UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Estrategias de aprendizaje automático aplicadas a videojuegos

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Autor: Adrián Valero Gimeno
Tutor: Vicent Botti Navarro
Director experimental: Javier Palanca Cámara

Curso 2018-2019

Resum

En aquest treball de fi de grau es realitza un estudi basat en l'aplicació de diferents tècniques d'aprenentatge per reforç sobre videojocs clàssics amb la finalitat de comprovar si és possible aconseguir un rendiment semblant o superior al d'un humà. A aquest efecte en ment, serà necessari dissenyar una interficie que faça ús de llibreries especialitzades d'aprenentatge per reforç que ens permeten crear agents independents a executar sobre una selección de videojocs de la consola *Atari 2600*. Addicionalment, s'ha modelat una solució que implementa un algorisme específic a executar sobre el famós videojoc *Doom Classic* (1993).

Paraules clau: Intel·ligència artificial, aprenentatge per reforç, videojocs, agents, simulador

Resumen

En este trabajo de fin de grado se realiza un estudio basado en la aplicación de diferentes técnicas de aprendizaje por refuerzo sobre videojuegos clásicos con el fin de comprobar si es posible conseguir un rendimiento similar o superior al de un humano. Con este fin en mente, será necesario diseñar una interfaz que haga uso de librerías especializadas de aprendizaje por refuerzo que nos permitan crear agentes independientes a ejecutar sobre videojuegos de la consola *Atari 2600*. Adicionalmente, se ha modelado una solución que implementa un algoritmo específico a ejecutar sobre el famoso *Doom Classic* (1993).

Palabras clave: Inteligencia artificial, aprendizaje por refuerzo, videojuegos, agentes, simulador

Abstract

In this final degree thesis we develop a study based on the application of different reinforcement learning techniques over classic videogames in order to test whether its performance can be equal or greater than that of a human agent. With this goal in mind, it will be necessary to develop an interface that will allow us to create independent agents capable of making decisions over certain videogame environments from the console *Atari 2600* using specialised machine learning libraries. Additionally, we modeled a solution that implements a specific algorithm which we will then test over the famous *Doom Classic* (1993).

Key words: Artificial Intelligente, reinforcement learning, videogames, agents, simulator

Índice general

Índice general v
Índice de figuras ix
Índice de tablas x

1 Introducción 1
1.1 Motivación . . . 1
1.2 Objetivos . . . 2
1.3 Metodología . . . 3
1.4 Estructura de la memoria . . . 3
2 Estado del arte. La situación del aprendizaje por refuerzo en la actualidad 5
2.1 ¿Qué es el aprendizaje por refuerzo? . . . 5
2.1.1 Tipos de entorno . . . 6
2.1.2 Modelo de aprendizaje por refuerzo . . . 6
2.1.3 Los elementos del Aprendizaje por refuerzo . . . 7
2.1.4 Modelos de comportamiento óptimo . . . 7
2.1.5 Procesos de Markov . . . 8
2.1.6 Procesos de decisión de Markov (MDPs) . . . 10
2.1.7 Recompensas y función de retorno . . . 11
2.1.8 Políticas (policies) . . . 12
2.1.9 Funciones de valor (value functions) . . . 12
2.1.10 La ecuación de Bellman . . . 13
2.2 Introducción al Q-Learning . . . 14
2.2.1 Redes neuronales . . . 14
2.2.2 Redes neuronales convolucionales (CNNs) . . . 16
2.2.3 Redes recurrentes (RNNs) . . . 18
2.3 Algoritmos existentes . . . 19
2.3.1 DQN: Deep Q-Network . . . 19
2.3.2 A2C - Advantage Actor Critic . . . 21
2.3.3 A3C: Asyncronous advantage actor-critic . . . 22
2.3.4 IMPALA: Importance Weighted Actor-Learner Architecture . . . 23
2.3.5 PPO - Proximal Policy Optimization . . . 23
2.3.6 Estrategias evolutivas . . . 24
2.4 Aplicación en videojuegos . . . 25
3 Análisis del problema 27
3.1 Introducción . . . 27
3.1.1 Perspectiva del trabajo . . . 27
3.2 Descripción general . . . 27
3.2.1 Experimentación sobre Atari . . . 28
3.2.2 Entornos de Atari . . . 29
3.2.3 El entorno ViZDoom . . . 32
3.3 Plan de trabajo . . . 34
3.4 Hardware disponible . . . 36

4 Diseño de la solución **39**
4.1 Introducción . . . 39
4.2 Tecnologías utilizadas . . . 40
4.2.1 Python . . . 40
4.2.2 Módulos adicionales . . . 40
4.3 Arquitectura de solución Atari . . . 41
4.3.1 Inicialización de los experimentos y condiciones de parada . . . 43
4.3.2 Parámetros de entrada . . . 44
4.4 Diseño de la solución para Doom . . . 45
4.4.1 Algoritmo a implementar: DRQN - Deep Recurrent Q-Network . . . 46
4.4.2 Optimización . . . 46
4.4.3 Arquitectura de algoritmo DRQN . . . 47
4.4.4 Diseño de entrenamiento e integración con librerías . . . 47
5 Implementación **49**
5.1 Introducción . . . 49
5.2 Solución Atari . . . 49
5.2.1 Implementación de script de pruebas . . . 49
5.3 Implementación de solución para DOOM . . . 50
5.3.1 Algoritmo DRQN . . . 50
5.3.2 Integración de aprendizaje con ViZDoom y Gym . . . 54
6 Diseño de pruebas **57**
6.1 Introducción . . . 57
6.2 Experimentaciones a realizar en entornos ATARI . . . 57
6.2.1 Algoritmos y entornos a testear . . . 57
6.2.2 Condiciones de parada . . . 58
6.2.3 Diseño de hiperparámetros . . . 58
6.2.4 Ejecución . . . 59
6.2.5 Procesamiento gráfico de los datos . . . 59
6.3 Pruebas en DOOM . . . 61
6.3.1 Diseño de hiperparámetros . . . 61
6.3.2 Escenarios a testear sobre DRQN . . . 61
6.3.3 Condiciones de parada . . . 61
6.3.4 Ejecución . . . 62
6.3.5 Procesamiento de los datos . . . 62
7 Evaluación de resultados **63**
7.1 Caso ATARI . . . 63
7.1.1 Resultados de Pong . . . 63
7.1.2 Resultados de Space Invaders . . . 65
7.1.3 Resultados de Breakout . . . 66
7.1.4 Resultados de Boxing . . . 67
7.1.5 Resultados de Atlantis . . . 69
7.1.6 Resultados de Seaquest . . . 70
7.1.7 Conclusiones . . . 71
7.1.8 Problemas encontrados durante la ejecución . . . 71
7.2 Caso DOOM . . . 72
7.2.1 Conclusiones . . . 73
8 Conclusiones **75**
8.1 Discusión del trabajo realizado . . . 75
8.1.1 Justificación de las técnicas aplicadas . . . 75
8.1.2 ATARI - Resumen de las pruebas y resultados . . . 75
8.1.3 DOOM - Conclusiones . . . 75

8.2 Relación con los estudios cursados 76
8.3 Trabajo futuro 77
8.3.1 Agradecimientos 77
Bibliografía **79**

Índice de figuras

2.1 Ejemplo de proceso de Markov . . . 9
2.2 Proceso de Markov con recompensa . . . 10
2.3 En esta extensión de nuestro ejemplo, añadimos acciones (decisiones) que el agente debe seleccionar, representadas como etiquetas en las aristas del autómata. . . . 11
2.4 Ejemplo de una red neuronal simple con una capa oculta de 5 nodos (neuronas). . . . 15
2.5 Funcionamiento interno de una neurona . . . 16
2.6 Ejemplos de funciones de activación. De izquierda a derecha, función step, linear, sigmoid y ReLU. . . . 16
2.7 Ejemplo de red convolucional . . . 17
2.8 Ilustración de la operación de convolución[3]. A la izquierda, el filtro utilizado para el ejemplo. Como se observa a la derecha, se reduce la dimensionalidad de la matriz inicial e introduce el resultado de la convolución como la suma de los vectores de pesos de cada píxel de la matriz: 1 x 1 + 1 x 4 + 1 x 2 + 1 x 5 = 12. . . . 17
2.9 A la izquierda, un modelo de red neuronal recurrente básica. A la derecha observamos el mismo modelo de red neuronal "desarrollado", donde observamos que los *outputs* de cada iteración son realimentados a la red neuronal, dotándola así de una suerte de memoria a corto plazo. . . . 18
2.10 Esquema de trabajo de actor worker en algoritmos actor-crítico. . . . 21
2.11 Diagrama de arquitectura de algoritmo A3C. Cada agente, sea *worker* o crítico cuenta con su propia red neuronal sobre la que ejecuta operaciones. Las salidas de las redes *worker* se utilizan como valores de entrada para el crítico. . . . 22
2.12 Comparativa de comunicación entre A3C e IMPALA . . . 23

3.1 Entorno Pong. . . . 30
3.2 Ejemplo de estado del entorno Space Invaders. . . . 30
3.3 Entorno Breakout. . . . 31
3.4 Entorno Boxing. . . . 31
3.5 Captura de pantalla del juego Atlantis. . . . 32
3.6 Captura de pantalla del entorno Seaquest. . . . 32
3.7 Mapa en escenario Deadly Corridor. . . . 33

4.1 Modelo de arquitectura de la solución . . . 42
4.2 Modelo de diagrama de flujo utilizado para la solución de Atari. . . . 43
4.3 Modelo de arquitectura de la solución . . . 44
4.4 Diagrama de dependencias de la solución Doom. . . . 45
4.5 Arquitectura de la red convolucional. . . . 47

6.1 Interfaz gráfica de la herramienta Tensorboard mientras se realizan experimentos. . . . 60
6.2 Ejemplo de muestra por terminal de resultados. . . . 62

7.1 Resultados de PongNoFrameskip-v4. 64
7.2 Puntuaciones medias relativas a la experimentación [4] sobre el servidor personal. Se puede observar que todos los algoritmos a excepción de DQN experimentan mejoras en la puntuación a lo largo de los escasos 2M de iteraciones realizadas, siendo IMPALA y A2C de nuevo los más eficientes. 64
7.3 Resultados de las experimentaciones de 3 horas sobre SpaceInvaders-v0 en función del tiempo. 65
7.4 Resultados de las experimentaciones de 16 horas sobre el entorno SpaceInvaders-v4. 66
7.5 Gráfica de número de episodios/iteración. Como se puede observar, IMPALA es el algoritmo más eficiente respecto a este factor, lo cual le proporciona potencialmente una gran ventaja sobre sus competidores. 66
7.6 Resultados relativos al tiempo de las experimentaciones en BreakoutNoFrameskip-v4. 67
7.7 Resultados relativos al tiempo de las experimentaciones en el entorno Boxing-v0 y BoxingNoFrameskip-v4. 68
7.8 Gráfico de recompensas medias obtenidas durante experimentos sobre AtlantisNoFrameskip-v4. 69
7.9 Gráfico de recompensas máximas obtenidas durante los experimentos sobre AtlantisNoFrameskip-v4. 70
7.10 Resultados en función de la puntuación media obtenida en Seaquest-v4. . 70
7.11 Resultados de la experimentación sobre DoomBasic-v0. 73
7.12 Resultados relativos al entorno *DoomDefendCenter-v0*. 73
7.13 Gráficas de resultados. A la izquierda, resultados relativos a *DoomBasic-v0*. A la derecha, resultados de *DoomDefendCenter-v0*. 73

Índice de tablas

6.1 Tabla de configuraciones de los experimentos en cada servidor. 59
6.2 Tabla de las experimentaciones lanzadas. 59
6.3 Tabla de configuraciones de los experimentos sobre DOOM. 62

CAPÍTULO 1
Introducción

En los últimos años, se ha podido observar un crecimiento notable en el uso de sistemas de inteligencia artificial en empresas de internet hasta el punto en el que se han convertido en un requisito casi indispensable para ser capaz de ofrecer un servicio específico; por ejemplo, los sistemas de recomendación en servicios de plataformas de Streaming o la publicidad personalizada que muchas páginas de compras por internet ofrecen basándose en la experiencia anterior de los usuarios en sus plataformas. Estos sistemas se basan en el entrenamiento de redes neuronales que permiten crear una relación entre un histórico de selecciones anteriores y productos afines.

El aprendizaje automático o "Machine Learning" ha sido durante los últimos años el foco de una extensa investigación, debido a su gran potencial en la aplicación a problemas del mundo moderno. Proyectos como los del equipo de Google DeepMind han conseguido hacer grandes avances en juegos clásicos de gran dificultad, gracias a la investigación en técnicas como el Deep Learning, siendo este avance liderado por el equipo de la empresa DeepMind recientemente adquirida por Google. Esta empresa recientemente desveló AlphaGo, una inteligencia artificial que fue capaz de ganar a algunas de las mentes más experimentadas del tradicional juego chino Go. Este juego, considerado uno de los más difíciles del mundo, se calcula que tiene sobre 10^{172} configuraciones posibles de las piezas sobre el tablero - un número superior al número estimado de átomos existentes en el universo -, haciéndolo extraordinariamente más complejo que juegos como el ajedrez. Recientemente, DeepMind también desveló AlphaStar, una nueva inteligencia artificial que por el momento es capaz de competir al mismo nivel que algunos de los mejores jugadores de StarCraft del mundo. En nuestro trabajo, haremos un repaso de las técnicas más conocidas de aprendizaje por refuerzo, haciendo un seguimiento histórico de los avances realizados e introduciendo los conceptos previos necesarios para la consecución de nuestros objetivos. Una vez asentadas nuestras bases, nuestro objetivo será la implementación de algunas técnicas ya conocidas sobre algunos juegos clásicos de la Atari 2600, así como de un entorno personalizado del Doom original, inicialmente publicado en el año 1993.

1.1 Motivación

Se ha elegido un trabajo de estas características debido a una motivación personal hacia los campos de la inteligencia artificial, así como una manera de extender el conocimiento en ciertos ámbitos del mundo de la computación los cuales no se encuentran necesariamente presentes en un plan de estudios tradicional en el campo de la Ingeniería Informática. De esta forma se busca además un desarrollo personal y profesional en el campo de la computación, que permita el acceso a un futuro laboral en este campo.

Se partía además de una necesidad personal de realizar un trabajo propio y, de alguna manera, novedoso con respecto a lo que podría conllevar la realización de un proyecto de fin de carrera típico. En definitiva, realizar un trabajo con unos objetivos y una metodología a seguir bien definidos y, por encima de todo, que contribuyera al desarrollo de las competencias necesarias para la formación en el campo de la investigación y el desarrollo de sistemas de estas características.

El campo de la inteligencia artificial está creciendo exponencialmente, y no dejará de hacerlo en los próximos años. De esta manera, se cree que el aprendizaje por refuerzo puede suponer una nueva revolución en la disciplina del aprendizaje automático, cosa que ya se puede observar en los avances realizados por empresas como DeepMind. De la misma manera, aplicar estos nuevos avances a los videojuegos supone un desafío muy alentador, que además se podría extrapolar más adelante a otros ámbitos como la robótica.

1.2 Objetivos

El propósito de este TFG consistirá en el estudio de las diferentes técnicas existentes de aprendizaje automático aplicadas a sencillos videojuegos en 2D, los cuales tendrán un número limitado de acciones a realizar.

Otro de los objetivos consistirá en el estudio y realización de pruebas sobre distintos entornos predefinidos de juegos arcade, haciendo uso de librerías que permiten el acceso a dicho tipo de juegos como la herramienta OpenAI desarrollada por Google, o las distintas librerías de Python relacionadas con la creación de entornos de redes neuronales y, más concretamente, la manipulación de hiperparámetros para encontrar los mejores resultados para cada uno de los entornos estudiados.

Resultaría además interesante aplicar estas nuevas técnicas estudiadas en la resolución de algún videojuego de mayor complejidad. Se plantea, por lo tanto, un estudio de los algoritmos más destacables estudiados en algún juego complejo como puede ser el famoso juego DOOM classic estrenado en el año 1993.

Se pretende aplicar estos u otros algoritmos de similares características a el entorno ViZDoom, siendo este una herramienta que nos permitirá acceder a algunos mapas predefinidos de Doom y sobre los cuales puede resultar muy interesante la aplicación de las técnicas mencionadas. Para ello, será necesario un trabajo de implementación propio o bien para introducir los algoritmos en el entorno, o para ser capaz de utilizar nuestras librerías de aprendizaje automático sobre este entorno independiente. Por ello, nuestros objetivos en este ámbito se podrían resumir en los siguientes:

- Implementación de un algoritmo de aprendizaje apto para entornos de observación parcial. Diseño de las estructuras necesarias de red neuronal y memoria para habilitar un proceso de aprendizaje.

- Diseño de una interfaz que permita la comunicación entre el videojuego y el algoritmo a estudiar.

- Experimentaciones sobre la solución creada para verificar el funcionamiento de los algoritmos y contrastar los distintos casos de estudio.

1.3 Metodología

Para la realización de este trabajo se pretende llevar a cabo una investigación en el campo de la inteligencia artificial, concretamente en el desarrollo de algoritmos de aprendizaje aplicados a entornos estocásticos. Para ello, se investigarán técnicas como el aprendizaje por refuerzo o Q-Learning, algoritmos evolutivos u otros algoritmos más complejos de aprendizaje (introducidos más adelante en el capítulo 2). Adicionalmente, se ha seguido un curso externo enfocado a la teoría de inteligencia artificial aplicada a distintos videojuegos, en los cuales se introducen los conceptos principales en los cuales se basa todo el campo del aprendizaje por refuerzo, desde la ecuación de Bellman hasta el diseño de redes neuronales convolucionales como método para acceder a la información de nuestros entornos.

Para complementar nuestra base de conocimientos, nos basaremos en artículos de investigación publicados por entidades como Google DeepMind, o el Massachussets Institute of Technology, en busca de ideas y nuevas técnicas para su posterior aplicación en nuestro trabajo.

Con respecto a las tareas de implementación que nos ocupan, comenzaremos por explorar las librerías ya existentes de aprendizaje por refuerzo, comenzando por el funcionamiento de las estructuras internas más básicas de entornos sobre los que se pueden aplicar, en este caso videojuegos de Atari (OpenAI Gym). Una vez con un conocimiento más extenso del funcionamiento de estas librerías, podremos comenzar a explorar herramientas que hacen uso de técnicas de inteligencia artificial por encima de estos entornos (Ray/RLLib) y desarrollar experimentaciones de distintos tipos de técnicas para tratar de obtener una mejor comprensión de qué técnicas funcionan mejor en función de las características de los entornos.

Por último, nuestro objetivo final será la implementación propia de algunas técnicas sobre un emulador del videojuego Doom. Para ello, haremos uso de la famosa librería Tensorflow, utilizada para la investigación en el campo de la inteligencia artificial. Una vez implementado, contrastaremos los resultados obtenidos con los de nuestras experimentaciones anteriores.

1.4 Estructura de la memoria

En el capítulo 2 - *Estado del arte. La situación del aprendizaje automático en la actualidad*, se hace una pequeña introducción a la materia del aprendizaje por refuerzo, introduciendo conceptos fundamentales para la comprensión del resto del documento. Se describirá la estructura de los diferentes algoritmos de aprendizaje existentes en la actualidad que tomaremos como base.

A continuación, en el capitulo 3 - *Análisis del problema*, describiremos los problemas a los que nos enfrentamos y haremos una primera aproximación de las soluciones a desarrollar durante nuestro trabajo, haciendo una distinción entre las pruebas en entornos existentes y todo el proceso que habremos de seguir para hacer acomodar el entorno externo de Doom a las librerías que utilizaremos.

En el capítulo 4 - *Diseño de solución*, se explica la estructura de la solución desarrollada para el estudio de los diferentes algoritmos, tomando como base librerías de Python ya existentes. Se detallará el papel de cada uno de los procesos que intervienen en la implementación real de algoritmos de aprendizaje por refuerzo, y se describirá con detalle la solución desarrollada.

En el capítulo 5 - *Implementación*, se describirá el desarrollo de la solución propuesta introducida en el capitulo 4, además de exponer nuestras aportaciones en forma de código. Se hará una breve explicación de cómo se conectan las diferentes librerías sobrae las que nos basamos para diseñar nuestra solución.

Utilizaremos el capítulo 6 - *Diseño de pruebas* para introducir la metodología que seguiremos en la realización de nuestros experimentos. Hablaremos de los distintos entornos en los que se centrará nuestra experimentación, así como la selección de hiperparámetros de la que haremos uso a la hora de llevar a cabo las tareas de aprendizaje.

En el capítulo 7 - *Evaluación de resultados* analizaremos los resultados obtenidos sobre los entornos ATARI y DOOM.

Por último, en el capítulo 8 expondremos nuestras conclusiones así como posibles mejoras o ampliaciones que se pudieran aplicar al trabajo realizado.

CAPÍTULO 2

Estado del arte. La situación del aprendizaje por refuerzo en la actualidad

2.1 ¿Qué es el aprendizaje por refuerzo?

El aprendizaje por refuerzo o *Reinforcement Learning* [12] es un área del aprendizaje automático cuya finalidad es resolver el problema de cómo una entidad o *agente* debe escoger acciones dentro de un *entorno* dado para maximizar una noción de *recompensa* o premio acumulado. Es considerado uno de los tres paradigmas principales de aprendizaje automático, junto al **aprendizaje supervisado** y el **aprendizaje no supervisado**.

El concepto de aprendizaje por refuerzo se remonta a los inicios de la cibernética, campo que se basa en principios de ámbitos tan variados como la estadística, psicología, neurociencia y, por supuesto, ciencia de computadores. És un método de programación en el que unos agentes independientes son castigados o recompensados en función de sus acciones sin tener que especificar *cómo* se ha de realizar una tarea. No obstante, existen enormes obstáculos computacionales para la implementación de esta premisa.

En esencia, el aprendizaje por refuerzo es **la tarea de un agente de aprender a través de métodos de prueba y error comportamientos específicos que maximicen su recompensa en un entorno dinámico**. A continuación, trataremos de explicar este concepto en mayor profundidad.

Es posible que, por razones de brevedad, durante este documento se mencione el concepto de aprendizaje por refuerzo por sus siglas en inglés, RL (Reinforcement Learning).

En primer lugar, necesitaremos definir las entidades básicas presentes en un modelo estándar de aprendizaje por refuerzo:

Agente , una entidad autónoma que percibe información de un entorno determinado y actuará acorde a la misma. En el contexto de nuestro trabajo, se trata de la entidad que sustituye a un jugador real.

Entorno es el término por el que se describe un **proceso de decisión de Markov** en el contexto del aprendizaje por refuerzo. Este concepto se explicará con detalle en el apartado 2.1.6.

2.1.1. Tipos de entorno

Dedicaremos esta sección a proporcionar una definición de los tipos de entorno existentes. Las distinciones entre tipos de entorno vendrán dadas por las propiedades que presenten:

- **Entornos deterministas/estocásticos:** Un entorno es determinista cuando un estado siguiente cualquiera viene completamente determinado por el estado actual y las acciones del agente. En otro caso, el estado se considera estocástico.
- **Parcial/Totalmente observables:** Se llamará totalmente observable a un entorno en el que el agente tenga acceso a toda la información del estado en el que se encuentra en todo momento de la ejecución. En otro caso, el entorno será parcialmente observable.
- **Discretos/continuos:** Si el número de estados es limitado y claramente definible, el entorno será **discreto** (por ejemplo, el ajedrez). En otro caso, el entorno será **continuo**
- **Episódicos/no episódicos:** En un entorno episódico, cada episodio consiste de la percepción y posterior actuación del agente en un estado individual, y la calidad de su acción dependerá de las acciones tomadas en el pasado. Por otra parte, los entornos no episódicos requieren memoria de las acciones tomadas en el pasado para determinar la mejor acción siguiente.
- **Mono/multiagente:** Es posible que un entorno sea capaz de soportar diversos agentes en un mismo espacio de tiempo, en estos casos el entorno se llama **multiagente**. En otro caso, se tratará de un entorno monoagente.

2.1.2. Modelo de aprendizaje por refuerzo

En un modelo estándar de RL, un agente se conecta a un entorno mediante percepción y acción. En cada paso de la interacción el agente recibirá unos valores de entrada i, consistentes en unas indicaciones del estado actual s, e información de la recompensa r recibida por la última acción realizada. Por su parte el agente en función de estos parámetros de entrada, así como de sus experiencias anteriores. El comportamiento del agente deberá por su parte tratar de **maximizar** las recompensas obtenidas a largo plazo; esto se puede conseguir a través de un proceso de prueba y error sistemático, guiado por un algoritmo determinado de selección de acciones que, dependiendo de sus características, realizará la tarea de búsqueda basándose en unas directrices predeterminadas. Estos algoritmos se explicarán en profundidad en secciones posteriores de este trabajo.

Formalmente, el modelo se puede definir como:

- Una serie de estados discretos del entorno, S
- Una serie de acciones discretas disponibles a realizar por el agente A
- Las señales de recompensa asociadas a cada par estado-acción

El trabajo del agente consistirá en encontrar una **política** π (sección 2.1.8) que maximice la recompensa a largo plazo. Después de seleccionar una acción, el entorno transmitirá al agente la recompensa inmediata así como el estado en el que ha desembocado su acción, pero no se le dice qué acción habría sido la óptima para maximizar sus intereses

a largo plazo. Es por ello que será necesario implementar mecanismos de memoria que permitan al agente recordar experiencias anteriores y ser así capaz de realizar acciones informadas sobre el entorno.

Algunos aspectos del aprendizaje por refuerzo tienen una fuerte relación con problemas de búsqueda heurística y planificación. Los algoritmos utilizados intentan generar una trayectoria con un resultado satisfactorio a través de un grafo de estados o **proceso de decisión de Markov**, introducido más adelante en la sección 2.1.6.

2.1.3. Los elementos del Aprendizaje por refuerzo

Como se ha mencionado anteriormente, el aprendizaje por refuerzo se basa en ideas de la neurociencia y la psicología, específicamente del campo de la psicología del comportamiento.

Para plantear el problema sobre un entorno sencillo, se suele modelar el problema del aprendizaje por refuerzo como un conjunto de elementos:

El conjunto de estados. Se trata de una colección de todos los estados posibles en los que un agente se puede encontrar, teniendo en cuenta toda la información del entorno.

El conjunto de acciones. Como su nombre indica, recoge todas las acciones posibles que un agente puede realizar en un entorno determinado.

Reglas de transición. Normalmente se componen de una serie de valores escalares que determinan la **recompensa** resultante de ejecutar una acción sobre un estado determinado.

Función de recompensa. Definirá el objetivo del problema. Consiste en una colección de relaciones estado-acción a las que se les asignará un valor numérico individual, dependiendo de si la acción tomada en un estado dado es favorable o no para la consecución del objetivo. El objetivo del agente será siempre intentar maximizar la recompensa total obtenida en un episodio[1].

En resumidas cuentas se podría decir que el aprendizaje por refuerzo consiste en introducir a un agente independiente una serie de directivas que le permitan atribuir un valor sobre una o un conjunto de acciones que ejecuta en un entorno. De esta manera, el agente podrá ser capaz de determinar qué serie de acciones le ofrezcen un resultado más favorable. El agente aprenderá a medida que ejecute acciones (las cuales en un primer momento serán escogidas aleatoriamente) y sopese las recompensas obtenidas que el entorno le devolverá en función de las acciones que vaya tomando.

El aprendizaje que el agente experimenta podrá variar dependiendo de las directrices (hiperparámetros) que se le introduzcan o de la aleatoriedad que el propio entorno puede presentar, en caso de que este fuera estocástico. Asimismo, existen una gran variedad de algoritmos de aprendizaje por refuerzo que se pueden aplicar. En la sección 2.3 mencionaremos algunos junto con su funcionamiento.

2.1.4. Modelos de comportamiento óptimo

Ante el problema de el aprendizaje por refuerzo, tarde o temprano se formulará la pregunta *¿Cómo se puede determinar que un agente tiene un comportamiento óptimo?* Es aquí,

[1] Al hablar de episodio, nos referimos a una determinada secuencia de estados dentro de un entorno hasta dar con un estado final.

por tanto, cuanto se tendrá que decidir qué modelo de optimalidad seguir. En concreto, estos modelos tratarán de especificar cómo se tomará en cuenta los estados futuros posibles.

El modelo de **horizonte finito** o *finite horizon* tomará como premisa que nuestro entorno tiene un número finito de estados. El agente intentará optimizar su recompensa esperada para los siguientes n pasos:

$$E(\sum_{t=0}^{n} r_t)$$

Como se puede observar, la aplicación consiste en realizar un sumatorio de las recompensas esperadas durante todo el recorrido, sin tener en cuenta mecanismos de control como pudiera ser un factor de descuento. En cada paso, r_t representa la recompensa que el entorno devuelve en un instante de tiempo futuro t. Este modelo no es demasiado apropiado, entre otros porque en la mayoría de ocasiones no se sabe con certeza la duración que tendrá la vida del agente.

El modelo de **horizonte finito con factor de descuento** tiene en cuenta las recompensas a largo plazo pero filtra las recompensas futuras a partir de un factor de descuento γ. Este modelo es más matemáticamente accesible ya que elimina la gran tarea computacional que supondría tener que sumar todas las recompensas, mientras que permite como se ha mencionado en secciones anteriores, centrarse en las acciones a tomar más próximas aún teniendo en cuenta las recompensas a largo plazo.

$$E(\sum_{t=0}^{\infty} \gamma^t r_t)$$

Un problema de este criterio es que no hay manera de distinguir entre dos políticas en las que por ejemplo una recibiera una gran cantidad de recompensa al inicio y después dejara de recibir y otra que mantiene una recompensa moderada durante gran parte de su recorrido. De todos modos, es posible modificar el criterio para que tenga en cuenta la recompensa media a largo plazo así como las recompensas iniciales. Llamaremos a este criterio que maximizará la recompensa media a largo plazo **modelo sesgado óptimo**.

2.1.5. Procesos de Markov

Llegados a este punto, es necesario abordar un concepto vital para conocer la implementación a bajo nivel de modelos de aprendizaje por refuerzo. Es por ello que en la siguiente sección intentaremos explicar de manera comprensible los **procesos de decisión de Markov**. A grandes rasgos, estos se podrían interpretar como un modelo de autómata finito de estados en el que la probabilidad de que ocurra un evento depende solamente del evento inmediatamente anterior.

Los procesos de Markov se definen a partir de los principios de la programación dinámica, cuya finalidad consiste en solucionar problemas complejos a partir de dividirlos en subproblemas, resolverlos, y combinar las soluciones para resolver el problema general. Se trata de una solución para problemas con dos propiedades principales: La primera, la existencia de una **subestructura óptima** donde se aplica el **principio de optimalidad**. En otras palabras, el concepto de que existiría un recorrido entre los estados que devolviera una recompensa máxima (óptima) al jugador.

La segunda propiedad sería la existencia de problemas solapados. En un entorno dinámico, podemos encontrar el mismo subproblema en distintas ocasiones, por lo que es

necesaria una estructura para identificarlos y poder reutilizar una solución que ya puede haber sido probada y validada anteriormente.

Los procesos de decisión de Markov satisfacen ambas propiedades. Por una parte, la **ecuación de Bellman** nos proporciona una descomposición recursiva del problema desde la cual es posible evaluar la acción óptima a tomar desde un momento t así como cual será el comportamiento óptimo desde los estados futuros $t+n$.

Adicionalmente, en estos sistemas se hace uso de una **función de valor**, diseñada para almacenar información sobre qué movimientos tienen una mayor probabilidad de aproximarse a una ejecución óptima, asistiendo así en el las decisiones a tomar por un agente.

Un proceso de Markov, vendrá modelado con los siguientes elementos:

- S, que representa una colección finita de estados.
- T es un modelo de transición, consistente en una tupla (estadoInicial, estadoFinal).
- P(s,s') es la **función de transición**. El valor de esta función nos denotará la probabilidad de acabar en cierto estado s' partiendo de un estado s. Hará, por lo tanto, uso de la tupla del modelo de transición.

En este tipo de problemas se asume la **propiedad de Markov**, que demuestra que los efectos de una transición entre estados dependen solamente del estado en el que el agente se encuentra, y no de los estados anteriormente visitados. Esto se ajusta a la definición dada anteriormente de los **entornos estocásticos**.

A continuación, expondremos un ejemplo sencillo de proceso de markov en el que cada nodo representa un estado con una probabilidad de transición entre 0 y 1 entre cada estado y el siguiente, y en el que el estado *Parada* representa un estado final.

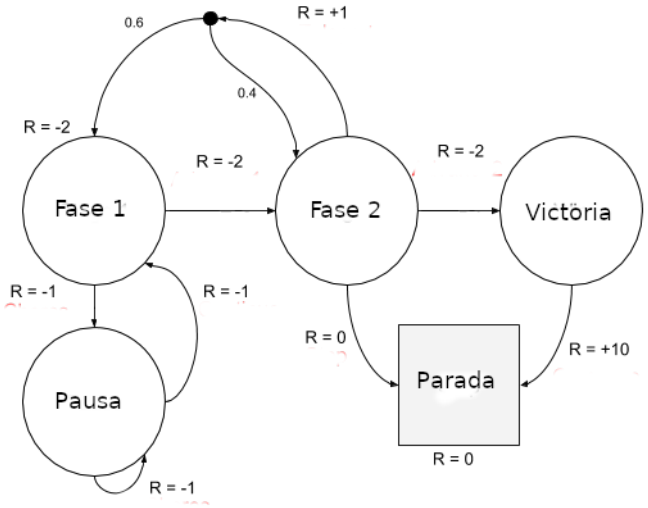


Figura 2.1: Ejemplo de proceso de Markov

En esta cadena no contamos con un valor de recompensa asociado a realizar una transición en un estado determinado para completar un objetivo.

Procesos de Markov con recompensa (MRPs)

Estos procesos añaden una **función de recompensa** a la tupla de la que se compone un proceso de markov estándar. Será, por tanto, una tupla (**S, P, R,** γ) donde **S** es el espacio de estados, **P** la probabilidad de transición y **R** la función de recompensa, que se aplicará con un factor de descuento γ:

$$Rs = \mathbb{E}[R_t + 1 | S_t = S]$$

Esta función nos dirá la recompensa inmediata se espera conseguir del estado S en un momento determinado.

Adicionalmente, introduciremos la noción de la función de retorno G_t basándonos en el **modelo de horizonte finito con función de descuento** que introducimos en la sección anterior. Es decir, aplicaremos una función de descuento a cada paso futuro del recorrido, con la finalidad de maximizar el retorno.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

Como ya se ha mencionado, el **factor de descuento** tomará un valor del intervalo [0,1]. En el caso de estar más cercano a 0, las acciones a corto plazo tendrán un peso mucho mayor que las futuras, mientras que en el caso de que el valor se aproxime a 1, se considerarán las acciones futuras con una mayor importancia.

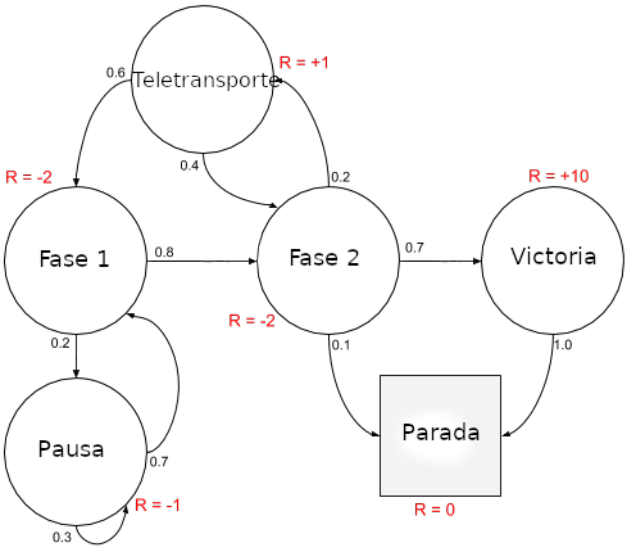


Figura 2.2: Proceso de Markov con recompensa

2.1.6. Procesos de decisión de Markov (MDPs)

Los procesos de decisión de markov son una extensión de los procesos con recompensa, con la adición de introducir **decisiones** que el agente debe tomar.

Así, un MDP es una tupla <S,A,P,R,γ>, cuyos elementos representan:

- Un conjunto de estados finito S.
- Un conjunto finito de acciones A.

- Una matriz de probabilidad de transición P(s,a,s').
- Un conjunto de funciones de recompensa R(s,a,s') asignados en función del estado y la acción.
- El factor de descuento $\gamma \in [0,1]$.

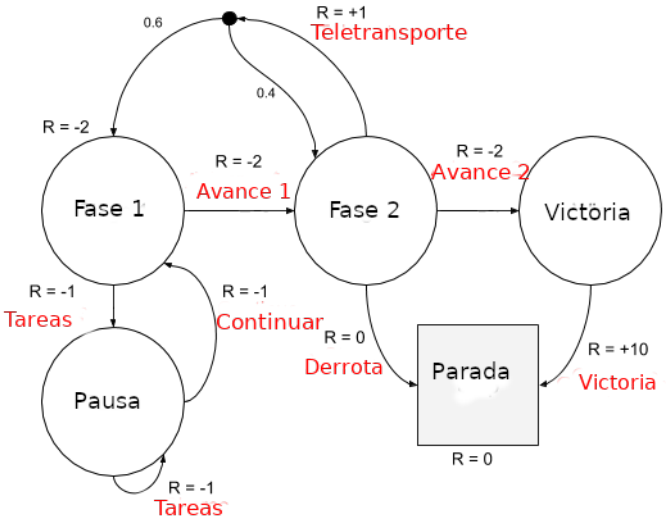


Figura 2.3: En esta extensión de nuestro ejemplo, añadimos acciones (decisiones) que el agente debe seleccionar, representadas como etiquetas en las aristas del autómata.

En los procesos de decisión se tiene más control sobre qué estados se visita. Tomando como ejemplo la figura anterior, si ejecutamos la acción *teleport*, habrá un 40 % de probabilidad de quedarse en el estado 2 y un 60 % de volver al estado 1. El resto de acciones ocurren con un 100 % de probabilidad.

Éstos modelos son los que se toman como base en el diseño de entornos de ejecución de aprendizaje por refuerzo. A continuación, introduciremos diversos conceptos adicionales que facilitarán la comprensión del problema.

2.1.7. Recompensas y función de retorno

Como ya se ha mencionado anteriormente, el objetivo de los agentes de aprendizaje por refuerzo consiste en **maximizar** la recompensa acumulativa futura. Nos referiremos a este parámetro como rendimiento **(return)**, y se puede definir de la siguiente manera:

$$R_t = R_{t+1} + R_{t+2} + R_{t+3} + R_{t+4} ... R_n = \sum_{k=0}^{n} r_t + k + 1$$

Esta definición asume que el número de pasos a seguir hasta llegar a un nodo final es **finito**, es decir, que se trata de una tarea **episódica**.

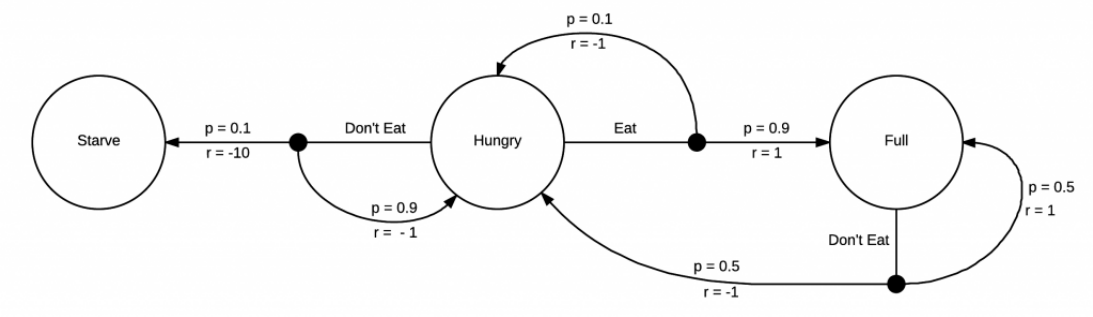
Normalmente estos valores no se introducen directamente sino que vienen dados en función a un **factor de descuento** γ donde 0 <γ <1.

$$R_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + ... + \gamma^{n-1} R_n = \sum_{k=0}^{n} \gamma^k r_t + k + 1$$

Este acercamiento resulta útil ya que da mayor importancia a las recompensas cercanas, de manera que se pueden conseguir mejores recompensas a corto plazo. Hay casos especiales en los que se puede utilizar un factor de descuento con valor 0 o 1. Si γ es 1, resultamos en la ecuación inicial en la que no teníamos en cuenta el factor de descuento, por lo que se tienen en cuenta todas las recompensas acumuladas sin importar cómo de lejos estén en el tiempo. Por el contrario, si nuestro γ es 0, solamente se tendrá en cuenta la recompensa inmediata, resultando en un comportamiento similar al de un algoritmo voraz.

2.1.8. Políticas (policies)

Una política π(s,a), describe una manera de actuar de un agente. Es una función que devuelve la probabilidad de tomar cierta acción *a* en un estado *s*. Tomando el siguiente ejemplo, desde el estado *hungry* podemos tomar dos acciones: comer o no comer.



Nuestra política describirá cómo actuar en cada estado. Una política aleatoria sería, por ejemplo: π(hambre, E) = 0.5, π(hambre, $\not$E) = 0.5, π(lleno, $\not$E) = 1.0, donde E sería la acción de comer y $\not$E la acción de no comer. Esto se traduce en que en caso de que el agente se encontrara en el estado *hungry*, escogería aleatoriamente entre comer y no comer.

La finalidad del aprendizaje por refuerzo es reconocer una política óptima π*. Una política óptima será aquella que nos especifique cómo actuar para maximizar la recompensa en cada estado.

2.1.9. Funciones de valor (value functions)

Para aprender la política óptima, hacemos uso de funciones de valor. En aprendizaje por refuerzo, se distingue entre dos tipos: La **función de valor de estado** V(s) y la **función de valor de acción** Q(s,a), también conocida como Q-Value.

La función de estado describe el valor de un estado al seguir cierta política. El resultado vendrá dado por la recompensa esperada desde el estado en que se encuentra cierto agente acorde a cierta política π'.

$$V^{\pi}(s) = \mathbb{E}_{\pi}[R_t|s_t = s]$$

Para un mismo entorno de entrenamiento, la función de estado puede variar en función de la política que se aplica. Esto se debe a el valor del estado cambia en función de los pasos que el agente planee tomar para llegar al estado final.

La otra función de valor se llama **función de valor de acción**, y nos proporcionará el valor asociado a tomar una acción en un estado al seguir una política específica π.

$$Q^{\pi}(s,a) = \mathbb{E}_{\pi}[R_t|s_t = s, a_t = a]$$

2.1.10. La ecuación de Bellman

Richard Bellman fue un investigador estadounidense que formuló lo que también se conoce como la **ecuación de la programación dinámica**, que describe una condición necesaria para la optimalidad en la resolución de problemas Markovianos. Esta ecuación constituye un elemento esencial para la comprensión de los algoritmos de aprendizaje por refuerzo, y aplica muchos de los conceptos introducidos en secciones anteriores. Definiremos P y R de la siguiente forma:

$$P^a_{ss'} = P_r(s_{t+1} = s' | s_t = s, a_t = a)$$

P es la función de probabilidad de transición. Tomando como inicio el estado s y al ejecutar la acción a, llegaremos al estado s' con una probabilidad $P^a_{ss'}$.

$$R^a_{ss'} = \mathbb{E}[r_{t+1} | s_t = s, s_{t+1} = s', a_t = a]$$

$R^a_{ss'}$ es otra manera de denotar la recompensa esperada que el agente recibirá empezando desde el estado s, tomando la acción a y pasando al estado s'.

A partir de estas ecuaciones podemos derivar las ecuaciones de Bellman. Primero, a partir de la definición de retorno podremos reescribir la función $V^\pi(s)$:

$$V^\pi(s) = \mathbb{E}_\pi[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... | s_t = s] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s]$$

Si eliminamos la primera recompensa del sumatorio, se escribirá de la siguiente manera:

$$V^\pi(s) = \mathbb{E}_\pi[r_{t+1} + \sum_{k=1}^{\infty} \gamma^k r_{t+k+1} | s_t = s]$$

Este valor describe lo que se espera recibir como recompensa si continuamos aplicando la política π a partir del estado s. Esto se describe como el sumatorio de las recompensas de todas las acciones que se espera realizar sobre todos los estados posteriores. Las siguientes ecuaciones nos ayudarán a explicar el siguiente paso:

$$\mathbb{E}_\pi[r_{t+1} | s_t = s] = \sum_a \pi(s,a) \sum_{s'} P^a_{ss'} R^a_{ss'}$$

$$\mathbb{E}_\pi[\gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} | s_t = s] = \sum_a \pi(s,a) \sum_{s'} P^a_{ss'} \gamma \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k r_{t+k+2|s_{t+1}=s'}]$$

Desde aquí, podemos manipular la ecuación a la siguiente fórmula:

$$V^\pi(s) = \sum_a \pi(s,a) \sum_{s'} P^a_{ss'} [R^a_{ss'} + \gamma \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k r_{t+k+2} | s_{t+1} = s']]$$

Como se puede observar, la parte final coincide con la formulación que hemos hecho de V(s), así que sustituimos:

$$V^{\pi}(s) = \sum_{a} \pi(s,a) \sum_{s'} P^{a}_{ss'}[R^{a}_{ss'} + \gamma V^{\pi}(s')$$

La ecuación de Bellman para la función de valor de acción se puede derivar de manera similar:

$$Q^{\pi}(s,a) = \sum_{s'} P^{a}_{ss'}[R^{a}_{ss'} + \gamma \sum_{a} \pi(s',a') Q^{\pi}(s',a')]$$

La importancia de las ecuaciones de Bellman viene de que nos permiten expresar valores de estado como valores de otros estados; es decir, si tenemos el valor del estado s_{t+1}, fácilmente podremos calcular el valor del estado s_t. Esto permite abordar el problema desde un punto de vista iterativo de una manera mucho más eficiente y directa que con los enfoques tradicionales.

2.2 Introducción al Q-Learning

El aprendizaje por refuerzo es un principio altamente utilizado en la actualidad en la investigación del campo de la Inteligencia Artificial. Este enfoque se inspira en el campo de la psicología de comportamiento y, como ya se ha explicado en secciones anteriores, pone el énfasis en cómo un agente independiente será capaz de tomar las acciones pertinentes basándose en un sistema de recompensas suministradas en función de las acciones que toma en cada momento. En este sentido, se intenta emular la capacidad de los seres vivos de hacer predicciones basándose en la información que se percibe del entorno. Estas predicciones [5] proporcionan a los animales tiempo para preparar reacciones de comportamiento, que en un futuro pueden mejorar las opciones de las que este disponga en el futuro. De esta manera, se habla de recompensa como un concepto que describe un valor positivo que una criatura le atribuye a un objeto, patrón de comportamiento o estado físico interno. Estas se manifiestan en una gran variedad de animales a través de un neurotransmisor llamado dopamina. En estos sistemas, esta se produce como respuesta a una recompensa inesperada que, si se produce de manera repetida a lo largo del tiempo, puede condicionar el comportamiento del organismo. Frecuentemente, se relaciona la dopamina con funciones vitales tan importantes como la motivación, la regulación de la memoria o en la toma de decisiones. Sirve, por lo tanto, como un método de refuerzo positivo, en situaciones en las que estas respuestas constituyan un patrón de comportamiento recurrente que se mantenga a lo largo del tiempo.

2.2.1. Redes neuronales

El concepto de red neuronal nace de la idea de conseguir un comportamiento autónomo similar al de un ser humano. Las redes neuronales intentan replicar el razonamiento humano creando una relación entre lo que el agente observa y las acciones que ha de realizar.

Las redes neuronales se basan en el modelo de aprendizaje supervisado. Consisten de una red de procesadores conectados llamados **neuronas**, pudiendo ser cada una de ellas activada en función de las señales de entrada que recibe. Estas vienen dadas a partir de información que se percibe del entorno, mientras que las neuronas de las capas ocultas se activan a partir de las interconexiones que se crean con las capas anteriores. Esto se puede observar claramente en la figura 2.4.

Dependiendo de la naturaleza de la información que se introduce, puede ser necesario modificar las redes para crear las estructuras específicas adaptadas al tipo de datos que se vayan a utilizar. Un ejemplo de esto serían las redes convolucionales.

Una red neuronal se compone esencialmente de tres componentes interconectados: Una capa de entrada, un conjunto de capas ocultas y una capa de salida.

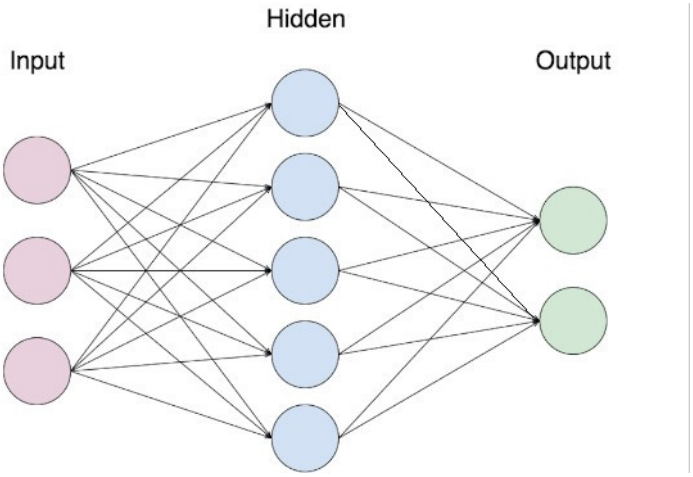


Figura 2.4: Ejemplo de una red neuronal simple con una capa oculta de 5 nodos (neuronas).

En la figura anterior se puede observar claramente que la capa de entrada recibirá los datos tal y como se perciben. Desde ese punto, estos datos irán siendo ponderados y transmitidos a la siguiente capa. En la mayoría de casos, existirá una conexión entre cada neurona y todas las neuronas de la capa siguiente. A su vez, cada neurona realizará una ponderación y normalización de las señales de entrada recibidas para a su vez emitir una señal de salida basada en estos *inputs* recibidos. Una vez la información es procesada a través de todas las capas ocultas, llegará la señal a la capa de salida. Esta será responsable entonces de activar las neuronas necesarias, que constituirán la predicción realizada y que, dependiendo del grado de entrenamiento por la que la red haya pasado anteriormente, sera más o menos acorde con el resultado real. Este proceso se conoce como *forward propagation*.

Con respecto a las neuronas, su valor se extraerá de realizar una **suma ponderada** de las señales que recibe, para ser pasadas posteriormente por una función de activación. Normalmente, a este sumatorio se le agrega un valor llamado sesgo (*bias*). Se ha demostrado que esta adición puede reducir el tiempo necesario de entrenamiento de una red neuronal. En la figura siguiente, se introduce el funcionamiento interno de una neurona individual.

La función de activación de nuestra neurona normalizará el resultado y devolverá un valor binario [0,1] dependiendo de la magnitud de entrada resultante del sumatorio de los distintos pesos. Existe un gran número de funciones de activación distintas cuyo funcionamiento repercutirá enormemente en el comportamiento general de la red. Será necesario, por tanto, realizar multitud de ensayos de prueba y error para determinar qué función de activación utilizar en cada caso particular.

El concepto de backpropagation

Dada la naturaleza de aprendizaje supervisado de nuestro problema, se puede determinar el valor correcto que las capas de salida de una red neuronal deben tomar. El *backpropagation* utiliza esta ventaja que nos ofrecen los entornos deterministas para realizar así una corrección de cómo el sistema maneja las señales de entrada. Así, la red

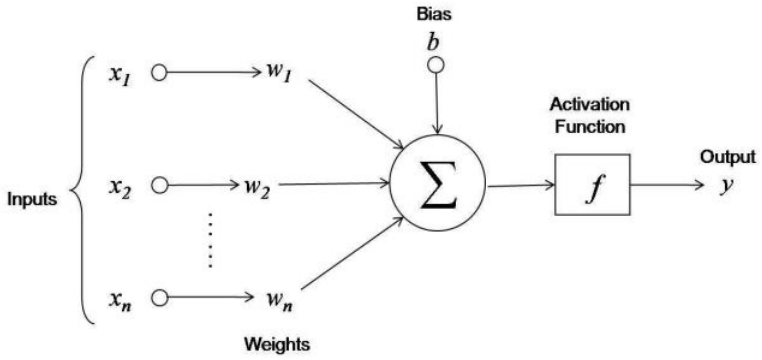


Figura 2.5: Funcionamiento interno de una neurona

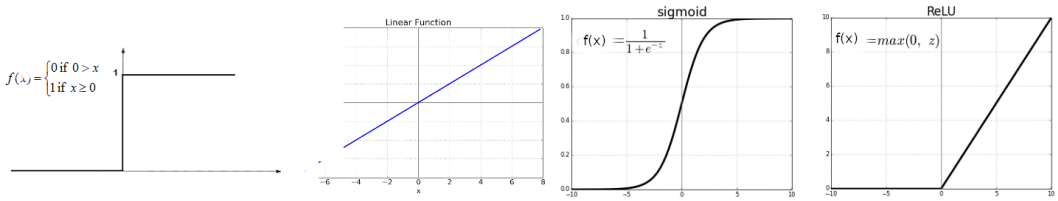


Figura 2.6: Ejemplos de funciones de activación. De izquierda a derecha, función step, linear, sigmoid y ReLU.

neuronal es capaz de ajustar sus funciones de activación para poder realizar una mejor clasificación en pruebas posteriores.

Una vez la tarea de predicción del forward propagation termina, una **función de pérdida** evaluará la diferencia entre el valor de salida y el valor real que la red debería tomar. Dado que se trata de un modelo de entrenamiento supervisado, se tendrá acceso al valor real a partir de una muestra de entrenamiento. Basado en el coste calculado por la función de pérdida, la red irá corrigiendo sus vectores de pesos capa por capa, aproximando así sus predicciones a un valor más óptimo de manera recursiva en cada iteración. [14]

2.2.2. Redes neuronales convolucionales (CNNs)

Las redes neuronales convolucionales tratan de resolver el problema de utilizar imágenes como datos de entrada para redes neuronales. Dado que las neuronas en la estructura de red tradicional sólamente están preparadas para soportar valores individuales, es necesaria la creación de una estructura adicional que sea capaz de manejar estas colecciones de píxeles. Para abordar este problema, se hace uso de un proceso de extracción de características anterior a la clasificación (que es llevada a cabo por una red neuronal tradicional). A continuación, abordaremos cada una de las capas y explicaremos su finalidad y funcionamiento interno.

El preprocesamiento de imágenes llevado a cabo viene ilustrado en la siguiente figura:

Normalmente, una CNN recibe una matriz tridimensional, o **tensor** como entrada, consistente en una imágen con H filas, W columnas y tres canales de color (R,G,B). Esta señal pasa por una serie de procesos antes de ser administrada a la red neuronal. Cuando hablamos de capas, nos referimos a cada uno de estos procesos independientes.

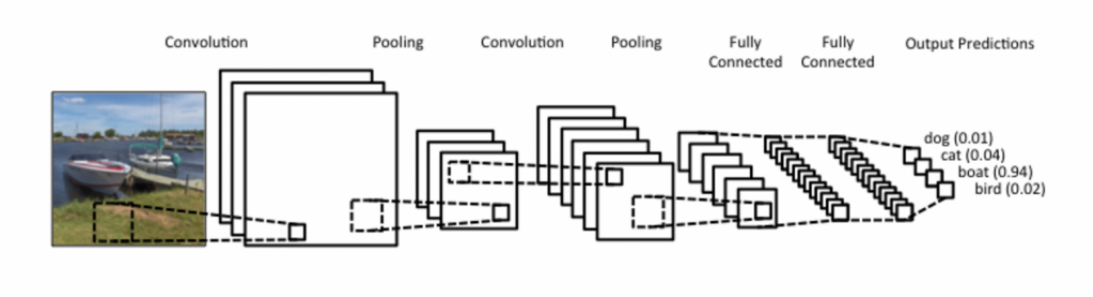


Figura 2.7: Ejemplo de red convolucional

Capa convolucional o detector de características

La capa convolucional esencialmente reúne información contenida en conjuntos de píxeles adyacentes. Para ello, se hace uso de un filtro, o **kernel**, consistente en una matriz de menor orden para la detección de características. Este filtro realizará una reducción de dimensionalidad de la matriz original, dando como resultado un **mapa de características**. En concreto, sopesa cada región de la imágen en función del filtro introducido, y devolverá un valor escalar a esa región en función a su afinidad con el filtro.

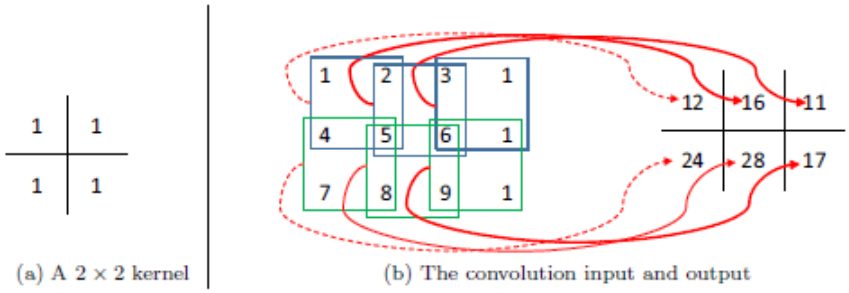


Figura 2.8: Ilustración de la operación de convolución[3]. A la izquierda, el filtro utilizado para el ejemplo. Como se observa a la derecha, se reduce la dimensionalidad de la matriz inicial e introduce el resultado de la convolución como la suma de los vectores de pesos de cada píxel de la matriz: 1 x 1 + 1 x 4 + 1 x 2 + 1 x 5 = 12.

Capa de agrupamiento (pooling)

Esta capa reduce de nuevo el tamaño de la matriz extrayendo de ella la información más remarcable de cada grupo adyacente de características.

Existen dos tipos de agrupamiento ámpliamente usados cuando hablamos de CNNs: agrupamiento máximo (max pooling) y agrupamiento promedio (average pooling). En el primero, se extrae el valor máximo encontrado en cada región del mapa, mientras que en el segundo se extrae un valor promedio. La matriz resultante tendrá una dimensionalidad menor que la introducida, siendo la magnitud de esta reducción dependiente del filtro aplicado.

Capa de aplanamiento

Una vez se ha realizado la tarea de reducción de dimensionalidad y se dispone de un mapa de características de dimensiones aceptables, es necesario pasar éste por un último filtro que transforme la matriz a un tamaño n x 1. A partir de este momento es posible introducir el mapa de características en una estructura de red neuronal clásica.

2.2.3. Redes recurrentes (RNNs)

Llegados a este punto es necesario describir un tipo de red neuronal particular que nos servirá para más adelante introducir un tipo de algoritmo a aplicar en el entorno Doom. Se trata de las redes neuronales recurrentes.

Se trata de una red neuronal estándar modificada para incluir bucles de realimentación. Es decir, esta red permite que la información que se ha procesado en el pasado por la red persista durante un número indeterminado de episodios y sea accesible como conocimiento para iteraciones posteriores. Esto se consigue incrustando los datos de salida resultantes de vuelta en los datos de entrada de la red (embedding).

Uno de los principales problemas que las redes neuronales tradicionales no han sido capaces de solucionar consiste en la **persistencia de información** entre estados. En muchos entornos, es trivial deducir que la información en los estados pasados puede de la misma manera ser útil para tomar decisiones en el presente. Los estudios sobre redes tradicionales históricamente han abarcado este problema alimentando a la red con un **historial de estados** que puede contener la información de un número reducido de estados anteriores. En entornos como el Pong esto es vital, ya que un solo *frame* no proporciona información sobre la dirección que sigue la bola y, por lo tanto, sin esta memoria el agente no sería capaz de tomar una decisión adecuadamente informada.

Las **redes neuronales recurrentes** no necesitan de estos mecanismos, ya que permiten que **la información persista** entre las diferentes iteraciones del entrenamiento.

Estas redes son capaces de tratar secuencias de información de tamaño indeterminado, sin ser necesario ajustar la capa de entrada para esta función. Esto se debe a que en estas redes no se tiene en cuenta desde qué capa llega la información y todos los *inputs* se procesan de la misma forma. Además, como ya hemos mencionado son muy útiles para tratar colecciones de datos secuenciales.

Uno de los problemas de utilizar este tipo de redes es que el tiempo de entrenamiento aumenta considerablemente, pero distintos estudios han corroborado su utilidad en entornos parcialmente observables, haciendo que el entrenamiento converja antes que DQN u otras variaciones del mismo.

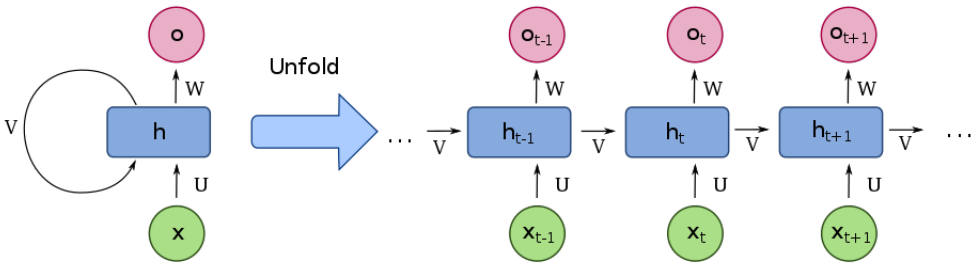


Figura 2.9: A la izquierda, un modelo de red neuronal recurrente básica. A la derecha observamos el mismo modelo de red neuronal "desarrollado", donde observamos que los *outputs* de cada iteración son realimentados a la red neuronal, dotándola así de una suerte de memoria a corto plazo.

A pesar de su diseño, las RRN no consiguen buenos resultados en entornos que requieren estrategias a más largo plazo. A continuación, introduciremos brevemente un tipo de red basada en RNN que nos podría ser útil para experimentaciones en ciertos entornos:

Redes LSTM

Uno de los principales reclamos de las redes recurrentes es la capacidad de çonectarïnformación del pasado con estados presentes. A pesar de que teóricamente esta conexión podría darse a través de estados muy distantes temporalmente, esto no es posible en la práctica. Estudios han corroborado que esta capacidad de inferencia es mucho más frecuente entre estados próximos. Es por ello que si se desea estudiar entornos donde las decisiones tengan una mayor influencia a largo plazo, es necesario utilizar una **estructura de red con memoria a largo-corto plazo** (Long-Short Term Memory networks o LSTM en inglés).

Esta red es un tipo especial de RNN que es capaz de aprender dependencias a largo plazo. El concepto fue introducido en 1997 por Hochreiter y Schimdhuber [17] y fue refinado y popularizado durante los años posteriores.

Las redes LSTM están diseñadas explícitamente para evitar el problema de la dependencia a largo plazo y son capaces de recordar información por periodos largos de tiempo.

2.3 Algoritmos existentes

Durante los últimos años, ha habido un auge de investigación en el campo del aprendizaje por refuerzo, y muchos algoritmos han sido diseñados con la intención de construir unos algoritmos eficientes e intuitivos. Éstos algoritmos se podrían dividir en distintas categorías, como por ejemplo, aquellos que utilizan técnicas de **descenso por gradiente** (A2C, A3C), aquellos que hacen uso de **redes neuronales de aprendizaje profundo** (DQN), o basados en **estrategias evolutivas**. En este capítulo se describirán algunos de los algoritmos que podrían ser empleados para resolver el problema.

2.3.1. DQN: Deep Q-Network

Podríamos decir que DQN [6] se trata de la versión más "pura"del aprendizaje por refuerzo que trataremos en este trabajo, además de la más simple. Para su implementación, generalmente se hace uso de una **red neuronal convolucional** (descrita en la sección 2.2.2) con **backpropagation**. Un agente comienza a ejecutarse sobre un entorno que le es desconocido, por lo que comienza a ejecutar acciones aleatorias del espacio de acciones. A medida que se suceden experiencias, el agente irá actualizando los valores de su red neuronal, asociando así las acciones más exitosas que haya experimentado con una mayor recompensa, lo que se traducirá en una mayor probabilidad de seleccionar dicha acción. La red convolucional se utiliza para reducir la dimensionalidad de los *frames* introducidos y así conseguir una red neuronal con un menor número de nodos de entrada. Se introduce además una estructura de *experience replay* (sección 2.3.1) donde se almacenan las experiencias pasadas del agente.

En algunas implementaciones se hace uso de una **segunda red neuronal *q-network***, que se utiliza para generar los valores Q objetivos que se espera alcanzar por la red principal. Esta red se ajusta frecuente y lentamente a medida que se realiza la tarea de exploración, ya que si se actualiza el objetivo con mayor frecuencia podría complicar la convergencia de las redes, resultando perjudicial para el aprendizaje [9]. Para lidiar con estos problemas, se hace uso de una serie de técnicas que a continuación se exponen:

El *experience replay*

Normalmente, para una correcto aprendizaje en entornos estocásticos no basta con la implementación de una red neuronal. En su definición más simple, los algoritmos de aprendizaje descartan la información respectiva a experiencias anteriores inmediatamente después de decidir la acción a tomar.

Resulta útil un mecanismo que sea capaz de almacenar y usar de nuevo las experiencias pasadas del agente, ya que soluciona distintos problemas que pueden surgir si no se sigue este enfoque. Por ejemplo, es interesante el tener constancia de estados poco comunes que requieran unas acciones específicas para conseguir un resultado favorable en posteriores iteraciones. Este mecanismo es el denominado *experience replay*.

Una de las ventajas de utilizar una implementación de estas características es que evita el sobreentrenamiento. Al no entrenar la red con los estados inmediatamente anteriores, enormemente relacionados al ser secuenciales, se consigue un aprendizaje más diverso al utilizar una selección aleatoria de experiencias pasadas.

Adicionalmente, [4] argumenta que priorizar las experiencias que el agente accede en memoria puede tener un efecto positivo en la rápida convergencia del aprendizaje, partiendo de la idea de que existirán experiencias de las que el agente podrá aprender de manera más efectiva que otras, o bien existir algunas que puedan no ser útiles para un estado determinado y que más tarde pasen a ser vitales para la correcta ejecución de cierta secuencia de acciones. El principal problema de esta ordenación por prioridad será entonces determinar el criterio por el cual se mide la importancia de cada transición de estados. Idealmente, esto se realizaría tomando como criterio la cantidad de información que el agente puede aprender de la transición desde el estado actual. Sin embargo, este no es un factor que podamos calcular. En su lugar, una aproximación adecuada podría ser tomar el error TD de una transición δ, lo cual nos daría una noción de cómo de 'sorprendente' o innovadora determinada acción resultará.

Esta solución no viene sin problemas, ya que este algoritmo de priorización acabará centrándose en un pequeño grupo de experiencias que se repetirán de manera frecuente. Además el error se reducirá lentamente con el paso de cada iteración, lo cual podría desembocar a su vez en problemas de sobreajuste[2].

[4] concluye después de realizar experimentaciones en el entorno de Atari 2600, que una correcta implementación del *experience replay* basado en prioridades podría aumentar el aprendizaje conseguido por un factor de 2.

La técnica del Target Network

Otra de las razones de la inestabilidad de DQN son los cambios frecuentes que se aplican a la función objetivo.

$$\theta_{t+1} = \theta_t + \alpha(Y_t^Q - Q(s_t, a_t; \theta_t))\nabla\theta_t Q(s_t, a_t; \theta_t)$$

$$Y_t^Q = r_{t+1} + \gamma \max_a Q(s_{t+1}, a; \theta_t)$$

El truco de la técnica del Target Network consiste en fijar los parámetro de la función objetivo $Q(s, a; \theta_t)$ a un valor constante durante un número de iteraciones previamente

[2] El sobreajuste o *overfitting* en inglés, es el efecto de sobreentrenar un algoritmo de aprendizaje para encontrar soluciones que ya se conocen.

definido. Al final de cada episodio, el valor de la función objetivo es actualizado con el último valor de la red.

$$Y_t^{DQN} = r_{t+1} + \gamma \max_a Q(s_{t+1}, a; \theta_t^-)$$

Reward clipping

Al aplicar DQN a entornos con diferente configuración en los que probablemente las recompensas no tengan el mismo peso, el entrenamiento se puede volver ineficiente. Esto se puede solventar aplicando una normalización de las funciones de recompensa. De esta manera, todas las recompensas positivas tendrán un valor de +1 y aquellas recompensas negativas un valor fijo de -1. Se consigue así evitar grandes actualizaciones de pesos y permite a la red actualizar sus parámetros de manera continua.

2.3.2. A2C - Advantage Actor Critic

El algoritmo A2C es una primera aproximación a lo que engloba los **algoritmos de tipo actor-crítico**, en los que existen múltiples procesos con funciones bien diferenciadas:

- En primer lugar, el proceso "**crítico**"tienen como función modificar la red a partir de las experiencias recogidas y **estimar la función de valor** que después se comunicará al resto de procesos.
- El "**actor**"se encarga de ejecutar las políticas estimadas por el crítico sobre una copia del entorno a resolver. Este proceso cuenta con una **red neuronal independiente** que toma como función de entrada la estimación del crítico para devolver la acción a ejecutar.

El concepto de **ventaja** presente en el nombre del algoritmo se debe a que, a diferencia de algoritmos que utilizan técnicas de descenso por gradiente (DQN, PPO), este tipo de algoritmos introducen éste concepto para estimar no solamente cómo de buenas o malas fueron las acciones tomadas, sino tambien en qué medida estas acciones serían mejores o peores de lo que se pudiera esperar. Esto permite al algoritmo centrarse en la parte de la red neuronal central (crítico) donde las predicciones se ajustan menos a la realidad e intentar corregirlas.

En la figura 2.10 se expone un esquema del proceso interno de funcionamiento que constituyen los procesos *worker*.

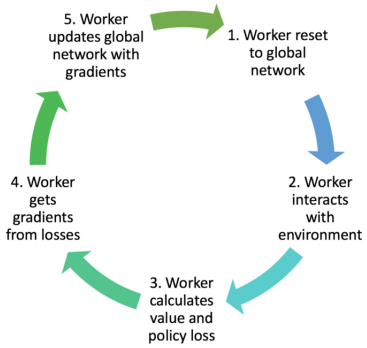


Figura 2.10: Esquema de trabajo de actor worker en algoritmos actor-crítico.

2.3.3. A3C: Asyncronous advantage actor-critic

A3C [1] se trata de uno de los algoritmos más competitivos que existen en la actualidad. Esencialmente supera a DQN en todos los aspectos, siendo este más rápido, simple, robusto y capaz de conseguir puntuaciones mucho más altas en una gran mayoría de tareas típicamente asignadas para la validación de estos algoritmos. En esencia, el algoritmo consiste en una **versión asíncrona de A2C** en la que es posible introducir múltiples entornos *worker* con la consecuente ventaja de poder obtener un significante aumento del número de experiencias procesadas.

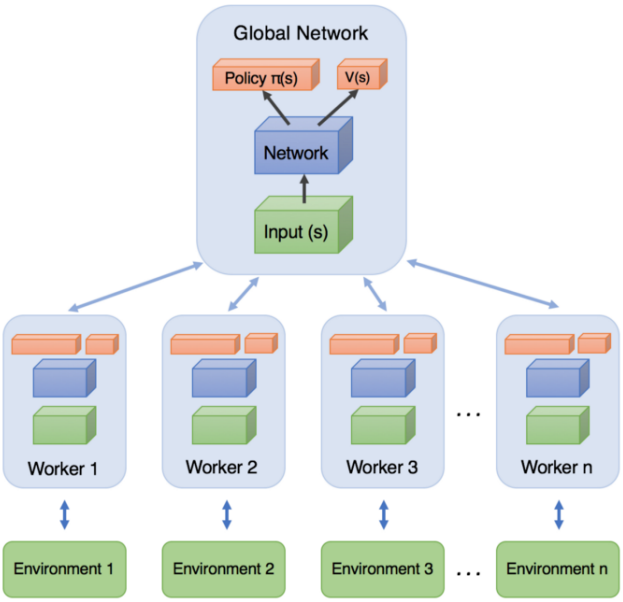


Figura 2.11: Diagrama de arquitectura de algoritmo A3C. Cada agente, sea *worker* o crítico cuenta con su propia red neuronal sobre la que ejecuta operaciones. Las salidas de las redes *worker* se utilizan como valores de entrada para el crítico.

A diferencia de DQN, donde un agente individual representado por una red neuronal interacciona con un solo entorno, A3C utiliza múltiples agentes para aprender de manera más eficiente. Al tratarse de un algoritmo del tipo actor-crítico, cuenta con una red global para estimar las funciones de valor y una serie de *workers* independientes a los que se les asigna su propia serie de parámetros de aprendizaje. Cada uno de éstos *workers* interacciona con su propia copia del entorno de manera simultánea. La premisa que justifica esta solución como una mejora ante A2C (excluyendo la aceleración del aprendizaje al realizar múltiples experiencias simultáneas) es que la experiencia de cada agente tomará decisiones independientes al resto, lo que permitirá unas experiencias más diversas, y por lo tanto, permitirá una exploración del entorno mucho más en profundidad. No obstante, diversos estudios han comprobado que ámbos algoritmos producen un rendimiento similar, siendo en muchos casos A2C una opción más eficiente.

2.3.4. IMPALA: Importance Weighted Actor-Learner Architecture

IMPALA [8] es uno de los algoritmos más recientes desarrollados por el equipo de DeepMind, basándose en el algoritmo A3C descrito anteriormente.

Como se ha descrito, en este tipo de modelos cada actor hace uso de una copia de los parámetros de la política del actor central. De manera periódica, los actores pausan la exporación para compartir los valores de gradiente actualizados con el servidor central de parámetros que aplica una actualización distribuida al resto de actores.

Se trata de un algoritmo cuyo objetivo consiste en conseguir el dominio de un mismo agente en una serie de entornos distintos. En este algoritmo se hace uso de distintos tipos de agentes: en primer lugar, una serie de actores que se ejecutan en el entorno y transmiten repetidamente las **experiencias** generadas a uno o varios actores centrales, llamados *learners*, que a su vez transmiten al resto los parámetros que constituyen la política.

A diferencia de A3C, en el que los *workers* comunican gradientes con respecto a la política del actor central, los actores de este algoritmo comunican "trayectorias de experiencia"(secuencias de estados, acciones y recompensas) a un actor de aprendizaje centralizado que resulta en un modelo con actores completamente independientes. La separación del aprendizaje en el servidor central de la actuación de los agentes independientes permite aumentar el rendimiento del sistema, ya que los agentes no deben esperar los resultados del aprendizaje ya que estos se proporcionan de manera síncrona y periódica.

Esta ”desconexión” entre los actores y el aprendizaje causa que las políticas de los primeros tengan un rendimiento inferior al que pudieran tener. Es por ello que se introduce la formulación de un factor llamado *V-trace* que compensa las trayectorias de los actores que no actúan dentro de una política determinada.

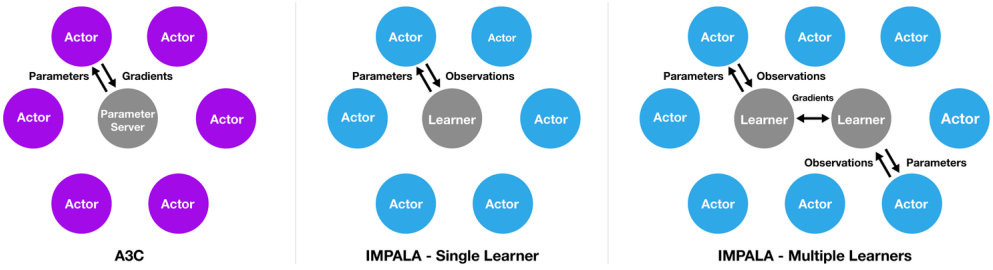


Figura 2.12: Comparativa de comunicación entre A3C e IMPALA

2.3.5. PPO - Proximal Policy Optimization

En 2017, el equipo de OpenAI propuso este nuevo algoritmo de aprendizaje con el objetivo de que fuera escalable, eficiente y robusto. La premisa principal de PPO [10] consiste en evitar grandes variaciones en las políticas aplicadas haciendo cambios de manera progresiva con la técnica de **descenso por gradiente**. De esta manera, se pretende influenciar al agente para que a medida que avance la exploración, ejecute acciones que aumente la recompensa total y evite el resto.

Este algoritmo, al igual que DQN utiliza la técnica del *backpropagation* para actualizar las funciones de pérdida y aplicar la optimización.

2.3.6. Estrategias evolutivas

A continuación, introduciremos una técnica de optimización conocida por décadas que se presenta como una de las alternativas al aprendizaje por refuerzo. Se trata de las denominadas **estrategias evolutivas** [11].

Las estrategias evolutivas son una clase de algoritmos de optimización de que utilizan procedimientos de búsqueda heurística. Esta técnica se pueden utilizar en problemas de una gran variedad de ámbitos y sigue los principios de la teoría de la evolución propuesta por Darwin en 1859 en su ensayo El Origen de las Especies.

En el ensayo de Darwin, un conjunto inicial de individuos habita un entorno. Cada individuo debe competir por los recursos con el resto de la población, y es en este contexto en donde entra en juego su adaptabilidad. Los individuos que poseen alguna ventaja evolutiva sobre el resto son más propensos a subsistir en el entorno. Es posible, por tanto, que estos individuos sean capaces de sobrevivir más tiempo y se reproduzcan transmitiendo aquellas ventajas a sus sucesores. Sin embargo, los cambios que puede sufrir un sujeto no siempre vienen dados por herencia, sino que pueden también aparecer fruto del azar en forma de mutaciones genéticas.

La población no es una entidad estática. A medida que el tiempo pase, las generaciones se irán sucediendo y sufriendo cambios. A continuación, diseccionaremos los diferentes procesos sobre los que se basan estas estrategias evolutivas:

- **Inicialización:** Se crea una población inicial que habita el entorno. Los individuos pueden ser muy diversos, al igual que su nivel de adaptación al medio.
- **Evaluación:** Representa el comportamiento de los especímenes ante el entorno. Se mide así su nivel de adaptación al mismo.
- **Selección:** Este proceso representa la competencia de los individuos para prosperar en un entorno. Aquellos individuos mejor adaptados tendrán mejores posibilidades de sobrevivir y reproducirse, mientras que aquellos menos adaptados son más propensos a desaparecer del entorno en futuras iteraciones.
- **Cruce:** Este paso hace referencia al proceso de reproducción entre dos individuos con el fin de generar uno o diversos descendientes con características de cada uno de los progenitores. La combinación de características es aleatoria y no garantiza que el ser descendiente sea mejor que sus progenitores.
- **Mutación:** Con una probabilidad aleatoria, hay ocasiones en las que se generarán cambios en las características de un individuo que pueden suponer una ventaja sobre el resto de la población. De la misma manera, es posible que la mutación reduzca el nivel de adaptación del individuo al entorno.
- **Reemplazo:** Se trata de la representación del paso generacional presente en la naturaleza. Al cabo del tiempo, los habitantes de una población acaban siendo sustituidos por una nueva generación de individuos resultantes del cruce y en algunos casos, la mutación. Una vez sucede el reemplazo, se vuelve a tomar la población resultante como inicial y se vuelve a repetir el proceso.

Es necesario aclarar de que, a pesar de utilizar estos mecanismos, las estrategias evolutivas difieren mucho de la forma en cómo la evolución funciona en la naturaleza. En nuestro caso, es más acertado ver las estrategias evolutivas como un **optimizador de caja negra**, con unos parámetros de entrada $\mathbf{w}$ (en nuestro caso la información del entorno,

pero puede adaptarse a problemas muy diferentes) y cuya meta es optimizar la funcion **f(w)** para que devuelva la recompensa máxima.

En el caso del aprendizaje por refuerzo, las mecánicas que el algoritmo sigue son básicamente prueba y error. Empezamos con unos parámetros aleatorios (población inicial) y de manera iterativa modificamos estos parámetros para acercarnos progresivamente a un mejor resultado con cada iteración. Al final de cada iteración, el algoritmo selecciona los parámetros que mejor han funcionado como población a modificar y a partir de éstos inicializar una nueva población.

Las estrategias evolutivas son más sencillas de implementar que la mayoría de algoritmos de aprendizaje por refuerzo, ya que no hay necesidad de implementar un método de *backpropagation*. Además, ofrecen una **exploración del entorno exhaustiva y estructurada** no alcanzable por los algoritmos descritos anteriormente.

2.4 Aplicación en videojuegos

Por norma general, los videojuegos constituyen entornos idóneos para la investigación en los campos del aprendizaje por refuerzo por diversas razones. En primer lugar, suelen ser productos inicialmente diseñados para ser controlados por jugadores humanos, por lo que pueden constituir un buen marco de referencia para la valoración del aprendizaje que un agente lleva a cabo. Además, estos sistemas proveen un entorno con unas reglas ya marcadas, en los que la información que se provee al jugador suele ser suficiente para valorar si se están tomando o no las decisiones correctas, bien a través de una puntuación atribuida a la partida, la duración de la misma o la consecución de una serie de objetivos previamente definidos. Por último, el reducido número de acciones que el jugador puede realizar (dimensionalidad) permite crear correlaciones mucho más exactas entre los pares acción recompensa, y facilita enormemente la tarea de aprendizaje.

En este documento, nos centraremos en la aplicación de los algoritmos previamente definidos a videojuegos de la videoconsola Atari 2600, comercializada en 1977 y con un extenso catálogo consistente en famosos juegos arcade como Pacman, Pong, Pinball o Space Invaders. Estos juegos fueron inicialmente diseñados para resultar desafiantes y difíciles de dominar por jugadores humanos, pero dado su baja dimensionalidad, constituyen entornos de estudio muy interesantes.

En estudios recientes [6], se demostró la capacidad de una variante del algoritmo DQN de superar exponencialmente el rendimiento de un jugador humano sobre los mismos entornos. En este estudio se hizo uso de una red convolucional que aplicaba los principio de la diferencia temporal para determinar factores como la trayectoria de los objetos o la diferencia de puntuación, atribuyendo así una correlación entre las acciones llevadas a cabo por el agente y su rendimiento a corto plazo. Adicionalmente, en estas implementaciones se hizo uso de un mecanismo de *experience replay* que aseguró que el agente fuera capaz de usar las experiencias del pasado para mejorar su rendimiento, y una función que actualizara múltiples valores de acción (*Q-values*) de manera periódica en lugar de realizar una actualización individual en cada instante de tiempo, con la intención de reducir las correlaciones entre acciones específicas y su resultado inmediato.

Para trabajar con estos entornos y basándonos en la implementación realizada por el equipo de DeepMind [7], tomaremos las imágenes del videojuego con una dimensión de 210x160 píxeles con una paleta de 128 colores y aplicaremos un preprocesado que reduce la dimensionalidad hasta una región de 84x84 que abarca el entorno del juego. En la aplicación de su algoritmo, en cada instante de tiempo se toman los últimos cuatro *frames* para este proceso, y se introducen como elementos de entrada a la red convolucional.

La dimensionalidad del problema puede variar entre 4 y 18 en función del juego que se seleccione.

La combinación de estas mejoras resultó en un rendimiento hasta un 2800 % superior con respecto al rendimiento máximo de un jugador humano en el entorno de Video Pinball.

CAPÍTULO 3
Análisis del problema

3.1 Introducción

En este capítulo se pretende realizar una descripción del problema al que nos enfrentamos, describiendo los entornos sobre los que vamos a experimentar y haciendo hincapié en aquellos requisitos indispensables que habremos de asegurarnos de reflejar en nuestra implementación para conseguir unos resultados válidos a analizar. Además, introduciremos el plan de trabajo seguido durante toda el proceso de realización del TFG.

En la sección 3.2 se introducirán los distintos entornos sobre los que trabajaremos, explicando las particularidades de cada uno y los potenciales desafíos a los que un agente se habrá de enfrentar para conseguir superar el aprendizaje sobre los mismos.

3.1.1. Perspectiva del trabajo

El estudio que realizamos durante toda la duración de este trabajo es un proyecto totalmente independiente que no va a ser usado con ningún fin más allá del ámbito académico, ni desempeña funciones para otro sistema.

3.2 Descripción general

En la próxima sección introduciremos por primera vez nuestro plan de solución para los entornos de Atari y Vizdoom, siendo necesario abarcar cada uno de manera diferente adaptándonos a los requisitos de las librerías que dependemos para cada uno de estos.

Por una parte, será necesario implementar un *script* que haga uso de ciertas librerías relacionadas con el aprendizaje por refuerzo, el cual nos permita realizar experimentaciones a través de diversos entornos predefinidos. Adicionalmente, se pretende realizar una implementación propia de algunos de los algoritmos vistos anteriormente - y posiblemente algunas variaciones de los mismos - haciendo uso únicamente de las librerías más básicas para el desarrollo de sistemas de aprendizaje automático, como podría ser el caso de Tensorflow, NumPy o Keras. Estas implementaciones serán testeadas sobre la librería ViZDoom, que a su vez habrá de ser modificada manualmente para su correcto funcionamiento. Además de un resultado correcto, este debería conseguirse preferiblemente haciendo un uso del tiempo de entrenamiento tan bajo como sea posible, por lo que valoraremos positivamente aquellos algoritmos que tomen menos tiempo en resolver cada entorno.

La **segunda parte de nuestro trabajo** consiste como ya se ha comentado en el diseño de un sistema de aprendizaje por refuerzo propio que implemente algoritmos ya descritos en secciones anteriores. Para ello, se harán uso de librerías relacionadas con el aprendizaje automático para, entre otros, el manejo y creación de redes neuronales, así como los cálculos que esto conlleva a más bajo nivel, o la librería OpenAI Gym para ser capaces de ejecutar nuestro entorno a medida de manera más sencilla e intuitiva.

Dado que todos los algoritmos descritos en la sección 2.3 ya vienen implementados en la librería RLLib, haremos uso de esta para comparar los resultados obtenidos con ejecuciones anteriores sobre los entornos Atari. Adicionalmente, se pretende realizar una implementación de cierta variación de DQN que, al parecer, proporciona un buen rendimiento en entornos que, como Doom, disponen de una gran cantidad de información en cada estado en el que el agente se encuentra, teniendo este solamente acceso a una fracción de la misma a diferencia de los entornos de Atari utilizados anteriormente en los que se tenía acceso a toda la información del entorno en cada estado que se exploraba. Esto es debido a que el agente está vinculado a una visión del entorno parcial, en primera persona y que por cada *frame* solamente ofrece información de lo que el agente tiene en su rango de visión de 90 grados, dejando los 270 grados a su alrededor como una incógnita. Esto viene agravado dado que tampoco se dispone de un sistema de radar, el cual pudiera potencialmente asistir en la recogida de información del entorno.

3.2.1. Experimentación sobre Atari

Como se ha adelantado en la sección de introducción, parte de nuestro problema consiste en resolver una serie de entornos predefinidos de juegos de la consola Atari 2600 mediante la aplicación de distintos algoritmos de aprendizaje por refuerzo. Después, se sopesarán los resultados obtenidos teniendo en cuenta las características que presenta cada juego; algunos podrían requerir acciones que ofrecen recompensas a largo plazo, otros pueden ofrecer una gran cantidad de recompensa a la consecución de una serie de acciones específicas, etc. Es por ello que es muy posible que obtengamos resultados distintos dependiendo del entorno que seleccionemos.

De la misma manera, estos problemas se tienen que tener en cuenta también desde el punto de vista de con cuánta potencia de procesamiento y memoria contamos en el dispositivo que realizaremos las pruebas, ya que cuanto más potente sea nuestra máquina, mas rápido se realizará un entrenamiento adecuado.

Aunque la forma concreta de la solución y las funciones del problema se desarrollarán más adelante en el capítulo 4, se puede dar una idea aproximada del funcionamiento general y las variables necesarias para la resolución. Estas serán las variables no aleatorias que habremos de especificar y consecuentemente influirán los resultados obtenidos:

- El **número de acciones disponibles**. Este parámetro definirá la dimensionalidad de nuestro problema. En un entorno como *pong*, el agente solamente tendrá tres acciones disponibles: moverse arriba, abajo o quedarse quieto. La red neuronal tendrá un número de nodos de salida igual al número de acciones.
- El **algoritmo a utilizar**. La librería RLLib, de la que hablaremos más adelante, nos ofrece un ámplio catálogo de algoritmos predefinidos listos para ejecutar. Esta será la base sobre la que compararemos nuestros resultados. Dependiendo de éste factor también podrán ser necesarias otras variables como el **número de workers**, la **tasa de aprendizaje** o el **tamaño de bloque de aprendizaje**.
- El **factor de descuento** a aplicar. Se propone un valor constante de 0.9 para evitar así discrepancias entre distintas ejecuciones.

- El **número de capas ocultas de la red**, así como su tamaño.
- La **condición de parada del entrenamiento**, que podrá ser en función de factores como el tiempo transcurrido, la recompensa media total o el número de iteraciones realizadas. Dado que el aprendizaje en estos sistemas resulta gradual, el conocimiento del agente sobre su entorno aumentará conforme más iteraciones realice sobre el mismo. Tradicionalmente, se realizan entrenamientos de al menos un millón de iteraciones (10^6).
- La **capacidad de GPU a utilizar**.
- La **cantidad de memoria dinámica disponible**. En los problemas de exploración normalmente se requiere una cantidad razonablemente elevada de memoria RAM para almacenar las experiencias pasadas del agente. Para evitar problemas de memoria, se establecerá el límite máximo de memoria RAM a utilizar a un 40 % de la capacidad máxima (6GB en un PC de 16GB de RAM).

A continuación, introduciremos brevemente algunos de los juegos clásicos de Atari implementados en OpenAI Gym que tenemos como objetivo estudiar.

3.2.2. Entornos de Atari

Todos los entornos sobre los que experimentamos tienen como entrada de datos una pantalla RGB con una resolución de 210x160 píxeles. Como se ha explicado en secciones anteriores, este elemento habrá de ser tratado por una **red convolucional** que normalice los datos de entrada a introducir.

Cada acción seleccionada por la red será repetida por una duración de k = [2, 3, 4] estados (o *frames*) para reducir la carga de la toma de decisiones y simplificar el entrenamiento

La librería Gym nos ofrece distintas versiones de un mismo entorno, diferenciadas por factores condicionantes que pueden potencialmente afectar las decisiones del agente a largo plazo. Algunas de estas son la probabilidad de repetición de una acción independiente de la acción seleccionada o el uso del *frame skipping*, una técnica consistente en realizar la misma acción durante un número indeterminado de estados y que en ciertos casos puede resultar beneficiosa para reducir el tiempo de entrenamiento.

En nuestro caso, utilizaremos principalmente la versión sin *frame skipping*, pero consideramos también realizar pruebas en otros formatos para comprobar hasta qué punto factores como este pueden condicionar realmente el entrenamiento.

Pong

Se trata de uno de los primeros juegos arcade de la historia, y el primer juego jamás desarrollado por la empresa Atari. El juego consiste en una partida de tenis entre dos jugadores donde al menos uno de ellos es controlado por un humano, o, en nuestro caso, un agente independiente. Las mecánicas consisten básicamente en conseguir que la pelota atraviese el campo del rival, evitando al mismo tiempo que el rival haga lo mismo.

Los únicos controles disponibles por el agente en este entorno son con respecto al movimiento de la raqueta": Arriba, abajo o quedarse quieto.

Un posible problema de ejecutar este tipo de entornos es que, como se puede observar, es imposible saber por solo un estado del entorno la trayectoria que sigue la pelota. Es

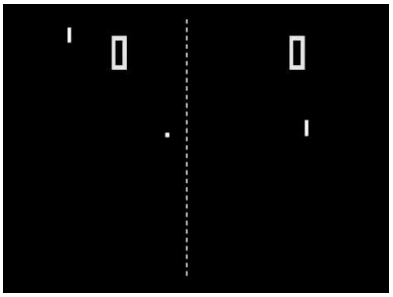


Figura 3.1: Entorno Pong.

posible que entornos de éste tipo no tengan un buen rendimiento en la librería Ray si esta no implementa una estructura de memoria que solucione este problema.

Space Invaders

Otro entorno muy popular de la Atari 2600 es el juego Space Invaders. En este, el jugador controla una torreta en la parte inferior de la pantalla que dispara de manera periódica capaz de moverse a izquierda y derecha. En el inicio de la partida, una serie de enemigos aparecen en la parte superior de la pantalla, y lentamente realizan un avance en dirección a la tierra. El objetivo a seguir es eliminar la mayor cantidad de enemigos posible antes de que lleguen al nivel de la torreta.

En la simulación de Gym, el agente recibe una recompensa cada vez que elimina un enemigo, así como una bonificación de tiempo cuando los ha eliminado a todos. Las acciones disponibles por el jugador son mover derecha, mover izquierda y quedarse quieto.

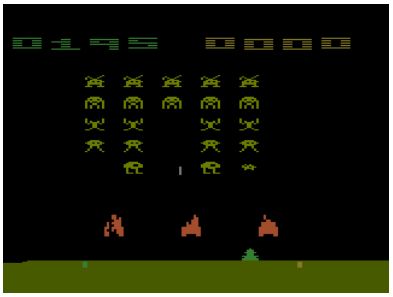


Figura 3.2: Ejemplo de estado del entorno Space Invaders.

Breakout

Este juego vino influenciado por el éxito inicial que tuvo Pong. El juego hace de nuevo uso de una raqueta en la parte inferior de la pantalla que el jugador puede desplazar de izquierda a derecha. A su vez, en la parte superior se encuentra una estructura de ladrillos que se destruyen cuando son golpeados por una bola que va desplazándose por el campo. La tarea del jugador es evitar que esta pelota se escape por la parte inferior y conseguir eliminar la mayor cantidad de ladrillos posible.

Boxing

El siguiente entorno muestra un ring de boxeo desde arriba. En él se encuentran dos boxeadores que se pueden golpear cuando están a corta distancia el uno del otro. La estrategia a seguir consiste en golpear al enemigo el mayor número de veces impidiendo

Figura 3.3: Entorno Breakout.

al mismo tiempo ser golpeado. Hay dos tipos de puñetazo: el largo, que suma un punto a nuestra puntuación, y el de corta distancia que suma dos puntos. A pesar de esto, solamente existe una acción de ataque que cuando es accionada se trata del entorno quien decide el tipo de puñetazo a lanzar. Adicionalmente, el agente es capaz de realizar movimientos en todas direcciones.

A pesar de la apariencia sencilla del entorno, la puntuación perfecta solo se consigue golpeando al rival 100 veces sin haber recibido ningún golpe a cambio. Será interesante observar si con nuestros experimentos alguno de los algoritmos es capaz de aproximarse a este nivel de conocimiento del entorno.

El juego termina o bien consiguiendo un *knockout* (100 golpes sobre el enemigo) o cuando transcurren dos minutos desde el inicio del asalto.

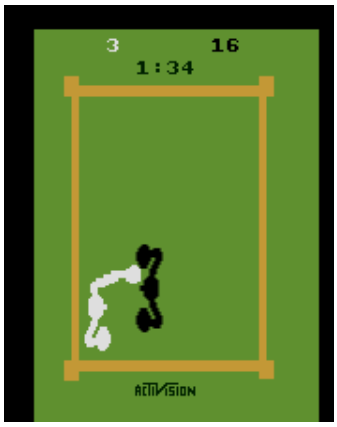


Figura 3.4: Entorno Boxing.

Atlantis

Atlantis fue un juego publicado en 1982 por la empresa Imagic[1] ambientado como un asedio a la ciudad perdida del mismo nombre. La ciudad cuenta con siete bases vulnerables a los ataques de los enemigos, tres de ellas equipadas con armamento para disparar a las naves enemigas, pero la trayectoria de los proyectiles disparados es inmutable.

El objetivo del juego es conseguir resistir el mayor tiempo posible el asedio de los enemigos que, conforme pasa el tiempo, aumentan en frecuencia y velocidad. Dado que no existe un final específico del juego más allá de cuando la base es destruida, diseñaremos la condición de parada basándonos en las puntuaciones más altas conseguidas por otros miembros de la comunidad realizando experimentaciones similares. [2]

[1] https://en.wikipedia.org/wiki/Imagic
[2] https://github.com/cshenton/atari-leaderboard

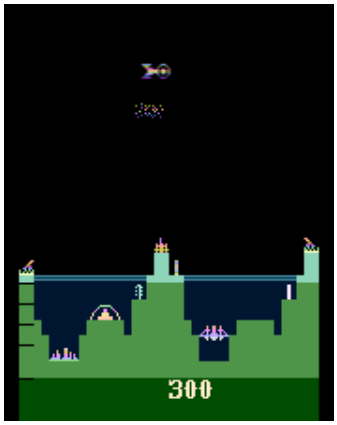


Figura 3.5: Captura de pantalla del juego Atlantis.

Seaquest

Este último entorno se trata posiblemente del más desafiante de nuestra selección. El juego es un *shooter* en el que el jugador controla un submarino con capacidad de disparar torpedos. Su objetivo es ir rescatando buceadores dispersados por el fondo del mar mientras elimina enemigos; el submarino cuenta con una capacidad de oxígeno limitada, por lo que es necesario que haga ascensiones a la superficie de manera regular. No obstante, cada vez que esto sucede, la frecuencia en la que aparecen enemigos aumenta, así que es necesario una buena gestión del oxígeno y de las emersiones que se realizan. Por último, si el submarino asciende a la superficie después de haber recogido seis buceadores, se recibe una recompensa adicional proporcional al oxígeno disponible.

Al igual que en Atlantis, nos basaremos en experimentaciones de terceros para diseñar las condiciones de parada.

Figura 3.6: Captura de pantalla del entorno Seaquest.

3.2.3. El entorno ViZDoom

Para la realización de la segunda parte de nuestro trabajo haremos uso de la librería de simulación ViZDoom. Este módulo desarrollado por estudiantes en la universidad tecnológica de Poznan, ofrece un modelo de entorno basado en el videojuego Doom, desde donde nos será posible realizar las experimentaciones mencionadas.

La librería nos permite acceder de manera casi automática a una versión de Doom personalizada orientada al estudio de técnicas de aprendizaje, desde donde se facilita el acceso a información tanto como de estados, recompensas u objetos presentes en el campo de visión del agente. Esta versión del juego cuenta además con una colección de escenarios personalizados con distintos objetivos a cumplir, ofreciendo así una variedad de problemas para solucionar por el agente de aprendizaje. El agente tendrá que aprender a ejecutar distintas estrategias para solucionar estos escenarios.

A continuación, haremos una rápida introducción de algunos escenarios que esta librería nos ofrece.

Basic

El escenario básico sobre el que realizar unas primeras pruebas y verificar el funcionamiento del aprendizaje. El agente aparece en el centro de un mapa rectangular con un enemigo en algún lugar aleatorio de la pared del mapa.

Recompensas:

- +101 por matar al enemigo
- -5 por no llegar al final del episodio
- **Living reward** -1

Movimientos disponibles: MOVER IZQUIERDA, MOVER DERECHA, DISPARAR

Deadly corridor

La finalidad de este escenario es enseñar al agente a navegar hacia su objetivo, en este caso consistente en un chaleco de armadura existente al otro lado de un corredor con enemigos en ambos lados. La recompensa en este caso es proporcional a la distancia entre el agente y el chaleco.

Recompensas:

- Penalización por muerte -100
- dX proporcional a la distancia al chaleco

Movimientos disponibles: MOVER/GIRAR IZQUIERDA, MOVER/GIRAR DERECHA, DISPARAR

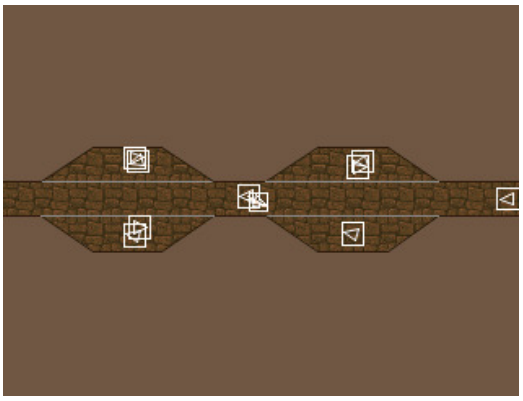


Figura 3.7: Mapa en escenario Deadly Corridor.

Defend the center

Este escenario tiene como objetivo enseñar al agente a eliminar enemigos evitando al mismo tiempo ser eliminado.

El mapa consiste en un rectángulo con nuestro agente apareciendo junto al centro del muro principal. A su vez, aparecen tres enemigos que atacan cuerpo a cuerpo y otros tres con armas a distancia. Al principio los enemigos mueren en un solo disparo, pero se vuelven más fuertes a medida que pasan las rondas. El episodio termina cuando el agente muere.

Recompensas:

- +1 por matar a un enemigo
- Penalización por muerte -1

Movimientos disponibles: GIRAR IZQUIERDA, GIRAR DERECHA, DISPARAR

Health gathering

Este escenario tiene como finalidad enseñar al agente a sobrevivir en un entorno hostil que inflige una cantidad de daño leve durante un periodo constante de tiempo. Para ello, aparecen sobre el mapa un conjunto de objetos que recuperan la vida al agente. El objetivo de este será aguantar todo el tiempo posible sin morir.

Living reward: +1

Movimientos disponibles: GIRAR IZQUIERDA, GIRAR DERECHA, AVANZAR

My way home

Este agente consiste en un camino laberíntico hacia la salida. El agente tendrá como objetivo navegar el mapa buscando la salida.

Recompensas:

- +1 por alcanzar la salida
- **Living reward:** -0.0001

3.3 Plan de trabajo

En esta sección, hablaremos del plan de trabajo que hemos seguido.

Debido a la naturaleza de nuestro trabajo, estructuramos la planificación en diversas partes bien diferenciadas. Dentro de cada etapa, establecimos distintos objetivos necesarios de superar antes de poder pasar a la siguiente etapa. Además, definimos una serie de plazos a cumplir con referencia a cada una de estas fases.

Fase I: Preparación

Esta primera etapa engloba distintas tareas a realizar de manera de introducción a la materia que nos ocupa. Como una primera aproximación, estimábamos que requeriría unas 70 horas de trabajo a realizar durante las primeras siete semanas del proyecto, tiempo que sería invertido en la realización del curso online **Artificial Intelligence A-Z: How to build an AI**[3], así como en la investigación del funcionamiento de librerías adicionales (OpenAI Gym, RLLib, VizDoom, etc.). Además, utilizaríamos este tiempo para

[3]https://www.udemy.com/artificial-intelligence-az/

informarnos sobre los avances realizados en el campo de la inteligencia artificial en los últimos años, qué nuevos algoritmos podrían haber surgido que resultarían útiles para poder aplicar a nuestro trabajo y qué conceptos adicionales podríamos tratar de aplicar o sería conveniente introducir para una comprensión global de nuestro problema más adecuada.

El flujo de trabajo estuvo bien definido, comenzando con el curso de introducción a la Inteligencia Artificial, el cual se constituye de 16 horas de lecciones en formato de video, en las cuales se introducen conceptos básicos tan variados como la ecuación de Bellman, pasando por una visión general de las redes neuronales y terminando con una explicación escueta de cómo aplicar un algoritmo A3C al entorno Breakout. Adicionalmente a estas 16 horas, también se requería la modificación de ciertas plantillas de código para superar algunas secciones del curso. Esto, añadido a la instalación del entorno desde cero, terminó resultando en aproximadamente unas 30 horas de trabajo repartidas durante las cuatro primeras semanas.

Una vez terminado el curso, iniciamos una serie de experimentaciones sencillas por nuestra cuenta en los entornos de OpenAI. Además, consideramos el uso de una serie de librerías externas que más adelante terminarían siendo descartadas, bien por su alta complejidad, obsolescencia o simplemente por no ajustarse exactamente a nuestros objetivos a largo plazo. Por poner algunos ejemplos, algunas de estas librerías fueron: **ml-agents**[4], **TRFL**[5] o **OpenAI Universe**[6].

Fase II: Primeras experimentaciones

Durante esta fase se nos proporcionó una serie de credenciales para el acceso remoto a un servidor sobre el que podríamos lanzar nuestras experimentaciones. El montaje del entorno en el nuevo servidor, así como las primeras pruebas y tutoriales para conseguir soltura en el entorno Jupyter llevaron alrededor de 5 horas.

Una vez con acceso a este nuevo servidor, comenzamos a hacer pruebas sobre la librería Ray, haciendo uso de una primera versión de nuestro script *custom_learning*. Ésta nos permitió realizar experimentaciones con distintos algoritmos en una gran variedad de entornos de Atari. No obstante, surgieron una gran cantidad de problemas de memoria, dado que se trataba de un servidor compartido en el que distintos usuarios lanzaban sus experimentaciones al mismo tiempo. Debido, pues, a la gran carga de memoria que se requiere para soportar las experiencias pasadas en cada ejecución de algunos de los algoritmos, resultó ser un gran problema que finalmente solventamos limitando la cantidad de memoria a utilizar por nuestro programa a solamente un 35-40 % de la memoria RAM total. De cualquier manera, hasta que pudimos aplicar esta solución nos encontramos con un gran número de experimentaciones que fallaban a mitad de ejecución y que básicamente tuvimos que desechar a pesar del considerable tiempo de ejecución invertido en muchas de ellas.

Paralelamente a estas experimentaciones, seguimos con nuestra formación particular en la librería Tensorflow, así como una recogida global de información a partir de una colección de más de quince *papers* académicos con enfoques muy distintos, a partir de los cuales nos inspiraríamos para la redacción de nuestro capítulo estado del arte.

Una vez solventados los problemas de memoria de RLLib, fuimos capaces de realizar una serie de experimentaciones relativamente largas sobre distintos entornos y recopilar

[4]https://github.com/Unity-Technologies/ml-agents
[5]https://github.com/deepmind/trfl/blob/master/docs/index.md
[6]https://openai.com/blog/universe/

unos primeros resultados sobre los que poder realizar valoraciones iniciales. Estimamos que el tiempo total de ejecución de nuestros experimentos puede superar las 100 horas.

Fase III: Verificación de resultados y pruebas con ViZDoom

Cuando ya hayamos hecho algunas experimentaciones y con una mejor comprensión de la librería y su funcionamiento interno, podremos pasar a la siguiente fase de nuestro trabajo. Esta fase consistirá de dos partes. En primer lugar, podremos ya verificar el comportamiento de los algoritmos a partir de las experimentaciones que ya habremos llevado a cabo y así poder llegar a ciertas conclusiones preliminares. En segundo lugar, trataremos de diseñar una solución capaz de realizar un modelo de entrenamiento por refuerzo similar a los que hemos utilizado hasta éste momento, pero en éste caso utilizando la librería Vizdoom como entorno para el aprendizaje. Para ello, será necesario aplicar los conocimientos de las librerías con las que experimentamos en la anterior fase, además de utilizar Tensorflow para el diseño de una red neuronal propia que implemente el algoritmo DRQN, al ser teóricamente útil en éste tipo de entornos.

Utilizaremos un modelo de desarrollo en cascada para el diseño de nuestra solución, centrándonos al inicio en implementar las funcionalidades básicas para que el entorno se pueda ejecutar, y a medida que avancemos le iremos añadiendo funcionalidades adicionales como la red neuronal DRQN, entrada de parámetros o mejoras generales en la estructura del algoritmo o el uso de la memoria. Calculamos que el diseño completo de nuestra solución llevará en torno a 25 horas de trabajo efectivo desde el inicio en la fase de especificación de requisitos.

Una vez tengamos nuestro modelo desarrollado y hayamos verificado su funcionamiento, realizaremos una serie de experimentaciones haciendo siempre uso de la misma red neuronal que hemos creado, pero sometiendo a nuestro agente a una selección de los escenarios propuestos por la librería Vizdoom. Calculamos que esto nos llevará aproximadamente otras 15 horas de trabajo efectivo, siendo realmente el tiempo de ejecución de nuestras experimentaciones por encima de las 50 horas.

En esta fase realizaremos también otra serie de experimentaciones de última hora sobre los entornos Atari en función a los problemas o lagunas de conocimiento que pudiéramos haber observado a partir de la primera tanda de resultados.

Fase IV: Análisis de resultados

Una vez las experimentaciones hayan sido finalizadas, podremos constrastar los resultados y trajar algunas conclusiones en función a lo que podamos observar. Haciendo uso de la herramienta Tensorboard, recogeremos la información pertinente de los resultados obtenidos y extraeremos las gráficas necesarias para plasmar nuestros resultados en este trabajo.

3.4 Hardware disponible

Debido a la naturaleza de nuestros objetivos, nos será necesaria una alta capacidad de procesamiento sobre la cual lanzar nuestras experimentaciones. Es por ello que el Grupo de Inteligencia artificial (GTI-IA) de la ETSINF nos proporcionó acceso remoto a una serie de servidores propios con las siguientes características:

Servidor GPU GTI-IA (1):

- **Unidad de procesamiento (CPU):** Intel(R) Core(TM) i7-7700 CPU @ 3.60GHz
- **Num. de núcleos:** 4
- **Tarjeta gráfica:** Nvidia Titan Xp
- **Memoria RAM:** 16 GB

Servidor GPU GTI-IA (2):

- **Unidad de procesamiento (CPU):** Intel(R) Xeon(R) W-2123 CPU @ 3.60GHz
- **Num. de núcleos:** 8
- **Tarjeta gráfica:** 2X Nvidia Titan V [7]
- **Memoria RAM:** 64 GB

PC Personal:

- **Unidad de procesamiento (CPU):** AMD Ryzen 5 2600 @ 3.90 GHz
- **Num. de núcleos:** 6
- **Tarjeta gráfica:** Nvidia Geforce GTX 970
- **Memoria RAM:** 32 GB

Todas las máquinas utilizadas llevarán instalado el mismo entorno de Anaconda listo para la ejecución, con las versiones idénticas de cada submódulo para evitar problemas de ejecución.

[7] https://www.nvidia.com/es-es/titan/titan-v/?nvid=nv-int-geo-es-titan-v

CAPÍTULO 4
Diseño de la solución

4.1 Introducción

En este capítulo se describe el enfoque propuesto para diseñar una implementación de aprendizaje por refuerzo conforme a lo introducido en el capítulo 3. La herramienta desarrollada a lo largo de este trabajo tiene como objetivo observar factores que pueden condicionar el rendimiento del aprendizaje, realizando un estudio entre diferentes entornos y algoritmos de aprendizaje.

Mediante el uso de Gym como base para el uso de los entornos, tanto aquellos nativos a la librería como algunos introducidos de manera manual, así como Ray para el lanzamiento de los procesos de aprendizaje, se ha conseguido recoger una colección de resultados suficiente para desarrollar conclusiones sobre la relación entre el tipo de entornos utilizados y el rendimiento de ciertos algoritmos de aprendizaje. Las herramientas mencionadas, habiendo estado desarrolladas en Python, nos permiten su combinación con otras librerías útiles como Tensorboard para la interpretación gráfica de los resultados o Numpy para el manejo de datos de manera cómoda y sencilla. Además, Ray proporciona una estructura de agentes modelados como hilos sobre los que nos es muy cómodo y relativamente sencillo realizar experimentos una vez las configuraciones de los agentes estén correctamente definidas.

La elección de la tecnología ha sido influida principalmente por las herramientas presentes en el lenguaje seleccionado para el desarrollo de soluciones de aprendizaje automático y computación científica en general. Ésto nos permite aprovechar funcionalidades ya desarrolladas por la comunidad, permitiéndonos así centrarnos en la aplicación del resto de funcionalidades y su ajuste a nuestro problema particular.

A continuación desarrollaremos un breve repaso sobre las tecnologías de las que hemos hecho uso para el desarrollo de nuestro trabajo, siendo algunas de ellas piezas fundamentales sin las cuales la correcta consecución de nuestros objetivos hubiera conllevado una carga de trabajo exponencialmente superior. Estas herramientas nos han dado la base necesaria para poder aplicar los conocimientos recogidos durante nuestra investigación previa, aportando características vitales y funciones indispensables.

4.2 Tecnologías utilizadas

El desarrollo de nuestro simulador ha implicado el uso de una variedad de tecnologías y herramientas que han condicionado en gran medida los entornos de programación a nuestro alcance. A continuación, presentaremos una breve descripción de esas herramientas y las ventajas que nos han brindado.

4.2.1. Python

Python 3.6 es un lenguaje interpretado de alto nivel altamente utilizado para este tipo de soluciones. Desarrollado en 1991 por Guido Van Rossum, que tuvo como objetivo crear un lenguaje donde fuera posible escribir código claro y conciso. Este lenguaje toma elementos del paradigma de programación orientada a objetos y de la programación funcional, tomando entre otras, las características funciones *lambda*, *filter* y *map*.

Se trata de un lenguaje interpretado que ofrece beneficios como el tipado dinámico y la independencia de la plataforma. Además, Python cuenta con un intérprete que permite la programación de forma interactiva facilitando la realización de pruebas y depuración de código.

Python provee un extenso catálogo de librerías de programación científica, es fácil de usar y su escritura es muy similar al lenguaje natural, y su **modularidad**, nos brindará la posibilidad de añadir fácilmente módulos no nativos al lenguaje que añadirán funcionalidades específicas muy útiles para nuestra implementación. Para esta finalidad, haremos uso del software *Anaconda*, una distribución personalizada de Python que se utiliza en infinidad de proyectos de ciencia de datos debido a la gran cantidad de librerías que incluye en su versión por defecto. Estas librerías además se pueden gestionar mediante *Conda*, un sistema de administración de entornos integrado que nos permitirá crear entornos con distintas librerías y evitar así posibles conflictos.

4.2.2. Módulos adicionales

A continuación, se procederá a enumerar y describir brevemente la funcionalidad de algunas librerías de Python, tanto nativas como comunitarias, así como algunos softwares externos que nos ayudarán a desarrollar nuestros scripts:

OpenAI Gym [1]. Ésta librería desarrollada sin ánimo de lucro provee una serie de entornos de entrenamiento predefinidos sobre los cuales se pueden experimentar distintas técnicas de aprendizaje por refuerzo. La librería se puede dividir en diferentes submódulos, cada uno con una serie de entornos de ejecución con distintas características. Nuestras experimentaciones se centrarán en la categoría de videojuegos de Atari que ésta plataforma nos ofrece.

ViZDoom [2]. Esta librería nos ofrece una implementación del juego DOOM de 1993 acomodada para el estudio de técnicas de aprendizaje por refuerzo. Esta librería ha sido introducida más detenidamente en la sección 3.2.3

[1] https://gym.openai.com/
[2] http://vizdoom.cs.put.edu.pl/

Ray [3], nos proveerá un sencillo entorno de ejecución para nuestras experimentaciones, enlazando Tensorflow con el Gym de OpenAI y permitiéndonos una simple programación de las funcionalidades de entrenamiento con las que cuenta Tensorflow sobre los entornos de Gym. El submódulo **RLLib** nos presentará una interfaz de desarrollo con una API extensa y de fácil uso que implementa la conexión con Gym. Esta librería permite una ejecución distribuida de las tareas de ejecución entre distintos servidores, cosa que podría ser útil para experimentaciones con un alto nivel de iteraciones y/o tiempo total de ejecución. Por último, Ray nos permite, si así lo deseamos, una fácil interfaz desde donde introducir entornos de ejecución personalizados. Ray cuenta con un submódulo llamado **Tune**, que provee una sencilla interfaz para modificar hiperparámetros de aprendizaje. Esta librería será capaz de hacer una búsqueda por descenso de gradiente de los hiperparámetros óptimos para cada problema particular.

Tensorflow [4], es una librería de Machine Learning de código abierto que permite el diseño de las estructuras de datos necesarias para la creación de redes neuronales. Esta librería nos proveerá de un extenso catálogo de distintas configuraciones por defecto para crear redes neuronales a medida y fácilmente modificables. Esta libreria cuenta con un submódulo de Keras nativo para la creación y el entrenamiento de los modelos que creemos.

Tensorboard [5] Herramienta integrada con Tensorflow, proporciona una visualización sencilla de resultados de experiementaciones mediante una extensa colección de gráficas que facilitan enormemente la comprensión de resultados.

Keras [6], API de alto nivel para construir y entrenar modelos de *Deep Learning*. En la actualidad se trata de una de las librerías de este tipo más utilizadas, principalmente para el prototipado, investigación y producción de este tipo de modelos.

Numpy [7] es la contracción de *Numerical Python*, módulo fundamental nativo de Python para el desarrollo de scripts de ámbito científico. Puede ser usado como un contenedor de datos multidimensional sobre el que aplicar funciones matemáticas complejas.

Git [8] esta herramienta es un software de control de versiones de código abierto. El uso de ésta viene justificado debido a que permite llevar un control de las modificaciones de código, la posibilidad de revertir a versiones anteriores, detección de errores o realizar un seguimiento global del trabajo realizado.

4.3 Arquitectura de solución Atari

En esta sección detallaremos la estructura interna de los scripts de entrenamiento que hemos desarrollado. Para ello, definiremos uno a uno los módulos de la herramienta.

Hemos dividido el funcionamiento de nuestro esquema de simulación en tres módulos funcionales distintos. En primer lugar, el módulo de inicialización transmite la información de entrada de los parámetros y un bloque de configuración para inicializar los procesos necesarios y poder dar comienzo la simulación. En segundo lugar, el módulo de entrenamiento hace uso de los agentes inicializados para realizar una simulación

[3] https://ray.readthedocs.io
[4] https://www.tensorflow.org/
[5] https://www.tensorflow.org/guide/summaries_and_tensorboard
[6] https://keras.io/
[7] https://www.numpy.org/
[8] https://git-scm.com/

con unas características dependientes de los datos de entrada. Por último, hacemos la distinción de utilizar un módulo interno de procesamiento de datos nativo a la librería Tensorflow, que a su vez es utilizada por el submódulo de Ray **RLLib**. En esta última capa, se procesan los resultados de cada iteración del entrenamiento y se actualizan los pesos. El modelo presentado puede variar ligeramente dependiendo del algoritmo de aprendizaje que se utilice.

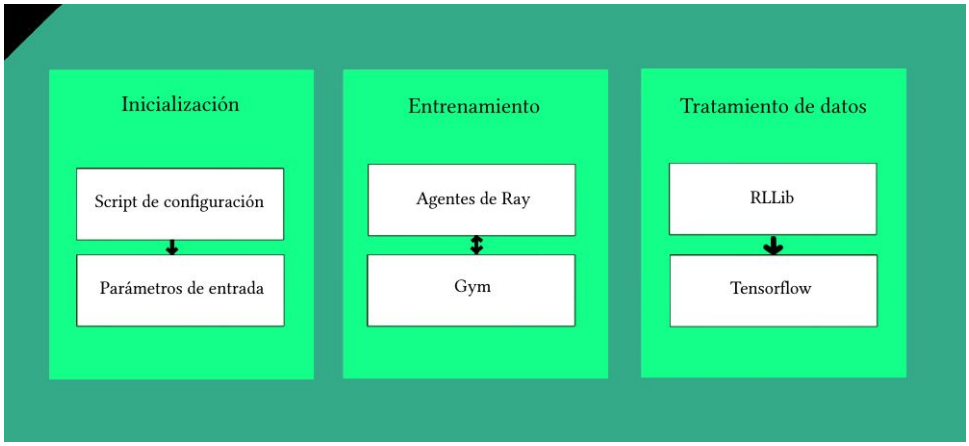


Figura 4.1: Modelo de arquitectura de la solución

El módulo config.py contiene la información de ejecución de cada algoritmo de aprendizaje. A partir de éste podremos experimentar distintas configuraciones de hiperparámetros para cada algoritmo. Simultáneamente, recibiremos por línea de comandos una serie de instrucciones con información sobre el tipo de experimentación a llevar a cabo.

A partir de estos datos, nuestro *script* hará una llamada a la librería Ray comenzar el entrenamiento. Se inicializarán los procesos (agentes) necesarios y se realizará en segundo plano el entrenamiento sobre el entorno introducido. Una vez terminado el entrenamiento, nos será posible acceder a algunos clips de estas iteraciones para observar así el progreso de nuestro programa.

Por último, el módulo de tratamiento de datos realizará una actualización de los pesos de la red neuronal con el fin de mejorar el rendimiento a largo plazo de los agentes. Esta operación la realizará basándose en la información recogida por las experiencias pasadas en intervalos predefinidos que podrán ser modificados en el fichero de configuración. Para más información, se recomienda revisar la sección 2.3.1.

A partir del modelo de arquitectura anterior, en la figura 4.2 se puede observar un diagrama de flujo del simulador con un funcionamiento más específico a partir de lo introducido anteriormente. En esta figura, la capa de entrada recibe los parámetros de entrada de la simulación a partir del fichero de configuración, además de los siguientes parámetros por línea de comandos:

1. El **algoritmo a ejecutar.**
2. El **entorno ATARI de ejecución.**
3. El **número de *workers*** a inicializar por la librería.
4. La **cantidad de memoria disponible para cada agente**
5. La **cantidad de GPU a utilizar**
6. El **directorio de destino** de los resultados recogidos.

7. Las **condiciones de parada** sobre las que termina la experimentación.

Con toda esta información, el programa inicia la simulación haciendo uso de las librerías mencionadas. A medida que se van acumulando experiencias, el programa actualiza los valores internos de la red dependiendo de la configuración introducida, a partir del módulo que hemos llamado gestión de workers. Una vez finalizado el proceso de aprendizaje, se generan las gráficas de salida a partir de la colección de experiencias pasadas y el log de la ejecución, además de una colección de repeticiones del rendimiento del agente sobre el entorno.

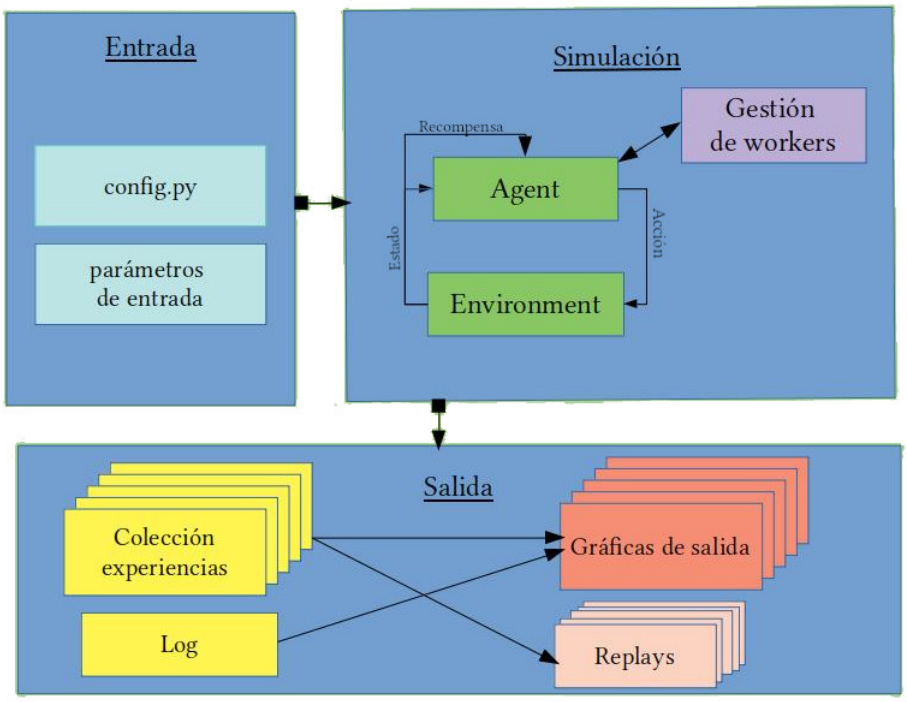


Figura 4.2: Modelo de diagrama de flujo utilizado para la solución de Atari.

A continuación, en la figura 4.3 se muestra el diagrama de flujo generalmente seguido por los procesos de Ray que se crean en una ejecución. El proceso ***trainer*** realiza la tarea de aprendizaje a partir de las muestras de entrenamiento que periódicamente le proporcionan los distintos ***workers***. Cada vez que esto sucede, el *trainer* es el responsable de actualizar los pesos de las redes de aprendizaje utilizadas por los *workers*.

La manera en que estos procesos se comunican depende en gran medida de el tipo de algoritmo que utilicemos, pudiendo ser así una comunicación tanto síncrona como asíncrona o con una estructura interna específica del proceso *trainer*.

4.3.1. Inicialización de los experimentos y condiciones de parada

Nuestra herramienta recibe una serie de parámetros de entrada que condicionarán la ejecución de distintas formas. Estos parámetros se introducirán más adelante en la sección 4.3.2.

Lo primero que hará nuestro programa será inicializar una sesión Ray con el entorno que se especifique, inicializando además los hilos necesarios y haciendo la reserva de memoria dinámica (RAM) que sea necesaria. Estos hilos o agentes serán los encargados de realizar las acciones en el entorno seleccionado para después pasarle la información de las experiencias al proceso *trainer*. A partir de ese momento, la libería será capaz de

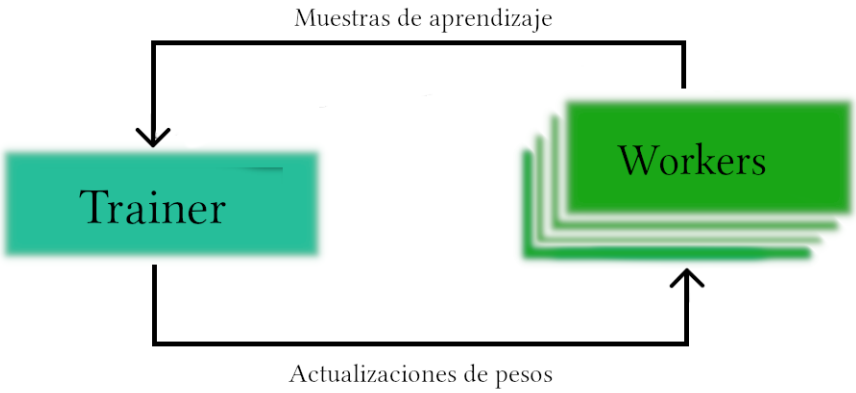


Figura 4.3: Modelo de arquitectura de la solución

ejecutar una experimentación y realizar la tarea de aprendizaje, recogiendo experiencias y aprendiendo de ellas.

Las condiciones de parada de la simulación dependerán de los parámetros de entrada que el usuario especifique. Adicionalmente, se crea en nuestro *script* unas condiciones de parada por defecto en caso de que estas no vinieran especificadas por línea de comandos. Estas condiciones pueden depender de elementos tan diversos como, entre otros, el tiempo total de ejecución, el número de iteraciones totales realizadas o la recompensa media conseguida por el agente. La ejecución también se abortará en caso de fallo de alguno de los hilos activos.

4.3.2. Parámetros de entrada

Previo a la ejecución de nuestros *scripts*, el usuario puede especificar algunas opciones por línea de comandos. Estas opciones influyen en el funcionamiento interno de nuestra herramienta, y es necesario tenerlos en cuenta en el momento de constrastar los resultados obtenidos. A continuación, enumeraremos los parámetros a introducir para el entorno de Atari:

- **Algorithm.** El algoritmo de RLLib a ejecutar. Para nuestras experimentaciones, nos centramos en A3C, A2C, DQN, PPO e IMPALA.
- **Environment.** El entorno sobre el que vamos a lanzar las experimentaciones. Se introduce el nombre de entorno nativo a la librería Gym. Ejemplo: *SpaceInvaders-v0*
- **Stop.** La condición de parada introducida. Puede recoger distintos argumentos:
  - time_total_s: Número de segundos totales de ejecución.
  - training_iteration: Número total de iteraciones realizadas entre todos los agentes implicados.
  - episode_reward_mean: Recompensa media conseguida entre todas o un número fijado de experiencias recientes.
- **Memory.** Cantidad de memoria RAM reservada para utilizar por cada proceso *worker*. Se introduce esta variable dado los problemas que surgieron al intentar ejecutar la librería en un servidor con 16 GB de RAM.
- **Gpu_usage.** Define si se hace uso de la GPU para la tarea de entrenamiento. Esto puede influir en gran medida en el tiempo necesario para realizar las iteraciones,

dependiendo de la capacidad de la GPU. Además, es posible introducir un valor decimal para utilizar así solamente una fracción de la capacidad total de la tarjeta.

- **Num_workers.** Número de procesos trabajadores a inicializar. Para una ejecución exitosa, habrá que tener en cuenta que el valor num_workers x memory_usage no sea mayor a la cantidad de memoria RAM total, ya que podría provocar errores en la ejecución.
- **Folder.** Fichero donde guardar los resultados recogidos por la librería. Este fichero, de no existir, sería creado dentro de una carpeta llamada *ray_results*.
- **All.** Parámetro introducido en caso de que se deseara realizar la ejecución de todos los algoritmos disponibles sobre el mismo entorno. Esto sobreescribiría lo introducido por el parámetro *algorithm* y realizaría de manera secuencial las experimentaciones de todos los algoritmos introducidos en el *script*.

4.4 Diseño de la solución para Doom

Como se ha comentado antes en el apartado 3.3, una vez realizado un primer conjunto de experimentaciones en Atari, pasaremos a diseñar una solución que haga uso de la librería ViZDoom. Para ello, nos será necesario utilizar Tensorflow para la creación de una red neuronal personalizada que implemente la estructura propia de una red profunda recurrente. Adicionalmente, es necesario crear las estructuras de comunicación necesarias entre el entorno Doom y nuestras estructuras de aprendizaje.

Para el acceso a la información del entorno, haremos uso de una librería comunitaria que ofrece una implementación del entorno a Gym, haciéndolo así más accesible para Tensorflow sin eso afectar a las posibilidades de modificar el funcionamiento interno del entorno.

La implementación de la red la hemos realizado haciendo un uso directo de Tensorflow, a diferencia de nuestra implementación de Atari donde hacíamos uso de la interfaz que ofrece Ray para acceder a sus configuraciones de Tensorflow ya creadas por defecto. Esto ha requerido una comprensión mucho más extensiva de la librería y generado una serie de nuevos problemas a solventar durante el desarrollo de nuestro código.

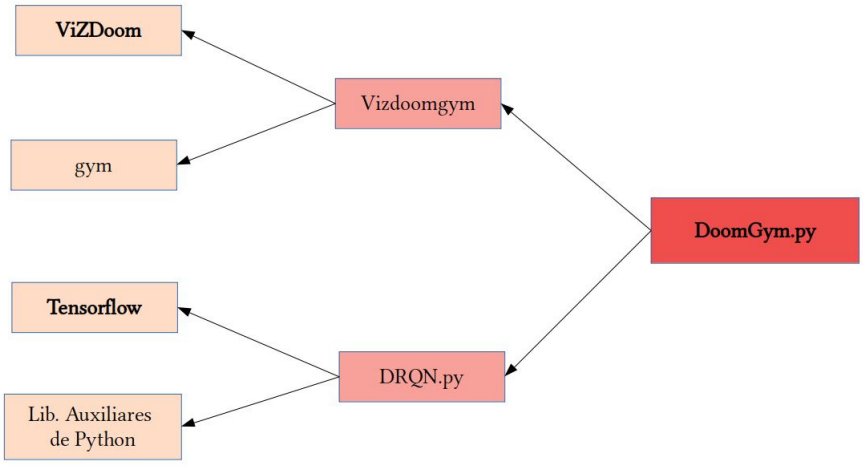


Figura 4.4: Diagrama de dependencias de la solución Doom.

4.4.1. Algoritmo a implementar: DRQN - Deep Recurrent Q-Network

A continuación introduciremos una variante del algoritmo DQN que se considera interesante para ejecutar sobre el entorno DOOM. El algoritmo DRQN [16] se trata de una variación del clásico DQN (2.3.1) que cuenta con la particularidad de introducir una **red neuronal recurrente (RNN) - sección 2.2.3** - que proporciona una mecánica de comunicación entre estados. Esta característica resulta práctica en entornos caracterizados por un elevado grado de información latente, ya que las redes recurrentes ofrecen una manera de rellenar las lagunas de conocimiento que el agente sufre especialmente durante las fases iniciales del entrenamiento.

En entornos caracterizados por un elevado grado de información latente, el uso de redes recurrentes ofrece una manera de rellenar las lagunas de conocimiento que el agente sufre durante las fases iniciales del entrenamiento. Es por ello que en estos casos se considera que en lugar de recibir información en base de estados, esta misma información viene dada por **observaciones parciales del entorno** del estado que se visita. El objetivo, por tanto, en estos entornos consistirá en estimarte el valor Q de la observación parcial $Q(o_t, a_t)$ en lugar del valor del estado. Esto pone al agente que realiza la tarea de aprendizaje en una posición de desventaja con respecto a la información que conoce del entorno. [15]

Por estas razones, consideramos que este algoritmo se trata de una muy buena opción para realizar pruebas sobre el videojuego Doom Classic. En las siguientes secciones entraremos más en detalle en el proceso de diseño de éste algoritmo y trataremos de verificar su supuesta conveniencia sobre entornos parcialmente observables.

4.4.2. Optimización

Como medidas a tener en cuenta para optimizar el rendimiento de nuestro algoritmo, introduciremos una estrategia de implementación propia de distintas técnicas ya mencionadas en secciones anteriores:

- En primer lugar, antes de comenzar el entrenamiento ejecutaremos una serie de episodios sobre el entorno con acciones aleatorias. De estas acciones se almacenarán ciertas experiencias que podrán ser usadas más adelante como muestras de entrenamiento

- También introduciremos una estructura de *frame skipping* para que el agente realice de manera consecutiva la acción seleccionada y así reducir el posible tiempo que requiere la decisión de la acción para cada *frame*.

- Haremos también uso de una estructura de entrenamiento por lotes de experiencias. Es decir, en el momento de actualizar la red introduciremos haciendo uso del método *sample(n)* de nuestra estructura de memoria (sección 5.3.1) un número n de experiencias que ajusten los valores estimados por nuestras redes. De esta manera se pretende reducir el tiempo de entrenamiento requerido por el algoritmo para alcanzar la resolución.

- Por último, será una prioridad diseñar una estructura de *experience replay* eficiente que no requiera un gran tiempo de acceso a las experiencias. Utilizaremos **Numpy** dado que se trata de una librería específicamente diseñada para manejar este tipo de computaciones.

4.4.3. Arquitectura de algoritmo DRQN

Como hemos comentado anteriormente, otro de nuestros objetivos sería la introducción de un nuevo algoritmo de aprendizaje para su uso en este entorno. El algoritmo seleccionado es **DRQN** que en teoría produce buenos resultados en entornos donde solamente se ofrece una observación parcial del entorno.

Hemos separado nuestra solución en dos clases distintas. Por un lado diseñamos un *script* que implementa la red neuronal y la estructura de memoria haciendo uso de Tensorflow y librerías auxiliares nativas de Python como Numpy. Se divide en dos clases:

- **class ExperienceReplay(capacity)**: Implementa las estructuras de datos que en tiempo de ejecución manejan las experiencias pasadas y mantienen un catálogo que la red puede consultar durante el entrenamiento. El método **appendToBuffer()** añadirá una experiencia al conjunto y el método **sample(n)** devuelve un conjunto de experiencias de tamaño n >= 1.
- **class DRQN()**: Implementa la red neuronal que más tarde se utiliza para el entrenamiento. Introducida en la sección 5.3.1

Para la clase DRQN hemos diseñado la arquitectura de una red neuronal convolucional con tres capas convolucionales. De esta manera, conseguimos reducir la dimensionalidad de la imágen inicial [256, 160, 3] a dimensiones mucho más manejables. En la siguiente figura introducimos la estructura general que tendrá nuestra red.

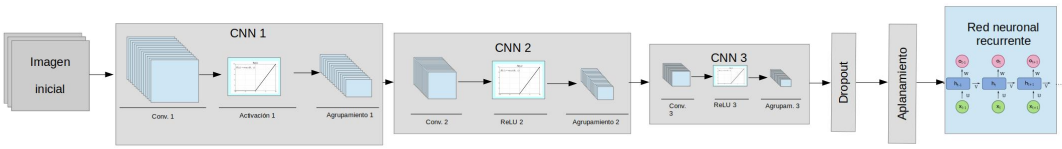


Figura 4.5: Arquitectura de la red convolucional.

Adicionalmente a las estructuras de convolución para el tratamiento de las imágenes, tendremos que crear la estructura interna de la red neuronal recurrente. Como se observa en la anterior figura, esta red tomará como entrada la imágen del juego una vez haya sido tratada por las capas convolucionales. Esta red contará con **dos capas ocultas** completamente conectadas. El número de *outputs* como ya se ha discutido vendrá dado por el número de acciones disponibles en el entorno.

4.4.4. Diseño de entrenamiento e integración con librerías

Por otra parte, tenemos el código de DoomGym_vX.py que hace uso de las estructuras que acabamos de describir. Este script recibe como parámetros el escenario a lanzar y realiza la inicialización de las variables (hiperparámetros) necesarias para el entrenamiento. Este script lanza una sesión de Tensorflow e inicializa dos redes DRQN, una para la selección de acciones y otra como objetivo, para así poder comparar el rendimiento de la red de selección con su comportamiento deseado.

La interfaz Vizdoomgym permite un manejo sencillo los diferentes escenarios disponibles en Vizdoom disponiendo de ellos haciendo una simple llamada a *gym.make(*nombre del escenario*-v0)*. Una vez el entorno se encuentre definido, es sencillo navegar por los

distintos estados. Por una parte, la llamada al método *step(a)* devolverá una tupla con información del estado resultante al ejecutar cierta acción *a*, la recompensa obtenida y si se trata o no de un estado final. Este método será nuestra principal vía de comunicación con el entorno junto a *env.reset()* que devolverá al agente al punto de partida del escenario.

A continuación introduciremos una pequeña traza del ciclo de entrenamiento para ilustrar su funcionamiento:

1. Se define el entorno, haciendo uso del método **gym.make()**.
2. Opcionalmente, realizaremos modificaciones sobre el escenario con los métodos disponibles en el objeto *gym.game*, que es con el que define el entorno ViZDoom.
3. Llamamos al método **env.reset()**, el cual nos devuelve el estado inicial del escenario. Esta llamada se hará en cada iteración del entrenamiento y guardaremos el estado devuelto en la variable *state*.
4. Inicialmente se realizarán una serie de episodios tomando acciones aleatorias y almacenándolas en memoria.
5. Mientras no se llegue a un estado final o *timeout*, y si ya se ha completado el proceso inicial de recolección de muestras aleatorias:
   a) Seleccionamos una acción *a* haciendo uso de los métodos de predicción de la red DRQN.
   b) Realizamos la acción seleccionada haciendo uso del método **env.step(action)**. Al mismo tiempo, preservamos las variables que nos devuelve el método con respecto al nuevo estado, la recompensa y la compleción de la ejecución.
   c) **Cada *n* estados (50)** guardamos la transición (estado, recompensas) en nuestra memoria.
   d) **Cada *i* pasos (5),** recibiremos una muestra de las experiencias pasadas y la utilizaremos para entrenar a la red.
   e) **Si la ejecución del entorno termina,** hacemos la suma de las recompensas acumuladas y la almacenamos en la memoria. Se reinicia el entorno y se sale del bucle de ejecución. Si no, volvemos al paso *a*.
6. Agregamos las recompensas de la última ejecución a memoria.
7. Cada 100 ejecuciones, guardaremos el modelo de la red neuronal en un fichero auxiliar.

Una vez terminado el entrenamiento, el *script* también realiza una fase de testeo donde prueba la red resultante. El modelo de red quedará guardado para poder acceder a él y realizar así la recolección de resultados o continuar el entrenamiento dado el caso.

CAPÍTULO 5
Implementación

5.1 Introducción

En este capítulo se presentará el diseño de nuestras soluciones usando el lenguaje Python 3.6. Para ello, introduciremos los *scripts* realizados que harán uso de los distintos módulos introducidos en secciones anteriores, y con los que pretendemos cumplir nuestros objetivos anteriormente explicados.

Las soluciones creadas se dividen en un módulo de entrenamiento para los entornos Atari y una implementación desde cero del algoritmo DRQN para aplicar a DOOM.

5.2 Solución Atari

En este apartado se muestra la implementación realizada de un script para realizar tareas de aprendizaje utilizando la librería RLLib. Ésta, como ya se ha comentado anteriormente, nos proporciona una interfaz de uso sencillo para realizar las tareas de aprendizaje, siendo necesario simplemente llamar a determinados métodos de la librería para comenzar la ejecución del aprendizaje.

5.2.1. Implementación de script de pruebas

Para las pruebas sobre entornos Atari, hemos implementado un sencillo script para lanzar pruebas de entrenamiento sobre la librería RLLib. Haciendo uso de los parámetros introducidos por el usuario o por los parámetros por defecto, se realiza un entrenamiento sobre un entorno específico. Es posible realizar la ejecución del aprendizaje sobre un solo algoritmo, o bien realizar una ejecución secuencial de cada uno de los algoritmos sobre el entorno introducido, haciendo uso de la función run_all_algos().

```
1 def run_all_algos():
2   algos = ["DQN", "ES", "PPO", "A2C", "A3C", "IMPALA"]
3   for algo in algos:
4     run_algo(algo)
```

Por otra parte, la función run_algo(algorithm) es la responsable de llamar a la ejecución de cada algoritmo de manera individual, haciendo uso de los métodos nativos de RLLib. El método get_config() recoge del fichero auxiliar la configuración específica recomendada para cada algoritmo. Dada la naturaleza de RLLib, la llamada al método run_experiments() con los valores de configuración correctos es suficiente para ejecutar la operación de aprendizaje.

Como se puede observar, la ejecución viene enormemente condicionada por los parámetros de entrada. Es por ello que en el capítulo **??** - Diseño de pruebas se analizará con más detenimiento los experimentos que llevaremos a cabo.

```
def run_algo(algorithm):
  run_experiments({args.folder: {
    "run": algorithm,
    "env": args.env,
    #El valor de "episode_reward_mean" se actualiza manualmente para cada
        entorno
    "stop": args.stop,
    "config": get_config(algo)
    }})
```

5.3 Implementación de solución para DOOM

Nuestro objetivo llegados a este punto es, como se ha comentado anteriormente, diseñar con éxito una implementación en Tensorflow del algoritmo Deep Reinforcement Q-Learning introducido en la sección (5.3.1) y, una vez realizado, testearlo sobre los diferentes escenarios del entorno ViZDoom. Esta sección del capítulo se centrará en el diseño de las clases que consitituyen la red neuronal que implementa nuestro algoritmo. Principalmente hemos hecho uso de la librería Tensorflow.

Durante esta sección introduciremos algunas partes del código implementado, pero por motivos de brevedad, evitaremos presentar el código completo, que se encuentra disponible en el siguiente enlace.

5.3.1. Algoritmo DRQN

La clase DRQN que hemos diseñado toma tres parámetros como argumentos: Por un lado, necesita saber el tamaño de la imagen que va a recibir como parámetro de entrada para la red, además del número de acciones de las que se dispone en un escenario concreto, y una tasa de aprendizaje normalizada a 0.005.

```
class DRQN():
  def __init__(self, input_shape, num_actions, inital_learning_rate=0.005):
    self.tfcast_type = tf.float32
    # forma de nuestra imagen de entrada (length, width, channels)
    self.input_shape = input_shape
    # numero de acciones en el entorno (heredado de vizdoom)
    self.num_actions = num_actions
    # tasa de aprendizaje de la red neuronal
    self.learning_rate = inital_learning_rate
```

A continuación, es necesario que inicialicemos los parámetros de los filtros, el tamaño del kernel que vamos a utilizar entre otros parámetros necesarios para Tensorflow, como el número de capas ocultas o los hiperparámetros de la función de pérdida.

Una vez tenemos todas las variables inicializadas, lo primero que hacemos es inicializar los inputs y los detectores de características de la red convolucional. Esto lo haremos haciendo uso de las estructuras de Tensorflow tf.placeholder() y tf.Variable().

```
self.input = tf.placeholder(shape = (self.input_shape[0], self.input_shape[1],
    self.input_shape[2]), dtype = self.tfcast_type)

#Inicializaremos tambien la forma del vector objetivo
self.target_vector = tf.placeholder(shape = (self.num_actions, 1), dtype = self
    .tfcast_type)
```

```
5
6  #Inicializacion de los mapas de caracteristicas para nuestros 3 filtros
7  self.features1 = tf.Variable(initial_value = np.random.rand(self.filter_size,
       self.filter_size, input_shape[2], self.num_filters[0]),
8  dtype = self.tfcast_type)
9
10 self.features2 = tf.Variable(initial_value = np.random.rand(self.filter_size,
       self.filter_size, self.num_filters[0], self.num_filters[1]),
11 dtype = self.tfcast_type)
12
13 self.features3 = tf.Variable(initial_value = np.random.rand(self.filter_size,
       self.filter_size, self.num_filters[1], self.num_filters[2]),
14 dtype = self.tfcast_type)
```

Adicionalmente, también es necesario inicializar los parámetros de nuestra red recursiva. Para ello, volvemos a hacer uso de la función tf.Variable() e introduciremos las variables de los pesos compartidos, el sesgo y una tasa de aprendizaje variable que disminuye a medida que la ejecución avanza.

A continuacion toca inicializar las redes convolucionales. Para ello, utilizaremos tres capas de preprocesamiento, cada una con su propia capa de activación y de agrupamiento. El *dropout* hará que nuestra red desactive algunas neuronas aleatoriamente para promover el aprendizaje y evitar el sobreentrenamiento.

```
1  # Primera capa convolucional
2  self.conv1 = tf.nn.conv2d(input = tf.reshape(self.input, shape = (1, self.
       input_shape[0], self.input_shape[1], self.input_shape[2])), filter = self.
       features1, strides = [1, self.stride, self.stride, 1], padding = "VALID")
3  self.relu1 = tf.nn.relu(self.conv1)
4  self.pool1 = tf.nn.max_pool(self.relu1, ksize = [1, self.poolsize, self.
       poolsize, 1], strides = [1, self.stride, self.stride, 1], padding = "SAME")
5
6  # Segunda capa convolucional
7  self.conv2 = tf.nn.conv2d(input = self.pool1, filter = self.features2, strides
       = [1, self.stride, self.stride, 1], padding = "VALID")
8  self.relu2 = tf.nn.relu(self.conv2)
9  self.pool2 = tf.nn.max_pool(self.relu2, ksize = [1, self.poolsize, self.
       poolsize, 1], strides = [1, self.stride, self.stride, 1], padding = "SAME")
10
11 # Tercera capa convolucional
12 self.conv3 = tf.nn.conv2d(input = self.pool2, filter = self.features3, strides
       = [1, self.stride, self.stride, 1], padding = "VALID")
13 self.relu3 = tf.nn.relu(self.conv3)
14 self.pool3 = tf.nn.max_pool(self.relu3, ksize = [1, self.poolsize, self.
       poolsize, 1], strides = [1, self.stride, self.stride, 1], padding = "SAME")
15
16 self.drop1 = tf.nn.dropout(self.pool3, self.dropout_probability[0])
17 self.reshaped_input = tf.reshape(self.drop1, shape = [1, -1])
```

Finalmente, es necesario inicializar las capas de la red recurrente, con bucles de realimentación para mantener información presente a través de las iteraciones. Esta red tomará como datos de entrada el *output* de la red convolucional una vez aplicado el *dropout*. Además, este también se aplicará a las conexiones de la última capa oculta con la capa de salida. Una vez realizada la conexión entre todas las capas, inicializaremos también la variable para acceder a la predicción de la red e introduciremos el optimizador y los gradientes.

```
1  self.h = tf.tanh(tf.matmul(self.reshaped_input, self.rW) + tf.matmul(self.h,
       self.rU) + self.rb)
2  self.o = tf.nn.softmax(tf.matmul(self.h, self.rV) + self.rc)
3  ...
```

```
self.output = tf.reshape(tf.matmul(self.drop2, self.fW) + self.fb, shape = [-1,
    1])
self.prediction = tf.argmax(self.output)
...
```

A continuación también introduciremos brevemente el código utilizado para crear la estructura de memoria, ya descrita en el apartado 4.4. Como se ha comentado, la clase se compone de una función de inicialización y dos funciones: una para agregar experiencias y otra para extraer muestras.

La función de inicialización requiere que se le pase como argumento la capacidad de memoria necesaria, es decir, la cantidad de experiencias máximas a almacenar por el objeto. El objeto resultante estructurará la información en base a tuplas con las distintas variables referentes al estado del agente dentro de la simulación, tomando como datos el estado anterior, el resultante, la acción tomada, la recompensa y una variable booleana que indique si se ha llegado a un estado terminal.

```
def __init__(self, capacity):
  channels = 3
  state_shape = (capacity, resolution[0], resolution[1], channels)

  self.s1 = np.zeros(state_shape, dtype=np.float32)
  self.s2 = np.zeros(state_shape, dtype=np.float32)
  self.a = np.zeros(capacity, dtype=np.int32)
  self.r = np.zeros(capacity, dtype=np.float32)
  self.isterminal = np.zeros(capacity, dtype=np.float32)

  self.capacity = capacity
```

La función *add_transition* asigna los valores de la transición introducida a una posición de memoria a la que se puede acceder sabiendo el valor de la misma. En el caso de que la memoria ya tenga toda su capacidad asignada, comenzará a sobreescribir los valores comenzando por los valores más antiguos.

Por otra parte, la función *get_sample(self, sample_size)* devolverá un número de experiencias igual al número introducido como *sample_size*.

A priori, el conjunto de las clases **ReplayMemory** y **DRQN** debería ser suficiente para introducir todas las estructuras internas necesarias para la implementación de nuestra solución. Esto se verá verificado en la próxima sección, donde pondremos nuestra solución a prueba y evaluaremos los resultados obtenidos.

Método train()

Antes de cerrar este capítulo, es necesario hacer un repaso de la implementación de la funcionalidad principal de nuestra solución, el método de aprendizaje.

Este método recibe como parámetros el número de episodios (*epochs*) que se pretende ejecutar, la duración máxima de lo episodios (*timeout*) y la tasa de aprendizaje inicial. Al realizar la llamada al método, inicializamos las variables globales donde almacenaremos las recompensas y pérdidas totales. Además, dado nuestro requerimiento de realizar una serie de episodios con acciones aleatorias, hemos de inicializar las variables que nos ayudarán a ello:

```
def gymTrain(epochs, episode_length, learning_rate, render = False):
  total_reward = 0
  total_loss = 0
  old_q_value = 0

```

```
pre_train_steps = 5000
total_steps = 0
learning_phase = False
```

Una vez con las variables inicializadas, se abre el **primer bucle de ejecución**, que en cada iteración reinicia el entorno haciendo uso de la librería Gym e implementa la traza de funcionamiento explicada en la sección 4.4.4. Dado que consideramos la explicación mencionada suficiente para la comprensión del funcionamiento interno, no mostraremos la totalidad del código implementado, a excepción del mecanismo de *frame repeat* implementado mostrado a continuación.

```
repeated = frame_repeat   #Indicador de acciones repetidas
if not learning_phase:
  a = env.action_space.sample()
elif repeated == frame_repeat:
  a = actionDRQN.prediction.eval(feed_dict = {actionDRQN.input: s_old})[0]
  print("\nSelecting the action took {:.2f} s".format((time.time() - start)))
  repeated = 1
else:
  repeated += 1
action = actions[a]
state, reward, done, info = env.step(action)
```

Consideramos necesario ahondar en la **implementación del aprendizaje a partir de las muestras de memoria**. Con este fin, diseñamos un bucle if que se ejecute con la frecuencia definida en la inicialización. Inicializaremos las variables a utilizar, relativas a la experiencia que se va a evaluar. Estas son el **estado** (*mem_frame*), la **acción tomada** (*mem_output*) y la **recompensa recibida** (*mem_reward*). El entrenamiento de la red se hace con el uso de la **función *eval()*** nativa de Tensorflow, que evalúa las decisiones tomadas.

```
if (learning_step % update_frequency) == 0:
  trainBatch = experiences.get_sample(10)
  for i in range (0, len(trainBatch) - 1):
    mem_frame = trainBatch[i, 0]
    mem_output = trainBatch[i, 2]
    mem_reward = trainBatch[i, 4]

    # network training
    Q1 = actionDRQN.output.eval(feed_dict = {actionDRQN.input: mem_frame})
    Q2 = targetDRQN.output.eval(feed_dict = {targetDRQN.input: mem_frame})
```

Con las evaluaciones computadas para todas las muestras, podemos aplicar la **ecuación de Bellman** (2.1.10) y calcular las pérdidas:

```
# Calculate Q Value and update
Qtarget = old_q_value + learning_rate * (mem_reward + discount_factor * Q2 -
    old_q_value)
old_q_value = Qtarget

# Loss function
loss = actionDRQN.loss.eval(feed_dict = {actionDRQN.target_vector: Qtarget,
    actionDRQN.input: mem_frame})
tf.summary.scalar('loss', loss)
total_loss += loss
```

Por último, realizamos el proceso de ***backtracking*** y actualizamos los pesos de las dos redes.

```
# Update networks
actionDRQN.update.run(feed_dict = {actionDRQN.target_vector: Qtarget,
    actionDRQN.input: mem_frame})
```

```
3   targetDRQN.update.run(feed_dict = {targetDRQN.target_vector: Qtarget,
        targetDRQN.input: mem_frame})
```

5.3.2. Integración de aprendizaje con ViZDoom y Gym

Para poder utilizar nuestro entorno, hemos integrado la librería VizDoom haciendo uso de una versión modificada de la interfaz Vizdoomgym. Esencialmente, la función de esta librería es inicializar los distintos escenarios de Vizdoom como entornos de Gym para el acceso a sus datos.

Para la implementación de esta herramienta en nuestro *script DoomGym* tenemos que seguir varios pasos: En primer lugar en el metodo *__init__()*, inicializamos el entorno haciendo una llamada al método gym.make() e introducimos las acciones disponibles:

```
env = gym.make(args.scenario)   # Entorno doom

n = env.game.get_available_buttons_size()   # Numero de acciones disponible
actions = np.zeros(n, n)  # Matriz de ceros nxn
count = 0

for i in actions:
  i[count] = 1
  count += 1
actions = actions.astype(int).tolist()
```

A continuación, hemos de inicializar las estructuras de memoria y las redes necesarias con llamadas a los métodos descritos anteriormente.

```
experiences = experience_buffer(buffer_size=5000)

actionDRQN = DRQN((160, 256, 3), env.game.get_available_buttons_size(),
    learning_rate)
targetDRQN = DRQN((160, 256, 3), env.game.get_available_buttons_size(),
    learning_rate)
```

Una vez se han inicializado estas variables, inicializamos una sesion de Tensorflow y hacemos la llamada al método *train()*.

```
with tf.Session(config=config) as sess:

  writer = tf.summary.FileWriter("logs", sess.graph)

  print("----------Starting the training----------")

  gymTrain(epochs, episode_length, learning_rate, render=False)
```

Por último y una vez el entrenamiento termine, se realiza una serie de pruebas para comprobar el rendimiento del algoritmo, mostrando el juego por pantalla.

```
  print("Training finished. It's time to watch!")
  env.game.set_window_visible(True)
  env.game.set_mode(vzd.Mode.ASYNC_PLAYER)
  #game.init()

  for _ in range(episodes_to_watch):
    env.reset()
    done = False
    while not done:
      state, reward, done, info = env.step(action)
      a = actionDRQN.prediction.eval(feed_dict = {actionDRQN.input: s_old})[0]

```

```
13          action = actions[a]
14          # Instead of make_action(a, frame_repeat) in order to make the animation
                smooth
15          env.game.set_action(a)
16          for _ in range(frame_repeat):
17            env.game.advance_action()
```

CAPÍTULO 6
Diseño de pruebas

6.1 Introducción

Dada la estructura de nuestro trabajo, consideramos necesario hablar de las clases de experimentaciones que vamos a llevar a cabo. Es por ello que en este capítulo nos centraremos en describir entre otros qué algoritmos hemos seleccionado para llevar a cabo nuestros experimentos y qué resultados esperamos conseguir basándonos en las experimentaciones llevadas a cabo anteriores a la redacción de este trabajo. [1], [7], [15]

Por razones de conveniencia, denominaremos **experimento** a una ejecución de cualquiera de las dos soluciones con un par algoritmo/entorno. Esta definición se mantendrá a lo largo de los próximos capítulos.

6.2 Experimentaciones a realizar en entornos ATARI

Sobre ATARI realizaremos una serie de experimentos haciendo uso de la herramienta *custom_learning* utilizando una serie de hiperparámetros comunes a todos ellos. Además, utilizaremos las mismas reglas de convergencia para todas las ejecuciones que pertenezcan a un mismo entorno.

6.2.1. Algoritmos y entornos a testear

Con respecto a los algoritmos que ejecutaremos, para esta parte realizaremos una selección del catálogo de algoritmos que la librería RLLib nos ofrece. Principalmente, realizaremos pruebas sobre algunos algoritmos asíncronos (A2C, A3C, IMPALA) así como algunos de los algoritmos más clásicos que hemos introducido (DQN, PPO). Adicionalmente, RLLib proporciona también una implementación del algoritmo de estrategias evolutivas sobre la que también realizaremos algunas pruebas.

Ejecutaremos nuestras experimentaciones sobre los mismos entornos introducidos en la sección 3.2.2.

La librería Gym nos ofrece distintas versiones de un mismo entorno, diferenciadas por factores condicionantes del entorno como la probabilidad de repetición de una acción independiente de la acción seleccionada, o . En nuestro caso utilizaremos principalmente las versiones *NoFrameskip*, pero a ser posible se pretende también realizar algunos experimentos en paralelo para cotejar las diferencias que esto puede suponer.

6.2.2. Condiciones de parada

A continuación discutiremos una de las secciones más críticas para conseguir resultados sólidos en nuestras experimentaciones. Se trata de las condiciones que introduciremos como parámetro para que dicten cuándo se debe finalizar la ejecución de cada experimento.

A medida que el entrenamiento progrese y la colección de experiencias aumente, es lógico pensar que llegará un momento que podremos decir que el agente ha **resuelto** el entorno, es decir, ha encontrado una serie de políticas a seguir que le llevan a completar el juego que se le presenta. Será necesario por tanto, darle un **valor numérico** a la recompensa total media que un agente debe recibir para considerar que el entorno ha sido resuelto. Este valor se introducirá como parámetro en cada ejecución como *episode_reward_mean*.

Debido a restricciones de tiempo, también limitaremos la ejecución de cada experimento a un máximo de 20 horas. Esta restricción nos permitirá recoger una gran variedad de resultados de distintos entornos con un entrenamiento medianamente considerable, pero el hecho de que no todos los algoritmos probados tengan la misma eficiencia conllevará posiblemente el problema de que en algunos casos existan resultados descompensados.

6.2.3. Diseño de hiperparámetros

Para esta solución, los hiperparámetros de cada algoritmo vendrán dados por el fichero de configuración *config.py*. En la siguiente tabla se recogen algunos de los más importantes.

Para esta solución, se han utilizado una serie de hiperparámetros globales que en algunos casos pueden ser sobreescritos por el fichero de configuración *config.py*. Estas configuraciones globales son las siguientes:

- **lr_schedule** (tasa de aprendizaje): [0, 0.0005]
- **sample_batch_size:** 50
- **train_batch_size:** 500

Adicionalmente, cada configuración particular puede implementar pequeñas variaciones de estos factores, o puede introducir nuevas variables de configuración como el factor de descuento o modificaciones al funcionamiento interno de la red neuronal (por ejemplo, al número de neuronas de las capas ocultas).

Cada uno de los servidores que utilizaremos para lanzar los entrenamientos cuenta con unas especificaciones particulares a las que habrá que adaptarse y consecuentemente limitarán el alcance del aprendizaje. Será por tanto necesario diseñar distintos parámetros de lanzamiento que limiten la cantidad de recursos utilizados en cada una de las ejecuciones para los servidores sobre los que lancemos pruebas. Estas restricciones vienen documentadas en la tabla 6.1.

Servidor de ejecución	Workers	GPUs	Memoria límite
Titan Xp + Intel^R Core i7-7700 @ 3.60Ghz	7	1	2GB
Titan V x2 + Intel^R Xeon W-2123 @ 3.60GHz	7	2	Sin restricciones
Geforce GTX 970 + AMD Ryzen 5 2600 @ 3.90 GHz	11	1	2GB

Tabla 6.1: Tabla de configuraciones de los experimentos en cada servidor.

6.2.4. Ejecución

En esta sección introduciremos la estructura de los experimentos que se plantean lanzar sobre los distintos servidores a los que tenemos acceso.

Durante la fase de experimentación se plantea lanzar ejecuciones para cada algoritmo utilizando distintos parámetros de condición de parada. En la tabla 6.2 se introduce el diseño de las distintas ejecuciones a evaluar con información sobre la duración máxima del entrenamiento, los algoritmos a estudiar y el servidor utilizado. El tiempo total de ejecución estimado para realizar la serie de experimentaciones completa aproxima las 450 horas.

Id	Entorno	Algoritmos	Servidor	Duración
1	**Breakout-v0**	DQN, A2C, A3C, IMPALA, ES	Titan Xp	2 horas
2	**Atlantis-v0**	DQN, A2C, A3C, IMPALA	Titan Xp	3 horas
3	**SpaceInvaders-v0**	DQN, A2C, A3C, IMPALA	Titan Xp	3 horas
4	**Pong-v4**	DQN, A2C, A3C, IMPALA	GTX 970	4 horas
5	**Seaquest-v4**	DQN, A3C, IMPALA, PPO	x2 Titan V	12 horas
6	**SpaceInvaders-v4**	DQN, A2C, A3C, IMPALA, PPO	Titan Xp	12 horas
7	**PongNoFrameskip-v4**	DQN, A2C, A3C, IMPALA, ES	Titan Xp	12 horas
8	**BreakoutNoFrameskip-v4**	DQN, A2C, A3C, IMPALA, ES	x2 Titan V	20 horas
9	**AtlantisNoFrameskip-v4**	DQN, A2C, A3C, IMPALA	x2 Titan V	20 horas
10	**BoxingNoFrameskip-v4**	A2C, A3C	Titan Xp	20 horas
11	**Boxing-v4**	DQN, IMPALA	Titan Xp	20 horas

Tabla 6.2: Tabla de las experimentaciones lanzadas.

6.2.5. Procesamiento gráfico de los datos

La librería Ray devuelve los ficheros de resultados de manera automática, consistentes en un *logger* de los eventos sucedidos y algunos ficheros con formato JSON con información de los parámetros de entrada y los resultados obtenidos.

Para acceder a los resultados, inicializaremos una sesión de la herramienta Tensorboard a la que podremos acceder con cualquier navegador de internet. Esta herramienta nos proporcionará un enorme catálogo de gráficas de resultados a partir de los ficheros de información devueltos por RLLib. Las gráficas más interesantes para la verificación de nuestros experimentos serán aquellas que reflejen las recompensas recogidas, los valores de error TD o la variación de los Q-Values estimados a lo largo del entrenamiento.

Tensorboard nos ofrece una herramienta de suavizado para evitar el ruido que se puede llegar a crear. **En todas las visualizaciones de resultados se utilizará un suavizado de 0.9.**

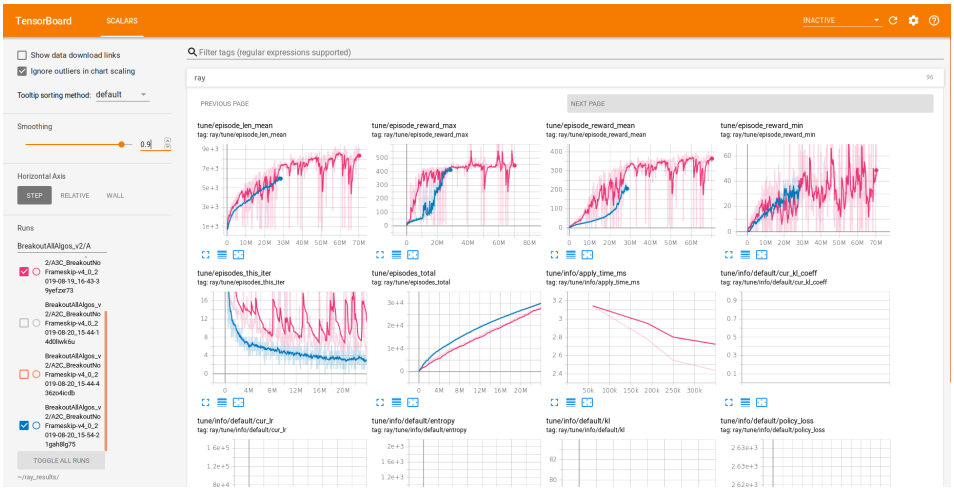


Figura 6.1: Interfaz gráfica de la herramienta Tensorboard mientras se realizan experimentos.

Debido a la naturaleza de los distintos algoritmos, es más que probable que las distintas ejecuciones realicen un número de iteraciones dispar en el periodo de tiempo que se les proporciona. Es por ello que en la visualización de los resultados haremos uso de la herramienta de muestra relativa de los resultados, que nos devolverá una gráfica en función del tiempo transcurrido a diferencia de la visualización estándar cuyo eje X sería el número de iteraciones realizadas.

6.3 Pruebas en DOOM

Adicionalmente a las experimentaciones que llevaremos a cabo sobre Atari, también es necesario probar nuestra solución DRQN para verificar su funcionamiento y extraer algunas conclusiones al respecto. Es por ello que una vez verificado el funcionamiento, trataremos de evaluar el comportamiento de nuestro algoritmo sobre algunos escenarios del catálogo que nos ofrece ViZDoom.

6.3.1. Diseño de hiperparámetros

Dado que se trata de una solución propia, es lógico hablar de la selección de hiperparámetros que introduciremos para llevar a cabo la tarea de aprendizaje.

Como se ha mencionado en la capítulo 4, nuestra clase DRQN() recibe como parámetros de entrada estos valores que condicionarán la ejecución:

- **Factor de descuento:** 0.99
- **Tasa de aprendizaje:** 0.005 (0.5 %)
- ***Frame repeat:*** 12
- **Número de episodios (epochs):** 10000

Utilizaremos el *frame repeat* como mecanismo para reducir la carga computacional del entrenamiento, dado que nos permite ignorar un conjunto de estados adyacentes que ofrecen informaciones muy similares. Reducir el valor de este factor nos resultaría en un agente más dinámico, pero no obstante requeriría un tiempo de entrenamiento mucho mayor.

Se ha seleccionado un número relativamente bajo de episodios de entrenamiento debido a las restricciones de tiempo encontradas durante la realización del trabajo. A pesar de ello, se espera poder reunir unos resultados de buena calidad sobre los que poder dibujar ciertas conclusiones.

6.3.2. Escenarios a testear sobre DRQN

Para las pruebas en Doom, vamos a realizar una pequeña selección de los escenarios ofrecidos que mencionamos en la sección 3.2.3. En concreto, se planea realizar pruebas sobre los entornos **Basic, Health Gathering, Defend Center** y **My Way Home.**

Debida la naturaleza de cada uno de los entornos seleccionados, se espera conseguir una serie de resultados que nos permita hacer conjeturas sobre si DRQN funciona mejor según qué tipo de estrategia sea necesaria seguir para completar el objetivo.

6.3.3. Condiciones de parada

En el caso de las experimentaciones con DOOM nuestra única condición de parada introducida consiste en la compleción del número de iteraciones (*epochs*) introducido como parámetro a la solución.

6.3.4. Ejecución

Se han diseñado una serie de experimentaciones que se pretende ejecutar para probar nuestra implementación del algoritmo DRQN. Adicionalmente a las pruebas de DRQN, se pretende hacer pruebas sobre algún otro algoritmo como IMPALA o A3C para, en la medida de lo posible, contrastar el comportamiento en éste tipo de entornos de observación parcial.

A continuación en la tabla 6.3 introducimos los experimentos que se plantean. Sus resultados serán descritos con detalle en la sección 7.2

id	Servidor de ejecución	Escenario	Parada	Algoritmo
1	**DoomBasic-v0**	Titan V x2	26 horas	DRQN
2	**DoomDefendCenter-v0**	Titan V x2	24 horas	DRQN

Tabla 6.3: Tabla de configuraciones de los experimentos sobre DOOM.

6.3.5. Procesamiento de los datos

A pesar de trabajar con Tensorflow, nuestra solución no implementa una recolección gráfica de datos con formato Tensorboard tan completa como la que ofrece la librería RLLib. Es por ello que para mostrar los resultados relativos a las puntuaciones conseguidas por los distintos experimentos mostraremos la información por línea de comandos en función del número de iteraciones (*epochs*) realizadas. Nuestra implementación también guarda periódicamente una copia del modelo de la red actualizada. Utilizaremos entradas con un formato similar al mostrado en la figura 6.2. En este formato se puede observar datos relativos a las recompensas obtenidas durante toda la ejecución. Nos centraremos en este dato ya que se considera el factor principal sobre el que podremos verificar el funcionamiento de nuestra solución.

```
 10%|                                                                                | 205/2100 [00:22<05:59,  5.27it/s]
Selecting the action took 1.05 s
 10%|                                                                                | 217/2100 [00:23<05:01,  6.24it/s]
Selecting the action took 1.05 s
 11%|                                                                                | 229/2100 [00:24<04:21,  7.17it/s]
Selecting the action took 1.05 s
 11%|                                                                                | 241/2100 [00:25<03:52,  8.00it/s]
Selecting the action took 1.10 s
 12%|                                                                                | 253/2100 [00:26<03:34,  8.60it/s]
Selecting the action took 1.05 s
 13%|                                                                                | 265/2100 [00:27<03:19,  9.19it/s]
Selecting the action took 1.07 s
 13%|                                                                                | 277/2100 [00:28<03:09,  9.61it/s]
Selecting the action took 1.09 s
 14%|                                                                                | 289/2100 [00:29<03:03,  9.86it/s]
TOTAL REWARD: -350.0

27 training episodes played.
Results: mean: -350.0±0.0, min: -350.0, max: -350.0,
TOTAL STEPS: 5498
```

Figura 6.2: Ejemplo de muestra por terminal de resultados.

CAPÍTULO 7
Evaluación de resultados

En este capítulo se proponen dos casos de estudio distintos, el primero basado en las experimentaciones realizadas sobre los entornos de la Atari 2600, haciendo uso de las librerías propuestas de aprendizaje por refuerzo, y un segundo caso de estudio tomando el entorno ViZDoom como base para el diseño de un algoritmo DRQN y la consecuente verificación de su funcionamiento. La metodología utilizada viene descrita en capítulo 6.

A lo largo de este capítulo, por tanto, explicaremos los experimentos realizados y contrastaremos los resultados obtenidos. Posteriormente, trataremos de realizar un análisis de las correlaciones entre los distintos factores que pueden afectar a la resolución de los entornos.

7.1 Caso ATARI

El principal objetivo de este caso de estudio consiste en analizar los datos obtenidos en las experimentaciones realizadas sobre ATARI con el objetivo de validar nuestra herramienta y contrastar el funcionamiento de las distintas configuraciones de experimento que hemos diseñado. Expondremos pues, en esta sección los resultados que se consideren más ilustrativos de los introducidos en la tabla 6.2.

7.1.1. Resultados de Pong

El entorno Pong presenta una estrategia de resolución relativamente sencilla, que consiste en aprender a predecir la posición de la pelota cuando esta atraviese la parte izquierda de la pantalla. La pelota adoptará una trayectoria particular en función de la posición relativa de la raqueta con respecto a la pelota.

Los resultados relativos a la media de puntuación del episodio pertenecientes a la experimentación [7] aparecen reflejados en la figura 7.2. En esta gráfica se observa que solamente los algoritmos IMPALA y A2C consiguen aplicar la estrategia ganadora, quedando DQN y A3C con un aprendizaje prácticamente inexistente con una recompensa media ligeramente superior a la mínima de -21. Por su parte, IMPALA y A2C atraviesan el umbral de aprendizaje a las pocas horas de comenzar el experimento, es decir, adquieren el conocimiento de las estrategias necesarias para la resolución, y mantienen una puntuación casi perfecta durante todo el resto de la ejecución.

Con respecto al algoritmo A3C, se desconoce por qué su aprendizaje no ha alcanzado los mismos valores que el algortimo A2C, similar en todos los aspectos exceptuando que A3C implementa una estructura asíncrona.

Adicionalmente se puede observar que en IMPALA y A2C el aprendizaje se acelera notablemente una vez el agente descubre que puede aumentar su puntuación posicionando su raqueta a la altura de la pelota. Una vez esto ocurre, el tiempo transcurrido entre una recompensa de -20 y la máxima de 20 es ligeramente superior a dos horas en los dos casos, con un pequeño intervalo de tiempo cuando las puntuaciones recibidas rondan el empate, en el que el aprendizaje se ralentiza. Consideramos que esto puede ser debido a que a pesar de que el agente ya haya aprendido a devolver la pelota, todavía no dispone de la experiencia necesaria para superar en habilidad al oponente. Una vez encuentra una estrategia favorable, el aprendizaje se reanuda y completa en un período de tiempo muy reducido.

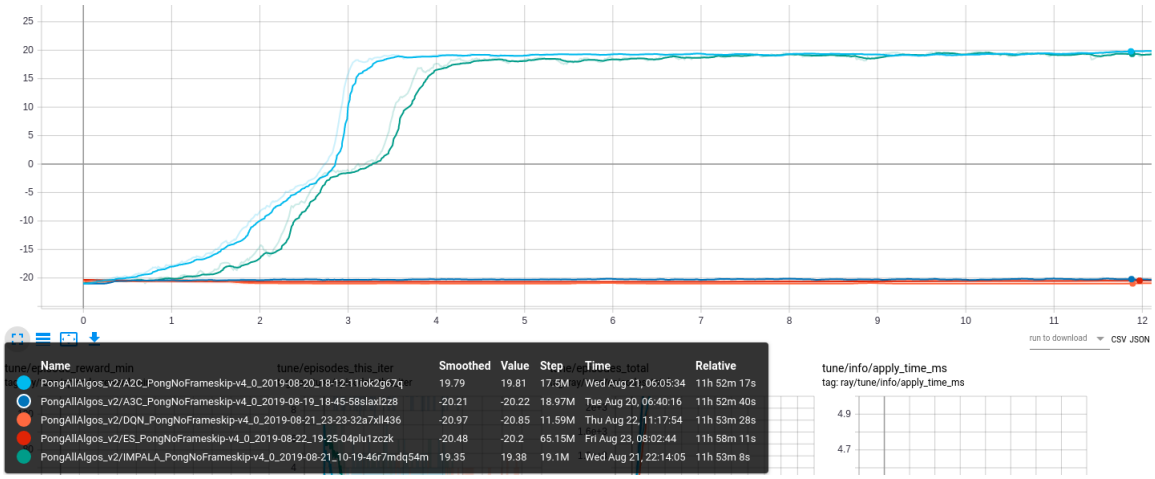


Figura 7.1: Resultados de PongNoFrameskip-v4.

En este caso la estrategia no requiere una gran precisión para devolver la pelota, por lo que a priori se podría considerar válido que una misma acción se repitiera durante un número indeterminado de estados. Es por ello que adicionalmente a los resultados expuestos en la figura 7.2 decidimos contrastar las repercusiones que el frame skipping podría tener durante el entrenamiento ejecutando una tanda adicional de pruebas [4] sobre el entorno *Pong-v4* que implementa *frame skipping*. Desafortunadamente, los resultados obtenidos no son concluyentes ya que con el tiempo de ejecución asignado en el servidor local (GTX 970) no ha sido posible realizar un número de episodios suficiente para extraer conclusiones.

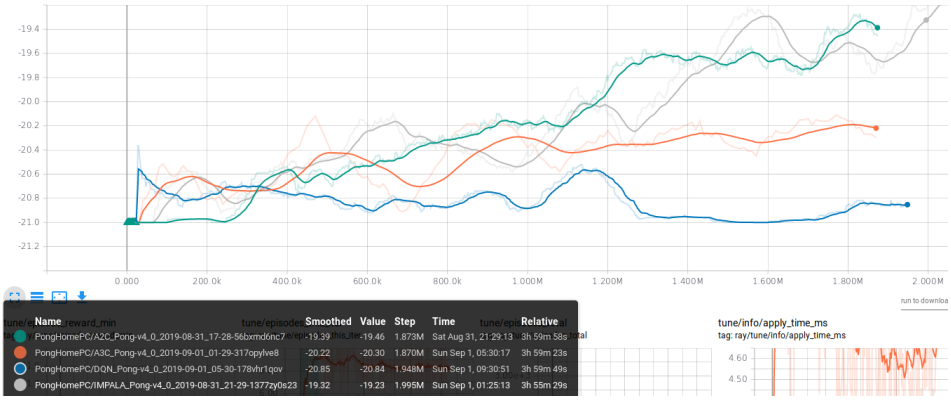


Figura 7.2: Puntuaciones medias relativas a la experimentación [4] sobre el servidor personal. Se puede observar que todos los algoritmos a excepción de DQN experimentan mejoras en la puntuación a lo largo de los escasos 2M de iteraciones realizadas, siendo IMPALA y A2C de nuevo los más eficientes.

7.1.2. Resultados de Space Invaders

Este entorno se basa en Space Invaders, uno de los videojuegos más emblemáticos de la época de Atari. En este caso, la estrategia de resolución puede variar, pero en esencia se trata de eliminar al mayor número de enemigos presentes en la parte superior de la pantalla, antes de que éstos sean capaces de alcanzar nuestra posición.

Sobre este entorno hemos realizado los experimentos número [3] y [6] de la tabla 6.2, los dos sobre una variante del entorno similar pero distinta duración de entrenamiento. A continuación comentaremos lo más destacable de nuestras observaciones utilizando por motivos de claridad tablas relativas al tiempo transcurrido de entrenamiento, dejando así constancia de que DQN en ambos casos ha tenido un rendimiento peor ejecutando solamente alrededor de un 60 % de las iteraciones realizadas por el resto de algoritmos.

En primer lugar, en la figura 7.3 podemos observar el comportamiento de los algoritmos las tres primeras horas del aprendizaje. Aquí, a pesar de una cierta superioridad en las puntuaciones del algoritmo IMPALA, no se puede observar una gran diferencia entre el resto de algoritmos al finalizar la ejecución, estando A2C y A3C prácticamente igualados.

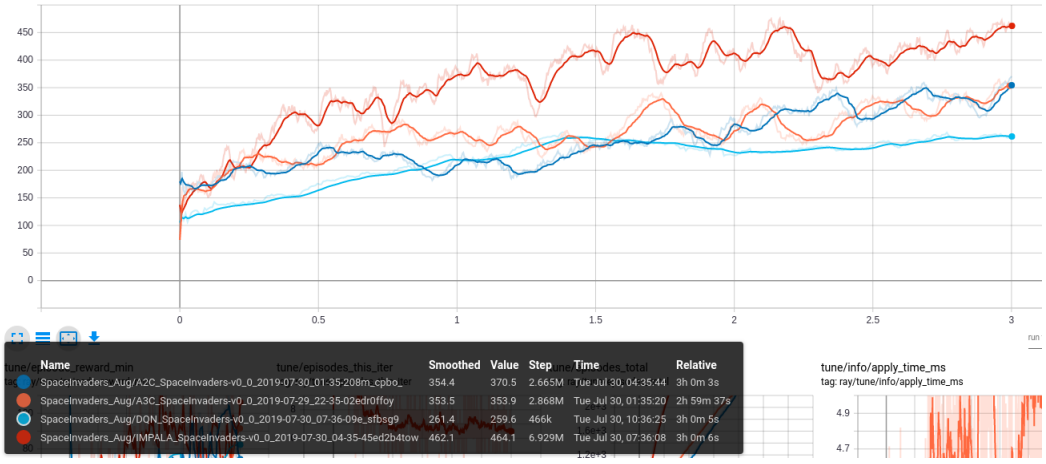


Figura 7.3: Resultados de las experimentaciones de 3 horas sobre SpaceInvaders-v0 en función del tiempo.

No obstante, una vez observamos la ejecución completa de 16 horas (fig. 7.4) se puede distinguir un empeoramiento en el rendimiento relativo de A3C, a partir de las 3 horas, igualando una puntuación media similar a DQN durante las últimas horas del entrenamiento. Por otra parte, A2C e IMPALA consiguen una puntuación media relativamente baja, pero muestran un comportamiento muy similar durante toda la ejecución.

Por otra parte, podemos observar los mismos resultados en función de las iteraciones realizadas y comprobar que, a pesar de que las puntuaciones son similares, las iteraciones realizadas por cada algoritmo varían enormemente. De hecho, se puede observar que el algoritmo DQN parece tener un mayor nivel de capacidad de aprendizaje por episodio de entrenamiento. No obstante, el factor del coste temporal que conlleva realizar cada episodio le impide alcanzar puntuaciones superiores a los otros algoritmos. De la misma forma, A2C y A3C presentan el mismo problema con respecto a IMPALA que se muestra como el algoritmo más rápido en cuanto a episodios/iteración (figura 7.5). En esta figura también se puede observar cómo a medida que el tamaño de la memoria aumenta, la tasa episodio/iteración se reduce en todos los algoritmos, posiblemente debido al tiempo adicional que requiere el acceso a la memoria.

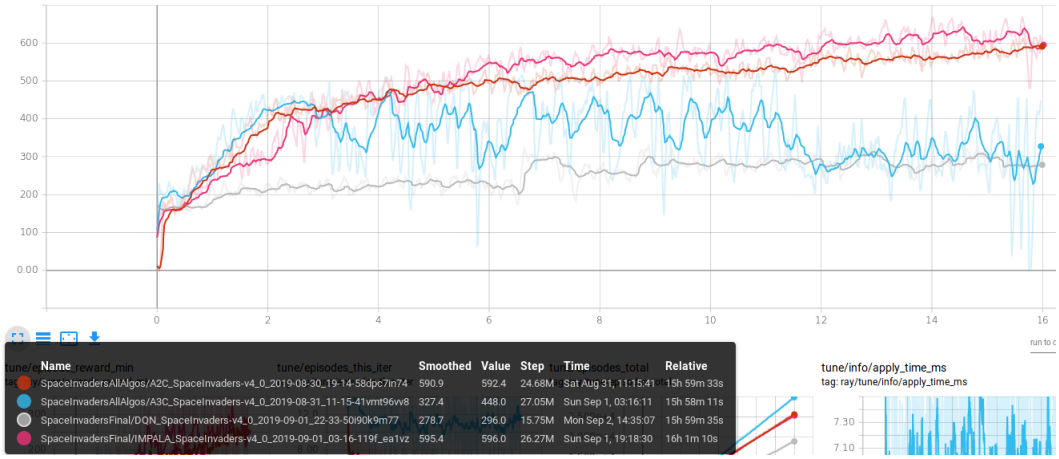


Figura 7.4: Resultados de las experimentaciones de 16 horas sobre el entorno SpaceInvaders-v4.

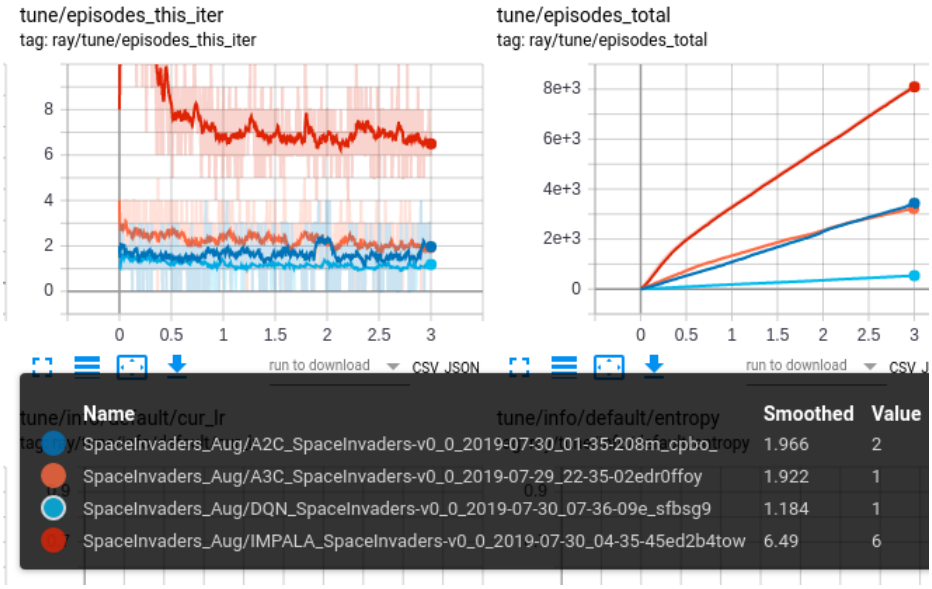


Figura 7.5: Gráfica de número de episodios/iteración. Como se puede observar, IMPALA es el algoritmo más eficiente respecto a este factor, lo cual le proporciona potencialmente una gran ventaja sobre sus competidores.

7.1.3. Resultados de Breakout

Breakout ha sido el entorno elegido para realizar un gran número de los experimentos preliminares para verificar el funcionamiento. Durante la fase de preparación realizamos algunas pruebas con tasa de aprendizaje variable, así como breves ejecuciones de los algoritmos de estudio. Finalmente, diseñamos unas pruebas extensas sobre el entorno *NoFrameskip* de duración máxima de 12 horas, pero cuya ejecución terminaría si se alcanzaba una puntuación media de 400.

La estrategia de resolución de este entorno se asemeja a Pong en el hecho de que el jugador controla una raqueta que debe utilizar para devolver la pelota cuando se aproxime al límite inferior de la pantalla. No obstante, a diferencia de Pong el objetivo principal consiste en eliminar bloques de la parte superior de la pantalla. El agente no puede fallar el impacto con la pelota en más de tres ocasiones, causando el final de la ejecución si esto ocurre. Es por ello que el agente deberá aprender distintas mecánicas: predecir el movimiento de la pelota, modificar la trayectoria de la misma en el impacto y, en el mejor

de los casos, conseguir pasar la pelota a la parte superior de la pantalla para que rebote en múltiples ocasiones contra los bloques y conseguir una gran cantidad de puntos. Consideramos que este entorno tiene una dificultad media de aprendizaje.

A pesar de haber realizado la prueba con todos los algoritmos introducidos en la memoria, no mostramos en la figura los datos de entrenamiento de las estrategias evolutivas. Esto se debe a que el formato de la información de los resultados de este algoritmo no es compatible con la herramienta Tensorboard y, por lo tanto, no nos ha sido posible comparar su funcionamiento. Los resultados del resto de algoritmos se encuentran expuestos en la figura 7.6.

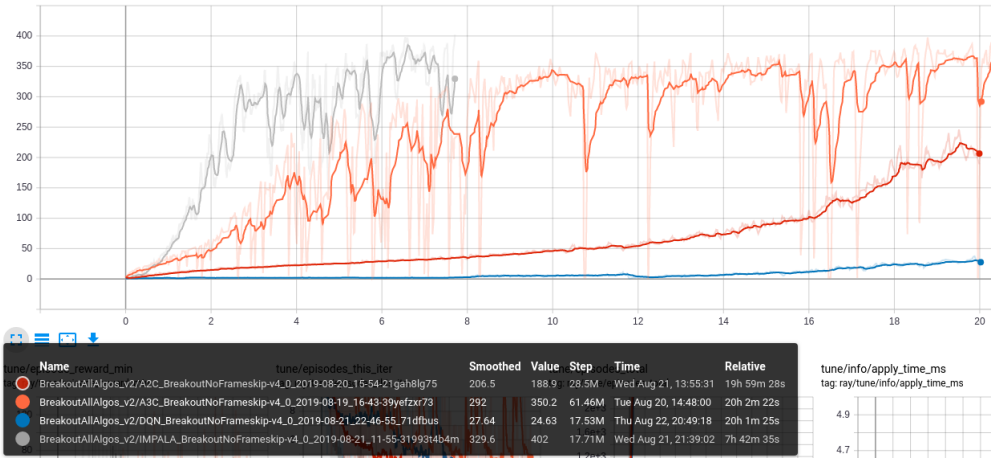


Figura 7.6: Resultados relativos al tiempo de las experimentaciones en BreakoutNoFrameskip-v4.

En primer lugar, se puede observar que la ejecución de IMPALA termina poco antes de las 8 horas de entrenamiento debido a que se alcanza el límite impuesto de puntuación media. Por otra parte A3C también presenta unos resultados positivos, pero no llega en ningún momento al límite de resolución que hemos impuesto rondando una recompensa media de 300 durante gran parte de su ejecución.

A2C presenta la particularidad con respecto a A3C de que se ha realizado un número de iteraciones (28.5 millones) considerablemente inferior que su versión asíncrona A3C. Esto es posiblemente debido a que, mientras que la estructura interna de A2C permite solamente la ejecución individual de un agente *worker* para realizar el entrenamiento, A3C inicializa múltiples agentes para la recogida de experiencias y, por tanto, le puede resultar más sencillo dar con estrategias favorables para la resolución del entorno.

Por último, de nuevo podemos observar que DQN se queda atrás con respecto al resto de algoritmos. Al igual que A2C, no cuenta con una estructura asíncrona que le permite dividir la recogida de experiencias entre múltiples actores, lo cual condiciona en gran medida la cantidad de información que conoce del entorno. Esto se ve también reflejado en que después de 20 horas de entrenamiento (17.5 millones de iteraciones) el agente no es capaz de superar la puntuación media de 30, cuando los algoritmos A3C e IMPALA fueron capaces de superar la misma marca después de aproximadamente una hora de entrenamiento.

7.1.4. Resultados de Boxing

Como se observa en la figura 7.7, el comportamiento de los distintos algoritmos en este entorno resulta muy similar. Todos ellos aprenden muy rápidamente la relación entre golpear al enemigo y el aumento de la puntuación, por lo que son capaces en muy poco tiempo de conseguir una puntuación cercana al empate.

A pesar del rápido aprendizaje que lleva a los experimentos a alcanzar una puntuación media de aproximadamente 0, ningún algoritmo ha sido capaz de extender este conocimiento y resolver el entorno. Nuestra hipótesis es que esto podría suceder rápidamente en el momento que los agentes aprendieran a esquivar los golpes del rival; no obstante, esto no se ha visto reflejado posiblemente porque en la implementación del entorno ofrecida por Gym los agentes nunca reciben una recompensa positiva al esquivar un golpe, y en consecuencia no es posible deducir la relación entre la técnica de esquive y una mayor puntuación al final de la partida.

En este entorno decidimos contrastar los resultados de ejecutar distintas versiones del entorno que implementaran la técnica del *frame skipping*, siendo A2C/A3C en el entorno *BoxingNoFrameskip-v4* [10] y DQN e IMPALA ejecutados en la variaente *Boxing-v4* [11]. A pesar de que durante las primeras horas no se observa nada fuera de lugar, el algoritmo DQN alcanza un punto en el que su recompensa media se ve reducida de manera significativa. Las causas de esto se desconocen, pero es posible que el uso del *frame skipping* haya tenido una influencia negativa en este entorno, dado que para su resolución requiere una precisión en las acciones considerable.

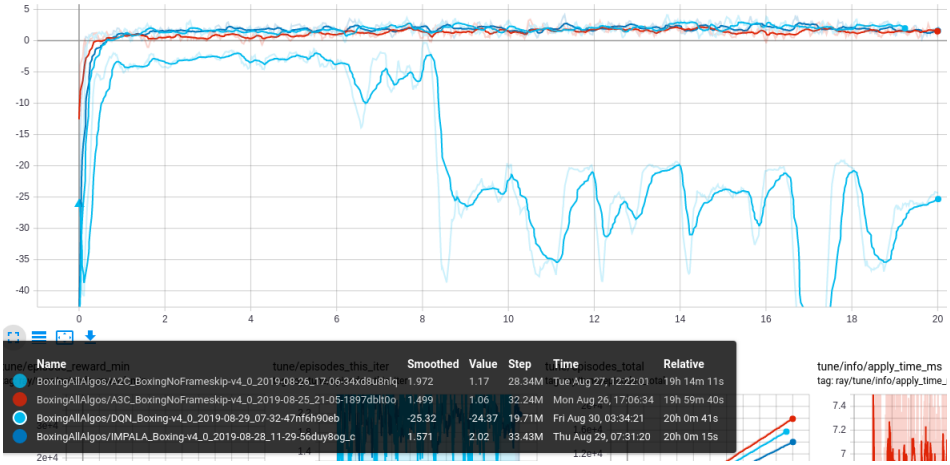


Figura 7.7: Resultados relativos al tiempo de las experimentaciones en el entorno Boxing-v0 y BoxingNoFrameskip-v4.

En conclusión, consideramos que sería posible para cualquier algoritmo alcanzar muy rápidamente la resolución con una simple modificación sobre el funcionamiento del entorno, dotándolo de la estructura para recompensar el esquive; no obstante, la implementación de Gym aporta una serie de recompensas pre-definidas cuya modificación podría conllevar una inversión de tiempo demasiado grande para que lo consideremos como una tarea adicional de nuestro trabajo.

Con respecto al *frame skipping*, consideramos los resultados insuficientes para sugerir su posible influencia sobre el aprendizaje.

7.1.5. Resultados de Atlantis

Previamente a la experimentación, consideramos este juego junto a Seaquest como aquellos con la dificultad más elevada para los agentes de aprendizaje por refuerzo, debido a la dificultad que conlleva utilizar correctamente las distintas torretas que el jugador puede accionar. Históricamente ninguna de las experimentaciones con agentes por refuerzo ejecutadas sobre este entorno han podido superar el rendimiento conseguido por un jugador experimentado en el mismo. Tomando estos datos previos a nuestro trabajo como referencia[1], diseñamos la condición de parada para terminar la ejecución cuando se llegase a una puntuación máxima de 2300000, o bien se alcanzara el límite de 20 horas de ejecución.

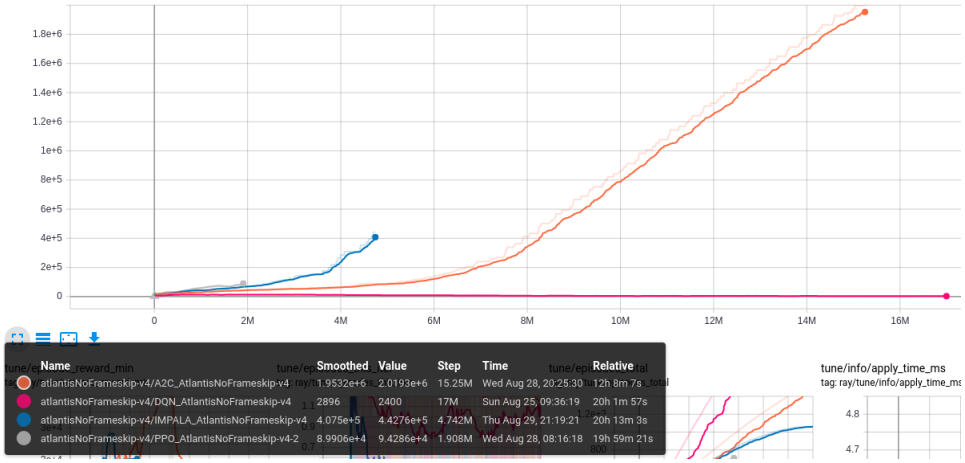


Figura 7.8: Gráfico de recompensas medias obtenidas durante experimentos sobre AtlantisNoFrameskip-v4.

En la figura 7.8 se observa en primer lugar el hecho de que A2C ha sido el algoritmo que mejor puntuación ha sido capaz de acumular a lo largo de su ejecución. A diferencia del resto de experimentaciones, observamos que en este caso IMPALA reduce considerablemente el número de iteraciones de aprendizaje realizadas en comparación con sus competidores, pero a su vez es el que mejor puntuación acumula en la marca de los 5 millones de iteraciones. Esto nos hace intuir que, a pesar de su bajo rendimiento temporal en este caso, tiene potencial para superar a A2C dadas las iteraciones necesarias.

Otra de las cosas remarcables de estos datos se puede observar revisando la gráfica de puntuaciones máximas, representada en la figura 7.9. Aquí se puede ver cómo tanto IMPALA como A2C llegan a un "tope"de aprendizaje, el primero alcanzando este límite en 240000 de puntuación total y el último en el valor de 270000. Se observa que, a partir de este límite, el agente no es capaz de aprender estrategias nuevas para superar esta marca, sino que en cambio consigue aumentar la recompensa media a base de repetir la estrategia óptima conocida.

Por su parte, PPO se presenta como un algoritmo bastante ineficiente, dado que no es capaz de ejecutar más de 2 millones de iteraciones a lo largo de las 20 horas de ejecución, mientras que DQN parece tener problemas para resolver el entorno, sin ser capaz de conseguir una puntuación superior a 4500. Adicionalmente, cabe añadir que el tiempo de A2C se vio mermado en un 40 % por un error en tiempo de ejecución, llegando sin embargo a abarcar un número de episodios de entrenamiento similar a DQN.

[1] https://github.com/cshenton/atari-leaderboard

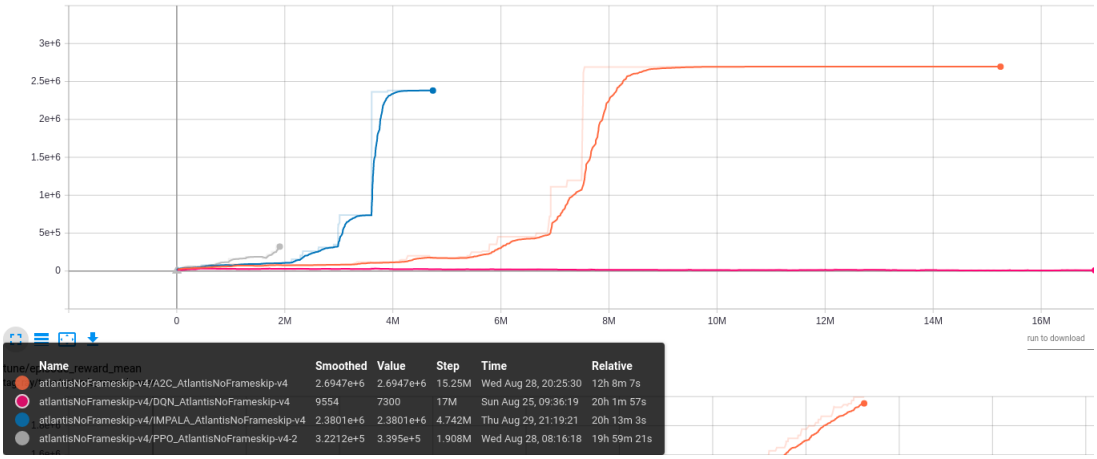


Figura 7.9: Gráfico de recompensas máximas obtenidas durante los experimentos sobre AtlantisNoFrameskip-v4.

7.1.6. Resultados de Seaquest

Como se ha mencionado en secciones anteriores, consideramos este entorno como el más desafiante para los modelos actuales de aprendizaje por refuerzo, dado que en ciertas situaciones requiere una toma de decisiones a medio/largo plazo que los algoritmos estudiados no son típicamente capaces de asimilar.

En este caso, hemos ejecutado una sola tanda de experimentaciones sobre el entorno *Seaquest-v4* y una duración de 16 horas, cuyos resultados aparecen mostrados en la figura 7.10. En esta se puede observar que, en este caso, es A2C el algoritmo que obtiene la mayor puntuación media, seguido en segundo lugar por A3C e IMPALA y con DQN ligeramente por detrás de estos últimos. Es necesario mencionar también que el algoritmo PPO presenta un comportamiento muy similar al de A3C durante su ejecución. No obstante, su baja optimización no permite una recolección de resultados tan extensa como con el resto de algoritmos. Adicionalmente, es necesario mencionar que las recompensas obtenidas son significativamente más bajas que la puntuación máxima de 50000 documentada por un algoritmo de estas características, aunque se desconoce el tiempo de entrenamiento dedicado para conseguir estos resultados.

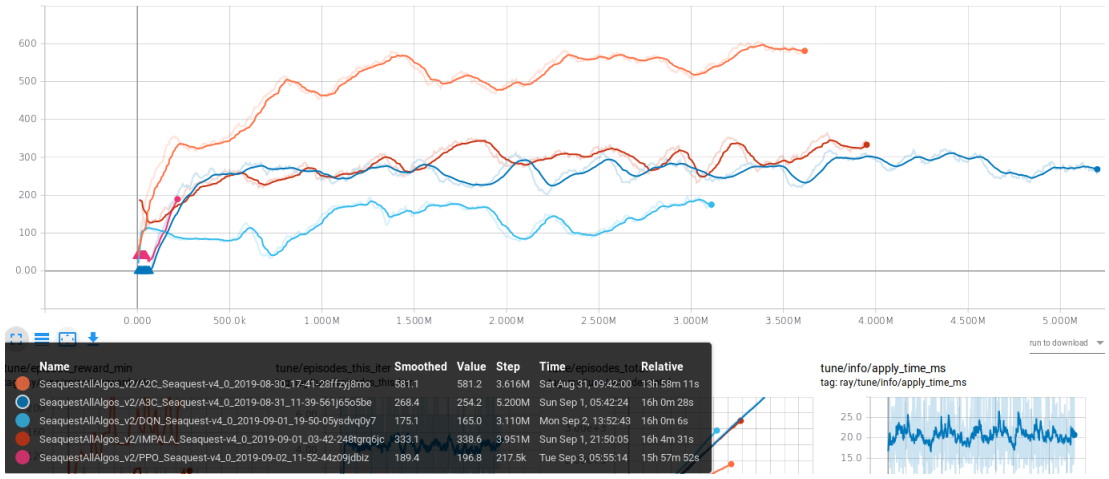


Figura 7.10: Resultados en función de la puntuación media obtenida en Seaquest-v4.

7.1.7. Conclusiones

Los resultados obtenidos en nuestras experimentaciones nos pueden ayudar a sacar ciertas conclusiones sobre la calidad y el comportamiento general de los distintos algoritmos seleccionados:

En primer lugar, podemos considerar que el algoritmo DQN no es capaz de conseguir resultados comparables al resto de algoritmos a largo plazo. Factores como la multiplicidad de los procesos que recogen experiencias o la asincronía tienen una relación importante con la cantidad de experiencia que un algoritmo procesa o la velocidad de actualización de las redes internas. El hecho de que DQN se trate de un algoritmo con un solo proceso limita el número de experiencias que se pueden ejecutar en un tiempo determinado, lo cual le proporciona una clara desventaja respecto a sus competidores.

Por otra parte, los algoritmos A2C y A3C son un tema central de discusión. A pesar de la aparente ventaja de A3C debida a su asincronía, nuestros resultados muestran que el algoritmo A2C consigue en la mayoría de casos puntuaciones mayores que su versión asíncrona, mientra que en otros como Boxing o Breakout consigue unos resultados similares o inferiores a esta. Estos resultados nos llevan a teorizar que la asincronía no aporta un beneficio real aplicado a este tipo de problemas, o que bien sería necesaria una modificación del algoritmo para utilizar de manera más eficiente este sobreexceso de información. No obstante, esto de nuevo podría crear problemas de optimización, al crear una mayor carga sobre el proceso *learner*.

Esta versión de IMPALA se muestra como una de los mejores algoritmos en función a la cantidad de experiencias acumuladas por unidad de tiempo, y también como el algoritmo más consistente y con unos mejores resultados globales, habiendo sido capaz de conseguir la mejor puntuación en una gran mayoría de entornos.

En general, el algoritmo PPO ha resultado demasiado lento y por tanto su ejecución se ha eliminado de la mayoría de experimentos.

Por último, con respecto a las estrategias evolutivas añadir que a pesar de no haber sido posible superponer los resultados al resto de algoritmos, se trata posiblemente del algoritmo que más iteraciones es capaz de realizar por unidad de tiempo, alcanzando los valores de 75 millones de iteraciones en un período de 20 horas.

7.1.8. Problemas encontrados durante la ejecución

Durante las experimentaciones realizadas, nos hemos encontrado con una serie de problemas que de una manera u otra nos han dificultado la recolección de resultados. A continuación, heremos un repaso de algunos de los principales impedimentos a los que nos hemos enfrentado.

Conflictos de versiones entre librerías

Problema: Nos hemos encontrado en algunas situaciones con diversos conflictos entre versiones de las librerías que hemos tenido que utilizar. Eventualmente, estos problemas se podían traducir en una imposibilidad de ejecutar nuestras soluciones.

Solución: Reinstalar versiones antiguas de las librerías implicadas (principalmente Ray y Tensorflow), modificar internamente la librería para soportar una funcionalidad desfasada o reintegrar el código a las nuevas versiones.

Sobrecarga de servidores

Problema: El grupo de Inteligencia Artificial amáblemente nos dotó acceso a algunos servidores utilizados para la investigación. Estos servidores eran frecuentemente utilizados por distintos investigadores del grupo u otros alumnos realizando pruebas, por lo que en ciertas ocasiones la memoria RAM se sobrecargaba. Esto causó en un gran número de ocasiones que nuestras experimentaciones se vieran interrumpidas pocas horas después de haberlas iniciado, lo cual dificultó en gran medida la recolección de resultados.

Solución: La mayoría de las experimentaciones mostradas en este documento fueron realizadas durante los meses de julio y agosto fuera del calendario académico, tiempo durante el cual tuvimos acceso sin restricciones a ámbos servidores para realizar nuestras experimentaciones. Esto causó una reducción significativa de las interrupciones causadas por fallos de memoria u otros factores similares, permitiéndonos realizar una recolección de datos relativamente extensa.

Durante el curso, otra de las soluciones probadas fue reducir el tamaño de memoria asignado a nuestros agentes. No obstante, esta solución no impidió que la ejecución se viera interrumpida en diversas ocasiones.

7.2 Caso DOOM

En esta sección, describiremos el proceso de experimentación realizado sobre nuestra propia implementación del algoritmo DRQN.

En nuestras pruebas iniciales con el algoritmo pudimos observar un notable descenso de la velocidad global de ejecución una vez la memoria hubiera recogido una cantidad considerable de experiencias, que se traducían en un acceso a memoria mucho más lento y un mayor tiempo de actualización de las redes neuronales. Es por ello que tomamos ciertas medidas para aplacar esto, como la introducción del *frame skipping* o un aumento al número de pasos entre cada actualización de la red.

Las restricciones de tiempo impuestas por nuestro servidor se tradujeron en una muestra reducida del número de episodios sobre los que se recogieron experiencias. No obstante, se considera que la muestra es suficiente para verificar el funcionamiento de nuestro algoritmo y extraer una serie de conclusiones iniciales.

En primer lugar, los resultados sobre DoomBasic-v0 fueron llevados a cabo en un período de tiempo de aproximadamente 26 horas, con un número de 475 episodios realizados. Durante este período de entrenamiento, el agente es capaz de aprender la estrategia de localizar al enemigo y dispararle, pero todavía no ha recibido el suficiente entrenamiento para realizarlo en todas las ocasiones ni para aprender que es imperativo localizar y disparar al enemigo cuanto antes, debido a la elevada penalización del *living reward* que en muchas ocasiones reduce considerablemente la recompensa recibida a pesar de la eliminación del enemigo.

En la figura 7.11 se observa la recompensa media obtenida de aproximadamente -200. Esto indica como se ha mencionado que el agente ha aprendido a eliminar el enemigo, pero no con la frecuencia necesaria como para que la puntuación mínima de -365 no influya notablemente en este factor. Como se puede observar también, el número total de pasos realizados por el agente de aprendizaje es ligeramente superior a 100000, lo cual no se considera un entrenamiento suficiente para resolver el entorno.

Como muestra adicional, se decidió recopilar otra serie de datos sobre el escenario *VizdoomDefendCenter-v0* ejecutado durante un período de 24 horas para observar el ren-

```
TOTAL REWARD: -325.0

471 training episodes played.
Results: mean: -204.0±165.4, min: -365.0, max: 95.0,
TOTAL STEPS: 103416

Epoch 472
-------
```

Figura 7.11: Resultados de la experimentación sobre DoomBasic-v0.

dimiento sobre un tipo de entorno con distintas reglas. En este caso, la ejecución resultó en un total de 332 episodios ejecutados con una recompensa media de 0.7 habiendo llegado a una máxima de 4, es decir, existiría al menos un episodio en el que el agente ha sido capaz de eliminar a cinco enemigos antes de ser derrotado. Esto de nuevo es indicativo de un aprendizaje efectivo por parte de nuestro algoritmo, si bien todavía no ha sido capaz de alcanzar puntuaciones mayores ni de aumentar significativamente su puntuación media, cosa que podría ocurrir fácilmente eliminando las restricciones temporales con las que contábamos.

```
331 training episodes played.
Results: mean: 0.7±1.1, min: -1.0, max: 4.0,
TOTAL STEPS: 113571

Epoch 332
-------
```

Figura 7.12: Resultados relativos al entorno *DoomDefendCenter-v0*.

7.2.1. Conclusiones

Hemos podido verificar el funcionamiento de nuestro algoritmo sobre el entorno ViZDoom observando que nuestra red es capaz de tomar las decisiones necesarias para obtener puntuaciones positivas. A pesar de las restricciones de tiempo impuestas, nuestras ejecuciones han reflejado un nivel de aprendizaje favorable que es indicativo de que la solución sería capaz de resolver una variedad de escenarios dado el tiempo de entrenamiento necesario.

En las siguientes figuras se muestran las trazas del aprendizaje durante nuestras experimentaciones, en función del número de iteraciones y la puntuación media obtenida.

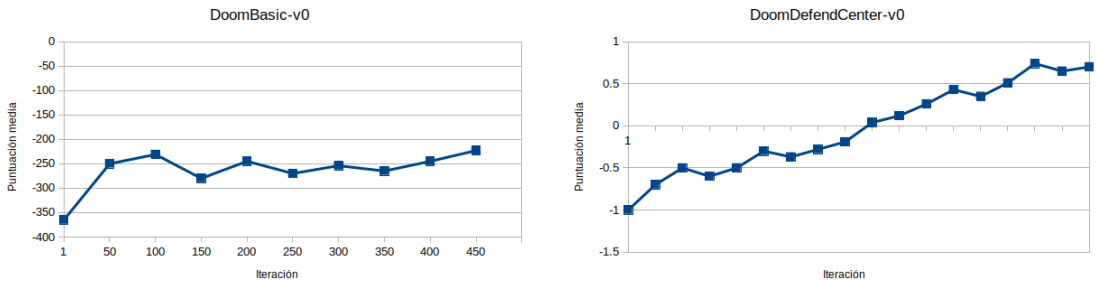


Figura 7.13: Gráficas de resultados. A la izquierda, resultados relativos a *DoomBasic-v0*. A la derecha, resultados de *DoomDefendCenter-v0*.

CAPÍTULO 8
Conclusiones

8.1 Discusión del trabajo realizado

8.1.1. Justificación de las técnicas aplicadas

Durante nuestro trabajo, nuestro principal objetivo ha consistido en poner a prueba distintas técnicas existentes en el campo del aprendizaje por refuerzo para contrastar su funcionamiento. Es por ello que en un primer momento seleccionamos una variedad de algoritmos con características distintivas para así intentar comprobar hasta qué punto existen diferencias en su rendimiento.

8.1.2. ATARI - Resumen de las pruebas y resultados

Con la intención de comprobar el funcionamiento de los distintos algoritmos, diseñamos una serie de experimentaciones sobre una selección de entornos con distintas estrategias de resolución.

Aún lejos de ser unos resultados concluyentes, se ha podido observar en la mayoría de experimentos un claro dominio de los algoritmos IMPALA y A3C sobre el resto, siendo el primero ligeramente superior en rendimiento. A su vez, DQN ha resultado en todos los casos el algoritmo a la vez más ineficiente y con una menor tendencia al aprendizaje sobre el entorno. Esto podría haber sido causado por ciertas discrepancias en la configuración interna del algoritmo. No obstante, las diferencias respecto al resto son tan significativas que sugieren que los algoritmos asíncronos aportan mejoras considerables en el campo del aprendizaje por refuerzo.

8.1.3. DOOM - Conclusiones

Se ha conseguido implementar con éxito una red neuronal basada en el algoritmo DRQN y realizar una serie de pruebas sobre distintos escenarios que verifican el funcionamiento del mismo

Los resultados de nuestras experimentaciones demuestran que se ha conseguido implementar con éxito un algoritmo de aprendizaje por refuerzo capaz de resolver entornos de observación parcial como en nuestro caso el entorno DOOM. Consideramos que la solución implementada sería capaz de resolver una gran variedad de entornos de este tipo eliminando las restricciones de tiempo encontradas.

8.2 Relación con los estudios cursados

Para la realización de este proyecto han sido necesarios los conocimientos adquiridos en el Grado de Ingeniería Informática ya sea en asignaturas obligatorias como las optativas de la rama de Computación. Para poder identificar cuáles han sido, a continuación se listan las materias y en qué asignaturas se han impartido:

- Lenguaje de programación Python, enseñado en las asignaturas de Algorítmica y Sistemas de almacenamiento y recuperación de información.
- Conocimientos estadísticos necesarios a la hora de analizar los argumentos, asignatura Estadística.
- Conocimientos de algoritmos de búsqueda con el objetivo de que el tiempo de respuesta fuera mínimo, asignatura Estructuras de datos y algoritmos y Sistemas de almacenamiento y recuperación de la información.
- Conocimientos de comunicación en sistemas multiagente para el envío de información llevado a cabo por nuestros algoritmos, asignatura Agentes inteligentes.
- Conocimientos de redes neuronales e introducción al ámbito de la Inteligencia artificial, asignaturas Sistemas inteligentes, Aprendizaje automático y Técnicas de la Inteligencia Artificial.
- Conocimientos de cómo gestionar un proyecto especialmente en la especificación de requisitos, asignatura Gestión de proyectos.

A pesar de estos conocimientos previos requeridos, el desarrollo del proyecto ha requerido adicionalmente una formación complementaria más detallada en los ámbitos del aprendizaje por refuerzo.

Como se ha mostrado, para la realización de este proyecto han sido necesarios los conocimientos adquiridos en diversas asignaturas del grado. En conclusión, gracias a la materia aprendida se ha podido dar solución a un problema real.

Para concluir esta sección, a continuación se indican las competencias transversales que se han puesto en práctica en la elaboración de dicho proyecto.

- CT-01. Comprensión e integración: la combinación de la materia de diferentes asignaturas ha sido necesario para la realización del TFG.
- CT-02. Aplicación y pensamiento práctico: para la realización del TFG ha sido necesario aplicar los conocimientos teóricos enseñados en diversas asignaturas para poder diseñar y crear una solución y, posteriormente, analizar e interpretar los datos para extraer conclusiones.
- CT-03. Análisis y resolución de problemas: cómo se puede observar en los capítulos Análisis del problema y Diseño de la solución esta competencia se ha puesto en práctica en el TFG.
- CT-05. Diseño y proyecto: el TFG se centra en el diseño y el desarrollo de un proyecto.
- CT-07. Responsabilidad ética, medioambiental y profesional: la responsabilidad del almacenamiento, uso y tratamiento de los datos cae sobre el desarrollador y el usuario administrador.

- CT-10. Conocimiento de problemas contemporáneos.
- CT-11. Aprendizaje permanente: como se ha comentado anteriormente, los conocimientos de desarrollo de sistemas de aprendizaje automático se han obtenido de forma autónoma.
- CT-12. Planificación y gestión del tiempo: con el fin de poder realizar el proyecto en el plazo esperado ha sido necesario una correcta planificación y gestión de tiempo.

8.3 Trabajo futuro

Para ahondar más en el funcionamiento interno de las redes neuronales recurrentes, se considera aplicar en un futuro el algoritmo DRQN a otro tipo de entornos. Principalmente sería de interés aplicarlo al entorno disponible en Gym "Monctezuma's Revenge", dado que las acciones tomadas tienen consecuencias a largo plazo y hasta el momento, el estudio sobre la aplicación de redes neuronales tradicionales no ha resultado fructífero con respecto a este entorno en particular.

Adicionalmente, se propone un estudio adicional sobre el entorno DOOM para probar el funcionamiento de otros algoritmos descritos como A2C o IMPALA, dado que teóricamente sus estructuras internas no son adecuadas para la resolución de entornos de observación parcial.

8.3.1. Agradecimientos

Como punto final de este trabajo, me gustaría agradecer a aquellas personas que han colaborado de una manera u otra en la redacción de este TFG. En primer lugar, agradecer enormemente a mis tutores Vicent Botti y, en especial, a Javi Palanca por la ayuda prestada durante todo el proceso de concepción de este trabajo, y al grupo de Inteligencia Artificial de la UPV por proporcionar los servidores sobre los que se han realizado las pruebas.

En una nota más personal, agradecer también a mi madre y hermano por su apoyo incondicional en los buenos y malos momentos. También a mis amigos, Guillermo, Iván, Carlos e Irene por su asistencia ejerciendo de *beta readers* del documento y por su apoyo en los momentos en los que los objetivos a cumplir parecían imposibles.

Por último, mandar un saludo a los chavalitos del podcast *Yo Interneto* por la compañía durante las interminables jornadas de realización de este trabajo.

Bibliografía

[1] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Tim Harley, Timothy P. Lillicrap, David Silver, Koray Kavukcuoglu. Asynchronous Methods for Deep Reinforcement Learning, febrero de 2016. *International conference on machine learning 1928-1937 (2016)*

[2] Michel Tokic Adaptive ε-greedy Exploration in Reinforcement Learning Based on Value Differences. *Annual Conference on Artificial Intelligence, pp. 203-210. Springer, Berlin, Heidelberg, 2010.*

[3] Jianxin Wu Introduction to Convolutional Neural Networks, mayo de 2017. *National Key Lab for Novel Software Technology. Nanjing University. China 5 (2017)*

[4] Tom Schaul, John Quan, Ioannis Antonoglou, David Silver Prioritized Experience Replay, febrero de 2016. *arXiv preprint arXiv:1511.05952 (2015).*

[5] Wolfram Schultz, Peter Dayan, P. Read Montague A Neural Substrate of Prediction and Reward Science 275, 1593-1599 (1997)

[6] Varios autores Human-level control through deep reinforcement learning *Nature 518, no. 7540 (2015): 529*

[7] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin Riedmiller Playing Atari with Deep Reinforcement Learning. *arXiv preprint arXiv:1312.5602 (2013).*

[8] Lesse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Volodymyr Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, Shane Legg, Koyah Kavukcuoglu IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures *arXiv preprint arXiv:1802.01561.*

[9] Timothy P. Lillicrap, Jonathan H. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, Daan Wierstra Continuous Control With Deep Reinforcement Learning *arXiv preprint arXiv:1509.02971.*

[10] John Schulman, Filip Woski, Prafulla Dhariwal, Alec Radford, Oleg Klimov Proximal Policy Optimization Algorithms OpenAI

[11] Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, Ilya Sutskever Evolution Strategies as a Scalable Alternative to Reinforcement Learning arXiv preprint arXiv:1703.03864 (2017).

[12] Leslie Pack, Michael L. Littman, Andrew Moore. Reinforcement Learning: A Survey Journal of Artificial Intelligence Research 4 (1996) 237-285

[13] ViZDoom Competitions: Playing Doom from Pixels, Wydmuch, Marek and Kempka, Michał and Jaśkowski, Wojciech, IEEE Transactions on Games (2018)

[14] Deep Learning Essentials Jianing Wei, Anurag Bhardwaj, Wei Di Packt Publishing, 2018

[15] ViZDoom: DRQN with Prioritized Experience Replay, Double-Q Learning & Snapshot Ensembling Christopher Schulze, Marcus Schulze Proceedings of SAI Intelligent Systems Conference (pp. 1-17). Springer, Cham.

[16] Deep Recurrent Q-Learning for Partially Observable MDPs Matthew Hausknecht and Peter Stone 2015 AAAI Fall Symposium Series. 2015.

[17] Long-Short term memory Sepp Hochreiter, Jürgen Schmidhuber Neural computation 9(8): 1735-1780, 1997