6. Aprenentatge profund

6.1. Introducció

En els darrers anys, la intel·ligència artificial ha guanyat un gran protagonisme en els mitjans d'informació generalistes gràcies a una sèrie d'impressionants assoliments en diferents àrees. La major part d'aquests assoliments es deuen als avanços en els mètodes denominats d'aprenentatge **profund.*** En aquest capítol veurem en què consisteixen els mètodes d'aprenentatge profund més importants, quins tipus hi ha i com es poden programar i aplicar.

*** En anglès, *deep learning* (DL).**

6.1.1. Assoliments recents

Els mètodes d'aprenentatge profund han protagonitzat nombroses notícies en els darrers anys a causa de diferents assoliments en molts tipus d'aplicació de la intel·ligència artificial (IA). Per citar alguns exemples destacables:

- En visió artificial han convertit en una tasca habitual el reconeixement de cares, caràcters, tipus d'objectes en una foto, vehicles i vianants, etc. Tot això amb una alta precisió. També han aconseguit superar els humans en tasques tan complexes com distingir entre diverses espècies d'aus molt semblants entre si.

- En medicina s'ha aplicat en diferents tasques, com buscar proteïnes que encaixin amb una determinada substància, identificar melanomes a partir de fotografies, detectar diferents patologies en imatges de ressonància magnètica, etc.

- En el sector de l'automoció s'estan desenvolupant sistemes de conducció automàtica que són capaços de conduir un cotxe per una autopista real, utilitzant informació senzilla (càmeres). Segons els desenvolupadors, s'han evitat nombrosos accidents gràcies a aquesta tecnologia. També s'estan desenvolupant robots capaços de dur a terme tasques complexes, per exemple, reaccionar encara que es trobin peces desordenades.

- En processament del llenguatge s'ha aconseguit una gran qualitat en reconeixement de veu, com és el cas dels famosos assistents per veu que inclouen els mòbils i altres dispositius. També ha millorat notablement la traducció automàtica, que actualment comença a ser útil per a diferents propòsits.

- En diferents ciències s'estan aplicant per accelerar el processament de dades, com per exemple per calcular la distorsió que un objecte massiu produeix en la llum que arriba d'objectes astronòmics llunyans.

- Un sistema d'aprenentatge profund (AlphaGo) va vèncer el campió del món de Go, també conegut com a dames xineses, un joc molt més complex que els escacs. També s'ha desenvolupat el sistema DQN, un sistema que és capaç d'aprendre a jugar a jocs de consola (concretament una antiga videoconsola Atari) per si mateix, sense utilitzar cap coneixement previ sobre el joc, simplement a partir del que veu en pantalla.

6.1.2. Causes

Els sistemes d'aprenentatge profund són una evolució de les **xarxes neuronals**, que són sistemes inspirats en les neurones del cervell per aconseguir respostes «intel·ligents». Les xarxes neuronals van començar a desenvolupar-se en la dècada de 1950, però no han aconseguit un interès tan generalitzat fins fa pocs anys (a partir del 2010). Per què ara sí que han tingut un gran èxit en tantes tasques de IA? Bàsicament es proposen tres raons per a aquest ressorgiment de les xarxes neuronals:

- La disponibilitat de grans conjunts de dades (el que es denomina *big data*). Els sistemes d'aprenentatge profund necessiten milers o milions de dades d'exemple per aconseguir resoldre les seves tasques amb un grau de precisió tan alt.

- Diferents millores teòriques que permeten entrenar xarxes neuronals més complexes (concretament, més *profundes*, com es veurà més endavant). Anteriorment els mètodes d'entrenament fracassaven amb xarxes complexes, la qual cosa impedia aconseguir l'aprenentatge per nivells de complexitat, característic dels sistemes d'aprenentatge profund.

- Les millores en el maquinari que permeten entrenar sistemes amb milions de dades d'exemple. Especialment destaquen els processadors gràfics,* que en principi es van desenvolupar per visualitzar videojocs, però que la comunitat de IA ha aprofitat per entrenar els seus mètodes. El gran avantatge de fer servir GPU davant de CPU es troba en el fet que les GPU tenen nombroses unitats de càlcul en paral·lel, la qual cosa accelera les operacions sobre vectors en diversos ordres de magnitud.

* En inglés, *Graphics Processing Units* (GPU).

6.1.3. Arquitectures

Un avantatge addicional dels sistemes d'aprenentatge profund és el seu disseny modular, que permet combinar diferents components de la manera més adequada a cada tasca. Aquesta modularitat dona lloc a diferents arquitectures. A continuació en destaquem les més importants:

- Xarxes **de propagació cap endavant** (*feed-forward networks*), també denominades **perceptró multicapa**: són sistemes generalment utilitzats per a tasques de classificació i de regressió, en les quals el flux d'informació va des de l'entrada fins a la sortida, sense bucles ni retrocessos.

- Xarxes **convolucionals**, especialitzades en processament d'imatge i altres dades estructurades, que inclouen elements especialitzats a capturar patrons repetits (per exemple, línies dins d'una imatge).

- Xarxes **recurrents**: en aquest tipus de xarxes es produeix una mena de recurrència o tornada enrere del flux de dades, la qual cosa els permet tenir memòria. S'utilitzen en processament de seqüències: senyals, text, tasques que requereixen múltiples passos.

- **Autocodificadors**: sistemes no supervisats especialitzats a generar representacions simplificades de les dades d'entrenament. El seu objectiu és similar al dels mètodes de reducció de la dimensionalitat clàssics que hem vist a l'assignatura, com PCA.

- Sistemes d'**aprenentatge per reforç** (*reinforcement learning*): estrictament no es tracta d'un tipus d'arquitectura, sinó d'un nou tipus de tasca en el qual el sistema s'entrena per interacció amb el seu entorn. Els sistemes d'aprenentatge profund s'utilitzen sovint en aquestes tasques.

- Sistemes **generatius**, en els quals hi ha dos sistemes oposats i es fa que un d'ells aprengui a «inventar-se» noves dades, tan semblants a les dades d'entrenament que semblin reals.

En apartats posteriors s'estudiaran amb més detalli aquestes arquitectures, la seva aplicació i la seva programació.

6.1.4. Biblioteques

L'ús de mètodes d'aprenentatge profund requereix la utilització de diferents mètodes matemàtics relativament complexos i que, a més, han d'executar-se amb la màxima velocitat i aprofitant el maquinari disponible perquè el seu ús sigui viable. Això fa que l'opció més habitual per utilitzar aquests mètodes passi per l'ús d'una o més biblioteques de programació que resolguin bona part dels problemes plantejats i permetin centrar-se en el desenvolupament del sistema que l'aplicació necessiti.

Avui dia existeixen nombroses biblioteques d'aprenentatge profund, de les quals destaquen les següents (el llenguatge de programació és Python, tret que s'indiqui el contrari):

- Theano: una de les primeres biblioteques, desenvolupada pels pioners de l'aprenentatge profund. Es tracta d'una biblioteca de baix nivell, en el sentit que ofereix diferents operacions matemàtiques però que requereix que l'usuari programi bastant per tenir un sistema en marxa.

- Tensorflow: el competidor de Theano, desenvolupat per Google. També és una biblioteca de baix nivell.

- Lasagne: biblioteca d'alt nivell que, utilitzant Theano, ofereix als usuaris instruccions senzilles per crear sistemes d'aprenentatge profund.

- Keras: equivalent a Lasagne, amb l'avantatge que pot operar sobre Theano i sobre Tensorflow.

- Caffe: biblioteca especialitzada en sistemes de visió artificial.

- nolearn: biblioteca de molt alt nivell, molt fàcil de fer servir però que, a canvi, permet pocs canvis.

- Torch: biblioteca en llenguatge Lua, utilitzada per Facebook i Twitter, entre altres.

- dl4j: biblioteca en Java, més orientada a l'entorn empresarial.

Com es veu, el llenguatge principal en aquesta àrea és Python. Els exemples que es presentaran als apartats següents utilitzen **Keras** sobre **Tensorflow**, ja que és potser la combinació més popular actualment, resulta senzilla però alhora permet dissenyar sistemes especialitzats.

Keras i Tensorflow

També es pot fer servir Keras sobre Theano, però es recomana Tensorflow perquè, en general, resulta més senzill de configurar i més senzill de configurar-lo per fer servir la GPU. Com que són biblioteques en actualització constant, es recomana consultar les instruccions d'instal·lació a las pàgines dels projectes: https://www.tensorflow.org/ https://keras.io/.

6.2. Xarxes neuronals

El cervell està format per diferents tipus de cèl·lules. Les **neurones** són les cèl·lules que, establint connexions entre elles i enviant senyals per aquestes connexions, donen lloc al comportament emergent que denominem intel·ligència.

L'objectiu de les **xarxes neuronals** és crear un anàleg computacional a les neurones biològiques per intentar crear intel·ligència en un ordinador. En aquest sentit, es defineixen uns elements anomenats neurones o simplement **unitats** amb connexions amb altres unitats per les quals es propaguen senyals. No obstant això, l'analogia no va gaire més enllà i l'experiència a l'àrea ha demostrat que el que funciona en biologia no té per què funcionar en informàtica.

En aquest apartat veurem què són i com funcionen les xarxes neuronals, que són la base dels sistemes d'aprenentatge profund.

6.2.1. Components d'una xarxa neuronal

Una xarxa neuronal bàsica es compon dels elements següents:

- Una capa d'unitats d'**entrada**, que reben les variables de l'exterior disponibles per tractar el problema: els píxels d'una imatge, la posició d'un objecte, els valors de certs productes, etc.
- Una capa d'unitats de **sortida**, composta per una o més unitats que produeixen una sortida a l'exterior, que és el «resultat» de la xarxa: la classe d'imatge, el valor de tensió elèctrica necessari per accionar un motor, si s'ha de comprar o vendre un producte, etc.
- Una capa d'unitats **ocultes**, que reben connexions de la capa d'entrada i es connecten a les unitats de sortida.
- Finalment, el conjunt de **connexions** entre capes. En general les connexions entre unitats són unidireccionals.

Com es veu, les unitats s'organitzen en capes. En el tipus de xarxa neuronal més senzilla, la xarxa neuronal prealimentada o perceptró, les connexions sempre van de les unitats de la capa d'entrada a les de la capa oculta i d'aquí a la capa de sortida; les connexions no tornen enrere.

Així, el funcionament del perceptró consisteix bàsicament en **la propagació** cap endavant dels senyals d'entrada, lògicament modificats a mesura que travessen capes.

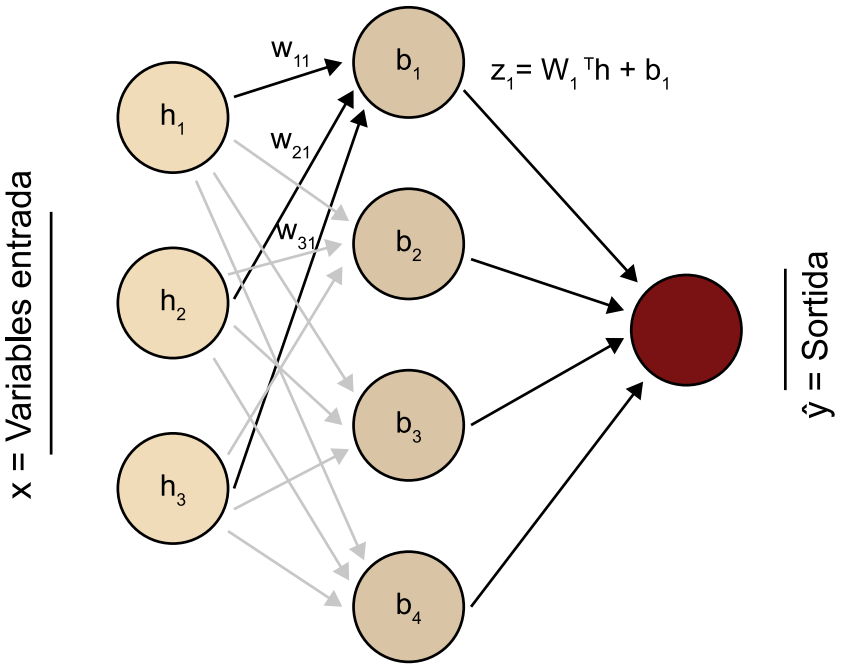
Quan totes les unitats d'una capa estan connectades a totes les unitats de la capa següent es diu que aquesta segona capa està **completament connectada**. Al perceptró les capes són així, però en altres arquitectures no té per què passar això.

De manera intuïtiva, i abans d'entrar en més detall, el funcionament del perceptró és el següent:

1) Les unitats d'entrada reben valors de l'exterior, i s'activaran —és a dir, produiran un determinat valor a la sortida— o no en funció de l'entrada rebuda.

2) Les sortides de la capa d'entrada són, al seu torn, les entrades de la capa oculta, de manera que aquestes unitats reben un conjunt d'entrades enfront de les quals reaccionaran. Per fer-ho, cada unitat de la capa oculta té un **vector de pesos**, un valor per cada connexió entrant, que combina amb els senyals corresponents i, com a resultat, produeixen l'activació o no de la sortida de cada unitat oculta.

Figura 61. Xarxa neuronal bàsica



3) Finalment, les unitats de la capa de sortida també reben els senyals de la capa oculta, realitzen una operació amb aquests senyals i els seus propis vectors de pesos i, com a resultat, calculen la seva pròpia sortida, que serà el resultat de la xarxa.

6.2.2. Funcions d'activació

La manera que té una unitat de combinar les seves entrades X amb el seu vector de pesos W per calcular la seva sortida z ve donada generalment per l'expressió:

$$z = W^T X + b \tag{72}$$

en què b és un valor escalar, denominat biaix (*bias*), necessari per garantir un valor de base. El valor $z \in \mathbb{R}$ s'utilitza al seu torn com a entrada per a la **funció d'activació**, que és la que decideix quina és la sortida final.

La funció d'activació més utilitzada en les unitats d'entrada i ocultes és la crida **ReLU** (de *Rectified Linear Unit*), que és tan senzilla com:

$$g(z) = \text{màx}(0,z) \tag{73}$$

És a dir, la sortida és 0 si l'entrada és negativa, i és igual a z si aquesta és positiva.

Hi ha variacions de la funció ReLU que milloren el seu comportament en algunes situacions, com les funcions ReLU amb pèrdua, en què la funció té un petit gradient negatiu quan $z < 0$, la qual cosa facilita l'entrenament de la xarxa. També resulta interessant la família de les **ELU** (*exponential linear units*), definides d'aquesta manera:

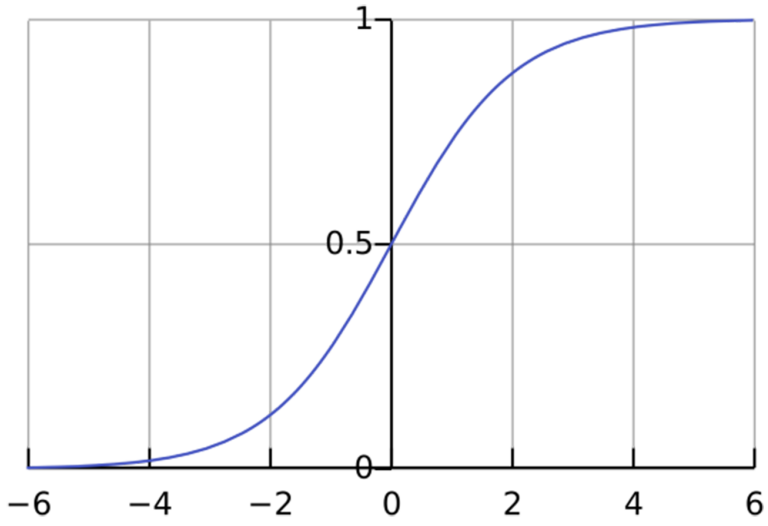
$$f(x) = \begin{cases} x & \text{si} \quad x \geq 0 \\ a(e^x - 1) & \text{si} \quad x < 0 \end{cases} \tag{74}$$

Experiments recents mostren que les ELU donen millors resultats, ja que el seu entrenament és més eficient gràcies al fet que el seu valor mitjà està més proper a zero.

En el cas d'unitats de sortida, se solen utilitzar altres funcions d'activació, que principalment depenen del tipus de sortida desitjada:

- Si es tracta d'un valor **real**, típic en problemes de regressió, la funció d'activació és lineal, és a dir $g(z) = z$.
- En el cas de valors **binaris**, per a classificació en dues classes o detecció d'anomalies, se sol utilitzar la funció logística sigmoide, mostrada a continuació.
- En valors **multiclasse**, com en un classificador, s'utilitza la funció **softmax**: hi ha una unitat de sortida per a cada classe, de manera que la sortida activada tindrà un valor proper a 1 i les altres tindran valors propers a 0.

Figura 62. Funció logística sigmoide



6.2.3. Entrenament d'una xarxa neuronal

Com s'ha explicat, en una xarxa neuronal de tipus perceptró els senyals es propaguen cap endavant, és a dir, de les entrades cap a les sortides, sense tornar enrere. També s'ha exposat que cada unitat té un vector de pesos W, amb un valor per a cadascuna de les seves entrades, així com un biaix b. La pregunta que sorgeix és: d'on sorgeixen W i b? Al cap i a la fi aquests valors són els que controlen el comportament de la xarxa. Doncs bé, aquests valors no apareixen sols ni es posen a mà, sinó que és la mateixa xarxa la que ha d'aprendre els valors òptims per aconseguir realitzar la tasca encomanada de la millor manera possible.

Per tant, en treballar amb xarxes neuronals es distingeixen dues fases: una primera fase d'**entrenament** en la qual la xarxa rep dades de les quals ha d'aprendre i en la qual va ajustant els pesos i biaixos, i una altra fase d'**execució**, en la qual s'utilitza la xarxa per dur a terme la tasca tal com l'ha après.

Els perceptrons són sistemes d'aprenentatge **supervisat**, per la qual cosa l'entrenament d'un perceptró requereix un conjunt de dades etiquetades. D'altra banda, els pesos i biaixos s'inicialitzen a valors aleatoris petits (al voltant de 0,1). A continuació, es van injectant els exemples d'entrenament a la xarxa i s'analitza la diferència entre la sortida obtinguda i la sortida esperada (segons l'etiqueta associada a cada exemple).

Aquesta diferència entre la sortida obtinguda i la sortida esperada s'expressa mitjançant una **funció de cost** C. En xarxes amb una sola sortida es pot utilitzar com a C la funció d'error quadràtic, mentre que en problemes amb diverses sortides solen utilitzar-se l'entropia creuada o la divergència de Kullback-Leibler.

L'objectiu de l'entrenament és reduir els valors de C, és a dir, la diferència entre la sortida obtinguda i l'esperada. Per això cal ajustar gradualment els pesos i biaixos de cada capa utilitzant el mètode de **descens de gradients**, vist al subapartat 5.3., concretament calculant el gradient de la funció de cost respecte als pesos i biaixos de la unitat de sortida:

$$\frac{\partial C}{\partial \hat{W}} \tag{75}$$

en què $\hat{W} = \{W,b\}$, és a dir, s'integren pesos i biaix en un únic vector per facilitar les operacions.

Així doncs, en determinar els $\hat{W}$ òptims per reduir la funció de cost C s'està ajustant la capa de sortida; no obstant això, les capes anteriors no han rebut

cap ajust. Aquí entra en joc el procés denominat **propagació cap enrere** (*back-propagation*): una vegada ajustada la capa de sortida, es procedeix a ajustar la capa oculta mitjançant el mateix procediment, i així es van ajustat les capes des de la sortida cap a l'entrada, per aquest motiu es fa «cap enrere».

D'aquesta manera s'acaben ajustant tots els paràmetres de la xarxa, fent que el conjunt de paràmetres produeixi el menor error possible amb les dades d'entrenament. Aquest **error d'entrenament** pot ser diferent de zero en funció de les dades i la complexitat del model; un model més complex serà capaç d'aprendre millor les dades d'entrenament, però com es veurà més endavant no és convenient portar aquesta idea a l'extrem per aconseguir un error d'entrenament igual a zero.

Una altra qüestió important és la manera en què es produeix l'optimització per descens de gradients, tenint en compte com s'utilitzen els exemples d'entrenament. Hi ha tres maneres de procedir:

- **Descens de gradients per lots** (*batches*). Consisteix a utilitzar tots els exemples d'entrenament alhora. Tot i que és el mètode que dona més bons resultats, computacionalment resulta molt costós, no només en temps sinó també en memòria, que és un factor crític si s'usen GPU.

- **Descens de gradients estocàstic**. Utilitza els exemples d'entrenament d'un en un. Tot i que és un mètode ràpid, provoca una gran variància en l'entrenament, ja que les característiques específiques de cada exemple condicionen enormement els pesos triats.

- **Descens de gradients per minilots** (*mini-batches*). És una solució intermèdia, en la qual els exemples s'agrupen en blocs i s'executa el descens de gradients sobre cada bloc. Així es redueix la variància però amb un cost computacional moderat. Convé triar una grandària de bloc per tal que els exemples càpiguen en la memòria de treball. És el mètode més utilitzat.

Un avantatge addicional de l'aproximació per minilots és que permet múltiples iteracions d'entrenament sobre el mateix conjunt de dades, bé prenent els mateixos lots, bé particionant les dades d'entrenament d'una manera diferent cada vegada. A aquestes iteracions d'entrenament se'ls dona el nom d'**èpoques** (*epochs*).

6.2.4. Problemes d'aprenentatge

A les xarxes neuronals, igual que en altres mètodes d'aprenentatge automàtic, poden aparèixer diferents problemes d'aprenentatge que és fonamental conèixer, identificar i saber resoldre adequadament, ja que en alguns casos les solucions són oposades i cal evitar canviar paràmetres a l'atzar fins que el sistema millori.

En primer lloc, recordem el protocol d'entrenament d'un sistema: per entrenar correctament un sistema de manera que posteriorment es pugui aplicar a noves dades amb èxit, és necessari dividir les dades en tres blocs:

- **Entrenament** (*train*): les dades que el mètode fa servir per aprendre ajustant els seus paràmetres interns.

- **Validació** (*validation*): dades que s'utilitzen per provar el sistema una vegada entrenat. Com que són dades diferents es poden fer servir per comprovar si el sistema és capaç de processar dades noves. Aquestes dades ajuden el desenvolupador a ajustar els hiperparàmetres del sistema, és a dir, a canviar la seva configuració, després de la qual cosa cal tornar a entrenar-lo.

- **Prova** (*test*): dades que es reserven fins que el sistema està completament entrenat i ajustat, serveixen per valorar les prestacions del sistema final.

També és fonamental que els tres blocs anteriors segueixin la mateixa distribució de dades, és a dir, que els exemples de diferents classes i característiques hi estiguin repartits per igual.

Figura 63. Entrenament, validació i prova

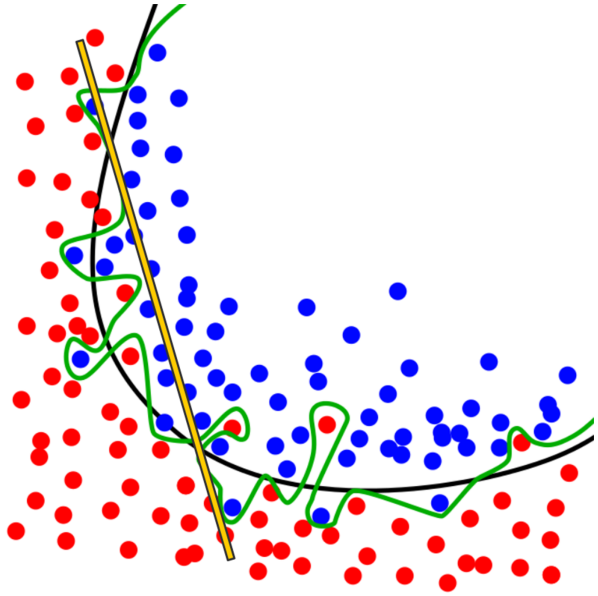
A grans trets, podem trobar-nos amb dos tipus d'errors, il·lustrats a la figura 64.

- **Error d'infraajust** (*underfitting*): indica que el sistema no és suficientment complex com per aprendre les dades d'entrenament. Per corregir-lo cal incrementar la complexitat del model (per exemple, més unitats en el cas d'una xarxa neuronal).

- **Error de sobreajust** (*overfitting*): indica que el sistema no és capaç de generalitzar el que ha après amb les dades d'entrenament a altres dades. Convé aplicar tècniques de **regularització**, explicades més endavant.

El comportament recomanable, seguint amb la figura, és l'indicat per la línia negra: un model suficientment complex com per adaptar-se a l'estructura general de les dades, però no tant com per perdre's intentant modelar els detalls de cadascun dels exemples, la qual cosa li impedeix tenir capacitat de **generalització**.

Figura 64. Tipus d'errors: infraajust (taronja) i sobreajust (verd)



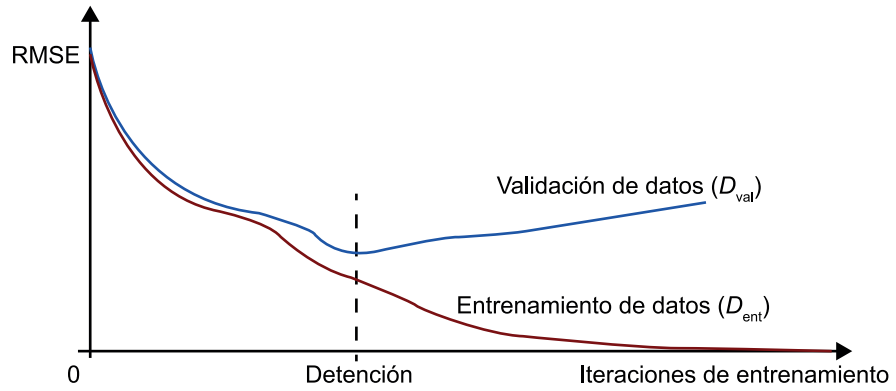
6.2.5. Algunes solucions

Dels problemes que hem vist anteriorment, sens dubte el més complicat d'esmenar és el del sobreajust. En general, les tècniques que tenen l'objectiu d'aconseguir un model amb més capacitat de generalització, i, per tant, sense sobreajust, reben el nom de tècniques de **regularització**. Vegem algunes de les tècniques de regularització més importants:

Moltes d'aquestes tècniques poden aplicar-se també a altres mètodes d'aprenentatge automàtic.

- **Detenció primerenca** (*early stopping*): és una idea tan senzilla com aturar l'entrenament quan el sistema està començant a sobreajustar-se a les dades d'entrenament. Es pot detectar que això passa quan l'error de **validació** comença a créixer, encara que l'error d'entrenament segueixi disminuint, com es pot veure a la figura 65. Aquest mètode sol aplicar-se a les èpoques d'entrenament, especialment quan s'utilitzen les dades d'entrenament diverses vegades.

Figura 65. Evolució dels errors d'entrenament i validació



Font: http://bit.ly/2ApsZar

- **Normalització de pesos**: és una tècnica senzilla que consisteix a limitar els valors dels pesos de les unitats de la xarxa neuronal evitant valors molt grans o molt petits. D'aquesta manera s'aconsegueix que la xarxa estigui més equilibrada i no doni excessiva importància a alguns exemples o combinacions de variables, la qual cosa al final evita el sobreajust.

- **Abandó** (*dropout*): és una tècnica sorprenent que s'aplica a les xarxes neuronals i que consisteix a desactivar unitats aleatòriament durant l'entrenament. Això provoca que la resta d'unitats es facin càrrec de les tasques de les unitats desactivades, la qual cosa al final redunda en una xarxa més robusta i més generalista. Aquesta tècnica funciona molt bé i té poc cost computacional, per la qual cosa s'usa amb molta freqüència, com es veurà en els exemples de codi.

- **Preparació de dades**: quan les dades disponibles no són suficients per aconseguir el nivell d'entrenament desitjat, de vegades convé reduir la complexitat del problema perquè la quantitat d'exemples sigui suficient. Per exemple, pot ser que un sistema de vigilància sigui impossible d'entrenar amb fotos completes de l'entrada a un edifici, així que potser convé fer servir només les fotos de cares i entrenar amb elles, és a dir, simplificant la tasca del sistema.

- **Augment de dades**: és una tècnica complementària de l'anterior, en aquest cas s'opta per augmentar el nombre de dades d'entrenament generant dades falses a partir dels exemples disponibles. Per exemple, afegint soroll a un enregistrament d'àudio, girant o retallant fotografies, etc.

6.2.6. Aprenentatge profund

En aquest apartat hem estudiat una xarxa neuronal senzilla, el perceptró, en la qual només hi ha tres capes. Una xarxa amb una sola capa oculta com aquesta és capaç d'aprendre qualsevol conjunt de dades finit, sempre que tingui un nombre suficientment gran d'unitats a la capa oculta.

El problema d'aquest plantejament, augmentar l'amplària de la capa oculta, és doble: primer, és una solució computacionalment costosa, ja que pot requerir una capa amb molts milers o fins i tot milions d'unitats; segon, un sistema així només s'aprenen les combinacions de variables presents en les dades d'entrenament. Així doncs, el seu poder de generalització és molt rudimentari.

Una estratègia alternativa per augmentar la complexitat d'una xarxa neuronal és incrementar el nombre de capes ocultes. Com a resultat, s'obté una xarxa que és capaç de generalitzar millor noves dades, ja que en cada capa es modela un nivell d'abstracció superior que en l'anterior. Així doncs, al final la xarxa és capaç de detectar característiques més generals.

Exemple

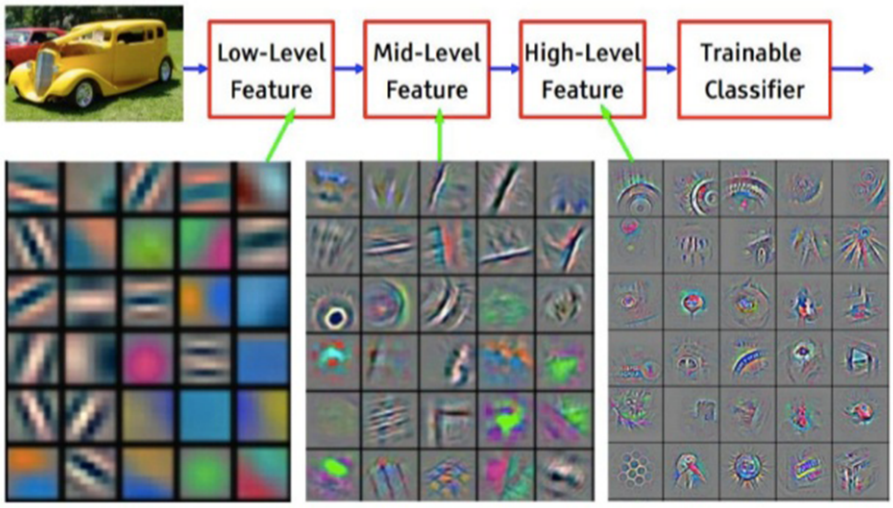
En un sistema de visió per ordinador amb moltes capes ocultes, la primera capa detectarà característiques senzilles com vores; la segona, elements composts amb els objectes de l'anterior, com cantonades, creus; la tercera, combinacions dels elements anteriors, com triangles, rectangles, etc. Així successivament, fins que les capes superiors són capaces de detectar característiques complexes com ulls, rodes, matrícules, senyals de trànsit, i així fins a produir un resultat global de gran qualitat per la seva capacitat d'abstracció.

Aquesta és la idea de l'aprenentatge profund: crear xarxes neuronals amb diverses capes ocultes. Per això s'anomena «profund». Depèn de l'autor, però generalment es considera profunda una xarxa amb deu o més capes.

Fins fa uns anys (al voltant del 2010) no s'havien utilitzat aquests sistemes de manera generalitzada per la dificultat i el cost computacional d'entrenar-los; no obstant això, diferents avanços teòrics i tècnics han obert les tècniques d'aprenentatge profund i han revolucionat la IA.

Als subapartats següents estudiarem diferents tipus de xarxes neuronals profundes i les seves aplicacions.

Figura 66. Característiques de baix, mitjà i alt nivell en anàlisi d'imatges



Font: http://bit.ly/2nreM7N

6.3. Perceptró multicapa

6.3.1. Idea

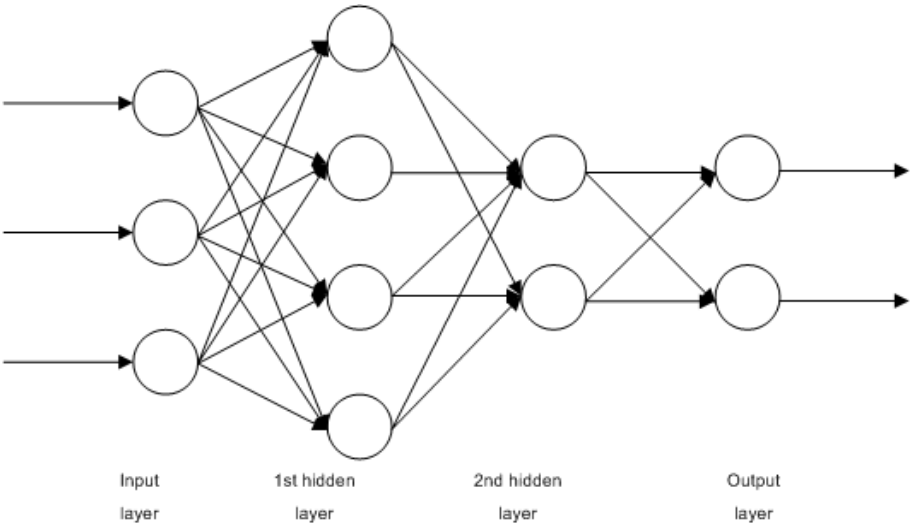
El tipus de xarxa neuronal profunda més senzill és el denominat **perceptró multicapa*** o també xarxa neuronal **prealimentada**.

* En anglès, *multilayer perceptron* (MLP) o bé *feed-forward neural network* (FNN).

Com el seu nom suggereix, simplement es tracta d'un perceptró amb diverses capes ocultes. L'avantatge d'aquesta configuració és que les successives capes ocultes van aprenent característiques cada vegada més complexes, ja que parteixen de les característiques apreses per les capes anteriors.

La seva estructura és molt senzilla: una capa d'entrada, diverses capes ocultes i una capa de sortida. Totes les capes estan completament connectades i només hi ha connexions cap endavant.

Figura 67. Exemple de perceptró multicapa



Font: Wikimedia.org

6.3.2. Exemple d'MLP

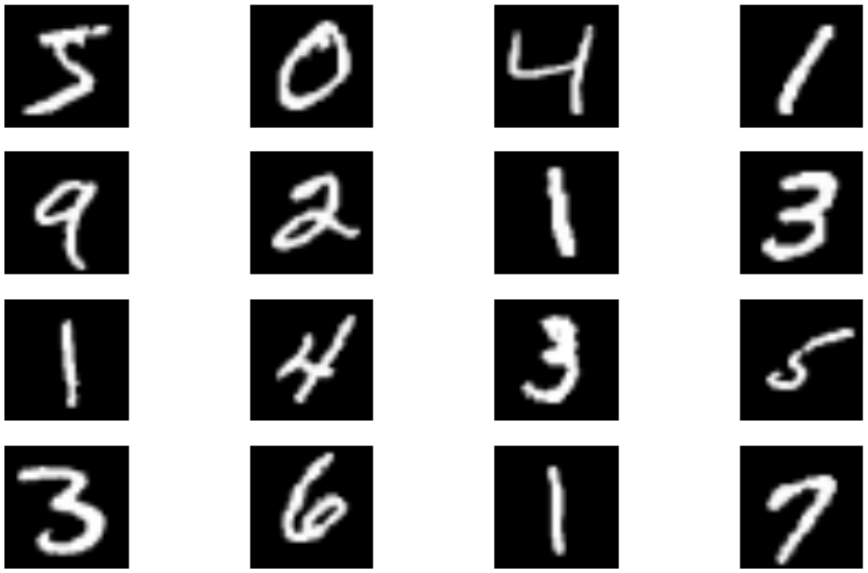
Un dels conjunts de dades estàndard per provar xarxes neuronals és la base de dades MNIST, que conté 70.000 imatges en escala de grisos de dígits manuscrits. Les imatges tenen una grandària de 28×28 píxels. Cada imatge té una etiqueta de classe de 0-9 igual al dígit representat a la imatge. Les dades estan repartides en seixanta mil imatges d'entrenament i deu mil imatges de test. El conjunt de dades s'inclou a la distribució de les llibreries Keras, la qual cosa permet treballar-hi d'una manera molt senzilla.*

*** Es pot accedir a la informació detallada a https://keras.io/datasets/.**

El programa següent mostra com crear un MLP que classifiqui les dades de MNIST en cadascuna de les deu classes. La xarxa s'entrenarà per minilots de cent vint-i-vuit exemples, i l'entrenament es repetirà vint vegades (èpoques). Després d'importar els paquets necessaris, normalitzar les dades i organitzar-les en entrenament i prova, es defineix el model de la xarxa. En aquest cas s'utilitza `keras.Sequential`, la qual cosa indica que es tracta d'un MLP, perquè les capes s'organitzen seqüencialment, sense salts ni tornar enrere. Concretament, aquesta xarxa té una primera capa d'entrada. A continuació, alterna una capa d'abandó (*dropout*) amb una capa densa. Finalment, afegeix una altra capa d'abandó prèvia a la capa de sortida, de tipus *softmax*, ja que es tracta d'un problema de classificació multiclasse.

La funció de cost o pèrdua (*loss*) és l'entropia creuada, a la versió dissenyada per a problemes de classificació, i l'optimitzador és `RMSprop`, que és una versió millorada del descens de gradients.

Figura 68. Setze primeres imatges de la base de dades MNIST



A Keras és necessari compilar el model, la qual cosa el prepara per a la seva execució de manera eficient, així com per definir quina serà la mètrica que s'usarà per avaluar la qualitat del model, en aquest cas la precisió en la classificació.

En aquest punt ja es pot entrenar el model amb les dades d'entrenament, i finalment s'avalua amb les dades de prova.

Capa d'abandó

Tot i que es generen com una capa més, les capes d'abandó no es compten com una capa més, sinó com un modificador de la capa següent.

Codi 6.1: perceptró multicapa com a classificador de digits de MNIST

```
1  import keras
2  from keras.datasets import mnist
3  from keras.models import Sequential
4  from keras.layers import Dense, Dropout
5  from keras.optimizers import RMSprop
6
7  # Configuracio general del programa
8  batch_size  = 128
9  num_classes = 10
10 epochs      = 20
11
12
13 # Organitzar dades en entrenament i prova
14 (x_train, y_train), (x_test, y_test) = mnist.load_data()
15
16 x_train  = x_train.reshape(60000, 784)
17 x_test   = x_test.reshape(10000, 784)
18
19 # Les dades d'entrada (pixels) son enteres.
20 # Cal convertir-les a reals per treballar-hi
21 x_train  = x_train.astype('float32')
22 x_test   = x_test.astype('float32')
23
24 # Normalitzar els valors a 0..1
25 x_train /= 255
26 x_test  /= 255
27 print(x_train.shape[0], 'train samples')
28 print(x_test.shape[0], 'test samples')
29
30 # Convertir cada numero de classe en un vector de 0s i 1s
31 # per poder-lo comparar amb la sortida softmax
32 y_train = keras.utils.to_categorical(y_train, num_classes)
33 y_test = keras.utils.to_categorical(y_test, num_classes)
34
```

```
35
36 # Definir el model de la xarxa, amb dues capes ocultes
37 model = Sequential()
38 model.add(Dense(512, activation='relu', input_shape=(784,)))
39 model.add(Dropout(0.2))
40 model.add(Dense(512, activation='relu'))
41 model.add(Dropout(0.2))
42 model.add(Dense(512, activation='relu'))
43 model.add(Dropout(0.2))
44 model.add(Dense(10, activation='softmax'))
45
46 model.summary()
47
48
49 # Compilar el model i entrenar-lo
50 model.compile(loss='categorical_crossentropy',
51               optimizer=RMSprop(),
52               metrics=['accuracy'])
53
54 history = model.fit(x_train, y_train,
55                     batch_size=batch_size,
56                     epochs=epochs,
57                     verbose=1,
58                     validation_data=(x_test, y_test))
59
60 # Finalment, avaluar-lo amb les dades de prova
61 score = model.evaluate(x_test, y_test, verbose=0)
62 print('Test loss:', score[0])
63 print('Test accuracy:', score[1])
```

Val la pena destacar algunes qüestions pràctiques d'aquest programa que cal tenir en compte a l'hora de treballar amb xarxes neuronals:

- És necessari **normalitzar** les dades d'entrada a la xarxa, preferiblement a un interval com 0..1, ja que si no l'optimització pot trigar molt més a convergir.

- En problemes de classificació múltiple cal convertir els valors de classe en vectors de zeros i uns.

- La funció de cost a l'hora d'entrenar la xarxa no té per què ser la mateixa que es faci servir per avaluar la qualitat del model. La funció de cost ha de ser fàcilment derivable, però pot ser que no sigui significativa per comparar l'execució de diversos sistemes. Per aquest motiu, a Keras s'especifica la funció de cost (paràmetre `loss`) i la mètrica d'avaluació (paràmetre `metrics`).

6.4. Classificació d'imatges amb xarxes neuronals convolucionals (CNN)

Les xarxes neuronals convolucionals* són un tipus particular de xarxes neuronals que s'utilitzen per a la classificació automàtica d'imatges. Així doncs, es tracta d'una tècnica de classificació supervisada, per la qual cosa és necessari disposar d'una base de dades amb imatges etiquetades segons un conjunt predefinit de classes.

* En anglès, *Convolutional Neural Networks* (CNN).

De la mateixa manera que a les xarxes neuronals convencionals, les CNN presenten una arquitectura en xarxa amb capes de neurones interconnectades entre si. En una xarxa neuronal, les interconnexions es realitzen mitjançant uns paràmetres (pesos i *offsets*) que poden ser apresos durant la fase d'entrenament. Una vegada entrenada, la xarxa neuronal pot ser utilitzada com a classificador de patrons aplicant dades a l'entrada i obtenint una sortida que prediu la classe a la qual pertany. En el nostre cas, aplicarem una imatge a l'entrada i obtindrem una etiqueta de la classe a la qual correspon. Per descomptat, durant la fase d'entrenament haurem de facilitar una quantitat suficient d'imatges representatives de cada classe que desitgem classificar.

A tall d'exemple d'aplicació, imagineu un classificador automàtic amb dues classes que distingeixi imatges de platges i de boscos. Per això necessitarem una base de dades amb un cert nombre d'imatges de platges i d'imatges de boscos. Durant la fase d'entrenament establirem els pesos de la xarxa a partir d'un conjunt d'imatges etiquetades segons la seva classe. Una vegada definida la xarxa, la podem utilitzar per classificar automàticament una nova imatge.

La diferència fonamental entre les xarxes neuronals convencionals i les xarxes neuronals convolucionals és que aquestes últimes estan específicament dissenyades perquè les dades d'entrada siguin imatges. Les CNN suposen un canvi de paradigma en el camp del reconeixement de patrons en imatges perquè la mateixa arquitectura de la xarxa s'encarrega de realitzar l'extracció d'atributs i de la seva posterior classificació automàtica. Les tècniques tradicionals de reconeixement de patrons en imatges requereixen una extracció d'atributs significatius de les imatges abans d'utilitzar un classificador supervisat per classificar-les. Aquests atributs poden estar basats en aspectes com formes, color o textures i solen requerir un esforç notable per part d'un expert que indiqui quines característiques són les més rellevants en un problema determinat.

Una imatge de grandària $NX \times NY$ píxels i NC canals de color es representa generalment mitjançant una matriu de dades de dimensió $NX \times NY \times NC$. Per exemple, una imatge de 32×32 píxels en un espai de color RGB ($NC = 3$, tres canals de color Red, Green i Blue) es representa mitjançant una matriu de grandària $32 \times 32 \times 3$. A cada canal, els píxels de la imatge poden prendre valors discrets entre un conjunt de 2^N nivells, en què N és la profunditat en bits de la imatge (per a una profunditat de $N = 8$ bits, per exemple, cada píxel pot prendre valors entre 0 i 255, això són 256 nivells). Resulta clar, llavors, que són necessaris un total de $NX \cdot NY \cdot NC \cdot N$ bits per emmagatzemar una imatge de grandària $NX \times NY \times NC$ amb una profunditat de N bits (per exemple, per emmagatzemar una imatge RGB 16 bits de grandària 32×32 es requereixen un total de $32 \cdot 32 \cdot 3 \cdot 16 = 49.152$ bits = 6,144 kB).

En una xarxa neuronal convencional amb una imatge com a dada d'entrada, cada neurona de la capa d'entrada podria arribar a tenir un total de $NX \times NY \times NC$ connexions, la qual cosa en una imatge de $256 \times 256 \times 3$ suposaria un total de 196.608 connexions. Resulta evident que en aquest cas la xarxa tindria un nombre excessiu de paràmetres per estimar, la qual cosa suposaria un enorme

cost computacional de la fase d'entrenament i un alt risc de sobreajust de les dades. Per evitar aquest problema, les xarxes convolucionals presenten una arquitectura en capes en la qual cada capa consisteix en un bloc de neurones organitzades en un volum regular.

El volum d'entrada, per exemple, és igual a la grandària de la imatge d'entrada $NX \times NY \times NC$.

A part de la capa d'entrada, hi ha tres tipus diferents de capes: capes convolucionals (*convolutional layers*), capes d'agrupació (*pooling layers*) i capes completament connectades (*fully-connected layers*), el funcionament de les quals es descriu breument a continuació:

1) Una **capa de convolució** aplica un conjunt de NF filtres al volum de la imatge d'entrada. Cada filtre consisteix en una matriu de pesos d'una certa grandària $M \times M \times NC$. A l'extensió espacial del filtre $M \times M$ se'l coneix com el camp receptiu d'una determinada neurona. Els pesos dels filtres seran determinats durant la fase d'entrenament de la xarxa. Cada filtre s'aplica a la imatge d'entrada mitjançant una operació matemàtica coneguda com a convolució, en la qual cada píxel de la imatge filtrada s'obté calculant la mitjana local ponderada utilitzant els coeficients del filtre com a matriu de pesos. Repetint aquesta operació per a cada píxel de la imatge d'entrada i cada filtre de la capa convolucional, vam acabar tenint un resultat en forma de volum de dimensions $(NX - M + 1) \times (NY - M + 1) \times NF$. Al final de les capes convolucionals s'aplica una funció no lineal d'activació que està inspirada en el funcionament de les neurones en sistemes de percepció sensorial. Aquesta funció té principalment dos objectius: d'una banda, evitar que els valors creixin indefinidament a mesura que les dades es propaguen per la xarxa; d'una altra, permetre un augment de la flexibilitat de l'arquitectura perquè la xarxa pugui aprendre relacions no lineals i, per tant, de més complexitat entre els atributs extrets en cada capa. Una de les funcions d'activació més utilitzades és la unitat de rectificació lineal (*rectifier linear unit*, comunament coneguda com a ReLU), que realitza una operació $f(x) = \max(0,x)$.

2) Les **capes d'agrupament** realitzen una operació de submostreig espacial de les dades, i, per tant, agrupen característiques en les capes ocultes de la xarxa. El resultat és, doncs, de dimensionalitat inferior a les dades d'entrada de la capa. Una de les més utilitzades és la funció *max-pooling*, que agrupa regions d'una certa grandària i les substitueix pel valor màxim de píxel a la regió.

3) La **capa completament connectada** consisteix en una capa que connecta totes les neurones d'entrada amb les de la sortida. Tot i que es pot utilitzar en el disseny de les capes interiors de la xarxa, habitualment s'utilitzen a la capa de sortida, proporcionant un índex de classe que codifica les NL classes del problema (proporcionant un volum de sortida de dimensions $1 \times 1 \times NL$).

6.4.1. Implementació de les CNN a Python utilitzant les llibreries Keras

En aquest exemple d'implementació de les CNN a Python tornem a utilitzar la base de dades MNIST, que conté setanta mil imatges en escala de grisos ($NC = 1$) d'una grandària de 28×28. Cada imatge té una etiqueta de classe de 0-9 igual al dígit representat a la imatge. Les dades estan repartides en seixanta mil imatges d'entrenament i deu mil imatges de prova. El *dataset* s'inclou en la distribució de les llibreries Keras, la qual cosa permet treballar d'una manera molt senzilla.*

*** Es pot accedir a la informació detallada a https://keras.io/datasets/.**

El codi de l'exemple comença carregant les llibreries Keras i la base de dades MNIST. A continuació, s'utilitza la instrucció per carregar les imatges d'entrenament i prova en dues variables *X_train* i *X_test* amb una grandària de $60.000 \times 1 \times 28 \times 28$ i $10.000 \times 1 \times 28 \times 28$ respectivament. El pas següent consisteix a redimensionar cada conjunt d'imatges en vectors de grandària $Nim \times NC \times NX \times NY$ píxels utilitzant la funció *reshape** de les llibreries NumPy. Posteriorment es normalitzen les dades perquè cada imatge es transformi d'escala de grisos entre 0-255 a valors reals entre 0-1.

*** Descrita a http://bit.ly/2jbFMnx**

Les etiquetes de classe han de ser codificades en format categòric en lloc de com a valor sencer. Per fer-ho s'ha d'utilitzar la funció *to_categorical* de les llibreries NumPy. La variable categòrica resultant consisteix en un vector columna amb tantes posicions com nombre de classes en el qual totes les posicions són zero excepte la que indica la classe que desitgem codificar.

La funció *baseline_model* és la que permet definir l'arquitectura del modelo CNN.* El model CNN que utilitzarem es compon de sis capes que es descriuen a continuació:

*** Les funcions de disseny de capes convolucionals que ofereixen les llibreries Keras estan detallades a http://bit.ly/2AMuMqq.**

Capa 0: la capa d'entrada consisteix en el volum de dades d'entrada, que en aquest cas és $Nim \times 28 \times 28 \times 1$.

Capa 1: capa convolucional 2D amb $NF = 32$ filtres de grandària 5×5 ($M = 5$) i una funció d'activació no lineal del tipus ReLU.

Capa 2: capa d'agrupament *max-pooling* amb una grandària 2×2 (funció MaxPooling2D).

Capa 3: capa de regularització que exclou aleatòriament el 20% de les neurones per evitar fenòmens de sobreajust de les dades.

Capa 4: capa de redimensionament de matrius 2D en vectors. L'objectiu és proporcionar el format adequat a la capa completament connectada que ve a continuació.

Capa 5: capa completament connectada amb cent vint-i-vuit neurones i una funció de rectificació ReLU com a funció no lineal d'activació.

Capa 6: capa de sortida amb deu neurones que codifiquen les deu classes del problema. S'aplica una funció no lineal d'activació del tipus exponencial

normalitzada (*soft-max*) per proporcionar una predicció probabilística de l'etiqueta de classe.

Al codi de l'exemple, la definició del model CNN es realitza a la funció definida a la línia 35. Una vegada definides les capes descrites anteriorment, s'especifiquen les característiques de l'algorisme d'optimització per a l'ajust dels paràmetres del model durant la fase d'entrenament: funció objectiu, algorisme d'optimització i mètrica per avaluar l'error. En aquest cas de l'exemple, s'utilitza una funció de cost d'entropia creuada, un algorisme de descens per gradient. L'actualització del gradient es realitza aplicant la regla d'actualització Adam, que utilitza un gradient suavitzat i aplica una correcció durant el règim transitori inicial. Finalment, l'exactitud (*accuracy*) com a mètrica d'error de predicció. Aquesta mètrica s'utilitza també durant la fase de validació del model.

A continuació, es realitza la instanciació del model (línia 55) i, finalment, es procedeix a l'ajust del model (fase d'entrenament, línia 58). En aquesta instrucció el paràmetre *epochs* indica el nombre de vegades que s'itera el procediment d'entrenament a partir de les dades d'entrenament. El paràmetre *batch_size* indica el nombre d'observacions que s'utilitzen en cada actualització del gradient en l'algorisme d'optimització. A la mateixa instrucció es valida els resultats del model utilitzant les dades de validació i aplicant aquests mateixos paràmetres.

```
1  # Carregar la base de dades MNIST i les llibreries Keras:
2  import numpy
3  from keras.datasets import mnist
4  from keras.models import Sequential
5  from keras.layers import Dense
6  from keras.layers import Dropout
7  from keras.layers import Flatten
8  from keras.layers.convolutional import Conv2D
9  from keras.layers.convolutional import MaxPooling2D
10 from keras.utils import np_utils
11 from keras import backend as K
12 K.set_image_dim_ordering('th')
13
14 # Establir la llavor del generador de nombres aleatoris per
15 #garantir la reproductibilitat dels resultats
16 seed = 7
17 numpy.random.seed(seed)
18
19 # Accedir a les imatges d'entrenament i test
20 (X_train, y_train), (X_test, y_test) = mnist.load_data()
21
22 # Dimensionar les imatges en vectors
23 X_train=X_train.reshape(X_train.shape[0],1,28,28).astype('float32')
24 X_test=X_test.reshape(X_test.shape[0],1,28,28).astype('float32')
25
26 # Normalitzar les imatges d'escala de grisos 0-255 (entre 0-1):
27 X_train = X_train / 255
28 X_test = X_test / 255
29
30 # Codificar les etiquetes de classe en format de vectors
31 # categorics amb deu posicions:
32 y_train = np_utils.to_categorical(y_train)
33 y_test = np_utils.to_categorical(y_test)
34 num_classes = y_test.shape[1]
```

```
35
36 # Funcio en la qual es defineix l'arquitectura del model:
37 def baseline_model():
38 # Capa d'entrada:
39         model = Sequential()
40 # Primera capa (convolucional):
41         model.add(Conv2D(32, (5, 5), input_shape=(1, 28, 28),
42                   activation='relu'))
43 # Tercera capa (agrupament):
44         model.add(MaxPooling2D(pool_size=(2, 2)))
45 # Quarta capa (regularitzacio):
46         model.add(Dropout(0.2))
47 # Cinquena capa (redimensionament):
48         model.add(Flatten()):
49 # Sisena capa (completament connectada)
50         model.add(Dense(128, activation='relu'))
51 # Capa de sortida:
52         model.add(Dense(num_classes, activation='softmax'))
53 # Compilar el model i especificar metode i metrica d'optimitzacio:
54         model.compili(loss='categorical_crossentropy',
55                       optimizer='adam', metrics=['accuracy'])
56         return model
57
58 # Crida al model:
59 model = baseline_model()
60
61 # Ajust del model:
62 model.fit(X_train, i_train, validation_data=(X_test, y_test),
63           epochs=10, batch_size=200, verbose=2)
64
65 # Avaluacio del model utilitzant les dades de prova:
66 scores = model.evaluate(X_test, y_test, verbose=0)
67 print("Exactitud del model: %.2f%%" % (100*scores[1]))
```

Durant l'execució de la fase d'entrenament apareixen en pantalla els resultats obtinguts en cadascuna de les deu iteracions. En el codi de l'exemple els resultats són els següents, que haurien de ser reproduïbles si s'aplica la mateixa llavor del generador de nombres aleatoris que apareix a les línies 15 i 16:

>Train on 60000 samples, validate on 10000 samples
Epoch 1/10
175s - loss: 0.2310 - acc: 0.9345 - val_loss: 0.0826 - val_acc: 0.9742
Epoch 2/10
180s - loss: 0.0737 - acc: 0.9780 - val_loss: 0.0471 - val_acc: 0.9839
Epoch 3/10
173s - loss: 0.0534 - acc: 0.9837 - val_loss: 0.0429 - val_acc: 0.9860
Epoch 4/10
173s - loss: 0.0403 - acc: 0.9878 - val_loss: 0.0405 - val_acc: 0.9866
Epoch 5/10
172s - loss: 0.0336 - acc: 0.9894 - val_loss: 0.0344 - val_acc: 0.9880
Epoch 6/10
172s - loss: 0.0274 - acc: 0.9916 - val_loss: 0.0308 - val_acc: 0.9898
Epoch 7/10
175s - loss: 0.0233 - acc: 0.9927 - val_loss: 0.0353 - val_acc: 0.9881
Epoch 8/10
173s - loss: 0.0203 - acc: 0.9936 - val_loss: 0.0326 - val_acc: 0.9887
Epoch 9/10

```
172s - loss: 0.0170 - acc: 0.9943 - val_loss: 0.0304 - val_acc: 0.9901
Epoch 10/10
172s - loss: 0.0144 - acc: 0.9956 - val_loss: 0.0317 - val_acc: 0.9902
```

A les dues últimes línies de codi s'avalua l'exactitud (*accuracy*) global del model, que en aquest cas és del 99.2%.

6.5. Xarxes recurrents

6.5.1. Idea

Les xarxes neuronals vistes fins ara només poden processar dades de longitud coneguda, definida per l'amplària de la capa d'entrada. No obstant això, no poden dur a terme tasques sobre tipus de dades seqüencials, com text, so, senyals, etc. Tot i que seria possible introduir cada element (per exemple, cada paraula) en una xarxa MLP o CNN seqüencialment, aquestes xarxes tracten cada exemple independentment dels anteriors, per la qual cosa no tindrien gens en compte l'ordre de les paraules en un text, per exemple. Per aquesta raó, MLP i CNN no són adequades per treballar amb dades seqüencials i estructurats, ja que el flux de senyals en elles només és cap endavant.

Perquè una xarxa neuronal tingui en compte la història de dades llegides i, per tant, sigui capaç de processar seqüències adequadament, és necessari que tingui algun tipus de **memòria** de les dades llegides anteriorment. En aquest context, això implica algun tipus de tornada enrere dels senyals cap a capes anteriors. Aquest tipus de xarxes es denominen **recurrents** (*recurrent neural networks*, RNN).

Hi ha moltes maneres de crear una RNN, de les quals la més senzilla és afegir una o més connexions cap enrere en una FNN. No obstant això, aquesta aproximació general és molt inestable i extremadament difícil d'entrenar per efecte de la realimentació de senyals, que provoquen que els valors alts creixin sense control i els baixos s'apaguin, fet que dona lloc al que es denomina el problema dels gradients evanescents.

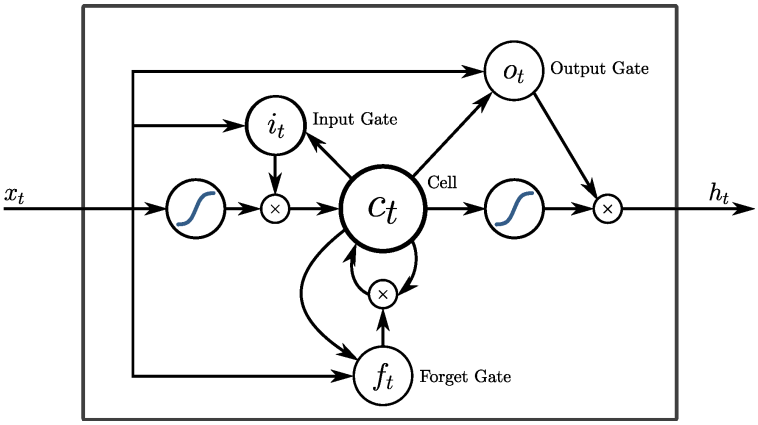
Per aquest motiu, a la pràctica la manera de crear una RNN és utilitzar unitats especials que tenen memòria i que s'ha comprovat que no tenen problemes d'entrenament.

El tipus d'unitat més habitual en RNN és l'anomenada unitat amb memòria a curt i llarg termini (*long short-term memory* o LSTM), dissenyada per convergir ràpidament en el seu entrenament alhora que és capaç d'aprendre dependències a llarg termini en les dades. La seva estructura ve donada a la figura següent, de la qual bàsicament es pot dir sense entrar en gaire detall que la clau de les LSTM per aconseguir estabilitat en l'entrenament i en els patrons

detectats consisteix a no aplicar la funció d'activació als seus components recurrents; en evitar aquesta realimentación, els valors numèrics són estables.

El comportament de la unitat LSTM està controlat per tres «portes» (*gates*): la porta d'entrada (i_t), que controla si un nou valor entra en la memòria; la porta d'oblit (f_t), que controla si un valor existent s'ha de reemplaçar per un valor nou, i la porta de sortida (o_t), que controla si el valor emmagatzemat s'usa per calcular la funció d'activació i, per tant, la sortida de la unitat. Els valors de les portes depenen, al seu torn, dels pesos apresos per la unitat.

Figura 69. Estructura d'una unitat LSTM



Font: Wikimedia.org

És molt senzill utilitzar LSTM per crear RNN. A Keras simplement es pot crear una capa de tipus LSTM. De fet, el més habitual és combinar capes LSTM amb capes normals (FNN), de manera que hi hagi un cert preprocés amb capes FNN i així la capa o capes LSTM rebin característiques més elaborades i significatives.

El 2014 es va proposar una simplificació de les LSTM denominada **unitat recurrent amb portes** (*gated recurrent unit*, GRU) que aconsegueix un comportament equivalent amb menys complexitat computacional.

Tot i que s'utilitzin unitats fiables com LSTM o GRU, les RNN segueixen sent més difícils d'ajustar i requereixen molta cura en els seus diferents hiperparàmetres.

6.5.2. Programació

El lloc web IMDB* (*Internet Movie DataBase*) és un dels llocs de cinema i pel·lícules més populars a internet. Els milers d'opinions escrites pels usuaris s'han aprofitat per crear un corpus d'opinions escrites al costat d'una valoració que indica si són positives o negatives. Utilitzarem aquest conjunt de dades per entrenar una RNN com a classificador binari.

* http://www.imdb.com/

En el cas de processament de seqüències, són necessaris alguns passos especials per preparar les dades:

- En primer lloc, pot haver-hi moltes paraules al corpus, però a l'exemple es faran servir només vint mil paraules diferents. Igualment, només es prendran les primeres vuitanta paraules de cada opinió.

- Com que algunes opinions poden tenir menys de vuitanta paraules, es fa necessari emplenar (*pad_sequence*) les opinions curtes perquè tots els exemples càpiguen en una matriu quadrada.

- En aquest exemple de processament de text hi ha moltes paraules diferents (vint mil). Si usem una codificació simple, haurem de representar cada paraula com un vector de 20.000 bits, en el qual tots valdran zero menys el corresponent a la paraula que entri en cada moment. Això és molt ineficient computacionalment i pobre des del punt de vista de la representació de la informació. Per aquest motiu, el que se sol fer en aquests casos és generar una **projecció** (*embedding*) de les dades en un espai vectorial de moltes menys dimensions, de l'ordre del centenar. Cada paraula del corpus es convertirà en un punt en aquest espai vectorial, la qual cosa a més de condensar la informació ajuda a associar conceptes propers. Per aconseguir-ho es crea una **capa de projecció**, la sortida de la qual són vectors a l'espai vectorial projectat. Això, a més, és molt més eficient computacionalment parlant i simplifica el disseny de la xarxa, ja que la sortida de la capa de projecció té una grandària raonable com a entrada per a les capes següents. És important destacar que és la mateixa capa la que aprèn la projecció més adequada durant l'entrenament de la xarxa.

D'altra banda, les capes d'una RNN que fa servir LSTM tenen una disposició seqüencial, com en el cas d'una FNN, i l'única diferència és que una de les capes és de tipus LSTM. De fet, es podrien afegir capes denses o de convolució si es considerés útil.

A la xarxa que es construeix a l'exemple següent hi ha tres capes: la de projecció, que redueix el vocabulari a un espai de cent vint-i-vuit dimensions; l'LSTM, amb factor d'abandó inclòs, i la capa de sortida, en aquest cas amb una sola unitat, perquè la resposta en aquest cas és binària.

La funció de cost és l'entropia creuada versió binària, i l'optimitzador és Adam, una de les millors variants del descens de gradients actualment.

Codi 6.2: classificador d'opinions de pel·lícules mitjançant RNN amb LSTM

```
1 from keras.preprocessing import sequence
2 from keras.models import Sequential
3 from keras.layers import Dense, Embedding
4 from keras.layers import LSTM
5 from keras.datasets import imdb
6
7 # Nombre de paraules diferents usades com a maxim
```

```
max_features = 20000
# De cada opinio s'agafen les 80 primeres paraules
maxlen = 80
# I les opinions s'agrupen en lots de 32
batch_size = 32

# Carregar les dades
print('Loading data...')
(x_train, y_train), (x_test, y_test) =
          imdb.load_data(num_words=max_features)
print(len(x_train), 'train sequences')
print(len(x_test), 'test sequences')

# Empaquetar els exemples en matrius quadrades (omplir)
print('Pad sequences (samples x time)')
x_train = sequence.pad_sequences(x_train, maxlen=maxlen)
x_test = sequence.pad_sequences(x_test, maxlen=maxlen)
print('x_train shape:', x_train.shape)
print('x_test shape:', x_test.shape)

# Crear el model amb tres capes
print('Build model...')
model = Sequential()
model.add(Embedding(max_features, 128))
model.add(LSTM(128, dropout=0.2, recurrent_dropout=0.2))
model.add(Dense(1, activation='sigmoid'))

# Compilar i entrenar
model.compile(loss='binary_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])

print('Train...')
model.fit(x_train, y_train,
          batch_size=batch_size,
          epochs=15,
          validation_data=(x_test, y_test))

# Avaluar amb les dades de prova
score, acc = model.evaluate(x_test, y_test,
                            batch_size=batch_size)
print('Test score:', score)
print('Test accuracy:', acc)
```

6.6. Altres arquitectures

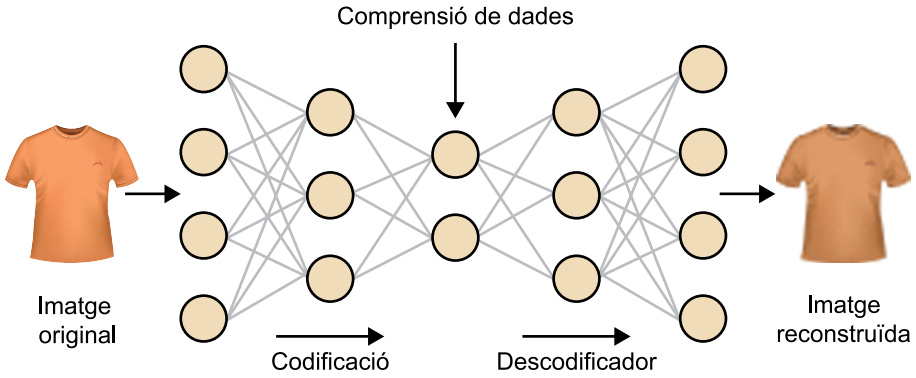
Hi ha moltes altres maneres de dissenyar una xarxa neuronal per resoldre diferents tasques. En aquest subapartat en veurem algunes de les més rellevants.

6.6.1. Autocodificadors

Els autocodificadors (*autoencoders*) són xarxes neuronals de propagació cap endavant no supervisades que tenen per objectiu que la sortida reprodueixi una còpia idèntica a l'entrada. Quin sentit té això? El detall important és que, tot i que lògicament la capa de sortida té la mateixa amplària que la d'entrada, les capes internes tenen menys unitats. Això obliga la xarxa a aprendre una representació més compacta de les dades d'entrenament, és a dir, una representació amb menys dimensions. D'aquesta manera, el seu objectiu és similar al dels mètodes de reducció de dimensionalitat com PCA.

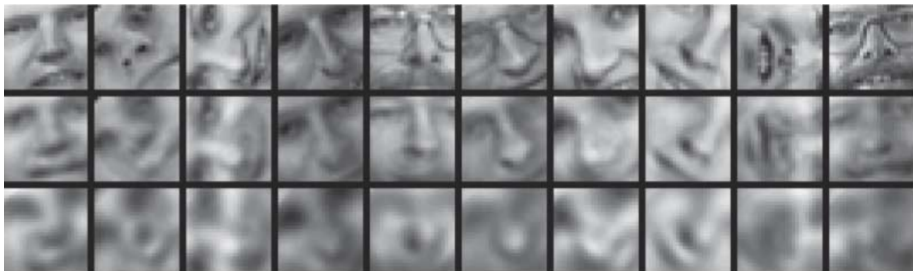
A la figura 70 es pot veure un diagrama senzill d'un autocodificador. A la pràctica poden tenir moltes més capes; és important que l'amplària de les capes es redueixi progressivament.

Figura 70. Estructura d'un autocodificador



La figura 71 compara la qualitat de la representació comprimida d'una sèrie de fotografies de cares. La primera fila són les fotografies originals; la segona, la representació generada per un autocodificador; la tercera, la representació generada per PCA. La dimensió de la capa interna de l'autocodificador és la mateixa que el nombre de components principals utilitzat. Com es veu, l'autocodificador és capaç de reproduir les imatges originals amb més fidelitat.

Figura 71. Comparació d'autocodificadors i PCA



Font: Wikimedia.org

6.6.2. Aprenentatge per reforç

Si bé l'aprenentatge per reforç (*reinforcement learning* o RL) ha experimentat un creixement enorme amb els sistemes d'aprenentatge profund, cal aclarir que són dos conceptes diferents. Vegem, en primer lloc, en què consisteix l'aprenentatge per reforç, per després explicar com se li pot aplicar solucions amb xarxes neuronals profundes.

A diferència de les situacions d'aprenentatge automàtic «clàssiques», en les quals hi ha un conjunt de dades, etiquetatges o no, amb els quals s'entrena un sistema, en l'aprenentatge per reforç un sistema aprèn per la seva interacció amb l'entorn, concretament per la situació que percep, les accions que executa i les conseqüències de les seves accions.

Imaginem un robot que ha de recollir les peces d'una habitació: suposem que el robot fa una imatge de l'habitació, decideix moure's cap a un punt determinat i agafar una peça; com a conseqüència, veurà que hi ha una peça menys a l'habitació i que aquesta operació ha tingut èxit. Per contra, si va cap a un punt en el qual no hi ha cap peça i intenta agafar alguna cosa, veurà que no ha avançat cap al seu objectiu final.

De la mateixa manera, en l'aprenentatge per reforç l'**agent**, és a dir, el sistema que aprèn i actua, guia el seu aprenentatge per les **recompenses** que rep a conseqüència de les seves accions. Les recompenses poden ser de naturalesa diversa, tot i que sempre orientades a premiar l'avanç cap a l'objectiu de l'agent: punts aconseguits en un joc, quilòmetres conduïts cap a la destinació sense accidents, peces acoblades per un robot, etc.

De manera més formal, un sistema d'aprenentatge per reforç es compon dels elements següents:

- Un conjunt d'**estats** del problema, S. Al joc del marro, per exemple, S serà tots els estats possibles de creu, cercle o casella buida, un total d'estats 3^9 possibles, tot i ser un joc tan senzill. No obstant això, el nombre d'estats possibles es dispara ràpidament, a unes 10^{40} posicions vàlides per als escacs o unes 10^{100} per al Go. Fora de l'àmbit dels jocs, el nombre d'estats possibles és impossible de calcular i, a la pràctica, infinit: quants estats possibles hi ha per al problema de la conducció automàtica? Quantes combinacions de vehicles, condicions ambientals, carretera, estat del vehicle, vianants, etc.?

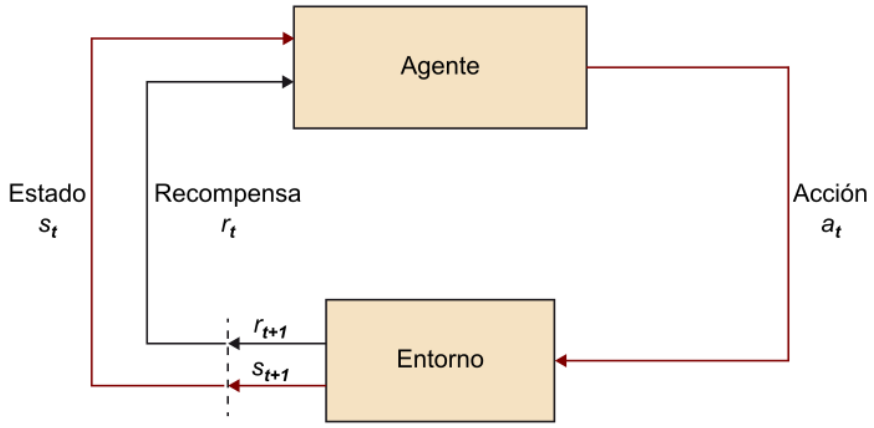
- Un conjunt de possibles **accions**, A. En un joc serien els possibles moviments. Amb un robot A són els moviments que pot executar; en un sistema de conducció automàtica, les diferents accions que pot ordenar al cotxe: accelerar, frenar, girar, etc.

- Una funció de **recompensa**, R, que retorna un nombre real en recompensa per l'acció presa per l'agent en un estat determinat, és a dir, $R : S \times A \to \mathbb{R}$. És molt important destacar que la recompensa depèn de l'acció presa i de l'estat actual, ja que de vegades serà bo que el cotxe acceleri, però altres vegades no.

- En teoria, un agent d'un sistema d'RL hauria d'aprendre's una **taula de recompenses** $Q = S \times A$ per saber perfectament quina és l'acció que s'ha d'aplicar en cada estat del sistema. No obstant això, com per a gairebé qualsevol problema S, té una grandària gairebé infinita, per la qual cosa és impossible emmagatzemar aquest coneixement en una taula. Per aquest motiu s'han proposat diferents mètodes per aprendre aproximacions raonablement bones a Q, mètodes que reben el nom de Q-Learning.

La figura 72 resumeix els elements d'un sistema RL que s'acaben d'explicar.

Figura 72. Diagrama de control d'un sistema d'aprenentatge per reforç



Font: extret dels materials d'IA per a videojocs (UOC)

Enllaç d'interès

La pàgina https://gym.openai.com/ conté gran quantitat de jocs preparats per entrenar sistemes d'aprenentatge per reforç.

Tot i que hi ha diverses estratègies per a Q-Learning, els sistemes que utilitzen xarxes neuronals profundes estan obtenint grans èxits en diferents àmbits, com el sistema DQN, que aprèn a jugar autònomament a jocs de consola; el sistema AlphaGo, que va vèncer al campió mundial de Go, o nombrosos sistemes de conducció automàtica i robòtica, entre altres.

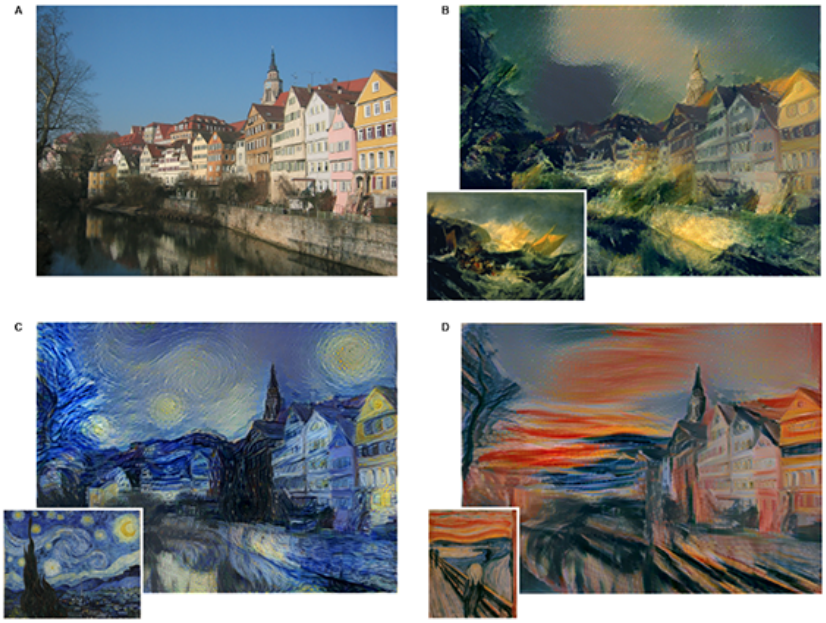
Els tipus de xarxes neuronals empleats per RL depenen de la tasca en qüestió; si per exemple l'entrada sensorial és d'imatge o vídeo, la primera fase del sistema sol ser una xarxa CNN, després de la qual es connecta una xarxa FNN. També poden utilitzar-se xarxes RNN si en tenir en compte els estats anteriors es pot millorar el rendiment del sistema.

6.6.3. Sistemes generadors

Les **xarxes generatives antagòniques** (*generative adversarial networks*, GAN) són sistemes no supervisats que tenen l'objectiu de generar falsos exemples, però que resultin creïbles i enganyin experts. Per exemple, entrenada amb fotos de cares, una GAN és capaç de crear cares noves que semblin reals. També pot utilitzar-se per modificar continguts, per exemple, redibuixar una fotografia normal de l'estil d'algun pintor famós, acolorir una fotografia en blanc i negre, o convertir un dibuix esquemàtic en una fotografia amb materials.

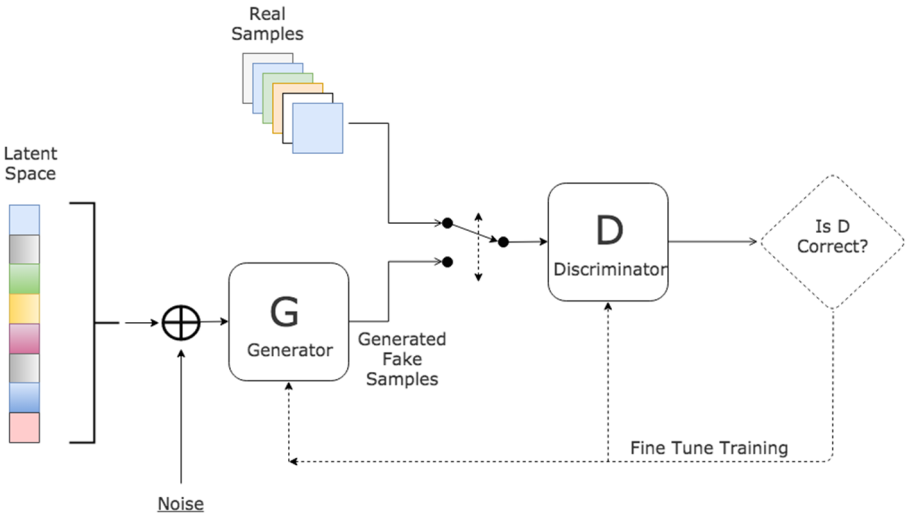
Una GAN es compon de dues xarxes independents: una crida **discriminador**, que s'especialitza a distingir exemples veritables de falsos, i una altra crida **generador**, que s'especialitza a generar exemples falsos a partir d'un espai d'exemples possibles. La clau d'una GAN és que les xarxes es posen en competència, de manera que el discriminador aprèn a distingir millor els exemples falsos dels veritables i el generador, a canvi, ha de generar cada vegada exemples més creïbles i, per tant, de més qualitat.

Figura 73. Quadres pintats per un sistema GAN a partir d'una fotografia



Respecte al tipus de xarxa, les GAN s'utilitzen principalment en processament i generació d'imatges; en aquest cas, el discriminador sol ser una CNN, mentre que el generador es construeix mitjançant una xarxa **deconvolucional**, que a grans trets és com una CNN però a l'inrevés, que passa de menys unitats a més en la sortida (similar a la meitat posterior d'una xarxa autocodificadora).

Figura 74. Estructura d'una xarxa generativa antagònica



Font: http://www.kdnuggets.com/2017/01/generative-adversarial-networks-hot-topic-machine-learning.html