Treball d’innovació

AlphaGo Zero

Intel·ligència Artificial

Q1 2022-23

Pere Carrillo

Marc Ordóñez

Laura Pérez

Índex

[**1. Introducció**](#_v97atk4mdqxa) **3**

[**2. Monte-Carlo Tree Search**](#_n3sw4cqygwey) **4**

[2.1. Minimax](#_2tezto8jg8fe) 4

[2.2. Estructura de l’arbre](#_qqe07tvm0eq9) 5

[**3. Xarxes neuronals**](#_tsy3tzn381sn) **7**

[3.1. Què és una xarxa neuronal?](#_k21xv13c40a2) 7

[3.2. Entrenament](#_oqzhlx1vgq83) 8

[3.3. Overfitting](#_drrgahabcg77) 9

[3.4. Tipus de capes](#_r6adbw96cqrj) 9

[3.4.1. Fully-connected](#_ibam4efq097h) 9

[3.4.2. Convolucional](#_tk4jgs8gjbls) 10

[3.4.3. Batch Normalization](#_onwvi89qzpdv) 11

[3.4.4. ReLU](#_z8nz4y8lhgx6) 11

[3.4.5. Softmax](#_fp8y54ahn63b) 11

[3.4.6. Residual Block](#_r2d86kgjmk45) 11

[**4. Funcionament**](#_f3tv5x7j1epf) **12**

[4.1. Entrada](#_xqe0qd4ulqw1) 13

[4.2. Policy Net](#_3ca0onr80t4x) 13

[4.3. Value net](#_6lu5d0pfiazh) 14

[4.4. Entrenament](#_jk0md1j79ue9) 14

[4.5. Reinforced Learning](#_o2l13m285mqp) 14

[**5. Futures versions**](#_eo3miobo97qw) **15**

[**6. Per què és innovador?**](#_lz8n33c0sigc) **17**

[**7. Beneficis per a DeepMind i Google**](#_4otsuyjh8tqz) **18**

[**8. Impacte en la societat**](#_umgtvgj5hxvv) **18**

[**9. Bibliografia**](#_ozqu24824u) **20**

# Introducció

AlphaGo Zero és una versió millorada del famós AlphaGo. Va ser publicat en el paper *“Mastering the game of Go without human knowledge”*, publicat a octubre de 2017. Es tracta d’un programa capaç de jugar al joc del Go en taulells de 19x19 (mida típica) i que és molt superior a l’antiga versió del programa coneguda com a AlphaGo. Això s’ha aconseguit amb l’ús de *reinforcement learning* i una xarxa neuronal.

L’essència de AlphaGo Zero és la mateixa que la seva antiga versió: utilitza una estructura anomenada Monte-Carlo Tree Search per a guardar-se els possibles estats de la partida als que pot accedir des de l’estat actual del taulell. Al primer nivell de profunditat estan els estats als que pugui arribar en un únic moviment, al següent nivell estaran els estats als que es puguin arribar en dos moviments (un moviment de fitxes blanques i un moviment de les negres) i així successivament. *Monte-Carlo Tree Search* (MCTS) és un tipus d’estructura en forma d’arbre que funciona molt bé en jocs petits com pugui ser el 3 en ratlla perquè pots estendre tots els nodes de l’arbre amb tots els possibles estats i veure per a cadascuna de les fulles (estats finals del joc) en quins casos guanyes i en quins perds per fer el moviment que saps amb certesa que et farà guanyar (o com a mínim que t’assegura que no perdràs). Es tracta de la mateixa lògica que hi ha darrere del MiniMax que ens van explicar a teoria.

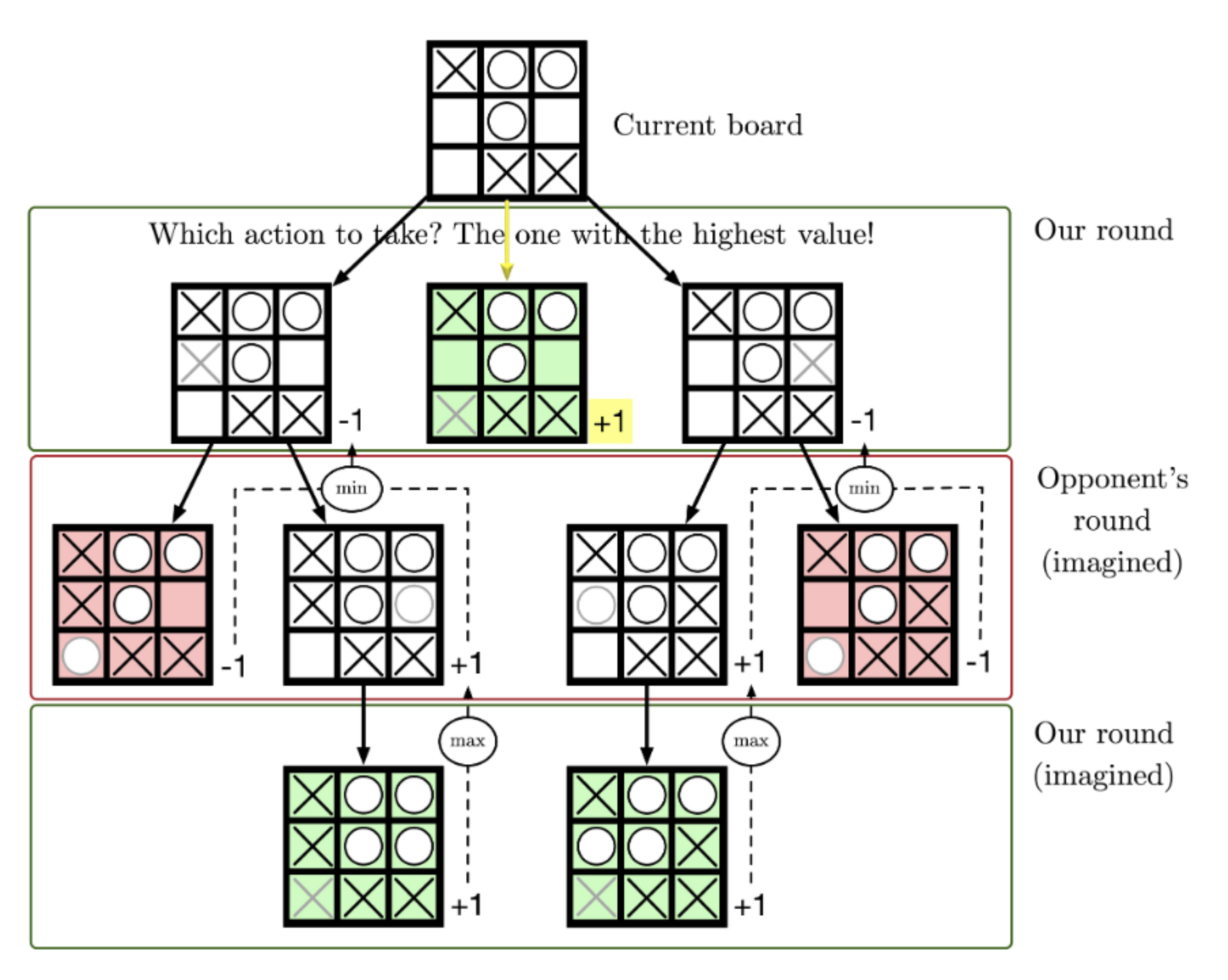
En jocs amb més possibles moviments i taulells més grans el mètode d’explorar tots els possibles estats del joc deixa de ser viable, ja que la ramificació és tan gran que no tenim capacitat per computar tota la informació. En els papers publicats per DeepMind, autors del AlphaGo i AlphaGo Zero, utilitzen una innovadora tècnica que fa ús de xarxes neuronals. La manera en què s’utilitzen aquestes xarxes i l’estructura de les mateixes és la principal diferència entre la primera versió anomenada AlphaGo i la que estem tractant nosaltres que és AlphaGo Zero.

En el cas d’AlphaGo Zero tenim una única xarxa neuronal anomenada policy-value net que s’encarrega de decidir quins són els moviments que creu que són els més probables que prengui l’adversari o nosaltres mateixos i d’aquesta forma aconseguim una primera poda que ens estalvia haver de mirar moviments que sabem amb casi total seguretat que no farem. Aquesta primera funcionalitat de la xarxa neuronal és la coneguda com a policy net. D’entre la resta de moviments que són més probables, cal veure quin és el que té una major probabilitat de fer que guanyem. Per fer-ho, a cada node del *Monte-Carlo Tree Search* (MCTS) se li assignarà una probabilitat d’arribar a un victòria si apliquem aquell moviment. La xarxa neuronal serà l’encarregada d’assignar les probabilitats als nodes.

# Monte-Carlo Tree Search

## Minimax

En essència, el Monte-Carlo Tree Search funciona igual que Minimax. És un mètode utilitzat per als jocs d'estratègia en què es té total coneixement de totes les variables del joc. Usualment es fa servir en jocs de taula en els que tenim caselles i fitxes, com per exemple tres en ratlla. Donat un taulell i un estat de les fitxes ens retorna el millor moviment que podem escollir per tal d’acabar guanyant la partida. Tot i que a priori pugui semblar que aquesta informació no es pot obtenir sense tenir en compte els futurs moviments de l’adversari, veurem que en jocs o taulells d’una mida no massa gran (o en els que no es puguin fer massa moviments) es pot saber quin moviment ens portarà a la victòria (o com a mínim no ens farà perdre) posant-nos sempre en el pitjor dels casos per a cadascun dels possibles moviments des de l’estat actual.



En l’exemple de la imatge, si fem la jugada de l’esquerra o la de la dreta, hi ha un possible resultat en el que perdrem i un altre en el que guanyem, però com ens posem en el cas pitjor aquests dos moviments els posem el valor de -1, ja que en algun cas ens poden portar a perdre. En canvi, realitzant el moviment central ens assegurem que en tots els possibles futurs estats (que en aquest cas ja no n’hi hauria més perquè s’acaba la partida) guanyariem si fem aquell moviment. Per realitzar aquest càlcul ens hem d’anar fins al nivell més profund, que serà aquell que un dels dos jugadors guanyi o en què empatin, i a partir d’allà anar assignant valor a les fulles: 1 si guanya l’usuari, -1 si guanya l’adversari i 0 en cas d’empat. Els valors de les fulles els anirem expandint cap als pares de forma que el pare es quedi amb el pitjor cas del seus fills en cas que sigui una jugada de les fitxes de l’adversari (mínim de tots els fills). Com hem de suposar que l’adversari sempre jugarà la seva millor opció, si hi ha algun cas en el qual ens guanyi (resultat -1 perquè nosaltres perdem) suposarem que escull aquella opció, de manera que hem de calcular el mínim dels fills. En cas de que el moviment sigui d’una fitxa nostre calcularem el màxim dels fills, ja que de totes les opcions nosaltres sempre escollirem la que ens beneficiï més. El pare d’un node és el moviment previ que s’ha de realitzar al taulell per arribar a la posició del fill.

Un ús clàssic de minimax seria tres en ratlla, on es poden estendre tots els possibles moviments del jugador i de l’adversari fins a arribar a un moviment que ens asseguri que no perdrem.

Aquesta mateixa lògica es podria aplicar a altres jocs de taulell que també son deterministes i on es juga amb posicions i fitxes fins a arribar a un resultat en què un dels dos jugadors guanya. Un exemple serien els escacs però en aquest cas no seria viable, o com a mínim no tindria un cost computacional i espaial baix, resoldre el joc amb minimax degut a la amplia ramificació que es genera en cada moviment. Aquest factor de ramificació depèn de la mida del taulell, del nombre de fitxes que podem moure i de tots els moviments que podem fer amb cadascuna. Com es pot observar, trobar el moviment que ens farà guanyar deixa de ser trivial.

Si ens posem en el supòsit d’un joc encara més complex com el Go, on el taulell és de 19x19, tenim més llibertat de moviment i més fitxes la ramificació es fa massa àmplia.

## Estructura de l’arbre

AlphaGo utilitza la mateixa estructura en forma d’arbre que hem explicat anteriorment. Rep el nom de *Monte-Carlo Tree Search*. Cada node representa un estat vàlid de la partida (distribució de fitxes en el taulell en aquest instant) i cada fill representa un moviment vàlid. En l’aresta que uneix un pare amb algun dels seus fills s’emmagatzema la següent informació:

* Un comptador **N** per representar el nombre de cops que s’ha realitzat l’acció *a* (passar de l’estat pare a l’estat del fill realitzant el moviment representat per l’aresta).
* Un valor **W** amb el valor total del següent estat (el fill).
* Un valor **Q** amb el valor mig del següent estat (el fill).
* La probabilitat **P** d’escollir l’acció *a.*

L’arbre es va construint en tres passes que s’aniran repetint successivament:

* Escollir un dels fills per veure què passaria a la partida si escollissim aquella opció. Aquesta elecció es fa en funció del valor que tingui el node fill juntament amb una certa incertesa que ens permet explorar fills que no tenen perquè ser els de valor més alt. Aquest procés es va repetint de manera que anem escollint fills dels fills fins a arribar a un estat terminal, que serà aquell en què un dels dos jugadors hagi guanyat.
* Un cop arribem a una de les fulles de l’arbre es generarà un valor en funció del resultat final del taulell (hem guanyat o hem perdut). Aquest valor s’anirà propagant des de la fulla fins al node arrel que ha estat el primer en el que hem començat a escollir futures accions. Per a cadascun dels nodes que van de la fulla fins al node arrel (seqüència de moviments que hauríem d’haver realitzat per acabar en aquell estat), s’incrementarà en 1 la *n* (el comptador del nombre de simulacions que han pres l’acció per arribar a l’estat d’aquell node) i la *W* s’haurà d’incrementar amb el valor de la nova fulla. La *Q* també es recalcularà dividint la nova *W* per la nova *n*.
* L’últim pas és l’expansió. Per a cada fulla a la que arribem (camí que ja no s’ha explorat en més profunditat) generarem un nou fill corresponent a un possible moviment des d’aquell estat. D’aquesta manera anirem ampliant l’arbre a cada iteració. Com més cops repetim aquest procés, més informació tindrem de com de bo o dolent que pot ser realitzar un cert moviment, ja que haurem explorat alguns dels possibles moviments que podríem prendre després de realitzar aquella acció i sabríem si generalment aquella acció ens porta a guanyar o a perdre.

Sempre s’intentarà explorar aquells nodes que tinguin un valor més alt perquè se suposa que són els que ens portaran a millors resultats. Per contra, pot haver-hi altres camins que siguin igualment bons però que encara no s’hagin explorat. És per això que es permet una certa aleatorietat a l’hora d’escollir el node a partir del qual expandim la simulació. La política de l’arbre explorarà moviments vàlids i es centrarà en aquells que semblen millors basats en les simulacions prèvies.

# Xarxes neuronals

L’important de l’estructura de l’arbre són les variables *policy* i *value:*

* ***Policy***: És una probabilitat que indica quan bo és un moviment. És a dir, quan probable seria realitzar aquesta jugada si volguéssim guanyar.
* ***Value***: És un valor entre -1 i 1 que representa quin jugador és més probable que guanyi si es realitza aquest moviment.

En el paper original d’AlphaGo aquestes dues variables s’obtenien fent ús de xarxes neuronals independents: la *policy net* i la *value net*. Totes dues rebien com a paràmetres l’estat del taulell en aquell node i en els darrers moviments de la simulació i amb aquesta informació les dues xarxes retornaven el valor de *policy* i de *value*.

## Què és una xarxa neuronal?

Es tracta d’un mètode empleat per la intel·ligència artificial, més concretament pel *deep learning* per replicar una estructura similar a la del nostre cervell: un conjunt de neurones distribuïdes per capes, on les neurones d’una capa s’interconnecten amb les de la següent per transmetre’s informació que aniran processant fins a arribar a l’última capa. S’intenta reproduir el produeix de sinapsis que es produeix a les neurones humanes, on les neurones reben certs impulsos nerviosos amb informació i s’encarreguen de transmetre aquesta informació a altres neurones. La primera capa és coneguda com a *input layer* i la última és la *output layer*. Totes les capes intermitges s’engloben en les *hidden layers*. La *input layer* és la que rebrà la informació a processar o a partir de la qual volem obtenir una certa informació, probabilitat o valor. La *output layer* és la darrera capa i ens retornarà la informació que s’ha extret gràcies a totes les capes de neurones.

Una neurona consta d’un pes, d'una funció d’activació (no sempre és necessària) i d’un *bias* (no sempre és necessari). La idea bàsica del funcionament de les neurones és el següent:

* La neurona rep un o diversos valors com a inputs i uns pesos que s’acostumen a multiplicar per cadascun dels inputs que rep la neurona. D’aquesta manera es pot decidir quines neurones tenen més pes o importància en els càlculs que es realitzaran.
* Realitza una operació amb aquests valors (normalment els suma però n’hi ha d’altres tipus).
* Se li afegeix un valor constant anomenat *bias*.
* Per acabar, al valor final de la neurona se li aplica una funció que pot ser una *sigmoid, softmax* o una *ReLU*, entre d’altres per tal de modificar el valor (convertir-lo en un decimal entre 0 i 1, fer que si és negatiu passi a valdre 0, etc.

Les neurones d’una mateixa capa no estan connectades amb altres neurones de la mateixa capa. Normalment, les neurones d’una capa es connecten a les neurones de la capa següent, tot i que no sempre és així. Per exemple, les capes residuals reben com a inputs valors de neurones que no corresponen a la capa anterior.

## Entrenament

La manera en què una xarxa s’entrena per a realitzar la funció que se li ha assignat correctament és complexa i hi ha involucrats diversos conceptes matemàtics com el gradient de les funcions. Per a fer-nos una idea, podem calcular una funció anomenada *Loss function* (funció de pèrdua) que representi lo lluny que està l’output de la xarxa del valor esperat. La nostra tasca serà disminuir al màxim aquesta funció per tal que no hi hagi diferència entre el que hauria de retornar la xarxa i el que retorna.

Aquí és on entren en joc les matemàtiques. Com tenim el valor esperat i tenim el valor de les neurones de la capa justament anterior i també coneixem els pesos de les neurones. És a dir, la importància que se li ha donat a cada connexió entre la penúltima capa i la *output layer*. Si variem els valors dels pesos obtindrem valors diferents a l’output, ja que se li donarà més importància a unes altres neurones que afectaran al resultat final del càlcul que es fa en aquella neurona. Si variem el *bias* de cada neurona també farem que variï el resultat.

La finalitat de l’entrenament serà trobar de quina manera es poden variar tots aquests paràmetres per fer que la funció de pèrdua disminueixi el més ràpid possible. Per exemple, pot haver un valor que per molt que l’augmenten o el disminuim casi no afecta al output final, mentres que igual n’hi ha un altre que amb una mínima variació ens acosta molt al resultat desitjat.

Per a trobar la manera en què s’haurien de canviar els pesos de la xarxa s’utilitza el descens del gradient de la funció de pèrdua, que és un mètode matemàtic que determina en quina “direcció” decrementa més ràpidament la *Loss function* depenent dels valors que es modifiquen de cada neurona.

## Overfitting

Un dels problemes que poden surgir durant l’entrenament de la xarxa neuronal és que convertim a la xarxa en una experta en resoldre els problemes que li hem donat per entrenar però que en canvi per a problemes que no formessin part del paquet d’entrenament no sigui capaç de resoldre’ls.

Posem com a exemple una xarxa que s’encarrega de dir si en una imatge hi ha un gos o no. En aquest cas podria tractar-se d’una xarxa que retorni un valor de 0 a 1 indicant la probabilitat de que en la imatge aparegui un gos. Per a probabilitat superiors a 0,5 suposarem que hi ha un gos i en cas contrari suposarem que no.

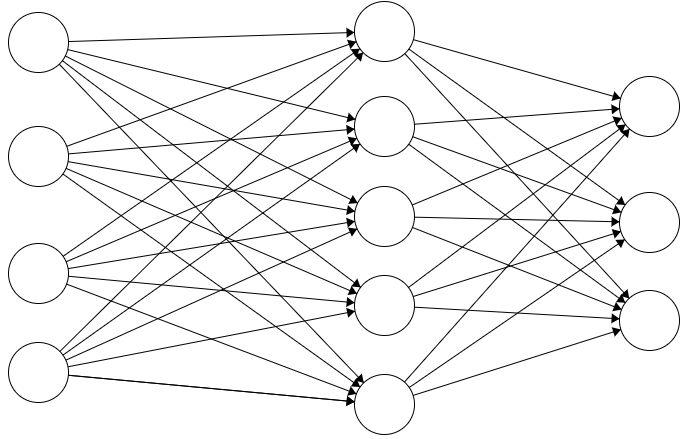
Si fem servir un grup d’imatges d’entrenament molt escàs o amb poca varietat (per exemple sempre surten el gossos amb un fons blanc, o només entrenem amb imatges d’una única raça de gos podem aconseguir que la xarxa tingui un comportament no desitjat, de manera que si li passem una imatge sense fons blanc o d’una altra raça no sigui capaç d’identificar el gos. Per evitar-ho, periòdicament es va comprovant amb imatges externes a les de l’entrenament que la xarxa també retorni el resultat esperat en aquelles imatges. Altrament, podríem estar fent que la xarxa associés el fet de que hi hagi o no un gos a patrons molt concrets que casualment tinguin les imatges d’entrenament, com pugui ser el color, el fons, la posició del gos, la perspectiva, el contrast…

## Tipus de capes

En l’actualitat s’utilitzen diversos tipus de capes en funció del propòsit que tingui la xarxa. S’acostuma a intercalar capes de diversos tipus. Ens centrarem en les que es fan servir en aquest projecte, que són la capa convolucional, la de *batch normalization* i la *ReLU*. Depenent de la llibreria o programa que es faci servir la *ReLU* es pot considerar que és una capa independent tot i que és la funció d’activació mencionada anteriorment.

### *Fully-connected*

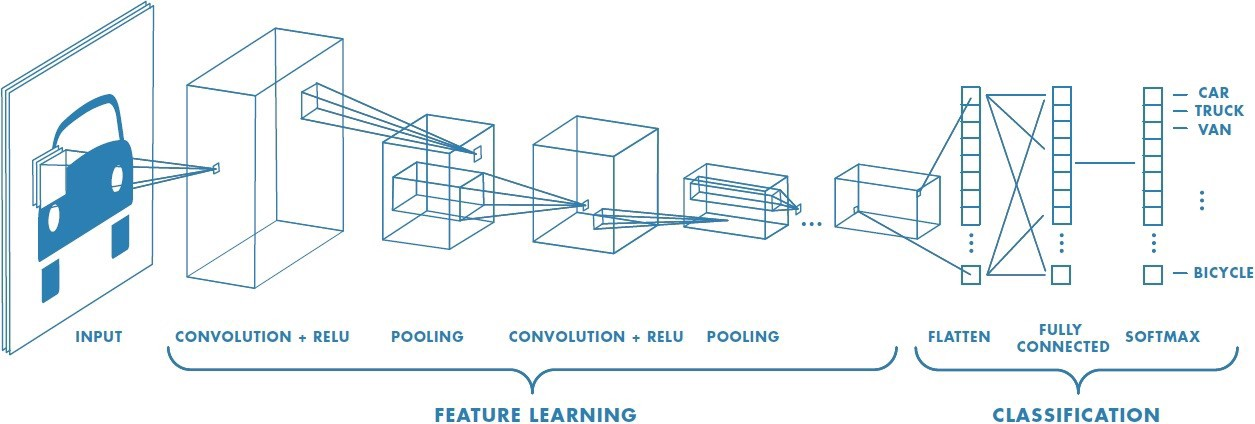
Es tracta d’una capa molt senzilla però que forma part de la majoria de xarxes neuronals. Consisteix en connectar cadascuna de les neurones d’una capa amb totes les neurones de la següent. En el següent cas tindríem dues capes completament connectades.



### Convolucional

És una de les capes més utilitzades en l’actualitat. S’utilitza per a diverses funcions entre les que destaquen: reconeixement d’imatge, reconeixement de text, reconeixement d’àudio, conducció autònoma i autenticació biomètrica.

La seva principal característica és que no es centra només en un valor concret, si no que també té en compte les valors que l’envolten. És per aquest motiu que és molt útil en tasques en què s’ha d’interpretar la informació com a conjunt i no individualment. Per exemple, en el reconeixement facial no ens serveix de res analitzar un píxel de color carn, però si estem tenint en compte la informació que envolta aquest píxel ens podrem adonar si es tracta d’un nas, del front, de la barbeta… S’acostuma a fer servir per analitzar imatges, ja sigui per transformar lletra manuscrita o per veure quin objecte hi ha en una imatge, o per identificar certs patrons, entre d’altres.



Les capes de la xarxa neuronals poden rebre com a entrada un tensor amb més d’una dimensió. Per exemple, en el cas d’una imatge l’entrada seria una matriu. O si fos una imatge RGB l’entrada podrien ser 3 matrius per cadascun dels colors. A cada matriu d’entrada se la coneix com a canal. Si fos una imatge quadrada de 512 píxels de costat en RGB l’entrada seria un tensor de 3 canals amb alçada 512 i amplada 512.

### *Batch Normalization*

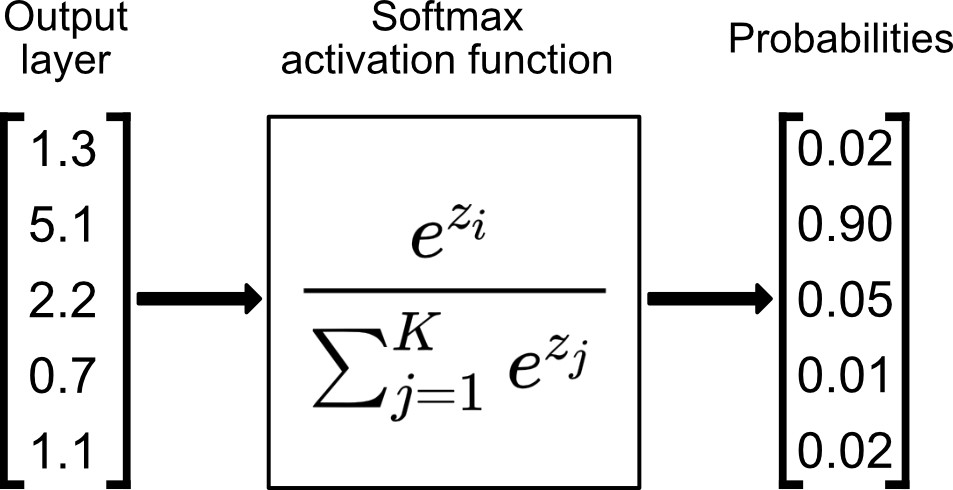
És un procés que s’aplica entre l’output d’una neurona i l’input que reben les neurones de la següent capa. Consisteix en normalitzar els valors que surten d’una capa per evitar l’overfitting. Normalment les entrades de la xarxa ja venen normalitzades, però entre capes no es fa. Per poder normalitzar entre capes s’utilitzarà el batch normalization.

### *ReLU*

És un dels tipus de funcions d’activació. Possiblement el més utilitzat en l’actualitat. S’encarrega de transformar *l’output* d’una neurona fent passar aquest output per una funció que transformarà el seu valor. Si el valor que rep la funció és negatiu, retornarà un 0 i si en canvi és positiu, retornarà el mateix valor que ha rebut. La funció representada gràficament seria:

### *Softmax*

És un altre tipus de funció d’activació que s’aplica a la sortida de les neurones. En aquest cas el que fa és transformar tots els valors que rep com a *input* per tal d’expressar cadascun d’aquests valors com a probabilitat. La suma de tots els *outputs* donarà probabilitat 1.



### *Residual Block*

Igual que en algun dels casos anteriors, no es tracta d’una xarxa com a tal, sinó una estructura o bloc format per capes de les ja esmentades. La característica d’aquesta estructura és que l’*input* passa a través de les capes de neurones que formen el bloc fins que finalment surt un *output* de l’última capa i a aquest *output* se li suma l’*input* inicial. Un exemple visual és més entenedor:

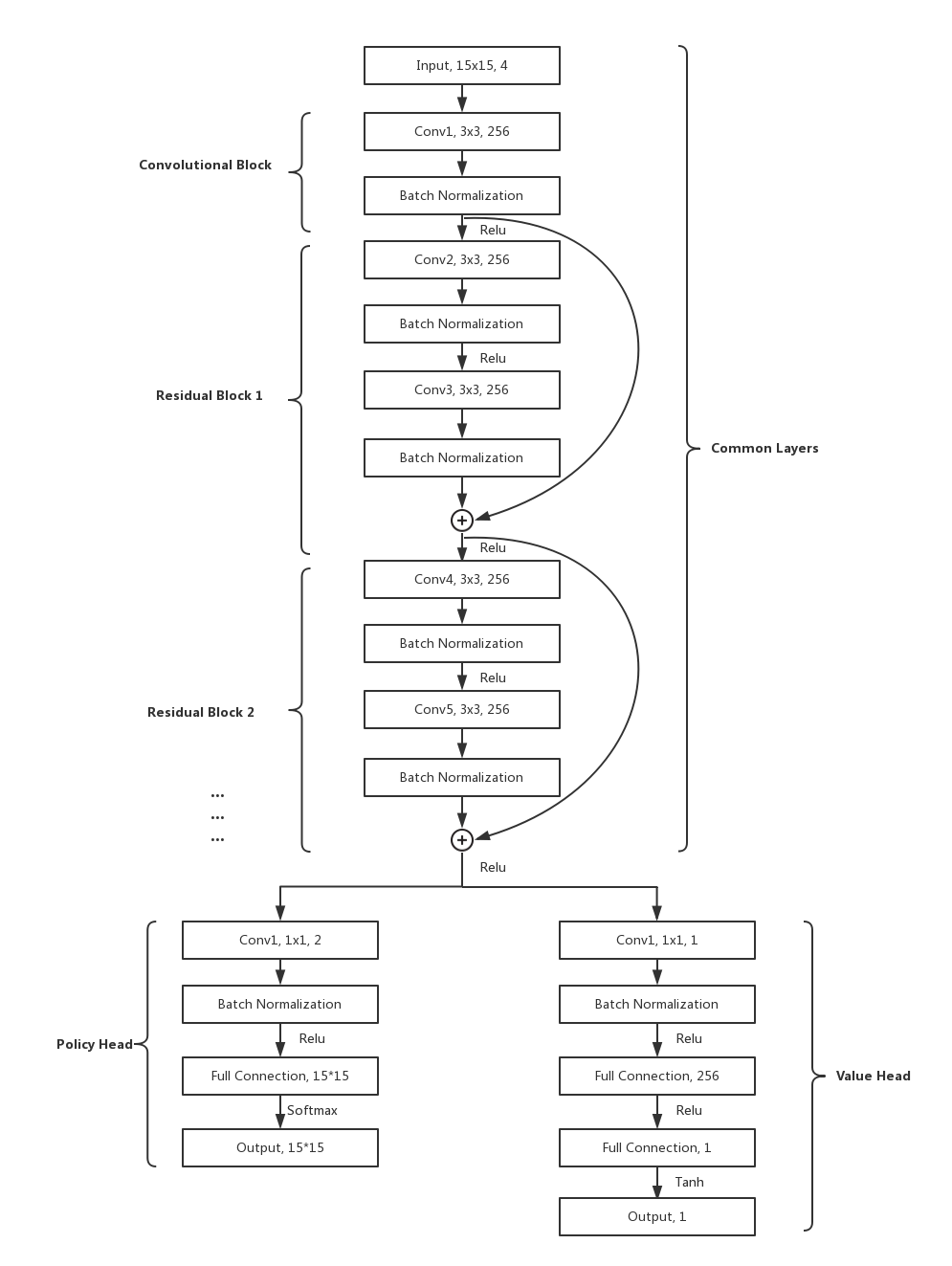
# 

# Funcionament

Tal i com havíem explicat, no és possible que el programa explori totes les possibles jugades possibles i que les vagi estenent fins al resultat final de totes les partides que es poden generar partint d’aquell estat. Per tal d’emmagatzemar menys informació i agilitzar el procés s’utilitza una poda que ens estalvia explorar jugades poc probables que passin. Per evitar simular aquest tipus de jugades haurem de guiar al programa cap a les millors jugades que pot fer. Gràcies als *values* dels nodes de l’arbre podrem fer aquesta poda.

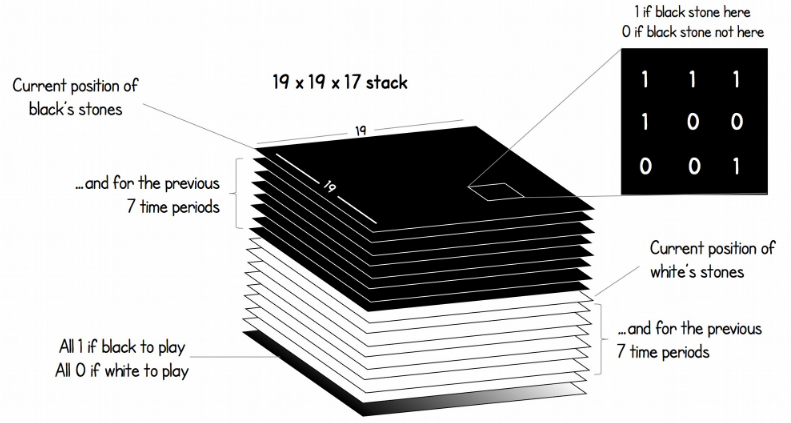
El problema serà doncs, decidir quant val l’atribut *value* i quin dels fills volem explorar (*policy*). En la versió original hi havia una xarxa neuronal encarregada de decidir el valor dels nodes i la política que decideix quin node explorar. En canvi, a la versió d’AlphaGo Zero es van compactar aquestes dues xarxes en una sola.

Aquesta xarxa es composa per una capa convolucional, batch normalization i relu, seguit de 40 blocs residuals. Cadascun d’aquests blocs està format per les capes: convolucional, batch normalization, relu, convolucional i batch normalization. Després d’aquesta darrera capa es suma aquest *output* amb l’*input* de la primera capa del bloc. Per acabar, s’aplica una funció d’activació al valor d’aquesta suma. En aquest cas, una ReLU. A continuació la xarxa es separa en dues parts. Un calcularà el *value* del node i l’altre calcularà el *policy*.



## Entrada

L’entrada del programa seran “imatges” de 19x19 píxels que representaran cadascuna de les posicions del taulell (pixel = posició taulell), de manera que podran valdre 1 si es troba una fitxa allà o 0 si no hi ha cap fitxa. Per diferenciar entre fitxes blanques i negres, tindrem dues capes amb l’estat actual: una per indicar on hi ha fitxes blanques i l’altra per les negres. Per a aportar més informació a la xarxa també se li passaran imatges 19x19 amb les darreres 7 jugades de les fitxes blanques i unes altres 7 imatges amb l’estat del taulell en els últims moviments de les fitxes negres. Per acabar, una capa que estarà tota a zero moure fitxa blanca i tot uns si toca moure fitxa negra. En conclusió, l’entrada serà un tensor de 17 canals d’alçada 19 i amplada 19.



Després de les 40 capes residuals l’*output* s’envia a dues capes diferents. Cadascuna d’elles està connectada a unes poques capes fins que obtenim els valors de *policy* i *value.*

## *Policy Net*

Ens retornarà la probabilitat de fer un moviment o com de bo que seria realitzar aquell moviment. Aquesta part de la xarxa rep com a input la sortida de l’últim bloc residual. Està format per una capa convolucional, *batch normalization,* ReLU, *fully-connected* i *softmax*. La sortida final acaba sent una matriu quadrada amb els possibles moviments des d’una certa posició i la probabilitat de dur-los a terme d’entre tots els possibles moviments (la suma de tots els possibles moviments dóna 1 gràcies al softmax que hi ha al final de la xarxa).

## *Value net*

Retorna un únic valor entre -1 i 1 indicant si guanyem o perdem. Com més proper sigui a 1, voldrà dir que la xarxa interpreta que tenim moltes possibilitats de guanyar si fem aquest moviment.

Aquesta part de la xarxa està formada per capa convolucional, *batch normalization*, ReLU, *fully-connected*, ReLU i *fully-connected*.

## Entrenament

L’únic que ens queda veure és com ho fem per entrenar a aquesta xarxa. En la versió original, per saber quins eren els moviments més probables i poder podar la cerca descartant a la resta, el que es feia és tenir un gran recull d’informació (*dataset*) de partides de jugadors professionals. A partir d’aquestes jugades es determinava quins eren els millors moviments que es podia fer en cada moment. La xarxa intentava replicar el comportament dels grans experts en aquest joc. En certa manera, això et limitava a les capacitats d’aquests professionals i no et deixava anar més enllà. En la versió AlphaGo Zero es va decidir canviar aquest *dataset* i utilitzar-ne un de més punt, ja que havia arribat un punt en què el programa era millor que els professionals, de forma que la informació de les seves jugades ja no el feia millorar.

Intuitivament, si volem entrenar a la xarxa amb jugades de gent encara millor que els professionals, l’única opció que tenim és utilitzar les jugades del propi AlphaGo per a fer-lo millorar. Aquesta tècnica es coneix com a aprenentatge per reforç.

## Reinforced Learning

Es traduiria com a aprenentatge per reforç i consisteix en fer a la xarxa entrenar-se contra ella mateixa fins que aconsegueixi ser millor que la darrera versió. Un cop tenim una versió que guanya a la seva antecessora, aquesta es convertirà en el nou model de referència i tornarem a començar el procés d’entrenament de la xarxa per arribar a un nou model que sigui encara millor.

El programa aprendrà a base de prova i error. Haurem de tenir un revisor que s’encarregarà de dir al programa lo encertada o desencertada que ha estat la decisió presa.

Un exemple de la vida real seria ensenyar a un gos a sentar-se. D’un inici l’animal no té cap idea del que ha de fer, i probablement provi coses sense sentit. Quan el gos realitza l’acció desitjada, l’amo el recompensarà amb un premi. En un cas ideal, el gos voldrà minimitzar el nombre de cops que s’equivoca per emportar-se tots els premis possibles.

En el cas de la xarxa, l’amo (aquell que decideix què està bé o malament) serà la darrera versió entrenada del model. I el que intentarà el gos (model que estem entrenant actualment) serà minimitzar la *Loss function.*

Per realitzar aquest tipus d’entrenament cal seguir una sèrie de passes:

* Crear un ambient adequat: El primer que necessitem és crear l’entorn que permetrà a la xarxa comunicar-se amb l’entorn. En el nostre cas una forma d’indicar els moviments que es realitzen i transmetre la informació de l’estat del taulell i tenir un control de les jugades i moviments vàlids.
* Definir la xarxa: Veure quina és la millor manera de construir la xarxa per a que ens permeti resoldre el problema desitjat. En el nostre cas al tractar-se d’una xarxa que representa l’estat inicial com una imatge ens és molt útil fer ús de capes convolucionals.
* Entrenar i validar a la xarxa: Entrenem a la xarxa i cada certs *steps* (passes d’entrenament) validem la xarxa per assegurar-nos que funcioni amb altres partides que no siguin les que es fan servir per entrenar per evitar l’*overfitting*.
* Un cop hem aconseguit entrenar a un model que es considera suficientment superior a la versió anterior crearem un nou punt de control i seguirem entrenant amb aquest nou model.

# 

# Futures versions

Després d’AlphaGo Zero va sortir una altra versió encara més bona i més generalista anomenada AlphaZero. Aquest nou programa era capaç d’aprendre a jugar a diferent tipus de jocs de taula similars al Go, com els escacs o quatre en ratlla sense necessitat de que el programa conegués les regles dels jocs.

Aquest paper es centrava en la generalització de l’AlphaGo Zero per a altres jocs i explicava com podíem fer servir la mateixa xarxa per a jugar a diferents jocs sense que el programa en conegui les normes. Bàsicament, tal i com hem vist amb el *reinforcement learning*, la xarxa pot aprendre a base de prova i error. Si partim d’un model des de zero que no té cap mena de coneixement i utilitzem un programa que li indiqui en quins casos ha guanyat i ens quins ha perdut i ens quins casos està realitzant moviments no permesos la xarxa per si sola serà capaç d’acabar executant sempre moviments que l’acostin cap a la victòria, de forma que evitarà fer moviments incorrectes o que el facin perdre. D’aquesta manera hem fet que el programa aprengui a jugar a un joc sense tenir cap coneixement previ sobre el joc o les seves normes.

# Per què és innovador?

Fins a abans de l’aparició del *paper* d’AlphaGo (la primera versió), guanyar a jocs més complexos que els escacs s’havia deixat de banda, ja que les limitacions que hem vist feien que no fos viable. En altres jocs de taula més senzills es podia intentar preveure tots els possibles moviments, o com a mínim tenir un heurístic el suficientment potent com per guiar-nos cap a la victòria. Per exemple, en el cas de les dames, el nombre de fitxes de cada jugador és un heurístic suficientment potent per crear un algorisme que pugui guanyar a una persona.

Amb el Go no passa això, ja que el nombre de fitxes no donen informació de qui està guanyant. Més aviat caldria fer una ullada al taulell en general per captar certs patrons que ens facin veure qui està en avantatge.

La novetat del programa creat per DeepMind és que es va entrenant i millorant contínuament per augmentar les seves capacitats fent-lo jugar a milions de jocs contra versions modificades de si mateix. Això permet a AlphaGo predir els propers moviments i determinar i avaluar aquestes posicions (si guanyarem o perdrem). El programa mira cap al futur i té en compte els possibles moviments i permutacions, passant per diverses simulacions abans de seleccionar aquella que considera més probable per portar-nos a la victòria. Les xarxes neuronals combinades en les diferents capes i en els dos extrems finals (*policy net* i *value net*) eviten que es faci feina de més: la xarxa que calcula el *policy* ajuda a reduir l'amplitud i ramificació dels moviments que s’analitzen, mentre que la xarxa que calcula el *value* ens estalvia haver de jugar internament la simulació de la partida complerta per arribar a la conclusió de si guanyarem o perdrem.

L’any 2015 va ser el primer cop que una màquina guanyava a un campió mundial de Go. Tot i que pugui no semblar massa rellevant, aquest fet va significar un punt d’inflexió en el món de la intel·ligència artificial, ja que després d’un llarg temps sense grans avenços es veure com les xarxes neuronals, el *deep learning* i la IA en general s’acostava cap a la seva finalitat: convertir-se en intel·ligències capaces de pensar i raonar per elles mateixes.

Alguns dels experts que van jugar contra el programa asseguraven que en els moviments i les estratègies que seguia es podia percebre un alt grau de comprensió del joc i de raonament per a generar jugades elaborades.

Al fer servir tècniques innovadores com el *reinforced learning* (entrenament reforçat), AlphaGo Zero va posar-se a un nivell molt superior que la resta de programes que també feien tasques similars.

És un model que es basa en la intuïció i en explorar aquells camins que li puguin semblar més adients. Aquesta forma de treballar ha servit de base per a futurs treballs que també han fet ús d’entrenament reforçat i d’estructures similars per a obtenir informació de la partida (guardar 17 capes d’una imatge del taulell).

De fet, la idea rere aquest model és tan potent que va permetre en poc temps estendre el programa per a que aprengués a jugar a altres tipus de jocs entrenant-se des de zero sense cap mena de coneixement previ del joc ni fent servir informació de jugades de persones.

AlphaGo ha demostrat que podem entrenar un sistema d'aprenentatge automàtic per emular les decisions o accions que seleccionaria un humà en un entorn relativament complex.

# Beneficis per a DeepMind i Google

DeepMind és una empresa de Google centrada en la recerca i innovació en intel·ligència artificial. Estan centrats en el món dels videojocs d’arcade i dels jocs clàssics. Els *papers* més importants que han publicat són els relacionats amb l’AlphaGo. Això els ha permès posicionar-se com a una empresa prestigiosa i reconeguda en el món de les IAs.

Gràcies a la publicació d’aquests articles i a la creació del programa han pogut seguir investigant en temes similar. Per exemple en entendre com funcionen certs jocs d’arcade únicament rebent com a entrada els píxels del joc i sent capaços d’aprendre i millorar fins al punt de guanyar el joc.

En l’actualitat DeepMind està centrat en altres projectes a banda del món dels videojocs, com pugui ser AlphaFold, que realitza prediccions de l’estructura de les proteines i dels seus plecs.

La victòria d’AlphaGo en el seu primer campionat es va produit una setmana abans de que Google fes pública la seva llibreria per a l’aprenentatge profund anomenada Tensorflow. Aquesta llibreria, juntament amb Tensorflow, són les més utilitzades per a les IAs més recents que veiem setmana rere setmana. Possiblement AlphaGo influenciés en la publicació de la llibreria després de l’èxit del model que havien creat. Aquestes llibreries públiques són les que han permès que a dia d’avui puguem parlar amb un xat de text i que sembli que parlem amb una persona, o que puguem crear imatges i retrats a partir d’una frase.

# Impacte en la societat

En definitiva, AlphaGo ha servit per a introduir nous conceptes com l’aprenentatge per reforç i ha fet que models més recents utilitzin el mateix tipus de tècniques. També ha afavorit al desestancament que hi havia fa uns anys en el món de la intel·ligència artificial amb la publicació de les llibreries més populars avui en dia i que ha impulsat tot aquest món a un ritme frenètic que a fet que en pocs anys s’hagin fet millores impensables que ens han portat a estar a l’era daurada de la intel·ligència artificial, de manera que mica en mica s’està introduint a la població i a la societat en general en aquest món i s’està aconseguint que guanyin interés, creant grans comunitats d’interessats en la matèria que dia a dia impulsen per a fer de les IAs un coneixement públic a l’abast de tothom.

# 

# Bibliografia

Per poder investigar la nostra font d’informació principal seran els dos papers que s’han publicat pels autors originals: “*Mastering the game of Go with deep neural networks and tree search”* (AlphaGo)i *“Mastering the game of Go without human knowledge”* (AlphaGo Zero). En aquests papers s’explica en detall el funcionament dels programes però creiem que també és necessari cercar en altres fonts d’informació que puguin explicar de manera més senzilla el funcionament del programa, ja que els papers originals tenen algunes parts massa tècniques que al principi costa entendre.

*Mastering the game of Go with deep neural networks and tree search* [Consulta: 10 d’octubre]*:* <https://www.nature.com/articles/nature16961>

*Mastering the game of Go without human knowledge* [Consulta: 10 d’octubre]*:* <https://www.nature.com/articles/nature24270>

*AlphaGo Zero Implementation Using Reinforcement Learning* [Consulta: 11 d’octubre]*:* <https://drive.google.com/file/d/1ySijuY1zqv6LzkQSo0Fgf7zP9mCWvexQ/view>

*The 3 Tricks That Made AlphaGo Zero Work* [Consulta: 20 d’octubre]*:* <https://hackernoon.com/the-3-tricks-that-made-alphago-zero-work-f3d47b6686ef>

*Training Deep Convolutional Neural Networks to Play Go* [Consulta: 23 d’octubre]*:* <https://github.com/brilee/MuGo/blob/master/papers/clark15.pdf>

*Why AlphaGo is a bigger game changer for Artificial Intelligence than many realize* [Consulta: 17 de Novembre]: <https://medium.com/@roelljr/why-alpha-go-is-a-bigger-game-changer-for-artificial-intelligence-than-many-realize-64b00f54a0>

*A Tutorial on AlphaGo Zero* [Consulta: 17 Novembre] <https://github.com/zhihanyang2022/alpha-zero/blob/main/alphazero.pdf>

*AlphaGo Zero: Starting from scratch* [Consulta: 17 Novembre] <https://www.deepmind.com/blog/alphago-zero-starting-from-scratch>

*Why AlphaGo is a bigger game changer for Artificial Intelligence than many realize* [Consulta: 4 Desembre] <https://medium.com/@roelljr/why-alpha-go-is-a-bigger-game-changer-for-artificial-intelligence-than-many-realize-64b00f54a0>

*Why is Google's Go win such a big deal?* [Consulta: 4 Desembre] <https://www.theverge.com/2016/3/9/11185030/google-deepmind-alphago-go-artificial-intelligence-impact>

*Why DeepMind AlphaGo Zero is a game changer for AI research*

[Consulta: 23 Desembre] https://hub.packtpub.com/deepmind-alphago-zero-game-changer-for-ai-research/