Universitat de Girona
Escola Politècnica Superior

Treball final de grau

Estudi: Grau en Disseny i Desenvolupament de Videojocs

Títol: Creació d'una eina de debug de videojocs

Document: Memòria

Alumne: Adrià Puntunet Linares

Tutors:Gustavo Patow i Fabio Lorella
Departament: Informàtica, Matemàtica Aplicada i Estadística
Àrea:Llenguatges i Sistemes Informàtics (LSI)

Convocatòria (mes/any) Setembre 2019

Capítol 1 Introducció

En l'actualitat el món dels videojocs presenta una característica molt interessant i buscada pels jugadors: un gran rang d'opcions, decisions i controls que el jugador pot executar. Aquest està encantat, ja que podrà atacar els diferents problemes que es trobi durant el joc de formes variades i cada una més interessant que l'anterior.

Però aquesta característica també comporta diferents problemes. El nombre de problemes i errors que trobarem dins del joc seran més elevats, ja que serà molt difícil poder provar els diferents controls disponibles per a cada escenari del joc.

Per exemple el Fallout 4, es tracta d'un joc de rol d'acció situat en un entorn postapocalíptic de món obert.

El jugador assumeix el control d'un personatge anomenat "Únic Supervivent", que surt d'una estasi criogènica de llarga durada a Vault 111, un refugi soterrat nuclear. Aquest s'aventura per buscar el seu fill desaparegut. El jugador explorarà el món del joc, realitzarà diverses missions, ajudarà a faccions i adquireix punts d'experiència per tal d'augmentar les habilitats del seu personatge.

Fallout 4 va rebre crítiques positives, molts van elogiar la profunditat del món, la llibertat dels jugadors, la quantitat contingut global… Cada jugador podrà tenir una història i un rol dins el joc completament diferent d'un altre, fins i tot el final del joc pot ser diferent. La crítica negativa es va dirigir principalment a aspectes visuals i tècnics del joc.

És per aquesta raó que he volgut crear una eina capaç de debugar nivells de videojocs. Una eina que busqui la solució a un nivell i mentre ho faci, podem veure els diferents errors i problemes que hi ha dins.

Existeix una saga de videojocs anomenada Assassin's Creed que a part de ser famosa pels seus jocs inicials i les mecàniques innovadores que van tenir, també és famós per la quantitat d'errors que hi trobem. Gràcies a una eina com la que vull crear, es podrien reduir aquest tipus de situacions. Entre d'altres podem trobar bugs visuals com el següent:

Abans de començar, però, posarem en context aquest treball. La base d'aquesta eina serà la intel·ligència artificial, la qual és dedicada al desenvolupament d'algorismes que permet a una màquina prendre decisions intel·ligents o, si més no, comportar-se com si tingués una intel·ligència semblant a la humana.

1.1 Motivacions

Per norma general, en un treball, recerca, estudi, etc., hi ha dos tipus de motivacions amagades al darrere: les personals i les necessitats professionals. Les personals solen estar més encarades en les ganes que té un mateix de treballar o investigar en un tema en concret. En aquest cas aquesta motivació neix de l'admiració. Últimament han aparegut moltes millores relacionades amb la intel·ligència artificial relacionades amb els jocs, entre elles la creació de l'AlphaZero, un programa capaç de guanyar a les millors intel·ligències artificials creades fins al moment en els jocs d'escacs, shogi i Go (les quals ja eren capaces de guanyar als millors jugadors del món). És clar que la intel·ligència artificial és una eina molt potent i a més a més ningú sap on són els límits realment. És per aquesta raó que he decidit endinsar-me en aquest món i saber tot el que s'amaga darrere.

La creació de nivells de jocs no és una feina fàcil, a més els jugadors cada vegada són més exigents, volen millors nivells i més interessants. És per això que combinar IA amb la creació de videojocs pot estalviar molt temps als creadors de nivells a l'hora de buscar errors i possibles solucions pel nivell en creació.

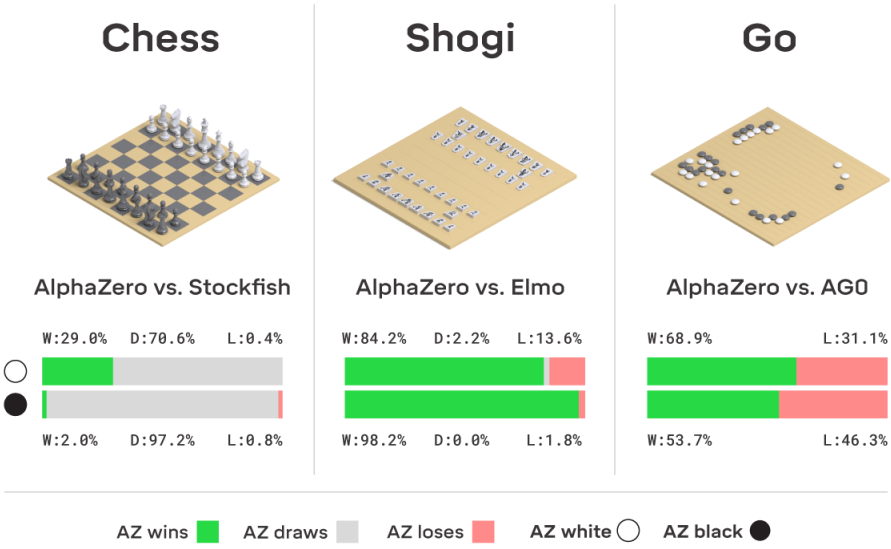


Figura 1.1.1: Resultats de l' AlphaZero contra la millor IA del moment en aquell joc.

1.2 Propòsit

Vist el gran potencial de la IA i la falta d'eines de debug de nivells de videojocs, el propòsit principal d'aquest treball serà desenvolupar una eina que ens permeti debugar nivells gràcies a l'ajuda de la intel·ligència artificial. Aquesta eina es tractarà d'un agent que utilitzarà xarxes neuronals. Haurem d'entrenar la xarxa per les diferents situacions que es pugui trobar en el joc i buscar maximitzar la puntuació que obtenim en cada una. Per aconseguir arribar a totes les situacions el que farem serà aplicar l'aleatorietat, de manera que l'ordinador tindrà el control del joc i anirà fent accions aleatòries per tal d'arribar al màxim d'opcions disponibles. L'aplicació utilitzarà el joc Cuphead per tal de testejar algorismes encara que s'intentarà fer el codi de la manera més general possible per tal que sigui fàcilment modificable per a utilitzar en altres jocs.

1.3 Objectius

L'objectiu principal d'aquest treball serà desenvolupar un programa que aconsegueixi arribar al més lluny possible i superar si pot, un nivell d'un videojoc gràcies a la intel·ligència artificial programada, en el nostre cas el Cuphead.

1. Utilitzar i aprendre el llenguatge de programació Python.
2. Aprendre a utilitzar la llibreria SerpentAI.
3. Estudiar el funcionament del joc escollit.
4. Desenvolupament d'un agent que jugui al joc escollit emprant SerpentAI.
5. Estudi i recerca d'eines d'intel·ligència artificial més adient, incloent-hi algoritmes genètics i xarxes neuronals.
6. Estudi i aprenentatge del funcionament d'una llibreria de xarxes neuronals com el Keras.
7. Desenvolupament d'una intel·ligència artificial adequada.
8. Finalització i arrodoniment de la documentació del treball realitzat.

1.4 Capítols de la memòria

La memòria està estructurada en els següents apartats:

- **Capítol 1. Introducció**. S'expliquen les motivacions, els propòsits i els objectius del projecte. També es presenta un resum dels diferents apartats del projecte.

- **Capítol 2. Estudi de viabilitat**. Es justifica la viabilitat del projecte en terme de recursos tant econòmics, tecnològics i humans.

- **Capítol 3. Metodologia**. S'explica la metodologia seguida per a desenvolupar el projecte.

- **Capítol 4. Planificació**. S'explica el *timing* del projecte, és a dir, el pla de treball, les tasques planificades i el temps estimat per cada etapa.

- **Capítol 5. Marc de treball i conceptes previs**. S'introdueixen els conceptes teòrics necessaris per a comprendre millor la resta de la memòria.

- **Capítol 6. Requisits del sistema**. Es descriuen els requisits funcionals i no funcionals del sistema per assolir els objectius.

- **Capítol 7. Estudis i decisions**. Es descriu el programari, el *framework*, l'IDE i les llibreries utilitzades en aquest projecte.

- **Capítol 8. Anàlisi i disseny del sistema**. S'expliquen les funcionalitats que haurà d'adquirir el sistema a partir de les necessitats dels usuaris, així com les solucions a partir dels diagrames dissenyats.

- **Capítol 9. Implementació i proves**. S'explica com s'ha implementat l'aplicació, amb els detalls més importants, i com s'han anat solucionat els problemes sorgits a partir d'exemples i proves gràfiques.

- **Capítol 10. Resultats**. Es mostren els resultats obtinguts, a partir d'exemples i imatges de l'aplicació.

- **Capítol 11. Conclusions**. S'exposen les conclusions de l'assoliment dels objectius i de l'assoliment dels requisits del projecte, així com una crítica dels resultats obtinguts.

- **Capítol 12. Treball futur**. S'expliquen les possibles ampliacions, millores o treballs futurs que es poden realitzar a partir de l'aplicació obtinguda.

- **Bibliografia**. Llistat de les referències utilitzades per desenvolupar el projecte.

- **Capítol 13. Manual d'usuari i instal·lació**. S'explica com s'ha d'utilitzar i instal·lar l'aplicació en un ordinador.

Capítol 2 Estudi de viabilitat

En aquest capítol es valorarà la viabilitat del projecte des de diferents aspectes com poden ser: els recursos tècnics, els recursos humans, els requisits tecnològics o el cost econòmic.

2.1 Recursos tècnics

Aquest treball s'ha realitzat en dos ordinadors:

- La part de programació i proves bàsiques que no requerien molta potència les he fet amb un MacBook Pro. Aquest té un processador 2,7 GHz Intel Core i5, 8 GB 1867 MHz DDR3 de memòria RAM i una targeta gràfica Intel Iris Graphics 6100 1536 MB. El sistema operatiu és macOS HighSierra 10.13.6.

- Per entrenar la xarxa neuronal ho he fet amb un Windows molt més potent per tal d'estalviar temps. La màquina té un processador Intel Core I5-4670 3.4GHz en un sistema operatiu Windows 7 de 64 bits i una targeta gràfica NVIDIA GeFroce GTX 760. Disposa de dues memòries RAM DDR3 de 4GB a 1866Hz cadascuna.

2.2 Recursos humans

Com la majoria de projectes informàtics, aquest treball requereix tres perfils de persona:

- **Analista**: Serà l'encarregat de definir com haurà de ser l'algorisme, quines funcionalitats i mètodes haurà de tenir i com s'haurà de dur a terme per aconseguir-les. També tindrà la responsabilitat d'estudiar les eines que es necessitaran juntament amb la seva documentació. Finalment s'encarregarà d'elaborar la memòria del projecte.

- **Programador**: Serà l'encarregat de crear l'algorisme de l'agent. A partir de les consignes de l'analista haurà de ser capaç d'entendre i desenvolupar l'aplicació a partir de les eines aportades.

- **Cap del projecte**: Serà l'encarregat de coordinar el projecte seguint la metodologia de treball escollida (Capítol 3), ajustar els terminis de les diferents etapes de desenvolupament i aconseguir un bon funcionament en el treball.

Com que aquest projecte ha estat realitzat per una mateixa persona li han correspost els perfils d'analista i programador. Pel que fa al perfil de cap de projecte l'han dut a terme els tutors del projecte.

2.3 Requisits tecnològics

A part de la màquina Windows (el Mac no pot entrenar la xarxa neuronal a una velocitat acceptable) especificada a l'apartat de 'Recursos tècnics', faran falta alguns altres requisits tecnològics per dur a terme aquest treball. S'ha utilitzat el llenguatge de programació Python pel qual cal tenir el compilador instal·lat a la màquina. Més endavant, a la Secció 7.3, veurem quin *framework* necessitarem per treballar i amb quines llibreries i *addons*.

2.4 Cost econòmic

En aquesta secció definirem el cost econòmic dels recursos tècnics, dels recursos humans i de la tecnologia que hem explicat en els apartats anteriors.

En el cas dels recursos tècnics, com que hem treballat amb un ordinador personal ens estalviarem el cost de comprar-lo, tot i això haurem de tenir en compte l'amortització dels recursos utilitzats. Per calcular-la utilitzarem la següent fórmula:

$$Amortitzacio = (mesosfeina * preurecurs)/36 = (8 * 768{,}77)/36 = 170{,}38€$$

Pel que fa al *software*, tots els programes utilitzats són gratuïts, tret del PyCharm encara que en aquest cas, he utilitzat la versió "*Education*".

En el cas dels recursos humans hem vist que, a banda del cap del projecte, són necessaris altres perfils: l'analista i el programador. En aquest cas, tots dos perfils han estat duts a terme per la mateixa persona, però si haguéssim de calcular el cost per independent, tindríem que:

- Sou de l'analista: 20€/h
- Sou del programador: 10€/h

Feines a realitzar:

- Estudi de les eines.

- Disseny de l'agent a partir del *framework* – Implementació.
- Disseny de la intel·ligència artificial a partir del *framework*. – Implementació.
- Entrenament.
- Documentació.

Per tant, el cost en funció de les tasques a realitzar és el següent:

Tasques	**Perfil**	**Hores**	**Cost/hora**	**Cost total**
Aprenentatge Python	Programador	40	10	400
Estudi Serpent	Analista	76	20	1520
Estudi Intel·ligència artificial	Analista	146	20	2920
Estudi Keras	Analista	53	20	1060
Implementació Agent	Programador	10	10	100
Implementació Intel·ligència Artificial	Programador	106	10	1060
Entrenament	Programador	100*	-	-
Documentació	Analista	156	20	3120
Total Programador		300	10	1560
Total Analista		420	20	8620
Amortització		-	-	170,38
TOTAL				10350,84

2.5 Conclusió de la viabilitat del projecte

Pel que fa al programari necessari s'utilitza programari gratuït i versions d'estudiant dels que no ho són. Per tant el cost és 0.

Pel que fa a la remuneració dels treballadors, en el cas hipotètic d'una contractació real, tampoc seria del tot ajustada envers el que podem veure a la taula anterior, ja que aquest treball moltes de les hores han estat invertides a fer recerca, investigar i provar noves tecnologies de les quals no se'n tenia un coneixement previ. Amb això, la contractació de treballadors sèniors o professionals en el sector no hauria suposat un preu tan elevat en nombre d'hores d'anàlisi, i segurament també menys en programació.

Com a conclusió, com que és un projecte d'estudiant i es disposa dels requisits tecnològics, el treball serà viable.

Capítol 3 Metodologia

Actualment hi ha moltes metodologies de desenvolupament eficients i diferenciables, des de les més antigues com la metodologia *Waterfall* fins a les més modernes tècniques *Agile*. Després d'estudiar diferents metodologies tals com *Spiral*, *Scrum* o *Iterative*, i les esmentades anteriorment, s'ha decidit no fer-ne servir en concret cap d'aquestes.

El que s'ha fet ha estat definir un tipus de metodologia convinguda amb el tutor, que funcionés bé amb les característiques del projecte, tal com es mostra en el diagrama de flux de la Figura 3.1.

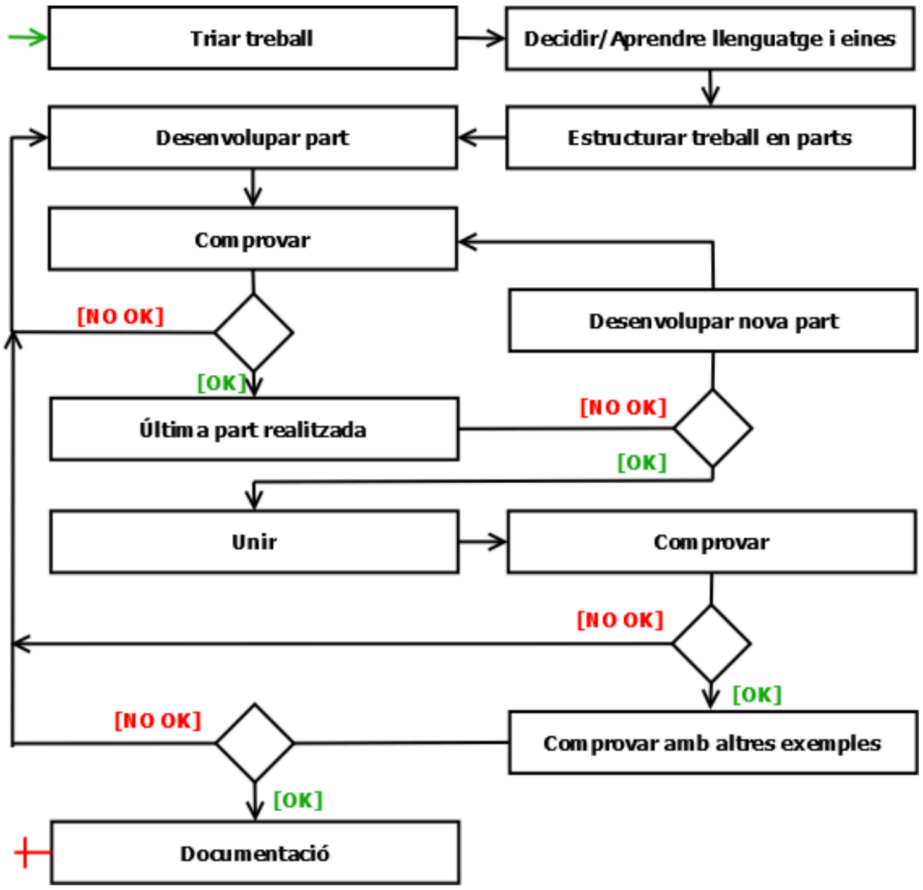


Figura 3.1: Diagrama de flux de la metodologia utilitzada

1. Triar el treball a desenvolupar.

2. Decidir el llenguatge de programació i les eines a utilitzar.

3. Aprendre el llenguatge de programació i el funcionament de les eines escollides.

4. Estructurar el treball en parts segons les funcions que ha de realitzar.

5. Desenvolupar la part corresponent seguint l'ordre de l'estructura del treball.

6. Fer comprovacions per tal de confirmar que el funcionament és correcte al finalitzar la part.

    - 6.1. Si al fer les comprovacions el resultat no és l'esperat, es torna al punt 5 per a realitzar els canvis oportuns en la última part desenvolupada o en les anteriors, si és convenient.

    - 6.2. Si al fer les comprovacions el resultat és l'esperat, es desenvolupa la part següent tornant al punt 5, si s'han finalitzat les parts amb les seves respectives comprovacions s'inicia el punt 7.

7. Unir totes les parts desenvolupades i comprovar que el funcionament és correcte.

    - 7.1. Si al fer les comprovacions el resultat no és l'esperat, es torna al punt 5 per a realitzar els canvis oportuns en l'última part desenvolupada o en les anteriors, si és convenient.

    - 7.2. Si al fer les comprovacions el resultat és l'esperat s'inicia el punt 8.

8. Generar diferents models d'exemple per a comprovar que el funcionament és el correcte.

    - 8.1. Si al fer les comprovacions el resultat no és l'esperat, es torna al punt 5 per a realitzar els canvis oportuns en l'última part desenvolupada o en les anteriors, si és convenient.

    - 8.2. Si al fer les comprovacions el resultat és l'esperat s'inicia el punt 9.

9. Arrodonir la documentació.

Tal com es pot veure consisteix en dividir el projecte en mòduls i organitzar en el temps el desenvolupament, segons el temps de verificació i de correcció. Tant durant el temps de desenvolupament com de verificació, es fa un seguiment mitjançant tutories setmanals o

bisetmanals depenent de l'etapa, ja que en els inicis són més lents que no pas en les etapes finals, i no sempre es necessari fer tutories setmanalment.

Quan s'acaba un mòdul, sempre que es pugui, es finalitza totalment de manera que no es torna a tocar. D'aquesta manera es garanteix que els errors que puguin sorgir en la resta de mòduls són únicament d'aquests i no de cap dels anteriors. I si es dóna el cas que ho és, al menys s'han minimitzat al màxim els efectes que s'hagin pogut propagar.

Pel procés de disseny s'utilitzarà el llenguatge de modelat estàndard dins el camp de l'enginyeria de programari, l'UML (*Unified Modeling Language*). L'UML s'utilitza per definir un sistema, per detallar els seus elements, per documentar i construir. Per aconseguir això, l'UML disposa de nombrosos tipus de diagrames que mostren diversos aspectes dels elements representats.

Capítol 4 Planificació

4.1 Planificació inicial

La planificació inicial del treball, datava l'entrega al mes de juny. Al mes de Gener es van dur a terme les primeres reunions en les quals parlàvem sobre possibles temes de TFG. Al cap de dues setmanes vam trobar un tema que ens semblava molt interessant tant als meus tutors com a mi, i tant bon punt el vam tenir escollit, vaig començar a aprendre tot el que em seria necessari per crear una intel·ligència artificial capaç de jugar a un videojoc.

La major part de codi d'IA està fet en Python i el framework sobre el qual treballem també utilitza aquest llenguatge. És per això que vaig decidir dedicar un temps en aprendre més a fons a codificar amb ell. Un cop triat això, només feia falta començar a estudiar intel·ligències artificials, els funcionaments, com codificar-les… L'última decisió que es va prendre va ser el joc, es va escollir el Cuphead perquè es tracta d'un joc de plataformes en 2D i és famós per la seva complexitat.

Per sort, un cop havia començat el TFG, començava a codificar l'agent i la xarxa neuronal podia ensenyar als tutors el que estava fent i comentàvem com ho anaven veient, d'aquesta manera tenia un feedback ràpid i segur per saber com anava el treball.

Així doncs, amb aquestes idees de base, es va dissenyar la planificació de treball següent:

1. Utilitzar i aprendre el llenguatge de programació Python.
2. Aprendre a utilitzar la llibreria SerpentAI.
3. Estudiar el funcionament del joc escollit.
4. Desenvolupament d'un agent que jugui al joc escollit emprant SerpentAI.
5. Estudi i recerca d'eines d'intel·ligència artificial més adient, incloent-hi behaviour trees i xarxes neuronals.
6. Estudi i aprenentatge del funcionament d'una llibreria de xarxes neuronals com el Keras.
7. Desenvolupament d'una intel·ligència artificial adequada.
8. Finalització i arrodoniment de la documentació del treball realitzat

En la Figura 4.1 es veuen les tasques repartides en el temps

Figura 4.1 Planificació inicial

4.2 Planificació final

Tot i la planificació inicial, sovint en un projecte les coses no surten com un ho planifica i voldria que sortissin des del principi. Sí que la planificació en general s'acaba respectant, però apareixen moltes subtasques, problemes, errors i algun contratemps que s'han de solucionar. Així doncs, finalitzat el treball la distribució de tasques dins la planificació va quedar com es pot veure en la Figura 4.2.

A partir de finals de febrer de 2019, un cop ja començava a entendre com funciona el framework Serpent, vam començar a investigar quina era la millor manera per crear la intel·ligència artificial. Trobem dues opcions: la primera consisteix a buscar sprites a la pantalla i obtenir la posició, i treballar a partir d'aquesta. La segona opció agafa tota la pantalla de joc i la processa directament. Cada un té els seus punts a favor i en contra. Vaig decidir apostar per la busca dels sprites i al cap d'un mes treballant-hi em vaig adonar que seria molt millor utilitzar tot el frame sencer.

Un cop decidit aquest canvi d'enfocament, tocava canviar l'agent i crear la intel·ligència artificial. La creació de l'agent és bastant simple, sempre depenendrà del joc i per tant un mateix agent no es podrà utilitzar directament si no hi apliquem canvis.

A mitjans de mes de maig, ja teníem una intel·ligència artificial creada a partir de xarxes neuronals i funcionant, però aquí va aparèixer un altre problema. Tot i entrenar-la per més de 100.000 iteracions no vam aconseguir que superés el nivell esperat ni aprenia gairebé res. La conclusió que vam extreure d'això és que la xarxa neuronal no estava ben construïda i per tant havíem de trobar la manera de crear-ne una d'òptima.

Per fer això vam decidir estudiar els algorismes genètics i aplicar-los en el codi per tal de trobar l'esmentat anteriorment, una xarxa neuronal òptima que potser tampoc aconseguia solucionar el nivell del joc, però sí que aconseguiria millors resultats que una xarxa neuronal base creada manualment i sense testejar-ne d'altres. Un cop aconseguit això vam poder dir que la part pràctica del treball estava finalitzada.

Figura 4.2 Planificació Final

Capítol 5
Marc de treball i conceptes previs

5.1 Videojocs de plataformes

Els videojocs de plataformes són un gènere de videojocs en el qual cal caminar, saltar o escalar sobre una sèrie de plataformes i penya-segats, amb enemics, mentre es recullen objectes per a poder completar el joc. Sol usar scroll horitzontal cap a esquerra o cap a la dreta. És un gènere molt popular de videojocs creat en els primers anys 80 i que continua mantenint bastant de popularitat en l'actualitat. Entre els seus exponents màxims trobem jocs com el Super Mario Bros, Rayman, Sonic…

5.1.1 Cuphead

El Cuphead és un videojoc de plataformes en 2D. Aquest videojoc funciona en forma de nivells trobats dins un món fantàstic. Aquests poden ser de dos tipus:

- Nivells run'n go: són els típics nivells dels jocs de plataformes en els quals s'ha d'anar avançant, esquivant obstacles i eliminant enemics per tal de superar-lo

- Nivells contra enemic poderós: són nivells en els quals només lluitem contra un sol enemic, el qual anomenem "boss", i l'haurem d'eliminar. Aquest tindrà molta vida i unes mecàniques pròpies.

5.1.1.1 Mecàniques del joc

El personatge podrà dur a terme les següents accions:

- Triar la direcció a la qual mirarà/apuntarà
- Moure's cap a la dreta o esquerra o quedar-se quiet
- Disparar
- Saltar
- Fer un "*dash*" (recórrer una petita distància a una gran velocitat).

Si el personatge és tocat per un enemic, per un míssil creat per l'enemic o cau en un forat, perdrà un punt de vida d'un total de 3 que en té.

5.2 Machine Learning

El Machine Learning és un subcamp de la intel·ligència artificial que proporciona al l'ordinador la capacitat d'aprendre sense que el programador indiqui les regles que ha de seguir per aconseguir l'objectiu, ho farà per si sol automàticament.

Per tant, el Machine Learning és capaç de desenvolupar un algoritme de predicció per a cada problema en particular. Aquest tipus d'algoritmes busquen patrons dins de les dades per tal de comprendre'ls i, a partir d'això, construir un model per tal de predir el resultat.

Hi ha moltes opcions d'algoritmes d'aquest tipus, però es solen classificar en tres classes: l'aprenentatge supervisat, l'aprenentatge no supervisat i l'aprenentatge per reforçament.

- En l'aprenentatge supervisat, les dades que utilitzem per a l'entrenament tenen la solució, la qual anomenem etiqueta.

- Quan ens referim a l'aprenentatge no supervisat, les dades d'entrenament no inclouen les etiquetes i l'algoritme intentarà classificar la informació per si mateix.

- L'aprenentatge per reforçament és quan el model s'implementa en forma d'un agent que haurà d'explorar un espai desconegut i determinar les accions a dur a terme mitjançant prova i error. El model aprendrà per si mateix gràcies a les recompenses i penalitzacions que rebi segons les seves accions. L'agent intentarà crear la millor estratègia possible per obtenir la màxima recompensa en el menor temps possible.

5.3 El concepte Agent

Un sistema d'IA està format per un agent i el seu entorn. Els agents actuen en l'entorn. L'entorn pot contenir més d'un agent. Un agent és qualsevol cosa que pot:

- Percebre l'entorn mitjançant sensors
- Actuar sobre aquest entorn mitjançant actuadors

Cada agent pot percebre les seves pròpies accions (però no sempre els efectes).

Per entendre l'estructura dels agents intel·ligents, hauríem de familiaritzar-nos amb l'arquitectura i l'algorisme dels agents. L'arquitectura és la maquinària on l'agent executa. És un dispositiu amb sensors i actuadors, per exemple: un cotxe robotitzat, una càmera, un PC. L'algorisme dels agents és una implementació d'una funció d'agent. Una funció d'agent és un mapa des de la seqüència de percepció (història de tot el que un agent ha percebut fins a la data) fins a una acció.

Exemples d'agent:

- Un agent de programari té pulsacions de teclat, contingut de fitxers, paquets de xarxa rebuts que actuen com a sensors i mostra a la pantalla, fitxers, paquets de xarxa enviats que actuen com a actuadors.

- Un agent humà té ulls, orelles i altres òrgans que actuen com a sensors i mans, cames, boca i altres parts del cos que actuen com a actuadors.

- Un agent robòtic disposa de càmeres i cercadors de gamma infraroja que actuen com a sensors i diversos motors que actuen com a actuadors.

5.3.1 Tipus d'agents

Els agents es poden agrupar en quatre classes en funció del grau d'intel·ligència percebuda i capacitat:

- Agents de reflex simple
- Agents de model reflex basat en models
- Agents basats en objectius
- Agents basats en utilitats
- Agent d'aprenentatge

Agents reactius simples

La funció d'agent es basa en la regla condició-acció. Una regla condició-acció és una regla que assigna un estat, i.e., condició, a una acció. Si la condició és certa, llavors es faran les accions, si no. Aquesta funció d'agent només té èxit quan l'entorn és completament observable. Per als agents simples que operen en entorns parcialment observables, sovint apareixen bucles infinits.

Agents reactius basats en models

Funciona trobant una regla on l'estat coincideixi amb la situació actual. Un agent basat en models pot gestionar entorns parcialment observables mitjançant un model sobre el món. L'agent ha de fer un seguiment de l'estat intern que s'ajusta per cada percepció i que depèn de l'historial de percepcions. L'estat actual s'emmagatzema a l'agent que manté algun tipus d'estructura que descriu la part del món que no es pot veure. L'actualització de l'estat requereix informació sobre com evoluciona el món independentment de l'agent i com aquesta acció afecta el món.

Agents basats en objectius

Aquest tipus d'agents prenen la decisió en funció de quina distància es troben actualment de l'objectiu. Les seves accions tenen com a objectiu reduir la distància respecte a la meta. D'aquesta manera, l'agent pot triar entre múltiples possibilitats, seleccionant aquella que arriba a un estat d'objectiu. El coneixement que dóna suport a les decisions es representa de manera explícita i es pot modificar, cosa que fa que aquests agents siguin més flexibles. Normalment requereixen cerca i planificació.

Agents basats en utilitats

Quan hi ha diverses alternatives possibles, aleshores, per decidir quina és la millor, s'utilitzen agents basats en utilitats. Ells opten per accions basades en una preferència (utilitat) per a cada estat. De vegades no és suficient aconseguir l'objectiu desitjat. És possible que busquem un viatge més ràpid, més segur i més econòmic per arribar a una destinació. Cal tenir en compte la "felicitat" de l'agent. La utilitat descriu com de "feliç" és l'agent.

Agent d'aprenentatge

Un agent d'aprenentatge en IA és el tipus d'agent que pot aprendre de les seves experiències passades o té capacitats d'aprenentatge. Comença a actuar amb coneixements bàsics per després actuar i adaptar-se automàticament a través de l'aprenentatge.

Un agent d'aprenentatge té principalment quatre components conceptuals, que són:

- Element d'aprenentatge: s'encarrega de crear millores aprenent des de l'entorn.
- Crític: l'element d'aprenentatge utilitza comentaris que descriuen el funcionament de l'agent respecte a un estàndard de rendiment.
- Element de rendiment: és responsable per seleccionar l'acció exterior
- Generador de problemes: aquest component és responsable de suggerir accions que condueixin a experiències noves i informatives.

Veure Figura 5.1

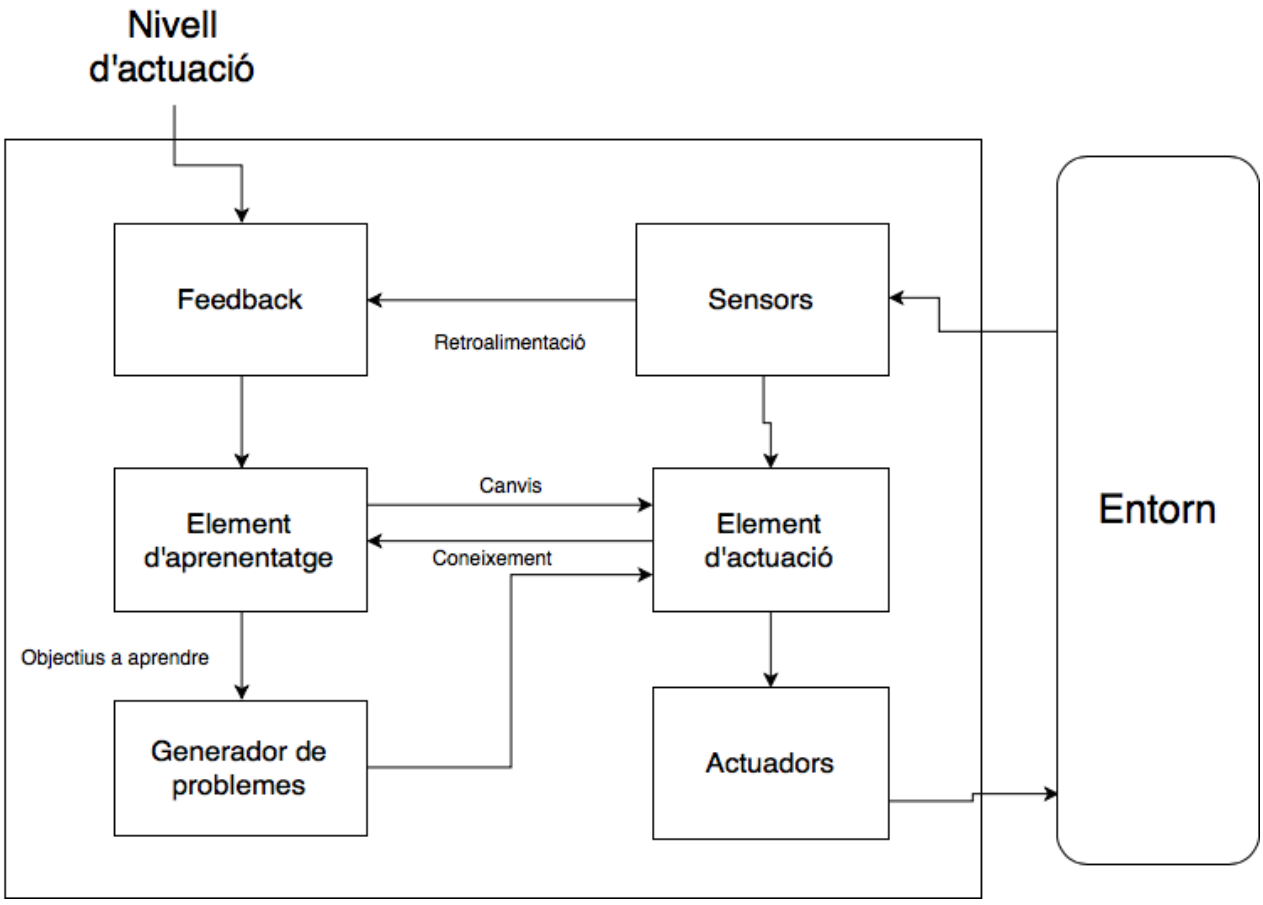


Figura 5.1

5.4 Xarxes neuronals artificials

Un tipus d'algorismes de Machine Learning són les xarxes de neurones artificials. Aquestes són un paradigma d'aprenentatge i processament automàtic inspirat en la forma en què funciona el sistema nerviós dels animals. Es tracta d'un sistema d'interconnexió de neurones en una xarxa que col·labora per produir un estímul de sortida.

Consisteixen en una simulació de les propietats observades en els sistemes neuronals biològics a través de models matemàtics recreats mitjançant mecanismes artificials. L'objectiu és aconseguir que les màquines donin respostes similars a les que és capaç de donar el cervell, que es caracteritzen per la seva generalització i la seva robustesa.

5.4.1 Com funcionen?

Una xarxa neuronal es compon d'unitats anomenades neurones. Cada neurona rep una sèrie d'entrades i emet una sortida. Veure Figura 5.2

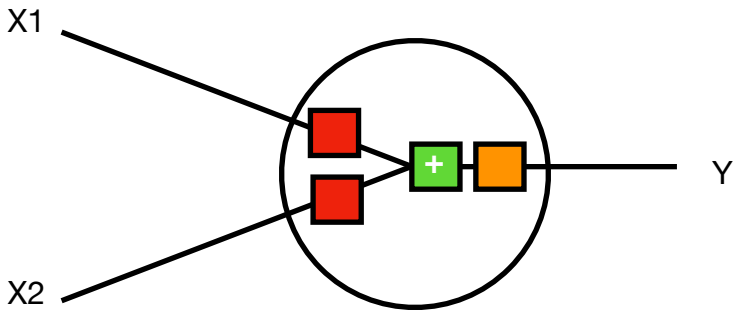


Figura 5.2 Neurona

En aquesta neurona hi passen 3 coses. Primer, cada input és multiplicat per un pes w:

$$x1 \text{ —> } x1 * w1$$
$$x2 \text{ —> } x2 * w2$$

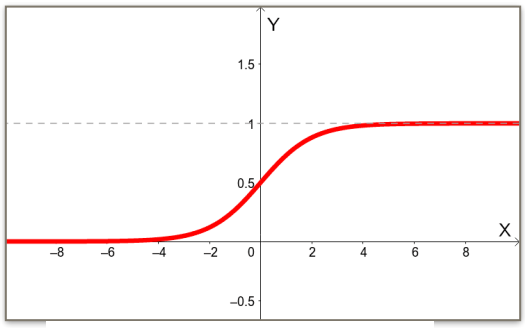
Després, després tots els pesos ponderats es sumen amb un biaix b:

$$(x1 * w1) + (x2 * w2) + b$$

Finalment la suma anterior es transforma per una funció d'activació:

$$y = f(x1 * w1 + x2 * w2 + b)$$

Aquesta funció s'utilitza per convertir una entrada il·limitada en una sortida predictible. Una de les funcions més utilitzada és la sigmoide. Veure en Figura 5.3



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

Figura 5.3 Funció Sigmoide

La funció sigmoide només dóna sortides entre 0 i 1. Per tant comprimeix el rang (-∞,+∞) a (0,1). Els nombres negatius grans es converteixen en ~0 i els positius grans en ~1.

Altres funcions d'activació que es poden utilitzar són:

- La tangent hiperbòlica: comprimeix el rang (-∞,+∞) a (-1,1) .

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

- La ReLU: anul·la els valors negatius i manté els valors positius.

$$f(x) = \max(0, \mathrm{x}) = \begin{cases} 0 \; for \;\; x < 0 \\ x \; for \;\; x \geq 0 \end{cases}$$

- La Softmax: transforma les sortides en una representació en forma de probabilitats, de manera que el sumatori de totes dona 1,

$$f(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$$

- Existeixen moltes més que podem triar segons el problema al qual ens estiguem enfrontant.

5.4.2 Combinar neurones per formar una xarxa neuronal

Les neurones s'agrupen en grups de manera que formen capes, les quals estan connectades entre elles. En trobem tres tipus diferents:

- D'entrada, reben dades i senyals que procedeixen de l'entorn.
- De sortida, proporcionen una resposta segons les dades entrades
- Ocultes, no reben ni donen informació a l'entorn (processament intern de la xarxa).

Una xarxa neuronal no és res més que un conjunt de neurones connectades. A la Figura 5.4 podem veure una xarxa neuronal bàsica:

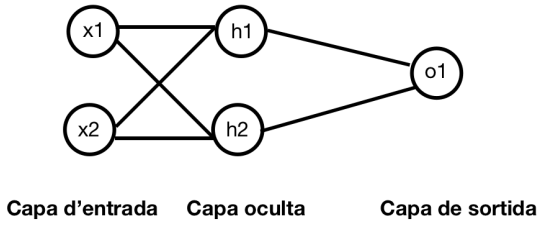


Figura 5.4

Aquesta xarxa està composta per dos inputs (x1 i x2), dues neurones ocultes (h1 i h2) i una capa de sortida amb una neurona (o1). Podem veure que els inputs de la capa de sortida que són els inputs de o1, són els outputs de h1 i h2. D'aquí prové el nom de xarxa.

Una xarxa neuronal pot tenir qualsevol nombre de capes i qualsevol nombre de neurones en aquestes capes. La idea bàsica és la mateixa, a partir de les capes d'entrada alimentar les capes ocultes i amb aquestes alimentar les següents fins a arribar a la capa de sortida.

5.5 CNN - Xarxes Neuronals Convolucionals

Una Xarxa Neuronal Convolucional és un algoritme que rep un senyal d'entrada, assigna importància (pesos que s'aprenen a mesura que s'entrena l'algoritme) a diversos aspectes i objectes de la imatge per tal de diferenciar-la d'altres.

El preprocessament requerit per una CNN és molt inferior en comparació amb qualsevol altre algoritme de classificació.

5.5.1 L'imatge d'entrada

Imaginem que tenim una imatge RGB de qualitat 8K és a dir, una imatge composta per tres capes (Red, Green, Blue) i de píxels 7680x4320 per a cada capa. Si hem de processar aquesta imatge en tots els seus tres components, serà molt costós computacionalment.

El rol de la Xarxa Convolucional és reduir les imatges de forma que siguin més fàcils de processar sense perdre les característiques que són crítiques per obtenir una bona predicció. Això és molt important quan dissenyem una arquitectura, la qual no només volem que sigui bona aprenguen característiques, sinó que també serà escalable per a grans bases de dades. Veure Figura 5.5

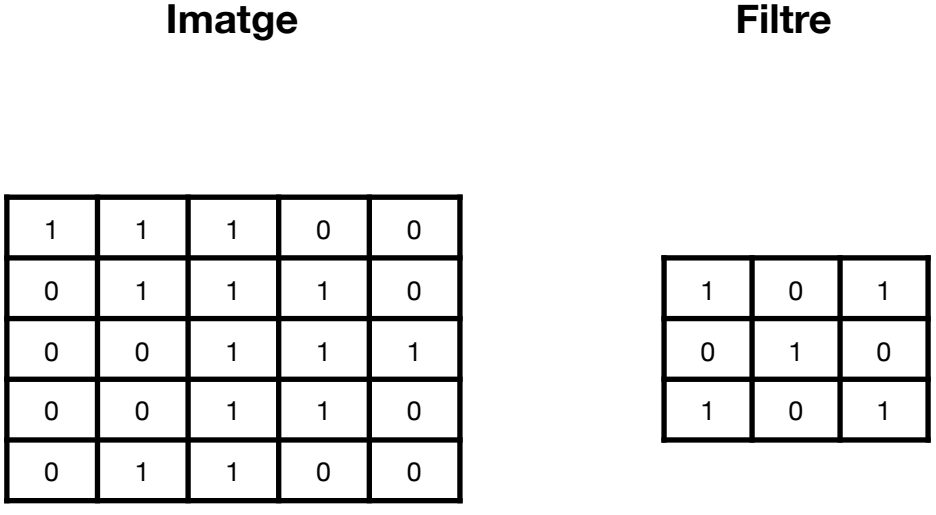
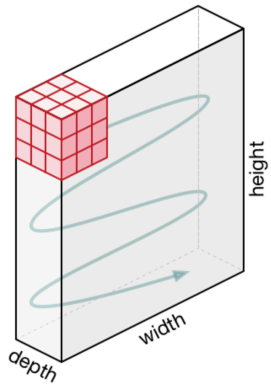


Figura 5.5
Tenim una imatge 5x5x1 a la qual li aplicarem un filtre 3x3x1 per obtenir la imatge convolucional. El filtre farà 9 moviments i cada cop farà una multiplicació de matrius entre ell i la porció de la imatge seleccionada (3x3x1)



Resultat

4	3	4
2	4	3
2	3	4

Figura 5.6. Podem veure el recorregut que farà el filtre a l'esquerra i a la dreta el resultat que ens donarà aplicar el filtre a la imatge de la Figura 5.5.

L'objectiu de les operacions de convolució és extreure característiques, com per exemple els contorns de la imatge. La primera capa serà la responsable de capturar colors, gradients, orientació... Amb les capes següents, l'arquitectura s'adapta a les característiques d'alt nivell, i ens dóna una xarxa que és capaç d'entendre les imatges de la base de dades de manera semblant a com ho faríem nosaltres.

5.6 Algorismes genètics

És una tècnica per a trobar solucions aproximades a problemes d'optimització i recerca. Els algorismes genètics són una classe particular d'algorismes evolutius que utilitzen tècniques inspirades per l'evolució biològica, imiten el procés de selecció natural, on els individus més aptes, se seleccionen per a la reproducció per produir els descendents de la següent generació.

La selecció natural

El procés de selecció natural comença amb la selecció dels individus més aptes de la població inicial. Aquests produeixen descendència que heretera les característiques dels seus pares. Aquest procés s'anirà duent a terme de forma iterativa fins que al final, trobarem una generació amb els individus més aptes.

Es consideren cinc fases en un algoritme genètic:

- Població inicial
- Funció d'aptitud (fitness)
- Selecció
- Encreuament
- Mutació

5.6.1 Població Inicial

El procés comença amb un conjunt d'individus anomenats població. Cada un correspon a una solució al problema que es vol resoldre.

Cada individu està caracteritzat per una sèrie de paràmetres coneguts com a gens. Aquests gens estan connectats i formen un Cromosoma el qual serà la solució. Normalment els valors usats per a cada gen són 1 o 0. Veure Figura 5.7.

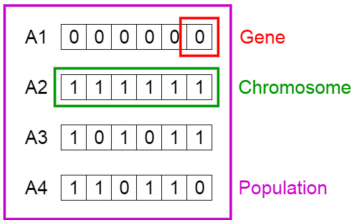


Figura 5.7. Elements d'un sistema genètic.

5.6.2 Funció d'aptitud

La funció d'aptitud determina l'aptitud d'un individu i atorga una puntuació per a cada un.

5.6.3 Funció de selecció

La idea principal és seleccionar els individus més aptes i passar els seus gens a les següents generacions. La probabilitat que un individu sigui escollit per a la reproducció, vindrà determinat per la seva puntuació d'aptitud, com més elevada, més possibilitats.

5.6.4 Encreuament

Es tracta de la fase més important en un algoritme genètic. Per cada parell de pares que s'encreuen, s'escull un punt d'encreuament de forma aleatòria entre els gens. Per exemple, en aquest cas el punt serà el 4. Veure Figura 5.8.

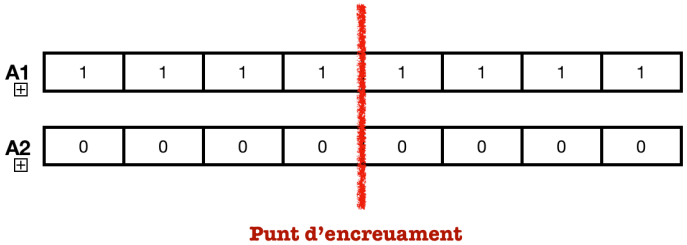


Figura 5.8: punt d'encreuament entre dos gens.

La descendència es crearà canviant els gens dels pares entre ells fins a arribar al punt d'encreuament. Veure Figura 5.9.

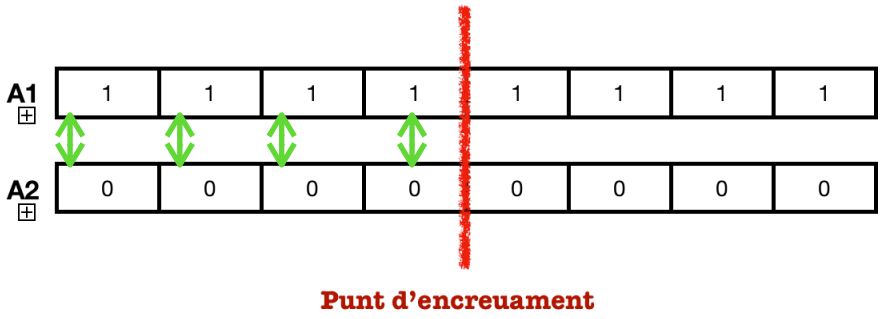


Figura 5.9: Gens que s'intercanviaran.

La nova descendència quedarà de la manera que es mostra a la Figura 5.10.

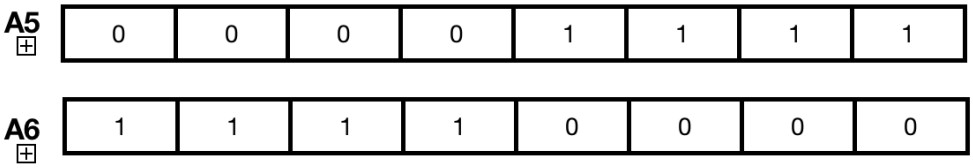


Figura 5.10: Nova descendència creada a partir dels cromosomes de la figura 5.9.

5.6.5 Mutació

Hi ha una petita probabilitat d'aplicar una mutació en certs descendents. Això implica que alguns bits de l'array del Cromosoma poden ser canviats. Veure Figura 5.11.

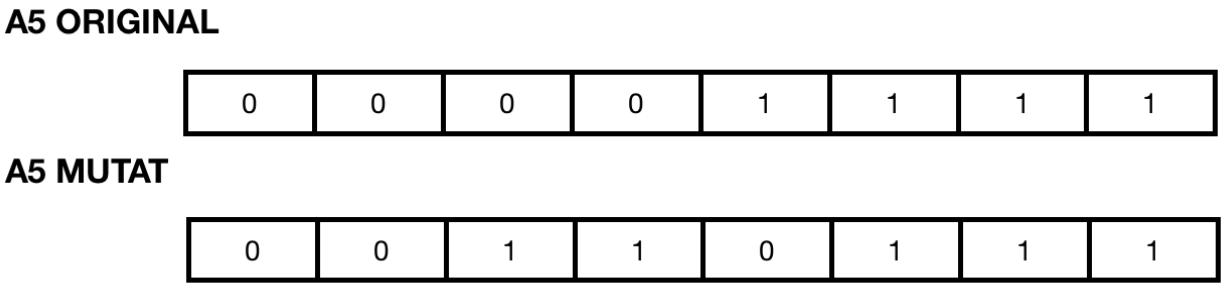


Figura 5.11. Mutació aplicada al cromosoma A5, el resultat és l'A5 mutat.

5.6.6 Fi

Direm que l'algorisme ha acabat quan la descendència no produeixi un canvi significatiu a la solució i per tant direm que hem trobat una solució més òptima possible.

5.6.7 Pseudocodi

Veure Figura 5.12.

```
Inici
Crear població inicial
Calcular l'aptitud
Mentre
        Seleccionar
        Encreuar
        Mutar
        Calcular l'aptitud
Fins que la població hagi arribat al límit d'aptitud
Fi
```

Figura 5.12. Pseudocodi d'un algorisme genètic

5.7 Q-Learning

Q-learning és una tècnica d'aprenentatge per reforç utilitzada en aprenentatge automàtic. L'objectiu del Q-learning és aprendre una sèrie de normes que li diguin a un agent quina acció prendre sota quines circumstàncies. No requereix un model de l'entorn i pot gestionar problemes amb transicions estocàstiques i recompenses sense requerir adaptacions.

El Q-Learning apren la funció acció-valor Q(s,a): En un estat determinat, determina quin valor ens retornarà prendre una acció o una altra.

Per exemple, en la imatge que podem veure a la Figura 5.13, es calcula el valor de moure el peó dues passes endavant.

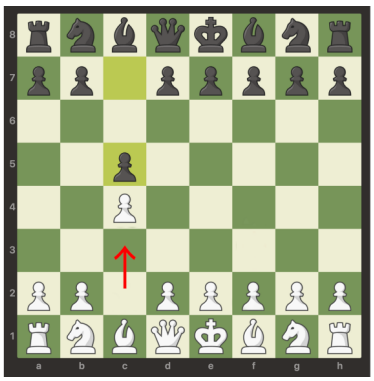


Figura 5.13. Busquem el valor de dur a terme l'acció de moure la peça blanca endavant.

Crearem una taula de memòria Q[s,a] per guardar els valors de Q per totes les possibles combinacions de "s" i “a”, on “s" és l’estat en el qual ens trobem i "a" és l’acció que prenem. Veure Figura 5.14

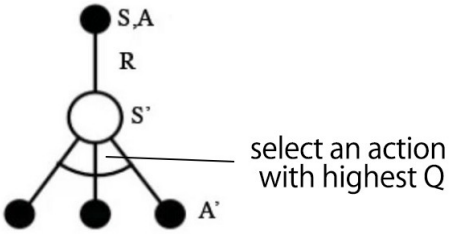


Figura 5.14. Representació gràfica del que fem l'utilitzar Q-learning

En una partida d’escacs, rebrem un punt si guanyem la partida i perdrem un si no ho fem. El Q-learning es basa a crear aquesta taula d’accions-valor. Veure Figura 5.15.

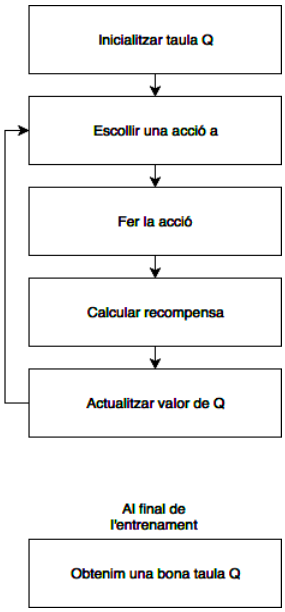


Figura 5.15. Representació del procés a seguir per obtenir la taula Q

El problema d'aquesta tècnica és que, si les combinacions d'estats i accions són molt llargs, la memòria i els requeriments computacionals seràn molt alts. Per solucionar això farem ús d'un algoritme anomenat DQN —> Deep Q Network , el qual es basa en aproximar el valor de Q(s,a).

5.8 DQN

Fent ús del Q-Learning, depenem de les funcions de valor per prendre accions. Aquests valors aniran canviant a mesura que sabem el que hem de fer i el que no. En el cas que ens trobem en una situació que no hàgim vist abans, l'algorisme anirà totalment perdut i no sabrà quina acció escollir. Per solucionar això, el DQN elimina la taula Q(s,a), i utilitzarà una funció d'estimació que serà la Figura 5.16.

$$Q^*(s,a) = r_0 + \gamma(r_1 + \gamma r_2 + \gamma^2 r_3 + \ldots) = r_0 + \gamma \max_a Q^*(s',a)$$

Figura 5.16. Funció d'estimació del DQN

El que calculem és, a partir de l'estat "s" i prenent una acció "a", quina serà la suma de recompenses des d'aquell estat fins a un estat x. La variable y, que sempre serà més petita que 1, determinarà fins quina futura acció tindrem en compte a l'hora de fer el càlcul, a mesura que les accions són més futures el seu valor és més pròxim a 0.

5.9 DDQN

El DDQN consisteix a crear dues xarxes neuronals que anomenarem θ- i θ. Utilitzem la primera per guardar els valors de Q i la segona inclourà totes les millores que es vagin fent a cada partida. Després de "x" iteracions les sincronitzarem. D'aquesta manera, els petits canvis que es puguin produir en una sola partida no tindrà efecte instantani i no crearan modificacions errònies dins la xarxa, sinó que, un cop en portem "x" passes, farem una actualització dels valors de la xarxa molt més precisa. Veure figura 5.17.

$$Q(s,a) \rightarrow r + \gamma \max_a \tilde{Q}(s',a)$$

Figura 5.17 Utilitzarem la xarxa ~Q per fer els càlculs i guardarem els resultats a Q.

Capítol 6
Requisits del sistema

En aquest capítol s'explica quina és la problemàtica que focalitza aquest treball, i com s'ha plantejat per resoldre-la, així com explicar els requisits funcionals i no funcionals que ha de complir el sistema.

6.1 Plantejament de la problemàtica

L'única eina que existeix per crear agents per a videojocs amb una base i controlats per intel·ligències artificials és el SerpentAI. Per tant, utilitzaré aquest framework per tal de crear una eina que sigui capaç d'aprendre a jugar a un videojoc i que ens ajudi a trobar errors o comportaments inesperats en aquest.

Aquesta eina s'intentarà fer el màxim general possible, de manera que es podrà utilitzar el mateix codi en un altre joc fent els mínims canvis possibles. El codi es basarà en tècniques de xarxes neuronals, explicades anteriorment. En concret s'utilitzarà DQN la mateixa tècnica que va utilitzar DeepMind per jugar a jocs de l'Atari el 2013 que es tracta d'una barreja de xarxes neuronals amb aprenentatge per reforç. Utilitzaré també algorismes genètics per tal de trobar la configuració més òptima per a la xarxa neuronal.

Per fer ús de l'eina per un altre joc no serà necessari conèixer les xarxes neuronals ni els algorismes genètics, ja que el codi d'aquests no s'haurà de modificar. El que si s'haurà de modificar, serà el codi de l'agent que controli el videojoc. En aquest hi haurem de posar els paràmetres que donaràn informació sobre el rendiment del jugador (per exemple la puntuació, el temps en partida…)

6.2 Plantejament de la solució

La solució plantejada per assolir es basarà en la llibreria SerpentAI. L'usuari podrà agafar el codi de l'agent creat per al Cuphead i utilitzar-lo per a un altre videojoc. L'usuari haurà de canviar la part del codi on s'indiquen el joc i els inputs d'entrada que inclouen els controls i la manera de puntuar. Un cop fet això, simplement haurà d'activar l'agent i deixar-lo fer. (Cada joc serà molt personalitzable de manera que aquest codi general serà molt extensible i molt diferent per a cada joc si es vol treure el màxim rendiment)

6.3 Requisits funcionals

Els requisits funcionals expliquen què ha de fer l'aplicació, és a dir, totes les funcionalitats que tindrà sense especificar com es farà. Aquestes són les funcionalitats que tindrà l'aplicació:

- L'agent serà capaç de jugar de forma autònoma al videojoc.
- L'agent serà capaç d'aprendre a jugar i maximitzar la puntuació que obté segons el sistema de recompenses que li hem determinat.
- Es pot escollir el % d'accions que es volen dur a terme de forma aleatòria i el % previstes gràcies a la xarxa neuronal.
- Es pot escollir el mode de joc de l'agent (entrenament o 100% previst).
- El codi és pel joc Cuphead, aquest codi es podrà utilitzar en altres jocs però s'hauran de canviar els inputs d'entrada (controls i recompenses), ja que són pròpies de cada joc. Es podran modificar una gran quantitat de paràmetres dins del codi entre les quals:
    - Dimensions dels frames del joc.
    - Mida del buffer d'imatges usades per a entrenar.
    - Velocitat d'entrenament.
    - Epsilon inicial —>% d'accions aleatòries inicials.
    - Epsilon final —> % d'accions aleatòries finals.
    - Canviar el fitxer model per a la xarxa neuronal.

6.4 Requisits no funcionals

Per poder utilitzar l'agent i el framework Serpent.AI, s'hauràn de complir els següents requisits:

6.4.1 Requisits de Hardware

Els requisits mínim seran els requisits que sol·licita el joc sobre el qual volem treballar més una càrrega extra de RAM, CPU i GPU.

Sistema Operatiu:

- Windows 7 o superior

- macOS El Capitan o superiors i instal·lar Homebrew (un gestor de paquets que ens serà molt útil durant la instal·lació)

- En Linux, tenir un escriptori basat en X11 i la comanda xwininfo, instal·lada de la següent manera " ***pacman -S xorg-xwininfo***"

RAM:
- 8 GB encara que, com més en tinguem millor per augmentar la velocitat.

CPU:
- Qualsevol CPU ens valdrà encara que com en el cas anterior, com millor sigui, més velocitat tindrem a l'hora de fer els càlculs per calcular la xarxa neuronal.

GPU:
- De nou, com millor sigui més ràpid farem els càlculs per tal de fer la nostra xarxa neuronal

6.4.2 Requisits de Software

- Python 3.6+
- Serpent.AI
- Tesseract, és el framework que s'utilitza per a reconeixement de caràcters
- Kivy, és el Framework usat pel debuggador visual del Serpent.AI
- Redis, s'utilitzarà com a memòria interna per a cada partida.

Només per Windows:
- Anaconda 5.2
- Build Tools for Visual Studio 2017

Per Windows i Linux:
- Dependències de GPU de Tensorflow si es volen utilitzar Deep Neural Networks
- Si s'utilitza una targeta Nvidia, instal·lar el Cuda

Capítol 7
Estudis i decisions

En aquest capítol descriurem el programari i les llibreries utilitzades durant el desenvolupament del projecte, i en justificarem l'elecció. En el cas de les llibreries, n'explicarem el funcionament bàsic per poder comprendre-les.

7.1 Serpent.AI

Serpent.AI és un framework per ajudar els desenvolupadors a crear agents de joc. Permet convertir qualsevol videojoc en un entorn adequat per a experimentar amb ell en codi Python.

Conté un ampli assortiment de mòduls de suport que proporcionen solucions als escenaris més habituals, així com eines de CLI (permet donar instruccions i interactuar amb el framework a partir de la línia de comandes), per accelerar el desenvolupament. El millor del Serpent és la flexibilitat: es pot utilitzar aprenentatge per reforç, tècniques de visió per computador, processament d'imatges i trigonometria, prémer aleatòriament els botons esquerre o dret…

Entre d'altres el SerpentAI permet:

- Generació de codi per a la plataforma de suport (Steam, Executables i Buscadors Web)
- Generació de codi per nous agents de joc
- Captures de frames del joc i de regions d'interès.

- Frame Grabbing en funcionament en alta resolució i alts fotogrames per segon
- Classificador de context que es pot entrenar fàcilment.
- Controlador d'inputs tan per teclat com per ratolí.
- Registre d'*Sprites*, identificació i localització
- Mòdul OCR
- Debugador visual per veure les imatges/frames guardades a memòria.

```
Adrias-MacBook-Pro:SerpentAI adri$ serpent

Serpent.AI v2018.1.2
Available Commands:

           setup: Perform first time setup for the framework
          update: Update the framework to the latest version
         modules: List the install status of the framework's optional modules
     grab_frames: Start the frame grabber
        activate: Activate a plugin
      deactivate: Deactivate a plugin
         plugins: List all locally-available plugins
          launch: Launch a game through a plugin
            play: Play a game with a game agent through plugins
          record: Record player input from a game
        generate: Generate code for game and game agent plugins
           train: Train a context classifier with collected context frames
         capture: Capture frames, screen regions and contexts from a game
 visual_debugger: Launch the visual debugger
     window_name: Launch a utility to find a game's window name
   record_inputs: Start the input recorder
```

Figura 7.1: Vista de les comandes disponibles al framework Serpent.AI

Com podem veure en la Figura 7.1 tenim tot un seguit de comandes que ens ajudaran a dur a terme les accions més bàsiques com ara:

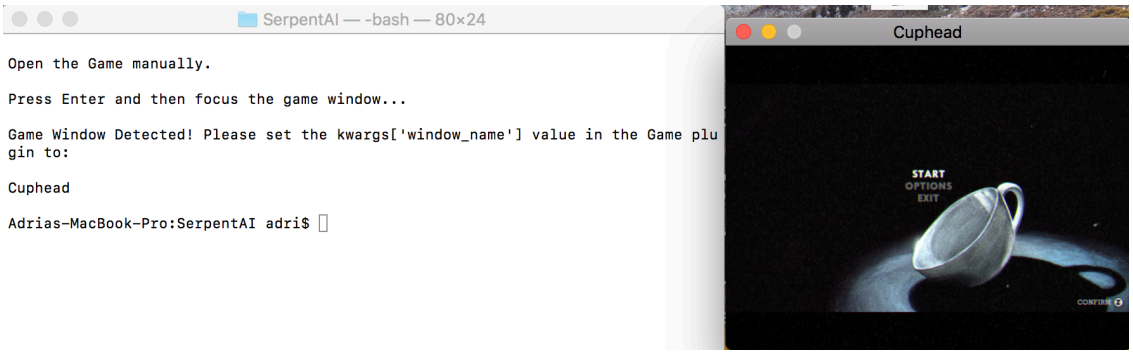
- Configurar el Serpent, actualitzar i mostrar els mòduls opcionals.
- Generar els plugins per a un joc determinat, activar-los o desactivar-los.
- Llistar tots els plugins disponibles
- Obrir el joc i/o jugar-hi a partir d'un plugin.
- Gravar la pantalla
- Entrenar l'agent

7.1.1 Hello Serpent.AI

1. El primer que haurem de fer serà instal·lar el joc sobre el qual vulguem treballar.

2. Després haurem d'obrir-lo i canviar les opcions gràfiques de manera que el joc quedi en mode finestra i seleccionar una resolució que ens sembli adequada.

3. Un cop fet això haurem de saber l'identificador de la finestra:
   3.1. A Linux i Windows serà el títol de la finestra
   3.2. En macOS serà el nom de procés del joc. Com que aconseguir aquest nom pot ser una mica complicat, hi ha una comanda al Serpent anomenada window_name la qual haurem d'utilitzar.



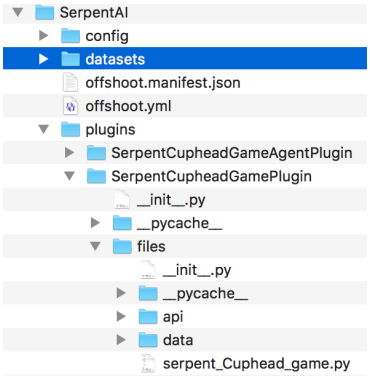
4. El serpent ens dóna una comanda per crear el plugin per qualsevol joc "***serpent generate game***"

5. Ens demanarà el nom del joc i la plataforma des de la qual l'utilitzarem.

```
What is the name of the game? (Titleized, No Spaces i.e. AwesomeGame):
Exemple
How is the game launched? (One of: 'steam', 'executable', 'web_browser'):
```

Un cop aquest pas, se'ns mostrarà el següent missatge: SerpentSuperHexagonGamePlugin was installed successfully!

Si mirem dins el directori de plugins podrem confirmar que s'han creat els fitxers per al plugin del joc.



6. Una vegada tenim el plugin, només falta fer-li unes petites modificacions. Haurem de canviar el paràmetre WINDOW_NAME amb l'identificador buscat anteriorment. Si el joc és de la plataforma Steam, cal buscar la APP_ID en aquesta pàgina web SteamDB i canviar-lo pel del plugin.

```
kwargs["platform"] = "steam"

kwargs["window_name"] = "Cuphead"

kwargs["app_id"] = "268910"
```

7. Un cop hàgim dut a terme tots els passos, només ens faltarà utilitzar la comanda "*serpent launch* ***(nom del joc)***"

7.1.2 Processament d'imatges

Tenim diverses opcions a l'hora de processar imatges:

- La més simple de totes és fer una captura general de la pantalla de joc o d'una zona determinada. Aquesta es guardarà a la carpeta */SerpentAI/datasets/collect_frames.*

  Comanda per capturar frames és:

  serpent capture frame <Nom del joc> <Interval de temps per captura>

  Comanda per capturar una zona del frame és:

  serpent capture region <Nom del Joc> <Interval de temps per captura> <Coordenades de la regió>

  Aquestes funcionalitats recauen sobre la llibreria Tesseract, encara que la utilitzarem a partir del framework de Serpent.

- Podem obtenir les coordenades d'interès d'una regió de la imatge. Per exemple, volem obtenir les coordenades on està situat el text "Retry" en la següent imatge cal fer:

  1. Obrirem la imatge amb la comanda:

     spritex datasets/collect_frames/<frame_file_name>.png

2. Podem utilitzar moltes funcions com ara fer aparèixer una graella, obtenir els colors característics de la imatge… El que volem serà seleccionar la regió RETRY, i per tant premem sobre Select Region. Veure Figura 7.2

Figura 7.2. Veiem com quedaria la selecció i les coordenades que obtindríem.

3. Obtenim els següents valors els quals podrem utilitzar en el nostre codi per saber, per exemple, la regió on hem de prémer per tornar a jugar, la posició on podria estar un enemic o per comparar entre aquesta zona de la imatge i aquesta mateixa zona d'una altra imatge… He decidit utilitzar dues opcions més per mostrar el potencial que té aquesta llibreria. Les opcions són les de crear un sprite i les de crear un sprite únic (la qual fa el mateix que l'anterior però elimina el fons de la imatge). Veure Figura 7.3

Figura 7.3: Comparació de crear un sprite i un sprite únic

7.1.3 Motius de la selecció

Serpent.AI és l'únic framework dirigit específicament a crear intel·ligències artificials per a videojocs. Hi ha altres eines que permeten fer coses semblants, però aquest ja té molt codi programat per a dur terme objectius que són comuns en molts jocs, com per exemple: seleccionar una regió de la pantalla, comparar imatges, seleccionar els frames que es volen capturar cada segon… Tot i això, sempre que vulguem treure el màxim de rendiment o crear una intel·ligència artificial molt específica per a un joc, etc, haurem de crear noves funcions a partir de les originals o des de 0.

7.2 PyCharm

PyCharm és un IDE (Integrated development environment) que s'utilitza en la programació informàtica, específicament per al llenguatge Python. És desenvolupat per la companyia txeca JetBrains. Proporciona anàlisi de codi, un depurador gràfic, un comprovador d'unitats integrat, integració amb sistemes de control de versions (VCSes) i suporta el desenvolupament web amb Django i Data Science amb Anaconda. PyCharm és multiplataforma, amb versions de Windows, MacOS i Linux.

Algunes de les característiques més importants del 'PyCharm' són:

- Assistència i anàlisi de codificació, amb completament de codi, ressaltat de sintaxi en l'error, integració de línies i solucions ràpides.
- Navegació de projectes i codis: visualitzacions de projectes especialitzats, vistes de l'estructura de fitxers i salt ràpid entre fitxers, classes, mètodes i usos
- Refactorització de Python: inclou el canvi de nom, el mètode d'extracció, la introducció de variables, la introducció de constants, la pujada, la baixada i altres
- Suport per a marcs web: Django, web2py i Flask
- Depurador Python integrat
- Proves unitàries integrades, amb cobertura de codi per línia
- Desenvolupament de Google App Engine a Python
- Integració de control de versions: interfície d'usuari unificada per Mercurial, Git, Perforce i CVS amb llistes de canvis i fusió

Altres alternatives a aquest IDE eren PyDev d'Eclipse i el IDE de Komodo més àmpliament enfocat.

7.2.1 Motius de la selecció

Treballarem en el framework de SerpentAI i el PyCharm ens permet poder desenvolupar el projecte sota una eina que treballa conjuntament amb moltes llibreries.

Podíem haver desenvolupat aquest projecte en qualsevol dels IDEs esmentats anteriorment, fins i tot a partir d'un editor de text com el BBEdit el qual va ser el primer que vaig utilitzar per a veure el codi i jugar-hi una mica, però a recomanació del tutor i després per pròpia experiència, vaig estar molt satisfet amb PyCharm.

7.3 Tensorflow

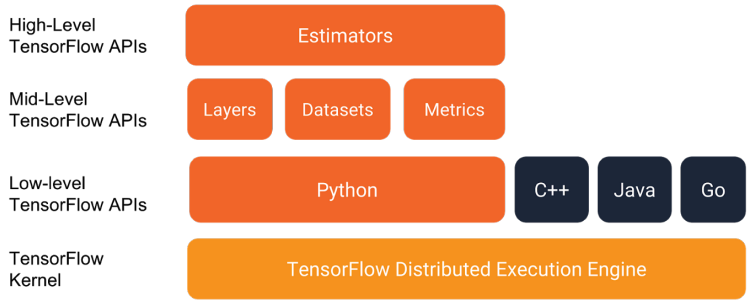
TensorFlow és una eina de computació numèrica creada per Google. S'utilitza per projectes de Machine Learning.

Hi ha alternatives a aquest com ara MXNet, Pytorch, Chainer y Caffee. Cada eina té els seus avantatges i els seus inconvenients, però he decidit decantar-me per aquesta perquè un dels meus tutors la domina molt bé i orientar.

TensorFlow té un excel·lent equilibri de flexibilitat i escalabilitat. La flexibilitat permet als desenvolupadors i investigadors provar idees noves en un temps curt. L'escalabilitat permet que els models desenvolupats puguin ser usats per milions d'usuaris. TensorFlow, a més, és portable, pel que pot ser utilitzat en molts de dispositius.

Addicionalment, TensorFlow compta amb un conjunt divers d'eines per poder tenir els projectes en producció d'una manera molt senzilla. Companyies com OpenAI, NVIDIA, Airbnb i Intel usen TensorFlow per aquestes raons.

TensorFlow consisteix d'APIs en diferents nivells. En el nivell més alt, hi ha els Estimators. Els Estimators permeten desenvolupar un pipeline de Machine Learning de manera molt senzilla. En el nivell més alt també tenim a Keras, el qual utilitzarem en aquest projecte i s'explica en el següent punt. En el nivell més baix tenim l'API de Python, el qual és el més usat, però també hi ha de C ++, Java, Go, JavaScript i Swift.



7.4 Keras

El Keras és una API de Xarxes Neuronals d'alt nivell i de Codi Obert escrita en Python. Es pot executar sobre TensorFlow (explicat al punt anterior), Microsoft Cognitive Toolkit o Theano.

Característiques principals:

- **Facilitat d'ús:** El Keras posa l'experiència de l'usuari al centre, segueix les millors pràctiques per reduir la càrrega cognitiva: ofereix API constants i simples, minimitza les accions de l'usuari necessàries per a casos d'ús comuns i proporciona comentaris clars sobre els errors que comet l'usuari.

- **Modularitat:** El Keras funciona en forma de models. Un model s'entén com una seqüència o un gràfic de mòduls autònoms configurables que es poden connectar amb les mínimes restriccions possibles. Per exemple, les capes neuronals, les funcions de cost, els optimitzadors, els esquemes d'inicialització, les funcions d'activació i els esquemes de regularització són mòduls autònoms que es poden combinar per crear nous models.

- **Fàcil extensibilitat:** Els mòduls nous són senzills d'afegir (com a noves classes i funcions). El fet de poder crear mòduls nous fàcilment permet una expressivitat total, fent que Keras sigui apte per a investigacions avançades.

- **Treballa amb Python:** No hi ha fitxers de configuració de models separats en un format declaratiu. Els models es descriuen en el codi Python, que és compacte, més fàcil de depurar i facilita l'extensibilitat.

7.4.1 Hello Keras

Com s'ha dit en el punt anterior, el Keras funciona a partir de models. El model més simple es tracta del seqüencial, el qual fa una pila de capes. Així seria un model seqüencial

```
from keras.models import Sequentialº

model = Sequential()
```

Ara podem afegir capes amb la funció `.add()`:

```
from keras.layers import Dense

model.add(Dense(units=64, activation='relu', input_dim=100))
model.add(Dense(units=10, activation='softmax'))
```

Un cop el model s'ajusti a les nostres necessitats, utilitzarem la funció .compile() per determinar el procés d'aprenentatge:

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

Ara podem iterar sobre les nostres dades d'entrenament en diferents lots (batches en anglès):

```
model.fit(x_train, y_train, epochs=5, batch_size=32)
```

N'Avaluem el rendiment:

```
loss_and_metrics = model.evaluate(x_test, y_test, batch_size=128)
```

Generem prediccions sobre nous models:

```
classes = model.predict(x_test, batch_size=128)
```

7.5 Python

Python és un llenguatge de programació interpretat el qual intenta que el codi sigui el màxim de llegible.

És un llenguatge de programació multiparadigma. Això vol dir que més que forçar als programadors a desenvolupar un estil particular de programació, permet diversos estils: programació orientada a objectes, programació imperativa i programació funcional. Altres paradigmes estan suportats mitjançant l'ús d'extensions.

Una característica important de Python és la resolució dinàmica de noms; és a dir, el que enllaça un mètode i un nom de variable durant l'execució del programa (també anomenat enllaç dinàmic de mètodes).

Un altre objectiu del disseny del llenguatge és la facilitat d'extensió. Es poden escriure nous mòduls fàcilment en C o C ++. Python pot incloure en aplicacions que necessiten una interfície programable.

7.5.1 Motius de la selecció

La veritat és que en el món de l'Intel·ligència Artificial el llenguatge predominant és el Python i no hi ha hagut gaire dubte en quin llenguatge utilitzar. Tot i això, hi havia altres opcions com ara el Lua, el qual s'ha utilitzat en un famòs projecte de IA sobre Mario Bros. També es podria haver utilitzat C++, Lisp, R… però com he dit anteriorment, el Python en aquest àmbit predomina molt més i per tant la quantitat d'informació disponible és molt superior.

7.6 GanttProject

GanttProject és una aplicació que permet fer l'organització de projectes. Està implementat en Java i és multiplataforma (Windows, Linux i OSX), sota la llicència GPL. Permet organitzar el projecte en tasques i sub-tasques, indicant el període de temps que es trigarà en completar-les.

7.7 Anaconda

Anaconda és una distribució lliure i oberta dels llenguatges Python i R, utilitzada en ciència de dades, i aprenentatge automàtic. Això inclou processament de grans volums d'informació, anàlisi predictiu i càlculs científics. Està orientat a simplificar el desplegament i administració dels paquets de software.

Les diferents versions dels paquets s'administren mitjançant el sistema de gestió de paquets Conda, el qual el fa bastant senzill d'instal·lar, utilitzar, i actualitzar programari de ciència de dades i aprenentatge automàtic, com per exemple tres llibreries que utilitzo que són: Scikit-team, TensorFlow i SciPy.

Instal·lar Python 3.6 o superior no és complicat, però el Serpent necessita moltes llibreries científiques que són extremadament difícils de compilar al Windows, i gràcies a l'Anaconda podem aconseguir-ho.

Anirà molt bé crear un "shortcut" per fer ús de l'Anaconda, ja que cada ho haurem d'utilitzar en cada comanda del Serpent i del Python.

Crearem un entorn per al serpent:

conda create --name serpent python=3.6

I per activar l'entorn dins la carpeta en la qual ens trobem, utilitzarem la següent comanda:

conda activate serpent

7.8 CUDA

CUDA (acrónim de Compute Unified Device Architecture (Arquitectura de còmput de dispositius unificats)) és una plataforma de computació paral·lela i model d'Interfície de programació d'aplicacions (API) creada per Nvidia per permetre a desenvolupadors i enginyers de software accelerar l'execució dels seus codis fent servir Unitats de processament gràfic (GPU).

La plataforma va ser desenvolupada amb l'objectiu de treballar conjuntament amb llenguatges de programació com C, C++ i Fortran, tot i que posteriorment ha acabat implementant-se en un ampli espectre de llenguatges de programació com Java, Lua, MATLAB... Aquesta accessibilitat facilita als especialistes de programació paral·lela l'ús dels recursos que ofereix una GPU, en contrast amb altres APIs natives pròpiament de la programació de gràfics (com Direct3D o OpenGL), les quals tenien un enfocament purament gràfic i requerien grans coneixements en aquest àmbit.

Només es pot fer ús d'aquest software en el cas que es tingui una targeta gràfica NVDIA.

Capítol 8
Anàlisi i disseny del sistema

En aquest capítol veurem de manera detallada com s'ha analitzat i dissenyat el sistema partint de la problemàtica i seguint la metodologia exposada. Per a fer-ho explicarem el diagrama de cas d'ús general amb les fitxes corresponents i el diagrama de classes del sistema.

8.1 Diagrama i fitxa de cas d'ús

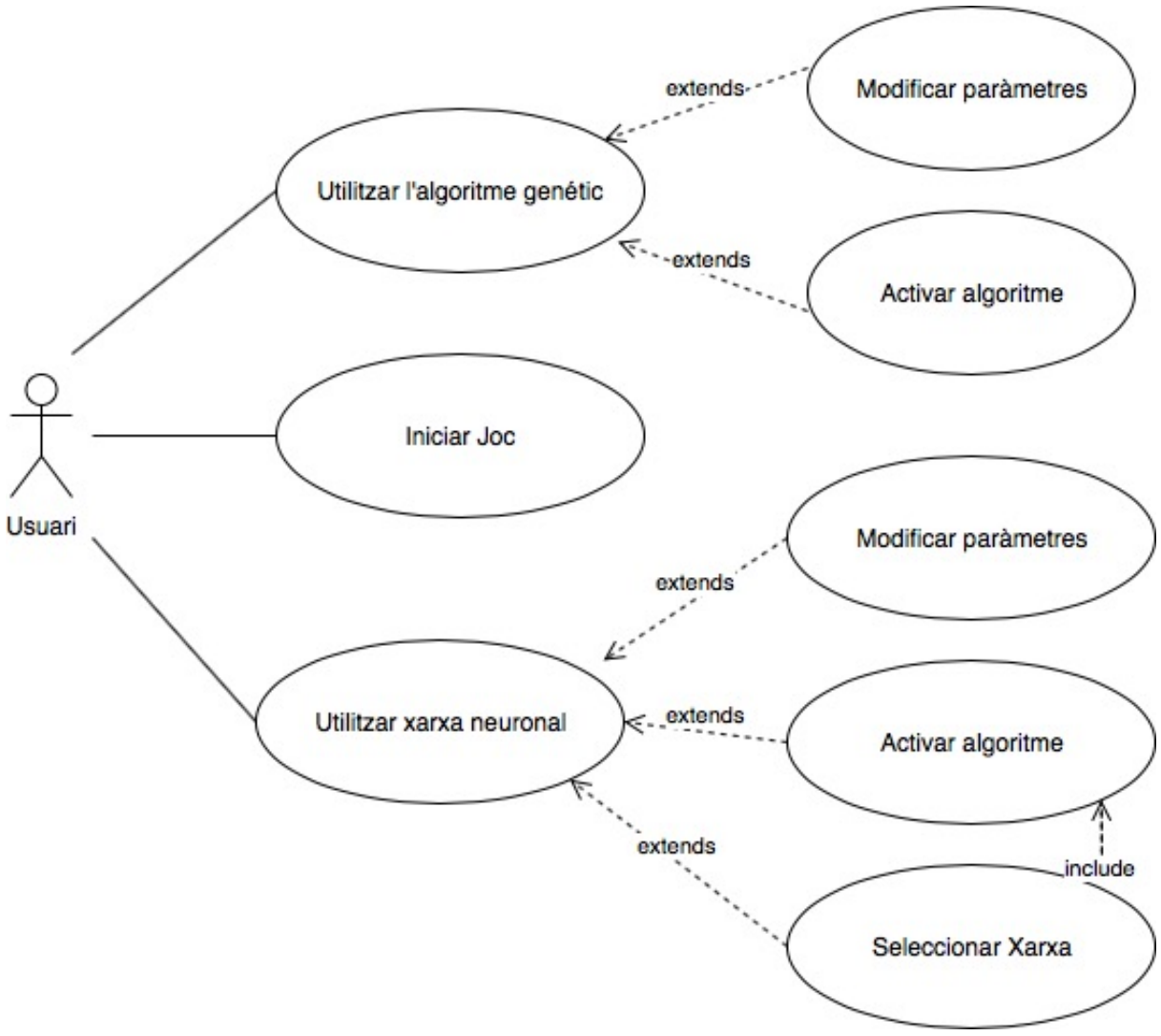


Figura 8.1. Diagrama de cas d'ús

Fitxa de cas d'ús	Iniciar Cuphead
Descripció	L'usuari vol iniciar el videojoc Cuphead
Actor	Usuari
Pre-condició	El plugin SerpentCupheadGame ha d'existir i ha d'haver estat instal·lat
Flux principal	1. Escriure a la línia de comandes, ***serpent launch Cuphead***
Post-condició	El joc s'ha iniciat

Fitxa de cas d'ús	Utilitzar Xarxa Neuronal
Descripció	L'usuari vol fer ús de la xarxa neuronal
Actor	Usuari
Pre-condició	El plugin SerpentCupheadGameAgent ha d'existir i ha d'haver estat instal·lat
Flux principal	1. Seleccionar que és vol fer: 1.1. Seleccionar Xarxa Neuronal 1.2. Activar Xarxa Neuronal 1.3. Canviar paràmetres
Post-condició	-

Fitxa de cas d'ús	Seleccionar Xarxa
Descripció	L'usuari vol seleccionar quina xarxa utilitzarà
Actor	Usuari
Pre-condició	El fitxer de la xarxa es troba dins la carpeta Serpent.AI
Flux principal	1. Introduir el "path" del fitxer.
Post-condició	-

Fitxa de cas d'ús	Activar Xarxa
Descripció	L'usuari vol entrenar la xarxa
Actor	Usuari
Pre-condició	-
Flux principal	1. Posar la variable genetic_algorithm_activated de la classe serpent_Cuphead_game_Agent a False 2. Escriure a la línia de comandes, ***serpent play Cuphead SerpentCupheadGameAgent***
Post-condició	Es pot veure text representatiu del que passa en el joc per la línia de comandes

Fitxa de cas d'ús	Utilitzar Algoritme Genètic
Descripció	L'usuari vol fer ús de l'algoritme genètic
Actor	Usuari
Pre-condició	El plugin SerpentCupheadGameAgent ha d'existir i ha d'haver estat instal·lat
Flux principal	1. Seleccionar que és vol fer: 1.1. Activar Algoritme Genètic 1.2. Canviar paràmetres
Post-condició	-

Fitxa de cas d'ús	Activar Algoritme Genètic
Descripció	L'usuari vol activar l'algoritme genètic
Actor	Usuari
Pre-condició	El plugin SerpentCupheadGameAgent ha d'existir i ha d'haver estat instal·lat
Flux principal	1. Posar la variable genetic_algorithm_activated de la classe serpent_Cuphead_game_Agent a True 2. Escriure a la línia de comandes, ***serpent play Cuphead SerpentCupheadGameAgent***
Post-condició	Es pot veure text representatiu del que passa en el joc i de l'estat de l'algoritme genètic per la línia de comandes.

Fitxa de cas d'ús	Modificar paràmetres
Descripció	L'usuari vol modificar els paràmetres d'entrenament de la xarxa
Actor	Usuari
Pre-condició	-
Flux principal	1. Buscar els paràmetres dins la llista de paràmetres 2. Seleccionar paràmetres a modificar 3. Modficar paràmetres
Post-condició	-

Fitxa de cas d'ús	Modificiar Paràmetres Genètics
Descripció	L'usuari vol fer activar l'algoritme genètic
Actor	Usuari
Pre-condició	El plugin SerpentCupheadGameAgent ha d'existir i ha d'haver estat instal·lat
Flux principal	1. Buscar els paràmetres dins la llista de paràmetres. 2. Seleccionar paràmetres a modificar 3. Modficar paràmetres
Post-condició	-

8.2 Diagrama de classes

En aquesta secció veurem com s'han dissenyat el diagrama de classes (Figura 8.2), tenint en compte l'estructura del Serpent i les llibreries addicionals utilitzades.

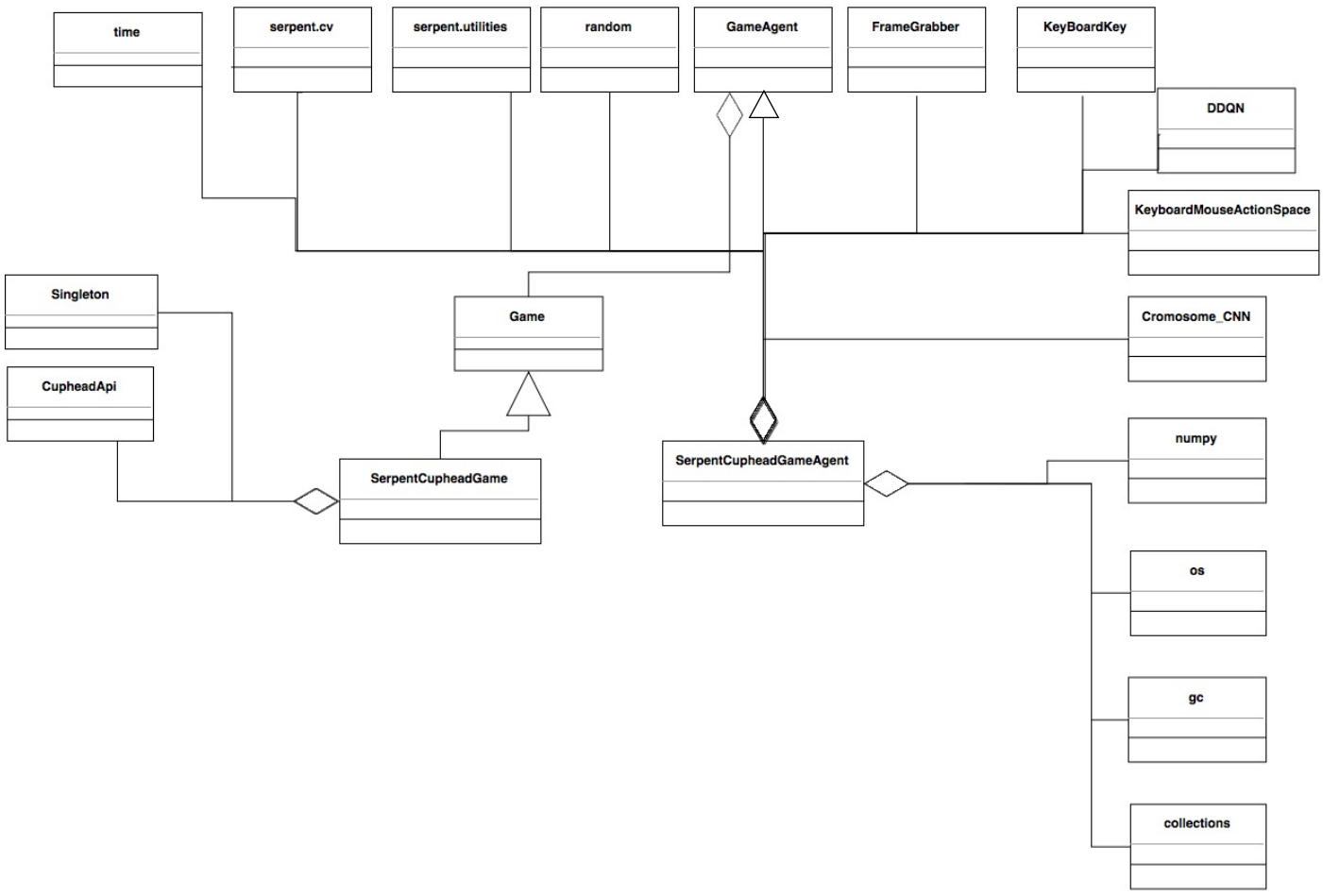


Figura 8.2 Diagrama de classes

Les dues classes principals són *SerpentCupheadGame* i *SerpentCupheadGameAgent.* La primera hereta de la classe *Game,* la qual conté tots els elements necessaris per utilitzar el SerpentAI en un joc. També hereta de la classe Singleton, que vol dir que només es podrà crear una instància d'aquesta. La classe serveix per inicialitzar el joc i definir diversos paràmetres que són necessaris per al funcionament de la segona classe.

La segona classe, *SerpentCupheadGameAgent*, és on combinarem el joc, les xarxes neuronals (classe *DDQN*) i els algoritmes genètics (classe *Cromosome_CNN*). També incorporà altres classes que seran de gran ajuda:

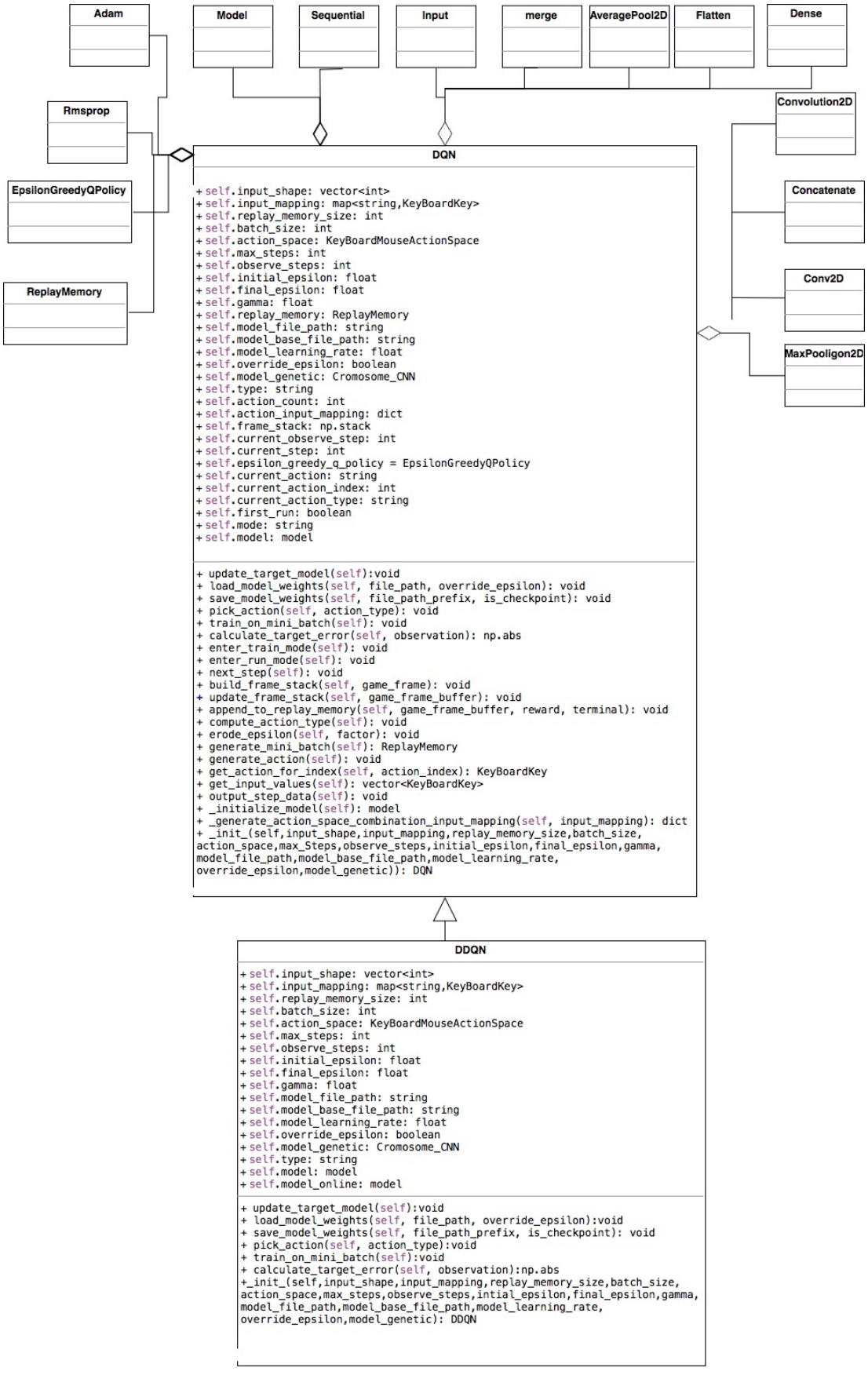
- *KeyboardKey*, la qual permet activar tecles del teclat sense haver-ho de fer manualment.
- *FrameGrabber*, la qual permet treballar amb els frames del joc.
- *Time*, la qual permet utilitzar el temps per calcular la duració de cada partida.
- *GameAgent*, serà la classe des de la qual heretarem les funcions.
- *Random*, per utilitzar elements aleatoris.
- gc, es tracta del "garbage collector".
- os, permet dur a terme funcionalitats del sistema operatiu, serveix per escriure i llegir arxius.
- numpy, és una llibreria de funcions matemàtiques, serveix sobretot per treballar amb vectors i matrius.

A continuació veurem individualment per a cada classe, els mètodes i atributs necessaris.

8.3 Les classes

En aquesta secció veurem en més detall les classes utilitzades, explicant els mètodes que s'han utilitzat i quina funcionalitat tenen.

8.3.1 DQN i DDQN



La classe DQN implementa l'algoritme esmentat en l'apartat 5.6 i la DDQN l'esmentat en el 5.7. Les xarxes neuronals es crearan dins d'instàncies de la classe DDQN, la qual hereterà la majoria dels mètodes de la casse DQN.

Les classes Dense, Flatten, Convolution2D, MaxPooling2D, AveragePooling2D, Input, merge, Concatenate, Conv2D, Adam, rmsprop, Model, Sequential són classes de la llibreria Keras i són les que permeten treballar amb xarxes neuronals.

Per últim, també tenim les classes EpsilonGreedyQPolicy i ReplayMemory, les quals van amb el Serpent i ajuden a treball amb aprenentatge per reforç.

Els atributs utilitzats són:

- **self.input_shape:** mida que ha de tenir l'entrada de la xarxa.
- **self.input_mapping:** conjunt de tecles que la xarxa podrà utilitzar.
- **self.replay_memory_size:** mida de la memòria on guardarem els conjunts de frames que anem obtinguen.
- **self.batch_size:** quantitat de frames que tindrà cada conjunt.
- **self.action_space:** combinacions de tecles que es poden arribar a dur a terme.
- **self.max_steps:** passos màxims fins als quals arribarà l'algoritme a l'hora d'entrenar la xarxa.
- **self.observe_steps:** contador de passos en mode observador.
- **self.initial_epsilon:** aleatorietat inicial.
- **self.final_epsilon:** aleatorietat final i en mode "run".
- **self.gamma:** constant.
- **self.replay_memory:** memòria on guardem els conjunts de frames.
- **self.model_file_path:** localització del fitxer que indica els pesos del model de la xarxa neuronal.
- **self.model_base_file_path:** localització del model de la xarxa neuronal.

- **self.model_learning_rate:** velocitat a la qual aprèn la xarxa neuronal.

- **self.override_epsilon:** en el cas que carreguem un model, sobreescriurem l'epsilon en el cas que aquest atribut sigui cert o utilitzarem la que tenia el model en el cas que sigui fals.

- **self.model_genetic:** guardarem el model que ha estat creat a partir de l'algorisme genètic.

- **self.type:** string que identifica el tipus d'algorisme, DQN o DDQN.

- **self.action_count:** nombre d'accions dutes a terme.

- **self.action_input_mapping:** conté totes les possibles combinacions de tecles possibles.

- **self.frame_stack:** conjunt de frames d'una determinada acció.

- **self.current_observe_step:** quantitat de passes en mode "observe".
- **self.current_step:** quantitat de passes en mode "train.

- **self.epsilon_greedy_q_policy:** variable on guardem totes les variables que facin referència a l'epsilon, és a dir, self.initial_epsilon, self.final_epsilon, self.max_steps

- **self.current_action:** combinació de tecles que es durà a terme en aquella acció.

- **self.current_action_index:** índex de la combinació que es durà terme (indica la posició de l'acció al diccionari action_input_mapping)

- **self.current_action_type:** string que indicarà si l'acció ha estat presa aleatòriament o no.

- **self.first_run:** és cert si es tracta de la primera partida.

- **self.mode:** mode en el qual ens trobem, pot ser "train", "observer" o "run".

- **self.model:** model de la xarxa neuronal que s'actualitza cada x iteracions (explicat al punt 5.9).

- **self.model_online:** model de la xarxa neuronal que s'actualitza a cada iteració.

Els mètodes utilitzats de les classes DQN i DDQN són els següents:

- **load_model_weights(self, file_path, override_epsilon):** En el cas que ja tinguem el model de la xarxa neuronal determinat i l'hàgim entrenat anteriorment, podem guardar el procés d'aquest entrenament i carregar-lo més endavant. La diferència entre la funció en DQN i DDQN és que en aquest últim actualitzarà les dues xarxes neuronals mentre que el primer ho farà amb l'única que té.

- **save_model_weights(self, file_path_prefix, is_checkpoint):** Guardarem els valors dels pesos de la xarxa amb el següent format (*f"{file_path_prefix}_dqn_{self.current_step}_{epsilon}_.h5").* D'aquesta manera indiquem on ho guardarem, els passos que portem i l'epsilon en la qual ens trobem, quan carreguem els pesos utilitzarem aquests valors. La diferència entre DDQN i DQN és la mateixa que en l'anterior funció, DDQN guarda els valors de les dues xarxes mentre DQN només ho fa de l'única que té.

- **pick_action(self,action_type):** segons els frames que hi hagi a buffer, el model provarà si és capaç de predir la següent acció o la seleccionarà de forma aleatòria, i triarà l'acció a fer. La funció és la mateixa tant per DDQN i DQN, però es redefineix perquè el nom de les xarxes és diferent i s'ha d'actualitzar.

- **train_on_mini_batch(void):** entrena la xarxa neuronal a partir de 8 conjunts d'imatges seleccionats aleatòriament, els quals estaran guardats al buffer, juntament amb la recompensa obtinguda amb cada una d'elles. En el cas del DDQN es farà l'explicat a l'apartat 5.7. Una xarxa buscarà la millor acció que ha de prendre mentre que l'altre buscarà la recompensa màxima que es pot obtenir a partir d'aquesta.

- **calculate_target_error(self,observation):** calcula la diferència de valors entre el conjunt d'imatges anterior (estat) i l'actual de manera que obté la recompensa per l'acció que ha dut a terme. La diferència entre DDQN i DQN torna a ser la mateixa, s'utilitza la xarxa actualitzada per al nou estat i la xarxa no actualitzada per a l'estat antic.

- **enter_train_mode(self):** canvia la variable self.mode a "TRAIN" de manera que ara la xarxa neuronal està en format d'entrenament. També guardarà el valor d'epsilon (aleatorietat) en el cas que no sigui 0.

- **enter_run_mode(self):** canvia la variable self.mode a "RUN" de manera que ara la xarxa neuronal està en format de màxim rendiment, només hi haurà un 1% d'accions aleatòries. També guardarà el valor d'epsilon (aleatorietat) en el cas que no sigui 0 i determinarà l'actual a 0.01.

- **next_step(self):** augmenta el comptador de steps en 1. Si es troba en mode entrenament, augmentarà la variable self.current_steps. Si es troba en moda observador, augmentarà la variable self.current_observer_step.

- **build_frame_stack(self, gameFrame):** construeix un conjunt de 4 frames en una taula np.stack.

- **update_frame_stack(self, game_frame_buffer):** afegeix nous frames al conjunt de frames guardats.

- **append_to_replay_memory(self, game_frame_buffer, reward, terminal):** utilitza el valor de la recompensa obtinguda i el conjunt de frames per tal de calcular el valor de dur a terme l'acció en aquell estat i ho afegeix a la memòria.

- **compute_action_type(self):** determina si l'acció serà aleatòria o no, segons el valor d'epsilon.

- **erode_epsilon(self,factor):** disminueix el valor d'*epsilon* un determinat valor -> *factor.*

- **generate_mini_batch(self):** retorna una estructura replayMemory que conté un conjunt de 8 frames seguits agafats aleatòriament.

- **generate_action(self):** selecciona l'acció a dur a terme segons la variable self.action_index.

- **get_action_for_index(self,action_index):** No s'utilitza, fa el mateix que l'anterior però passant l'índex de l'acció per paràmetre enlloc de tractar-se d'un atribut de DQN.

- **get_input_values(self):** retorna les tecles que s'han de prémer.

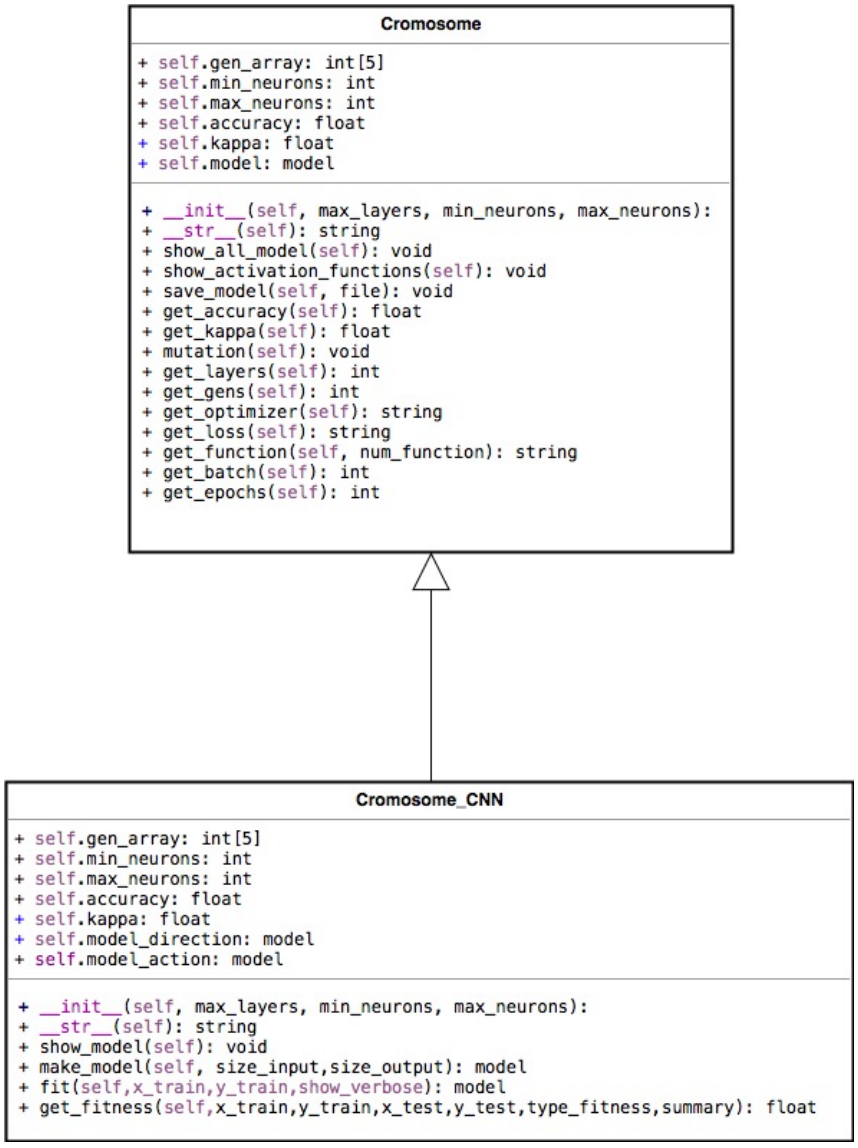
- **output_step_data(self):** mostra diverses dades de la xarxa per pantalla: el mode, els passos, l'epsilon i la probabilitat d'aleatorietat.

- **_initialize_model(self):** crea el model de la xarxa neuronal. Podem escollir si crear una xarxa neuronal per defecte o bé crear-ne una aleatòriament dins d'uns límits. La funció retorna el model creat.

- **__generate_action_space_combination_input_mapping(self, input_mapping):** crea un diccionari amb totes les possibles combinacions de tecles que la xarxa pot prémer.

8.3.2 Cromosome i CromosomeCNN

Aquesta classe representa un cromosoma d'un algorisme genètic, explicats en l'apartat 5.5.



La classe Cromosome_CNN es tracta d'una versió de la classe Cromosme però amb xarxes convolucionals. Aquesta serà la classe que s'utilitza per crear diferents xarxes neuronals i comprovar quina és la més òptima per a resoldre el problema al qual ens enfrontem, en aquest cas el nivell del joc.

Les variables d'aquestes classes són:

- **self.gen_array:** guardem l'esquelet de la xarxa neuronal
- **self.min_neurons:** neurones mínimes que tindrà cada capa.
- **self.max_neurons:** neurones màximes que tindrà cada capa.
- **self.accuracy:** precisió de la xarxa.
- **self.kappa:** coeficient.
- **self.model_direction:** model que afecta les tecles de moviment.
- **self.model_action:** model que afecta les tecles d'acció.

Les funcions d'aquestes classes són:

- **__init__(self,max_layers,min_neurons,max_neurons):** Es determinarà l'esquelet de la xarxa neuronal de forma aleatòria dins dels límits establerts als paràmetres d'entrada, és a dir, les capes que tindrà i el número de neurones per capa.
- **__str__(self):** mostra el nombre de capes i les neurones de cada capa de la xarxa neuronal per pantalla.
- **show_all_model(self):** mostra el nombre de capes, les neurones de cada capa de la xarxa neuronal per pantalla, les funcions d'activació i l'optimitzador.
- **show_activation_functions(self):** mostra les funcions d'activació de cada element de la xarxa.
- **save_model(self, file):** guarda el model a disc**.**
- **get_accuracy(self):** retorna la precisió.
- **get_kappa(self):** retorna el coeficient de kappa.
- **mutation(self):** redefineix un dels paràmetres modificables de la xarxa.

- **get_layers(self):** retorna el nombre de capes de la xarxa.
- **get_gens(self):** retorna lA taula on hi ha l'esquelet de la xarxa guardat.
- **get_optimizer(self):** retorna l'optimitzador utilitzat.
- **get_loss(self):** retorna la funció de cost utilitzada.
- **get_function(self, num_function):** retorna la funció d'activació utilitzada.
- **get_batch(self):** retorna el nombre de proves que farà abans d'actualitzar els paràmetres interns.
- **get_epochs(self):** retorna el nombre d'iteracions que es faran sobre totes les dades de prova

8.3.3 serpent_Cuphead_game

La idea d'aquesta classe és representar un videojoc dins el framework Serpent.AI.

Les responsabilitats d'una instància d'aquesta classe són:

- Iniciar el joc mitjançant la classe GameLauncher,
- Recollir i emmagatzemar informació de la finestra de joc.
- Iniciar i aturar instàncies de la classe FrameGrabber, la qual treballa sobre els frames de la pantalla.
- Alimentar la classe SerpentCupheadGameAgent (l'agent de joc) a partir d'instàncies de la classe GameFrame i controlar la velocitat a la qual es fa.
- Identificar i processar els sprites que es troben en el directori determinat i segueixen un model de nom determinat.
- Donar informació sobre les regions d'interès determinades als frames.

Frame Grabbing

Quan s'inicia una instància de la classe serpent_Cuphead_game, també s'inicia una instància de la classe FrameGrabber com a un procés separat. La seva funció serà capturar frames a una freqüència constant i guardar les imatges a memòria. Aquesta s'eliminarà tan bon punt la classe serpent_Cuphead_game sigui eliminada.

Sprites

Podem haver extret sprites prèviament de forma que la instància pugui treballar amb ells. Aquests han d'estar guardats al directori files/data/sprites. Haurà de seguir també un model de nom preestablert el qual funcionarà de la següent manera (sprite_<nom_de_l'sprite>_<index_de_l'animacio>.png), d'aquesta manera quedaran automàticament registrats i instanciats com a objectes de la classe Sprite i guardats en la variable self.sprites. Això servirà per identificar i/o localitzar-los a partir de l'agent del joc , s'explica més endavant.

Screen Regions

Introduint les coordenades en el codi, podem tenir controlades regions dels frames i, gràcies a això, obtenir informació d'alta rellevància.

Les variables són les següents:

- **self.config:** s'hi guarda el nom de la classe que hereta d'aquesta. És a dir, SerpentCupheadGame.
- **self.platform:** plataforma des de la qual utilitzem el joc.
- **self.input_controller:** conté el controlador d'inputs del joc.
- **self.window_id:** identificador de la finestra.
- **self.window_name:** nom de la finestra.
- **self.window_geometry:** contindrà la geometria de la finestra de joc.
- **self.window_controller:** hi guardarem el controlador de la finestra de joc.
- **self.is_launched:** serà cert quan el joc estigui en marxa, altrament serà fals.
- **self.frame_grabber_process:** subprocess que capturarà els frames de la pantalla.
- **self.frame_transformation_pipeline_string:** contindrà un string que indicarà si s'ha de dur a terme alguna transformació al frame capturat.
- **self.game_frame_limiter:** limitador de frames per segon.
- **self.api_instance:** guarda una instància de la classe api corresponent al joc.
- **self.sprites:** conjunt d'sprites definits prèviament.
- **self.redis_client:** sistema de memòria el qual ens servirà per guardar frames.
- **self.kwargs:** hi guardarem els **kwargs. Es tracta d'un conjunt d'elements identificables a partir d'una paraula clau o "keyword".

Les funcions són les següents:

- **game_launcher(self):** retorna la plataforma des de la qual s'executa el videojoc (pot ser Steam, executable o web)

- **game_launchers(self):** retorna un diccionari amb les 3 formes possible d'executar un videojoc

- **screen_regions(self):** Regions de pantalla que són característiques de cada videojoc,i per tant a la classe Game no està implementat. En la classe serpent_Cuphead_game retorna les regions definides amb les coordenades respectives.

- **api(self):** retorna una instància de l'api.

- **is_focused(self):** retorna cert en el cas que l'usuari tingui el focus en la finestra de joc.

- **launch(self, dry_run):** inicia el joc.

- **before_launch(self):** en el nostra cas no s'utilitza però si es volgués dur a terme alguna operació abans d'iniciar el joc, s'hauria de fer d'indicar dins d'aquesta funció.

- **after_launch(self):** un cop s'ha inicialitzat el joc, es posa el focus sobre ell, es situa a dalt a l'esquerra de la pantalla (coordenades (0,0)) i s'agafa la geometria de la finestra.

- **play(self, game_agent_class_name, frame_handler, **kwargs):** El primer que fa és buscar si el joc sobre el qual estem treballant té una classe game_agent. En el cas que n'hi hagi, s'activarà tot el procés de captura de frames i inicialització de l'agent.

- **extract_window_geometry(self):** retorna la geometria de la pantalla.

- **start_frame_grabber(self, pipeline_string):** inicialitza la captura de frames.

- **stop_frame_grabber(self):** para la captura de frames.

- **grab_latest_frame(self, frame_type):** obté l'últim frame reproduït.

- **_discover_sprites(self):** retorna un diccionari amb tots els sprites que compleixin les regles esmentades anteriorment en l'apartat sprites d'aquest mateix punt.

- **_handle_signal(self, signum, frame, do_exit):** el procés frameGrabber envia un senyal

8.3.4 serpent_Cuphead_game_agent

La idea d'aquesta classe és interactuar amb el joc.

Les responsabilitats d'una instància d'aquesta classe són:

- Utilitzar els models de machine learning implementats.
- Teniu una instància de SpriteIdentifier que coneix tots els sprites registrats del joc.
- Tenir el control dels fotogrames de la pantalla de joc gràcies al frameHandler.
- Enviar inputs d'entrada al videojoc a partir de la instància de InputController.
- Mostra informació pel terminal de què està succeint a la pantalla.

Les variables són les següents:

- **self.frame_handlers:** guardarem el nom del frame_handler que utilitzem.

- **self.frame_handler_setups:** guardem el nom de les funcions de setup.
- **self.input_mapping:** diccionari que identifica cada tecla amb un string, s'utilitza per a la creació de la xarxa neuronal. Exemple: **"UP": [KeyboardKey.KEY_UP],**
- **self.key_mapping:** identifica amb el paràmetre nom de cada tecla l'string creat en el mapa anterior. Exemple: **KeyboardKey.KEY_UP.name: "UP",**
- **self.direction_action_space:** conté les tecles de direcció en format string.
- **self.action_space:** conte les tecles d'acció en format string.
- **self.number_cromosomes:** número de cromosomes que crearem.
- **self.number_selection:** nombre de cromosomes d'una generació que passaran a la següent.
- **self.cromosome_position:** nombre de cromosoma al qual ens trobem.
- **self.taxe_mutation:** taxa de mutació (un nombre entre 0 i 1).
- **self.array_cromosome:** llista de cromosomes.
- **self.array_fitness:** llista de fitness de cada cromosoma (en aquest cas és el temps que dura cada xarxa).
- **self.best_cromosome:** guardem el millor cromosoma en el cas que estiguem treballant amb algoritmes genètics.
- **self.best_acuracy:** guardem el millor temps fet per una xarxa neuronal en el cas que estiguem treballant amb algoritmes genètics.
- **self.dqn_direction:** aquí es troba la xarxa que s'encarregarà de controlar les tecles direccionals del joc.
- **self.dqn_action:** aquí es troba la xarxa que s'encarregarà de controlar les tecles d'acció del joc.
- **self.input_controller:** contindrà el controlador d'inputs i farem ús d'ell quan vulguem dur a terme una acció amb les tecles.

- **self.game_state:** es tracta d'un diccionari que conté diferents paràmetres sobre el que esta passant al joc: **“**health”, “run_reward_direction”, “run_reward_action”, “current_run”, “current_run_steps”, “run_predicted_actions", “last_run_duration", “last_run_distance", “record_time_alive", “record_distance_alive", “run_timestamp".

- **self.dqn_change:** variable booleana que serà certa quan hàgem de fer un canvi de xarxa neuronal (això només passarà quan estiguem fent ús de l'algorisme genètic)

Les funcions són les següents:

- **__init__(self, **kwargs):** inicialitza tots els paràmetres d’una instància gameAgent cridant la funció __init__ de la classe gameAgent, i també inicialitza les variables que equivalen a elements del joc, com ara les vides, el temps que ha estat viu, les passes fetes…

- **setup_play(self):** inicialitza les variables referents a la intel·ligència artificial, les xarxes neuronals, els algoritmes genètics, juntament amb les seves variables i l’InputController el qual decidirà les tecles que es poden prémer.

- **handle_play(self, game_frame):** es tracta de la classe més important de tot el treball. La seva funció és controlar tot el que passa mentre el joc està en marxa, es tractarà d'un loop finit, però el més probable és que no arribem mai al final i que per aturar farem manualment. Serà l’encarregada de saber en quin estat del joc ens trobem i, per tant dur a terme una acció o una altra. Per exemple en el cas que acabem d'acabar una partida, alimentarà la xarxa neuronal amb els nous frames i recompenses per a cada un. Si ens trobem dins la partida, mostrarà per pantalla elements que ens ajudaran a entendre el que està passant. També controlarà l’algorisme genètic i determinarà quan creem una nova població, quan canviem de cromosoma …

- **_reset_game_state(self):** reinicialitzarà les variables que tinguin únicament relació amb una partida, és a dir, la duració d’aquesta, la partida en la qual ens trobem, la probabilitat d'aleatorietat que té…

- **_measure_cuphead_hp(self, game_frame):** retorna la vida que té el personatge mentre estiguem en la partida.

- **_calculate_reward(self):** calcula la recompensa segons l’acció feta i la vida.

- **_new_dqn_system(self):** crea una nova xarxa neuronal a partir d'un cromosoma determinat.

- **sort_cromosomes(self,type):** ordena els cromosomes segons el temps que han durat, com més temps, millor posició.

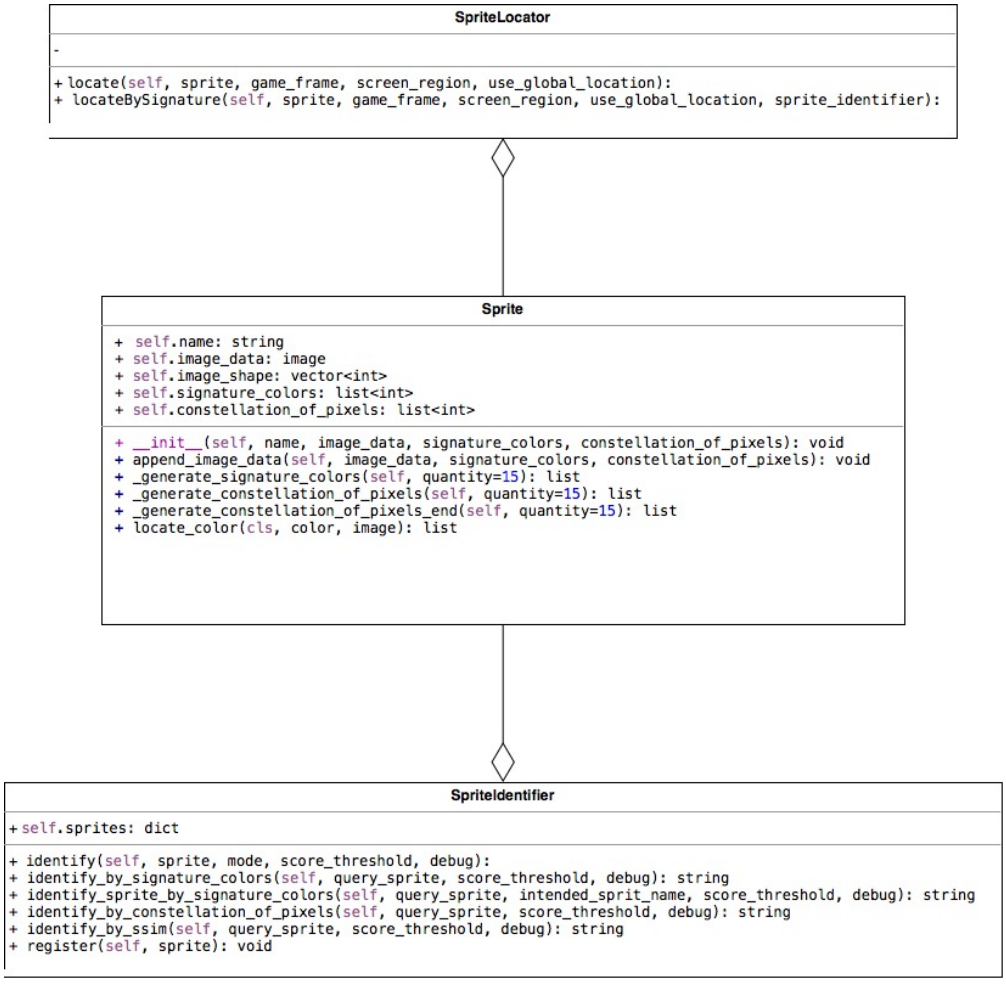
- **next_poblation(self):** crea una nova població de cromosomes a partir de la població inicial.

- **get_best_cromosome(self):** retorna el cromosoma amb un temps més elevat.

- **get_best_accuracy(self):** retorna el millor temps que han fet entre tots els cromosomes.

8.3.5 Sprite, SpriteIdentifier i SpriteLocator

La idea d'aquestes classes és treballar amb els sprites prèviament definits dins la carpeta files/data/sprites i que segueixin el model de nom preestablert (sprite_<nom_de_l'sprite> _<index_de_l'animacio>.png).



La classe Sprite els definirà com a tal i ens permetrà utilitzar-los dins el codi gràcies a les diferents variables que permetran treballar amb ells. La classe SpriteIdentifier tindrà la funció d'identificar un Sprite dels que hàgem definit prèviament, dins del joc. Per últim, la classe SpriteLocator tindrà la funció de buscar un Sprite determinat dins la pantalla de joc.

8.3.5.1 Sprite

Les variables són les següents:

- **self.name:** String amb el nom de l'Sprite.
- **self.image_data:** conté els valors dels píxels de la imatge.
- **self.image_shape:** mida de la imatge.
- **self.signature_colors:** conté una llista d'enters. Cada element correspon a un color. Es tracta d'un color significatiu, és a dir, que té un alt índex de presència en l'sprite.
- **self.constellation_of_pixels:** conté un diccionari que guarda els colors significatius de la variable anterior amb les posicions en la imatge.

Les funcions són les següents:

- **__init__(self, name, image_data, signature_colors, constellation_of_pixels):** inicialitza les variables corresponents a cada conjunt d'Sprites (entenem com a conjunt d'Sprites un mateix objecte i les seves diferents animacions).
- **append_image_data(self, image_data, signature_colors, constellation_of_pixels):** afegeix una imatge al conjunt.
- **_generate_signature_colors(self, quantity):** retorna la quantitat colors característics (els que apareixen més) d'un Sprite.
- **_generate_constellation_of_pixels(self, quantity):** crea un diccionari on trobem els colors característics i les seves posicions en l'Sprite.
- **_generate_constellation_of_pixels_end(self, quantity):**
- **locate_color(cls, color, image):** retorna les posicions de la imatge on s'ha trobat el color entrat per paràmetre.

8.3.5.2 SpriteIdentifier

Les variables són les següents:

- **self.sprites:** hi ha guardats tots els Sprites definits abans d'obrir el videojoc.

Les funcions són les següents:

- **identify(self, sprite, mode, score_threshold, debug):** retorna el nom de l'Sprite identificat a la imatge entrada per paràmetre "sprite" a partir del mètode entrat pel paràmetre "mode". Score_threshold marcarà el llindar a partir del qual acceptem que un Sprite és igual a un altre.

- **identify_by_signature_colors(self, query_sprite, score_threshold, debug):** retorna el nom de l'Sprite en el cas que l'hagi trobat utilitzant el mètode de "signature_colors". Buscarà els colors més significatius de cada element de la llista de Sprites dins la imatge querySprite. Si troba un percentatge de colors que supera el llindar establert a l'score_threshold, direm que es tracta del mateix sprite.

- **identify_sprite_by_signature_colors(self, query_sprite, intended_sprite_name, score_threshold, debug):** a diferència de l'anterior, només provarà si el "query_sprite" coincideix amb l'Sprite de nom "intended_sprite_name".

- **identify_by_constellation_of_pixels(self, query_sprite, score_threshold, debug):** retorna el nom de l'Sprite en el cas que l'hagi trobat utilitzant el métode de "signature_colors". Aquest busca que la posició i el color coincideixi en les dues imatges. Si troba un percentatge de coincidència que supera el llindar establert a l'score_threshold, direm que es tracta del mateix sprite.

- **identify_by_ssim(self, query_sprite, score_threshold, debug):** retorna el nom de l'Sprite en el cas que l'hagi trobat utilitzant el métode anomenat "ssim". Aquest és un mètode de comparació que es troba a la llibreria skimage.

- **register(self, sprite):** registrarem l'Sprite amb el nom que entrem al parametre "sprite".

8.3.5.3 SpriteLocator

Les funcions són les següents:

- **locate(self, sprite, game_frame, screen_region, use_global_location):** busca l'sprite dins el frame o regió de la pantalla introduïda, buscant per "pixel_constellation" (és a

dir, buscant que els “signature_colors” i la posició d'aquests siguin els mateixos en les dues imatges) i retorna la posició en el cas que l'hagi trobat.

- **locateBySignature(self, sprite, game_frame, screen_region, use_global_location, sprite_identifier):** fa el mateix que la funció anterior però només busca els “signature_colors”, si els troba ja diem que es tracta del mateix Sprite. És menys precisa, però en el cas que un mateix Sprite tingui moltes animacions, ens estalviarà molta feina

8.4 Disseny de l’aplicació

En aquesta secció plantejarem el pseudocodi de l’aplicació per assolir l’objectiu del treball. A més, també s’explicaran més al detall els atributs de les classes.

La idea principal del projecte era la creació d'una intel·ligència artificial que fos capaç d'aprendre a jugar i aconseguís testejar els diferents camins que permet en provar un nivell. Per fer això, el primer que s'havia de construir era una xarxa neuronal.

8.4.1 DQN i DDQN

El funcionament de la xarxa ja ha estat explicat al punt 5.8 i 5.9, però veure el pseudocodi anirà molt bé per acabar d'entendre com funciona.

El primer que voldrem fer quan creem una instància d’aquesta classe serà inicialitzar tots els paràmetres i crear una xarxa, o carregar-ne una ja definida.

8.4.1.1 Inicialitzar model

En el cas que vulguem crear una xarxa o vulguem fer ús de l'algoritme genètic, es durà a terme aquesta funció.

Aquesta funció el primer que farà, serà comprovar si estem fent ús d'un model genètic, si no és el cas, crearà una xarxa neuronal definida prèviament, crearà les neurones, les capes, les funcions d’activació... Un cop fet, es compilarà i es retornarà. En el cas que estiguem fent ús del model genètic, l'estructura de la xarxa neuronal que volem fer vindrà dins de la variable model_genetic i el que haurem de fer és iterar dins d'aquesta estructura i fer exactament el mateix. Un cop la tinguem creada, la compilarem i la retornarem.

```
FUNCIÓ _initialize_model(self):

  Si self.model_genetic és None LLAVORS : // crearem una xarxa neuronal per defecte.
        Creació de les capes i neurones del model, juntament amb les funcions d'activació.
        Creació del model a partir de les capes anteriors.
        Compilar Model
  ALTRAMENT : // fem ús de l'algoritme genètic
        Creació de les capes i neurones del model a partir del cromosoma genètic
        Creació del model a partir de les capes creades
        Compilar model
  FSI
    Retorna model m
  FFUNCIO
```

8.4.1.2. Carregar model de xarxa neuronal

En el cas que vulguem carregar una xarxa que tinguem guardada a l'ordinador utilitzarem aquesta funció.

El paràmetre "*file_path_model*" contindrà el directori i el nom del fitxer que conté el model, haurem de carregar-lo des d'aquí i indicar que volem aquesta xarxa en les variables "*model"* i "*model_online".* Un cop carregada, la compilarem i llegirem el fitxer que ens indica la variable "*file_path_weights*", el qual ens dirà els pesos corresponents a cada element que conforma el model. A més a més, aquest fitxer està guardat amb la peculiaritat que en el seu nom hi indicarem els passos fets i l'epsilon en el moment de guardat, això ho podrem utilitzar o bé utilitzar una epsilon nova.

```
FUNCIÓ.load_model_complet(self,model_base_file_path,model_file_path,override_epsilon):
        Llegir model des de "file_path_model"
        Compilem el model_online i el model
        Llegim el fitxer de pesos "file_path_weights"
        Actualitzem els pesos dels models per tal que siguin iguals
        Acutalitzem passos que hem obtingut a partir del fitxer de pesos
        SI override_epsilon és cert LLAVORS:
              Actualitzem valor d'epsilon
        FSI
        FFUNCIO
```

8.4.1.2 Carregar pesos de xarxa neuronal

En el cas que vulguem carregar només els pesos d'una xarxa utilitzarem aquesta funció. Llegirem el fitxer que ens indica la variable "*file_path*", el qual ens dirà els pesos corresponents a cada element que conforma el model. A més a més, aquest fitxer està guardat amb la peculiaritat que en el seu nom hi indicarem els passos fets i l'epsilon en el moment de guardat, això ho podrem utilitzar o bé utilitzar una epsilon nova.

```
FUNCIÓ load_model_weights((self, file_path, override_epsilon):

        Llegim el fitxer de pesos file_path
        Actualitzem els pesos dels models per tal que siguin iguals
        Acutalitzem passos que hem obtingut a partir del fitxer de pesos
        SI override_epsilon és cert LLAVORS:
            Actualitzem valor d'epsilon
        FSI
FFUNCIO
```

8.4.1.3 Actualitzar model de xarxa

Aquesta funció actualitzarà els valors del model amb els del model_online.

```
FUNCIO update_target_model(self):
        Escriure ("Updating target model...")
        model.actualitzarPesModel(self.model_online.get_weights())
FFUNCIO
```

8.4.1.4 Guardar pesos del model

També haurem de guardar els pesos de la xarxa i la xarxa en la qual treballem. Aquesta funció ens permet guardar els pesos, juntament amb els passos que portem i l'epsilon actual. Guardar la xarxa es fa simplement amb la funció .save(file_path).

```
FUNCIÓ save_model_weights(self, file_path_prefix=, is_checkpoint):
        epsilon := obtenirValorEpsilon
        SI is_checkpoint = Cert LLAVORS:
            file_path := "{file_path_prefix}_dqn_{passos}_{epsilon}_.h5"
        ALTRAMENT:
            file_path := "{file_path_prefix}_dqn_{epsilon}_.h5"
        Guardar pesos del model a disc al path (file_path)
FFUNCIO
```

8.4.1.5 Canviar a mode “train" i canviar a mode ”run"

Un cop tinguem la xarxa neuronal llesta, podem triar entre dos modes gràcies a les funcions, el mode run i el mode train.

El mode “train”, com indica el nom, serà el que utilitzarem per entrenar la xarxa i arribar al màxim d'escenaris possibles. Aquest triarà si l'acció a fer és aleatòria o no segons el valor que tingui la variable epsilon. En el cas que no sigui aleatòria, la xarxa neuronal farà la predicció de l'acció a fer. En el mode “run”, el 99% de les accions seran prediccions de la xarxa.

La funció enter_Train_mode el primer que fa és comprovar si hi ha un valor d’espsilon previ a l’actual, si no és així, simplement canvia la variable de mode a "TRAIN". En el cas que no sigui així, indicarem que la epsilon actual és igual a la prèvia, la prèvia passarà a ser nul·la i també canvia el mode a “TRAIN".

La funció enter_run_mode fa el contrari que l'anterior, guarda l'epsilon actual a la variable d'epsilon prèvia i posa l'actual a valor 0.01. Per últim, canvia el mode a “RUN"

```
FUNCIO enter_train_mode(self):
        SI self.previous_epsilon no és None LLAVORS: //Si teníem una epsilon prèvia, la
        utilitzem

                self.epsilon_greedy_q_policy.epsilon := self.previous_epsilon
                self.previous_epsilon := None
        FSI
        mode = “TRAIN”
FFUNCIO
```

```
FUNCIO enter_run_mode(self): //guarda l’epsilon que estava utilitzant i posa l’actual a
0.01 —> 1%
        self.previous_epsilon := self.epsilon_greedy_q_policy.epsilon
        self.epsilon_greedy_q_policy.epsilon := 0.01
        mode := “RUN"
FFUNCIO
```

8.4.1.6 Generar espai de combinacions d'inputs d'entrada

La sortida de la xarxa serà determinar quin conjunt de tecles haurà de prémer l'agent. Els conjunts de tecles que podrà prémer seran definits per aquesta funció, la qual retorna un diccionari amb totes les combinacions de tecles possibles.

El que farem serà iterar dins les combinacions de tecles disponibles. Per cada combinació calcularem les tecles en format string que haurem de prémer. Un cop calculat, direm que el valor d'entrada o input serà igual a cada tecla en format Keyboard.KEY dins el diccionari input_mapping equivalent a cada valor_combinacio dins valors_combinacions sempre que aquest sigui diferent a None. Un cop s'acabin les combinacions, iterarem dins valors_entrada i guardarem cada valor dins el diccionari d'accions, juntament amb la combinació corresponent.

```
FUNCIO _generate_action_space_combination_input_mapping(self, input_mapping):
        Crear diccionari d'accions
        PER combinacio DINS DE combinacions FER:
                valors_combinacios := valorsPerCombinacio(combinacio)
                valors_Entrada=input_mapping[valor_combinacio] PER CADA
                valor_combinacio DINS DE valors_combinacios SI valor_combinacio és
                diferent a None
        FIPER
        diccionari d'accions := llista(per cada combinacio de valors_entrada)
RETORNA diccionari d'accions
FFUNCIO
```

8.4.1.7 Escollir acció

Un cop tinguem tot el diccionari d'accions, haurem de prendre una acció cada vegada que juguem.

El primer que farem serà mirar si l'action_Type està definit, en el cas que no ho estigui el calcularem. Pot ser de dos tipus "RANDOM" o "PREDICTED". En el cas que es tracti d'una acció random, triarem una acció de la llista d'accions de forma aleatòria. Altrament, buscarem l'acció que ens retorni una recompensa màxima calculada a partir de l'últim conjunt de frames vist.

```
FUNCIO pick_action(self, action_type=None):
    SI action_type ÉS None LLAVORS:
        calcularTipusAccio()

    ALTRAMENT:
        tipus_accio = action_type
    FSI

    qs = model.PredirValorMaxim(self.frame_stack)

    SI tipus_accio = “RANDOM" LLAVORS:
      seleccionar accio aleatoriament
    ALTRAMENT SI tipus_accio = "PREDICTED":
      seleccionar accio calculada
FFUNCIO
```

8.4.1.8 Calcular tipus d'acció

Per saber si l’acció serà aleatòria o no simplement fem ús de l’epsilon. Com més proper a 1 sigui el valor, més probable serà una acció aleatòria.

```
FUNCIO compute_action_type(self):
        use_random := self.epsilon_greedy_q_policy.use_random() // calculem si fem random o no

        SI use_random=True LLAVORS
            tipus_accio := "RANDOM"
        ALTRAMENT
            tipus_accio := “PREDICTED"
FFUNCIO
```

8.4.1.9 Següent pas

Avancem el comptador de passes segons el mode en que ens trobem. Si ens trobem en mode “TRAIN" augmentarem els passos generals en 1, en el cas que ens trobem en mode "OBSERVE" augmentarem els passos d'observador en 1. El mode observador només està pensat per fer petites proves i si aquest supera un nombre de passos, el mode ja passarà a ser "TRAIN"

```
FUNCIO next_step(self):
        SI mode := “TRAIN” LLAVORS:
                passos += 1
        ALTRAMENT SI mode = “OBSERVE" LLAVORS:
                passos_actuals_observador += 1

        SI mode := "OBSERVE" I passos_actuals_observadors >= passos_observador
        LLAVORS:
                mode := “TRAIN"
FFUNCIO
```

8.4.1.10 Construir conjunt de frames

A part de dur a terme l'acció, també volem guardar els frames de quan l'hem dut a terme. Per això crearem un frame_stack o conjunt de frames.

```
FUNCIO build_frame_stack(self,game_frame):
        frame_stack:=Crear stack(game_frame)
FFUNCIO
```

8.4.1.11 Actualitzar conjunt de frames

Actualitza els frames de l’stack amb els frames del buffer.

```
FUNCIO update_frame_stack(self,game_frame_buffer):
        game_frames = [game_frame.frame PER CADA game_frame DINS
        game_frame_buffer.frames]
        frame_stack:=Crear stack(game_frames)

FFUNCIO
```

8.4.1.12 Afegir conjunt a memòria

Hem de guardar a memòria el conjunt de fames, juntament amb l'acció presa i la recompensa obtinguda utilitzant la següent funció.

Guardarem els frames actuals com a frames previs, actualitzarem els frames actuals, crearem una variable observation on posarem tots els valors que ens serveixen per entrenar la xarxa i afegirem aquesta variable a memòria

```
FUNCIO append_to_replay_memory(self, game_frame_buffer, reward, terminal):
        conjunt_previ_frames := conjunt_actual_frames
        actualitzarConjuntFrames(game_frame_buffer)

        observation=[conjunt_previ_frames, index_accio_actual, recompensa,
        conjunt_actual_frames, terminal]

       afegirAMemoria(observation)

FFUNCIO
```

8.4.1.13 Calcular error de l'acció

Per cada acció, haurem de calcular-ne l'error. La funció és diferent per DQN i DDQN, tal com s'ha explicat als punts 5.8 i 5.9, només canviarà el codi dins l'ALTRAMENT. Per entendre el pseudocodi anirà molt bé entendre la variable observation.

Aquesta variable ens serveix per guardar tots els elements a partir dels quals entrenarem una xarxa, hi guardem: conjunt de frames previs a l'acció, índex de l'acció presa, recompensa, conjunt de frames actuals, i si l'acció és terminal o no, és a dir, si la nostra vida arriba a 0.

```
observation = [
   previous_frame_stack, // conjunt de frames previs
   self.current_action_index, // index d'acció presa
   reward, // recompensa
   self.frame_stack, // conjunt de frames
   terminal // es tracta d'una acció terminal
]
```

El primer que farem serà predir el valor que obtindríem fent l'acció guardada a la variable observation i tenint en compte els frames previs. En el cas que es tracti d'una acció terminal, direm que el valor final és la recompensa de la variable observation. Altrament direm que el valor final és la recompensa més el valor que obtindríem en el cas que la xarxa predís l'acció a fer Un cop tenim aquests dos valors, buscarem el valor absolut entre els dos i aquest serà l'error.

```
DQN
FUNCIO calculate_target_error(self, observation):

        previous_target := predirValor(observation[0])[0][observation[1]]

        SI observation[4] LLAVORS:
                target := observation[2] // en el cas que sigui una acció terminal, agafa el
                valor de la recompensa.
        ALTRAMENT:
                target:=observation[2]+self.gamma*maxim(predirValor(observation[3])) //
                en el cas que no sigui acció terminal, agafa el valor de la recompensa +
                gamma*el valor retornat de la funció predict
        FSI
        RETORNA valorAbsolut(target - previous_target)
FFUNCIO
```

En el cas del DDQN es fa exactament el mateix però utilitzant les dues xarxes neuronals. Farem la predicció de la millor acció amb el model no actualitzat i busquem la recompensa que tindríem en el model actualitzat, per últim tornaríem a fer el valor absolut entre el valor previous_target i el target obtindríem el resultat.

```
DDQN
        ALTRAMENT:
                best_action = np.argmax(self.model_online.predict(observation[3])) // es
                calcula la millor acció amb el model no actualitza

                q = self.model.predict(observation[3])[0][best_action] // es calcula el valor
                amb el model actualitzat

                target = observation[2] + self.gamma * q // es calcula el valor com en
                l'anterior funció
        FSI
```

8.4.1.14 Reduir epsilon

Cada cop que hàgim fet una acció, i per tant entrenat una mica més la xarxa neuronal, disminuirem l'aleatorietat de les accions. Per fer això simplement disminuirem la variable epsilon.

```
FUNCIO erode_epsilon(self, factor):
        SI mode = "TRAIN":
                epsilon.reduir(factor)
        FSI
FFUNCIO
```

8.4.1.15 Informació de sortida

Aquesta funció ens permetrà conèixer l'estat de la xarxa mentre estigui en funcionament i ho mostrarà pel terminal.

Mostrarem el mode de joc, els passos que portem, en el cas que estiguem en mode observador mostrarem els passos als quals hem d'arribar per passar a mode observador i els passos que portem fins ara en mode observador. Mostrarem el valor d'epsilon arrodonit a 6 decimals i la probabilitat actual d'aleatorietat que és igual a l'epsilon multiplicada per 100.

```
FUNCIO output_step_data(self):
        Escriure("CURRENT MODE: {self.mode}")
        Escriure("CURRENT STEP: {self.current_step}")

        SI mode = “OBSERVE” LLAVORS:
                Escriure("CURRENT OBSERVE STEP: {self.current_observe_step}")
                Escriure("OBSERVE STEPS: {self.observe_steps}”)
        FSI

        Escriure("CURRENT EPSILON: {round(self.epsilon_greedy_q_policy.epsilon,
        6)}")
        Escriure("CURRENT RANDOM ACTION PROBABILITY:
        {round(self.epsilon_greedy_q_policy.epsilon * 100.0, 2)}%”)
        Escriure("LOSS: {self.model_loss}”)
FFUNCIO
```

8.4.1.16 Generar acció:

L'acció final serà agafar l'índex d'acció i buscar-lo dins de l'espai de combinacions.

```
FUNCIO generate_action(self):
   self.current_action := obtenirAccio(indexAccio)
FFUNCIO
```

8.4.1.17 Obtenir valors d'entrada :

Ens retornarà les tecles a apretar.

```
FUNCIO get_input_values(self):
        RETORNA self.action_input_mapping[self.current_action] // acció obtinguda gràcies a la funció anterior i obtinguent la seva correspondència dins l'input mapping.
FFUNCIO
```

8.4.1.18 Generar mini batch

Obtenim un conjunt de frames amb acció i recompensa de memòria.

```
FUNCIO generate_mini_batch(self):
        SI mode = "OBSERVE":
                RETORNA None
        FSI
        RETORNA exemplarDeMemoria(self.batch_size) // agafa un conjunt de frames de la memòria de mida self.batchsize=8
FFUNCIO
```

8.4.1.19 Entrenar xarxa a partir de mini batch(self):

Per últim haurem d'entrenar la xarxa. Per fer-ho crearem un conjunt de 32 fotogrames amb les seves corresponents recompenses i accions.

El primer serà obtenir un conjunt de frames del buffer, tot seguit agafarem 6 valors entre 0 i la mida del conjunt (8) i els guardarem a la variable flashback_indices. Iterarem dins tot el conjunt, si la posició del frame equival a un valor de flashback_indices, guardarem la imatge per mostrar-la pel debuggador. Tot seguit agafarem els valors de la variable

conjunt[i] que funcionen de la mateixa manera que la variable observation explicada al punt 8.4.1.13. Així doncs calcularem el valor que obtindriem fent ús de la xarxa sobre els frames “previous_frame_stack” i el valor fent ús de la xarxa sobre “frame_stack”. Un cop tenim això, mirem si es tracta d'un estat terminal o no. Si ho és, guardarem coma target[action_index] la variable reward, altrament target[action_index] serà igual a reward + el valor màxim de futuresRecompenses.

Per últim calcularem el valor absolut entre previous_target i target[action_index], actualitzarem la memòria amb aquest valor i entrenarem la xarxa.

Un cop creada la xarxa neuronal, ja es va provar d'entrenar el joc. Tot i això, no sabíem si la xarxa neuronal que estàvem utilitzant seria la més òptima per tal aconseguir el que es volia, i per tant vam optar per fer ús dels algoritmes genètics per optimitzar-la.

```
FUNCIO train_on_mini_batch(self):

    conjunt = generarConjunt()
   flashback_indices = randomEntre(self.batch_size), 6)
    PER i DINS DE rangEntre(0, longitud(conjunt)) FER:
        SI i DINS flashback_indices LLAVORS:
            guardarImatgePelDebuggador()
        FSI
        previous_frame_stack = conjunt[i][1][0] // agafem els valors del conjunt
        action_index = conjunt[i][1][1] // agafem els valors del conjunt
        reward = conjunt[i][1][2] // agafem els valors del conjunt
        frame_stack = conjunt[i][1][3] // agafem els valors del conjunt
        terminal = conjunt[i][1][4] // agafem els valors del conjunt

        target = predirValorModel(previous_frame_stack)
        previous_target = target[action_index]
        futuresRecompenses = predirValorModel(frame_stack)

        SI terminal és cert LLAVORS:
            target[action_index] = reward
        ALTRAMENT:
            target[action_index]=reward+self.gamma*obtenirValorMaxim(futuresRecompens
            es)
        FSI
        error = valorAbsolut(target[action_index] - previous_target)
        actualitzarMemoria(error)
        entrenarModel()
    FPER
FFUNCIO
```

8.4.2 Cromosome_CNN i Cromosome

La classe Cromosome_CNN representa una xarxa neuronal convolucional que és el tipus de xarxa que volem utilitzar. La classe Cromosome és una base de la qual heretaran altres classes per crear xarxes neuronals, per si sola només pot definir l'estructura però no crear el model.El funcionament de l'algoritme genètic està explicat a l'apartat 5.6.

8.4.2.1 Inicialitzar cromosoma

El primer que voldrem serà definir l'estructura de la xarxa. Per això entrarem per paràmetres el màxim de capes que pot tenir una xarxa i les neurones màximes i mínimes per capa. Abans faré una petita explicació de la variable taulaGenètica que serà la base d'aquestes classes.

La taulaGenètica és un array d'enters de mida 5 que ens definirà l'estructura de la xarxa neuronal.

- TaulaGenètica[0] ens indicarà el nombre de capes que tindrà la xarxa.

- TaulaGenètica[1] ens indicarà l'optimitzador de la xarxa, 0—>sgd, 1—>rmsprop, 2—>adagrad, 3—>adadelta, 4—> adam.

- TaulaGenètica[2] la funció de cost ón, 0—>categorical_crossentropy, 1—>binary_crossentropy, 2—>mean_squared_error, 3—>mean_absolute_error, 4 —> categorical_crossentropy.

- TaulaGenètica[3] ens indica el batch que és nombre d'exemples amb els quals treballa abans d'actualitzar la xarxa.

- TaulaGenètica[4] ens indica els epochs que són el nombre de vegades que l'algoritme iterarà sobre tots els exemples disponibles.

- TaulaGenètica[5….i] nombre de neurones que té cada capa. La primera capa correspon a TaulaGenètica[5], la segona a TaulaGenètica 6…

La funció inicialitzar cromosoma crearà aquesta taula. Per fer-ho utilitzarà la funció de creació de nombres aleatoris dins d'un rang determinat pels paràmetres d'entrada

```
FUNCIO __init__(self, max_layers = 5, min_neurons = 1, max_neurons = 10):
        NombreCapes := Random(1, max_layers)
        taulaGenetica := taulaZeros(dimensioTaula,tipus=enters)
        taulaGenetica[0] := NombreCapes
        taulaGenetica[1] := Random(0,4)
        taulaGenetica[2] := Random(0,4)
        taulaGenetica[3] := Random(2,32)
        taulaGenetica[4] := Random(1,50)

        PER i DINS  rangEntre(5, dimensioTaula):
                taualaGenetica[i] =  Random(min_neurons, max_neurons)
        FPER

        self.min_neurons := min_neurons
        self.max_neurons := max_neurons

        self.model := []
FFUNCIO
```

8.4.2.2 Variable en format text

Per tal d'obtenir tots els valors en format de text, iterarem sobre la taula i ho guardem a una variable.

```
FUNCIO __str__(self):
        cadena := ''
        PER i DINS rangeEntre(0, taulaGenetica[0]):
                cadena := cadena+ str(taulaGentica[i])
        FPER
        RETORNA cadena
FFUNCIO
```

8.4.2.3 Mostrar model

Si volem mostrar tot el model, haurem de fer la funció anterior i mostrar també les funcions d'activació i l'optimitzador.

```
FUNCIO show_all_model(self):
        cadena := ''
        PER i DINS rangeEntre(0, taulaGenetica[0]):
                cadena := cadena+ str(taulaGentica[i])
        FPER
        Escriure(cadena)
        mostrarFuncionsActivacio()
        Escriure(obtenirOptimitzador())
FFUNCIO
```

8.4.2.4 Mostrar funcions d'activació

Si volem mostrar les funcions d'activació, cal iterar dins la taula i obtenir les funcions d'activació de cada capa, les funcions dependran del nombre de neurones:

0 —> elu
1 —> relu
2 —> tanh
3 —> selu
4 o més —> sigmoid

```
FUNCIO show_activation_functions(self):
        cadena := ''
        PER i DINS rangeEntre(5, taulaGenetica[0]+5):
                fun = self.gen_array[i]
                cadena= cadena + obtenirFuncioActivacio(fun)
        FPER
        Escriure('Activation Function')
        Escriure(cadena)
FFUNCIO
```

8.4.2.5 Guardar model

Necessitem una funció per poder guardar el model. En el cas que vulguem guardar el model i no n'hi hagi cap, escriurem per pantalla “Model empty”, altrament el guardarem i per pantalla mostrar "model saved in disk”

```
FUNCIO save_model(self, file):
        SI model = [] LLAVORS:
                Escriure('Model empty')
        ALTRAMENT:
                model.guardar(file)
                Escriure(‘Model saved in disk')
FFUNCIO
```

8.4.2.6 Obtenir capes:

Retorna el nombre de capes que componen la xarxa

```
FUNCIO get_layers(self):
        RETORNA self.gen_array[0]
FFUNCIO
```

8.4.2.7 Obtenir estructura de la xarxa

Retorna la taula estructural de la xarxa.

```
FUNCIO get_gens(self):
        RETORNA self.gen_array
FFUNCIO
```

8.4.2.8 Obtenir optimitzador de la xarxa

Retorna l'optimitzador utilitzat a la xarxa. Agafarem el valor [1] de la taula genètica i segons ell retornarem un optimitzador o un altre.

0 —> ‘sgd’,
1 —> rmsprop’
2 —> 'adagrad'
3 —> 'adadelta'
4 —> 'adam'

```
FUNCIO handle_play(self):
      SI primerPartida LLAVORS:
          premerTecla(KeyboardKey.KEY_ENTER)
          primerPartida = False
      FSI
      SI conjuntDeFrames és None LLAVORS:
         conjuntDeFrames:=crearConjuntDeFrames()
       ALTRAMENT:
          actualitzarConjuntDeFrames(conjuntDeFrames)
          SI mode= “TRAIN" LLAVORS:
                   recompensa=obtenirRecompensa()
                  dqn_direction.afegirMemoria(conjuntDeFrames,recompensa)
                  dqn_action.afegirMemoria(conjuntDeFrames,recompensa)

                    #Cada 500 steps, guardem a disc
                     SI passes mod 500 = 0 LLAVORS:
                              guardarXarxa(dqn.action)
                              guardarXarxa(dqn.direction)
                      FSI
                     SI algoritmeGeneticActivat i passes mod 1500=0 LLAVORS
                              canviarXarxa:=Cert
                     FSI

          ALTRAMENT self.dqn_direction.mode == "RUN":
                    dqn_direction.actualitzarConjuntFrames(game_frame_buffer)
                    dqn_action.actualitzarConjuntFrames(game_frame_buffer)
         FSI
         escriureInformacioXarxa(dqn.action)
         escriureInformacioXarxa(dqn.direction)
        SI algoritmeGeneticActivat LLAVORS:
            esciureInformacioGenetica()
         FSI

         SI vida <= 0:
              self.game_state["last_run_duration"] = timestamp_delta.seconds
              Guardar valors de la partida

              SI mode == “TRAIN" LLAVORS:
                 PER i DINS rang(8) FER:
                    self.dqn_direction.train_on_mini_batch()
                    self.dqn_action.train_on_mini_batch()
                 FPER
              FSI
```

```
FUNCIO get_optimizer(self):
        optimitzador := self.gen_array[1]
        SI optimitzador = 0 LLAVORS:
                RETORNA 'sgd'
        ALTRAMENT SI optimitzador == 1 LLAVORS:
                RETORNA 'rmsprop'
        ALTRAMENT SI optimitzador == 2 LLAVORS:
                RETORNA 'adagrad'
        ALTRAMENT SI optimitzador == 3 LLAVORS:
                RETORNA 'adadelta'
        ALTRAMENT:
                RETORNA 'adam'
FFUNCIO
```

8.4.2.9 Obtenir funció de cost

Retorna la funció de cost de la xarxa. Agafarem el valor [2] de la taula genètica i segons ell retornarem una funció o una altra.

0 —> ‘categorical_crossentropy’,
1 —> ‘binary_crossentropy’
2 —> 'mean_squared_error'
3 —> 'mean_absolute_error'
4 —> 'categorical_crossentropy'

```
FUNCIO get_loss(self):
        loss := self.gen_array[2]
        SI loss = 0 LLAVORS:
                RETORNA ‘categorical_crossentropy'
        ALTRAMENT SI loss == 1 LLAVORS:
                RETORNA ‘binary_crossentropy'
        ALTRAMENT SI loss == 2 LLAVORS:
                RETORNA 'mean_squared_error'
        ALTRAMENT SI loss == 3 LLAVORS:
                RETORNA ‘mean_absolute_error'
        ALTRAMENT:
                RETORNA 'categorical_crossentropy'
FFUNCIO
```

8.4.2.10 Obtenir funció d'activació

Retorna la funció d'activació de la xarxa. Depenent del paràmetre num_function retornarà un funció o una altra:

0 —> elu
1 —> relu
2 —> tanh
3 —> selu
4 o més —> sigmoid

```
FUNCIO get_function(self,num_function):
        SI num_function = 0 LLAVORS:
                RETORNA ‘elu'
         ALTRAMENT SI num_function == 1 LLAVORS:
                RETORNA ‘relu'
         ALTRAMENT SI num_function == 2 LLAVORS:
                RETORNA 'tanh'
         ALTRAMENT SI num_function == 3 LLAVORS:
                RETORNA ‘selu'
         ALTRAMENT:
                RETORNA 'sigmoid'
FFUNCIO
```

8.4.2.11 Obtenir batch

Retorna el batch de la xarxa. Indica el nombre d'exemples amb els quals treballa abans d'actualitzar la xarxa.

```
FUNCIO get_batch(self):
        RETORNA self.gen_array[3]
FFUNCIO
```

8.4.2.12 Obtenir epochs

Retorna els epochs de la xarxa. Indica el nombre de vegades que l'algoritme iterarà sobre tots els exemples disponibles

```
FUNCIO get_epochs(self):
        RETORNA self.gen_array[4]
FFUNCIO
```

8.4.2.13 Mutar xarxa

Aplica l'operació de mutació sobre un d'aquests paràmetres: nombre de neurones, funció d' activació, epochs, batch o funció de cost/loss.

La funció decidirà quin paràmetre de la xarxa modificarà de forma aleatòria entre 1 i la longitud de l'array -1. Un cop seleccionat l'índex del paràmetre simplement ens mourem dins de condicionals per veure de quin tipus de paràmetre tracta. Si és més gran de 4, es tracta d'una capa i canviarem el nombre de neurones d'aquesta, si és igual a 3, canviarem el nombre de batch amb un random entre 1 i 32. Si és 4, canviarem el nombre d'epochs amb un random entre 1 i 50, per últim, si l'índex és 2 o 1 farem un random entre 0 i 4.

```
FUNCIO mutation(self):
        pos = randomEntre(1,len(self.gen_array)-1)
         SI pos > 2 LLAVORS:
                SI pos > 4 LLAVORS:
                        self.gen_array[pos]=randomEntre(self.min_neurons,
                        self.max_neurons)// canvia el nombre de neurones d'una capa
                ALTRAMENT:
                        SI pos == 3 LLAVORS:
                                self.gen_array[pos] = randomEntre(1,32)// canvia el
                                nombre de batch.
                        ALTRAMENT:
                                self.gen_array[pos] = randomEntre(1,50) // canvia el
                                nombre de epochs.
                        FSI
                FSI
        ALTRAMENT:
                self.gen_array[pos] = randomEntre(0,4)
        FSI
FFUNCIO
```

8.4.2.14 Crear xarxa a partir de l'estructura

No l'utilitzarem, ja que el model el crearem des de la classe DQN o DDQN. Tot i això el codi és exactament el mateix vist al punt 8.4.1.1 i el podríem fer servir des d'aquesta classe si volguéssim.

```
FUNCIO make_model(self, size_input, size_output, show = 0):
        Creació de les capes i neurones del model a partir del cromosoma genètic
        Creació del model a partir de les capes creades
        Compilar model Creació del model a partir de les capes creades
        Compilar model
  Retorna model
 FFUNCIO
```

8.4.2.15 Mostrar model(self):

Mostra el model un cop ja ha estat creat

```
FUNCIO show_model(self):
        self.model.summary()
FFUNCIO
```

8.4.3 serpent_Cuphead_game_agent

Per últim toca ajuntar tota la feina feta anteriorment i combinar-ho. Això ho farem dins la classe serpent_Cuphead_game_agent, l'encarregada de controlar-ho tot.

8.4.3.1 Configurar joc:

Aquesta serà la funció de configuració i serà la primera que farem només començar. Aquí definirem les xarxes neuronals a utilitzar, els controls que podrà utilitzar la xarxa per jugar, diferents variables relacionades amb l'algoritme genètic i una variable que ens indicarà si farem ús d'aquest algoritme genètic o només entrarem una xarxa.

```
FUNCIO setup_play(self):
      Crear mapes d'entrada
       algoritmeGeneticActivat=True
       iteracionsGenetiques=5
       nombreCromosomes = 7
       nombreDeSeleccionats = 3
       posicioCromosoma= 0
       taxaMutacio = 0.25
       taulaCromosomes = []
       taulaTemps = []
       millorCromosoma = []
       millorTemps = 0
       canviarXarxa=False
     PER i DINS  rangEntre(0, nombreCromosomes) FER:
```

```
        c = crearCromosoma
        taulaCromosomes.Afegir(c)
    FPER
    self.dqn_direction = CrearXarxaDDQN()
    self.dqn_action = CrearXarxaDDQN()
    SI algoritmeGeneticActivat LLAVORS:
        taulaCromosomes[0].model_action=self.dqn_action
        taulaCromosomes[0].model_direction=self.dqn_direction
    FSI
    poisicioCromosoma+=1
    self.dqn_action.entrarModeEntrenament()
    self.dqn_direction.entrarModeEntrenament()
FFUNCIO
```

8.4.3.2 Controlar joc

Aquesta funció serà l'encarregada d'ajuntar tot el que hem parlat fins ara.

El primer que farà serà comprovar si estem en la primera partida per prémer la tecla Enter, per així començar a jugar. Crearem un conjunt de frames en el cas que no hi sigui i si existeix, l'actualitzarem. Si estem en mode "TRAIN", calcularem la recompensa en aquests frames i ho guardarem en una variable. A més, afegirem aquest conjunt (frames, acció, recompensa) a memòria de les xarxes.

Si ens trobem en una passa mod 500=0, guardarem les xarxes a disc. Si és mod 1500=0 i estem utilitzant l'algoritme genètic, posarem la variable canviarXarxa a cert. En el cas que estiguéssim en mode "RUN" simplement actualitzarem el conjunt de frames de les xares. Després d'això, mostrarem informació referent a la xarxa i al joc per pantalla, en el cas que també fem ús de l'algoritme genètic, també mostrarem informació sobre aquest. Quan la vida sigui 0 s'acaba la partida, guardarem els resultats d'aquesta i si ens trobem en mode "TRAIN" farem 8 vegades la funció d'entrenament explicada al punt 8.4.19. Posarem totes les variables de la partida a 0. En el cas que estiguem en mode "TRAIN" o "RUN" i el nombre de partides sigui mod 20 = 0 i diferent de 0, actualitzarem les xarxes neuronals i farem una partida en mode "RUN", altrament entrarem en mode "TRAIN".

Tot seguit, si la variable canviarXarxa és certa, si la posició del cromosoma és inferior a 7, crearem noves xarxes i treballarem sobre elles. Altrament, si les iteracions genètiques són diferents a 0 crearem una nova població i crearem noves xarxes, disminuirem el nombre d'iteracions genètiques restants en 1 i posarem la posició del cromosoma de 0. En el cas que les iteracions genètiques siguin 0, ja haurem acabat i guardarem el millor model.

Per últim, escollirem l'acció a fer, la farem i actualitzarem els paràmetres de les passes, és a dir, reduir l'aleatorietat, sumar 1 a les passes fetes.

```
  Posar variables de partida a 0
            SI mode és “TRAIN” o”RUN" LLAVORS:
               SI partidaActual > 0 partidaActual mod 20 == 0 LLAVORS:
                  actualitzarModels()
                   dqn_direction.enter_run_mode()
                   dqn_action.enter_run_mode()
               ALTRAMENT:
                  dqn_direction.enter_train_mode()
                   dqn_action.enter_train_mode()
                FSI
            SI canviarXarxa LLAVORS:
               SI posicioCromosoma<7 LLAVORS:
                        crearNovesXarxes()
               ALTRAMENT:
                           SI iteracionsGenetiques=0 LLAVORS:
                                guardarMillorModel()
                           ALTRAMENT:
                              novaPoblacio()
                              posicioCromosoma=0
                              crearNovesXarxes()
                              iteracionsGenetiques=iteracionsGenetiques-1
                         FSI
                  FSI
            FSI
         accio=seleccionarAccio()
         inputController.fer(accio)
         Actualitzar paràmetres d’acció i passes
FFUNCIO
```

8.4.3.3 Mesurar vida

Per saber la vida que tenim, agafarem la regió del frame on apareix l'indicador de vida. Tot seguit farem una iteració per tots els sprites predefinits i els compararem la regió on apareix l'indicador de vida. Si la seva similitud és superior a la similitud màxima trobada fins aquell moment, guardarem aquest com a màxim i el nombre de vides ens ve indicat pel nom d'aquest sprite. Retornarem el nombre de vides.

```
 FUNCIO _measure_cuphead_hp(self, game_frame):
      hp_area_frame = extreureRegio(game_frame,[“HP_REGION”])
      vida = None
      max_similitud = 0

     PER sprite DINS self.game.sprites.items() FER:
           similitud = comparar(sprite,hp,_area_frame)

           SI similitud > max_ similitud LLAVORS:
              max_similitud = similitud
              cuphead_hp=obtenir valor del nom // el nombre de vides està al nom
               FSI

         FPER
        FPER
RETORNA cuphead_hp
FFUNCIO
```

8.4.3.4 Calcular recompensa

El sistema de recompenses ha estat una part bastant delicada. El joc no té una puntuació i només indica a quin punt del nivell s'ha arribat quan es perd. És per això que no es podia utilitzar com a sistema de recompensa que ja el mètode d’entrenament que utilitzem recau en un seguit de frames i la recompensa immediata.

Per això el sistema escollit ha estat el d'evitar rebre mal, és a dir, que la vida no disminueixi. Si rebem mal, rebrem recompensa negativa. Si no rebem mal, la recompensa serà positiva.

```
 FUNCIO calculate_reward(self):
  recompensa := 0
  SI vidaEnEstat[0] < vidaEnEstat[1] LLAVORS:
       recompensa :=-4
  ALTRAMENT
       recompensa:=0.5
  FSI
  RETORNA recompensa, recompensa
FFUNCIO
```

8.4.3.5 Crear nou sistema DQN

Només s'utilitzarà en el cas que estiguem fent ús de l'algoritme genètic. Serà l'encarregada de cridar als constructors de les noves xarxes a partir de l'estructura de cromosoma que toqui en aquell moment.

```
FUNCIO _new_dqn_system(self):
        self.dqn_direction = CrearXarxaDDQN()
        SI algoritmeGeneticActivat LLAVORS:
                taulaCromosomes[0].model_direction=self.dqn_direction
        FSI
        self.dqn_action = CrearXarxaDDQN()
        SI algoritmeGeneticActivat LLAVORS:
                taulaCromosomes[0].model_action=self.dqn_action
        FSI
        poisicioCromosoma+=1
        self.dqn_action.entrarModeEntrenament()
        self.dqn_direction.entrarModeEntrenament()
```

8.4.3.6 Nova població

Quan fem ús de l'algoritme genètic necessitarem crear descendència un cop acabem la iteració dins la primera població. Aquesta funció crea aquesta descendència i a més també inclourà els 3 millors cromosomes de la població anterior. Alguns descendents seran mutats, això dependrà de l'aleatorietat i la taxa de mutació.

El primer que farem serà ordenar les taules de temps i en funció d'aquest fer el mateix amb els cromosomes. Un cop ordenats, farem una selecció dels 3 millors. Tot seguit buidarem les taules de cromosomes i de temps. Crearem tants cromosomes nous com cromosomes hagi de tenir la població menys els seleccionats com a millors anteriorment.

Per últim mutarem els cromosomes, iterarem per tots els elements de la taula de cromosomes i farem un random entre 0 i 1. Si el valor que apareix és inferior a la taxa de mutació, mutarem el cromosoma altrament passarem al següent

```
FUNCIO next_poblation(self):

      //Seleccio millors cromosomes
    ordenarCromosomes()
    millorSeleccioCromosomes = taulaCromosomes[0:nombresSeleccionats]

    //Creació nous cromosomes
    taulaCromosomes = []
    taulaTemps = []
    total = nombreCromosomes - nombresSeleccionats
    PER i DINS rang(0, total) FER:
        c = crearXarxa()
        taulaCromosomes.afegir(c)
    FPER

    “Afegir cromosomes seleccionats"
    PER c DINS millorSeleccioCromosomes:
        taulaCromosomes.afegir(c)
     FPER

    “Mutem la població"
    index = 0
    PER c DINS self.array_cromosome FER:
        r = randomEntre(0, 1)
        SI r < taxa mutacio LLAVORS:
                c.mutar()
                taulaCromosomes[i] = c
                 index = index + 1
       FSI
    FPER
FFUNCIO
```

8.4.3.7_get_best_cromosome(self) i_get_best_accuracy(self):

Les dues funcions fan el mateix, iterar sobre tots els elements de la taula per buscar el que hagi fet un temps més elevat, per guardar-lo com a millor cromosoma.

```
FUNCIO get_best_cromosome(self):
        mida = longitud(taulaTemps) - 1
        MENTRE mida >= 0 FER:
           temps = taulaTemps[mida]
           c = taulaCromosomes[mida]
           SI temps > millorTemps:
              millorTemps = temps
              millorCromosoma = c
           mida=mida-1
          FMENTRE
RETORNA millorCromosoma // en el cas del best accuracy, es retorna millorTemps
```

8.4.3.8 ordenar cromosomes:

Aquesta funció la utilitzarem un cop hàgem acabat d'entrenar tota una població de cromosomes, els ordenarem per tal d'agafar els millors.

Per ordenar, farem un mentre des de la mida de la taula de fitness/temps fins a 0 sempre que la variable interchange sigui certa. Només entrar al bucle ja posarem aquesta variable com a falsa. Ara farem un per que iterarà sobre el nombre d'iteracions que quedin al mentre. Comprovarem si el temps de la posició i es superior al de la i+1, si és així, els canviarem i posarem la variable interchange a cert. Quan aquest bucle faci una iteració completa sense canviar cap element, la variable interchange quedarà igual a fals i voldrà dir que està ordenat.

```
FUNCIO  sort_cromosomes(self, type=1):
          interchange = True
           iterations = len(self.array_fitness) - 1
           MENTRE iterations > 0 I interchange FER:
              interchange = False
              PER i DINS rang(iterations) FER:
                 SI self.array_fitness[i] > self.array_fitness[i + 1] LLAVORS :
                    interchange = True
                    aux = self.array_fitness[i]
                    c_aux = self.array_cromosome[i]
                    self.array_fitness[i] = self.array_fitness[i + 1]
                    self.array_cromosome[i] = self.array_cromosome[i + 1]
                    self.array_fitness[i + 1] = aux
                    self.array_cromosome[i + 1] = c_aux
                  FSI
               FPER
             iterations -= 1
             FMENTRE
FFUNCIO
```

8.5 Diagrama de seqüència Usuari-Aplicació

8.5.1 Diagrama fent ús de l'algoritme genètic

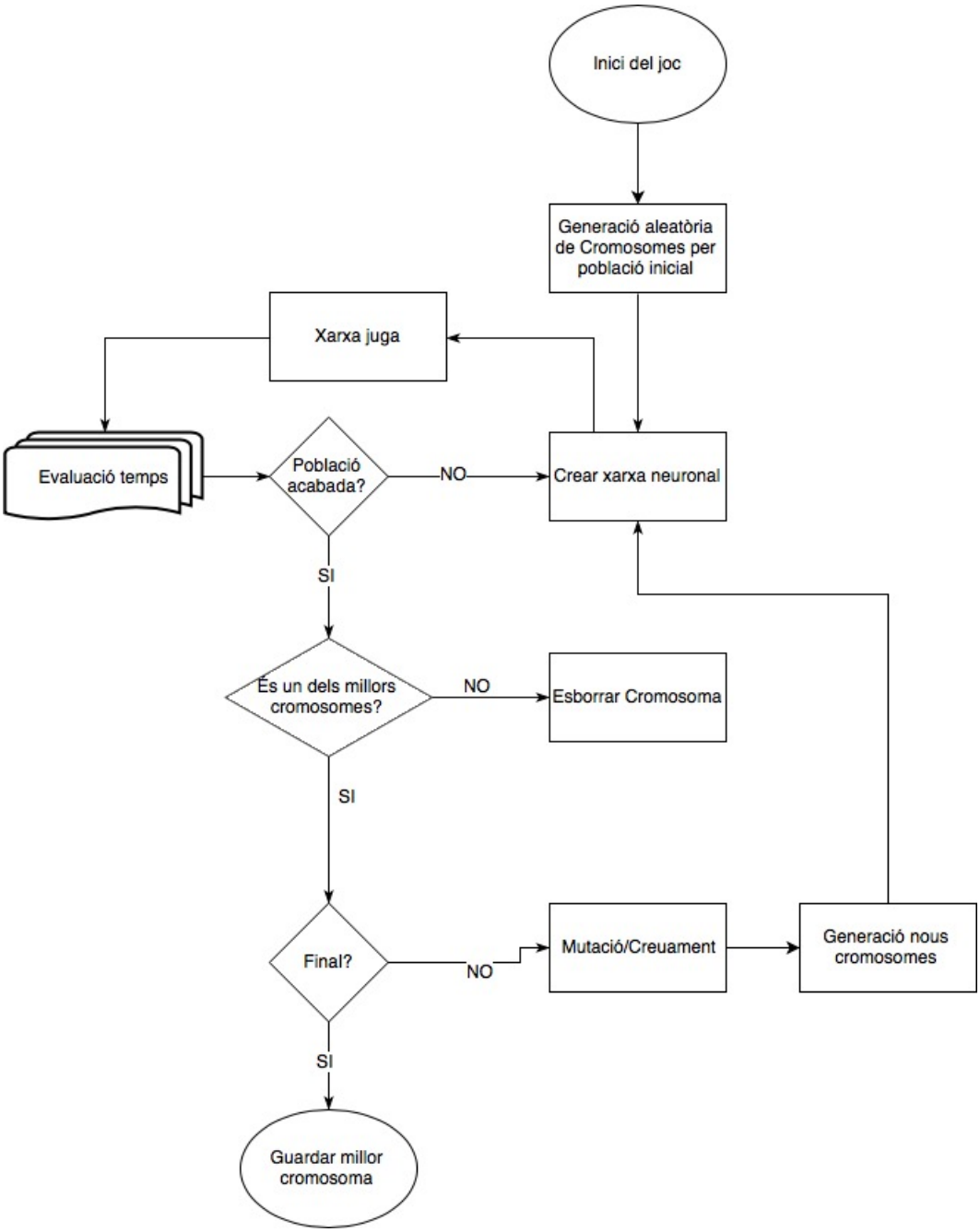


Figura 8.2. Diagrama de seqüència de l'algoritme genètic

1. L’usuari iniciarà el sistema a partir de la línia de comandes.
2. La classe serpent_Cuphead_game_agent crearà una població de cromosomes.
3. Cridarem el constructor de la classe CromosomeCNN i crearem tantes instàncies com nombre d'elements hagi de tenir la població.
4. Agafarem un cromosoma i crearem una xarxa DDQN a partir d’ell .
5. L'entrenarem i avaluarem el temps. (explicat al punt 8.5.2)
6. Si la població no s'ha acabat, tornem al punt 3 i avancem al següent cromosoma.
7. Si la població s'ha acabat, ordenem els cromosomes, seleccionem els millors i eliminem la resta.
8. Comprovem si és el final de l'algoritme genètic.
9. Si no ho és:
   9.1. Mutem els cromosomes.
   9.2. Creem nous cromosomes.
10. Si ho és, guardem el millor cromosoma a disc.

8.5.2 Diagrama entrenament xarxa neuronal

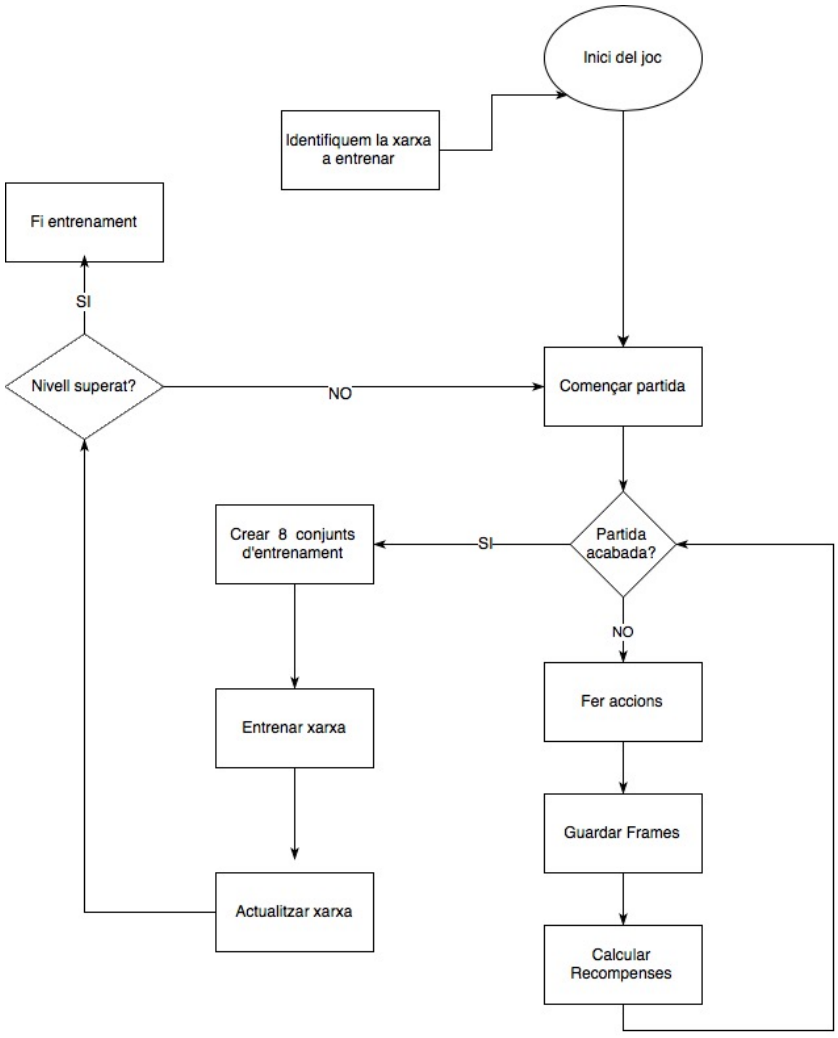


Figura 8.3. Diagrama de seqüència de l'entrenament de la xarxa neuronal

1. L’usuari determinarà la xarxa que entrenarà.
2. L’usuari iniciarà el sistema a partir de la línia de comandes.
3. Començarà la partida.
4. Mentre la partida no s'hagi acabat.
   4.1. La xarxa DDQN calcularà i farà les accions.
   4.2.Guardarem els frames a memòria.
   4.3.Calcularem les recompenses segons la vida.
5. Quan la partida s'hagi acabat.
   5.1.Crearem 8 conjunts d’entrenament que contindran el mateix que la variable observation explicada ala punt 8.4.1.13)
   5.2.Actualitzarem els valors de la xarxa
6. Si el nivell està superat, diem que l'entrenament ha acabat.
7. Altrament tornem a començar una partida.

Capítol 9 Implementació i proves

En aquest capítol s'explicaran els passos que s'han seguit per desenvolupar el codi, des de crear una xarxa neuronal fins a construir un algoritme genètic que vagi creant xarxes i sigui capaç de jugar al joc. Tenint en compte els pseudocodis del capítol anterior, desenvoluparem els codis en el llenguatge corresponent, veurem exemples pas a pas i s'explicaran les problemàtiques que poden anar sortint.

Seguirem l'ordre amb el que ens vam decidir enfrontar al projecte. Primer veurem implementacions la implementació del game_Agent. Tot seguit parlarem de l'enfocament principal, la idea era no enviar tot el frame a la xarxa neuronal sinó que volíem enviar les posicions dels sprites en pantalla, tot seguit veurem la creació d'una xarxa neuronal bàsica i el canvi d'enfocament a agafar tot el frame, per últim veurem la inserció de l'algoritme genètic.

9.1 serpent_Cuphead_game_Agent

Començarem parlant per la funció principal handle_play(self,game_frame). Per entendre-la millor l'anirem desglossant en petites parts.

9.1.1 handle_play(self,game_frame)

Recordem que l'agent és el controlador i per tant no haurem d'ajudar-lo en cap moment quan estigui entrenant la xarxa o quan faci ús de l'algoritme genètic. Això va molt bé per introduir la primera part de la funció, la qual mirarà si estem en la primera partida de totes. Si aquest és el cas, premerà la tecla Enter i adormirà el programa durant 4 segons. Aquests 4 segons és el temps de transició que necessita el joc per poder començar a jugar.

```
if self.dqn_direction.first_run: //si es tracta de la primera partida
    self.input_controller.tap_key(KeyboardKey.KEY_ENTER) //premèr enter

    self.dqn_direction.first_run = False // posem les variables de primera partida a fals
    self.dqn_action.first_run = False

    time.sleep(4) // adormim el programa 4 segons
    return None
```

En aquest moment ja estarem agafant els frames de la pantalla i una de les coses que volem obtenir d'aquest frame és la vida que tenim. La vida la guardarem com a una variable d'estat "self.game_state[**"health"**]" (dins self.game_state guardem totes les variables d'estat del joc, a mesura que vagin apareixent s'aniran explicant). Aquesta variable es tracta d'una cua de 2 elements, la raó d'això és per poder comparar la vida actual amb la del frame anterior.

```
cuphead_hp = self._measure_cuphead_hp(game_frame) //obtenim la vida
self.game_state["health"].appendleft(cuphead_hp) // afegim la vida a la cua
```

Mirarà si existeix un conjunt de frames, en el cas que no existeixi, cridarem la funció FrameGrabber.get_frames la qual retornarà un buffer que contindrà el frame amb índex 0 (el primer frame), de mida [self.game.frame_height self.game.frame_width] i amb un frame_type = "PIPELINE". Els frame_type del tipus "PIPELINE" han estat prèviament definits a la classe SerpentCupheadGame i en auqest cas el frame es veurà reduït a mida 100x100 i a escala de grisos per tal de que el cost computacional es redueixi dràsticament.

self.frame_transformation_pipeline_string = "RESIZE:100x100|GRAYSCALE"

Un cop fet això, construirem el frame stack, explicat en el punt 8.4.1.10. Li donem el mateix frame stack a les dues xarxes neuronals.

En el cas que ja hi hagi frame_stack, cridarem la mateixa funció per obtenir el buffer d'imatges però en aquest cas amb 4 frames d'índex [0,4,8,12].

```
if self.dqn_direction.frame_stack is None:
        pipeline_game_frame = FrameGrabber.get_frames(
            [0],
            frame_shape=(self.game.frame_height, self.game.frame_width),
            frame_type="PIPELINE"
        ).frames[0]

        self.dqn_direction.build_frame_stack(pipeline_game_frame.frame)
        self.dqn_action.frame_stack = self.dqn_direction.frame_stack

    else:
        game_frame_buffer = FrameGrabber.get_frames(
            [0, 4, 8, 12],
            frame_shape=(self.game.frame_height, self.game.frame_width),
            frame_type="PIPELINE"
        )
```

```
if self.dqn_change:
                self.array_fitness.append(self.game_state[‘record_time_alive'].get('value')
                )

                self.get_best_cromosome().model_direction.model.save(f”datasetsProve
                s9/MODELcuphead_direction_{self.cromosome_position-1}
                _temps_{self.game_state['record_time_alive'].get('value')}
                _geneticiteration_{self.genetic_iterations-6},)

                self.get_best_cromosome().model_action.model.save(f”datasetsProves9/
                MODELcuphead_action_{self.cromosome_position-1}
                _temps_{self.game_state['record_time_alive'].get('value')}
                _geneticiteration_{self.genetic_iterations-6}",)

            self.game_state["record_time_alive"] = {

                "value": 0,
                "run": 0,
                "predicted": self.dqn_direction.mode == "RUN"
            }

            if self.cromosome_position<7:
                        self._new_dqn_system()
            else:
                        if self.genetic_iterations==0:

                                    self.get_best_cromosome().model_direction.model.save("
                                    millorModelDirection.h5")

                                    self.get_best_cromosome().model_action.model.save("mill
                                    orModelAction.h5")
                                    self.cromosome_position=0
                                    self.next_poblation()
                        else:
                                    self.next_poblation()
                                    self.cromosome_position=0
                                    self._new_dqn_system()
                                    self.genetic_iterations=self.genetic_iterations-1

self.input_controller.tap_key(KeyboardKey.KEY_ENTER)
time.sleep(4)
```

Si estem en mode “TRAIN”, calcularem la recompensa explicada al punt 8.4.3.4 i ho sumarem a la recompensa total de la partida. Un cop fet això, ho afegirem a memòria utilitzant la funció append_to_replay_memory explicada al punt 8.4.1.12.

```
if self.dqn_direction.mode == "TRAIN":

            reward_direction, reward_action = self._calculate_reward()
            self.game_state["run_reward_direction"] += reward_direction
            self.game_state["run_reward_action"] += reward_action

            self.dqn_direction.append_to_replay_memory(
                game_frame_buffer,
                reward_direction,
                terminal=self.game_state["health"] == 0
            )

            self.dqn_action.append_to_replay_memory(
                game_frame_buffer,
                reward_action,
                terminal=self.game_state["health"] == 0
            )
```

Si ens trobem en una passa mod 500 = 0 guardarem les xarxes neuronals. En el cas que estiguem utilitzant l’algoritme genètic, el file_path_prefix indicarà el cromosoma i la població a la qual estem. Altrament no ho farà, a part també guardarem el fitxer de pesos.

```
        if self.dqn_direction.current_step % 500 == 0:
                if self.genetic_algoritm_activated:

                        file_path_prefix=f”datasetsProves12\cuphead_direction_{self.cro
                        mosome_position-1}_{self.genetic_iterations}”
                else:
                        file_path_prefix=f”datasetsProves12\cuphead_direction"


                self.dqn_direction.model.save(f”datasetsProves12\cuphead_direction_{s
                elf.cromosome_position-1}_{self.genetic_iterations}.h5”)

                self.dqn_direction.save_model_weights(
                        file_path_prefix,
                        is_checkpoint=True
                    )
```

```
                if self.genetic_algoritm_activated:

                        file_path_prefix=f”datasetsProves12\cuphead_action_{self.cromo
                        some_position-1}}_{self.genetic_iterations}”
                else:
                        file_path_prefix=f”datasetsProves12\cuphead_action"

                self.dqn_action.model.save(f”datasetsProves12\cuphead_action_{self.cr
                omosome_position-1}_{self.genetic_iterations}.h5")

                self.dqn_action.save_model_weights(
                        file_path_prefix,
                        is_checkpoint=True
                    )
```

Si tenim l'algoritme genètic activat i les passes mod 1500=0, ens indicarà que hem de canviar de xarxa, ho indicarem a la variable dqn_change que passarà a ser certa.

Tronant a l‘inici de la pàgina anterior, si en lloc d'estar en mode “TRAIN”, està en mode "RUN" simplement actualitzarem el conjunt de frames de cada xarxa.

```
        if self.genetic_algoritm_activated and self.dqn_direction.current_step%1500==0:
                    self.dqn_change=True

elif self.dqn_direction.mode == "RUN":

        self.dqn_direction.update_frame_stack(game_frame_buffer)

        self.dqn_action.update_frame_stack(game_frame_buffer)
```

Tot seguit ve molt codi que mostra informació per pantalla. És molt útil mirar-lo per entendre tot el que està passant mentre la xarxa s'està entrenant o està funcionant l'algoritme genètic.

El primer que farem serà saber el temps que portem en marxa, a la variable self.started_at hi tenim guardat el temps d’inici i el compararem amb datetime.now() que és el temps actual, d'aquí aconseguirem saber el temps que portem. Mostrarem també les xarxes a partir de les seves funcions d’output.

```
 run_time = datetime.now() - self.started_at

 print(f"SESSION RUN TIME: {run_time.days} days, {run_time.seconds // 3600} hours, {(run_time.seconds // 60) % 60} minutes, {run_time.seconds % 60} seconds")
print("")

print("DIRECTION NEURAL NETWORK:\n")
 self.dqn_direction.output_step_data()

print("")

print("ACTION NEURAL NETWORK:\n")
self.dqn_action.output_step_data()
```

En el cas que l'algoritme genètic estigui activat, mostrarem el cromosoma al qual ens trobem i la població. Seguirem mostrant la partida a la qual estem, la recompensa que tenim, les accions predites, la vida, la duració de l'última partida, la duració màxima i el mode de joc .

```
if self.genetic_algoritm_activated:
        print(f"CURRENT CROMOSOME: {self.cromosome_position}")
        print(f"CURRENT Iteration: {self.genetic_iterations-6}")

print(f"CURRENT RUN: {self.game_state['current_run']}")
print(f"CURRENT RUN REWARD: {round(self.game_state['run_reward_direction'] + self.game_state['run_reward_action'], 2)}")

print(f"CURRENT RUN PREDICTED ACTIONS:{self.game_state[‘run_predicted_actions']}")

print(f"CURRENT HEALTH: {self.game_state['health'][0]}")
print("")
print(f"LAST RUN DURATION: {self.game_state['last_run_duration']} seconds")
print("")

print(f"RECORD TIME ALIVE: {self.game_state['record_time_alive'].get('value')} seconds (Run {self.game_state['record_time_alive'].get('run')}, {'Predicted' if self.game_state['record_time_alive'].get('predicted') else 'Training'})")

print("")

print(self.dqn_direction.mode)
```

Si la vida és igual o inferior a 0, voldrà dir que la partida ha acabat. Netejarem el terminal i agafarem el temps actual. Comparem el temps actual amb el de l'inic de partida per obtenir la duració d'aquesta.

En el cas que estiguem en mode "TRAIN" o "RUN", comprovarem que el temps que hem fet no sigui superior al temps rècord. Si ho és, actualitzarem la variable rècord.

Reinicialitzarem la variable dels passos i li direm al inputController que no activi cap tecla. Ara entrenarem la xarxa en el cas que estiguem en mode "TRAIN". Iterarem 8 cops dins dels quals farem la funció train_on_mini_batch(). També informarem l'usuari si la següent partida és en mode "RUN", això passarà quan el nombre de partides mod 20=0.

```
if self.game_state["health"][0] <= 0:
        serpent.utilities.clear_terminal()
        timestamp = datetime.utcnow()
        timestamp_delta = timestamp - self.game_state["run_timestamp"]
        self.game_state["last_run_duration"] = timestamp_delta.seconds
        if self.dqn_direction.mode in ["TRAIN", "RUN"]:
                if(self.game_state[“last_run_duration"]>self.game_state["record_time_aliv
                e"].get("value", 0)):

                        self.game_state["record_time_alive"] = {
                            "value": self.game_state["last_run_duration"],
                            "run": self.game_state["current_run"],
                            "predicted": self.dqn_direction.mode == "RUN"
                        }

         self.game_state["current_run_steps"] = 0
        self.input_controller.handle_keys([])
        if self.dqn_direction.mode == "TRAIN":
                for i in range(8):
                        serpent.utilities.clear_terminal()
                        print(f"TRAINING ON MINI-BATCHES: {i + 1}/8”)

                        print(f"NEXT RUN: {self.game_state['current_run'] + 1} {'- AI RUN'
                        if (self.game_state['current_run'] + 1) % 20 == 0 else ''}")

                         self.dqn_direction.train_on_mini_batch()
                        self.dqn_action.train_on_mini_batch()
```

Seguirem amb la re-inicialització de paràmetres referents a la partida. Agafarem el nou temps d'inici, incrementarem en 1 el comptador de partides, posarem a 0 les recompenses, les accions predites i els dos elements de la cua de vida.

```
self.game_state["run_timestamp"] = datetime.utcnow()
self.game_state["current_run"] += 1
self.game_state["run_reward_direction"] = 0
self.game_state["run_reward_action"] = 0
self.game_state["run_predicted_actions"] = 0
self.game_state["health"] = collections.deque(np.full((2,), 0), maxlen=2)
```

Com s'ha dit anteriorment, si ens trobem en una partida mod 20=0, farem que sigui totalment predita per la xarxa i a més a més actualitzarem les xarxes neuronals. Altrament la farem en mode "TRAIN"

```
if self.dqn_direction.mode in ["TRAIN", "RUN"]:
        if self.game_state["current_run"] > 0 and self.game_state["current_run"]
        % 20 == 0:
                if self.dqn_direction.type == "DDQN":
                        self.dqn_direction.update_target_model()
                        self.dqn_action.update_target_model()
                self.dqn_direction.enter_run_mode()
                self.dqn_action.enter_run_mode()
        else:
                self.dqn_direction.enter_train_mode()
                self.dqn_action.enter_train_mode()
```

El següent que comprovarem és si la xarxa neuronal a de canviar, ho sabrem si la variable dqn_change és certa (això passarà quan portem 1500 passes). Afegirem el temps màxim que hagi aconseguit la xarxa a la taula de temps i posarem la variable d'estat de rècord a 0 per utilitzar-la amb la següent xarxa.

Mirarem que la posició del cromosoma sigui inferior a 7 (és la mida de la nostra població), si és així, cridarem a la funció _new_dqn_system i s'encarregarà de fer la nova xarxa a partir del següent cromosoma. Altrament, haurem de crear una nova població, indicar que el següent cromosoma serà el 0, crear una nova xarxa i disminuir el nombre d'iteracions restants en 1.

Independentment de la variable dqn_change, activarem la tecla Enter i esperarem 4 segons (temps que tarda a començar la partida).

Recordem que fins ara érem dins un condicional en el qual la nostra vida era 0, si no haguéssim entrat dins d'ell, ens trobem el següent.

Les dues xarxes neuronals seleccionant accions i generant-les. També afegirem la tecla de disparar i la del poder extra dins el conjunt de tecles que activarà, això ho fem per estalviar feina a la xarxa neuronal (la xarxa se centra a sobreviure amb el sistema de recompenses que li hem donat, si li donéssim recompensa per prémer la tecla de disparar i poder extra faria exactament el mateix que si les activem constantment). Un cop determinada l'acció, l'inputController la durà a terme. Si el tipus d'acció ha estat "PREDICTED" aleshores augmentar el comptador d'accions predites en 1. També reduirem l'epsilon en 10 i canviarem de passa.

```
        self.dqn_direction.pick_action()
        self.dqn_direction.generate_action()

        self.dqn_action.pick_action(action_type=self.dqn_direction.current_action_type)
        self.dqn_action.generate_action()

        keys = self.dqn_direction.get_input_values() + [KeyboardKey.KEY_X] +
[KeyboardKey.KEY_V]

        actions = self.dqn_action.get_input_values()
        print("")
        print(" + ".join(list(map(lambda k: self.key_mapping.get(k.name), keys + actions))))
        self.input_controller.handle_keys(keys)
        self.input_controller.tap_keys(actions)

        if self.dqn_direction.current_action_type == "PREDICTED":
            self.game_state["run_predicted_actions"] += 1
        self.dqn_direction.erode_epsilon(factor=10)
        self.dqn_action.erode_epsilon(factor=10)
        self.dqn_direction.next_step()
        self.dqn_action.next_step()
        self.game_state["current_run_steps"] += 1
```

9.1.2 setup_play(self)

Posarà a punt tots els elements que conformen el game_Agent juntament amb les xarxes neuronals i l'algoritme genètic si el fem servir.

El primer que farem serà definir l'input mapping que són les accions que podem fer i el key mapping que identifica el nom de cada tecla amb l'acció que farà. Definirem un espai de tecles de direcció i un espai de tecles d'acció. Realment treballarem sobre dues xarxes neuronals, una controlarà els inputs d'acció i l'altra els de direcció

```
def setup_play(self):

    self.input_mapping = {
        "UP": [KeyboardKey.KEY_UP],
        "LEFT": [KeyboardKey.KEY_LEFT],
        "DOWN": [KeyboardKey.KEY_DOWN],
        "RIGHT": [KeyboardKey.KEY_RIGHT],
        "UP-LEFT": [KeyboardKey.KEY_UP, KeyboardKey.KEY_LEFT],
        "UP-RIGHT": [KeyboardKey.KEY_UP, KeyboardKey.KEY_RIGHT],
        "DOWN-LEFT": [KeyboardKey.KEY_DOWN, KeyboardKey.KEY_LEFT],
        "DOWN-RIGHT": [KeyboardKey.KEY_DOWN, KeyboardKey.KEY_RIGHT],
        "SHOOT": [KeyboardKey.KEY_X],
        "JUMP": [KeyboardKey.KEY_Z],
        "EX": [KeyboardKey.KEY_V],
        "DASH": [KeyboardKey.KEY_LEFT_SHIFT]
    }
    self.key_mapping = {
        KeyboardKey.KEY_UP.name: "UP",
        KeyboardKey.KEY_LEFT.name: "LEFT",
        KeyboardKey.KEY_DOWN.name: "DOWN",
        KeyboardKey.KEY_RIGHT.name: "RIGHT",
        KeyboardKey.KEY_X.name: "SHOOT",
        KeyboardKey.KEY_Z.name: "JUMP",
        KeyboardKey.KEY_V.name: "EX",
        KeyboardKey.KEY_LEFT_SHIFT.name: "DASH"
    }
    self.direction_action_space = KeyboardMouseActionSpace(
            direction_keys=[None, "UP", "LEFT", "DOWN", "RIGHT", "UP-LEFT", "UP-
RIGHT", "DOWN-LEFT", "DOWN-RIGHT"]
    )
    self.action_space = KeyboardMouseActionSpace(
        action_keys=[None, "JUMP", "DASH", "EX"]
    )
```

Inicialitzarem les variables que ens seran necessàries.

self.genetic_algoritm_activated —> Booleà que indica si l'algoritme genètic està activat.
self.genetic_iterations —> Nombre de poblacions que crearà l'algoritme genètic.
self.number_cromosomes —> Nombre de cromosomes per població.
self.number_selection —> Nombre de millors cromosomes que passarem de població.
self.cromosome_position —> Posició de cromosoma que ens trobem.
self.taxe_mutation —> Taxa de mutació.
self.array_cromosome —> Taula de cromosomes.
self.array_fitness —> Taula de temps per cromosoma.
self.best_cromosome —> Millor cromosoma.
self.best_acuracy —> Millor temps.
self.dqn_change —> Variable que ens indicarà si hem de canviar cromosoma.

En el cas que l'algoritme genètic estigui activat, farem un per de 0 fins a self.number_cromosomes i crearem un cromosoma en cada iteració per afegir-lo a la llista de cromosomes.

```
self.genetic_algoritm_activated=True
self.genetic_iterations=5
self.number_cromosomes = 7
self.number_selection = 3
self.cromosome_position= 0
self.taxe_mutation = 0.25
self.array_cromosome = []
self.array_fitness = []
self.best_cromosome = []
self.best_acuracy = 0
self.dqn_change=False

if self.genetic_algoritm_activated:
        for i in range(0, self.number_cromosomes):
            c = Cromosome_CNN.Cromosome_CNN()
            self.array_cromosome.append(c)
```

Si volem carregar fitxers amb el model i amb pesos els escriurem a les variables direction_model_file_path i direction_model_base_file_path pel model de direcció. Ho farem en action_model_file_path i action_model_base_file_path pel model d'acció.

El següent a fer és crear les dues xarxes neuronals ja sigui a partir del cromosoma, o a partir d'un fitxer. En el cas que hàgim entrat un nom de fitxer incorrecte, el paràmetre quedarà buit i crearà una xarxa nova.

Si fem ús de l'algoritme genètic, guardarem el model creat dins del cromosoma. Recordem que quan creem un cromosoma, només indiquem l'estructura del model, no creem.

```
direction_model_file_path = "directori/nomFitxer.h5".replace("/", os.sep)
direction_model_base_file_path = "directori/nomFitxer.h5".replace("/", os.sep)
self.dqn_direction = DDQN(
        model_file_path=direction_model_file_path if
        os.path.isfile(direction_model_file_path) else None,
        model_base_file_path=direction_model_base_file_path if
        os.path.isfile(direction_model_base_file_path) else None,
        input_shape=(100, 100, 4),
        input_mapping=self.input_mapping,
        action_space=self.direction_action_space,
        replay_memory_size=5000,
        max_steps=1000000,
        observe_steps=1000,
        batch_size=32,
        model_learning_rate=2.5e-4,
        initial_epsilon=0.8,
        final_epsilon=0.1,
        override_epsilon=True,
        model_genetic=self.array_cromosome[0]
    )
if self.genetic_algoritm_activated:
        self.array_cromosome[0].model_direction=self.dqn_direction

action_model_file_path = "directori/nomFitxer.h5".replace("/", os.sep)
action_model_base_file_path = "directori/nomFitxer.h5".replace("/", os.sep)

self.dqn_action = DDQN(
        model_file_path=action_model_file_path if
        os.path.isfile(action_model_file_path) else None,
        model_base_file_path=action_model_base_file_path if
        os.path.isfile(action_model_base_file_path) else None,
        //mateixos paràmtres que en dqn.direction
          )
if self.genetic_algoritm_activated:
        self.array_cromosome[0].model_action=self.dqn_action
```

Per útil mostrarem les xarxes creades pel terminal, posarem les xarxes en mode entrenament i augmentarem en 1 la posició del cromosoma.

```
        self.dqn_direction.model.summary()
        self.dqn_action.model.summary()

        self.cromosome_position+=1
        self.dqn_action.enter_train_mode()
        self.dqn_direction.enter_train_mode()
```

9.1.3 Problemes i proves

El Serpent.AI és un framework que posa molt fàcil el desenvolupament d'agents i els problemes principals no han recaigut sobre aquesta classe. Tot i això, fins que no entès tot el funcionament i com està construït el sistema de variables, s'ha d'estudiar el codi i totes les classes. És molt útil entrar al GitHub del framework, ja que hi ha diversos exemples de jocs amb intel·ligències artificials que et permeten entendre com funciona tot.

```
def handle_play(self, game_frame):
        print("Hello World!")
```

Una de les primeres proves va ser que la funció handle_play mostrés Hello World!. És molt interessant, ja que cada vegada que el game_agent rebi un frame, ho mostrarà pel terminal. També apareix el concepte de quants frames pot rebre l'agent per segon. El joc es reprodueix a uns 30 FPS, d'aquests només n'agafarem 2, per tant durem a terme 2 accions per segon, que són 120 per minut que serà més que suficient en aquest joc.

Una altre prova molt interessant, és veure els frames que utilitzem i comprovar que es veuen correctament. Simplement obrint un altre terminal, escrivint la comanda "serpent visual_debugger" i afegint el següent codi. Veure la Figura 9.1

```
for i, game_frame in enumerate(self.game_frame_buffer.frames):
            self.visual_debugger.store_image_data(
                game_frame.frame,
                game_frame.frame.shape,
                str(i)
            )
```

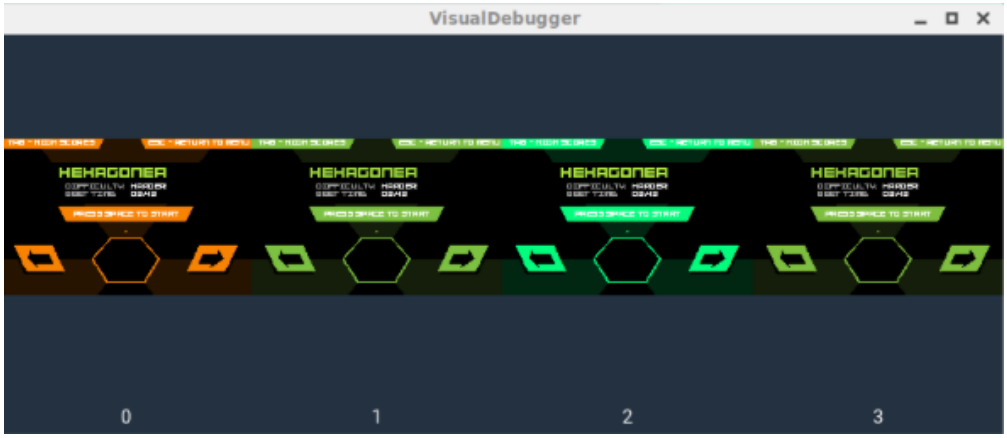


Figura 9.1 Visual Debugger

9.2 Sprite, SpriteIdentifier, SpriteLocator

D'aquest conjunt de classes només en farem ús per saber la vida que el personatge té. Per capturar els sprites vaig fer servir el mètode explicat al punt 7.1.2 mentre jugava. Un cop obtinguts, els vaig posar a la carpeta d'sprites amb el nom corresponent.

El nom ha d'estar escrit de la següent manera per tal, que la classe sprite treballi correctament sprite_<label>_<animation_state_index>.png

SPRITE640_HP0_0.png SPRITE640_HP2_0.png SPRITE640_HP3_0.png SPRITE640_HP11_0.png SPRITE640_HP12_0.png

Figura 9.2 Sprites de vida

Per tant simplement amb la classe Sprite podem aconseguir-ho.

9.2.1 _init__(self, name, image_data=None, signature_colors=None, constellation_of_pixels=None):

Aquesta funció ens permetrà utilitzar els sprites que hem guardat a la carpeta.

El primer que farà la funció és comprovar que la imatge sigui vàlida. La guardarà amb el seu nom corresponent i guardarà també la seva mida. Tot seguit buscarà els seus colors

significatius i la seva constel·lació de colors que són dos conceptes que explicaré juntament amb les seves funcions en els dos punts següents.

```
def __init__(self, name, image_data=None, signature_colors=None,
constellation_of_pixels=None):
    if not isinstance(image_data, np.ndarray):
        raise SpriteError("'image_data' needs to be a 4D instance
of ndarray...")

    if not len(image_data.shape) == 4:
        raise SpriteError("'image_data' ndarray needs to be
4D...")

    self.name = name
    self.image_data = image_data
    self.image_shape = image_data.shape[:2]

    self.signature_colors = signature_colors or
self._generate_signature_colors()

    self.constellation_of_pixels = constellation_of_pixels or
self._generate_constellation_of_pixels()
```

9.2.2_generate_signature_colors(self, quantity=15):

Cada imatge té una sèrie de colors, el que volem obtenir d'aquesta funció són els colors que més apareixen en l'sprite. Per fer-ho utilitzarem la llibreria numpy.

Per totes les animacions que tingui el personatge buscarem el nombre de cops que apareix cada color. Np.unique ens retorna el color i el nombre de cops que apareix. Ho ordenarem per cops que ha aparegut i guardarem l'índex del color de la taula values a la variable maximum_indices. Això ho farem fins a arribar al nombre d'elements marcat per quantity o bé quan hàgim acabat tots els sprites. Un cop acabat, buscarem els colors a partir dels seus índexs i els afegirem a la llista de signature_colors.

```
def _generate_signature_colors(self, quantity=15):
    signature_colors = list()
    height, width, pixels, animation_states =
self.image_data.shape

    for i in range(animation_states):
        values, counts = np.unique(self.image_data[...,
i].reshape(width * height, pixels), axis=0, return_counts=True)
```

```
        if len(values[0]) == 3:
            maximum_indices = np.argsort(counts)[::-1][:quantity]
        elif len(values[0]) == 4:
            maximum_indices = list()

            for index in np.argsort(counts)[::-1]:
                value = values[index]

                if value[3] > 0:
                    maximum_indices.append(index)

                    if len(maximum_indices) == quantity:
                        break

        colors = [tuple(map(int, values[index][:3])) for index in
maximum_indices]
        signature_colors.append(set(colors))

    return signature_colors
```

9.2.3 _generate_constellation_of_pixels(self, quantity=15):

Ara volem obtenir una posició de cada element de la llista de color significatius. Per fer-ho crearem una llista anomenada constellation_of_pixels, tornarem a iterar dins totes les animacions disponibles i afegirem un diccionari a la llista per a cada una. Dins de cada iteració, crearem una altra iteració de longitud igual a la variable quantity. En aquesta, agafarem un signature_color de forma aleatòria i buscarem en quines posicions apareix de l'sprite. Un cop ho tinguem, agafarem una posició aleatòria i afegirem la combinació de posició + color a la variable constellation_of_pixels.

Tot i això, la nostra idea principal era buscar tots els sprites per la pantalla de joc i obtenir la seva posició. D'aquesta manera hauríem alimentat la xarxa neuronal a partir de posicions en lloc del frame sencer.

```
def _generate_constellation_of_pixels(self, quantity=15):
    constellation_of_pixels = list()
    height, width, pixels,animation_states =
self.image_data.shape

    for i in range(animation_states):
        constellation_of_pixels.append(dict())
        for ii in range(quantity):
           signature_color=
           random.choice(list(self.signature_colors[i]))
```

```
            signature_color_locations =
            Sprite.locate_color(signature_color,
            np.squeeze(self.image_data[:, :, :3, i]))
             y, x = random.choice(signature_color_locations)
             constellation_of_pixels[i][(y, x)] = signature_color
    return constellation_of_pixels
```

Doncs la teoria era fàcil, havíem d'iterar cada frame per tota la pantalla buscant els sprites. El primer problema va ser l'obtenció dels sprites, agafar els 5 sprites de la vida és fàcil, ja que sempre estàn en la mateixa posició de la pantalla, però agafar sprites en moviment és molt més complicat,a més a més cada personatge és possible que tingui més de 20 animacions. Fàcilment hauríem hagut de capturar més de 140 imatges i separar-les del fons. Per sort la comunitat dins del món dels videojocs fa molta feina i vam trobar tots els sprites penjats a internet.

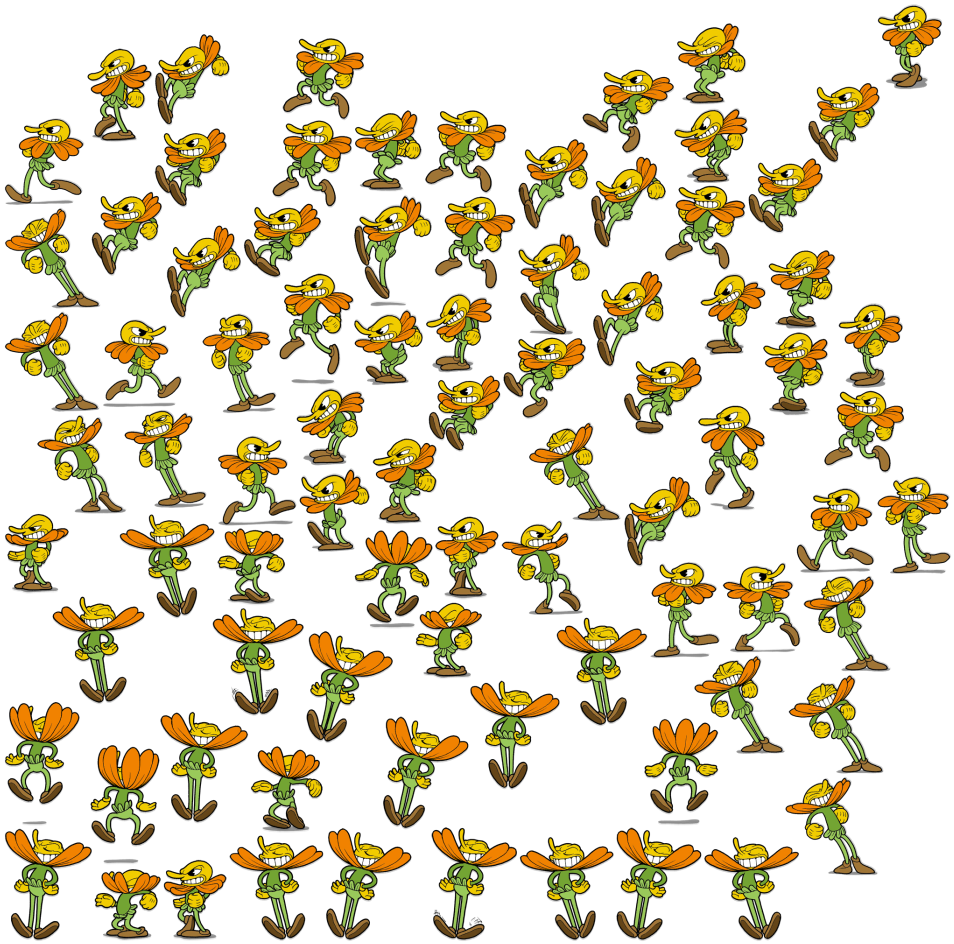


Figura 9.3 Animacions de l'enemic Girasol

El personatge de la figura 9.3 va ser el primer amb el qual vam treballar. Vam separar els sprites de la imatge original i els vam posar dins la carpeta corresponent per ser utilitzats dins el joc.Tot semblava anar bé fins que vaig provar de localitzar el Girasol mentre el joc estava en marxa, no el detectava.

La raó era perquè la funció locate(self, sprite, game_frame, screen_region, use_global_location) busca tots els constellation_pixels de l'sprite a la imatge i si n'hi ha un que no és exactament igual, ja el dóna per erroni.

9.2.4 locate(self, sprite, game_frame, screen_region, use_global_location):

No es tracta d'una funció que utilitzi i per tant l'explicaré a alt nivell. El que fa la funció és buscar l'sprite que hi haurà dins la variable sprite dins el game_frame que hi haurà a la variable game_Frame. Per fer-ho agafarà cada un dels colors dins els constellation_pixels de l'sprite i el buscarà per tota la imatge. En el cas que el trobi retornarà la posició on l'ha trobat i a partir d'aquesta posició buscarà que tots els altres colors de la llista de constellation of pixels siguin a la imatge i que la posició relativa dels colors sigui la mateixa tant en l'sprite com en la imatge. En el cas que tot això sigui cert, retornarà la posició de l'sprite en pantalla altrament retorna None.

```
locate(self, sprite=None, game_frame=None, screen_region=None,
use_global_location=True):

      constellation_of_pixel_images =
      sprite.generate_constellation_of_pixels_images()
      location = None

      frame = game_frame.frame

      if screen_region is not None:
            frame = serpent.cv.extract_region_from_image(frame,
            screen_region)

      for i in range(len(constellation_of_pixel_images)):
            constellation_of_pixels_item =
            list(sprite.constellation_of_pixels[i].items())[0]

            query_coordinates = constellation_of_pixels_item[0]
            query_rgb = constellation_of_pixels_item[1]


            rgb_coordinates = Sprite.locate_color(query_rgb,
            image=frame)


            rgb_coordinates = list(map(lambda yx: (yx[0] -
            query_coordinates[0], yx[1] - query_coordinates[1]),
            rgb_coordinates))
```

```
    for y, x in rgb_coordinates:
         if y < 0 or x < 0 or y > maximum_y or x > maximum_x:
                    continue
               for yx, rgb in
               sprite.constellation_of_pixels[i].items():
                 if tuple(frame[y + yx[0], x + yx[1], :]) != rgb:
                           break

                 else:
                       location = (
                           y,
                           x,
                          y +
          constellation_of_pixel_images[i].shape[0],
                              x +
          constellation_of_pixel_images[i].shape[1]
                          )

              if location is not None and screen_region is not
          None and use_global_location:
                   location = (
                       location[0] + screen_region[0],
                       location[1] + screen_region[1],
                       location[2] + screen_region[0],
                       location[3] + screen_region[1]
                   )
        return location
```

Hi havia dues maneres de solucionar això, aconseguint tots els sprites disponibles del joc o bé creant una funció que fos molt menys exigent.

9.2.5 locateBySignature(self, sprite, game_frame, screen_region,use_global_location,sprite_identifier):

Doncs aquesta va ser la manera escollida. El codi seria exactament el mateix que l'anterior però en el cas que trobem un constellation pixel,

El que farem serà agafar una regió de mida igual a l'sprite des del píxel que hem trobat i l'identificarem amb la funció *identify_sprite_by_signature_colors,* la qual compara els signature colors i no té en compte les posicions d'aquests. Per tant serà molt més flexible però la posició de l'sprite que retorni no serà exacte. Si aquesta funció retorna el nom d'un sprite, voldrà dir que l'ha trobat i retornarem la posició, altrament seguirem iterant pels constallation pixels fins a acabar-los i si no trobem res, retornarem None.

```
for y, x in rgb_coordinates:
    if y < 0 or x < 0 or y > maximum_y or x > maximum_x:
        break
    else:
     frame2 = serpent.cv.extract_region_from_image(frame, (y,x,y
     + constellation_of_pixel_images[i].shape[0],x +
     constellation_of_pixel_images[i].shape[1]))

     frame2=frame2[:,:,:,np.newaxis]

     query_sprite = Sprite("QUERY", image_data=frame2)

     sprite_name=sprite_identifier.identify_sprite_by_signature_
     colors(query_sprite,sprite.name,score_threshold=24,debug=Tr
     ue)  # Will be "UNKNOWN" if no match

     if sprite_name!="UNKNOWN":
            location = (
                y,
                x,
                y + constellation_of_pixel_images[i].shape[0],
                x + constellation_of_pixel_images[i].shape[1]
            )
            break

if location!= None:
    break
```

Aquesta funció ens va permetre aconseguir millors resultats, però encara teníem molts errors. Va ser en aquell moment que vam estudiar els valors dels píxels que obteníem per pantalla i vam veure que anaven canviant constantment encara que el personatge estigués quiet. El Cuphead utilitza un filtre el qual provoca blur i que no havíem tingut en compte, això fa que el color dels píxels canvií a cada frame.

Figura 9.4 Imatge en joc on podem veure efectes del filtre

Per solucionar això, vam tornar a fer ús de la comunitat del joc. A alguns usuaris aquest filtre els molestava molt i van crear un modificador que l'eliminava i deixava la imatge neta.

Un cop fet això, els resultats ja eren millors però no havíem ni començat a crear una xarxa neuronal després de tota aquesta feina. A més, l'eficiència del codi era nul·la, ja que iterava per tot el frame per buscar constellation pixels, després tornava a iterar per buscar l'sprite i això ho havíem de fer tants cops com sprites tinguéssim guardats.

Per tant teníem dues opcions de nou, redefinir tot el codi un altre cop per poder treballar amb sprites o que l'input de la xarxa fós tot el frame en lloc de les posicions dels sprites. En aquest cas vam optar per agafar tot el frame.

9.3 DDQN i DQN

Mostraré del codi de les dues funcions més interessants i que poden crear més dubte d'aquestes classes.

9.3.1 _initialize_model(self):

Aquesta serà la funció que crearà la xarxa neuronal. Utilitzarem codi de la llibreria Keras per a la creació de la xarxa.

El primer que farem serà definir la mida de l'input d'entrada que en aquest cas serà de 100x100 en blanc i negre. La funció la dividirem a partir d'un condicional el qual pregunta si estem utilitzant l'algoritme genètic o no.

Si no utilitzem l'algoritme genètic, l'estructura està totalment definida en el codi. Es tracta d'una xarxa neuronal que hem aconseguit que durés 152 segons dins el nivell en mode "RUN" és a dir, controlat totalment per la intel·ligència artificial.

La creació de la xarxa es basa a anar creant capes i ajuntar-les amb una sortida de mida igual a les tecles que pugui activar la xarxa.

```
def _initialize_model(self):

    input_layer = Input(shape=self.input_shape)
    if self.model_genetic == None:
```

```
    tower_1 = Convolution2D(16, 1, 1, border_mode="same",
    activation="elu")(input_layer)
    tower_1 = Convolution2D(16, 3, 3, border_mode="same",
    activation="elu")(tower_1)
    tower_2 = Convolution2D(16, 1, 1, border_mode="same",
    activation="elu")(input_layer)
    tower_2 = Convolution2D(16, 3, 3, border_mode="same",
    activation="elu")(tower_2)
    tower_2 = Convolution2D(16, 3, 3, border_mode="same",
    activation="elu")(tower_2)

    tower_3 = MaxPooling2D((3, 3), strides=(1, 1),
    border_mode="same")(input_layer)
    tower_3 = Convolution2D(16, 1, 1, border_mode="same",
    activation="elu")(tower_3)
    merged_layer =  Concatenate(axis=1)([tower_1, tower_2,
    tower_3])

    output = AveragePooling2D((7, 7), strides=(8, 8))
    (merged_layer)
    output = Flatten()(output)
    output = Dense(self.action_count)(output)

    m = Model(input=input_layer, output=output)
    m.compile(rmsprop(lr=self.model_learning_rate,
    clipvalue=1), "mse")
```

En el cas que fem ús de l'algoritme genètic, la xarxa que haurem de construir ens vindrà determinada pel cromosoma. Crearem un model seqüencial, iterarem des de 5 fins a 5 + el nombre de capes que tingui l'estructura del model. Agafarem les neurones de cada capa, la funció d'activació, el filtre i crearem capes de tipus Conv2D amb aquests paràmetres. Les capes diferents tornaran a ser la primera i l'última per la mida de l'input i la de l'output.

```
else:
    layers = self.model_genetic.gen_array[0]
    m = Sequential()
    for i in range(5, (layers+5)):
       num_neuronas = self.model_genetic.gen_array[i]
       function_activacion =
       self.model_genetic.gen_array[i+layers]
       filtro = self.model_genetic.filter
       if i == 5:
            conv = Conv2D(num_neuronas, (filtro, filtro),
            activation =
            self.model_genetic.get_function(function_activac
            ion), padding='same')(input_layer)
```

```
            else:
                  conv = Conv2D(num_neuronas, (filtro, filtro),
                  activation =
                  self.model_genetic.get_function(function_activac
                  ion), padding='same')(conv)
      conv = Flatten()(conv)
      output = Dense(self.action_count, activation='softmax')
      (conv)
      m = Model(input_layer, output)
              #m.add(Dense(size_output, activation='softmax'))

      m.compile(loss=self.model_genetic.get_loss(),optimizer=self
      .model_genetic.get_optimizer(),metrics=['accuracy'])
      m.summary()


     return m
```

9.3.2 train_on_mini_batch(self):

Aquesta funció està explicada al punt 8.4.1.19 es bastant simple d'entendre amb l'explicació feta anteriorment. El més interessant és que connecta tots els conceptes explicats també en el punt 5.7, 5.8 i 5.9

```
def train_on_mini_batch(self):
      mini_batch = self.generate_mini_batch()
      flashback_indices = random.sample(range(self.batch_size),
      6)
      sumError=0
            for i in range(0, len(mini_batch)):
                  if i in flashback_indices:
                        flashback_image = np.squeeze(mini_batch[i]
                        [1][3][:, :, :, 1])

                        self.visual_debugger.store_image_data(
                        flashback_image,
                        flashback_image.shape,
                        f"flashback_{flashback_indices.index(i) +
                        1}"
                        )

                  del flashback_image

      previous_frame_stack = mini_batch[i][1][0]
      action_index = mini_batch[i][1][1]
      reward = mini_batch[i][1][2]
```

```
        frame_stack = mini_batch[i][1][3]
        terminal = mini_batch[i][1][4]
        target = self.model_online.predict(previous_frame_stack)
        previous_target = target[0][action_index]

        if terminal:
              target[0][action_index] = reward
        else:
              best_action =
              np.argmax(self.model_online.predict(frame_stack))
              q = self.model.predict(frame_stack)[0][best_action]

              target[0][action_index] = reward + self.gamma * q
        error = np.abs(target[0][action_index] - previous_target)
        sumError+=error
        self.replay_memory.update(mini_batch[i][0], error)
        self.model_online.fit(previous_frame_stack, target,
        epochs=1, verbose=0)

    self.model_loss = 0
    print(sumError/self.batch_size)
```

Capítol 10 Resultats

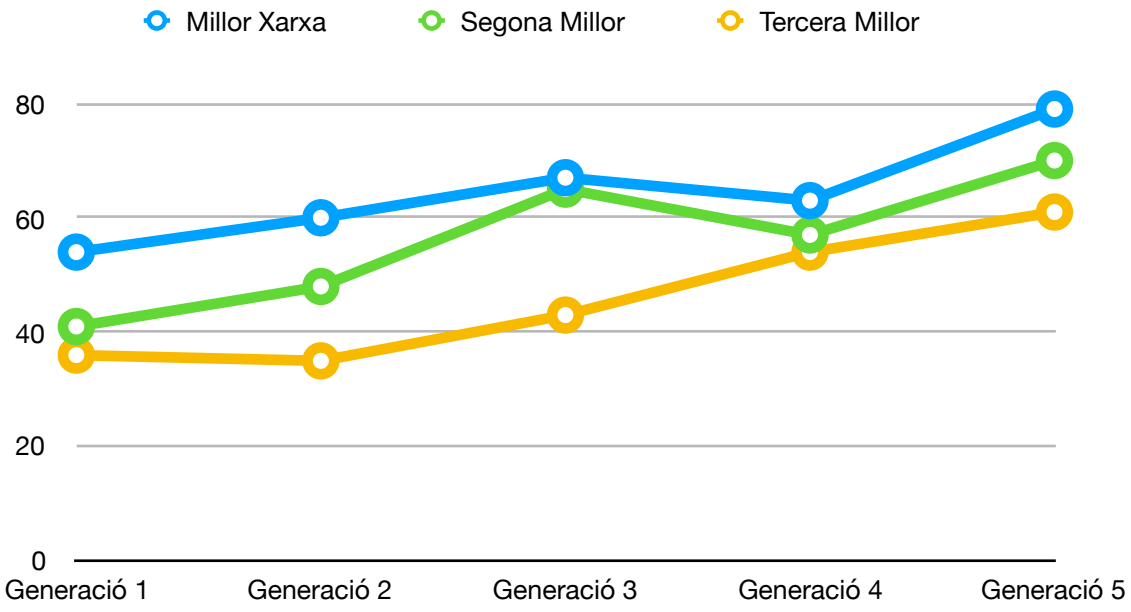
En aquest capítol es mostrarà el grau d'assoliment dels objectius.

10.1 Objectiu algoritme genètic

L'algoritme genètic l'hem utilitzat de la següent manera. Es fa una població de 7 cromosomes i s'entrena cada cromosoma fins X passes. S'agafen els 3 millors cromosomes amb millor temps i s'utilitzen per a fer la següent població, això fins a arribar a 5 poblacions i quedar-nos amb el millor.

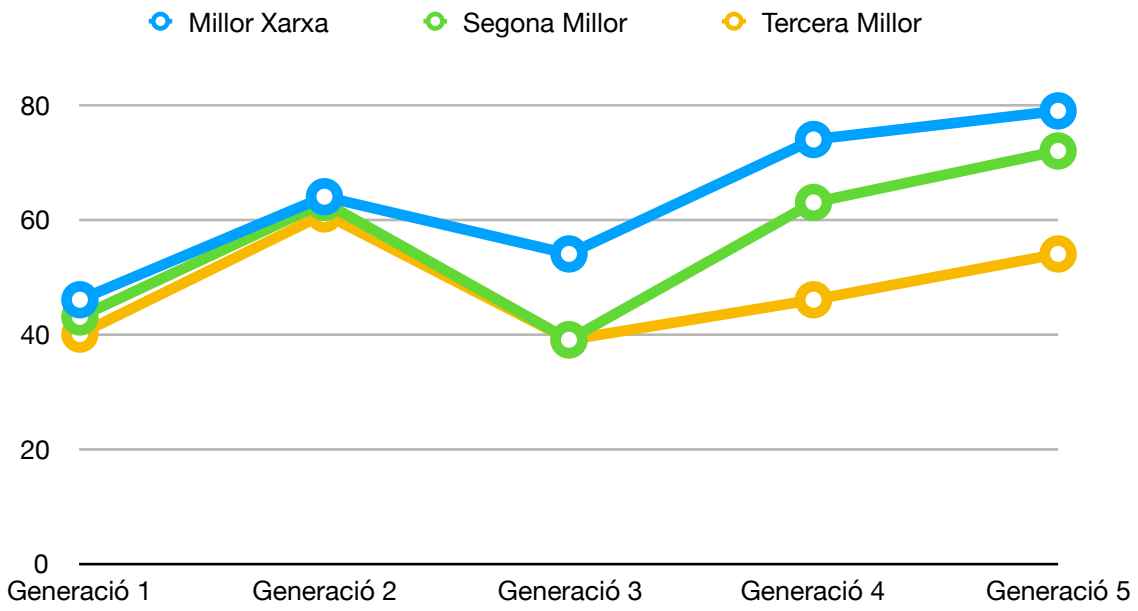
Per veure millor els resultats, a cada iteració guardava cromosomes els tres millors cromosomes i així es pot veure l'evolució en el temps.

CAS 1 Entrenament fins 1500 passes



Generació	Millor Xarxa	Segona Millor	Tercera Millor
1	54 segons	41 segons	36 segons
2	60 segons	48 segons	35 segons
3	67 segons	65 segons	43 segons
4	63 segons	57 segons	54 segons
5	79 segons	70 segons	61 segons

CAS 2 Entrenament fins a 1000 passes



Generació	Millor Xarxa	Segona Millor	Tercera Millor
1	46 segons	43 segons	40 segons
2	64 segons	63 segons	61 segons
3	54 segons	39 segons	39 segons
4	74 segons	63 segons	46 segons
5	79 segons	72 segons	54 segons

Podem veure com a mesura avancem en generacions, la duració en partida també augmenta. També és cert, que trobem alguna generació com la 3 del cas 1 que té uns valors molt més bons dels esperats. Això es deu al fet que hi ha molts paràmetres aleatoris i fent un entrenament de 1500 passes o 1000 passes per xarxa trobem molt soroll a les nostres dades (1500 passes són aproximadament 25 minuts). Un entrenament de 15000 passes per cada xarxa ens permetria veure com la gràfica incrementa sense tant soroll.

L’algoritme genètic sencer va començar a les 11:05 del matí i va acabar a la 01:54 de la matinada següent, van ser 14 hores i 49 minuts fent 1500 passes per xarxa, en el cas que féssim 15000 passes per xarxa hauríem de multiplicar el temps anterior per 10, 148 hores i 10 minuts. Aquest és el gran problema dels algoritmes genètics, les xarxes neuronals i el món de machine learning. El temps que tardem a dur a terme un objectiu com aquest és molt elevat i mentre s’està duent a terme l'ordinador queda totalment inutilitzable. A més a més, hem d’esperar que no hi hagi cap error mentre s'està executant.

10.2 Xarxa neuronal

L'objectiu de la xarxa neuronal era la de no rebre mal. Això ens permetia provar el màxim d'opcions dins un nivell. El millor resultat l'hem obtingut amb la xarxa més entrenada i ha durat 152 segons. Podem veure-ho a la Figura 10.1

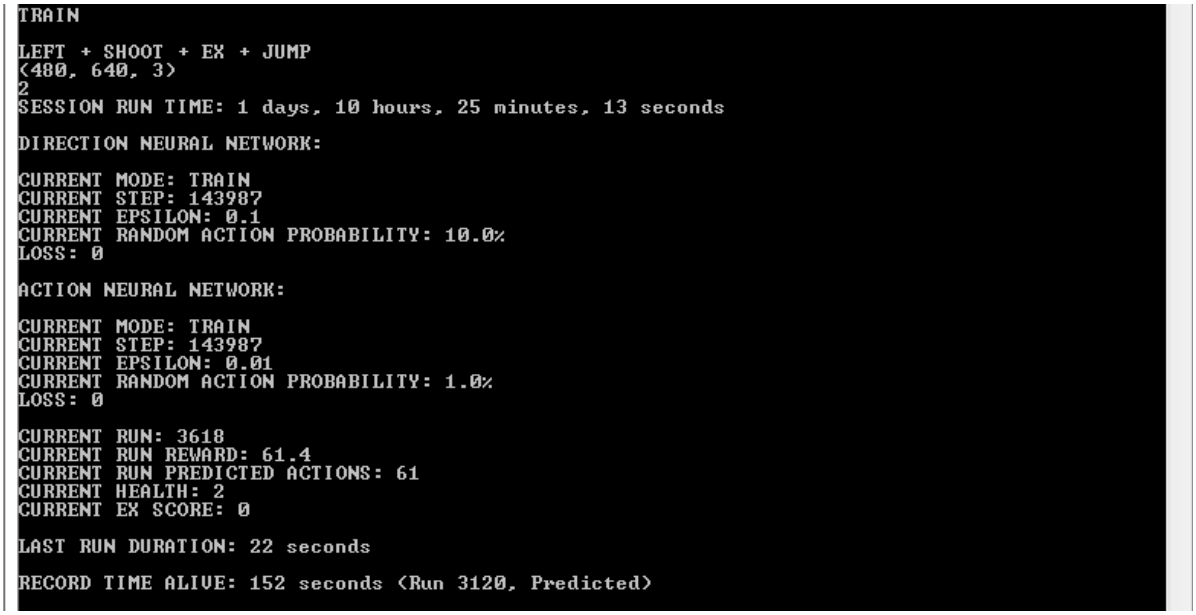


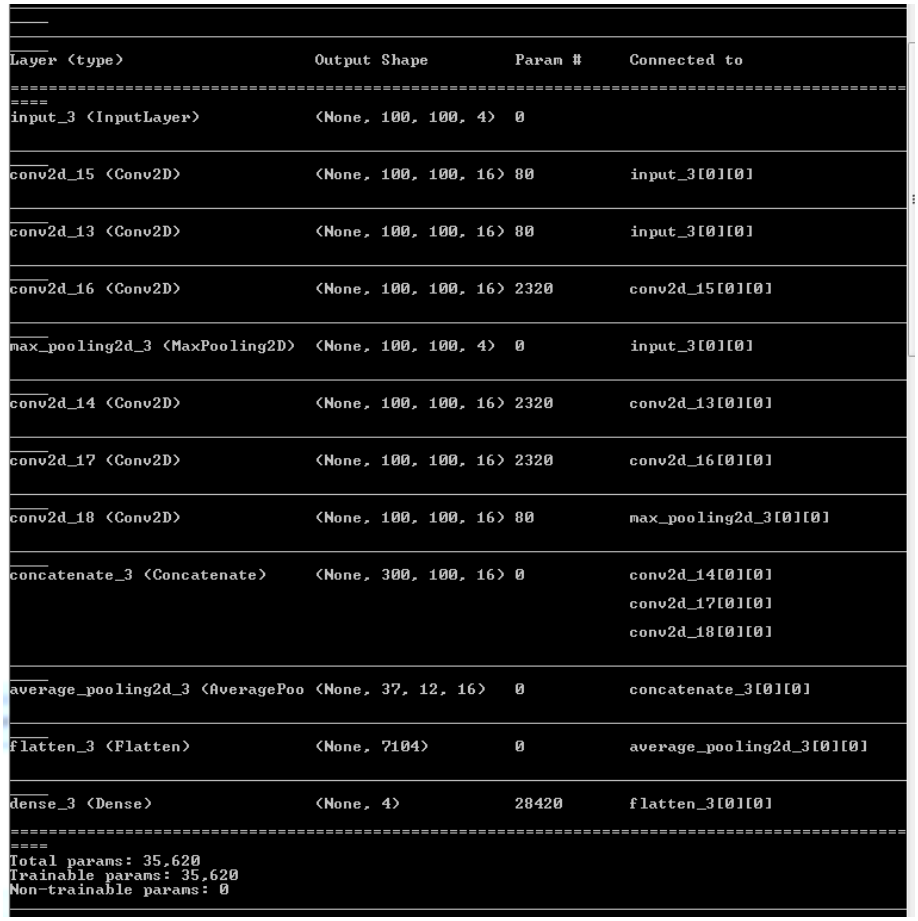
Figura 10.1 Millor resultat xarxa

La xarxa va estar entrenant fins a arribar a 1 dia i 11 hores. A diferència de l'algoritme genètic aquest tipus d’entrenament era molt més fàcil de guardar mentre estava en marxa i per tant si hi havia algun problema, podíem reprendre la marxa des d’un punt anterior.

Un altre problema interessant és saber si estem entrenant una xarxa de manera inútil o no, ja que pot ser que a causa de la seva estructura, no sigui capaç de resoldre el problema que li hem presentat.

10.2.1 Estructura de la xarxa

La xarxa neuronal que ha durat 152 segons en mode “PREDICTED” és la següent:



```
Layer (type)                 Output Shape              Param #   Connected to
==================================================================================================
input_3 (InputLayer)         (None, 100, 100, 4)       0
__________________________________________________________________________________________________
conv2d_15 (Conv2D)           (None, 100, 100, 16)      80        input_3[0][0]
__________________________________________________________________________________________________
conv2d_13 (Conv2D)           (None, 100, 100, 16)      80        input_3[0][0]
__________________________________________________________________________________________________
conv2d_16 (Conv2D)           (None, 100, 100, 16)      2320      conv2d_15[0][0]
__________________________________________________________________________________________________
max_pooling2d_3 (MaxPooling2D)  (None, 100, 100, 4)    0         input_3[0][0]
__________________________________________________________________________________________________
conv2d_14 (Conv2D)           (None, 100, 100, 16)      2320      conv2d_13[0][0]
__________________________________________________________________________________________________
conv2d_17 (Conv2D)           (None, 100, 100, 16)      2320      conv2d_16[0][0]
__________________________________________________________________________________________________
conv2d_18 (Conv2D)           (None, 100, 100, 16)      80        max_pooling2d_3[0][0]
__________________________________________________________________________________________________
concatenate_3 (Concatenate)  (None, 300, 100, 16)      0         conv2d_14[0][0]
                                                                 conv2d_17[0][0]
                                                                 conv2d_18[0][0]
__________________________________________________________________________________________________
average_pooling2d_3 (AveragePoo (None, 37, 12, 16)     0         concatenate_3[0][0]
__________________________________________________________________________________________________
flatten_3 (Flatten)          (None, 7104)              0         average_pooling2d_3[0][0]
__________________________________________________________________________________________________
dense_3 (Dense)              (None, 4)                 28420     flatten_3[0][0]
==================================================================================================
Total params: 35,620
Trainable params: 35,620
Non-trainable params: 0
```

Podem veure que té l’input_3 que és la capa d’entrada, després tenim capes convolucionals, conv2d_15 i conv2d_13 les quals van connectades a la capa d’input i una capa max_pooling2d que el que fa és rebaixar la qualitat de la imatge i es centra en els elements més importants. Tot seguit trobem tres capes més conv2d_14, conv2d_17 i conv2d_18 que van conectades a les altres 3 respectivament. Aquestes tres capes les ajuntarem a concatenate_3. Tornarem a fer una funció de reducció de qualitat per minimitzar paràmetres i enfocar-nos els elements més significatius, average_pooling2d. Els últims dos passos seràn el flatten_3 i dense_3, la seva funció és agafar les característiques identificades en les capes anteriors, posar-les en un vector i predir la sortida.

Es tracta de la xarxa utilitzada per dqn.action i ho podem veure perquè la sortida només te 4 valors, en el cas de dqn.direction, la sortida té 9 valors.

10.2.2 Altres xarxes

Hem trobat altres xarxes que tenen comportaments que són bastant interessants i la veritat és que donen basant bons resultats. Per exemple:

- Hi ha una xarxa en què el personatge es posa en un punt de la pantalla i l'única acció que fa és : DOWN+RIGHT+SHOOT+EX. Això fa que el personatge quedi quiet i ajupit en la posició d'inici. Aquesta posició és el lloc amb què menys freqüència passa el nostre enemic. D'aquesta manera la vida tarda molt en ser modificada i el personatge té temps de fer molt mal a l'enemic. Normalment no aconsegueix eliminar a l'enemic però sí que arriba a treure-li un 80-90% de la vida.

- Una altra xarxa enviarà al personatge a la cantonada dreta de la pantalla i premerà constantment la tecla per fer l'acció DASH. Aquesta acció el que fa és un moviment molt ràpid endavant i mentre el fa, el personatge desapareix. D'aquesta manera fins i tot quan l'enemic fa algun atac cap al nostre personatge aquest no l'afectarà, i només li farà mal quan es mou just sobre la seva posició.

Capítol 11 Conclusions

Per assolir els objectius d'aquest treball final de grau, es van marcar unes tasques a realitzar (vegeu Secció 1.3 Objectius). Aquestes tasques s'han pogut dur a terme, no per això sense trobar entrebancs.

S'ha estudiat a fons el funcionament del framework SerpentAI, veient com és el funcionament de les seves classes i com es poden unir totes per forma agents de joc. També s'ha pogut veure una part del funcionament d'un llenguatge amb el qual no havia treballat gaire, el Python.

S'han obert portes a tècniques d'intel·ligència artificial, com són els algoritmes genètics i les xarxes neuronals. La informació sobre aquestes tècniques és molt abundant, encara que és bastant confusa i s'ha de buscar bé per no agafar informació equivocada. Tant les xarxes neuronals com els algoritmes genètics tenen un ventall molt gran de possibilitats, i en aquest projecte només se n'ha vist una petita part.

L'elecció de les eines ha estat bastant senzilla, l'única que hi havia disponible per dur a terme aquest tipus de projectes és el SerpentAI. Per la creació de xarxes neuronals el Keras sempre ha estat l'eina escollida, els tutors em van plantejar aquesta opció des del principi i gràcies al fet que un d'ells és un expert en aquest món l'aprenentatge m'ha sigut més fàcil.

S'ha assolit la creació d'un algoritme genètic que és capaç de crear poblacions de xarxes neuronals, mutar-les, crear descendència de les millors xarxes de poblacions anteriors. Un altre cop gràcies al tutor experimentat en aquest món, el plantejament del codi i del funcionament de l'algoritme ha estat molt més fàcil d'entendre.

Respecte a l'objectiu més visible que és arribar el més lluny possible en el nivell i/o superar-lo. L'agent aconsegueix passar el nivell depenent de quina xarxa utilitzem encara que no el 100% dels intents. A causa del sistema de recompenses que hem donat a les xarxes neuronals l'agent buscarà sobreviure i no arribar al final del nivell. Hem aconseguit demostrar que gràcies als algoritmes genètics i a les xarxes neuronals podem aconseguir bons resultats.

L'entrenament de les xarxes neuronals és un tema molt important, la potència de l'ordinador i el temps disponible juguen un factor clau, amb una millor màquina s'haguessin pogut dur a terme més proves. Molts projectes d'aquest món no es poden dur a terme per culpa del gran cost computacional i si bé és cert que les grans empreses si que disposen d'aquesta potència de càlcul, la resta de nosaltres no som capaços d'aconseguir-ho.

El videojoc escollit ha portat més problemes dels esperats. Si haguéssim agafat un videojoc sense filtre a la pantalla i amb un sistema de puntuació o distància recorreguda visible dins la partida, la xarxa donaria millors resultats. El sistema de recompenses de la xarxa depèn totalment de la vida del personatge i d'aquesta manera no aprèn a passar el nivell sinó que aprèn a no morir.

Com a reflexió final s'han obtingut resultats bons, tot i els problemes i les incerteses de no saber si el que s'estava fent funcionaria. S'ha aconseguit crear un codi que gràcies als algoritmes genètics és capaç d'obtenir una xarxa neuronal òptima, la qual té la capacitat de jugar al joc i aprendre mentre ho està fent.

Capítol 12 Treball futur

El projecte pot avançar de moltes maneres entre elles:

- Canviar l'estil d'entrenament, podríem fer que no hi hagués cap acció aleatòria i totes fossin predites per la xarxa neuronal. D'aquesta manera s'entrenarà sempre seguint el mateix recorregut. Una de les opcions és arribar al final del nivell, però també pot arribar a un punt sense sortida i anar repetint el mateix recorregut infinites vegades perquè no durà a terme accions aleatòries.

- Seguint amb la línia de les xarxes neuronals, en lloc de fer xarxes utilitzant el mètode DDQN, utilitzar un altre mètode com per exemple podria ser el PPO el qual utilitza una altra idea d'entrenament i podríem obtenir resultats diferents.

- Es podria crear una aplicació a partir del que hi ha fet fins ara, que permetés canviar els paràmetres tant de l'algoritme genètic com de la xarxa neuronal i no haver-ho de fer tot per codi. Aquesta aplicació podria ser capaç d'entrenar una xarxa per qualsevol joc.

- Perfeccionar el sistema de recompenses en aquest mateix joc de manera que el personatge no aprengui a sobreviure sinó a passar el nivell.

- Desenvolupar més a fons les classes de captura, identificació i localització d'sprites per tal que siguin més eficients i permetin noves formes d'entrar inputs a la xarxa neuronal.

Bibliografia

Hado van Hasselt, Arthur Guez, David Silver (2015) Deep Reinforcement Learning with Double Q-learning. https://arxiv.org/abs/1509.06461

Greg Surma (2018) Atari - Solving Games with AI. https://towardsdatascience.com/atari-reinforcement-learning-in-depth-part-1-ddqn-ceaa762a546f

Thomas Simonini (2018) Recollit de les 7 parts https://www.freecodecamp.org/news/improvements-in-deep-q-learning-dueling-double-dqn-prioritized-experience-replay-and-fixed-58b130cc5682/

Vijini Mallawaarachchi (2017) Introduction to Genetic Algorithms https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3

Getting started with the Keras Sequential model https://keras.io/getting-started/sequential-model-guide/

Capítol 13
Manual d'usuari i d'instal·lació

Manual d'instal·lació

Requeriments inicials

- Windows 7 o superior

Python 3.6+ (amb Anaconda)

Serpent.AI està desenvolupat per Python 3.6 així que només funciona amb aquest i versions superiors.

Instal·lar Python 3.6+ no és realment difícil però instal·lar les llibreries que van amb ell a Windows és molt complicat . Per sort, Anaconda existeix i ens soluciona aquests problemes

Instal·lar Anaconda 5.2.0 (Python 3.6)

Descarregar la versió Python 3.6 de l'Anaconda 5.2.0 i utilitzar l'instal·lador gràfic

Les següents comandes s'hauran de dur a terme a la consola d'Anaconda amb privilegis d'administrador. Crearem un shortcut perquè cada cop que fem servir una comanda de Python i Serpent s'haurà de dur a terme des d'allà a partir d'ara.

Crear Conda Env pel Serpent.AI

conda create --name serpent python=3.6 ('serpent' can be replaced with another name)

Crear directori per guardar els projectes

mkdir SerpentAI && cd SerpentAI

Activar el Conda Env

conda activate serpent

Redis

Anar a https://github.com/microsoftarchive/redis/releases, descarregar i instal·lar la versió 3.2.100.

Build Tools for Visual Studio 2017

Anar a https://www.visualstudio.com/downloads/ i descarregar *Build Tools for Visual Studio 2017*. En aquesta seleccionar la secció *Visual C++ build tools* i marcar els següents paquets:

- Visual C++ Build Tools core features
- VC++ 2017 version 15.7 v14.14 latest v141 tools
- Visual C++ 2017 Redistributable Update
- VC++ 2015.3 v14.00 (v140) toolset for desktop
- Universal CRT

Installing Serpent.AI

Ara ja estem llestos per instal·lar el framework. Obrim una nova consola d'Anaconda i anem al directori on guardarem els projectes de SerpentAI i escrivim la comanda:

pip install SerpentAI

Després instal·larem les dependències restants fent:

serpent setup

Instal·lació de mòduls

Tesseract

S'han de seguir les següents passes per instal·lar el Tesseract:

1. Anar a https://digi.bib.uni-mannheim.de/tesseract/
2. Descarregar tesseract-ocr-setup-3.05.02-20180621.exe
3. Instal·lar utilitzant el fitxer descarregat.
4. Afegir el path de *tesseract.exe* a %PATH% environment variable / variable d'entorn

Es pot comprovar la instal·lació fent la comanda tesseract --list-langs.

Un cop instal·lat, utilitzar la comanda serpent setup ocr

GUI

Per instal·lar el debuggador visual, utilitzarem la comanda serpent setup gui.

Tensorflow GPU Dependencies (Opcional)

Només s'han d'instal·lar en el cas que tinguem una NVIDIA GPU amb càlcul computacional 3.0+ (GTX600 o superiors)

NVIDIA Drivers

El primer que hem de fer és tenir els drivers instal·lats, en el cas que no els tinguem haurem de fer el següent:

1. Anar http://www.nvidia.com/Download/index.aspx
2. Buscar la nostra GPU a la llista de drivers i descarregar el corresponent
3. Utilitzar l'instal·lador

CUDA

1. Fer un compte a https://developer.nvidia.com/
2. Anar a https://developer.nvidia.com/cuda-toolkit-archive
3. Seleccionar CUDA Toolkit 9.0
4. Selccionar Windows, x86_64, 10, exe (local) i descarregar l'instal·lador
5. Utilitzar l'instalador en mode personalitzable i només seleccionar CUDA.
6. Podem provar el cuda des de una consola Anaconda i amb la comanda nvcc --version.

cuDNN

1. Fer un compte o identificar-te a https://developer.nvidia.com/
2. Anar a https://developer.nvidia.com/rdp/cudnn-archive
3. Descarregar cuDNN v7.0.5 (Dec 5, 2017), for CUDA 9.0 i instal·lar.
4. Descarregar cuDNN v7.0.5 Library per Windows
5. Copiar bin, include and lib que hi ha dins de l'arxiu ZIP al directori on hem instal·lat el CUDA hauria de ser(C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v9.0)
6. Assegurar-nos que el .DLL està inclòs a %PATHEXT% environment variable / variables d'entorn.

Instal·lació

Utilitzar la comanda serpent setup ml. Tot seguit desinstal·lar la versió de Tensorflow 1.4.0 pip uninstall tensorflow-gpu i instal·lar la següent versió 1.5.1 pip install tensorflow-gpu==1.5.1.

Link amb un tutorial per a fer la instal·lació: https://www.youtube.com/watch?v=5d4Ceq1L8hg

Instal·lació dels plugins

1. Entrem i descarreguem el zip del següent link -https://drive.google.com/file/d/1ixZP8tcEguER6qMAvjOBwGSxKpBPqB3A/view?usp=sharing

2. Extraiem el contingut de la carpeta, entrem a codi i copiem els arxius SerpentCupheadGamePlugin i SerpentCupheadGameAgent a la carpeta on hem instal·lat el Serpent dins del directori plugins. Tot seguit entrem al directori de Seroent a partir de la consola d'Anaconda i escrivim les comandes “serpent activate SerpentCupheadGamePlugin” i “serpent activate SerpentCupheadGameAgent”

3. Copiem el codi de la carpeta DQN del zip descarregat i l’enganxem al següent path : C:\Users\NomUsuari\Anaconda3\envs\serpent\Lib\site-packages\serpent\machine_learning\reinforcment_learning

Manual d’usuari

Per utilitzar-lo simplement haurem d'obrir una consola Anaconda i escriurem el codi “serpent launch Cuphead”, això obrirà el Cuphead. Tot seguit haurem d’anar manualment fins al nivell que volem jugar. Entrem en ell i tan bon punt som dins premerem la tecla “Esc” per parar el joc.

Si volem que l’agent faci ús de l’algoritme genètic haurem de fer:

- Posar la variable self.genetic_algorithm_Activated a True
- Posar el paràmetre model_genetic=self.array_cromosome[0] en les dues declaracions de les xarxes neuronals, és a dir, a self.dqn_doirection i self.dqn_action.

Si volem que l’agent faci ús de l’entrenament de xarxes neuronals haurem de fer:

- Posar la variable self.genetic_algorithm_Activated a False
- Posar el paràmetre model_genetic=None en les dues declaracions de les xarxes neuronals, és a dir, a self.dqn_doirection i self.dqn_action.

Si volem fer ús d'una xarxa definida en un fitxer:

- Introduir nom del fitxer del model a direction_model_file_path, action_model_file_path.
- Introduir nom del fitxer dels pesos a direction_model_base_file_path, action_model_base_file_path. El nom del fitxer dels pesos serà de la següent manera: cuphead_{action o direction}_{passes}_{epsilon}_.h5. Si no s'escriu amb aquest format, el programa no el llegirà

Si no volem fer ús d'una xarxa definida en un fitxer:

- Introduir noms inexistents a les variables dels fitxers.

Escriurem la comanda "serpent play Cuphead SerpentCupheadGameAgent" per activar el nostre agent i començarà a fer la seva feina.