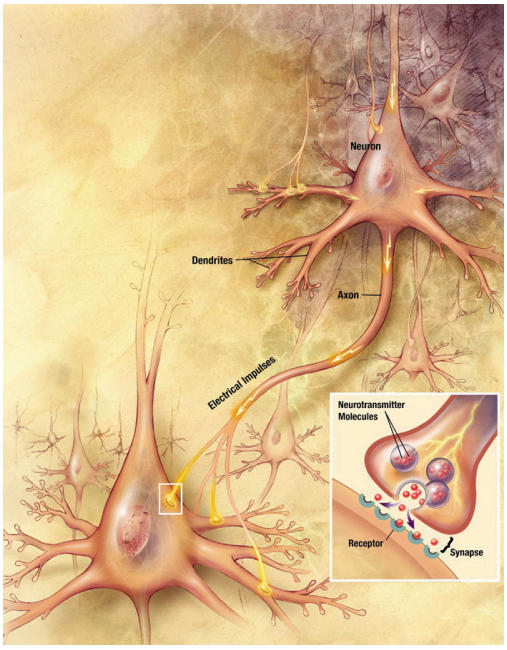
Xarxes neuronals i aprenentatge profund

Iloc:	Institut d'Ensenyaments a Distància de les Illes Balears	Imprès per:	Carlos Sanchez Recio
Curs:	Sistemes d'aprenentatge automàtic	Data:	divendres, 7 de febrer 2025, 20:20
Llibre:	Xarxes neuronals i aprenentatge profund		

Taula de continguts

1. Què és una xarxa neuronal?

Les neurones són els constituents estructurals del cervell. Al cervell sempre se li ha associat una gran capacitat de càlcul, i es pot definir mitjançant el símil que és un ordinador molt complex, no lineal i paral·lel. Les seves qualitats les aconsegueix mitjançant una sofisticada xarxa de neurones amb interaccions sinàptiques. A la il·lustració següent veim que la sinapsi és la connexió entre diferents neurones.



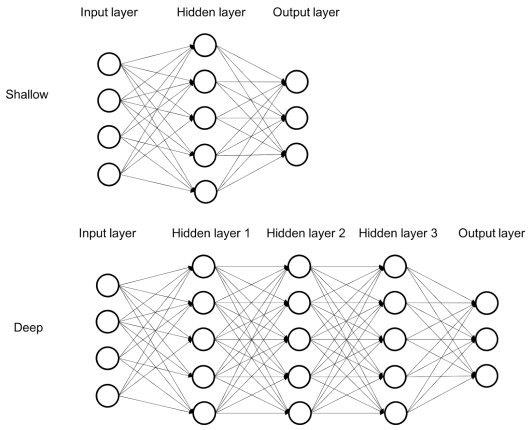
L'impuls elèctric es transmet a través de l'axon d'una neurona cap al cos i les dendrites d'unes altres neurones.

Una xarxa neuronal artificial és un algorisme matemàtic amb capacitat per recordar experiències i fer-les disponibles per al seu ús. Recorda el cervell humà en dos aspectes:

- El coneixement és adquirit per la xarxa mitjançant un procés d'aprenentatge.
- La força de la connexió entre neurones (pesos sinàptics) és usada per emmagatzemar el coneixement.

Una xarxa neuronal aprèn mitjançant la modificació dels pesos sinàptics. També es pot modificar la topologia. Altres maneres d'anomenar xarxes neuronals són sistemes connexionistes, processadors distribuïts paral·lels o neurocomputadors.

Les xarxes neuronals s'estructuren en capes. Una xarxa neuronal que només tengui capa d'entrada, com a màxim una capa intermèdia i una capa de sortida s'anomena *shallow neural network*, que podem traduir com a xarxa neuronal superficial. Quan hi ha més d'una capa interna ja es parla de *deep neural network* (xarxa neuronal profunda) i *deep learning*.



Cada una de les fletxes del diagrama correspon al pes que té assignada la connexió entre les neurones que uneix. L'entrada d'una neurona és una combinació lineal de les sortides de les neurones de la capa anterior, afegint-hi un biaix. A més, la neurona aplicarà una funció no lineal a la combinació lineal d'entrada.

Vegem per tant que hi ha una relació estreta que hi ha entre una neurona i els models de regressió lineal i logística que vàrem veure al lliurament anterior.

Una neurona amb funció d'activació la funció logística correspon a una regressió logística.

Una neurona amb funció d'activació lineal, la identitat, correspon a una regressió lineal.

1.1. Cronologia

A continuació presentam cronològicament una sèrie de fets rellevants en el desenvolupament de les xarxes neuronals, des dels seus inicis fins al ressorgiment dels anys 80 i l'eclosió actual de l'aprenentatge profund generatiu.

- 1943 primer treball de **McCulloch i Pitts**. McCulloch, psiquiatre i neuroanatomista, i Pitts, matemàtic, van descriure un càlcul lògic de xarxes neuronals. Von Neumann va utilitzar elements neurals ideals a base d'elements discrets *switch-delayed* per al desenvolupament de l'EDVAC que va resultar a l'ENIAC.
- 1958 publicació de Perceptron treball de **Rosenblatt** i del teorema de convergència del perceptró.
- Anys 1970, són anys de silenci. Motius tecnològics i l'article molt crític de **Minsky i Papert** (1969).
- Anys 80, ressorgiment. Hi ha treballs en diverses línies: Grossberg, **Hopfield** (1982), Kohonen, el 1986 l'algorisme de back-propagation de Rumelhart, **Hinton** i Williams i el 1988 les Radial Basis Functions de Broomhead i Lowe.
- **LeNet** xarxa neuronal convolucional de Yann LeCun. Tractarem les xarxes neuronals convolucionals al lliurament sisè de Programació d'Intel·ligència Artificial.
- **AlexNet** va ser una arquitectura de xarxes convolucionals que aconseguí un gran èxit en visió artificial.
- **Long-Short Term Memory**, de Hochreiter i Schmidhuber.
- **Generative Adversarial Networks**, d'Ian Goodfellow.
- **Arquitectura transformer,** presentada a l'article *Attention Is All You Need*. Els transformers són la base del models estesos del llenguatge, entre els quals GPT, BART o LlaMA.
- **Models de difusió**, a la base de la generació d'imatges estàtiques o en moviment. Els tractarem al lliurament setè de Models d'Intel·ligència Artificial, IA generativa.

1.2. Avantatges i inconvenients

Les xarxes neuronals tenen diverses característiques que les fan molt adequades en l'aprenentatge automàtic actual.

- Modelitzen relacions no lineals entre l'entrada i la sortida. Aquestes relacions poden ser arbitràriament complexes, en funció de les dimensions de la xarxa.
- Tenen capacitat d'adaptació. Poden modificar tant el valor dels paràmetres, per adaptar-se a les dades, com fins i tot l'estructura de la xarxa, eliminant neurones per millorar la generalització dels models.
- Hi ha la possibilitat d'implementar-les en dispositius VLSI, així com executar-les sobre dispositius com les GPU i, més recentment, les TPU.
- Intensives en dades. El seu rendiment millora molt quan hi ha disponibles moltes dades d'entrenament. En el context actual, d'una disponibilitat de dades cada vegada més gran, això les fa destacar.

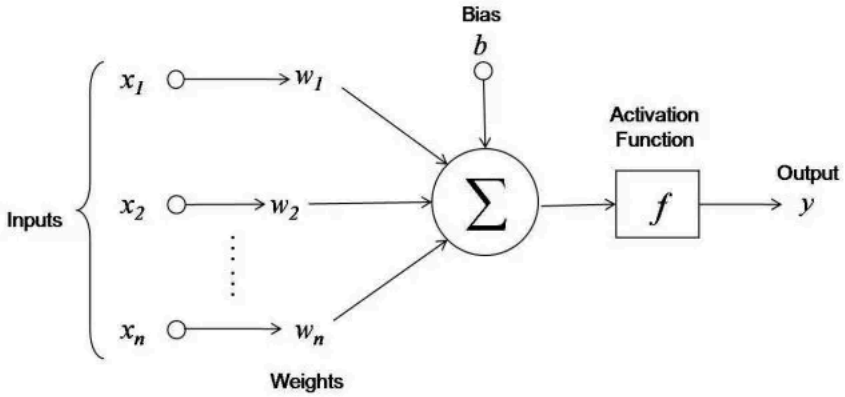
Així mateix, el desenvolupament tecnològic basat en xarxes neuronals, especialment profundes, també presenta diversos inconvenients.

- La interpretació del funcionament és difícil, cosa que dificulta l'**explicabilitat** dels resultats obtinguts amb aquests models, que es consideren de caixa negra. Sovint no podem determinar **per què** o **com** una xarxa neuronal ha pres la decisió que ha pres.
- **Biaix**. El fet d'estar entrenades sobre conjunts de dades que poden contenir biaixos de diversos tipus (racial, de gènere...) poden contribuir a la perpetuació i amplificació de discriminacions dels col·lectius afectats. Per exemple, en sistemes entrenats amb dades de recursos humans que reflecteixin una tendència a contractar homes blancs, és més difícil que contractin dones negres amb unes mateixes qualificacions i característiques professionals. De forma semblant, en sistemes d'avaluació de risc creditici, determinats col·lectius es poden veure afectats negativament de forma injustificada.
- L'entrenament de models cada vegada més grans, a causa de la gran quantitat de dades disponible, genera un **consum d'energia** molt gros.
- Només un grapat d'empreses pot assumir els costos milionaris associats a l'entrenament de grans sistemes d'aprenentatge profund a partir de dades massives. Això afavoreix una concentració de poder en poques mans, un **oligopoli tecnològic**.

1.3. Model de neurona artificial

La neurona artificial és la unitat bàsica de processament en una xarxa neuronal artificial. Els seus elements són els següents.

- Sinapsis o connexions, cadascuna d'elles amb un pes associat w.
- Un biaix b que s'afegirà a les entrades de la neurona
- Un sumador que integrarà la contribució de totes les entrades ponderades i el biaix.
- Una funció d'activació que pot limitar la sortida de la neurona dins un determinat marge.



1.4. Eclosió del Deep Learning

D'ençà que John McCarthy proposà el terme intel·ligència artificial a la dècada dels 1950 molts d'esdeveniments han contribuït a l'evolució de l'aprenentatge profund fins al moment en què som ara. El 1958 Frank Rosenblatt construí un prototipus de xarxa neuronal, el perceptró. A més, les idees fonamentals de les xarxes neuronals profundes per a visió per computador ja es coneixien a finals dels 1980. Els algorismes principals de Deep Learning per a sèries temporals com les xarxes LSTM (Long-Short Term Memory) es desenvoluparen el 1997, per posar alguns exemples. Aleshores, què ha fet possible ara l'eclosió de l'aprenentatge profund?

Seguint l'exposició de Jordi Torres al seu llibre Python Deep Learning, assenyalarem quatre grans factors que expliquen l'auge actual de l'aprenentatge profund.

- La supercomputació
- Les dades, combustible de la intel·ligència artificial
- La computació al núvol
- Una comunitat de recerca i desenvolupament oberta i col·laborativa

Capacitat computacional

La tasca d'entrenar xarxes d'aprenentatge profund demana una gran quantitat de computació i, sovint, usen el mateix tipus d'operacions matricials que les aplicacions intensives en càlcul numèric. Per tant, les aplicacions Deep Learning funcionen molt bé en sistemes amb acceleradores com les GPU (Graphic Processing Units).

Per això, a partir de 2012 i fins a 2014, els investigadors en aprenentatge profund començaren a usar sistemes amb GPU. A més, l'avantatge és que aquests algorismes escalen perfectament quan es pot posar més d'una GPU en un node. Això vol dir que si duplicam el nombre de GPU augmenta gairebé el doble la velocitat del procés d'aprenentatge dels models.

La gran capacitat computacional disponible permeté avançar i poder dissenyar xarxes neuronals cada vegada més i més complexes, i tornar a requerir més capacitat de computació que la que podia oferir un servidor amb múltiples GPU. Per això, a partir de 2014, per accelerar encara més el càlcul, es va començar a distribuir la càrrega entre múltiples màquines amb diverses GPU connectades per una xarxa.

A partir de 2016, a més, començaren a aparèixer xips de processament especialment pensats per a algorismes d'aprenentatge profund. Per exemple, el 2016 Google auntià que havia construït un processador dedicat anomenat *Tensor Processing Unit* (TPU). Des d'aleshores, ja s'han desenvolupat diverses versions de TPU.

Les dades, matèria primera per a la intel·ligència artificial

A més de la computació, la qualitat i la disponibilitat de les dades han estat factors clau en el desenvolupament de l'aprenentatge profund.

En general, la intel·ligència artificial necessita grans conjunts de dades per a l'entrenament dels models. La creació i disponibilitat de dades ha crescut exponencialment gràcies a la gran reducció de cost i a l'augment de fiabilitat en la generació de dades: fotos digitals, sensors més barats i precisos, etc.

A més de la disponibilitat de dades deguda a Internet, els recursos de dades especialitzats per a Deep Learning han facilitat el progrés de l'àrea. Moltes bases de dades obertes han impulsat el desenvolupament ràpid d'algorismes d'intel·ligència artificial. Un exemple n'és ImageNet, una base de dades de més de deu milions d'imatges etiquetades a mà. El que fa ImageNet especial no és tant la seva mida, sinó la competició anual que es realitzava, una gran motivació per als equips de recerca.

Mentre que els primers anys les propostes es basaven en algorismes de visió per computador tradicionals, el 2012 Alex Krizhevsky usà una xarxa neuronal Deep Learning, ara coneguda com a AlexNet, que va reduir la taxa d'error a menys de la meitat dels resultats de l'època. El 2015, l'algorisme guanyador ja estava a l'alçada de les capacitats humanes, i actualment els algorismes d'aprenentatge profund superen amb escreix les taxes d'error que tenen les persones.

Altres bases de dades populars són MNIST, CIFAR, STL o IMDB.

També és important mencionar com a recurs Kaggle, una plataforma que allotja competicions d'anàlisi de dades on companyies i investigadors aporten les seves dades, mentre que equips d'arreu del món competeixen per crear els millors models de predicció o classificació.

Computació al núvol

I què passa si una empresa no disposa de tota aquesta capacitat de computació? La intel·ligència artificial fins ara ha estat en mans de les grans companyies de tecnologia com ara Amazon, Baidu, Google o Microsoft, a més d'algunes noves empreses que disposaven d'aquesta capacitat. Per a molts d'altres negocis i parts de l'economia, els sistemes d'intel·ligència artificial fins ara han estat massa costosos i massa difícils d'implementar per complet.

Però ara entram en una altra era de distribució de la computació, i les empreses poden disposar d'accés a grans centres de processament de dades amb centenars de milers de servidors, en el que es coneix com a Cloud Computing.

El Cloud Computing ha revolucionat la indústria mitjançant la distribució de la computació i ha canviat completament la manera d'operar dels negocis. Les petites empreses tenen ara una gran oportunitat, ja que no poden construir aquestes infrastructures pel seu compte, però hi poden accedir mitjançant la computació al núvol.

Els proveïdors de Cloud ofereixen el que es coneix com AI-as-a-Service, serveis d'intel·ligència artificial que poden entrellaçar-se i treballar amb aplicacions internes de les empreses mitjançant API REST.

Es tracta d'un servei en què es paga per ús; això és disruptiu perquè permet als desenvolupadors de programari usar i posar qualsevol algorisme d'intel·ligència artificial en producció ràpidament.

Amazon, Microsoft, Google i IBM estan liderant els serveis AI-as-a-Service, que permeten des d'entrenaments fins a posades en producció de forma ràpida. Totes les capacitats importants de les aplicacions d'intel·ligència, des de l'experimentació fins a l'optimització, es poden trobar com a serveis al núvol en aquestes plataformes.

Col·laboració global amb codi obert

Fa alguns anys, per treballar amb Deep Learning feia falta conèixer llenguatges com ara C++ o CUDA; avui en dia, amb unes nocions de Python n'hi ha prou. Això ha estat possible gràcies al gran nombre de *frameworks* de codi obert que han anat apareixent, com ara Keras. Aquests *frameworks* faciliten enormement la creació i entrenament dels models, alhora que permeten d'abstraure les peculiaritats del maquinari al dissenyador de l'algorisme.

Podem destacar TensorFlow, Keras i PyTorch com els frameworks més rellevants actualment.

TensorFlow es va desenvolupar a Google Brain, per facilitar la recerca en Machine Learning i accelerar la transició d'un prototip d'investigació a un sistema de producció.

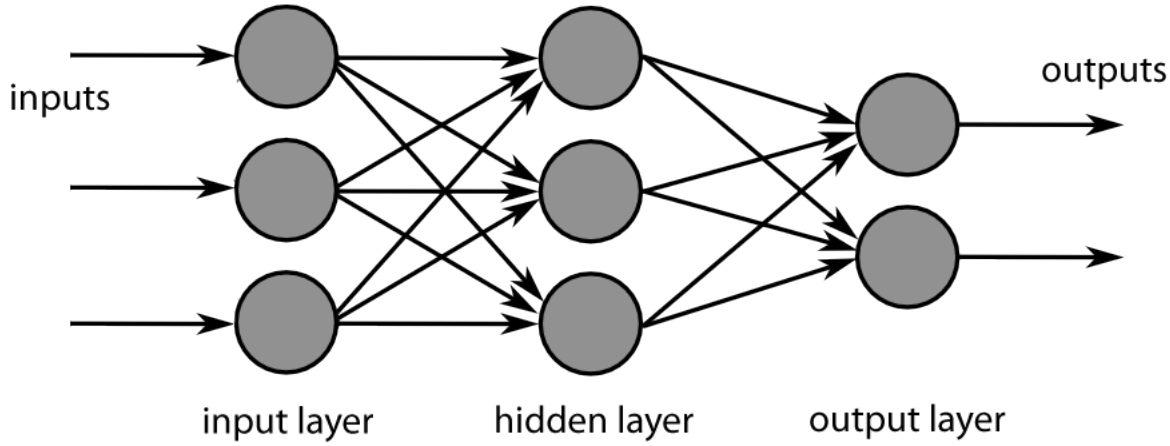
Keras té una API d'alt nivell per a xarxes neuronals, cosa que el fa l'entorn perfecte per iniciar-se en aquest tema. El codi s'especifica en Python. Keras en principi es pot executar sobre tres entorns destacats: TensorFlow, CNTK o Theano. Ara bé, TensorFlow ha adoptat Keras com la seva API principal, a més d'incorporar el creador de Keras, François Chollet. Això fa pensar que Keras es limitarà a TensorFlow en un futur.

PyTorch és un entorn de Machine Learning implementat en C, usant OpenMP i CUDA per aprofitar infrastructures altament paral·leles. És basat en Python i desenvolupat per Facebook. És un entorn popular en aquest camp de recerca, ja que permet molta flexibilitat en la construcció de xarxes neuronals i té tensors dinàmics, entre d'altres característiques.

2. Arquitectures

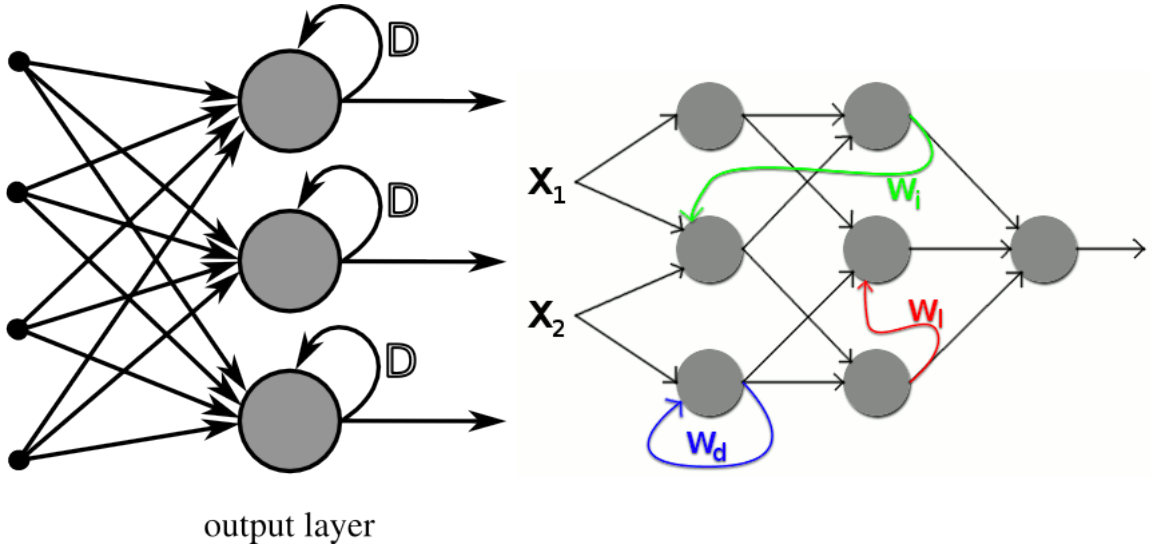
Presentam a continuació algunes arquitectures importants de xarxes neuronals.

- **Xarxes neuronals directes**, *feedforward*. Les connexions entre els estats no presenten cap cicle recurrent o de realimentació.

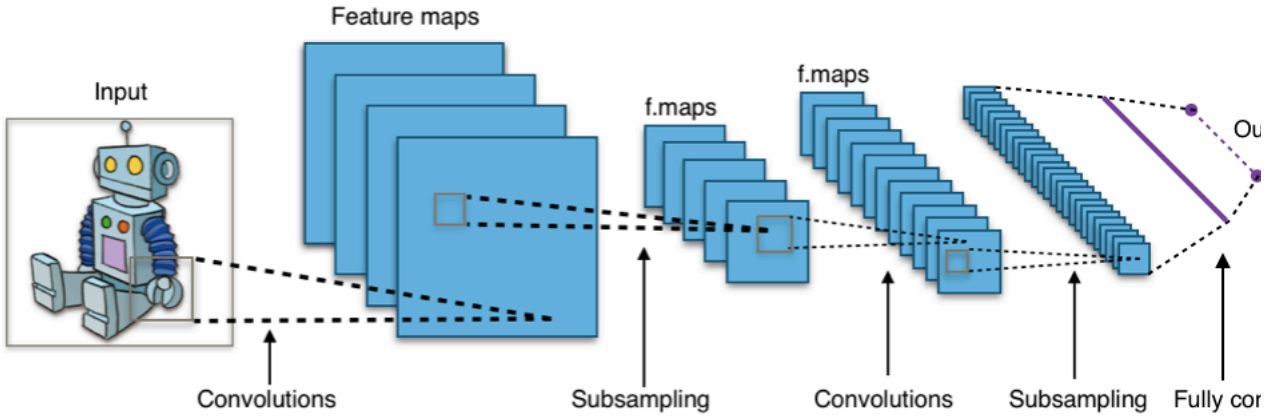


Aquesta arquitectura és la que usam en el present lliurament.

- **Xarxes neuronals recurrents** (*RNN*, *Recurrent Neural Networks*). Les connexions entre els estats presenten un o més cicles recurrents. Aquests cicles de realimentació fan la funció de memòria interna del sistema.



- **LSTM**, *Long Short-Term Memory*. Són un cas particular de RNN, molt usades i eficaces en problemes de modelatge de senyals temporals, per exemple en reconeixement de la parla. RNN, LSTM i també una variant anomenada GRU les veurem al lliurament de Deep Learning per a NLP de Models d'Intel·ligència Artificial.
- **Xarxes neuronals convolucionals** (*Convolutional Neural Networks*, *CNN*). Són molt usades en reconeixement d'imatges. Les usarem al lliurament 6è de Programació d'Intel·ligència Artificial.



2.1. Xarxes neuronals directes

Feedforward

2.2. Xarxes neuronals recurrents

- RNN
- LSTM
- GRU

2.3. Xarxes neuronals convolucionals

Convolutional Neural Networks

- Yann LeCun: LeNet
- Alex Khrizevsky (with Geoffrey Hinton)
- Turing Award

2.4. Transformers

- Catàleg de Xavier Amatriain
- Illustrated Transformer
- Arbre de família

3. Entrenament d'una xarxa neuronal

En aquest apartat d'entrenament de xarxes neuronals veurem els elements bàsics a partir dels quals aprenen els sistemes. Començarem amb una visió intuïtiva de l'aprenentatge per acabar descrivint els components principals del procés.

No és necessari saber de memòria totes les operacions implicades en l'entrenament de les xarxes. Sí que és important, però, entendre que darrere els mètodes que les llibreries ens ofereixen i utilitzam en una sola línia de codi, hi ha tota una seqüència de càlculs basats en idees matemàtiques ben desenvolupades.

3.1. Procés d'aprenentatge

Una **xarxa neuronal** és un model que mira de relacionar una entrada amb una etiqueta. Per exemple, aquesta entrada pot ser la imatge d'un dígit manuscrit i l'etiqueta és el dígit corresponent, entre zero i nou. En aquest cas, l'aplicació és de classificació. En aplicacions de regressió, en canvi, l'etiqueta serà un valor continu, com per exemple quan realitzam la predicció del preu d'un habitatge en funció de la seva superfície i la renda mitjana disponible de les unitats familiars del barri.

L'aprenentatge, en una xarxa neuronal, consisteix a determinar els paràmetres òptims (pesos w i biaixos b) observant moltes mostres i la seva etiqueta corresponent, durant el procés d'entrenament. Una capa de neurones, aplica a les seves dades d'entrada (les sortides de la capa anterior) els pesos i biaixos que tenen emmagatzemats les seves neurones. Són els paràmetres de la capa. El procés d'aprenentatges consisteix en trobar valors adequats d'aquests paràmetres.

Com s'aconsegueix l'aprenentatge d'aquests paràmetres? Són uns paràmetres entrenables, que s'inicialitzen per defecte amb valors aleatoris. A partir d'aquí, el seu valor es va ajustant gradualment. El resultat serà que aquests paràmetres contenen la informació apresa per la xarxa quan ha estat exposada a les dades d'entrenament. D'aquesta forma, **aprendre** significa trobar un conjunt de valors per als paràmetres de totes les capes de la xarxa. Amb aquests valors, la xarxa neuronal relacionarà correctament les mostres d'entrada amb les seves etiquetes corresponents.

Ara bé, una xarxa neuronal pot contenir milions i milions de paràmetres. Per poder trobar un valor adequat per a tots ells, s'ha de tenir en compte la influència que exerceixen els uns sobre els altres. Per exemple, a la petita xarxa de dues entrades que calculava la funció OR, vàrem trobar que una solució venia donada pels pesos 1 i el biaix $1/2$. Si modificàssim el valor d'algun d'aquests paràmetres, els altres també haurien de canviar per mantenir la funcionalitat de la xarxa.

Per poder regular la variació dels paràmetres, hem d'observar la sortida de la xarxa neuronal, mesurant la distància entre el resultat obtingut i el resultat desitjat. Aquesta és la tasca de la **funció de pèrdua** (*loss function* en anglès). La funció de pèrdua pren les prediccions de la xarxa i el valor veritable, que esperaríem que produís la xarxa, i calcula l'error comès en cada mostra d'entrada concreta.

Aquesta mesura de l'error s'usa com a senyal de retroalimentació del sistema per ajustar el valor dels paràmetres. Aquest ajust s'ha de realitzar en la direcció de reducció de l'error del conjunt de les mostres.

Aquest ajust és la tasca de l'**optimitzador**, que implementa la **retropropagació** (*backpropagation*) de l'error, des de la capa de sortida cap a les capes internes, per optimitzar els paràmetres.

En general, s'assignen inicialment valors aleatoris als paràmetres de la xarxa. Per això, l'error calculat és alt. Poc a poc, amb cada mostra d'entrada que la xarxa processa, els pesos s'ajusten una mica en la direcció correcta i l'error calculat va davallant progressivament. Aquest és el cicle d'entrenament, que es repeteix un nombre suficient de vegades per produir valors de paràmetres per minimitzar el valor de la funció de pèrdues.

Després d'aquesta descripció general, vegem amb més detall cada part del procés.

Aprenentatge iteratiu

Entrenar la xarxa neuronal és un procés d'**anar i tornar** per les capes de neurones. L'acció d'anar cap endavant, per obtenir a partir de l'entrada de la xarxa la sortida, li direm *forward propagation*. L'acció de tornar, a partir de la capa de sortida cap a la d'entrada, transmetent cap enrera la informació necessària per actualitzar els paràmetres, li direm *backward propagation*.

A la primera fase, *forward propagation*, s'exposa la xarxa a les dades d'entrenament i aquestes dades travessen tota la xarxa neuronal per calcular finalment les etiquetes o prediccions. A mesura que les dades travessen la xarxa, cada capa aplica la seva transformació corresponent, aplicant pesos i biaixos seguit de la funció d'activació.

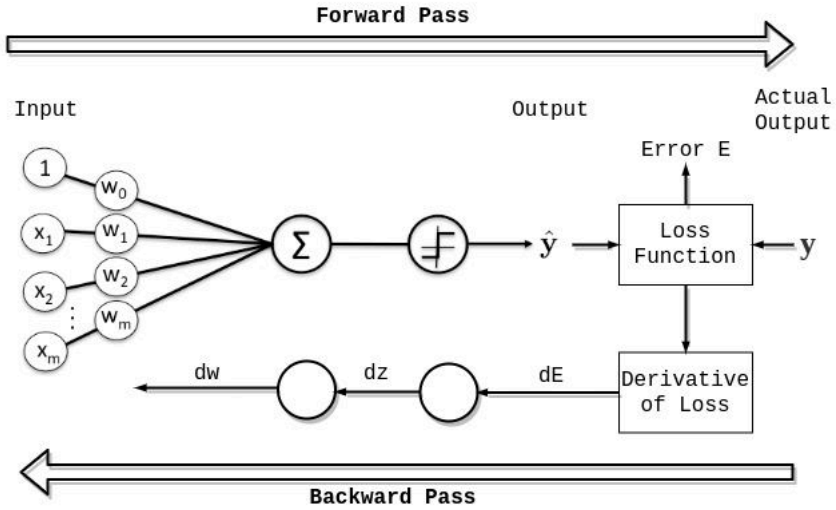
A continuació, usam una **funció de pèrdua** (*loss*) per calcular l'error d'estimació. Això mesura com de bo ha estat el nostre resultat de predicció en relació amb el resultat correcte. Pel fet que som en un entorn d'aprenentatge supervisat, disposam de l'etiqueta que ens indica el valor esperat.

Idealment, ens interessa que el nostre error calculat sigui zero, és a dir, sense divergència entre l'estimació i el valor esperat. Això s'aconsegueix a mesura que s'entrena el model, que anirà ajustant els pesos de les interconnexions de les neurones amb la informació que es va propagant cap enrere (retropropagació, *backpropagation*) començant per la capa de sortida.

A l'inici de la retropropagació, partint de la capa de sortida, que és la capa final de la xarxa, la informació de quant d'error s'ha comès es propaga cap a totes les neurones de la capa oculta que contribueixen directament a la capa de sortida. No obstant això, les neurones d'aquesta capa oculta només reben una fracció del senyal total de l'error, que es basa en la

contribució relativa que hagi aportat cada neurona a la sortida original d'acord amb els pesos. Aquest procés de propagació es repeteix, capa a capa, cap enrere, fins que totes les neurones de totes les capes de la xarxa han rebut un senyal de l'error comès.

Una vegada que s'ha propagat cap enrere aquesta informació calculada per la funció de pèrdua, podem ajustar els pesos de les connexions entre neurones. D'aquesta forma aconseguim que l'error calculat es reduesqui la propera vegada que tornem a usar la xarxa per realitzar una predicció. Per aconseguir-ho, usam una tècnica anomenata **descens del gradient** (*gradient descent* en anglès). Aquesta tècnica va ajustant els pesos en petits increments amb l'ajuda del càlcul de la derivada (o gradient) de la funció de pèrdua. Això permet a la xarxa avançar en la direcció adequada cap al mínim. Això es va fent en general en lots de dades, i no d'una en una.



Com a resum, l'algorisme d'aprenentatge consisteix en les passes següents.

1. Començar amb uns valors (sovint aleatoris) per als paràmetres de la xarxa (pesos i biaixos).
2. Prendre un conjunt d'exemples de les dades d'entrada (lot) i passar-lo per la xarxa per obtenir la seva predicció.
3. Comparar aquesta predicció obtinguda amb els valors d'etiquetes esperats i calcular-hi l'error comès mitjançant la funció de pèrdua.
4. Propagar cap enrere aquest error perquè arribi a tots els paràmetres que formen el model de la xarxa neuronal.
5. Usar aquesta informació propagada per actualitzar (amb l'algorisme de descens del gradient) els paràmetres de la xarxa neuronal, de forma que es vagi reduint progressivament l'error calculat.
6. Iterar les passes anteriors fins a aconseguir un bon model.

Elements clau de la retropropagació

Hem presentat la retropropagació (backpropagation) com un mètode per modificar els paràmetres de la xarxa neuronal en la direcció adequada. Després d'aplicar la funció de pèrdua, i de calcular el terme d'error, els paràmetres de la xarxa s'ajusten en ordre invers (de darrere cap a davant) amb un algorisme d'optimització que té en compte justament aquest terme d'error calculat.

A Keras comptam amb el mètode `compile()` per especificar la configuració de l'aprenentatge. Per exemple, podem usar el codi següent.

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

S'hi passen tres arguments: una funció de pèrdua (argument *loss*), un optimitzador (argument *optimizer*) i la llista de mètriques (argument *metrics*) que usarem per avaluar el model. En realitat, hi ha més arguments, que es poden consultar a https://keras.io/api/models/model/#compile. Només són obligatoris la funció de pèrdua i l'optimitzador. Abans de veure'ls amb més detall, repassem l'algorisme d'optimització per descens de gradient.

3.2. Descens del gradient

En **Deep Learning** (en **Machine Learning** en general) s'utilitza una optimització iterativa anomenada descens del gradient (*Gradient Descent* en anglès). Aquest algorisme ajusta gradualment els paràmetres del model per minimitzar la funció de cost sobre el conjunt de dades d'entrenament. Tot seguit presentam aquest algorisme i també algunes de les seves variants:

1. Descens del gradient en lots (*Batch Gradient Descent*)
2. Descens del gradient en minilots (*Minibatch Gradient Descent*)
3. Descens del gradient estocàstic (*Stochastic Gradient Descent*)

Descens de gradient bàsic

El descens de gradient és un algorisme d'optimització genèric capaç de trobar solucions òptimes a un ventall ampli de problemes. La idea general és ajustar paràmetres de forma iterativa per minimitzar una funció de pèrdua. El descens del gradient aprofita el fet que totes les operacions utilitzades a la xarxa neuronal són diferenciables i calculen el gradient de la funció de pèrdua respecte dels paràmetres de la xarxa neuronal. Aquest fet permet moure els paràmetres en la direcció oposada al gradient, de forma que l'error es redueixi.

Una comparació que pot servir per mostrar com funciona el descens de gradient és la d'un excursionista perdut dalt d'una muntanya, amb una boira densa que no permeti veure-hi gaire enfora. Només nota el pendent del terra sota els seus peus, i el que pot fer és anar davallant en la direcció de més pendent, fins a arribar al fons de la vall. Això fa l'algorisme de descens del gradient: mesura el gradient local de la funció de pèrdua respecte del vector de paràmetres, i va en la direcció del gradient descendent. Una vegada que s'arriba a un punt en què el gradient és zero, hem arribat a un mínim (local, si més no).

El descens del gradient usa la primera derivada (gradient) de la funció de pèrdua quan actualitza els paràmetres. El gradient dona el pendent de la funció en aquest punt. El procés consisteix a encadenar la derivada de la funció de pèrdua amb les derivades de cada capa de la xarxa neuronal. A la pràctica, una xarxa neuronal consisteix en moltes operacions de tensors encadenades. Cada una d'elles coneix la seva derivada. Per això les funcions d'activació de les neurones també han de ser derivables. Aquesta cadena de funcions es pot derivar precisament aplicant l'anomenada regla de la cadena del càlcul matemàtic.

En realitat l'aplicació de la regla de la cadena és la base de l'algorisme de *backpropagation*, perquè la seva tasca principal és fer la diferenciació en mode invers (*reverse-mode differentiation*). La retropropagació comença amb el valor final de pèrdua i avança cap enrere des de les capes finals de la xarxa cap a les capes inicials, aplicant la regla de la cadena per calcular la contribució de cada paràmetre en el valor de pèrdua, mitjançant la derivació simbòlica.

IMPORTANT

Quan usam entorns com per exemple TensorFlow no farà falta que implementem l'algorisme de *backpropagation*. Com a màxim, ens pot ser útil cridar alguna funció que ens retorni els gradients. Per això no val la pena entrar en més detalls de la matemàtica de la retropropagació. Quedem-nos amb la idea que l'optimització està basada en gradients.

A cada iteració, una vegada que totes les neurones tenen el valor del gradient de la funció de pèrdua que els correspon, s'actualitzen els valors dels paràmetres en sentit contrari al que indica el gradient. El gradient, en realitat, sempre apunta en el sentit que s'incrementa el valor de la funció de pèrdua. Per tant, usant l'oposat, el negatiu del gradient aconseguim el sentit en què es redueix la funció de pèrdua.

Tipus de descens del gradient

Amb quina freqüència s'ajusten els valors dels paràmetres? Aplicam l'algorisme d'ajust a cada mostra del conjunt de dades d'entrada? O aplicam l'ajust amb tot el conjunt d'entrada a cada iteració?

En el primer cas, parlam de **descens del gradient estocàstic** (SGD, de l'anglès *Stochastic Gradient Descent*), quan s'estima el gradient a partir de l'error observat en cada mostra de l'entrenament. Se'n diu estocàstic perquè cada dada s'extreu a l'atzar, aleatòriament.

En el segon cas, parlam de descens del gradien en lots (*Batch Gradient Descent* en anglès). Usam tot el conjunt de dades d'entrenament en cada pas de l'algorisme d'optimització, que calcula l'error amb la funció de pèrdua.

Tanmateix, si les dades d'entrenament estan ben distribuïdes, un petit subconjunt ens ha de donar una idea prou bona del gradient. Per ventura no se n'obté la millor estimació possible, però és més ràpid i, pel fet que estam iterant, aquesta aproximació pot ser suficient. Per això s'usa sovint una tercera opció, que ajusta els paràmetres a un subconjunt de mostres del conjunt d'entrenament, coneguda com **descens del gradient en minilots** (*Mini Batch Gradient Descent* en anglès).

Aquesta opció sol ser millor que la de descens del gradient estocàstic i calen més pocs càlculs per actualitzar els paràmetres de la xarxa neuronal. A més, el càlcul simultani del gradient per a molts d'exemples d'entrada es pot obtenir operant amb matrius, un càlcul que es pot implementar de forma molt eficient en paral·lel, mitjançant GPU.

ALERTA

Encara que **Batch Gradient Descent** significa tot el conjunt de dades d'entrenament de cop, en un sol lot, el paràmetre **batch_size** que s'usa al codi significa un subconjunt, per tant, un **minibatch**.

La majoria d'aplicacions usen el descens de gradient estocàstic amb un subconjunt de mostres del conjunt de dades d'entrenament, que anomenarem lot (*batch* en anglès). Per assegurar que s'usen totes les daes, el que es fa és repartir les dades d'entrenament en diversos subconjunts d'una mida determinada. Aleshores, es pren el primer lot, es passa per la xarxa, es calcula el gradient de la funció de pèrdua i s'actualitzen els paràmetres de la xarxa neuronal, successivament fins al darrer lot.

SGD és molt bo d'implementar en Keras. Al mètode `compile()` s'indica que l'optimitzador és `sgd`, i al mètode `fit()` s'especifica la mida del lot (batch) que s'agafarà a cada iteració dins l'argument `batch_size`.

```
model.fit(X_train, y_train, epochs=5, batch_size=100)
```

En aquest exemple de codi, estam dividint les nostres dades en lots de mida 100 amb l'argument `batch_size`. Amb l'argument `epochs` indicam quantes de vegades realitzam aquest procés damunt totes les dades.

3.3. Funció de pèrdua

Fa falta una funció de pèrdua (*loss function* en anglès) per guiar el procés d'entrenament de la xarxa, i per quantificar com de prop està una determinada xarxa neuronal del seu ideal mentre està en el procés d'entrenament.

A la pàgina del manual de Keras https://keras.io/losses podem trobar informació de les funcions de pèrdua de Keras, que estan totes disponibles en el mòdul `tf.keras.losses`. La tria de la funció de pèrdua ha de coincidir amb el problema específic de modelatge que tractem, ja sigui regressió o classificació. A més, la configuració de la capa de sortida també ha de ser adequada per a la funció de pèrdua triada.

Per exemple, per reconèixer els dígits de MNIST, s'usa categorical_crossentropy com a funció de pèrdua, ja que la sortida ha de ser en format categòric. La variable de sortida ha de prendre un valor entre els 10 possibles. Aquesta serà la funció de pèrdua adequada quan tenguem un problema de classificació en diverses classes. Hi ha d'haver el mateix nombre de neurones a la darrera capa que classes. A més, la sortida de la capa final ha de passar a través d'una funció d'activació **softmax** perquè cada node generi un valor de probabilitat entre 0 i 1.

En canvi, quan tenim una tasca de classificació binària, una de les funcions de pèrdua que se sol utilitzar és binary_crossentropy. Si usam aquesta funció de pèrdua, només necessitam un node a la darrera capa per classificar les dades en dues classes. El valor de sortida ha de passar a través d'una funció d'activació sigmoid, amb el rang de sortida entre 0 i 1.

En aplicacions de regressió, una de les funcions de pèrdua que es poden usar és Mean Squared Error. Aquesta pèrdua es calcula prenent la mitjana del quadrat de la diferència entre els valors reals i els pronosticats.

Resumim en una taula les funcions de pèrdua mencionades.

Tipus de problema	Funció d'activació	Funció de pèrdua
Classificació múltiple	softmax	categorical_crossentropy
Regressió	linear	mse
Classificació binària	sigmoid	binary_cross_entropy

Taula: *Funcions d'activació i de pèrdues segons el tipus de problema.*

3.4. Optimitzadors

L'optimitzador és un altre dels arguments que fan falta en el mètode `compile()`. A TensorFlow, amb l'API de Keras es poden usar uns altres optimitzadors, a més del SGD: RMSprop, AdaGrad, Adadelta, Adam, Adamax, Nadam. Podem consultar la informació sobre cada un a la documentació https://keras.io/optimizers. Aquests optimitzadors són variants o optimitzacions de l'algorisme de descens del gradient.

Una opció molt habitual actualment és utilitzar l'optimitzador Adam amb una taxa d'aprenentatge de 0.001 (https://keras.io/api/optimizers/adam).

4. Paràmetres i hiperparàmetres

Motivació

Suposem que hem desenvolupat un model per al problema del reconeixement de dígits MNIST amb els hiperparàmetres proposats al quadern de Colab. Hem obtingut un resultat al voltant del 90%. Com podem millorar aquesta precisió?

Haurem de modificar els paràmetres i hiperparàmetres de la xarxa. Per exemple modificant la **funció d'activació** de **sigmoid** a **ReLu**, podem obtenir un 2% addicional amb el mateix temps aproximat de càlcul.

```
model.add(Dense(10, activation='relu', input_shape=(784,)))
```

També es pot augmentar el nombre d'**epochs**, les vegades que s'utilitzen totes les dades d'entrenament. O augmentar les neurones en algunes capes, o afegir capes. Per exemple, podem usar 512 neurones a la capa intermèdia.

```
model.add(Dense(512,activation='relu', input_shape=(784,)))
```

Podem veure, amb el mètode summary(), que augmenta el nombre de paràmetres, i que el temps d'execució és bastant superior. Amb aquest model la precisió arriba al 94%. I si augmentam a 20 epochs ja aconseguim una precisió del 96%.

Paràmetres i hiperparàmetres

Quina és la diferència entre un paràmetre del model i un hiperparàmetre? Els paràmetres del model són interns a la xarxa neuronal, per exemple, els pesos i biaixos de les neurones.

En canvi, els hiperparàmetres són paràmetres externs al model, establerts a la configuració de la xarxa neuronal. Per exemple, són hiperparàmetres el tipus de funció d'activació o la mida de lot (*batch_size*) usada en l'entrenament.

Grups d'hiperparàmetres

Podem distingir dos grans grups d'hiperparàmetres.

- De l'estructura i la topologia de la xarxa neuronal: nombre de capes, nombre de neurones per capa, funcions d'activació, inicialització dels pesos, etc.
- De l'algorisme d'aprenentatge: *epochs*, *batch_size*, *learning rate*, etc.

Epochs

El nombre d'èpoques (*epochs*) indica quantes vegades les dades d'entrenament passen per la xarxa neuronal durant l'entrenamnet. És important determinar un valor adequat d'aquest paràmetre. Un nombre massa alt d'èpoques provoca que el model s'ajusti massa a les dades i tengui problemes de generalització als conjunts de validació i prova. També pot causar problemes de *vanishing gradient* i *exploding gradient*, és a dir, que els gradients prenguin valors massa petits o massa grossos. Un valor massa petit pot limitar el potencial del model, amb un entrenament insuficient que no permeti realitzar bones prediccions.

Una bona pista per trobar aquest valor òptim d'*epochs* és anar incrementant fins que la mètrica de precisió amb les dades de validació comenci a davallar, fins i tot encara que la precisió amb les dades d'entrenament continuï creixent.

Batch size

Podem repartir les dades d'entrenament en lots (*batches*) per entrenar la xarxa més àgilment que si en cada *epoch* s'utilitzàs tot el conjunt d'entrenament. A Keras, el *batch_size* és un argument del mètode `fit()`.

Learning rate i *learning rate decay*

El vector de gradient té una direcció i una magnitud. Els algorismes de descens del gradient multipliquen la magnitud del gradient per un escalar, anomenat taxa d'aprenentatge o *learning rate* (també de vegades *step size*) per determinar el següent valor del paràmetre.

Per exemple, si la magnitud del gradient és 1.5 i el *learning rate* és 0.01, el següent valor del paràmetre serà a 1.5×0.01 del punt anterior.

El valor adequat d'aquest hiperparàmetre depèn molt del problema. En general un *learning rate* massa gran provoca canvis massa grans en els paràmetres i ens podem botar el mínim de la funció de pèrdues, anant a posicions completament aleatòries.

En canvi, si la taxa d'aprenentatge és massa petita, els avanços seran molt petits, i la convergència molt lenta.

Un bon sistema per garantir velocitat de l'entrenament i convergència al mínim és reduir el *learning rate* a mesura que avança l'entrenament. L'hiperparàmetre que governa aquesta reducció és el *learning rate decay*.

El valor de la taxa d'aprenentatge depèn també de l'optimitzador utilitzat. Per exemple, amb l'optimitzador **sgd**, 0.1 pot ser un bon valor, mentre que amb Adam funciona millor 0.001.

Inicialització del pes dels paràmetres

La inicialització del pes dels paràmetres no és exactament un hiperparàmetre, però també té una gran influència en l'entrenament de les xarxes neuronals. És recomanable inicialitzar els pesos i biaixos amb uns valors aleatoris i petits, per rompre la simetria entre diferents neurones. Si dues neurones tenguessin exactament els mateixos valors inicials als paràmetres, sempre seguirien tenint valors iguals entre elles, i no podrien aprendre característiques diferents de les entrades.

4.1. Funcions d'activació

La funció d'activació propaga cap endavant la sortida d'una neurona. Aquesta sortida la reben les neurones de la següent capa a la qual hi ha connectada aquesta neurona, fins a la capa de sortida inclosa. La funció d'activació serveix per introduir la no linealitat en les capacitats de modelatge de la xarxa. A continuació veurem les funcions d'activació més usades actualment. Totes elles es poden usar en una capa de Keras.

Podem aprofundir aquesta informació consultant la referència de la pàgina de Keras https://www.tensorflow.org/api_docs/python/tf/keras/activations

Els tipus de funció d'activació que desenvoluparem són els següents.

- Lineal
- Sigmoide
- Tanh
- Softmax
- ReLU

En els apartats següents, z és l'entrada de la funció d'activació, la combinació lineal de les entrades de la neurona ponderades pels pesos més el biaix de la neurona. Si les activacions de les neurones de la capa anterior són a_i, per a i des d' 1 fins a N,

$$z = w_1 a_1 + w_2 a_2 + \ldots + w_N a_N + b$$

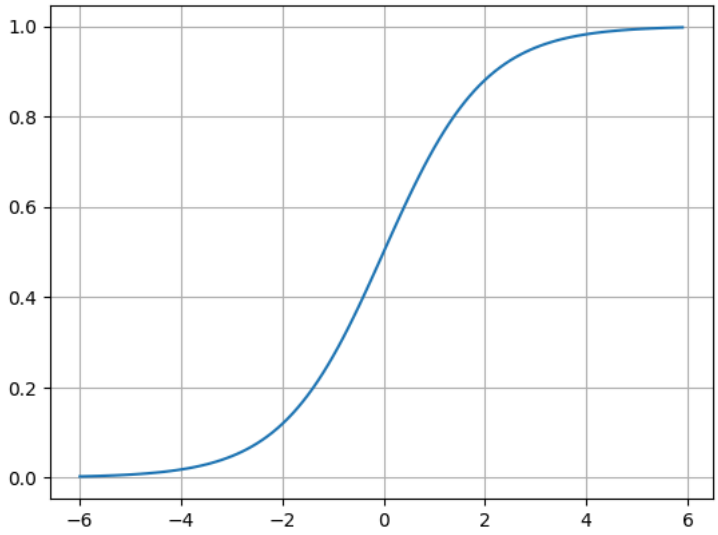
Observem que a la capa inicial no hi ha activacions de la capa anterior, sinó directament les entrades de la xarxa x_i

Escriurem l'activació de la neurona actual com a

$$a = g(z)$$

4.2. Sigmoide

Sigmoide. Les funcions amb forma de S en general reben el nom de sigmoides. La que s'utilitza més habitualment és la funció logística. Per això sovint quan es diu sigmoide ens referim a la funció logística.



Imatge: *Sigmoide logística*

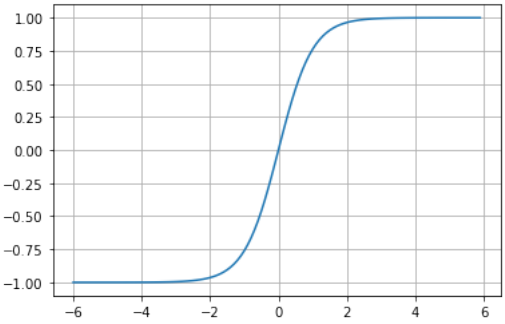
$$ g(z) = \frac{1}{1+e^{-z}} $$

És una aproximació suau a la funció esglaó. Pren valors molt propers a zero per a entrades negatives de mòdul gran i valors molt propers a la unitat per a entrades positives grans. Té l'avantatge que és derivable, i això fa que s'hi pugui aplicar l'algorisme de **descens de gradient**. S'utilitza a la capa de sortida en les aplicacions de classificació binària.

Una única neurona amb aquesta funció d'activació és equivalent a la regressió logística.

4.3. Tanh

La tangent hiperbòlica, tanh, té qualitativament la mateixa forma que la sigmoide logística. Però si les observam atentament, veurem que, mentre que la funció logística tendeix als valors \(0\) i \(1 \) als extrems, la tanh tendeix a \(-1\) i \(1 \). Per tant, és una aproximació suau a la funció signe.



\[g(z) = \tanh(z) = \frac{\sinh(z)}{\cosh{z}} = \frac{e^z-e^{-z}}{e^{z}+e^{z}}\]

La tangent hiperbòlica s'usava a les capes internes de les xarxes neuronals profundes abans de l'ús generalitzat de la ReLU.

4.4. Softmax

La funció softmax generalitza la regressió logística. Mentre que la regressió logística permet una classificació binària, softmax s'aplica a classificació en múltiples classes. Aquesta funció d'activació es trobarà a la capa de sortida de classificadors en múltiples classes. Softmax normalitza les sortides de les N neurones de la capa de sortida del classificador, d'acord amb l'expressió següent.

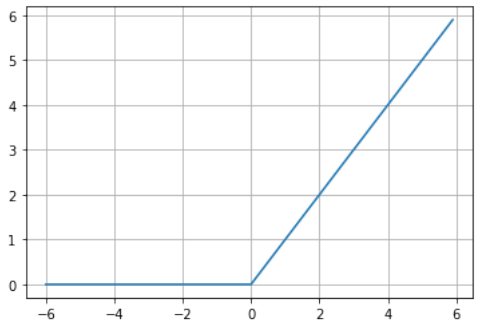
$$ g_i(z) = \frac{e^{z_i}}{\sum_{j=1}^K e^{z_j}} $$

Aquí, l'índex i es refereix a la neurona de sortida i-èsima, hi ha K neurones i z és un vector que conté les entrades z_i de totes aquestes neurones de la capa de sortida.

La funció exponencial té la utilitat de transformar un rang d'entrada entre $-\infty$ i ∞ a un rang entre 0 i ∞. I l'aplicació de la funció softmax garanteix per construcció que la suma de tots aquests valors positius de la sortida de la xarxa és la unitat. Per tant, es pot interpretar com la probabilitat que l'entrada pertany a cada una de les K classes del classificador.

4.5. ReLU

- **ReLU**, Rectified Linear Unit. Unitat lineal rectificada. Deixa passar les entrades positives mentre que limita a zero les negatives. És una no-linealitat senzilla que es va començar a usar recentment i va accelerar molt el procés d'entrenament de les xarxes neuronals profundes, per la seva simplicitat. A més, té propietats matemàtiques interessants, com la independència de l'escala. S'utilitzen a les capes internes de les xarxes, tant en aplicacions de regressió com de classificació.



$$ g(z)=max(0,z) $$

5. Llibreries

Python ofereix diverses llibreries potents per implementar xarxes neuronals. Entre les més populars es troben **TensorFlow**, **Keras** i **PyTorch**.

TensorFlow

TensorFlow és una llibreria de codi obert desenvolupada per Google per a computació numèrica i aprenentatge automàtic. És molt utilitzada en aplicacions d'aprenentatge profund, gràcies a la seva capacitat per construir i entrenar xarxes neuronals complexes.

Característiques clau:

- **Ecosistema ampli:** TensorFlow inclou eines per a tot el flux de treball de l'aprenentatge automàtic, des del preprocesament de dades fins al desplegament de models.
- **Suport per a GPU i TPU:** Pot aprofitar la potència del hardware avançat per a càlculs paral·lels.
- **TensorFlow Lite i TensorFlow.js:** Permeten portar models a dispositius mòbils i aplicacions web, respectivament.
- **Flexibilitat:** Ofereix una API de baix nivell per personalitzar completament els models.

Exemple bàsic:

```
1  1  import tensorflow as tf
2
3  # Creació d'un tensor
4  x = tf.constant([[1, 2], [3, 4]])
5  print(x)
```

Keras

Keras és una API d'alt nivell integrada a TensorFlow des de la versió 2.0, però també pot funcionar com una biblioteca independent. Va ser dissenyada per ser fàcil d'utilitzar, modular i extensible

Característiques clau:

- **Simplicitat:** Ideal per a principiants gràcies a la seva sintaxi clara i intuïtiva.
- **Flexibilitat:** Permet construir models seqüencials o amb arquitectures més complexes utilitzant el model funcional.
- **Integració:** Amb TensorFlow, ofereix una experiència completa d'entrenament i optimització de models.
- **Extensió:** Compatible amb múltiples backends (TensorFlow, Theano, etc., en versions anteriors).

Exemple bàsic:

```
1  1  from tensorflow.keras.models import Sequential
2  from tensorflow.keras.layers import Dense
3
4  # Creació d'un model seqüencial
5  model = Sequential([
6      Dense(32, activation='relu', input_shape=(784,)),
7      Dense(10, activation='softmax')
8  ])
```

PyTorch

PyTorch és una llibreria de codi obert desenvolupada per Facebook que s'ha convertit en una opció popular per a investigadors i desenvolupadors gràcies a la seva flexibilitat i fàcil depuració.

Característiques clau:

- **Dinàmica:** Utilitza un gràfic computacional dinàmic que permet modificar la topologia de la xarxa durant l'execució.
- **Alt rendiment:** Ofereix suport per a operacions en GPU i una API de baix nivell per a personalitzacions avançades.
- **Comunitat:** Té un fort suport en investigació acadèmica i és popular en projectes d'avantguarda.
- **Integració amb Python:** L'ús de PyTorch és similar al treball amb NumPy, cosa que fa que sigui intuïtiu per a desenvolupadors.

Exemple bàsic:

```
1  1
```

```python
import torch
import torch.nn as nn

# Creació d'un tensor
x = torch.tensor([[1.0, 2.0], [3.0, 4.0]])

# Exemple d'una capa lineal
layer = nn.Linear(2, 3)
output = layer(x)
print(output)
```

A les pàgines següents donam dos exemples d'implementació de xarxes neuronals amb Keras

Els enllaços a les tres llibreries són els següents.

- TensorFlow
- Keras
- PyTorch

6. Xarxes neuronals en Keras

En aquest apartat, construirem de forma pràctica un exemple de reconeixement de dígits manuscrits de la base de dades MNIST. L'exemple s'organitza en els apartats següents.

1. Precàrrega de les dades a Keras.
2. Preprocessament de les dades d'entrada.
3. Definició del model
4. Configuració de l'aprenentatge
5. Entrenament del model
6. Avaluació del model

6.1. Precàrrega

Keras disposa de diversos conjunts de dades precarregades. Això facilita molt l'inici en aplicacions d'aprenentatge profund, ja que ens podem centrar directament en el modelatge de les dades.

Un d'aquests conjunts disponibles és MNIST, que es troba precarregat en forma de quatre arrays NumPy i es pot obtenir amb el següent codi.

```
import tensorflow as tf
mnist = tf.keras.datasets.mnist

(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

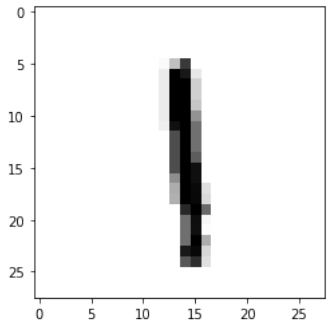
x_train i y_train formen el conjunt d'entrenament, mentre que x_test i y_test són el conjunt de prova. En aplicacions en què s'optimitzen també els hiperparàmetres dels models s'han de distingir tres subconjunts de les dades: entrenament per a l'ajust dels paràmetres, validació per a l'optimització dels hiperparàmetres i test o prova per a l'avaluació de la capacitat de generalització. De moment, en aquesta primera aplicació, quedem-nos en els dos conjunts principals: entrenament i prova.

Les imatges estan codificades com a arrays NumPy i les seves etiquetes corresponents (*labels*), de 0 a 9.

Si volem inspeccionar quins valors hem carregat, podem triar qualsevol de les imatges del conjunt. Per exemple, prenguem la vuitena, amb el codi següent.

```
import matplotlib.pyplot as plt
plt.imshow(x_train[8], cmap=plt.cm.binary)
```

Obtenim la imatge següent.



Comprovem que la seva etiqueta és 1 amb el codi següent.

```
print(y_train[8])
```

Per conèixer millor les dades, podem inspeccionar-les, obtenint la seva dimensionalitat, forma i tipus de dades.

Al quadern de Colab amb l'exemple complet podem executar el codi necessari i obtenir els resultats corresponents.

A continuació, anem a la passa segona: preprocessament.

6.2. Preprocessament

En general, sempre hi sol haver un preprocessament de les dades amb l'objectiu d'adaptar-les a una format que permeti explotar-les més bé pel model. Alguns dels preprocessaments més habituals en deep learning són la vectorització, normalització o extracció de característiques.

Aquestes imatges de MNIST tenen una mida de 28x28 píxels, una matriu de valors de 8 bits, per tant, entre 0 i 255 de tipus uint8. Per facilitar la convergència de l'entrenament de les xarxes neuronals els escalarem de forma que el rang quedi entre 0 i 1. Abans, però, per no perdre resolució, haurem de passar els enters a valors de coma flotant.

```
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train/=255
x_test/=255
```

Una altra transformació que convé fer de vegades és canviar la forma dels tensors sense canviar les dades. Per exemple, en aquest cas, passarem de matrius de 28x28 píxels a un sol array de 784 valors. Ho podem fer amb la funció `numpy.reshape()`.

```
x_train = x_train.reshape(60000, 784)
x_test = x_test.reshape(10000,784)
```

Després de canviar la forma dels tensors, podem comprovar que efectivament la forma que tenen és la desitjada.

```
print(x_train.shape)
print(x_test.shape)
```

A més, també hem de transformar les etiquetes de cada dada d'entrada, que ara són nombres entre 0 i 9. La xarxa neuronal tendrà deu neurones a la capa de sortida, i per a cada entrada només s'haurà d'activar la neurona corresponent. Per això, ens interessa transformar aquests nombres utilitzant la codificació **one-hot**. Per exemple, si una etiqueta és 2, el vector one-hot corresponent serà $[0,0,1,0,0,0,0,0,0,0]$, amb un únic 1 a la posició 2.

Amb la funció `to_categorical` de `tensorflow.keras.utils` realitzam aquesta operació, com il·lustra el codi del quadern a la secció 2.

6.3. Definició

L'estructura de dades principal de Keras és la classe Sequential, que permet la creació d'una xarxa neuronal.

El model en Keras es representa com una seqüència de capes. Cada una d'elles va destil·lant gradualment les dades d'entrada fins a obtenir la sortida. Construirem per començar un model molt simple de tan sols dues capes, amb el codi següent.

```
model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(10,activation='sigmoid',input_shape=(784,)))
model.add(tf.keras.layers.Dense(10,activation='softmax'))
```

La xarxa s'ha definit com una seqüència de dues capes denses, completament connectades. Totes les neurones de la primera capa estan connectades amb totes les neurones de la segona capa, d'acord amb els pesos corresponents. La primera capa és la capa d'entrada, per això té \(784\) valors d'entrada, que venen dels \(24x24\) píxels de les imatges.

A cada capa indicam el nombre de nodes (neurones) que té i la funció d'activació que s'hi aplicarà.

La segona capa té una funció d'activació softmax de 10 neurones, per tant tornarà un vector de 10 valors de probabilitat. Aquestes sortides indicaran la probabilitat que una entrada pertany a la classe corresponent a cada neurona.

Una vegada definit el model, és molt útil emprar el mètode `summary()` per veure la seva estructura, com es pot veure a l'apartat 3 del quadern Colab.

6.4. Configuració

Després de construir el model, hem de definir com serà el procés d'aprenentatge, mitjançant el mètode `compile()`.

El primer dels arguments que passarem a aquest mètode és la funció de pèrdues (*loss function*), que usarem per avaluar l'error entre les sortides calculades i les desitjades.

També s'especifica l'optimitzador, que determina l'algorisme amb què es van obtenint les actualitzacions dels paràmetres de la xarxa.

Finalment, hem d'indicar la mètrica, la mesura en base a la qual avaluarem el model. En aquest exemple, només tendrem en compte la precisió (*accuracy*).

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

6.5. Entrenament

Una vegada que tenim el model definit i hem configurat l'aprenentatge, ja podem entrenar-lo, utilitzant el metode `fit()`, que vol dir ajust.

```
model.fit(x_train, y_train, epochs=5)
```

En els dos primers arguments indicam les dades en forma d'array NumPy amb què s'entrenarà el model. Amb el paràmetre epochs indicam quantes vegades s'usaran totes les dades durant el procés d'aprenentatge.

Aquest és el mètode que pot arribar a tardar més temps a executar-se. Keras ens permetrà veure com avança mitjantçant l'argument verbose (per defecte, igual a 1).

6.6. Avaluació

Després de l'entrenament, podem avaluar com es comporta el model amb les dades de prova, que no hauran estat usades durant l'entrenament. D'aquesta forma, tendrem una mesura de la capacitat de generalització del model.

```
test_loss, test_acc = model.evaluate(x_test, y_test)
```

En aquest exemple, obtenim una precisió lleugerament superior al 80%. A la tasca realitzarem alguns canvis al model i l'entrenament amb l'objectiu de millorar aquest resultat.

6.7. Predicció

El darrer bloc de codi de l'exemple il·lustra la classificació d'una imatge concreta.

S'hi pot veure com en el vector de deu posicions que conté la probabilitat de cada classe, el valor més gran és a la posició que correspon al dígit.

7. Exemple: Fashion-MNIST

En aquest apartat veurem la creació d'un model en Keras aplicant una darrera l'altra totes les passes a un altre conjunt de dades. El conjunt de dades serà Fashion-MNIST, que ja està precarregat dins Keras, i això ens permetrà centrar-nos en les passes necessàries per al model.

Fashion-MNIST és un conjunt de dades de les imatges dels articles de Zalando, una botiga de moda en línia d'Alemanya que ven roba i sabates. Conté 70000 imatges en escala de grisos en 10 categories. Les imatges mostren peces de roba en baixa resolució, 28x28 píxels. S'usen 60000 imatges per entrenar la xarxa i 10000 imatges per avaluar la precisió amb què la xarxa aprèn a classificar les imatges.

Al quadern de Colab Fashion-MNIST tenim desenvolupat tot el procés, amb les passes següents.

1. Preparació de les dades
2. Definició del model
3. Configuració del model
4. Entrenament del model
5. Avaluació i millora del model
6. Ús del model per realitzar prediccions
7. Millora del model

8. Experts

Per tancar aquest lliurament, presentem diversos investigadors destacats en l'àrea de les xarxes neuronals i l'aprenentatge profund.

- **Geoffrey Hinton**, **Yoshua Bengio** i **Yann LeCun** varen rebre el premi Turing l'any 2018, pel seu treball pioner en l'aprenentatge profund.
- **Fei Fei Li** impulsà la base de dades ImageNet, fonamental en el desenvolupament de la visió per computador.
- **Alex Khrizevsky**, estudiant de Geoffrey Hinton, és el primer autor de la publicació en què es presenta l'arquitectura coneguda com AlexNet. Fou la primera xarxa convolucional profunda que suposà un punt d'inflexió en la classificació d'imatges.
- **Ilya Sutskever**, també estudiant de Hinton, és una persona clau en el desenvolupament tècnic dels grans models del llenguatge a OpenAI.
- **Andrej Karpathy** ha fet una gran tasca de divulgació de l'aprenentatge profund publicant exemples de codi i ha estat responsable de visió artificial a Tesla.
- **Ian Goodfellow** és l'autor de les xarxes generatives antagonistes, que són a la base de la producció amb un gran realisme de cares de persones que no existeixen a la realitat.
- **Jürgen Schmidhuber** és responsable de molts de desenvolupaments en xarxes neuronals, entre els quals destaquen les LSTM.
- **Andrew Ng** té un gran impacte a la docència, a través de deeplearning.ai, i al desenvolupament d'empreses, a través del seu AI Fund.