

POTENCIAL DE LES XARXES NEURONALS ARTIFICIALS

De què és capaç una IA?

Biel Altés Grifoll

DESEMBRE 2020, BARCELONA

Tutora: María Plazas



RESUM

Ha arribat un punt en què les xarxes neuronals artificials estan molt presents a la nostra vida diària, tot i que molt poca gent sàpiga que existeixen, i menys com funcionen. Així doncs, l'objectiu principal d'aquest treball ha estat fer recerca en aquest àmbit i donar a conèixer al lector, de la manera més simple possible, el funcionament d'aquestes xarxes.

El treball va començar a través dels seus fonaments matemàtics. D'aquesta manera es va fer palès que per a poder entendre bé les eines punteres, abans s'havia de conèixer el seu rerefons. Seguidament, mitjançant TensorFlow, una API puntera en la matèria usada per professionals, es van poder veure aplicacions reals i els seus rendiments en ordinadors estàndard.

Gràcies a les dades obtingudes, s'ha pogut constatar el potencial real d'aquestes xarxes i s'ha entès perquè estan tan esteses al nostre dia a dia, i com ho estaran encara més en un futur pròxim.

Paraules clau: intel·ligència artificial (IA), perceptró, retropropagació, TensorFlow, Keras, python

ABSTRACT

It has come to a point where artificial neural networks are really present in our daily lives, even if very few people are aware of their existence and, much less, of how they work. Therefore, the main objective of this paper has been to do research in this field and to make it known to the reader, in the easiest way possible, how this algorithms work.

The research started through their mathematical foundations. This way, we clearly saw that, in order to understand these advanced tools properly, we had to start learning their background. After that, making use of TensorFlow, a cutting-edge API used by professionals, it was possible to see real applications and their performances on ordinary computers.

Thanks to the collected data, it has been possible to confirm the real potential of these structures, so we could understand why they are so used in our everyday lives and how they will be used even more in the near future.

Keywords: artificial intelligence (AI), perceptron, backpropagation, TensorFlow, Keras, python

Sumari

1 Introducció al treball.....	4
2 Inicialització a les xarxes neuronals artificials.....	5
2.1 Comprensió del funcionament d'una xarxa neuronal.....	5
2.2 Parts del funcionament.....	6
2.3 La neurona primitiva, el perceptró.....	6
2.3.1 Fórmula per a l'aprenentatge segons Frank Rosenblatt.....	10
2.3.2 Els límits del perceptró.....	11
2.4 El perceptró multicapa.....	12
2.4.1 Esquema d'un perceptró multicapa.....	15
2.4.2 Paràmetres i hiperparàmetres.....	15
2.4.3 Simbologia.....	16
2.4.4 Sigmoid.....	17
2.4.5 Softmax.....	19
2.4.6 Teorema de l'aproximació universal.....	19
2.4.7 Output d'un perceptró multicapa.....	20
2.5 Retropropagació.....	22
2.5.1 Funció cost (error quadràtic mitjà).....	22
2.5.2 Descens del gradient.....	23
2.6 Implementació d'un perceptró multicapa.....	31
2.6.1 Programació.....	31
2.6.1.1 def __init__(self).....	32
2.6.1.2 def execucio(self).....	34
2.6.1.3 def sortida(self).....	35
2.6.1.4 def retropropagacio(self).....	36
2.6.1.5 def guardar(self).....	37
2.6.2 Proves.....	38
2.6.2.1 Aprendent quin número es més gran (0-1000).....	39
2.6.3 Conclusions del perceptró multicapa amb python.....	40
3 Deep learning amb TensorFlow i Keras.....	41
3.1 Què és TensorFlow?.....	41
3.2 Què és Keras?.....	41
3.3 Estructura del codi TensorFlow.....	42
3.4 Paràmetres i hiperparàmetres d'un model.....	42
3.5 Estructures neuronals.....	43
3.5.1 MLP (multi layer perceptron).....	43
3.5.1.1 Boston housing price dataset.....	43
3.5.2 CNN (Convolutional Neural Networks).....	45
3.5.2.1 Chest X-Ray Images (Pneumonia) dataset.....	47
3.5.3 RNN (Recurrent Neural Networks).....	51
3.5.3.1 LSTM.....	52
3.5.3.2 Wikipedia-sentences dataset.....	53
4 Projecte final: IA conduceix en videojocs.....	59
4.1 Programa de creació del dataset.....	61
4.2 Programa de definició i entrenament de la xarxa neuronal.....	65
4.3 Programa d'execució en temps real.....	67
4.4 Proves.....	69
4.4.1 Proves primer model.....	69
4.4.2 Millores al model.....	69
4.4.3 Proves segon model.....	70
5 Últims avenços en IA arreu del món.....	71
5.1 Sistemes de predicció: Algorisme de YouTube.....	71
5.2 Processament d'imatges: detecció i reconeixement.....	71
5.2.1 Autopilot.....	72
5.2.2 Sistemes de control i detecció de masses.....	72
5.3 Models d'NLP: GPT-3 i Blenderbot.....	72
5.3.1 GPT-3.....	73
5.3.2 BlenderBot.....	74
6 Conclusions.....	76
7 Bibliografia.....	77
7.1 Llibres.....	77
7.2 Documents del web.....	77
7.3 Pàgines del web.....	77

1 Introducció al treball

El meu treball de recerca es titula "Potencial de les xarxes neuronals artificials" i amb ell tinc la intenció de conscienciar al lector de tot el que es pot fer amb un tipus d'algorisme que tot i que és utilitzat per empreses d'arreu del món, és poc conegut pel ciutadà corrent, les xarxes neuronals.

Aquest és un tema de plena rellevància actual a causa de la por sostinguda de la pèrdua de capacitats de l'ésser humà envers als robots a l'hora de poder oferir serveis i, per tant, tenir una feina a la societat. És un tema que desperta un gran interès en mi, ja que és un moviment tecnològic que busca aconseguir facilitar la vida de les persones, i això sempre és part de l'evolució humana, que lluita any rere any i segle rere segle per tal de millorar i facilitar la nostra vida.

L'**objectiu** d'aquesta recerca és arribar a conculoure un treball amb 3 apartats principals, estructurats de la següent manera:

- Un aprofundiment en les matemàtiques que mouen tots aquests grans projectes, accompanyades d'una aplicació d'aquestes mateixes de la manera tan senzilla com sigui possible. Tractat en l'apartat: «Inicialització a les xarxes neuronals artificials».
- Utilitzar l'API de Google “TensorFlow 2.0” i la llibreria “Keras” per tal de facilitar la programació d'una xarxa neuronal artificial de certes dimensions. Tractat en l'apartat: «Deep learning amb TensorFlow i Keras».
- Amb els coneixements adquirits en l'apartat anterior programar una IA que condueixi en el popular simulador Assetto Corsa. Tractat en l'apartat: «Projecte final: IA conduceix en videojocs».
- Per a finalitzar, una petita vista ràpida a les últimes tecnologies potenciades per IA que més impacte tenen sobre les nostres vides per tal d'acabar de veure'n el potencial real. Tractat en l'apartat: «Últims avenços en IA arreu del món».

Personalment sempre m'ha agradat la tecnologia informàtica i tot el que l'envolta, però de vegades penso que l'usuari comú és massa poc rigorós a l'hora d'intentar estar al cas dels progressos tecnològics i de la seva comprensió. A parer meu, ser concient en aquests temes és un requeriment indispensable en una societat tan avançada tecnològicament parlant com en la que vivim. És per això que m'he proposat investigar sobre aquest tema. Cal avisar que es tracta d'un temari dens i tot i que les matemàtiques que es veuen no són del tot complicades, la seva aplicació pot resultar una mica enrevessada. També s'ha d'entendre que si mai s'ha vist un codi en python o algun altre llenguatge, serà difícil entendre tots els matisos de la part de la programació.

Finalment abans de començar, voldria agrair tant al Domènec Mollà com al Jordi Torres per haver-me guiat a definir el treball quan vaig començar-lo.

2 Inicialització a les xarxes neuronals artificials

En aquesta primera part del treball, ens introduirem en les xarxes neuronals artificials a partir de les seves estructures més bàsiques. D'aquesta manera, comprendrem el funcionament matemàtic i serà molt més fàcil posar-li cara al que farem posteriorment.

A continuació desenvoluparem els apartats que veurem en aquesta primera part:

- Comprensió del funcionament d'una xarxa neuronal: introducció a les xarxes neuronals a partir d'un símil amb les xarxes neuronals biològiques.
- Parts del funcionament: veurem les parts més importants de les xarxes neuronals artificials.
- La neurona primitiva, el perceptró: començarem amb l'estructura més simple de xarxa neuronal artificial.
- El perceptró multicapa: tractarem conceptes necessària per a la comprensió de les xarxes més complexes.
- Retropropagació: veurem l'algorisme necessari per a l'entrenament dels perceptrons multicapa.
- Implementació d'un perceptró multiaca: aplicació pràctica d'un perceptró mutlicapa.

2.1 Comprensió del funcionament d'una xarxa neuronal

Una xarxa neuronal artificial, com el seu nom indica, és una imitació en un entorn informàtic del funcionament de les neurones al cervell. Igual que les neurones es relacionen amb impulsos electromagnètics entre si amb la finalitat de prendre decisions, farem que a través d'unes entrades i un annexionat de neurones artificials, es calculin sortides que compleixin les nostres necessitats.

Volem que a partir d'un model que va modificant una sèrie de dades, aconseguir una sortida que concordi amb el que vulguem obtenir.

Amb això ens podem proposar de fer qualsevol cosa, des d'un simple programa que distingeixi números, fins a una xarxa que amb píxels com a entrada controli un cotxe autònom.

2.2 Parts del funcionament

Atès que aquestes dades que manipulen les entrades no es coneixen en un inici, el funcionament de tota xarxa neuronal artificial es pot diferenciar en dues parts:

- **Aprendentatge:**

El primer que ha de fer una xarxa neuronal és aprendre. Pot aprendre de moltes maneres, però bàsicament en aquesta fase la xarxa aprèn a partir d'informació de la qual se sap la sortida desitjada, calibrant així les dades preestablertes per a trobar patrons entre entrades i sortides i, en un futur, poder deduir sortides tot introduint unes entrades que no ha vist mai.

- **Inferència:**

Un cop la nostra xarxa neuronal ja ha passat amb èxit la fase d'aprenentatge, està preparada per a donar sortides a totes les entrades apreses i, a partir d'aquestes entrades, és capaç de deduir-ne de noves.

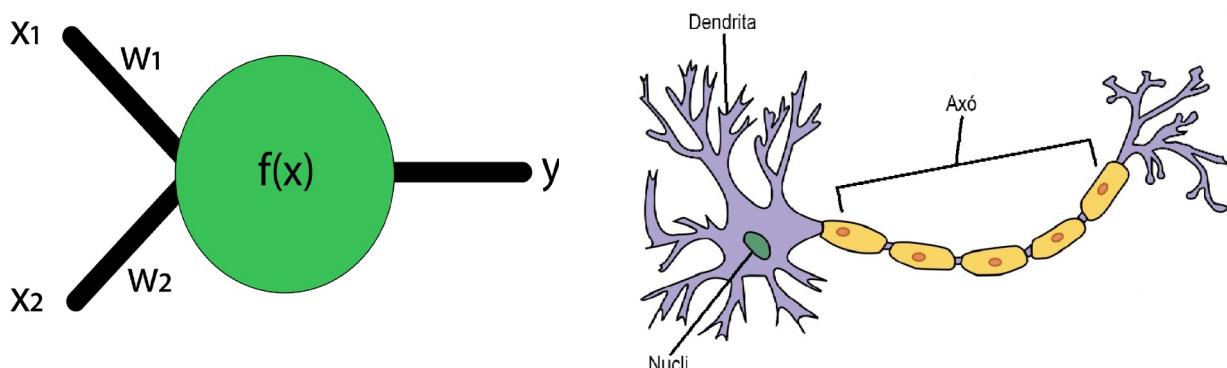
Un cop està entrenada la xarxa, es solen provar valors dels quals se sap la sortida per a poder calcular així el percentatge de predicció de la xarxa.

Aquestes parts no han d'estar separades necessàriament, si bé és veritat que en la majoria de casos ho estan, també hi ha algorismes que són capaços d'anar aprenent de dades noves a mesura que van deduint-ne els resultats.

2.3 La neurona primitiva, el perceptró

El perceptró està inspirat en una simple neurona. És la forma més bàsica de xarxa neuronal i una de les coses que això comporta és una limitació en diversos àmbits com veurem més endavant, però va ser el primer model de xarxa neuronal i per tant, inspiració per als següents.

Va ser inventat per Frank Rosenblatt (1928-1971) i està format per uns pesos, un llindar i el nucli del perceptró. A continuació podeu veure una foto de la representació gràfica d'un perceptró i una neurona al costat.



Imatge de wikipedia (modificada)¹

En una neurona, la dendrita rep els estímuls, el nucli intervé i es transmeten nous estímuls a les altres neurones a partir de l'axó.

Amb la mateixa idea, donades diverses entrades (x_1, x_2), les connectem a la neurona a partir d'un pes que adquireix un valor real qualsevol (w_1, w_2), i quan arriben a la neurona en forma de sumatori, depenent de si sobrepassen el llindar o no, la sortida (y) d'aquesta neurona serà un valor o un altre.

¹ <https://ca.wikipedia.org/wiki/Neurona>

Així doncs, imitant a una neurona real, tenim diverses entrades que connectem a la neurona mitjançant un pes. Com a sortida, aquesta neurona artificial, donarà el sumatori de totes les entrades multiplicades pels seus respectius pesos.

Aquests pesos són una part fonamental en les xarxes neuronals artificials, atès que són les dades que nosaltres podem modificar per a obtenir les sortides desitjades en cada moment. El llindar o el bias que veurem més endavant fan una funció similar.

D'aquesta manera, el que hem de fer és calibrar els pesos de tal manera que donades unes entrades, les sortides siguin les desitjades. Pel que fa al llindar, el qual es defineix com el grau d'inhibició de la neurona, el situarem nosaltres segons conveniència. La idea és que si les sortides que volem obtenir són majoritàriament 0, que el llindar sigui alt i si les sortides són majoritàriament 1, que el llindar sigui baix.

La sortida la calculem de la següent manera, essent n les entrades del perceptró:

$$y=1 \text{ si } \left(\sum_{j=1}^n x_j \cdot w_j > t \right)$$

$$y=0 \text{ si } \left(\sum_{j=1}^n x_j \cdot w_j \leq t \right)$$

Com es pot veure, multipliquem cada entrada (x) pel pes (w) per on passa, sumem els resultats i després l'avaluem respecte al llindar (t, de l'anglès threshold).

Bé, doncs ara que ja sabem com calcular una sortida i com funciona un perceptró, ja podem encomanar-li feines simples,.En aquest cas li ensenyarem la porta lògica AND.

X₁	X₂	Y
1	1	1
1	0	0
0	1	0
0	0	0

Llavors només haurem de trobar 2 pesos aleatoriament tals que es compleixi la sortida desitjada en cada moment, aquesta no és la millor manera, més endavant veurem com perfeccionar-ho.

Podem desenvolupar un petit programa amb python per a exemplificar-ho, establirem al llindar a 0,5, com mostrem a continuació.

```
import random

pesos = [random.uniform(0,1), random.uniform(0,1)]
llindar = 0.5
aprenent = True
bucles = 0
acerts = 0

taula = [[1,1,1], [1,0,0],[0,1,0],[0,0,0]]

while aprenent == True:
    for x in taula:
        if (x[0]*pesos[0]+x[1]*pesos[1]) > llindar:
            sortida_polaritzada = 1
            bucles += 1
        else:
            sortida_polaritzada = 0
            bucles += 1

        if sortida_polaritzada == x[2]:
            acerts += 1
        else:
            pesos = [random.uniform(0,1), random.uniform(0,1)]
            acerts = 0

    if acerts == 4:
        print(pesos)
        print(bucles -4)
        aprenent= False0
```

I quan l'executem obtenim:

```
[0.4765911580339237, 0.22047016991757284]
34
```

```
Process finished with exit code 0
```

En executar-lo veiem que és molt irregular, així com de vegades en té prou amb menys de 20 bucles, pot arribar a tardar-ne 90, això és degut al factor aleatori.

2.3.1 Fórmula per a l'aprenentatge segons Frank Rosenblatt

Per no haver d'anar provant pes a pes aleatoriament fins a trobar un resultat que satisfaci les nostres dades d'entrenament (cosa que és molt poc eficient), Frank Rosenblatt va idear un mecanisme per a poder anar una mica més ràpid en la nostra feina, és un mètode iteratiu que s'assembla molt al que farem servir quan tinguem una xarxa molt més gran. Així que cal entendre-ho bé com a fonament.

Comencem amb l'algorisme per a entrenar el nostre perceptró simple. La següent fórmula serveix per a trobar uns valors tals que l'error es vegi en el seu mínim.

Primer definim l'error:

$$\text{error} = \text{sortida desitjada} - \text{sortida real}$$

I posteriorment podem calcular el valor d'un pes (w_n) en la següent iteració a partir de l'entrada (x_{n-1}) que passa per aquell pes i el pes de la iteració anterior(w_{n-1}).

$$w_n = w_{n-1} + \alpha \cdot \text{error} \cdot x_{n-1}$$

Aquí podem veure un concepte que no s'havia descrit abans, la taxa d'aprenentatge, que normalment es simbolitza amb la lletra grega alfa (α). Aquesta taxa pren un valor, usualment al voltant de 0,1. Com més gran és aquest valor, més ràpid aprendrà el perceptró, però més possibilitats hi ha que se salti el mínim i la resposta no sigui acurada.

```
import random

pesos = [random.uniform(0,1), random.uniform(0,1)]
llindar = 0.5
aprenent = True
bucles = 0
acerts = 0

taula = [[1,1,1], [1,0,0],[0,1,0],[0,0,0]]

while aprenent == True:
    for x in taula:
        if (x[0]*pesos[0]+x[1]*pesos[1]) > llindar:
            sortida_polaritzada = 1
            bucles += 1
        else:
            sortida_polaritzada = 0
            bucles += 1

        if sortida_polaritzada == x[2]:
            acerts += 1
        else:
            error = x[2] - sortida_polaritzada
            pesos[0] += (0.1 * error * x[0])
            pesos[1] += (0.1 * error * x[1])
            acerts = 0

    if acerts == 4:
        print(pesos)
        print(bucles -4)
        aprenent= False
```

En executar-lo veiem patent la millora de rendiment i sobretot d'estabilitat, ja que se situa sempre entre unes 5 i 15 iteracions.

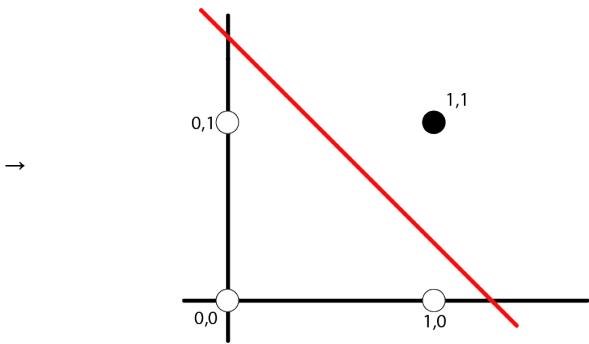
```
[0.44489187205137937, 0.420576042251777]
7
```

```
Process finished with exit code 0
```

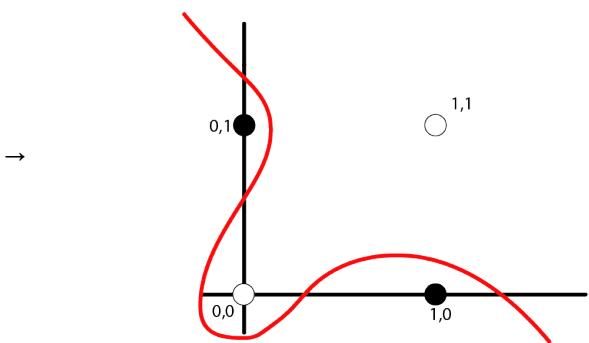
2.3.2 Els límits del perceptró

El perceptró, com hem dit abans va ser presentat per Frank Rosenblatt l'any 1959, però pocs anys després, Marvin Minsky i Seymour Papert demostraven en el seu llibre "Perceptrons" que un perceptró no és capaç d'aprendre una gran quantitat de problemes, ja que actua com a divisor lineal sobre el pla bidimensional, fent impossible per exemple d'aprendre la taula XOR i qualsevol altre problema que no sigui linealment divisible. Això ho podem veure millor amb un exemple com aquest:

AND		
1	1	1
1	0	0
0	1	0
0	0	0



XOR		
1	1	0
1	0	1
0	1	1
0	0	0



Com veiem en el gràfic, el perceptró no pot aprendre a separar entrades que requereixen més d'una recta per a classificar, o dit d'una altra manera, aquells que la disposició dels objectes sigui tal que no es puguin separar amb una recta. Amb la combinació de diversos perceptrons simples podien resoldre alguns problemes no lineals, però no existia un mecanisme per a calibrar els perceptrons que no es trobaven a l'entrada.

Si executem la nostra pràctica amb la taula XOR, veiem com el programa mai acaba amb la seva feina. Les conseqüències d'això van ser que tot i que es començava a tenir el hardware necessari per a provar una xarxa neuronal, no existien uns algorismes realment eficients. Això no va canviar fins a finals de la dècada dels vuitanta amb l'aparició de l'algorisme de retropropagació, que podia calibrar eficientment les neurones aquestes xarxes més complexes. Amb això es podia començar a parlar del perceptró multicapa.

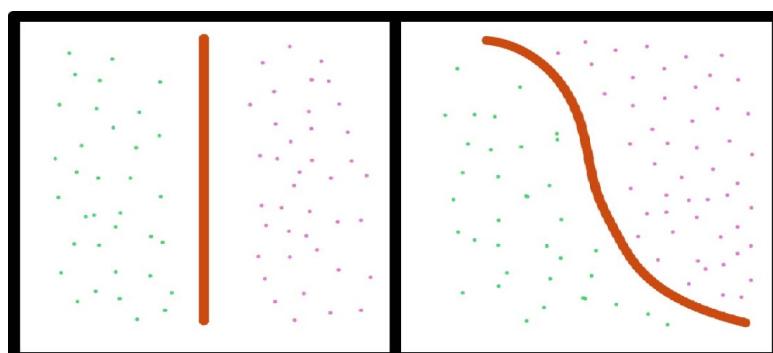
2.4 El perceptró multicapa

Un perceptró multicapa no deixa de ser la unió de diversos perceptrons simples els quals es classifiquen en diferents capes per a poder trobar solució a qualsevol mena d'entrada. Així doncs, la principal característica que no tenia el perceptró simple era la capacitat de poder resoldre problemes no divisibles linealment.

Recopilant, un perceptró multicapa és un conjunt de neurones connectades per columnes o capes. Cada neurona suma tots els valors provinents de la capa anterior i els introduceix a una funció, la sigmoide. El resultat de la funció l'envia a la següent capa per a repetir exactament el mateix procés.

Per a la implementació del perceptró multicapa, la part més important és l'algorisme d'aprenentatge, retropropagació.

D'aquesta manera, això no només serveix per a les portes lògiques que hem ensenyat abans, sinó que també pot amb problemes que no són divisibles linealment. En el següent possible cas es veuen dos situacions hipòtètiques. El primer es pot solucionar amb un perceptró simple, mentre que el segon requereix d'un perceptró multicapa.



En els següents punts s'explicarà el funcionament del perceptró, podreu veure que la part més senzilla és el funcionament del mateix perceptró, el qual no és res més que un graf que descriu una funció.

La part complicada del perceptró multicapa és l'aprenentatge. Ja veurem més endavant com funciona, però l'aprenentatge serà la part on minimitzar l'error que ens dona la nostra funció respecte als valors de sortida desitjats.

Una altra cosa digna de menció és que ja no farem servir llindars. En un perceptró multicapa es fa servir la negativa del llindar, el que en anglès s'anomena bias (b). Així doncs, el bias és com si agaféssim el llindar i el passéssim a l'altra banda de l'igual positivament, d'aquesta manera tenim:

Perceptró simple

$$y=1 \text{ si } \left(\sum_{j=1}^n x_j \cdot w_j > t \right)$$

$$y=0 \text{ si } \left(\sum_{j=1}^n x_j \cdot w_j \leq t \right)$$

x:entrada, w:pes, t:llindar

Perceptró multicapa

$$y=f\left(\sum_{j=1}^n x_j \cdot w_j + b\right)$$

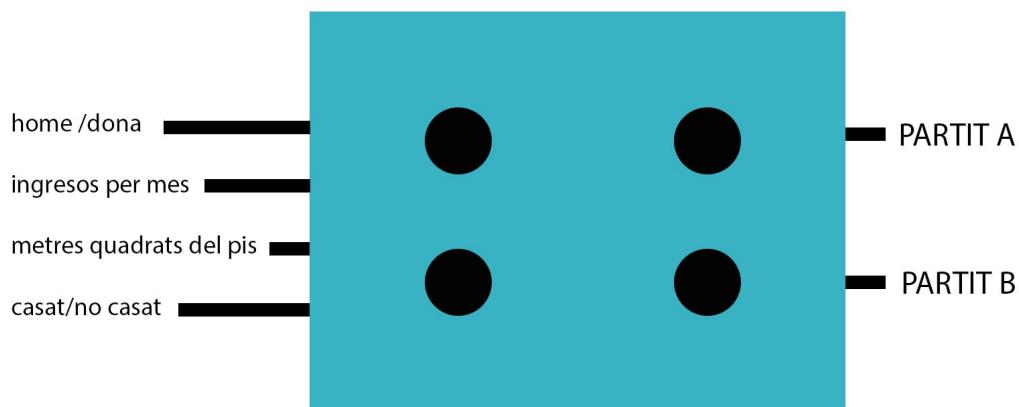
x:entrada, w:pes, b:bias

On aquesta f, representa la funció sigmoide, que ens polaritzarà els valors entre 0 i 1.

Una manera senzilla de veure el bias, és imaginant-lo com la n de l'equació de la recta ($y = mx + n$), que ens regula l'altura.

Per entendre el funcionament ens podem imaginar que el perceptró és una cosa física. Imaginem que hem comprat una màquina de la qual se'n diu que és capaç de dir a quin partit polític ha votat un ciutadà a partir de 4 dades, si és home o dona, el que cobra, els metres quadrats de la casa on viu i si està casat o no (aclarir que les dades que es veuen en aquest exemple són inventades amb la finalitat d'entendre la matèria).

Així doncs quan a nosaltres ens arriba la màquina, veiem una caixa amb 4 entrades, 2 sortides i 4 potenciómetres a la superfície.



Just agafem l'aparell, el primer que fem és introduir les nostres dades a veure si encerta i veiem que s'equivoca. En aquest moment llegim les instruccions a veure si trobem què passa. Comencem a llegir i trobem el problema, aquesta màquina no està calibrada, és a dir, el nostre percepçó està per entrenar. Juntament amb les instruccions ens inclouen una base de dades les quals són certes sobre persones reals. Ens indiquen que hem d'anar movent els 4 potenciòmetres fins que tots els resultats de la base de dades siguin correctes. Arribat a aquest punt la màquina ja serà capaç d'encertar resultats amb persones noves amb un elevat percentatge d'encerts.

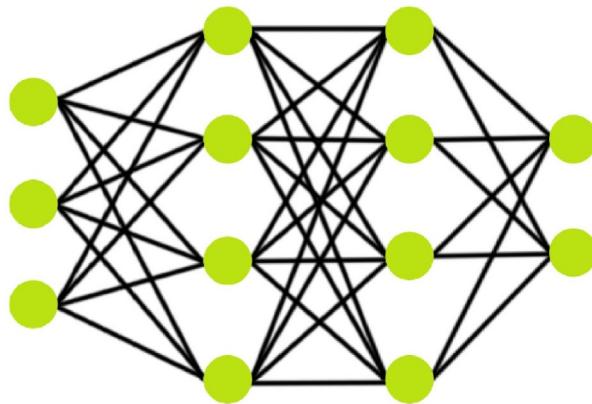
El que hem estat fent és calibrar el percepçó multicapa. En un entorn real, el nombre d'aquests potenciòmetres (el que anomenem pesos i bias) és molt superior i amb les dades d'entrada ens passa el mateix. Òbviament aquesta feina no pot ser feta per un humà i requereix moltíssima potència de càlcul en un ordinador si la xarxa és mitjanament gran. No cal ni dir, que aleatoriament com hem vist al principi, no es pot fer això.

A continuació veurem els fonaments matemàtics del percepçó multicapa, els organitzarem per:

- Esquema d'un percepçó multicapa: veurem la seva estructura.
- Paràmetres i hiperparàmetres: veurem els dos tipus de variables que intervenen en el percepçó multicapa.
- Simbologia: nomenclatura utilitzada en la descripció del percepçó multicapa.
- Sigmoide: funció que ens proporciona la sortida de cadascuna de les neurones (funció d'activació).
- Softmax: funció necessària per normalitzar les respostes quan hi ha més d'una sortida.
- Teorema de l'aproximació universal: teorema que proporciona la base tipus de problema no lineal.
- Output d'un percepçó multicapa: aprendrem a calcular la sortida d'un percepçó multicapa.

2.4.1 Esquema d'un perceptró multicapa

El perceptró multicapa no deixa de ser una estructura, per la qual fem passar dades i les modifiquem amb els pesos per a obtenir sortides desitjades, vist així es veu clar perquè la feina més gran és l'entrenament de la xarxa neuronal. En aquesta estructura tenim diverses capes de neurones les quals estan interconnectades només amb la capa anterior i la capa següent.



Les neurones de la capa d'entrada no actuen com a tal, ja que no efectuen cap operació, simplement introduceixen els valors.

Cada segment simbolitza un pes i cada neurona una funció. Recordem que cada neurona també té un bias variable assignat.

2.4.2 Paràmetres i hiperparàmetres

Quan parlem d'una xarxa neuronal artificial, hi ha dos tipus de dades necessàries per al seu funcionament, les anomenem paràmetres i hiperparàmetres.

Els paràmetres són totes aquelles dades que no estan predefinides sinó que es van ajustant segons l'algorisme d'aprenentatge. Aquí hi tenim tot el conjunt de pesos i bias.

Els hiperparàmetres són els valors que defineixen estructuralment una xarxa neuronal. Aquest tipus de dades seran preestablertes per nosaltres i tenen un impacte directe al rendiment del model. Per exemple aquí hi podem trobar la taxa d'aprenentatge o l'algorisme d'entrenament. D'aquestes dades en la primera part en parlarem poc, ja que el perceptró estarà programat per a funcionar amb uns donats hiperparàmetres i l'hauríem de modificar substancialment per a canviar-ne algun. De fet, només podrem variar la taxa d'aprenentatge, el nombre de capes i el nombre de neurones per capa. En canvi, quan treballem amb TensorFlow, se'n permetrà variar molts hiperparàmetres al nostre gust amb la finalitat d'aconseguir els resultats esperats de la nostra xarxa.

2.4.3 Simbologia

Hem de tenir en compte que tindrem molts pesos, moltes neurones i molts bias. Així doncs, hem de simbolitzar cada part del nostre percepçó d'una manera senzilla i eficaç. Ho farem de la següent manera.

- Nombre de neurones per capa:

El nombre de neurones per capa el simbolitzarem amb la lletra n i el subíndex de la capa on ens trobem.

- Neurones d'entrada:

Les neurones d'entrada es simbolitzen amb una x i un subíndex (i) començant a comptar de dalt a baix.

$$x_i$$

- Neurones de sortida:

Les neurones de sortida es simbolitzen amb una y i un subíndex (i) començant a comptar de dalt a baix.

$$y_i$$

- Neurones de capes ocultes:

Les neurones ocultes es simbolitzen amb una “z” la qual conté un superíndex que ens diu la capa a la qual pertany i un subíndex (i) que ens diu la posició que ocupa en aquella capa.

$$z_i^{(n)}$$

- Pesos

Els pesos els simbolitzarem amb la lletra “w” (weight), trobem un superíndex entre parèntesis que ens diu a la capa que pertany, i un subíndex que ens indica la neurona de la capa que surt (i) i seguidament la neurona a la que va (j) de la següent capa.

$$w_{i,j}^{(n)}$$

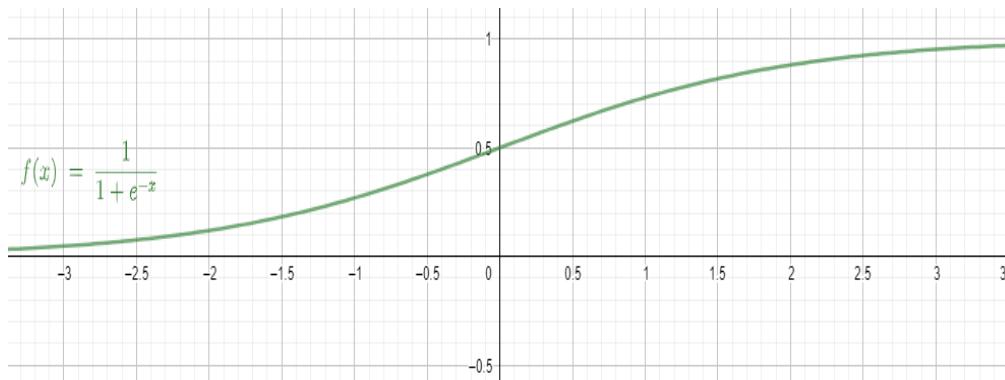
- Bias

Els bias els simbolitzarem amb la lletra “b” (bias), la qual conté un superíndex que ens diu la capa a la qual pertany i un subíndex que ens diu de quina neurona pertany, en cas que pertanyi a la capa de sortida, el superíndex serà y .

$$b_i^{(n)}$$

2.4.4 Sigmoide

La sortida de cada neurona la calcularem amb el que s'anomena funció d'activació. Aquesta s'utilitza per a polaritzar la sortida entre dues sortides, normalment 0 i 1 i trencar d'aquesta manera la linealitat de la xarxa. Nosaltres farem servir una que s'anomena sigmoide, que és molt utilitzada. Té la següent forma:



Així doncs per a entrades molt baixes ens donarà 0 i per a entrades molt altes ens donarà 1 com a resposta.

L'equació és la següent:

$$f(x) = (1 + e^{-x})^{-1}$$

Llavors tenint en compte que cada neurona ens donarà un sortida que podem anomenar $g(x)$, haurem d'avaluar la composta de $g(x)$ amb $f(x)$, o sigui:

$$(f \circ g)(x) \rightarrow f(g(x))$$

Bé, tot això l'únic que vol dir que donada una sortida d'una neurona, n'haurem de polaritzar els valors amb aquesta funció. És a dir, agafar el valor que ens doni la neurona i posar-lo com a variable a la sigmoide, la qual ens donarà un resultat entre el 0 i l'1. La corba ja l'hem observat anteriorment.

La sigmoide a més a més té una peculiaritat molt important que ens facilitarà molt les coses. El que buscarem més endavant és reduir l'error d'una funció, és a dir, minimitzar la diferència entre la sortida desitjada o la que ens dona. Una de les coses requerides per a aconseguir això és derivar, i la sigmoide té la peculiaritat que la seva derivada té la següent forma:

$$f(x)' = f(x) \cdot (1 - f(x))$$

Això es pot demostrar fàcilment seguint el següent procediment matemàtic:

Recordem que la regla de la cadena ens diu que per a derivar una funció, derivem primer la funció sencera i posteriorment el que ens trobem dins de la funció.

$$f(x)' = -1(1 + e^{-x})^{-2} \cdot e^{-x}$$

Això ho podem expressar com a:

$$f(x)' = \frac{e^{-x}}{(1+e^{-x})^2}$$

O el que és el mateix:

$$f(x)' = \frac{1 \cdot e^{-x}}{(1+e^{-x}) \cdot (1+e^{-x})}$$

Llavors podem separar els dos termes, quedant de la següent manera:

$$f(x)' = \frac{1}{(1+e^{-x})} \cdot \frac{e^{-x}}{\cdot (1+e^{-x})}$$

I seguidament acollint-nos a la certesa que $1-1=0$, podem fer:

$$f(x)' = \frac{1}{(1+e^{-x})} \cdot \frac{e^{-x} + 1 - 1}{\cdot (1+e^{-x})}$$

I llavors separar:

$$f(x)' = \frac{1}{(1+e^{-x})} \cdot \left(\frac{(1+e^{-x})}{(1+e^{-x})} + \frac{-1}{(1+e^{-x})} \right)$$

I ara simplificant les fraccions:

$$f(x)' = \frac{1}{(1+e^{-x})} \cdot \left(1 - \frac{1}{(1+e^{-x})} \right)$$

O el que és el mateix:

$$f(x)' = (1+e^{-x})^{-1} \cdot (1 - (1+e^{-x})^{-1})$$

Això ens anirà molt bé quan derivar sigui un problema a causa de la quantitat de variables, ja que podrem calcular la derivada de forma ràpida i senzilla.

Aquesta funció és molt important ja que ens trenca linealitat del sistema. Aquesta solució comporta afegir complexitat matemàtica a l'hora d'entrenar la xarxa però ens dona la capacitat de donar resposta a problemes no lineals, sinó utilitzéssim la sigmoidal, es comportaria com un percepçó simple amb les limitacions que hem vist.

2.4.5 Softmax

Quan tenim un problema en què el model només té una neurona en la seva capa final, com per exemple que per a dos nombres donats hagi de dir quin és més gran, en el que la resposta és a o b, 0 o 1, la sigmoide és plenament capaç de treballar en totes les capes. Ara bé, quan en l'última capa necessitem més d'una neurona, com un problema que requereixi classificació en més de dos àmbits, la sigmoide pot no funcionar com voldríem.

El problema principal ve donat per incoherències lògiques. Si tenim un model que ens diferenciï entre els diferents tipus d'animals vertebrats i la sortida per a una determinada entrada és 0.2 per als mamífers, 0.1 per a les aus, 0.6 per als peixos, 0.3 per als rèptils i 0.2 per als amfibis, no podríem dir que hi ha un 60% (0.6) de possibilitats que sigui un peix, perquè no tindria sentit, ja que la suma de totes les probabilitats seria major a 1. Per a solucionar això fem servir la funció d'activació softmax en l'última capa, que ens farà que totes les sortides sumin 1 mantenint les proporcions inicials.

Per a aplicar la funció softmax el que es fa és «one-hot encoding» o codificació one-hot. Aquest mètode consisteix en tenir una llista amb tants valors com sortides possibles hi hagi i assignar un 0 a totes les sortides menys a la desitjada, que pren un 1 com a valor.

Matemàticament parlant, per a aplicar softmax sobre una de les sortides (y_i) n'hem de calcular l'exponencial i dividir-la entre la suma dels exponencials de totes les sortides, la funció és la següent:

$$\sigma(y_i) = \frac{e^{y_i}}{\sum_{\substack{j=1 \\ y_j=1}}^k e^{y_j}}$$

2.4.6 Teorema de l'aproximació universal

El teorema de l'aproximació universal ens diu que existeix una xarxa neuronal d'almenys una capa oculta la qual és capaç d'aproximar qualsevol funció contínua sempre que es faci servir una funció d'activació no lineal.

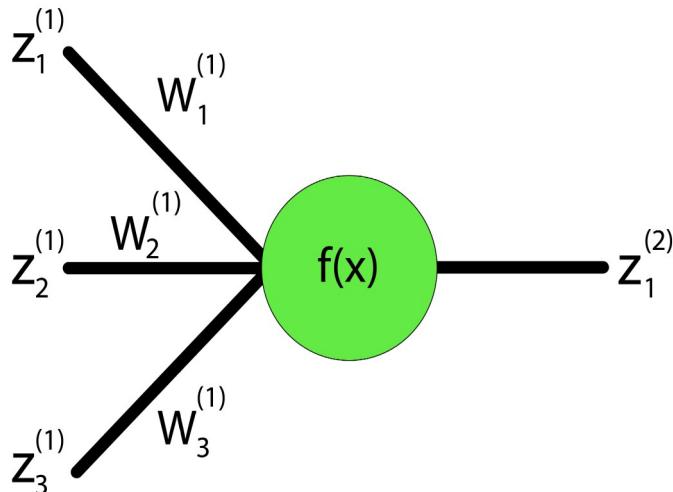
És un teorema dels que s'anomenen "d'existència", ja que ens diu que existeix tal x que compleix y , però no ens diu les propietats d'aquesta x . És a dir, sabem que existeix una xarxa neuronal que pot solucionar el problema que li posem, però no sabem quantes neurones per capa hem de posar ni quantes capes. Això és un problema que està obert i que per tant no té solució fins el moment. Ara com ara, la millor solució és provar diferents opcions i avaluar-ne els resultats. Si posem moltes neurones, el que farà la xarxa és memoritzar i no generalitzarà a altres casos, i si en posem molt poques, no podrà aprendre.

Sobre el terme que ens parla que la funció d'activació no pot ser lineal, totes les que es fan servir no ho són, en el nostre cas farem servir la sigmoide, com ja hem vist.

2.4.7 Output d'un perceptró multicapa

Ara veurem com calcular una sortida per al nostre perceptró multicapa.

Per començar farem una mica de repàs de com funciona una neurona, que és molt similar, per no dir igual de com ho feia amb el perceptró simple.



En una neurona tenim diverses entrades (sortides de neurones anteriors o bé les dades que introduïm a la xarxa) les quals ens arriben a través d'uns pesos. Fem el producte de l'entrada pel pes per on passa i els sumem tots afegint el bias. Tot això ho introduïm a la funció sigmoïdal de la neurona i ja tenim aquesta desitjada sortida.

Amb la neurona de la imatge, el que hem dit es representa de la següent manera:

$$z_1^{(2)} = f(z_1^{(1)} \cdot w_{1,1}^{(1)} + z_2^{(1)} \cdot w_{2,1}^{(1)} + z_3^{(1)} \cdot w_{3,1}^{(1)} + b_1^{(2)})$$

Com que realment això no es res més que un sumatori, això ho podem simplificar de la següent manera:

$$z_i^{(k)} = f\left(\sum_{j=1}^{n_{k-1}} (z_j^{(k-1)} \cdot w_{j,i}^{(k-1)}) + b_i^{(k)}\right)$$

Aquesta és una funció dependent, és a dir, necessitarem els resultats de les neurones anteriors per a poder calcular les neurones de més endavant. Així doncs, haurem de començar calculant des de $i=2$ (no comencem a la primera capa, ja que les primeres neurones només reben el que nosaltres introduïm a la xarxa i per tant no s'ha de calcular res) fins a la sortida de la xarxa neuronal.

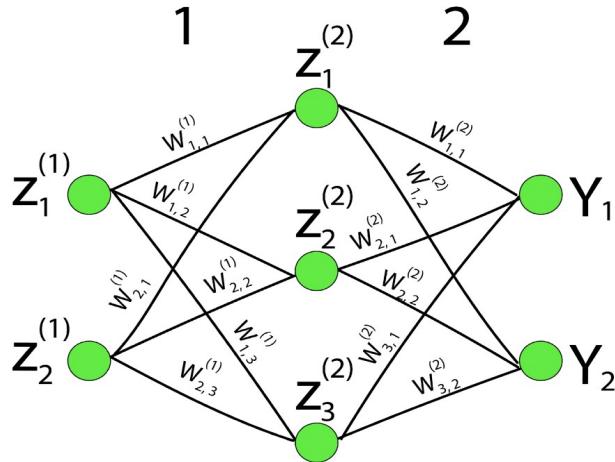
Recordem que aquesta funció és:

$$f(x) = (1 + e^{-x})^{-1}$$

Com hem vist, per tal de calcular les sortides d'un perceptró multicapa haurem de calcular la sortida de les neurones que es troben al final, però si ens hi fixem, veurem que per a calcular-la necessitarem la sortida de les neurones anteriors i així successivament fins a l'entrada.

Ara en veurem un exemple. Per a poder entendre-ho millor, farem servir un perceptró multicapa amb una configuració de 2 entrades, una capa oculta amb 3 neurones i 2 sortides.

Té la següent forma:



Com hem dit, necessitarem la sortida de les neurones del mig per a poder calcular la sortida final, així que primer les calculem:

$$z_1^{(2)} = f(z_1^{(1)} \cdot w_{1,1}^{(1)} + z_2^{(1)} \cdot w_{2,1}^{(1)} + b_1^{(2)})$$

$$z_2^{(2)} = f(z_1^{(1)} \cdot w_{1,2}^{(1)} + z_2^{(1)} \cdot w_{2,2}^{(1)} + b_2^{(2)})$$

$$z_3^{(2)} = f(z_1^{(1)} \cdot w_{1,3}^{(1)} + z_2^{(1)} \cdot w_{2,3}^{(1)} + b_3^{(2)})$$

Ara que ja tenim quan valen les sortides de la capa oculta ja podem calcular les sortides final del perceptró:

$$y_1 = f(z_1^{(2)} \cdot w_{1,1}^{(2)} + z_2^{(2)} \cdot w_{2,1}^{(2)} + z_3^{(2)} \cdot w_{3,1}^{(2)} + b_1^{(y)})$$

$$y_2 = f(z_1^{(2)} \cdot w_{1,2}^{(2)} + z_2^{(2)} \cdot w_{2,2}^{(2)} + z_3^{(2)} \cdot w_{3,2}^{(2)} + b_2^{(y)})$$

Ara mateix si ja sabéssim el valor d'aquests pesos i aquests bias, ja podríem posar en funcionament el nostre perceptró, ja que per a les entrades que se'n proposessin, en podríem calcular el resultat. Cal dir que aquest perceptró és molt simple i ha estat només un exemple. Al món real veurem perceptrons amb més capes i moltes més neurones per capa.

2.5 Retropropagació

L'algorisme de retropropagació és l'algorisme més utilitzat per a entrenar xarxes neuronals artificials. Consisteix en minimitzar l'error que es produeix a la xarxa neuronal. Quan tinguem el nostre error, el que farem és fer ús de mètodes per minimitzar la funció a partir de les capes de sortida fins a les d'entrada. El procediment és el següent:

- Calculem la sortida per a una entrada sabent-ne el resultat real.
- Busquem la diferència entre la sortida desitjada i la que realment obtenim.
- Comencem minimitzant l'error per als bias i pesos que estan més propers a la sortida.
- Anem traslladant aquest error inicial cap a les capes del principi, on s'acabarà minimitzant.

Aquest és un petit resum per a poder entrar ara en profunditat. En els termes de minimitzar una funció farem l'ús de derivades, en aquest cas parcials perquè tindrem moltes variables.

Com hem dit per a entrenar la xarxa haurem de trobar l'error a partir de les sortides desitjades i les que hem obtingut. Hi ha molts tipus d'error, nosaltres farem servir l'error quadràtic mitjà com veurem a continuació. Un cop explicat l'error que utilitzarem ja podrem començar amb l'algorisme del descens del gradient per a minimitzar aquest error.

2.5.1 Funció cost (error quadràtic mitjà)

La funció cost serveix per a determinar la diferència entre la sortida que obtenim i la sortida que volem obtenir. En concret farem servir l'error quadràtic mitjà.

Ara mirarem com obtenir aquest error i com introduir-lo a la nostra funció cost.

Per a obtenir la diferència entre l'entrada que obtenim i la que volem, farem servir la distància pitagòrica.

En dues dimensions, el teorema de Pitàgores, segurament una de les fórmules més famoses de la història, ens diu que:

Siguin S_1 i S_2 les sortides desitjades:

Siguin y_1 i y_2 les sortides reals:

$$\text{error} = \sqrt{(S_1 - y_1)^2 + (S_2 - y_2)^2}$$

Ara bé, el que ens interessarà a nosaltres és derivar aquest error per a poder-lo minimitzar. Si derivem la equació anterior i la igualem a zero:

$$\text{error}' = \frac{\sum_{i=1}^{S_n=y_n} (S_n - y_n)^2}{2 * \sqrt{\sum_{i=1}^{S_n=y_n} (S_n - y_n)^2}} = 0$$

I l'única manera de que això es compleixi, es quan el nostre numerador val zero. Així doncs, a l'hora de minimitzar l'arrel quadrada perd importància envers al que té al seu interior.

Llavors ens queda:

$$\text{error} = \sum_{i=1}^{S_n=y_n} (S_n - y_n)^2$$

I finalment l'error quadràtic mitjà afegeix un mig al sumatori. Això es fa perquè realment el que volem és una funció que decreixi al reduir l'error, llavors afegir aquest un mig no ens varia aquest aspecte i ens anirà molt bé a l'hora de derivar, tenim:

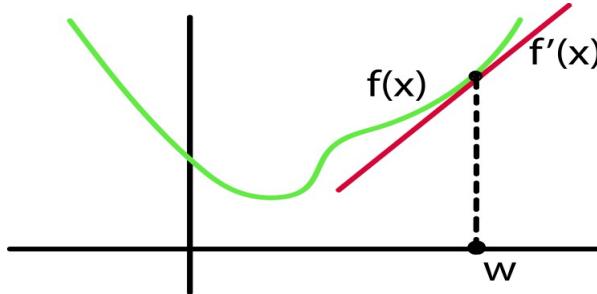
$$\text{error} = \sum_{i=1}^{S_n=y_n} \frac{1}{2} (S_n - y_n)^2$$

Quan el derivem, ens quedarà com l'error mateix, ja que el dos baixarà multiplicant i quedarà 1.

2.5.2 Descens del gradient

El descens del gradient és un mètode iteratiu per a trobar mínims o màxims a funcions. En xarxes neuronals, voldrem minimitzar l'error i no maximitzar-lo, per tant haurem de buscar els mínims. Això ho podem fer anant en contra de la derivada com ens diu el descens del gradient:

$$\Delta x = -f'(x)$$



Llavors podem marcar les iteracions en cerca del mínim de les diferents parts les quals ajustem segons:

$$x_{n+1} = x_n - f'(x_n)$$

Però si utilitzéssim això tindríem molts problemes perquè els salts que faria entre iteració i iteració serien prou grans com per a poder-se saltar el mínim que estem buscant.

És per això que s'incorpora la raó d'aprenentatge, simbolitzat amb la lletra grega α .

Llavors la fórmula ens queda com el següent:

$$x_{n+1} = x_n - \alpha \cdot f'(x_n)$$

Ara bé, amb això podem trobar mínims a una funció qualsevol, però per a minimitzar l'error a una xarxa neuronal no ho tindrem tan fàcil. Sobretot les complicacions són degudes a la gran quantitat de variables amb les quals treballem.

Per aplicar aquest descens del gradient a un perceptró multicapa haurem de fer servir derivades parcials.

Una derivada parcial sobre una funció amb moltes variables, és la derivada respecte a una de les variables deixant les altres com a constants.

Així doncs, haurem de derivar cadascuna de les nostres parts a calibrar, pesos i bias respecte a l'error que tinguem. Una derivada parcial es llegeix com «com canvia x quan vario y » en aquest cas, com canvia l'error quan en modifico els pesos o els bias.

Essent z un bias o bé un pes. la fórmula ens queda:

$$z_{n+1} = z_n - \alpha \cdot \frac{\partial \text{error}}{\partial z_n}$$

Però aquí trobem un problema. Hem dit que una derivada parcial és la derivada d'una funció respecte a una de les seves variables, però de fet els pesos i els bias no es troben a la funció error. L'error depèn de les sortides les quals sí que depenen dels pesos i els bias.

Quan tenim una funció G (en el nostre cas l'error) que depèn de variables, per exemple A, B , (en el nostre cas les sortides y_i) les quals són funcions i depenen d'altres variables a, b , (en el nostre cas els pesos i els bias), i volem derivar G en funció d' a, b ... ho podem fer de la següent manera.

$$\frac{\partial G}{\partial a} = \frac{\partial G}{\partial A} \cdot \frac{\partial A}{\partial a} + \frac{\partial G}{\partial B} \cdot \frac{\partial B}{\partial a}$$

O bé:

$$\frac{\partial G}{\partial b} = \frac{\partial G}{\partial A} \cdot \frac{\partial A}{\partial b} + \frac{\partial G}{\partial B} \cdot \frac{\partial B}{\partial b}$$

Així doncs això ho utilitzarem per a poder derivar els pesos i els bias respecte a l'error.

Ho podem generalitzar de la següent forma aplicant-ho ja a la xarxa neuronal:

$$\frac{\partial \text{error}}{\partial z_n} = \sum_{i=1}^{n_y} \left(\frac{\partial \text{error}}{\partial y_i} \cdot \frac{\partial y_i}{\partial z_n} \right)$$

Cal veure que aquesta funció és realment el mateix que havíem vist anteriorment, l'únic que s'ha afegit un sumatori per a generalitzar per a qualsevol número de sortides.

Encara ho podem simplificar una mica més:

Sabem que:

$$\text{error} = S_i - y_i$$

Llavors si derivem l'error respecte a y_i tenim:

$$\frac{\partial \text{error}}{\partial y_i} = y_i - S_i$$

Degut a que derivem respecte a y_i i no respecte $-y_i$ canviem tots els signes.

Llavors podem simplificar l'equació anterior amb el que hem deduït ara, llavors ens queda:

$$\frac{\partial \text{error}}{\partial a_n} = \sum_{i=1}^{n_y} \left((y_i - S_i) \cdot \frac{\partial y_i}{\partial a_n} \right)$$

Bé, recopilem una mica. Per a calcular iteracions amb l'equació del descens del gradient (és a dir per a dirigir-nos al mínim de la funció error), necessitem calcular la derivada parcial de l'error respecte a cadascun dels pesos i bias. Així doncs ara veiem que l'únic que ens falta per a poder-la calcular és la derivada parcial de la sortida que obtenim envers el pes que estem calibrant.

Per a veure un exemple de com es fa, hem de delimitar les capes d'un perceptró qualsevol al qual poder aplicar l'algorisme. Hem de fer això, ja que com veurem l'equació canvia segons la capa on es trobin els bias o els pesos. No cal que definim un nombre de neurones per capa, ja que això ho podem definir després segons necessitat canviant el rang dels sumatoris. Ho farem amb dues capes ocultes.

El primer que farem serà definir quant valen les sortides a partir de les entrades per a després poder-ne derivar parcialment els pesos i bias.

Farem servir la fórmula que hem vist anteriorment per a calcular les sortides de les neurones, la recordem:

$$z_i^{(k)} = f \left(\sum_{j=1}^{n_{k-1}} (z_j^{(k-1)} \cdot w_{j,i}^{(k-1)}) + b_i^{(k)} \right)$$

Començarem per les entrades i anirem calculant la sortida capa per capa:

$$z_i^{(2)} = f \left(\sum_{j=1}^{n_1} (x_j \cdot w_{j,i}^{(1)}) + b_i^{(2)} \right)$$

$$z_i^{(3)} = f \left(\sum_{j=1}^{n_2} (z_j^{(2)} \cdot w_{j,i}^{(2)}) + b_i^{(3)} \right)$$

I finalment:

$$y_i = f \left(\sum_{j=1}^{n_3} (z_j^{(3)} \cdot w_{j,i}^{(3)}) + b_i^{(y)} \right)$$

Ara tenim les 3 equacions que ens descriuen les sortides. Si ens hi fixem, totes depenen entre si, així que les podem escriure com una sola funció com veiem a continuació:

$$y_i = f \left(\sum_{j=1}^{n_3} \left(f \left(\sum_{p=1}^{n_2} (z_p^{(2)} \cdot w_{p,i}^{(2)}) + b_p^{(3)} \right) \cdot w_{j,i}^{(3)} + b_i^{(y)} \right) \right)$$

$$y_i = f \left(\sum_{j=1}^{n_3} \left(f \left(\sum_{p=1}^{n_2} \left(f \left(\sum_{q=1}^{n_1} (x_q \cdot w_{q,i}^{(1)}) + b_q^{(2)} \right) \cdot w_{p,i}^{(2)} + b_p^{(3)} \right) \cdot w_{j,i}^{(3)} + b_i^{(y)} \right) \right) \right)$$

Arribats a aquest punt, ja que anem a derivar la funció respecte a cadascun dels pesos, hem de recordar com funciona la regla de la cadena:

Donada una funció:

$$f(x)$$

On podem dir que:

$$x = g(k)$$

L'enunciat ens diu:

$$f'(x) = f'(g(k)) \cdot g'(k)$$

Però, és clar, veurem com $g'(k)$ és una altra funció polinòmica i així doncs tocarà aplicar el mateix procediment que hem aplicat anteriorment, llavors:

$$f'(x) = f'(g(k)) \cdot g'(k) \cdot k'$$

I això ho haurem de fer per a totes les funcions que anem trobant de forma encadenada.

Ara bé: calcular aquestes derivades seria molt complicat. Per tant, aprofitant que abans hem explicat i demostrat quant valia la derivada d'una sigmoide i constatant que totes aquestes funcions son sigmoïdals, ho podem aplicar a l'hora de derivar-les, amb la qual cosa ens queda:

$$f'(g(k)) = f(g(k)) \cdot (1 - f(g(k)))$$

$$g'(k) = g(k) \cdot (1 - g(k))$$

Si ho apliquem al que havíem obtingut anteriorment ens quedará:

$$f'(x) = f(g(k)) \cdot (1 - f(g(k))) \cdot g(k) \cdot (1 - g(k)) \cdot k'$$

Si k fos una altra funció continuaríem aplicant el mateix procés.

Així doncs, en el nostre cas:

$$f(x) = y_i$$

$$x = \sum_{p=1}^{n_2} \left(f \left(\sum_{q=1}^{n_x} (x_q \cdot w_{q,m}^{(1)}) + b_m^{(2)} \right) \cdot w_{p,k}^{(2)} \right)$$

Aleshores quan derivem:

$$f'(x) = y_i \cdot (1 - y_i) \cdot \left[f \left(\sum_{p=1}^{n_2} \left(f \left(\sum_{q=1}^{n_x} (x_q \cdot w_{q,m}^{(1)}) + b_m^{(2)} \right) \cdot w_{p,k}^{(2)} \right) \right) \right]$$

I com que x' torna a ser una funció, haurem d'aplicar el mateix procediment.

Llavors ara podem començar a derivar per cada filera de pesos i bias:

Primera capa:

$$\frac{\partial y_i}{\partial w_{j,k}^{(1)}} = y_i \cdot (1 - y_i) \cdot \left[\sum_{p=1}^{n^3} w_{p,i}^{(3)} \cdot z_p^{(3)} \cdot (1 - z_p^{(3)}) \cdot w_{k,p}^{(2)} \right] \cdot z_k^{(2)} \cdot (1 - z_k^{(2)}) \cdot x_j \quad \begin{matrix} i=1 \dots n^4 \\ j=1 \dots n^1 \\ k=1 \dots n^2 \end{matrix}$$

Com podem veure, primer calculem la derivada de la funció y_i , que com sabem és una sigmoide, després hem de derivar el que hi ha a l'interior, és a dir, les sortides de la capa 3, que les trobem al sumatori. Els pesos els posem dins del sumatori perquè la seva quantitat depèn del nombre de neurones que hi hagi a la tercera capa. Finalment tenim la neurona de la segona capa, la qual conté el pes que estem derivant. Com que és un sumatori, al derivar, totes les altres neurones d'aquesta capa s'han anul·lat. Així com els pesos de totes les capes que al estar sumant s'anul·len al derivar. Acabem amb la neurona d'entrada la qual es veu modificada pel pes que estem evaluant. Recordem que les neurones d'entrada no actuen com a tal, ja que no modifiquen el valor que reben a través de la sigmoide com sí ho fan les altres neurones.

Així doncs, ara podem continuar amb la següent capa. Cal veure que aquí l'entrada és la segona capa de neurones, ja que el que es veu influenciat per aquests pesos és el que va darrere seu i no pas abans, que no canvia al variar el pes. Per tant, els pesos $z^{(2)}$ actuen com a constants, quedant així:

$$\frac{\partial y_i}{\partial w_{j,k}^{(2)}} = y_i \cdot (1 - y_i) \cdot w_{k,i}^{(3)} \cdot z_k^{(3)} \cdot (1 - z_k^{(3)}) \cdot z_j^{(2)}$$

I finalment, els últims pesos, en aquest cas $z^{(3)}$ que també actuen com a constant:

$$\frac{\partial y_i}{\partial w_{j,k}^{(3)}} = y_i \cdot (1 - y_i) \cdot z_j^{(3)}$$

Ja tenim les equacions per a poder calcular l'ajust dels pesos. Ara ens falta l'altra part que podem calibrar: els bias.

Veiem que l'estructura és la mateixa, menys en la neurona a la qual estem calibrant el pes. A l'estar la neurona sumant, s'anul·la:

$$\frac{\partial y_i}{\partial b_k^{(2)}} = y_i \cdot (1 - y_i) \cdot \left[\sum_{p=1}^{n^3} w_{p,i}^{(3)} \cdot z_p^{(3)} \cdot (1 - z_p^{(3)}) \cdot w_{k,p}^{(2)} \right] \cdot z_k^{(2)} \cdot (1 - z_k^{(2)})$$

$$\frac{\partial y_i}{\partial b_k^{(3)}} = y_i \cdot (1 - y_i) \cdot w_{k,i}^{(3)} \cdot z_k^{(3)} \cdot (1 - z_k^{(3)})$$

$$\frac{\partial y_i}{\partial b_i^{(4)}} = y_i \cdot (1 - y_i)$$

Ja hem acabat de trobar totes les equacions. Ara bé, si recordeu el que volem era la derivada parcial de l'error respecte a les parts mòbils i ara tenim la derivada parcial de la sortida. Per tant hem de recuperar la següent formula que ja havíem deduït i simplement substituir el que hem fet ara.

$$\frac{\partial \text{error}}{\partial a_n} = \sum_{i=1}^{n_y} \left((y_i - S_i) \cdot \frac{\partial y_i}{\partial a_n} \right)$$

Llavors, per als pesos tenim:

- Tercera capa

$$\frac{\partial \text{error}}{\partial w_{p,i}^{(3)}} = \sum_{i=1}^{n_y} ((y_i - S_i) \cdot y_i \cdot (1 - y_i) \cdot z_p^{(3)})$$

En aquest cas, com que ja estem contemplant una i en concret, el sumatori no te sentit, ens queda:

$$\frac{\partial \text{error}}{\partial w_{p,i}^{(3)}} = (y_i - S_i) \cdot y_i \cdot (1 - y_i) \cdot z_p^{(3)}$$

- Segona capa

$$\frac{\partial \text{error}}{\partial w_{k,p}^{(2)}} = \sum_{i=1}^{n_y} ((y_i - S_i) \cdot y_i \cdot (1 - y_i) \cdot w_{p,i}^{(3)} \cdot z_p^{(3)} \cdot (1 - z_p^{(3)}) \cdot z_k^{(2)})$$

Com hem pogut veure, el que tenim són estructures molt repetitives. Ara el que farem serà modificar aquesta equació per a poder obtenir l'anterior dins d'aquesta i després poder establir una relació. Farem servir les següents propietats dels sumatoris:

$$\sum_{n=s}^t C \cdot f(n) = C \cdot \sum_{n=s}^t f(n) \quad D'on \quad C = \text{constant}$$

$$\sum_{i=k_0}^{k_1} \sum_{j=l_0}^{l_1} a_{i,j} = \sum_{j=l_0}^{l_1} \sum_{i=k_0}^{k_1} a_{i,j}$$

Així doncs, manipulant obtenim:

$$\frac{\partial \text{error}}{\partial w_{k,p}^{(2)}} = z_k^{(2)} \cdot z_p^{(3)} \cdot (1 - z_p^{(3)}) \cdot \left(\sum_{i=1}^{n_y} w_{p,i}^{(3)} \cdot (y_i - S_i) \cdot y_i \cdot (1 - y_i) \right)$$

- Primera capa

$$\frac{\partial \text{error}}{\partial w_{j,k}^{(1)}} = \sum_{i=1}^{n_y} \left((y_i - S_i) \cdot y_i \cdot (1-y_i) \cdot \left[\sum_{p=1}^{n_3} w_{p,i}^{(3)} \cdot z_p^{(3)} \cdot (1-z_p^{(3)}) \cdot w_{k,p}^{(2)} \right] \cdot z_k^{(2)} \cdot (1-z_k^{(2)}) \cdot x_j \right)$$

Igual que hem fet anteriorment, manipulem per a trobar les dues estructures anteriors:

$$\frac{\partial \text{error}}{\partial w_{j,k}^{(1)}} = x_j \cdot z_k^{(2)} \cdot (1-z_k^{(2)}) \cdot \sum_{p=1}^{n_3} \left[w_{k,p}^{(2)} \cdot z_p^{(3)} \cdot (1-z_p^{(3)}) \cdot \left(\sum_{i=1}^{n_y} w_{p,i}^{(3)} \cdot (y_i - S_i) \cdot y_i \cdot (1-y_i) \right) \right]$$

Amb els bias fem el mateix:

- Quarta capa

$$\frac{\partial \text{error}}{\partial b_i^{(4)}} = \sum_{i=1}^{n_y} (y_i - S_i) \cdot y_i \cdot (1-y_i)$$

Com hem dit abans, retirem el sumatori.

$$\frac{\partial \text{error}}{\partial b_i^{(4)}} = (y_i - S_i) \cdot y_i \cdot (1-y_i)$$

- Tercera capa

$$\frac{\partial \text{error}}{\partial b_p^{(3)}} = \sum_{i=1}^{n_y} \left((y_i - S_i) \cdot y_i \cdot (1-y_i) \cdot w_{p,i}^{(3)} \cdot z_p^{(3)} \cdot (1-z_p^{(3)}) \cdot z_j^{(2)} \right)$$

$$\frac{\partial \text{error}}{\partial b_p^{(3)}} = z_p^{(3)} \cdot (1-z_p^{(3)}) \cdot \sum_{i=1}^{n_y} \left((y_i - S_i) \cdot y_i \cdot (1-y_i) \cdot w_{p,i}^{(3)} \right)$$

- Segona capa

$$\frac{\partial \text{error}}{\partial b_k^{(2)}} = \sum_{i=1}^{n_y} \left((y_i - S_i) \cdot y_i \cdot (1-y_i) \cdot \left[\sum_{p=1}^{n_3} w_{p,i}^{(3)} \cdot z_p^{(3)} \cdot (1-z_p^{(3)}) \cdot w_{k,p}^{(2)} \right] \cdot z_k^{(2)} \cdot (1-z_k^{(2)}) \right)$$

$$\frac{\partial \text{error}}{\partial b_k^{(2)}} = z_k^{(2)} \cdot (1-z_k^{(2)}) \cdot \sum_{p=1}^{n_3} \left[w_{k,p}^{(2)} \cdot z_p^{(3)} \cdot (1-z_p^{(3)}) \cdot \left(\sum_{i=1}^{n_y} w_{p,i}^{(3)} \cdot (y_i - S_i) \cdot y_i \cdot (1-y_i) \right) \right]$$

Com hem vist, tenim estructures que es van repetint. De fet, l'única diferència entre un pes i el bias és que als pesos hi afegim la neurona de la qual surt el pes i en els bias no, ja que es troba sumant.

En conseqüència, podem establir estructures per capes que després ens facilitaran la programació. Aquestes estructures es simbolitzen amb la lletra grega delta de la següent manera:

El superíndex (n) ens indica la capa a la que equival aquesta estructura i el subíndex (i) ens diu la lletra que li hem assignat a la capa:

$$\delta_i^{(n)}$$

Com veiem, el que hem fet manipulant les equacions és aconseguir una estructura com la següent:

		$\delta^{(4)}$	z
	$\delta^{(3)}$	y	z
$\delta^{(2)}$	x	y	z

Ara doncs, a partir de les repetitions a les equacions que s'han vist anteriorment podem establir delta com a:

$$\delta_k^{(2)} = z_k^{(2)} \cdot (1 - z_k^{(2)}) \cdot \sum_{p=1}^{n_3} \left[w_{k,p}^{(2)} \cdot z_p^{(3)} \cdot (1 - z_p^{(3)}) \cdot \left(\sum_{i=1}^{n_y} w_{p,i}^{(3)} \cdot (y_i - S_i) \cdot y_i \cdot (1 - y_i) \right) \right]$$

$$\delta_p^{(3)} = z_p^{(3)} \cdot (1 - z_p^{(3)}) \cdot \sum_{i=1}^{n_y} ((y_i - S_i) \cdot y_i \cdot (1 - y_i) \cdot w_{p,i}^{(3)})$$

$$\delta_i^{(4)} = (y_i - S_i) \cdot y_i \cdot (1 - y_i) \cdot z_j^{(3)}$$

D'aquesta manera ja podem escriure la fórmula final que ens permetrà reduir l'error a partir d'iteracions:

Pesos:

$$w_{j,k}^{(1)} \leftarrow w_{j,k}^{(1)} - \alpha \cdot x_j \cdot \delta_k^{(2)}$$

$$w_{k,p}^{(1)} \leftarrow w_{k,p}^{(1)} - \alpha \cdot z_k^{(2)} \cdot \delta_p^{(3)}$$

$$w_{p,i}^{(1)} \leftarrow w_{p,i}^{(1)} - \alpha \cdot z_p^{(3)} \cdot \delta_i^{(4)}$$

Bias:

$$b_k^{(2)} \leftarrow w_{j,k}^{(1)} - \alpha \cdot \delta_k^{(2)}$$

$$b_p^{(3)} \leftarrow w_{k,p}^{(1)} - \alpha \cdot \delta_p^{(3)}$$

$$b_i^{(4)} \leftarrow w_{p,i}^{(1)} - \alpha \cdot \delta_i^{(4)}$$

2.6 Implementació d'un perceptró multicapa

A continuació, a partir de tots els fonaments matemàtics que hem vist, començarem amb la programació d'un perceptró multicapa amb el llenguatge de programació python. Posteriorment s'entrenarà amb diverses dades i es provarà la seva eficiència.

2.6.1 Programació

L'estructura del perceptró serà la següent:

```
class MLP:  
    def __init__(self):  
    def ejecucion(self):  
    def salida(self):  
    def retropropagacion(self):  
    def guardar(self):
```

Tenim un objecte, el nostre perceptró. Aquest objecte consta de 5 funcions diferents:

- def __init__ : És el constructor, quan cridem al nostre perceptró s'executará. En ell hi introduirem els diàlegs i muntarem el perceptró segons se'ns demani o bé l'executarem si ja ha estat entrenat.
- def ejecucion: Aquesta funció controlarà els bucles que fem i anirà cridant a salida, retropropagació i a guardar quan sigui necessari:
- def salida: Agafa els valors d'entrada que li passem i calcula la salida
- def retropropagació: a partir de la salida que obtenim i la que hauríem d'haver obtingut recalcula els pesos i el bias (grau d'inhibició) de cada neurona.
- def guardar: guardem el perceptró en un fitxer .txt.

Cal dir que s'utilitzen 3 mòduls natius del python, com són json (per a poder llegir i escriure fitxers) numpy (per a operacions matemàtiques com l'exponent o la tolerància) i random (per a generar números pseudoaleatoris)

Aquí anirem passant funció per funció i mirant-les en detall. Es recomana visitar els repositoris de github que es troben a continuació per a poder veure i descarregar el codi complert.

www.github.com/bielaltes/TdR

2.6.1.1 def __init__(self)

El que volem en aquesta funció és definir-hi les variables globals i cridar a l'execució perquè controli el programa per a entrenar la xarxa o bé per a executar-la amb un seguit de valors.

Les variables globals són molt importants, potser les que ho són més són les referides a l'estruatura del perceptró, aquestes són:

```
self.perceptró = []
```

Aquesta variable és una llista que consta d'un element per a cada capa que trobem al perceptró. Cadascun d'aquests elements contenen al seu interior els bias i els pesos de la capa que representen.

```
self.dades_entrenament = []
```

```
self.dades_execucio = []
```

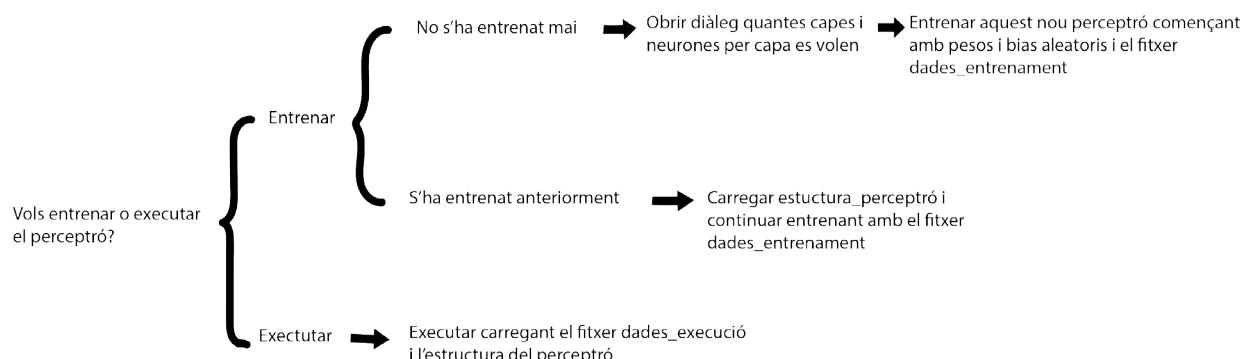
Unes altres dos variables molt importants seran on guardem les dades d'entrenament o d'execució. Quan s'executa el programa es bolquen els dos fitxers «dades_entrenament» i «dades_execució» en dues llistes que són les que després es revisaran al programa.

```
self.sortides = []
```

Aquesta variable torna a ser una llista i també té els mateixos elements que capes té el perceptró. En el seu interior anem guardant les sortides que tenen les diferents neurones de cada capa.

D'altres variables que hem d'inicialitzar són l'error, la taxa d'aprenentatge i una booleana per determinar si estem executant o entrenant el perceptró.

Així doncs, després de definir les variables globals, crearem un diàleg on es pregunti què vol fer l'usuari, les possibilitats són les següents:



D'aquesta manera obtenim el següent codi: observem que la variable mode serveix per a indicar quan s'està entrenant o executant el perceptró. Quan entri en acció la funció definició, activarà retropropagació dependent si s'està entrenant o executant, és a dir, si la variable mode es True o False.

```

import random, json, numpy

class MLP:
    def __init__(self):
        print("#### YOU'RE RUNNING BIEL'S PERCEPTRON ####")
        self.perceptrone = []
        self.sortides = []
        self.dades_entrenament = []
        self.dades_execucio = []
        self.alpha = 0.1
        self.S_d = []
        self.mode = 'bool'

    entrenar_executar = input("Vols entrenar (1) o executar (2) el perceptrone multicapa?: ")
    if entrenar_executar == "1":
        self.mode = True
        with open('dades_entrenament.txt', 'r') as training:
            self.dades_entrenament = json.load(training)
        nova_creada = input("El MLP es nou (1) o ja l'havies entrenat abans (2)?: ")

        if nova_creada == "1":
            numero_capes = int(input("Selecciona el numero de capes: "))
            for x in range(numero_capes):
                capa = []
                z = []
                w = []
                b = []
                neurones_per_capa = []
                neurones_per_capa.append(int(input('numero de neurones a la capa ' + str(x) + ' : ')))
                for y in range(int(neurones_per_capa[-1])):
                    b.append(random.random())
                    z.append(random.random())
                    w_per_capa = []
                    if x != 0:
                        for k in range(len(self.perceptrone[x-1][0])):
                            w_per_capa.append(random.random())
                    w.append(w_per_capa)
                capa.append(b)
                capa.append(w)
                self.perceptrone.append(capa)
                self.sortides.append(z)
            self.execucio()
            self.guardar()

        elif nova_creada == "2":
            with open('estructura_perceptrone.txt', 'r') as estructura_perceptrone:
                estructura_completa = json.load(estructura_perceptrone)
                self.perceptrone = estructura_completa[0]
                self.sortides = estructura_completa[1]
            self.execucio()
            self.guardar()

        else:
            print("error")

    elif entrenar_executar == "2":
        self.mode = False
        with open('dades_execucio.txt', 'r') as executing:
            self.dades_execucio = json.load(executing)
        with open('estructura_perceptrone.txt', 'r') as estructura_perceptrone:
            estructura_completa = json.load(estructura_perceptrone)
            self.perceptrone = estructura_completa[0]
            self.sortides = estructura_completa[1]
        self.execucio()

    else:
        print("error")

```

2.6.1.2 def execucio(self)

Ara, tal com havíem dit continuarem amb la funció execució.

L'execució ha de controlar el funcionament del perceptró. Té dues opcions, segons s'estigui executant o entrenant el perceptró.

- Entrenament: Quan el perceptró s'està entrenant, s'entrena sobre un rang decidit per l'usuari. D'aquesta manera s'obre un quadre de text preguntant-li quants bucles vol executar. Però què és un bucle? Un bucle és l'execució i correcció per retropropagació de totes les dades que conformen la variable dades_entrenament. Cada x bucles es guarden les dades d'entrenament per no perdre-ho tot si tenim algun problema. Cal veure com per a cada dada de la variable dades_entrenament es constitueixen com a entrades els valors d'entrenament i es calcula la sortida. A partir d'aquí, es defineix la sortida desitjada i s'executa la funció retropropagació.
- Execució: En cas que el perceptró s'estigui executant, es farà tot l'anterior menys cridar a retropropagació. A més a més es porta un recompte del percentatge dels cops que s'ha encertat. Es contempla una petita tolerància, ja que difícilment s'obtindrà el 0 o l'1 exacte.

```
def execucio(self):
    nombre_bucle = 0
    if self.mode == True:
        for y in range(int(input("Quants cops vols entrenar el bucle?: "))):
            for x in range(int(len(self.dades_entrenament))):
                self.sortides[0] = self.dades_entrenament[x][0]
                self.sortida()
                self.S_d = []
                self.S_d = self.dades_entrenament[x][1]
                self.retropropagacio()
            nombre_bucle += 1
            print("Portes " + str(nombre_bucle) + " bucles")
            if nombre_bucle % 10 == 0:
                self.guardar()
    else:
        encerts = 0
        errors = 0
        for x in range(int(len(self.dades_execucio))):
            self.sortides[0] = self.dades_execucio[x][0]
            self.sortida()
            for i in range(len(self.sortides[-1])):
                if numpy.isclose(self.sortides[-1][i], self.dades_execucio[x][1][i], atol=0.1):
                    encerts += 1
                else:
                    errors += 1
                    print("error")
        percentatge_encerts = ((encerts / (encerts + errors)) * 100)
        print(percentatge_encerts)
```

2.6.1.3 def sortida(self)

La idea d'aquesta funció és que simplement agafi les entrades a la xarxa i vagi calculant sortides capa per capa fins a obtenir la sortida final.

Com es pot veure, no es fa sempre ús de la sigmoide com a tal. En el cas que ho féssim, es crearien valors tan grans que acabarien causant overflow (s'excedeix la capacitat màxima d'una variable double). Es considera que si la sortida abans d'aplicar la sigmoide és major a 20 o menor a -20 ja es pot considerar la sortida 0 i 1 respectivament, ja que prement $x=20$ la sortida de la sigmoide és 0,9999999979 i amb $x=-20$ és 0,0000000206.

En el cas que estiguem executant la xarxa, s'imprimeix el valor que pren la sortida.

```
def sortida(self):
    for k in range(len(self.perceptró)-1):
        for i in range(len(self.sortides[k+1])):
            sortida_neurona = 0
            for j in range(len(self.sortides[k])):
                sortida_neurona += (self.sortides[k][j] * self.perceptró[k+1][1][i][j])
            sortida_neurona += self.perceptró[k+1][0][i]
            if sortida_neurona > 20:
                sortida_neurona = 1
            elif sortida_neurona < -20:
                sortida_neurona = 0
            else:
                sortida_neurona = (1/(1 + numpy.exp(-sortida_neurona)))
            self.sortides[k+1][i] = sortida_neurona

    if self.mode == False:
        print(self.sortides[-1])
```

2.6.1.4 def retropropagacio(self)

Quan cridem a la funció retropropagació, el que es fa és agafar el resultat sortint de la funció self.sortida() i la sortida desitjada i recalcular els pesos i els bias.

Com es pot veure costa de dos bucles:

- Es calcula la primera delta, la qual serà sempre necessària, ja que el mínim de capes són dues (una d'entrada i una de sortida). Seguidament, es calcula la delta_y i s'aplica a cadascun dels pesos i bias de l'última capa. Com que el número de bias correspon al nombre de neurones de la capa en la qual estem, el nombre de pesos és el producte de la capa anterior i la que estem. Per això els pesos es modifiquen dins d'un bucle amb les repeticions equivalents al nombre de neurones de la capa anterior.
- Quan ja tenim la delta_y i la última capa calibrada, ara hem de fer-ho amb les altres. Com que és un perceptró «adaptatiu», és a dir, l'usuari pot triar les capes que vol, depèndrà de les capes que hagi triat l'usuari. La idea del segon bucle és anar aplicant delta cadascuna de les capes següents (del final fins a l'inici) i anar afegint a delta_y a cadascuna de les parts que corresponen a les següents deltes. Quan ja tenim la delta de la capa que estem modificant, en calibrem els pesos i els bias.

```
def retropropagacio(self):
    delta_y = []
    for k in range(len(self.sortides[-1])):
        delta_y.append(self.sortides[-1][k] * (1 - self.sortides[-1][k]) * (self.sortides[-1][k] - self.S_d[0]))

        self.perceptró[len(self.perceptró)-1][0][k] -= self.alpha * delta_y[k]
        for y in range(len(self.sortides[-2])):
            self.perceptró[len(self.perceptró)-1][1][y] -= self.alpha * self.sortides[-2][y] * delta_y[k]

    for p in range(len(self.sortides)-2):
        delta_t = []
        sumatori_retropropagacio = 0

        for k in range(len(self.sortides[-2-p])):
            for j in range(len(self.sortides[-3-p])):
                for i in range(len(self.sortides[-1-p])):
                    sumatori_retropropagacio += delta_y[i] * self.perceptró[len(self.perceptró)-1-p][1][i][k]
            delta_t.append((self.sortides[-2-p][k] * (1-self.sortides[-2-p][k])) * sumatori_retropropagacio)

            self.perceptró[len(self.perceptró)-2-p][0][k] -= self.alpha * delta_t[k]
            for n in range(len(self.sortides[-3-p])):
                self.perceptró[len(self.perceptró)-2-p][1][k][n] -= self.alpha * self.sortides[-3-p][n] * delta_t[k]
        delta_y = delta_t
```

2.6.1.5 def guardar(self)

Aquesta ja és l'última funció. Quan la cridem, s'encarrega simplement d'obrir el fitxer estructura_perceptró.txt i sobreescriu les dades amb l'actual estructura. Quan acaba imprimeix una nota que ens diu que aquest procediment s'ha dut a terme correctament.

```
def guardar(self):
    estructura_perceptró = [self.perceptró, self.sortides]
    with open('estructura_perceptró.txt', 'w') as perceptró:
        json.dump(estructura_perceptró, perceptró)
    print("Guardat correctament")
```

2.6.2 Proves

A continuació provarem el perceptró multicapa que hem programat. Primer, veiem què necessitem per a executar-lo i quins programes i fitxers requereix. La compatibilitat ha estat testejada amb Windows 10, però en principi amb qualsevol altre sistema operatiu hauria de funcionar.

Primer, cal tenir instal·lat python 3 a l'ordinador i alguna IDE com pot ser PyCharm, Visual Studio, Jupyter o qualsevol altra. En el cas que es vulgui executar des del cmd de Windows, caldria retirar els int() que es troben davant dels inputs(), en qualsevol cas es recomana l'execució des de l'IDE.

El programa requereix tres fitxers que han d'estar situats a la mateixa carpeta que es troba el programa. Aquests fitxers són:

- estructura_perceptró

Consta de dues llistes. En la primera s'emmagatzemen els valors que prenen tant els pesos com els bias de cada capa. En la segona es carrega l'última sortida de cada capa. Això es fa perquè quan carreguem la xarxa neuronal, ja estigui creada l'estructura de les sortides i no tenir cap problema. Els valors que tinguin seran sobreescrits i per tant no tenen importància.

Aquest fitxer es crea automàticament quan es forma la xarxa neuronal per primer cop. En el cas que es vulgui fer ús d'un model preentrenat del repositori de github, n'hi ha prou amb descarregar el fitxer i copiar-lo a la carpeta on es troba el programa.

- dades_entrenament

Aquest fitxer s'ha de crear prèviament a l'entrenament i en ell hi trobem totes les entrades i sortides amb les quals vulguem entrenar la nostra xarxa, ha d'estar format de la següent manera:

[[[entrada₁, entrada₂, entrada_n],[sortida₁, sortida₂, sortida_n]],[...]]

Així doncs amb la taula AND per exemple seria:

[[[1, 1], 1],[[1, 0], 0],[[0, 1], 0],[[0, 0], 0]]]

- dades_execucio

Aquest fitxer es fa servir quan s'executa la xarxa neuronal i té la mateixa estructura del fitxer anterior. Realment es podria utilitzar sense la resposta final, però incorporem la sortida desitjada al fitxer per a poder calcular el percentatge d'encerts que té la xarxa neuronal.

2.6.2.1 Aprenent quin número es més gran (0-1000)

La idea d'aquesta prova és que el model aprengui a diferenciar quin número és més gran quan li donem dos valors entre 0 i 1000.

El primer que hem de preparar són les dades d'entrenament i d'execució, això ho farem amb un altre petit programa de python.

```
import random, json

llista = []

for i in range(10000):
    entrada = []
    entrades = []
    final = []
    sortida= []
    a = random.randint(0,1000)
    b = random.randint(0,1000)
    a_n = a/1000
    b_n = b/1000
    entrades = [a_n,b_n]
    entrada.extend(entrades)
    if a >= b:
        sortida.append(1)
    else:
        sortida.append(0)
    final.append(entradas)
    final.append(sortida)
    llista.append(final)

with open('dades_entrenament.txt', 'w') as training:
    json.dump(llista, training)
```

Aquest programa ens crea la llista que volem i la guarda al fitxer perquè després la pugui obrir el perceptró multicapa. Simplement canviem el nom del fitxer quan volem les dades d'entrenament o bé les dades d'execució. En aquest cas es van fer servir 10000 entrades per a entrenar i 1000 per a comprovar els resultats. Dividim l'entrada entre 1000 per a poder tenir millor controlats els valors i que pugui entrenar una mica més ràpid a l'estar els valors compresos entre 0 i 1.

En total hi ha 1 milió de possibilitats (1000^2) i la xarxa només en veu 10000. Tot i això, al provar el perceptró veiem que encerta tots els valors que li hem introduït diversos cops. No cal dir que òbviament l'operador $<$ i $>$ no l'hem introduït, sinó que ha après el patró.

El model preentrenat està disponible al github, consta de 3 capes ocultes amb 4 neurones cadascuna i ha estat entrenat durant 3 hores amb uns 20000 bucles. Si es vol provar amb altres dades, també està disponible el generador de llistes.

2.6.3 Conclusions del perceptró multicapa amb python

Com hem pogut veure, aquest perceptró funciona de manera excel·lent a l'hora de dur a terme la funció que li hem assignat. Tot i això, la xarxa ha necessitat molt temps d'entrenament. Aquest fet el converteix en una estructura gens viable. Aquest excés de temps és degut a molts factors, el primer dels quals és la manera en què ha estat programat el perceptró. Quan es busca rapidesa, sempre es busca el càlcul matricial que tan bé efectuen els nostres processadors i gràfiques. El problema és que per a fer-ho d'aquesta manera python no és el llenguatge adequat i fent-ho en algun altre llenguatge (C o C++) el programa se'n complicaria molt. A més a més, el perceptró ha estat funcionant en un sol nucli del processador i amb bucles de python que són famosos pel seu baix rendiment.

Tot i això, la pràctica ens ha servit per a comprovar l'eficàcia d'aquest model aplicat directament des dels fonaments matemàtics, cosa que hem aconseguit perfectament i per tant podríem dir que el resultat ha estat molt satisfactori.

D'aquesta manera, concloem aquesta primera part del treball i ens endinsem en el món del TensorFlow, un ecosistema que sí que està creat pensant en el rendiment.

3 Deep learning amb TensorFlow i Keras

Aquesta és la segona part del treball de recerca. En la primera part, hem fet una introducció a les xarxes neuronals i posteriorment n'hem vist els fonaments matemàtics per a poder programar un model relativament senzill i amb molt poca eficiència. A continuació veurem una eina que ens permet crear models complexos de manera senzilla i que és molt utilitzada tant per usuaris domèstics com per usuaris professionals.

A continuació veurem les diferents parts que tractarem en aquest apartat:

- Què és TensorFlow?: plataforma que utilitzarem per executar el codi.
- Què és Keras?: llibreries que farem servir i que s'utilitzen sobre TensorFlow.
- Estructura del codi TensorFlow: veurem les parts bàsiques del codi.
- Paràmetres i hiperparàmetres: repasarem aquests dos tipus de variables.
- Estructures neuronals: veurem les possibilitats que ens proporcionen TensorFlow i Keras.

3.1 Què és TensorFlow?

TensorFlow és una API de codi obert desenvolupada per l'equip Google Brain, propietat de Google. Està desenvolupada en C i normalment es fa servir sobre python, tot i que també és compatible amb altres llenguatges de programació com JavaScript.

Avui en dia TensorFlow és la plataforma dirigida a l'aprenentatge profund més utilitzada del món. TensorFlow opera mitjançant gràfics de flux de dades formats per matrius multidimensionals connectades entre elles.

Concretament estarem utilitzant la segona versió, TensorFlow 2.0, presentada el setembre de 2019. Per a fer la programació encara més senzilla, farem servir una llibreria molt intuïtiva d'alt nivell anomenada Keras.

3.2 Què és Keras?

Keras és una biblioteca de xarxes neuronals artificials escrita en python que pot córrer sobre TensorFlow i facilita la implementació dels models. És desenvolupada en el projecte ONEIROS (Open-ended Neuro-Electronic Intelligent Robot Operating System) amb François Chollet (un reconegut enginyer de Google) al capdavant.

3.3 Estructura del codi TensorFlow

Quan veiem un codi TensorFlow, observem que està dividit en 4 parts bàsiques:

- Tractament del dataset: suposant que ja tenim un dataset preparat, quasi sempre n'haurem de tractar les dades per a normalitzar-les o distribuir-les segons la nostra conveniència per a entrenar el model.
- Definició del model: definició les capes i les característiques de cada capa.
- Compilació i entrenament: definició de com volem que s'entreni el model i posterior entrenament amb el dataset que teníem.
- Testeig i ús del model: un cop acabat el procés d'entrenament podrem testar el model i començar a fer-lo servir per a inferència.

3.4 Paràmetres i hiperparàmetres d'un model

Com hem vist abans, en una xarxa neuronal trobem dos tipus de dades, els paràmetres i els hiperparàmetres. Recordem que els paràmetres són els que nosaltres no decidim directament, com pot ser el valor que pren cadascun dels pesos o els bias, que s'actualitzen seguint retropropagació.

D'altra banda també hem de recordar que hi ha una altra mena de dades que són molt importants, els hiperparàmetres. En el perceptró multicapa que hem programat anteriorment hem explicat algun hiperparàmetre com la taxa d'aprenentatge. Un hiperparàmetre és tot allò del model que definim nosaltres abans de començar la fase d'entrenament. Així doncs, el nombre de capes, el nombre de neurones per capa, l'algorisme d'aprenentatge (optimitzador) o la funció d'activació entre d'altres són hiperparàmetres. Abans no hem profunditzat gaire en els possibles hiperparàmetres, ja que la majoria ja estaven predefinitos i per tant eren invariables. Ara amb TensorFlow es tornen de vital importància, ja que són moltes les opcions per a poder triar i tenen un impacte directe al rendiment del model.

3.5 Estructures neuronals

Fins ara hem vist l'estructura que segueix un perceptró multicapa, un número definit de capes densament connectades. Utilitzant TensorFlow juntament amb Keras podrem fer servir diferents tipus d'estructures neuronals segons la tasca que vulguem dur a terme. En aquest apartat en veurem 3, amb les seves respectives pràctiques:

- MLP: Farem ús d'un perceptró multicapa (Multi-Layer Perceptron) com a estructura més senzilla per a dur a terme un problema de regressió lineal.
- CNN: També programarem un model basat en una xarxa neuronal convolucionada (Convolutional Neural Network), utilitzada en el tractament d'imatges.
- RNN: Finalment farem ús d'un model basat en neurones recurrents (Recurrent Neural Network), que s'utilitza en tractament de textos.

3.5.1 MLP (multi layer perceptron)

Un MLP (multi-layer perceptron) és un conjunt de capes densament connectades. Així doncs, és el mateix que hem vist a la primera part del treball. Aquesta estructura la farem servir per a crear un model regressiu que predigi el preu de cases basant-se en un dataset molt famós inclòs dins de la llibreria Keras. És una pràctica molt senzilla que ens permet veure com funciona Tensorflow i Keras a mode de petita introducció. Tant per aquesta pràctica com per les següents farem servir el Google Colab², una plataforma de Google on se'ns deixa una targeta gràfica i un entorn de desenvolupament TensorFlow de manera totalment gratuïta tan sols iniciant sessió amb un compte de Google. Això ens facilitarà molt la feina i per a fer pràctiques senzilles és ideal.

3.5.1.1 Boston housing price dataset

El primer que farem serà importar les dependències, en aquest cas tan sols Tensorflow, Keras i numpy.

```
import tensorflow as tf
import numpy as np
from tensorflow import keras
print(tf.__version__)
```

Seguidament importarem el dataset a partir dels conjunt de datasets de prova de Keras:

```
boston_housing = tf.keras.datasets.boston_housing
x_train, y_train), (x_test, y_test) = boston_housing.load_data()
```

² <https://Colab.research.Google.com/>

Com veiem importem el dataset fent servir la funció load_data. El dataset és un conjunt de tuples de numpy les quals separen en dades d'entrenament i de test. Definirem les entrades a la xarxa neuronal com a x i les sortides desitjades com a y. Exemple: x_train (entrades entrenament), y_train (sortides desitjades entrenament). La relació entre dades d'entrenament i test la podríem ajustar mitjançant l'argument test_split, que per defecte està definit a 0.2.

Mitjançant la funció shape podem veure la forma que pren aquest dataset:

```
print(f'Training data : {x_train.shape}')
print(f'Test data : {x_test.shape}')
```

Output:

```
Training data : (404, 13)
Test data : (102, 13)
```

Com veiem tenim 404 cases diferents en entrenament i 102 en test. Cada casa té com a característica una array de 13 valors, com els metres quadrats de la casa entre d'altres. El llistat complet d'atributs es pot comprovar a la pàgina web originària del dataset³.

Ara el que haurem de fer és normalitzar les dades per tal que la nostra xarxa neuronal les pugui processar millor, farem servir la mitjana i la desviació estàndard calculada automàticament per numpy.

```
mean = x_train.mean(axis=0)
x_train -= mean
x_test -= mean
std = x_train.std(axis=0)
x_train /= std
x_test /= std
```

Això ens permet tenir valors que mantenen la proporció inicial, però són molt més petits, pròxims a 0.

Seguidament ja podem definir el model, utilitzarem un parell de capes densament connectades amb una capa de sortida d'una sola neurona que farà la predicción correspondiente. Com a funció d'activació per a trencar la linealitat farem servir la sigmoide, com ja l'havíem utilitzat anteriorment. El model = Sequential() fa referència al tipus de model que estem construint.

```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense

model = Sequential()
model.add(Dense(64, activation='sigmoid', input_shape=[13]))
model.add(Dense(64, activation='sigmoid'))
model.add(Dense(1))

model.summary()
```

3 <http://lib.stat.cmu.edu/datasets/boston>

Un cop tenim el model definit l'hem de compilar, això amb TensorFlow vol dir definir l'optimitzador, la funció cost i les mètriques que vulguem fer servir, en aquest cas farem servir l'Adam, un optimitzador que resumidament aplica un descens del gradient, però amb una taxa d'entrenament personalitzada a cadascun dels paràmetres, cosa que dona molt bons resultats.

```
from tensorflow.keras.optimizers import Adam  
  
model.compile(loss='mse',  
               optimizer=Adam(lr=1e-4),  
               metrics=['mae'])
```

Fem servir com a funció cost el Mean Squared Error, que va bé per evitar grans errors al model, però per a veure l'error en sí com a resultat farem servir l'error absolut, que és el que busquem. Veiem que li diem que volem 50 epochs. Una epoch és allò que definíem com a bucles en el nostre perceptró multicapa, és a dir, les iteracions que el model fa sobre el dataset d'entrenament,

Finalment ja podem entrenar i veure els resultats de la regressió amb el dataset de test.

```
EPOCHS = 50  
history = model.fit(x_train, y_train, epochs=EPOCHS,  
                      validation_split = 0.2, verbose=0)  
  
loss, mae = model.evaluate(x_test, y_test)  
print(f'Error absolut mitjà en test : {round(mae, 2)}')
```

Podem veure com l'error en el dataset de test està al voltant de 3, és a dir, 3000 dòlars, ja que en el dataset d'entrenament els valors els trobàvem en milers de dòlars. Comtant que estem veient preu de cases que ronden els 70 o 80 mil dòlars (el dataset és dels anys setanta) és un error bastant petit i comprensible que ens atorga uns bons resultats.

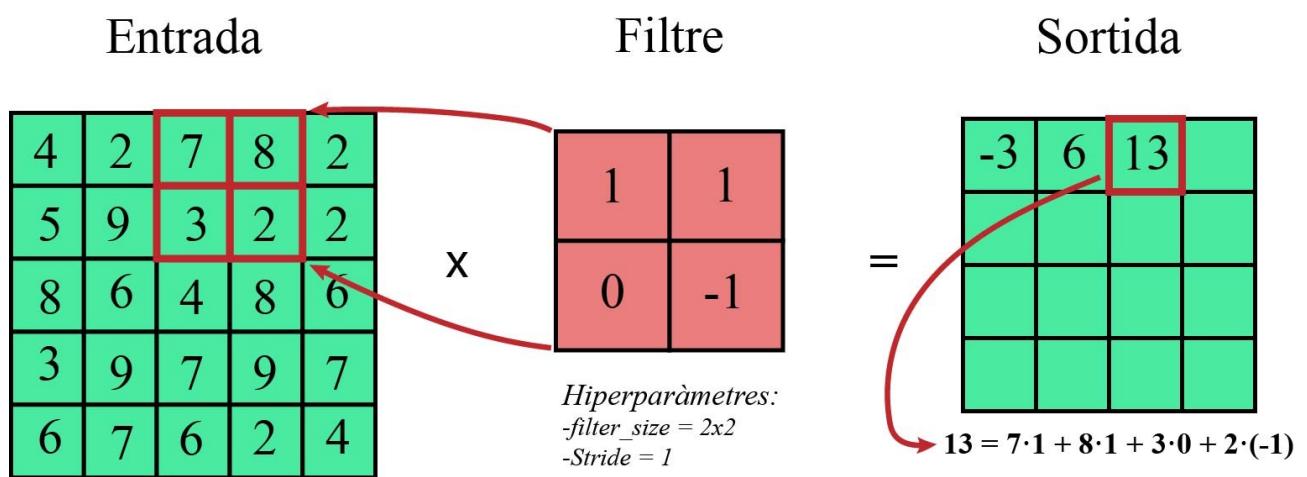
3.5.2 CNN (Convolutional Neural Networks)

Com s'ha dit amb anterioritat, les xarxes neuronals convolucionals són xarxes pensades per al tractament d'imatges. El seu funcionament és una mica diferent que el de les capes densament connectades. Així que abans de posar-les en pràctica explicarem el seu funcionament.

Les xarxes neuronals convolucionals funcionen molt bé per a la identificació i tractament d'imatges perquè tenen la capacitat d'aprendre formes i patrons a les imatges que després poden generalitzar. És a dir, amb una cara per exemple, una xarxa neuronal convolucional entrenada és capaç de detectar ulls, boca, nas... i les seves proporcions per a després poder reconèixer la cara igual que ho fem els humans.

Una convolució és allò que es fa en cadascuna de les capes convolucionals. Es tracta d'agafar una matriu de pesos, anomenada filtre, que va fent salts de esquerra a dreta i de dalt a baix fins a recórrer tota la imatge d'entrada. El nombre de píxels que avança de dreta a esquerra i quan acaba la fila de dalt a baix és un hiperparàmetre que podem definir anomenat «stride». Posem un exemple d'una foto de 28 píxels com a entrada. Si tenim un filtre de 5x5, obtindrem una matriu associada a la foto d'entrada i al filtre de 24x24 (el filtre només es podrà moure 23 cops a la dreta i a baix abans d'arribar al final de la foto). Això ho haurem d'aplicar per tots els filters que vulguem tenir, per exemple 32 en aquest cas explicatiu.

Aquí podeu veure una convolució d'una imatge de 5x5 amb un filtre de 2x2, aquests són valors molt petits que no tenen aplicació real, però ens van bé per a poder-ho entendre.



Reprenent l'exemple anterior, veiem que la primera capa oculta tindrà un total de 576 neurones repartides en 32 filters. El valor de cada punt de la primera capa oculta és el resultat del producte escalar entre el filtre i cada conjunt de neurones de la capa d'entrada.

Així doncs l'input de la primera capa havia estat de $28 \times 28 \times 1$ i l'output serà de $24 \times 24 \times 32$.

Seguidament aplicarem l'operació pooling, que consisteix en simplificar la matriu de sortida de la capa convolucional. Per exemple, amb un pooling de 2×2 simplificarem la matriu anterior de $24 \times 24 \times 32$ a una de $12 \times 12 \times 32$.

Amb la concatenació de capes convolucionals i capes de pooling (solen anar una darrera l'altra) el que estem fent és que a mesura que anem ampliant el nombre de filters reduint la resolució d'aquests mateixos. Això fa que la primera capa trobi més els detalls de les imatges i a mesura que ens anem apropiant al final la xarxa aprengui els trets més generals.

Les capes convolucionals normalment acaben amb una capa flatten() que ens converteix el vector de sortida de les convolucionals a un de tan sols dues dimensions per a poder-hi concatenar un seguit de capes denses que a partir de les descripcions en forma de vectors creades per les convolucionals puguin classificar les imatges d'entrada.

3.5.2.1 Chest X-Ray Images (Pneumonia) dataset

Per tal de posar en pràctica i acabar d'entendre com funcionen les xarxes neuronals convolucionals farem una pràctica amb un dataset de radiografies pulmonars. Així com a l'anterior pràctica fèiem servir dades de la biblioteca Keras a mode simplement introductiu, ara farem servir un dataset molt més pròxim al món real. Tant aquest com el següent estan extrets de la pàgina web kaggle⁴, una mena de xarxa social de datasets oberta a tot el públic. En ella pots navegar entre milers de datasets amb els que alimentar moltíssims projectes de deep learning. En concret farem servir «chest-xray-pneumonia»⁵, un dataset de radiografies amb pneumònia i sanes.

Primer de tot, declarem els mòduls que farem servir:

- Tensorflow 2.x
- Keras
- os
- zipfile

Un cop aclarit aquest punt, comentarem pas per pas tot el programa. El primer que haurem de fer serà importar el dataset. Com que estem treballant amb Colab, tenim dues opcions: pujar el dataset al Colab cada cop que vulguem fer servir el programa (cosa que és molt pesada perquè et tanquen sessió després de pocs minuts d'inactivitat) o bé tenir els fitxers al Drive i importar-los. Farem servir la segona, així doncs muntem l'extensió del Google Drive:

```
from google.colab import drive  
drive.mount('/content/drive')
```

Ara que ja tenim el Drive, hem de trobar els fitxers i descomprimir-los:

```
local_zip = '/content/drive/My Drive/Treball de recerca/Datasets/17810_23812_bundle_archive.zip'  
zip = zipfile.ZipFile(local_zip, 'r')  
zip.extractall('/content')  
zip.close()
```

⁴ www.kaggle.com

⁵ <https://www.kaggle.com/paultimothymooney/chest-xray-pneumonia?>

Com que l'estructura del fitxer està marcada per carpetes (carpeta train, validation i test) i dins de cadascuna tenim una carpeta amb radiografies normals i una altra amb radiografies amb pneumònia, declarem una variable amb cadascun dels directoris per a facilitar-nos la feina i posteriorment imprimim el número de fotos que tenim a cada carpeta per veure la forma del dataset.

```
dir = '/content/chest_xray'

train_dir = os.path.join(dir, 'train')
train_normal_dir = os.path.join(dir, 'train/NORMAL')
train_pneumonia_dir = os.path.join(dir, 'train/PNEUMONIA')

validation_dir = os.path.join(dir, 'val')
validation_normal_dir = os.path.join(dir, 'val/NORMAL')
validation_pneumonia_dir = os.path.join(dir, 'val/PNEUMONIA')

test_dir = os.path.join(dir, 'test')
test_normal_dir = os.path.join(dir, 'test/NORMAL')
test_pneumonia_dir = os.path.join(dir, 'test/PNEUMONIA')

print('total training normal images :', len(os.listdir(train_normal_dir)))
print('total training pneumonia images :', len(os.listdir(train_pneumonia_dir)))

print('total validation normal images :', len(os.listdir(validation_normal_dir)))
print('total validation pneumonia images :', len(os.listdir(validation_pneumonia_dir)))

print('total test normal images :', len(os.listdir(test_normal_dir)))
print('total test pneumonia images :', len(os.listdir(test_pneumonia_dir)))
```

output:

```
total training normal images : 1341
total training pneumonia images : 3875
total validation normal images : 8
total validation pneumonia images : 8
total test normal images : 234
total test pneumonia images : 390
```

Ara a l'observar el dataset veiem que no està balancejat, és a dir, que no té les mateixes dades d'entrenament per a totes les sortides, cosa que pot provocar biaixos i un funcionament erroni del model. Per a solucionar-ho, tenim un parell d'opcions: el que s'anomena DataAugmentation o ajustar el class_weight. El DataAugmentation consisteix en rotar o invertir les imatges per a fer-ne copies diferents, aquesta tècnica funciona molt bé quan tenim un dataset petit però balancejat i el volem augmentar, però no és el més òptim per a inflar alguna de les sortides en la que tenim poques dades.

Així doncs, optarem per ajustar el class_weight, que consisteix en ponderar les sortides, és a dir, li diem al model quantes dades tenim de cadascuna de les classes perquè ell ponderi els errors després. Per a calcular el class_weight es segueix la següent fórmula:

$$\text{classweight}_i = \frac{\text{number of samples}}{\text{number of classes} \cdot \text{dades classe}_i}$$

Això aplicat al nostre dataset és:

```
weight_for_0 = (len(os.listdir(train_normal_dir)) + len(os.listdir(train_pneumonia_dir))) / (2 * len(os.listdir(train_normal_dir)))

weight_for_1 = (len(os.listdir(train_normal_dir)) + len(os.listdir(train_pneumonia_dir))) / (2 * len(os.listdir(train_pneumonia_dir)))

class_weight = {0: weight_for_0, 1: weight_for_1}

print('Weight for class 0: {:.2f}'.format(weight_for_0))
print('Weight for class 1: {:.2f}'.format(weight_for_1))
```

Ara que ja hem arreglat el problema del balanç de dades, crearem a partir de les carpetes amb imatges que tenim tensors amb els quals puguem entrenar el nostre model. Això ho fem mitjançant la utilitat `flow_from_directory()` de Keras (per a importar les imatges) juntament amb `ImageDataGenerator` (per a normalitzar-les). Un dels paràmetres de `flow_from_directory` és la mida de les fotos que vulguem importar, en aquest cas li diem que ens les retallí totes a 200x200 píxels.

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

train_datagen = ImageDataGenerator(rescale=1.0/255.)
validation_datagen = ImageDataGenerator(rescale=1.0/255.)
test_datagen = ImageDataGenerator(rescale=1.0/255.)

train_generator = train_datagen.flow_from_directory(train_dir,
                                                    batch_size=20,
                                                    class_mode='binary',
                                                    target_size=(200, 200))

validation_generator = validation_datagen.flow_from_directory(validation_dir,
                                                               batch_size=20,
                                                               class_mode='binary',
                                                               target_size=(200, 200))

test_generator = test_datagen.flow_from_directory(test_dir,
                                                   batch_size=20,
                                                   class_mode='binary',
                                                   target_size=(200, 200))
```

Un cop tenim totes les dades preparades, definirem el model que utilitzarem:

```
import tensorflow as tf
from tensorflow.keras import Model

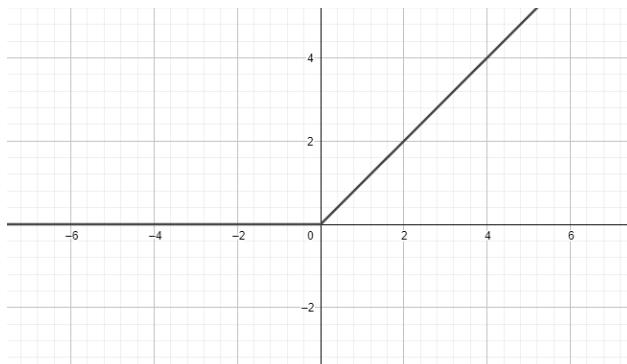
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense

model = Sequential()
model.add(Conv2D(32, (3,3), activation='relu', input_shape=(200, 200, 3)))
model.add(MaxPooling2D(2, 2))
model.add(Conv2D(64, (3,3), activation='relu'))
model.add(MaxPooling2D(2, 2))
model.add(Conv2D(128, (3,3), activation='relu'))
model.add(MaxPooling2D(2, 2))
model.add(Flatten())
model.add(Dense(64, activation='relu'))
model.add(Dense(64, activation='relu'))
model.add(Dense(64, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.summary()
```

Aquí cal veure com fem servir una funció d'activació que no havíem fet servir abans, la relu. És molt utilitzada pel resultat que dona i el fàcil que és de computar, la seva equació és:

$$f(x) = \begin{cases} x & : x > 0 \\ 0 & : x < 0 \end{cases}$$

I la seva representació gràfica és:



Havent aclarit això ja simplement ens queda compilar i entrenar el model:

```
from tensorflow.keras.optimizers import Adam

model.compile(optimizer=Adam(lr=1e-4),
              loss='binary_crossentropy',
              metrics = ['acc'])

history = model.fit(
    train_generator,
    epochs=3,
    validation_data=validation_generator,
    class_weight=class_weight,
    verbose=1)
```

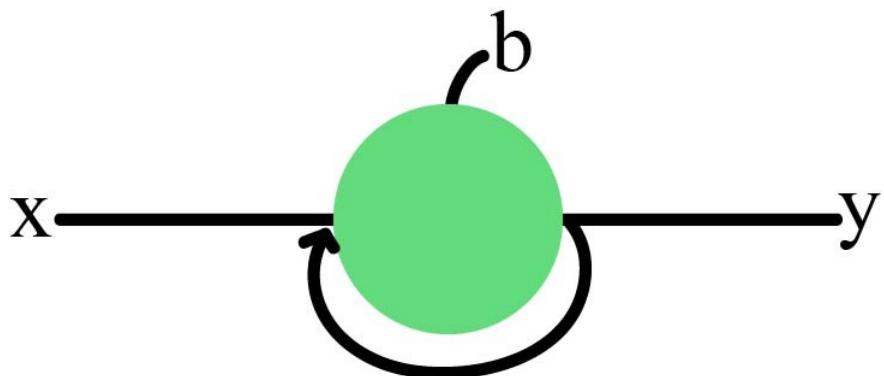
Un cop acabat l'entrenament veiem que obté un 93,7% d'encerts a en validation i prop d'un 95% en train. Ara anem a provar-lo en el dataset que encara no ha vist per a comprovar la fiabilitat:

```
test_lost, test_acc= model.evaluate(test_generator)
print("TestAccuracy:", test_acc)
```

Un cop s'ha executat, veiem que obté més o menys un 85% d'encerts sobre un banc d'unes 500 mostres, que és un bon resultat partint des del punt de vista que algú que no hagi estudiat medicina, mirant les fotos directament li costa molt diferenciar entre unes i altres, almenys en el meu cas.

3.5.3 RNN (Recurrent Neural Networks)

Les xarxes neuronals recurrents (Recurrent Neural Networks) són un altre tipus de xarxes molt comunes. Tal com s'ha dit anteriorment, aquestes es fan servir generalment en el tractament de text. Per què? Bé, el que fa especial a una RNN es la seva capacitat de tractar amb la dimensió temporal, la qual cosa aconsegueixen introduint la sortida anterior ponderada en la següent iteració. Una sola neurona es veuria així:



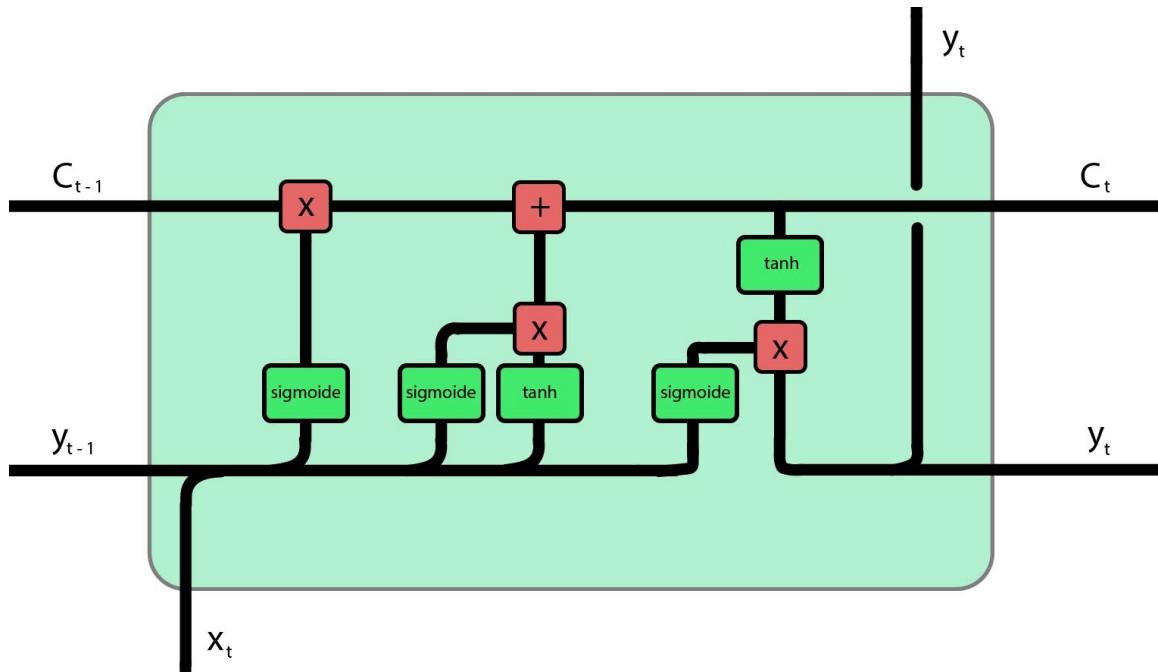
Com veiem, els resultats d'una RNN depenen de les iteracions anteriors, per això va bé per a tractament de text, ja que per a escriure una lletra o una paraula nova té una noció de les que s'han escrit anteriorment. Òbviament l'equació de sortida de cada neurona es veu modificada a la d'una neurona normal i té la següent forma:

$$y_t = f(w_t \cdot x_t + w_{t-1} \cdot y_{t-1} + b)$$

Com veiem, aquesta estructura és molt senzilla i realment funciona, però té un petit problema: tendeix a oblidar molt fàcilment matisos de generacions passades que de vegades són necessaris. Per exemple, si tenim la frase «a la carretera hi ha molts», la RNN podrà fàcilment afegir «cotxes» correctament a l'oració, però si per exemple diem «sóc català» a l'inici d'un text i després d'un paràgraf tenim «la meva llengua materna és el» una RNN no recordarà aquest concepte que havíem dit anteriorment i segurament no respondrà bé «català». És en aquest context on neixen xarxes recurrents més complexes com les LSTMs, les més famoses.

3.5.3.1 LSTM

Les LSTMs (Long-Short Term Memory) són neurones que neixen amb l'única finalitat de poder recordar aquests conceptes o matisos passats, això comporta que la seva estructura sigui bastant més complexa. Una LSTM consta de 3 entrades: l'entrada de la iteració actual, la sortida de l'anterior i el que s'anomena «cell state». Aquest cell state és la clau que diferencia a les LSTM: és una mena de cinta transportadora que va passant de generació en generació amb petites modificacions. L'esquema seria el següent.



Així doncs, primer ens fixarem en el carril que calcula la sortida y_t . Com veiem agafem la sortida anterior (y_{t-1}) i l'entrada de l'actual iteració (x_t) i la passem per la sigmoide. Un cop hem obtingut el resultat el multipliquem pel cell state (C_t) en composició amb la tangent hiperbòlica.

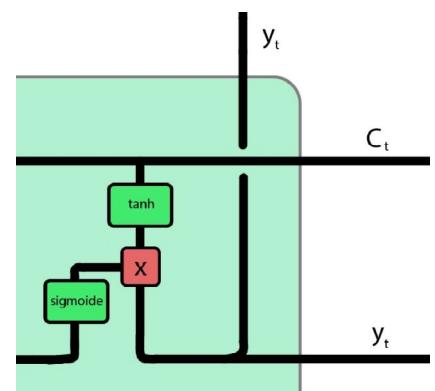
La tangent hiperbòlica és una altra funció d'activació amb la mateixa forma que la sigmoide, però amb diferent rang: en aquest cas és $(-1, 1)$ i no $(0, 1)$. La seva equació és:

$$\tanh = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

D'aquesta manera, la sortida de la nostra neurona serà:

$$y_t = \text{sigmoide}(w \cdot (y_{t-1} + x_t) + b_t) \cdot \tanh(C_t)$$

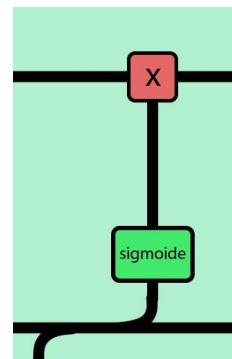
On C_t és el cell state que pren qualsevol valor real. Ara bé, com el calculem?



Veiem que hi ha dos procediments a fer. El primer s'anomena «forget gate» i com el seu nom indica ens permetrà decidir que s'oblida del cell state.

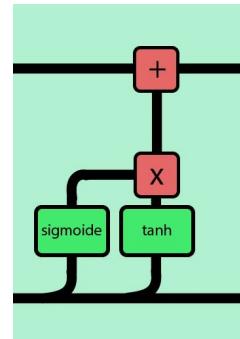
Com veiem, composem la funció sigmoide amb la sortida anterior i l'entrada actual sumada al bias de la iteració actual.

$$C_t = C_{t-1} \cdot \text{sigmoide}(w(y_{t-1} + x_t) + b_t)$$



Ara bé, encara hem de fer un procediment més per a obtenir el nostre cell state de l'actual generació: aquest l'anomenem «input gate». Consisteix en agafar la sortida anterior i l'entrada i multiplicar-les passades per una sigmoide i una tangent hiperbòlica per a després sumar-ho tot al cell state que teníem. Això matemàticament és:

$$C_t = C_{t-1} + [\text{sigmoide}(w(y_{t-1} + x_t) + b_t) \cdot \text{sigmoide}(w(y_{t-1} + x_t) + b_t)]$$



Finalment, ja tenim el cell state que ens faltava per a calcular la sortida de l'actual generació i per tant de la següent i així successivament.

3.5.3.2 Wikipedia-sentences dataset

Ara, com hem fet anteriorment, farem una petita pràctica de «NLP» (Natural Language Processing), és a dir, tractament de text. Per a poder dur-ho a terme, utilitzarem un dataset de la pàgina web kaggle.com ho hem fet anteriorment. En aquest cas, farem servir un dataset anomenat «wikipedia-sentences»⁶ en el que trobem al voltant de 7,8 milions de frases. El seu creador ens diu que ha agafat la base de dades de la Wikipedia i l'ha netejat amb tasques com eliminar les frases amb menys de 3 o més de 255 caràcters i ha eliminat les frases que quedaven duplicades. També ens diu que estan ordenades alfabèticament i que cada línia del .txt és una frase. Així doncs, ens ha facilitat molta feina de tractament de dades.

Com ja hem fet anteriorment, veiem quins mòduls son necessaris per al projecte:

- Tensorflow 2.x
- Keras
- os
- Time
- Random

⁶ <https://www.kaggle.com/mikeortman/wikipedia-sentences>

Igual que abans, començarem important el fitxer des del Google Drive.

```
from google.colab import drive
drive.mount('/content/drive')
```

El primer cop haurem de retallar el dataset, ja que és molt gran i el Google Colab es queda sense ram a l'intentar processar-lo. Aquest petit script modifica el fitxer d'arrel, per tant només s'ha d'executar el primer cop, ja que si no anirem fent cada cop el dataset més petit.

```
path_to_fileDL ='/content/drive/My Drive/Treball de recerca/Datasets/wikisent2.txt'
lines = open(path_to_fileDL).readlines()
random.shuffle(lines)
del lines[len(lines)//20:]
open(path_to_fileDL, 'w').writelines(lines)
```

A partir d'aquí sí que haurem d'executar-ho tot cada cop que vulguem obrir sessió al Colab. Primerament obrim el fitxer i imprimim tots els diferents caràcters que el contenen:

```
path_to_fileDL ='/content/drive/My Drive/Treball de recerca/Datasets/wikisent2_20.txt'

text = open(path_to_fileDL, 'rb').read().decode(encoding='utf-8')
print('Longitud del text: {} caràcters'.format(len(text)))

vocab = sorted(set(text))

print ('El text està compost d'aquests {} caràcters:'.format(len(vocab)))
print (vocab)
```

output:

```
Longitud del text: 46677954 caràcters
El text està compost d'aquests 96 caràcters:
['\n', ' ', '!', '"', '#', '$', '%', '&', "''", '(', ')', '*', '+', ':', ',', ';', '<', '=', '>', '/',
'0', '1', '2', '3', '4', '5', '6', '7', '8', '9', '?', '@', 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z', '[', '\\', ']', '^', '`', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z', '{', '[', '}', '^', '~']
```

Com que al model no li podem donar caràcters com a input, hem de crear un diccionari relacionant tots aquests caràcters amb un número:

```
char2idx = {u:i for i, u in enumerate(vocab)}
idx2char = np.array(vocab)

for char, _ in zip(char2idx, range(len(vocab))):
    print(' {:4s}: {:3d}'.format(repr(char), char2idx[char]))
```

output:

```
'\n': 0, ' ': 1, '!': 2, ..... '{': 92, '|': 93, '}': 94, '~': 95,
```

Ara que ja tenim el diccionari, anem a utilitzar-lo per a convertir tot el text que teníem en un inici a números:

```
text_as_int = np.array([char2idx[c] for c in text])
```

Al tenir el text en una numpy array, el convertirem en un tensor amb l'eina Dataset.from_tensor_slices de TensorFlow. Farem seqüències de 100 caràcters per a introduir-les al model.

```
char_dataset = tf.data.Dataset.from_tensor_slices(text_as_int)
seq_length = 100
sequences = char_dataset.batch(seq_length+1, drop_remainder=True)
```

Com que volem que el nostre model predigi paraules, és a dir, que donant-li una lletra o una paraula ell continuï escrivint amb normalitat, agafarem frases i li traurem la primera lletra perquè el model vagi predient. D'aquesta manera tot tenint la frase completa com a output desitjat el model anirà rectificant i per tant, aprenent a escriure. Això ho farem amb la funció map().

```
def split_input_target(chunk):
    input_text = chunk[:-1]
    target_text = chunk[1:]
    return input_text, target_text

dataset = sequences.map(split_input_target)
```

Ara organitzarem les dades en batch i les mesclarem totes perquè a cada batch tinguem varietat de seqüències. El paràmetre buffer_size serveix per indicar cada quant fa grups aleatoris. És a dir, si tinguéssim 100 imatges de gats i 100 de gossos (ordenades primer les de gats i després les de gossos) i poséssim un buffer_size de 100, barrejaria entre elles les fotos de gats i entre elles les fotos de gossos, cosa que no volem, ja que continuaran unes darrere les altres. Així doncs, ens hem d'assegurar que el buffer_size sigui tan gran com dades tinguem sempre que sigui possible.

```
BATCH_SIZE = 64
BUFFER_SIZE = len(text)
dataset = dataset.shuffle(BUFFER_SIZE).batch(BATCH_SIZE, drop_remainder=True)
```

Seguidament haurem de definir el model, abans de la capa LSTM, haurem de definir un embedding que molt resumidament és una capa que ens transformarà els nostres números d'entrada en vectors bidimensionals que pot tractar la LSTM. L'entrada de l'embedding serà el número de caràcters possibles i les seves dimensions són un hiperparàmetre que podem definir segons convinència. Com podem veure, la capa LSTM té un parell de paràmetres. El return_sequences està lligat a que el model ens retorna una sortida per cada iteració mentre que stateful es refereix al fet que com que els nostres batch tenen relació entre si, al començar el batch nou faci servir com a cell state inicial el cell state de l'últim batch.

Finalment farem servir una capa densa per a trobar una sortida.

```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Embedding, LSTM, Dense

vocab_size = len(vocab)
embedding_dim = 256
LSTM_units = 1024

model = Sequential()
model.add(Embedding(input_dim=vocab_size,
                     output_dim=embedding_dim,
                     batch_input_shape=[BATCH_SIZE, None]))
model.add(LSTM(LSTM_units,
               return_sequences=True,
               stateful=True))
model.add(Dense(vocab_size))
model.summary()
```

Un cop definit el model toca compilar-lo. Farem servir la sparse_categorical_crossentropy com a funció cost, ja que realment volem classificar entre diversos caràcters, és una funció molt comuna. Ara bé, el detall està en l'argument from_logits: l'hem de tenir com a True perquè no estem tractant amb una distribució probabilística com a sortida (com quan fem servir softmax) sinó que hem de classificar valors de versemblança. La suma de probabilitats de totes les possibles lletres no ha de donar 1, perquè el que hem de mesurar és com de bé queda aquella lletra allà, per tant pot haver-hi més d'una opció.

```
def loss(labels, logits):
    return tf.keras.losses.sparse_categorical_crossentropy(labels, logits, from_logits=True)

model.compile(optimizer='adam', loss=loss)
```

Ara ja només queda entrenar. Al acabar, guardem el valor dels pesos pel que veurem just a continuació.

```
EPOCHS=5
history = model.fit(dataset, epochs=EPOCHS, verbose= 1)
weights = model.get_weights()
```

El model que hem entrenat té definit un batch_size de 64, això vol dir que espera com a input aquest batch_size de 64. Ara bé, al fer inferència, és a dir, al provar el model, aquest batch_size haurà de ser de només 1, ja que voldrem que a partir d'una sola lletra completi frases i no pas que l'hi introduïm 64 lletres perquè completi 64 frases. Així doncs, haurem de tornar a definir un model per a inferència amb aquest hiperparàmetre, el batch_size, a 1.

```
model_prediction = Sequential()
model_prediction.add(Embedding(input_dim=vocab_size,
                               output_dim=embedding_dim,
                               batch_input_shape=[1, None]))
model_prediction.add(LSTM(LSTM_units,
                         return_sequences=True,
                         stateful=True, ))
model_prediction.add(Dense(vocab_size))
model_prediction.set_weights(weights)
```

Un cop tenim això fet, hem de crear la funció que ens generi text a partir d'una lletra o una paraula, emmagatzemada en la variable start_string. Primer amb la variable num_generate li diem el nombre de caràcters que volem que generi, seguidament convertim start_string a números amb el nostre diccionari. Ara introduïm la temperatura. La temperatura és alguna cosa semblant al grau de llibertat del model. Si és molt elevat (sobre 1) tindrà tanta llibertat que cometrà faltes ortogràfiques i s'inventarà paraules... Si és molt baix (prop de 0) entrarà en un bucle per falta de capacitat creativa. El reset_states ens reinicia el cell_state de manera que no tingui influències que poden ser negatives de la fase d'aprenentatge.

Després tenim el bucle: cada iteració ens afegirà una lletra a la string de sortida. Primer comença passant-li al model input_eval, que conté totes les sortides anteriors i aquest retorna una matriu de versemblança.

A aquesta matriu se li suprimeix la dimensió de size 1 (passem de tenir una matriu «`[[]]`» a una amb forma «`()`») i es divideix entre la temperatura. Un cop fet això ja simplement passem la matriu pel `tf.random.categorical` que ens tria la següent lletra que és afegida al input_eval i a la string de text generat.

```
def generate_text(model, start_string):
    num_generate = 300
    input_eval = [char2idx[s] for s in start_string]
    input_eval = tf.expand_dims(input_eval, 0)
    text_generated = []
    temperature = 0.5
    model.reset_states()

    for i in range(num_generate):
        predictions = model_prediction(input_eval)
        predictions = tf.squeeze(predictions, 0)
        predictions = predictions / temperature
        predicted_id = tf.random.categorical(predictions, num_samples=1)[-1,0].numpy()
        input_eval = tf.expand_dims([predicted_id], 0)
        text_generated.append(idx2char[predicted_id])

    return (start_string + ''.join(text_generated))
```

Finalment, ara que ja tenim la funció, només l'hem de cridar amb la start_string pertinent.

```
print(generate_text(model, start_string='text inicial'))
```

Si provem amb diferents entrades inicials:

- Football:

```
Football Championship and became officially recognized by the Department of Historic Places in 1988.  
He is also the first for the school and industrial election of 1996.  
The school was founded in 1993 and was a former American football club from Australia.  
He was appointed to the public to make a tennis tou
```

- Obama:

```
Obama since 1966 by South Henry IV, and is currently owned by Danny Crawford in 1999.  
A constant can be conserved on the characters in the modern star league to the Canadian Cup competition.  
As of 2006, the institution has been elected to participate in the top ten final appearances for the county season
```

- Amazon:

```
Amazon Company is a private hotel in the Cape County Courthouse in the U.S. state of West Virginia. It is also the first for an American main character of the state association of the United States Army Air Force (NRL). The song was produced by Stephen Kershaw and was released by the University of Dance
```

Com es pot veure després de provar amb diverses frases d'inici, el model tendeix a escriure sorprendentment bé tot i el simple que és. És a dir, totes les frases no tenen sentit per si mateixes, però veiem com no fa faltes d'ortografia i compleix bé normes bàsiques com són posar articles o l'ordre dins de l'oració entre subjectes i predicats. Per tant, podem dir que el resultat és molt satisfactori. Cal recordar que el programa es troba al repositori de GitHub⁷ i és tan fàcil com penjar-lo al Drive i obrir-lo amb el Colab per a provar-lo.

⁷ <https://github.com/bielaltes/TdR>

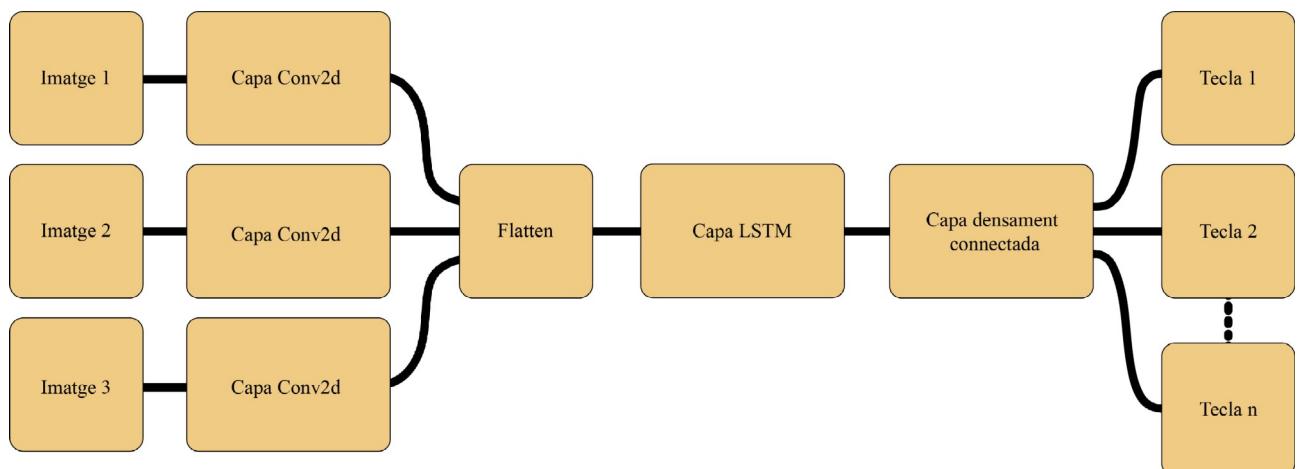
4 Projecte final: IA condueix en videojocs

En aquesta part del treball de recerca programarem amb TensorFlow una intel·ligència artificial que juga a jocs de conduir. En aquest cas s'utilitzarà el famós simulador de conducció Assetto Corsa.

La idea principal és crear un dataset amb un conjunt d'imatges amb tecles associades a partir d'una persona conduint per a poder-les introduir a una xarxa neuronal i que així aprengui a conduir tal com ho fa un humà. Així doncs, el primer que s'ha de fer és el programa encarregat de crear el dataset. Arribat a aquest punt, ens trobem amb un problema: si tu veus una imatge aïllada d'una carretera, sabràs si has de frenar, accelerar o girar? La resposta és no.

Nosaltres tenim dos ulls per a poder calcular distàncies, al tenir dos ulls tenim dos punts de vista diferents, i d'aquesta manera el nostre cervell pot crear una imatge tridimensional de la zona on ens trobem. Això amb una imatge no es pot aconseguir. Davant de l'impossibilitat d'obtenir dos punts de vista diferents en el joc, farem grups de 3 imatges espaiades en el temps amb una tecla associada (la que s'estigui prement en el moment de captura de l'última imatge) perquè així la IA pugui aprendre el concepte de proximitat i també el de velocitat.

Un cop tenim això haurem de preparar la xarxa neuronal en sí. Per a poder tractar amb imatges distribuïdes en el temps farem servir la capa TimeDistributed de Keras. Cadascuna de les imatges passarà per un seguit de capes convolucionals i seguidament una capa flatten la qual ens descriurà cadascuna de les imatges amb un vector bidimensional. Quan tinguem el vector bidimensional de cadascuna de les imatges, passarem els 3 vectors per una capa de neurones recurrents que ens donaran una sola sortida interpretada per un seguit de capes densament connectades per a acabar deliberant quina tecla s'ha de premer. Gràficament això té la següent forma:



Com hem vist abans les RNN són xarxes dedicades a la comprensió i escriptura de textos en general. Això és degut al fet que integren una petita memòria temporal que ens permet recordar sortides anteriors. Quan tractem amb RNN i textos abans hem de crear embeddings, és a dir, convertir les paraules o les lletres en vectors. Si ens hi fixem, estem fent el mateix però amb imatges, és a dir, estem convertint les imatges en vectors tal com ho faríem amb un text a través de les CNN.

Finalment, després del procés d'entrenament, necessitem un programa que ens permeti en temps real capturar seqüències d'imatges per a introduir a la xarxa fent inferència i que a partir de la predicció de la xarxa premi en cada moment la tecla corresponent.

Cal dir que per a executar aquest programa ja no farem servir el Google Colab com fins ara, ja que al executar-se als servidors de Google, no li podem administrar en temps real el que està capturant el nostre programa. Ho haurem d'executar en local. Així doncs, haurem de tenir instal·lat TensorFlow i totes les dependències necessàries. En concret, necessitem TensorFlow-gpu que ens permetrà executar la xarxa neuronal en la nostra targeta gràfica de manera molt més eficient que amb una cpu (les gràfiques són molt bones multiplicant matrius, ja que estan pensades per a això). Necessitarem una gràfica amb els seus determinats drivers, en el cas de Nvidia, el Nvidia Cuda Toolkit i CUDNN i en el cas de AMD, ROCm. Cal dir que els drivers per a xarxes neuronals d'AMD només estan disponibles en Linux i el suport està molt limitat, per tant no podem fer ús d'una gràfica AMD.

En aquest cas per a executar la xarxa es farà servir un entorn Windows 10 amb: una Nvidia RTX 2060 amb 6gb de vram (no es recomanen menys de 6gb, si no no podrem carregar la xarxa neuronal) i un ryzen 5 2600 (la cpu no ha de ser necessàriament potent) junt amb 16gb de ram a 3200mhz (la ram si que és important a l'hora de carregar el dataset, no es recomanen menys de 16gb).

A continuació, trobarem:

- Programa de creació del dataset: el programa que ens crearà el dataset d'entrenament.
- Programa de definició i entrenament de la xarxa neuronal: configurarem la xarxa neuronal i l'entrenarem amb el dataset que hem creat.
- Programa d'execució en temps real: executarem el model en temps real.
- Proves: provarem el model en diferents entorns.
 - Proves primer model: provarem el model i en veurem les seves limitacions.
 - Millores al primer model: intentarem pal·liar les limitacions.
 - Proves model millorat: provarem el model millorat.

4.1 Programa de creació del dataset

Com hem dit aquest programa s'encarregarà d'anar creant el dataset mentre nosaltres anem jugant. Per a programar-lo, farem servir els següents mòduls o dependències.

- Time
- cv2
- numpy
- win32api (aquest mòdul no és instal·lable a través del sistema de gestió de paquets python pip)
- grabber (petit script que podem importar com a mòdul posant-lo a la mateixa carpeta que el programa, ens permetrà fer captures de pantalla de manera molt més ràpida que amb PIL o cv2; aquest script es troba al github amb les respectives referències als programadors)

Així doncs, ara que tenim vistos els requisits anem amb l'estructura del programa. El programa consta de diverses funcions com la que fa les captures de pantalla, la que mira quines tecles s'estan prement, etc. La funció `__main__()`, és la que cridarem per a iniciar el programa i la que s'encarregarà de fer de director d'orquestra. Seguidament anirem mostrant cadascuna de les funcions així com per a què serveixen. El codi complet es pot trobar al GitHub mostrat anteriorment.

Aquesta funció s'encarrega de buscar antics fitxers per a continuar escrivint en ells i no haver de crear tot el dataset de cop. En el cas que no trobi el fitxer `training_data`, el que fa és crear una array nova on començar a enregistrar les dades.

```
def data_finder():

    global old_training_data_balanced
    global training_data

    previous_data = "training_data.npy"

    if os.path.isfile(previous_data):
        print("Loading previous data")
        old_training_data_balanced = list(np.load(previous_data, allow_pickle=True))
    else:
        print("Starting new files")
        training_data = []
```

Quan es crida aquesta funció, es registra una captura de pantalla amb les mesures 1,24,801,626, per tant de 800x600 i mitjançant `cv2.resize` es redimensiona a una imatge de 200x150, una mesura molt més assequible per una xarxa neuronal entrenada amb el hardware que tenim disponible.

```
def get_grb():
    grb = Grabber(bbox=(1, 26, 801, 626))
    grabbedscreen = None
    grabbedscreen = grb.grab(grabbedscreen)
    img_resized = cv2.resize(grabbedscreen, (200, 150))
    final_img = np.array(img_resized)
    return final_img
```

Amb aquesta funció el que fem és capturar les tecles que estan premudes en el moment que la cridem i ens les retorna en forma d'array.

```
keyList = []
for char in "WASDP":
    keyList.append(char)

def key_check():
    keys = []
    for key in keyList:
        if win32.GetAsyncKeyState(ord(key)):
            keys.append(key)
    return keys
```

Un cop tenim les tecles premudes en forma d'array [a,b,c...] les hem de codificar utilitzant la tècnica one-hot de manera que les pugui classificar una xarxa neuronal. Això vol dir que a la funció hi introduirem les tecles premudes i ens retornarà un vector del tipus [0,0,0,...] amb un 1 a la posició que correspon a la tecla o combinació de tecla premuda.

```
def key_to_multihot():
    multihot = [0,0,0,0,0]
    keys = key_check()
    if len(keys) == 1:
        if 'W' in keys:
            multihot[0] = 1
        elif "P" in keys:
            multihot[4] = 1
        else:
            multihot[3] = 1
    elif len(keys) == 2:
        if "W" and "A" in keys:
            multihot[1] = 1
        elif "W" and "D" in keys:
            multihot[2] = 1
        else:
            multihot[3] = 1
    else:
        multihot[3] = 1

    return multihot
```

Quan hem acabat l'entrenament, cridem a la funció balance_data. Serveix per a equilibrar el dataset, si no ho féssim, com que la majoria de temps estem accelerant, la IA aprendria que sempre ha de tirar endavant, ja que així encertaria una bona part dels cops i per tant es minimitzaria l'error. Així doncs, necessitem un programa que ens equilibri el nombre de seqüències de cada tecla premuda. Amb les que sobrin, simplement les eliminarem. Acabem fent un shuffle() per a barrejar totes les seqüències.

```
def balance_data():
    global training_data
    global training_data_balanced
    global old_training_data_balanced
    global new_training_data_balanced
    w = []
    wa = []
    wd = []

    for data in range(len(training_data)):
        img = training_data[data][0]
        output = training_data[data][1]

        if output == [1,0,0,0,0]:
            w.append([img, output])
        elif output == [0,1,0,0,0]:
            wa.append([img, output])
        elif output == [0,0,1,0,0]:
            wd.append([img, output])
        else:
            print("error")

    w = w[:len(wa)][:len(wd)]
    wa = wa[:len(w)]
    wd = wd[:len(w)]

    print(len(w))
    print(len(wa))
    print(len(wd))

    training_data_balanced = w+wa+wd
    new_training_data_balanced = training_data_balanced + old_training_data_balanced
    random.shuffle(new_training_data_balanced)

    np.save('training_data.npy',new_training_data_balanced)
    print("Finished")
```

Finalment hem arribat a la funció `_main_()`. Aquesta funció es la que s'encarrega de cridar ordenadament a totes les altres i per tant podríem dir que és el cervell del nostre programa. Com podem veure cada `1/capturerate` capture una imatge i quan en porta 3, mira la tecla associada. Tot això ho va emmagatzemant en una llista i al acabar l'entrenament simplement balanceja el dataset i el guarda en un fitxer numpy.

```
def __main__():
    global training_data
    data_finder()
    capturerate = 30
    running = bool
    img_seq = []
    training_data = []
    time.sleep(10)
    print("Starting...")
    while running:
        for i in range(3):
            last_time = time.time()
            img_seq.append(get_grb())
            if i > 2:
                waittime = (1.0/capturerate)-(time.time()-last_time)
                if waittime>0.0:
                    time.sleep(waittime)
            img_seq = []
            multihot = key_to_multihot()
            if multihot[3] == 1:
                pass
            elif multihot[4] == 1:
                running = False
                print(len(training_data))
                balance_data()

        else:
            training_data.append([img_seq, multihot])

        waittime = (1.0/capturerate)-(time.time()-last_time)
        if waittime>0.0:
            time.sleep(waittime)

__main__()
```

4.2 Programa de definició i entrenament de la xarxa neuronal

Abans de començar, definirem les dependències que hem de tenir en el nostre ordinador per a poder executar el programa.

- Tensorflow 2.x
- Keras (farem servir la versió que incorpora tensorflow)
- numpy

Ara, un cop tenim el dataset, hem de definir la xarxa neuronal que farem servir i posteriorment entrenar-la.

Abans però, no se'ns pot oblidar normalitzar les dades. Cada conjunt de píxels RGB té un valor entre 0 i 255. Això pot ser difícil d'entendre per a la xarxa neuronal al ser valors molt dispersos. Així doncs, per a normalitzar entre valors més manejables, dividirem cada valor entre 255. Fem servir float 16 per a estalviar espai en memòria, fer els càlculs en float 32 és innecessari, ja que la precisió que guanyem ens seria irrelevat.

```
training_data = np.load('training_data.npy', allow_pickle=True)
x_train = []
y_train = []

for x in range(len(training_data)):
    x_train.append(training_data[x][0])
    y_train.append(training_data[x][1][:3])

x_train = np.array(x_train, dtype=np.float16)
y_train = np.array(y_train, dtype=np.float16)

for a in range(len(x_train)):
    for b in range(len(x_train[a])):
        for c in range(len(x_train[a][b])):
            x_train[a][b][c] = x_train[a][b][c] / 255
```

Ara que ja tenim les dades preparades, ja podem definir la xarxa neuronal artificial; en aquest cas fem servir les TimeDistributed layers.

```
from keras.models import Sequential
from keras.layers import TimeDistributed, LSTM, Flatten, Dense, MaxPooling2D, Dropout, Activation
from keras.layers.convolutional import Convolution2D
from keras import optimizers
from keras.models import load_model
from keras import initializers

model = Sequential()

model.add(TimeDistributed(Convolution2D(32, (4,4), data_format='channels_last'), input_shape=(3, 150, 200, 3)))
model.add(TimeDistributed(Activation('relu')))
print(model.output_shape)

model.add(TimeDistributed(Convolution2D(32, (4,4), data_format='channels_last')))
model.add(TimeDistributed(Activation('relu')))
print(model.output_shape)

model.add(TimeDistributed(MaxPooling2D(pool_size=(5, 5), data_format='channels_last')))
model.add(TimeDistributed(Dropout(0.25)))
print(model.output_shape)

model.add(TimeDistributed(Convolution2D(16, (3,3), data_format='channels_last')))
model.add(TimeDistributed(Activation('relu')))
print(model.output_shape)

model.add(TimeDistributed(MaxPooling2D(pool_size=(5, 5), data_format='channels_last')))
model.add(TimeDistributed(Dropout(0.25)))
print(model.output_shape)

model.add(TimeDistributed(Flatten()))
print(model.output_shape)

model.add(LSTM(256, kernel_initializer=initializers.RandomNormal(stddev=0.001))) #128
model.add(Dropout(0.25))
print(model.output_shape)

model.add(Dense(100))
print(model.output_shape)

model.add(Dense(80))
print(model.output_shape)

model.add(Dense(40))
print(model.output_shape)

model.add(Dense(3, activation='sigmoid'))
print(model.output_shape)

model.summary()
```

Amb el model definit, seguidament haurem de compilar-lo. Farem servir l'optimitzador Adam i com a funció cost l'error quadràtic mitjà. Això ho establim de la següent manera.

```
opt = optimizers.rmsprop(lr=0.001)
model.compile(loss='mean_squared_error', optimizer=opt, metrics=['accuracy'])
```

Finalment ja només ens quedarà entrenar el model. Ho farem de la manera més simple possible i amb 10 epochs.

```
model.fit(x_train,y_train,
           batch_size=10,
           epochs=10,
           Verbose=1)
```

Al ja tenir el model entrenat, l'hem de guardar per a poder executar-lo. Això ho fem de la següent manera:

```
model.save("prova.h5")
```

4.3 Programa d'execució en temps real

Com hem fet anteriorment, ara definirem les dependències que cal tenir instal·lades per a fer servir aquest programa, aquestes són:

- Tensorflow 2.x
- Keras (farem servir la versió que incorpora tensorflow)
- numpy
- cv2
- time
- win32api
- grabber (el mateix que al primer programa, ja que haurem de capturar imatges en temps real)

Aquest programa realment és una barreja dels dos anteriors. La part d'obtenció de la seqüència serà exactament la mateixa al primer. L'únic que canviarem és que si bé al primer programa un cop obtinguda la seqüència el que es feia era afegir-la a l'array que contenia totes les anteriors, ara simplement la passarem al model mitjançant `model.predict()`. Això ens retornarà com a output una matriu de possibilitats de la qual agafarem la més alta. Tant la funció per a obtenir la captura com les funcions per detectar i prémer les tecles són igual que al primer programa, així que ho donarem per entès. Un cop més recordar que el codi sencer es pot trobar al repositori de GitHub.

La diferència es troba en la funció `__main__()`, que és la que controla el programa. De fet aquesta també és igual fins al comentari que diu «adaptació en temps real». Aquí veiem com converteix l'array a float 16 i en normalitza els valors, i tot seguit simplement afegeix una dimensió amb valor -1. Aquest -1 indica que el predict es farà de seqüència en seqüència, i no en forma de batch com s'ha entrenat. Finalment fem servir la funció `predict_classes` que ens retornarà el valor més gran dels possibles com a sortida a la xarxa neuronal, és a dir, la tecla que el model ens diu que és més probable que sigui correcta. Després simplement premem la tecla que toca i repetim el bucle. Quan s'acaba el bucle (perquè premem la tecla P), simplement tanca el programa desseleccionant totes les tecles que haguessin pogut quedar premudes.

```
keyboard = Controller()
def __main__():
    running = True
    time.sleep(5)
    print("Starting")
    while running:
        #capture seq
        grab_seq = []
        for i in range(3):
            last_time = time.time()
            grab_seq.append(get_grb())
            if i > 2:
                waittime = (1.0/capturerate)-(time.time()-last_time)
                if waittime>0.0:
                    time.sleep(waittime)

        #adaptació en temps real
        img_seq = np.array(grab_seq, dtype=np.float16)

        for a in range(len(img_seq)):
            for b in range(len(img_seq[a])):
                img_seq[a][b] = img_seq[a][b] / 255

        img_seq = img_seq.reshape(-1, 3, 150, 200, 3)
        prediction = model.predict_classes(img_seq)

        key_activator(prediction)

        keys = key_check()
        if "P" in keys:
            running = False
            keyboard.release("w")
            keyboard.release("a")
            keyboard.release("d")
            print("Stopped")
    __main__()
```

4.4 Proves

Ara veurem les proves que s'han fet al model tal com l'hem programat i les seves limitacions. Per augmentar el rendiment de la xarxa aplicarem un seguit de millores que veurem reflectides en les segones proves.

4.4.1 Proves primer model

Primer de tot parlarem del dataset sobre el que s'efectuen les proves. Són un parell d'hores conduint en diversos circuits del joc. La gràcia és que en el dataset d'entrenament no hi ha cap imatge del circuit que provarem per així veure si el model ha generalitzat correctament.

Per a comprovar l'eficàcia de la xarxa neuronal artificial el que farem serà simplement deixar-la conduit al simulador de conducció Assetto Corsa, el mateix del qual hem extret el dataset d'entrenament. Els vídeos demostratius es poden trobar al GitHub, ja que aquí és impossible inserir-los. Així doncs, abans de continuar es recomana veure'ls, ja que aquí simplement parlarem sobre els punts forts i fluixos de la IA conduit.

Podem veure com el rendiment de la xarxa és raonablement bo per ser el primer. No acostuma a sortir-se de la carretera tot i que va donant molts tomb. També podem observar que ha entès bé el concepte de carretera i s'aferra a no sortir d'ella, però li costa molt mantenir-se recte i agafar certa velocitat. Com que no agafa mai la velocitat a la qual ha estat entrenat es produeixen errades de proximitat, ja que aquest concepte l'havia après a una determinada velocitat que ara no s'està complint.

4.4.2 Millores al model

Com hem pogut veure, el nostre model és millorable. Això ens va perfecte per entendre la idea que en aquest món de la intel·ligència artificial res surt a la primera a causa de la quantitat d'opcions que tenim disponibles. Un model no té una solució única, en té de més bones i de més dolentes. Així doncs, ara amb la informació que hem pogut obtenir del primer model, el millorarem. El problema principal ha estat que itera moltíssim entre les tecles «wa» i «wd» cosa que a altes velocitats fa que es desequilibri. D'aquesta manera el que farem serà limitar les tecles que pot prémer, és a dir, farem que només pugui anar endavant (w), dreta (wd) i esquerra (wa), simplificant molt els moviments. A més a més baixarem el les imatges que es capturen per segon a 15, de manera que es premin 5 i no pas 10 tecles per segon, que hem vist que li era perjudicial. Finalment, baixarem la velocitat màxima del cotxe, ja que considerem que el que hem de fer realment és veure si la IA assoleix els coneixements bàsics de conducció, és a dir, si sap què ha de fer quan ve una corba o una recta.

Per a aconseguir això simplement haurem de canviar el paràmetre «capturerate» al codi i limitar les tecles que es capten i les que es premen. Per a reduir la velocitat, ho fem directament des del joc posant un límit. Tots els canvis es poden trobar al repositori de GitHub que ja ha estat esmentat amb anterioritat.

4.4.3 Proves segon model

Un cop aplicats els canvis esmentats en el punt anterior, el model ha millorat el seu rendiment considerablement com podem veure en els vídeos disponibles en el GitHub.. Veiem que la majoria d'imatges han estat enregistrades en circuits amb moltes corbes en les quals realment la IA pot mostrar el seu potencial. Veiem que els objectius mencionats anteriorment es compleixen perfectament i que el model aconsegueix no sortir de la carretera i prendre les corbes amb una velocitat considerable.

Dit això, obviament aquest model no és perfecte. De fet, hi ha infinites possibilitats que el podrien fer millor en tots els casos. El problema principal és la llargada del dataset, el qual podria ser molt més gran, però això significaria moltes hores enregistrades que per una sola persona és molt difícil, però també uns recursos extres que estan fora de l'abast d'un treball de recerca, degut al fet que el model actual ja frega el límit de ram i vram disponible en el sistema, que com s'ha dit anteriorment són de 16 i 6 gb respectivament.

Una altra via de millora seria tenir accés als fitxers del joc per a saber quan el cotxe surt de la carretera i poder-lo entrenar amb això. D'aquesta manera, eliminaríem el problema que la IA no sap tornar a entrar a la pista quan surt, ja que mai ha vist ningú sortir (no pot haver-hi cap seqüència sortint de la pista en el dataset, ja que la IA entendria que és bo sortir). El problema és que bàsicament el joc no és de codi obert i per tant no es poden accedir als seus fitxers interns.

Òbviament, no ens podríem pensar que algú en un ordinador personal a casa programaria una IA que conduís millor que una persona, quan ni tan sols les grans marques del sector de l'automoció ho aconsegueixen. Tot i això hem vist com s'assolien els objectius inicials i per tant la pràctica ha estat tot un èxit.

5 Últims avenços en IA arreu del món

Fins aquí hem vist què pot fer un estudiant de segon de batxillerat amb el seu ordinador personal en el seu temps lliure. Dit això, què poden fer investigadors amb grans superordinadors amb milers de gràfiques al seu abast? Efectivament, coses impressionants. Ara en veurem un petit recull per a poder fer-nos una idea del que és la IA en el món real i com està afectant la societat.

Estan classificats en les 3 categories que hem anat tractant:

- MLP: Sistemes de predicción: Algorisme de YouTube
- CNN: Processament d’imatges: detecció i reconeixement: on comentarem l’Autopilot i els sistemes de control i detecció de masses.
- RNN: Models d’NLP (processament de text natural): on tractarem el GPT-3 i el BlenderBot, aquest darrer el podrem provar.

5.1 Sistemes de predicción: Algorisme de YouTube

L'algorisme de YouTube avui dia no deixa de ser una gran xarxa neuronal que a partir de diferents dades et recomana el que sigui més probable que vulguis mirar. Cal recalcar el tema de les dades, ja que ells tenen absolutament totes les dades necessàries per a entrenar una xarxa neuronal. Saben quins vídeos mires, quins t'agraden molt (els dones al like), quins veus completament, quins t'avorteixen (si els deixes per la meitat) o bé quins no t'agraden (els que els dónes dislike). Amb això ja saben quina temàtica de vídeos t'agraden i per tant que t'han de recomanar. Cal destacar que aquesta és una xarxa viva, és a dir, si algun dia per la raó que sigui comences a veure algun altre tipus de vídeos, la xarxa aprendrà que has tingut un canvi de gustos i et començarà a mostrar aquells vídeos que realment t'agraden.

El funcionament d'aquesta xarxa és molt bo i realment fa que YouTube tingui aquest component addictiu de veure un vídeo rere l'altre.

Hem posat YouTube com a exemple perquè és molt clar, però dins d'aquest sac també podríem posar el sistema de mostratge d'anuncis publicitaris o l'algorisme de quasi bé totes les xarxes socials com poden ser Instagram, TikTok entre d'altres.

5.2 Processament d’imatges: detecció i reconeixement

L’aplicació més directa de les xarxes neuronals convolucionals és el tractament d’imatges i la detecció d’objectes. Ara veurem un parell d’exemples que podem trobar en el nostre dia a dia.

5.2.1 Autopilot

En els últims anys hem vist com una empresa revolucionava el mercat automobilístic. Tesla ha entrat al mercat amb cotxes elèctrics que posseeixen tecnologia puntera a un preu no excessivament car, sobretot als Estats Units, la qual cosa ha fet que guanyi molta popularitat en molt poc temps. D’aquesta empresa ens fixarem en el seu «autopilot»⁸, un algorisme punter que permet conduir de manera autònoma. Doncs bé, aquest sistema pot funcionar gràcies a xarxes neuronals que detecten i reconeixen els elements que es van trobant per la carretera podent fer així el sistema viable. Cal dir que totes les altres grans empreses de l’automoció també estan treballant en sistemes semblants, però Tesla ha estat la capdavantera.

5.2.2 Sistemes de control i detecció de masses

Així com la intel·ligència artificial ens pot beneficiar molt com a societat, si se'n fa un mal ús pot suposar un retall de les nostres llibertats personals. Així com hem dit que amb xarxes neuronals podem detectar objectes, també podem detectar cares. D’aquesta manera, molts règims autoritaris estan utilitzant càmeres equipades amb aquesta tecnologia per tal de controlar les masses poblacionals. Això suposa una clara retallada a les nostres llibertats individuals, ja que el govern d'un determinat país o fins i tot alguna empresa privada pot saber exactament per on passem i què fem. Fins ara, si a un govern no li interessava alguna manifestació, no podia reprimir a tota la multitud més enllà d'uns quants detinguts. Ara bé, si tens càmeres que identifiquin a la població amb aquest tipus de tecnologia, pots reprimir a tothom per igual. Això ja està passant a la Xina on per exemple els manifestants pro-democràcia de Hong Kong han d'anar tapats fins dalt i equipats amb làsers per a inhibir les càmeres que es troben en la via pública.

Tot això a Europa ens ho veiem molt lluny, però en realitat i salvant les distàncies estem més a prop del que sembla. Aquest any 2020 a Barcelona ciutat s’han començat a posar càmeres amb aquest tipus de tecnologia en espais públics com pot ser el transport públic, els camps de futbol, fires, concerts o inclús en manifestacions. De moment els precursors del projecte asseguren que és per tal de vetllar per la salut pública i combatre activitats delictives com furts o actes potencialment terroristes. D’entrada sembla una bona iniciativa si es limita a fer-ne l’ús que se’n diu, però un cop es tenen les eines és molt fàcil que aquesta bona voluntat es corrompi.

8 <https://www.tesla.com/autopilot>

5.3 Models d’NLP: GPT-3 i Blenderbot

Aquest últim any estem vivint una explosió en el món del tractament de text, l'anomenat NLP (Natural Language Processing). Parlarem de dos en concret. Primer del GPT-3 d'Open AI (empresa sense ànim de lucre del CEO de Tesla Elon Musk) perquè bàsicament és el més avançat del món, ja que aconsegueix uns resultats extremadament bons. Això el fa molt difícilment diferenciable d'un humà escrivint. D'altra banda parlarem de Blenderbot, un bot de Facebook que si bé no és tan bo i espectacular com ho és el GPT-3, aconsegueix resultats molt bons i el podem executar en la nostra màquina local o al Google Colab.

5.3.1 GPT-3

Primer de tot farem un petit aclariment. Des dels inicis de les xarxes neuronals artificials, tots els grups de recerca punters en la matèria publicaven com a codi obert totes les seves noves investigacions de manera que tothom sabia què feia cadascú i per tant se'n beneficiava tota la comunitat. Amb el model GPT-3 això ja no ha estat així per una simple raó: és una eina massa poderosa que no pot acabar en mans de qualsevol. Ara bé, tot i que Open AI es va fundar com una empresa sense ànim de lucre, ha decidit llicenciar el codi font de GPT-3 a Microsoft, aquest fet ha indignat a bona part de la comunitat. Això significa que avui en dia els únics que poden fer servir aquesta eina són els enginyers d'una gran empresa com és Microsoft, que està creant un monopoli no desitjat.

No s'ha publicat gairebé res de com funciona GPT-3 o amb què ha estat entrenat més enllà del que ha dit Open AI, és a dir, que és un model amb 175 bilions (americans) de paràmetres i que ha estat entrenat amb una quantitat ingent de dades (prop de 600gb de text equivalent a un trilió americà de paraules). Així doncs, com sabem que és una eina tan potent? Doncs bé, per les reviews i les aplicacions que li han fet de presentació o durant la petita beta tancada. Entre elles podem destacar:

- «Project december» de Jason Rohrer⁹:

Project December és una IA impulsada pel model GPT-3 d'ambient retro creada per Jason Rohrer. Mitjançant a un pagament es pot accedir a ella de manera online i veure així el seu rendiment.

- Un article escrit per GPT-3 al diari britànic «The Guardian»¹⁰:

El famós diari anglès va tenir accés al GPT-3 i a més de fer-ne una review, van deixar que el model escrivis la totalitat d'un l'article amb la següent premissa: explicar-nos perquè els robots són pacífics, el resultat és impressionant com a mínim.

9 <https://projectdecember.net/>

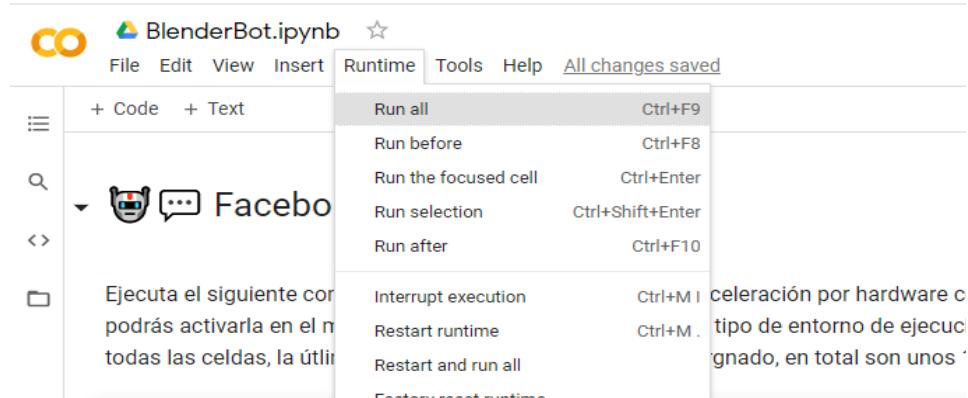
10 <https://www.theguardian.com/commentisfree/2020/sep/08/robot-wrote-this-article-gpt-3>

- Aplicació AI dungeon:

AI dungeon és una aplicació disponible tant a la Google Play Store d'android com a la App Store de IOS o bé a través d'internet a la seva pàgina web¹¹. Aquesta aplicació tracta d'un joc de rol creat des de zero on cada cop que s'inicia una partida nova es va prenent una direcció o una altra depenent del que vagis contestant tu dins de la història. En ella tot el text, és a dir, la història, és creada pel model GPT-2, l'antecessor a GPT-3, tot i que des del 22 de juny del 2020 ja està disponible la «dragon model upgrade», la qual ja fa servir el GPT-3 millorant així els seus resultats que ja de per si eren molt bons.

5.3.2 BlenderBot

Blenderbot és una xarxa de tractament de text molt més senzilla, però que al ser de codi obert podem executar en el nostre ordinador i això ens permet parlar directament amb ell sense cap complicació. Òbviament no és ni semblant al GPT-3, però també ofereix un bon resultat. Està disponible en versions de 2,7 i de 9,4 bilions americans de paràmetres. Nosaltres utilitzarem el model petit perquè amb aquestes dimensions la memòria del sistema és un problema real. Així doncs, perquè el lector ho pugui fer sense tenir coneixements previs a casa, farem servir un notebook¹² del Google Colab, en el que només hem de donar-li a executar tot, esperar uns 10 o 15 minuts perquè s'instal·li en els servidors de Google i començar ja a xatejar amb ell (òbviament en anglès). D'aquesta manera, un cop hem entrat a l'adreça electrònica proposada anteriorment només li haurem de donar a runtime → run all:



11 <https://play.aidungeon.io/main/home>

12 <https://cutt.ly/LhbFapN>

Potencial de les xarxes neuronals artificials – Biel Altés Grifoll

Un cop fets aquests senzills passos i després d'esperar al fet que es carregui el model ja podrem tenir conversacions amb ell. Cada cop que el reiniciem, pren un rol nou (treball i hobbies) cosa que li atorga molta flexibilitat i per tant les converses amb ell no es fan repetitives. Si volem acabar una conversa i començar-ne una altra, no fa falta reiniciar tot el script, simplement escrivint «[DONE]» com a missatge es reiniciarà la conversa.

```
Loading /content/ParlAI/ParlAI/data/blended_skill_talk/train.json.  
Saving to /content/ParlAI/ParlAI/data/blended_skill_talk/train.txt  
Loading /content/ParlAI/ParlAI/data/blended_skill_talk/valid.json.  
Saving to /content/ParlAI/ParlAI/data/blended_skill_talk/valid.txt  
Loading /content/ParlAI/ParlAI/data/blended_skill_talk/test.json.  
Saving to /content/ParlAI/ParlAI/data/blended_skill_talk/test.txt  
[context]: your persona: i got married last year.  
your persona: i live on a boat.  
Enter Your Message:
```

Aclarir que aquesta IA no ha estat programat per mi. L'hem executat com a pràctica per a conscienciar-nos sobre el potencial actual de les xarxes neuronals. Tota la documentació i codi es pot trobar al seu github¹³.

13 <https://github.com/facebookresearch/ParlAI>

6 Conclusions

La recerca feta durant el període de desenvolupament del treball ha permès assolir el repte més important de tots: comprendre el funcionament de les xarxes neuronals artificials. A més a més, s'ha pogut explicar d'una manera resumida i el més entenedora possible. Així doncs hem pogut posar cara a una tecnologia que sense saber-ho utilitzem quotidianament.

Com a objectius secundaris hem assolit el ple funcionament en totes les pràctiques que s'havien proposat, des dels programes amb python per a aplicar directament els fonaments matemàtics fins a l'ús de TensorFlow per a programar diversos models funcionals.

Hem acabat veient les últimes notícies en intel·ligència artificial arreu del món, cosa que ens ha permès de finalitzar el treball entenent projectes reals i el seu rendiment, que no deixa de ser espectacular. A partir d'aquí doncs, hem pogut veure perquè estan tan esteses aquestes xarxes i perquè tot apunta a que en un futur molt proper ho estarán encara més.

Durant tots aquests apartats hem pogut anar veient, de menys a més, el potencial d'aquestes xarxes neuronals a partir d'aplicacions reals que ens han permès reflexionar sobre el paper de la IA en el nostre dia a dia i com realment té i tindrà un gran impacte a la nostra vida.

Arribats a aquest punt, voldria destacar la pràctica de conducció i el perceptró programat des de zero, que personalment m'ha suposat un repte majúscul degut a la seva complexitat. Veient-ne el resultat, estic molt satisfet amb el que he obtingut d'aquestes pràctiques i tot el que he après durant aquests mesos. Tot i que ja tenia coneixements en programació, vaig començar des de zero en el món de la IA havent d'aprendre tots els coneixements necessaris per al desenvolupament del treball.

Les hores dedicades a l'estudi, recerca i aplicació d'aquestes xarxes han fet que aquest treball hagi estat realment un repte a nivell personal i m'ha proporcionant un aprenentatge en la matèria de la intel·ligència artificial molt valuós.

7 Bibliografia

7.1 Llibres

- FAUSETT, Laurene, *Fundamentals of neural networks*, New Jersey, Prentice Hall, 1994.
- TORRES, Jordi, *Python Deep Learning: Introducción práctica con Keras y TensorFlow 2*, Barcelona, Marcombo, 2020.
- OSORIO, Pablo; CASTRO, Sergio, *Unas matemáticas para todos*, Granada, Alfa-Omega, 2019.

7.2 Documents del web

- NIELSEN, Michael, *Neural Networks and Deep Learning* [en línia],
«<http://neuralnetworksanddeeplearning.com>».
- MORENO PARRA, Rafael Alberto, Redes Neuronales [en línia],
«<https://openlibra.com/es/book/redes-neuronales-parte-1>»

7.3 Pàgines del web

- <https://www.tensorflow.org/>
- <https://keras.io/>
- <https://en.wikipedia.org/wiki/Perceptron>
- https://en.wikipedia.org/wiki/Sigmoid_function
- https://en.wikipedia.org/wiki/Universal_approximation_theorem
- <https://www.youtube.com/channel/UCYOv9HwOFwK0lY2dUQIZSpq>
- <https://www.geogebra.org/classic>
- <https://en.wikipedia.org/wiki/GPT-3>
- <https://openai.com/>
- <https://github.com/openai/gpt-3>
- <https://openai.com/blog/openai-licenses-gpt-3-technology-to-microsoft/>
- <https://ai.facebook.com/>
- https://www.elconfidencial.com/tecnologia/2019-08-09/protestas-hong-kong-hackear-inteligencia-artificial_2169083/
- <https://www.lavanguardia.com/vida/20200906/483329209528/camaras-videovigilancia-interior.html>
- <https://wallpaperaccess.com/ai> (imatge de la portada, modificada)