x = layers.add([x, residual]) # Add back residual
previous_block_activation = x # Set aside next residual

# Add a per-pixel classification layer
outputs = layers.Conv2D(num_classes, 3, activation="softmax", padding="same")(x)

# Define the model
model = keras.Model(inputs, outputs)
return model

# Build model
model = get_model(img_size, num_classes)
model.summary()
```

Separam una fracció de les dades com a conjunt de validació, per contrastar el funcionament del model.

```
import random

# Split our img paths into a training and a validation set
val_samples = 1000
random.Random(1337).shuffle(input_img_paths)
random.Random(1337).shuffle(target_img_paths)
train_input_img_paths = input_img_paths[:-val_samples]
train_target_img_paths = target_img_paths[:-val_samples]
val_input_img_paths = input_img_paths[-val_samples:]
val_target_img_paths = target_img_paths[-val_samples:]

# Instantiate dataset for each split
# Limit input files in `max_dataset_len` for faster epoch training time.
# Remove the `max_dataset_len` arg when running with full dataset.
train_dataset = get_dataset(
    batch_size,
    img_size,
    train_input_img_paths,
    train_target_img_paths,
    max_dataset_len=1000,
)
valid_dataset = get_dataset(
    batch_size, img_size, val_input_img_paths, val_target_img_paths
)
```

Ara estam en situació d'entrenar el model.

```

# Configure the model for training.
# We use the "sparse" version of categorical_crossentropy
# because our target data is integers.
model.compile(
    optimizer=keras.optimizers.Adam(1e-4), loss="sparse_categorical_crossentropy"
)

callbacks = [
    keras.callbacks.ModelCheckpoint("oxford_segmentation.keras", save_best_only=True)
]

# Train the model, doing validation at the end of each epoch.
epochs = 50
model.fit(
    train_dataset,
    epochs=epochs,
    validation_data=valid_dataset,
    callbacks=callbacks,
    verbose=2,
)

```

I finalment vegem com es comporta el model amb una imatge del conjunt de validació, que no ha format part del conjunt d'entrenament.

```

# Generate predictions for all images in the validation set

val_dataset = get_dataset(
    batch_size, img_size, val_input_img_paths, val_target_img_paths
)
val_preds = model.predict(val_dataset)

def display_mask(i):
    """Quick utility to display a model's prediction."""
    mask = np.argmax(val_preds[i], axis=-1)
    mask = np.expand_dims(mask, axis=-1)
    img = ImageOps.autocontrast(keras.utils.array_to_img(mask))
    display(img)

# Display results for validation image #10
i = 10

# Display input image
display(Image(filename=val_input_img_paths[i]))

# Display ground-truth target mask
img = ImageOps.autocontrast(load_img(val_target_img_paths[i]))
display(img)

# Display mask predicted by our model
display_mask(i) # Note that the model only sees inputs at 150x150.

```

A continuació tenim la imatge, la seva segmentació de referència i la segmentació obtinguda pel model.



S'obté un resultat sorollós, amb la separació blanca entre imatge i fons discontínua.

7.3. Detecció d'objectes

En aquesta aplicació es tracta de determinar la posició dels objectes d'interès a través d'uns punts de referència

https://keras.io/examples/vision/keypoint_detection/

Començam carregant les llibreries i dades.

```
!pip install -q -U imgaug
```

```
!wget -q http://vision.stanford.edu/aditya86/ImageNetDogs/images.tar
```

```
!tar xf images.tar
!unzip -qq ~/stanfordextra_v12.zip
```

```
from keras import layers
import keras

from imgaug.augmentables.kps import KeypointsOnImage
from imgaug.augmentables.kps import Keypoint
import imgaug.augmenters as iaa

from PIL import Image
from sklearn.model_selection import train_test_split
from matplotlib import pyplot as plt
import pandas as pd
import numpy as np
import json
import os
```

Definim els paràmetres del sistema

```
IMG_SIZE = 224
BATCH_SIZE = 64
EPOCHS = 5
NUM_KEYPOINTS = 24 * 2 # 24 pairs each having x and y coordinates
```

I mostrem les coordenades dels punts de referència per a una imatge concreta.

```
IMG_DIR = "Images"
JSON = "StanfordExtra_V12/StanfordExtra_v12.json"
KEYPOINT_DEF = (
    "https://github.com/benjebob/StanfordExtra/raw/master/keypoint_definitions.csv"
)

# Load the ground-truth annotations.
with open(JSON) as infile:
    json_data = json.load(infile)

# Set up a dictionary, mapping all the ground-truth information
# with respect to the path of the image.
json_dict = {i["img_path"]: i for i in json_data}
```

```
'n02085782-Japanese_spaniel/n02085782_2886.jpg':
{'img_bbox': [205, 20, 116, 201],
'img_height': 272,
'img_path': 'n02085782-Japanese_spaniel/n02085782_2886.jpg',
'img_width': 350,
'is_multiple_dogs': False,
'joints': [[108.66666666666667, 252.0, 1],
[147.66666666666666, 229.0, 1],
[163.5, 208.5, 1],
[0, 0, 0],
[0, 0, 0],
[0, 0, 0],
[54.0, 244.0, 1],
[77.333333333333, 225.333333333334, 1],
[79.0, 196.5, 1],
[0, 0, 0],
[0, 0, 0],
[0, 0, 0],
[0, 0, 0],
[0, 0, 0],
[150.66666666666666, 86.66666666666667, 1],
[88.66666666666667, 73.0, 1],
[116.0, 106.333333333333, 1],
[109.0, 123.333333333333, 1],
[0, 0, 0],
[0, 0, 0],
[0, 0, 0],
[0, 0, 0],
[0, 0, 0],
[0, 0, 0]],
'seg': ...}
```

```
# Load the metdata definition file and preview it.
keypoint_def = pd.read_csv(KEYPOINT_DEF)
keypoint_def.head()

# Extract the colours and labels.
colours = keypoint_def["Hex colour"].values.tolist()
colours = ["#" + colour for colour in colours]
labels = keypoint_def["Name"].values.tolist()

# Utility for reading an image and for getting its annotations.
def get_dog(name):
    data = json_dict[name]
    img_data = plt.imread(os.path.join(IMG_DIR, data["img_path"]))
    # If the image is RGBA convert it to RGB.
    if img_data.shape[-1] == 4:
        img_data = img_data.astype(np.uint8)
        img_data = Image.fromarray(img_data)
        img_data = np.array(img_data.convert("RGB"))
    data["img_data"] = img_data

    return data
```

Ja podem visualitzar les dades d'entrenament

```
# Parts of this code come from here:
# https://github.com/benjebob/StanfordExtra/blob/master/demo.ipynb
def visualize_keypoints(images, keypoints):
    fig, axes = plt.subplots(nrows=len(images), ncols=2, figsize=(16, 12))
    [ax.axis("off") for ax in np.ravel(axes)]

    for (ax_orig, ax_all), image, current_keypoint in zip(axes, images, keypoints):
        ax_orig.imshow(image)
        ax_all.imshow(image)

        # If the keypoints were formed by `imgaug` then the coordinates need
        # to be iterated differently.
        if isinstance(current_keypoint, KeypointsOnImage):
            for idx, kp in enumerate(current_keypoint.keypoints):
                ax_all.scatter(
                    [kp.x],
                    [kp.y],
                    c=colours[idx],
                    marker="x",
                    s=50,
                    linewidths=5,
                )
        else:
            current_keypoint = np.array(current_keypoint)
            # Since the last entry is the visibility flag, we discard it.
            current_keypoint = current_keypoint[:, :2]
            for idx, (x, y) in enumerate(current_keypoint):
                ax_all.scatter([x], [y], c=colours[idx], marker="x", s=50, linewidths=5)

    plt.tight_layout(pad=2.0)
    plt.show()
```

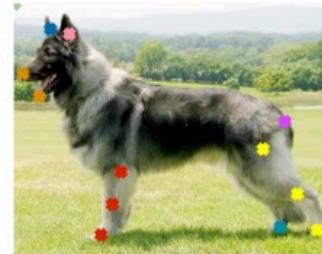
```
# Select four samples randomly for visualization.
samples = list(json_dict.keys())
num_samples = 4
selected_samples = np.random.choice(samples, num_samples, replace=False)

images, keypoints = [], []

for sample in selected_samples:
    data = get_dog(sample)
    image = data["img_data"]
    keypoint = data["joints"]

    images.append(image)
    keypoints.append(keypoint)

visualize_keypoints(images, keypoints)
```



A continuació preparació el generador de dades, que permetrà obtenir més variants d'imatges que completen l'entrenament.

```
class KeyPointsDataset(keras.utils.PyDataset):
    def __init__(self, image_keys, aug, batch_size=BATCH_SIZE, train=True, **kwargs):
        super().__init__(**kwargs)
        self.image_keys = image_keys
        self.aug = aug
        self.batch_size = batch_size
        self.train = train
        self.on_epoch_end()

    def __len__(self):
        return len(self.image_keys) // self.batch_size

    def on_epoch_end(self):
        self.indexes = np.arange(len(self.image_keys))
        if self.train:
            np.random.shuffle(self.indexes)

    def __getitem__(self, index):
        indexes = self.indexes[index * self.batch_size : (index + 1) * self.batch_size]
        image_keys_temp = [self.image_keys[k] for k in indexes]
        (images, keypoints) = self._data_generation(image_keys_temp)

        return (images, keypoints)
```

```

def __data_generation(self, image_keys_temp):
    batch_images = np.empty((self.batch_size, IMG_SIZE, IMG_SIZE, 3), dtype="int")
    batch_keypoints = np.empty(
        (self.batch_size, 1, 1, NUM_KEYPOINTS), dtype="float32"
    )

    for i, key in enumerate(image_keys_temp):
        data = get_dog(key)
        current_keypoint = np.array(data["joints"])[ :, :2]
        kps = []

        # To apply our data augmentation pipeline, we first need to
        # form Keypoint objects with the original coordinates.
        for j in range(0, len(current_keypoint)):
            kps.append(Keypoint(x=current_keypoint[j][0], y=current_keypoint[j][1]))

        # We then project the original image and its keypoint coordinates.
        current_image = data["img_data"]
        kps_obj = KeypointsOnImage(kps, shape=current_image.shape)

        # Apply the augmentation pipeline.
        (new_image, new_kps_obj) = self.aug(image=current_image, keypoints=kps_obj)
        batch_images[i,] = new_image

        # Parse the coordinates from the new keypoint object.
        kp_temp = []
        for keypoint in new_kps_obj:
            kp_temp.append(np.nan_to_num(keypoint.x))
            kp_temp.append(np.nan_to_num(keypoint.y))

        # More on why this reshaping later.
        batch_keypoints[i,] = np.array(kp_temp).reshape(1, 1, 24 * 2)

        # Scale the coordinates to [0, 1] range.
        batch_keypoints = batch_keypoints / IMG_SIZE

    return (batch_images, batch_keypoints)

```

```

train_aug = iaa.Sequential(
    [
        iaa.Resize(IMG_SIZE, interpolation="linear"),
        iaa.Fliplr(0.3),
        # `Sometimes()` applies a function randomly to the inputs with
        # a given probability (0.3, in this case).
        iaa.Sometimes(0.3, iaa.Affine(rotate=10, scale=(0.5, 0.7))),
    ]
)

test_aug = iaa.Sequential([iaa.Resize(IMG_SIZE, interpolation="linear")])

```

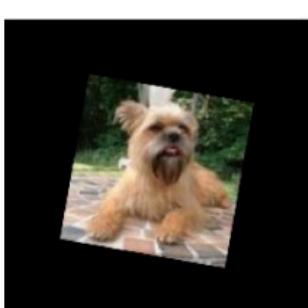
```

np.random.shuffle(samples)
train_keys, validation_keys = (
    samples[int(len(samples) * 0.15) :],
    samples[: int(len(samples) * 0.15)],
)

```

```
train_dataset = KeyPointsDataset(  
    train_keys, train_aug, workers=2, use_multiprocessing=True  
)  
validation_dataset = KeyPointsDataset(  
    validation_keys, test_aug, train=False, workers=2, use_multiprocessing=True  
)  
  
print(f"Total batches in training set: {len(train_dataset)}")  
print(f"Total batches in validation set: {len(validation_dataset)}")  
  
sample_images, sample_keypoints = next(iter(train_dataset))  
assert sample_keypoints.max() == 1.0  
assert sample_keypoints.min() == 0.0  
  
sample_keypoints = sample_keypoints[:4].reshape(-1, 24, 2) * IMG_SIZE  
visualize_keypoints(sample_images[:4], sample_keypoints)
```

```
Total batches in training set: 166  
Total batches in validation set: 29
```



Construïm el model.

```
def get_model():
    # Load the pre-trained weights of MobileNetV2 and freeze the weights
    backbone = keras.applications.MobileNetV2(
        weights="imagenet",
        include_top=False,
        input_shape=(IMG_SIZE, IMG_SIZE, 3),
    )
    backbone.trainable = False

    inputs = layers.Input((IMG_SIZE, IMG_SIZE, 3))
    x = keras.applications.mobilenet_v2.preprocess_input(inputs)
    x = backbone(x)
    x = layers.Dropout(0.3)(x)
    x = layers.SeparableConv2D(
        NUM_KEYPOINTS, kernel_size=5, strides=1, activation="relu"
    )(x)
    outputs = layers.SeparableConv2D(
        NUM_KEYPOINTS, kernel_size=3, strides=1, activation="sigmoid"
    )(x)

    return keras.Model(inputs, outputs, name="keypoint_detector")
```

Compilam i entrenam el model

```
model = get_model()
model.compile(loss="mse", optimizer=keras.optimizers.Adam(1e-4))
model.fit(train_dataset, validation_data=validation_dataset, epochs=EPOCHS)
```

```
Epoch 1/5
166/166 [=====] 84s 415ms/step - loss: 0.1110 - val_loss: 0.0959
Epoch 2/5
166/166 [=====] 79s 472ms/step - loss: 0.0874 - val_loss: 0.0802
Epoch 3/5
166/166 [=====] 78s 463ms/step - loss: 0.0789 - val_loss: 0.0765
Epoch 4/5
166/166 [=====] 78s 467ms/step - loss: 0.0769 - val_loss: 0.0731
Epoch 5/5
166/166 [=====] 77s 464ms/step - loss: 0.0753 - val_loss: 0.0712

<keras.src.callbacks.history.History at 0x7fb5c4299ae0>
```

```
sample_val_images, sample_val_keypoints = next(iter(validation_dataset))
sample_val_images = sample_val_images[:4]
sample_val_keypoints = sample_val_keypoints[:4].reshape(-1, 24, 2) * IMG_SIZE
predictions = model.predict(sample_val_images).reshape(-1, 24, 2) * IMG_SIZE

# Ground-truth
visualize_keypoints(sample_val_images, sample_val_keypoints)

# Predictions
visualize_keypoints(sample_val_images, predictions)
```

I observam els resultats, primer els punts de referència i després els obtinguts automàticament. S'hi aprecien diferències importants, cosa que fa pensar que el sistema és bastant millorable.



