



UNIVERSIDAD NACIONAL DE EDUCACIÓN A DISTANCIA

UNIVERSIDAD COMPLUTENSE DE MADRID

Máster en Ingeniería de Sistemas y Control

Proyecto Fin de Máster

Runn3r

Resolución Inteligente de Laberintos en Arduino

Autor: FRANCISCO MANUEL GARCÍA ÁLVAREZ

Directora: DRA. MATILDE SANTOS PEÑAS

2020/2021

Máster en Ingeniería de Sistemas y Control

Proyecto Fin de Máster

Runn3r

Resolución Inteligente de Laberintos en Arduino

Autor: FRANCISCO MANUEL GARCÍA ÁLVAREZ

Directora: DRA. MATILDE SANTOS PEÑAS

AUTORIZACIÓN

Autorizo a la Universidad Nacional de Educación a Distancia (UNED) y a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la memoria de este Trabajo Fin de Máster, como el código, la documentación y/o el prototipo desarrollado.

Firmado: Francisco Manuel García Álvarez

Resumen

En el presente trabajo se diseñará un **robot autónomo móvil** de **coste contenido** que, haciendo uso de **laberintos** creados para desarrollar, probar y perfeccionar su funcionamiento, servirá de base para crear **estrategias de navegación** más avanzadas, adaptables e **inteligentes**.

En su primera parte, se introducirá este **trabajo**, su motivación, los objetivos perseguidos, la metodología planteada y la estructura seguida por el propio trabajo.

Tras ello, se realizará un estudio de algunas de las **tecnologías** y **desarrollos** más relevantes en la actualidad en el campo de los ***micromouses***. Se **estudiarán** y **trabajarán** sus diseños, características, arquitectura, electrónica, sensores, estrategias de navegación que emplean y componentes de los que hacen uso, incidiendo sobre las soluciones adoptadas por aquellos más **innovadores** y **eficientes**.

Una vez analizadas, se mostrará una propuesta ***open source*** y de **coste asequible**, centrándonos en sus componentes y diseño. Dicha propuesta servirá de base para el siguiente apartado, en el que se revisarán algunos de los **algoritmos** y **técnicas** de resolución de **laberintos** más utilizados. Se tendrá en cuenta la inteligencia mostrada, su eficacia y las posibles optimizaciones, **seleccionando** aquellas que mejor encajen en nuestra propuesta.

Los algoritmos y técnicas seleccionadas y el diseño del robot inicial serán **probados** en un laberinto diseñado **específicamente** para ello. Sus **resultados** serán analizados, mostrando sus virtudes y defectos para, posteriormente, proponer otra **alternativa** más capaz.

Así, gracias a las conclusiones obtenidas, se desarrollará una opción **más adaptable** e **inteligente**, sustentada por una versión **evolucionada** de la plataforma inicial, que combina y hace **realidad** en este tipo de diseños algunas de las técnicas más **avanzadas**, como el **Aprendizaje por Refuerzo** y las **Redes Neuronales**, presentando **resultados** y **conclusiones** sobre su capacidad de adaptación y rendimiento.

Por último, y tras analizar todo el trabajo desarrollado, se plantearán algunas líneas de **investigación** y **actuación** que podrían resultar muy interesantes en el **futuro**, ya que es un campo en permanente **evolución** y que suscita un gran **interés** por parte de profesionales y desarrolladores de todo el mundo.

Palabras Clave

Robot, *micromouse*, laberintos, navegación, Arduino, Aprendizaje por Refuerzo, Redes Neuronales

Índice de contenidos

1. Runn3R: Resolución Inteligente de Laberintos en Arduino 21

1.1. Introducción 22

1.2. Motivación 23

1.3. Objetivos 24

1.4. Metodología 24

1.5. Estructura del trabajo 24

1.6. Publicaciones relacionadas 25

2. Estado del Arte 26

2.1. Artículos académicos 27

2.1.1. Path Tracking Control of Obstacle Avoiding Robot Using Ultrasonic Sensor 27

2.1.2. Path Finding and Turning with Maze Solving Robot 27

2.1.3. Wheeled Mobile Robot Obstacle Avoidance Using Compass and Ultrasonic 28

2.1.4. Implementation of Q-Learning Algorithm for Solving Maze Problem 28

2.1.5. Human-level Control Through Deep Reinforcement Learning 28

2.1.6. Application of Deep Reinforcement Learning in Mobile Robot Path Planning 29

2.2. Proyectos 29

2.2.1. aMAZEing robot: A method for automatic maze solving 29

2.2.2. Autonomous Maze Solving Robot 30

2.2.3. Design and Implementation of a Robot for Maze-Solving using Flood-Fill Algorithm .. 30

2.2.4. The project of an improved omnidirectional modular robot for the RoboCupJunior Rescue Maze 2017 31

2.2.5. Autonomously Solving Mazes with Robots 31

2.2.6. Design and Development of Intelligent Navigation Control Systems for Autonomous Robots Using Neural Networks 32

2.3. Observaciones 33

3. Análisis del Problema 37

3.1. Requisitos 38

3.2. Propuesta 39
4. El Robot: 41
Componentes y Montaje 41
4.1. El robot 42
4.1.1. Arduino Uno Rev3 43
4.1.2. Sensor ultrasónico HC-SR04 45
4.1.3. Servomotor SG-90 47
4.1.4. Sensor de infrarrojos LED TCRT5000L 47
4.1.5. Módulo L298N 48
4.1.6. Motores DC 49
4.1.7. Baterías Li-Ion 18650 50
4.1.8. Mando IR 51
4.2. Diagrama de bloques y montaje 51
5. Algoritmos y Laberintos 53
5.1. Algoritmos sin conocimiento previo 54
5.1.1. Aleatorio o del "ratón" 54
5.1.2. Seguidor de paredes *(Wall-Follower)* 55
5.1.3. Pledge 56
5.1.4. Trémaux 57
5.2. Algoritmos con conocimiento previo 58
5.2.1. Inundación *(Flood-Fill)* 58
5.2.2. Seguidor de paredes optimizado *(Optimized Wall-Follower)* 58
5.3. Selección de algoritmos laberínticos 59
5.4. Los laberintos 60
5.4.1. Virtual+Estándar 60
5.4.2. Real+Estándar 60
5.5. Implementación de los algoritmos 62
5.5.1. Entorno simulado 62

5.5.2. Entorno real 63
6. Algoritmos Laberínticos: Resultados 67
6.1. Pruebas iniciales 68
6.1.1. Entorno simulado 68
6.1.2. Entorno real 68
6.2. Resultados 69
6.2.1. Laberinto Virtual+Estándar 69
6.2.2. Laberinto Real+Estándar 71
6.3. Análisis de los resultados 74
6.4. Mejoras propuestas 76
6.4.1. Arduino Mega 2560 Rev.3 77
6.4.2. *Encoder* óptico FC-03 79
6.4.3. Pantalla OLED SH1106 80
6.4.4. Inventario de mejoras 82
7. Aprendizaje por Refuerzo, Laberintos y Arduino 83
7.1. Aprendizaje por Refuerzo *(Reinforcement Learning)* 84
7.1.1. Elementos 84
7.1.2. Objetivo 85
7.1.3. El algoritmo general 85
7.2. El Aprendizaje Profundo *(Deep Learning)* 86
7.2.1. Redes Neuronales Artificiales 86
7.3. Objetivo: El laberinto 87
7.4. Aprendizaje por Diferencia Temporal 89
7.4.1. SARSA *(State-Action-Reward-State'-Action')* 91
7.4.2. Q-Learning 93
7.4.3. SARSA vs Q-Learning 93
7.5. Trazas de Elegibilidad *(Eligibility Traces)* 95
7.5.1. SARSA(λ) 98

7.5.2. Q(λ) 99
7.6. *Deep Q-Learning* 100
7.7. En los límites de Arduino 103
8. Aprendizaje por Refuerzo: Resultados 107
8.1. Laberintos 108
8.1.1. Virtual+Estándar 108
8.1.2. Real+Estándar 109
8.1.3. Virtual+Desafiante 109
8.2. Implementación de los algoritmos 111
8.2.1. Entorno simulado 111
8.2.2. Entorno real 113
8.3. Pruebas iniciales 114
8.3.1. Entorno simulado 114
8.3.2. Entorno real 115
8.4. Resultados 116
8.4.1. Laberinto Virtual+Estándar 117
8.4.2. Laberinto Real+Estándar 122
8.4.3. Laberinto Virtual+Desafiante 124
8.5. Análisis de los resultados 127
8.5.1. Laberinto Virtual+Estándar 128
8.5.2. Laberinto Real+Estándar 129
8.5.3. Laberinto Virtual+Desafiante 130
8.5.4. Algoritmos Laberínticos vs Aprendizaje por Refuerzo 131
9. Conclusiones y Trabajo Futuro 137
9.1. Conclusiones 138
9.2. Trabajo Futuro 139
Referencias Bibliográficas 141
Glosario de Términos 147

Anexo A: Montaje del Elegoo Smart Robot Car V3 151

1) Instalación de los motores 152

2) Instalación del módulo L298N 153

3) Instalación de los sensores de líneas 153

4) Instalación de la Arduino y la placa de extensión 154

5) Instalación del receptor IR y del compartimento de baterías 154

6) Instalación del servo y del receptor de ultrasonidos 155

7) Conexión del cableado de los motores y preparación de las dos planchas MDF 155

8) Conexión de los sensores a la Arduino 156

9) Conexión del receptor Bluetooth 156

10) Ensamblaje de las ruedas 157

Y el resultado final es… 157

Anexo B: Tabla de Conexiones para el Elegoo Smart Robot Car V3 159

Anexo C: Instalación del Entorno de Aprendizaje por Refuerzo 163

VirtualBox 6.1: Instalación 165

VirtualBox 6.1: Cómo crear una máquina virtual 165

Lubuntu 20: Instalación e inicio del sistema 169

Lubuntu 20: Instalación de las Guest Additions 174

Lubuntu 20: Instalación de matplotlib, numpy, keras y tensorflow con VirtualEnv 175

Índice de Ilustraciones

Ilustración 1. aMAZEing robot 29
Ilustración 2. Autonomous Maze Solving Robot 30
Ilustración 3. Robot for Maze-Solving using Flood-Fill Algorithm 31
Ilustración 4. Improved Omnidirectional Modular Robot 31
Ilustración 5. Autonomously Solving Mazes with Robots 32
Ilustración 6. Autonomous Robots Using Neural Networks 32
Ilustración 7. Elegoo Smart Robot Car Kit v3: Componentes 42
Ilustración 8. Arduino Uno R3: Conexiones 44
Ilustración 9. Modulación PWM en Arduino 45
Ilustración 10. Sensor ultrasónico HC-SR04 45
Ilustración 11. Funcionamiento de un sensor ultrasónico 47
Ilustración 12. Servomotor SG-90 47
Ilustración 13. Sensor de infrarrojos LED TCRT5000L 48
Ilustración 14. Módulo L298N 48
Ilustración 15. Módulo L298N: Conexiones 49
Ilustración 16. Conexionado de los motores 50
Ilustración 17. Baterías 18650 junto a un cargador 50
Ilustración 18. Mando IR (Amazon, 2020) 51
Ilustración 19. Runn3r: Diagrama de bloques 51
Ilustración 20. Runn3r: Vista lateral y frontal 52
Ilustración 21. RMD: Resolución de un laberinto 55
Ilustración 22. Pledge: Resolución de un laberinto 57
Ilustración 23. Trémaux: Reglas y ejemplo de un laberinto 57
Ilustración 24. Inundación y laberinto marcado 58
Ilustración 25. Seguidor de paredes optimizado: Reglas 59
Ilustración 26. Laberinto Virtual+Estándar: Diseño 60
Ilustración 27. Laberinto Real+Estándar: Vista general del diseño de 2x2 celdas 61
Ilustración 28. Laberinto Real+Estándar: Materiales empleados 61
Ilustración 29. Controlador PID: Concepto 64
Ilustración 30. Controlador PID: Valores empleados 65
Ilustración 31. Resultados: Laberinto Virtual+Estándar - Algoritmo del Ratón 69
Ilustración 32. Resultados: Laberinto Virtual+Estándar - Algoritmo Seguidor de Paredes 70
Ilustración 33. Resultados: Laberinto Virtual+Estándar - Alg. Seguidor de Paredes Optimizado 70

Ilustración 34. Resultados: Laberinto Virtual+Estándar - Algoritmo de Inundación 71
Ilustración 35. Resultados: Laberinto Real+Estándar - Algoritmo del Ratón 72
Ilustración 36. Resultados: Laberinto Real+Estándar - Alg. Seguidor de Paredes con RMI 73
Ilustración 37. Resultados: Laberinto Real+Estándar - Al. Seguidor de Paredes Optimizado 73
Ilustración 38. Resultados: Laberinto Real+Estándar - Algoritmo de Inundación 74
Ilustración 39. Arduino mega2560 Rev.3: Conexiones 78
Ilustración 40. Encoder óptico FC-03 y disco ranurado 79
Ilustración 41. Protocolo I2C: Ejemplo 81
Ilustración 42. Pantalla OLED SH1106 en funcionamiento 81
Ilustración 43. Aprendizaje por Refuerzo: Modelo 84
Ilustración 44. Neuronas: Datos (x), pesos (w) y sigmoide 87
Ilustración 45. Red neuronal 87
Ilustración 46. Aprendizaje por Refuerzo en un laberinto 89
Ilustración 47. Diferencia Temporal: Algoritmo general 90
Ilustración 48. Aprendizaje por Refuerzo: Tipos de modelos 90
Ilustración 49. Tabla-Q: Proceso de actualización 91
Ilustración 50. SARSA: Algoritmo 92
Ilustración 51. Política ε-greedy 92
Ilustración 52. Q-Learning: Algoritmo 93
Ilustración 53. Cliff Walking Example 94
Ilustración 54. Trazas de Elegibilidad: Decaimiento 96
Ilustración 55. Algoritmo T(λ) 97
Ilustración 56. Algoritmo T(λ): Retropropagación 97
Ilustración 57. SARSA(λ): Algoritmo 98
Ilustración 58. SARSA vs SARSA(λ) 99
Ilustración 59. Q(λ): Algoritmo 100
Ilustración 60. Google DeepMind 101
Ilustración 61. Arduino Nano 33 BLE Sense 103
Ilustración 62. Laberinto Virtual+Estándar: Diseño 108
Ilustración 63. Laberinto Real+Estándar: Vista general del diseño de 2x2 celdas 109
Ilustración 64. Laberinto Virtual+Desafiante: Diseño 110
Ilustración 65. Mejoras: Nuevos anclajes en el chasis 115
Ilustración 66. Runn3r: Aspecto final, preparado para recorrer el laberinto 116
Ilustración 67. Resultados: Laberinto Virtual+Estándar - SARSA 117

Ilustración 68. Resultados: Laberinto Virtual+Estándar - SARSA(λ) 118
Ilustración 69. Resultados: Laberinto Virtual+Estándar - Q-Learning 119
Ilustración 70. Resultados: Laberinto Virtual+Estándar - Q-Learning(λ) 120
Ilustración 71. Resultados: Laberinto Virtual+Estándar - DQN 121
Ilustración 72. DQN: Información de aprendizaje 122
Ilustración 73. Resultados: Laberinto Real+Estándar - SARSA, SARSA(λ), Q-Learn., Q(λ) y DQN .. 123
Ilustración 74. Resultados: Laberinto Virtual+Desafiante - SARSA 124
Ilustración 75. Resultados: Laberinto Virtual+Desafiante - SARSA(λ) 125
Ilustración 76. Resultados: Laberinto Virtual+Desafiante - Q-Learning 126
Ilustración 77. Resultados: Laberinto Desafiante - Q-Learning(λ) 126
Ilustración 78. Entorno de Aprendizaje: Estadísticas 128
Ilustración 79. Algoritmos Laberínticos: Movimientos empleados en los recorridos óptimos 132
Ilustración 80. Algoritmos Laberínticos: Movimientos desde las entradas hasta la salida 132
Ilustración 81. Elegoo Smart Robot Car v3: Presentación del kit 152
Ilustración 82. Elegoo Smart Robot Car v3: Instalación de los motores 153
Ilustración 83. Elegoo Smart Robot Car v3: Instalación del módulo L298N 153
Ilustración 84. Elegoo Smart Robot Car v3: Instalación de los sensores de líneas 153
Ilustración 85. Elegoo Smart Robot Car v3: Instalación de la Arduino y de la placa de extensión 154
Ilustración 86. Elegoo Smart Robot Car v3: Instalación del receptor IR y baterías 154
Ilustración 87. Elegoo Smart Robot Car v3: Instalación de servo y receptor de ultrasonidos 155
Ilustración 88. Elegoo Smart Robot Car v3: Cableado de motores y preparación de MDF 155
Ilustración 89. Elegoo Smart Robot Car v3: Conexión de los sensores a la Arduino 156
Ilustración 90. Elegoo Smart Robot Car v3: Conexión del receptor Bluetooth 156
Ilustración 91. Elegoo Smart Robot Car v3: Ensamblaje de las ruedas 157
Ilustración 92. Elegoo Smart Robot Car v3: Montaje final 157
Ilustración 93. Máquina virtual: Creación 166
Ilustración 94. Máquina virtual: Ajustes básicos 167
Ilustración 95. Máquina virtual: Disco duro virtual 167
Ilustración 96. Máquina virtual: Creada 168
Ilustración 97. Máquina virtual: Ajuste adicional 1 168
Ilustración 98. Máquina virtual: Ajuste adicional 2 169
Ilustración 99. Máquina virtual: Ajuste adicional 3 169
Ilustración 100. Lubuntu 20: Arranque de la VM 170
Ilustración 101. Lubuntu 20: Idioma de instalación 170

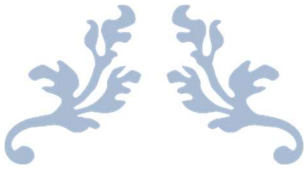
Ilustración 102. Lubuntu 20: Tipo de arranque 171
Ilustración 103. Lubuntu 20: Pantalla principal en el primer arranque 171
Ilustración 104. Lubuntu 20: Pantalla principal en el primer arranque 172
Ilustración 105. Lubuntu 20: Instalador 172
Ilustración 106. Lubuntu 20: Resumen de las opciones de instalación 173
Ilustración 107. Lubuntu 20: Fin de la instalación 173
Ilustración 108. Lubuntu 20: Eliminación del disco de instalación 174
Ilustración 109. Lubuntu 20 – Guest Additions: Insertar CD virtual 174
Ilustración 110. Lubuntu 20 – Guest Additions: Autoarranque 175
Ilustración 111. Lubuntu 20 – Guest Additions: autorun.sh 175
Ilustración 112. Lubuntu 20 – VirtualEnv con SW instalado 176
Ilustración 113. Lubuntu 20 – VirtualEnv con SW instalado 177

Índice de Tablas

Tabla 1. Algoritmos Laberínticos: Resultados medios 75
Tabla 2. Arduino Uno vs Mega2560: Diferencias 79
Tabla 3. Inventario de mejoras 82
Tabla 4. Características de los Modelos de Aprendizaje 111
Tabla 5. Parámetros configurables de los Modelos de Aprendizaje 112
Tabla 6. Algoritmos de Aprendizaje por Refuerzo: Resultados medios 128
Tabla 7. Algoritmos Laberínticos vs Aprendizaje por Refuerzo: Resultados 134
Tabla 8. Conexiones - Elegoo Smart Robot Car v3 161

Índice de Ecuaciones

Ecuación 1. Diferencia Temporal: Actualización de valores de los estados 89
Ecuación 2. SARSA: Regla de actualización 91
Ecuación 3. Q-Learning: Regla de actualización 93
Ecuación 4. Trazas de Elegibilidad 96
Ecuación 5. Trazas de Elegibilidad: Errores por diferencia temporal 96
Ecuación 6. Trazas de Elegibilidad: Generalización de errores 96
Ecuación 7. SARSA(λ): Regla de actualización 98
Ecuación 8. SARSA(λ): Errores por diferencia temporal 98
Ecuación 9. SARSA(λ): Trazas de Elegibilidad 98
Ecuación 10. Q(λ): Trazas de Elegibilidad 100
Ecuación 11. Q(λ): Regla de actualización 100
Ecuación 12. Q(λ): Errores por diferencia temporal 100
Ecuación 13. DQN: Regla de actualización 102
Ecuación 14. DQN: Función de pérdida 103
Ecuación 15. Algoritmos Laberínticos: Cálculo de movimientos empleados 132
Ecuación 16. Algoritmos Laberínticos: Tiempo empleado por movimiento (Arduino) 133
Ecuación 17. Algoritmos Laberínticos: Tiempo empleado por movimiento (PC) 133
Ecuación 18. Algoritmos Laberínticos: Tiempos empleados desde todas las celdas (Arduino) 133
Ecuación 19. Algoritmos Laberínticos: Tiempos empleados desde todas las celdas (PC) 133



1. RUNN3R: RESOLUCIÓN INTELIGENTE DE LABERINTOS EN ARDUINO



1. Runn3r: Resolución Inteligente de Laberintos en Arduino

1.1. Introducción

Un **laberinto** podría verse como una **red de caminos**, que normalmente nos lleva desde una **entrada** hasta una **salida**. El concepto de laberinto fue inventado en Egipto y, desde mediados del siglo XX, es un campo de mucha relevancia en la **robótica** actual (Covaci, Harja, & Nascu, 2020), ya que la resolución de laberintos es una forma de **entrenar y mejorar** la capacidad que poseen los robots autónomos de **desplazarse** por su entorno. Para ello, emplean un algoritmo de "toma de **decisiones**" adaptado a su situación concreta, ya sea por tamaño, ubicación o potencia de proceso, por poner algunos ejemplos (Pfeiffer, Schaeuble, Nieto, Siegwart, & Cadena, 2017).

En este contexto, desde hace ya tiempo tienen gran relevancia los ***micromouses*** (Wyard-Scott & M. Meng, 1995), unos pequeños robots diseñados para encontrar la salida de un laberinto **sin ayuda** y creados para poner **a prueba** la inventiva y creatividad de estudiantes e ingenieros de todo el mundo (Juing-Huei, Chyi-Shyong, & Hsin-Hsiung, 2011), que siempre están buscando la mejor forma de **resolver** los problemas que se van encontrando durante su desarrollo.

Los *micromouses* se componen, esencialmente, de un motor, dirección y giro para moverse (Christiansen, 2014). Tienen **sensores** para detectar paredes y una lógica de **control** para encontrar el posible camino. Cuanto menos tiempo tarden en llegar a la meta, mejor será el **algoritmo** de toma de decisiones. Los primeros eran mucho menos tecnológicos y electrónicamente avanzados que los de hoy, y actualmente los microprocesadores los permiten funcionar y hacer cálculos no concebibles cuando se idearon, hace ya más de treinta años. Además, los motores, sensores, circuitos integrados y otros componentes han mejorado enormemente las posibilidades del diseño de los robots, siendo más **pequeños**, más **rápidos** y más **inteligentes** que sus homólogos de tiempos pasados.

Por otra parte, el **Aprendizaje por Refuerzo** o *Reinforcement Learning* (RL[1], en adelante) es un gran aliado en tareas complejas, como la **navegación**, máxime si es en entornos controlados en cierta forma, como un **laberinto** (Osmanković & Konjicija, 2011). Una de sus grandes ventajas es que el comportamiento de los agentes **no** necesita ser programado explícitamente, sino que pueden definirse recompensas para ciertas acciones o estados específicos. Gracias a ello se puede reducir en gran medida el tiempo de desarrollo, además de permitir un comportamiento mucho más **complejo** que la mayoría de los códigos programados manualmente (Arulkumaran, Deisenroth, Brundage, & Bharath, 2017). El problema es que las recompensas

[1] RL: *Reinforcement Learning*

que deben especificarse no contienen mucha información y generalmente son de naturaleza indirecta, lo que hace que los robots necesiten mucha **experiencia** para cumplir con sus tareas. Y mucha experiencia es **tiempo** de computación y altos **costes** operativos, lo que no es nada deseable.

Pero existe un método relativamente reciente que podría paliar en parte este problema: Emplear **redes neuronales** antes del Aprendizaje por Refuerzo o, lo que es lo mismo, emplear **Aprendizaje por Refuerzo Profundo** o *Deep Reinforcement Learning* (DRL[2], en adelante) (Jing, Huan, Ding, & Minqi, 2017). La gran ventaja de ésto es que el DRL no calcula exhaustivamente todas las posibilidades, sino que puede **aproximarlas**, lo que permitiría agilizar los **cálculos** y la **velocidad** de procesado (Mnih, Silver, Veness, Fidjeland, & Petersen, 2015).

El objetivo de este proyecto será abordar la creación de un ***micromouse*** de **coste asequible** sustentado por una plataforma ***open source***. Se trabajará sobre su **diseño**, la elección de los **componentes** que integrará y, por supuesto, la implementación de las diferentes **estrategias y algoritmos** necesarios para gobernarlo mientras se desplaza. Se presentarán algunas de las más habituales en cuanto a **navegación** por laberintos y se **compararán** con el RL y DRL, para conocer de qué es capaz un robot de este tipo dotado de una inteligencia más **avanzada** y **adaptable**.

Tras ello, presentaré los **resultados** obtenidos, tanto de forma teórica como práctica, sirviendo como una comparación **real** entre las técnicas clásicas de resolución de laberintos y el RL.

Por último, se extraerán **conclusiones** sobre todo el trabajo, tratando de reflexionar sobre cómo ir un paso más allá en posibles líneas de trabajo **futuras**, algo siempre posible y deseable.

1.2. Motivación

Este trabajo pretende, especialmente, estudiar los **algoritmos** de resolución de laberintos que se emplean habitualmente en los *micromouses*, sus ventajas y desventajas, proponiendo posteriormente una alternativa más **avanzada**, **inteligente** y **adaptable**. Para ello, se emplearán y analizarán diversas técnicas, estudiándolas primero en un entorno **virtual** para posteriormente evaluar sus capacidades y posibilidades en un homólogo **real**, con los retos y desafíos que ello implica.

Además, gracias a este tipo de proyectos es posible adquirir habilidades y conocimientos en **múltiples disciplinas**, como informática, electrónica, física, mecánica, etc., una suerte de **aprendizaje "oculto"**, que

[2] DRL: *Deep Reinforcement Learning*

he podido **estudiar** gracias a un trabajo **paralelo** al desarrollo de este TFM (García Álvarez & Santos Peñas, Educational-Oriented Mobile Robot: Hidden Lessons, 2020), más enfocado hacia el ámbito **educativo**, pero de cuyos resultados también se ha visto **beneficiado**.

Por último, este trabajo también servirá para mostrar que, incluso en plataformas **asequibles** y ***open source,*** es posible desarrollar estrategias de **navegación autónoma** realmente **avanzadas**, y que un buen trabajo de **diseño** y **optimización** permitirá **conocer** y **superar** sus aparentes limitaciones.

1.3. Objetivos

Los **objetivos específicos** a alcanzar en este trabajo son los siguientes:

a) Diseño y construcción de un **robot autónomo asequible**, sustentado por una plataforma ***open source*** y equipado con lo necesario para recorrer el laberinto de una forma **eficiente** y **eficaz**.
b) Planteamiento y justificación de posibles **mejoras** en cuanto a **materiales** empleados.
c) Diseño e implementación de los **algoritmos** más **habituales** y **eficientes** en este tipo de entornos.
d) Planteamiento y justificación de algoritmos más **avanzados**, **inteligentes** y **adaptables**, capaces de **superar** las limitaciones de las plataformas de este tipo.
e) Pruebas del **robot** y de todos los **algoritmos**, en entornos **virtuales** y en sus homólogos **reales**.
f) **Resultados** y **conclusiones**.

1.4. Metodología

Dado que es un trabajo en el que hay **muchas** alternativas y posibilidades que analizar, implementar, probar y evaluar, se seguirá una metodología **evolutiva**. Para ello, se darán los siguientes **pasos**:

1. Diseño y construcción del **robot autónomo** en su versión **estándar**.
2. Diseño e implementación de los **algoritmos** laberínticos más **habituales** y **eficientes**.
3. **Prueba** de los algoritmos, de forma **virtual** y **física**, para conocer sus **fortalezas** y **debilidades**.
4. Identificación, justificación e implementación de posibles **mejoras** en los **componentes**.
5. Justificación, diseño e implementación de **algoritmos** más **inteligentes** y **adaptables**.
6. **Prueba** de los algoritmos inteligentes, en el trazado original y en uno **irresoluble** mediante los algoritmos laberínticos vistos con anterioridad.
7. **Resultados** y **conclusiones**.

1.5. Estructura del trabajo

En este trabajo se realiza un estudio de las tecnologías más relevantes en la actualidad en el campo de los ***micromouses***. Se estudiarán y trabajarán sus características, arquitectura, electrónica, sensores y

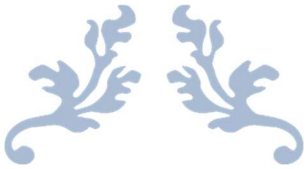
mecanismos de navegación. Además, se revisarán algunos de los **algoritmos** y **técnicas** de resolución de laberintos más utilizados en este ámbito, teniendo en cuenta su inteligencia, eficacia y posibles optimizaciones. Finalmente, se **desarrollará** una opción que, aunque de coste **asequible**, será más **avanzada**, **adaptable** e **inteligente**, y combinará algunas de las técnicas más **punteras** de la actualidad, presentando **resultados** y **conclusiones**.

Puede dividirse en cinco **partes** diferenciadas:

1) **Capítulos 1, 2 y 3:** Presentación del Trabajo, Estado del Arte y Análisis del Problema.
2) **Capítulo 4**: El Robot.
3) **Capítulo 5 y 6**: Algoritmos Laberínticos, Laberintos, Pruebas, Resultados y Posibles Mejoras.
4) **Capítulos 7 y 8**: Algoritmos de Aprendizaje por Refuerzo, Pruebas y Resultados.
5) **Capítulo 9**: Conclusiones y Trabajo Futuro.

1.6. Publicaciones relacionadas

- Francisco Manuel García-Álvarez and Matilde Santos. *Educational-oriented Mobile Robot: Hidden Lessons*. ICEUTE 2020. Advances in Intelligent Systems and Computing, vol. 1266, pp. 61-71, 2021. Springer.
- Francisco M. García-Álvarez, Matilde Santos. *Technology-enhanced learning by experiments with real systems*. 14th International Technology, Education and Development Conference, INTED2020, 272-280, 2020.
- Francisco M. García Álvarez, Matilde Santos Peñas. *Robots Inteligentes como Plataforma de Aprendizaje*. ROBOCITY 2030 Robótica e Inteligencia Artificial, UPM, Madrid, España, 10/12/2019.



2. ESTADO DEL ARTE



2. Estado del Arte

Existen **múltiples** formas de resolver laberintos, dependiendo de las necesidades y posibilidades concretas. Actualmente se suelen emplear técnicas que hacen uso de algoritmos sencillos en cuanto a concepción, pero muy **eficientes** en cuanto a coste computacional / resultados, pero también hay otras que tratan de ir más allá y trabajan con soluciones realmente **innovadoras**, aunque en ocasiones pueden **no** resultar tan eficientes.

Además de los que pasaremos a ver a continuación, existen muchos otros ejemplos que **comparten** o **coinciden** plenamente con el planteamiento de los mostrados. Si se desea obtener información sobre los mismos, puede hacerse uso de la biliografía adjunta, en la que se incluyen unos cuantos ejemplos más.

Finalmente, se **propondrá** una solución que, partiendo de algunas de las ideas de los artículos y proyectos expuestos, sea más **capaz** y **adaptable**, más **inteligente** en definitiva, que las existentes en la actualidad.

2.1. Artículos académicos

2.1.1. Path Tracking Control of Obstacle Avoiding Robot Using Ultrasonic Sensor

El trabajo de (Htike Htike & Min Oo, 2018) propone un **algoritmo** para guiar a un robot móvil con ruedas por una **ruta prefijada**. El principal objetivo de este trabajo es obtener una trayectoria **libre de colisiones** desde el punto de partida hasta el objetivo.

Se adoptan **sensores ultrasónicos** para implementar un sistema de esquiva de obstáculos en tiempo real. El motor se controla mediante un **controlador PID**, que se emplea en el algoritmo ***wall-follower*** para lograr un diseño de ruta optimizado.

Finalmente, los **resultados** experimentales del control de motores y los resultados de la simulación de la trayectoria del robot móvil se muestran gracias al entorno que proporciona Matlab.

2.1.2. Path Finding and Turning with Maze Solving Robot

En el trabajo de (Yee Mon & Nu Nu, 2016) se muestra el desarrollo e implementación de un robot capaz capaz de resolver de **laberintos**. Además, debe **regresar** una y otra vez al punto de partida, tratando de encontrar y asegurarse de que ha encontrado el recorrido **óptimo**.

Diferentes **técnicas** se han utilizado para resolver el problema laberinto por robot, destacando el algoritmo de **Inundación**, que se utiliza como método de búsqueda de la ruta para llegar al objetivo marcado.

2.1.3. Wheeled Mobile Robot Obstacle Avoidance Using Compass and Ultrasonic

El artículo de (Yousef Moh. & Ali, 2018) muestra el uso de una placa **Arduino** para controlar un robot móvil autónomo. El robot utiliza dos **sensores** para navegar y evitar los obstáculos:

1) Una **brújula digital** HMC5883L para detectar la dirección norte y actualizar la situación del robot durante su movimiento.
2) Y un **sensor ultrasónico** para evitar los obstáculos más cercanos en el camino del robot.
3) Aparte de ello, se utiliza **lenguaje C** usado para programar el sistema Arduino.

Los **resultados** obtenidos muestran que el robot pudo navegar y evitar los obstáculos del perímetro.

2.1.4. Implementation of Q-Learning Algorithm for Solving Maze Problem

Los autores (Osmanković & Konjicija, 2011) presentan una implementación de **Q-Learning** para modelos gráficos abstractos de resolución de laberintos, es decir, de forma simulada.

El trabajo consiste en la **conversión** de la matriz de laberinto a la matriz de recompensa de Q-Learning, además de la implementación del **algoritmo** Q-Learning para la matriz de recompensa. Esta implementación se encuentra en un nivel superior de abstracción, por lo que se pueden utilizar **otras** representaciones (redes neuronales artificiales, árbol, etc.).

Para la prueba del algoritmo Q-Learning, el problema de resolución de laberintos, y la **trayectoria** encontrada para el laberinto, se mostró por pantalla gracias al lenguaje de programación Matlab. Además, el **rendimiento** del algoritmo se midió para diferentes escalas del problema.

2.1.5. Human-level Control Through Deep Reinforcement Learning

Si bien los **agentes** de **Aprendizaje por Refuerzo** han logrado algunos éxitos en una variedad de dominios, su aplicación se ha **limitado** a dominios en los que se pueden crear características útiles de forma sencilla, o dominios con espacios de estado de pequeñas dimensiones totalmente controlables.

En el trabajo de (Mnih, Silver, Veness, Fidjeland, & Petersen, 2015) se utilizan algunos de los avances más **recientes** en las Redes Neuronales Profundas para desarrollar un novedoso agente artificial, denominado ***Deep Q-Learning***, que puede **aprender** buenas políticas de múltiples entradas sensoriales.

El agente fue puesto a **prueba** en el retador mundo de los videojuegos clásicos de la Atari 2600. Y se demostró que el agente de la red **DQN**, que recibió sólo los píxeles y la puntuación del juego como entradas,

pudo **superar** el rendimiento de todos los algoritmos anteriores y alcanzar un nivel comparable al de un probador de juegos humano profesional en un conjunto de 49 juegos.

2.1.6. Application of Deep Reinforcement Learning in Mobile Robot Path Planning

Para que un robot **inteligente** sea capaz de calcular la próxima acción óptima a seguir directamente desde de su percepción visual original, sin ninguna modificación ni características añadidas, el estudio de (Jing, Huan, Ding, & Minqi, 2017) propone un método novedoso para planificar una ruta completa: el uso del **Aprendizaje por Refuerzo Profundo**.

Se siguen tres **etapas**:

1) Una *Deep Q-Network* (**DQN**) se diseña y entrena para aproximar la función de valor de los pares estado/acción necesarios.
2) Tras ello, el **valor Q** correspondiente a cada posible acción del robot móvil (es decir, girar a la izquierda, girar a la derecha, adelante) lo determina dicha DQN una vez entrenada. En este caso, la entrada de la DQN es la imagen RGB original (los píxeles de imagen) capturada del entorno del robot, sin añadidos ni modificaciones.
3) Finalmente, la **acción** óptima a seguir se selecciona mediante una estrategia de selecciones de acciones específica. Así, el robot móvil llegará a la meta mientras evita los obstáculos en su camino.

Se realizaron treinta **experimentos** de planificación de ruta en el entorno de pruebas gracias a la plataforma **Google DeepMind**. Los resultados experimentales muestran que este método de planificación de rutas de robots basado en DRL es un método realmente **eficaz**.

2.2. Proyectos

2.2.1. aMAZEing robot: A method for automatic maze solving

El propósito del trabajo de (Ottoson & Renström, 2019) es idear un método para resolver laberintos de forma automática, mediante el robot mostrado en la *Ilustración 1*, capaz de encontrar un **objetivo** específico dentro de un laberinto.



Ilustración 1. aMAZEing robot (Ottoson & Renström, 2019)

Su **funcionamiento** es el siguiente:

- Una placa **Arduino** controlará el robot.
- Una dirección **diferencial** permite giros de radio cero.
- La detección de las **intersecciones** se logra gracias a:

- Un sensor **IR**[3] para la detección de **objetivos**.
- Y un sensor **ultrasónico** para mantener una **distancia** concreta con la pared.

El resultado fue que el robot pudo detectar el tipo intersección y hacer **lo esperado** (según el algoritmo implementado) en un **80-100%** de las ocasiones, y fue capaz de encontrar su **objetivo** el **100%** de las veces.

Ahora bien, hay que tener en cuenta que el **error de rotación** fue de entre 3.5° y -9.5°, y existió un **problema de oscilación** en el movimiento del robot cuando el sensor de distancia izquierdo medía más de 19.5°, lo que hizo al robot menos capaz de seguir un camino y enderezarse correctamente.

2.2.2. Autonomous Maze Solving Robot

En el trabajo de (Rahman, 2017), los desarrolladores presentan un robot de resolución de laberintos (ver *Ilustración 2*).

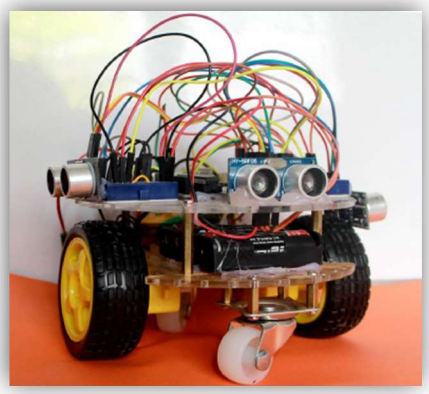


Ilustración 2. Autonomous Maze Solving Robot (Rahman, 2017)

Su **funcionamiento** es el siguiente:

- Para la navegación se aplica el **seguimiento de paredes** basado en el algoritmo **LSRB**[4] y **RSLB**[5].
- El robot obtiene la información que necesita de un sensor **ultrasónico** y uno **infrarrojo**, además de ***encoders*** de rotación situados en los ejes.
- Se **procesa** dicha información mediante una placa **Arduino** integrada en su diseño.

El robot resolvió el laberinto y logró obtener el camino más **corto**, **almacenando** la ruta y sus detalles para utilizarlos en futuras ocasiones y como referencia adicional.

2.2.3. Design and Implementation of a Robot for Maze-Solving using Flood-Fill Algorithm

En la *Ilustración 3* puede observarse el robot ideado por (Elshamarka & Sayuti Saman, 2012), diseñado para resolver un laberinto mediante una **variante** del Algoritmo de **Inundación**.

Su **funcionamiento** es el siguiente:

- Para corregir la **trayectoria** se emplean ***encoders*** y un controlador **PID**[6].

[3] IR: *Infra-Red*
[4] PID: *Proportional–Integral–Derivative*
[5] LSRB: *Left-Straight-Right-Back*
[6] RSLB: *Right-Straight-Left-Back*

- De **detectar** las paredes y la salida del laberinto se ocupan sensores **ultrasónicos**.
- Aparte de ello, para procesar la **lógica** de decisiones necesaria se utiliza una placa **Arduino**.

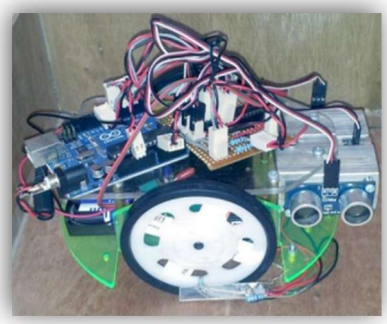


Ilustración 3. Robot for Maze-Solving using Flood-Fill Algorithm (Elshamarka & Sayuti Saman, 2012)

Gracias a la combinación de todo ello, el robot es capaz de **aprender** el laberinto, encontrar las rutas posibles y resolverlo empleando la más **corta**.

2.2.4. The project of an improved omnidirectional modular robot for the RoboCupJunior Rescue Maze 2017

El trabajo de (Siqueira & Rodrigues, 2017) presenta el robot modular **omnidireccional** de la *Ilustración 4*, pensado para participar en la competición RoboCupJunior "Maze Rescue" de 2017.

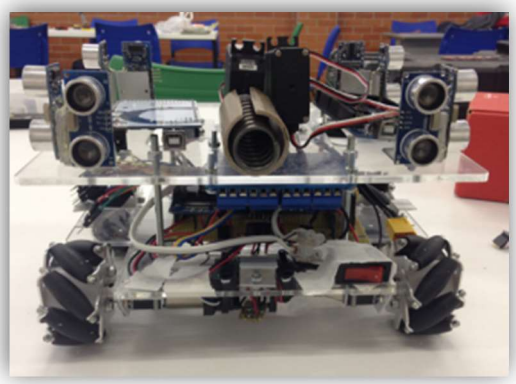


Ilustración 4. Improved Omnidirectional Modular Robot (Siqueira & Rodrigues, 2017)

Su **funcionamiento** es el siguiente:

- El algoritmo para controlar el robot se escribió en C++ y emplea **DFS**[7] y un controlador **PID**.
- Se ejecuta en una placa **Teensy**, que se ocupa de gestionar las lecturas de los sensores de **distancia** y **temperatura**, así como de los **motores**.
- En cuanto a la **alineación** del robot y la detección de **obstáculos**, se emplean 8 sensores de **distancia** Sharp y 4 sensores de **infrarrojos**, dos a cada lado del chasis.
- Para la estructura mecánica del robot se utiliza **MDF**[8] y **aluminio** en un chasis de tipo modular, por lo que es posible hacer cualquier **cambio** necesario fácilmente.
- Tiene 4 ruedas **omnidireccionales** para permitir movimientos en cualquier dirección sin rotar.

Como conclusión, gracias al uso de DFS se visitan **todas** las estancias planteadas.

2.2.5. Autonomously Solving Mazes with Robots

En el proyecto de investigación de (Arya, y otros, 2017), un pequeño robot autónomo (ver *Ilustración 5*) trata de explorar un laberinto empleando un **mapeo experimental** y ciertos algoritmos de **navegación**. Y,

[7] DFS: *Depth First Search*
[8] MDF: *Medium Density Fibreboard*

al hacerlo, se tiene en cuenta un concepto clave al programar y construir estos robots: la **eficiencia** con la que registran obstáculos a su alrededor.

El **funcionamiento** es el siguiente:

- El robot crea y utiliza un mapa de **referencia** para poder guiarse y navegar a través del laberinto.
- Los algoritmos de **mapeo** utilizados son la **exploración por cuadrículas** y la creación de **grafos para entornos desconocidos** (similares a las técnicas BFS[9] o DFS).
- Las opciones de **navegación** son los giros **aleatorios**, la **Regla de la Mano Derecha** (o RSLB), **A*** y el algoritmo de **Dijkstra**.
- Para gestionar todo ello se emplea una placa **Arduino**.

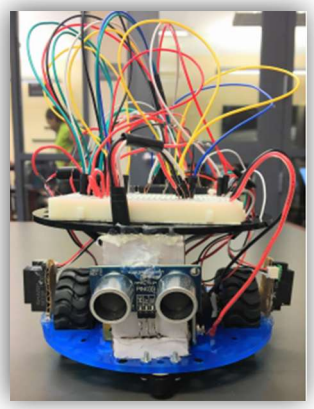


Ilustración 5. Autonomously Solving Mazes with Robots (Arya, y otros, 2017)

Desafortunadamente, las piezas que conformaban el robot eran demasiado **frágiles** para el proceso de prueba completo, lo que resultó en un estudio **no concluyente**.

2.2.6. Design and Development of Intelligent Navigation Control Systems for Autonomous Robots Using Neural Networks

El trabajo de (Viji, 2018) propone una navegación basada en **Redes Neuronales Artificiales** o ANN[10] para robots móviles autónomos inteligentes, como el mostrado en la *Ilustración 6.*

El **funcionamiento esperado** es el siguiente:

- Las **ANN** se ocupan de tareas cognitivas como el **aprendizaje** y la **adaptación**.
- Al diseñar la navegación mediante redes neuronales, la capacidad de aprendizaje de las mismas deberá proporcionar al robot capacidades para navegar con **éxito** en el entorno del laberinto propuesto.
- Además, el robot debe ser capaces de **aprender** durante el proceso de navegación mediante sensores, **actualizarlo** y emplearlo para planificar y controlar la navegación a través de la **retropropagación** de la Red Neural que incorpora el diseño.

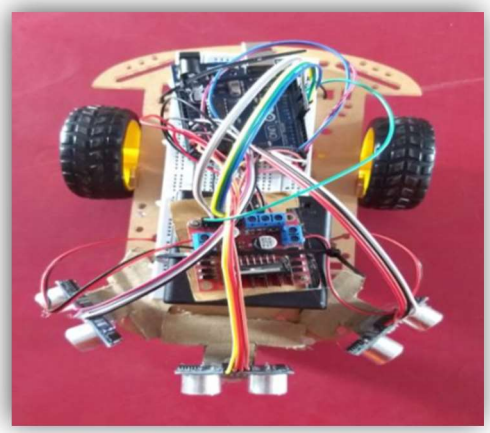


Ilustración 6. Autonomous Robots Using Neural Networks (Viji, 2018)

[9] BFS: *Breadth First Search*
[10] ANN: *Artificial Neural Networks*

El controlador se simuló primero en **Matlab** utilizando Simulink Toolbox, y se utilizó una **Arduino** para implementar el controlador neuronal desarrollado y obtener resultados tangibles.

Los resultados muestran la capacidad que proporciona una red neuronal a los robots móviles autónomos para navegar de manera **inteligente** en varios entornos. Además, el método empleado contribuye positivamente a **reducir** el tiempo requerido para **entrenar** las redes.

2.3. Observaciones

Existen **múltiples** opciones, puntos de vista, perspectivas y soluciones para llegar a lograr que un **robot móvil** sea capaz de navegar de forma **autónoma**. Aunque en lo que concierne a robots de un **coste asequible**, y suficientemente **versátiles** para permitir experimentar con cierta variedad de **algoritmos** de resolución de laberintos, su número se reduce considerablemente. Y si hablamos de combinarlos con técnicas avanzadas de **inteligencia artificial**, las experiencias, publicaciones y estudios en este campo se **reducen** en mayor medida.

Pero, una vez conocido el actual estado del arte, pueden encontrarse ciertos **parecidos** y **diferencias**, aunque éstas últimas más por **concepción** que por necesidad, y un buen **análisis** nos va a permitir extraer ciertas **ideas** que podrían resultar realmente **útiles** de cara a desarrollar el presente trabajo.

Así, pueden extraerse las siguientes **similitudes** :

a) **Algoritmos**: Si es factible, para encontrar la ruta a seguir se emplean algoritmos **sencillos**. En los trabajos de (Htike Htike & Min Oo, 2018), (Ottoson & Renström, 2019) y (Rahman, 2017) se hace uso del algoritmo de la RMI o su variante optimizada, fácil de entender y asequible de implementar.

b) **Q-Learning** y **DQN**: Son algoritmos complejos, pero factibles de programar, como muestran (Osmanković & Konjicija, 2011), (Mnih, Silver, Veness, Fidjeland, & Petersen, 2015) y (Jing, Huan, Ding, & Minqi, 2017). Pero la pregunta que deberemos hacernos es… ¿**es posible** implementarlos en placas de coste **asequible**, con todas las limitaciones que ello conlleva? ¿y qué **herramientas** existen en la actualidad para tal fin, a ser posible ***open source***?

c) **Grafos**: El concepto de grafo también está presente. (Elshamarka & Sayuti Saman, 2012), (Siqueira & Rodrigues, 2017) y (Arya, y otros, 2017) emplean variantes de algoritmos de **profundidad**, como el de inundación, el DFS o el BFS, y todos ellos tienen como base a los grafos.

d) ***Backtracking***: Esta técnica puede tener cierta **complejidad** a la hora de implementarse, pero el resultado que suele obtener es una ruta **óptima**, como muestran (Yee Mon & Nu Nu, 2016), (Elshamarka & Sayuti Saman, 2012), (Siqueira & Rodrigues, 2017) y (Arya, y otros, 2017).

e) **Procesamiento**: Actualmente es posible emplear placas de coste **asequible**, como las Arduino o las Teensy, que deberían ser capaces de obtener buenos resultados si se utilizan adecuamente.

f) **Sensores**: También pueden integrarse sensores de **bajo coste**, pero muy capaces. Así, los *micromouses* actuales pueden integrar arrays de sensores ultrasónicos, infrarrojos o incluso brújulas digitales para aumentar las capacidades y precisión de sus mediciones de forma asequible. De una u otra forma, los trabajos vistos integran múltiples sensores para obtener datos del entorno, como en (Htike Htike & Min Oo, 2018), en (Yee Mon & Nu Nu, 2016) o en (Yousef Moh. & Ali, 2018), por citar algunos.

g) ***Encoders***: Son capaces de obtener datos más precisos sobre el giro de las ruedas, permitiendo conocer la **posición** del robot con mayor detalle, como en los trabajos de (Rahman, 2017) y (Elshamarka & Sayuti Saman, 2012).

h) **PID**: Los controladores de este tipo pueden resultar muy interesantes para **corregir desviaciones** de rumbo (como muestran (Htike Htike & Min Oo, 2018), (Elshamarka & Sayuti Saman, 2012) y (Siqueira & Rodrigues, 2017)), siempre y cuando actúen en conjunción con las mediciones obtenidas por los sensores implicados.

i) **Chasis**: Generalmente, no resultan necesarios materiales muy avanzados para mantener la integridad del robot, porque el **bajo peso** final permite el uso de metacrilato o madera, por ejemplo. De los robots analizados, tan sólo uno (Siqueira & Rodrigues, 2017) emplea algo de aluminio para el chasis. Aún así, conviene conocer bien lo que podría necesitar el robot, para evitar situaciones como la de (Arya, y otros, 2017), en la que no se llegó a completar el estudio por emplear piezas demasiado frágiles.

Aparte de ello, como se ha comentado anteriormente, también existen ciertas **diferencias** e **ideas** originales que resultan interesantes, e incluso quizá **útiles** de aplicar:

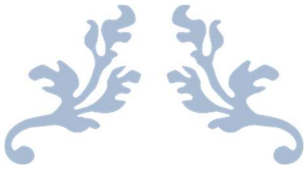
- **Lenguajes de programación**: Suelen emplearse lenguajes habituales y **estándar**, como C, C++ o Matlab, aunque para el caso de las ANN el más empleado es Python, como en (Jing, Huan, Ding, & Minqi, 2017), que emplea Google DeepMind, en el que es el lenguaje predominante (ver (Google DeepMind, 2021)).
- **Separación entre mapeo y navegación**: Es posible separar ambos conceptos para encontrar la ruta óptima (como en (Arya, y otros, 2017)), algo que en algunos casos puede resultar **interesante**.
- **Redes Neuronales**: Como puede verse en (Viji, 2018), la aplicación de esta técnica es **factible**, y sus posiblidades está despertando mucho interés entre los investigadores.

- **Ruedas omnidireccionales**: Un laberinto es un espacio reducido, normalmente. En el trabajo de (Siqueira & Rodrigues, 2017) se ha aprovechado para evitar que el robot necesite **rotar** sobre sí mismo gracias a este tipo de ruedas.

Llegados a este punto, es posible **profundizar** en los diseños y resultados de los trabajos presentados, representativos del estado del arte actual. Así, se pueden hacer las siguientes **observaciones**:

- Es importante tener en cuenta la **placa** que se empleará para gobernar el robot autónomo. Si bien es cierto que la tecnología actual ha mejorado mucho en este sentido, y que ejemplos como la Arduino son muy capaces, conviene conocer sus **limitaciones** en cuanto a velocidad de procesamiento, memoria, etc. Y debe pensarse en ello antes de diseñar el algoritmo que se empleará, porque pueden surgir limitaciones que dificulten su implementación.
- Junto a la CPU[11] empleada, la **memoria** disponible es uno de los factores más limitantes. El uso de mapas optimizados, grafos y la técnica de backtracking pueden ofrecer muy buenos resultados, incluso óptimos, pero el laberinto a resolver **no** podrá ser demasiado **amplio**, porque debe almacenarse en memoria antes de aplicar cualquiera de estas técnicas.
- Es cierto que emplear más **sensores** permite capturar más datos sobre el entorno del robot, pero se debe reflexionar acerca de cuál es el número **realmente** necesario para obtener buenos resultados. Hacer un uso excesivo e innecesario de los mismos se traducirá en un mayor coste, más datos a procesar y mayor peso a integrar en el chasis, lo que implica un aumento del gasto energético y la consecuente reducción de la autonomía del robot.
- Siempre se debe tratar de **posicionar** lo mejor posible el robot en el laberinto. El uso mayoritario de *encoders* o de controladores PID demuestra que siempre hay que tener todo bajo control, porque un pequeño error de navegación puede dar al traste con un algoritmo bien diseñado.
- Las ruedas omnidireccionales podrían resultar útiles, pero su **mayor peso** y **complejidad** es algo que debe tenerse en cuenta. Por ello, un mecanismo de dirección **diferencial** suele resultar más conveniente, y no es un factor limitante para el desplazamiento de un robot por un laberinto.
- Por ejemplo, el algoritmo de la **Regla de la Mano Izquierda no es complejo** de entender ni de implementar y, si es capaz de resolver el laberinto planteado, es muy **eficiente** computacionalmente. Por ello, resultará muy útil conocer qué algoritmos pueden resultar más interesantes en este sentido.
- La **separación** entre mapeo y navegación puede resultar ser muy, muy interesante, porque podrían **superarse** las limitaciones que existen en ambos casos. Si se combina adecuadamente con las técnicas más novedosas, como las ANN o DQN, los resultados podrían ser **sorprendentes**.

[11] CPU: *Central Processing Unit*



3. ANÁLISIS DEL PROBLEMA



3. Análisis del problema

Para lograr crear un robot **autónomo** que se desplace y resuelva **laberintos** complejos, y lograr en última instancia que sea capaz de adaptarse a cualquiera de los propuestos, es necesario dotarlo de cierta "**inteligencia**". En este sentido, serán fundamentales los **algoritmos** que se diseñen para posibilitar que sea capaz de tomar decisiones por sí mismo y que sea realmente autónomo.

Existen **multitud** de algoritmos diferentes para resolver laberintos (Mishra & Bande, 2008), algunos diseñados para ser utilizados **sin conocimiento** previo del propio laberinto y otros empleados cuando el robot sí posee un **conocimiento previo** del laberinto, siendo capaz de resolverlo mucho más rápido o más eficientemente. Además, los laberintos pueden verse como un **árbol** en la **Teoría de Grafos** (Sadik, Dhali, Farid, Rashid, & Syeed, 2010), y es por esta razón por la que muchos algoritmos de resolución de laberintos están estrechamente relacionados con ella. Ahora bien, debemos **seleccionar** aquellos algoritmos que, de alguna forma, puedan implementarse en un *micromouse* **asequible** y ***open source***.

Así pues, se diseñarán e implementarán los algoritmos seleccionados de forma **simulada**, para luego adaptarlos al **robot real**. El traslado a un entorno real supondrá un **reto** considerable, ya que lo que suele funcionar en entornos simulados sin problema, en un entorno real no suele ser tan preciso ni exacto debido al conjunto de pequeñas **imperfecciones** y **errores** de medición que se encuentran.

Todos los datos que obtengamos nos permitirán extraer **resultados** y **conclusiones**, a raíz de las observaciones efectuadas, que serán utilizados posteriormente para crear una versión más **avanzada**, tanto desde el punto de vista de los **algoritmos** como del propio **robot**, capaz de **superar** a las soluciones habituales e ir un paso más allá. El uso del **Aprendizaje por Refuerzo** y las **Redes Neuronales** deberá proporcionarle una mayor capacidad de adaptación a diversos laberintos **complejos**.

En la fase final del trabajo se **analizarán** dichas mejoras respecto del conjunto del proyecto. La **separación** entre el **mapeo** y la propia **navegación** hará posible su implementación en una placa tan sencilla y asequible como **Arduino**, logrando un robot más adaptable, eficiente y, en definitiva, más **inteligente** y **capaz**.

3.1. Requisitos

Se han de tener en cuenta **múltiples** variables para que el proyecto llegue a buen puerto. Pero, desde un punto de vista global, deberá **cumplir** con lo siguiente:

- La plataforma empleada y el entorno de desarrollo serán ***open source***, para garantizar su disponibilidad y el apoyo necesario por parte de la comunidad de desarrolladores.

- Las **piezas** del robot serán **estándar**, fácilmente reemplazables y económicas. Se busca optimizar su desarrollo, mostrando que con poco se puede hacer mucho.
- El **chasis** empleado **no** requerirá de conocimientos, técnicas o herramientas demasiado complejas.
- La **placa base** que integrará el robot deberá ser **programable** en diferentes lenguajes, suficientemente **potente** para manejar los sensores y ser de coste **asequible**.
- Los sensores, motores y ruedas empleados serán lo suficientemente precisos como para lograr una **navegación fluida** por el laberinto.
- Los **algoritmos** que se diseñen en un primer momento deberán **mostrar** las diferentes técnicas y posibilidades existentes en la **actualidad**, logrando que el robot alcance su **objetivo**.
- El laberinto real que se construya deberá ser **idéntico** en concepción al virtual, para poder mostrar las **diferencias** existentes entre ambos en cuanto a adaptaciones y tiempos de forma **directa**.
- Los **resultados** obtenidos en las diferentes pruebas deberán ser **estudiados** desde diferentes perspectivas para poder plantear **mejoras** justificadas y de relevancia.
- La versión final del robot estará equipada con todas las mejoras necesarias para dotarle de una **inteligencia** capaz de **adaptarse** a cualquier laberinto, por complejo que resulte.
- Por último, el resultado **final** deberá ser **innovador** y **diferente** a cualquier otro, haciendo uso de algunas de las técnicas más **avanzadas** existentes en la actualidad.

3.2. Propuesta

Si se analizan todos los requisitos planteados, y los artículos y trabajos que se han presentado como referencia del actual estado del arte, es posible delimitar en gran medida una **solución** que sea capaz de dar respuesta a todas las **necesidades** planteadas a priori para el presente proyecto. Además, se deberá tener presente que este trabajo será **evolutivo** y de tipo ***Plan-Do-Check-Act*** (Srivannaboon, 2009), tratando de aprovechar la capacidad de adaptación y mejora continua que aporta al desarrollo de proyectos, por lo que se diseñará una buena **base** inicial que se **mejorará** y **perfeccionará** conforme avance el mismo.

El proyecto integrará, de partida, los siguientes **componentes** y **tecnologías**:

a) **Chasis**: Será fabricado en **MDF**, porque es suficientemente resistente, manejable y asequible.

b) **Placa base**: Se empleará **Arduino**, *open source* y que goza de una gran comunidad de desarrolladores (Arduino Stack Exchange, 2020), (GitHub, 2020), (Stack Overflow, 2020).

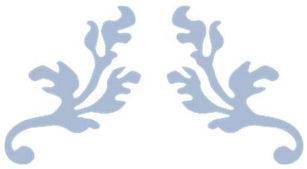
c) **Sensores y dispositivos**:
   - **Ultrasonidos**: Para calcular distancias a objetos y obstáculos.
   - **Infrarrojos**: Añadirán puntos de referencia que ayudarán en la navegación.
   - **Servomotores**: Aportarán versatilidad a los elementos de medición.

d) **Navegación**:
   - **Motores DC**: Son imprescindibles para mover las ruedas y lograr que el robot avance.
   - **Baterías**: Aportarán la energía necesaria para alimentar los componentes electrónicos.

e) **Algoritmos**: Los más habituales son el Aleatorio, el Seguidor de Paredes y el de Inundación, así que serán los elegidos para ser el punto de **referencia** inicial.

f) **Laberintos**: El virtual será simulado en la propia placa **Arduino**, para homogeneizar su desarrollo con el real. Éste último será elaborado con materiales **básicos**, como cartón, cinta aislante o pinzas, facilitando así su diseño y construcción.

Tras tener el diseño a punto, éste será **probado** y **analizado**. Y con las conclusiones que se extraigan, sin duda se descubrirán nuevas **necesidades** y posibles **soluciones**, a las que se deberá dar respuesta en forma de nuevos elementos del **robot** y nuevos diseños de **algoritmos**, más eficientes, adaptables e **inteligentes**.



4. EL ROBOT:

COMPONENTES Y MONTAJE



4. El Robot: Componentes y Montaje

Existe un gran número de **kits comerciales** de robots que integran una placa **Arduino** y todo lo **necesario** para desarrollar casi cualquier proyecto. Así pues, he decidido partir de un buen kit de los disponibles en el mercado e ir haciendo las **modificaciones** oportunas, a medida que vayan siendo necesarias. Esto permitirá **agilizar** su **desarrollo**, **limitar costes** y, por qué no decirlo, tener **mejores acabados** que los que suelen conseguirse de forma artesanal.

Además, se aprovechará la ocasión para saber hasta dónde se puede llegar con un kit comercial bien **programado** y al máximo de sus **posibilidades**. Evidentemente, tendrá sus **limitaciones** y puede que no cubra todos los aspectos que seguramente se necesitarán, pero se irán **superando** con un buen trabajo de ingeniería a medida que vayan surgiendo.

4.1. El robot

Partiré de una **buena base**, capaz y flexible, para poder ir haciendo las **modificaciones** oportunas, si fueran necesarias. El seleccionado es un **kit** basado en **Arduino**, el Elegoo Smart Robot Car Kit v3, cuyos componentes pueden verse con detalle en la *Ilustración 7*.



Ilustración 7. Elegoo Smart Robot Car Kit v3: Componentes (Elegoo, 2019)

Ha sido elegido por las siguientes **razones**:

- **Elegoo** es una compañía que desde 2011 está especializada en el diseño, la producción y la comercialización de productos basados en **Arduino**, y sus kits gozan de muy buena **reputación** entre los desarrolladores de todo el mundo.
- El kit integra una **Arduino UNO**, que puede ser programada y modificada para cambiar el comportamiento del robot en función de los algoritmos requeridos.
- Aporta un buen **tutorial**, con instrucciones de buena calidad, para ensamblar el robot desde cero. Además, se emplean puertos XH2.54, lo que hace facilita su construcción y se reducen los errores.
- Más de 10 tipos de **mejoras** sobre la versión previa, como un sensor remoto IR integrado en la placa, 3 módulos de seguimiento de línea o una batería mejorada con placa de protección.

Entre los **componentes** más destacados para comenzar el proyecto, se encuentran:

a) 1 x placa **Arduino Uno** Rev3.
b) 1 x **sensor ultrasónico** HC-SR04.
c) 1 x **servo motor** SG90.
d) 3 x **sensores** de **infrarrojos** LED[12] TCRT5000L.
e) 1 x **módulo** L298N para controlar los **motores**.
f) 4 x **motores DC**.
g) 2 x **baterías** Li-Ion 18650 de 3700 mAh.

4.1.1. Arduino Uno Rev3

La parte más **importante** de todo el proyecto. Es una **placa** electrónica de las que tiene Arduino disponibles y con la que es sencillo introducirse en el mundo de la programación electrónica (Llamas, 2020). Además, es en conjunto una plataforma de **código abierto** u *open source* (Arduino CC, 2020), lo que permite realizar modificaciones tanto de hardware como de software sin problema, perfecto para el presente proyecto.

Sus **características técnicas** son las siguientes:

- Microcontrolador: ATmega328
- Voltaje de operación: 5V
- Voltaje de entrada (recomendado): 7-12V
- Voltaje de entrada (límites): 6-20V
- Pines de E/S[13] digitales: 14 (de los cuales 6 proporcionan salida PWM)
- Pines de entrada analógica: 6
- Corriente DC[14] por pin de E/S: 40 mA
- Corriente DC para 3.3V Pin: 50 mA
- Memoria Flash: 32 KB[15] de los cuales 0,5 KB son utilizados por el *bootloader*
- SRAM[16]: 2 KB (ATmega328)
- EEPROM[17]: 1 KB (ATmega328)
- Velocidad de reloj: 16 MHz[18]
- Longitud: 68,6mm

[12] LED: *Light-Emitting Diode*
[13] E/S: Entradas/Salidas
[14] DC: *Direct Current*
[15] KB: KiloBytes
[16] SRAM: *Static Random Access Memory*
[17] EEPROM: *Electrically Erasable Programmable Read-Only Memory*
[18] MHz: MegaHertzios

- Anchura: 53,4mm
- Peso: 25 g

Hay que reseñar que la placa puede funcionar con una **alimentación** externa de 6 a 20 voltios. Sin embargo, si se suministra con menos de 7V, la clavija de 5V puede suministrar menos de 5V y la placa puede ser inestable. Si se utilizan más de 12V, el regulador de voltaje puede sobrecalentarse y dañar la placa. Por todo ello, el rango **recomendado** es de **7** a **12** voltios.

En cuanto a las **conexiones** de la placa, claves en nuestro proyecto, son las de la *Ilustración 8*.

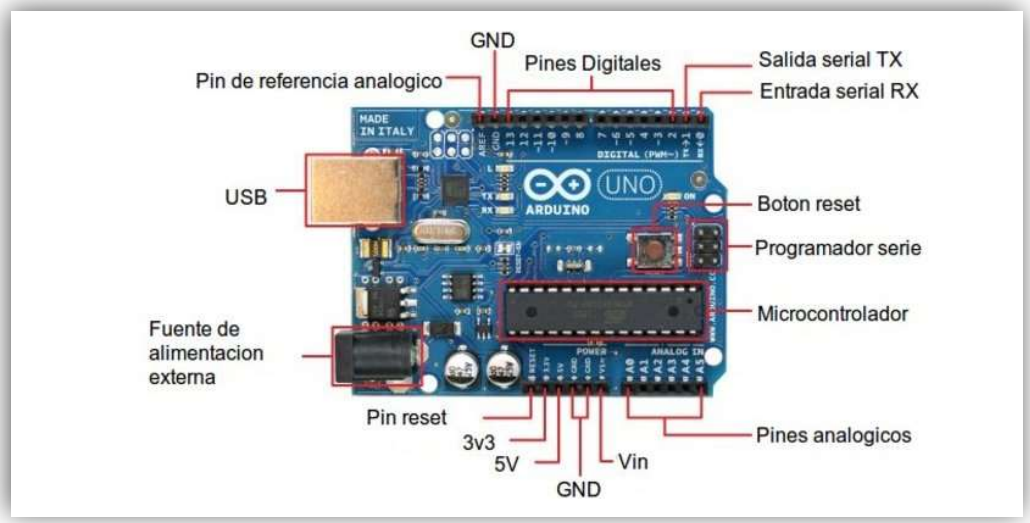


Ilustración 8. Arduino Uno R3: Conexiones (Arduino CC, 2020)

También conviene conocer cuáles son los **pines** de **E/S** que podrían resultar útiles, destacando:

- **Analógicos**: Van desde el pin A0 al A5 y pueden comportarse como entradas.
- **Digitales**: Son 14, desde el pin 0 al 13 y actúan como entradas o salidas.
- **0 RX** y **1 TX**: Son empleados para transmitir datos en serie a la propia placa.

Los **pines** empleados para suministrarle **energía** son los siguientes:

- **VIN:** Suministra el voltaje de entrada a la placa Arduino cuando se está usando una fuente de alimentación externa (a diferencia de los 5V de USB u otra fuente de alimentación regulada).
- **5V**: Esta clavija emite un 5V regulado a la placa, que puede alimentarse desde el conector de alimentación de CC (7-12V), el conector USB (5V) o la clavija VIN de la tarjeta (7-12V). La alimentación de tensión a través de las clavijas de 5V o 3,3V puentea el regulador y puede dañar la placa, lo que no es nada aconsejable.
- **3,3V**: Una alimentación generada por el regulador, con un consumo máximo de 50 mA.

- **GND**[19]: Toma de tierra.

Es interesante mencionar en este punto que, para controlar aquellos componentes a los que se precise cambiar su intensidad, ángulo, velocidad, etc., las placas Arduino suelen emplear una modulación **PWM**[20] (Modulación por Ancho de Pulso). Es así porque gracias a PWM se pueden **simular voltajes** entre siempre encendido (5V) y apagado (0V), cambiando el **porcentaje** de tiempo durante el que la señal está encendida respecto al que pasa apagada (Bricolabs Wiki, 2015).

En Arduino, los pulsos PWM pueden generarse en los **pines** 3, 5, 6, 9, 10, y 11 (para Arduino UNO) utilizando la función **analogWrite** (PIN, CICLO_DE_TRABAJO), donde CICLO_DE_TRABAJO puede ser cualquier número entero entre **0** y **255**, de la forma indicada en la *Ilustración 9*.

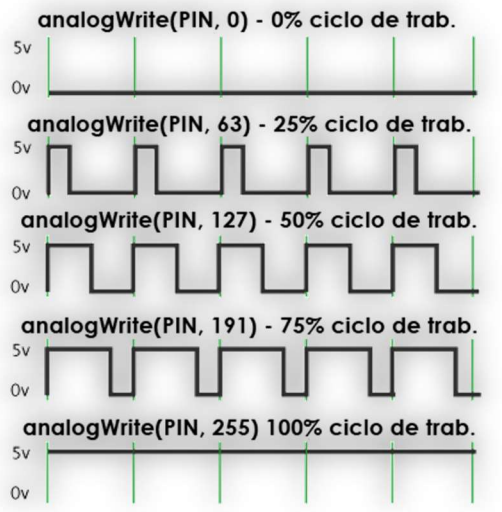


Ilustración 9. Modulación PWM en Arduino

En cuanto a la **programación** de la placa, el entorno de programación más utilizado es **Arduino IDE**[21], descargable desde la página oficial de Arduino. Es **fácil** de entender y **sencillo** de manejar, siempre que se tengan conocimientos de programación y cierta familiaridad con el lenguaje **C**, algo bastante habitual entre los desarrolladores.

4.1.2. Sensor ultrasónico HC-SR04

Los sensores de ultrasonidos suelen ser sensores de **baja precisión**, debido a que la orientación de la superficie a medir puede provocar que la onda se refleje y no resultan adecuados en entornos con gran número de objetos, dado que el sonido rebota en las superficies generando ecos y falsas mediciones. Además, tampoco son apropiados para el funcionamiento en el exterior y al aire libre.



Ilustración 10. Sensor ultrasónico HC-SR04 (Arduino CC, 2020)

Pese a esta baja precisión, son sensores **ampliamente empleados** (Diosdado, 2018). En robótica es habitual montar uno o varios de estos sensores, por ejemplo, para la detección de obstáculos o resolver **laberintos**. Por ello, nos valdremos de este

[19] GND: *Ground*
[20] PWM: *Pulse-Width Modulation*
[21] IDE: *Integrated Development Environment*

tipo de sensor para aportarle al robot **conocimiento** sobre su **entorno**, porque será perfecto en el entorno controlado en el que estará (Infootec, 2017).

El **HC-SR04** (ver *Ilustración 10*) es un sensor ultrasónico que destaca por su **pequeño tamaño**, **bajo consumo** energético, **buena precisión** y **excelente precio**. Es el **más utilizado** dentro de los sensores de tipo ultrasonido, principalmente por la cantidad de información y proyectos disponibles en la web. Posee dos transductores: un **emisor** y un **receptor** piezoeléctricos, además de la electrónica necesaria.

Sus **características técnicas** son las siguientes:

- Voltaje de Operación: 5V DC
- Corriente de reposo: < 2mA
- Corriente de trabajo: 15mA
- Rango de medición: 2cm a 450cm
- Precisión: +- 3mm
- Ángulo de apertura: 15°
- Frecuencia de ultrasonido: 40 KHz
- Duración mínima del pulso de disparo TRIG (nivel TTL): 10 µS
- Duración del pulso ECO de salida (nivel TTL): 100-25000 µS
- Dimensiones: 45mm x 20mm x 15mm
- Tiempo mínimo de espera entre una medida y el inicio de otra 20ms (recomendable 50ms)

En cuanto a los **pines** que resultan relevantes, son éstos:

- **VCC** (+5V DC)
- **TRIG** (disparo del ultrasonido)
- **ECHO** (recepción del ultrasonido)
- **GND** (0V)

El sensor **opera** de la siguiente forma:

a) El **emisor** piezoeléctrico emite 8 pulsos de ultrasonido (40 KHz) tras recibir la orden en el pin **TRIG**.
b) Las ondas de **sonido** viajan en el aire y **rebotan** al encontrar un objeto.
c) El sonido de rebote es **detectado** por el receptor piezoeléctrico, tras lo que el pin **ECHO** cambia a HIGH (5V) por un **tiempo** igual al que le llevó a la onda ir y volver.
d) Finalmente, dicho **tiempo** del pulso **ECHO** es **medido** por el microcontrolador para que podamos emplearlo en la aplicación que necesitemos.

En la *Ilustración 11* puede verse una idea esquemática del **funcionamiento** de un sensor de este tipo, incluyendo los **datos** que se deben tener en cuenta para calcular la distancia medida con mayor precisión.

Notar que el sensor **no** se ve afectado por la **luz** solar o material de color **negro**, aunque los materiales **blandos** acústicamente como tela o lana pueden llegar a ser **difíciles** de detectar, algo que conviene tener en cuenta.

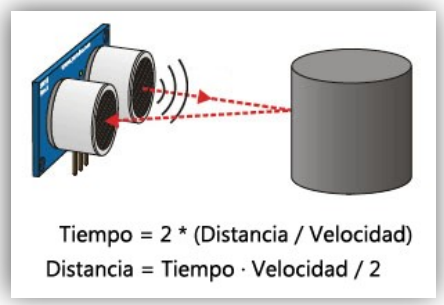


Ilustración 11. Funcionamiento de un sensor ultrasónico

4.1.3. Servomotor SG-90

Puede servir para múltiples fines, pero se hará uso de este servomotor para **orientar** el sensor ultrasónico del robot y recibir **diferentes lecturas** del entorno por el que se desplazará.

El servomotor SG-90 (ver *Ilustración 12*) es de **pequeño** tamaño, perfecto para proyectos que no necesiten desarrollar mucho par de fuerza para mover objetos y donde se requiera poco peso. Muy usado en aeromodelismo, pequeños brazos robóticos y muy apropiado para **Arduino** (Cheung, 2016).

Sus **características técnicas** son las siguientes:

- Voltaje de operación: 3 – 7,2V
- Velocidad: 0,1seg / 60 grados
- Par: 1,3Kg x cm (4.8V) – 1,6Kg (6.0V)
- Ancho de pulso: 4 µseg
- Peso: 9g
- Engranajes: Nylon
- Dimensiones: 22*11,5*27mm
- Longitud del conductor: 150mm
- Conector universal tipo “S”



Ilustración 12. Servomotor SG-90 (Elegoo, 2019)

Aparte de ello, y aunque el servo puede moverse con una resolución de más de **1 grado**, éste es el máximo que se logrará, debido a la **limitación** de la señal **PWM** que es capaz de generar la Arduino UNO.

4.1.4. Sensor de infrarrojos LED TCRT5000L

El TCRT5000L (ver *Ilustración 13*) es un sensor **óptico reflectivo**, sencillo, que detecta la **diferencia** de **color** en un objeto mediante la reflexión de la luz en el mismo. Dispone de un **LED emisor** de luz infrarroja y de un **fototransistor** que **recibe** la luz reflejada, pudiendo distinguir entre zonas claras y oscuras de un objeto.

Estos sensores suelen proporcionarse con una placa de **medición** estándar con el comparador LM393, que permite obtener la lectura como un valor digital cuando se supera un cierto umbral, regulado a través de un **potenciómetro** ubicado en la placa.

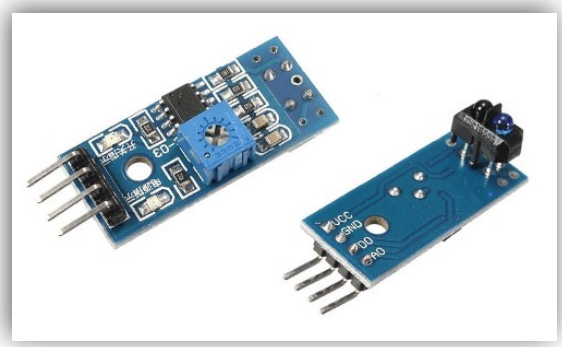


Ilustración 13. Sensor de infrarrojos LED TCRT5000L (Elegoo, 2019)

La cantidad de luz infrarroja tiene una fuerte **dependencia** con el color, material, forma y posición del obstáculo, por lo que **no** disponen de **precisión** suficiente para proporcionar una estimación de la **distancia** a un objeto, simplemente es capaz de su **detección**. En cuanto a su **alcance**, varía entre 0.2 y 15mm, siendo la distancia óptima 2.5mm. Por tanto, es un sensor de muy **corta distancia**.

Los sensores TCRT5000L son ampliamente **utilizados** para hacer robots **seguidores de líneas** (Feiticeir0, 2018), aunque también pueden emplearse para detectar cualquier otro tipo de objeto, si fuera necesario.

En el presente caso, se empleará para **corregir** la **trayectoria** del robot y evitar posibles desvíos de su rumbo al cometer pequeños errores de giro, avance, etc.

4.1.5. Módulo L298N

El módulo controlador L298N (ver *Ilustración 14*) es el **más utilizado** para manejar motores DC de **hasta 2 amperios**, y resultará **imprescindible**, ya que la placa Arduino **sólo** puede suministrar hasta 40 mA de corriente.

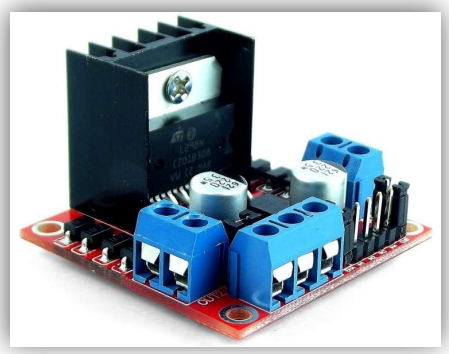


Ilustración 14. Módulo L298N (Elegoo, 2019)

El chip L298N posee **dos puentes H** completos que permiten controlar el **sentido** y **velocidad** de giro de **motores** mediante señales TTL provenientes de placas como **Arduino** (Nay Lamp Mechatronics, 2017).

Tiene integrado un **regulador** de voltaje LM7805 de 5V encargado de alimentar la parte lógica del L298N, perfecto para alimentar la Arduino sin tener que utilizar otra fuente adicional. Eso sí, este regulador sólo funciona si alimentamos el módulo con una **tensión máxima** de **12V**. En nuestro caso se emplearán 2 baterías de 3.7V, es decir, un total de **7.4V**, así que **no** habrá problema.

En cuanto a sus **especificaciones técnicas**, son las siguientes:

- Chip: L298N
- Canales: 2 (soporta 2 motores DC o 1 motor PAP)
- Voltaje de potencia (V motor): 5V - 35V DC
- Consumo de corriente: 0 a 36mA
- Capacidad de corriente: 2A (picos de hasta 3A)
- Potencia máxima: 25W
- Peso: 30g
- Dimensiones: 43*43*27mm

Posee un **conector** de 6 pines para las señales TTL que controlan los motores, una clema de 3 pines para la alimentación, y dos clemas de 2 pines para los motores. Puede verse su **conexionado** en la *Ilustración 15*:

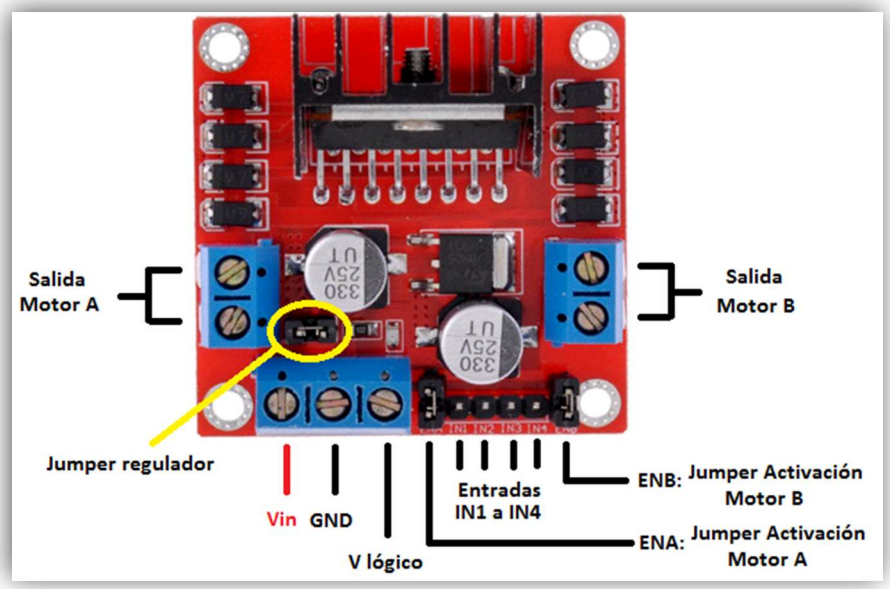


Ilustración 15. Módulo L298N: Conexiones

Para el **control** del módulo, se requiere de los siguientes **pines**:

- **IN1**, **IN2** corresponden a las entradas para controlar el **motor A** (OUT1 y OUT2).
- De igual manera **IN3**, **IN4** permiten controlar el **motor B** (OUT3 y OUT4).
- **ENA** y **ENB** servirán para habilitar o deshabilitar sus respectivos motores, aunque generalmente se utilizan para controlar la **velocidad** mediante una señal PWM.

4.1.6. Motores DC

Un motor de corriente continua DC convierte la energía **eléctrica** en **mecánica** y, por las características del presente proyecto, que emplea una placa Arduino, será el tipo de motores que se utilizarán. Concretamente se emplearán 4 (uno por rueda), que otorgarán tracción, velocidad y rotación al robot.

Para todo ello se empleará del módulo **L298N**, quedando el circuito entre los motores, el módulo, la placa Arduino y las baterías como puede verse en la *Ilustración 16*:

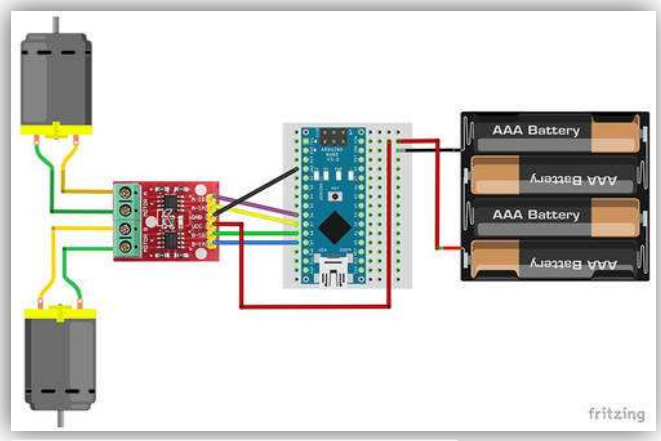


Ilustración 16. Conexionado de los motores

4.1.7. Baterías Li-Ion 18650

Las baterías de litio 18650 (ver *Ilustración 17*) son conocidas por alimentar linternas y láser de mano, y son una de las **mejores opciones** como solución de alimentación para **Arduino**

Dos baterías de litio 18650 puestas en serie proporcionan **7.4-8.2V**, que es un voltaje **perfecto** para alimentar Arduino. Además, se pueden encontrar **porta baterías** que permiten adaptarlas con facilidad a nuestras necesidades, como en nuestro caso concreto.

Tienen la **ventaja** de proporcionar una **alta capacidad** de carga. Las baterías de primeras marcas proporcionan hasta 4800 mAh, aunque en nuestro caso nos quedaremos con **3700 mAh** de capacidad por cada una de las 2 que incorporaremos, tratando de **equilibrar** el coste y el rendimiento.



Ilustración 17. Baterías 18650 junto a un cargador (Amazon, 2020)

Proporcionan una **capacidad de descarga** de entre 1C-2C, según modelos. Esto supone una **intensidad** máxima de hasta **10A**, aunque por seguridad no resulta aconsejable emplear más de 2-4A sin estar muy seguros de la calidad y características de la batería, dado que el uso incorrecto de baterías de litio puede ser peligroso, causando sobrecalentamientos o incluso incendios.

En resumen, son una **opción** bastante más **avanzada** que las típicas pilas de 9V o las baterías AA, y muy interesante para un proyecto como el nuestro, con **mayores requisitos** de capacidad y corriente.

4.1.8. Mando IR

Su **alcance** suele ser **limitado**, típicamente inferior a 3m, dependiendo fuertemente del **ángulo** de emisión. Y **no** existe un **único protocolo** adoptado como estándar, ya que cada fabricante ha desarrollado los suyos propios. Así, tenemos el RC-5 y RC-6 de Philips, el SIRC de Sony, el protocolo NEC desarrollado por la compañía japonesa Nippon Electronic Company, entre muchos otros.



Ilustración 18. Mando IR (Amazon, 2020)

De todas formas, los que se emplean para manejar las placas Arduino (Pattabiraman, 2017) suelen ser bastante **asequibles** económicamente, como el de la *Ilustración 18*, así que siempre podremos encontrar fácilmente uno que resulte compatible.

En el caso que nos ocupa, se utilizará para **cambiar** de **algoritmos** empleados y para **controlar** el **movimiento** del robot de forma directa (similar a un coche teledirigido), para comprobar su correcto funcionamiento global, las ruedas, las baterías, etc.

4.2. Diagrama de bloques y montaje

Una vez decididos los componentes a integrar en el proyecto, podemos ver el diseño del **diagrama de bloques** asociado a los mismos en la *Ilustración 19*, para clarificar su montaje y las ideas que aplicaremos:

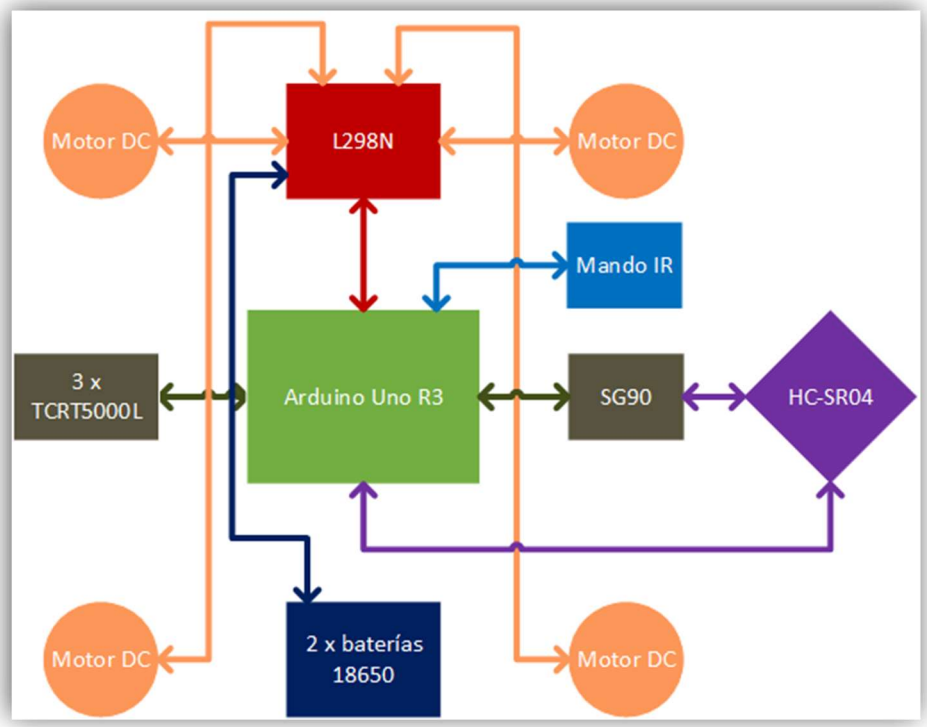


Ilustración 19. Runn3r: Diagrama de bloques

En cuanto al **resultado** final físico de aplicar dicho diagrama de bloques a los componentes que tenemos entre manos, podemos verlo en la *Ilustración 20*:

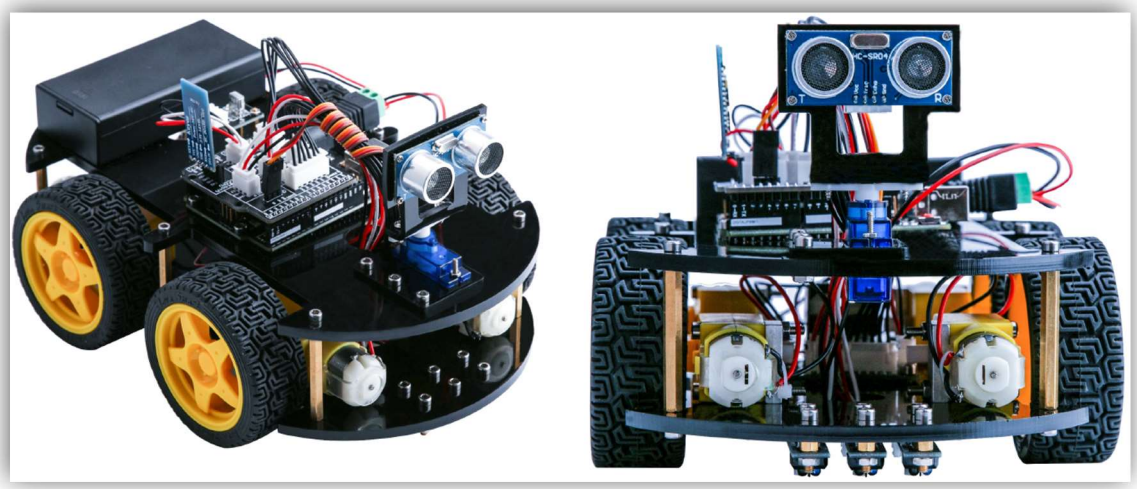
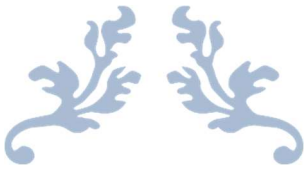


Ilustración 20. Runn3r: Vista lateral y frontal

El **montaje** es bastante claro, dividiéndose en dos **mitades**:

- **Inferior**: Módulo L298N, 4x motores, 3x sensores de seguimiento de líneas y 4x ruedas.
- **Superior**: Arduino Uno R3, placa de conexionado, sensor HC-SR04, servo SG90, receptor IR, 2x baterías, cable de alimentación y caja portapilas.



5. ALGORITMOS Y LABERINTOS



5. Algoritmos y Laberintos

La idea fundamental de este trabajo es hacer un robot **autónomo** que se desplace y resuelva laberintos **complejos**. Así pues, es necesario dotarlo de suficiente "**inteligencia**", mediante **algoritmos** que posibiliten que sea capaz de tomar decisiones por sí mismo.

Es cierto que existe una gran variedad de algoritmos diferentes para resolver laberintos, algunos diseñados para ser utilizados **sin conocimiento** del laberinto y otros cuando el robot sí posee un conocimiento **previo** del mismo, siendo capaz de resolverlos mucho más rápido, evidentemente.

Pero, de todos ellos, se deben **seleccionar** aquellos algoritmos que puedan implementarse **eficientemente** en **Arduino** porque, debido a sus características concretas, **no** todos son recomendables. Para ello, se **mostrarán** algunos de los más empleados en la actualidad, tras lo que se **analizarán** para saber si es posible adaptarlos a las características del presente proyecto.

Así pues, se empleará la siguiente **metodología**:

a) **Presentación** de los algoritmos de resolución de laberintos más habituales:
   - **Sin** conocimiento previo.
   - **Con** conocimiento previo.

b) **Estudio** de dichos algoritmos.

c) **Selección** e **implementación** de los algoritmos en **Arduino**.

5.1. Algoritmos sin conocimiento previo

5.1.1. Aleatorio o del "ratón"

Este es un método **trivial** inspirado en el comportamiento de un ratón imaginario al que se introduce en un laberinto y desconoce cómo salir del mismo, ni tiene referencia alguna sobre él.

Su **secuencia** de pasos es la siguiente:

a) El robot sale del punto de partida del laberinto y avanza en línea **recta**.

b) Cuando llegue a un cruce, toma una decisión **aleatoria** sobre la siguiente dirección a seguir.

c) Continúa avanzando de este modo hasta llegar a la **meta**.

Aunque este método **siempre** acaba encontrando la **solución** correcta, este algoritmo suele ser bastante **lento** y **nada predecible**, por lo que **no** suele ser empleado.

5.1.2. Seguidor de paredes *(Wall-Follower)*

También conocido como **RMI**[22]**/LSRB** o **RMD**[23]**/RSLB**, según la **dirección** de **preferencia** a seguir (ver *Ilustración 21*), requiere que el laberinto esté **simplemente conectado**, es decir, que todas sus paredes están conectadas entre sí o al límite exterior del laberinto.

La **idea** es sencilla: Se debe mantener una "mano" **en contacto** con una pared del laberinto. Así se **garantiza** que el robot no se perderá y que alcanzará una **salida**, siempre que exista. De no existir, el algoritmo **volverá** a la **entrada** habiendo atravesado cada corredor al menos una vez.

Notar que, como se ha comentado, si el laberinto **no** está simplemente conectado (es decir, si existen bucles en el camino a la salida), este método **no** alcanzará la meta.

Entre sus **ventajas**, destacan las siguientes:

- Es **fácil** de seguir.
- Siempre encuentra una **salida**, incluso si eso significa volver a la entrada (sigue siendo una salida).
- **No** necesita un **mapa** del laberinto.

Por otra parte, entre sus **desventajas** se encuentran:

- **No** es eficiente en laberintos **grandes**.
- Puede que haya salidas más **cercanas** que sean **ignoradas**.
- **No** suele encontrar la mejor solución.
- Se ve **afectado** por los **bucles** en los laberintos.

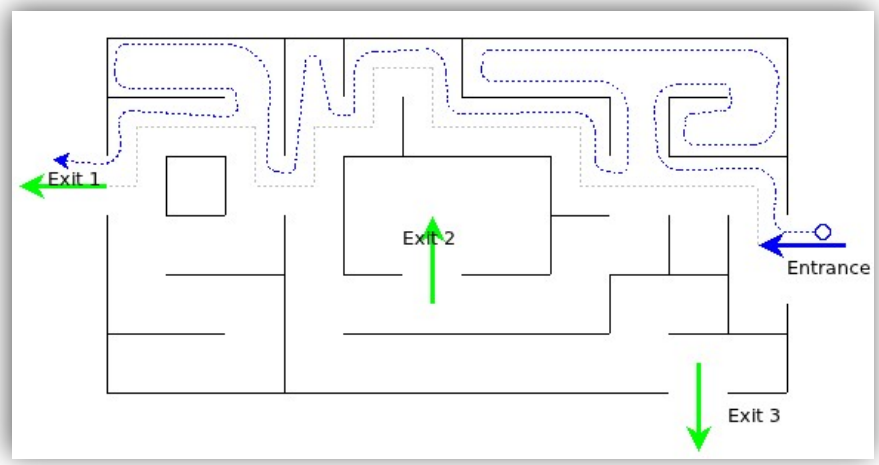


Ilustración 21. RMD: Resolución de un laberinto (m0rd0r, 2019)

[22] RMI: Regla de la Mano Izquierda
[23] RMD: Regla de la Mano Derecha

Como apunte final, comentar que a veces resulta oportuno **cambiar** entre la RMI/RMD y viceversa, porque puede que el robot encuentre una salida en **menos** tiempo y de forma más **rápida**.

5.1.3. Pledge

Los laberintos que **no** estén **simplemente conectados** se pueden resolver con el método del seguidor de paredes, pero **sólo** cuando la entrada y la salida estén en las paredes **exteriores** del laberinto.

Sin embargo, como se ha comentado, el robot podría estar en una sección que **no** esté **conectada** a la salida por ninguna pared, en cuyo caso estará dando vueltas eternamente, en un **bucle infinito**.

Pero el algoritmo **Pledge** (llamado así por Jon Pledge) es capaz de **resolver** este problema. Está expresamente diseñado para **sortear obstáculos**, y requiere elegir una dirección **al azar** para seguir adelante, que será **preferencial**.

Su **funcionamiento** es el siguiente:

a) Se elige una **dirección** favorita: Norte, Sur, Este, Oeste.
b) Se elige qué **mano** vamos a emplear, la derecha o la izquierda.
c) Se **avanza** en esa dirección.
d) Cuando se encuentra un **obstáculo**, se emplea la mano elegida a lo largo del obstáculo mientras se **cuentan** los **grados** girados. Pero hay **diferencia** entre elegir una mano y otra:
   - Mano **derecha**: El giro en sentido horario será positivo y el antihorario será negativo.
   - Mano **izquierda**: El giro en sentido antihorario será positivo y el horario será negativo.

e) Si el robot llega a girar **360º**: Si mira hacia la dirección **favorita** y la **suma** angular de los giros realizados es **0**, el robot debe **abandonar** el obstáculo y **continuar** su rumbo original.
f) Se continúa desde **c)** hasta encontrar la **salida**.

Es decir, que la **mano** se **retira** de la pared sólo cuando la "suma de giros realizados" y el "rumbo actual" son cero. Y ésto es justamente lo que permite que el algoritmo **evite** posibles "trampas" en el recorrido, normalmente con la forma de la letra "**G**".

En el mundo real, este algoritmo podría permitir a una persona con una **brújula** encontrar su camino desde **cualquier** punto dentro del laberinto hasta una salida en cualquier laberinto 2D[24] finito.

[24] 2D: *2-Dimensional*

Una **comparación** con la RMD puede verse en la *Ilustración 22*:

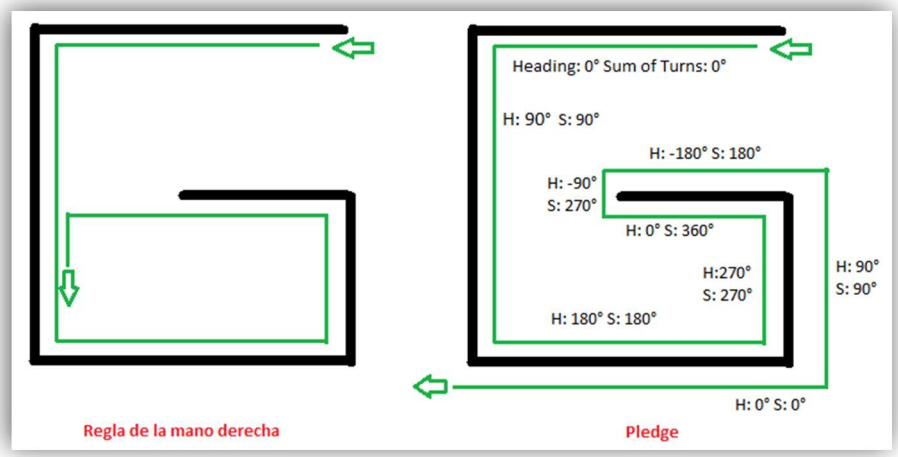


Ilustración 22. Pledge: Resolución de un laberinto

5.1.4. Trémaux

Inventado por Charles Pierre **Trémaux**, requiere dibujar **indicaciones** en el suelo para **marcar** ciertos recorridos, y funciona para **todos** los laberintos que estén bien definidos. Aunque, por otra parte, **no** garantiza encontrar la ruta **más corta**.

El algoritmo funciona de acuerdo con las siguientes **reglas**:

1) Un camino desde un cruce puede estar "**no visitado**", "marcado **una vez**" o "marcado **dos veces**".
2) Cuando se **siga** un camino, se debe **marcar** ambos **extremos** del mismo, su inicio y su fin.
3) **Nunca** se debe entrar en un camino que tenga **dos marcas**.
4) Si se llega a un cruce **sin marcas**, se debe **elegir** una ruta sin marcar al azar, seguirla y marcarla.
5) Si el camino al que se llega tiene solo **una marca**, se debe **dar la vuelta** y regresar por dicho camino, **marcándolo** nuevamente. Ésto debería ocurrir al encontrar un callejón sin salida.

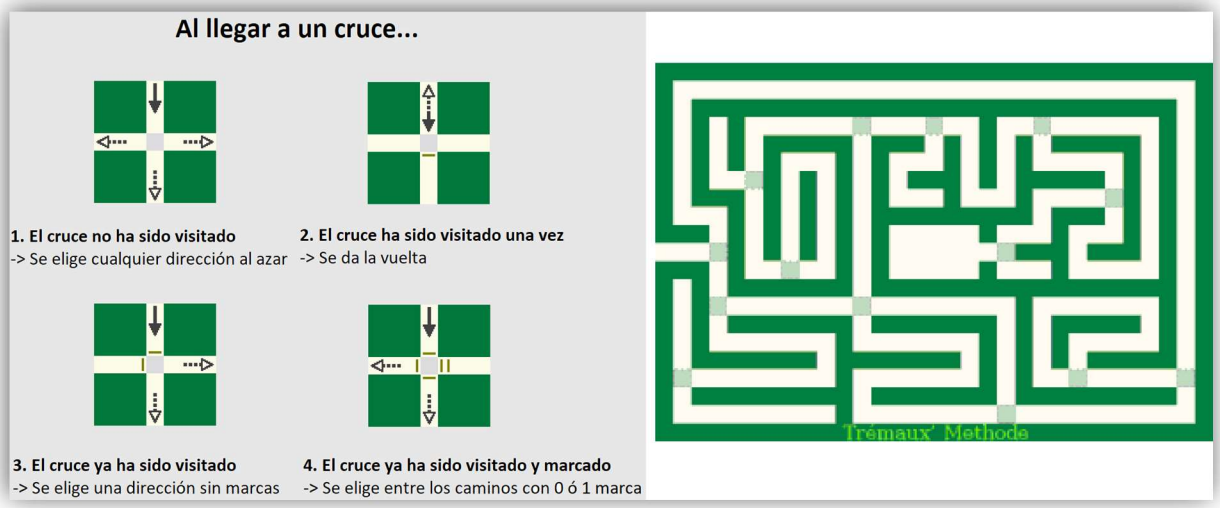


Ilustración 23. Trémaux: Reglas y ejemplo de un laberinto

6) Si no, hay que elegir **al azar** una de las rutas restantes con la **menor** cantidad de marcas (cero si es posible, de lo contrario, una), luego **seguir** dicha ruta y **marcarla**.

Cuando finalmente se llegue a la **salida**, las rutas marcadas exactamente **una vez** indicarán el camino de **regreso** al inicio. Si **no** hay salida, podremos volver al inicio y todas las rutas se marcarán **dos veces**.

En la *Ilustración 23* pueden verse las **reglas** que deberían seguirse para hallar la salida del mismo y un **ejemplo** de laberinto adaptado a los cruces que sugiere Trémaux.

5.2. Algoritmos con conocimiento previo

5.2.1. Inundación *(Flood-Fill)*

El algoritmo se basa en **dividir** el laberinto en **celdas**, cada una con un **valor heurístico**. Como **condiciones** iniciales se tienen las **dimensiones** del laberinto y la **posición** de la meta.



Ilustración 24. Inundación y laberinto marcado (Automouse, 2016)

El término de inundación se le da a este algoritmo precisamente porque se encarga de "**inundar**" el laberinto con los valores, como si se vertiera agua desde la celda solución y se fuera desplazando a través del laberinto por diferentes caminos, pasando por diferentes niveles.

Dichos valores se **calculan** de la siguiente **forma**:

a) La **salida** del laberinto vale **0**, lo que quiere decir que se requieren 0 **pasos** para llegar a ella.
b) Las celdas **adyacentes** a la meta, ya sean de manera **vertical** u **horizontal**, tendrán un valor de **1** (primer nivel); las celdas adyacentes a las del primer nivel tendrán un valor de **2** (segundo nivel); las celdas adyacentes a éstas últimas tendrán valor de **3** (tercer nivel), y así **sucesivamente**.
c) Sin embargo, si existe un **muro** entre dos celdas, éstas **no** se considerarán **adyacentes** visibles.

La *Ilustración 24* muestra un laberinto **marcado** con los diferentes niveles hasta llegar a la meta.

5.2.2. Seguidor de paredes optimizado *(Optimized Wall-Follower)*

Es, básicamente, el **mismo** algoritmo que el seguidor de paredes normal, pero con la **información** obtenida al llegar a la meta. Es decir, se tiene **conocimiento** exacto del recorrido hecho previamente, sus giros, cambios de sentido, etc. Dicho conocimiento será empleado en un **segundo recorrido**, **optimizando** la ruta seguida y eliminando cambios de sentido y giros innecesarios en la búsqueda de la meta.

Para ello, se emplean las **reglas** de la *Ilustración 25*:

I = Izquierda **R** = Línea recta

D = Derecha **C** = Cambio de sentido

a) ICD = **C**
b) ICR = **D**
c) DCI = **C**
d) RCI = **D**
e) RCR = **C**
f) ICI = **R**

Ilustración 25. Seguidor de paredes optimizado: Reglas

Aplicando dichas **reglas**, se puede seguir un camino mucho más **corto** y **sin** cambios de dirección innecesarios, ahorrando mucho **tiempo** en la resolución del laberinto.

5.3. Selección de algoritmos laberínticos

Llegados a este punto, se **descartarán** dos de los algoritmos analizados por los siguientes **motivos**:

- **Pledge**: Como el robot se va a desplazar por laberintos en los que la entrada y la salida estarán conectadas mediante paredes no resultará necesario implementarlo, al **no poder comparar** los resultados del resto de algoritmos con él ya que no existen posibles "trampas".
- **Trémaux**: Debido a su concepción, se requiere de una cantidad de **memoria** reseñable, al tener que llevar un registro de marcas, caminos, cruces, etc. En nuestro caso, se emplea una Arduino Uno, y está algo limitada si se manejan laberintos de cierto tamaño.

Una vez descartados, los **seleccionados** para ser implementados y probados serán los siguientes:

- **Aleatorio** o del "**ratón**": Fácil de diseñar y apenas tiene requerimientos de memoria, porque el hecho de ser aleatorio implica que **no** precisa de almacenar ningún recorrido.
- **Seguidor de paredes**: Algo más elaborado que el aleatorio, pero sus necesidades están **al alcance** de una Arduino, casi en cualquiera de sus versiones.
- **Seguidor de paredes optimizado**: En este caso, el recorrido sí deberá **almacenarse** en memoria para poder encontrar la ruta óptima. Es totalmente factible, pero habrá que tenerlo **presente** al diseñarlo, tratando de aprovechar los recursos al máximo.
- **Inundación**: Es el más **complejo** de todos, de diseño y de implementación, pero si se diseñan las **fases** que lo integran de la forma adecuada, no deberían existir grandes limitaciones.

5.4. Los laberintos

5.4.1. Virtual+Estándar

Para mostrar el correcto funcionamiento de los algoritmos planteados **no** es necesaria la creación de laberintos de **grandes** dimensiones. Es más, sería **contraproducente**, ya que las **limitaciones** de **memoria** existentes en las Arduino Uno podrían hacer que **no** fuera posible ni efectuar las primeras pruebas.

Por ello, se ha optado por diseñar un **buen** laberinto inicial, no muy grande, pero que servirá perfectamente para el propósito planteado, ya que es **controlable** y sus resultados **varían** lo necesario para obtener conclusiones lo suficientemente relevantes de cara a plantear posibles **mejoras**.

Cabe reseñar que, debido a su particularidad, en el caso del Algoritmo de **Inundación** se han dado valores iniciales a las diferentes casillas del laberinto empleando **letras**, "**a**" para la **meta** y letras sucesivas dependiendo de la **distancia** a la misma, que serán evaluadas y actualizadas por el algoritmo.

Su **diseño** es el de la *Ilustración 26*:

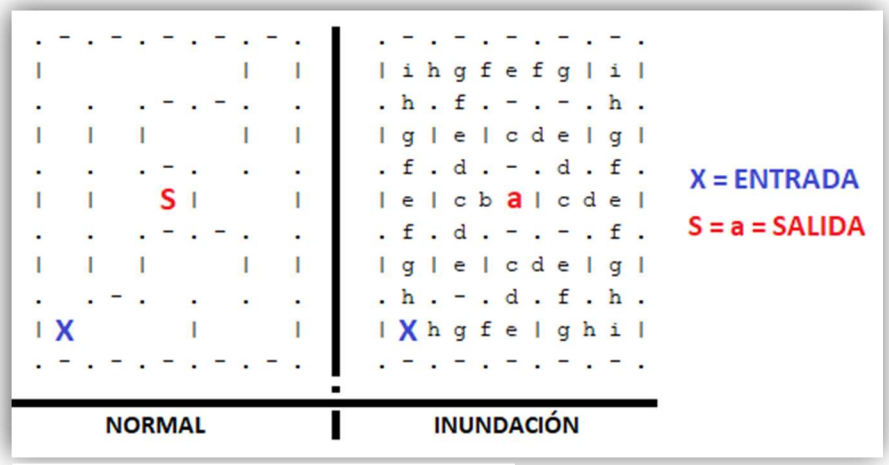


Ilustración 26. Laberinto Virtual+Estándar: Diseño

Como puede observarse, su tamaño final ha sido de **11x11**, con la peculiaridad de que se ha tenido que emplear dicho tamaño para almacenarlo porque se hacía necesario reflejar la posición y la ubicación de los diferentes **obstáculos** que, evidentemente, también han sido considerados **celdas** (ocupadas, en este caso).

5.4.2. Real+Estándar

El diseño y el trazado de **ambos** laberintos, el Virtual+Estándar y el Real+Estándar, es **idéntico** en su **concepto**, porque se ha buscado probar las decisiones, giros y cambios de sentido de **igual** forma, sin importar por dónde se mueva el robot. Pero **no** es así en su **recorrido**, ya que se han tratado de **reducir** al

máximo las decisiones en cuanto a su avance, aprovechando el hecho de que siempre puede hacerse de **dos en dos celdas** (ver *Ilustración 27*), **ahorrando** la mitad de **tiempo**. De todas formas, se ha creado de tal forma que resulta fácilmente **editable**, para probar los algoritmos bajo muchos **otros** diseños.

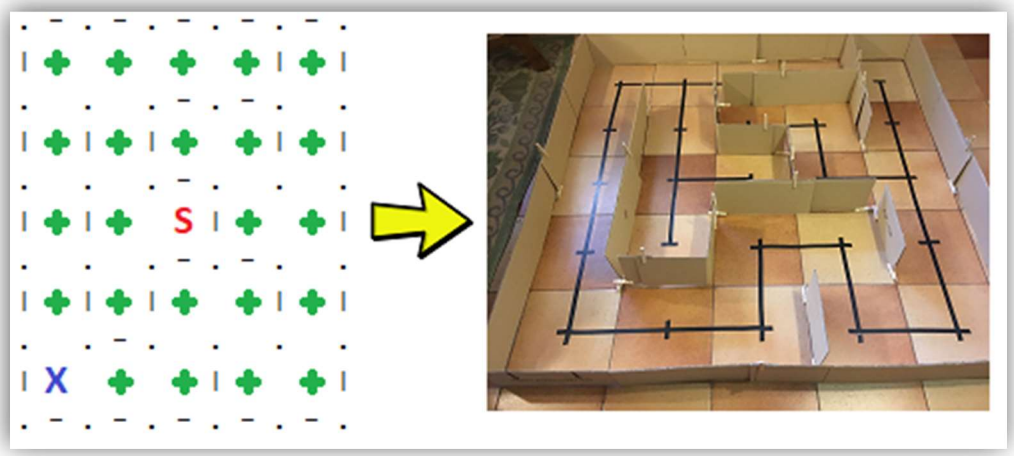


Ilustración 27. Laberinto Real+Estándar: Vista general del diseño de 2x2 celdas

Las **baldosas** miden aproximadamente 60*60 cm y, a pesar de que han resultado de bastante ayuda al delimitar el laberinto, seguramente sería preferible algo más de **margen** respecto del robot.

En cuanto a la traslación al mundo **real** del laberinto, se ha procurado elegir bien los **materiales** para las paredes y para las líneas de seguimiento (ver *Ilustración 28*), con la intención de no interferir con los **sensores**, y con el mismo **suelo**, porque su diferente **rozamiento** influye sobremanera en los movimientos del robot, especialmente los **giros**, en los que el robot puede sufrir desviaciones en su rumbo fácilmente.

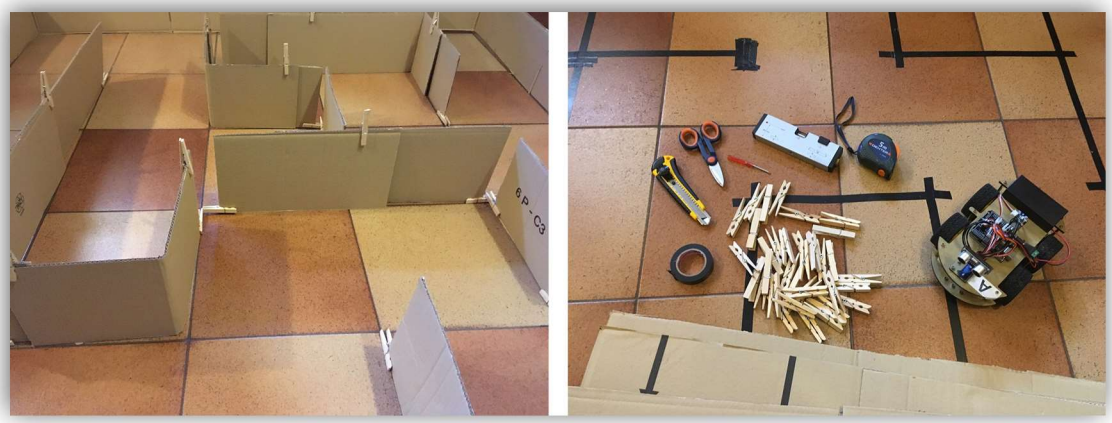


Ilustración 28. Laberinto Real+Estándar: Materiales empleados

Se ha empleado poco más que cartón, pinzas de madera y cinta aislante, aparte de **herramientas** como una tijera, un cúter, un metro y un nivel láser, para hacer los cortes en los cartones lo más homogéneos posibles.

5.5. Implementación de los algoritmos

5.5.1. Entorno simulado

Para poder hacer **comparaciones** de forma adecuada, ciertos aspectos siempre serán **comunes** entre los algoritmos desarrollados, como:

a) Arduino IDE y lenguaje **C**, con las características de Arduino.
b) **Direcciones** (Norte – Sur – Este – Oeste).
c) **Laberinto**, aunque puede variarse su diseño de forma sencilla.
d) Posiciones de los **obstáculos**, que **ocupan** espacio en el laberinto virtual. Este hecho hay que tenerlo en cuenta, ya que deberá evitarse colisionar con ellos.
e) Muestra de datos por **pantalla**, a través del puerto serie.

Aleatorio o "del ratón"

Debería ser, a priori, el **menos óptimo** de todos los algoritmos. El hecho de que sea **aleatorio** significa que puede variar entre obtener una ruta óptima o seguir la peor que sea posible. Por otra parte, su aleatoriedad **simplifica** bastante su **programación** porque reduce la toma de decisiones.

Por último, decir que para que el número obtenido sea **aleatorio realmente**, conviene emplear la instrucción **`randomSeed`** sobre un puerto **analógico** de la placa Arduino, que siempre obtendrá valores diferentes debido a sus constantes cambios de estado.

Seguidor de paredes *(Wall-Follower)*

Su fortaleza reside en que **siempre** va a llegar a la **salida**, lo va a hacer de una forma **predecible** y, generalmente, empleará un tiempo **medio**.

En cuanto a las **direcciones** seguidas, se ha desarrollado un algoritmo para la mano **izquierda** y otro para la mano **derecha**, que son, básicamente, **simétricos** a la hora de elegir dichas direcciones.

Seguidor de paredes optimizado *(Optimized Wall-Follower)*

Su **dificultad** de implementación se debe a que necesita una **segunda pasada**, lo que **implica**:

a) Es necesario ir almacenando la **ruta** seguida para poder **optimizarla**.
b) Se requiere de un **algoritmo** paralelo de **optimización** de la ruta.
c) Se debe crear un **segundo laberinto** con las direcciones óptimas a seguir.
d) Hay que devolver al robot a la posición **inicial**, para que pueda **demostrar** que ha sido capaz de encontrar la ruta **óptima** a seguir.

Ahora bien, una vez bien diseñado, en la segunda pasada será capaz de alcanzar la meta recorriendo la ruta **óptima**, demostrando su capacidad de **aprendizaje**.

Inundación (Flood-Fill)

También es **complejo** de implementar porque, en él, los **valores** de las casillas son fundamentales. Por ello, a pesar de que la asignación **inicial** de **valores** a las casillas del laberinto es **estándar**, sí habrá que estudiar con detalle la forma en la que el robot **modifique** y **actualice** los valores de las mismas, porque un valor erróneo hará que siga una ruta que no sea la óptima.

En el desarrollo llevado a cabo, se ha **dividido** este algoritmo en tres **partes**:

1) **Exploración** del **laberinto**: Para modificar y asignar valores a las casillas que encuentre el robot.
2) **Vuelta** a la **salida**: Al llegar a la meta, el robot debe retroceder hasta la salida, modificando y actualizando los valores que crea oportunos para determinar la **mejor ruta**.
3) **Recorrido optimizado**: Una vez que se conoce la mejor ruta, tan sólo queda **recorrerla**.

- Estas tres partes en las que se ha dividido servirán para **observar** mejor el comportamiento del algoritmo sobre el robot y facilitará su traslación al mundo **real**, dada su mayor **complejidad**.

5.5.2. Entorno real

Tras llevar a cabo la implementación del robot en un entorno simulado, el paso al robot **real** conlleva la **gestión**, común a todos los algoritmos, de los siguientes **apartados**:

a) **Algoritmos**: Los movimientos físicos serán de **2 en 2** celdas, para mejorar los tiempos.
b) **Arduino**: Gestión de tiempos, optimización de cálculos, aprovechamiento de la memoria, etc.
c) **Sensor de ultrasonidos**: Medición de distancias, posicionamiento, etc.
d) **Servo**: Control de posición del sensor de ultrasonidos.
e) **Sensores de líneas**: Calibración de sensibilidad, distancia al suelo, materiales a emplear, etc.
f) **Motores**: Velocidad, potencia suministrada, sentido del giro, control de ejes, etc.
g) **Baterías**: Alimentación y gestión de la energía suministrada.

Debido a ello, se han tenido que efectuar **pruebas** de todos y cada uno de los puntos comentados, teniendo que ajustar, entre otros, los siguientes **puntos**:

- El **servo** que gobierna el sensor de ultrasonidos **gira** algo más a la izquierda que a la derecha, además de tener cierta **desviación**, por lo que en vez de emplear **180-90-0** grados para comprobar distancias a izquierda-frente-derecha, se debe posicionar a **170-88-15** grados.

- El sensor de **ultrasonidos** no es nada apto para operar con ciertos **materiales** blandos (tipo tela o paños), por lo que hay que emplear materiales más rígidos y fáciles de medir.
- Las **ruedas** tienen más **potencia** en el eje derecho, por lo que hay que tenerlo en cuenta y ajustarlo correctamente para los tiempos de giro, avance, etc.
- Los **sensores de líneas** necesitan una **calibración** algo diferente, porque el de la izquierda es algo más sensible que los otros dos.
- La **potencia** de los **motores** debe manejarse de forma **analógica** mediante pulsos PWM.

En cuanto al **diseño** del **laberinto**, comentar que resulta muy importante la **superficie** elegida y el **material** empleado, porque pueden **interferir** con los diferentes **sensores**. Además, para tratar de replicar el entorno simulado, se han creado **celdas virtuales** dentro del laberinto, apoyado por ciertas **marcas** en el suelo que servirán para delimitarlas, gracias a los sensores de **líneas**.

Por otro lado, el disponer de un **único** sensor de **ultrasonidos** ha obligado a reflexionar acerca de que el robot avanza y su **trayectoria no** es **perfecta**. De haber tenido tres sensores, uno de frente y dos a los lados, se podría corregir su rumbo con relativa facilidad, midiendo distancias continuamente, pero no es el caso. Así pues, se ha optado por **combinar** el sensor de ultrasonidos con los sensores de **líneas**, que serán los encargados de aportar la **posición**, y así el posible desvío del robot de su rumbo ideal. Ha sido complejo, pero el resultado es muy **satisfactorio**.

En este punto, para tratar de **suavizar** el movimiento del robot tanto en su avance como en los giros aún más, se ha optado por integrar un **controlador PID** en el movimiento del robot (ver *Ilustración 29*).

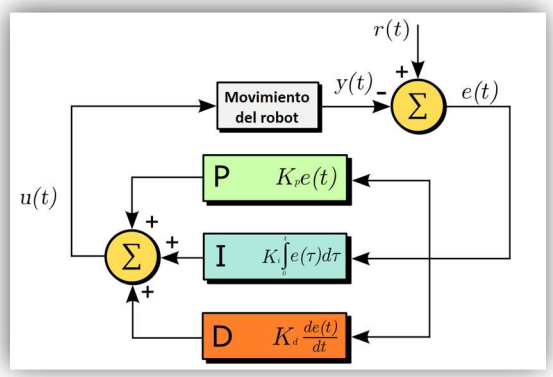


Ilustración 29. Controlador PID: Concepto

Así, para lograr que el robot dibuje sus **trayectorias** de la mejor forma posible se han **empleado**:

a) **Sensores de líneas**: Comprueban la **posición** del robot.
b) **Controlador PID**: Según los datos obtenidos por los sensores de líneas, trata de **corregir** la trayectoria del robot en mayor o menor medida.
c) **Motores**: Dependiendo del controlador PID, se enviará más o menos **potencia** a los motores izquierdo y derecho, para corregir la posible desviación del robot de su trayectoria ideal.

Y tras someter el robot a unas cuantas **pruebas**, **errores** y **ensayos** en diferentes situaciones en cuanto a materiales, rozamientos, potencia de los motores y capacidad de giro, los **valores** que he acabado empleando para el propio controlador **PID** son los de la *Ilustración 30*:

$K_p = 2.5$

$K_i = 0.02$

$K_d = 12$

Ilustración 30. Controlador PID: Valores empleados

Comentar también que se ha implementado un **menú** para **elegir** los algoritmos desde el **mando IR**, tanto para al emplear un entorno simulado como uno real, debiendo **ajustar** la **memoria** al máximo.

Aleatorio o "del ratón"

Su **dificultad** radica en que ahora, a diferencia del entorno simulado, para saber si el robot puede darse **media vuelta** desde una posición debe conocer **si puede** hacerlo. Para ello, y debido a que se dispone de un **único** sensor de ultrasonidos, habrá que **cambiar** su **orientación** y comprobar lo que tiene de **frente**.

En este apartado el controlador **PID** resulta muy **útil**, ya que puede **corregir** exceso o defecto de giro debido a la potencia de los motores, al rozamiento de la superficie donde esté el robot, etc.

Seguidor de paredes *(Wall-Follower)*

Su funcionamiento es muy **similar** al visto en el apartado simulado, salvo por la integración del PID, las mediciones de los sensores, el avance por las líneas, etc.

Seguidor de paredes optimizado *(Optimized Wall-Follower)*

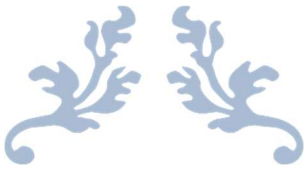
En este caso, el planteamiento se **complica** considerablemente. Al igual que en el entorno simulado, el hecho de manejar **dos** laberintos (el inicial y el optimizado), obliga a **adaptarse** a ellos.

Decir que se ha incorporado un **temporizador** de 10s para poder **reposicionar** el robot en la salida tras la vuelta de reconocimiento, justo antes de empezar con la ruta óptima.

Inundación (Flood-Fill)

El último algoritmo implementado es más **complejo**, como ya sucedió con el entorno simulado. De todas formas, se han vuelto a respetar las **tres fases** en las que se ha dividido, lo que ha posibilitado crear **dos variantes** del mismo:

a) **Con recorrido**: El robot **explora** el laberinto, asigna valores y, una vez de vuelta en la salida, recorre la ruta **óptima** que ha encontrado.
b) **Sin recorrido**: Ya que es un algoritmo informado, **no** hay necesidad de que el robot **explore** físicamente el laberinto, por lo que, en un primer momento, lo recorrerá y asignará valores de forma virtual. Tras ello, sólo le restará recorrer la ruta **óptima** para llegar a la salida.



6. ALGORITMOS LABERÍNTICOS: RESULTADOS



6. Algoritmos Laberínticos: Resultados

6.1. Pruebas iniciales

6.1.1. Entorno simulado

Se ha tenido que:

- Optimizar el funcionamiento de los **algoritmos**.
- **Estructurar** el **código** de forma lo más modular y reutilizable posible.
- **Optimizar** el funcionamiento de los algoritmos y **adaptar** su código a la placa Arduino Uno.
- Elaborar una **documentación** que sea comprensible y útil.

6.1.2. Entorno real

Al tratar de trasladar los algoritmos del mundo virtual al **real**, han surgido ciertos problemas iniciales de diferentes tipos, la mayoría esperables. Son de **destacar**:

- La gestión de la **memoria** es fundamental en Arduino, porque los **2 Kbytes** de los que dispone para las **variables** pueden parecer suficientes en un primer momento (de hecho, suelen serlo), pero si el programa se complica, se acabarán muy pronto. De todas formas, una buena aliada es la función **F( )** empleada justo antes de **mostrar** cualquier cadena de **caracteres**. En vez de almacenarla toda en memoria antes de mostrarla por pantalla, la recorre carácter a carácter, logrando una **eficiencia** considerable, y **rebajando** considerablemente el **uso** de memoria.
- El **receptor IR** del mando a distancia ha dado bastantes más problemas de los previstos. Si bien en la parte simulada funciona sin problema, en la parte real **interfiere** con el **`timer`** del servo. Se ha tratado de cambiar los que emplean uno y otro, pero todas las librerías que manejan receptores IR emplean el mismo, el `timer2` (Minhaj, 2018). Así pues, se ha tenido que crear una función aparte que **alterna** su uso, porque la placa Arduino Uno tiene **menos** *timers* de los que serían deseables.
- Por otra parte, el **receptor IR** que venía con el kit se **fundió** tras un par de usos, y hubo que encargar otro que fuera **compatible** con el mando que tengo. Por suerte, a la primera ya estaba emparejado.
- El **servo** del sensor de ultrasonidos resultó estar algo **inclinado** hacia adelante, lo que hacía que no midiera correctamente las distancias. Fue corregido gracias a la flexibilidad de sus materiales.
- Además, dicho servo es capaz de **girar** algo más hacia la izquierda que hacia la derecha, lo que se corroboró tomando medidas con un haz láser. Dichos apuntes fueron trasladados al código, pudiendo ahora apuntar a la derecha, al frente y a la izquierda sin problema.
- El **eje de las ruedas** se acopla de forma demasiado **holgada** a los motores, lo que en un principio hacía que deslizasen sobre el propio eje y que no aportasen la potencia deseada. Pudo ser arreglado dando la vuelta al eje de giro y a los propios motores, ya que son bastante sencillos en su interior. Al final tenían incluso cierto exceso de potencia, solucionado mediante pulsos **PWM**.

- Los **cables de alimentación** de las baterías a la placa Arduino se **sueltan** con cierta facilidad. Ello me obligó a emplear el soldador y a reestañarlos para ganar resistencia.
- Uno de los **sensores de líneas** se **fundió** debido a un exceso de corriente. Por ello, hubo que emplear un polímetro y soldar una pequeña resistencia para solucionarlo.

Pero, aparte de todo ello, el kit de **robot** elegido se comporta realmente **bien** y es de construcción **sólida**.

6.2. Resultados

6.2.1. Laberinto Virtual+Estándar

Aleatorio o "del ratón"

En este caso, se debe ejecutar **varias** veces para extraer conclusiones. Su naturaleza **aleatoria** hace que los tiempos empleados no sean los mismos en ninguna ocasión.

Los **resultados** han sido los de la *Ilustración 31*:

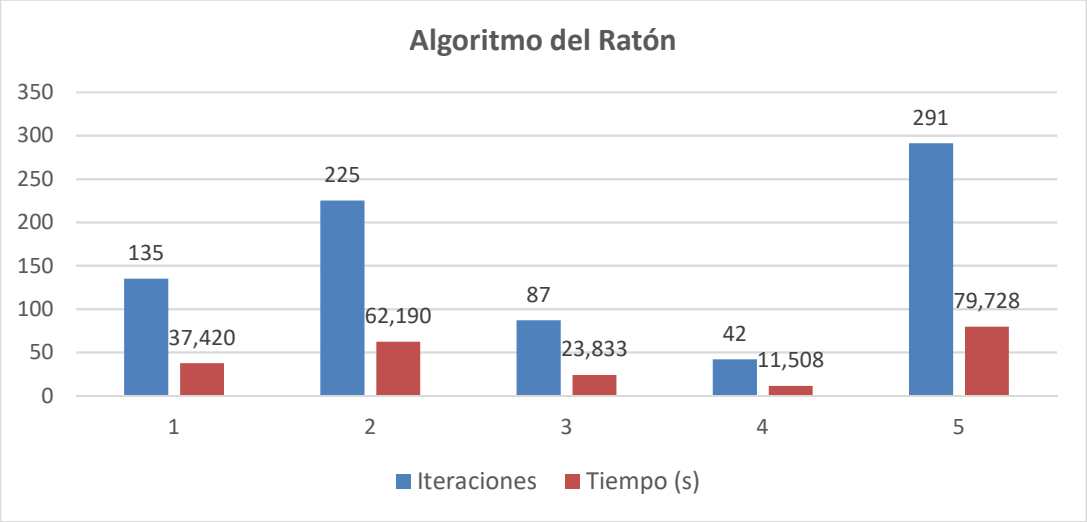


Ilustración 31. Resultados: Laberinto Virtual+Estándar - Algoritmo del Ratón

Como puede observarse, los **resultados** obtenidos son muy **dispares**. A pesar de ello, puede apreciarse que las **iteraciones** conllevan entre **0,27** y **0,28 segundos**, un tiempo de **computación** bastante **estable**.

Seguidor de paredes *(Wall-Follower)*

Su funcionamiento es **predecible**, si está bien implementado. Así pues, no se necesitará realizar varias pasadas, a diferencia del caso anterior. Además, también se mostrará la **diferencia** entre emplear la mano **izquierda** y la mano **derecha** porque, a **igualdad** de laberinto, sus resultados pueden ser muy **diferentes**.

El **tiempo** y las **iteraciones** empleadas pueden verse en la *Ilustración 32*. Como era de esperar, debido al diseño de este laberinto existe bastante **diferencia** entre elegir la RMI y la RMD, por lo que conviene hacer **pruebas antes** de decidirse por uno de los dos.

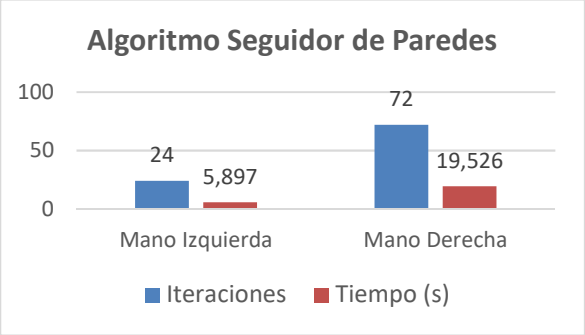


Ilustración 32. Resultados: Laberinto Virtual+Estándar - Algoritmo Seguidor de Paredes

Notar que el tiempo de computación es **similar** al caso anterior, situándose entre **0,245** y **0,271** segundos.

Seguidor de paredes optimizado *(Optimized Wall-Follower)*

Este algoritmo se divide en **dos partes**, la común a la **RMI** y la que llega a la meta recorriendo la ruta **óptima**. Por ello, éstos serán los **tiempos** de referencia para los resultados, que pueden verse en la *Ilustración 33*:

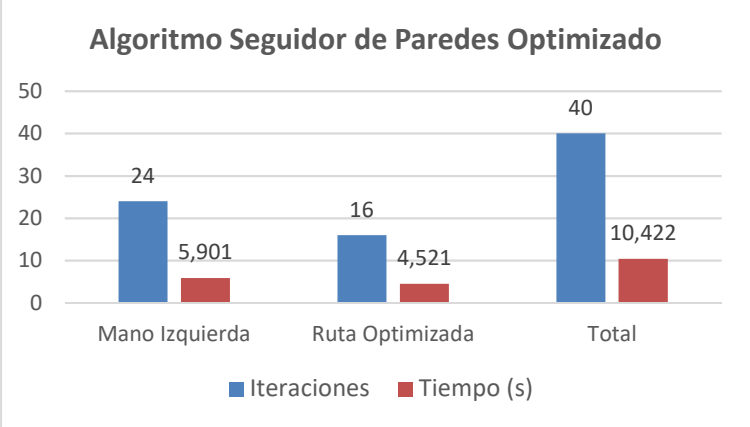


Ilustración 33. Resultados: Laberinto Virtual+Estándar - Algoritmo Seguidor de Paredes Optimizado

Sus resultados son bastante **dispares**, evidentemente, porque el algoritmo **optimizado** sabe dónde llegar, cómo y de la mejor forma, lo que hace que **recorte** tiempos claramente.

Ahora bien, hay que decir que **primero** debe ejecutarse el algoritmo normal y, **tras ello**, se podrá obtener la ruta optimizada, por lo que el tiempo **total** será, en un primer momento, **superior**.

Inundación (Flood-Fill)

Consta de **tres partes**, por lo que su **tiempo** de computación conjunto es **superior**, algo **esperable** en los algoritmos que buscan la **ruta óptima**. Así, los **resultados** obtenidos han sido los de la *Ilustración 34*:

Al igual que en caso anterior, el tiempo **total** es **superior**, pero si se parte de la **misma** base y no hay cambios, se **podrá** aplicar el algoritmo optimizado las veces que se desee.

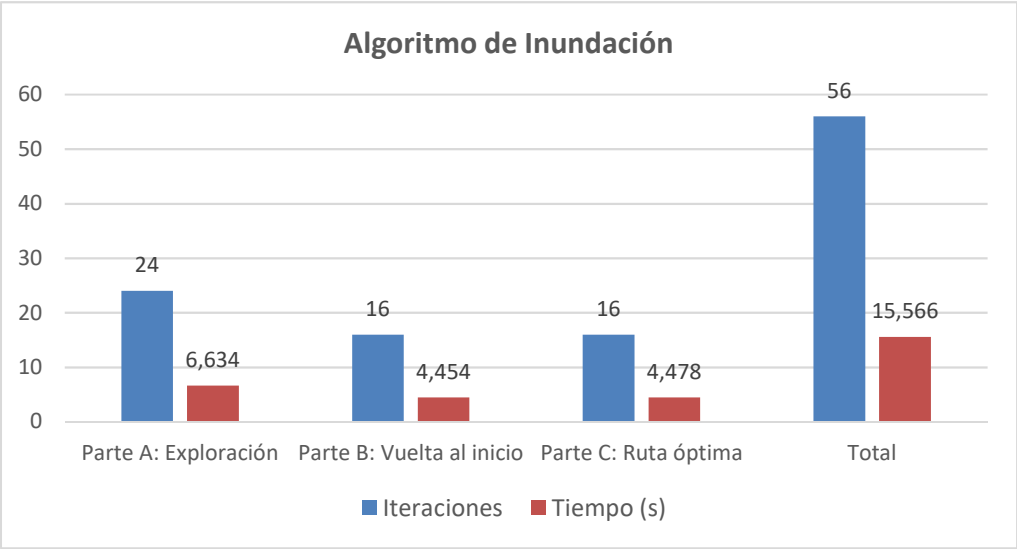


Ilustración 34. Resultados: Laberinto Virtual+Estándar - Algoritmo de Inundación

Aparte de ello, decir que es bastante **eficiente**, dado que la parte B y la C son básicamente **simétricas**, por lo que podría obtenerse la ruta óptima **a partir** de los resultados de B, ahorrando tiempo. De todas formas, se ha preferido hacer de esta forma porque resulta más **claro** de **entender** visual y conceptualmente.

6.2.2. Laberinto Real+Estándar

Para pasar del entorno simulado al entorno **real** se ha tratado de hacer los **mínimos cambios** en su lógica, para poder compararlos de la forma más **similar** posible, pero tal y como se ha comentado, el robot avanzará de **2 en 2** celdas, ahorrando **tiempo** de desplazamiento, ya que el diseño del laberinto lo **permite**.

Y también se ha aprovechado y se ha desarrollado una versión **optimizada** de los algoritmos que gobiernan al robot en el entorno **real** porque, si bien en la simulación **no** existe **diferencia** en comprobar todas las casillas alrededor del robot porque resulta casi **inmediato**, en el entorno real **sí**, debido a que el sensor de **ultrasonidos** tarda cierto **tiempo** en comprobar **distancias**.

Así, y tomando como ejemplo el algoritmo de la **RMI**, se seguirá esta **secuencia**:

1) En caso de no encontrar obstáculo a la **izquierda**, **no** se comprobarán más distancias.
2) De haberlo, se comprobará el **frente**, y **no** se seguirá comprobando más.
3) Si lo hay al frente, se comprueba a la **derecha**, y **nada más**.
4) En caso de tener obstáculos **alrededor**, el robot girará sobre sí mismo y dará **media vuelta**.

De esta forma, el robot se **ahorrará** cierto **tiempo** de comprobación de **distancias**, acortando el total que emplea para resolver el laberinto.

Decir que para la **RMD** y el **RMI optimizado** también se ha implementado dicha **mejora** alternativa.

Aleatorio o "del ratón"

Al igual que para el caso simulado, se debe ejecutar **varias** veces para extraer conclusiones, algo a tener bastante en cuenta porque los **tiempos** oscilan de forma **considerable**, tanto de exploración como de navegación por el laberinto. Y, debido a tal hecho, las **baterías** han **sufrido** más de lo debido, teniendo que parar a **recargarlas** en alguna ocasión, aunque era algo de suponer.

Los **resultados** han sido los de la *Ilustración 35*:

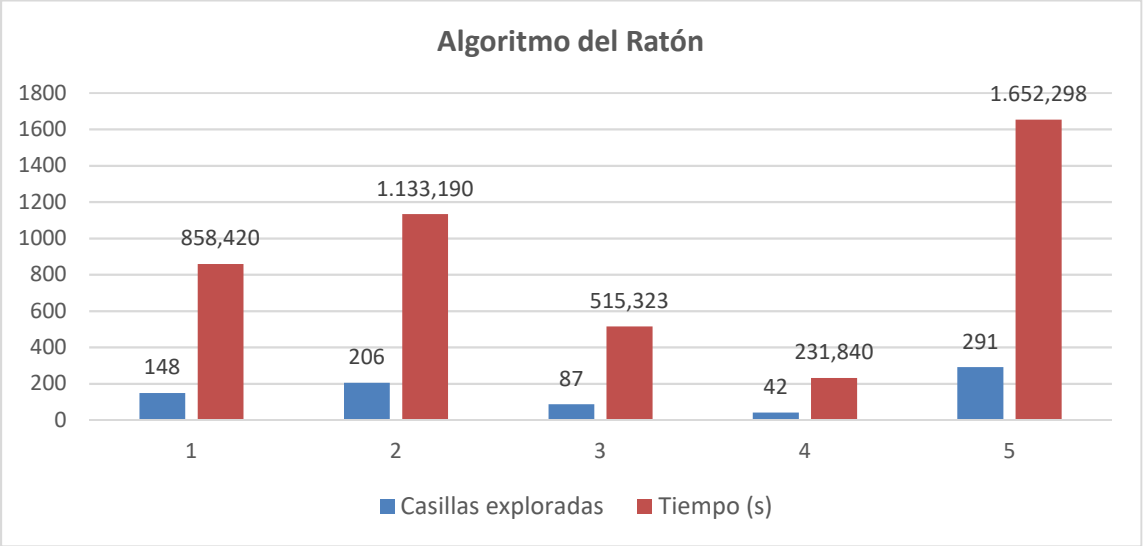


Ilustración 35. Resultados: Laberinto Real+Estándar - Algoritmo del Ratón

Como puede observarse, los **resultados** obtenidos son **demasiado** aleatorios. Y los **tiempos**, evidentemente, han sido **muy superiores** a los de las simulaciones, debido a la necesidad de **explorar** y **navegar** físicamente por el laberinto.

Por ello, explorar cada **casilla** ha conllevado un **tiempo** que ha oscilado entre los **5,5** y los **5,8** segundos aproximadamente, contando con el tiempo de computación necesario.

Seguidor de paredes *(Wall-Follower)*

Su funcionamiento es igual de **predecible** que para el caso simulado. Por ello, tampoco necesitaremos de realizar varias pasadas.

Siguiendo la **RMI**, el **tiempo** empleado y las **casillas exploradas** han sido los de la *Ilustración 36*:

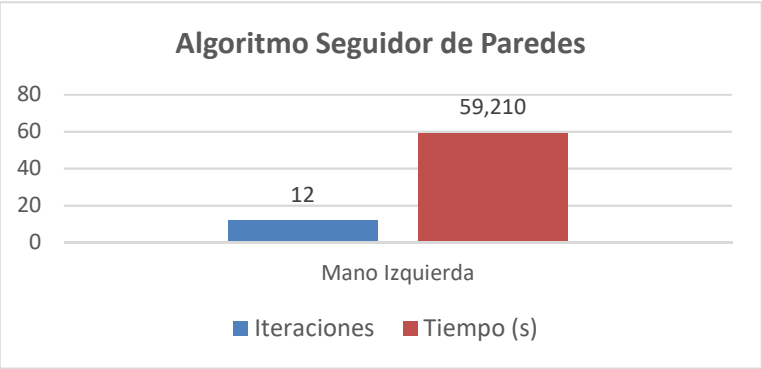


Ilustración 36. Resultados: Laberinto Real+Estándar - Algoritmo Seguidor de Paredes empleando la RMI

Como era de esperar debido al diseño de este laberinto, es bastante mejor emplear la RMI. De no hacerlo, exploraría innecesariamente la otra mitad del laberinto. Y es patente el tiempo **ahorrado** si se emplea la estrategia de avanzar de **2 en 2** celdas que, como era de esperar, se queda en la **mitad** (12 casillas, no 24)

Notar que el tiempo de computación es algo **inferior** al caso anterior, situándose en **4,9** segundos. Es lógico suponer que el hecho de introducir **aleatoriedad** a la hora de tomar decisiones obligue a que el robot deba **comprobar** primero su resultado y, posteriormente, ponerlo en **práctica**.

Seguidor de paredes optimizado *(Optimized Wall-Follower)*

Para este algoritmo se debe recordar que se divide en **dos partes**, la que es común a la **RMI** y la que llega a la meta recorriendo la ruta **óptima**. Por ello, éstos serán los tiempos que tomaré para observar resultados.

Así, tendremos los resultados que pueden verse en la *Ilustración 37*:

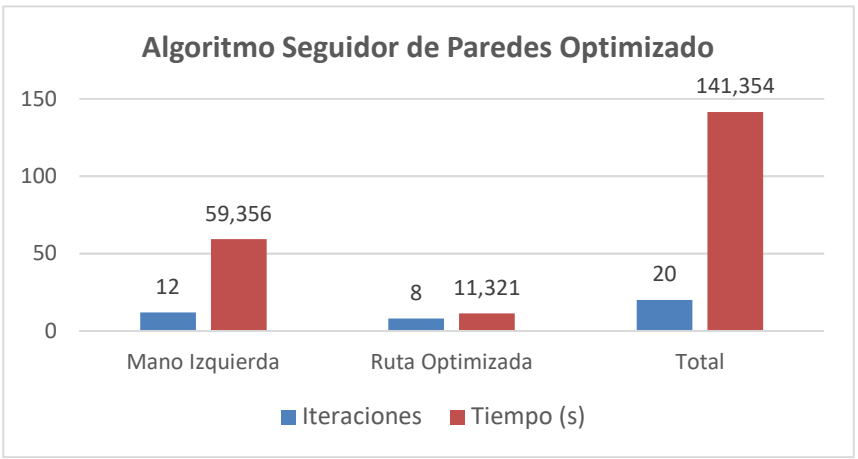


Ilustración 37. Resultados: Laberinto Real+Estándar - Algoritmo Seguidor de Paredes Optimizado

Sus resultados son aún **más dispares** que en el entorno simulado. El hecho de que el robot **no** tenga que **comprobar** dónde hay obstáculos y que pueda avanzar a **máxima velocidad**, dado que conoce cuáles son las líneas rectas a seguir, logra que **recorte** tiempos de forma muy notable.

Inundación (Flood-Fill)

No se puede conocer exactamente cuánto **tiempo** lleva cada una de las **tres partes** de las que se compone el algoritmo, por lo que se tomarán como **referencia** los tiempos del caso simulado, ya que deberían ser muy similares, incluso idénticos, al **no** tener que **desplazarse** el robot.

Así, los **resultados** obtenidos han sido los que pueden observarse en la *Ilustración 38*:

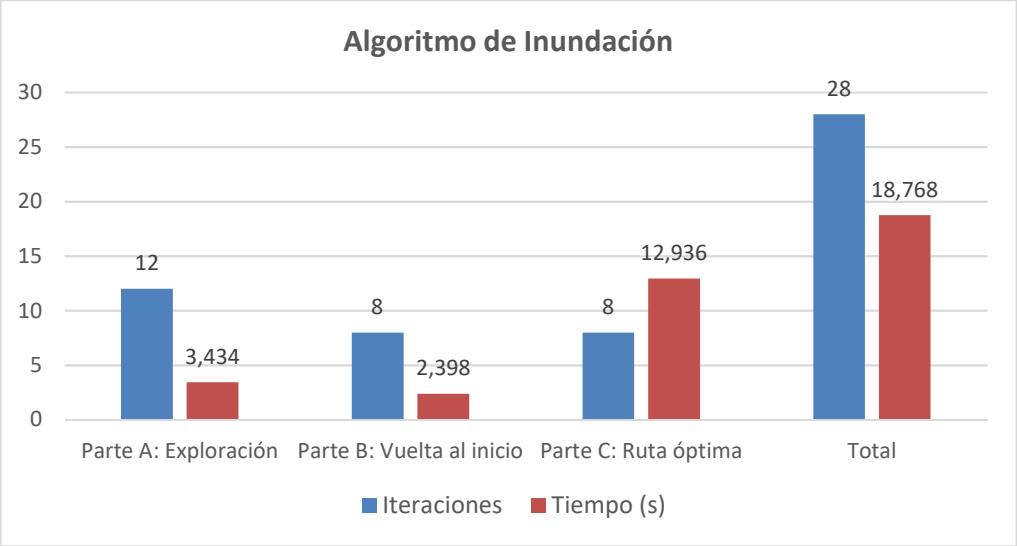


Ilustración 38. Resultados: Laberinto Real+Estándar - Algoritmo de Inundación

Con este algoritmo se pone de relevancia la **diferencia** de que el robot posea información **previa** sobre el laberinto, ya que el tiempo que se pierde explorando y navegando es **considerable**. Pero, una vez más, el tiempo de recorrido se reduce a la **mitad**, al avanzar de **2 en 2** celdas (8 en lugar de 16 de la ruta óptima).

Por ello, sería muy interesante que, en caso de que las condiciones fueran a ser las **mismas** en veces sucesivas, se explorase **previamente** el laberinto para encontrar la ruta óptima.

6.3. Análisis de los resultados

Tal y como ya se ha comentado, se ha empleado el **mismo** laberinto y las **mismas** condiciones para que los resultados no se vieran afectados por posibles variaciones en el trazado, el equipo empleado o los materiales, por poner algunos ejemplos. Gracias a ello, podrán extraerse **conclusiones** realistas, que servirán para tomar nota de los aspectos que pueden **optimizarse** y **mejorarse**.

Laberinto	Virtual+Estándar		Real+Estándar	
Algoritmo	**Iteraciones**	**Tiempo (s)**	**Iteraciones**	**Tiempo (s)**
Aleatorio	223,8	61,79	154,8	831,84
RMI	12	3,433	12	59,20
RMI Optimizado	12 + 8 (opt)	3,433 + 2,38	12 + 8 (opt)	59,35 + 11,32
Inundación	12 + 8 + 8	3,43 + 2,39 + 2,39	12 + 8 + 8	3,43 + 2,39 + 12,93

Tabla 1. Algoritmos Laberínticos: Resultados medios

En lo que respecta a los **algoritmos** (ver *Tabla 1*), puede decirse que:

a) **Aleatorio** o **del "Ratón"**: Debe emplearse únicamente como **último** recurso. Su **aleatoriedad** puede llegar a ser **desesperante**, con unos **tiempos** demasiado **elevados** y con la posibilidad de **agotar** las **baterías** incluso antes de llegar a la meta.

b) **Seguidor de Paredes**: Los algoritmos de la RMD y la RMI deberían **probarse antes** de decantarse por cualquiera de ellos. Es muy posible que, si uno no resulta satisfactorio en un laberinto dado, el otro sí lo haga, por lo que conviene tenerlo en cuenta.

c) **Seguidor de Paredes Optimizado**: La optimización debe hacerse **sólo** si se pretende trabajar sobre el mismo laberinto en veces **sucesivas** y **no se cambian** sus parámetros iniciales. Si es así, resultan perfectos para alcanzar la meta de forma rápida y óptima.

d) **Inundación**: También resulta muy útil en determinadas condiciones. El hecho de ser informado permite **no** hacer **avanzar** el robot **hasta** conocer cuál es la ruta óptima, lo que **ahorrará** mucho tiempo de exploración y gasto de baterías. Este hecho se ha reflejado implementando **dos** algoritmos de inundación, uno que obliga al robot a explorar el laberinto y otro que sólo lo hace de forma virtual, para luego avanzar hasta la meta físicamente.

En cuanto a las **funciones** desarrolladas para el robot, puede decirse que:

- **Mando IR**: El manejo es relativamente sencillo, pero la posibilidad de **interferir** con componentes como servos o sensores en una placa Arduino Uno es algo a tener **muy en cuenta**. Tanto es así, que para el entorno **real** se ha preferido que el robot ejecutase los algoritmos **uno a uno**, para **evitar** que el *timer* del mando IR introdujera variaciones en los motores y arruinasen la fase de pruebas.
- **Servo**: Se ha comportado bastante bien, pero requiere de una **calibración** previa para saber exactamente hacia dónde mira. Si no se hace, puede que "izquierda" o "derecha" no lo sean.
- **Sensor** de **ultrasonidos**: Es bastante **preciso**, porque ha llegado a medir distancias que coincidían al centímetro con las medidas de un metro real.
- **Sensores** de **líneas**: Es preferible manejarlos de forma **digital**, con "1" y "0" para detectar o no líneas. En caso de emplear señales PWM se corre el riesgo de que no se hagan mediciones correctas

por el simple hecho de detectar un pequeño pliegue en los **materiales** empleados para guiar al robot, como la cinta aislante negra, que funciona relativamente bien.

- **Motores**: Realmente no son necesarios cuatro y con dos habría bastado, pero se ahorra tiempo a la hora de mover el robot, porque son bastante **potentes**. Ahora bien, deben emplearse señales **PWM** o se corre el riesgo de que su movimiento **se descontrole**.
- **Controlador PID**: Ha funcionado de forma muy **satisfactoria**, ya que puede corregir giros, avances, etc., de forma **suave** y **controlada**, por lo que hay que habrá que tenerlo muy en cuenta.

6.4. Mejoras propuestas

A medida que se han ido realizando las pruebas, tanto en el entorno simulado con en el real, he ido haciendo diversas **anotaciones**, bien por ser algún tipo de **mejora** sobre la base existente o por posibilitar otro tipo de enfoque o **utilidad** para un robot que resuelva laberintos.

En cuanto a los **algoritmos**, se hace patente plantear mejoras que incidan en ciertas **carencias** que, si bien para el laberinto planteado hasta ahora no suponen diferencia, en el momento en el que se encuentren con un trazado más complejo, para el que no han sido diseñados, **no podrán** encontrar solución alguna.

Así, algunas de las **características** de un laberinto que, a priori, podrían resultar conflictivas son:

- **Bucles**: Si existe un recorrido que no acabe llevando a la salida, y que lleve al robot una y otra vez por el mismo camino, tendremos un problema al que no se podrá dar solución.
- **Salida "aislada"**: Si la salida no está en contacto con ninguna pared adyacente, los algoritmos planteados van a tener problemas para llegar hasta ella.
- **Cruces "trampa"**: Pueden existir laberintos con cruces de caminos en los que el robot no sepa cuál es la mejor dirección a seguir, por lo que unas veces podrían encontrar la salida y otras no.

En cuanto a los **componentes** mejorables, destacan los siguientes:

a) **Placa**: Una placa más **capaz** que la Uno sería perfecta. Más memoria, más velocidad, más posibilidades de conexionado, más *timers* para más dispositivos…

b) **Sensores de ultrasonidos**: Convendría disponer de, al menos, **tres sensores**. Situados uno de frente y dos a los lados sería lo ideal, aunque un **cuarto** estaría bien en la parte de detrás del robot. De esta forma, se podría prescindir del guiado por líneas cuando el robot avance en línea recta, ya que se tendrían mediciones de distancia continuas a ambos lados y se podría corregir su rumbo sobre la marcha de forma natural. Además, el cuarto sensor, situado en la parte posterior, podría tomar medidas en caso de cambiar de sentido de forma más rápida, sin necesidad de girar el robot.

c) **Servo**: Si se dispusiera de más sensores de ultrasonidos, podría **prescindirse** del servo, porque no habría que girar ninguno para obtener mediciones de otros puntos.

d) **Sensores de líneas**: Para un guiado por líneas óptimo, convendría disponer de **cinco** o incluso **siete sensores**, para detectar movimientos y desviaciones aún más pequeñas. Aparte de ello, los cruces y detenciones resultarían más naturales, situando los sensores con el mayor ancho posible.

e) **Ejes** y **ruedas**: Existen **acoplamientos** más profesionales que los que vienen en el kit de Elegoo, y resultarían perfectos para disponer de más par, potencia y control sobre las ruedas. Además de ahorrar tiempo desmontando los motores y tratando de desarrollar otro tipo de acople.

f) **Posicionamiento**: Afortunadamente, una vez arreglados los ejes de las ruedas y los motores, y gracias a medir tiempos, se han podido controlar bastante bien los giros del robot. Pero para hacerlos con más precisión y para posicionar mucho mejor el robot sobre el laberinto, sería preferible optar por dos componentes que suelen funcionar muy bien: ***encoders*** para las ruedas y una **brújula digital** para el robot. Con esta combinación se corregiría aún más la velocidad de las ruedas y el robot podría conocer en todo momento su dirección, sin necesidad de ajustar casi "artesanalmente" los grados que gira, como con el kit de Elegoo.

g) **Pantalla LCD**: Situada en la parte posterior del robot, sería perfecta para mostrar mensajes informativos sobre el robot, como qué algoritmo está empleando, qué giro está efectuando, por qué lugar del laberinto está yendo en ese momento, etc.

h) **Bluetooth**: Es posible que los comandos que ahora se envían a través del mando IR pudieran enviarse mejor a través de **Bluetooth**, sin interferencias y sin necesidad de apuntar al robot.

i) **Baterías**: La **conexión y alimentación** debe ser mejor, quizá mediante cables soldados y mejor protegidos que los que vienen en el kit.

Como hemos visto, existen múltiples **opciones** que podrían aportar un comportamiento mejorado y/o potenciado al robot. Todas son interesantes, pero algunas son más necesarias que otras, por lo que aportan, posibles incompatibilidades o si son realmente viables, por prestaciones, tamaño o, sencillamente, por coste económico, que debería ser lo más contenido posible.

Así pues, veamos un poco más en detalle las **propuestas** que hemos planteado.

6.4.1. Arduino Mega 2560 Rev.3

Es una placa de desarrollo basada en el microcontrolador ATmega2560 (de ahí su nombre). Pertenece a la extensa familia de placas Arduino, siendo junto a la Arduino UNO de las más representativas, pero ha sido diseñada para llevar a cabo proyectos más **complejos**.

Sus **características técnicas** son las siguientes:

- Microcontrolador: ATmega2560
- Voltaje Operativo: 5V
- Voltaje de Entrada: 7-12V
- Pines digitales de E/S: 54 (de los cuales 15 proveen salida PWM)
- Pines analógicos de entrada: 16
- Corriente DC por cada Pin Entrada/Salida: 40 mA
- Memoria Flash: 256 KB (8KB usados por el *bootloader*)
- SRAM: 8KB
- EEPROM: 4KB
- Velocidad de reloj: 16 MHz

Lo más importante es que la placa Mega2560 está diseñada para ser **compatible** con la mayoría de los componentes diseñados para la Uno. Por ejemplo, los pines digitales 0 a 13 (y los pines adyacentes AREF y GND), las entradas analógicas 0 a 5, la alimentación y el ICSP[25] se encuentran todos en ubicaciones equivalentes. Además, el UART[26] principal (puerto serie) está situado en los mismos pines (0 y 1), al igual que las interrupciones externas 0 y 1 (pines 2 y 3 respectivamente). De todas formas, el I2C[27] no se encuentra en los mismos pines necesariamente porque, de hecho, existen dos pares de pines asociados.

En cuanto a las **conexiones** de la placa, son las de la *Ilustración 39*:

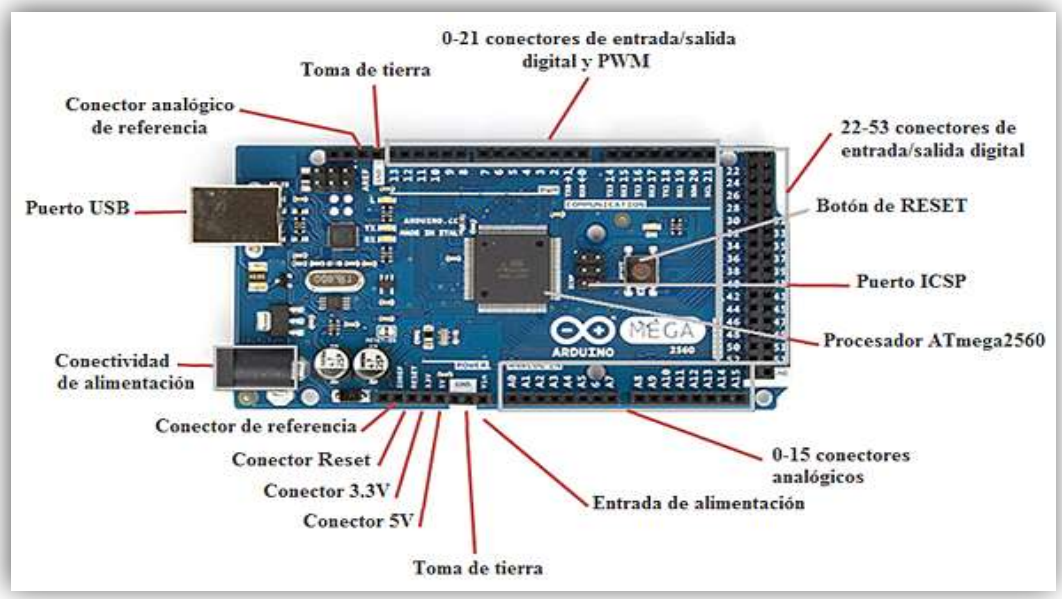


Ilustración 39. Arduino mega2560 Rev.3: Conexiones (Arduino CC, 2020)

[25] ICSP: *In-Chip Serial Programmer*
[26] UART: *Universal Asynchronous Receiver-Transmitter*
[27] I2C: *Inter-Integrated Circuit*

Conviene conocer también cuáles son los **pines** de **E/S** que podrían resultar útiles, destacando:

- **Analógicos**: Van desde el pin A0 al A15 y pueden comportarse como entradas.
- **Digitales**: Son 54, de los cuales 15 proveen salida PWM, y actúan como E/S.
- **0 RX** y **1 TX**: Son empleados para transmitir datos en serie a la propia placa.

En resumen, las **diferencias** más importantes entre la Arduino Uno y la Mega2560 son las de la *Tabla 2*:

Placa	Uno	Mega2560
Microcontrolador	ATmega328	ATmega2560
Memoria Flash	32 KB	256 KB
Memoria SRAM	2 KB	8 KB
Memoria EEPROM	1 KB	4 KB
Pines analógicos	6	16
Pines digitales	14	54
Pines digitales PWM	6	15
Timers	3	6

Tabla 2. Arduino Uno vs Mega2560: Diferencias

Su potencia, su mayor capacidad de memoria y, sobre todo, su amplio número de puertos de E/S resultarán perfectos para **mejorar** la "inteligencia" del robot y hacerlo así más capaz.

6.4.2. *Encoder* óptico FC-03

Conocer la **velocidad** de un motor, y por extensión de una posible **rueda** acoplada al mismo, es muy importante en robótica. Por ello, existen diversas alternativas, siendo una de las más comunes, y habitual en este tipo de proyectos, el uso de *encoders* de tipo **óptico**.

El FC-03 utiliza el opto interruptor infrarrojo MOCH22A, que cuenta con dos **partes**: un emisor IR y un receptor o sensor IR. Entre ambos existe un espacio para el objeto que bloqueará el paso de luz, típicamente un disco ranurado (ver *Ilustración 40*). Los pulsos son digitalizados por un **comparador** LM393 mediante pulsos TTL, que pueden ser interpretados por un microcontrolador como Arduino.

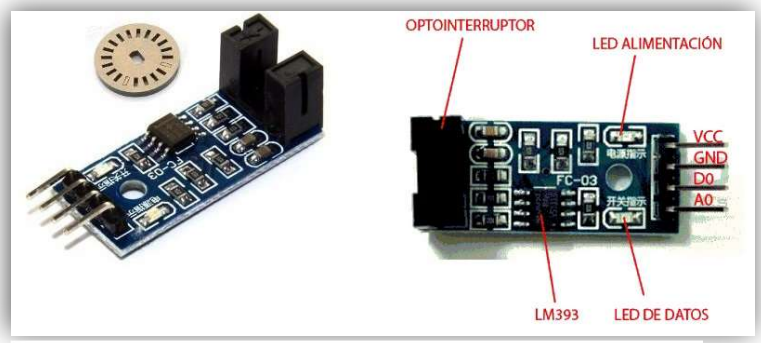


Ilustración 40. Encoder óptico FC-03 y disco ranurado (Elegoo, 2019)

Es recomendable utilizar **interrupciones** que empleen flancos de **subida/bajada** para detectar los pulsos, porque mejora la precisión de las medidas. En Arduino Uno y Mega2560 los pines de interrupción por flanco son el 2 y el 3. Se recomienda agregar un capacitor de 100nF entre la línea de salida D0 y tierra a modo de filtro pasa-bajo, y así evitar falsos disparos en la interrupción. Otra recomendación es alimentar el módulo con 3.3V, algo idóneo en Arduino.

Sus **características técnicas** son las siguientes:

- Voltaje de Operación: 3.3V - 5V DC
- Salidas: Analógica y Digital TTL
- Sensor: MOCH22A
- Modelo de placa: FC-03 / FZ0888
- Tipo de emisor/detector: Fotodiodo IR /fototransistor
- Peso: 8 gramos
- Dimensiones: 3.2*1.4*0.7 cm
- Ranura de 5mm
- Comparador óptico: LM393
- Led indicador de alimentación y pulso

Este modelo de *encoder* ha sido diseñado para trabajar con plataformas de **robótica móvil**, como el del caso que nos ocupa, aunque también son utilizados en contadores de RPM (revoluciones/minuto) en motores DC/AC, por si se desea conocer la velocidad de giro del motor al que van acoplados.

Por todo ello, y para maximizar el conocimiento de cada movimiento del robot, vamos a utilizar **cuatro** *encoders*, uno en cada rueda. De esta forma podremos conocer la velocidad de giro de cada rueda de cada eje, lo que nos permitirá ser mucho más **precisos** a la hora de calcular el giro exacto que deberá dar el robot cuando se encuentre con los obstáculos del laberinto o deba cambiar de rumbo, simplemente. Y esto, combinado con el controlador **PID**, debería proporcionar unas trayectorias mucho mejores para el robot.

6.4.3. Pantalla OLED SH1106

Es una pequeña pantalla monocromo de tan sólo 1.3" de diagonal y 128×64 píxeles. Pequeña pero muy **visible** desde cualquier ángulo gracias al contraste que le proporciona la tecnología OLED[28]. Está basado en el driver SH1106 y se comunica por el bus I2C, aunque también existe versión SPI[29].

[28] OLED: *Organic Light-Emitting Diode*
[29] SPI: *Serial Peripheral Interface*

I2C (Prometec, 2018) es un método de comunicación que **sólo** utiliza **4 pines** . En este protocolo, varios "maestros" (microcontroladores como Arduino) pueden ser conectados a varios "esclavos" (sensores, normalmente) usando 4 pines: VCC, GND, SDA y SCL, lo que nos **ahorrará** emplear unos cuantos cables y pines adicionales, que siempre viene bien (ver *Ilustración 41*).

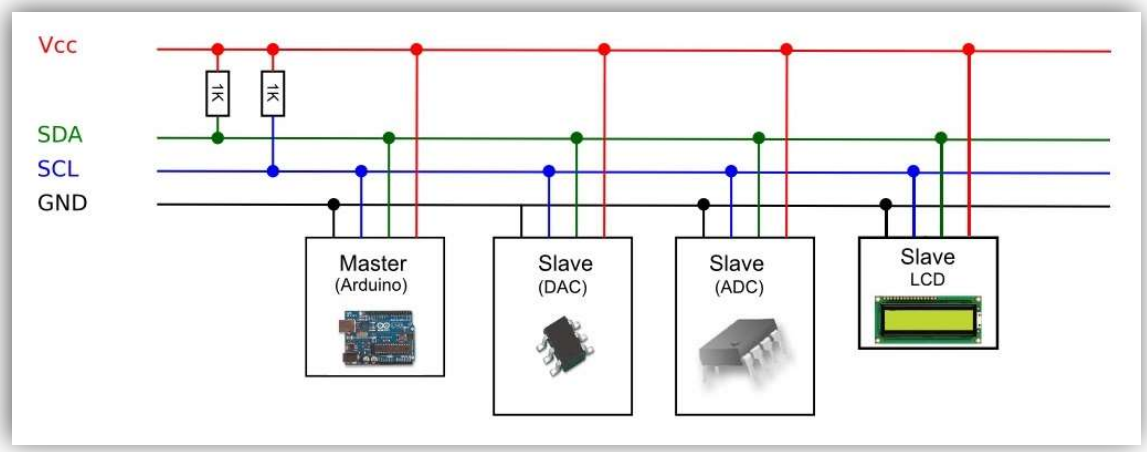


Ilustración 41. Protocolo I2C: Ejemplo (Prometec, 2016)

Comparadas con las LCD, las pantallas OLED tienen una serie de **ventajas** (ver *Ilustración 42*) tales como alto **brillo**, alta relación de **contraste**, **ángulo de visión** amplio, buena gama de **temperatura** y un **consumo** de energía bajo. Por contra, esta tecnología **no** es recomendable para un uso **continuado** (por ejemplo, relojes), pues sólo permite unos pocos miles de horas de funcionamiento antes de degradarse la imagen.

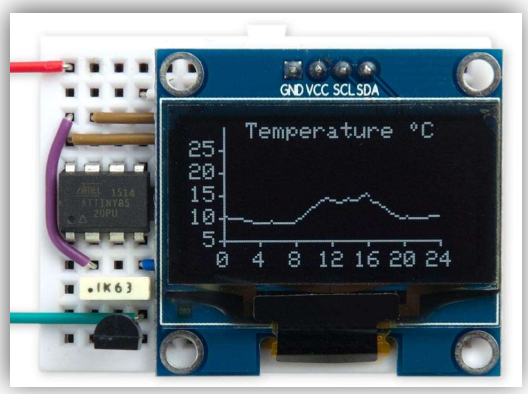


Ilustración 42. Pantalla OLED SH1106 en funcionamiento (Technoblogy, 2018)

Sus **características** técnicas son las siguientes:

- Tecnología: OLED monocromo (blanco)
- Tamaño: 1.3 pulgadas
- Resolución: 128 * 64
- Ángulo de visión: > 160 grados
- Driver: SH1106
- Interfaz: I2C

- Alimentación: 3.3V ~ 5VDC
- Corriente de trabajo: 25mA (máx.)

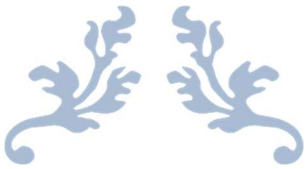
Como vemos, es compatible con una alimentación de **3.3v** y además cuenta con un consumo **muy reducido**, lo que la hace perfecta para integrarla en nuestro proyecto y que el robot nos vaya mostrando **información**.

6.4.4. Inventario de mejoras

La **relación** final de todas las mejoras que vamos a aplicar es la de la *Tabla 3*:

Mejora	Tipo	Unidades	Total
Placa	Arduino Mega2560	+1	1
Posicionamiento	Sensores de ultrasonidos HC-SR04	+2	3
	Sensores de líneas TCRT5000L	+2	5
	Encoder FC-03	+4 (uno por motor)	4
Información	Pantalla OLED SH1106	1	1

Tabla 3. Inventario de mejoras



7. APRENDIZAJE POR REFUERZO, LABERINTOS Y ARDUINO



7. Aprendizaje por Refuerzo, Laberintos y Arduino

7.1. Aprendizaje por Refuerzo *(Reinforcement Learning)*

El Aprendizaje por Refuerzo es una **rama** de las técnicas de **Aprendizaje Automático** (Luna González, 2018) o *Machine Learning* (TechVidvan, 2020). Estos sistemas de aprendizaje **exploran** y adquieren **datos** sobre el problema por propia iniciativa, diseñando **automáticamente** estrategias que le den solución.

Un sistema de Aprendizaje por Refuerzo (ver *Ilustración 43*) está formado por un "**agente**" inteligente que aprende a optimizar un proceso de decisión. Así, el agente interactúa con un "**entorno**", que puede ser el proceso de decisión real o una simulación del mismo. El agente trabaja observando el entorno, y tomando una **decisión** para comprobar qué **efectos** produce. Si el resultado de esa decisión es **beneficioso**, el agente aprende automáticamente a **repetir** esa decisión en el futuro, mientras que si el resultado fuera **perjudicial evitará** volver a tomar la misma decisión.

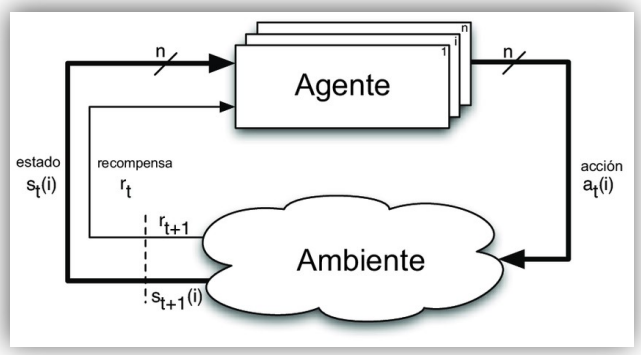


Ilustración 43. Aprendizaje por Refuerzo: Modelo

De este modo, sigue un proceso de aprendizaje inspirado en la psicología **conductista** (Berridge, 2001), cuya ocupación es determinar qué acciones debe escoger un agente en un entorno dado, con el fin de maximizar una noción de "**recompensa**". Es decir, al agente no se le dice qué acciones tomar, sino al contrario: él debe **experimentar** para encontrar qué acciones lo llevan a una mayor recompensa.

El **modelo** básico (ver *Ilustración 43*) consiste en:

a) Un conjunto de **estados** de entorno (**s**).
b) Un conjunto de **acciones** (**a**).
c) Reglas de **transición** entre estados.
d) Reglas que determinan la **recompensa** inmediata (**r**) de una transición.
e) Reglas que describen lo que **observa** el agente.

7.1.1. Elementos

Política

- Define el modo que el agente se **comporta** en un momento definido, es decir, "mapea" qué **acciones** se deberían llevar a cabo dados los **estados** que se han percibido del entorno. Representa el núcleo del agente y es suficiente para determinar el comportamiento de éste.

- Una política puede representar una **función** simple o una **tabla de búsqueda**, y en la mayoría de casos involucra un **cálculo** extensivo.

Recompensa

- La recompensa es lo que define los **objetivos** a corto plazo del problema que se quiere solucionar.
- Es uno de los recursos más **importantes** de la política, ya que, si una acción conlleva una recompensa baja, la política deberá variar para futuros eventos que involucren a esa acción en la misma situación.

Función de valor

- A diferencia de la recompensa, la función de valor se centra en los valores a **largo** plazo, representando la cantidad de recompensa **futura** que podría esperarse desde el estado actual.
- Es decir, se ocupa de introducir el concepto de "**valor**" y darle una mayor importancia respecto de una "recompensa", porque dichas acciones son las que conducirán a una mayor recompensa a largo plazo.

Modelo de entorno

- Ayuda a **inferir** resultados, ya que el modelo puede **predecir** el siguiente estado y recompensa.
- Normalmente los utilizan las políticas para planificar las acciones que se llevarán a cabo (*model-based policies*), pero también existen políticas **sin** un modelo establecido (*model-free policies*), que trabajan gracias al método de prueba y error, algo, en principio, opuesto a la planificación.

7.1.2. Objetivo

El objetivo de un agente que aprende por refuerzo es obtener tanta **recompensa** como sea posible. Está conectado con su entorno a través de la **percepción** y la **acción**. En cada paso, el agente recibe una entrada y alguna indicación del estado actual del entorno, tras lo que elige una acción siguiendo algún criterio de selección de acción, generándose una salida, que le servirá como **experiencia** para el futuro.

7.1.3. El algoritmo general

Los **pasos** que deberá dar el agente son los siguientes:

1) Observa el **entorno**.
2) Decide cómo actuar, empleando algún tipo de **estrategia**.
3) **Actúa** de acuerdo a esa estrategia.
4) Obtiene una **recompensa** o una **penalización**
5) Aprende de las **experiencias** y **refina** su estrategia.
6) Itera los pasos anteriores hasta encontrar una estrategia **óptima**.

Así, puede elegir **cualquier** acción en función de su historia, e incluso puede **aleatorizar** su selección de acciones, aunque la única manera de recopilar información sobre el medio ambiente es mediante la **interacción** con él. Aparte de ello, para poder actuar de la manera más óptima posible debe pensar acerca de las consecuencias a **largo** plazo de sus acciones. Por ello, el RL es especialmente adecuado para los problemas que incluyen un razonamiento a largo plazo frente a uno a corto plazo.

Y este modelo, y las **decisiones** secuenciales en forma de **estados/acciones** que se irán tomando, se pueden **caracterizar** y **modelar** mediante dos conceptos matemáticos que veremos a continuación: los **Procesos de Decisión de Markov** y las **Ecuaciones de Bellman** (Bellman, 1957).

7.2. El Aprendizaje Profundo *(Deep Learning)*

Es una forma de ML [30]que **entrena** a un sistema para que aprenda por sí mismo, a través del reconocimiento de **patrones**. Para lograrlo, el DL[31] utiliza una clase específica de concepto: las "**redes neuronales**". Estas redes están inspiradas en la red neuronal biológica del cerebro humano, lo que posibilita un proceso de aprendizaje más **avanzado** que los modelos básicos de Aprendizaje Automático.

Las principales **diferencias** entre ambas tecnologías, ML y DL, son:

- El Aprendizaje Automático utiliza algoritmos para analizar datos, aprender de ellos y tomar decisiones basadas **únicamente** en lo **aprendido**.
- El Aprendizaje Profundo estructura los algoritmos en capas, para crear una red neuronal artificial, que puede aprender y tomar decisiones **por sí misma**.
- Así, los algoritmos de Aprendizaje Profundo se caracterizan por el número de **transformaciones** aplicadas a la señal mientras se propaga desde la capa de entrada a la de salida.

Aun así, **no** existe un estándar de facto para el número de transformaciones (o capas) que convierte a un algoritmo en "profundo", pero la mayoría de investigadores en el campo considera que aprendizaje profundo implica **más de dos** transformaciones intermedias.

7.2.1. Redes Neuronales Artificiales

Las ANN están constituidas por entidades a las que llamamos "**neuronas**". Las neuronas reciben datos de entrada de otras neuronas, los **procesan**, y producen datos de salida que envían a otras neuronas, y así, con tantos niveles o capas como deseemos, hasta que la información llega a las últimas neuronas de la red,

[30] ML: *Machine Learning*
[31] DL: *Deep Learning*

que muestran el **resultado** que buscamos. Es, evidentemente, un intento de buscar un paralelismo con el comportamiento de las neuronas en un cerebro humano.

Las neuronas se **caracterizan** por tener (ver *Ilustración 44*):

a) **Peso**: Caracteriza a cada neurona, cuyos pesos son distintos.
b) **Umbral**: También es distinto, y tenemos que encontrar su valor óptimo para que la red funcione.
c) **Sigmoide**: Función matemática encargada de procesar los datos que llegan a cada neurona.

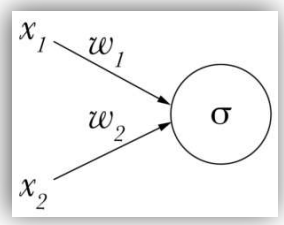


Ilustración 44. Neuronas: Datos (x), pesos (w) y sigmoide

Básicamente, una red neuronal (ver *Ilustración 45*) es como una gran **función de regresión adaptable** a cualquier circunstancia, en la que cada neurona por la que pasan datos es un factor de esa función. Pero los valores de las neuronas de salida nunca serán un 0 y un 1 absoluto, siempre habrá un margen de **error**. Y dicho error es **medible**, eligiendo el número de neuronas, pesos de cada conexión y sigmoides adecuados para que el error sea el menor posible, probando valores hasta dar con el mejor resultado.

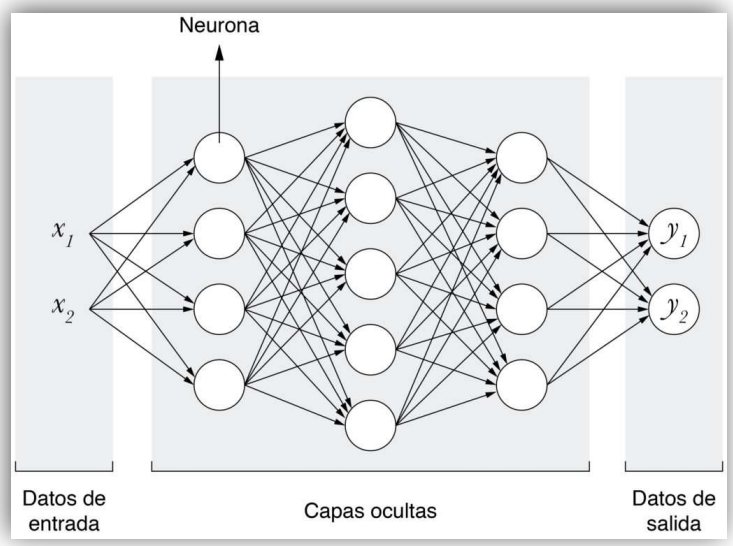


Ilustración 45. Red neuronal

A este proceso se le llama "**entrenar** la red", y para hacerlo necesitamos un conjunto de **datos** cuyo resultado **conozcamos**. Pero entrenar una red requiere muchos **datos** y bastante **tiempo**, aunque existen técnicas para que el proceso de prueba y error sea más eficiente que depender enteramente del azar.

7.3. Objetivo: El laberinto

Según hemos visto, dos **supuestos** importantes de los algoritmos de RL son:

a) Un **entorno estacionario**, es decir, las probabilidades de realizar transiciones de estado o de recibir una recompensa específica no cambian con el tiempo.
b) Todo el **espacio de estados** se puede **enumerar** y **almacenar** en memoria.

Es decir, son requisitos que **encajan** a la perfección con un **laberinto**, ya que este tipo de entorno es lo suficientemente limitado como para permitirnos enumerar todos los estados posibles, pero, al mismo tiempo, lo suficientemente rico como para permitirnos evaluar los algoritmos en diferentes situaciones, que es exactamente el objetivo que se persigue en esta parte del trabajo.

Si ahora **aplicamos** la idea de RL, el laberinto tendrá las siguientes **características**:

- **Formado** por:
  - **Posiciones**/estados (**s**)
  - **Acciones** (**a**)
  - Posiciones/estados **resultantes** tras ejecutar una acción (**s'**)
  - **Recompensas** (**r**)
- Estructura de **cuadrícula**, de un número arbitrario de celdas/casillas.
- De entre las **celdas** posibles, hay dos **especiales**:
  1. El inicio/**entrada**, desde donde el agente iniciará su camino.
  2. La meta/**salida**, la recompensa final.
- El **objetivo** es navegar por el laberinto desde el punto de partida hasta la meta.
- Sus celdas tienen dos **valores** posibles:
  1. **Obstáculo**/pared.
  2. Celda **vacía**, donde el agente puede situarse.

- Desde cada posición/estado (s) del laberinto hay 4 **acciones** (a) posibles, que cambian el estado resultante (s'): Norte, Sur, Este y Oeste.
- El estado **no cambiará** si un movimiento trata de situar al agente en una pared o fuera del laberinto, ya que en cuyo caso no se moverá.
- La recompensa (r) es **cero** para todas las transiciones, **excepto** para las que se encuentran en estados (s) objetivo, es decir, las que conducen a recompensas (r), como la **salida**.

Así, el ciclo que empleará el robot/agente encargado de recorrer el laberinto, es decir, la **secuencia** de estados, acciones y recompensas, es la que puede verse en la *Ilustración 46*:

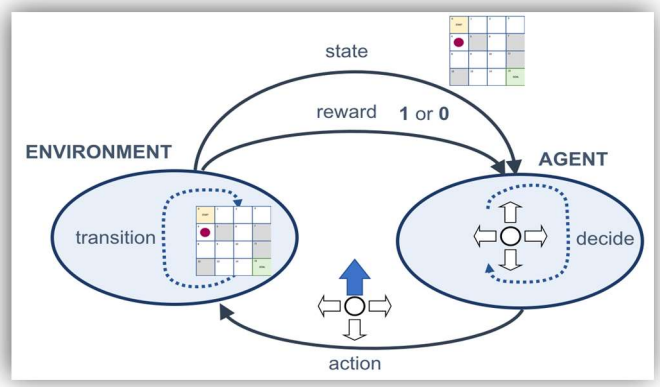


Ilustración 46. Aprendizaje por Refuerzo en un laberinto (Torres, 2020)

Pero la acción que el agente escoja no deberá depender sólo de la recompensa que vaya a recibir a **corto** plazo, sino la que a **largo** plazo le proporcione la máxima recompensa posible en su recorrido (o "**episodio**"). Para ello, tendrá que emplear una función de valor $V(s)$, una **estimación** de la recompensa que obtendrá el agente hasta el final del episodio, empezando desde el estado s. Si conseguimos estimar este valor correctamente, podremos decidir ejecutar la acción que nos lleve al estado con el valor **más alto**.

Teniendo todo ello en cuenta, de entre los algoritmos de RL, nos vamos a centrar en lo que se conoce como **Aprendizaje por Diferencia Temporal** o **TDD**[32].

7.4. Aprendizaje por Diferencia Temporal

Es un algoritmo de Aprendizaje por Refuerzo **sin** modelo. Esto significa, como se ha comentado con anterioridad, que el agente aprende a través de la **experiencia** real en lugar de a través de una tabla de transición entre estados, su "manual de instrucciones". Y se **actualiza** el conocimiento del agente en **cada** paso temporal (acción) en lugar de en cada episodio (alcanzar la meta o el estado final).

La idea general, en cuanto a la **actualización** del valor de un estado, es la de la *Ecuación 1*:

$$V(s_t) \leftarrow V(s_t) + \alpha\left[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)\right]$$

Ecuación 1. Diferencia Temporal: Actualización de valores de los estados (Sutton & Barto, 2015)

Lo que viene a **significar**:

Nuevo_Valor = Valor_Actual + Tasa_de_Aprendizaje[Valor_Objetivo – Valor_Actual]

[32] TDD: *Temporal Difference Learning*

Así, el agente **no** tiene conocimiento del estado, las recompensas y las transiciones. **Interactúa** con el entorno (realiza acciones aleatorias o informadas) y **aprende** nuevas estimaciones (valores de pares estado-acción), **actualizando** su conocimiento del entorno continuamente **después** de tomar cada acción.

Su **algoritmo** general es el de la *Ilustración 47*:

Initialize $V(s)$ arbitrarily, π to the policy to be evaluated
Repeat (for each episode):
  Initialize s
  Repeat (for each step of episode):
    $a \leftarrow$ action given by π for s
    Take action a; observe reward, r, and next state, s'
    $V(s) \leftarrow V(s) + \alpha\left[r + \gamma V(s') - V(s)\right]$
    $s \leftarrow s'$
  until s is terminal

Ilustración 47. Diferencia Temporal: Algoritmo general (Sutton & Barto, 2015)

Pero, ¿**cómo** elegirá el agente las acciones, es decir, **qué acción** tomará el agente en un estado (política) en particular? Aquí es donde entran en juego **SARSA** y ***Q-Learning***, las dos políticas de control sin modelo (ver *Ilustración 48*) que **elegiremos** para guiar a nuestro agente por el laberinto.

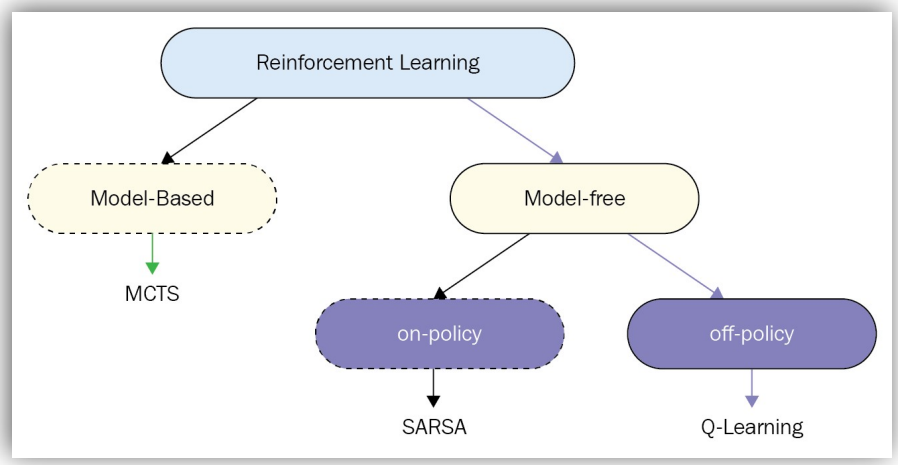


Ilustración 48. Aprendizaje por Refuerzo: Tipos de modelos (Packtpub, 2018)

La Tabla-Q

En realidad, es el nombre que se le da a una simple **tabla de búsqueda** donde se calculan las **recompensas futuras** máximas esperadas por acción en cada estado. Como ya se ha comentado, esta tabla **guiará** a nuestro agente/robot hacia la mejor acción en cada estado.

Cada **entrada** de la Tabla-Q será la recompensa futura máxima esperada que obtendrá el robot si realiza una **acción** concreta en un **estado** concreto. Éste es un proceso **iterativo**, ya que necesitamos **mejorar** la Tabla-Q en cada iteración. Notar que normalmente las **columnas** son las **acciones** y las **filas** son los **estados**.

Ahora bien, ¿cómo **calculamos** los valores de la Tabla-Q? ¿Los valores están disponibles o predefinidos? Para ello, tal y como apuntamos anteriormente, se van a emplear los algoritmos **SARSA** y ***Q-Learning*** y sus funciones de actualización/evaluación correspondientes.

Las Funciones-Q

Cuando **comenzamos**, todos los valores de la Tabla-Q son **ceros**. Mediante un proceso **iterativo** de actualización de valores, mientras se explora el entorno, la Función-Q nos brinda cada vez mejores aproximaciones al **actualizar** continuamente los valores Q en la tabla.

El **proceso** de **actualización** es el que puede observarse en la *Ilustración 49*:

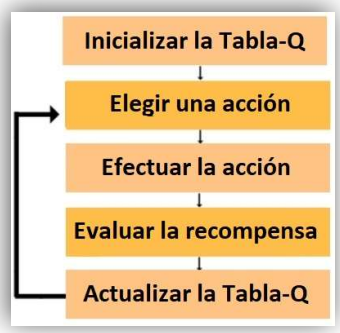


Ilustración 49. Tabla-Q: Proceso de actualización

7.4.1. SARSA *(State-Action-Reward-State'-Action')*

Es un método de control de Aprendizaje por Diferencia Temporal que, en este caso, emplea una **política** (*on-policy*), es decir, tuplas de pares estado-acción que **mapean** las acciones a tomar en cada estado, siendo capaz de este modo de **calcular** el valor de la política que se está siguiendo.

El objetivo es **estimar** $Q\pi(s,a)$ para la política actual π y todos los pares estado-acción (s-a), siendo Q un mapeo entre un par estado-acción y un número real, que denota su **utilidad**. SARSA y Q-Learning son métodos que tratan de evaluar el valor Q **óptimo** para todos los pares de acción-estado.

La **regla** de actualización para SARSA es la de la *Ecuación 2*:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\left[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\right]$$

Ecuación 2. SARSA: Regla de actualización (Sutton & Barto, 2015)

Es decir, si un estado S es terminal (estado objetivo o estado final), entonces $Q\ (S,a)\ =\ 0\ \forall\ a\ \in\ A$, donde A es el conjunto de todas las acciones posibles.

En cuanto al **algoritmo** que sigue SARSA, puede verse en la *Ilustración 50*:

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Initialize $Q(s,a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state},\cdot) = 0$
Repeat (for each episode):
  Initialize S
  Choose A from S using policy derived from Q (e.g., ϵ-greedy)
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ϵ-greedy)
    $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma Q(S',A') - Q(S,A)\right]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

Ilustración 50. SARSA: Algoritmo (Sutton & Barto, 2015)

De este modo, una **experiencia** en SARSA tiene la forma $[S, A, R, S', A']$, lo que significa que el agente estaba en el estado S, realizó la acción A, recibió la recompensa R y terminó en el estado S', momento en el que decidió hacer la acción A'. Y ésto **proporciona** una nueva experiencia para actualizar $Q\ (s, a)$, con un nuevo valor $R\ +\ \gamma Q\ (S', A')$, aportado por dicha experiencia.

En el algoritmo de la *Ilustración 50*, la acción A' sucede siguiendo una política, ya que SARSA es un método ***on-policy***. La política aplicada a los valores Q suele ser la denominada ***ε-greedy*** ("ávida") con pasos aleatorios (ver *Ilustración 51*), pero dicha política *ε-greedy* podría **cambiar** a una más conservadora si se detectan potenciales penalizaciones futuras, a **diferencia** de Q-Learning, que aplicará **siempre** la misma.

Política *ε-greedy*

1) Se genera un número aleatorio r ∈ [0,1]
2) Si r <ε se elige la acción a seguir gracias a los valores de Q (de la máxima utilidad)
3) De lo contrario, se elige una acción aleatoria

Ilustración 51. Política ε-greedy

En una política, como la de la *Ilustración 51*, el valor de ε determina el **ratio** de exploración/explotación:

- Si ε es grande, el número aleatorio r casi nunca será mayor que ε, por lo que **casi nunca** se ejecutará una acción **aleatoria**. Es decir, menos exploración y más **explotación**.
- Si ε es pequeño, el número aleatorio r a menudo será mayor que ε, lo que hará que el agente elija más acciones **aleatorias**. Esta característica estocástica permitirá al agente **explorar** aún más el entorno.

Como regla general, se elige ε normalmente para que sea **0.9**, pero se puede **variar** según el tipo de entorno. En algunos casos, ε se trata de **equilibrar** con el tiempo para permitir una mayor exploración tras varios episodios de una mayor explotación.

7.4.2. Q-Learning

Es un método de control de Aprendizaje por Diferencia Temporal al igual que SARSA, pero en este caso es ***off-policy***. Es decir, **no** sigue una **política** para encontrar la siguiente acción A', sino que **siempre** elige la acción a seguir de forma "voraz" o ***greedy***.

La regla de actualización que sigue Q-Learning puede verse en la *Ecuación 3*:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha(r_{t+1} + \gamma max_a Q(s_{t+1}, a) - Q(s_t, a_t))$$

Ecuación 3. Q-Learning: Regla de actualización (Sutton & Barto, 2015)

En cuanto al **algoritmo** que sigue el método Q-Learning, es el de la *Ilustración 52*:

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Initialize $Q(s,a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(terminal\text{-}state, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Take action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q(S',a) - Q(S,A)\right]$
    $S \leftarrow S'$
  until S is terminal

Ilustración 52. Q-Learning: Algoritmo (Sutton & Barto, 2015)

Tal y como puede observarse, siempre trata de buscar las celdas con un **mayor valor** Q, las que hacen que sea su valor sea máximo, **sin importarle** cualquier otra consideración (es "ávido").

Así, Q-Learning aprenderá una política óptima sin importarle lo que haga el agente, siempre que explore lo suficiente. Pero **podría** haber casos en los que dicho comportamiento sea "**peligroso**", con grandes recompensas negativas si no se tienen en cuenta las posibles consecuencias de tomar un determinado camino, más allá de las recompensas inmediatas.

7.4.3. SARSA vs Q-Learning

Como ya hemos comentado, SARSA y Q-Learning coinciden en muchos aspectos, y SARSA podría verse como una generalización de Q-Learning. Pero, más allá de eso, ¿en qué se **diferencian**?

En **Q-Learning**:

1) El agente comienza en el estado S.
2) Realiza la acción A.

3) Ve cuál es la recompensa R más alta posible por realizar cualquier acción desde su nuevo estado.
4) Y actualiza su valor para S/A gracias a un nuevo valor más alto posible.

En cambio, en **SARSA**:

1) El agente comienza en el estado S.
2) Realiza la acción A y obtiene una recompensa R.
3) Pasa al estado S'.
4) Realiza la acción A' y obtiene una recompensa R'.
5) Regresa al estado S para actualizar el valor de S/A en función del valor real de la recompensa que obtuvo al realizar la acción A'.

Un ejemplo muy famoso para ilustrar sus diferencias es el del "**Paseo por el Acantilado**" o *"Cliff Walking Example"* de la *Ilustración 53*:

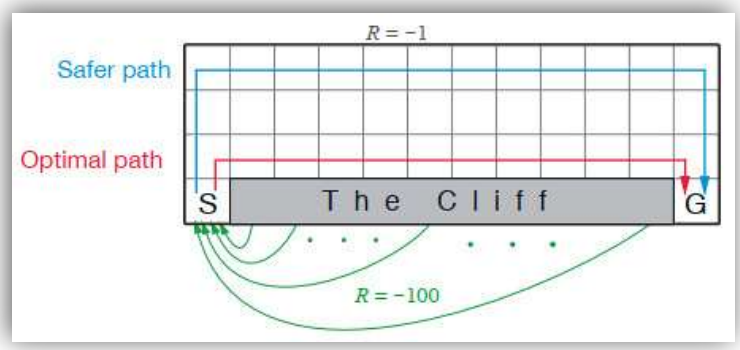


Ilustración 53. Cliff Walking Example (Sutton & Barto, 2015)

En él, puede observarse que hay una penalización de -1 por cada paso que da el agente y una penalización de -100 por caer por el precipicio. El camino **óptimo** es, por tanto, correr exactamente a lo largo del **borde** del acantilado y alcanzar la recompensa lo más rápido posible. Esto **minimiza** el número de pasos que da el agente y maximiza su recompensa siempre que **no se caiga** al precipicio en ningún momento.

Q-Learning tomará el camino **óptimo**, mientras que SARSA tomará el camino **seguro**. El resultado es que, si se emplea una política *ε-greedy* u otra política basada en la exploración, existe cierto **riesgo** de que en cualquier momento un agente que está empleando Q-Learning se caiga por el precipicio, como resultado de elegir explorar las zonas afectadas.

SARSA, a diferencia de Q-Learning, **anticipa** la siguiente acción para ver qué hará (o podría hacer) el agente en el siguiente paso, y actualiza el valor Q de su par estado/acción actual en consecuencia. Por esta razón,

aprende que el agente podría caer por el precipicio y que ésto conduciría a una gran recompensa negativa, por lo que **reduce** los valores Q de esos pares de acción/estado en consecuencia.

El resultado es que Q-Learning **asume** que el agente está siguiendo la mejor política posible sin intentar "pensar" o evaluar cuál es realmente esa política, mientras que SARSA **tiene en cuenta** la política real del agente, es decir, lo que termina haciendo cuando se mueve al siguiente estado, en lugar de lo mejor posible que podría suponerse que va a hacer (evitar el precipicio, por ejemplo).

Cuándo elegir SARSA y cuándo elegir Q-Learning

Como se mencionó anteriormente, Q-Learning y SARSA son algoritmos muy **similares** y, de hecho, Q-Learning a veces se llama "SARSA-Max". Cuando la política del agente es simplemente "ávida" (es decir, que elige la acción de mayor valor del siguiente estado sin importar cuál sea éste), Q-Learning y SARSA producirán los mismos resultados. En la práctica, no se suele utilizar una estrategia únicamente ávida, sino que se opta por una de tipo ***ε-greedy***, donde algunas de las acciones se eligen al **azar**.

Por lo tanto, se puede pensar en SARSA como una versión más **general** de Q-Learning. Los algoritmos son muy similares y, en la práctica, modificar una implementación de Q-Learning para que se comporte como SARSA no implica más que cambiar el método de actualización de los valores Q. Sin embargo, la diferencia de rendimiento puede ser importante, como hemos visto en el *"Cliff Walking Example"*.

En resumen, SARSA funcionará **mejor** que Q-Learning cuando haya grandes posibilidades de que el agente elija una acción **subóptima** al azar en el siguiente paso. Si éste es el caso, SARSA convergerá más rápido y con menos errores. Si **no** lo fuera, Q-Learning será más **rápida**, por ser más inmediato y "ávido".

7.5. Trazas de Elegibilidad *(Eligibility Traces)*

Hasta ahora, para elegir su **próximo** estado nuestro agente se ha fijado en lo que percibe como la **mejor** recompensa de forma *greedy* o *ε-greedy*. Para los métodos que emplean el Aprendizaje por Diferencia Temporal, como Q-Learning y SARSA, se tiene en cuenta lo que se va a obtener tan sólo **un** paso **adelante**, o lo que se conoce como **TD(0)**. Ahora mismo, con los métodos que hemos visto, el agente sólo "recapacitará" sobre su **última** acción en el momento en el que **cambie** de estado. Es decir, que sólo al **finalizar** cada episodio se podrá hacer uso de los valores actualizados de la Tabla-Q de valores, para que en el siguiente episodio el agente puede tomar mejores decisiones. Queremos que nuestro agente sea lo más inteligente posible, que sea capaz de emplear de una forma mucho mejor la **experiencia** que va teniendo. Y esto es algo que quizá pudiera ser mejorado, a modo de **realimentación** continua.

Así que sería muy interesante que nuestro agente fuera capaz de tener en cuenta **no sólo** el próximo paso que diera (el estado al que cambie), sino **varios** pasos de antemano. Si es posible hacer esto, seguramente nuestro agente tome decisiones mucho mejores. ¿Y es posible? Sí, gracias a las **Trazas de Elegibilidad**.

Las Trazas de Elegibilidad **combinan** dos cosas: la **frecuencia** y la **antigüedad** de un **estado**. Ello implica que se deben **actualizar** las **estimaciones** de la función de estado/valor para cada estado en cada paso. Aparte de ello, los estados que no se visitan o que no se han visitado durante un tiempo tendrán un valor de elegibilidad igual o cercano a cero, lo que significa que la estimación para ellos apenas variará. Y ésto es exactamente lo que queremos.

$$e_t(s) = \begin{cases} \gamma\lambda e_{t-1}(s) & \text{si } s \neq s_t; \\ \gamma\lambda e_{t-1}(s) + 1 & \text{si } s = s_t, \end{cases}$$

Ecuación 4. Trazas de Elegibilidad (Sutton & Barto, 2015)

Las Trazas de Elegibilidad se **definen** en la *Ecuación 4*, siendo **γ** el **factor de descuento** y **λ** el parámetro conocido como **decaimiento** (o traza-decaimiento). Este tipo de elegibilidad es de **acumulación** (hay otros tipos), porque se acumula cada vez que se visita un estado y se va "**desvaneciendo**" gradualmente cuando no se visita el estado, como observa en la *Ilustración 54*:

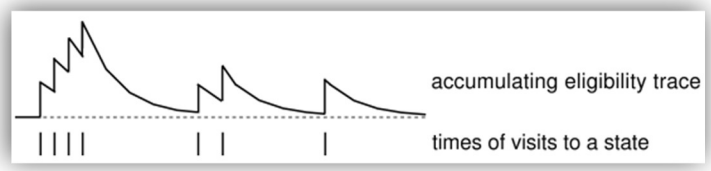


Ilustración 54. Trazas de Elegibilidad: Decaimiento (Sutton & Barto, 2015)

En un momento dado, las trazas **registran** qué estados han sido visitados **recientemente**, donde "recientemente" se define en términos de **γλ**. Así, las trazas indican el grado en que cada estado es "**elegible**" como **experiencia** de aprendizaje cuando se produzca un refuerzo, fijándonos sobre todo en los **errores** por diferencia temporal **δ** que se produzcan, definidos como se puede ver en la *Ecuación 5*:

$$\delta_t = r_{t+1} + \gamma V_t(s_{t+1}) - V_t(s_t)$$

Ecuación 5. Trazas de Elegibilidad: Errores por diferencia temporal (Sutton & Barto, 2015)

Que, en caso de generalizarse darían lugar a la ***Ecuación*** *6*:

$$\Delta V_t(s) = \alpha \delta_t E_t(s)$$

Ecuación 6. Trazas de Elegibilidad: Generalización de errores (Sutton & Barto, 2015)

Siendo α la **tasa de aprendizaje**, como en el caso general del Aprendizaje por Diferencia Temporal.

Así, el **algoritmo** que integra el concepto de "Trazas de Elegibilidad" a los métodos anteriores se conoce como **Algoritmo T(λ)**, que referencia el parámetro de **decaimiento**. Puede verse en la *Ilustración 55*:

Initialize $V(s)$ arbitrarily and $e(s) = 0$, for all $s \in \mathcal{S}$
Repeat (for each episode):
  Initialize s
  Repeat (for each step of episode):
    $a \leftarrow$ action given by π for s
    Take action a, observe reward, r, and next state, s'
    $\delta \leftarrow r + \gamma V(s') - V(s)$
    $e(s) \leftarrow e(s) + 1$
    For all s:
      $V(s) \leftarrow V(s) + \alpha\delta e(s)$
      $e(s) \leftarrow \gamma\lambda e(s)$
    $s \leftarrow s'$
  until s is terminal

Ilustración 55. Algoritmo T(λ) (Sutton & Barto, 2015)

Para comprender mejor esta **retropropagación**, pensemos en qué sucede para ciertos **valores** de λ:

- Si **λ = 0**: Estaremos en el caso **TD(0)**, el caso **general** del Aprendizaje por Diferencia Temporal, donde sólo el estado **anterior** cambia debido a los errores en las diferencias.
- Si **λ = 0** y **λ < 1**: Estaremos en el caso **TD(λ)**, ya cambiando valores de los estados **anteriores** pero, a medida que aumenta la distancia temporal con cada estado, éste se cambiará menos, porque su **traza de elegibilidad** será más pequeña, lo que es nuestra intención (ver *Ilustración 56*).

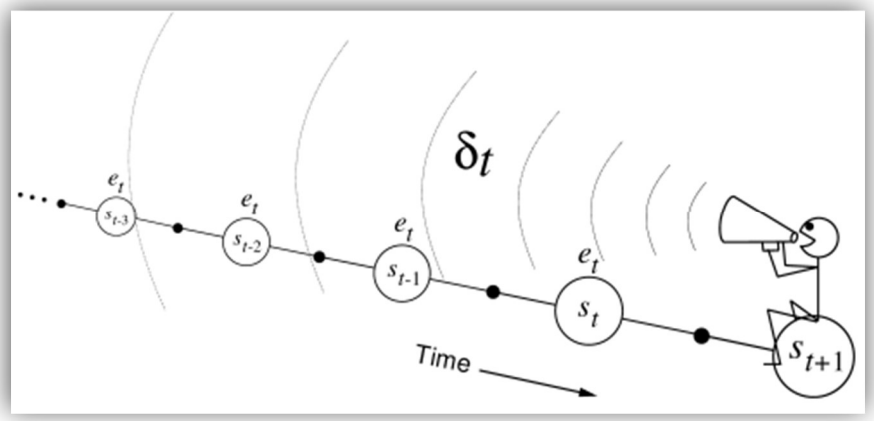


Ilustración 56. Algoritmo T(λ): Retropropagación (Sutton & Barto, 2015)

- Si **λ = 1**: Estaremos en el caso **TD(1)**, muy similar al conocido como "Método de Monte Carlo", en el que sólo se actualizarán los valores de los estados al **finalizar** cada episodio de aprendizaje.

Por ello, y volviendo a TD(λ), el valor de **elegibilidad** se utilizará a modo de **factor de escala** para los errores cometidos cuando se empleaba un Aprendizaje de tipo TD(0), como SARSA o Q-Learning. Pero ahora todos los estados se actualizarán **a la vez**, **propagando** la información del error actual a los estados ya visitados. Y éste hecho es clave para mejorar nuestros algoritmos de aprendizaje, pero ¿sería posible emplear el

concepto de Trazas de Elegibilidad **no sólo** para la **predicción**, como en TD(λ), sino para **controlar** las acciones del agente (su **política**, al fin y al cabo), como era posible con Q-Learning y SARSA?

7.5.1. SARSA(λ)

La idea de SARSA(λ) es aplicar el método de predicción de TD(λ) a pares **estado/acción** en lugar de sólo a los estados, por lo que se requerirá un **seguimiento** de los valores de dichos pares de estados/acciones. Así, si introducimos dichas ideas en la regla de actualización de SARSA, nos queda la *Ecuación 7*, donde los **errores** por diferencia temporal δ se definen como se ve en la *Ecuación 8*. En cuanto a las **Trazas de Elegibilidad**, se definen en la *Ecuación 9*, siendo γ el factor de descuento y λ el parámetro de traza-decaimiento:

$$Q_{t+1}(s,a) = Q_t(s,a) + \alpha\delta_t E_t(s,a) \text{ para todo } s,a$$

Ecuación 7. SARSA(λ): Regla de actualización (Sutton & Barto, 2015)

$$\delta_t = r_{t+1} + \gamma Q_t(s_{t+1}, a_{t+1}) - Q_t(s_t, a_t)$$

Ecuación 8. SARSA(λ): Errores por diferencia temporal (Sutton & Barto, 2015)

$$e_t(s,a) = \begin{cases} \gamma\lambda e_{t-1}(s,a) + 1 & \text{si } s = s_t \text{ y } a = a_t; \\ \gamma\lambda e_{t-1}(s,a) & \text{en otro caso.} \end{cases} \text{ para todo } s,a$$

Ecuación 9. SARSA(λ): Trazas de Elegibilidad (Sutton & Barto, 2015)

Por otra parte, el **algoritmo** de funcionamiento de SARSA(λ) es el de la *Ilustración 57*:

Initialize $Q(s,a)$ arbitrarily and $e(s,a) = 0$, for all s,a
Repeat (for each episode):
  Initialize s, a
  Repeat (for each step of episode):
    Take action a, observe r, s'
    Choose a' from s' using policy derived from Q (e.g., ε-greedy)
    $\delta \leftarrow r + \gamma Q(s',a') - Q(s,a)$
    $e(s,a) \leftarrow e(s,a) + 1$
    For all s,a:
      $Q(s,a) \leftarrow Q(s,a) + \alpha\delta e(s,a)$
      $e(s,a) \leftarrow \gamma\lambda e(s,a)$
    $s \leftarrow s'$; $a \leftarrow a'$
  until s is terminal

Ilustración 57. SARSA(λ): Algoritmo (Sutton & Barto, 2015)

El uso de Trazas de Elegibilidad puede aumentar sustancialmente la **eficiencia** de los algoritmos de control, hecho que se puede **observar** en la *Ilustración 58*:

1) La primera imagen muestra el camino recorrido por un agente en **un** solo episodio, que termina en una ubicación con una **recompensa**, marcada con *****. En este ejemplo, los valores son inicialmente 0, al igual que las recompensas, excepto una positiva en la ubicación *.

2) En las otras dos imágenes, las flechas muestran los valores/acción que se **refuerzan** al recorrer la ruta mediante los métodos SARSA y SARSA(λ). Así, el método habitual de SARSA sólo refuerza la **última** acción de la secuencia de acciones que condujo a la recompensa que se buscaba, mientras que el método SARSA(λ) refuerza muchas **otras** acciones de la secuencia. Aparte de ello, es interesante notar que el grado de fortalecimiento (indicado por el tamaño de las flechas) **disminuye** a medida que se alejan de la recompensa, debido a las **Trazas** de Elegibilidad.



Ilustración 58. SARSA vs SARSA(λ) (Sutton & Barto, 2015)

7.5.2. Q(λ)

Hay que recordar que Q-Learning es un método *off-policy*, lo que significa que la política aprendida no tiene por qué ser la misma que se empleó para seleccionar las acciones que sigue el agente. Además, normalmente Q-Learning emplea una política "ávida" o *greedy* al explorar, lo que puede provocar que seleccione acciones **subóptimas** en más ocasiones de las que sería deseable. Debido a esto, hay que tener especial **cuidado** al introducir el concepto de Trazas de Elegibilidad.

¿Y por qué se debe tener especial cuidado al hacerlo? Supongamos que estamos almacenando duplas estado/acción de la forma habitual, es decir, **momento a momento**. Ahora, imaginemos la siguiente secuencia: en dos pasos consecutivos cualesquiera, el agente selecciona una acción "ávida", pero al llegar a un tercero, el agente **cambia** y opta por una acción exploratoria **no** "ávida", algo perfectamente posible al emplear Q-Learning. El **problema** es que, si el agente aprende el valor de emplear una política *greedy*, **sólo** se puede utilizar dicha experiencia mientras se siga esa **misma** política "ávida", es decir, sólo para los dos primeros pasos, y no para el tercero. Por lo tanto, habrá que tener este tipo de situaciones muy en cuenta cuando se emplee la idea de Trazas de Elegibilidad.

Así, en Q(λ) las trazas se utilizan como en SARSA(λ), **excepto** que se **fijan a cero** cuando se realiza una acción exploratoria que **no** sea "ávida". Sería similar a que la actualización de trazas se realizase en 2 **pasos**:

1) Las **trazas** de todos los pares estado/acción **decaerán** mediante el uso de los parámetros **γλ**. Pero, si se optó por **explorar**, se fijarán a **0**.

2) La **traza** del estado/acción se **incrementa** en 1.

Siendo el **resultado** el de la *Ecuación 10*:

$$e_t(s,a) = \mathcal{I}_{ss_t} \cdot \mathcal{I}_{aa_t} + \begin{cases} \gamma\lambda e_{t-1}(s,a) & \text{si } Q_{t-1}(s_t,a_t) = \max_a Q_{t-1}(s_t,a) \\ 0 & \text{en otro caso} \end{cases}$$

Ecuación 10. Q(λ): Trazas de Elegibilidad (Sutton & Barto, 2015)

El resto del algoritmo está definido por la **regla** de actualización de la *Ecuación 11*, en la que los **errores** por diferencia temporal **δ** se definen como se ve en la *Ecuación 12*.

$$Q_{t+1}(s,a) = Q_t(s,a) + \alpha\delta_t e_t(s,a)$$

Ecuación 11. Q(λ): Regla de actualización (Sutton & Barto, 2015)

Por otra parte, el **algoritmo** de funcionamiento de Q(λ) es el de la *Ilustración 59*:

$$\delta_t = r_{t+1} + \gamma \max_{a'} Q_t(s_{t+1}, a') - Q_t(s_t, a_t)$$

Ecuación 12. Q(λ): Errores por diferencia temporal (Sutton & Barto, 2015)

Initialize $Q(s,a)$ arbitrarily and $e(s,a) = 0$, for all s, a
Repeat (for each episode):
  Initialize s, a
  Repeat (for each step of episode):
    Take action a, observe r, s'
    Choose a' from s' using policy derived from Q (e.g., ε-greedy)
    $a^* \leftarrow \arg\max_b Q(s',b)$ (if a' ties for the max, then $a^* \leftarrow a'$)
    $\delta \leftarrow r + \gamma Q(s',a^*) - Q(s,a)$
    $e(s,a) \leftarrow e(s,a) + 1$
    For all s, a:
      $Q(s,a) \leftarrow Q(s,a) + \alpha\delta e(s,a)$
      If $a' = a^*$, then $e(s,a) \leftarrow \gamma\lambda e(s,a)$
                    else $e(s,a) \leftarrow 0$
    $s \leftarrow s'$; $a \leftarrow a'$
  until s is terminal

Ilustración 59. Q(λ): Algoritmo (Sutton & Barto, 2015)

Desafortunadamente, **no** hacer uso de las trazas cada vez que se realiza una acción **exploratoria** hace que se **desperdicie** gran parte de la ventaja de emplearlas. Así que, si las acciones exploratorias son frecuentes, como en las **primeras** etapas del aprendizaje, **rara vez** se aprovecharán más allá de uno o dos, y el aprendizaje será sólo **un poco** más rápido que en el caso del Q-Learning habitual.

7.6. *Deep Q-Learning*

Como hemos visto con anterioridad, Q-Learning es un algoritmo sencillo, pero con la suficiente capacidad como para crear una "hoja de ruta" (la **Tabla-Q**) para nuestro agente, que determina exactamente qué **acción** realizar y según en el **estado** donde se encuentre.

Pero imaginemos un entorno con 10.000 estados y 1.000 acciones por estado, resultaría una Tabla-Q de 10 millones de celdas. Algo que, evidentemente, requeriría de una **cantidad** de **memoria** bastante elevada y de un **acceso** más rápido a ella, debido a su tamaño. Y no es precisamente el caso de nuestro laberinto, porque tiene unas dimensiones pensadas para poder almacenarlo en una Arduino, pero ¿qué sucedería en caso de querer ir **más allá** y probar otros laberintos?

Si bien los algoritmos laberínticos que hemos visto en capítulos anteriores tendrían que ser modificados para adaptarse a la nueva situación (un laberinto mayor), y también tendrían un límite en cuanto a la memoria de la Arduino, en el caso de Q-Learning también se **sumaría** el hecho de que se necesita una **Tabla-Q** con dos entradas por cada estado/celda (el propio estado/celda y su valor asociado) y las actualizaciones asociadas. Y, evidentemente, **agotaría** la memoria de la Arduino rápidamente.

¿Y qué solución se podría buscar? Está claro que **no** podemos **inferir** el valor Q de nuevos estados a partir de estados ya explorados, por los motivos que acabamos de ver. Pero exactamente para ello nos valdremos del enfoque que aporta el algoritmo que desarrolló la división de inteligencia artificial de **Google**, **DeepMind** (ver *Ilustración 60*), y al que llamó **DQN**[33] (Mnih, Silver, Veness, Fidjeland, & Petersen, 2015).



Ilustración 60. Google DeepMind

DQN utiliza lo que se conoce como "**repetición de experiencias**", un mecanismo inspirado en la propia biología de los seres vivos, que utiliza una muestra **aleatoria** de acciones anteriores **en lugar** de la acción más **reciente** para seguir avanzando. Ésto **elimina** las **correlaciones** en la secuencia de pasos a dar, y **suaviza** los cambios en la distribución de los datos que se almacenan. Así, actualizaciones iterativas en los valores irán **ajustando** Q para irse asemejando cada vez más a los valores **objetivo**. Valores que ahora se actualizarán sólo **periódicamente**, reduciendo **aún más** las correlaciones necesarias con el valor objetivo. Así, DQN **no** emplea una gran tabla de valores Q, sino que utiliza una **red neuronal** para **aproximar** la función del valor Q a partir de cada acción/estado dados. Generalmente, el **estado** se da como **entrada** y el valor **Q** de todas las acciones posibles se genera como **salida**.

Así, DQN se **caracteriza** por:

1) Cada **experiencia** es almacenada en **memoria**.
2) Cada **acción** está determinada por la **salida máxima** de la **red neuronal**.
3) El **error** que se produce al **estimar** cada valor es un problema de **regresión**.

[33] DQN: *Deep Q-Networks*

Además, la **regla de actualización** del valor Q derivada de la ecuación de Bellman habrá **cambiado**, como puede observarse en la *Ecuación 13*, y ahora para **DQN** tenemos que:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right]$$

Ecuación 13. DQN: Regla de actualización (Sutton & Barto, 2015)

Si nos fijamos en la ecuación anterior, la sección en verde representa el valor **objetivo**. Y podríamos pensar que está prediciendo su propio valor, pero dado que R es la verdadera recompensa, gracias a este método la red **actualizará** sus valores mediante **retropropagación** (básicamente de forma similar a la recursividad en algunos algoritmos de programación) para finalmente **converger** en los valores buscados.

Ahora bien, seguimos dependiendo de las **políticas** o valores del Aprendizaje por Refuerzo **tradicional** para **evaluar** el resultado de las acciones. Pero estos resultados van **cambiando** a medida que se aprende qué explorar, cuáles son los mejores caminos, por lo que la salida también estará cambiando. Entonces, ¿cómo es posible mapear una entrada/salida que cambia **constantemente**?

DQN lo **resuelve** empleando no una, sino **dos redes** neuronales para **estabilizar** el proceso de aprendizaje:

1) **Red Neuronal Principal**: Representada por los parámetros θ, se utiliza para **estimar** los valores-Q del estado S y la acción A actuales, mediante: $Q(S, A;\ \theta)$
2) **Red Neuronal Objetivo**: Parametrizada por θ', tendrá la misma arquitectura que la red principal, pero se usará para **aproximar** los valores-Q del siguiente estado S' y la siguiente acción A'.

El **aprendizaje** ocurre en la red **principal** y no en la objetivo. La red objetivo se "**congela**" (sus parámetros no se cambian) durante varias **iteraciones**, y después los parámetros de la red principal se **copian** a la red objetivo, **transmitiendo** así el **aprendizaje** de una a otra, haciendo que las estimaciones calculadas por la red objetivo sean más **precisas** y **sin** tantas **fluctuaciones** como serían esperables.

El **aprendizaje** ocurre en la red **principal** y no en la objetivo. La red objetivo se "**congela**" (sus parámetros no se cambian) durante varias **iteraciones**, y después los parámetros de la red principal se **copian** a la red objetivo, **transmitiendo** así el **aprendizaje** de una a otra, haciendo que las estimaciones calculadas por la red objetivo sean más **precisas** y **sin** tantas **fluctuaciones** como serían esperables.

En resumen, el **proceso** completo podría verse de la siguiente forma:

1) Se **pre-procesa** y "alimenta" de **estados** nuestro DQN: De este modo, la red devolverá los valores Q de todas las acciones posibles en cada estado.

2) Se selecciona una **acción** utilizando la política ***ε-greedy***: Con probabilidad ε se selecciona una acción aleatoria A, y con probabilidad 1- ε, se selecciona una acción que tiene un valor Q máximo.
3) Tras ello, hay que **moverse** a un **nuevo** estado S' (para recibir una recompensa): Se almacena esta transición en memoria como $< S, A, R, S' >$

   Se **muestrean** algunos valores **aleatorios** de transiciones existentes en memoria y se **calcula** la diferencia entre el valor estimado y el objetivo, o "**Función de Pérdida**" de la *Ecuación 14* (notar que es el cuadrado de la diferencia entre ambos lados de la ecuación de Bellman para DQN):

$$Pérdida = (r + \gamma max_{a'} Q(s', a'; \theta') - Q(s, a; \theta))^2$$

Ecuación 14. DQN: Función de pérdida (Sutton & Barto, 2015)

4) Se emplea el método del **Gradiente Descendente** para ML (Martínez Heras, 2020) respecto de los parámetros de la red para realizar la **regresión** de valores y **minimizar** la pérdida.
5) Después de cierto número de iteraciones, se **copian** los valores de la red principal a la objetivo.
6) Los pasos anteriores deben **repetirse** el número de episodios suficientes para alcanzar la **convergencia** de valores que se desea.

Gracias a este método, realmente **complejo** de implementar, pero factible, trataremos de ir un paso más allá y cumplir con el objetivo último de este trabajo: Dotar a nuestro agente/robot de una **inteligencia** capaz de **adaptarse** a cualquier laberinto, de cualquier forma y tamaño.

7.7. En los límites de Arduino

Actualmente existe mucha gente interesada en la posibilidad de ejecutar algoritmos de Aprendizaje Automático en una placa **Arduino**, porque es asequible, fácil de encontrar, muy versátil y está presente en todo tipo de proyectos. Y, tras investigar lo suficiente, he podido llegar a las siguientes **conclusiones**:

1) Los **recursos** de una Arduino son demasiado **limitados** para el Aprendizaje Automático.
2) En cuanto a **redes neuronales**, es posible una implementación **sencilla** de un perceptrón multicapa.
3) Y, recientemente, es posible usar el *software* TensorFlow **Lite** para tareas, digamos, "**sencillas**".

Y todas las conclusiones son válidas y no excluyentes. Es decir, una Arduino siempre estará **limitada** para el Aprendizaje Automático por su capacidad de **procesamiento** o por su **memoria**, principalmente. Por eso, todas las soluciones al respecto van a tratar de "sortear" este hecho como, por ejemplo, TensorFlow Lite (TensorFlow , 2020), una variante de TensorFlow especialmente adaptada para Arduino

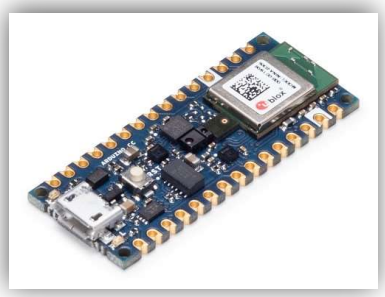


Ilustración 61. Arduino Nano 33 BLE Sense (Arduino.cc, 2020)

IDE. La idea detrás de TensorFlow Lite es utilizar la capacidad de **Arduino** para obtener datos de sus sensores, **transmitir** dicha información a través de la placa auxiliar Arduino Nano 33 BLE Sense (ver *Ilustración 61*) a un **PC** por Bluetooth para que la procese y, una vez procesada, **devolvérsela** a la Arduino para que siga operando con ella (Arduino Team, 2020).

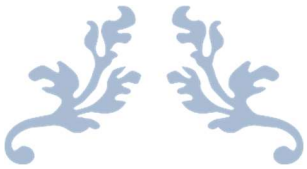
Aunque es una buena idea, todavía necesita bastante desarrollo y, sobre todo, está **limitada** por su propio concepto de funcionamiento, ya que:

- TensorFlow **Lite** es una versión **limitada** de TensorFlow, por lo que **no** van a estar disponibles todas sus **funcionalidades**, habrá limitaciones en cuanto a APIs externas y la **comunidad** de desarrolladores no está, ni por asomo, tan involucrada como en la versión principal.
- El uso de **Bluetooth** 4.0 es una buena idea, fácil de conectar, relativamente robusto y asequible. Pero cuando se trata de manejar grandes cantidades de datos, **no** es especialmente **rápido**.
- La placa auxiliar tiene un **coste** muy similar a la de la propia Arduino, además de un **consumo** energético considerable, lo que **reduciría** la **autonomía** de nuestro robot.

Así pues, ¿cuál es la **solución** que propongo? Tratar de **aprovechar** al máximo su idea, pero intentando **evitar** sus limitaciones en cuanto a APIs, velocidades, consumos, etc. Y, para ello, en esta parte del trabajo, voy a tratar de exprimir todos los **recursos** de los que disponemos, trabajando sobre las siguientes **ideas**:

1) El **laberinto** diseñado **no va a cambiar**, ni el simulado ni el físico, porque la idea es poder comparar los resultados con los algoritmos para laberintos con los que hemos trabajado. Es decir, el robot va a **conocer** el entorno **previamente**, antes de empezar a moverse.
2) Por ello, en un primer momento podemos "separarnos" de la placa Arduino y emplear toda la potencia, versatilidad y recursos de un **PC**, algo inalcanzable para ella. Así, **no** tendremos **limitaciones** en cuanto a *software*, APIs, kits de desarrollo, etc., para implementar nuestros algoritmos de RL y, sobre todo, de DRL, algo que demanda bastante capacidad de procesamiento.
3) Así, podremos emplear algunas de las **herramientas** y **lenguajes** más adaptados a este campo, como Python (Python.org, 2020), TensorFlow (Google, 2020), Keras (Keras Team, 2020), NumPy (NumPy, 2020) y MatPlotLib (MatPlotLib, 2020).
4) Una vez que el PC haya "aprendido" cuál es la mejor ruta gracias a los algoritmos que implementemos, se la trasladaremos a la **Arduino** en forma de **secuencia** a seguir mediante un cable USB, Bluetooth, Wi-Fi o la tecnología que resulte más eficiente y eficaz.
5) Por último, el robot, gracias a la placa Arduino que integra y a su especialización en cuanto a **sensores**, se ocupará de recoger todos los datos posibles de su entorno y **guiarle** desde la salida hasta la meta del laberinto, de forma rápida y segura, evitando obstáculos, haciendo los giros

oportunos, etc. Algo que recuerda, inevitablemente, a la forma en la que se trabajaba con el Algoritmo de Inundación, aunque en ese caso el pre-procesado del laberinto se hacía en la Arduino.



8. APRENDIZAJE POR REFUERZO: RESULTADOS



8. Aprendizaje por Refuerzo: Resultados

8.1. Laberintos

8.1.1. Virtual+Estándar

Inicialmente vamos a emplear el **mismo** laberinto que hemos utilizado hasta ahora para nuestras pruebas. El primer objetivo es saber si en él **es posible** aplicar los algoritmos de Aprendizaje por Refuerzo. Además, como ya se ha comentado, y a pesar de que no es muy grande, tiene un diseño que servirá perfectamente para obtener **conclusiones** lo suficientemente relevantes y conocer su comportamiento.

Si recordamos, el **diseño** del laberinto Virtual+Estándar es el de la *Ilustración 62*:

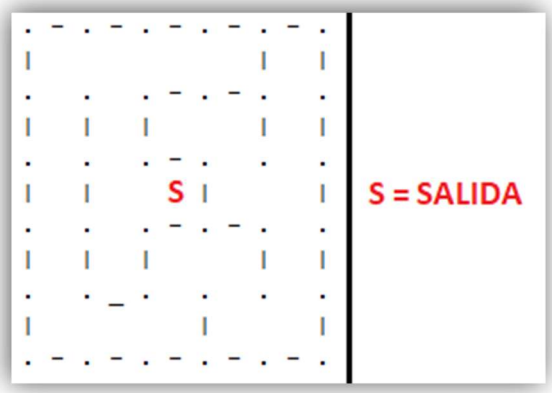


Ilustración 62. Laberinto Virtual+Estándar: Diseño

Tiene un tamaño de **11x11**, y el diseño y el trazado de este laberinto y el del Real+Estándar siguen siendo **idénticos** conceptualmente. Pero, además, vamos a aprovechar el hecho de que ahora empleamos **Aprendizaje por Refuerzo**, y **no** va a haber una entrada al laberinto **fija**, es decir, que se va a probar a salir desde **todas** las celdas posibles. Por lo tanto, al finalizar el **entrenamiento** podríamos obtener una ruta desde **cualquier** celda del laberinto, una **mejora** deseable y una gran **ventaja** sobre los algoritmos previos.

Pero, si bien es cierto que el laberinto es el mismo, la situación y la forma de resolver el laberinto ha **cambiado**. Ahora, un **agente** decidirá qué **acción** tomar (N, S, E, O) en cada paso. Para ello, se "**entrenará**", y aprenderá una **política** Q que le indicará cuál es el mejor paso a seguir. Así, en cada uno de ellos el agente puede recibir una **penalización** o una **recompensa** (si logra llegar a la salida). Y, tal y como se ha comentado, el agente se entrenará partiendo de todas las celdas, resolviendo el laberinto **completamente**.

Las políticas (o modelos) que utilizaremos van a emplear **Q-Learning**, **SARSA** y **DQN**. Durante el entrenamiento, los algoritmos de aprendizaje **actualizarán** los valores/acciones para cada **estado** que se visite, y se deducirá/predecirá la **acción** más preferible, según el algoritmo empleado. Notar que, al emplear métodos que hacen uso de un Aprendizaje por Diferencia Temporal, el agente aprenderá en **cada paso** que

dé, **no sólo** cuando alcance la salida. Y dicho aprendizaje se **acelerará** una vez que se alcance la salida por primera vez, especialmente cuando se obtenga el **suficiente** grado de **convergencia** en los resultados.

8.1.2. Real+Estándar

Al igual que en el caso de los algoritmos laberínticos originales, se empleará la estrategia de avanzar físicamente **de 2 en 2** celdas, para ahorrar **tiempos** de desplazamiento. Aparte de ello, he aprovechado el hecho de que ahora el robot conocerá el recorrido óptimo **previamente** para diseñar una versión "**reducida**" y poner a prueba sus capacidades de forma más sencilla y clara.

El **diseño** final del trazado Real+Estándar puede verse en la *Ilustración 63*:

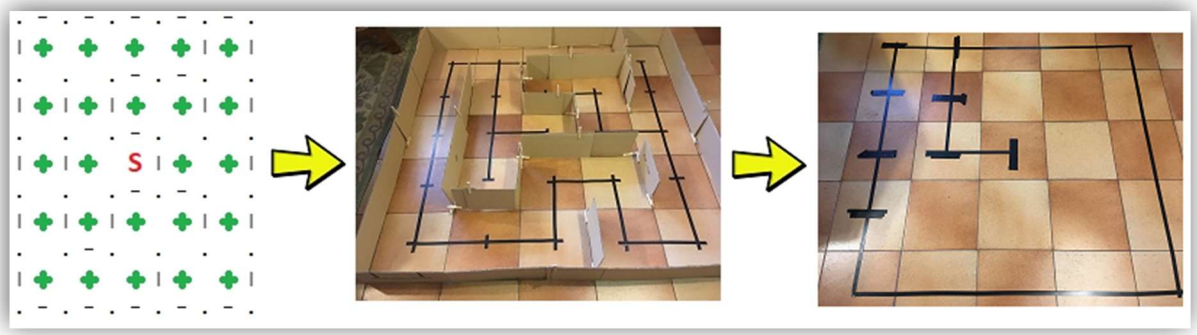


Ilustración 63. Laberinto Real+Estándar: Vista general del diseño de 2x2 celdas (izq: virtual, centro: completo, dch: reducido)

En cuanto al problema que existía con el **suelo**, debido a que su diferente **rozamiento** influye en cierta manera sobre los movimientos del robot, especialmente en los **giros**, espero que ahora sea posible **corregirlo** en parte gracias a los nuevos ***encoders*** instalados en los ejes de las ruedas.

8.1.3. Virtual+Desafiante

Una vez que tenemos claro cómo vamos a abordar la resolución del laberinto que venimos empleando hasta ahora, detengámonos un momento y **reflexionemos** sobre los siguientes **puntos**:

- Disponemos de un **PC**, así que la limitada **potencia** de cálculo de Arduino ahora no es un problema.
- Por el mismo motivo, tampoco existen limitaciones en cuanto a **memoria**, dentro de un orden.
- Los algoritmos que se han diseñado emplean RL, lo que implica que deberían ser capaces de **aprender** cuál es la mejor ruta, **independientemente** del laberinto.
- Los potenciales **problemas** que presentaban los algoritmos laberínticos, como bucles en los giros, mejores tiempos si se opta por girar hacia la derecha o la izquierda, discontinuidades en las paredes, etc., ahora deberían poder **solventarse** sin dificultad.
- Y si optamos por un diseño **virtual** únicamente podremos dotarle de las características que más convengan para llevar los algoritmos **al límite**, sin limitaciones de tamaño, materiales, etc.

Así que ¿por qué no diseñar un laberinto que ponga **a prueba** todas las capacidades de los algoritmos que estamos utilizando para trabajar el Aprendizaje por Refuerzo? Con esta idea presente, he diseñado un laberinto del **doble** de **tamaño** efectivo que el anterior y con bastante **dificultad** añadida, siendo el **laberinto** resultante el que puede verse en la *Ilustración 64* y al que he llamado "**Virtual+Desafiante**":

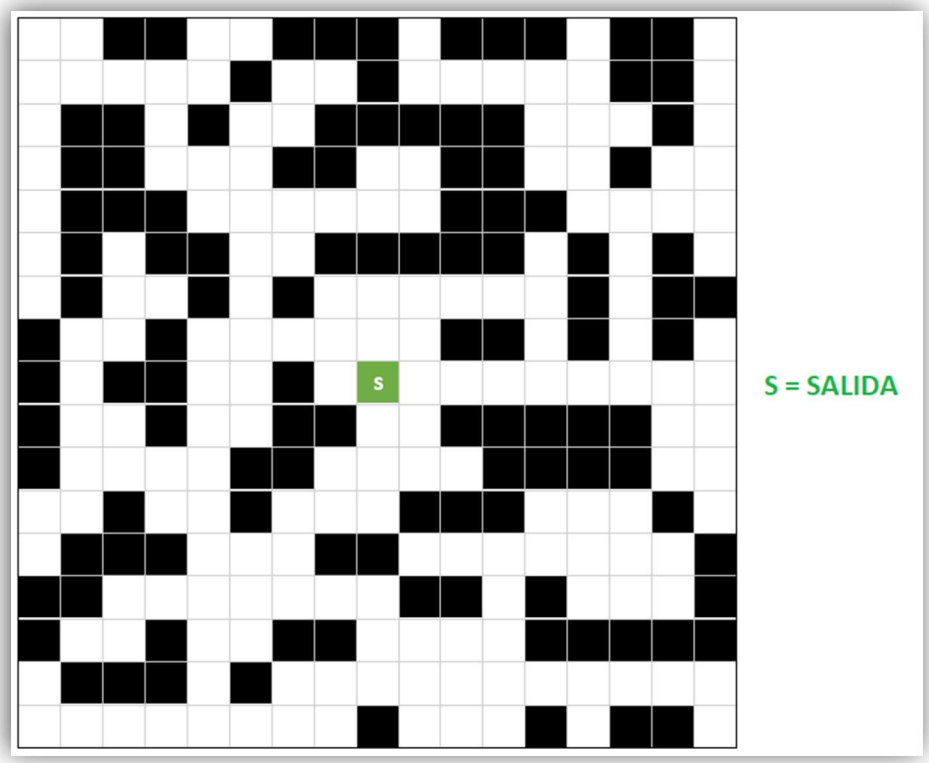


Ilustración 64. Laberinto Virtual+Desafiante: Diseño

Y si bien el anterior tenía un tamaño de 11x11, las celdas del perímetro del laberinto se empleaban para delimitar sus "paredes", por lo que el laberinto que se resolvía era de un tamaño de 9x9 efectivo. Así, finalmente tiene un tamaño de **17x17** celdas, no de 18x18, como cabría esperar. Es así porque prefería que hubiera un **centro** exacto por motivos más bien "estéticos", por lo que debía tener un tamaño **impar**.

Su **composición** ha sido determinada de forma **semi-aleatoria**, es decir:

1) He creado una rejilla **vacía** de tamaño **17x17**.
2) Sobre ella he **generado** un laberinto de forma totalmente **aleatoria**.
3) He creado **caminos** desde las casillas **aisladas**, si las hay, hasta la **salida**.
4) Por último, he **revisado** el trazado para alcanzar la mayor **dificultad** posible.

Y lo he diseñado así porque se deben evitar las casillas aisladas para alcanzar la **convergencia** de forma efectiva, además de para poder probar que es posible resolver **cualquier** laberinto, un reto imposible hasta ahora. Además, es un laberinto con muchas "**trampas**" (bucles, callejones sin salida, celdas aisladas, etc.),

perfectas para que cualquiera de los algoritmos con los que hemos trabajado con anterioridad **fallen**. Ahora ya no debe perseguirse tanto el resolver el laberinto de la forma más rápida, sino el poner **a prueba** la "**inteligencia**" del agente a la hora de resolverlos, el **objetivo** final del presente trabajo.

8.2. Implementación de los algoritmos

Para poder hacer **comparaciones** de forma adecuada, ciertos aspectos siempre serán **comunes** entre los laberintos Virtual+Estándar y Real+Estándar, como:

a) **Direcciones** (Norte – Sur – Este – Oeste).
b) **Trazado** y posiciones de los **obstáculos**.
c) **Programación** de los algoritmos y modelos de aprendizaje en **Python**.
d) Arduino IDE y lenguaje **C** empleados para la navegación física.
e) Muestra de datos por **pantalla**, ahora ya empleando la librería gráfica PyPlot.

Hay que recordar que el laberinto **Virtual+Desafiante** sólo será implementado y puesto a prueba de forma **simulada**. Y es así porque de la posible mejora en la navegación **física** sacaremos **conclusiones** más acertadas y relevantes si comparamos únicamente el **Virtual+Estándar** con su homólogo **Real+Estándar**.

8.2.1. Entorno simulado

Toda la **teoría** vista sobre el RL vista hasta el momento será **necesaria**. Y en ello **Python** nos resultará de gran ayuda, porque nos permitirá aislarnos un poco más del código para centrarnos mejor en la propia implementación de cada caso.

Las **características** de los **modelos de aprendizaje** implementados pueden resumirse en la *Tabla 4*:

	Modelos					
Características	**Q-Aleat**	**SARSA**	**SARSA(λ)**	**Q-Learn**	**Q(λ)**	**DQN**
Entrada desde cualquier celda	X	X	X	X	X	X
Tabla-Q con estados/acciones	X	X	X	X	X	
Red neuronal con estados/acciones						X
Valores iniciales = 0	X	X	X	X	X	X
Valores actualizados con recompensa/penalización		X	X	X	X	X
Política *greedy*		X	X	X	X	X
El entrenamiento finaliza al llegar a n episodios	X	X	X	X	X	X
El entrenamiento finaliza con 100% de victorias	X	X	X	X	X	X
Trazas de Elegibilidad			X		X	
Tasa de Decaimiento en la Elegibilidad			X		X	X
Posibilidad de guardar/cargar el modelo						X

Tabla 4. Características de los Modelos de Aprendizaje

En cuanto a qué **parámetros** son configurables en el código asociado, son los de la *Tabla 5*:

Valores configurables	Modelos					
	Q-Aleat	SARSA	SARSA(λ)	Q-Learn	Q(λ)	DQN
Número de transiciones/experiencias a almacenar						X
Número de acciones para la capa de salida						X
Valores objetivo						X
Tasa de descuento		X	X	X	X	X
Tasa de exploración		X	X	X	X	X
Tasa de decaimiento en la exploración		X	X	X	X	
Tasa de aprendizaje		X	X	X	X	
Tasa de decaimiento en la elegibilidad			X		X	
Número de episodios máximo	X	X	X	X	X	X
Tamaño de la muestra a recuperar						X
Episodios tras los que comprobar la convergencia	X	X	X	X	X	X
Datos cargados de anteriores experiencias						X

Tabla 5. Parámetros configurables de los Modelos de Aprendizaje

Q-Aleatorio

He implementado un algoritmo que va "a ciegas" para comprobar sus resultados, como en las anteriores pruebas. Y debería ser el **menos óptimo** de todos los algoritmos, ya que el hecho de que sea **aleatorio** significa que puede variar entre obtener una ruta óptima o la peor posible. Por otra parte, su aleatoriedad **simplifica** bastante la **programación** de su modelo, porque reduce al mínimo las decisiones.

Por último, debido a que ahora se emplea un lenguaje mucho más moderno y amigable, como es Python, existen multitud de librerías casi para todo, por lo que se ha empleado la librería **`random`** para obtener un número/dirección **aleatoria** sin mayor problema.

SARSA, SARSA(λ), Q-Learning y Q(λ)

Se han implementado siguiendo sus **conceptos** al máximo, tratando de **adaptarlos** a la situación concreta.

DQN

Es el modelo más **complejo** de implementar, lo que era esperable. Aparte de todo lo ya visto, hace uso de **TensorFlow** en combinación con **Keras** para crear las dos redes neuronales que necesitamos, la **principal** y la **objetivo**, perfectas para nuestros propósitos, aunque seguramente **exigentes** computacionalmente.

La red **objetivo** se emplea, como hemos visto en la teoría, a modo de respaldo o *backup*, reflejándose este hecho en el **código** con la estructura necesaria. Aparte de ello, de esta red se extraen **muestras** para **mejorar** el aprendizaje del agente, de forma aleatoria o dirigida, según nuestras necesidades.

Por último, he implementado la posibilidad de **cargar/guardar** el **modelo** creado para poder reutilizarlo. Esto resulta muy interesante si el modelo es muy **pesado** o si no se dispone de **tiempo** suficiente para poder entrenarlo y se prefiere **proseguir** con su entrenamiento **más tarde**, si se desea.

8.2.2. Entorno real

Tras llevar a cabo la implementación del robot en un entorno simulado, el paso al robot **real** conlleva la **gestión**, común a todos los algoritmos, de los siguientes **apartados**:

a) **Algoritmos**: Los movimientos físicos serán de **2 en 2** celdas, para mejorar los tiempos.
b) **Arduino**: Gestión de tiempos, optimización de cálculos, aprovechamiento de la memoria, etc.
c) **Sensores de ultrasonidos**: Medición de distancias, posicionamiento, etc.
d) **Sensores de líneas**: Calibración de sensibilidad, distancia al suelo, materiales a emplear, etc.
e) ***Encoders***: Calibración, conteo de *clicks*, ajuste al giro de las ruedas, etc.
f) **Pantalla OLED**: Inicialización, cálculo de dimensiones, formateado, etc.
g) **Motores**: Velocidad, potencia suministrada, sentido del giro, control de ejes, etc.
h) **Baterías**: Alimentación y gestión de la energía suministrada.

Debido a ello, he tenido que efectuar **pruebas** y **ajustes** de todos y cada uno de los puntos comentados:

- Los sensores de **ultrasonidos** siguen sir ser nada aptos para operar con ciertos **materiales** blandos (tipo tela o paños), por lo que seguiremos empleando materiales más rígidos y fáciles de medir.
- Las **ruedas** siguen teniendo algo más **potencia** en el eje derecho, pero su avance, los giros, y todos los movimientos del robot en general, ahora están un poco más bajo control, gracias a los ***encoders***, que han resultado ser una aportación relevante. Eso sí, si el robot avanza demasiado rápido, **no** habrá **tiempo** de detenerse en un punto exacto del *encoder*.
- Los sensores de **líneas** no necesitan de calibración adicional, ya que ahora se emplea un modelo de 5 sensores integrados que viene **precalibrado** de fábrica, y resulta mucho más **preciso**. Aparte de ello, se **integra** con el controlador **PID** a la perfección.
- La **potencia** de los **motores** debe manejarse de forma **analógica** mediante pulsos PWM, al igual que sucedía en las anteriores pruebas.

En cuanto al **diseño** del **laberinto**, la **superficie** elegida y el **material** empleado, éstos serán los mismos que los empleados con anterioridad, para poder maximizar las posibles comparaciones.

Por otro lado, ahora disponemos de **tres** sensores de **ultrasonidos**, midiendo distancias continuamente, lo que debería ayudar a que el robot se mueva más rápido y a que se comentan menos errores de

trayectoria, aunque seguramente siga sin ser 100% **perfecta**. Por ello, seguiremos **combinando** los sensores de ultrasonidos con los sensores de **líneas**, ahora cinco (dos más que antes), que serán los encargados de aportar la **posición**, y así el posible desvío del robot de su rumbo ideal. Aparte de ello, he **incorporado** al control de trayectorias los valores que captan los ***encoders***, para hacerlas más **precisas** y lo más suavizadas posibles.

Por último, también se ha implementado un **menú** para **elegir** el modo de funcionamiento del robot desde el **mando IR**, que ahora funciona mucho **mejor** gracias a la Arduino Mega y sus múltiples ***timers***.

Q-Aleatorio

Su **dificultad** radica en saber si el robot puede darse **media vuelta** desde cualquier posición, aunque ahora disponemos de tres sensores, lo que **agilizará** el proceso de comprobación.

SARSA, SARSA(λ), Q-Learning, Q(λ) y DQN

Debido a que se han implementado estos cuatro algoritmos de RL en un **PC**, por los motivos comentados en apartados anteriores, para su traslación al laberinto real se seguirá una política **similar** al caso del algoritmo de **Inundación**. Es decir, se seguirá el siguiente **procedimiento**:

1) Ejecución del algoritmo de RL deseado en el **PC**.
2) Extracción/volcado de la **ruta óptima** entre la entrada y la salida establecidas.
3) Carga de dicha ruta en la placa **Arduino**.
4) Recorrido **físico** del robot por el laberinto, empleando la ruta óptima.

Su funcionamiento es muy **similar** al visto en el apartado simulado, salvo por la integración del PID, las mediciones de los sensores, el avance por las líneas, etc.

8.3. Pruebas iniciales

8.3.1. Entorno simulado

Se ha tenido que:

- Adaptar el funcionamiento de los **algoritmos** a los nuevos requisitos planteados.
- Estructurar el **código** en función de las características de Python, Keras, etc.
- **Optimizar** el funcionamiento de los algoritmos, realmente complejos.
- Dotar al nuevo código de una **documentación** lo más comprensible posible.

8.3.2. Entorno real

Al tratar de trasladar los algoritmos del mundo virtual al **real**, han surgido ciertos **problemas** iniciales de diferentes tipos, la mayoría esperables. No tanto en cuanto a los algoritmos diseñados, sino debido a los **componentes** del robot y las **peculiaridades** de los mismos. Son de **destacar**:

- La gestión de la **memoria**, aunque en este caso, ya no es tan relevante. El disponer de GBytes de RAM en un **PC** actual posibilita manejar **tamaños** de laberinto impensables para una Arduino.
- En cuanto a los *encoders*, he tenido que optar por utilizar sólo **dos** de los cuatro inicialmente previstos, debido a que surgieron problemas de **interferencias** en las **interrupciones** entre ellos. Pero dos **bien situados**, a cada lado y en ambos ejes, deberían funcionar perfectamente y tomar las **referencias** de avance y retroceso de las ruedas sin problema.
- Muchos tornillos, tuercas y espaciadores **nuevos** han sido necesarios. He tenido que **modificar** el **chasis**, debido al aumento de sensores, al cambio de la placa Arduino y del cableado, y resultan fundamentales para ajustar todo correctamente. La precisión a la hora de tomar **medidas** para poder situar bien los **anclajes** de los componentes es clave (ver *Ilustración 65*).

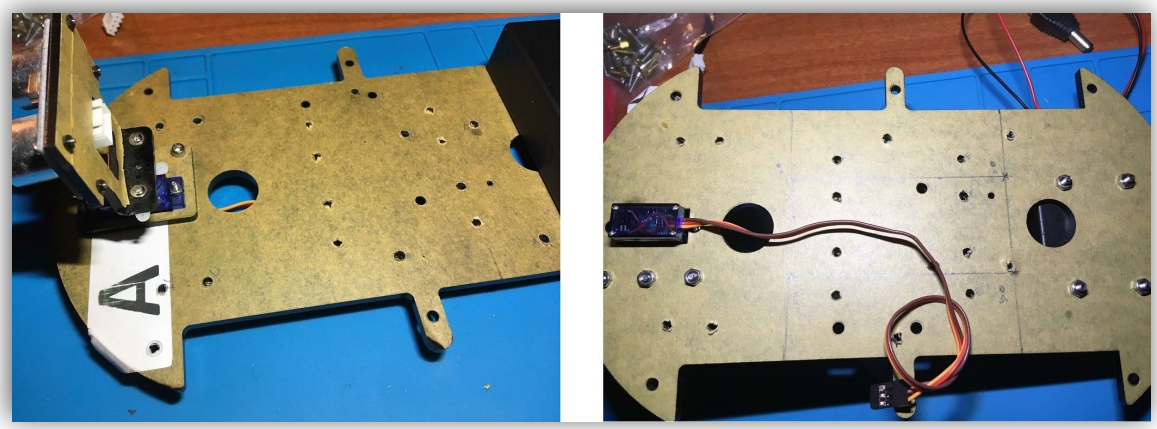


Ilustración 65. Mejoras: Nuevos anclajes en el chasis

- Además, un **exceso** de apriete en la placa Arduino **impedía** poder **programarla** bien. Una tarde entera buscando fallos de programación inexistentes y mucha paciencia fueron la clave para poder resolver algo tan imprevisto como trivial.
- La alimentación ha tenido que **puentearse** gracias a una ***breadboard*** externa, porque las conexiones de la Arduino **no** han sido suficientes para alimentar todos los sensores. Ahora bien, la Mega tiene unos **reguladores** de voltaje **muy buenos**, y ha podido suministrar la energía necesaria sin problemas ni fluctuaciones en la misma.
- La pantalla **OLED** se ha comportado de forma fantástica. Es pequeña, pero posee buena **definición** y tiene un consumo **ínfimo**.
- El haber integrado tantos sensores, obliga a hacer pruebas **individuales** del cableado y a revisar todos ellos uno por uno casi **cada vez** que se pone a funcionar el robot.

- El **peso** de los componentes extra ha obligado a trabajar sobre la entrega de **potencia** de los motores, que ahora es bastante **diferente** a los resultados obtenidos con anterioridad. Además, los **ejes** de las ruedas también han sido revisados para comprobar que se ajustan como deben.
- El **receptor IR** del mando a distancia se ha comportado bastante bien, porque la placa Arduino Mega dispone de **más** de un **`timer`**, por lo que **no** interfiere con ningún otro. Aparte de ello, ahora ya no empleamos el **servo**, por lo que se evitarán aún más interferencias en este sentido.

Pero, aparte de todo ello, las nuevas modificaciones hechas en el kit del **robot** son bastante robustas, por lo que todos los componentes montados (ver *Ilustración 66*) deberían funcionar a pleno rendimiento.

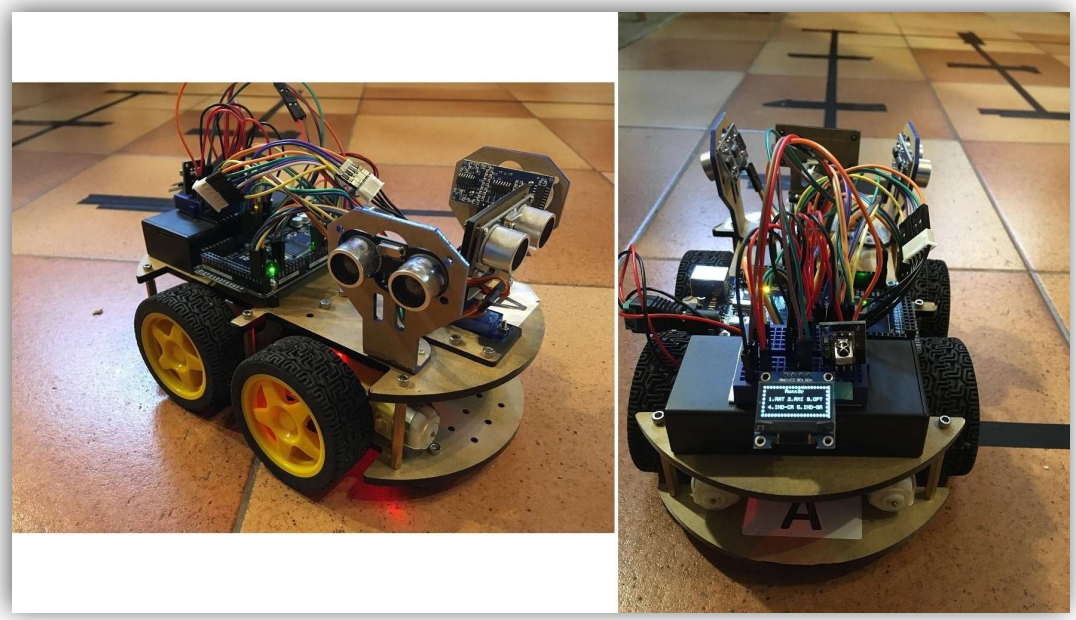


Ilustración 66. Runn3r: Aspecto final, preparado para recorrer el laberinto

8.4. Resultados

Todo lo visto debe ser probado y analizado. Para ello, se observará el **funcionamiento** de los algoritmos y el **tiempo** empleado en resolver el laberinto. Y deberá tenerse presente que los tiempos obtenidos son los que el agente emplea en entrenarse **por completo** y alcanzar la salida desde **todas** las posibles celdas de **entrada**. Por ello, los tiempos totales serán **superiores** al caso de los algoritmos laberínticos, que se centraban en una única entrada, y **no** deberían ser comparados sin situarles en su **contexto** particular.

Aparte de ello, recordar una vez más que los laberintos Virtual+Estándar y Real+Estándar tienen un trazado **idéntico** entre sí y respecto de los empleados con los algoritmos laberínticos. Pero, por otra parte, el caso del laberinto Virtual+Desafiante será **exclusivo** de un entorno **simulado**, ya que lo que se busca es saber **si** es **posible** resolverlo, en cuantos **episodios** y en cuánto **tiempo**, más que compararlo con el resto de casos.

8.4.1. Laberinto Virtual+Estándar

Q-Aleatorio

En principio, para este caso se debería ejecutar el algoritmo **varias** veces para extraer conclusiones, porque su naturaleza **aleatoria** hace que los tiempos empleados **no** sean los mismos en ninguna ocasión.

Pero esta vez no es, en absoluto, necesario: este algoritmo es de todo **menos fiable**, no encontrando casi **ninguna** vez la salida. Es así porque el RL se basa en, precisamente, la **experiencia** del agente al ponerse a prueba. Y en este caso **no** puede hacerlo porque en este caso no hay política alguna a seguir, no hay recompensas que acumular, no hay rumbo mejor que otro. De no encontrar la salida en sus primeros pasos, todo su aprendizaje será severamente **penalizado** y se dará por **finalizado**.

Por ello, no mostraré estadísticas de resultado alguno, porque tratar de aplicar el RL a un agente que no tiene ningún tipo de "inteligencia" es prácticamente **imposible** e incluso **impracticable**, y sus **pésimos** resultados (ha encontrado la salida sólo una vez en más de 50 intentos…) lo demuestran.

SARSA

Su funcionamiento va a ser interesante de conocer, y de comparar con Q-Learning cuando veamos sus resultados. En principio, debería ser más **lento** que Q-Learning, porque emplea una política más "**conservadora**", pero **podría** suceder que precisamente por emplear una política "ávida", Q-Learning acabe siendo más lento que SARSA, si recibe demasiadas penalizaciones por movimientos erróneos.

Así pues, veamos sus **resultados** en la *Ilustración 67*:

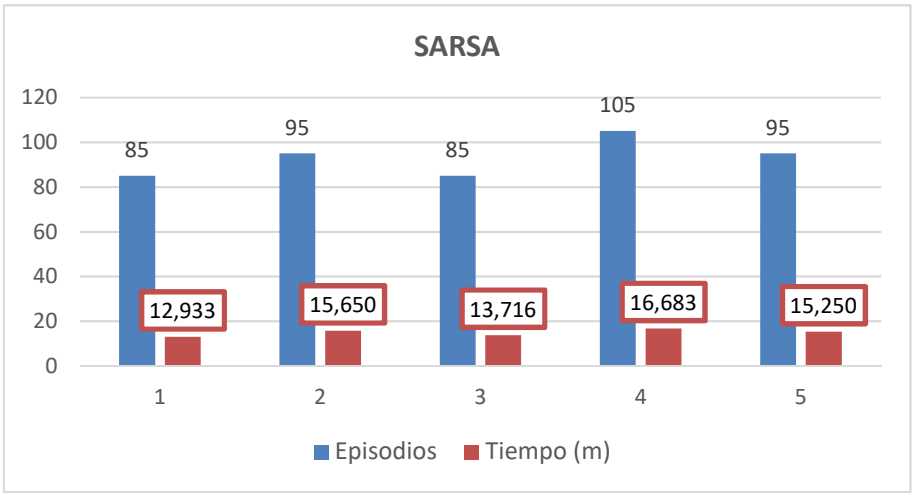


Ilustración 67. Resultados: Laberinto Virtual+Estándar - SARSA

Como puede observarse, los resultados son bastante **estables**, promediando cerca de los 93-94 episodios y los 14-casi-15 minutos. A priori, parecen bastante **buenos**, pero esperemos a ver lo que pasa con Q-Learning para poder sacar conclusiones más tangibles.

SARSA(λ)

En este caso se hace uso de las Trazas de Elegibilidad, lo que debería llevar a mejores tiempos de ejecución. Pero veamos sus **resultados** en la *Ilustración 68*:

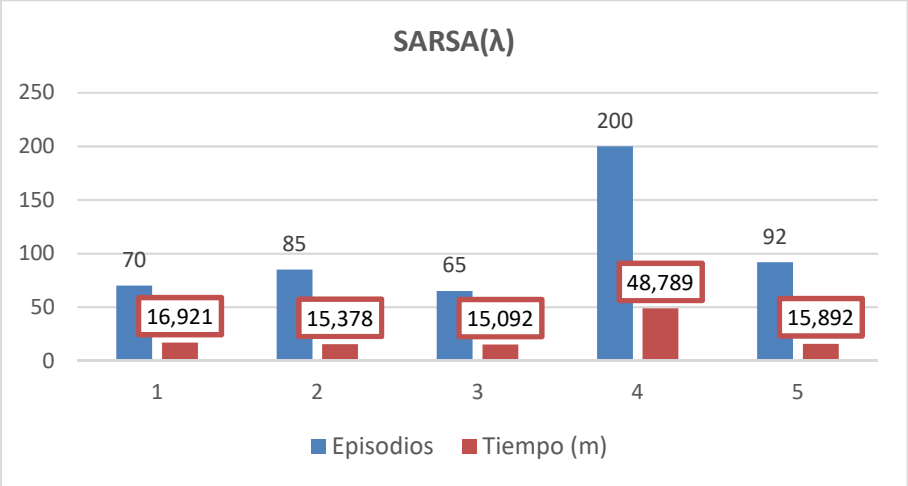


Ilustración 68. Resultados: Laberinto Virtual+Estándar - SARSA(λ)

Sus **resultados** son realmente **sorprendentes** por varios motivos:

- SARSA(λ) converge en un **menor** número de episodios que SARSA.
- A pesar de ello, los tiempos que emplean son **similares** a la versión "estándar".
- Y, sobre todo, hay un caso que ha sido detenido antes de converger, porque el **límite** máximo de episodios que he impuesto es de 200. Por ello, **no** ha llegado a alcanzar una salida.

He tratado de **investigar** y recopilar información sobre SARSA(λ) para averiguar a qué pueden. Y, sobre todo, destacaría el **estudio** de (Grzes & Kudenko, 2008), en el que se analiza la robustez de los métodos que emplean Trazas de Elegibilidad. En el caso de SARSA(λ), entre otras, llegan a ciertas **conclusiones** estrechamente **relacionadas** con gran parte de lo que nos pasa.

Tras estudiar la información presentada, y si se **relaciona** con nuestra **situación**, se puede decir que:

a) SARSA(λ) siempre converge en un **menor** número de episodios que SARSA.
b) SARSA(λ) **no** es necesariamente más rápido que SARSA, debido a que emplea una política más "**conservadora**", lo que hace que se vuelva más "precavido" de lo normal, penalizando sus tiempos.

c) Por último, los métodos que emplean Trazas de Elegibilidad **no** siempre son capaces de alcanzar la **convergencia** en determinadas situaciones o antes de cierto número de episodios de entrenamiento, lo que sucede en el caso 4.

Así que puede decirse que los resultados que obtenemos son totalmente **normales** si se emplea SARSA(λ). Y, es más, conviene tener presente que en algunas ocasiones **no** es recomendable emplear las Trazas de Elegibilidad, porque quizá **no** lleguen a alcanzar una convergencia de resultados.

Q-Learning

En este caso, los resultados deberían **diferir** de SARSA. A priori, no sabemos si para mejor o para peor, porque depende del trazado, ya que el empleo de una política más agresiva, ***greedy***, puede hacer que se vea penalizado más veces o que alcance la salida de forma más rápida.

En este caso, los **resultados** obtenidos han sido los de la *Ilustración 69*:

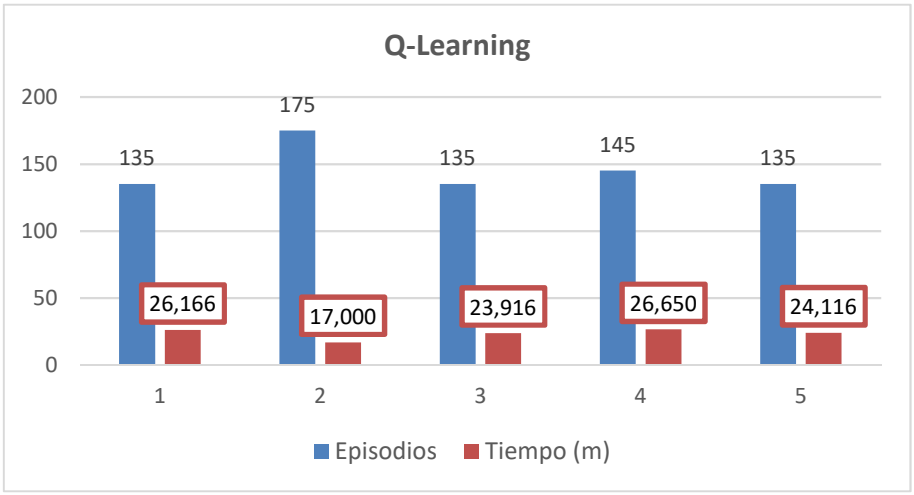


Ilustración 69. Resultados: Laberinto Virtual+Estándar - Q-Learning

Los tiempos obtenidos han sido **similares** en los casos 1, 3, 4 y 5, empleando unos 25 minutos y 140 episodios para entrenar al agente desde todas y cada una de las celdas posibles.

En el caso 2, los resultados difieren de forma "**inusual**" porque, si bien se entrena al agente durante **más** episodios (170), tarda bastante **menos** tiempo en llevarlos a término. Podría parecer extraño, pero **no** lo es si se tiene en cuenta un hecho del que ya hemos hablado: cuanto **mejor** "entrenado" está el agente, su convergencia es **mayor**, por lo que encontrará la salida más **rápidamente**. Y éste es precisamente el hecho que explica lo que ocurre con el caso 2, en el que el agente se entrena durante más episodios, pero con una mejor experiencia, por lo que tarda menos en ejecutarlos.

Aparte de ello, es claramente más **lento** que SARSA, tanto por su mayor número de episodios de entrenamiento como por el tiempo empleado, llegando a ser incluso de más del doble. Y ha sido así porque el laberinto planteado puede hacer que el agente reciba más **penalizaciones** de las que cabría esperar, porque existen unos cuantos "callejones sin salida" que a buen seguro habrán lastrado su desempeño.

Q-Learning(λ)

Para este caso, los resultados deberían ser **mejores** que los del caso "estándar". El uso de Trazas de Elegibilidad debería permitir mejores resultados, pero **quizá** suceda como con SARSA(λ), y algún caso puede que no llegue a alcanzar la convergencia necesaria. Aparte de ello, también es interesante saber si otra de las conclusiones a las que llega el estudio de (Grzes & Kudenko, 2008) es cierta: Q-Learning(λ) es más **rápido** que SARSA(λ) en las **mismas** condiciones.

En este caso, los **resultados** obtenidos han sido los de la *Ilustración 70*:

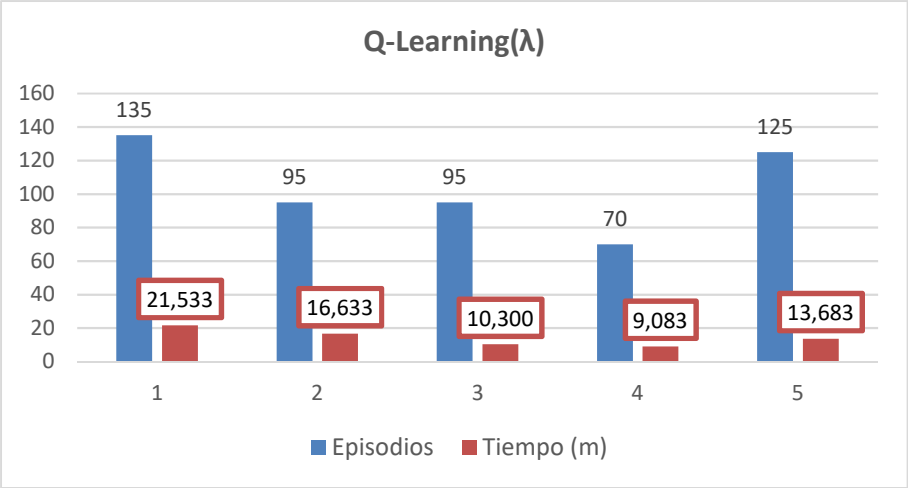


Ilustración 70. Resultados: Laberinto Virtual+Estándar - Q-Learning(λ)

Y sí, efectivamente los resultados han sido **mejores** que para el algoritmo estándar de Q-Learning, **mucho** mejores. Si bien es cierto que en el caso 1 el agente ha efectuado el mismo número de episodios, lo ha hecho en bastante menos tiempo (entre 3 y 5 minutos menos). Pero si nos fijamos en otros casos, como el 3 o el 4, los resultados han sido realmente buenos, reduciéndose el número de episodios a casi la **mitad** (en el 4) y el tiempo empleado a… ¡**menos** de la mitad! (casos 3 y 4).

Así pues, el uso de Trazas de Elegibilidad en el caso del Algoritmos Q-Learning está plenamente **justificado**, y sí resulta, generalmente, más **rápido** que SARSA(λ).

DQN

Éste es un caso, digámoslo así, "**especial**". El hecho de emplear Q-Learning conjuntamente con redes neuronales **no** implica necesariamente que sea más rápido. Dependerá de muchos **otros** factores, como el tipo de laberinto o la máquina que estemos manejando, porque podría hacerse uso de una **GPU** especializada para tal fin (gracias a TensorFlow) y ser su aprendizaje mucho más veloz. En este caso he optado por **no** hacerlo, para **igualar** al máximo las condiciones con el resto de algoritmos, así que todo el peso recaerá en la **CPU** que, si bien es potente, no lo es tanto para estos fines.

En este caso, los **resultados** obtenidos han sido los de la *Ilustración 71*:

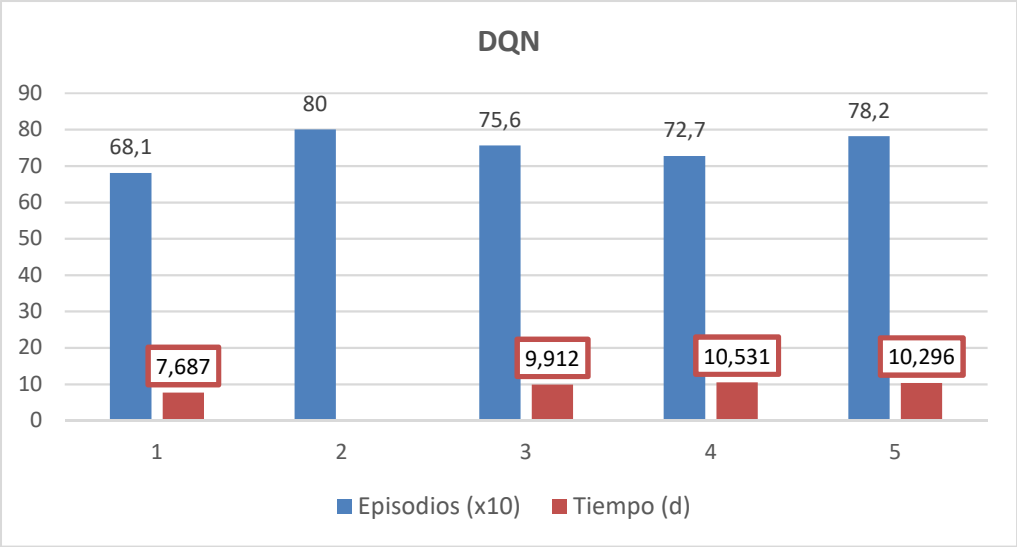


Ilustración 71. Resultados: Laberinto Virtual+Estándar - DQN

Así pues, en cuanto al **tiempo** invertido, los resultados han sido mucho, **mucho mayores**, porque ya estamos hablando de **días** de computación. Como se ha comentado, es cierto que se podría haber acelerado el proceso empleando la potencia de una GPU potente, pero la intención era **compararlo** en igualdad de condiciones, así que se ha empleado sólo la CPU, al igual que en los casos anteriores.

En cuanto a los casos en sí, en general han tardado una media de 9,6 días de computación y 736,5 episodios, resultados más o menos **homogéneos** y dentro de lo esperable, que han variado debido a la velocidad de aprendizaje en unos casos o en otros, que puede que hayan sido más acertados a la hora de elegir valores (ver *Ilustración 72*). Ahora bien, el segundo caso **no** ha llegado a finalizar su entrenamiento, ya que ha alcanzado el **límite** de episodios (800) sin haber alcanzado la convergencia, por lo que ha **parado**.

Así pues, los resultados son bastante claros y, para resolver este laberinto con una potencia de proceso **estándar**, DQN **no** es una solución **recomendable** dentro de las existentes en cuanto a RL, quedando más como una **muestra** del funcionamiento de las redes neuronales que como algo práctico y utilizable.

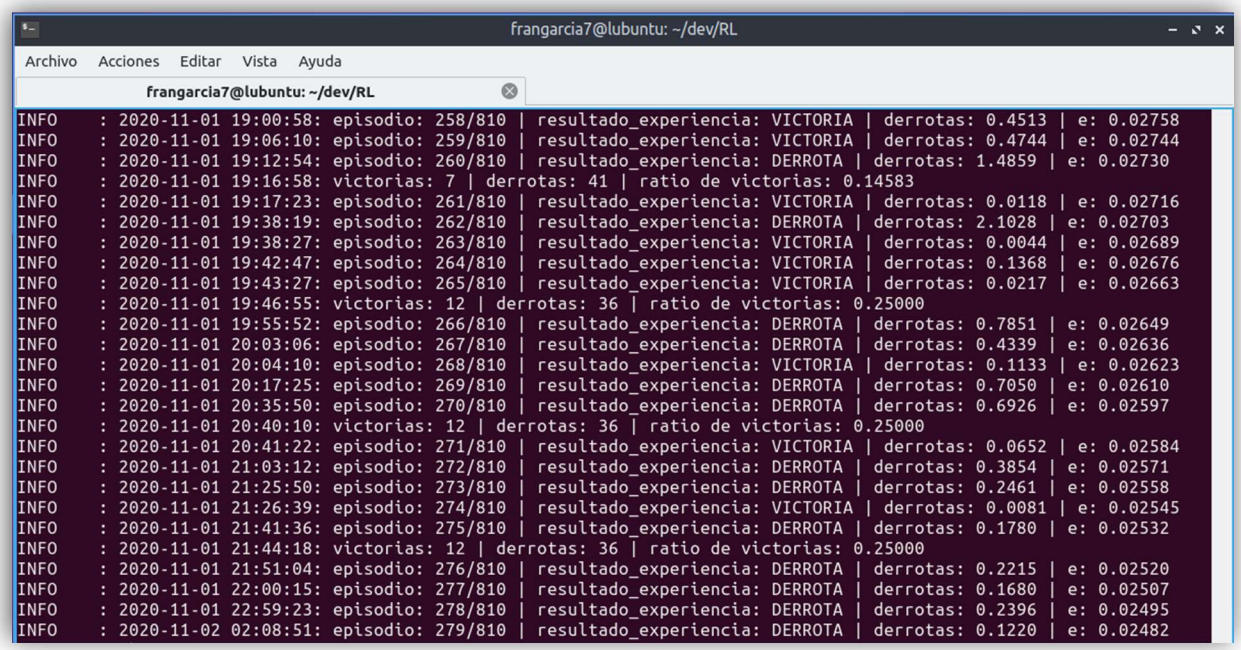


Ilustración 72. DQN: Información de aprendizaje

8.4.2. Laberinto Real+Estándar

Para pasar del entorno simulado al entorno **real** se ha tratado de hacer los **mínimos cambios** en su lógica, para poder compararlos de la forma más **similar** posible, pero tal y como se ha comentado, el robot avanzará de **2 en 2** celdas, ahorrando **tiempo** de desplazamiento, ya que el diseño del laberinto lo **permite**.

Se ha empleado el laberinto **Real+Estándar**, idéntico en concepción al Virtual+Estándar, para poder **comparar** los resultados con los vistos antes de implementar ninguna mejora. Sobre todo, lo que más interesará es conocer cómo se ha visto afectada la **velocidad** y la **precisión** de la **navegación**, ya que el hecho de disponer de ***encoders*** y de más sensores de **líneas** debería permitir al robot dibujar **trayectorias** mucho mejores, evitando correcciones y posibles pérdidas de tiempo.

Q-Aleatorio

Una vez más, este algoritmo sigue demostrando que es realmente **pésimo**, y en los más de 20 intentos que he vuelto a llevar a cabo, no se ha podido llegar a la meta ni una sola vez. Los resultados son demasiado **aleatorios**, y resulta totalmente **impracticable** y **nada** recomendable.

Por ello, este algoritmo quedará más como una mera **anécdota** y un recordatorio de que, por mucha potencia de la que se disponga, un buen planteamiento lo es todo.

SARSA, SARSA(λ), Q-Learning, Q(λ) y DQN

En este punto cabe recordar que, para estas cuatro estrategias, el **procedimiento** seguido es el siguiente:

1) Ejecución del **algoritmo** de RL deseado en el **PC**.
2) Extracción/volcado de la **ruta** óptima entre la entrada y la salida establecidas.
3) Carga de dicha ruta en la placa **Arduino**.
4) Recorrido **físico** del robot por el laberinto, empleando la ruta óptima.

Así, se tomarán como **referencia** los tiempos del caso simulado para la resolución de los aprendizajes, ya que deberían ser promedios muy similares, al **no** tener que **desplazarse** el robot. Tras ello, desglosaremos el tiempo que tarda el robot en seguir una ruta óptima con todas las mejoras efectuadas. Que, al fin y al cabo, va a servir para **comparar** el tiempo que invierte el robot **mejorado** respecto de su versión previa (la construida exclusivamente con las piezas del kit comercial) en recorrer el laberinto.

Así, los **tiempos** obtenidos, teniendo en cuenta únicamente los movimientos de desplazamiento del robot, han sido los que pueden observarse en la *Ilustración 73*:

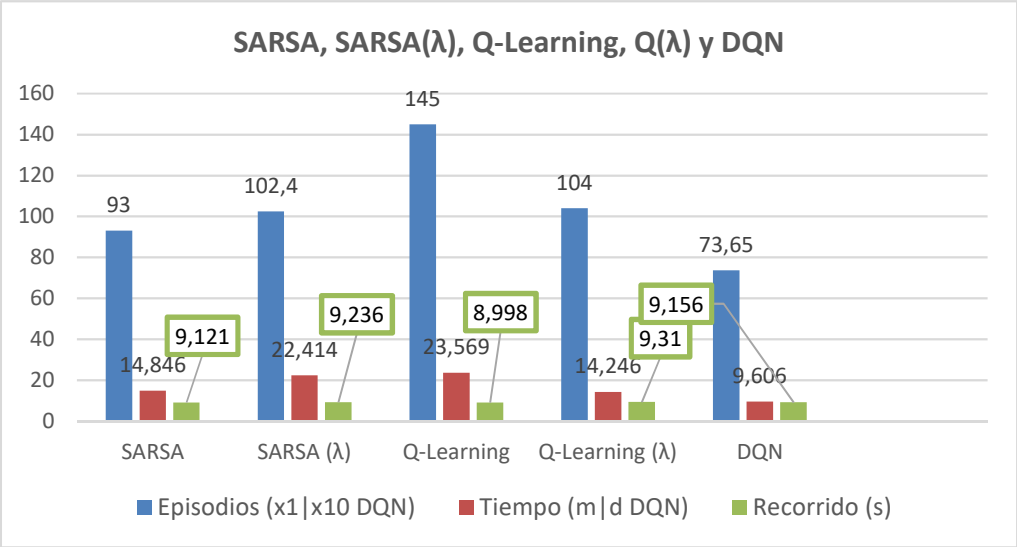


Ilustración 73. Resultados: Laberinto Real+Estándar - SARSA, SARSA(λ), Q-Learning, Q(λ)

Si comparamos el tiempo invertido en la navegación por el laberinto, las **mejoras** están plenamente **justificadas**. Una media de casi **3 segundos** es el tiempo que se **ahorra** al recorrerlo, que en poco más de 9 segundos de media supone un **25%** del tiempo total. Y eso a pesar de que el **peso** extra del robot debe tenerse en cuenta en su avance, y las numerosas modificaciones que he tenido que hacer en el código para controlar los **motores** y ajustar el **PID** así lo atestiguan. Pero su reserva de **potencia**, unida a un uso inteligente de los ***encoders*** (dos, finalmente) y los sensores de **líneas** extra, han marcado la **diferencia**.

8.4.3. Laberinto Virtual+Desafiante

Q-Aleatorio

Al igual que en el caso del laberinto Virtual+Estándar, porque si ya era misión imposible aplicar el Aprendizaje por Refuerzo en dicha ocasión, ahora es directamente **impensable**. La complejidad de este laberinto hace que este algoritmo resulte totalmente **inútil** en este caso.

SARSA

Su funcionamiento va a ser interesante de comparar con Q-Learning cuando veamos sus resultados, porque **quizá** en esta ocasión **no** reciba tantas **penalizaciones** debidas a su política "ávida". Así pues, veamos sus **resultados** en la *Ilustración 74*:

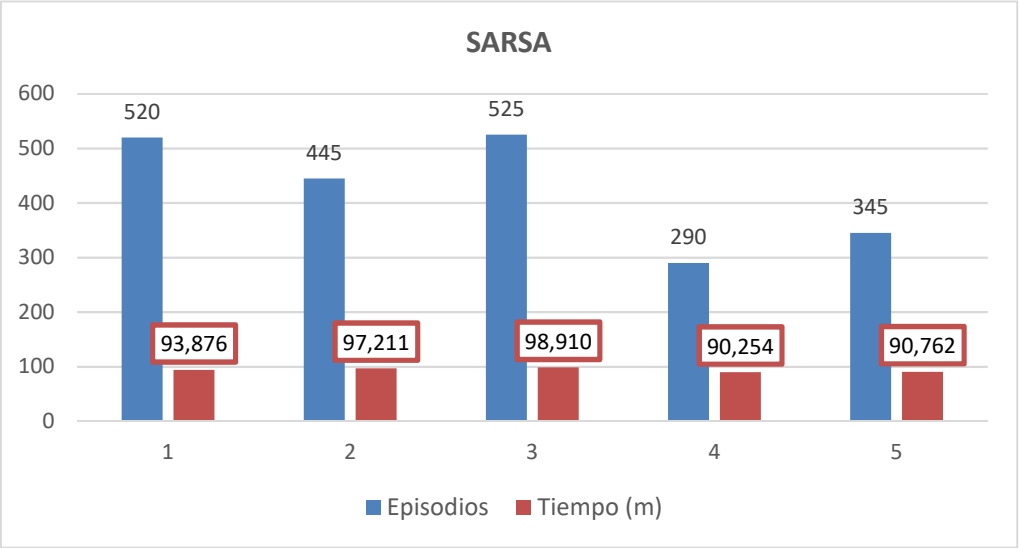


Ilustración 74. Resultados: Laberinto Virtual+Desafiante - SARSA

Como puede observarse, los resultados son bastante **estables** en cuanto al **tiempo** empleado, promediando cerca unos 93 minutos. Pero, por otra parte, los **episodios** invertidos en alcanzar la convergencia oscilan entre los 290 y los 525, una **diferencia** considerable. Así que ¿cómo es posible que en casi el doble de episodios se invierta casi el mismo tiempo? La respuesta es sencilla: existe un punto común a todos los casos en el que las rutas importantes están "**aprendidas**", y al llegar a él, el aprendizaje se acelera rápidamente. De este modo, y a pesar de que el número de episodios es diferente, el tiempo final tenderá a **igualarse**, que es exactamente lo que puede observarse en la gráfica.

SARSA(λ)

En este caso se hace uso de las Trazas de Elegibilidad, lo que debería llevar a **mejores** tiempos de ejecución, pero veamos sus **resultados** en la *Ilustración 75*:

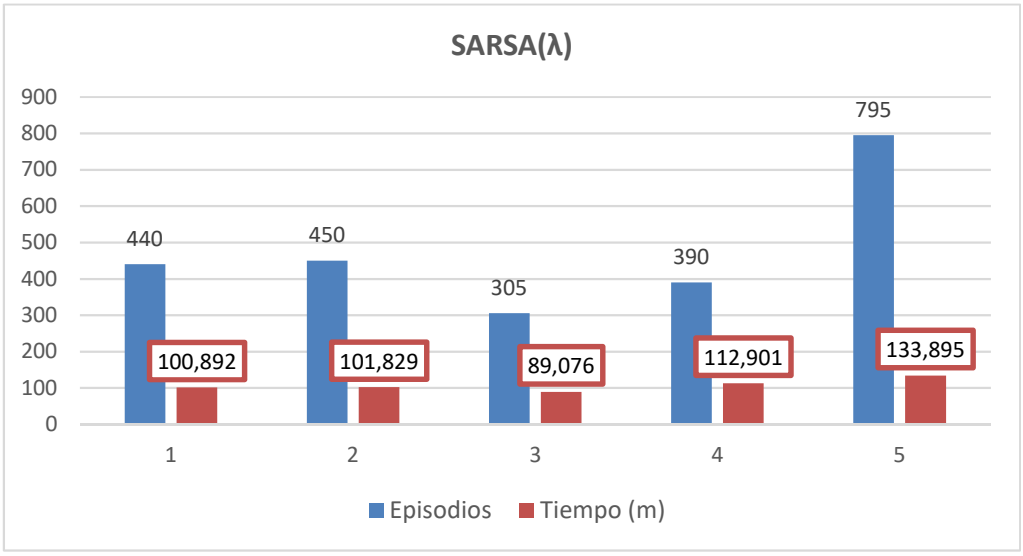


Ilustración 75. Resultados: Laberinto Virtual+Desafiante - SARSA(λ)

Sus resultados, en cuanto al **tiempo** empleado, son relativamente **estables**, salvo en el caso 5. **No** es así en cuanto al número de **episodios**, por lo que es de suponer que suceda algo similar al caso **anterior**, en el que los tiempos que se obtienen tienden a **equilibrarse** cuando se acelera el aprendizaje. Aparte de ello, los tiempos son **superiores** al caso estándar, aunque no por mucho, lo que indica que el empleo de las Trazas de Elegibilidad puede que **no** beneficie a laberintos **grandes** o **complejos**. El hecho de tener que actualizar **todos** los estados previos es posible que pase factura a los tiempos totales.

Por último, el caso 5 vuelve a reflejar lo que vimos en el laberinto Virtual+Estándar, en el que hablamos del **estudio** de (Grzes & Kudenko, 2008), y es que en algunas ocasiones **no** es recomendable emplear las Trazas de Elegibilidad, porque quizá **no** se llegue a alcanzar una **convergencia** de resultados o tarde demasiado.

Q-Learning

En este caso, los resultados deberían **diferir** de SARSA. Al igual que antes, no sabemos si para mejor o para peor, porque **depende** del trazado, y éste ha variado sustancialmente. En este caso, los **resultados** obtenidos han sido los de la *Ilustración 76*.

Los **tiempos** obtenidos han sido similares en los casos 2, 3 y 4, mientras que en el caso 1 son algo mejores y en el 5 algo peores, por lo que podríamos decir que son bastante **estables**, en torno a los 126-127 minutos.

En cuanto a los **episodios**, vuelve a pasar algo similar al caso de SARSA, así que seguramente suceda algo parecido, que al llegar a cierto punto el aprendizaje se acelera lo suficiente como para llegar a **equilibrar** los tiempos finales que se obtienen.

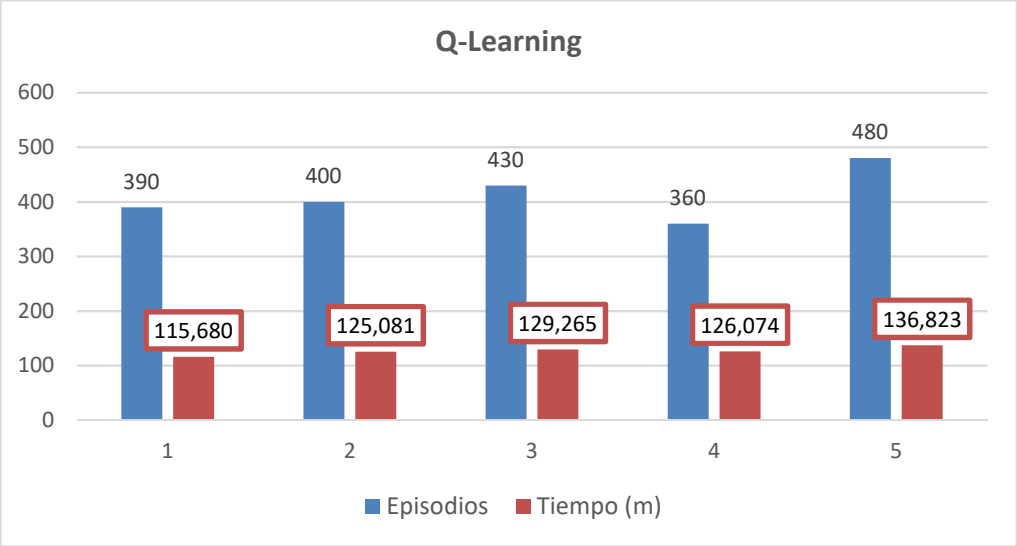


Ilustración 76. Resultados: Laberinto Virtual+Desafiante - Q-Learning

Aparte de ello, sí es claramente más **lento** que SARSA, sobre todo por el tiempo empleado, del orden de 30 minutos superior, aproximadamente un 36%. Una vez más, el agente ha recibido más **penalizaciones** de las que cabría esperar, seguramente debidas a todas las "trampas" de las que este laberinto hace gala, y su política "ávida" le ha acabado pasando factura.

Q-Learning(λ)

Para este caso, los resultados deberían ser **mejores** que los del caso "estándar". Y será interesante saber si es capaz de mejorar los tiempos de su versión "estándar", además de saber si Q-Learning(λ) finalmente puede ser más rápido que SARSA(λ) en las **mismas** condiciones, como el estudio de (Grzes & Kudenko, 2008) mencionado con anterioridad sugiere. En este caso, los **resultados** han sido los de la *Ilustración 77*.

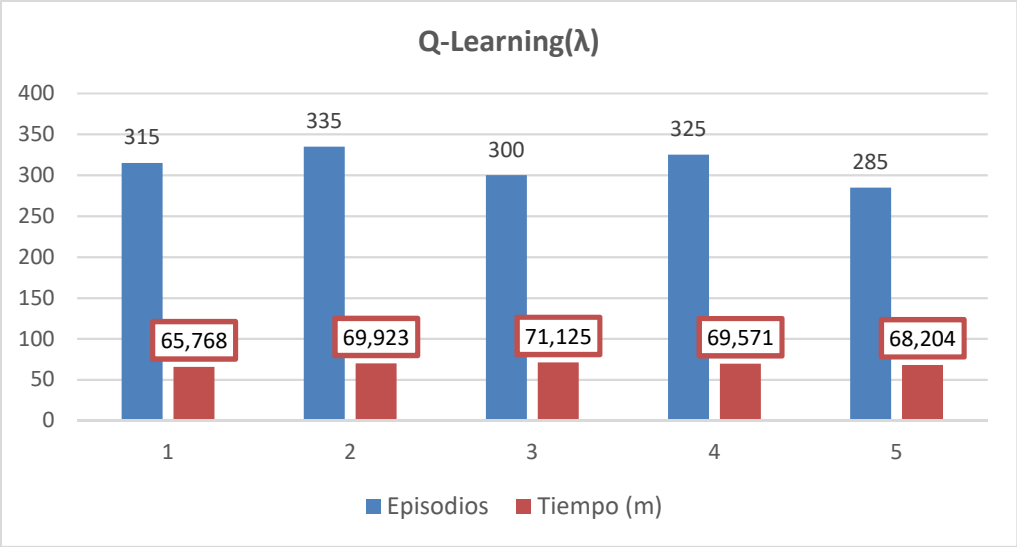


Ilustración 77. Resultados: Laberinto Desafiante - Q-Learning(λ)

Y sí, efectivamente los resultados han sido **mejores** respecto del algoritmo estándar de Q-Learning, y también respecto de cualquiera de las variantes de SARSA. Menos **episodios** invertidos y, sobre todo, **tiempos** mucho menores, invirtiendo casi 30 minutos menos, lo que representa un 26% menos.

Así pues, el uso de Trazas de Elegibilidad en el caso del Algoritmo Q-Learning está plenamente **justificado** en este laberinto, porque su rapidez así lo acredita. Una política *greedy* en conjunción con la **actualización** de los estados previos llevada a cabo por las Trazas de Elegibilidad es capaz de **mejorar** los resultados de forma ostensible cuando nos enfrentamos a un laberinto de ciertas **dimensiones**.

DQN

Si ya en el caso del laberinto "Estándar" hablábamos de **días** para alcanzar la convergencia necesaria, en el caso del laberinto "Desafiante" resulta un algoritmo **impracticable**, si no es mediante un montaje de GPUs en ***array*** o, directamente, de un **supercomputador** dedicado a entrenar redes neuronales.

Así pues, para resolver el laberinto actual con una potencia de proceso estándar, DQN **no** es una solución **recomendable** dentro de las existentes en cuanto a Aprendizaje por Refuerzo.

8.5. Análisis de los resultados

Se ha empleado el **mismo** trazado en los laberintos Virtual+Estándar y Real+Estándar, por lo que podrán extraerse **conclusiones** realistas. Hay que notar que los tiempos del laberinto Real+Estándar son únicamente los que se emplean en **navegar** por el laberinto de forma **física**, ya que el aprendizaje y la obtención del recorrido óptimo se habrá llevado a cabo con anterioridad y **precargado** en la placa Arduino gracias al **Virtual+Estándar**, gracias a haber **separado** el mapeo de la propia navegación.

Aparte de ello, y tal y como ya se ha comentado, se ha desarrollado un laberinto más "desafiante", para poner **a prueba** las capacidades de los algoritmos de RL, contando con una versión únicamente **virtual** (el laberinto Virtual+Desafiante), **reproducible** fácilmente en la realidad si se desea.

Los **resultados** medios pueden verse en la *Tabla 6*:

Laberinto	**Virtual+Estándar**		**Real+Estándar**	**Virtual+Desafiante**	
Algoritmo	**Episodios (x1|x10 DQN)**	**Tiempo (m|d DQN)**	**Recorrido (s)**	**Episodios**	**Tiempo (m)**
Q-Aleatorio	---	---	---	---	---
SARSA	93	14,846	9,121	425	94,202
SARSA(λ)	102,4	22,414	9,236	476	107,718

Laberinto	Virtual+Estándar		Real+Estándar	Virtual+Desafiante	
Q-Learning	145	23,569	8,998	412	126,584
Q-Learning(λ)	104	14,246	9,31	312	68,918
DQN	73,65	9,606	9,156	---	---

Tabla 6. Algoritmos de Aprendizaje por Refuerzo: Resultados medios

8.5.1. Laberinto Virtual+Estándar

La implementación de **estadísticas** al finalizar la ejecución de cada algoritmo me ha servido para observar el comportamiento de cada uno de ellos de forma más **precisa**, tal y como puede verse en la *Ilustración 78*, en la que se presenta el **entorno** visual de *testing* desarrollado gracias a un uso extensivo de PyPlot.

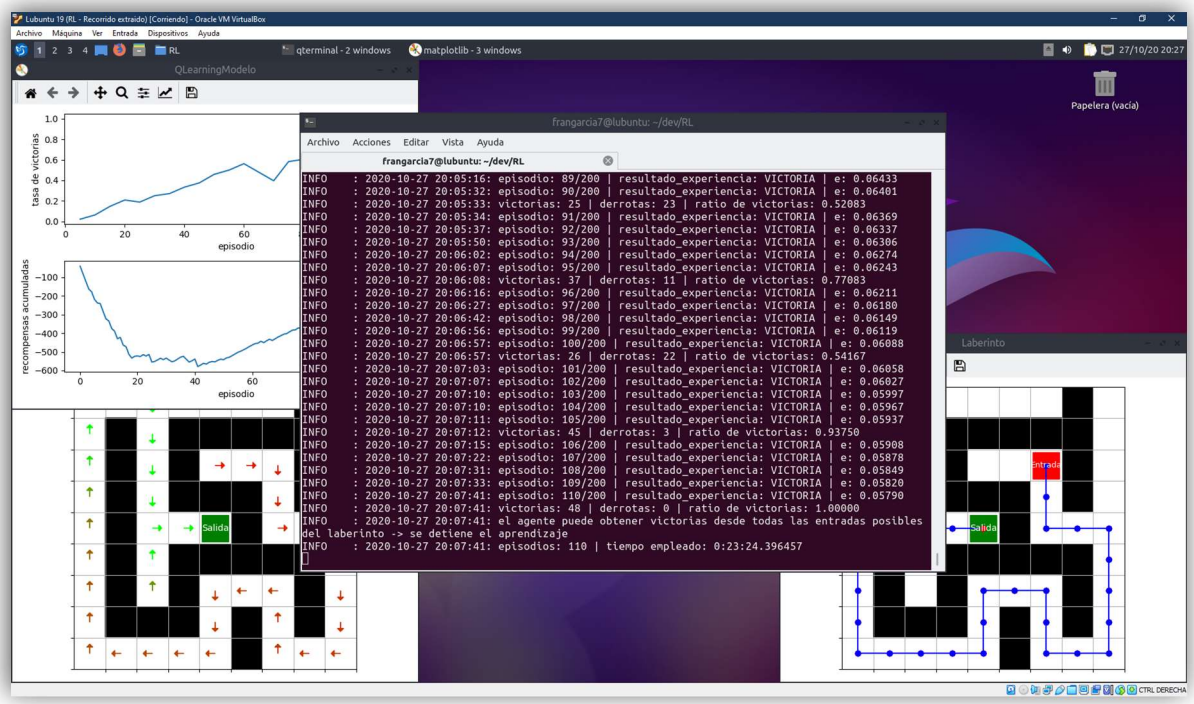


Ilustración 78. Entorno de Aprendizaje: Estadísticas

Concretamente, este caso **muestra** cómo gracias a Q-Learning se ha alcanzado la convergencia en 110 episodios y en 23 minutos y 24 segundos.

Así, en lo que respecta a los **algoritmos**, puede decirse que:

a) **Q-Aleatorio**: Debe emplearse únicamente como **último** recurso. Su **aleatoriedad** es **desesperante**, con unos **tiempos** extremadamente **elevados**, si es que el robot acaba llegando a la meta.

b) **SARSA**: Es un **buen** algoritmo, a tenor de los resultados. Además, el trazado del laberinto, que tiene bastantes "callejones sin salida", le **beneficia** en bastantes ocasiones, porque utiliza una política **conservadora** y no sufre de tantas penalizaciones como sí sucede, en general, con Q-Learning, que optará por asumir demasiados riesgos.

c) **SARSA(λ)**: El uso de las Trazas de Elegibilidad es **buena** idea. El hecho de poder "recordar" resultados y de mejorar su entrenamiento antes de que acabe un episodio suele resultar **beneficioso**, lo que le permite recortar los **tiempos** en la mayoría de ocasiones. **No** ha sido así en este caso, pero **tampoco** sus resultados se han visto muy penalizados, por lo que, en mi opinión, debería optarse por SARSA(λ) en lugar de SARSA siempre que sea **posible**.

d) **Q-Learning**: A pesar de que a priori podría haber resultado mejor que SARSA, en este caso **no** lo ha sido. Como he comentado, su política ***greedy*** hace que el agente asuma demasiados **riesgos**, por lo que recibirá muchas penalizaciones de no encontrar pronto la salida. Y es justo lo que ha sucedido, en este caso, debido al trazado particular del laberinto.

e) **Q-Learning(λ)**: Su uso está plenamente justificado. Una política *greedy* resultará perfecta para avanzar de forma rápida, y el uso de las Trazas de Elegibilidad ayudará a **controlar** en cierta forma su progreso de aprendizaje, evitando riesgos innecesarios. En este caso, ha resultado ser una **gran** combinación, **reduciendo** los tiempos y los episodios de aprendizaje de forma clara.

f) **DQN**: El uso de redes neuronales en el Aprendizaje por Refuerzo resulta muy flexible, porque es el diseño más **adaptable** de todos los que hemos visto. Ahora bien, el esfuerzo **computacional** que requiere es muy importante, por lo que se debe disponer de bastantes **recursos**, y dedicados a ser posible, para entrenar las redes con la suficiente **velocidad**.

8.5.2. Laberinto Real+Estándar

A la vista de los resultados, puede decirse que la incorporación de los nuevos componentes ha sido plenamente **satisfactoria**, recortando los tiempos de navegación claramente. Aparte de, por supuesto, tener un coste **asequible**, uno de los objetivos que me marqué para esta fase del trabajo.

De los **componentes mejorados** implementados en el robot, puede concluirse que:

- **Arduino Mega2560**: Una auténtica maravilla. Bastante más memoria, algo más rápida (aunque la Uno no era para nada lenta), muchas más conexiones, más *timers*… Seguramente, la mejora más recomendable de todas, sobre todo si se piensa en incorporar gran cantidad de sensores y de actuadores al proyecto que tengamos entre manos.
- **Sensores** de **ultrasonidos**: Son un componente bastante **asequible** en cuanto a coste, y su programación tampoco es especialmente compleja una vez que se conoce su funcionamiento. Pero el resultado es muy bueno, porque el prescindir del servo hace que el robot tome las referencias que necesita de forma **instantánea**, lo que hace que el robot se mueva a mucha más velocidad por el laberinto. De todas formas, en este caso **no** han podido brillar tanto como habría sucedido si se volvieran a ejecutar los algoritmos **laberínticos** originales, en los que los **giros** y **cambios** de

orientación eran casi constantes. Ahora el robot parte con la ruta **óptima** precargada, por lo que no tendrá que corregir su rumbo a medida que avanza, y éstos sensores son **prescindibles**. De ahí, que finalmente haya empleado el laberinto Real en su versión "**reducida**", porque el robot es capaz de moverse **sin** referencias sobre sus obstáculos, necesitando tan sólo de **marcas** en el suelo para guiarle gracias a sus sensores de líneas.

- **Sensores** de **líneas**: Seguramente sea porque los sensores son diferentes, pero sus mediciones son más **precisas** que las de los sensores empleados con anterioridad, aparte de que **no** necesitan de **calibración** previa. Decir que, en todas las pruebas que he hecho, no he visto que hayan cometido error ninguno a la hora de detectar la cinta aislante empleada, son muy **fiables**.
- ***Encoders***: Si bien presuponía que iban a ser una de las mejoras que más relevancia iban a tener, porque son capaces de calcular el avance de las ruedas, la velocidad de desplazamiento del robot hace que giren demasiado rápido para poder detener el robot en el momento exacto en el que sería deseable. Por ello, siempre se va a introducir algo de error en el avance y los giros y, por pequeño que éste sea, hay que tenerlo siempre presente.
- **Controlador PID**: Sigue resultando bastante **recomendable** y hasta **necesario**, porque los *encoders*, a pesar de que resultan de bastante ayuda, **no** son precisos al 100%, especialmente en los giros. Por suerte, **combinado** con los sensores ha suavizado algo más los movimientos del robot.
- **Motores**: Han tenido que soportar bastante **peso** extra, y todos los componentes adicionales que se han incorporado se notan, especialmente en los **giros**. Por suerte, son cuatro y tenían bastante **potencia** de reserva, así que el **margen** de ajuste-mejora que proporcionaban ha resultado ser suficiente para mover el conjunto del robot de forma adecuada.
- **Mando IR**: Gracias a los *timers* adicionales de la Arduino Mega, ahora el mando IR funciona a la **perfección**, pudiendo obtenerse sus códigos de señalización sin problema. Aparte de ello, el hecho de que ahora es posible prescindir del servo hace que las posibles **interferencias** entre dichos *timers* se reduzcan a la mínima expresión.

8.5.3. Laberinto Virtual+Desafiante

Este laberinto es el auténtico **reto**, creado ex profeso para poner **a prueba** la capacidad de los algoritmos y de la máquina sobre la que se ejecutan. Múltiples bucles, callejones sin salida aparente, rutas que se solapan con otras, penalizaciones casi a la vuelta de cada giro… En este caso, lo que más interesa es su **eficacia**, si son capaces de **encontrar** la salida, y no tanto sus tiempos.

Teniendo presente este hecho, de los **resultados** obtenidos al aplicar los algoritmos puede decirse que:

a) **Q-Aleatorio**: Debido a su naturaleza aleatoria, no se tratado ni de ejecutar. Un laberinto de este tamaño, con estas características, es algo inalcanzable para este algoritmo.

b) **SARSA**: Es un **buen** algoritmo, y lo ha **demostrado** una vez más. El hecho de ser más "conservador" que Q-Learning le ha **beneficiado** en este laberinto, porque está plagado de **penalizaciones**, y algunas de ellas son incluso más severas de lo esperado.

c) **SARSA(λ)**: El uso de las Trazas de Elegibilidad en este caso **no** ha producido mejores resultados. Y **no** es que sea **peor** que el método "estándar" de SARSA, porque sus tiempos han sido **similares**, sino que más bien es posible que se vea algo lastrado por tener que hacer **tantas** actualizaciones en los estados, lo que repercute en no poder obtener mejores tiempos.

d) **Q-Learning**: Era de esperar que un laberinto como éste, pensado para **penalizar** más que para recompensar, haya dejado a Q-Learning como el **peor** algoritmo a emplear en él. Si bien el número de episodios es similar, no lo son los **tiempos**, reflejando el hecho de que tarda más en aprender cuál es la ruta óptima, debido a que **no** es capaz de encontrar el mejor camino con tanta rapidez como lo hace SARSA, en este caso.

e) **Q-Learning(λ)**: Y, de nuevo, la **combinación** de una política *greedy* y el uso de las Trazas de Elegibilidad es **imbatible**. Si se toman decisiones rápidas y se **aprende** sobre la marcha de los errores pasados, acabará repercutiendo en **mejores** resultados. Y lo hace de forma clara, mejorando por bastante tanto el número de **episodios** de aprendizaje como los **tiempos** que se invierten en alcanzar la convergencia.

f) **DQN**: El esfuerzo **computacional** que requiere es muy, muy elevado. Y un laberinto como éste **no** es el mejor para probarlo, porque con los mismos recursos que se han dedicado a los casos anteriores, seguramente estuviéramos hablando de **semanas**, más que de días de computación.

8.5.4. Algoritmos Laberínticos vs Aprendizaje por Refuerzo

Llegados a este punto, y gracias a las múltiples **pruebas** con **todos** los algoritmos y los laberintos presentados, ya es posible empezar a establecer una **comparación** entre los AL[34] seleccionados y los que emplean **RL**, uno de los objetivos principales del presente trabajo. Y, para poder hacerlo, debemos **equiparar** los resultados al máximo.

Por ello y, antes de nada, deben hacerse ciertos **apuntes** al respecto:

a) El código de los **AL** escrito originalmente para la Arduino ha sido **reescrito** en lenguaje **C++** (el más cercano al Arduino IDE), para emplear la **misma CPU** y establecer comparaciones reales. Además, se mostrarán los datos **originales**, para poder comparar la Arduino con un **PC**.

b) En cuanto a tiempos de resolución, nos centraremos en los laberintos en su variante **Virtual+Estándar**, ya que el Virtual+Desafiante **sólo** es posible resolverlo mediante **RL**.

[34] AL: Algoritmos Laberínticos

c) Al resolver el laberinto **Virtual+Estándar** mediante **RL** se ha obtenido la **mejor ruta** hasta la **salida** desde **todas** las celdas de **entradas** posibles, y no sólo desde **una**, como en el caso de los **Algoritmos Laberínticos**. Para poder establecer una comparación adecuada, habrá que **calcular**:

1. **Número** de **movimientos** totales empleados:

   Siendo:

   M_{ES} = Movimientos empleados por el **recorrido óptimo** (entrada → salida)
   M_{RMI} = Movimientos del algoritmo **RMI**
   M_{IOPT} = Movimientos del algoritmo **RMI Optimizado**
   M_{IND} = Movimientos del algoritmo de **Inundación**

$$M_{RMI} = \text{es directo}$$
$$M_{IOPT} = M_{RMI} + M_{ES}$$
$$M_{IND} = M_{RMI} + M_{ES} * 2 \text{ (ida y vuelta)}$$

Ecuación 15. Algoritmos Laberínticos: Cálculo de movimientos empleados

Números que, plasmados **sobre** el propio laberinto, se verían de la siguiente forma:

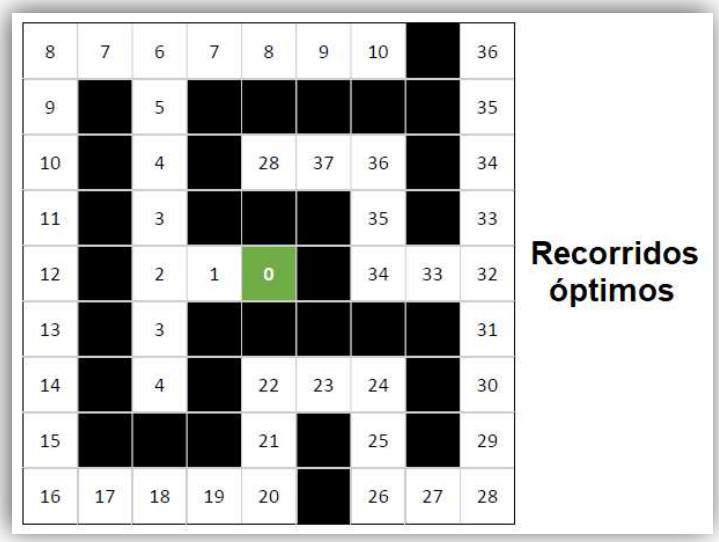


Ilustración 79. Algoritmos Laberínticos: Movimientos empleados en los recorridos óptimos

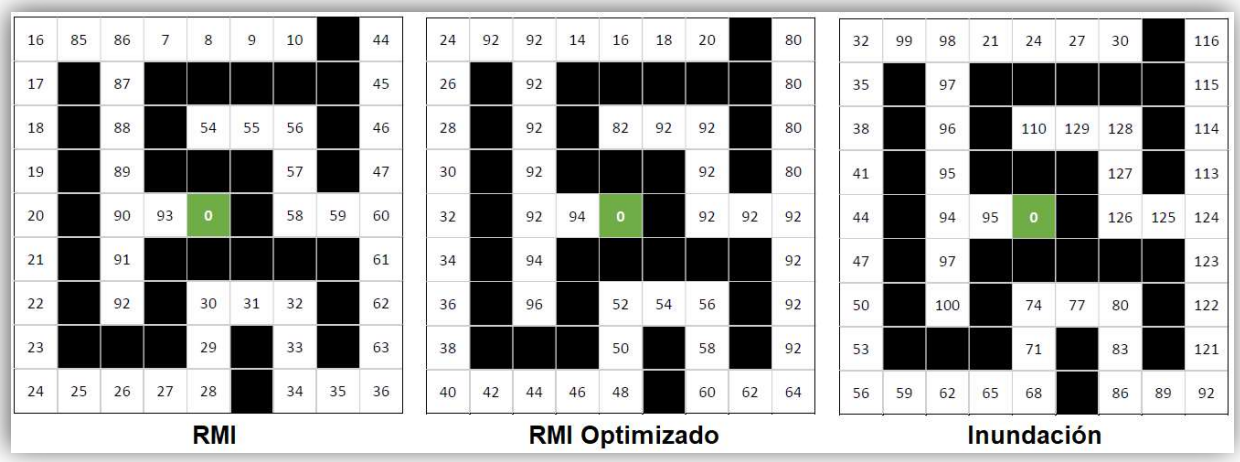


Ilustración 80. Algoritmos Laberínticos: Movimientos desde las entradas hasta la salida

2. **Tiempo** empleado en cada **movimiento** del laberinto (con RMI, por ejemplo):

   Siendo:

   $\mathbf{T_{MOV}}$ = Tiempo empleado por movimiento
   $\mathbf{T_{RMI}}$ = Tiempo total del algoritmo RMI

$$\mathbf{T_{MOV}} = T_{RMI} / M_{RMI} = 3{,}43s / 24mov = \mathbf{0{,}142s / mov}$$

Ecuación 16. Algoritmos Laberínticos: Tiempo empleado por movimiento (Arduino)

$$\mathbf{T_{MOV}} = T_{RMI} / M_{RMI} = 0{,}464s / 24mov = \mathbf{0{,}01933s / mov}$$

Ecuación 17. Algoritmos Laberínticos: Tiempo empleado por movimiento (PC)

   Notar que este cálculo sufrirá ínfimas variaciones en **cualquier** punto del laberinto porque, en el caso del laberinto **virtual**, el robot **no** hace uso de sensores ni actuadores físicos, y la comprobación de obstáculos o los giros es **instantánea**.

3. **Tiempo** total empleado en resolver el laberinto **Virtual+Estándar** desde **todas** las posibles entradas, para poder **equiparar** los resultados a los de **RL**:

   Siendo (desde **todas** las **entradas** hasta llegar a la **salida**):

   $\mathbf{MT_{RMI}}$ = Movimientos de **RMI**
   $\mathbf{MT_{IOPT}}$ = Movimientos de **RMI Optimizado**
   $\mathbf{MT_{IND}}$ = Movimientos de **Inundación**
   $\mathbf{TT_{RMI}}$ = Tiempo de **RMI**
   $\mathbf{TT_{IOPT}}$ = Tiempo de **RMI Optimizado**
   $\mathbf{TT_{IND}}$ = Tiempo de **Inundación**

$$\mathbf{TT_{RMI}} = MT_{RMI} * T_{MOV} = 2.148mov * 0{,}142s/mov = 305{,}016s / 60s = \mathbf{5{,}083m}$$
$$\mathbf{TT_{IOPT}} = MT_{IOPT} * T_{MOV} = 3.058mov * 0{,}142s/mov = 434{,}236s / 60s = \mathbf{7{,}237m}$$
$$\mathbf{TT_{IND}} = MT_{IND} * T_{MOV} = 3.968mov * 0{,}142s/mov = 563{,}456s / 60s = \mathbf{9{,}390m}$$

Ecuación 18. Algoritmos Laberínticos: Tiempos totales empleados desde todas las celdas (Arduino)

$$\mathbf{TT_{RMI}} = MT_{RMI} * T_{MOV} = 2.148mov * 0{,}01933s/mov = 41{,}528s / 60s = \mathbf{0{,}692m}$$
$$\mathbf{TT_{IOPT}} = MT_{IOPT} * T_{MOV} = 3.058mov * 0{,}01933s/mov = 59{,}111s / 60s = \mathbf{0{,}985m}$$
$$\mathbf{TT_{IND}} = MT_{IND} * T_{MOV} = 3.968mov * 0{,}01933s/mov = 76{,}701s / 60s = \mathbf{1{,}278m}$$

Ecuación 19. Algoritmos Laberínticos: Tiempos totales empleados desde todas las celdas (PC)

d) Los tiempos obtenidos por los **AL** en el caso del **PC** han sido obtenidos tras efectuar la **media** de **10 tiempos** elegidos **al azar** de entre todos los resultados.

e) Respecto de los tiempos obtenidos en el laberinto Real+Éstandar, conviene recordar que en el momento en el que se probaron los **AL** el robot **no disponía** de las **mejoras** que sí tenía al hacerlo con los de **RL**. Por ello, nos centraremos en los algoritmos informados y sólo en el tiempo que

emplean de **navegación** entre la entrada y la salida del laberinto, **no** en el de **mapeo**, para no establecer comparaciones que no sean realistas entre ambos tipos de algoritmos.

f) Aunque quizá podría **optimizarse** el funcionamiento de los algoritmos **informados** para **aprovechar** ciertos recorridos y ahorrar tiempo (especialmente en el caso del de Inundación), para no alterar las posibles conclusiones se va a considerar que en **todos** los casos estudiados el robot **parte de cero**, con un **desconocimiento** total del laberinto.

g) **No** se van a comparar las **iteraciones** efectuadas por los Algoritmos Laberínticos y los **episodios** de los RL, porque son conceptualmente **distintos** y también en cuanto a su propio diseño.

Así, la **comparación** se va a centrar en los siguientes **aspectos**:

a) **Tiempo** empleado en **resolver** el laberinto **virtual**: Servirá para comparar las diferencias existentes entre los **AL** y los de **RL** en el mismo laberinto virtual y en las mismas condiciones.

b) **Diferencia** de **tiempo** de resolución **Arduino vs PC**: Es una buena referencia para poder comparar mejor la **potencia** real de cada sistema en este tipo de situaciones.

c) **Tiempo** empleado únicamente en la **navegación física**: Servirá para comparar las diferencias entre el robot en su versión **inicial** y la versión **mejorada** (más sensores, *encoders*, etc).

Dicho esto, pasemos a ver los **resultados**:

Laberinto	**Virtual+Estándar**			**Real+Estándar**
Algoritmo	Arduino	PC	**Diferencia**	
	Tiempo de resolución (m|d DQN)		**(Ard vs PC)**	**Tiempo de navegación (s)**
Aleatorio	---	---	---	---
RMI	5,083	0,692	734,53%	---
RMI Optimizado	7,237	0,985	734,72%	11,32
Inundación	9,390	1,278	734,74%	12,93
Q-Aleatorio	---	---	---	---
SARSA	---	14,846	---	9,121
SARSA(λ)	---	22,414	---	9,236
Q-Learning	---	23,569	---	8,998
Q-Learning(λ)	---	14,246	---	9,31
DQN	---	9,606	---	9,156

Tabla 7. Algoritmos Laberínticos vs Aprendizaje por Refuerzo: Resultados

Con ellos en la mano, si los **analizamos** podemos hacer las siguientes **observaciones**:

a) **Tiempo de resolución: AL vs RL**

- El tiempo que emplean los **AL** en resolver el laberinto es, con mucha diferencia, **inferior** al que emplean los de **RL** (hasta un **3.405,92%** menor si RMI vs Q-Learning), en las mismas condiciones.

- El porcentaje de **diferencia** de **tiempo** empleado entre la Arduino y el PC es muy **estable**, siendo de un **734,5%**, y mostrando que los algoritmos en PC están **diseñados** de forma **equivalente**. Además, en este caso, hay que decir que la Arduino se comporta **realmente bien**, porque la diferencia de **capacidad de cómputo** real es mucho más **elevada** que este dato concreto.
- El algoritmo **Aleatorio** es más lento debido a su propia **aleatoriedad**, que lo vuelve totalmente impredecible. En cuanto al **DQN**, que se mide ya en días de tiempo, su "lentitud" se debe a requerir **mucha potencia** de proceso, tal y como hemos explicado.
- En cuanto a qué algoritmos son "mejores", puede decirse que los **AL** son más **eficientes** que los de **RL**. Evidentemente, están mucho más **enfocados** para este propósito **concreto**.
- Pero, por otra parte, no hay que olvidar que los algoritmos de **RL** son mucho más **adaptables**, siendo los **únicos** capaces de resolver el laberinto **Virtual+Desafiante**, algo **imposible** para los **AL**.

b) **Tiempo de navegación: Robot Inicial vs Mejorado**

- El robot **mejorado** ha empleado **menos** tiempo que su versión **inicial** (alrededor de un **25%** menos).
- Los tiempos empleados son bastante **estables** en la versión **mejorada**, lo que demuestra que el robot ya posee más "**control**" de su entorno, y no es tan propenso a sufrir desviaciones de rumbo y a tener que reorientarse debido a aspectos como el rozamiento de las ruedas, la potencia de los motores o las superficies por las que pasa.

Por todo ello, podemos **concluir** que:

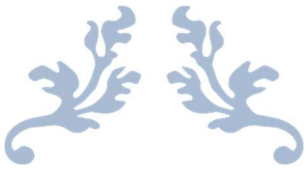
1) La **aleatoriedad** en la resolución y navegación de laberintos debe **evitarse** a toda costa.

2) Los **Algoritmos Laberínticos** son **mucho** más **eficientes** en las **condiciones adecuadas**.

3) Los algoritmos de **Aprendizaje por Refuerzo** son más **adaptables** a **todo tipo** de laberintos.

4) Los algoritmos de **RL** requieren de mucha más **potencia de proceso** respecto de los **AL**.

5) Las **mejoras de navegación** introducidas en el robot están plenamente **justificadas**.



9. CONCLUSIONES Y TRABAJO FUTURO



9. Conclusiones y Trabajo Futuro

9.1. Conclusiones

La **robótica** es ya casi omnipresente en la actualidad, gracias al **impulso** de las grandes compañías tecnológicas, que buscan tanto reducir **costes** y maximizar **beneficios** como proporcionarnos una mejor **calidad** de vida, automatizada hasta donde hace nada era totalmente impensable. Y este hecho se hace especialmente patente cuando hablamos de **robots autónomos móviles**, desde los sencillos, pero a la par complejos aspiradores domésticos hasta llegar a verdaderas obras de la ingeniería moderna como los ya no tan futuros coches autónomos de niveles 4 y 5.

En este contexto es donde se sitúa un campo de gran relevancia en la robótica **actual**: la resolución de **laberintos**, entornos controlables y gracias a los que es posible **probar** todo tipo de soluciones, para **entrenar y mejorar** la capacidad que poseen los robots autónomos de **desplazarse** por su entorno de la forma más **eficiente** posible, algo de sumo interés. Y en todo ello, y desde hace ya tiempo, tienen gran relevancia los ***micromouses***, unos pequeños robots diseñados para encontrar la salida de un laberinto **sin ayuda** y creados para poner **a prueba** la inventiva y creatividad de ingenieros de todo el mundo.

Por otra parte, el **Aprendizaje por Refuerzo** resulta ser muy interesante para resolver tareas complejas, como la **navegación**, permitiendo un comportamiento mucho más **complejo** del que se le presupone. El problema es que el robot suele necesitar mucha **experiencia**, lo que repercute en **tiempo** de computación y altos **costes** operativos, como sucede en ocasiones cuando se emplean **redes neuronales** antes del RL o, lo que es lo mismo, emplear **Aprendizaje por Refuerzo Profundo** o **DRL**.

A lo largo de los diferentes capítulos que componen este trabajo hemos podido abordar la creación de un ***micromouse***, teniendo en cuenta todo lo comentado, trabajando sobre su **diseño**, la elección de sus **componentes** y, por supuesto, la implementación de las diferentes **estrategias y algoritmos** necesarios para gobernarlo mientras se desplaza. Se han presentado algunas de las más habituales en cuanto a **navegación** por laberintos, y se han **comparado** con el RL y DRL, para conocer de qué es capaz un robot ***open source*** de este tipo dotado de una inteligencia más **avanzada** y **adaptable**.

Tras ello, se han presentado los **resultados** obtenidos, tanto de forma teórica como práctica, sirviendo como comparación **real** entre las técnicas clásicas de resolución de laberintos y el RL.

Sobre todo se ha hablado y experimentado de primera mano a lo largo de este Trabajo Fin de Máster, en cada uno de sus capítulos. Ha sido un trabajo **extenso** y, en más ocasiones de las previstas, de una **exigencia**

técnica destacable. Ello ha motivado algunas **variaciones** en el calendario de tareas e hitos, con ciertas modificaciones en extensión y detalle. Pero debido a que ya en la planificación inicial se tuvo en cuenta esta posibilidad, el trabajo ha resultado ser lo suficientemente flexible y adaptable para dar cabida a todas las correcciones y necesidades alternativas que han ido surgiendo, dando como resultado un **mejor** trabajo, en contenido y forma, hasta llegar a convertirse en el TFM que tenemos entre manos.

9.2. Trabajo Futuro

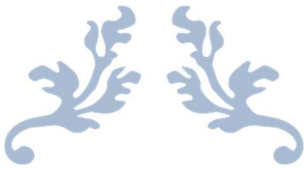
En este proceso de desarrollo del trabajo, y a lo largo de todos sus apartados, se han podido vislumbrar muchas de las **carencias** y **virtudes** que están presentes en este tipo de desarrollos. Como resultado, resultaría muy conveniente trabajar en un momento futuro sobre las siguientes **áreas**:

- **Investigación en el campo de los algoritmos de RL**: Quizá sea posible optimizar su funcionamiento para hacer que sean más fácilmente manejables por plataformas que no sean tan potentes como las que finalmente resultan necesarias. Aunque, por otra parte, las GPUs están mejorando su relación coste/potencia a un ritmo increíble en los últimos tiempos, por lo que podría pensarse en desarrollos optimizados para GPUs de bajo coste, directamente.
- **Investigación en mejoras básicas para ciertos componentes**: La verdad es que resulta bastante frustrante el hecho de que, por ahorrar un poco de coste del producto final, algunos de los componentes que existen en la actualidad para integrar en un robot móvil acaben siendo casi más perjudiciales que beneficiosos. Sensores IR quemados en el primer encendido, ejes para las ruedas con acoplamientos defectuosos que hacen que se salgan en dos giros, cables que se sueltan de sus terminaciones según se cogen, motores con rangos de potencia que oscilan entre no girar apenas o parecer una turbina, todo ello comprobado de primera mano.
- **Colaboración en el desarrollo de mejores IDEs para Arduino**: Si bien el Arduino IDE resulta suficiente, no lo es tanto cuando de lo que se trata es de aprovechar al máximo sus recursos. La posibilidad de emular dicha plataforma y de conocer su comportamiento sin necesidad de hacer pruebas físicas aceleraría los desarrollos ostensiblemente. Es algo que, por ejemplo, lleva años haciendo Google con Android, y su kit de desarrollo para aplicaciones móviles es realmente agradecido, funcional e intuitivo.
- **Colaboración en el desarrollo de *software* de ML para plataformas de bajo coste**: Las soluciones que existen en la actualidad no dejan de ser versiones demasiado simplificadas respecto de las estándar, y bastantes de sus funcionalidades se echan en falta antes o después, por lo que al final siempre se acaba desarrollando pensando casi exclusivamente en un PC.
- **Colaboración en el desarrollo de guías informativas avanzadas de Arduino**: Si bien es cierto que la comunidad de Arduino es muy activa, y existen multitud de foros y páginas web, no es tan

sencillo encontrar información sobre algo que se salga de los ejemplos habituales o que trate de ir más allá. Y, como pasa en ocasiones, el hecho de no encontrar apenas información sobre una librería a tiempo puede hacer que acabemos tratando de reinventar la rueda.

- **Aplicación de todo el conocimiento adquirido con este TFM a drones**: Ha sido un gran trabajo, y la verdad es que, al profundizar como lo he llegado a hacer, me han sorprendido muchos aspectos de Arduino para bien. Tiene muchas posibilidades que aún tengo pendientes de explotar, así que, ¿por qué no pasar de las 2D a las 3D en la robótica móvil?

Todo ello podría servir para mejorar las **capacidades** de las plataformas móviles, su **navegación** y, sobre todo, su **facilidad de uso**, algo siempre necesario y deseable. A buen seguro, el futuro nos deparará aún más **tipologías** de ellas, seguramente más potentes, eficientes y esperemos que suficientemente asequibles. Y la idea de aplicar los **conocimientos** que he ido adquiriendo sobre Arduino, la gran variedad de sensores que existen y las herramientas *open source* para el Aprendizaje por Refuerzo a un **dron** de bajo coste me resulta cada vez más interesante… ;)



REFERENCIAS BIBLIOGRÁFICAS



Referencias Bibliográficas

Amazon. (2020). *Amazon*. Obtenido de https://www.amazon.es

Arduino CC. (Septiembre de 2020). *Arduino CC.* Obtenido de https://www.arduino.cc/

Arduino Stack Exchange. (Diciembre de 2020). *Arduino Stack Exchange*. Obtenido de https://arduino.stackexchange.com/

Arduino Team. (2020). *Get started with machine learning on Arduino.* Obtenido de Arduino.cc: https://blog.arduino.cc/2019/10/15/get-started-with-machine-learning-on-arduino/

Arduino.cc. (2020). *Arduino Nano 33 BLE Sense.* Obtenido de Arduino.cc: https://store.arduino.cc/arduino-nano-33-ble-sense?utm_source=blog&utm_medium=blog&utm_campaign=ai&utm_content=getting%20started

Arulkumaran, K., Deisenroth, M., Brundage, M., & Bharath, A. (2017). Deep Reinforcement Learning: A Brief Survey. *IEEE Signal Processing Magazine* , 26-38.

Arya, I., Gillespie, F., Kapoor, S., Mahesh, K., Ruzinov, D., & Battipaglia, J. (21 de Julio de 2017). *Autonomously Solving Mazes with Robots.* Obtenido de Semantic Scholar: https://pdfs.semanticscholar.org/0306/d72b917053771ef00e6db84d6309e8cf0f75.pdf

Automouse. (2016). *Automouse.* Obtenido de San Diego State University: http://automouse.sdsu.edu/design.html

Berridge, K. C. (2001). Reward Learning: Reindorcement, Incentives, and Expectations. En *The Psychology of Learning and Motivation, Vol. 40* (págs. 223-278). Academic Press.

Bricolabs Wiki. (2015). *Control de motores con Arduino.* Obtenido de Bricolabs Wiki: https://bricolabs.cc/wiki/guias/control_de_motores

Cheung, P. (2016). *Servo motor SG90.* Obtenido de Imperial College of London: http://www.ee.ic.ac.uk/pcheung/teaching/DE1_EE/stores/sg90_datasheet.pdf

Choudhary, A. (18 de Abril de 2019). *A Hands-On Introduction to Deep Q-Learning using OpenAI Gym in Python.* Obtenido de Analytics Vidhya: https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/

Christiansen, D. (23 de Abril de 2014). *The Amazing MicroMouse Contest.* Obtenido de IEEE Spectrum: https://spectrum.ieee.org/consumer-electronics/gadgets/the-amazing-micromouse-contest

Covaci, R., Harja, G., & Nascu, I. (2020). Autonomous Maze Solving Robot. *2020 IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR)*.

Diosdado, R. (2018). *Sensor de ultrasonidos HC-SR04.* Obtenido de Zona Maker: https://www.zonamaker.com/arduino/modulos-sensores-y-shields/ultrasonido-hc-sr04

Elegoo. (Septiembre de 2019). *Elegoo*. Obtenido de https://www.elegoo.com/

Elshamarka, I., & Sayuti Saman, A. B. (Octubre de 2012). *Design and Implementation of a Robot for Maze-Solving using Flood-Fill Algorithm.* Obtenido de CiteseerX: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.245.1506&rep=rep1&type=pdf

Feiticeir0. (2018). *IR Line tracking sensor.* Obtenido de Feiticeir0's Blog: http://blog.whatgeek.com.pt/arduino/ir-tracking-sensor/

García Álvarez, F., & Santos Peñas, M. (2020). *Educational-Oriented Mobile Robot: Hidden Lessons.* Burgos: Springer.

GitHub. (Diciembre de 2020). *GitHub*. Obtenido de https://github.com/

Google. (2020). *Tensorflow*. Obtenido de https://www.tensorflow.org/

Google Brain. (14 de Enero de 2019). *What's coming in TensorFlow 2.0.* Obtenido de Tensorflow Blog: https://blog.tensorflow.org/2019/01/whats-coming-in-tensorflow-2-0.html

Google DeepMind. (2021). *DeepMind*. Obtenido de Github: https://github.com/deepmind

Grzes, M., & Kudenko, D. (Septiembre de 2008). *Robustness Analysis of SARSA(λ): DifferentModels of Reward and Initialisation*. Obtenido de ResearchGate.net: https://www.researchgate.net/publication/221656193_Robustness_Analysis_of_SARSAlambda_Different_Models_of_Reward_and_Initialisation

Guzmán, G. (8 de Febrero de 2019). *Programación dinámica con Q-learning.* Obtenido de Planeta ChatBot: https://planetachatbot.com/programaci%C3%B3n-din%C3%A1mica-con-q-learning-9e1474312287

Htike Htike, A., & Min Oo, Z. (2018). Path Tracking Control of Obstacle Avoiding Robot Using Ultrasonic Sensor. *International Journal of Science and Engineering Applications*, 499-503.

Infootec. (2017). *Sensor ultrasonidos HC-SR04.* Obtenido de Infootec: https://www.infootec.net/sensor-ultrasonidos-hc-sr04/

Instituto de Ingeniería del Conocimiento. (2019). *Aprendizaje por Refuerzo*. Obtenido de Instituto de Ingeniería del Conocimiento: https://www.iic.uam.es

Jing, X., Huan, Z., Ding, L., & Minqi, L. (2017). Application of deep reinforcement learning in mobile robot path planning. *2017 Chinese Automation Congress (CAC)*.

Juing-Huei, S., Chyi-Shyong, L., & Hsin-Hsiung, H. (2011). A Micromouse Kit for Teaching Autonomous Mobile Robots. *The International Journal of Electrical Engineering & Education*, 188-201.

Keras Team. (2020). *Keras*. Obtenido de https://keras.io/

Llamas, L. (2020). *Tutoriales de Arduino.* Obtenido de Luis Llamas: https://www.luisllamas.es/tutoriales-de-arduino/

Luna González, J. (8 de Febrero de 2018). *Tipos de aprendizaje automático*. Obtenido de Medium: https://medium.com/soldai/tipos-de-aprendizaje-autom%C3%A1tico-6413e3c615e2

m0rd0r. (2019). *m0rd0r's bl0g.* Obtenido de http://www.m0rd0r.eu/

Martínez Heras, J. (20 de Septiembre de 2020). *Gradiente Descendiente para aprendizaje automático.* Obtenido de IArtificial.net: https://www.iartificial.net/gradiente-descendiente-para-aprendizaje-automatico/#:~:text=funci%C3%B3n%20de%20coste.-,M%C3%A9todo%20del%20Gradiente%20Descendiente,%2C%20el%20deep%20learning%2C%20etc.

MatPlotLib. (2020). *MatPlotLib.* Obtenido de https://matplotlib.org/

Minhaj, A. (31 de Mayo de 2018). *How to Use ServoTimer2 Library (Simple Explain) Servo Sweep.* Obtenido de Hackster.io: https://www.hackster.io/ashraf_minhaj/how-to-use-servotimer2-library-simple-explain-servo-sweep-512fd9

Mishra, S., & Bande, P. (2008). Maze Solving Algorithms for Micro Mouse. *2008 IEEE International Conference on Signal Image Technology and Internet Based Systems*.

Mnih, V., Silver, D., Veness, J., Fidjeland, A., & Petersen, S. (2015). Human-level control through deep reinforcement learning. *Nature*, 529–533.

Nay Lamp Mechatronics. (2017). *Driver Puente H L298N.* Obtenido de Nay Lamp Mechatronics: https://naylampmechatronics.com/drivers/11-driver-puente-h-l298n.html

NumPy. (2020). *NumPy.* Obtenido de https://numpy.org/

Oracle. (2020). *VirtualBox End-user documentation.* Obtenido de VirtualBox : https://www.virtualbox.org/wiki/End-user_documentation

Osmanković, D., & Konjicija, S. (2011). Implementation of Q — Learning algorithm for solving maze problem. *2011 Proceedings of the 34th International Convention MIPRO*.

Ottoson, J., & Renström, N. (2019). *School of Industrial Engineering and Management.* Obtenido de KTH Royal Institute of Technology: http://www.diva-portal.org/smash/get/diva2:1373793/FULLTEXT01.pdf

Packtpub. (2018). *SARSA versus Q-learning – on-policy or off?* Obtenido de Packtpub: https://subscription.packtpub.com/book/data/9781789345803/1/ch01lvl1sec13/sarsa-versus-q-learning-on-policy-or-off

Pattabiraman, K. (Octubre de 2017). *How to set up an IR remote and receiver on an Arduino.* Obtenido de Circuit Basics: http://www.circuitbasics.com/arduino-ir-remote-receiver-tutorial/

Pfeiffer, M., Schaeuble, M., Nieto, J., Siegwart, R., & Cadena, C. (2017). From perception to decision: A data-driven approach to end-to-end motion planning for autonomous ground robots. *2017 IEEE International Conference on Robotics and Automation (ICRA)*.

Prometec. (2016). *Arduino y L298N: Montaje y conexión.* Obtenido de Prometec: https://www.prometec.net/coche-arduino-l298n/

Prometec. (2018). *El Bus I2C*. Obtenido de Prometec: https://www.prometec.net/bus-i2c/

Python.org. (2020). *Python.org*. Obtenido de https://www.python.org/

Rahman, M. (Mayo de 2017). *Autonomous Maze Solving Robot.* Obtenido de University of Liberal Arts Bangladesh (ULAB): https://www.researchgate.net/profile/Musfiqur_Rahman3/publication/316664613_AUTONOMOUS_MAZE_SOLVING_ROBOT/links/590acddcaca272f6580b7b7f/AUTONOMOUS-MAZE-SOLVING-ROBOT.pdf

Sadik, A., Dhali, M., Farid, H., Rashid, T., & Syeed, A. (2010). A Comprehensive and Comparative Study of Maze-Solving Techniques by Implementing Graph Theory. *2010 International Conference on Artificial Intelligence and Computational Intelligence*.

Siqueira, A. S., & Rodrigues, I. B. (2017). *The project of an improved omnidirectional modular robot for the RoboCupJunior Rescue Maze 2017.* Obtenido de Sistema Olimpo: http://sistemaolimpo.org/midias/uploads/74756631e3b102f0591f152e859eda6f.pdf

Srivannaboon, S. (2009). Achieving Competitive Advantage through the use of Project Management under the Plan-do-Check-act Concept. *Journal of General Management*, 1-20.

Stack Overflow. (Enero de 2018). *Plotting multiple lines in python.* Obtenido de Stack Overflow: https://stackoverflow.com/questions/47449741/plotting-multiple-lines-in-python

Stack Overflow. (Diciembre de 2020). *Stack Overflow*. Obtenido de https://stackoverflow.com/

Sutton, R. S., & Barto, A. G. (2015). *Reinforcement Learning: An Introduction.* Obtenido de Stanford University: https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf

Technoblogy. (28 de Marzo de 2018). *Tiny Graphics Library for SH1106 OLED Display*. Obtenido de Technoblogy: http://www.technoblogy.com/show?23OS

TechVidvan. (2020). *Types of Machine Learning – Supervised, Unsupervised, Reinforcement*. Obtenido de TechVidvan: https://techvidvan.com/tutorials/types-of-machine-learning/

TensorFlow . (2020). *TensorFlow Lite para microcontroladores.* Obtenido de TensorFlow : https://www.tensorflow.org/lite/microcontrollers

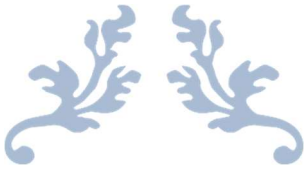
Torres, J. (15 de Mayo de 2020). *A gentle introduction to Deep Reinforcement Learning*. Obtenido de Towards Data Science: https://towardsdatascience.com/drl-01-a-gentle-introduction-to-deep-reinforcement-learning-405b79866bf4

Viji, R. (2018). *Design and Development of Intelligent Navigation Control Systems for Autonomous Robots Using Neural Network.* Obtenido de Semantic Scholar: https://www.semanticscholar.org/paper/Design-and-Development-of-Intelligent-Navigation-Viji/75ddaad5b04e3789cea1558b8ebf10c245029d66#paper-header

Wyard-Scott, L., & M. Meng, Q. (1995). A potential maze solving algorithm for a micromouse robot. *IEEE Pacific Rim Conference on Communications, Computers, and Signal Processing. Proceedings*.

Yee Mon, N., & Nu Nu, W. (2016). Path Finding and Turning with Maze Solving Robot. *International Journal of Science, Engineering and Technology Research (IJSETR)*.

Yousef Moh., A., & Ali, H. (2018). Wheeled Mobile Robot Obstacle Avoidance Using Compass and Ultrasonic. *Universal Journal of Control and Automation*, 13-18.



GLOSARIO DE TÉRMINOS



Glosario de Términos

2D: *2-Dimensional*

AL: Algoritmos Laberínticos

ANN: *Artificial Neural Networks*

BFS: *Breadth First Search*

CPU: *Central Processing Unit*

DC: *Direct Current*

DFS: *Depth First Search*

DL: *Deep Learning*

DQN: *Deep Q-Networks*

DRL: *Deep Reinforcement Learning*

EEPROM: *Electrically Erasable Programmable Read-Only Memory*

E/S: Entradas/Salidas

GND: *Ground*

I2C: *Inter-Integrated Circuit*

ICSP: *In-Chip Serial Programmer*

IDE: *Integrated Development Environment*

IR: *Infra-Red*

KB: *KiloBytes*

LED: *Light-Emitting Diode*

LSRB: *Left-Straight-Right-Back*

MDF: *Medium Density Fibreboard*

MHz: *MegaHertzios*

ML: *Machine Learning*

OLED: *Organic Light-Emitting Diode*

PID: *Proportional–Integral–Derivative*

PWM: *Pulse-Width Modulation*

RL: *Reinforcement Learning*

RMD: Regla de la Mano Derecha

RMI: Regla de la Mano Izquierda

RSLB: *Right-Straight-Left-Back*

SPI: *Serial Peripheral Interface*

SRAM: *Static Random Access Memory*

TDD: *Temporal Difference Learning*

UART: *Universal Asynchronous Receiver-Transmitter*

ANEXO A:

MONTAJE DEL ELEGOO SMART ROBOT CAR V3

Anexo A: Montaje del Elegoo Smart Robot Car v3

Toda la **teoría** que hemos visto a lo largo de este trabajo ha sido puesta en **práctica**, para poder medir, evaluar y comparar los **resultados** obtenidos. Aparte de ello, han sido muchos, muchos **componentes** los que al final al acabo formando parte de él. Algunos buenos, otros mejores y otros… mejor los dejaremos en el olvido. Pero de lo que no me cabe duda es de que una **muy buena idea** es **no** partir totalmente de cero cuando no se tienen medios y/o experiencia en cuanto al montaje de un robot móvil basado tanto en **Arduino** como en cualquier tipo de plataforma similar a ella.

Por ello, este anexo va a estar dedicado íntegramente a detallar el **montaje** del **kit** de robot móvil que he elegido como base para todo el trabajo desarrollado: El **Elegoo Smart Robot Car v3**. Es un kit que tiene un coste realmente **asequible** para todo lo que lleva (alrededor de 60-70€), está muy bien **terminado** y está listo para ser **montado**. Además, todos los componentes y piezas están **etiquetados** y **embalados** según su ubicación y uso final (ver *Ilustración 81*), así que más no se puede pedir.

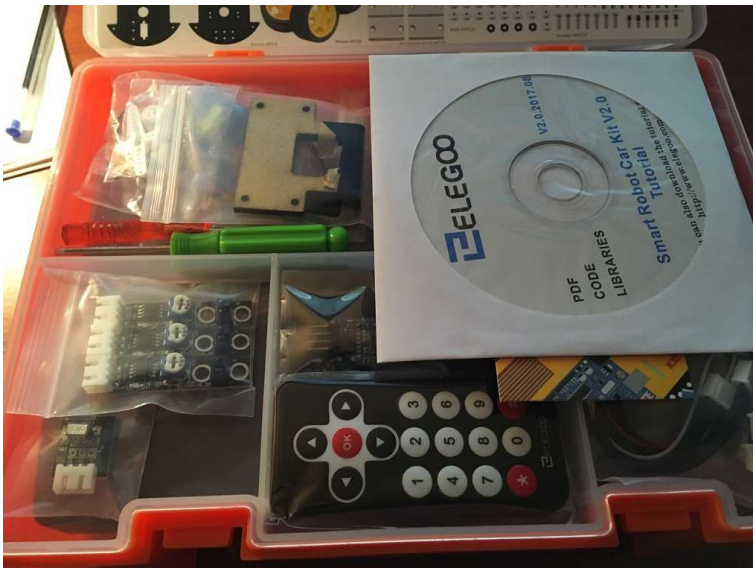


Ilustración 81. Elegoo Smart Robot Car v3: Presentación del kit

Y si bien se incluye también una **guía** de montaje en el CD que viene con el propio kit, a veces es algo **confusa** por la secuencia que se sigue, por la perspectiva de las fotos o porque los desarrolladores de Elegoo han utilizado más de la cuenta el traductor automático de Google…

Dicho esto, trataré de ser lo más **claro** y **conciso** posible, así que… **¡vamos a ello!**

1) Instalación de los motores

No reviste demasiada dificultad, tan sólo identificar los **tornillos** correspondientes, localizar los **pilares** de sujeción de los motores en la plancha de MDF y los tendremos a punto (ver *Ilustración 82*).

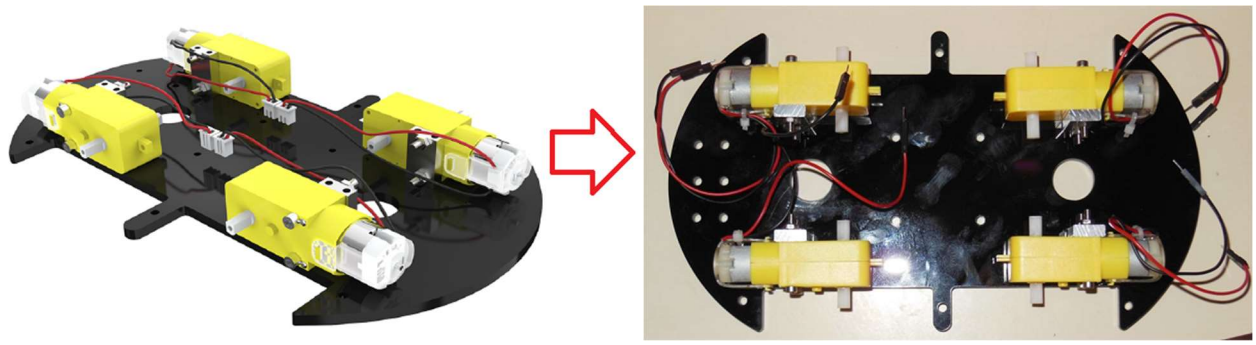


Ilustración 82. Elegoo Smart Robot Car v3: Instalación de los motores

2) Instalación del módulo L298N

El paso lógico tras montar los motores es proseguir con la placa de que **alimentará** los motores de nuestro robot (ver *Ilustración 83*), gracias a las **baterías** que vienen en el kit.

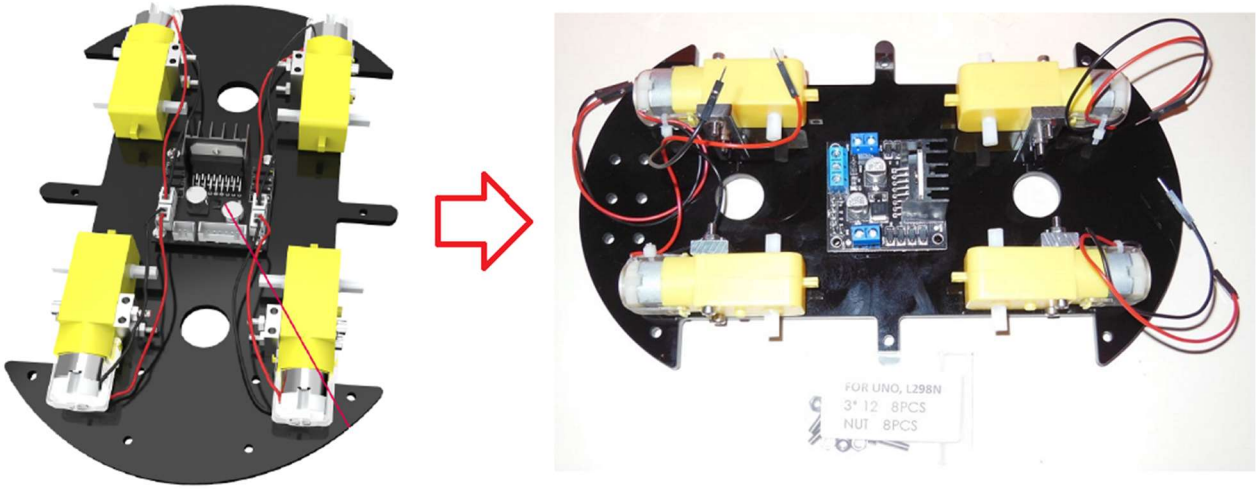


Ilustración 83. Elegoo Smart Robot Car v3: Instalación del módulo L298N

3) Instalación de los sensores de líneas

Hay que tener en cuenta que son **tres**, aunque montados de forma **idéntica** (ver *Ilustración 84*). La única dificultad radica en tener en cuenta hacia dónde deben **mirar** y que los **cables** que los alimentan deben pasar a través del agujero que ya está hecho previamente en la plancha de MDF.



Ilustración 84. Elegoo Smart Robot Car v3: Instalación de los sensores de líneas

4) Instalación de la Arduino y la placa de extensión

La clave de todo el proceso es **no apretar** demasiado los tornillos de sujeción de la Arduino ya que, de hacerlo, podríamos **dañar** la placa. O, si entra en **contacto** la cabeza de una tuerca con la propia placa, podría suceder lo que me pasó en su momento con la Mega, **imposibilitar** la carga de programas en memoria, así que es posible que necesitemos **espaciadores** de nylon o similares (ver *Ilustración 85*). Por suerte, el kit que tenemos entre manos ya cuenta con ellos, así que no debería haber problemas.

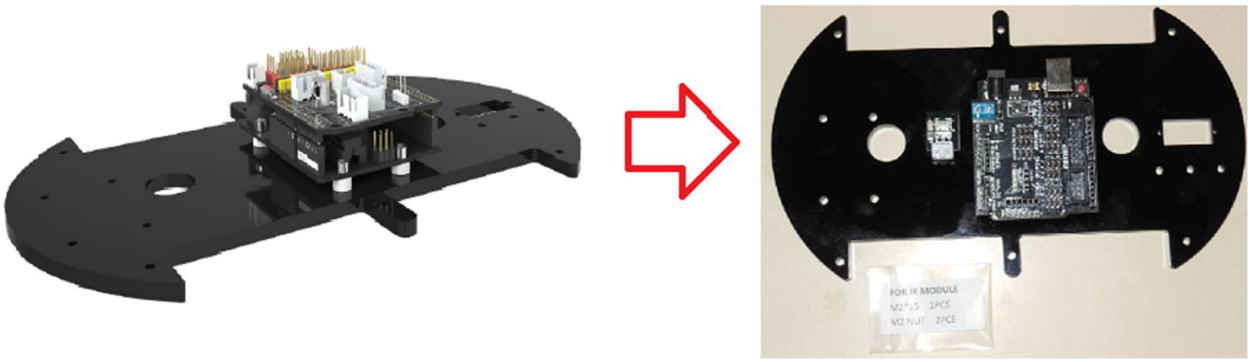


Ilustración 85. Elegoo Smart Robot Car v3: Instalación de la Arduino y de la placa de extensión

5) Instalación del receptor IR y del compartimento de baterías

Se anclarán a la plancha de MDF **superior**, justo al lado de la Arduino (ver *Ilustración 86*). Es así porque de las baterías va a depender toda la **alimentación** de todos los componentes, a través de la propia Arduino. Por ello, está situada en una ubicación estratégica, al alcance de todo lo necesario. Aparte de ello, el receptor IR también necesita estar posicionado de tal forma que el mando IR puede enviarle **señales**. De nada serviría que estuviera en la zona inferior, porque no podría recibir comando alguno.

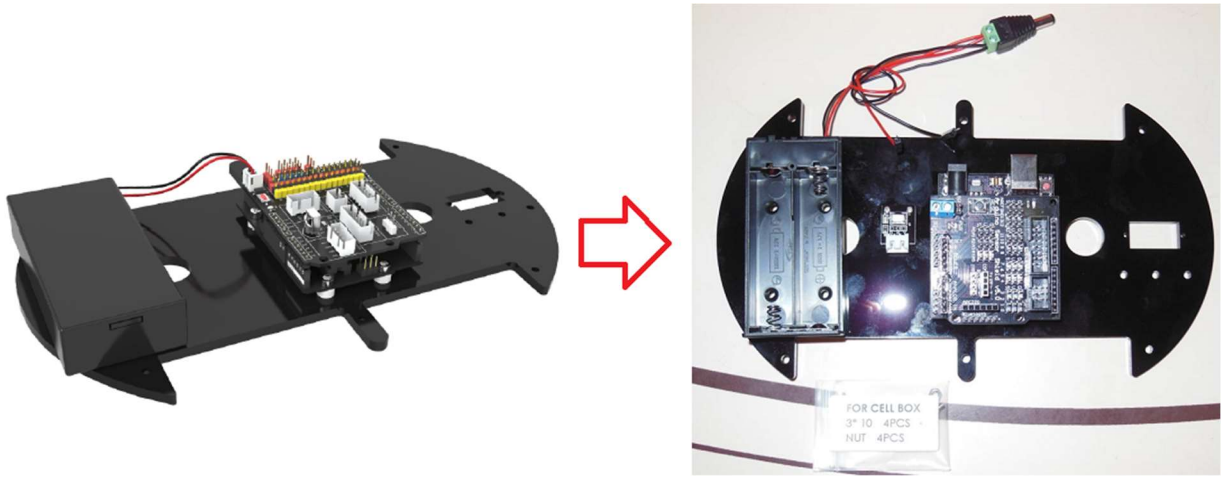


Ilustración 86. Elegoo Smart Robot Car v3: Instalación del receptor IR y del compartimento de baterías

6) Instalación del servo y del receptor de ultrasonidos

Son un conjunto **solidario**, porque el sensor de ultrasonidos va a necesitar del servo para que el robot pueda orientarse en un arco de casi **180 grados** a su alrededor. Gracias a él, podrá moverse a derecha e izquierda para comprobar los obstáculos a los que se enfrente. Su ubicación (ver *Ilustración 87*) se debe a que el robot se va a mover hacia adelante, y necesitará anticiparse a lo que se encuentra a su paso.

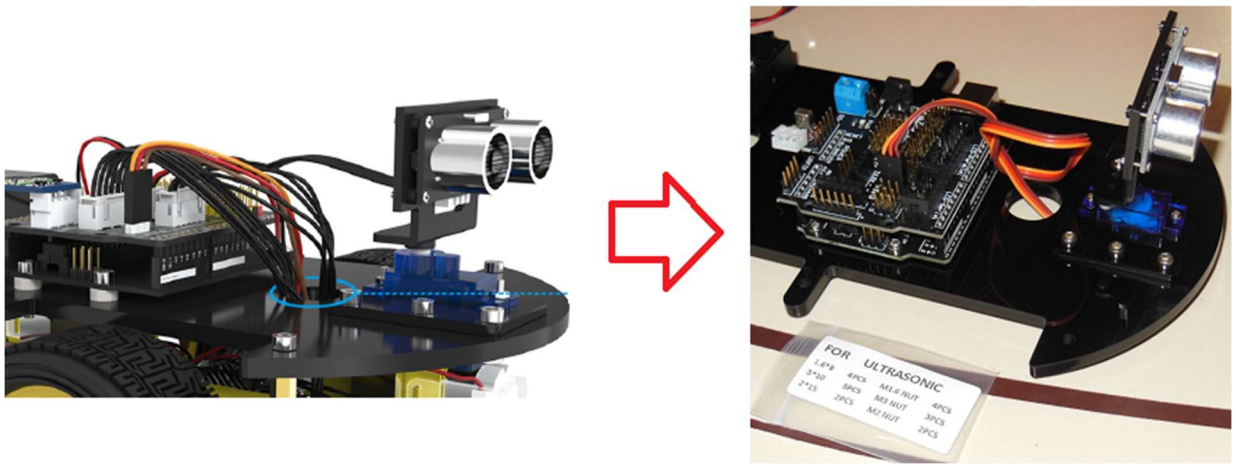


Ilustración 87. Elegoo Smart Robot Car v3: Instalación del servo y del receptor de ultrasonidos

7) Conexión del cableado de los motores y preparación de las dos planchas MDF

Es una parte sencilla si se sigue el **orden** establecido por el fabricante y aplicamos algo de lógica y sentido común (ver *Ilustración 88*). El punto a tener en cuenta en el proceso es **no confundirnos** con las conexiones porque, si lo hacemos, al enviar comandos desde la Arduino para mover el robot en una dirección u otra puede que lo hagamos de forma incorrecta, lo que provocaría un auténtico caos. Si sucede, tocará desmontar de nuevo las dos mitades y volver a comprobar este apartado…

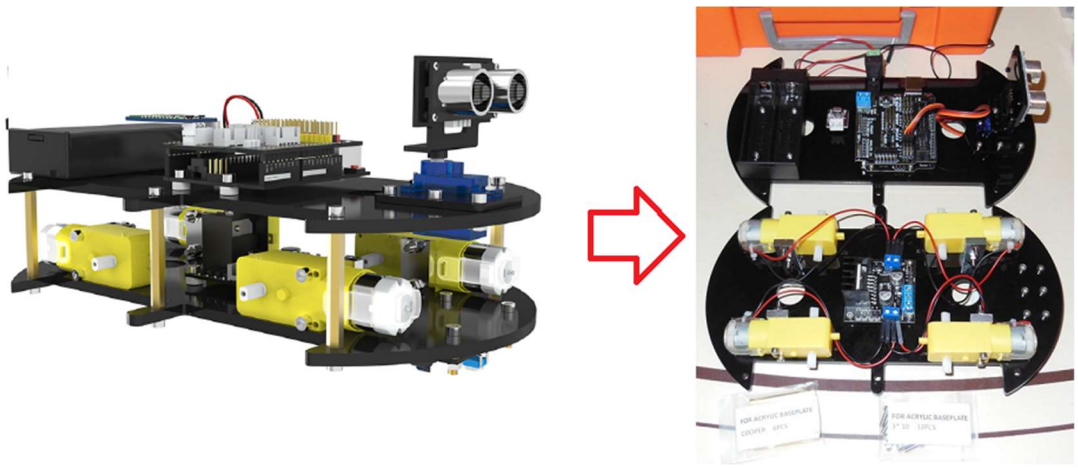


Ilustración 88. Elegoo Smart Robot Car v3: Conexión del cableado de los motores y preparación de las dos planchas MDF

8) Conexión de los sensores a la Arduino

Con cuidado, hay que pasar ahora a conectar los **sensores** de los que el robot dispone **a la placa** Arduino. Es decir, los tres sensores de líneas y el sensor de ultrasonidos, aprovechando también para conectar el receptor IR situado en la zona superior de montaje (ver *Ilustración 89*).

En este punto es bastante importante consultar el manual de **conexionado** del fabricante (o la tabla del adjunto **Anexo B**), porque una equivocación en un número de pin hará que el componente conectado no funcione correctamente, teniendo que revisar el cableado uno a uno.

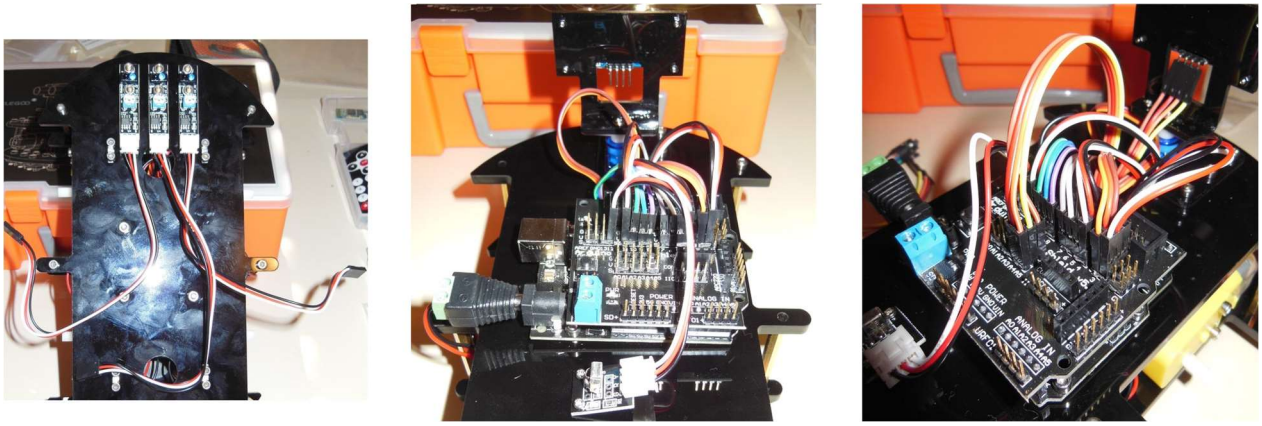


Ilustración 89. Elegoo Smart Robot Car v3: Conexión de los sensores a la Arduino

9) Conexión del receptor Bluetooth

Este paso es, bajo mi punto de vista, totalmente **opcional**. Si bien es cierto que en el kit se incluye este receptor (ver *Ilustración 90*), la verdad es que sirve para bastante **poco** según viene de fábrica. El módulo en sí no está mal, es capaz de recibir comandos básicos a través del puerto serie (requiriendo de cierta programación adicional), pero lo que más limita su funcionalidad es la aplicación de la que dispone para que el robot sea manejado desde un móvil: es muy simple, la interfaz suele fallar y sirve para unos pocos comandos preestablecidos.

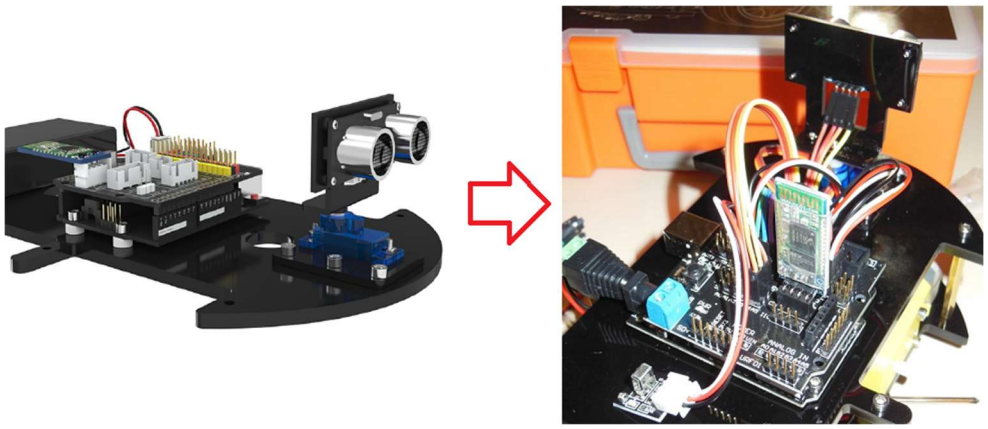


Ilustración 90. Elegoo Smart Robot Car v3: Conexión del receptor Bluetooth

10) Ensamblaje de las ruedas

Es el último paso, pero resulta uno de los más **importantes**. De nada sirve una buena lógica de control, unos sensores muy bien calibrados o potencia de sobra en los motores si todo esto no se traduce en un buen movimiento del robot. Así que conviene tomarnos nuestro tiempo, revisar los ejes, abrir los motores si fuera necesario para ello y dejar todo bien **ajustado** (ver *Ilustración 91*). Por mi experiencia, y tal y como he reflejado en diferentes apartados del trabajo, un buen funcionamiento de las ruedas no es nada fácil de conseguir, y puede dar muchos dolores de cabeza…

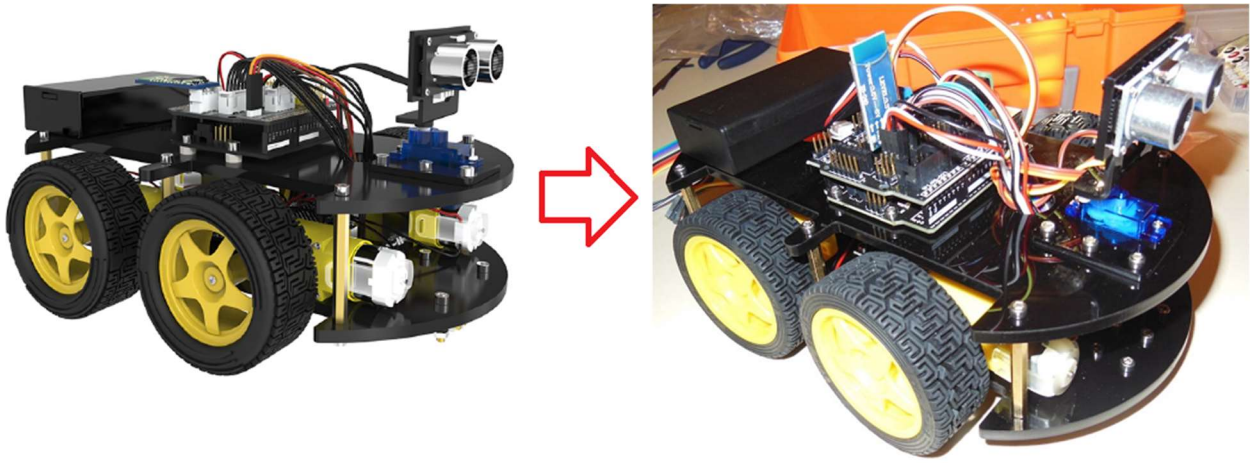


Ilustración 91. Elegoo Smart Robot Car v3: Ensamblaje de las ruedas

Y el resultado final es…

Nos va a llevar un **buen rato**, seguro, porque una cosa es montarlo y otra es montarlo bien, tratando de evitar errores futuros, ajustando todos los componentes, pasando los cables por los mejores sitios posibles (ver *Ilustración 92*). Así que, si has llegado hasta aquí… ¡**Enhorabuena**!

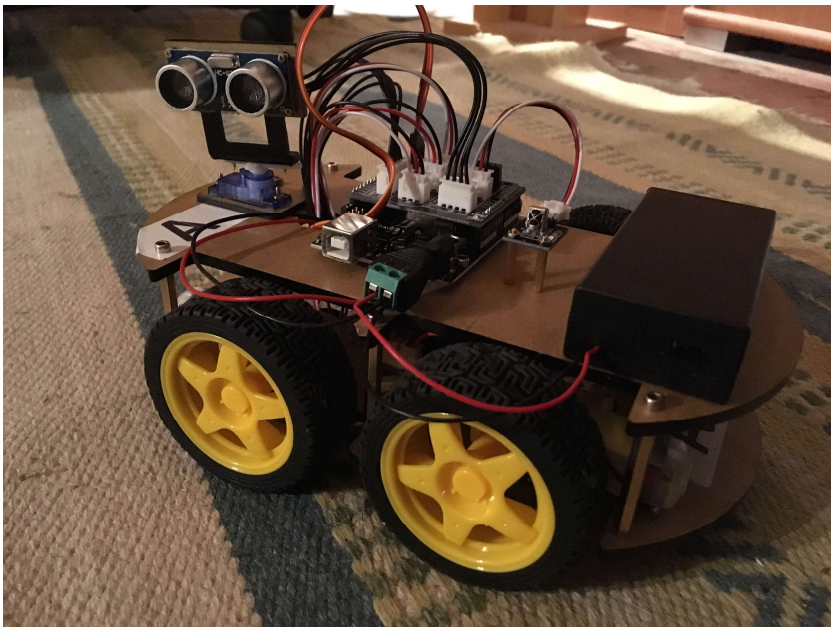
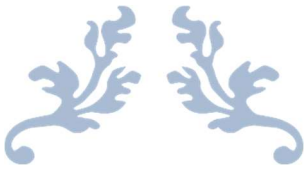


Ilustración 92. Elegoo Smart Robot Car v3: Montaje final



ANEXO B:

TABLA DE CONEXIONES PARA EL ELEGOO SMART ROBOT CAR V3



Anexo B: Tabla de Conexiones para el Elegoo Smart Robot Car v3

Tal y como he comentado en apartados previos, conectar correctamente todos los **componentes** del robot que vamos a utilizar como base es **fundamental**. Si no lo hacemos bien, aparecerán fallos en las medidas de los sensores, comportamientos erráticos, problemas de alimentación… Y cuando hablamos de un kit de estas características, en el que la **cantidad** de conexiones y de pines que hay que tener en cuenta es muy **elevada**, es **muy fácil** cometer errores.

Por eso, he decidido elaborar una **tabla de conexiones** de pines lo más clara posible, para que no haya dudas a la hora de saber qué va conectado donde, porque según cuál sea el manual concreto del kit de Elegoo que compremos se verá mejor, peor o, directamente, no estará.

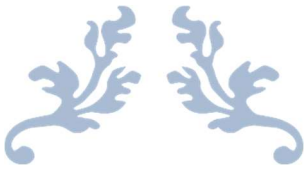
Así pues, vamos a **ello**:

Tabla de Conexiones – Elegoo Smart Robot Car v3			
Arduino Uno	**Placa de E/S**		**Componente**
D10	S10		**Sensor de líneas 1**
D4	S4		**Sensor de líneas 2**
D2	S2		**Sensor de líneas 3**
	GND		**Servo SG90**
	VCC		
D3	S3		
D5	S5	ENA	**Placa L298N**
D6	S6	ENB	
D7	S7	N1	
D8	S8	N2	
D9	S9	N3	
D11	S11	N4	
	GND		**Receptor IR**
	VCC		
D12	S12		
	VCC	Vcc	**Sensor de ultrasonidos**
A5	A5	Trig	
A4	A4	Echo	
	GND	Gnd	
	+	VCC	**Módulo Bluetooth**
	-	GND	
D0 (RX)	RX	TXD	
D1 (TX)	TX	RXD	
Placa L298N	Motor		**Pin de control**
OUT1	Cable negro		**N3, N4**
	Cable rojo		

Tabla de Conexiones – Elegoo Smart Robot Car v3			
Arduino Uno	**Placa de E/S**		**Componente**
OUT2	Cable negro		
	Cable rojo		
OUT3	Cable negro		**N1, N2**
	Cable rojo		
OUT4	Cable negro		
	Cable rojo		
Nota: Los pines A0-A3 no se emplean			

Tabla 8. Conexiones - Elegoo Smart Robot Car v3

Como apunte final, decir que el montaje de los componentes **mejorados** ha resultado ser bastante más **complejo** que el del kit (que también lleva su tiempo), requiriendo la conexión de todos ellos **uno a uno**, probándolos y comprobando su funcionamiento a cada paso. Más que nada porque son **más d**e **cincuenta** cables los que he acabado empleando, y si pensamos en su falta de rigidez estructural, que las conexiones se pueden soltar con relativa facilidad y que no hay ningún tipo de guía al respecto porque todo está hecho **a medida**, es fácil imaginar que se acaban necesitando ingentes cantidades de **paciencia** y **persistencia**. Pero como suele decirse, “sarna con gusto, no pica… aunque mortifica” ;)



ANEXO C:

INSTALACIÓN DEL ENTORNO DE APRENDIZAJE POR REFUERZO



Anexo C: Instalación del Entorno de Aprendizaje por Refuerzo

A lo largo del trabajo hemos ido viendo cómo desarrollar la **inteligencia** de nuestro robot autónomo, desde los algoritmos iniciales, más clásicos, hasta llegar al Aprendizaje por Refuerzo. Pero ¿qué tipo de **entorno** necesitaríamos para poder seguir desarrollando el presente trabajo?

Desde el punto de vista **hardware**, especialmente en el caso del Aprendizaje por Refuerzo, resultaría interesante disponer de cierta potencia computacional, cumpliendo al menos ciertos **mínimos**:

- **CPU**: Intel Core i5 8th gen o AMD Ryzen 1500.
- **GPU** (interesante, pero opcional): Nvidia GTX 1060
- **Memoria**: 8 GB de RAM.
- **Disco duro**: 60 GB.

Sin embargo, una de las mejores opciones por las que podríamos decantarnos sería emplear una **máquina virtual**, ya que lograríamos independizar la máquina física de la que emplearemos para realizar los cálculos, algo muy interesante debido a múltiples **factores**, como:

- **Portabilidad**.
- **Escalabilidad**.
- **Seguridad**.
- **Compatibilidad**.
- **Entorno** de pruebas **controlable**.

En cuanto al entorno sobre el que desplegar dicha máquina virtual, las **opciones** más relevantes son:

- **VirtualBox** (Oracle): Es software libre, y se lleva especialmente bien con Linux.
- **VMware** Workstation (VMware): Es de pago, pero hay quien la prefiere por ser más amigable de cara al usuario. Está disponible para múltiples plataformas y tiene un gran servicio técnico.

Para el trabajo me he decidido por emplear **VirtualBox 6.1**, ya que he trabajado anteriormente con él y siempre ha funcionado como esperaba, pudiendo exportar las máquinas virtuales de unos sistemas a otros sin mayores problemas y siendo siempre muy fluido. Según Oracle, sus **requisitos** hardware mínimos son los siguientes (Oracle, 2020):

- Procesador: Intel Core™ i5 8th gen, AMD Ryzen™ 1500X o superior.
- Memoria: 4 GB de RAM, 8 GB recomendados.
- Disco duro: 60 GB.

En el apartado **software**, una buena **distribución Linux**, a ser posible ligera, es todo lo que necesitaremos. A estas alturas, Linux está muy probado, es muy versátil y no necesita demasiados recursos. Aparte de ello, el SW **científico** suele estar sólo disponible para él. Las que personalmente **recomendaría** son:

- Lubuntu (https://lubuntu.net/)
- Ubuntu Mate (https://ubuntu-mate.org/)
- Linux Mint Xfce (https://www.linuxmint.com/rel_tessa_xfce_whatsnew.php)

Están todas derivadas de **Ubuntu**, que a su vez deriva de **Debian**, por lo que son estables, tienen un gran soporte y están siendo desarrolladas activamente. Lubuntu es una de las más reconocidas y utilizadas del sector, sobre todo en cuanto a su equilibrio rendimiento/facilidad de uso, así que he optado por emplearla. Y no me ha defraudado, ni por herramientas incluidas ni por velocidad.

Una vez dispongamos de los elementos necesarios, y ya con el archivo ISO de Lubuntu descargado en algún lugar de nuestro disco duro, ya podemos comenzar a preparar el entorno de pruebas, comenzando por instalar **VirtualBox**, para proceder a instalar la **MV**[35] después.

VirtualBox 6.1: Instalación

Es un proceso bastante **sencillo**, tanto en entornos Windows como Mac o Linux, no diferenciándose demasiado de la instalación de otro tipo de programas. Los **pasos** a dar son los siguientes:

1) Aceptar el **acuerdo** final de usuario.
2) Elegir la **opción** de instalación deseada (típica, normalmente).
3) Elegir el **directorio** de instalación.
4) Elegir dónde crear **accesos** directos (escritorio y menú de inicio, por ejemplo).
5) ¡Instalación **finalizada**!

VirtualBox 6.1: Cómo crear una máquina virtual

Una vez instalado VirtualBox, ya podemos crear nuestras propias máquinas virtuales. Siempre que creemos una, los pasos a dar serán los mismos, aunque deberemos **adaptar** el hardware virtual a las necesidades del sistema que vayamos a instalar.

En nuestro caso, y como ya se ha comentado con anterioridad, trataremos de instalar **Lubuntu** como máquina virtual en su versión más actual, la 20. De dicha versión hay dos **variantes** principales:

- **Intel x86**: Para sistemas de 32-bits, más antiguos y limitados.

[35] MV: Máquina Virtual

- **AMD 64-bit**: Para sistemas de 64-bits, como casi cualquiera de los actuales.

Decir que, a pesar de la nomenclatura, tanto los procesadores Intel como los AMD pueden ejecutar **cualquiera** de las dos versiones, aunque la versión de 32 bits puede funcionar en procesadores de 64-bit, pero no a la inversa. Y básicamente, para nuestros propósitos, van a funcionar de forma muy **similar**, aunque los paquetes a instalar no serán los mismos (suelen ser específicos para cada versión).

En este caso, optaré por instalar la versión de **64-bit**, porque actualmente las incompatibilidades existentes en antiguas versiones han desaparecido por completo, y las versiones de 64-bit suelen funcionar **mejor** cuando se trabaja con programas y herramientas que hacen uso extensivo de la **memoria** y de los recursos de la **CPU**. Y, como en todas las distribuciones basadas en Debian, siempre podremos hacer uso del excelente gestor de paquetes **Synaptic** y su `apt-get` para instalar todos los paquetes adicionales que nos hagan falta. Algo diferente a los entornos Windows, pero mucho, mucho más potente.

Los **pasos** a dar, en modo experto (recomendado), son los siguientes:

1) En la pantalla principal, seleccionar `Máquina` → **`Nueva`**:



Ilustración 93. Máquina virtual: Creación

2) Ahora elegiremos los **ajustes** básicos que, en nuestro caso, los fundamentales serán:
   - **Nombre**: Un nombre descriptivo, como "`Lubuntu 20`".
   - **Carpeta de máquina**: La ubicación donde deseamos que se instale.
   - **Tipo**: `Linux`.
   - **Versión**: `Ubuntu 64-bit`.
   - **Memoria (RAM)**: 2 GB = `2048 MB` (podrían ser menos, pero es una cantidad razonable).
   - **Disco duro**: `Crear un disco duro virtual ahora`.

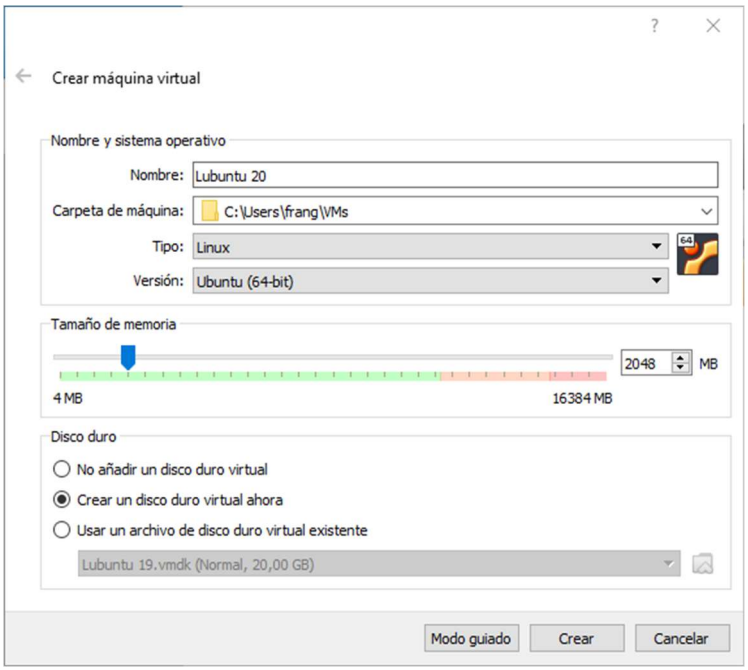


Ilustración 94. Máquina virtual: Ajustes básicos

3) En cuanto a las opciones del **disco duro virtual**:

- **Ubicación de archivo**: La carpeta que hemos seleccionado anteriormente.
- **Tamaño de archivo**: El espacio máximo que tendrá el disco duro a crear. En este caso, `20 GB` deberían ser suficientes, aunque hay opciones para aumentarlo en el futuro.
- **Tipo de archivo de disco duro**: `VMDK`. Hay más opciones, pero es la más compatible y estable.
- **Almacenamiento en unidad de disco duro física**: `Reservado dinámicamente`. De este modo, sólo se ocupará aquel espacio que sea necesario en cada momento.

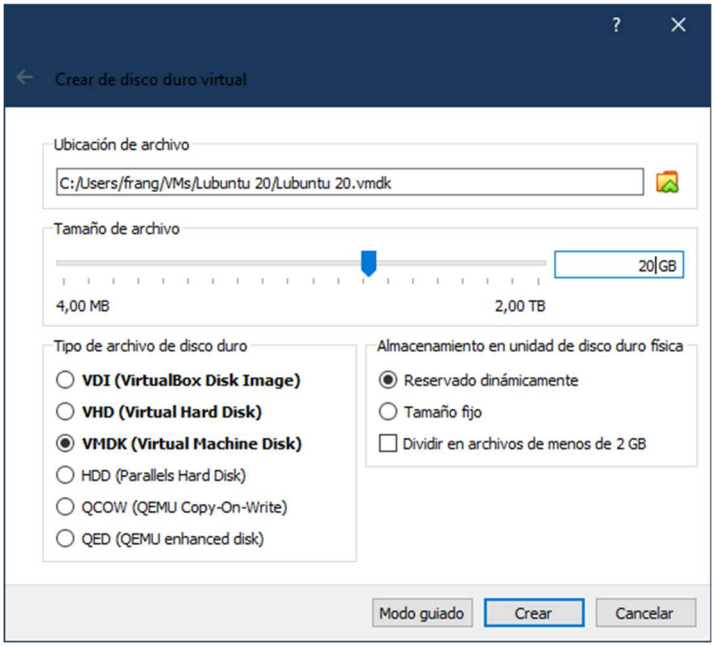


Ilustración 95. Máquina virtual: Disco duro virtual

4) Tras ello, el asistente nos indicará que la MV ya está creada, con las opciones que hemos puesto:

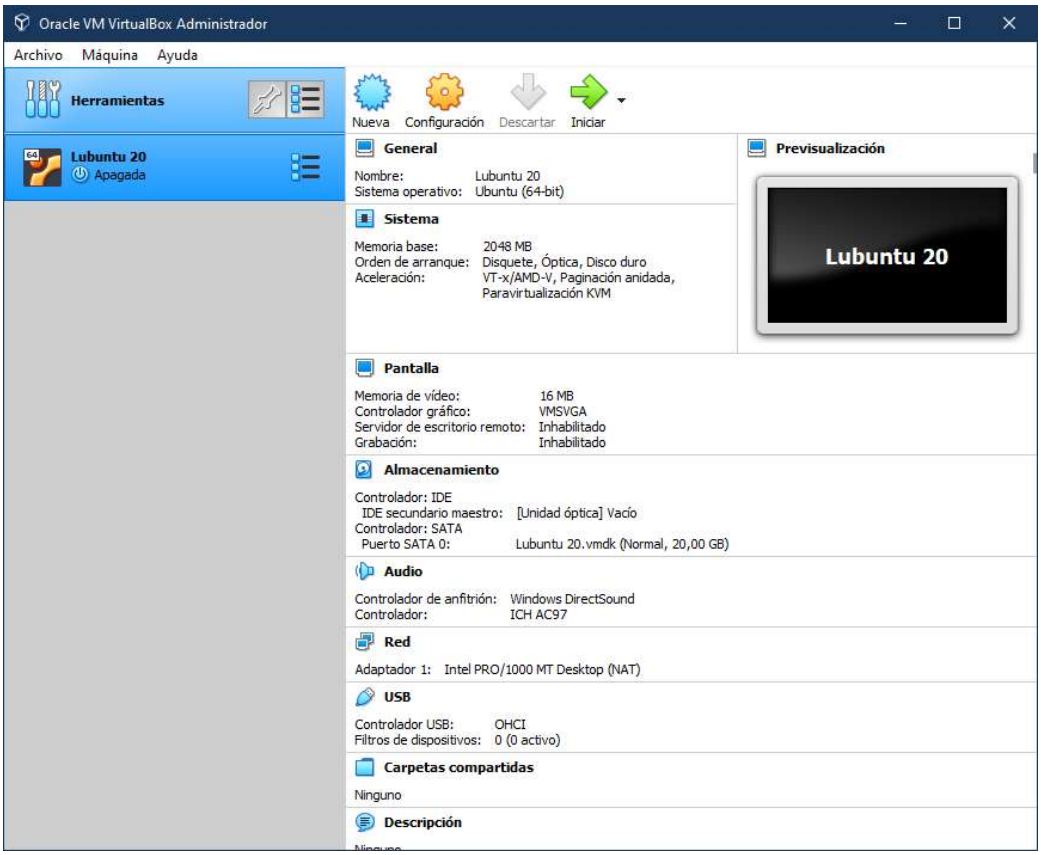


Ilustración 96. Máquina virtual: Creada

5) Ahora la MV ya podría **arrancar** sin problema (aunque aún **sin** sistema operativo cargado), pero conviene ajustar algunos **detalles** en la pestaña de `Configuración`, como:

- `General` → `Avanzado`:
    - Compartir portapapeles: Bidireccional

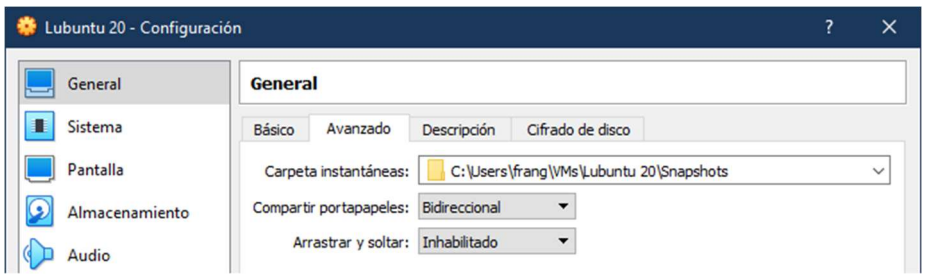


Ilustración 97. Máquina virtual: Ajuste adicional 1

- `Sistema` → `Procesador`:
    - Procesador(es): 2
    - Características extendidas: Habilitar PAE/Nx y VT-x/AMD-V

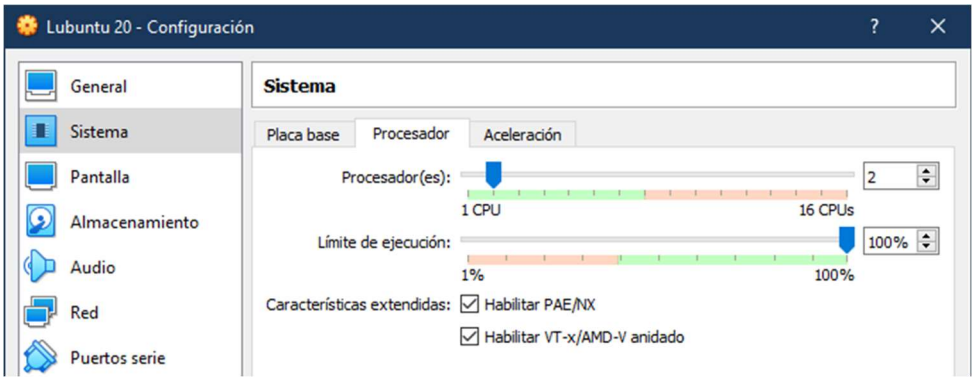


Ilustración 98. Máquina virtual: Ajuste adicional 2

6) En este punto, debemos indicar a VirtualBox dónde está **ubicado** el archivo **ISO** de Lubuntu que nos hemos descargado desde la web. Y dado que vamos a arrancar el proceso de instalación típico de cualquier distribución Linux, le indicaremos a VirtualBox que disponemos de una unidad de CD/DVD (**virtual**) donde está ubicado el disco. Para ello, debemos dar los siguientes **pasos**:
   1. Almacenamiento → Controlador IDE → IDE secundario maestro
   2. Click-izquierdo sobre el icono del disco → Seleccionar un archivo de disco
   3. Seleccionamos el archivo ISO de Lubuntu que nos hemos descargado.

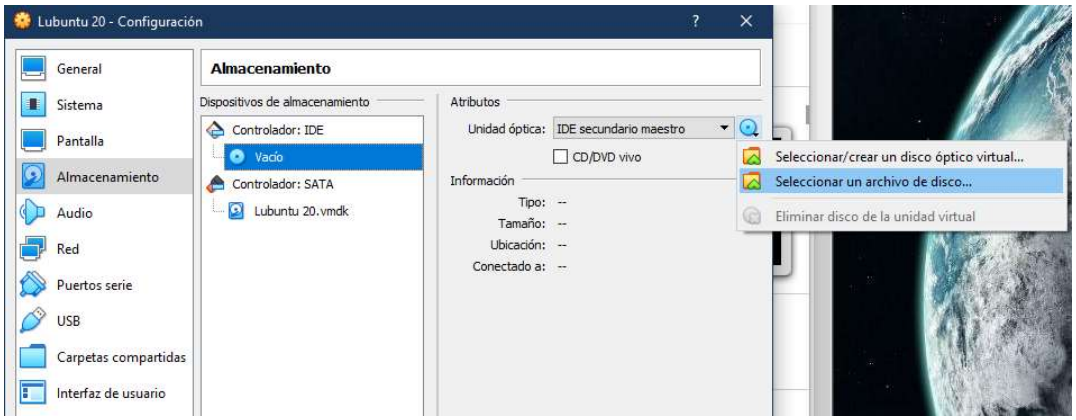


Ilustración 99. Máquina virtual: Ajuste adicional 3

7) Y ya estamos **listos** para instalar Lubuntu, como una MV, en nuestra máquina.

Lubuntu 20: Instalación e inicio del sistema

Una vez configurado el apartado **hardware** subyacente gracias a VirtualBox, procederemos a instalar Lubuntu 20 de la **misma** forma que lo haríamos en una máquina física.

Notar que siempre que queramos volver al **escritorio** de nuestra máquina anfitrión por cualquier motivo, como parar la máquina, reconfigurarla o, simplemente, pausarla para recuperarla posteriormente, podremos hacerlo pulsando la tecla `Ctrl-dch`.

Los **pasos** a dar son éstos:

1) Desde el menú de la VM, debemos pulsar en `Iniciar` → `Inicio Normal`, lo que hará que se **arranque** por primera vez la máquina:



Ilustración 100. Lubuntu 20: Arranque de la VM

2) En este punto, Lubuntu nos pregunta por el **idioma** de instalación, "Español" en nuestro caso:

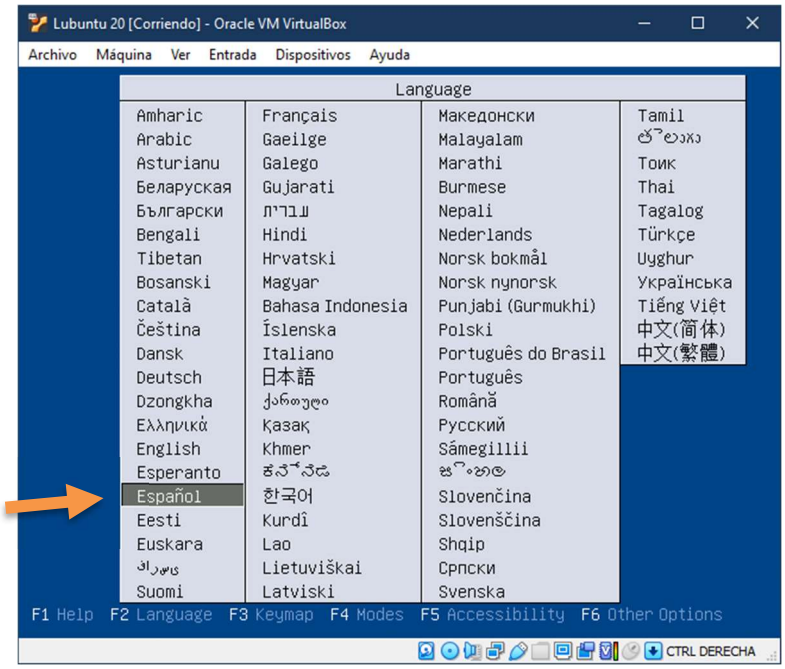


Ilustración 101. Lubuntu 20: Idioma de instalación

3) Ahora se nos preguntará por el tipo de arranque, y optaremos por el **normal** ("Start Lubuntu"):

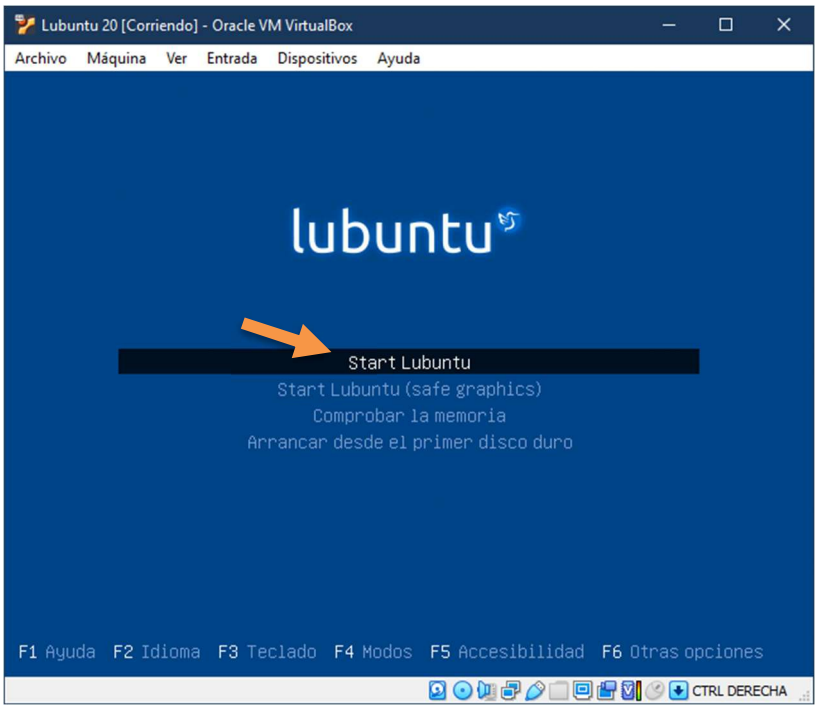


Ilustración 102. Lubuntu 20: Tipo de arranque

4) Tras un rato, **identificando** nuestro sistema para arrancarlo sin problema (Linux ha mejorado muchísimo en este sentido), aparecerá la siguiente pantalla (típica de los antiguos Live-CD):

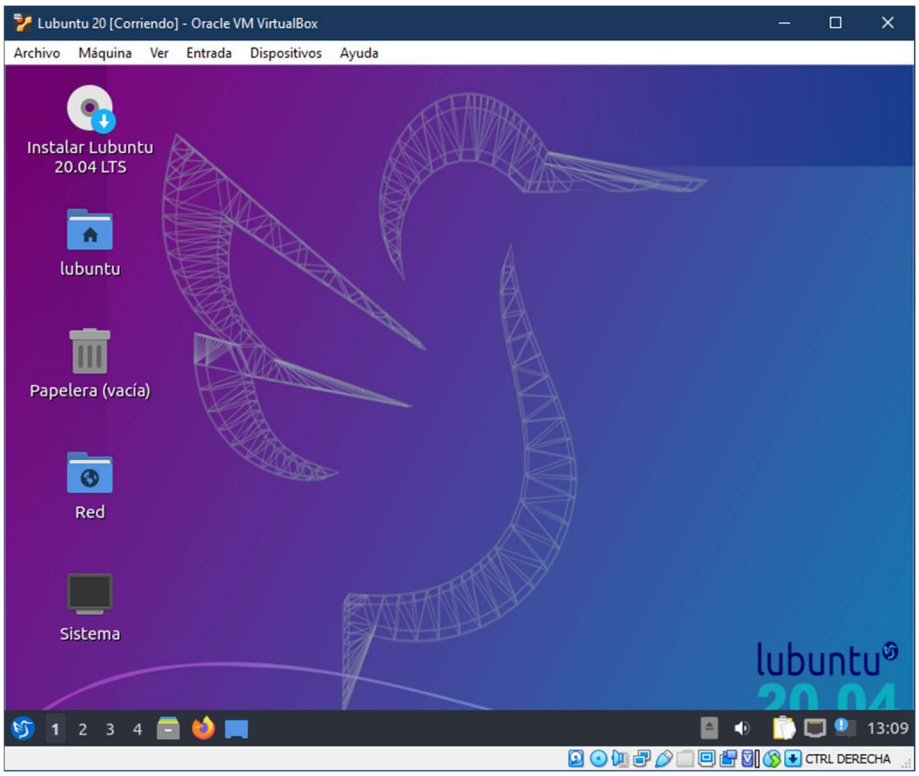


Ilustración 103. Lubuntu 20: Pantalla principal en el primer arranque

5) Ahora debemos pulsar sobre el icono de **instalación**:

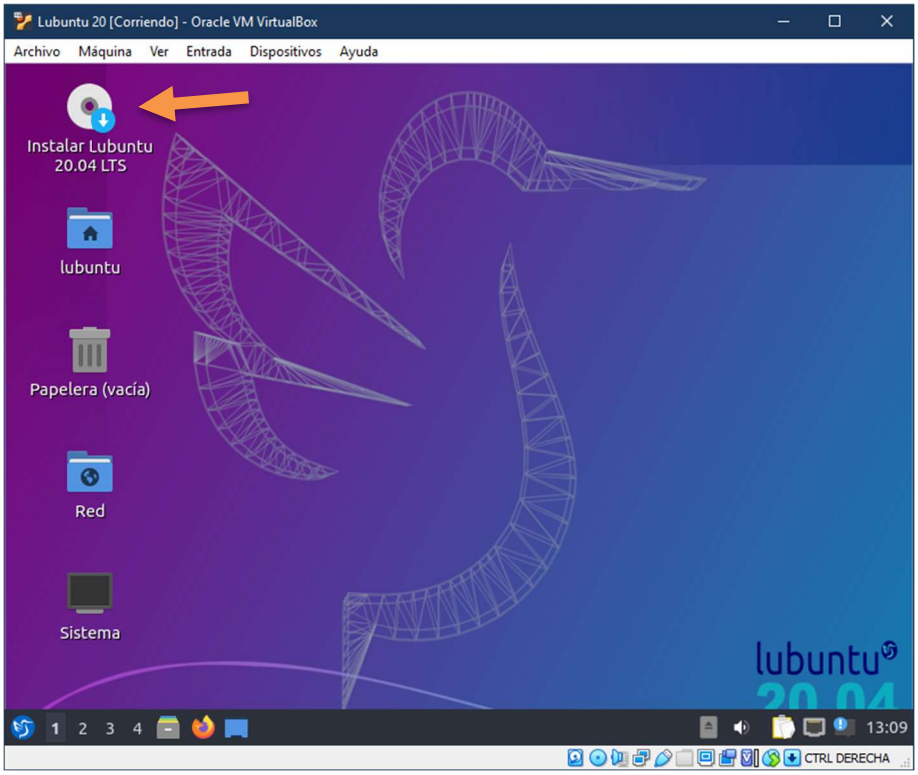


Ilustración 104. Lubuntu 20: Pantalla principal en el primer arranque

6) En este punto, debemos proceder a introducir los **parámetros** de la instalación:

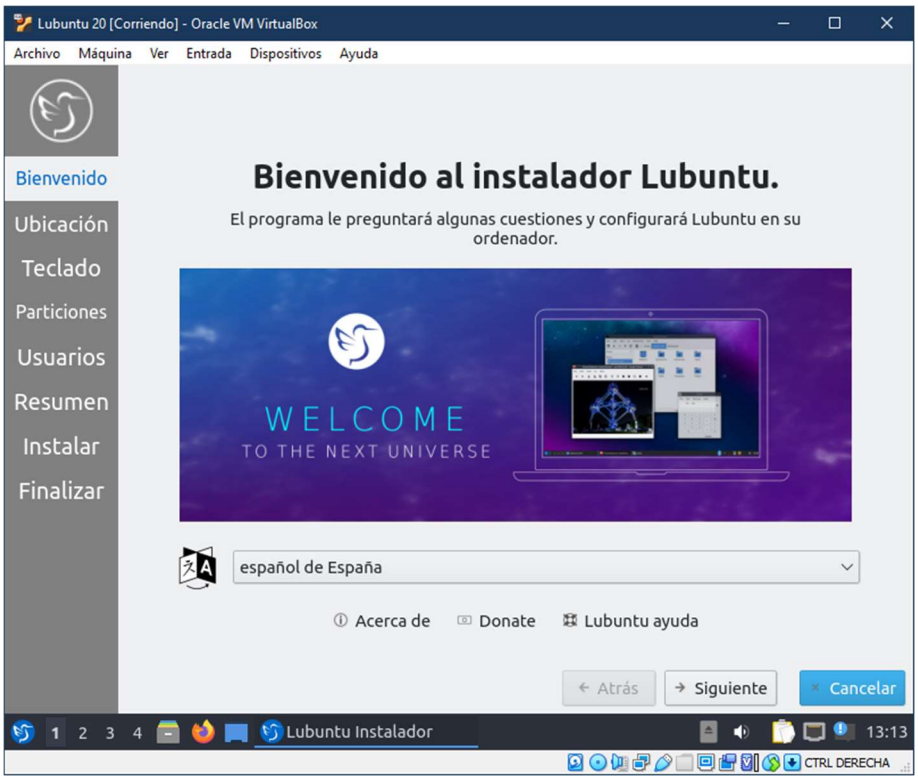


Ilustración 105. Lubuntu 20: Instalador

- **Idioma**: Español de España.
- **Ubicación**: Europe → Madrid
- **Teclado**: Spanish → Por defecto
- **Particionado**: Borrar disco
- **Nombre, nombre de equipo, password**: Los que se deseen.

7) Una vez ajustado, ya estaremos preparados para **instalar** Lubuntu, pulsando sobre `Instalar`:

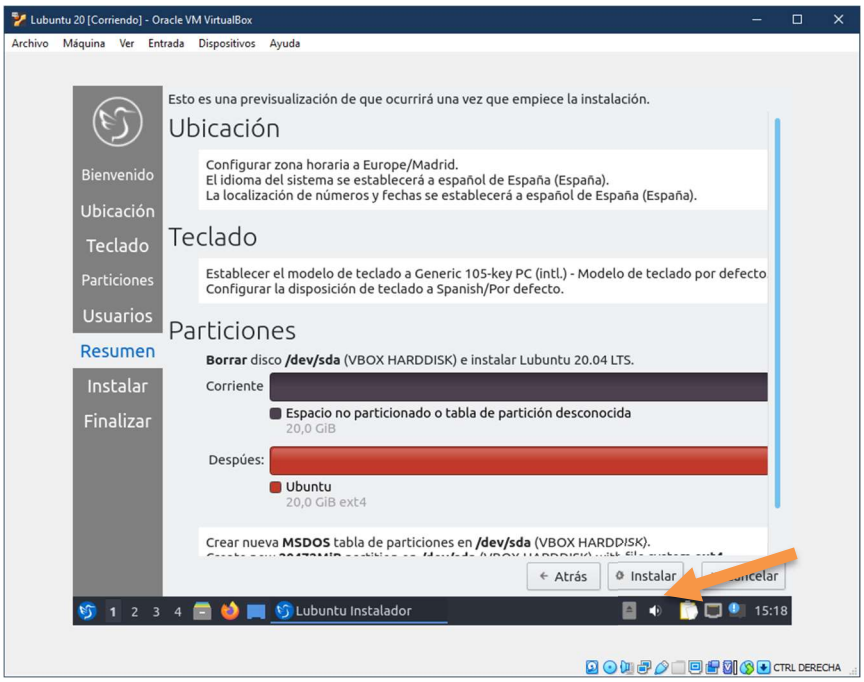


Ilustración 106. Lubuntu 20: Resumen de las opciones de instalación

8) Tras **acabar** la instalación, se mostrará la siguiente pantalla, y estará **listo**:

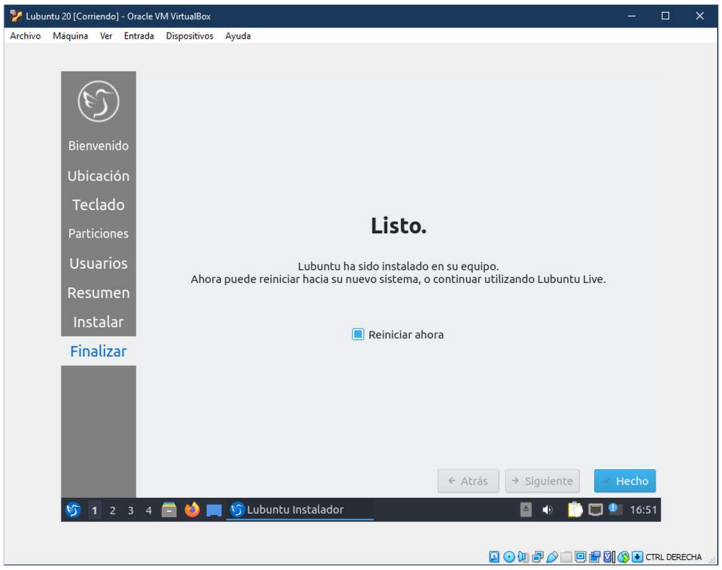


Ilustración 107. Lubuntu 20: Fin de la instalación

En este punto es mejor no optar por “reiniciar” el sistema, y es preferible **apagarlo**. Es así, porque aún quedará “sacar” el CD/DVD con la ISO de la unidad **virtual** desde la que se ha hecho la instalación, y es mejor hacerlo con la MV apagada.

Puede hacerse desde `Archivo` → `Administrador de Medios Virtuales`, haciendo `click-izq` sobre el botón `Eliminar`, una vez elegida la imagen ISO del instalador de Lubuntu:

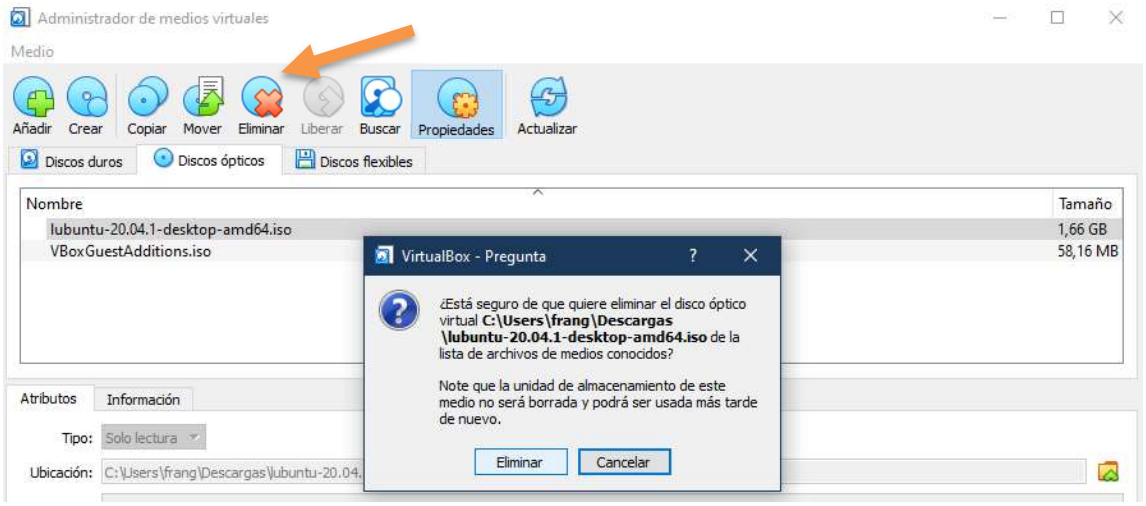


Ilustración 108. Lubuntu 20: Eliminación del disco de instalación

Lubuntu 20: Instalación de las Guest Additions

Una vez instalado Lubuntu, el último paso a dar es instalar las **extensiones** propias de VirtualBox para habilitar ciertas funciones adicionales, llamadas “Guest Additions”. Sirven para poder reajustar el tamaño de pantalla “al vuelo”, para que funcionen las carpetas compartidas que se habiliten o para que no haya problema con “copiar-y-pegar” texto desde la máquina anfitriona a la máquina MV y viceversa.

En Linux puede resultar algo “complejo”, pero si se siguen estas **indicaciones**, no habrá problema:

1) Con la MV arrancada, click-izq en `Dispositivos` → `Insertar imagen de CD…` :



Ilustración 109. Lubuntu 20 – Guest Additions: Insertar CD virtual

2) El CD virtual tratará de **autoarrancarse**, para lo que se deberá pulsar sobre `Aceptar`:

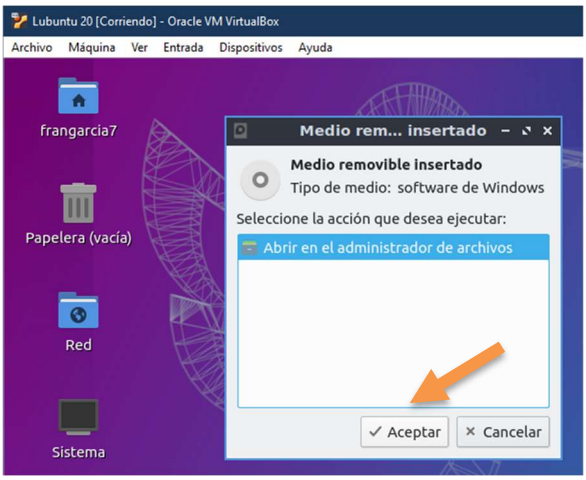


Ilustración 110. Lubuntu 20 – Guest Additions: Autoarranque

3) Hacemos `doble-click` sobre `autorun.sh` (o sobre `VBoxLinuxAdditions.run`, en caso de que el autoarranque no se ejecute correctamente) → `Ejecutar` en una `terminal`:

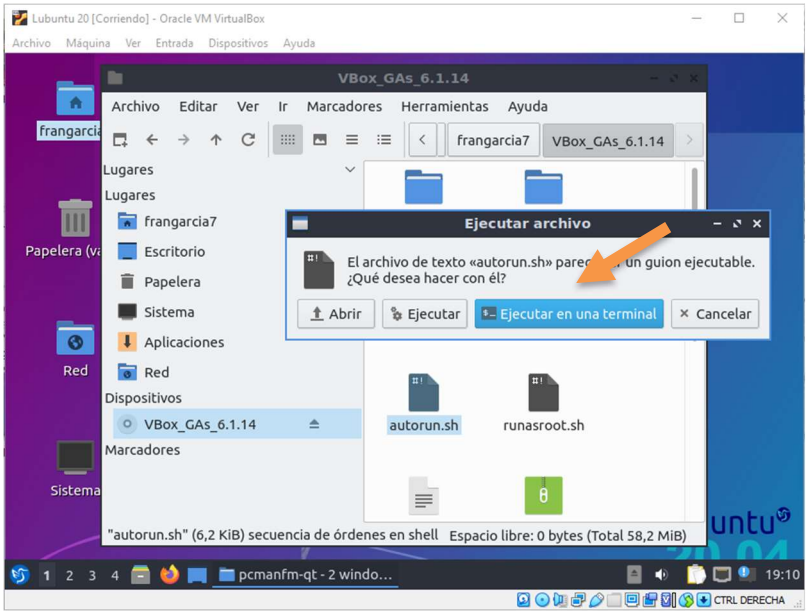


Ilustración 111. Lubuntu 20 – Guest Additions: autorun.sh

4) Y en un instante estará listo, tan sólo quedará **reiniciar** el sistema y cerciorarse de que las opciones que necesitamos (portapapeles, etc), están **activadas** para la MV que estamos ejecutando.

Lubuntu 20: Instalación de matplotlib, numpy, keras y tensorflow con VirtualEnv

Bien, Lubuntu ya está instalado. Pero para poder tener un **entorno** apto para desarrollar y ejecutar aplicaciones que empleen Aprendizaje por Refuerzo hacen falta ciertas **aplicaciones**, como:

- **Pip3**: Gestor de paquetes basado en Python muy potente y flexible.
- **Numpy**: Implementa funciones matemáticas de alto nivel.

- **Tensorflow**: Biblioteca para Aprendizaje Automático, capaz de construir y entrenar redes neuronales para detectar y descifrar patrones y correlaciones.
- **Keras**: Biblioteca de Redes Neuronales, capaz de ejecutarse sobre TensorFlow.
- **Matplotlib**: Genera gráficos de forma potente y sencilla.

Existen muchas formas de instalar estas aplicaciones, pero siempre acaban surgiendo **problemas** de versiones, compatibilidades, interacciones entre unas y otras… Por eso, mi recomendación es instalar lo que se conoce como un "**entorno virtual**" gracias a, por ejemplo, **VirtualEnv**.

Virtualenv es una herramienta utilizada para crear un **entorno** Python **aislado**. Este entorno tiene sus propios directorios de instalación que **no comparten** bibliotecas con otros entornos virtualenv o las bibliotecas instaladas globalmente en el servidor, lo que permite tener entornos aislados, sin ninguno de los problemas antes mencionados. Es como si fueran, a su vez, pequeñas MVs enfocadas a ejecutar ciertas aplicaciones dentro de la propia MV, y esto es algo realmente útil.

Así pues, si queremos instalar Virtualenv y todas las aplicaciones/programas antes mencionados en un entorno virtual a medida, deberemos seguir los siguientes **pasos** desde una terminal Linux:

1) Instalar Pip3: `sudo apt install python3-pip`
2) Instalar Virtualenv: `sudo apt-get install virtualenv`
3) Añadir Virtualenv a la variable $PATH de Linux: `source ~/.profile`
4) Crear un entorno virtual llamado "RL": `virtualenv RL`
5) Activar el entorno virtual: `source RL/bin/activate`

   Al activar el entorno, se verá su nombre al principio de la línea de comandos ("RL", en este caso)

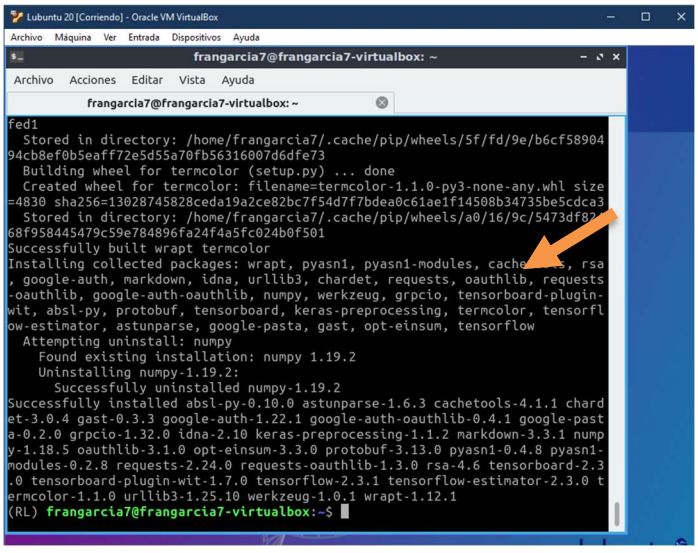


Ilustración 112. Lubuntu 20 – VirtualEnv con SW instalado

6) Instalar MatPlotLib: `pip3 install --upgrade matplotlib`
7) Instalar Numpy: `pip3 install --upgrade numpy`
8) Instalar Keras: `pip3 install --upgrade keras`
9) Instalar Tensorflow: `pip3 install --upgrade tensorflow`

Y **listo**, ya tendremos todos los paquetes necesarios para desarrollar y ejecutar aplicaciones que emplean *Machine Learning* y Aprendizaje por Refuerzo, con las tecnologías más actuales (ver *Ilustración 113*).

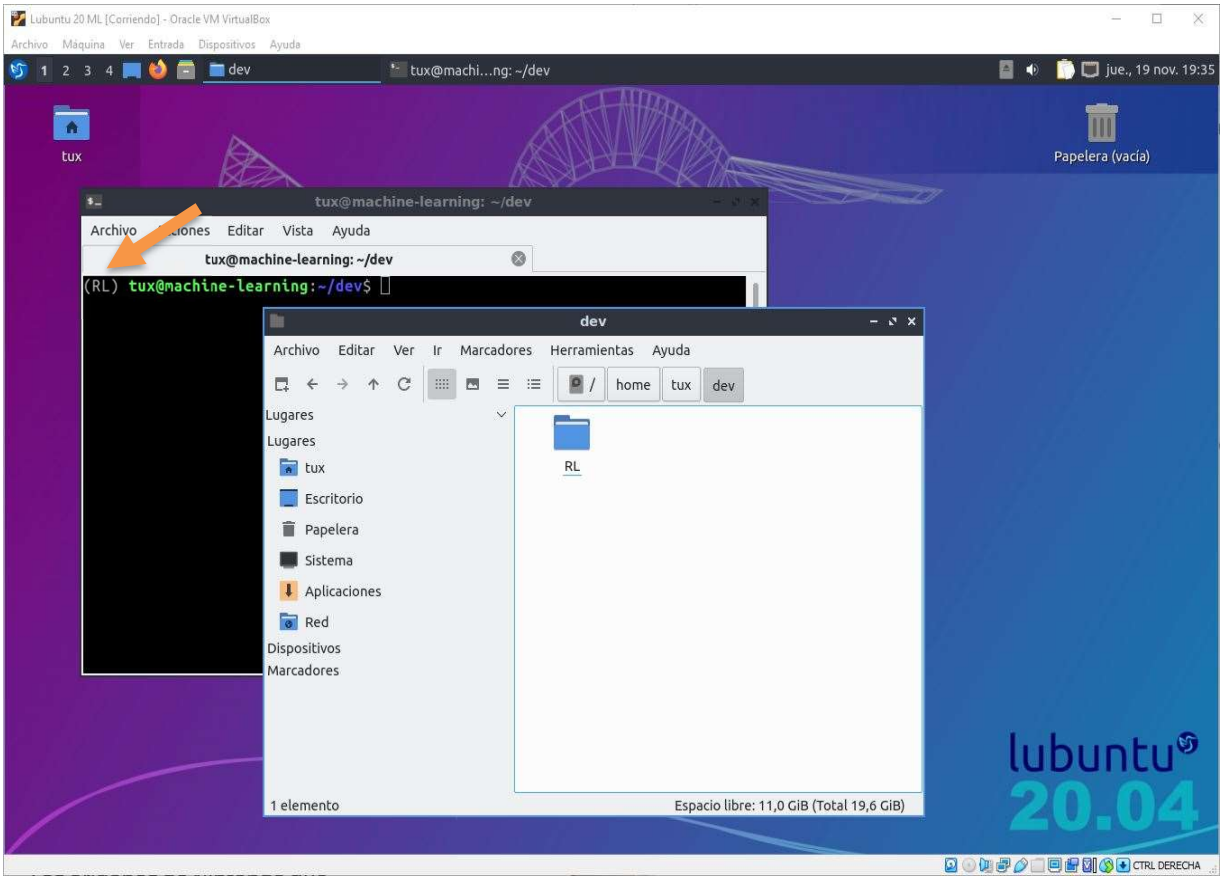


Ilustración 113. Lubuntu 20 – VirtualEnv con SW instalado