TREBALL FINAL DE GRAU

Implementació d'una IA basada en Machine Learning

Implementació d'un agent que aprèn a jugar a l'Snake amb l'ús d'aprenentatge per reforç

Adrià Aumedes Floreta
Tutora: Ester Bernadó Mansilla
Grau en Disseny i Producció de Videojocs

CURS 2020-21

TecnoCampus
Escola Superior Politècnica

Centre adscrit a la

upf. Universitat Pompeu Fabra *Barcelona*

Abstract

This project explores the implementation of the Deep Q-Learning algorithm developed by DeepMind with the goal of getting an agent to play Snake optimally. Deep larning libraries are used to create a model based on artificial neural networks that has a formal description of the environment as input and tries to predict the value for each possible action. Agent averaged 12 points per game.

Resum

En aquest projecte s'explora la implementació de l'algorisme Deep Q-Learning desenvolupat per DeepMind amb l'objectiu que un agent aconsegueixi jugar a l'Snake de forma òptima. Es fa ús de llibreries de deep larning per crear un model basat en xarxes neuronals artificials que té com a input una descripció formal de l'entorn i com a output intenta predir el valor per cada acció possible. L'agent ha obtingut una mitjana de 12 punts per partida.

Resumen

En este proyecto se explora la implementación del algoritmo Deep Q-Learning desarrollado por DeepMind con el objetivo que un agente consiga jugar al Snake de forma óptima. Se hace uso de librerías de deep larning para crear un modelo basado en redes neuronales artificiales que tiene como input una descripción formal del entorno y que intenta predecir el valor por cada acción posible. El agente ha obtenido una media de 12 puntos por partida.

Índex

Índex de figures

Índex de taules

1 Introducció

La Intel·ligència artificial (IA) sempre s'ha considerat un camp important dels videojocs i des dels inicis ha estat en constant desenvolupament. La indústria ha utilitzat la IA com una forma més de donar una experiència al jugador atorgant-li comportaments interessants que augmentaven el valor del joc. La IA en videojocs s'ha utilitzat en jocs com Mario (Nintendo, 2020) amb el comportament dels enemics, Pacman i els seus fantasmes o durant l'última dècada amb sagues com Grand Theft Auto (Marquez, 2021) i Need for Speed (EA, 2021) on s'hi poden trobar NPCs (Non-Player Character), que són personatges controlats per la IA i no disponibles pel jugador, que condueixen o altres que disparen amb certa habilitat.

Hi ha diferents algorismes que s'utilitzen per programar IA entre els que hi ha les màquines d'estats i els arbres de comportament. Aquests, són dos tipus d'algorismes semblants, molt utilitzats per la seva senzillesa però tot i així capaços de crear comportaments útils que per la major part de la IA necessària en els videojocs és suficient. Un exemple típic és el d'un NPC amb tres comportaments: patrullar entre uns punts, perseguir el jugador i atacar el jugador. La senzillesa d'aquests comportaments fa que la forma més òptima de programar-los sigui utilitzant aquests algorismes on cada comportament pot correspondre a un estat dins la màquina d'estats.

Un dels àmbits de la IA que està sorgint amb més força és el deep reinforcement learning que és un tipus de Machine Learning en el que s'agrupen algorismes capaços d'aprendre a partir de la interacció d'un agent amb un entorn utilitzant una recompensa com a feedback de l'aprenentatge per saber si l'agent ha realitzat una acció més o menys bona. Aquets algorismes tenen el punt fort de poder crear comportaments molt complexos que serien impossibles de crear utilitzant màquines d'estats o arbres de comportament.

Aquest projecte s'introdueix al concepte de deep reinforcement learning i concretament al deep Q-Learning com a alternativa a l'ús d'altres tipus d'IA més senzills sense pretendre substituir-los. Aquests tipus d'IA, encara que continuen sent útils, no són gens flexibles o dinàmics sinó només limitats al comportament

exacte que se'ls ha programat. Quan aquests algorismes estan dissenyats, implementats i testejats, tenen limitacions d'adaptabilitat i evolució si canvien les condicions de l'entorn, fet que produeixen comportaments predictibles. Per contra, els algorismes deep reinforcement learning són altament adaptables a diferents condicions ja que són capaços de crear una abstracció de l'entorn per utilitzar-la de forma més genèrica i eficient. La capacitat de generalització prové de l'ús de funcions d'aproximació, que acostumen a ser xarxes neuronals i, que permeten, com a exemple, que un agent entengui una posició, no com un punt fix a l'espai, sinó com una àrea.

El deep reinforcement learning ha sigut força explorat els últims anys amb projectes com el d'Openai (Openai, 2020), DeepMind (Mnih et al, 2013) (Mnih et al, 2015), TD Gammon (Tesauro, 1995) i AlphaGo Zero (Silver et al, 2016). Els projectes anteriors tracten d'implementacions d'agents que aprenen a jugar a nivell humà o superhumà a jocs coneguts com el Go, el BackGammon i Space Invaders o en el cas d'Openai implementacions d'agents amb la capacitat de cooperar en un entorn competitiu (més informació als referents del capítol 3).

Les investigacions han utilitzat videojocs com a base però sempre amb la intenció de demostrar el funcionament d'un algorisme, no amb la intenció de poder ser implementat al videojoc per millorar l'experiència de l'usuari. És per això que aquestes propostes tenen una finalitat més de fonamentació teòrica que de jugabilitat, amb les excepcions d'AlphaStar (DeepMind, 2020) i Openai Five (Openai, 2019) que són implementacions que exploren els límits del deep reinforcement learning amb tasques com la creació d'un agent entrenat per jugar a Starcraft 2 o un equip de 5 agents entrenats per jugar al joc Dota 2. Aquests dos casos anteriors són considerats actualment l'extrem al que s'ha arribat fins ara en la investigació del deep reinforcement learning. El que indiquen aquestes propostes és el potencial que té utilitzar models de deep learning per entrenar agents en un entorn que compleix la propietat de Markov, és a dir, qualsevol futur estat del joc pot dependre només de l'estat actual (Yannakakis i Togelius, 2018).

Els algorismes d'aprenentatge per reforç i sobretot la versió amb *deep learning* no són algorismes trivials, ni d'entendre el funcionament teòric ni d'implementar-los. Aquest projecte parteix des de zero per poder arribar a obtenir la base teòrica i pràctica necessària per aprofundir en el tema. Una vegada aconseguits els

coneixements bàsics, el següent pas (més enllà d'aquest projecte) ha de ser poder adaptar un algorisme de deep reinforcement learning a un videojoc no existent on la IA tingui un paper important i no sigui només una complementació del joc.

1.1 Objectiu del projecte

Per abordar correctament l'objectiu del treball, convé definir la diferència que hi ha entre una IA científica, aplicada a la resolució de problemes amb agents entrenats per adaptar-se a un entorn de forma òptima i una IA per videojocs, que no només ha de ser funcional sinó que també s'ha d'assegurar una experiència d'usuari i un cert control per part del dissenyador. En aquest projecte no s'abordarà com aplicar una IA complexa a un videojoc, amb la intenció de ser jugat per un jugador, sinó com crear una IA complexa que solucioni una tasca senzilla amb la finalitat d'estudiar-ne el funcionament. De forma genèrica i conclusiva s'avaluarà la perspectiva de poder ser implementada en un videojoc amb les modificacions necessàries.

Sabent això, l'objectiu del projecte és crear una tasca basada en deep Q-learning on un agent ha de jugar al famós joc de l'Snake de forma òptima, és a dir, aconseguint la màxima quantitat de punts mentre intenta no morir.

1.2 Estructura de l'obra

Aquest apartat exposa com s'estructura tot el document.

El primer capítol correspon a la introducció on s'introdueix la IA en els videojocs i els seus algorismes més típics. Després s'entra en el tema central d'aquest projecte, el deep reinforcement learning, on s'ofereix una visió genèrica i alguns exemples d'èxit a la indústria. Per últim s'explica de forma genèrica l'objectiu principal i de quina manera s'aborda.

El següent capítol conté tota la teoria referent a la intel·ligència artificial, el Machine Learning i més concretament a l'aprenentatge per reforç. D'això últim se'n detallen tots els elements que en permeten el funcionament i s'expliquen els dos algorismes utilitzats, Q-Learning i Deep Q-Learning. També s'hi pot trobar informació sobre les eines i les llibreries utilitzades i sobre la metodologia amb la que es fonamenta el treball. En el capítol 3 es presenten els referents del treball i la relació que tenen a

nivell de prototip i desenvolupament dels algorismes. També s'explica què en destaca i perquè són importants per aquest projecte.

A continuació, es presenten les contribucions del projecte pròpiament dites. En primer lloc, es presenten els objectius, que s'han descompost en un objectiu principal i varis objectius secundaris. Seguint en la línia del treball propi, el capítol 5 presenta el disseny metodològic que s'ha seguit per desenvolupar aquest projecte. També es defineixen les fases del projecte i aquestes es poden veure exposades a través del temps amb un cronograma.

El sisè capítol conté la implementació dels prototips i del projecte final de l'Snake. Cada una de les tres implementacions té un apartat de disseny, on es defineix el funcionament de la tasca i, en el cas del projecte final, també s'explica com funciona l'Snake. El següent apartat està destinat a mostrar com s'han configurat els paràmetres de la tasca i com s'ha traslladat el disseny a la implementació amb codi. El tercer i últim apartat analitza els resultats obtinguts després que l'agent realitzi l'entrenament, mitjançant gràfiques i taules de valors on es modifiquen els paràmetres de la tasca.

El penúltim capítol correspon a les conclusions del treball. Es tornen a mostrar els objectius i s'indica si han estat completats. També s'exposa quina pot ser la implementació, dels algorismes utilitzats en aquest treball, en videojocs on complementin al jugador i per acabar s'esmenten les possibles línies de futur que es poden explorar. Finalment, a l'últim capítol es poden trobar totes les referències bibliogràfiques que s'han utilitzat.

2 Marc teòric

En aquest apartat s'exposa tota la teoria relacionada amb la intel·ligència artificial, tan a nivell general com en videojocs, i el Machine Learning. Concretament es detalla el concepte d'aprenentatge per reforç i els seus components, a més de presentar els dos algorismes utilitzats en aquest projecte. Finalment s'explica el material que s'utilitza per dur a terme el projecte i la metodologia utilitzada.

2.1 Fonaments de la Intel·ligència artificial

La intel·ligència artificial (IA) es definida per la Real Academia Española com la "disciplina científica ocupada de crear programes informàtics que executen operacions comparables a les que realitza la ment humana, com l'aprenentatge o el raonament lògic" (Real Academia Española, 2021).

El primer que va intentar definir la IA va ser Alan Turing, creador de la màquina de Turing i també del Test de Turing. El test de Turing (Oppy i Dowe, 2020) és una proba que permet afirmar si una màquina és intel·ligent o no. En el test, si una persona, després de parlar una estona amb una entitat, no sap distingir si parlava amb una màquina o amb una persona, significa que la màquina es considera intel·ligent. Per aconseguir passar aquesta prova la IA ha de tenir capacitats com el reconeixement del llenguatge natural, aprenentatge, raonament i representació del coneixement (Serrano, 2016).

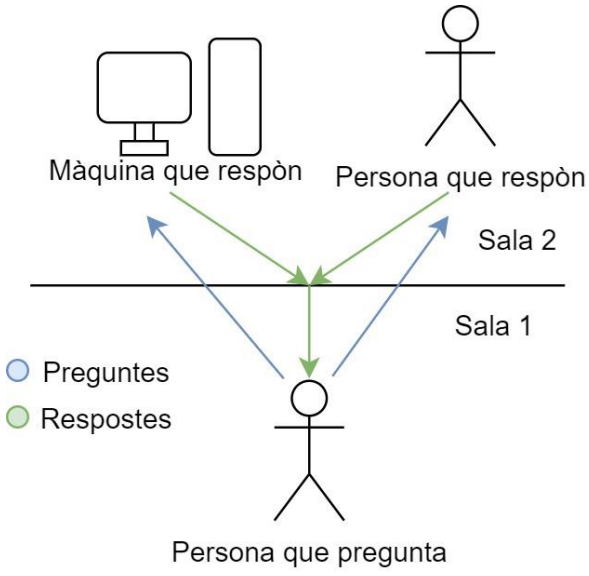


Figura 2.1 Diagrama de funcionament del Test de Turing. Font: Elaboració pròpia.
Turing no va ser l'únic ha definir què és una màquina intel·ligent, fins a dia d'avui han aparegut altres propostes com que la màquina és intel·ligent si pensa com un humà o que la màquina és intel·ligent si pensa i actua de forma racional. Aquesta última proposta es fonamenta en el concepte d'agent, que aprèn basant-se en la informació que té de forma local i a partir d'aquí pren decisions. Si en comptes d'un agent és un grup d'agents es considera un sistema multiagent (Serrano, 2016).

Segons investigadors de la Universitat d'Alacant, consideren que hi ha cinc grans categories dins la intel·ligència artificial (Escolano et al, 2003) (Pannu, 2015). Aquestes són:

- **Processament del llenguatge natural**: definit com l'habilitat per entendre i donar resposta a un llenguatge natural. Dins d'aquesta àrea hi ha la lingüística computacional, el reconeixement del discurs i la traducció automàtica entre d'altres sub-categories.
- **Sistemes experts**: engloba els sistemes que incorporen l'experiència de personal qualificat per aproximar les deduccions a la realitat.
- **Robòtica**: sistemes que permeten la navegació sobre el terreny i l'habilitat de manipular objectes. Aquí s'inclouen les tecnologies militars, industrials i de transport i navegació entre d'altres.
- **Percepció**: l'habilitat per analitzar una escena detectada relacionant-la amb un model intern que representa el coneixement sobre el món. Inclou la visió artificial i el reconeixement de patrons entre d'altres.
- **Machine Learning**: sistemes que permeten desenvolupar una representació interna i una sèrie de regles que poden ser utilitzades per predir comportaments i relacions entre entitats o objectes reals o virtuals.

2.2 La Intel·ligència artificial en els videojocs

2.2.1 Història de la Intel·ligència Artificial en els videojocs

La intel·ligència artificial apareix en els videojocs juntament amb els inicis de la informàtica als anys 50. Nim és considerat el primer videojoc en contenir una IA senzilla basada en la teoria de jocs combinatòria (Ramírez, 2020). Aquest és un joc d'estratègia matemàtica per torns en el que els jugadors o el jugador i la IA van

agafant fitxes de diferents pilons, la quantitat que creguin i, guanya el que agafa la última fitxa. Al 1952, un any després de Nim, apareix una versió digital del tres en ratlla on el software podia masteritzar el joc. Arthur Samuel va desenvolupar, poc després, un software per les dames basat en una forma primitiva d'aprenentatge per reforç. A partir d'aquí, la recerca va estar sempre enfocada a la creació d'algorismes per als jocs de taula clàssics, per la gran complexitat, les regles estructurades i la informació perfecta en tot moment (Yannakakis i Togelius, 2018).

Amb les millores, als anys 70, en la potència i qualitat del hardware i la capacitat del software, també van arribar millores en els videojocs en els que es van començar a incloure algorismes senzills que permetien donar certa millora en l'experiència de l'usuari, com en el cas de Pong, Space Invaders i Galaxian (Ramírez, 2020).

El següent pas important per la IA va arribar amb el joc Pac-Man al 1980 (Ramírez, 2020). En aquest joc, el jugador controla un personatge amb el propòsit de menjar tots els punts del laberint per passar el nivell. Per complicar-ho hi ha quatre fantasmes que intentaran acabar amb ell. Cada un d'aquests té un comportament diferent que demostra la seva personalitat; el vermell es dedica a perseguir directament el jugador ja que sempre té com a punt objectiu la tile en la que està el Pac-man, el rosa té com a objectiu anar a on es dirigeix el jugador i no on està en aquest moment tenint en compte la seva posició i direcció, el blau té un comportament complicat ja que intenta anar on es dirigeix el jugador però té en compte la posició del fantasma vermell per calcular més bé la seva tile objectiu i així tallar millor el jugador, tot i que al principi pot fallar una mica. Per últim hi ha el fantasma taronja que pel comportament erràtic que té, dona la impressió que no persegueix al jugador, sinó que fa voltes pel laberint al seu compte. En realitat si que persegueix al jugador si aquest es troba a més de 8 tiles de distància i si es troba a menys de 8, intentarà evitar-lo dirigint-se a la cantonada inferior esquerra del laberint (Birch, 2010).

Figura 2.2 Nivell del videojoc Pacman amb els quatre fantasmes movent-se pels camins. Font: El País.

La IA de Pac-man va ser important sobretot degut als patrons que utilitzaven els fantasmes (Ramírez, 2020). Als 80 també es va començar a utilitzar tècniques de generació de contingut procedural, on un algorisme crea part del contingut en temps d'execució del joc en comptes de ser predissenyat. Exemples d'això en són Rogue i Elite (Yannakakis i Togelius, 2018).

Als 90, gràcies a les noves millores en el hardware i el software, es van millorar els algorismes bàsics com les màquines d'estat finites o els sistemes de cerca de camins. A més a més es van incloure els arbres de cerca Monte Carlo als algorismes ja que són capaços de conèixer totes les possibles opcions en un cert moment de la partida i, juntament amb una màquina d'estats finita permetia a jocs com Wolfstein i Doom millorar l'experiència general del joc (Ramírez, 2020). Al 1997 es va aconseguir per primer cop que un ordinador obtingués un nivell superhumà jugant als escacs gràcies al software d'IBM Deep Blue, basat en l'algorisme Minimax (Yannakakis i Togelius, 2018).

2.2.2 La intel·ligència artificial als videojocs actualment i la seva utilització

Des del punt de vista del rendiment de la IA, actualment els algorismes han millorat molt i en totes les tipologies d'especialització. A grans trets tenim els següents tipus d'algorismes (Yannakakis i Togelius, 2018):

- **Creació de comportaments**: màquines d'estat finites, arbres de comportament i IA basada en utilitat. Aquests algorismes són senzills d'implementar (inclús pel dissenyador) i tenen una capacitat decent d'oferir comportaments interessants. Solen utilitzar estats o nodes on es programen els comportaments. Entre un comportament i un altre es canvia mitjançant una condició. Poden ser utilitzats en NPCs que tinguin un número finit d'estats de comportament. D'aquests, el més utilitzat avui en dia és l'arbre de comportament.
- **Arbres de cerca**: cerca uniformada, Minimax, arbre de cerca Monte Carlo, etc. Aquests algorismes es basen en desplegar totes les opcions de resolució de la tasca en format d'arbre i a partir d'aquí buscar el camí més òptim per resoldre-la. Són útil per resoldre tasques on la intel·ligència artificial té coneixement total de l'estat del joc com és el cas dels escacs, el Go i el Shogi entre d'altres.
- **Computació evolutiva**: cerca local i algorismes genètics. Són algorismes que intenten trobar una solució entre un conjunt, optimitzant la cerca amb l'ús d'heurístiques. Útil sobretot en sistemes de cerca de camins pels NPCs (pathfinding en anglès) com el cas de l'algorisme A*.
- **Aprenentatge supervisat**: xarxes neuronals artificials, màquina de vector de suport. Algorismes que intenten classificar predir valors basant-se en unes dades correctament etiquetades. S'utilitza aquesta forma d'aprenentatge juntament amb les versions de deep learning de l'aprenentatge per reforç. (veure secció 2.3)
- **Aprenentatge no supervisat**: K-Means, xarxa de Hopfield (Serrano, 2016), etc. Algorismes que extreuen característiques útils d'unes dades d'entrada per després agrupar-les segons la seva semblança. (veure secció 2.3)
- **Aprenentatge per reforç**: Q-Learning, Deep Q-Learning, Sarsa, Policy Gradient, etc. Són algorismes basats en la interacció amb un entorn, que retorna un senyal numèric per permetre un aprenentatge. En els videojocs són capaços de proporcionar comportaments molt complexos impossibles de realitzar amb altres metodologies explicades als punts anteriors. (veure secció 2.3)

Tots aquests tipus d'algorismes s'utilitzen una quantitat de vegades semblant però per raons diferents. A la taula següent, extreta de (Yannakakis i Togelius, 2018), es veu què pot fer cada tipus d'algorisme:

	Play Games		Generate Content		Model Players	
	Winning	Experience	Autonomously	Assisted	Experience	Behavior
Behavior Authoring	●	●				
Tree Search	●	○	○	○		
Evolutionary Computation	●	○	●	●	●	
Supervised Learning	○	●			●	●
Reinforcement Learning	●	○				
Unsupervised Learning				○	○	●
Total (Dominant)	5 (4)	5 (2)	2 (1)	3 (1)	3 (2)	2 (2)

Taula 1 Mètodes de IA segons si són dominants (●) o secundaris (○) per cada àrea important en què pot ser utilitzada la IA . Font: Yannakakis i Togelius, 2018, p.261.

Tal com es mostra a la taula 1, els diferents tipus d'algorismes presentats a la secció anterior, s'utilitzen per diferents raons segons les seves fortaleses. Tots els espais buits de la taula indiquen àrees amb potencial per ser explorades i oferir encreuaments prometedors entre les àrees i els diferents tipus d'algorismes. Així podem dividir l'ús segons si:

- **Juguen a jocs**: inclou jugar per guanyar i jugar per l'experiència. A més d'això la IA també pot controlar el personatge jugador o el personatge no jugador.
- **Generen contingut**: inclou el contingut procedural i el contingut assistit, és a dir que modifiquen paràmetres.
- **Modelen al jugador**: inclou formar l'experiència del jugador i modelar-ne el comportament.

Com resumeixen Georgios N. Yannakakis i Julian Togelius, des del punt de vista de l'experiència d'usuari també hi ha diferents maneres d'enfocar la utilització de la IA.

Per una banda la IA millora el joc. Això és perquè ha de jugar bé i de forma atractiva pel jugador, cosa que permet crear sistemes de dificultat i de balanceig dinàmics que personalitzen i milloren l'experiència del jugador. A més a més, pot controlar el

personatge del jugador o un personatge no jugable que, a partir d'aquí, es pot definir si la IA juga contra el jugador o el complementa d'alguna manera.

Per altra banda la IA pot augmentar el contingut del joc i millorar-lo. Augmenta el contingut perquè els algorismes poden generar contingut proceduralment o múltiples instàncies del mateix però variant simples paràmetres, fet que permet crear jocs amb rejugabilitat. Millora el contingut perquè quan la generació de contingut està associat amb aspectes del joc, permet crear que sigui molt més personalitzable i/o adaptatiu.

2.3 Machine Learning

El Machine Learning (ML) és definit com "una forma d'estadística aplicada amb major èmfasis en l'ús d'ordinadors per estimar estadísticament funcions complicades i en menor èmfasis en la demostració dels intervals de confiança al voltant d'aquestes funcions" (Goodfellow et al, 2016) p.96. El Machine Learning també es pot definir de forma menys tècnica com un tipus d'intel·ligència artificial que proporciona a les computadores la capacitat d'aprendre des de les dades sense ser programades explícitament (Beunza et al, 2020).

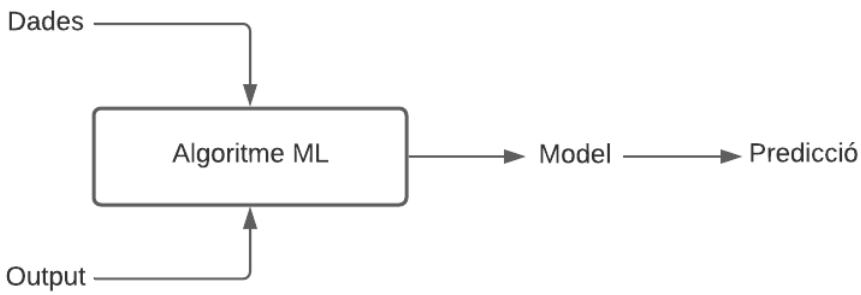


Figura 2.3 Estructura d'un procés de Machine Learning basat en aprenentatge supervisat. Font: Elaboració pròpia a partir de **(Beunza et al, 2020)**, p.10.

Els algorismes de Machine Learning es divideixen en tres branques (Goodfellow et al, 2016):

- **Aprenentatge supervisat**: és el conjunt d'algorismes que utilitza dades etiquetades com a entrada per després predir un resultat. Pot aprendre gràcies al còmput de l'error entre el resultat obtingut i el que hauria d'obtenir. Solen resoldre tasques de classificació quan l'algorisme ha d'especificar la

categoria a la que pertany un valor d'entrada o de regressió quan l'algorisme ha de predir un valor numèric basant-se en certes dades d'entrada dependents, entre altres tipus de tasques.

- **Aprenentatge no supervisat**: són els algorismes que es basen en detectar patrons en un conjunt de dades no etiquetades que després han de comparar amb la memòria interna. Acostumen a resoldre tasques de clusterització quan busquen i agrupen conjunts de dades numèriques semblants o diferents a les dades d'entrada per saber a quin grup pertanyen.
- **Aprenentatge per reforç**: conjunt d'algorismes que interaccionen amb l'entorn per aprendre.

2.3.1 Conceptes

El Machine Learning està ple de conceptes complexos i alguns s'utilitzen en aquest projecte. Tant a l'explicació del marc teòric on s'utilitzen com a la implementació, es donen per entès. Per tant en aquest sub-apartat s'expliquen els conceptes que és important saber abans de llegir la resta del treball.

Deep Learning: és un tipus de Machine Learning (Goodfellow et al, 2016) i s'entén com la implementació més potent dels algorismes de ML (Beunza et al, 2020) en el que s'utilitzen models amb gran capacitat d'aprenentatge per resoldre problemes de gran complexitat.

Tasca: és el resultat final que es vol obtenir però no com (Goodfellow et al, 2016). Per exemple, és una tasca aconseguir predir, amb un cert grau d'encert, una malaltia a partir d'unes dades d'entrada. Com s'arriba a predir la malaltia, és a dir, utilitzant un o altre algorisme, no defineix la tasca sinó que es considera la manera d'afrontar-la.

Entrenament: fase en la que el model aprèn i optimitza els paràmetres interns mitjançant un conjunt de dades d'entrada i, en alguns casos, també de sortida.

Testeig: fase en la que el model ja ha sigut entrenat i per tant pot realitzar prediccions més o menys encertades sobre la tasca que es porta a terme a partir de noves dades d'entrada.

Generalització: habilitat d'un algorisme de Machine Learning de fer bones prediccions per les noves dades d'entrada no observades prèviament. Més

generalització implica més flexibilitat i capacitat d'adaptació de l'algorisme (Bishop, 1995).

Sobre entrenament (overfitting): ocorre un sobre entrenament quan l'error produït per la diferència entre les prediccions i els valors objectius del model no és prou petit durant la fase d'entrenament (Goodfellow et al, 2016). Dit d'una altra manera, a partir d'un cert moment el model deixa d'optimitzar i comença a aprendre característiques particulars i pertorbacions de les dades d'entrada que n'empitjoren el rendiment. Un model sobre entrenat perd la capacitat de generalitzar i falla quan fa prediccions amb dades d'entrada significativament diferents a les dades d'entrada utilitzades durant la fase d'entrenament (Allamy i Khan, 2014).

Infraentrenament (underfitting): ocorre un infraentrenament quan la diferència entre l'error de la fase d'entrenament i l'error de la fase de testeig és massa gran (Goodfellow et al, 2016). En altres paraules, el model no aconsegueix aprendre les característiques rellevants de les dades d'entrada perquè el model utilitzat no és adequat per la tasca (Allamy i Khan, 2014). Un model poc entrenat no aconsegueix optimitzar els paràmetres i per tant no pot encertar les prediccions.

2.3.2 Introducció a les xarxes neuronals artificials

Són models matemàtics i deterministes, adaptats a la computació com a representació del cervell biològic (Goodfellow et al, 2016). L'algorisme de xarxa neuronal es basa en una estructura formada per una certa quantitat de neurones agrupades per capes. Les neurones de cada capa estan enllaçades amb les neurones de la capa anterior i les de la posterior excepte les neurones d'entrada i de sortida que tenen només connexió amb les neurones de la capa adjacent. Cada neurona té una entrada per on arriba la senyal i una sortida per on es propaga la senyal. Tenen la capacitat de ponderar la senyal d'entrada que reben, constituint la base de l'aprenentatge de la xarxa neuronal (Flórez López i Fernández Fernández, 2008).

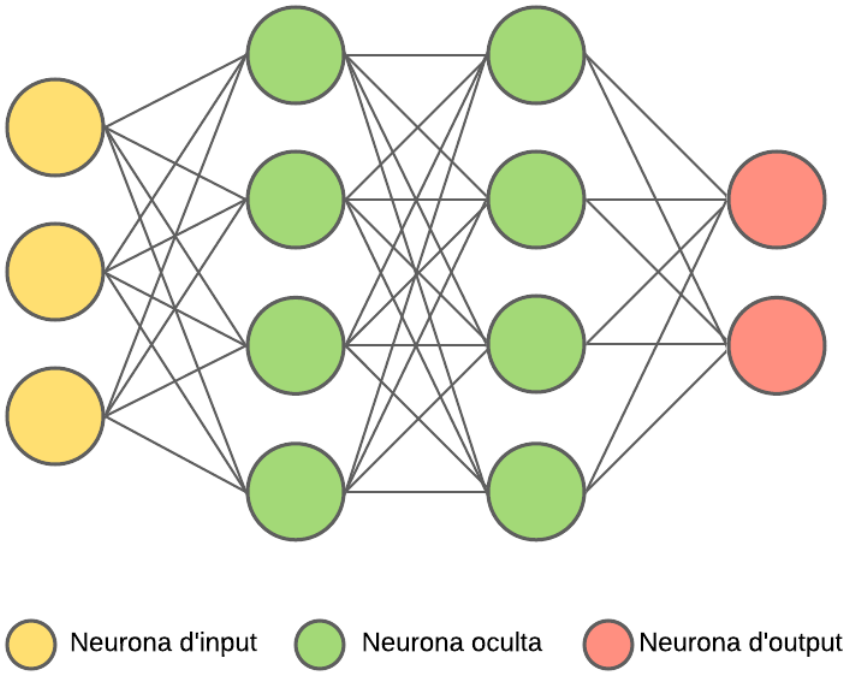


Figura 2.4 Arquitectura d'una xarxa neuronal profunda densament connectada o *FeedForward*. Font: Elaboració pròpia.

2.3.3 Introducció a l'aprenentatge per reforç

L'aprenentatge per reforç es defineix com aprendre a assignar accions segons la situació en la que es troba l'aprenent per tal de maximitzar una recompensa numèrica. L'aprenent no coneix quines accions ha de seleccionar, sinó que ha de descobrir quines accions li retornen la major recompensa realitzant-les (Sutton i Barto, 2018). Això es pot entendre com l'aprenentatge en base a la prova i error tal com ho fem els humans en moltes ocasions. Per exemple en el cas d'un nen que aprèn a caminar si aconsegueix caminar fins a un punt, rebrà una recompensa positiva ja que veurà com els seus pares se n'alegren. En canvi si cau, es farà mal i la recompensa per l'acció es considera negativa.

L'objectiu de l'aprenentatge per reforç és aprendre una estratègia per l'agent, experimentant amb l'entorn i reben recompenses simples. Amb l'estratègia òptima, l'agent és capaç d'adaptar-se a l'entorn per maximitzar les futures recompenses (Weng, 2018).

2.4 Xarxes neuronals artificials

Els següents conceptes són alguns dels termes més importants relacionats amb les xarxes neuronals.

Pesos i bias: Els pesos són els principals paràmetres d'aprenentatge de la xarxa neuronal. Tenen un valor decimal entre -1 i 1 que van optimitzant durant l'entrenament. Cada neurona té un pes per cada connexió amb una altra neurona però entre dos neurones que estan connectades hi ha el mateix pes en ambdós sentits. Els pesos controlen la quantitat d'influència que exerceixen els valors d'entrada en les prediccions (Kobran, 2020). Quan la xarxa està entrenada, els pesos acaben amb uns valors que es poden entendre com una contrasenya que permet crear prediccions encertades per una tasca específica. L'objectiu de l'entrenament, doncs, és balancejar els pesos correctament perquè acabin amb els valors més òptims per la tasca que s'ha de resoldre. El bias és un altre paràmetre de la xarxa neuronal i, com els pesos, també té el rang de valors decimal -1 a 1 i s'ha d'optimitzar. Hi ha un bias per cada neurona, que serveix per fer més robust l'aprenentatge garantint que la neurona pot arribar a activar-se (Kobran, 2020).

Propagació cap endavant (forward propagation): la propagació cap endavant és un mètode de la xarxa neuronal que fa prediccions a partir d'uns valors d'entrada. El procés d'aquest mètode és basa en acceptar uns valors d'entrada $\boldsymbol{x}$ que proporcionen la informació inicial, aquesta informació es propaga cap a les neurones de la següent capa i així fins a arribar a l'ultima capa que produeix les prediccions $\hat{\boldsymbol{y}}$ (Goodfellow et al, 2016).

Propagació cap endarrere (backpropagation): la propagació cap endarrere (Rumelhart et al, 1986) és un mètode de la xarxa neuronal que ajusta els paràmetres d'aprenentatge de la xarxa per minimitzar l'error entre els valors predits i els valors objectius del model. El procediment comença per calcular els gradients que són uns valors que mesuren el pendent i la direcció d'actualització dels paràmetres. Per això s'utilitza la funció de cost i la derivada de la funció d'activació de les neurones, explicades posteriorment. Els primers gradients calculats són els de la capa de sortida, a partir de l'error obtingut amb la diferència entre les prediccions $\hat{\boldsymbol{y}}$ i els valores objectius $\boldsymbol{y}$. Després es segueix propagant l'error per

cada capa de la xarxa per reduir el valor dels paràmetres entrenables, fins arribar a la capa d'entrada, fet que s'haurà complert una iteració de l'algorisme de propagació cap endarrere.

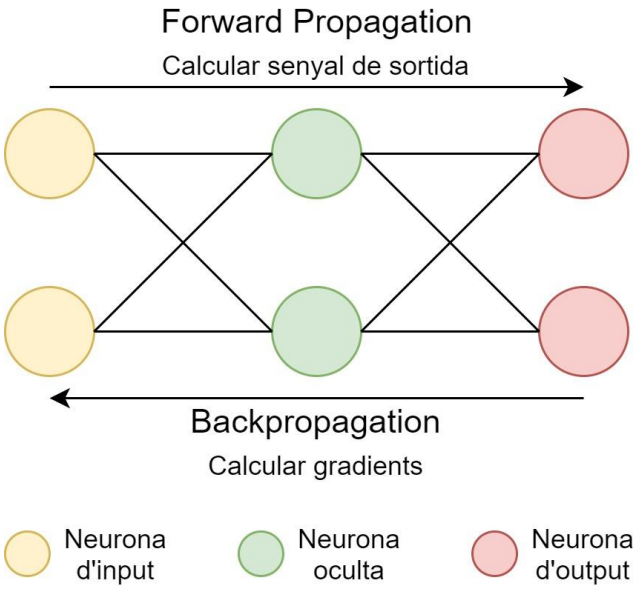


Figura 2.5 Arquitectura d'una xarxa neuronal densament connectada o Feedforward. Les fletxes indiquen la direcció del flux de la informació durant els mètodes de propagació cap endavant i propagació cap endarrere. Font: elaboració pròpia.

Gradient descent: gradient descent és una estratègia d'optimització de xarxes neuronals que té per objectiu minimitzar o maximitzar una funció per arribar a un mínim o màxim global. En xarxes neuronals, optimitzar significa ajustar els paràmetres del model per produir valors òptims que després seran usats per fer prediccions (Pai, Machine Learning Works, 2020). D'aquesta estratègia d'optimització se'n deriven diferents algorismes que es mostren més endavant (Goodfellow et al, 2016). Hi ha diferents paràmetres que s'utilitzen per optimitzar: els pesos, el bias i el gradient, explicats anteriorment i el learning rate.

El learning rate és un paràmetre modificable que es descriu com la quantitat de passos fets per aconseguir assolir el mínim (Pandey, 2019). Usant aquests paràmetres, la funció d'actualització de pesos és la següent:

$$w = w - \alpha * g \tag{2.1}$$

on la primera *w* és el pes resultant, la segona *w* és el pes actual, *α* (alfa) és el learning rate, *g* és el gradient (també escrit $\nabla_w f(w)$) i el símbol negatiu indica un

canvi de direcció en l'actualització dels pesos. Aquesta funció s'utilitza en cada iteració de l'algorisme amb totes les dades d'entrenament.

Funció d'activació: la funció d'activació és la funció que decideix si una neurona s'activa o no a partir de la senyal que rep d'entrada (Jordan, 2017). Per xarxes neuronals amb capes ocultes s'utilitzen funcions d'activació derivables i en la major part dels casos també funcions no lineals. La derivada de la funció d'activació s'utilitza durant el procés de propagació cap endarrere per calcular els gradients que permeten optimitzar els pesos. Existeixen vàries funcions d'activació que són derivables, les següents en són algunes:

- **ReLu**

  La funció ReLu (rectifier linear unit) és una de les funcions d'activació més utilitzades. Tot i el nom, és una funció d'activació no lineal que s'utilitza principalment en tasques de regressió. Encara que té un gran ús, és una funció que pateix d'un problema important que és transformar tots els valors d'entrada negatius en zero fent que no s'apliquin del tot bé als valors de sortida (Sharma, 2017).

  Funció:

  La funció ReLu es pot escriure de dos maneres diferents.

$$f(x)=\begin{cases} 0 \ \ per \ \ \&x<0 \\ x \ \ per \ \ \&x\geq 0 \end{cases} \tag{2.2}$$

$$f(x)=max(x,\ 0) \tag{2.3}$$

  Derivada:

$$f'(x)=\begin{cases} 0 \ \ per \ \ \&x<0 \\ 1 \ \ per \ \ \&x\geq 0 \end{cases} \tag{2.4}$$

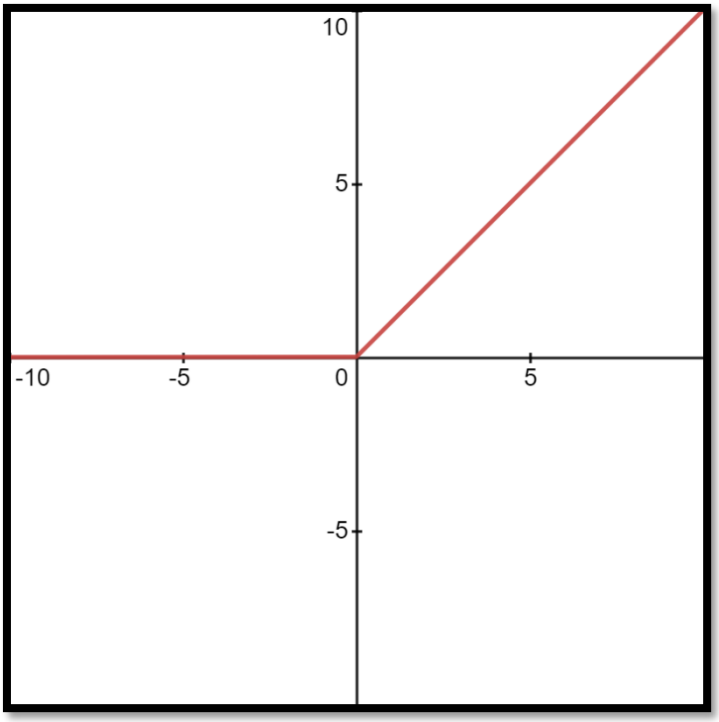


Figura 2.6 Representació gràfica de la funció ReLu. Font: elaboració pròpia.

- **Sigmoid**

  La funció sigmoid, també anomenada funció logística, és una funció d'activació no lineal que és molt utilitzada pels models de xarxes neuronals actuals. S'utilitza sobretot en tasques de classificació ja que el que es busca és establir probabilitats que funcionen amb valors entre 0 i 1, com els que extreu aquesta funció (Sharma, 2017).

  Funció:

$$f(x)= \frac{1}{1+e^{-x}} \tag{2.5}$$

  Derivada:

$$f'(x)=f(x)(1-f(x)) \tag{2.6}$$

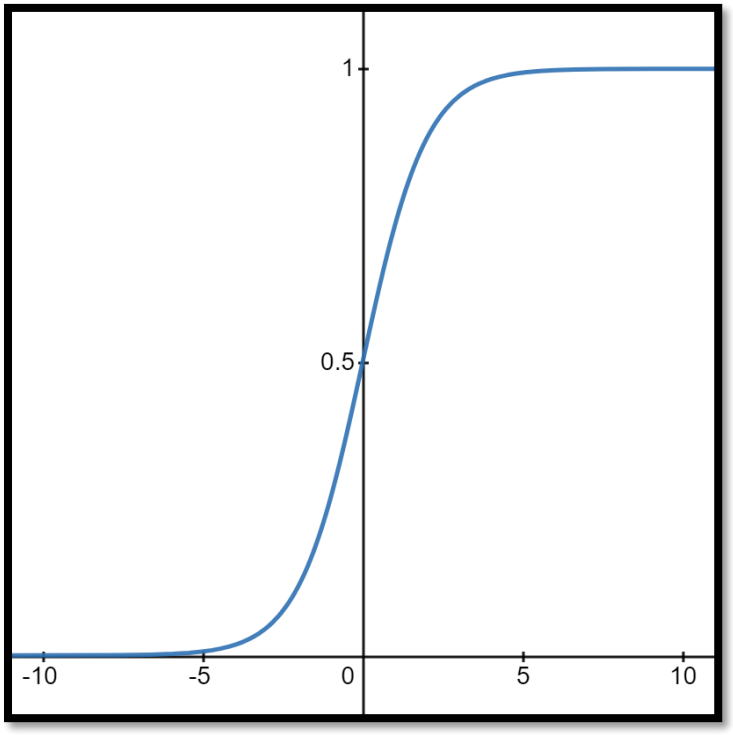


Figura 2.7 Representació gràfica de la funció Sigmoid. Font: elaboració pròpia.

- **Tangent hiperbòlica**

  La tangent hiperbòlica és molt semblant a la sigmoid però millor degut a que el rang de valors està entre -1 i 1 fent que el 0 quedi en el centre. Té l'avantatge que una senyal d'entrada negativa proporciona una senyal de sortida molt negativa mentre que una entrada de zero proporciona valors de sortida pròxims al zero. Com la funció sigmoid s'utilitza per tasques de classificació, però en aquest cas només per dos classes diferents (Sharma, 2017).

  Funció:

$$f(x) = \tanh(x) \tag{2.7}$$

  Derivada:

$$f'(x) = 1 - f(x)^2 \tag{2.8}$$

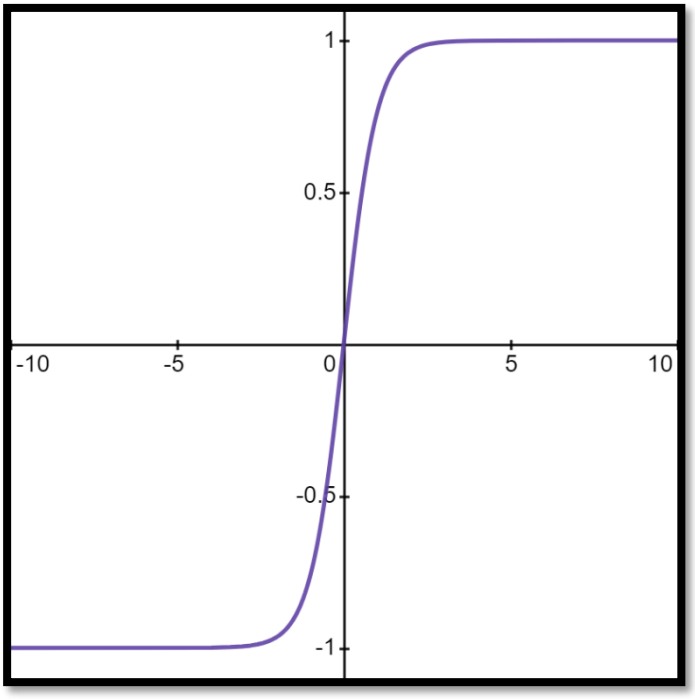


Figura 2.8 Representació gràfica de la funció Tangent hiperbòlica. Font: elaboració pròpia.

- **Leaky ReLu**

  La funció d'activació leaky ReLu és una versió millorada de la ReLu, destinada a minimitzar els problemes que té, explicats anteriorment. La funció leaky ReLu té un rang de valors de -infinit a infinit a diferència de la funció ReLu que té un rang entre 0 i infinit. Com es veu a la equació de la funció, el valor que acompanya x és alfa (α) que pot ser qualsevol valor. Generalment és de 0.01, però si no és el cas, la funció s'anomena Randomized ReLu (Sharma, 2017).

  Funció:

$$f(x)=\begin{cases} \alpha x \ \ per \ \ \&x<0 \\ x \ \ per \ \ \&x\geq 0 \end{cases} \tag{2.9}$$

  Derivada:

$$f'(x)=\begin{cases} 0.1x \ \ per \ \ \&x<0 \\ 1 \ \ per \ \ \&x\geq 0 \end{cases} \tag{2.10}$$

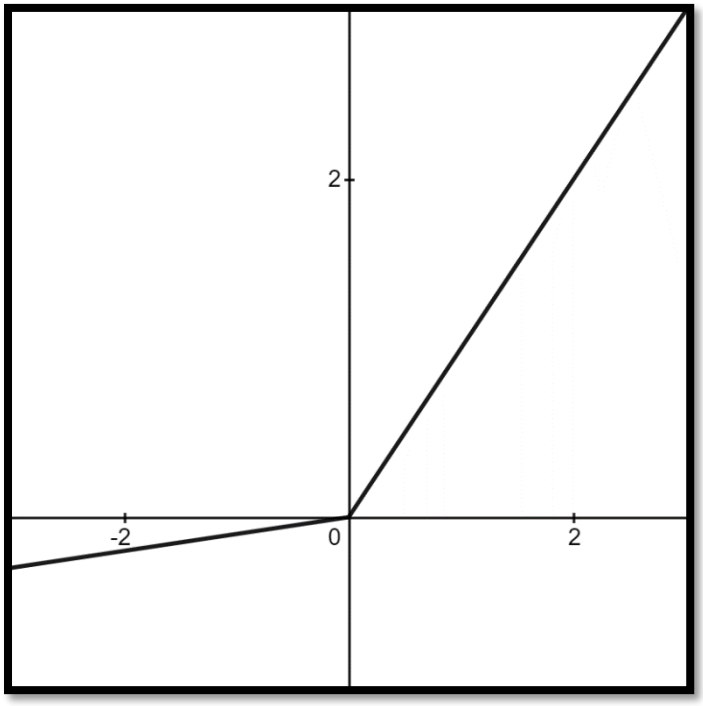


Figura 2.9 Representació gràfica de la funció Leaky ReLu. Font: elaboració pròpia.

- **ELU**

  La funció ELU (exponential linear unit), com la leaky ReLu, també és una versió millorada de la ReLu. La ELU té valors negatius que empenyen la mitjana de les activacions pròxima a 0 i, això permet un aprenentatge més ràpid i més precís ja que acosten el gradient al seu valor natural. Com la funció leaky ReLu, la ELU també utilitza el paràmetre alfa (α) que per defecte pot estar a 1 (Clevert et al, 2016).

  Funció:

$$f(x)=\begin{cases}\alpha(e^{x}-1) & per\ \ \&x<0\\ x & per\ \ \&x\geq 0\end{cases} \tag{2.11}$$

  Derivada:

$$f'(x)=\begin{cases}f(x)+\alpha & per\ \ \&x<0\\ 1 & per\ \ \&x\geq 0\end{cases} \tag{2.12}$$

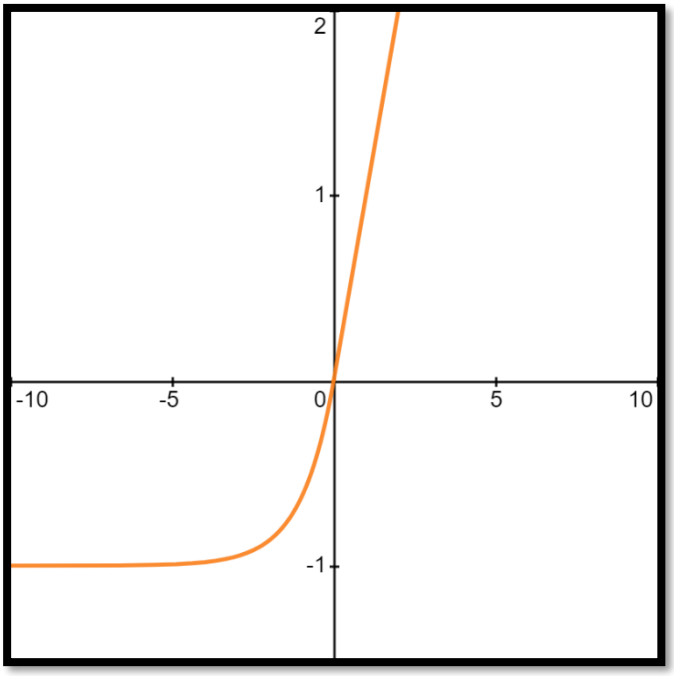


Figura 2.10 Representació gràfica de la funció ELU. Font: elaboració pròpia.

- **Lineal**

  La funció lineal o d'identitat és una funció que a diferència de les anteriors transforma els valors d'entrada linealment. Degut a això només es fa servir a la capa de sortida en tasques de regressió, ja que les tasques més complexes necessiten funcions d'activació no lineals.

  Funció:

$$f(x)=x \tag{2.13}$$

  Derivada:

$$f'(x)=1 \tag{2.14}$$

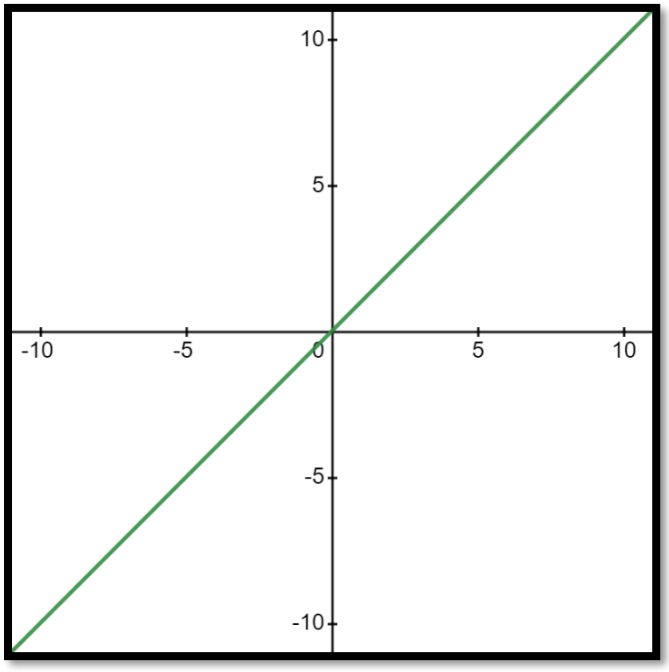


Figura 2.11 Representació gràfica de la funció Lineal o Identitat. Font: elaboració pròpia.

Funció del cost (Loss function): la funció del cost és una funció matemàtica que serveix per calcular l'error del model, és a dir, l'encert que té la xarxa neuronal al fer prediccions. L'error es calcula utilitzant el valor predit i el valor objectiu que ha d'aconseguir predir el model (Pai, Machine Learning Works, 2020), fent que com més petit sigui l'error, més bones prediccions fa el model. Existeixen diferents funcions per calcular l'error com. Per tasques de regressió hi ha l'error quadrat mitjà (mean squared error), l'error absolut mitjà (mean absolute error) o l'error absolut suau (smooth absolute error), entre d'altres. Per tasques de classificació s'utilitzen funcions com l'entropia creuada binària (binary cross entropy), l'entropia creuada categòrica (categorical cross entropy) o la probabilitat logarítmica negativa (negative log likelihood) (Agrawal, 2017).

De les anteriors, la més utilitzada en tasques de regressió és la funció de l'error quadrat mitjà (mean squared error en anglès, abreviat MSE). Aquesta funció intenta minimitzar l'error per cada parella de valors predicció-objectiu utilitzant la diferència entre els dos valors i fent el quadrat del resultat. Llavors, els valors resultants es sumen i es divideixen per m, que és el número de dades d'entrenament (Pai,

Machine Learning Works, 2020). La següent funció mostra més gràficament l'explicació anterior (Goodfellow et al, 2016).

$$\text{MSE}= \frac{1}{m}\sum_{i=1}^{m}(\hat{y}_i-y_i)^2 \tag{2.15}$$

on $\hat{y}_i$ és el valor de la predicció a *i* enèsima dada d'entrada, y_i és el valor objectiu de la predicció a *i* enèsima dada d'entrada i m és el número de dades d'entrenament.

Optimitzador: Un optimitzador (Agrawal, 2017) és un algorisme utilitzat per actualitzar els paràmetres del model per reduir l'error comès entre les prediccions i l'objectiu que es vol aconseguir. Si l'optimitzador està basat en gradient descent, hi ha dos tipus d'algorismes d'optimització: els que tenen un learning rate constant i els que tenen un learning rate adaptatiu.

En el primer cas podem trobar exemples com Stochastic Gradient Descent (SGD) i Momentum (Goodfellow et al, 2016). En aquests algorismes el paràmetre del learning rate l'ha de modificar el programador manualment basant-se en la seva experiència o provant la millor configuració. Si el valor del learning rate és massa petit, el model pot tardar molt a convergir, mentre que si és massa gran, el model pot fluctuar al voltant del mínim global o directament divergir.

En el segon cas hi ha algorismes com Adagrad (Duchi et al, 2011), Adadelta, RMSprop i Adam (Goodfellow et al, 2016). Aquests són una evolució dels anteriors ja que no és necessari que el programador modifiqui manualment el learning rate, sinó que el valor s'adapta a les necessitats de la tasca.

Dels algorismes adaptatius, el més recent i interessant per aquest projecte és l'algorisme Adam (Adaptative Moment Estimation) (Kingma i Ba, 2014). Aquest algorisme combina l'habilitat de l'algorisme Adagrad per fer front a gradients dispersos i l'habilitat de l'algorisme RMSprop per fer front als objectius no estacionaris. És un mètode eficient, robust, ràpid i senzill d'implementar. La següent figura mostra el pseudocodi per implementar l'algorisme Adam.

Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
$m_0 \leftarrow 0$ (Initialize 1st moment vector)
$v_0 \leftarrow 0$ (Initialize 2nd moment vector)
$t \leftarrow 0$ (Initialize timestep)
while θ_t not converged **do**
$t \leftarrow t + 1$
$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
$\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
$\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
end while
return θ_t (Resulting parameters)

Figura 2.12 Pseudocodi de l'algorisme d'optimització Adam. Font: (Kingma i Ba, 2014).

2.5 Aprenentatge per reforç

En aquesta secció s'expliquen tots els elements que conformen l'aprenentatge per reforç.

Markov Decision Process (MDP)

L'aprenentatge per reforç utilitza l'estructura anomenada Markov Decision Process (MDP) (Sutton i Barto, 2018) que és un model de presa de decisions seqüencial amb cinc components fonamentals.

- **Agent**
  Entitat que aprèn i que pren decisions.
- **Entorn**
  És amb el que interacciona l'agent comparable a un entorn real. La regla general és que qualsevol cosa que no pot ser canviada arbitràriament per l'agent es considera fora d'aquest i per tant part de l'entorn.
- **Estat**
  És la representació de l'estat actual de l'entorn on l'agent pot estar, comparable a fer una captura de l'entorn en un moment donat per poder ser analitzat de forma computacional. Això inclou per exemple la posició, la velocitat i la mida d'objectes tan mòbils com estàtics o la captura dels frames

de la pantalla per després ser analitzats, entre d'altres. Els estats en una tasca poden ser finits o infinits.

- **Accions**

  És un conjunt de possibles decisions que l'agent pot prendre en un estat.

- **Recompensa**

  És la recompensa positiva o negativa que l'entorn retorna a l'agent com a resultat de les seves accions i permet comunicar-li què ha d'aconseguir però no com. Una bona acció té una recompensa positiva mentre que una mala acció retorna una negativa i l'estat terminal de la simulació té la major recompensa. L'objectiu de l'agent serà sempre intentar maximitzar aquest valor durant la tasca escollint les millors accions a cada estat. Això significa maximitzar la recompensa acumulada a llarg termini i no la immediata. Posant com a exemple els escacs, se li donarà una recompensa positiva a l'agent només quan guanyi i una de negativa només quan perdi. Si es dona una recompensa per un sub-objectiu com és matar una peça rival l'agent pot aprendre a explotar aquests sub-objectius sacrificant l'objectiu principal que és sempre guanyar la partida.

El funcionament de MDP és bàsicament la relació entre l'agent i l'entorn i els tres senyals, tal com es veu a la figura 2.13: el senyal que representa les decisions de l'agent (accions), un senyal per representar la base sobre la qual es prenen decisions (estats) i un senyal per definir l'objectiu de l'agent (recompensa). Donat un State (S) l'agent escull una acció (A) entre les possibles, l'executa a l'entorn i aquest li retorna una recompensa (R) positiva o negativa i el següent State (S'). Finalment, el següent estat (S') passa a ser l'estat actual (S). Aquest procés porta a crear seqüències o trajectòries que comencen de la següent manera:

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3$$

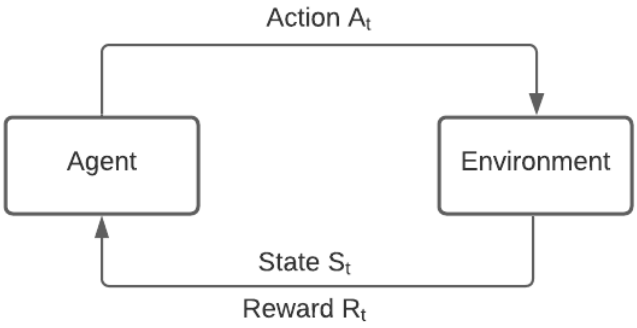


Figura 2.13 Representació estructural del procés de decisió de Markov. Font: Elaboració pròpia.

Iteracions, episodis i retorn

Tot el procés anterior, entre que l'agent escull una acció en un estat fins que el següent estat passa a ser l'estat actual, completa un pas o iteració de l'algorisme, anomenat en anglès timestep o step (Sutton i Barto, 2018) que en la durada de la tasca pot variar. Pot ser, per exemple, un procés que es realitza a cada frame. Finalitzat un pas, comença un altre cop el cicle des d'escollir l'acció a l'estat actual, que era el següent estat de la iteració anterior.

En el cas de les tasques finites, des de que l'agent comença a explorar l'entorn fins que passen un cert nombre de passos o aconsegueix l'objectiu final de la tasca arribant a l'estat terminal, aquest període s'anomena episodi (no significa que acabi la fase d'entrenament). Després d'acabar un episodi comença el següent restablint tots els valors inicials tan de l'entorn com de l'agent, excepte els valors on guarda l'experiència acumulada. Múltiples episodis conformen la tasca episòdica que ha de realitzar l'agent, que acaba amb un número exacte d'episodis o quan es considera que ha après suficient i ja no millorarà més. A partir d'aquí l'agent es considera entrenat.

En el cas de les tasques infinites o contínues no hi ha un estat terminal, sinó que l'agent està aprenent d'aquest entorn sempre. Un exemple pot ser un agent que aprèn el funcionament de la borsa.

El retorn (G) és l'acumulació de recompenses durant l'episodi i el que l'agent ha d'intentar maximitzar (Sutton i Barto, 2018). De manera ràpida es calcula sumant la recompensa de cada pas:

$$G = R_1 + R_2 + R_3 + \cdots + R_n \tag{2.16}$$

El retorn calculat d'aquesta manera només té sentit en tasques episòdiques on hi ha un estat terminal. Per tots els tipus de tasca s'utilitza un paràmetre anomenat *discount rate (γ)* o gamma que serveix per determinar el valor de les futures recompenses i que el retorn total no creixi infinitament. Aquest paràmetre està en el rang $0 \leq \gamma \leq 1$ i es manté constant durant tota la tasca. Si $\gamma = 0$ significa que l'agent només es preocupa de les recompenses immediates mentre que si γ és pròxim a 1, l'agent valora més les recompenses futures. Utilitzant el discount rate, l'anterior fórmula queda de la següent manera:

$$G = \gamma R_1 + \gamma^2 R_2 + \gamma^3 R_3 + \cdots + \gamma^n R_n \tag{2.17}$$

Aquest paràmetre també anima a l'agent a donar més importància a les recompenses futures en comparació amb la recompensa immediata, ja que aquestes es descomptaran més. Això significa que les recompenses immediates són més valuoses per l'agent però les recompenses que ofereixen el retorn total més elevat estan per davant de les immediates.

Estratègia i valor

El principal problema que busquen resoldre els algorismes d'aprenentatge per reforç és quines accions ha d'escollir l'agent per resoldre la tasca maximitzant el retorn. L'agent té diferents estratègies per escollir una acció o altra. Pot, per exemple, seleccionar una acció aleatòria entre les disponibles, escollir sempre l'acció que ofereix la màxima recompensa o seleccionar una acció no realitzada durant l'episodi amb la intenció d'explorar nous estats, entre d'altres. L'estratègia que utilitza l'agent per seleccionar una acció s'anomena, en anglès, policy(π) (Sutton i Barto, 2018) i porta a relacionar un estat donat amb l'acció a realitzar. Cada estat queda relacionat amb l'acció pertinent, seguint una estratègia, amb la notació següent:

$$\pi_t(A|S)$$

Solucionar una tasca de d'aprenentatge per reforç implica trobar una estratègia que aconsegueixi una gran recompensa a la llarga. Una estratègia π és millor que una

altra estratègia π' si el retorn esperat de π és més gran o igual que el retorn de π' per tots els estats. Sempre n'hi ha una que és millor o igual que totes les altres, aquesta és l'estratègia òptima π*, que pot ser més d'una.

Les policies poden ser deterministes, quan l'agent sempre escull la mateixa acció quan arriba a un estat en concret o estocàstiques quan l'agent escull una acció en un estat en concret basant-se en una distribució de probabilitats per totes les accions possibles.

El valor (Sutton i Barto, 2018) és el retorn esperat obtingut en un estat seguint una estratègia concreta al llarg de molts episodis, es a dir, permet saber com de bo és un estat o una acció en un estat concret. Com més episodis passen, més experiència acumula l'agent i per tant més òptims són els seus valors. Tant l'estratègia com el valor poden ser estructurats utilitzant una taula o matriu que pot ser de dos tipus:

- Valor Estat: el valor esperat en un estat concret executant accions basades en una estratègia π.
- Valor Estat-Acció: el valor esperat seleccionant una acció determinada d'un estat específic seguint una estratègia π. Aquest valor també s'anomena, en anglès, Q-Value o valor de qualitat (Q).

Equació de Bellman

L'equació de Bellman és una part fonamental dels algorismes d'aprenentatge per reforç. Permet relacionar el valor d'un estat S amb el valor de l'estat S' i serveix per al càlcul dels valors de qualitat més òptims per un estat concret. Els valors òptims s'aconsegueixen utilitzant l'equació de Bellman en múltiples iteracions de l'algorisme fins que el model convergeix en els valors més adequats. La notació és:

$$V(s) = \max_a \left(R(s,a)+\gamma V(s')\right)$$

(2.18)

2.6 Classificació dels algorismes d'aprenentatge per reforç

L'aprenentatge per reforç conté diversos algorismes per solucionar els diferents problemes que necessiten ser abordats amb aquest mètode de Machine Learning. Aquests algorismes es classifiquen principalment de la següent manera (Lapan, 2018):

- Si utilitzen un model de l'entorn, es a dir, coneixen el funcionament intern des d'un principi (model-based) o en canvi necessiten interactuar amb l'entorn per aconseguir coneixement (model-free).
- Si intenten aproximar directament l'estratègia de l'agent representada amb una distribució de probabilitats sobre totes les accions possibles (policy-based) o calculen el valor de qualitat de cada acció i escullen l'acció amb el valor més gran (value-based).
- Si intenten avaluar o millorar l'estratègia que pren les decisions (on-policy) o si intenten avaluar o millorar una estratègia diferent a la utilitzada per generar dades (off-policy).

Els algorismes amb més aplicacions al món real són els que no utilitzen un model de l'entorn (model-free) ja que sol ser massa complex per conèixer-lo sencer. Dins d'aquesta classificació hi ha algorismes com Monte Carlo Control, Sarsa, Q-Learning o el grup d'algorismes Policy Gradient entre d'altres. Aquest projecte explorarà l'algorisme Q-Learning basat en una taula i sobretot la seva versió més potent, el Deep Q-Learning que no està basat en una taula.

2.7 Aprenentatge amb valors de qualitat

2.7.1 Temporal-Difference Learning

Un dels conceptes que ha revolucionat l'aprenentatge per reforç és el de Temporal-Difference (Sutton i Barto, 2018). Aquest mètode aprèn directament interactuant amb l'entorn amb prova i error sense tenir-ne un model. Es basa en actualitzar estimacions basades en part en altres estimacions apreses sense esperar a un resultat final. El funcionament és que en cada iteració calcula la recompensa futura utilitzant l'equació de Bellman adaptada sense haver de tenir en compte el retorn

total de l'episodi. Això permet avaluar més acuradament una trajectòria d'estats i accions ja que s'avalua individualment cada acció. Utilitzant el retorn total de l'episodi pot portar a descartar trajectòries amb accions positives o a acceptar-ne amb accions massa negatives.

2.7.2 Algorisme Q-Learning

Segons la classificació de l'apartat 2.6, el Q-Learning és un algorisme que no utilitza un model, està basat en l'ús de valors de qualitat i no intenta millorar una estratègia. És l'algorisme més interessant dels representats en una taula. Utilitza una taula anomenada Q-table, o taula de valors de qualitat, on les files són els estats i les columnes les accions per cada estat, fet que permet relacionar cada estat i acció amb un valor de qualitat o Q-value.

El funcionament de l'algorisme és (Sutton i Barto, 2018):

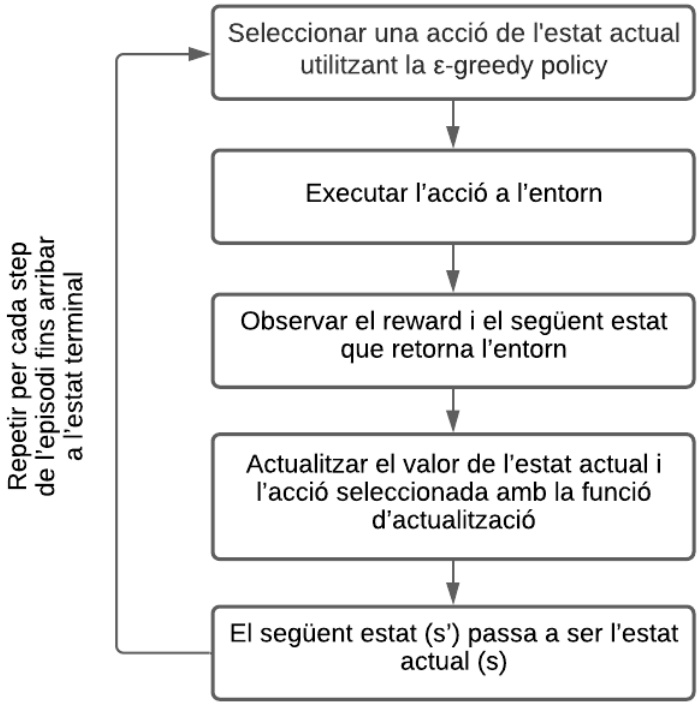


Figura 2.14 Algorisme de Q-Learning. Font: Elaboració pròpia.

La funció d'actualització dels Q-values basada en l'equació de Bellman és:

$$Q(S,A) = Q(S,A) + \alpha[R + \gamma \max Q(S',a) - Q(S,A)]$$

(2.19)

on Q(S,A) és el valor actual de la taula per l'estat i acció seleccionats, α és el learning rate, R és la recompensa aconseguida al realitzar l'acció a l'estat seleccionat, γ és el discount rate o gamma que normalment té un valor de 0.9 o 0.99 i maxQ(S', a) és la màxima recompensa esperada en el futur donat l'estat següent (S') i totes les possibles accions. Aquesta equació basada en l'equació original de Bellman ens permet relacionar el valor d'un estat (S) i una acció (A) amb el valor d'un estat (S'). L'objectiu serà trobar Q* que és la funció acció-valor òptima independentment de l'estratègia que es segueix.

En el Q-Learning, per seleccionar l'acció és comú utilitzar l'estratègia ε-greedy on ε és la probabilitat de seleccionar una acció aleatòria en aquest tipus d'estratègia. El seu valor està entre 0, on es maximitza la selecció d'accions ja conegudes amb el que es coneix com a explotació de l'entorn o estratègia greedy, i 1, on es maximitza la selecció d'accions de forma aleatòria amb el que es coneix com a exploració de l'entorn. El valor mínim sol estar a 0.1 fet que assegura que hi haurà un mínim d'exploració durant tota la tasca, però el Q-Learning, al ser tant senzill, el valor final d'èpsilon pot ser 0. El funcionament és:

- S'inicialitza la variable ε a 1 per assegurar que es comença la tasca escollint accions aleatòries per explorar l'entorn.
- S'obté un valor aleatori entre $0 \geq x \leq 1$.
- El valor de x es compara amb ε
    - Si $x < \varepsilon$ l'agent escull una acció aleatòria a l'estat actual.
    - Si $x \geq \varepsilon$ l'agent escull una l'acció amb el Q-value més gran dins l'estat actual.
- Es redueix gradualment la variable ε segons la necessitat al llarg dels episodis.

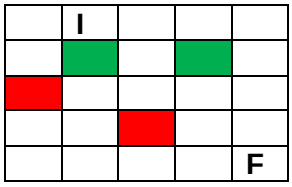
Equilibri entre exploració i explotació

Utilitzant l'estratègia anterior, l'agent pot passar d'una fase d'exploració, a una fase d'explotació de l'entorn. La variable èpsilon és la que regula aquesta transició de forma lineal però necessita ser reduïda gradualment de la forma més adequada per la tasca que ha de resoldre l'agent. Si èpsilon es redueix massa ràpid, l'agent no té temps d'explorar prou estats i no arribarà a aprendre a resoldre la tasca. Per contra

si èpsilon es redueix massa lentament, l'aprenentatge deixa de ser òptim, cosa que en tasques senzilles no és problema però en tasques molt complexes pot augmentar molt el temps d'entrenament fins a l'ordre de desenes d'hores. Per aquest motiu és important reduir èpsilon en la mesura correcta perquè l'agent acabi entrenat per resoldre la tasca. Sinó també es poden utilitzar altres estratègies de forma temporal que assegurin en un principi l'exploració i, al final, l'explotació de l'entorn.

Exemple

Per exemplificar aquest algorisme partim d'un entorn basat en una quadricula de 5x5 cel·les i d'un agent que ha de complir una tasca. A la quadricula hi ha una cel·la d'inici per l'agent i una de final on ha d'arribar. Per complicar l'entorn hi ha cel·les que proporcionen una recompensa positiva de 10 i algunes amb una recompensa negativa de -10 mentre que la resta proporcionen una recompensa de 0. Finalment l'agent té les possibles accions de moure's cap amunt, cap avall, a l'esquerra i a la dreta i que utilitzarà per complir la tasca d'anar de l'inici a la cel·la final.



Taula 2 Entorn quadricular com a exemple de Q-Learning. Font: Elaboració pròpia.

Utilitzant l'exemple, l'algorisme procedeix de la següent manera:

- S'inicialitza la taula de valors a 0. La taula té tantes files com estats possibles (25 estats) i tantes columnes com accions possibles (4 accions).

	←	↑	→	↓
Cel·la 0,0	0	0	0	0
Cel·la 0,1	0	0	0	0
Cel·la 0,2	0	0	0	0
…	0	0	0	0

Taula 3 Q-table amb els valors inicials. Hi ha un valor de qualitat per cada parella estat-acció possible. Font: Elaboració pròpia.

- S'escull una de les 4 accions possibles utilitzant l'estratègia ε-greedy. Per exemple cap avall una cel·la.

- S'executa l'acció a l'entorn. L'agent es mou de l'inici una cel·la cap avall.
- S'observa la recompensa i el següent estat que retorna l'entorn. La cel·la és verda (cel·la 0,1), per tant la recompensa és positiva. A més, també rep una recompensa segons la distància a la que es troba de l'objectiu, per agilitzar la convergència del model (en aquest cas recompensa negativa de -7). Aquesta cel·la verda és també el següent estat.
- S'utilitza l'equació de Bellman per actualitzar la taula:
  $Q(\text{cel·la 0,1, cap avall}) = Q(\text{cel·la 0,1, cap avall}) + \alpha[R + (\gamma * \max Q(\text{cel·la 1,1 , per totes les accions})) - Q(\text{cel·la 0,1, cap avall})]$
  $0 = 0 + 0.7[3 + (0.99 * 0) - 0]$

(2.20)

	←	↑	→	↓
Cel·la 0,0	0	0	0	0
Cel·la 0,1	0	0	0	2.1
Cel·la 0,2	0	0	0	0
…	0	0	0	0

Taula 4 Q-table a la primera iteració de l'algorisme. Font: Elaboració pròpia.

- Finalment el següent estat passa a ser l'estat actual.
- Després d'això s'ha de tornar a repetir des de seleccionar les accions necessàries fins que l'agent resolgui la tasca. La tasca s'ha de repetir fins que els valors de la taula hagin convergit i l'agent es consideri entrenat, que pot ser al llarg d'una certa quantitat d'episodis o infinitament.

L'algorisme Q-Learning funciona molt bé quan l'espai d'estats és discret i no gaire abundant però té un problema d'escalabilitat en espais d'estats grans o infinits com en tasques contínues o entorns massa grans. Per solucionar aquest problema existeix l'enfocament basat en *deep learning* sobre aquest algorisme.

2.7.3 Deep Q-Learning

El *deep Q-Learning* parteix de la base del Q-Learning però amb un enfocament diferent. En aquest algorisme ja no s'utilitza una taula on guardar els valors sinó que s'utilitza una funció d'aproximació per aconseguir-los. Com a funció s'utilitzen xarxes neuronals que aquí s'anomenen *Deep Q-Networks* (DQN). A la DQN se li passa com a input un estat (S) i extreu com a output un valor de qualitat per cada

acció. L'objectiu és variar els pesos de la xarxa neuronal per aconseguir extreure els valors més òptims.

L'arquitectura d'una DQN és (Mnih et al, 2013):

- Una xarxa neuronal anomenada *Q Network* que s'encarrega de calcular i optimitzar els valors per entrenar l'agent.
- Una xarxa neuronal idèntica a la *Q Network* anomenada *Target Network* que s'encarrega de calcular els valors per utilitzar com a etiqueta de la primera xarxa neuronal. Els pesos de la *Target network* es mantenen fixes durant unes quantes iteracions per donar la possibilitat a la Q Network a optimitzar els seus pesos aproximant-les a les de la *Target Network* oferint un objectiu semi-estàtic.
- *Experience replay* és una tècnica per guardar les experiències de l'agent a cada pas en forma de tupla $e_t = (s_t, a_t, r_t, s_{t+1}, d)$ en un conjunt de dades anomenat *replay memory* M. Cada experiència està formada per l'estat actual, l'acció seleccionada, la recompensa i el següent estat obtinguts i un valor booleà per saber si realitzant aquesta acció s'ha arribat a un estat terminal de l'episodi.

El funcionament de l'algorisme és:

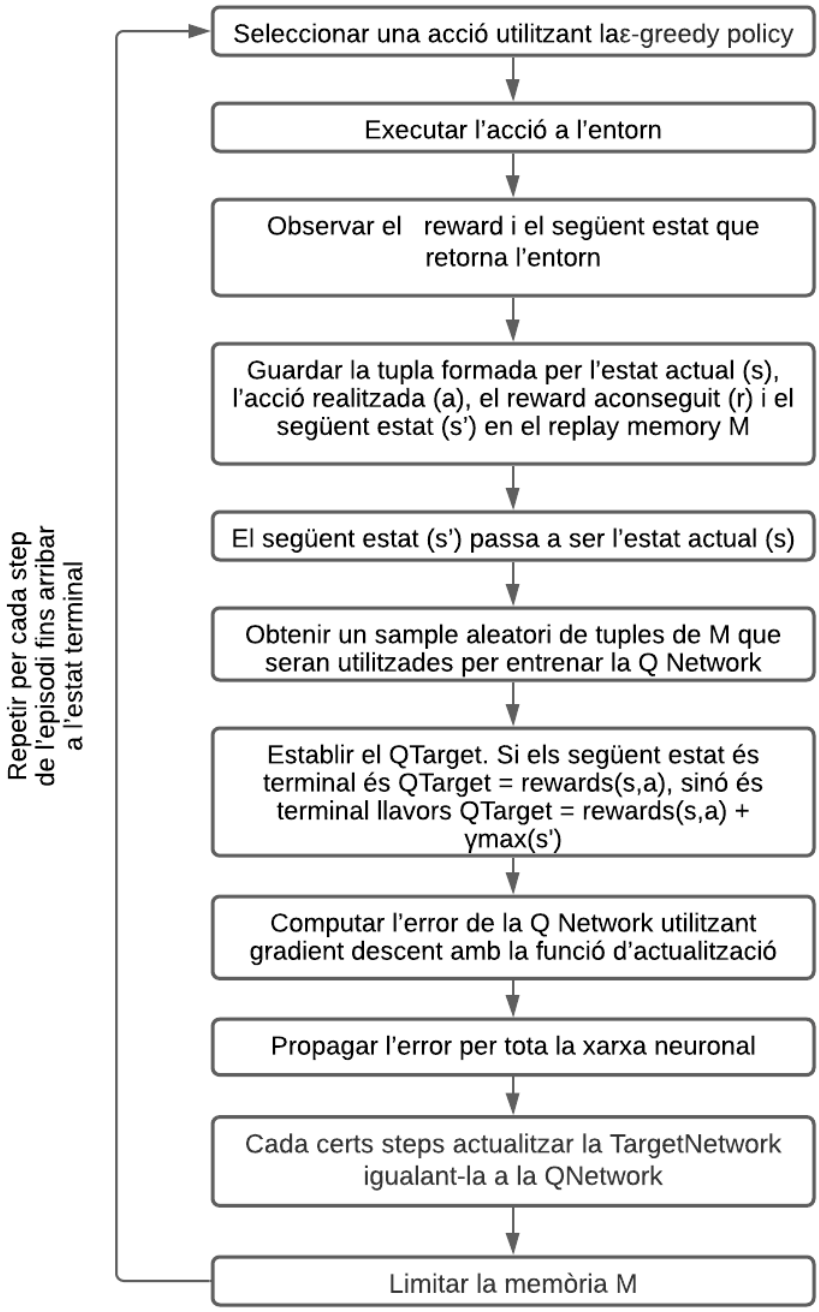


Figura 2.15 Algorisme de Deep Q-Networks amb ús d'una memòria d'experiències. Font: Elaboració pròpia.

L'equació d'actualització dels pesos és una modificació de la utilitzada pel Q-learning per l'algorisme gradient descent estàndard:

$$\nabla w = \alpha \left[\left(R + \gamma \max \hat{Q}(s', a, w) \right) - \hat{Q}(s, a, w) \right] \nabla_w \hat{Q}(s,a,w)$$

(2.21)

on ∇w és el canvi als pesos, $R + \gamma \max \hat{Q}(s', a, w)$ és el Q-target o label per calcular l'error, $\hat{Q}(s, a, w)$ és el valor actual predit per la *Q Network* i $\nabla_w \hat{Q}(s, a, w)$ és el gradient del Q-value actual. Aquesta funció se sol fragmentar en diferents parts en el codi.

2.8 Eines i llibreries

2.8.1 Visual Studio

Visual Studio (Microsoft, 2019) és un entorn integrat de desenvolupament (IDE) creat per Microsoft. Admet la programació en C#, C++, Visual Basic i d'altres sempre amb la orientació de crear software compatible amb la plataforma .NET. L'última versió d'aquest IDE és la 2019 que conté millores respecte les versions anteriors com l'intellicode, una forma d'auto completat que sempre ofereix la millor resposta al programador mentre està escrivint el codi. (Microsoft, 2020).

2.8.2 C Sharp

C# (Microsoft, 2021) és un llenguatge de programació modern, orientat a objectes i de tipus estàtic. Va ser desenvolupat l'any 2000 per Microsoft i actualment forma part de la plataforma .NET. Aquest llenguatge es caracteritza respecte altres de semblants per la seva senzillesa i pel gran potencial per crear aplicacions multi-plataforma ràpidament.

2.8.3 Python

Python (Python Software Foundation, 2021) és un llenguatge de programació d'alt nivell, interpretat i orientat a objectes que serveix com a llenguatge de propòsit general. Va ser creat al febrer de l'any 1991 per Guido van Rossum però actualment està gestionat per la Python Software Foundation (PSF). Python és un llenguatge utilitzat per la seva sintaxi senzilla, pel gran potencial al crear aplicacions complexes, per l'adaptabilitat a diferents plataformes i per la gran quantitat de llibreries associades que faciliten la programació de tot tipus de tasques. Durant el primer quart de 2021 ha sigut el segon llenguatge més utilitzat a Github (Beuke, 2021). En aquest projecte s'utilitza la versió 3.8.

2.8.4 Tensorflow i Keras

Tensoflow (Tensorflow, 2021) és una llibreria d'alt nivell creada per Google i implementada per ser utilitzada amb Python, que està orientada al desenvolupament de *machine learning*.

Keras (Keras, 2021) també és una llibreria de *machine learning* d'alt nivell però que és utilitzada com a capa superior de Tensorflow amb la intenció d'incrementar la velocitat d'implementació dels models de *deep learning*.

2.8.5 Numpy

Numpy (Numpy, 2021) és una llibreria que permet la computació numèrica al llenguatge Python. Es va crear el 2005 per substituir dos llibreries més senzilles i actualment és quasi imprescindible en qualsevol projecte. Algunes de les seves característiques importants són: la creació de vectors i matrius, realitzar fàcilment operacions amb vectors, utilitzar operacions estadístiques i la connexió amb les llibreries de Tensorflow i Keras, entre d'altres característiques interessants.

2.8.6 Matplotlib

Matplotlib (Hunter, 2007) és una llibreria per crear gràfics 2D a Python d'una manera senzilla i ràpida. Entre els gràfics que es poden crear hi ha: gràfics de línia, histogrames, gràfics de dispersió i gràfics polars, entre molts altres.

2.8.7 Imageio

Imageio (Imageio, 2020) és una llibreria de Python que permet la gestió de dades en format d'imatge. Pot llegir i generar dades d'imatge entre les que hi ha imatges animades en format de gif, realitzar captures de la web cam o canviar el color d'una pel·lícula.

2.9 Metodologia de desenvolupament

En aquest apartat es detalla la metodologia més adequada per la implementació dels prototips i el projecte final.

Model de desenvolupament incremental

El model de desenvolupament incremental (Laboratorio Nacional de Calidad del Software de INTECO, 2009) és una metodologia de gestió de projectes que es basa

en augmentar progressivament la funcionalitat del software. El model utilitza seqüències lineals on es realitzen diverses fases que produeixen un increment. Cada increment ha de ser significativament superior a l'anterior per notar que hi ha una evolució en el desenvolupament. Es diferencia del model de desenvolupament en cascada per la gestió interna de l'equip i pel fet que en aquest últim es realitza tot el desenvolupament sencer d'una vegada, validant només al final, cosa que augmenta molt els riscos del projecte.

Aquesta metodologia permet (Laboratorio Nacional de Calidad del Software de INTECO, 2009):

- Crear software funcional més ràpidament i en un estat primerenc del desenvolupament del producte final.
- Tenir més flexibilitat a l'hora d'adaptar el projecte a les condicions inicials.
- Reaccionar i gestionar més fàcilment als problemes que sorgeixen encara que el projecte estigui en una etapa avançada.
- Provar i depurar el codi resulta més senzill progressant amb passos petits.
- Cada iteració sembla fàcilment abordable. Aquesta metodologia és considerada una de les millors per al desenvolupament de software perquè ofereix la possibilitat d'anar revisant la feina feta i el feedback del producte a mesura que es realitza el projecte per així poden tenir més marge de reacció davant dels errors.

El funcionament del model de desenvolupament incremental és iterar principalment entre quatre fases (Pérez, 2016):

- **Anàlisis**: s'analitzen els requeriments per aquesta iteració per saber que són compatibles amb la iteració anterior.
- **Disseny i especificació de l'increment**: establir les tasques per assegurar que hi ha un increment de millora respecte l'anterior iteració.
- **Desenvolupament**: implementació de les tasques previstes per aconseguir l'increment establert a la fase anterior.
- **Validació**: el fragment desenvolupat a la iteració ha de ser provat per confirmar que funciona. En cas de resultats inesperats s'ha de tornar enrere a la mateixa iteració i buscar-ne les causes.

Aquestes fases es repeteixen per cada iteració augmentant les funcionalitats del software cada vegada. Com a part extra del model, abans de la fase de disseny, s'han de definir els requeriments i definir quines seran les iteracions necessàries. Al finalitzar, les iteracions realitzades han de quedar compactades al projecte (Pérez, 2016).

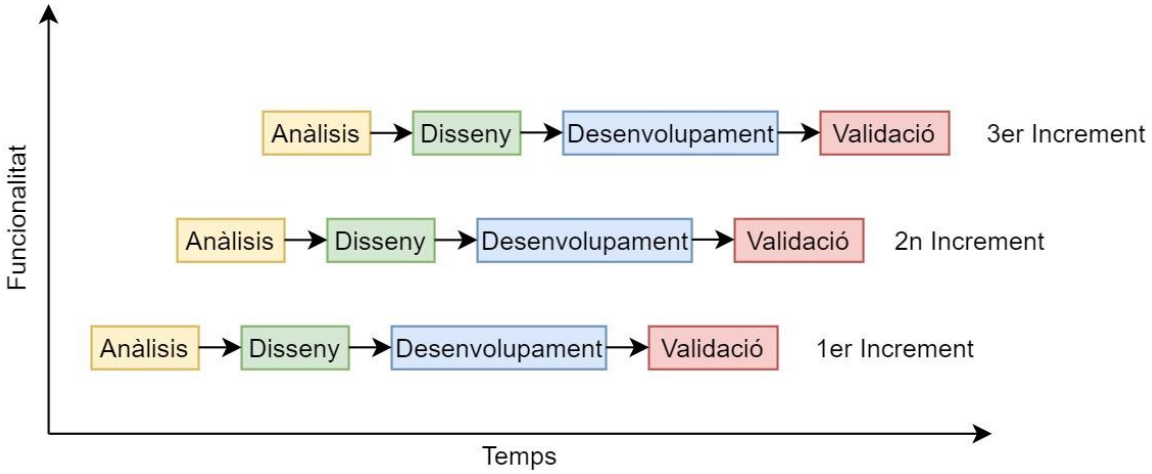


Figura 2.16 Esquema del desenvolupament incremental. Font: Elaboració pròpia.

2.10 Resum

En aquest apartat s'ha exposat la teoria relacionada amb la intel·ligència artificial per videojocs més enfocada als algorismes capaços d'aprendre, englobats dins el Machine Learning. S'ha posat el focus en l'aprenentatge per reforç, detallant les parts que permeten el funcionament d'aquest tipus de Machine Learning i dos dels algorismes existents, el Q-Learning i el Deep Q-Learning. Finalment també s'ha explicat la metodologia, les llibreries i les eines utilitzades per desenvolupar el treball.

3 Anàlisi de referents

En aquest apartat s'exposen els referents utilitzats i què és el que en destaca per ser útils en aquest projecte.

3.1 Openai: Hide and Seek

El primer referent utilitzat per aquest treball és la proposta basada en aprenentatge per reforç realitzada per l'empresa Openai anomenada "hide and seek" (Openai, 2020) basada en el joc tradicional infantil del gat i la rata (Fundació Enciclopèdia Catalana, 2021). Aquesta empresa ha creat una simulació on quatre agents autònoms dividits en dos equips, que comparteixen un mateix entorn, han d'aconseguir un objectiu diferent segons la tasca que se'ls ha assignat. Per un costat hi ha els agents perseguidors (seekers) que han d'atrapar als agents que s'oculten i per l'altra hi ha els agents que s'oculten (hiders) que han d'impedir que els agents perseguidors els atrapin. Al principi, l'entorn conté una gran sala tancada on hi ha els seekers (vermell) i una sala més petita, creada de forma aleatòria amb algunes forats, dins la sala gran i on hi ha els hiders (blau). A l'entorn també hi ha blocs (quadrat groc) i una rampa (triangle groc) que els agents poden agafar, moure i bloquejar perquè els agents enemics no ho agafin. L'algorisme que implementen és el Proximal Policy Optimization (Schulman et al, 2017), un tipus d'algorisme de deep reinforcement learning policy-based que no és el que s'utilitza en aquest treball. Tot i així, és un referent interessant per com estructuren l'entrenament de la tasca i per les dades d'entrada que se'ls hi passen als agents, que són bàsicament numèriques, com en aquest projecte.

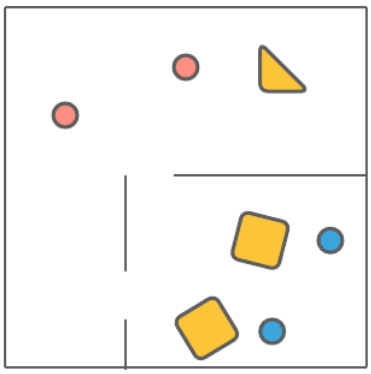


Figura 3.1 Punt inicial de l'entorn amb vista top down de la simulació Hide and Seek d'Openai. Font: Elaboració pròpia.

En aquesta simulació els agents no obtenen recompensa per interaccionar amb l'entorn, sinó que han d'aprendre a fer servir aquest entorn en benefici propi sense cap motivació extra, amb la idea d'aconseguir els objectius esmentats abans per cada grup d'agents. La recompensa que s'estableix és principalment d'equip; si els hiders aconsegueixen estar tots amagats obtenen una recompensa de 1 i una recompensa de -1 si algun hider no està amagat. En el cas dels seekers, obtenen una recompensa de -1 si tots els hiders estan amagats i d'1 en qualsevol altra cas. Tots els agents també obtenen individualment una recompensa negativa de -10 si surten fora dels marges de la zona de joc. Al principi hi ha un temps anomenat fase de preparació, on els hiders poden moure's i intentar amagar-se amb una recompensa de 0, després s'activen els seekers. Els episodis tenen una durada fixa de 240 passos. Les observacions que conformen l'estat són: la posició, orientació i el vector velocitat de l'agent, informació dels punts d'impacte d'un lidar que detecta els obstacles per l'agent, la posició, orientació i vector velocitat dels altres agents de l'entorn, la posició, orientació, vector velocitat i mida dels blocs de l'entorn i la posició, orientació i vector velocitat de les rampes. Aquestes dades, després de ser introduïdes a diferents xarxes neuronals, es concatenen i se'ls hi aplica una màscara per impedir que l'agent sàpiga més informació de la que pot saber, com en el cas de tenir un agent darrere seu. Es disminueix la mida del vector resultant, que s'introdueix a una xarxa neuronal de curta memòria que acaba extraient una probabilitat per cada una de les tres accions possibles: moure's, agafar i bloquejar un objecte. La següent figura mostra com aquesta informació anterior és estructurada i analitzada pel model de presa de decisions de cada agent.

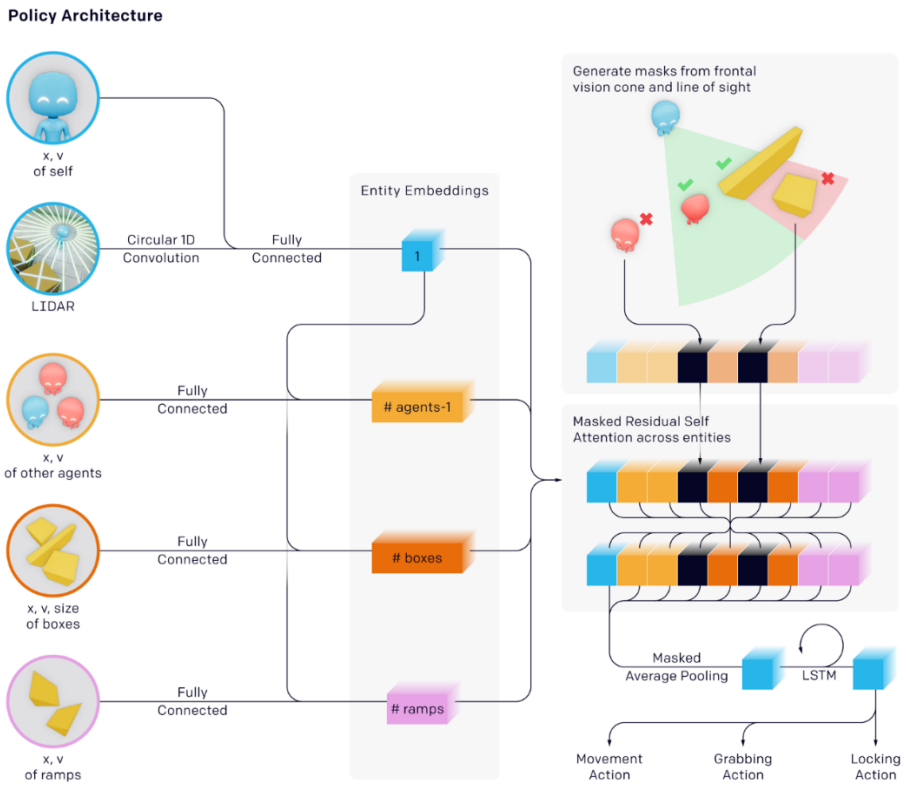


Figura 3.2 Imatge que mostra l'arquitectura del model que pren les decisions per cada agent a la simulació “Hide and seek” d’Openai. Font: **(Openai, 2020)**.

Els resultats d’aquesta simulació mostren uns quants comportament emergents (comportament no previstos) després d’un total de 167.7 milions d’episodis que corresponen a 98 hores d’entrenament en el pitjor dels casos. Els comportaments emergents que s’han trobat és la capacitat de moure’s damunt d’un bloc, els hiders marxant de la zona de joc amb un bloc, els hiders utilitzant errors en les físiques per enviar la rampa fora de l’abast dels seekers i finalment els seekers utilitzant també els errors en les físiques de la rampa per saltar molt alt i arribar cap als hiders que estan amagats. Els investigadors han trobat diferències significatives durant l’entrenament, segons la quantitat de dades d’entrada que utilitzaven (batch size). En aquest cas, com més dades utilitzades més ràpid es realitza l’entrenament, arribant fins a les 20 hores.

3.2 DeepMind: Deep Q Network amb Experience replay

El segon referent, i potser el més important, és el projecte relacionat amb deep reinforcement learning desenvolupat per DeepMind. En aquest cas la tasca està

basada en el joc d'atari "Space Invaders" (Mnih et al, 2013). El document on es descriu aquesta tasca està considerat un referent dins del deep reinforcement learning ja que va aportar un nou algorisme que va motivar la investigació massiva d'aquest tipus de Machine Learning. La seva contribució és l'ús d'una memòria on guardar les experiències i la gestió que se'n fa, com es detalla a l'apartat de deep Q-Learning del marc teòric. La memòria és en essència una estructura de dades en format de cua, en un primer moment buida ja que a l'aprenentatge per reforç no existeix un conjunt de dades d'entrenament inicial i, que és limita a la capacitat requerida segons el tipus de tasca. Cada valor d'aquesta cua és una tupla d'experiències formada per l'estat actual, l'acció escollida, la recompensa i el següent estat retornats per l'entorn i un valor booleà que indica si el següent estat és terminal. Quan s'ha realitzat un pas de l'algorisme, es recullen les cinc dades anteriors i s'afegeixen dins la cua, formant dinàmicament, durant tota la tasca, el conjunt de dades d'entrenament de la Q Network. Tal com indiquen els investigadors de DeepMind, els avantatges que té utilitzar aquesta memòria d'experiències són:

- Millora l'eficiència de l'algorisme, ja que cada tupla d'experiències es pot utilitzar potencialment en moltes actualitzacions dels pesos de la xarxa neuronal.
- Trenca amb les correlacions entre les mostres ocasionades per llegir-ne de consecutives. L'aproximació que utilitzen és escollir de forma aleatòria les experiències que formen part de la mostra d'entrenament de la Q Network per evitar les correlacions de dades i reduir la variància de les actualitzacions.
- Amb l'ús de la memòria d'experiències es fa una mitja dels comportaments sobre molts estats anteriors per suavitzar l'aprenentatge i evitar oscil·lacions en els paràmetres.

Els responsables d'aquesta investigació utilitzen frames de la pantalla com a estat, en comptes de valors numèrics com la posició o la distància entre objectes. Això fa que no puguin utilitzar una xarxa neuronal bàsica sinó que han d'utilitzar una xarxa neuronal convolucional, típicament utilitzada per analitzar imatge. L'arquitectura del model utilitzat consisteix en unes dades d'entrada emmagatzemades en un tensor

de 84 x 84 x 4 corresponents a la mida dels frames després de ser processats i, el número 4 són els quatre frames que formen l'estat. Després, el model conté dos capes de convolució que extreuen les característiques rellevants de les imatges i filtres per reduir la mida de les matrius de dades. Finalment hi ha acoblada una xarxa neuronal densament connectada amb una capa oculta formada per 256 neurones i una capa de sortida amb entre 4 i 18 neurones segons les accions que l'agent pot fer a cada joc. Aquest tipus d'arquitectura utilitzada amb l'algorisme de la figura 3.3 és el que defineixen com a Deep Q-Networks (DQN).

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

Figura 3.3 Algorisme en pseudocodi ideat per els investigadors de DeepMind per ser utilitzat en deep reinforcement learning. Font: **(Mnih et al, 2013)**.

Els resultats que ofereixen mostren que, en alguns jocs, els agents entrenats amb Deep Q-Networks aconsegueixen un nivell de joc sobre humà. En la resta obtenen un nivell similar o inferior al d'una persona. Tot i així, demostren que la metodologia de les Deep Q-Networks funciona.

3.3 Keras: Implementació Deep Q-Learning

El tercer referent és el projecte creat per Jacob Chapman i Mathias Lechner (Chapman i Lechner, 2020) per exemplificar el funcionament de la llibreria Keras. El projecte té com a objectiu implementar un model de deep Q-Learning utilitzant les llibreries de Python, Tensorflow i Keras, basat en el paper de DeepMind (Mnih et al, 2013). La tasca que realitzen és l'entrenament d'un agent que aprèn a jugar

al joc Atari Breakout. Aquest referent és útil per visualitzar el funcionament d'aquestes llibreries i l'estructuració de codi per un algorisme de deep Q-Learning.

3.4 Resum

En aquest apartat s'han presentat els tres referents útils pel projecte, dos dels quals són importants a nivell teòric i l'últim és important a nivell pràctic. A més a més, s'ha explicat perquè són importants per desenvolupar aquest treball.

4 Objectius

El principal objectiu d'aquest projecte és:

- Dissenyar, prototipar i analitzar una tasca relacionada amb un joc que utilitzi intel·ligència artificial basada en algorismes de machine learning.

  La tasca està basada en el joc Snake on un agent és entrenat perquè aconsegueixi jugar amb una mínima habilitat per aconseguir punts, augmentar de mida, i mantenir-se sense morir el màxim temps possible.

Com a objectius secundaris:

- Estudiar les formes d'analitzar l'aprenentatge de l'agent per extreure conclusions sobre el rendiment.
- Avaluar la conveniència d'aquest tipus d'algorismes en els videojocs i contrastar-los amb altres alternatives i amb els referents proposats al projecte.

5 Disseny metodològic i cronograma

L'estratègia utilitzada per desenvolupar el projecte és el model de desenvolupament incremental amb una iteració inicial més senzilla i les dos següents més complexes.

5.1 Fases del projecte

5.1.1 Investigació

Per començar el projecte es realitza una investigació sobre les bases de l'aprenentatge per reforç. Tot i així, per complir els objectius del projecte es necessita models més potents que els que ofereixen les bases del RL. El següent pas és aprofundir en la cerca d'informació sobre Deep Reinforcement Learning. En aquesta branca del deep learning hi ha mètodes que, amb unes condicions molt específiques i controlades, permeten obtenir els resultats que s'esperen.

5.1.2 Documentació

Durant tota l'evolució del projecte es prepara la documentació de la memòria. S'hi poden trobar apartats teòrics, els objectius proposats, l'explicació de la part pràctica, els referents i les conclusions a les que s'ha arribat, entre d'altres coses.

5.1.3 Cerca de referents

Donada la dificultat del projecte s'han tingut en compte referents tan teòrics com pràctics. Els referents teòrics serveixen per conèixer quins són els algorismes adequats i perquè funcionen de la manera com ho fan. Els referents pràctics estan més enfocats a visualitzar formes d'estructuració de codi, programació de models de deep Learning i com a exemples de deep Q-Learning.

5.1.4 Elecció d'eines i llibreries

L'elecció d'eines i llibreries ha sigut complicada. Per un costat hi ha el llenguatge C#, molt senzill d'utilitzar i potent per qualsevol tipus de projecte. Tot i que aquest llenguatge és molt útil, els requeriments del projecte demanen llibreries especialitzades en Machine Learning, que no és comú programar-ho amb C#.

Per l'altre costat, l'estàndard de la indústria a l'hora de programar Machine Learning és utilitzar com a llenguatge de programació Python. Aquest llenguatge té moltes llibreries potents associades, com Tensorflow o Pytorch, que fan que sigui molt senzill desenvolupar models avançats.

Coneixent això, s'ha escollit C# per crear el prototip 1 basat en Q-Learning. S'ha preferit escollir Python per el prototip 2 i el projecte final, pel fet de poder fer ús de les llibreries Tensorflow i Keras que permeten programar models de deep learning ràpids i fàcils de parametritzar.

5.1.5 Prototip 1: Q-Learning

El primer prototip del projecte té la utilitat de veure la funcionalitat més bàsica de l'algorisme de Q-Learning i provar l'estructura de codi més adequada. L'agent té l'objectiu de dirigir-se a un punt de l'entorn de la forma més òptima possible. Es considera que l'objectiu s'ha complert si es veuen signes d'aprenentatge durant l'entrenament sense massa exigència en la optimització.

5.1.6 Prototip 2: Deep Q-Learning

El segon prototip aprofita la implementació del primer prototip ja que la tasca a resoldre és la mateixa però utilitzant deep Q-Learning en comptes de Q-Learning. El prototip 2 es considera acabat, com en el cas del primer prototip, quan es veuen signes d'aprenentatge durant l'entrenament ja que l'objectiu d'aquest prototip és comparar l'algorisme de Q-Learning amb l'algorisme de deep Q-Learning per una mateixa tasca.

5.1.7 Projecte final: Snake

El projecte final parteix del segon prototip però la tasca a complir augmenta considerablement de dificultat. La tasca és que l'agent aprengui a jugar al joc de l'Snake, en el que haurà de recollir menjar per créixer i farà més complicat la manera de moure's per l'entorn. En el projecte final es recullen tots els coneixements dels prototips, tan d'estructuració de codi, com aplicació de l'algorisme de deep Q-Learning fins a l'ús de les llibreries.

5.1.8 Conclusions

Aquest apartat està destinat a explicar quins objectius s'han aconseguit i a valorar els resultats de forma genèrica. A més, també es mostra com es podrien implementar en videojocs els algorismes explicats en aquest projecte i quines són les línies d'investigació futures.

5.2 Cronograma

Activitats	Desembre	Gener	Febrer	Març	Abril	Maig	Juny
Investigació sobre RL							
Documentació del treball							
Cerca de referents							
Elecció d'eines i llibreries							
Prototip 1 Q-Learning							
Prototip 2 Deep Q-Learning							
Projecte final Snake							
Conclusions							

Taula 5 Cronograma amb les diferents fases del projecte. Font: Elaboració pròpia.

6 Disseny i desenvolupament dels prototips basats en aprenentatge per reforç

En aquest apartat es mostra el disseny, la implementació i els resultats obtinguts de la creació dels prototips i el projecte final requerits per aquest treball.

El disseny de cada implementació està estructurat en 6 parts seguint els elements de l'aprenentatge per reforç comentats a l'apartat 2.5:

1. Definició de l'agent, l'objectiu i la tasca que ha de resoldre.
2. Disseny de l'entorn.
3. Definició de l'estat.
4. Definició de les accions disponibles.
5. Elecció de l'estratègia adequada.
6. Definició dels passos i els episodis.

6.1 Prototip 1: Q-Learning

6.1.1 Disseny Prototip

Com s'ha especificat en l'apartat 5, el primer prototip és una implementació bàsica de l'algorisme Q-Learning en què la prioritat és conèixer la implementació de l'algorisme més que aconseguir que l'agent realitzi una tasca molt complexa. El disseny de les diferents parts és:

1. En aquest prototip, l'agent és l'aprenent, que pren decisions per maximitzar una recompensa personal. El seu objectiu és una posició estàtica dins l'entorn. La tasca és arribar fins a l'objectiu de la forma més òptima possible.
2. L'entorn està format per un una quadrícula de 10 x 10 cel·les. Si l'agent surt d'aquesta quadrícula obté una recompensa negativa de -10, en canvi si aconsegueix complir l'objectiu de la tasca obté una recompensa positiva de 10. A més per cada acció que realitza obté una recompensa negativa que varia segons la distància a la que es troba respecte l'objectiu. La posició inicial escollida per l'agent és la posició (2, 2) de la quadrícula i és a la que

retornarà cada vegada que comenci un nou episodi. La posició escollida per l'objectiu és la (7, 8) i resta fixa durant tota la tasca.

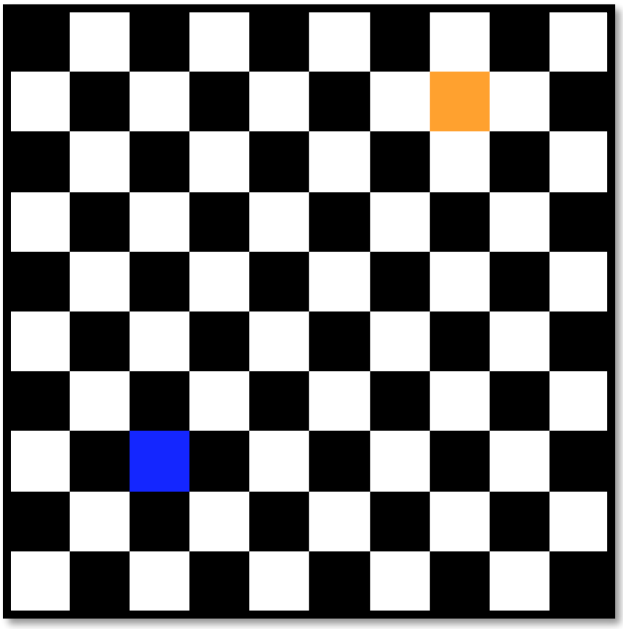


Figura 6.1 Estat inicial de l'entorn format per una quadrícula de 10 x 10 cel·les. En aquesta representació, l'agent és el quadrat blau i l'objectiu el quadrat taronja. Font: elaboració pròpia.

3. Com a estat només es considera la posició actual de l'agent en els eixos X i Y.
4. L'agent pot complir el seu objectiu movent-se per les cel·les d'una en una. Per tant les accions disponibles són: anar a la dreta, a l'esquerra, amunt i avall.
5. La policy o estratègia utilitzada en aquest prototip és la ε -greedy, explicada amb detall a l'apartat 2.7.2 més 500 iteracions extres d'estratègia únicament greedy per assegurar que els valors de qualitat que resolen la tasca estan prou balancejats.
6. Cada iteració de l'algorisme compleix un pas. En aquest pas, l'agent escull una acció dins de les quatre accions disponibles definides al punt 4 seguin l'estratègia definida al punt 5. No hi ha límit de passos per episodi sinó que l'agent pot recórrer als que necessiti per completar-lo. L'episodi s'acaba quan l'agent arriba a la posició de l'objectiu o quan surt dels marges de la quadrícula, llavors comença el següent. La tasca finalitza quan la variable èpsilon és més petita que 0.

6.1.2 Implementació del Prototip 1

Definició dels híper-paràmetres

A la taula 6 hi ha els sis atributs constants utilitzats com a híper-paràmetres amb els seus respectius valors i descripcions. Els valors utilitzats s'han aconseguit valorant quina era la mesura adequada per la tasca a resoldre.

Híper-paràmetre	**Valor**	**Descripció**
Discount factor - gamma	0.9	Valor utilitzat per actualitzar els valors de qualitat
Learning rate - alfa	0.7	Valor utilitzat per actualitzar els valors de qualitat
GridSize	10	Mida de la quadrícula
Recompensa positiva	10	Valor que retorna l'entorn quan l'agent realitza una bona acció i l'ajuda a aprendre
Recompensa negativa	-10	Valor que retorna l'entorn quan l'agent realitza una mala acció i l'ajuda a aprendre
Epsilon decay rate (Taxa Reducció Epsilon)	0.0005	Valor amb el que decau ε a cada pas de l'entrenament per l'estratègia ε-greedy

Taula 6 Híper-paràmetres utilitzats en el prototip 1 amb els valors corresponents. Font: elaboració pròpia.

Implementació en codi

Per estructurar el codi s'ha utilitzat un esquema de classes com es veu a la figura 6.2. En aquest esquema UML (Llenguatge Unificat de Modelat) es poden veure 4 classes (Agent, Program, State, Vector2). L'ús d'aquestes 4 classes és per una millor estructuració del codi però, una implementació tan senzilla com la d'aquest prototip, es pot realitzar amb una sola classe.

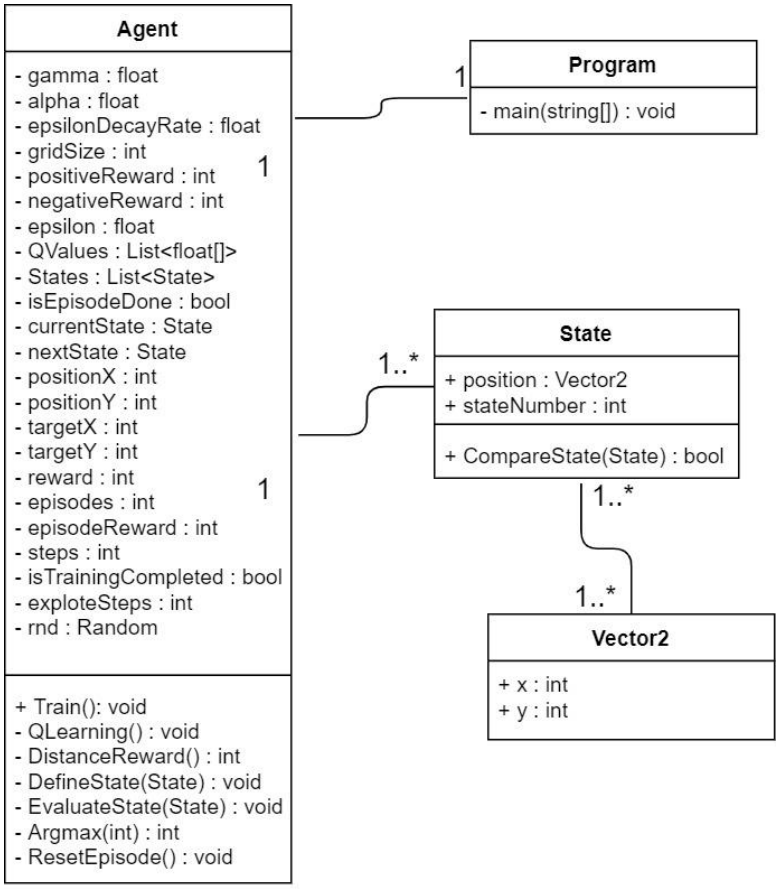


Figura 6.2 Diagrama de classes UML per la implementació del prototip de Q-Learning. Font: elaboració pròpia.

La classe principal és la classe Agent. Aquesta conté una sèrie de mètodes que permeten desenvolupar l'algorisme Q-Learning que són els següents:

Train: mètode que conté el bucle per realitzar tots els episodis de la tasca. També avalua si un episodi ha acabat. Si és cert crida la funció ResetEpisode i si és fals, crida la funció QLearning.

QLearning: aquest mètode conté tota la sintaxi relacionada amb l'aprenentatge per reforç. Primer escull una acció utilitzant l'estratègia ε-greedy i disminueix la variable èpsilon. Després avalua si el valor èpsilon és més petit que 0. En cas de ser cert, la variable isTrainingCompleted es posa com a certa i s'acaba la tasca. En cas de ser fals s'executen 500 passos extres. El següent pas és executar l'acció i avaluar quina és la recompensa retornada i si ha acabat l'episodi. Més enllà de la recompensa que ja pugui tenir l'agent, es crida la funció DistanceReward per assegurar una recompensa per distància per ajudar l'agent a saber més fàcilment on es troba

l'objectiu. Finalment s'utilitza la funció d'actualització dels valors de qualitat de l'algorisme Q-Learning i el següent estat passa a ser l'actual.

DistanceReward: mesura i retorna la distància entre l'agent i l'objectiu.

DefineState: Afegeix un estat passat per paràmetre a la llista d'estats i en crea un vector per els valors de qualitat associats a aquest estat i que també afegeix a la llista de QValues.

EvaluateState: Avalua si un estat ja ha sigut creat.

Argmax: mètode que busca l'índex del valor més alt entre els valors de qualitat d'un estat.

ResetEpisode: mètode que reinicia l'episodi.

En relació a les variables, les principals són la recompensa, èpsilon, l'estat actual i el següent i les llistes que emmagatzemen els estats coneguts i els valors de qualitat.

La segona classe és la classe State. Aquesta és més senzilla que l'anterior i està implementada de tal manera que se'n puguin crear les instàncies necessàries. Conté només un mètode que serveix per comparar l'estat amb qualsevol altre. Les variables que emmagatzema l'estat són: un vector de la posició de l'agent i el número de l'estat en el moment de ser creat.

La tercer classe és la Vector2 que representa un vector de dos posicions. Té dos variables: x i y, corresponents a la posició.

L'última classe és la classe Program que només conté el mètode main per crear una instància de la classe Agent i executar el mètode Train.

Algorisme

Al mètode QLearning conté la part més fonamental de l'algorisme Q-Learning, tot i així, és a la funció Train on es troba el bucle de l'algorisme que es mostra a continuació:

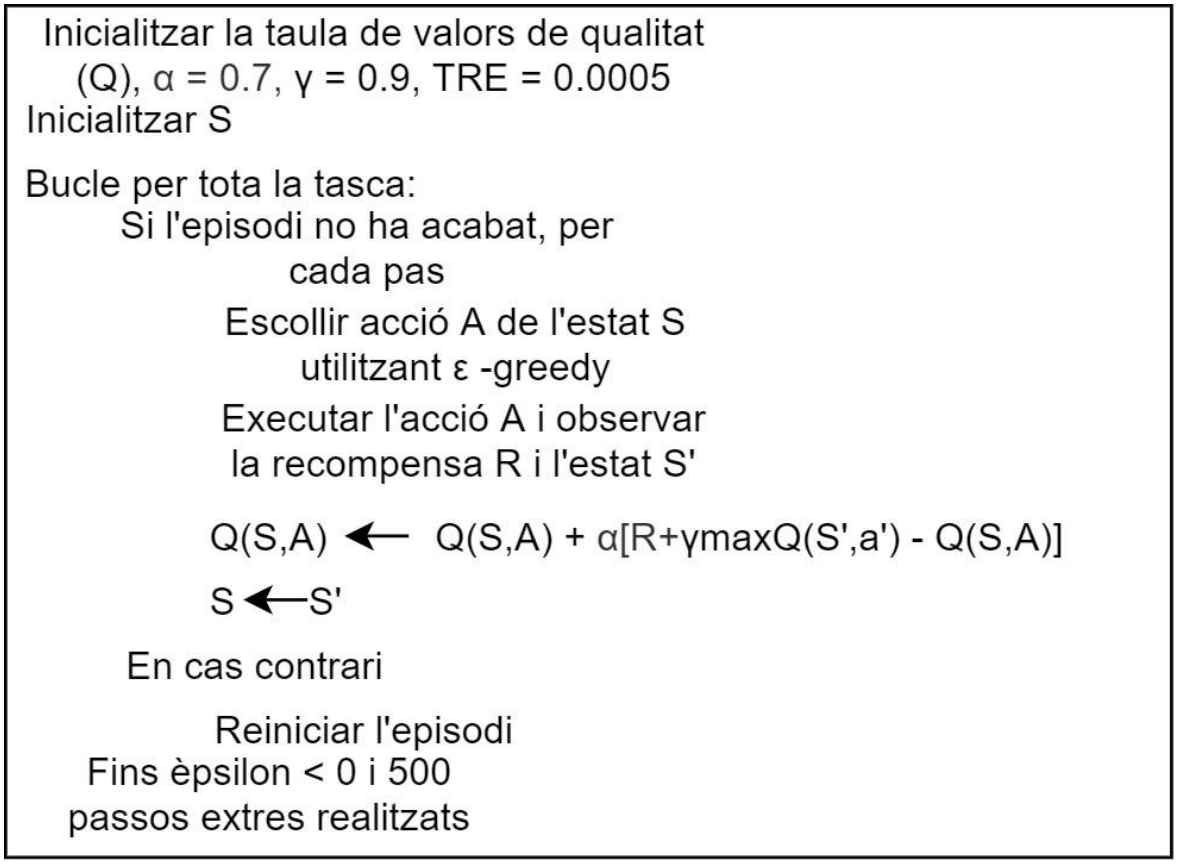


Figura 6.3 Pseudocodi de l'algorisme Q-Learning. Font: elaboració pròpia

La implementació del prototip 1 ha seguit el pseudocodi de la figura 6.3 i l'explicació proporcionada a l'apartat 2.7.2. Apart d'inicialitzar la taula de valors de qualitat en format de llista de vectors per poder ser ampliable dinàmicament, també s'inicialitza una llista on es guarden tots els estats visitats. Quan l'agent observa el següent estat S', donat que pot ser que ja l'hagi visitat, es compara amb tots els estats de la llista d'estats utilitzant la funció EvaluateState. En cas d'existir a la llista, es retorna l'estat que ja existeix, mentre que si no està a la llista, es crea un nou estat amb la funció DefineState. El mètode DefineState afegeix el nou estat, li assigna un número a la variable stateNumber i crea un vector que afegeix a la taula de valors de qualitat corresponent a l'estat creat. Dins el mètode QLearning és on es s'observa la recompensa retornada. Primer s'avalua si l'agent ha sortit del perímetre de l'entorn i després s'avalua si la posició de l'agent és la mateixa que la de l'objectiu. En els dos casos, si la condició és certa, l'episodi s'acaba i comença el següent. La part d'escollir l'acció funciona com està explicat a l'apartat 2.7.2 utilitzant la variable èpsilon per l'estratègia èpsilon-greedy, que just abans

d'executar l'acció, es redueix una certa quantitat fixa utilitzant la constant taxa de reducció d'èpsilon (TRE).

6.1.3 Anàlisi de resultats

Els resultats d'aquest prototip estan dividits en 4 gràfiques i un mapa de calor. Han estat obtinguts d'un sol entrenament després de comprovar que els diferents entrenaments coincidien en els resultats, ja que el més important dels següents gràfics és la tendència i el rendiment general i no els valors concrets en un moment del temps. Això és degut a que a l'inici dels entrenaments, l'agent tendeix a explorar més i per tant sempre escull accions aleatòries que no tenen un valor d'anàlisi concret, si no que el seu valor està en comparar aquestes dades amb les dades resultants de quan l'agent està explotant l'entorn. L'entrenament per els següents resultats s'ha realitzat utilitzant els híper-paràmetres inicials mostrats a l'apartat anterior.

Gràfiques de resultats

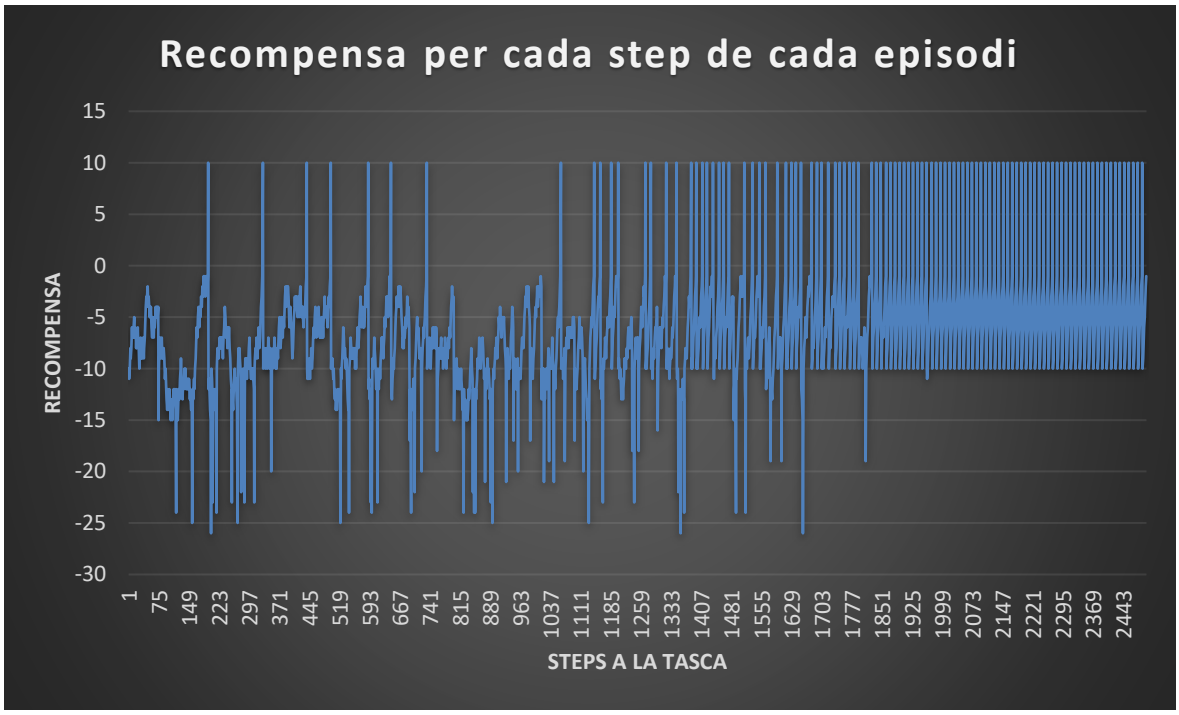


Figura 6.4 Gràfic sobre la recompensa que rep l'agent per cada pas de cada episodi. Font: elaboració pròpia.

A la gràfica de la figura 6.4 es pot veure la recompensa obtinguda a cada pas que hi ha hagut durant l'entrenament de la tasca. Cada punt situat a l'altura 10 és un

pas on l'agent ha arribat a l'objectiu mentre que la resta de punts la recompensa és negativa, ja sigui per haver sortit dels marges de la quadrícula o per la distància amb la que es troba de l'objectiu. Sabent això es pot veure com la primera meitat de la gràfica és caòtica degut a la tria d'accions aleatòries per explorar que de forma puntual l'acosten a l'objectiu. En canvi la segona meitat de la gràfica tendeix a tenir punts amb valor 10 de forma més regular, fet que significa que l'agent està aprenent que és positiu arribar a l'objectiu i ho fa de forma més freqüent.

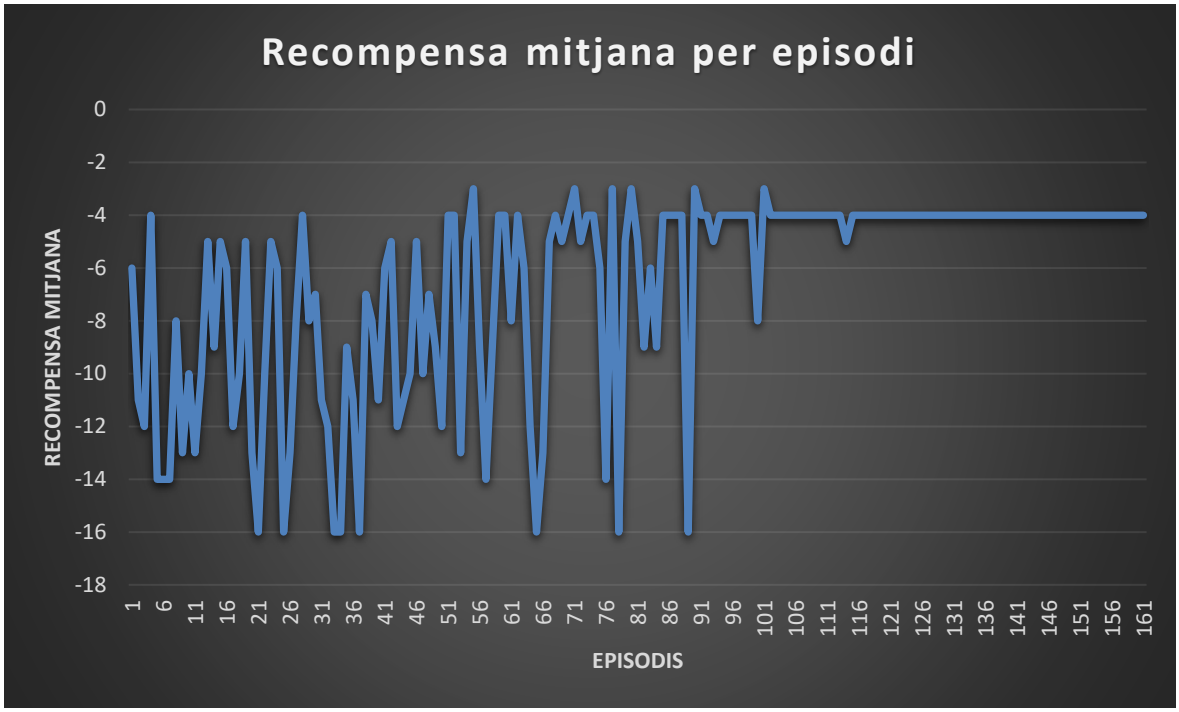


Figura 6.5 Gràfic sobre la recompensa mitjana per episodi. Font: elaboració pròpia.

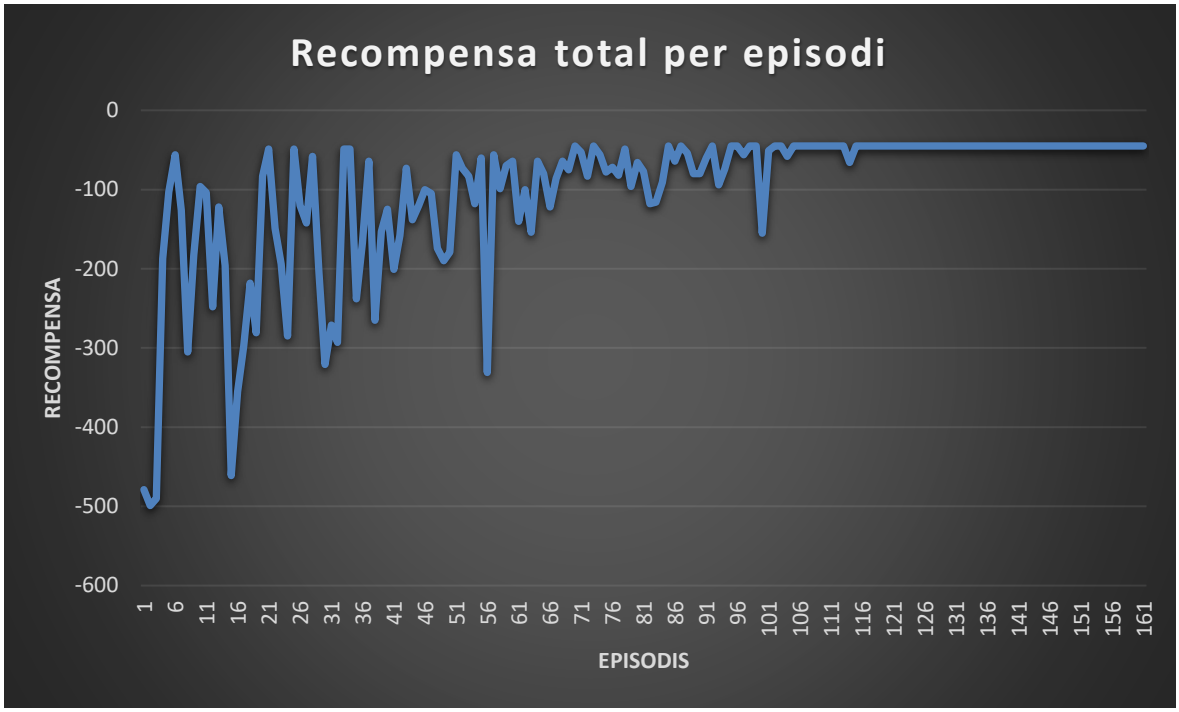


Figura 6.6 Gràfic sobre la recompensa total per episodi. Font: elaboració pròpia.

Les gràfiques de les figures 6.5 i 6.6 mostren la recompensa per episodi i la recompensa mitjana per episodi. Aquestes gràfiques també mostren com la primera part és caòtica mentre que la segona meitat de la gràfica tendeix cada cop més a estabilitzar-se a un valor, fet que en aquest entorn és positiu.

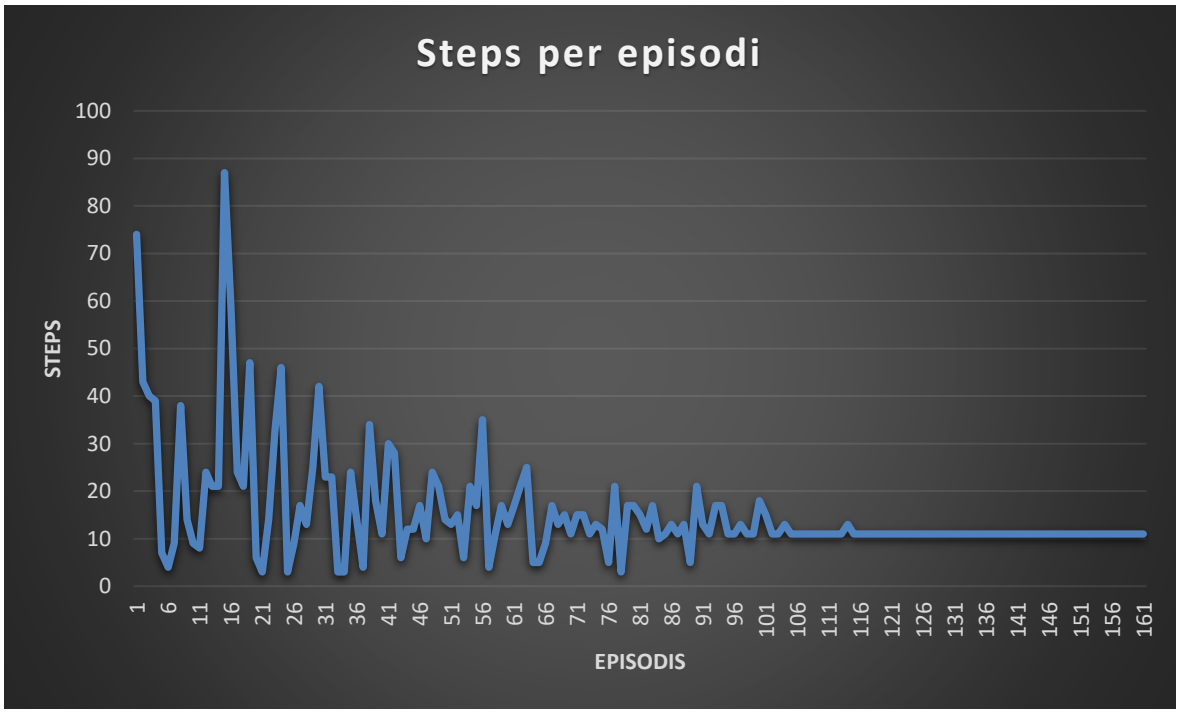


Figura 6.7 Gràfic sobre la quantitat de passos per episodi. Font: elaboració pròpia.

El gràfic de la figura 6.7 mostra la quantitat de passos per episodi que han ocorregut durant l'entrenament. En aquest gràfic també es nota una tendència general en la que l'agent, conforma passen els episodis, necessita menys passos per resoldre la tasca. Cap al principi es pot veure com necessita gairebé 90 passos mentre que cap al final es manté regular a 11 passos, que és el mínim per resoldre la tasca òptimament.

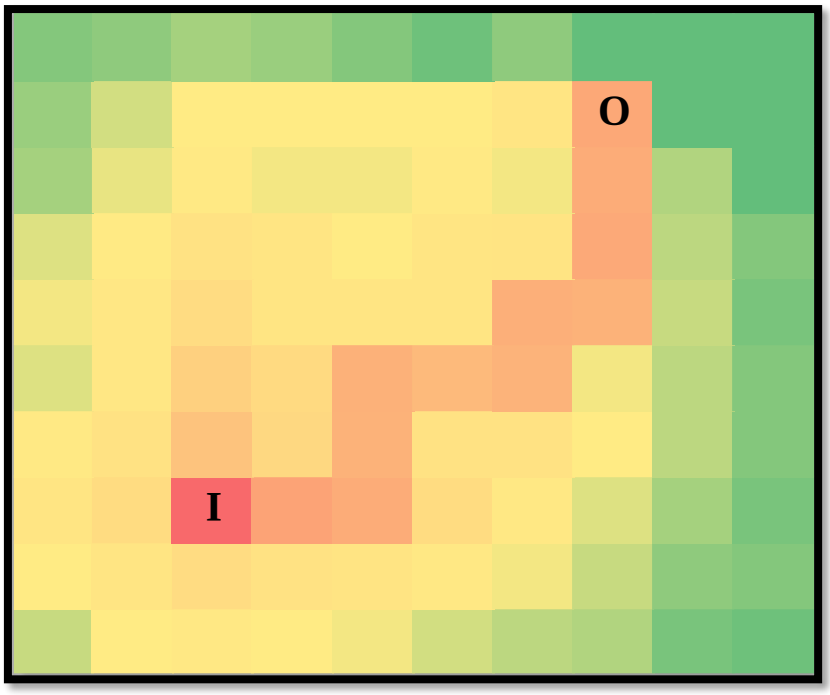


Figura 6.8 Mapa de calor segons la posició en la que ha estat l'agent. En vermell les posicions on més vegades ha passat i en verd les que menys o no hi ha passat cap vegada. La I és el punt d'inici i la O l'objectiu. Font: elaboració pròpia.

La figura 6.8 és un mapa de calor que representa totes les posicions per on ha passat l'agent durant l'entrenament. Les cel·les més vermelloses és per on ha passat més vegades mentre que les cel·les més verdes és per on no ha passat cap vegada o hi ha passat poques vegades. Com es pot veure al mapa de calor, l'agent ha passat moltes vegades per el punt d'inici ja que cada vegada que es reinicia l'episodi torna al punt d'inici. A partir d'aquí, entre la posició d'inici i la posició de l'objectiu ha definit el camí més curt per el que ha passat més vegades que per els

altres camins de la mateixa llargada. En aquesta tasca el camí més curt té una llargada d'11 cel·les i n'hi ha més d'un.

Modificació d'híper-paràmetres

Com a resultat extra, s'ha provat a modificar dos dels híper-paràmetres inicials: alfa i la taxa de reducció d'èpsilon (TRE) tornant a realitzar l'entrenament.

Híper-paràmetre	**Valor per defecte**	**Valor modificat**	**Notes**
Alfa	0.7	1	No s'observa diferència amb el valor donat per defecte.
		0.1	Com en l'anterior, tampoc s'observa diferència amb el valor per defecte. La hipòtesis és que la tasca és massa senzilla com perquè aquestes modificacions siguin rellevants per canviar la optimització dels valors de qualitat i afectar l'aprenentatge de l'agent.
Taxa reducció èpsilon (TRE)	0.0005	0.1	No és possible realitzar la tasca, l'agent no explora suficient l'entorn per poder aprendre.
		0.01	Millora respecte l'anterior però continua sense arribar a aprendre ja que l'exploració tampoc és suficient.
		0.001	L'agent resol la tasca només algunes vegades. La resta no, degut a que els valors de qualitat no estan prou balancejats per la falta d'exploració. Per balancejar els valors l'agent ha de passar vàries vegades per un mateix estat, que és el que permet aproximar-los al valor òptim. Pot ser considerat el llindar entre que l'agent aprengui o no aprengui en aquesta tasca.
		0.0008	Aquest és el primer valor on l'agent explota l'entorn de forma mínimament òptima. Amb valors més petits està assegurat l'aprenentatge, com el valor per defecte, però si són massa petits, l'algorisme perd eficiència.

Taula 7 Taula que mostra el resultat de dos paràmetres modificats. Font: elaboració pròpia.

6.2 Prototip 2: Deep Q-Learning

6.2.1 Disseny prototip

El segon prototip té per objectiu realitzar una primera implementació de l'algorisme de deep Q-Learning per veure'n els seu funcionament essencial. Com a tasca s'utilitza la mateixa que en el prototip 1 per així poder establir una comparació entre els dos prototips. Seguint l'estructura presentada al principi de l'apartat 6, el disseny té les següents parts:

1. Aquest prototip, com l'anterior, no es visualitza. Com en el primer, l'agent també és l'aprenent, que es mou per un entorn amb la tasca d'arribar a un objectiu estàtic de forma òptima.
2. L'entorn és semblant al del prototip 1. Així, està format per una quadrícula de 10x10 amb l'agent a la posició (2, 2) i l'objectiu a la posició (7, 8). La recompensa obtinguda per l'agent a l'arribar a l'objectiu és de 100, si surt de la quadrícula de -10 i 0 per la resta de casos. Aquí no hi ha recompensa per distància.
3. L'estat està definit per 4 variables: posició X i posició Y de l'agent i distància X i Y de l'agent amb l'objectiu.
4. Les accions continuen sent les mateixes que al prototip 1, és a dir, dreta, esquerra, amunt i avall.
5. L'estratègia utilitzada també és la ε -greedy amb reducció lineal de la variable èpsilon.
6. En aquest prototip, com en l'anterior, quan l'agent realitza una acció es compleix un pas. Però, a diferència, no comença un nou episodi quan l'agent arriba a l'objectiu, sinó que només es reinicia la posició de l'agent. L'episodi canvia quan l'agent surt dels límits de la quadrícula. La tasca finalitza quan la variable èpsilon és menor que 0.

6.2.2 Implementació del prototip 2

Definició dels híper-paràmetres

Híper-paràmetre	Valors per defecte	Descripció
Discount factor - gamma	0.995	Valor que regula el descompte de les recompenses futures.
Epsilon decay rate (Taxa Reducció Epsilon)	0.0001	Valor amb el que decau ε a cada pas de l'entrenament per l'estratègia ε-greedy.
Batch Size	64	Valor que determina la quantitat d'experiències aleatòries que utilitza l'algorisme cada vegada que realitza un aprenentatge a cada pas.
Memòria mínima	400	Valor que limita el mínim d'experiències necessàries perquè el model comenci l'aprenentatge.
Memòria màxima	5000	Valor que limita el màxim d'experiències que emmagatzema l'agent.
Grid Size	10x10	Mida de la quadrícula.
Recompensa positiva	100	Valor que retorna l'entorn quan l'agent realitza una bona acció i l'ajuda a aprendre
Recompensa negativa	-10	Valor que retorna l'entorn quan l'agent realitza una mala acció i l'ajuda a aprendre
Neurones capa oculta	12	Quantitat de neurones a la capa oculta del model
Funció activació capa oculta	ReLu	Tipus de funció d'activació de neurones per la capa oculta del model
Funció activació capa sortida	Linear	Tipus de funció d'activació de neurones per la capa de sortida del model
Optimitzador	Adam	Tipus d'optimitzador utilitzat per disminuir el valor dels pesos, basat en Gradient Descent

Taula 8 Taula d'híper-paràmetres pel prototip 2. Font: elaboració pròpia.

Arquitectura i configuració del model

Com està indicat a l'apartat 2.7.3, el deep Q-Learning requereix l'ús de funcions d'aproximació com són les xarxes neuronals. En aquest prototip s'utilitza les llibreries Tensorflow i Keras que permeten construir models senzills i ràpidament.

El model de xarxa neuronal utilitzat és un model seqüencial que permet 4 valors d'entrada a la primera capa. Després conté, per defecte, una capa oculta amb 12

neurones i una capa de sortida amb 4 neurones. Això fa un total de 112 paràmetres optimitzables entre pesos i bias. La capa oculta utilitza la funció d'activació *relu* mentre que la capa de sortida utilitza la funció d'activació *linear*. Com a funció per establir el cost s'utilitza l'error quadrat mitjà (mean squared error) i com a optimitzador es fa servir un basat en gradient descent anomenat *Adam* (explicat a la secció 2.4).

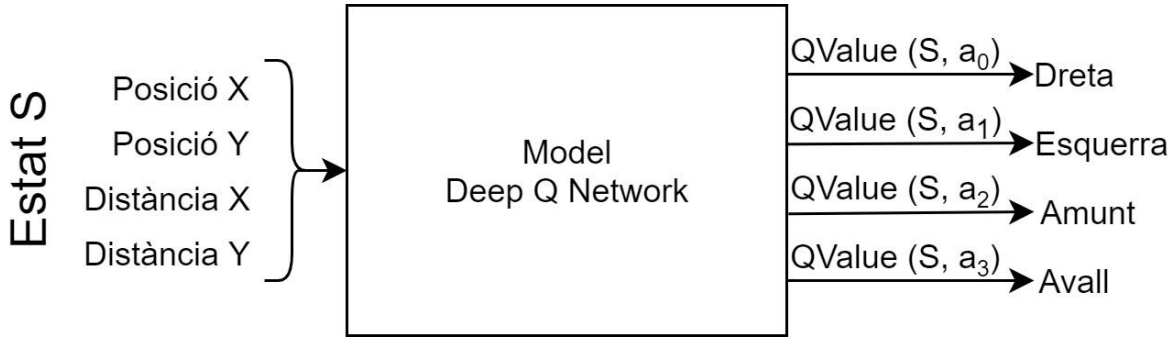


Figura 6.9 Esquema que mostra la informació d'entrada i de sortida pel model del prototip 2. Font: elaboració pròpia.

Com es pot veure a la figura 6.9, el model rep 4 entrades, dos per la posició de l'agent en els eixos X i Y i dos entrades més corresponents a la distància entre l'agent i l'objectiu també en els dos eixos. Llavors extreu 4 valors de qualitat (Q), un per cada acció disponible corresponent a les quatre direccions possibles.

Implementació en codi

Per aquest prototip s'ha utilitzat una única classe anomenada Agent que conté tots els mètodes, variables i híper-paràmetres. Els mètodes utilitzats són els següents:

init: s'inicialitzen els híper-paràmetres, es crea la memòria d'experiències, que és una llista que funciona com una *queue* i es creen els dos models de xarxa neuronal, anomenats qModel el que aprèn i targetModel el que calcula l'objectiu semi-estàtic del model.

buildModel: es crea un model seqüencial amb l'ajuda de la llibreria Keras tal com s'indica en el sub-apartat anterior.

copyWeights: els pesos de la xarxa neuronal aprenent es copien a la xarxa neuronal que proveeix l'objectiu semi-estàtic per calcular l'error durant la fase d'aprenentatge.

addExperience: se li passen com a paràmetres l'estat actual, l'acció escollida, la recompensa i el seu estat retornats i si s'ha acabat l'episodi després d'executar aquesta acció. Després afegeix aquestes dades a la memòria d'experiències.

getState: calcula la distància en l'eix X i la distància en l'eix Y entre la posició de l'agent i la posició de l'objectiu. Llavors afegeix en una llista les dos distàncies i la posició en els dos eixos de l'agent per acabar retornant l'estat.

takeAction: aquest mètode agafa com a paràmetre un estat i s'escull una acció utilitzant l'estratègia èpsilon-greedy. En el primer cas s'escull una de les 4 accions possibles de forma aleatòria. En el segon cas, la xarxa neuronal aprenent agafa l'estat passat per paràmetre i prediu una acció basant-se en el model intern de la xarxa. Finalment redueix la variable èpsilon i retorna l'acció escollida.

resetEpisode: estableix la posició en X i en Y de l'agent.

executeAction: agafa per paràmetre una acció i l'executa a l'entorn canviant la posició de l'agent. La funció retorna una tupla formada per dos valors booleans. El primer indica si l'agent ha arribat a la posició de l'objectiu i el segon indica si l'agent ha sortit dels marges de la quadrícula.

backpropagateLearning: aquesta funció realitza tota la fase d'aprenentatge de l'algorisme. Comença avaluant si la quantitat d'experiències que hi ha a la memòria supera el mínim. En cas de ser cert s'executa la resta de la funció. Primer es selecciona un sub-conjunt d'experiències de la memòria, de forma aleatòria. Es creen cinc llistes, una per cada tipus de dada dins les experiències, i s'omplen amb les experiències seleccionades prèviament. Es fan les prediccions del target utilitzant la llista de l'estat actual per emplenar la llista de targets fent ús del qModel. També es fan les prediccions utilitzant la llista amb el següent estat utilitzant els dos models de xarxa neuronal. Iterant sobre cada experiència seleccionada, s'avalua si l'experiència actual és terminal, com es pot veure a la figura 29. Si el valor és fals es computa la llista de targets utilitzant l'equació de Bellman, que és la recompensa sumada a la multiplicació de gamma per l'índex amb el valor màxim de la predicció de valors del target, comunament anomenat argmax. Si és cert no s'utilitza l'equació de Bellman sinó que directament el target actual passa a ser la recompensa actual ja que l'episodi ha acabat i per tant no hi haurà més recompenses futures. Finalment, es crida la funció d'entrenament del model utilitzant com a paràmetres

la llista d'estats actuals, la llista de targets prèviament calculats, el batch size, el número d'epochs, qué és el número d'iteracions sobre totes les dades seleccionades, que en aquest cas es deixa a 1 per realitzar només una iteració i finalment hi ha el valor verbose que pot mostrar feedback durant l'entrenament si s'utilitzen moltes epochs però, en aquest cas, es manté silenciat amb el valor 0.

train: inicialitza les variables score, numEpisode i isTaskCompleted. A continuació hi ha dos bucles; el primer itera mentre la tasca no ha sigut completada, sumant un episodi a cada iteració. El segon bucle està dins del primer, executa cada pas dins d'un mateix episodi i itera fins que s'executa una sentència break. Dins d'aquest bucle hi ha tot el codi directament relacionat amb l'aprenentatge per reforç. L'algorisme comença escollint una acció i executant-la a l'entorn. Segons l'acció, es retorna una recompensa i el següent estat. La informació de l'estat actual, l'acció realitzada, la recompensa i el següent estat retornats i si s'ha acabat l'episodi s'afegeix a la memòria d'experiències amb la funció addExperience. Abans d'executar la funció d'aprenentatge, el seguent estat passa a ser l'estat actual i es suma un pas al comptador. Llavors, es crida la funció backpropagateLearning, explicada abans. Finalment hi ha tres condicionals; el primer reinicia l'episodi si l'agent ha arribat a l'objectiu. El segon utilitza la funció copyWeights, reincia l'episodi, mostra informació sobre el rendiment per pantalla i executa un break per sortir del bucle de l'episodi. El tercer avalua si la variable èpsilon és més petita que 0, en cas afirmatiu, la variable isTaskCompleted es posa com a certa ja que la tasca es considera finalitzada, també mostra informació sobre el rendiment i executa el break.

Algorisme

S'ha implementat l'algorisme en base al pseudocodi de la figura 6.10 en els mètodes train, que conté els bucles i la sintaxi que hi ha abans de seleccionar les experiències i, backpropagateLearning que conté l'aprenentatge del model, des de seleccionar un conjunt d'experiències fins a entrenar el model.

Inicialitzar la memòria d'experiències M amb capacitat N = 5000, inicialitzar la Q Network amb pesos aleatòris, γ = 0.995, TRE = 0.0001

Bucle per cada episodi:

Inicialitzar estat S

Bucle per cada pas de l'episodi

Escollir acció A partint de l'estat S utilitzant ε-greedy

Executar l'acció A i observar la recompensa R i el següent estat S'

Afegir l'experiència (S, A, R, S', D) a M

$S \leftarrow S'$

Seleccionar un conjunt aleatori de M

Calcular $y_i = \begin{cases} R_i & \text{Si S' és terminal} \\ R_i + \gamma \max Q(S', a'; w) & \text{Si S' no és terminal} \end{cases}$

Entrenar el model utilitzant backpropagation amb un algoritme basat en gradient descent per disminuir els pesos de la Q Network

Fins que S és terminal

Fins èpsilon < 0

Figura 6.10 Pseudocodi de l'algorisme Deep Q-Learning pel prototip 2. Font: elaboració pròpia basat en **(Mnih et al, 2013)**.

6.2.3 Anàlisi de resultats

Els resultats d'aquest prototip, com a l'anterior, estan dividits en 4 gràfiques i un mapa de calor. També han estat obtinguts d'un sol entrenament després de comprovar que els diferents entrenaments obtenien uns resultats semblants. L'entrenament per els següents resultats s'ha realitzat utilitzant els híper-

paràmetres inicials mostrats a l'apartat anterior que, pel tipus de tasca proposada, l'entrenament pot ser realitzat tan ràpidament com ho permeti la màquina.

Gràfiques de resultats

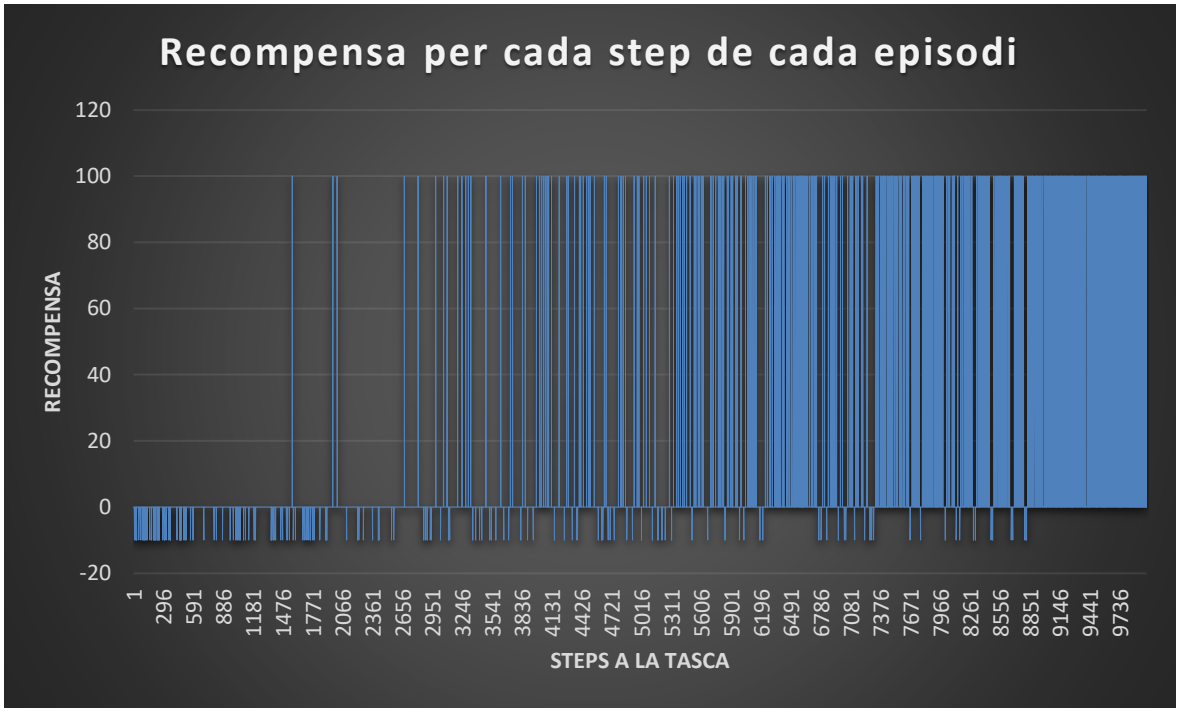


Figura 6.11 Gràfic sobre la recompensa que rep l'agent per cada pas de cada episodi. Font: elaboració pròpia.

A la gràfica de la figura 6.11 es pot veure la recompensa obtinguda a cada pas que hi ha hagut durant l'entrenament de la tasca. Cada punt situat a l'altura 100 és un pas on l'agent ha arribat a l'objectiu, alguns punts l'agent ha aconseguit una recompensa negativa de -10 per haver sortit dels marges de la quadrícula, mentre que la resta de punts, tot i que no s'aprecia per la gran quantitat de passos que hi ha hagut durant la tasca, la recompensa és de 0. Al primer prototip es veia un canvi de comportament en l'agent a partir de la meitat de la tasca però en aquest prototip es pot veure com en el primer terç de la gràfica és caòtica i no mostra signes d'aprenentatge, degut a la tria d'accions aleatòries per explorar. Després del primer terç, l'agent comença a mostrar signes d'aprenentatge conforma explota més l'entorn, cosa que significa que està aprenent que és positiu arribar a l'objectiu. A partir de la segona meitat de la gràfica, l'agent insisteix en dirigir-se a l'objectiu cada com amb més freqüència i de manera més directa.

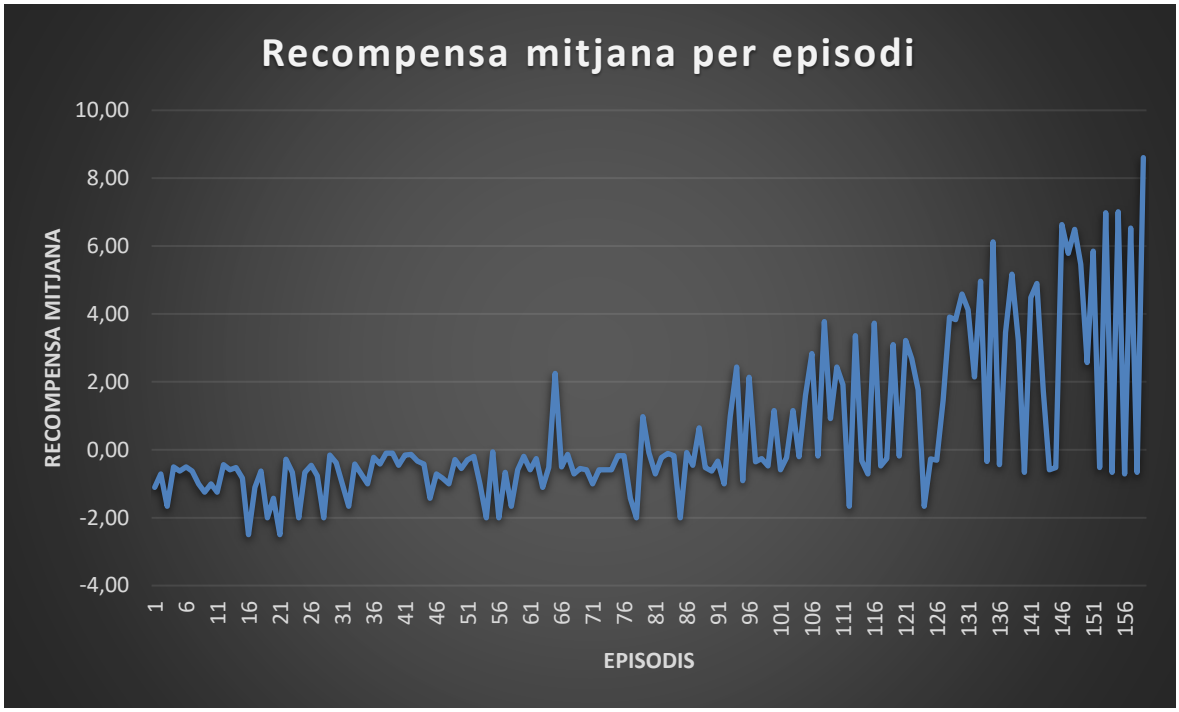


Figura 6.12 Gràfic sobre la recompensa mitjana per episodi. Font: elaboració pròpia.

El gràfic 6.12 és potser el gràfic que mostra els resultats més clars. A primer cop d'ull es pot veure com la recompensa mitjana generalment augmenta cap a l'últim terç de la tasca. Això és degut a que l'agent es torna eficient aconseguint recompensa positiva i tot i que el total es divideix entre més passos, el resultat acaba sent prou elevat. Això es demostra observant la primera meitat de la gràfica; la recompensa també es divideix amb una bona quantitat de passos però l'agent sol aconseguir una recompensa de 0 o negativa fent que el resultat acabi sent negatiu.

Figura 6.13 Gràfic sobre la recompensa total per episodi. Font: elaboració pròpia.

A diferència del gràfic anterior, en el gràfic 6.13 no s'aprecia correctament el progrés de l'agent durant la tasca, excepte per l'última part on els valors tenen una tendència més alta ja que l'agent aconsegueix més recompensa positiva. El que porta a entendre és que la variable èpsilon que regula l'estratègia èpsilon-greedy ha d'estar amb valors molt pròxims al 0 per poder notar canvis significatius en l'aprenentatge, que és quan està explotant més l'entorn.

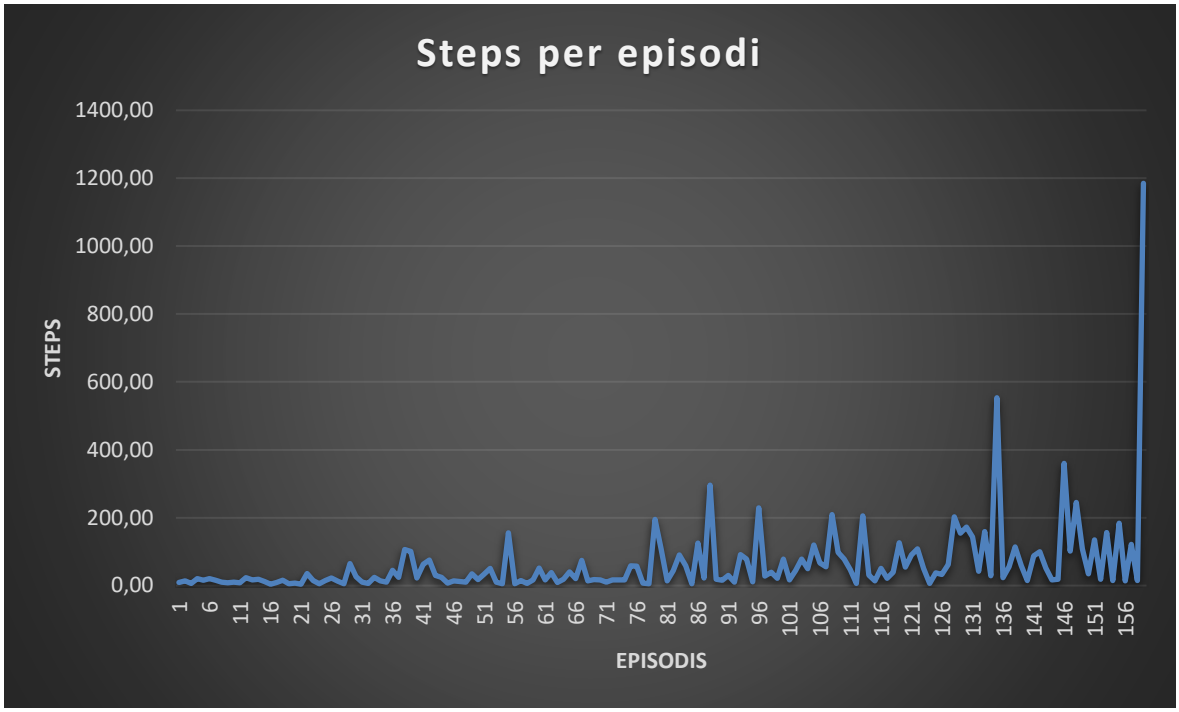


Figura 6.14 Gràfic sobre la quantitat de passos per episodi. Font: elaboració pròpia.

Tal com mostra la gràfica 6.14, la quantitat de passos per episodi augmenta considerablement cap al final de la tasca fins arribar a 1200 passos a l'últim episodi de la tasca, quan la variable èpsilon ja ha arribat a 0 i l'agent està explotant completament el coneixement que disposa de l'entorn. En el primer prototip que un episodi tingués pocs passos podia ser o perquè l'agent anava directament en direcció els marges de l'entorn o perquè anava directament en direcció a l'objectiu. En aquest segon prototip, el funcionament dels episodis ha sigut replantejat fent que no comenci un nou episodi quan l'agent arriba a l'objectiu. Això comporta que com més curts siguin els episodis, pitjor ho està fent l'agent i, com més llargs, més bé ho fa ja que ha après a sobreviure a l'entorn.

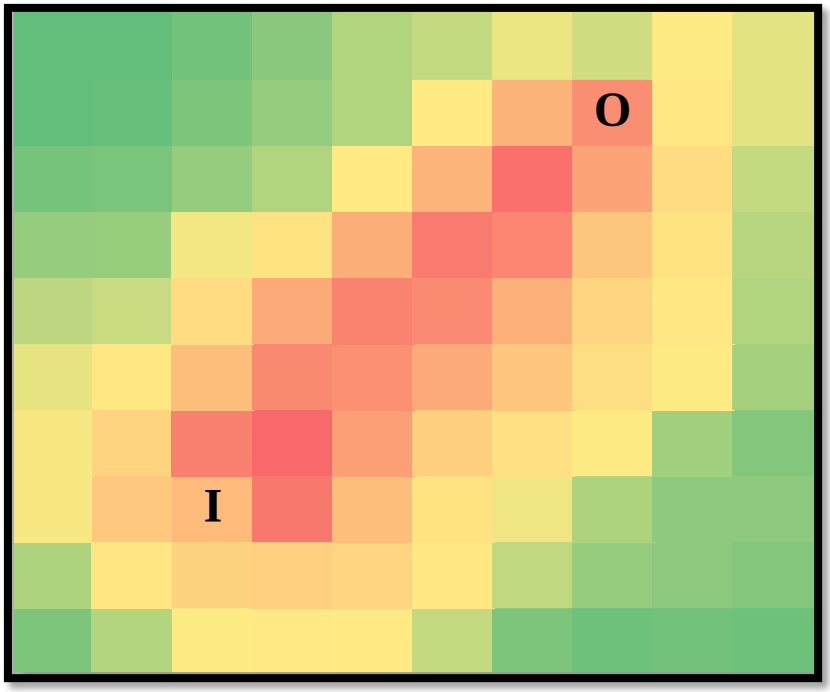


Figura 6.15 Mapa de calor segons la posició en la que ha estat l'agent. En vermell les posicions on més vegades ha passat i en verd les que menys o no hi ha passat cap vegada. La I és el punt d'inici i la O l'objectiu. Font: elaboració pròpia.

La figura 6.15 és un mapa de calor que representa totes les posicions per on ha passat l'agent durant l'entrenament. Les cel·les més vermelloses és per on ha passat més vegades mentre que les cel·les més verdes és per on no ha passat cap vegada o hi ha passat poques vegades. El mapa de calor té una zona entre el punt d'inici i el punt objectiu que és clarament més vermellosa que la resta de l'entorn. Aquesta zona és un conjunt de cel·les difuminades entre els dos punts que no delimiten un camí concret, com passa en el primer prototip, sinó que l'agent ha sigut capaç de generalitzar el camí cada vegada que havia d'anar cap a l'objectiu gràcies a l'ús de funcions d'aproximació que permeten a l'agent generalitzar el moviment en un espai. En aquesta tasca el camí més curt té una llargada d'11 cel·les i n'hi ha més d'un, que l'agent possiblement ha utilitzat durant la tasca. És interessant visualitzar com les cel·les per les que ha passat més no són ni el punt d'inici ni el punt objectiu sinó que són justament les dos cel·les que estan directament en diagonal i situades entre aquests dos punts.

Modificació d'híper-paràmetres

La taula 9 mostra els paràmetres i valors que s'han modificat per aconseguir diferents resultats. El rendiment és avaluat amb un valor anomenat puntuació episòdica mitjana amb la idea d'aconseguir visualitzar els resultats de tota la tasca concentrats en una sola variable. Per aconseguir aquest valor és utilitzada una variable anomenada score que suma un punt cada vegada que l'agent arriba a l'objectiu. Al principi de cada episodi la variable score es reinicia a 0 i a l'acabar-lo es suma el valor resultant a una variable que acumula tota la puntuació dels episodis, anomenada totalScore. El valor d'aquesta última variable dividida pel nombre d'episodis de la tasca dona com a resultat la puntuació episòdica mitjana. S'agafa com a valor de referència de la puntuació episòdica mitjana el que s'aconsegueix utilitzant els valors per defecte descrits a l'apartat de definició d'híper-paràmetres, que dona un resultat de 2.39 (145 episodis, 347 puntuació total).

Híper-paràmetre o configuració	**Valors per defecte**	**Valor modificat**	**Puntuació episòdica mitjana (episodis, puntuació total)**	**Notes**
Gamma	0.995	0.5	0.52 (205, 106)	S'evidencia que aquest paràmetre és molt important per un bon aprenentatge. Si no és pròxim a 1, no puntua correctament les recompenses futures i l'algorisme perd eficàcia.
Taxa de reducció d'èpsilon (TRE)	0.0001	0.0002	0.18 (77, 14)	Per aquesta tasca, el valor per defecte és el valor mínim on està assegurat l'aprenentatge de l'agent. Pel valor modificat, l'agent no arriba a explorar suficientment.
Batch size	64	32	0.7 (162, 113)	El resultat demostra que modificar aquest valor és rellevant per la fase d'aprenentatge.

				L'algorisme pot aprendre més ràpid utilitzant més experiències a la vegada. Masses experiències augmenta el temps d'entrenament.
Memòria mínima	400	100	1.8 (149, 267)	La diferència entre aquesta puntuació episòdica mitjana i la de referència pot estar influenciada per la aleatorietat, ja que el numero d'episodis és semblant i només ha aconseguit menys puntuació. Quan es comença a entrenar el model no hauria d'afectar el resultat.
Memòria màxima	5000	2000	3.3 (115, 380)	Degut a que la quantitat màxima d'estats en els que pot estar l'agent és de 100, una límit de memòria a 2000 és suficient per encabir totes les possibles experiències i per tant torna molt més eficient l'algorisme.
Mida quadrícula + posició de l'agent + posició de l'objectiu	10x10 + 2,2 + 7,8	15x15 + 5,4 + 12,14	2.19 (160, 350)	Aquest resultat s'ha obtingut utilitzant una TRE de 0.00005 ja que utilitzant el valor per defecte no arriba a convergir. Per quadrícules més grans es requereix una TRE encara més petita.
Recompensa positiva	100	10	0.144 (125, 18)	Poca puntuació, aconseguida per aleatorietat, no per aprenentatge. Mantenir un valor alt ajuda a que l'agent diferenciï millor l'objectiu.
Recompensa negativa	-10	-1	1.6 (156, 249)	Pujar la recompensa negativa de -10 a -1 no causa una

				diferència significativa. El número d'episodis és pròxim al valor de referència i la puntuació pot estar relacionada amb la aleatorietat. Tot i així, el valor de -10 ajuda a l'agent a entendre millor que s'ha de mantenir dins la quadrícula.
Neurones a la capa oculta	12	8	2.94 (109, 321)	Utilitzar menys neurones a la capa oculta és significativament millor que el valor per defecte. S'han necessitat pocs episodis i s'ha aconseguit molta puntuació, fet que fa que sigui més eficient que la tasca amb valors per defecte.
Funció activació capa oculta	ReLu	Sigmoid	0.144 (173, 25)	El model no arriba a convergir. La funció sigmoid no és útil en aquest tipus de tasca.
Funció activació capa sortida	Lineal	ReLu	2.37 (127, 301)	El valor resultant és molt semblant al valor de referència. En aquest cas concret, és innecessari utilitzar una altra funció que no sigui lineal.
Optimitzador	Adam	Stochastic Gradient Descent (SGD)	0.042 (403, 17)	Utilitzant SGD, el model no arriba a convergir. S'han necessitat molts episodis amb pocs passos a cada un. Els 17 punts aconseguits corresponen a la aleatorietat d'accions del principi de la tasca, no a un aprenentatge.

Taula 9 Taula on s'analitzen alguns híper-paràmetres i configuracions utilitzades al prototip 2. Font: elaboració pròpia.

6.3 Projecte final: Snake

6.3.1 Disseny del projecte

El projecte final té per objectiu la implementació d'un agent que aprèn a jugar a l'Snake per després acabar analitzant el rendiment al realitzar la tasca. Com en els dos prototips anteriors, el projecte final també segueix la mateixa estructura de disseny, que és la següent:

1. Com en els prototips, l'agent és l'aprenent i el que pren les decisions, que en aquesta tasca és una serp que va creixent quan agafa menjar de l'entorn. L'objectiu és el menjar que funciona de manera semi-estàtica ja que es mou quan l'agent se l'arriba a menjar i es manté estàtic la resta del temps. La tasca és que l'agent aconsegueixi el màxim de puntuació per partida.
2. Per defecte, l'entorn està format per una quadrícula de 10x10 amb l'agent situat a la meitat de la quadrícula i l'objectiu situat en alguna cel·la aleatòria en que no hi estigui situat l'agent. Si l'agent arriba a l'objectiu, aconsegueix una recompensa positiva de 100 mentre que si mor, és a dir, surt dels marges de la quadrícula o xoca amb si mateix, la recompensa que rep és de -10. Rep una recompensa de 0 en la resta de casos.
3. El projecte final és significativament més complicat que el prototip anterior, per tant, l'estat també es compon de més variables, per així assegurar un correcte aprenentatge per l'agent. Les variables són: obstacle a l'esquerra del cap, obstacle a la dreta del cap, obstacle al davant del cap, posició de l'agent en l'eix X, posició de l'agent en l'eix Y, direcció de l'agent en l'eix X, direcció de l'agent en l'eix Y i finalment la llargada de la cua de la serp.
4. Les accions canvien respecte els prototips. L'agent pot anar, de forma relativa a si mateix, cap a l'esquerra, capa a la dreta i cap endavant.
5. L'agent utilitza com a estratègia la ε-greedy amb reducció lineal de la variable èpsilon.
6. El projecte final té un funcionament semblant al prototip 2. Quan l'agent realitza una acció es compleix un pas. El següent episodi comença quan l'agent mor, però ja no es reinicia quan l'agent arriba a l'objectiu. La tasca finalitza quan la variable èpsilon és menor que 0.

Snake

L'Snake (Banerjee, 2017) és un videojoc arcade, actualment considerat un clàssic, que està basat en un joc creat l'any 1976 anomenat Blockade que partia de la premissa en què dos jugadors dirigien cada un una serp en un mateix espai amb la intenció de bloquejar l'altre. El videojoc Snake va aparèixer al 1997 al model de telèfon 6110 de Nokia però va esdevenir viral l'any 2000 amb l'aparició del Nokia 3310, que també portava el joc instal·lat de base i era una versió millorada de la que apareixia al model 6110. A partir d'aquí van anar sortint diferents versions amb més o menys encert amb la intenció d'engrandir aquest joc.

Les regles de l'Snake són molt senzilles; el jugador controla una serp que es mou per una quadrícula amb la intenció d'aconseguir menjar. Si aconsegueix menjar, la serp es fa més llarga i si xoca contra els marges de la quadrícula o contra si mateixa, mor.

Per el projecte final s'utilitza l'Snake com a base on provar un agent que funciona amb deep reinforcement learning perquè és un joc amb unes regles senzilles però difícil de masteritzar, inclús per una persona. Qualsevol joc amb regles senzilles però difícil de masteritzar valdria, com pot ser el Pong, Atari Breakout, Space invaders, Pac-man, etc... S'ha acabat escollint l'Snake per la facilitat de modelització de l'entorn i perquè és senzill veure quan l'agent compleix la tasca correctament donat que no hi ha gaires dependències, només és la serp i el menjar.

6.3.2 Implementació del projecte

Definició de la configuració i els híper-paràmetres

Híper-paràmetre	Valors per defecte	Descripció
Discount factor - gamma	0.85	Valor que regula el descompte de les recompenses futures.
Epsilon decay rate (Taxa Reducció Èpsilon)	0.00006	Valor amb el que decau ε a cada pas de l'entrenament per l'estratègia ε-greedy.
Batch Size	64	Valor que determina la quantitat d'experiències aleatòries que utilitza l'algorisme cada vegada que realitza un aprenentatge a cada pas.
Memòria mínima	1000	Valor que limita el mínim d'experiències necessàries perquè el model comenci l'aprenentatge.

Memòria màxima	10000	Valor que limita el màxim d'experiències que emmagatzema l'agent.
Grid Size	10x10	Mida de la quadrícula.
Recompensa positiva	100	Valor que retorna l'entorn quan l'agent realitza una bona acció i l'ajuda a aprendre
Recompensa negativa	-10	Valor que retorna l'entorn quan l'agent realitza una mala acció i l'ajuda a aprendre
Neurones capa oculta	12	Quantitat de neurones a la capa oculta del model
Funció activació capa oculta	ReLu	Tipus de funció d'activació de neurones per la capa oculta del model
Funció activació capa sortida	Linear	Tipus de funció d'activació de neurones per la capa de sortida del model
Optimitzador	Adam	Tipus d'optimitzador utilitzat per disminuir el valor dels pesos, basat en Gradient Descent

Taula 10 Taula que mostra la configuració i els híper-paràmetres per l'Snake. Font: elaboració pròpia.

Arquitectura i configuració del model

Els models de xarxa neuronal pel projecte final són semblants als utilitzats en el prototip 2. També estan basats en un model seqüencial format per capes densament connectades de la llibreria Keras, però en comptes de 4 entrades com al prototip, té una primera capa amb 8 entrades, una per cada neurona. Conté una capa oculta amb 12 neurones i una capa de sortida amb 3 neurones. La capa oculta utilitza la funció d'activació *ReLu* mentre que la capa de sortida utilitza la funció *linear*. El cost es calcula amb l'error quadrat mitjà i l'optimitzador utilitzat és l'*Adam*.

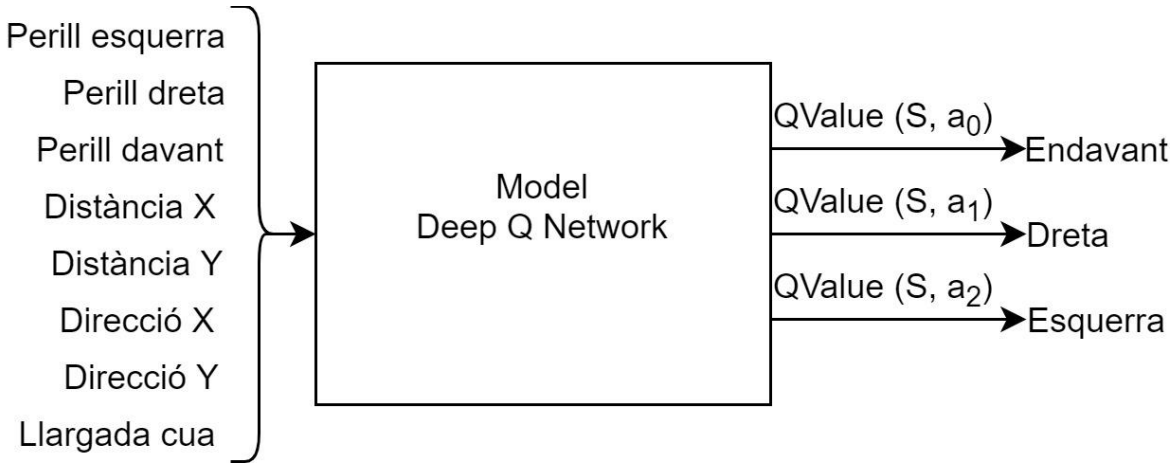


Figura 6.16 Esquema que mostra la informació d'entrada i de sortida pel model de deep Q Network utilitzat a l'algorisme de l'Snake. Font: elaboració pròpia.

Com mostra la figura 6.16, el model rep 8 entrades i extreu 3 valors de qualitat (Q), un per cada acció disponible corresponent a les direccions possibles relatives a l'agent, endavant, esquerra i dreta. En relació amb les dades d'entrada, els tres primers valors informen a l'agent de si té un perill pròxim en les direccions en les que es pot dirigir i serveixen perquè aprengui a sobreviure dins la quadrícula i no mori contra si mateix. Els dos valors d'entrada següents corresponen a la distància entre l'agent i l'objectiu en els eixos X i Y que serveixen perquè tingui en compte on s'ha de dirigir. Els següents valors corresponen a la direcció que porta l'agent en els dos eixos que serveixen perquè l'agent sàpiga cap a on s'està movent i finalment la llargada de la serp, que fa que les altres dades d'entrada acabin depenent d'aquesta última quan l'agent està en punt avançats de la tasca. L'interessant d'aquestes dades d'entrada és la dependència entre elles que és el que fa que l'agent tingui una descripció prou detallada de l'entorn.

Implementació en codi

El projecte final és més complex que els prototips, per tant s'han utilitzat diverses classes necessàries perquè l'algorisme sigui funcional. Les classes amb tots els mètodes i variables estan a l'esquema UML de la figura 6.17.

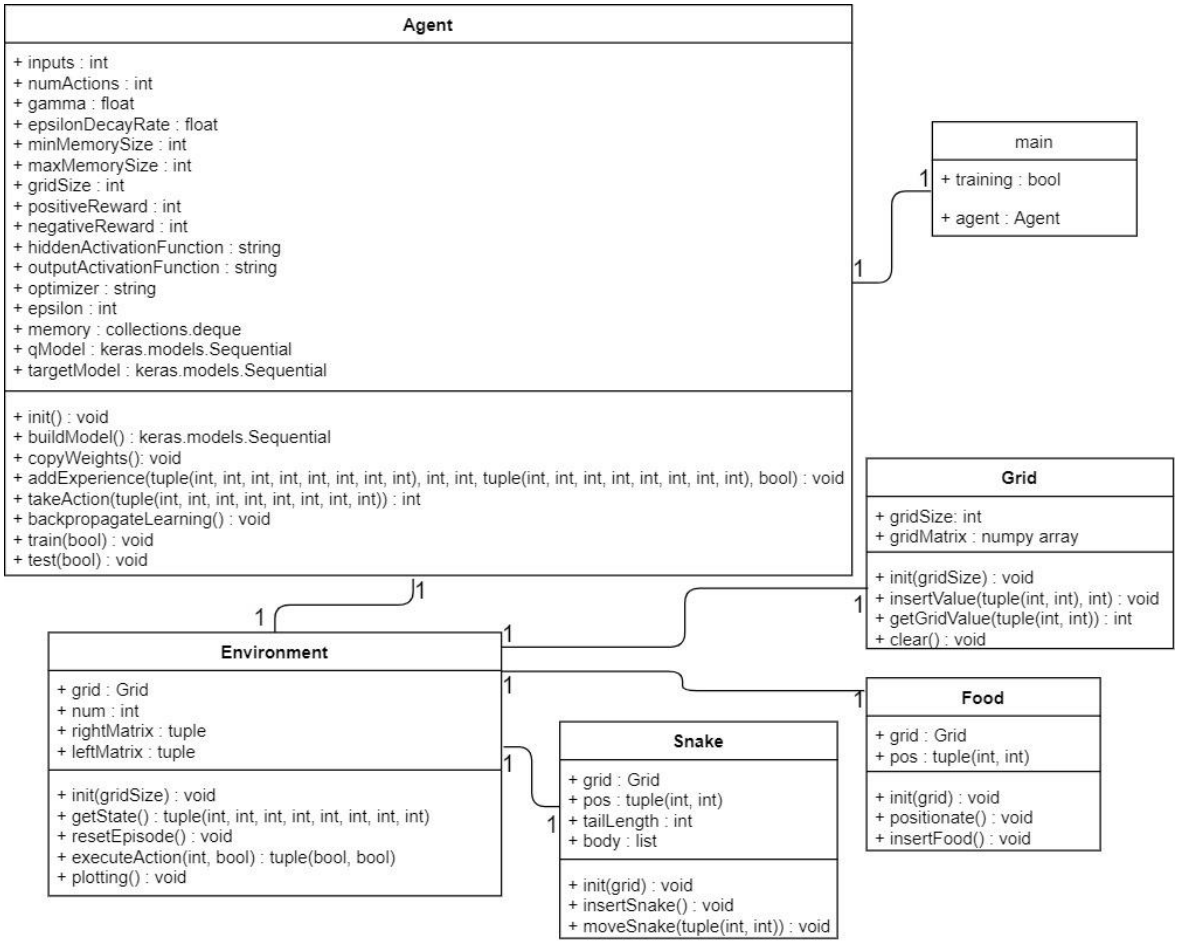


Figura 6.17 Diagrama de classes UML del projecte Snake. Font: elaboració pròpia.

La classe principal és la classe Agent. Aquesta és molt semblant a la classe del mateix nom del prototip 2 però sense els mètodes getState, resetEpisode i executeAction que, al ser més extensos, pertanyen a una altra classe. Tots els altres mètodes, explicats a l'apartat d'implementació del prototip 2, són exactament iguals excepte el mètode train, que se li afegeix la creació d'una instància d'Environment i la possibilitat de guardar el model en format .h5. L'últim mètode, anomenat test, s'encarrega de carregar el model entrenat, l'executa a l'entorn ,prèviament creat, per comprovar el rendiment de l'agent quan el model ja és òptim i extreu un gif per visualitzar el resultat.

La classe Environment estableix el comportament de l'entorn i té els següents mètodes:

init: mètode que per la creació de la classe Environment. Crea una una instància de la classe Grid de mida gridSize, crida el mètode resetEpisode i estableix dos matrius de rotació per el mètode executeAction.
getState: aquest mètode avalua si hi ha algun obstacle davant, a dreta o a esquerra de la serp, observant si a la matriu que representa l'entorn hi ha algun obstacle a la posició adjacent de les tres que ha de comprovar, i calcula la distància entre el cap de la serp i el menjar. Finalment retorna vuit variables en forma de tupla mostrades a l'apartat anterior de disseny del model.
resetEpisode: aquest mètode inicia un nou episodi. Neteja la matriu que serveix per fer els càlculs sobre l'entorn, inicialitza la direcció per defecte i l'aplica també a la direcció anterior i per últim crea una serp i menjar i els afegeix a la matriu d'entorn.
executeAction: el mètode comença per avaluar el número d'acció que és; si la variable action passada per paràmetre és 0, la direcció passa a ser la darrera direcció utilitzada. Si és 1, es fa el producte vectorial entre la matriu de rotació per rotar la direcció 90 graus a la dreta, en canvi si és 2, utilitza l'altra matriu de rotació per rotar la direcció 90 graus a l'esquerra. El següent pas és cridar el mètode moveSnake per moure una posició la serp i establir la direcció actual com darrera direcció. Llavors comprova si la serp ha sortit de la quadrícula o ha xocat contra si mateixa establint la variable isDead com a certa. Si ha mort, el mètode acaba i retorna una tupla de dos valors booleans, un per si ha arribat al menjar i l'altre per si ha mort. Si no ha mort, comprova si la posició del cap de la serp és la mateixa que la del menjar per així posicionar altra cop el menjar a l'entorn, augmentar la cua de la serp en una unitat i establir la variable booleana targeted com a certa. Finalment, neteja la matriu de l'entorn, col·loca la serp i el menjar a les noves coordenades i retorna la tupla de valors booleans anterior actualitzada.
plotting: aquesta funció crea un gràfic de la llibreria matplotlib, que mostra la serp i el menjar en una quadrícula, i el guarda en format .jpg en una carpeta.

La classe Grid gestiona la representació de l'entorn i té els següents mètodes:
init: permet crear una instància de la classe Grid. Té una variable que emmagatzema la mida de la quadrícula i una altra que és una matriu que emmagatzema les posicions del menjar i de la serp.

insertValue: col·loca un valor dins la posició de la matriu escollida per paràmetre.
getGridValue: comprova si una posició passada com a paràmetre està fora de la quadícula. Si és així, retorna -1 i en cas contrari retorna el valor que hi ha dins la matriu en aquesta posició.
clear: reinicia la matriu de l'entorn.

La classe Snake gestiona la posició de la serp i té els següents mètodes:
init: permet crear una instància de la classe Snake. Guarda en una variable la instància de la Grid que s'està utilitzant, la posició del cap de la serp, la llargada de la cua de la serp, que per defecte és 3, i una llista de tuples amb les posicions en X i Y de les parts del cos de la serp.
insertSnake: insereix un 2 a la matriu de l'entorn a la posició del cap i un 1 a les posicions del cos de la serp.
moveSnake: afegeix la posició actual del cap a la llista body i escurça aquesta llista per actualitzar-la a la llargada correcta del cos. Per últim, estableix de nou la posició del cap sumant les coordenades X i Y de la direcció, passada com a paràmetre, a les de la posició de la serp.

La classe Food gestiona la posició del menjar i té els següents mètodes:
init: permet crear una instància de la classe Food. Emmagatzema en una variable la instància de la Grid que s'està utilitzant i la posició del menjar. També crida el mètode positionate per canviar la posició del menjar.
positionate: estableix un valor aleatori per cada coordenada de la posició del menjar amb valors entre 0 i la mida de la quadrícula. Si a la posició escollida hi ha alguna part de la serp, es torna a cridar aquesta funció de forma recursiva.
insertFood: insereix un 4 a la matriu de l'entorn a la posició del menjar.

Finalment hi ha el mètode main que crea una instància d'Agent i executa el mètode train o test segons el valor de la variable booleana training.

Com al prototip 2, el pseudocodi de la figura 6.18 és en el que s'ha basat la implementació per els mètodes train i backpropagateLearning. La diferència està

en els valors dels híper-paràmetres com la capacitat de la memòria d'experiències, el valor gamma i la taxa de reducció d'èpsilon, entre d'altres.

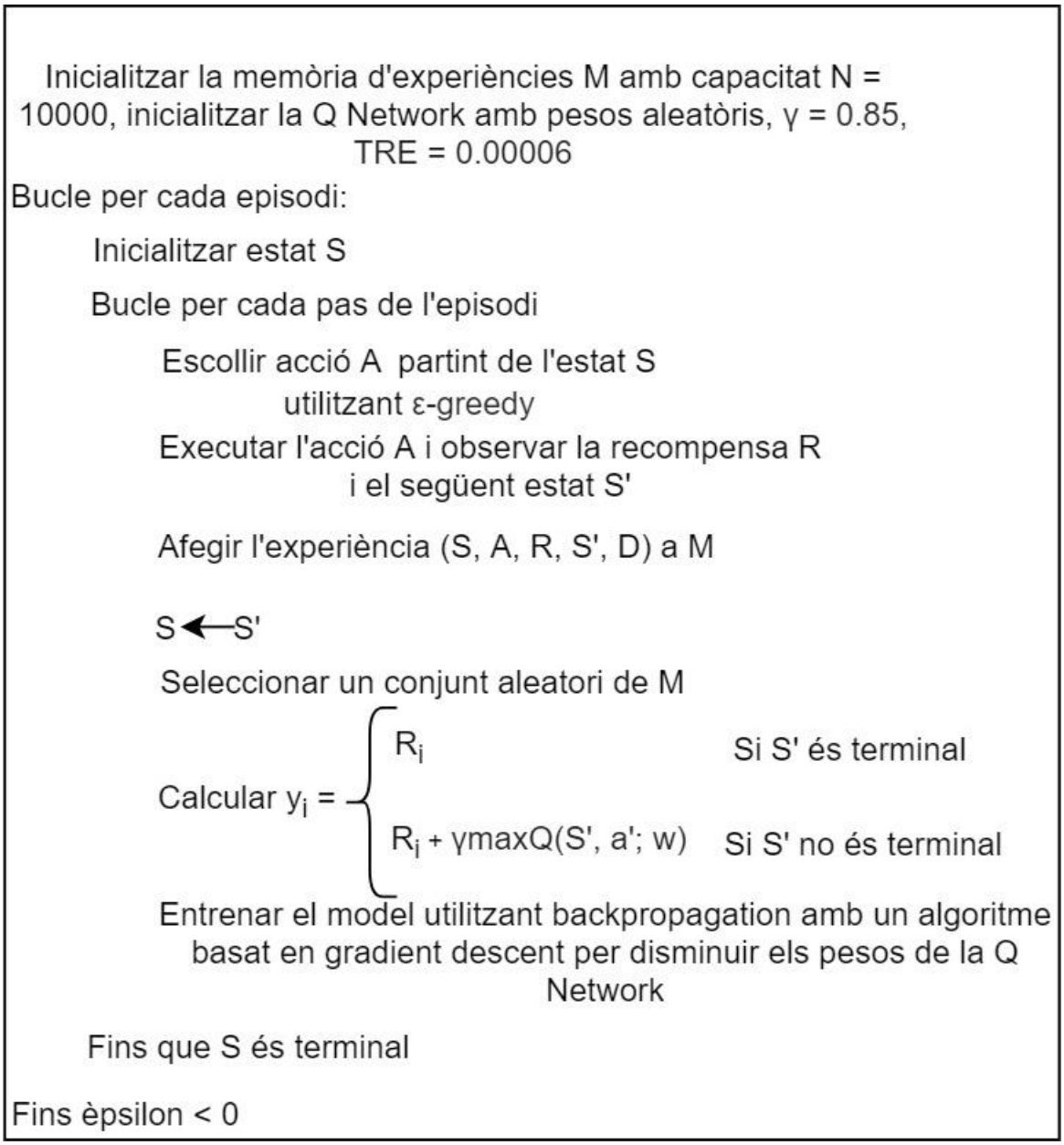


Figura 6.18 Pseudocodi de l'algorisme Deep Q-Learning per l'Snake. Font: elaboració pròpia basat en **(Mnih et al, 2013)**.

6.3.3 Anàlisi de resultats

Els resultats de l'Snake estan dividits en 5 gràfiques, diverses taules que modifiquen la configuració de l'algorisme i una taula que compara els resultats de testeig del model amb una persona. Han estat obtinguts d'un sol entrenament després de comprovar que els diferents entrenaments obtenien uns resultats semblants.

L'entrenament per els següents resultats s'ha realitzat utilitzant la configuració i els híper-paràmetres inicials mostrats a l'apartat anterior. Cal destacar que la serp comença amb una llargada de cua de 3 perquè tingui una progressió inicial més ràpida, tot i així, aquest valor no s'afegeix a la puntuació que aconsegueix l'agent.

Gràfiques de resultats

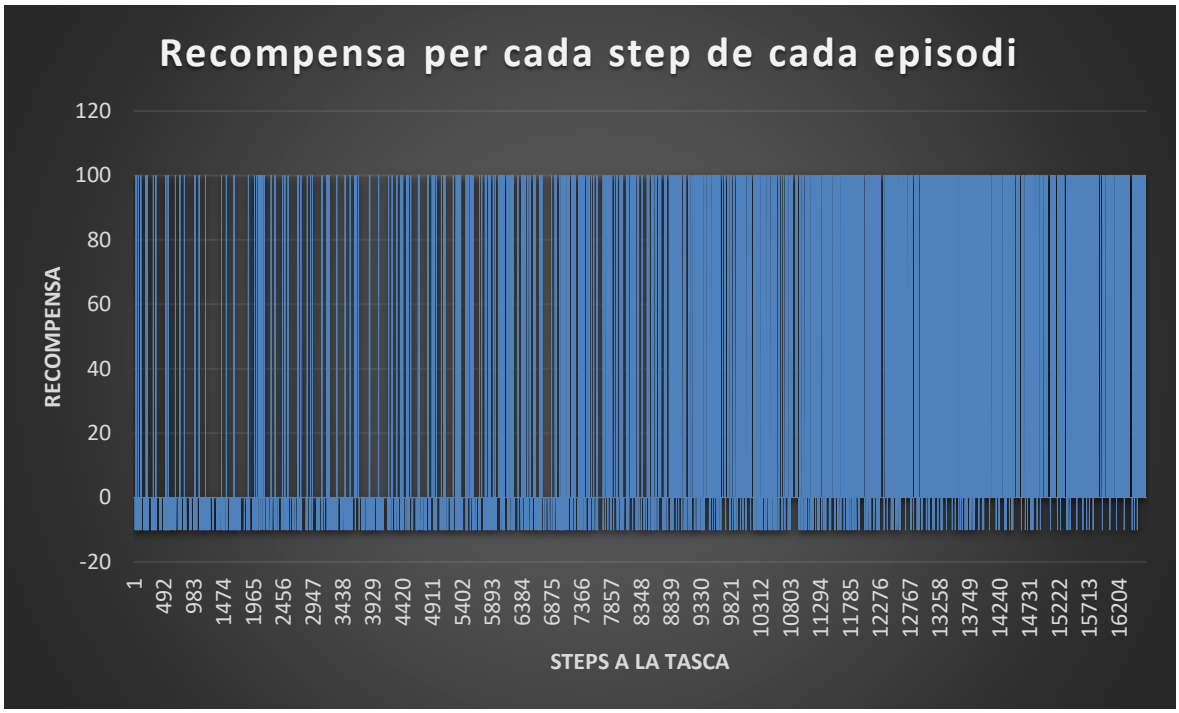


Figura 6.19 Gràfic sobre la recompensa que rep l'agent per cada pas de cada episodi. Font: elaboració pròpia.

A la gràfica de la figura 6.19 es pot veure la recompensa obtinguda a cada pas que hi ha hagut durant l'entrenament de la tasca. Cada punt situat a l'altura 100 és un pas on l'agent ha arribat a l'objectiu, alguns punts l'agent ha aconseguit una recompensa negativa de -10 per haver sortit dels marges de la quadrícula, mentre que la resta de punts, tot i que no s'aprecia per la gran quantitat de passos que hi ha hagut durant la tasca, la recompensa és de 0. Com als prototips, aquesta gràfica també mostra un canvi significatiu de recompensa per cada pas de cada episodi a partir d'un cert moment. Tot i que hi ha una gran quantitat de passos a la tasca, és poden apreciar signes d'aprenentatge entre la primera i la segona meitat de la gràfica abans dels 7000 passos.

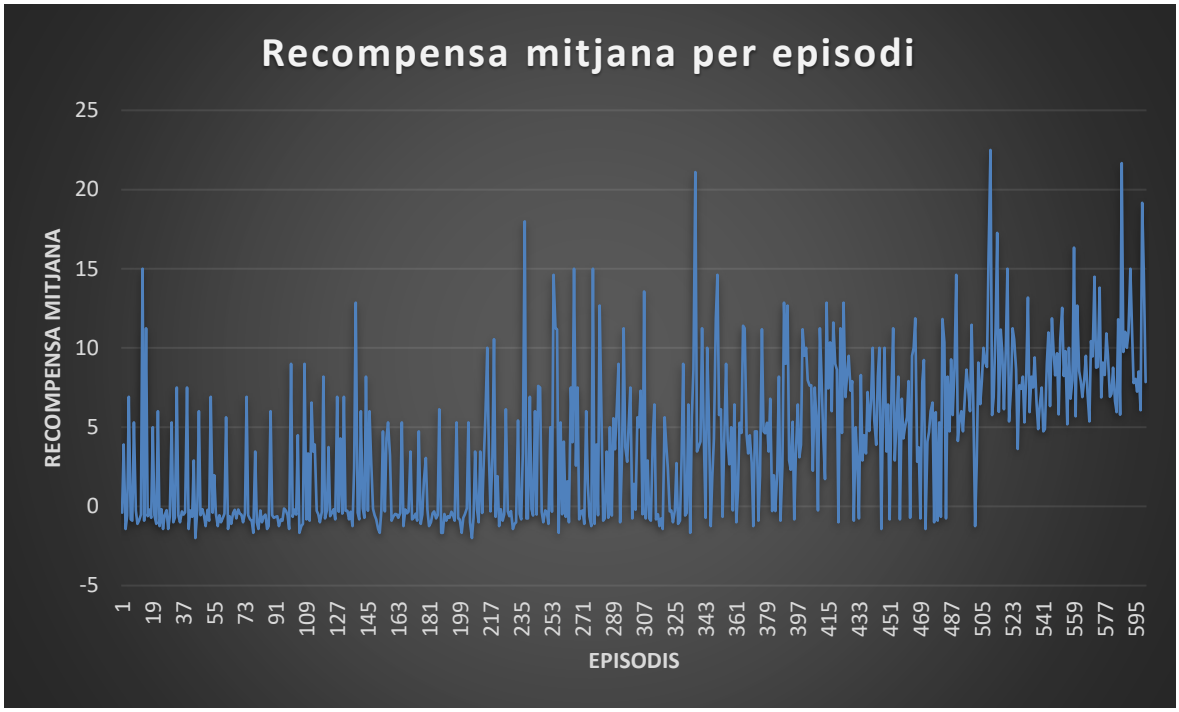


Figura 6.20 Gràfic sobre la recompensa mitjana per episodi. Font: elaboració pròpia.

El gràfic 6.20 també mostra uns resultats bastant clars. S'aprecia com en quasi tota la gràfica hi ha molts episodis amb una recompensa mitjana pròxima o per sota del zero i alguns episodis amb valors per sobre de 5, de 10 i inclús de 15. A partir d'un episodi pròxim al 487, tendeixen a augmentar tots els valors mitjans situant-los en molts casos amb un mínim per sobre de 5 i màxims per sobre de 15 i 20. És en aquest punt on el model ja ha convergit i només s'està acabant d'optimitzar, fent que ja sigui útil per jugar a l'Snake de forma òptima.

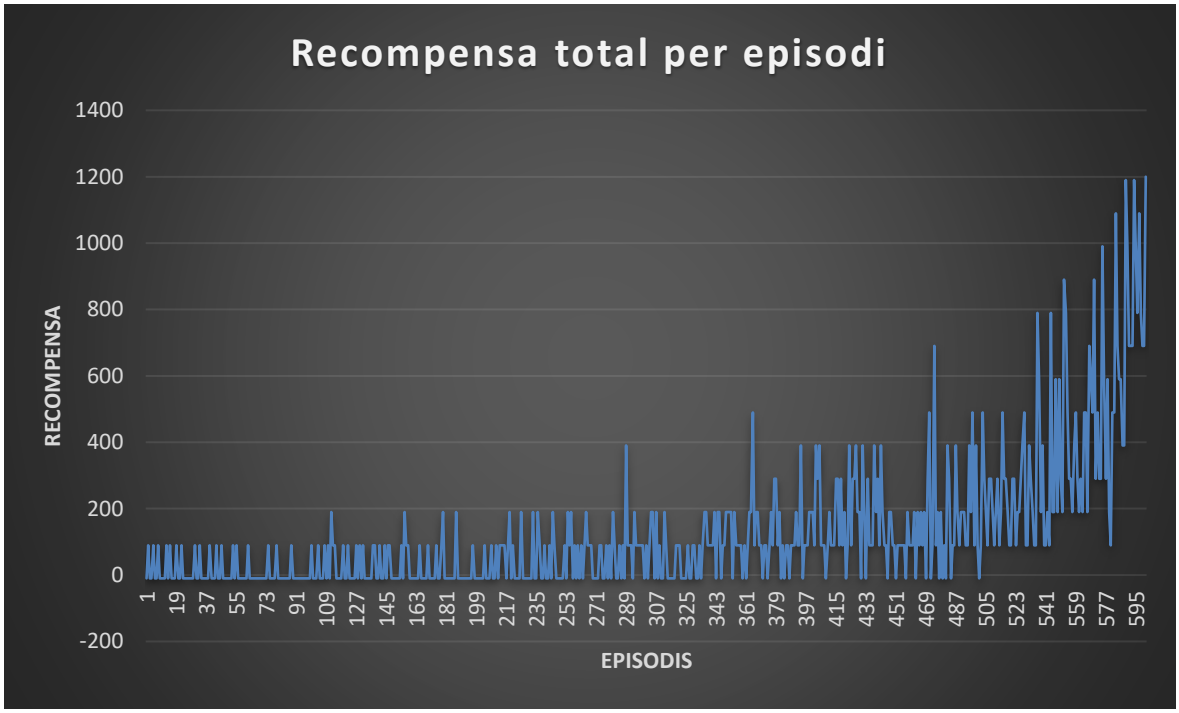


Figura 6.21 Gràfic sobre la recompensa total per episodi. Font: elaboració pròpia.

En el gràfic 6.21 s'aprecia el progrés de l'agent a l'últim terç de la tasca on aconsegueix més recompensa positiva. En la majoria d'episodis l'agent aconsegueix una recompensa de 0 i en molts casos de -10 i fins a l'episodi 670, quan aconsegueix recompensa sol ser de 100 o 200. Al final es poden visualitzar episodis amb recompenses de 400 i fins a 1200 que correspon a que la serp aconsegueixi menjar entre 4 i 12 vegades en un mateix episodi. La variable èpsilon, que regula l'estratègia èpsilon-greedy, no és tant necessària que estigui amb valors pròxims al 0, però quan hi arriba, és quan es poden notar els canvis més significatius en l'aprenentatge ja que l'agent pot explotar més l'entorn.

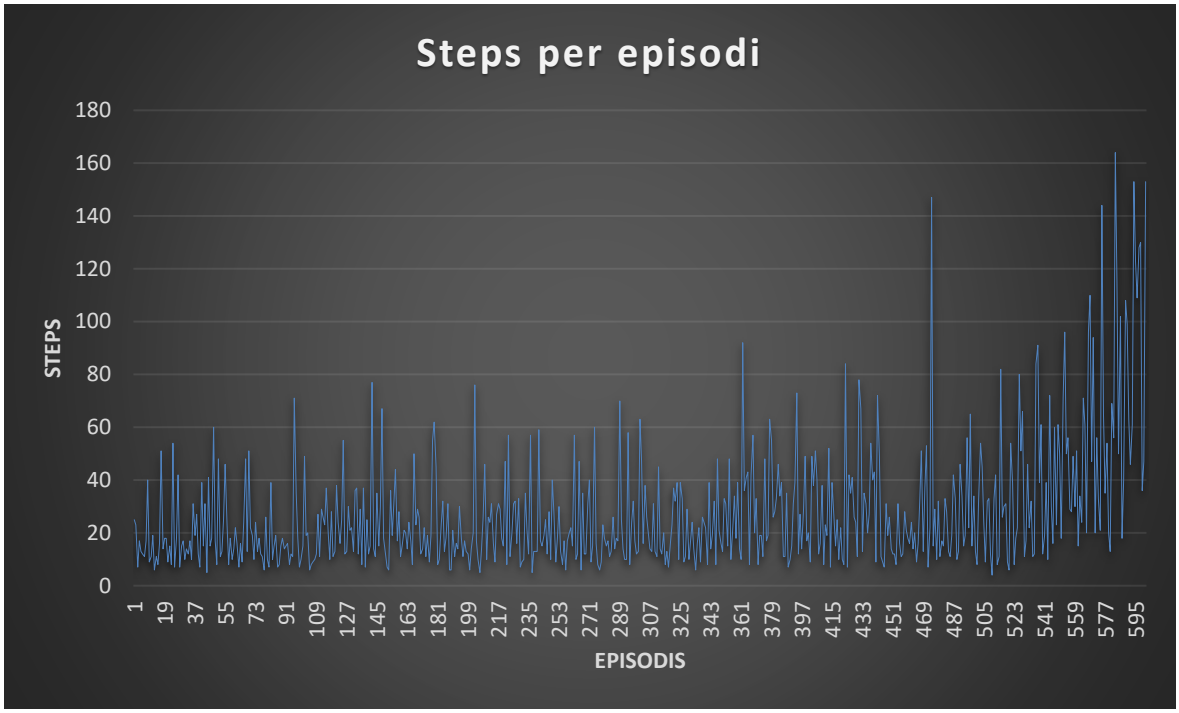


Figura 6.22 Gràfic sobre la quantitat de passos per episodi. Font: elaboració pròpia.

Tal com mostra la gràfica 6.22, la quantitat de passos per episodi augmenta considerablement cap al final de la tasca fins arribar a 164 passos a l'episodi 584 i 153 a l'últim episodi de la tasca, quan la variable èpsilon ja ha arribat a 0 i l'agent està explotant completament el coneixement que disposa de l'entorn. Encara que la quantitat de passos sembla poca, és suficient perquè l'agent aprengui a resoldre la tasca amb èxit. A diferència del prototip 2, en el projecte final, que l'agent sobrevisqui molts passos ja no indica que ho està fent bé ja que el rendiment es mesura amb la puntuació que aconsegueix la serp quan aconsegueix menjar.

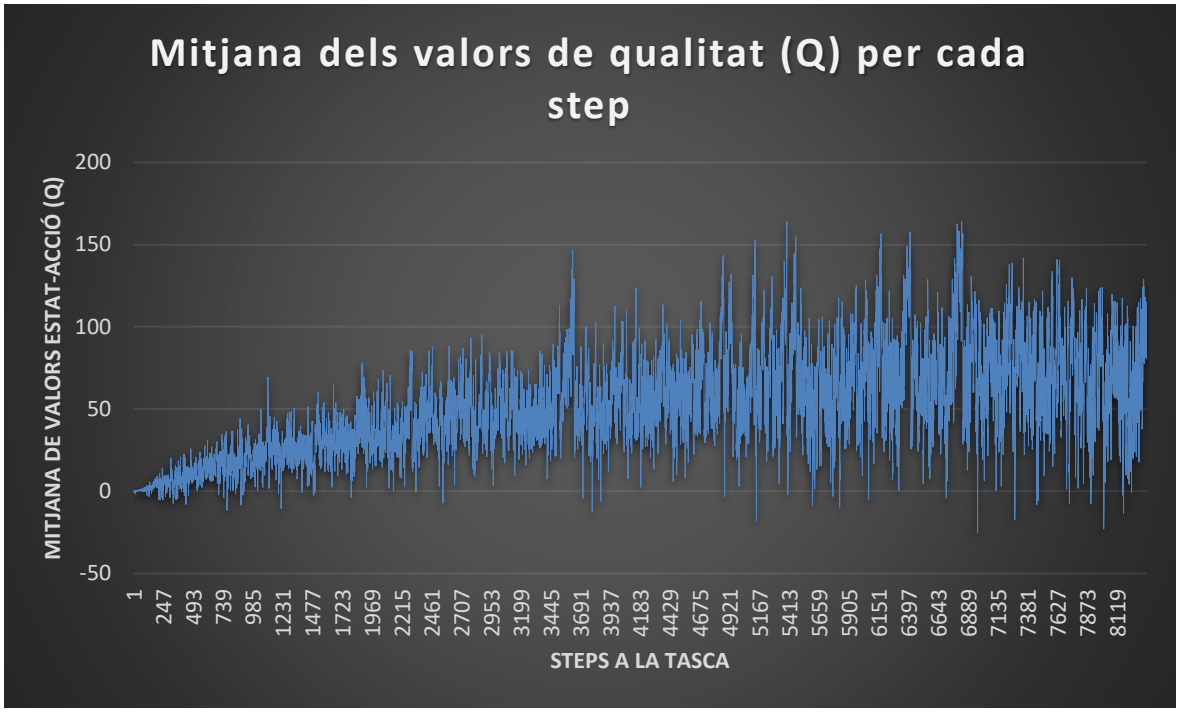


Figura 6.23 Gràfic sobre la mitjana entre els valors de qualitat (Q) de cada acció per cada pas de la tasca. Font: elaboració pròpia.

La gràfica de la figura 6.23 és potser la gràfica més rellevant de totes les exposades. Mostra una estimació de la recompensa descomptada que l'agent pot obtenir seguint l'estratègia utilitzada a qualsevol estat. Es pot veure com la mitjana dels valors que ha extret el model per cada predicció van en augment durant la primera meitat de la tasca. Al principi, el valor mitjà és pròxim al 0 i conforme avança l'entrenament el valor augmenta fins a estabilitzar-se pròxim al 50 tot i que, durant tota la tasca, la gràfica es manté turbulenta. Els valors de qualitat permeten veure evidències de l'aprenentatge de l'agent però són valors massa inestables per considerar-ho com a única prova dels resultats obtinguts.

Modificació d'híper-paràmetres

De la mateixa manera que en el prototip 2, aquí també s'ha utilitzat el paràmetre de la puntuació episòdica mitjana (PEM) com a valor aproximat per comparar els entrenaments amb els diferents híper-paràmetres i configuracions de l'algorisme. S'han aconseguit tres valors PEM resultat de realitzar l'entrenament amb els híper-paràmetres i configuració per defecte, que són segons l'ordre d'execució 1.35 (601, 811), 0.78 (600, 472), 1.17 (585, 686) que produeixen un valor mitjà de 1.1. Els números entre parèntesis corresponen, el primer al número d'episodis i el segon a la puntuació. Aquests valors es consideren els valors de referència per realitzar la

comparació amb altres híper-paràmetres i configuracions mostrades a les següents taules.

Híper-paràmetre o configuració	Gamma
Valor per defecte	0.85
Valor modificat 1	0.5
Puntuació episòdica mitjana 1	0.57 (663, 382)
Valor modificat 2	0.99
Puntuació episòdica mitjana 2	0.6 (612, 365)
Notes	Els valors obtinguts per les dos modificacions són significativament més baixos que els valors de referència. En aquesta tasca s'evidencia que gamma ha de tenir un valor pròxim al 0 però sense ser massa pròxim ja que un valor molt baix o molt alt redueix el rendiment del model.

Taula 11 Taula que mostra els resultats de les modificacions pel paràmetre gamma. Font: elaboració pròpia.

Híper-paràmetre o configuració	Taxa de reducció d'èpsilon (TRE)
Valor per defecte	0.00006
Valor modificat 1	0.0001
Puntuació episòdica mitjana 1	0.7 (364, 258)
Valor modificat 2	0.0005
Puntuació episòdica mitjana 2	0.275 (80, 22)
Notes	Les dos modificacions són massa justes per la tasca de l'Snake però es nota que comencen a realitzar un aprenentatge sense arribar a optimitzar el model. En aquest cas, el valor per defecte assegura que el model s'entrena completament sense arribar a ser el valor més òptim.

Taula 12 Taula que mostra els resultats de les modificacions pel paràmetre de la taxa de reducció d'èpsilon. Font: elaboració pròpia.

Híper-paràmetre o configuració	Batch size
Valor per defecte	64
Valor modificat 1	32
Puntuació episòdica mitjana 1	0.58 (618, 359)
Valor modificat 2	128
Puntuació episòdica mitjana 2	1.05 (609, 642)
Notes	Utilitzar 32 experiències durant l'entrenament del model porta a uns resultats inferiors als esperats mentre que utilitzar més experiències de les proposades per defecte, ofereix un rendiment acceptable similar als valors de referència.

Taula 13 Taula que mostra els resultats de les modificacions pel paràmetre batch size. Font: elaboració pròpia.

Híper-paràmetre o configuració	Memòria mínima
Valor per defecte	1000
Valor modificat 1	400
Puntuació episòdica mitjana 1	1.02 (576, 589)
Valor modificat 2	50
Puntuació episòdica mitjana 2	0.78 (583, 457)
Notes	Els resultats per les dos modificacions tendeixen a ser pròxims als valors de referència, fet que porta a pensar que canviar aquest paràmetre no aporta canvis significatius. El millor valor continua sent el de la configuració per defecte.

Taula 14 Taula que mostra els resultats de les modificacions pel paràmetre de la memòria mínima. Font: elaboració pròpia.

Híper-paràmetre o configuració	Memòria màxima
Valor per defecte	10000
Valor modificat 1	5000
Puntuació episòdica mitjana 1	1.04 (584, 611)
Valor modificat 2	2000
Puntuació episòdica mitjana 2	1.12 (617, 692)
Notes	Les dos modificacions obtenen bons resultats que fan pensar que la quantitat màxima de memòria per aquesta tasca no és massa rellevant, mentre es mantingui en aquests valors.

Taula 15 Taula que mostra els resultats de les modificacions pel paràmetre de la memòria màxima. Font: elaboració pròpia.

Híper-paràmetre o configuració	Mida quadrícula
Valor per defecte	10x10
Valor modificat 1	15x15
Puntuació episòdica mitjana 1	0.6 (320, 193)
Valor modificat 2	20x20
Puntuació episòdica mitjana 2	0.66 (233, 155)
Notes	En els dos casos l'agent aconsegueix un resultat decent pel fet que altres paràmetres com la TRE estan preparats per una quadrícula de mida 10x10 en el seu valor per defecte. El resultat mostra pocs episodis, que significa que es desenvolupen més passos per episodi que en els valors per defecte. Per les dos modificacions, el model no arriba a convergir però, la puntuació aconseguida, es l'evidència que amb una TRE que proporcioni els suficients passos, ho arribaria a fer.

Taula 16 Taula que mostra els resultats de les modificacions pel paràmetre de la mida de la quadrícula. Font: elaboració pròpia.

Híper-paràmetre o configuració	Recompensa positiva
Valor per defecte	100
Valor modificat 1	10

Puntuació episòdica mitjana 1	1.32 (597, 789)
Valor modificat 2	1
Puntuació episòdica mitjana 2	0.76 (578, 441)
Notes	Una recompensa positiva de 10 obté un valor similar amb el valor de referència més elevat. En canvi una recompensa de 1, aconsegueix un valor bastant baix ja que no ajuda a que l'objectiu destaqui més durant l'aprenentatge.

Taula 17 Taula que mostra els resultats de les modificacions pel paràmetre de la recompensa positiva. Font: elaboració pròpia.

Híper-paràmetre o configuració	Recompensa negativa
Valor per defecte	-10
Valor modificat 1	-1
Puntuació episòdica mitjana 1	1.14 (590, 676)
Valor modificat 2	-100
Puntuació episòdica mitjana 2	0.79 (584, 464)
Notes	El resultat per la primera modificació és prou bona degut a que al principi de l'entrenament succeeixen moltes males accions que fa que un valor de -1 sigui suficient perquè l'agent entengui aquestes accions. Per la raó anterior, un valor de -100 és massa gran i redueix el rendiment del model.

Taula 18 Taula que mostra els resultats de les modificacions pel paràmetre de la recompensa negativa. Font: elaboració pròpia.

Híper-paràmetre o configuració	Neurones a la capa oculta
Valor per defecte	12
Valor modificat 1	8
Puntuació episòdica mitjana 1	0.76 (614, 470)
Valor modificat 2	15
Puntuació episòdica mitjana 2	1.46 (598, 875)
Notes	Mentre utilitzar menys neurones a la capa oculta disminueix el rendiment del model, utilitzar-ne més genera la situació contrària, el model rendeix molt millor que utilitzant el valor per defecte ja que aconsegueix molta puntuació per una quantitat d'episodis semblant a la de referència.

Taula 19 Taula que mostra els resultats de les modificacions pel paràmetre de les neurones de la capa oculta. Font: elaboració pròpia.

Híper-paràmetre o configuració	Funció activació capa oculta
Valor per defecte	ReLu
Valor modificat 1	Elu
Puntuació episòdica mitjana 1	1.48 (593, 880)
Valor modificat 2	Linear
Puntuació episòdica mitjana 2	0.3 (652, 251)

Notes	El resultat aconseguit utilitzant la funció ELU és clarament superior als valors de referència i a més ha aconseguit molta puntuació per pocs episodis. Això fa que sigui prioritari utilitzar aquesta funció en detriment de la ReLu ja que en soluciona els problemes derivats. En canvi, utilitzar la funció lineal disminueix de manera significativa el rendiment del model ja que no és un tipus de funció preparada per ser utilitzada a la capa oculta.

Taula 20 Taula que mostra els resultats de les modificacions per la configuració de la funció d'activació de la capa oculta. Font: elaboració pròpia.

Híper-paràmetre o configuració	Funció activació capa de sortida
Valor per defecte	Linear
Valor modificat 1	ReLu
Puntuació episòdica mitjana 1	0.94 (649, 612)
Valor modificat 2	Tangent hiperbòlica
Puntuació episòdica mitjana 2	0.7 (555, 393)
Notes	Tot i que cap dels dos resultats és molt baix, les dos modificacions no estan preparades com a funcions per la capa de sortida i se'n ressent el rendiment del model.

Taula 21 Taula que mostra els resultats de les modificacions per la configuració de la funció d'activació de la capa de sortida. Font: elaboració pròpia.

Híper-paràmetre o configuració	Optimitzador
Valor per defecte	Adam
Valor modificat 1	Stochastic Gradient Descent (SGD)
Puntuació episòdica mitjana 1	1.27 (642, 816)
Valor modificat 2	Adagrad
Puntuació episòdica mitjana 2	0.26 (571, 150)
Notes	En el paper de Deepmind utilitzat com a referència (Mnih et al, 2013), utilitzen SGD com a optimitzador amb bons resultats. En aquest treball, no s'ha utilitzat SGD per defecte, però l'ús també ofereix bons resultats, semblants als millors resultats de referència. Per el plantejament d'aquesta tasca, utilitzar Adagrad assegura la divergència del model.

Taula 22 Taula que mostra els resultats de les modificacions per la configuració de l'optimitzador del model. Font: elaboració pròpia.

Després d'aconseguir els resultats anteriors, s'ha intentat entrenar de nou l'agent utilitzant la millor configuració possible. S'ha decidit deixar el valor per defecte si el resultat de la puntuació episòdica mitjana de referència més gran (1.35) supera a la puntuació episòdica mitjana de les dos modificacions. S'ha usat algun dels valors

de les modificacions per la resta de casos. La següent taula mostra finalment quina és aquesta configuració:

Híper-paràmetre o configuració	**Valor amb el millor resultat**
Gamma	0.85
Taxa de reducció d'èpsilon	0.00006
Batch Size	64
Memòria mínima	1000
Memòria màxima	10000
Mida quadrícula	10x10
Recompensa positiva	100
Recompensa negativa	-10
Neurones a la capa oculta	15
Funció activació capa oculta	ELU
Funció activació capa sortida	Linear
Optimitzador	Adam

Taula 23 Taula que mostra la millor configuració del model i dels híper-paràmetres basat en els resultats anteriors. Font: elaboració pròpia.

Utilitzant la configuració anterior, l'agent aconsegueix durant l'entrenament una puntuació episòdica mitjana de 1.48 amb 618 episodis realitzats i una puntuació total de 918. Tal com s'espera, el resultat és superior al millor valor de referència. Només s'ha canviat la funció d'activació i les neurones de la capa oculta però és suficient per ser considerada una millor configuració que la utilitzada per defecte.

Resultats de la fase de testeig

Per fer una bona valoració dels resultats obtinguts per l'agent s'ha considerat de comparar-los amb el rendiment d'una persona, en unes condicions de joc semblants. Per jugar s'ha utilitzat el joc Snake que està integrat a la interfaç del buscador Google, ja que és suficient per la comparació entre l'agent i el jugador. Aquesta implementació té una petita diferència en la mida de la quadrícula que és de 10x9 en comptes de 10x10 però que no ha de suposar un problema per avaluar-ne el rendiment. La comparació es realitza amb 10 partides seguides.

Nº partida	**Puntuació persona**	**Puntuació agent: configuració per defecte (passos)**	**Puntuació agent: millor configuració (passos)**
1	28	14 (194)	13 (143)
2	18	10 (91)	8 (75)
3	19	12 (126)	15 (129)
4	17	9 (93)	16 (127)
5	31	8 (59)	13 (116)
6	25	9 (93)	12 (104)
7	21	11 (173)	14 (135)

8	29	11 (130)	13 (138)
9	19	19 (238)	11 (81)
10	23	12 (205)	9 (85)

Taula 24 Taula que mostra el rendiment de l'agent i d'un jugador comparant la puntuació aconseguida durant 10 partides utilitzant la configuració i paràmetres per defecte. Font: elaboració pròpia.

La taula 12 mostra, aparentment, resultats similars entre la configuració per defecte i la millor configuració. Observant la mitjana de puntuació per les deu partides la mitjana de la millor configuració és de 12.4 mentre que la configuració per defecte ha aconseguit una mitjana de 11.5, que indica que la millor configuració és lleugerament més òptima. El jugador obté, a nivell general, molta més puntuació a cada partida que l'agent amb les dos configuracions, amb una mitjana de 23. Cal destacar que l'agent amb la configuració per defecte obté un valor màxim de 19, que supera dos de les puntuacions del jugador. Tot i així, no és suficient per afirmar que l'agent està al mateix nivell que el jugador.

La quantitat de passos que sol realitzar l'agent és baixa en comparació amb la puntuació aconseguida, sense arribar a ser del tot òptim en els moviments. A més, s'ha observat que quan la serp mor, és per xocar-se contra si mateixa i no per haver sortit de la quadrícula, ja que confinar-se dins l'entorn és un dels punts bàsics de la seva supervivència.

Optimització en el videojoc Snake

L'Snake és un videojoc amb final. En una quadrícula de 10x10 i, contant que la serp comença ocupant només una cel·la, pot menjar com a màxim 99 vegades. A nivell general, hi ha tres formes de jugar a l'Snake; la primera és l'enfocament directe. Tal com es veu a la figura 6.24, un jugador actuant de forma directa intenta dirigir-se cap al menjar amb el mínim de moviments possibles, cosa que assegura uns moviments òptims. La segona forma és l'enfocament algorítmic. Aquesta manera intenta crear un moviment programable fàcilment amb la intenció d'assegurar la victòria en el joc, sacrificant els moviments. Un exemple d'això es veu a la figura 6.24 en vermell. El tercer enfocament és el de crear un mix entre els dos anteriors agafant-ne els millors avantatges. Un jugador que utilitza la tercera forma comença amb un enfocament directe i com més creix la serp, va canviant a un enfocament algorítmic.

Utilitzar tota la partida la primera forma acaba condemnant la serp a morir xocant contra si mateixa. La segona forma assegura la victòria però és completament

ineficient i té una manca de repte, al seguir tota l'estona els mateixos moviments. La tercera forma comença sent molt òptima i conforme creix la serp, ja no es poden realitzar moviments tan directes i s'ha de començar a planejar per evitar que xoqui contra si mateixa, assegurant la optimització al llarg de tota la partida.

Visualitzant els moviments de l'agent entrenat, es pot veure que realitza un enfocament directe pràcticament durant tota la partida i només quan és prou llarga comença a planejar mínimament. Justament és en aquest punt en què la serp mor al xocar-se contra si mateixa ja que no arriba a ser capaç de planejar suficient els pròxims moviments tenint en compte la llargada que té. Aquest resultat indica una mínima optimització, com es buscava al realitzar aquest treball, sense arribar a assolir la capacitat de planificació que tenim els humans que fa que juguem de forma òptima.

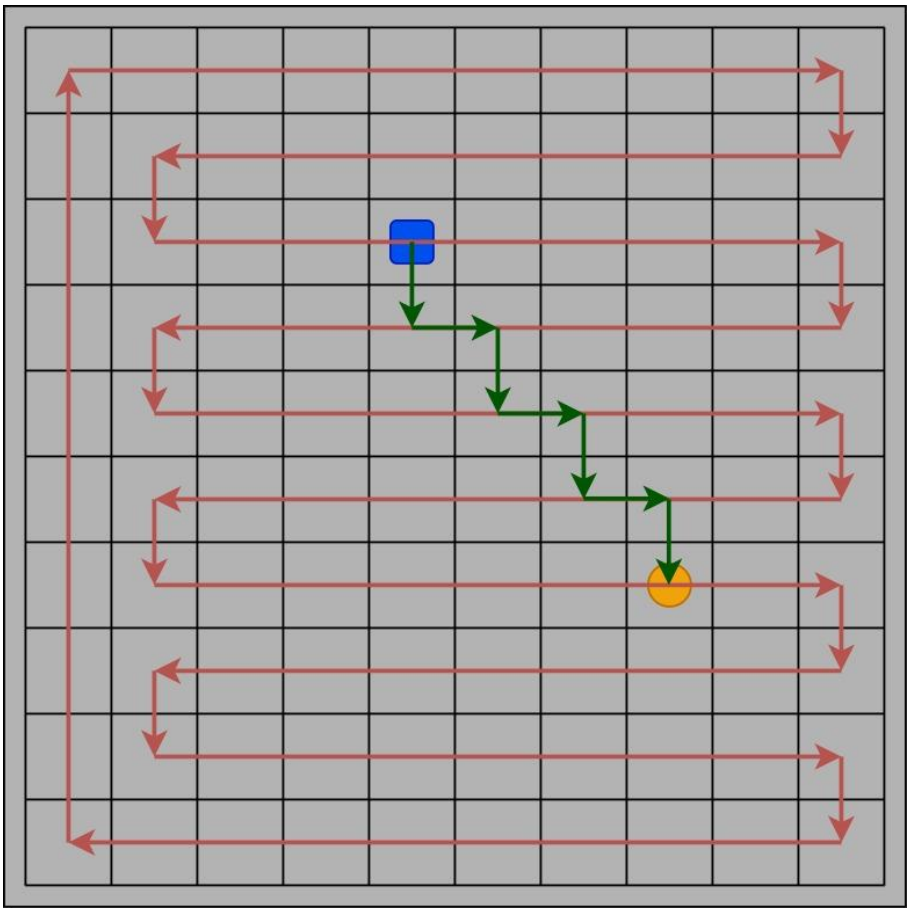


Figura 6.24 Imatge que mostra el recorregut òptim (verd) de l'agent (blau) per anar cap al menjar (taronja). També es mostra el camí menys òptim que assegura la victòria a l'Snake (vermell). Font: elaboració pròpia.

7 Conclusions

Aquest treball tenia un objectiu principal basat en el disseny, el prototipatge i l'anàlisi d'un agent que aprèn a jugar a l'Snake de forma mínimament òptima, utilitzant algorismes de Machine learning. Després s'havien proposat dos objectius secundaris, un que per estudiar l'anàlisi de l'aprenentatge de l'agent i l'altre per avaluar la conveniència dels algorismes utilitzats en els videojocs i per contrastar-los amb altres tipus d'algorismes d'IA.

Finalitzat el treball, es pot afirmar que s'ha complert el principal objectiu proposat i el primer objectiu secundari. S'ha aconseguit dissenyar, crear i analitzar dos prototips que demostren el funcionament essencial de dos algorismes de Machine learning: Q-Learning i Deep Q-Learning. A més s'ha aconseguit realitzar la implementació i l'anàlisi d'un agent que aprèn a jugar a l'Snake d'una manera mínimament òptima utilitzant també Deep Q-Learning.

Després de comprovar el funcionament dels dos algorismes, és possible imaginar quina pot ser la seva adaptació en un videojoc, complementant l'experiència del jugador.

Per un costat, el Q-Learning és un algorisme molt senzill que no pot realitzar tasques gaire complexes. Aquest algorisme pot realitzar tasques que tinguin un número finit d'estats possibles. Pot arribar a ser útil en videojocs, regulant sistemes senzills de dificultat dinàmica si els estats han estat prèviament definits. Per exemple, en un joc de tipus roguelike pot definir la quantitat d'enemics i algunes de les seves característiques, basant-se en l'actuació del jugador.

Per l'altre costat, el Deep Q-Learning és un algorisme molt més complex, potent i amb moltes més possibilitats que el Q-Learning. Aquest algorisme no té limitació d'estats, sinó que pot realitzar tasques amb un espai d'estats infinit. En un videojoc, segons l'arquitectura que tingui el model de xarxa neuronal, l'algorisme pot arribar a controlar un agent en qualsevol situació com ho podria fer una persona. Per exemple, en un joc de tipus shooter, l'algorisme pot controlar un bot aliat o enemic o inclús un conjunt d'ells com si fos una persona.

L'exemple anterior seria una tasca que no es pot realitzar amb algorismes molt utilitzats en els videojocs a dia d'avui, com les màquines d'estat o els arbres de decisió, tots dos IA basada en comportament, ja que el comportament és massa complex i conté un espai d'estats que pot ser infinit. Els algorismes de deep reinforcement learning no substitueixen aquests algorismes de IA més senzills sinó que els poden complementar en tasques més complexes. De forma genèrica, com més simple és la tasca, més òptims es tornen els algorismes amb una IA més de comportament i més ineficients els algorismes d'aprenentatge per reforç, en canvi, com més complexa és, més òptims es tornen els algorismes de deep reinforcement learning i més inútils els algorismes amb una IA de comportament.

Amb aquestes conclusions queda assolit l'últim objectiu secundari que faltava.

A part dels objectius proposats, a nivell personal, aquest treball també ha servit per realitzar un aprenentatge sobre diversos temes diferents. Ha sigut la primera presa de contacte amb el concepte de deep reinforcement learning i de la implementació de xarxes neuronals. També ha sigut una introducció al llenguatge Python, molt útil per programar models de deep learning i analitzar dades.

Futures investigacions

Aquest treball ha servit per aconseguir el coneixement base necessari sobre deep learning i aprenentatge per reforç. Tot i així, existeixen algorismes molt més complexos, potents i interessants que els que s'han presentat aquí. El següent pas, és obtenir un coneixement més ample de l'aprenentatge per reforç investigant aquests algorismes més evolucionats com, per exemple, els algorismes basats en Policy Gradient. L'objectiu final, és acabar reflectint tot aquest coneixement en un joc on la IA provoqui una experiència nova i diferent al jugador, de la que estem acostumats i traspassi la frontera del que es pot aconseguir amb algorismes que no estiguin basats en Machine Learning.

8 Referencies

Agrawal, A. (29 / 9 / 2017). *Medium.* Recollit de Loss Functions and Optimization Algorithms: https://medium.com/data-science-group-iitr/loss-functions-and-optimization-algorithms-demystified-bb92daff331c

Allamy, H., & Khan, R. (1 / 1 / 2014). *ResearchGate.* Recollit de ResearchGate: https://www.researchgate.net/publication/300719195_Methods_to_Avoid_Over-Fitting_and_Under-Fitting_in_Supervised_Machine_Learning_Comparative_Study

Asensio, I. (8 / 11 / 2019). *MasterD*. Recollit de MasterD: https://unity.com/products/unity-platform

Banerjee, P. (27 / 2 / 2017). *Digit*. Recollit de A brief history of Snake: https://www.digit.in/features/mobile-phones/a-brief-history-of-snake-33913.html

Beuke, F. (2021). *Github*. Recollit de Githut 2.0: https://madnight.github.io/githut/#/pull_requests/2021/1

Beunza et al. (2020). *Manual Práctico de Inteligencia Artificial En Entornos Sanitarios.* Elsevier Health Sciences.

Birch, C. (2 / 12 / 2010). *Game Internals*. Recollit de Game Internals: https://gameinternals.com/understanding-pac-man-ghost-behavior

Bishop, C. M. (1995). Neural Networks for Pattern Recognition. En C. M. Bishop, *Neural Networks for Pattern Recognition* (pág. 332). Nova York: Oxford University Press, Inc. Obtenido de https://dl.acm.org/doi/10.5555/525960#cited-by-sec

Chapman, J., & Lechner, M. (17 / 6 / 2020). *Github.* Recollit de Github: https://github.com/keras-team/keras-io/blob/master/examples/rl/deep_q_network_breakout.py

Clevert et al. (22 de 2 de 2016). *arXiv.* Obtenido de arXiv: https://arxiv.org/abs/1511.07289

DeepMind. (8 / 2 / 2020). *Nature.* Recollit de Nature: https://www.nature.com/articles/s41586-019-1724-z

Duchi et al. (1 de 7 de 2011). *ResearchGate.* Obtenido de ResearchGate: https://www.researchgate.net/publication/220320677_Adaptive_Subgradient_Methods_for_Online_Learning_and_Stochastic_Optimization

EA. (2021). *EA*. Recollit de EA: https://www.ea.com/es-es/games/need-for-speed

Escolano et al. (2003). *Inteligencia artificial: modelos, técnicas y áreas de aplicación.* Editorial Paraninfo.

Flórez López, R., & Fernández Fernández, J. (2008). *Las Redes Neuronales Artificiales.* Netbiblo.

Fundació Enciclopèdia Catalana. (2021). *Gran enciclopèdia catalana*. Recollit de Gran enciclopèdia catalana: https://www.enciclopedia.cat/ec-gec-0185759.xml

Goodfellow et al. (2016). *Deep Learning.* MIT Press.

Hunter. (2007). *Matplotlib*. Obtenido de Matplotlib: https://matplotlib.org/stable/users/history.html

Imageio. (2020). *Imageio*. Recollit de Imageio: https://imageio.readthedocs.io/en/stable/index.html

Jordan, J. (4 / 7 / 2017). *Jeremy Jordan*. Recollit de Jeremy Jordan: https://www.jeremyjordan.me/neural-networks-activation-functions/

Keras. (4 / 2 / 2021). *Keras*. Recollit de https://keras.io/about/

Keras. (2021). *Keras*. Recollit de Keras Losses: https://keras.io/api/losses/

Kobran, D. (2020). *AI Wiki.* Recollit de AI Wiki: https://docs.paperspace.com/machine-learning/wiki/weights-and-biases

Laboratorio Nacional de Calidad del Software de INTECO. (3 de 2009). *Scribd.* Obtenido de Scribd: https://es.scribd.com/document/62931905/Guia-de-Ingenieria-Del-Software

Lapan, M. (2018). *Deep Reinforcement Learning Hands-On: Apply modern RL methods, with deep Q-networks, value iteration, policy gradients, TRPO, AlphaGo Zero and more.* Packt Publishing Ltd.

Marquez, J. (3 / 2 / 2021). *Hipertextual*. Recollit de Hipertextual: https://hipertextual.com/2021/05/todos-los-juegos-de-grand-theft-auto-antes-de-gta-6

Microsoft. (2019). *Visual Studio Microsoft*. Obtenido de Visual Studio Microsoft: https://visualstudio.microsoft.com/es/vs/

Microsoft. (2020). *Visual Studio Code*. Obtenido de Visual Studio Code: https://code.visualstudio.com/docs

Microsoft. (2021). *Microsoft Docs*. Obtenido de Microsoft Docs: https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/

Mnih et al. (19 / 12 / 2013). *ArXiv.* Recollit de ArXiv: https://arxiv.org/abs/1312.5602

Mnih et al. (25 / 2 / 2015). *Nature.* Recollit de Nature: https://www.nature.com/articles/nature14236

N. Yannakakis, G., & Togelius, J. (2018). *Artificial Intelligence and Games.* Springer.

Nintendo. (2020). *Nintendo*. Recollit de Nintendo: https://mario.nintendo.com/es/history/

Numpy. (2021). *Numpy*. Recollit de Numpy: https://numpy.org/about/

Openai. (13 / 12 / 2019). *ArXiv.* Recollit de ArXiv: https://arxiv.org/abs/1912.06680

Openai. (11 / 2 / 2020). *ArXiv.* Recollit de ArXiv: https://arxiv.org/abs/1909.07528

Oppy, G., & Dowe, D. (2020). *Stanford Encyclopedia of Philosophy.* Recollit de Stanford Encyclopedia of Philosophy: https://plato.stanford.edu/archives/win2020/entries/turing-test/

P. Kingma, D., & Ba, J. L. (22 / 12 / 2014). *Cornell University.* Recollit de Cornell University: https://arxiv.org/abs/1412.6980

Pai, A. (2 / 8 / 2020). *Machine Learning Works.* Recollit de Gradient Descent: https://www.machinelearningworks.com/tutorials/gradient-descent

Pai, A. (26 / 10 / 2020). *Machine Learning Works.* Recollit de Mean Squared Error Cost Function: https://www.machinelearningworks.com/tutorials/mean-squared-error-cost-function

Pandey, P. (18 / 5 / 2019). *TowardsDataScience.* Recollit de TowardsDataScience: https://towardsdatascience.com/understanding-the-mathematics-behind-gradient-descent-dde5dc9be06e

Pannu, A. (10 / 4 / 2015). *Semantic Scholar.* Recollit de Semantic Scholar: https://www.semanticscholar.org/paper/Artificial-Intelligence-and-its-Application-in-Pannu-Student/9a4d9a755134e612854db1897c03adb3983413df

Pérez, A. (16 / 8 / 2016). *OBS Business School*. Recollit de OBS Business School: https://www.obsbusiness.school/blog/caracteristicas-y-fases-del-modelo-incremental

Python Software Foundation. (28 de 5 de 2021). *Python.org*. Obtenido de General Python FAQ: https://docs.python.org/3/faq/general.html

Ramírez, F. (19 / 8 / 2020). *Think Big Empresas*. Recollit de Think Big Empresas: https://empresas.blogthinkbig.com/breve-historia-de-la-ia-en-los-videojuegos/

Real Academia Española. (2021). *Diccionario de la lengua española*. Obtenido de Diccionario de la lengua española: https://dle.rae.es/inteligencia#2DxmhCT

Rumelhart et al. (9 / 10 / 1986). *Semantic Scholar.* Recollit de Semantic Scholar: https://www.semanticscholar.org/paper/Learning-representations-by-back-propagating-errors-Rumelhart-Hinton/052b1d8ce63b07fec3de9dbb583772d860b7c769

S. Sutton, R., & G. Barto, A. (2018). *Reinforcement Learning: An Introduction second edition.* Cambridge: The MIT Press.

Schulman et al. (28 / 8 / 2017). *arXiv.* Recollit de arXiv: https://arxiv.org/abs/1707.06347

Serrano, A. G. (2016). *Inteligencia Artificial. Fundamentos, práctica y aplicaciones.* Madrid: RC Libros.

Sharma, S. (6 de 9 de 2017). *TowardsDataScience.* Obtenido de TowardsDataScience: https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6

Silver et al. (27 / 1 / 2016). *Nature.* Recollit de Nature: https://www.nature.com/articles/nature16961

Tensorflow. (26 / 1 / 2021). *Tensorflow*. Recollit de Tensorflow: https://www.tensorflow.org/about

Tesauro, G. (3 / 1995). *ACM Digital Library home.* Recollit de ACM Digital Library home: https://doi.org/10.1145/203330.203343

Unity. (2019). *Unity*. Obtenido de Unity: https://unity.com/products/unity-platform

Weng, L. (19 / 2 / 2018). *Lil'Log*. Recollit de Lil'Log: https://lilianweng.github.io/lil-log/2018/02/19/a-long-peek-into-reinforcement-learning.html

TREBALL FINAL DE GRAU

Annexes

Implementació d'una IA basada en Machine Learning

Adrià Aumedes Floreta
Tutora: Ester Bernadó Mansilla
Grau en Disseny i Producció de Videojocs

CURS 2020-21

TecnoCampus
Escola Superior Politècnica

Centre adscrit a la

upf. Universitat Pompeu Fabra *Barcelona*

Annex 1: Codi font

Annex 2: Models de xarxa neuronal

Models de xarxa neurona de la millor configuració per l'Snake i de la configuració per defecte, en format .h5. Per carregar els models, s'ha de posar la variable training = False i posar la ruta a la primera línia de codi de la funció test. Fent això, el codi crea una imatge de cada pas i la guarda a la mateixa carpeta on hi ha el codi. Amb aquestes imatges, quan es finalitza la partida de testeig, es crea automàticament un vídeo en format de gif.

Annex 3: Vídeos per visualitzar el funcionament del projecte de l'Snake

Tres vídeos que mostren el funcionament de l'agent per l'Snake. Un vídeo que mostra l'agent en els 10 primers episodis de l'entrenament, un altre que mostra el testeig de l'agent pel model amb la millor configuració i un tercer que mostra el testeig de l'agent pel model amb la configuració per defecte.

Annex 4: Llibreries i software

S'han utilitzat les següents llibreries i software per el desenvolupament d'aquest treball:

Visual Studio: Software de programació per C# desenvolupat per Microsoft. Versió 2019. Obtingut de: https://visualstudio.microsoft.com/es/downloads/

Python: Llenguatge de programació desenvolupat per la Python Software Fundation. Versió 3.8.5 Obtingut de: https://www.python.org/downloads/

Numpy: Llibreria de Python de codi obert. Versió 1.19.2. Obtingut de: https://numpy.org/install/

Tensorflow: Llibreria de Python desenvolupada per Google. Versió 2.3.0. Obtingut de: https://www.tensorflow.org/install

Keras: Llibreria de Python utilitzada amb Tensorflow. Versió 2.4.3. Obtingut de: instal·lada juntament amb Tensorflow.

Matplotlib: Llibreria de Python utilitzada per crear gràfics. Versió 3.3.2. Obtingut de: https://matplotlib.org/stable/users/installing.html

Imageio: Llibreria de Python per crear vídeos en format gif a partir d'imatges. Versió 2.9.0. Obtingut de: https://imageio.readthedocs.io/en/stable/installation.html

Draw.io: Eina online de creació de diagrames i altres formes de dibuix. Obtingut de: https://app.diagrams.net/