



UNIVERSIDAD DE GRANADA

TRABAJO FIN DE GRADO
INGENIERÍA INFORMÁTICA

Implementación de sistema empotrado de control robot en plataforma de procesamiento de altas prestaciones

Un estudio de técnicas de inteligencia artificial biológicamente inspiradas

Autor

Antonio José Blánquez Pérez

Directores

Eduardo Ros Vidal

Richard R. Carrillo Sánchez



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍAS INFORMÁTICA Y DE
TELECOMUNICACIÓN

Granada, 8 de julio de 2021

Implementación de sistema empotrado de control robot en plataforma de procesamiento de altas prestaciones inspirados: un estudio de técnicas de inteligencia artificial biológicamente inspiradas

Antonio José Blánquez Pérez

Palabras clave: robótica, sistema empotrado, interacción humano-robot, redes neuronales por impulsos, NVIDIA Jetson, ROS, EDLUT

Resumen

Actualmente el control de robots industriales o robots colaborativos sofisticados se realiza desde servidores de control que incluyen componentes hardware (para procesamiento de altas prestaciones si es necesario) y componentes software como bibliotecas de control y comunicaciones que facilitan la implementación de esquemas de control. Hasta hace poco los robots industriales tenían herramientas y software de control específico y propietario, lo cual hacía difícil trasladar esquemas de control de un tipo de robot a otro. Actualmente ya existen entornos y middleware ampliamente utilizados como ROS (Robotic Operating System) que facilitan la comunicación con los sensores y actuadores de diversos robots a un nivel de abstracción mayor.

En los últimos años han surgido plataformas de procesamiento de altas prestaciones (integrando CPU y GPU) para sistemas empotrados. En este trabajo fin de grado se afronta la integración de esquemas de control robótico neuronal y software de comunicación con el robot, en plataforma de computación empotrada de altas prestaciones. Esto permitirá avanzar en darle mayor autonomía a robots que tradicionalmente se controlan desde un servidor central.

Embedding robot control system in high performance processing platform: a study of biologically inspired artificial intelligence techniques

Antonio José Blánquez Pérez

Keywords: robotics, embedded systems, human-robot interaction, spiking neural networks, NVIDIA Jetson, ROS, EDLUT

Abstract

Currently, the control of sophisticated industrial and collaborative robots is performed from control servers, which include hardware (for high performance when necessary) and software components such as control and communications libraries that ease the control scheme implementation. Until recently almost all industrial robots had custom and proprietary control hardware and software, which prevent porting control schemes from a type of robot to another. Currently, there already exist widely-used middleware and environments, such as ROS (Robotic Operating System), that facilitate the communications with sensors and actuator from different robots at a higher level of abstraction.

In recent years a variety of single-board computers include high performance processing units (integrating CPU and GPU). This final project faces the integration of neural robotic control schemes and communication software in a high-performance embedded computing platform. This integration aims at making progress in the availability of more autonomous robots, which were traditionally controlled from a central server.

Agradecimientos

Antes de comenzar, quería agradecer su apoyo a todos aquellos que me han acompañado a lo largo de esta etapa de mi vida. Si hoy estoy donde estoy, es gracias a todos vosotros.

A mis tutores, Eduardo y Richard, por la dedicación, por toda la ayuda y el conocimiento que me habéis aportado, por encontrar huecos donde no los hay. Simplemente gracias, nunca pensé que llegaría a hacer un proyecto de estas características.

Al resto del grupo de investigación, en especial a Ignacio y Paco, por la acogida, por el trato, por ayudarme y formarme en las herramientas más complejas del proyecto.

A mi familia, Fefa, Antonio y Samuel, por enseñarme el valor del esfuerzo y el trabajo, por el apoyo mutuo, por dejarme conducir mi vida, por los sacrificios y dolores de cabeza, por criarme en valores de libertad e inconformismo. Y por todo lo demás, sin vosotros hoy no estaría aquí.

A vosotros, es imposible listar a todos, que habéis estado y seguís ahí, por todos esos momentos inolvidables y por el apoyo cuando se necesitaba. Tanto a los que os he conocido estos últimos años como los que estáis ahí desde siempre, porque no todo en la vida es trabajo y estudio.

Índice general

I	Introducción	1
1.	Introducción y objetivos	3
1.1.	Motivación	3
1.2.	Propuesta y objetivos	5
1.3.	Planificación	7
2.	Dominio y estado del arte	13
2.1.	Plataformas de procesamiento	13
2.2.	Robots en la industria	16
2.3.	Frameworks de control para robots	20
2.4.	Entornos de simulación	22
2.5.	Paradigmas de control	24
2.5.1.	Control prealimentado	24
2.5.2.	Control realimentado: controlador PID	25
2.5.3.	Otros esquemas de control	27
2.6.	Redes neuronales artificiales	27
3.	Material	31
3.1.	El robot Baxter	31
3.2.	El entorno de simulación Gazebo	33
3.3.	La librería cROS	34
3.4.	La plataforma empotrada NVIDIA Jetson	34
3.5.	Otros materiales	36
3.6.	Presupuesto	36
3.6.1.	Recursos materiales	36
3.6.2.	Recursos humanos	36
3.6.3.	Coste total	37
II	Desarrollo: sistema de control robot	39
4.	Ciclo de control de partida	41
4.1.	Control en lazo abierto	41

4.1.1.	Tiempo de simulación y retardo por conexión inalámbrica simulado	41
4.1.2.	Generación de trayectorias	45
4.1.3.	Traducción de mensajes de control	46
4.1.4.	Monitorización	47
4.1.5.	Funcionamiento	47
4.2.	Control en lazo cerrado: controlador PD	48
4.2.1.	El controlador PD	48
4.2.2.	Ajuste del controlador PD	51
4.2.3.	Funcionamiento	51
4.3.	Problemática, obsolescencia y adecuación	52
5.	cROS como alternativa	53
5.1.	Planteando la adaptación del diseño base	53
5.2.	Adaptación de nodos ROS a cROS	54
5.2.1.	Funcionalidades de tiempo	54
5.2.2.	Acceso a los campos de los mensajes	55
5.2.3.	Contexto entre <i>publishers</i> y <i>subscribers</i>	57
5.2.4.	Envío de mensajes periódicos	59
5.2.5.	Otras consideraciones	62
5.3.	Diseño final de los nodos en cROS	63
5.3.1.	Ciclo en lazo abierto	63
5.3.2.	Ciclo en lazo cerrado: controlador PD	64
5.3.3.	Extracción de resultados	65
III	Desarrollo: simulador neuronal en plataforma empotrada	67
6.	Definición del modelo de red neuronal	69
6.1.	La neurona	69
6.2.	El modelo LIF	72
6.3.	EDLUT: Event-Driven LookUp Table	73
6.4.	Un cerebro artificial	75
7.	Puesta a punto de la plataforma NVIDIA Jetson	79
7.1.	Instalación y puesta en marcha	79
7.2.	Medidas y perfiles de consumo	80
7.2.1.	Herramientas de medida de consumo eléctrico	80
7.2.2.	Herramientas usadas durante el proyecto	81
7.2.3.	Consumo por parte de los periféricos	83
7.2.4.	Perfiles de consumo	83
7.3.	Instalación y configuración de EDLUT	84
7.4.	Test preliminares: topología de las redes	85

7.5. Test preliminares: impulsos de entrada	87
7.6. Test preliminares: paralelización	87
7.7. Test preliminares: ED vs TD	89
7.8. Test preliminares: densidad de spikes	92
7.9. Rendimiento en términos de potencia computacional y consumo	93
7.10. Perfil de potencia <i>ad hoc</i>	94
7.11. Estudio de viabilidad	96
IV Resultados definitivos y conclusiones	99
8. Resultados	101
8.1. Rendimiento del sistema de control en lazo abierto	101
8.2. Rendimiento del sistema de control en lazo cerrado con PD .	103
8.3. Comparativa de EDLUT-Jetson con otras plataformas	104
8.3.1. Comparativa de rendimiento y consumo	105
8.3.2. Comparativa de eficiencia	109
8.3.3. Forma de exponencial decreciente	112
8.4. Comparativa de EDLUT-Jetson con SpiNNaker	113
9. Conclusiones	119
9.1. Objetivos logrados	119
9.2. Conclusión	121
9.3. Trabajo futuro	122
Bibliografía	129
A. Guia de usuario de EDLUT-cROS	131
A.1. Introducción	131
A.2. Instalación	131
A.3. Ejecución	132
B. Guía de usuario de Jetson Tools	133
B.1. Introducción	133
B.2. Instalación y ejecución	133
B.3. Descripción de los archivos	134

Índice de figuras

1.1. A la izquierda, robots industriales en un perímetro de seguridad, a la derecha, un robot colaborativo acompañado de un trabajador. Fuentes: https://maquinasyequipos.com.ar/seguridad-en-robots-industriales/ y https://www.ioteca.com/smart-factory/historia-robot-colaborativo/	4
1.2. Diagrama de Gantt.	7
1.3. Tablero de Trello en un instante de trabajo	9
2.1. De izquierda a derecha y de arriba a abajo: Nvidia Jetson AGX Xavier, Raspberry Pi 4, Arduino Uno y SpiNNaker. Fuentes: https://www.nvidia.com/es-es/autonomous-machines/jetson-store/ , https://es.wikipedia.org/wiki/Raspberry_Pi , https://commons.wikimedia.org/wiki/File:Arduino_Uno_-_R3.jpg , https://www.researchgate.net/figure/A-SpiNNaker-board-with-48-chips-SpiNN-5_fig1_301559712	14
2.2. Número de robots instalados en un año. Fuente: [36]	17
2.3. De izquierda a derecha y de arriba a abajo: Universal Robot UR16e, ABB IRB 14000 YUMI, Kuka LBR iiwa y Rethink Robotics Baxter. Fuentes: https://www.universal-robots.com/es/productos/ , https://www.tecnoplcc.com/yumi-robot-colaborativo-de-abb/ , https://www.kuka.com/es-es/productos-servicios/sistemas-de-robot/robot-industrial/lbr-iiwa y https://cobotsguide.com/2016/06/rethink-robotics-baxter/	18
2.4. Transmisión elástica. Fuente: [30].	19
2.5. Arquitectura de una red ROS	21
2.6. Ejemplo de comunicación a través de ROS. Fuente: http://wiki.ros.org/Master	23
2.7. Sistema de control prealimentado.	25

2.8.	Ciclo de control genérico en lazo cerrado con controlador PID. Fuente: https://www.researchgate.net/figure/A-generic-closed-loop-process-control-system-with-PID-controller_fig1_242727578	26
2.9.	Estructura de una red neuronal artificial. Fuente: Yaser[2] . .	28
3.1.	Robot Baxter. Fuente: https://www.elempaque.com/temas/Baxter,-el-robot-colaborativo-que-opera-en-equipo-con-el-personal+103502	31
3.2.	Articulaciones del robot Baxter. Fuente: [38]	32
3.3.	SoC NVIDIA Tegra. Fuente: https://en.wikichip.org/wiki/nvidia/tegra/xavier	35
4.1.	Ciclo de control en lazo abierto	42
4.2.	Ciclo de control en lazo abierto simplificado	44
4.3.	Esbozo de trayectorias deseadas, vistas en planta. A la izquierda la trayectoria circular, a la derecha la trayectoria en forma de ocho	45
4.4.	Traza de los comandos de trayectoria durante el ciclo de control en lazo abierto	48
4.5.	Ciclo de control en lazo cerrado con control por PD	49
4.6.	Ciclo de control en lazo cerrado con control por cerebelo artificial	52
5.1.	Ciclo de control en lazo abierto implementado con la librería cROS.	64
5.2.	Ciclo de control en lazo cerrado con controlador PD implementado con la librería cROS.	65
6.1.	Estructura de una neurona. Fuente: https://www.xeridia.com/blog/redes-neuronales-artificiales-que-son-y-como-se-entrenan-parte-i	70
6.2.	Potencial de membrana de una neurona biológica. Fuente: https://tareasfisiologialefuentes.home.blog/2019/02/03/potenciales-de-membrana-y-potenciales-de-accion/	71
6.3.	Circuito que representa un modelo LIF. Fuente: [12]	72
6.4.	A la derecha, la microcircuitería de un cerebelo real, a la izquierda, la topología de la red neuronal artificial que lo modela. Fuentes:[1] y http://en.wikipedia.org/wiki/File:CerebCircuit.png , CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=8010900	77

7.1. Por orden de izquierda a derecha: vatímetro digital de corriente continua, vatímetro digital de corriente alterna adaptado a enchufes, vatímetro de pinza y SAI. Fuentes: http://www.amazon.es/Dinam%C3%B3metro-Meter-1Pc-Wattmeter-potencia-equilibrio/dp/B07RJFXGY7 , https://www.amazon.com/-/es/Digital-Wattmeter-Monitor-Electricity-Analyzer/dp/B07P89KZQ6 , https://peakttech-rce.com/es/multimetros-digital-a-pinza/197-peakttech-1640-pinza-amperimetrica-vatimetro-ccca-3-digitos-17mm-1000a-40-240kw-30mm-trms.html , http://www.fadrell.com/etiqueta/sistema-de-alimentacion-ininterrumpida/	81
7.2. Comparación entre monitores de consumo	82
7.3. Ahorro de energía evitando el uso de periféricos	84
7.4. Perfiles de potencia por defecto para la NVIDIA Jetson. Fuente: documentación oficial[9]	85
7.5. Arriba, los tiempos de ejecución para EDLUT sin generar spikes de entrada, abajo, el mismo experimento introduciendo inputs.	88
7.6. Comparación de ejecuciones usando multi-threading.	89
7.7. Comparación de tiempos de ejecución para redes de varios tamaños usando las tres opciones y distintos muestreos de ED. Se marca en línea negra discontinua el tiempo simulado en las redes, que ha sido en todo caso de 10 segundos.	91
7.8. Comparación de potencia consumida en la Jetson para redes de varios tamaños usando las tres opciones.	91
7.9. Comparación de spikes/s para redes de varios tamaños usando las tres opciones.	92
7.10. Comparación de tiempo de ejecución para distintos intervalos inter-spike usando las tres opciones.	93
7.11. Comparación de potencia consumida para distintos intervalos inter-spike usando las tres opciones.	94
7.12. Comparación de spikes/s/W para redes de varios tamaños usando las tres opciones.	95
7.13. Comparación de spikes/s/W para distintos intervalos inter-spike usando las tres opciones.	95
7.14. Comparación de spikes/s/W entre el perfil de potencia de máximo rendimiento de la Jetson (MAXN) y el perfil personalizado.	96
8.1. Trayectorias seguidas por el brazo del Baxter simulado para el control en lazo abierto. A la izquierda para la trayectoria circular, a la derecha para la trayectoria en forma de ocho . .	102

8.2. Posición y velocidad de algunas articulaciones del Baxter simulado para el control en lazo abierto.	103
8.3. Trayectoria seguidas por el brazo del Baxter simulado para el control en lazo cerrado (trayectoria circular).	103
8.4. Posición y velocidad de las articulaciones del Baxter simulado para el control en lazo cerrado.	104
8.5. Comparativa de rendimiento para las distintas plataformas en simulación ED.	106
8.6. Comparativa de rendimiento para las distintas plataformas en simulación TD-CPU.	107
8.7. Comparativa de rendimiento para las distintas plataformas en simulación TD-GPU.	108
8.8. Comparativa de consumo para las distintas plataformas. . . .	110
8.9. Comparativa de eficiencia para las distintas plataformas en simulación ED.	111
8.10. Comparativa de eficiencia para las distintas plataformas en simulación TD-CPU.	111
8.11. Comparativa de eficiencia para las distintas plataformas en simulación TD-GPU.	112
8.12. Arriba, medida de spikes producidos en una red respecto a las neuronas de las que está compuesta, abajo, dichos valores normalizados respecto a la red más pequeña.	114

Índice de cuadros

1.1.	Lista de <i>issues</i> relacionadas con el sistema de control.	10
1.2.	Lista de <i>issues</i> relacionadas con el estudio de EDLUT en la Jetson.	10
1.3.	<i>Sprints</i> definidos durante el proyecto.	11
3.1.	Recursos utilizados durante el proyecto.	37
3.2.	Horas dedicadas al proyecto.	37
3.3.	Coste total del proyecto.	38
7.1.	Redes usadas en los experimentos con EDLUT en la NVIDIA Jetson	86
8.1.	Tabla de especificaciones de las distintas plataformas usadas en la comparativa	105
8.2.	Comparativa entre la red de ejemplo de SpiNNaker y 10_100 en la Jetson.	115
8.3.	Consumo de la SIMULACIÓN de distintas redes con EDLUT en la Jetson frente a consumo teórico en SpiNNaker. Se muestra el porcentaje de potencia que consume la Jetson respecto a SpiNNaker.	116
8.4.	Consumo TOTAL de distintas redes con EDLUT en la Jetson frente a consumo teórico en SpiNNaker. Se muestra el porcentaje de potencia que consume la Jetson respecto a SpiNNaker. 116	

Parte I

Introducción

Capítulo 1

Introducción y objetivos

1.1. Motivación

Robot fue el término que usó Čapek, en *R.U.R.* (*Robots Universales Rossum*), para referirse a los humanos artificiales que aligeraban la carga de trabajo del resto de trabajadores en la citada representación teatral. A esta se le atribuye la primera aparición de la palabra robot, cuyo origen parece ser el término checo *robota*, que se puede traducir como “servidumbre” o “trabajo duro”. En concordancia con esta definición, la utilidad principal de los robots es liberar a los humanos de labores pesadas y repetitivas. Además, proporcionan una mayor precisión y fiabilidad en trabajos largos debido a que estos no experimentan cansancio, así como un ahorro en costes de producción y una reducción del riesgo humano en ciertas aplicaciones.

Poco a poco distintos tipos de robots se están introduciendo tanto en la industria como en nuestra vida cotidiana. Desde los robots aspiradora que comienzan a limpiar nuestros suelos, pasando por los brazos soldadores en la industria, hasta los coches autónomos que nos permitirán realizar viajes sin conductor en el futuro, este tipo de soluciones tecnológicas nos facilitan la vida en ámbitos impensables hace algunos años. Actualmente el campo de aplicación más extendido es la utilización de brazos robóticos en la industria, donde la gran mayoría de robots están acotados por un perímetro de seguridad al que no pueden acceder trabajadores humanos. Un ejemplo de ello puede ser el de los robots de la Figura 1.1, que ocupan un espacio cercado que no puede ser utilizado por los trabajadores de la fábrica mientras la cadena esté en funcionamiento.

En los últimos años han aparecido una serie de robots, llamados cobots o robots colaborativos, que buscan poner solución a este problema de la interacción humano-robot. La *HRI* (*Human-Robot Interaction*) es un campo interdisciplinar que estudia la interacción entre individuos humanos y ro-



Figura 1.1: A la izquierda, robots industriales en un perímetro de seguridad, a la derecha, un robot colaborativo acompañado de un trabajador. Fuentes: <https://maquinasyequipos.com.ar/seguridad-en-robots-industriales/> y <https://www.ioteca.com/smart-factory/historia-robot-cola> borativo/

bots. Algunos de los problemas que intenta resolver son el intercambio de información, sea verbal o escrita, y la formación de equipos[24]. Sin embargo, la parte interesante para este proyecto es el problema del contacto físico.

Para considerar que existe colaboración entre humano y robot se requieren cuatro elementos. El primero es que ambos compartan horario de trabajo, es decir, estén trabajando al mismo tiempo. Si además cumple el segundo, que es un espacio de trabajo común, podemos hablar de coexistencia entre humano y robot. Asimismo el tercer elemento, el enfoque, requiere que estén ejerciendo la misma tarea, lo que nos permite hablar de cooperación. Finalmente, pero no menos importante, existe una cuarta variable para que podamos hablar de colaboración: el contacto.

El contacto físico entre humanos y robots es un reto debido al riesgo que supone. Los robots tradicionales están compuestos de articulaciones rígidas y son muy pesados, lo que implica grandes inercias y una mínima amortiguación de sus movimientos ante colisiones. Este es el motivo por el que existe un gran riesgo para el trabajador: un despiste o un fallo en la unidad del control del robot pueden implicar graves lesiones en la componente humana. Para minimizar esta posibilidad, los robots colaborativos tienen mecanismos de seguridad pasivos y activos. La integración de componentes elásticas en sus articulaciones es un mecanismo de seguridad pasivo que mejora su capacidad de absorción en caso de impacto. Los mecanismos de seguridad activos se basan en una capacidad de sensorización y reacción muy rápida para detección y reacción en caso de colisión (estos mecanismos son más caros y más sofisticados). Los actuadores en los sistemas biológicos son elásticos, por lo que a nivel de control los sistemas han evolucionado a esquemas de control que les puedan sacar el mayor partido a nivel de precisión y fuerza.

Sin embargo, los cobots son difíciles de controlar. La introducción de

componentes elásticas dificulta en gran medida su modelado y, en definitiva, su control. Por otra lado, hasta hace unos años los robots industriales tenían herramientas y software de control propietario y específico. Este hecho hacía que la adaptación de los esquemas de control entre robots llegara a ser una tarea compleja y tediosa. Actualmente, como añadido a las utilidades propietarias de cada una de las empresas, la mayoría de ellas incorporan una interfaz de comunicación estándar basada en ROS (Robot Operating System). Esto torna interesante el estudio de este tipo de software en el contexto del control robótico, dado que la adaptación a otros modelos de robots ahora es viable. El uso de técnicas de inteligencia artificial llega a ser muy útil en estos casos donde la regulación tradicional tiene dificultades, consiguiendo movimientos suaves, precisos y seguros. Además, si se consigue integrar el sistema de control en una plataforma empotrada de bajo consumo, es posible incrementar la autonomía y la eficiencia de este tipo de soluciones.

El conjunto de todos los elementos comentados anteriormente permite la introducción de robots en ambientes hasta ahora inviables. Estos pueden ser rehabilitación, ayuda a personas mayores, o tareas domésticas, además por supuesto de todas las tareas industriales que puedan ser llevadas a cabo por humanos y robots en cooperación. Todo esto hace que para los próximos años se estime un crecimiento en el mercado de los cobots de 981M\$ en 2020 a los 7,972M\$ en 2026. Usando el Google Patents vemos que ya existen aproximadamente unas 1000 con el término cobot. En definitiva, un sin fin de posibilidades que nos permitirá facilitar la vida de las personas y aumentar la seguridad en su entorno de trabajo.

1.2. Propuesta y objetivos

Este Trabajo de Fin de Grado se afronta desde un enfoque multidisciplinar, aunando uso y estudio de sistemas empotrados, robótica e inteligencia artificial, todo ello desde un enfoque inspirado en la biología. El proyecto constará de dos líneas de trabajo que confluirán en una conclusión final: ¿es posible controlar un robot colaborativo desde un sistema empotrado de manera efectiva, eficiente y en tiempo real?

Durante la primera línea de trabajo se propone el *desarrollo de un esquema de control para el robot Baxter*. Dicho sistema deberá trabajar desde una plataforma empotrada como la NVIDIA Jetson, por lo que se afrontará la implementación mediante una biblioteca de ROS adecuada para sistemas embebidos: cROS. Dado que este último es relativamente reciente y no se ha testeado demasiado, esto incluye comprobar la interconexión de cROS con las librerías estándar de ROS.

Por otro lado, la segunda línea de trabajo abordará un *estudio del ren-*

dimiento de redes neuronales biológicamente inspiradas integradas en una plataforma empotrada. Este tipo de redes son una gran herramienta para el control de robots colaborativos, dada su gran complejidad que dificulta el uso de controladores clásicos. Este estudio incluirá medidas de rendimiento y rendimiento-consumo, de manera que sea posible la comparación respecto a otras plataformas como PC estándar (además de compararlo de forma indirecta con plataformas “neuromórficas” basadas en hardware de propósito específico para sistemas neuronales.

Teniendo en cuenta ambas líneas de trabajo en conjunto, debería ser posible responder a la pregunta anteriormente planteada. Es importante aclarar que ahora estamos hablando conceptualmente de elementos que aún no hemos introducido, elementos que se verán explicados más detalladamente en capítulos posteriores.

Definida esta propuesta pasamos a definir los objetivos que se intentarán cumplir durante el desarrollo del proyecto:

- Implementación de un sistema de control para el Baxter mediante una biblioteca de funciones ligera de interfaz robótica (cROS).
- Validación de la interoperabilidad entre la interfaz robótica ligera (cROS) y las librerías estándar de ROS.
- Integración y evaluación del control en una plataforma de computación autónoma (*stand-alone*), como la familia NVIDIA Jetson.
- Integración del simulador neuronal EDLUT en la plataforma NVIDIA Jetson.
- Validación de simulación de una red neuronal con EDLUT en tiempo real en la NVIDIA Jetson.
- Comparativa del rendimiento, en términos de potencia y consumo, de EDLUT en esta plataforma frente a otras plataformas de propósito general (computador de propósito general).
- Comparativa de los datos obtenidos frente a plataformas con arquitecturas específicas para sistemas neuronales basados en impulsos.
- Responder a la pregunta: ¿es posible controlar un robot colaborativo con un sistema neuronal basado en impulsos (SNN) en un sistema empotrado de manera efectiva, eficiente y en tiempo real?

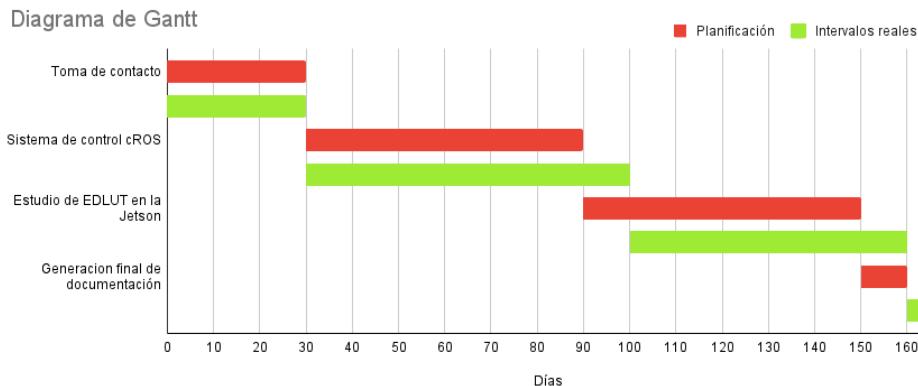


Figura 1.2: Diagrama de Gantt.

1.3. Planificación

Este proyecto se planteó de manera prematura debido a que se tenía claro a qué se quería enfocar prácticamente al comenzar el curso académico. Tras las gestiones pertinentes y la adquisición de los dispositivos necesarios, finalmente se comenzó a trabajar en este proyecto alrededor del 11 de enero de 2021. La finalización del proyecto se prevé para el 20 de junio del mismo año, aproximadamente.

El proyecto se puede separar en cuatro grandes tareas: la toma de contacto con las herramientas hardware y software, la implementación del sistema de control cROS, la integración y el estudio de EDLUT en la Jetson y, finalmente, la generación de la documentación. Por tanto, del total de 160 días que se estiman para el desarrollo, se asignará una duración aproximada a cada una de las tareas. Por la carga de trabajo que implica cada tarea, se estima que se necesitará un mes para tomar contacto con los recursos, debido a que no se tiene experiencia con ninguna de ellas. Por otro lado, se dejarán 10 días centrados en finalizar la memoria, que realmente será desarrollada poco a poco durante el resto de tareas, con la generación de informes parciales y revisión con los supervisores del Trabajo Fin de Grado. El resto del tiempo se repartirá de manera equitativa entre las dos tareas centrales del proyecto. Esta planificación resulta en el diagrama de Grantt de la Figura 1.2, que muestra la planificación inicial y el tiempo real dedicado a cada tarea. Cada una de las tareas, que son muy generales, se dividirán en subtareas una vez se aborden.

Tras finalizar el proyecto, y como se puede ver en el diagrama de Grantt, solo se ha dado un retraso destacable (de unos diez días) en la implementación del sistema de control utilizando cROS. Este retraso fue debido a que en principio se subestimó la dificultad de esta tarea, dado que se pensaba que

el traslado de una biblioteca a otra sería relativamente directo. No obstante, como se detalla en la sección 5.2, cROS tiene algunas peculiaridades que han requerido más tiempo para comprenderse. En consecuencia se abordó la última tarea de manera más exhaustiva para evitar un retraso demasiado severo, que ha acabado siendo de tan solo cinco días.

En otro orden de cosas, utilizaremos una metodología ágil como paradigma de trabajo durante el desarrollo. Se va a asumir una metodología basada en SCRUM, pero siendo flexible dado que sólo somos tres personas en el proyecto. Para las tareas de toma de contacto y generación final de la memoria no hay nada que destacar, simplemente se mantendrán reuniones quincenales en las que yo, el autor, transmitiré las dudas surgidas durante la semana y los tutores las resolverán *in situ* o, si es necesario, posteriormente después de estudiar el problema concreto. No obstante, se permitirá algo de flexibilidad en las fechas dado que sólo somos tres personas, sin fijar exactamente un día concreto y permitiendo alterar la frecuencia de estos encuentros dependiendo del avance del proyecto o del tiempo del que disponga el equipo de trabajo. Aparte de eso se ha tenido una comunicación fluida paralela por email que ha permitido avanzar de forma continuada a lo largo de cada sprint o período de desarrollo.

En la primera reunión en la que se aborde cada una de las dos tareas de desarrollo principales, se generará un *backlog* con las subtareas, las *issues*, en las que dividirá cada tarea y debatirá el contenido del primer *sprint* de trabajo. En las posteriores reuniones se comentará el avance en el *sprint* trabajado durante la quincena y se decidirán las *issues* que contendrá el siguiente *sprint*, introduciendo nuevas *issues* si así fuera necesario. Dado que se va a ser flexible con la metodología, y dado que no tendrá efectos adversos en otros miembros del equipo de trabajo, se permitirá alargar un *sprint* si es necesario o cualquier otra alteración que sea pertinente, siempre con el visto bueno de las tres partes. Además, a lo largo del proyecto se mantendrán reuniones con personal especializado del grupo de investigación en distintas partes del proyecto con objetivo de clarificar aspectos relacionados con su campo de aplicación específico.

Para cada una de las dos tareas se utilizará un tablero de Trello para gestionar las *issues* y el flujo de trabajo. Se puede ver un ejemplo en la Figura 1.3. Cada tablero constará de cuatro listas:

- *Backlog*. Mantendrá todas las *issues* que aún no han comenzado a desarrollarse y que no se pretenden trabajar en el *sprint* actual, ordenadas por orden de prioridad.
- *To Do*. Estarán listadas las *issues* que se van a trabajar en el *sprint* actual, pero que aún no han comenzado a desarrollarse.
- *Developing*. Tendrá listadas las *issues* que se están desarrollando y

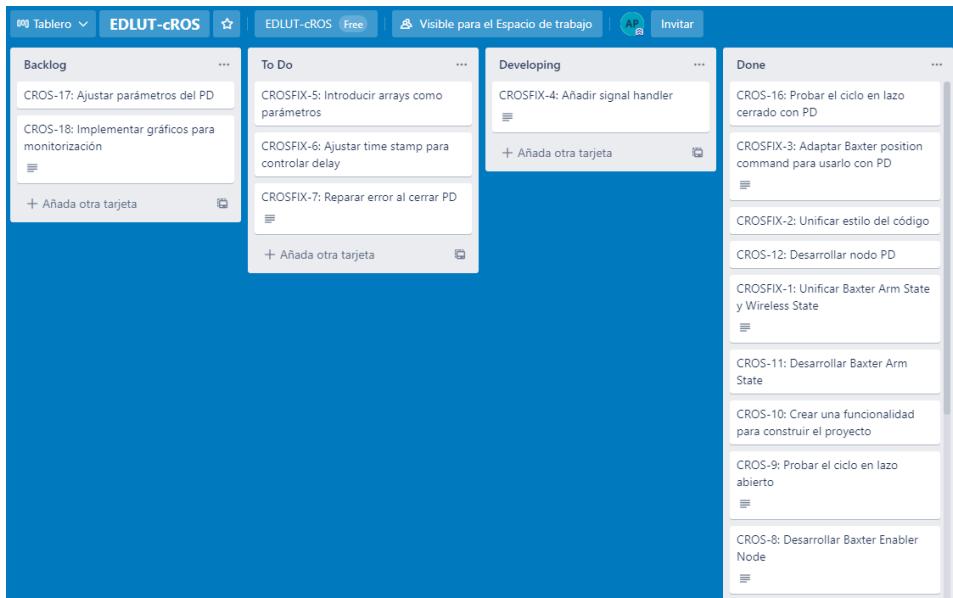


Figura 1.3: Tablero de Trello en un instante de trabajo

están sin finalizar.

- *Done*. Acumulará las *issues* ya finalizadas.

Las *issues* tendrán nombres codificados y fáciles de reconocer. Usaremos el formato **XXXX-n**, donde **XXXX** será **CROS** para las *issues* relacionadas con el desarrollo del sistema de control, **CROSFIX** para los arreglos que vayan siendo necesarios durante el desarrollo, **JETSON** para el estudio de EDLUT en la Jetson y **JETSONFIX** para las reparaciones de los problemas ocurridos en esta tarea. La **n** será la numeración, que se irá asignando desde el 1 hacia delante para cada una de las tres etiquetas por orden de creación, usando el 0 para la tareas pre-implementación.

Al ser un equipo pequeño, no tiene sentido utilizar *Planning Poker* para acordar la dificultad de cada *issue*. Se optará directamente por establecer en cada reunión qué *issues* se podrían llevar a cabo en el tiempo asignado al siguiente *sprint*.

Una vez finalizado el proyecto, podemos exponer tanto la lista de *issues* como los *sprints* que se han definido durante el desarrollo. Esta información está presente en las tablas 1.1, de *issues* relacionadas con el sistema de control, 1.2, de *issues* relacionadas con el estudio de EDLUT en la Jetson y 1.3 de *sprints*.

<i>Issue</i>	Descripción
CROS-0	Instalar ROS y Gazebo
CROS-1	Instalar cROS
CROS-2	Crear el proyecto EDLUT-cROS
CROS-3	Definir mensajes
CROS-4	Plantear adaptación de la lógica de clases a cROS
CROS-5	Desarrollar Wireless State Node
CROS-6	Desarrollar Q Trajectory Node
CROS-7	Desarrollar Baxter Position Command
CROS-8	Desarrollar Baxter Enabler Node
CROS-9	Probar el ciclo en lazo abierto
CROS-10	Crear una funcionalidad para construir el proyecto
CROS-11	Desarrollar Baxter Arm State
CROS-12	Desarrollar nodo PD
CROS-16	Probar el ciclo en lazo cerrado con PD
CROS-17	Ajustar parámetros del PD
CROS-18	Implementar gráficos para monitorización
CROS-19	Generar gráficos e interpretarlos
CROSFIX-1	Unificar Baxter Arm State y Wireless State
CROSFIX-2	Unificar estilo del código
CROSFIX-3	Adaptar Baxter position command para usarlo con PD
CROSFIX-4	Añadir signal handler
CROSFIX-5	Introducir arrays como parámetros
CROSFIX-6	Ajustar time stamp para controlar delay
CROSFIX-7	Reparar error al cerrar PD

Cuadro 1.1: Lista de *issues* relacionadas con el sistema de control.

<i>Issue</i>	Descripción
JETSON-0	Instalar SO y herramientas
JETSON-1	Instalar y probar EDLUT
JETSON-2	Definir redes y pesos a usar
JETSON-3	Implementar scripts para generar inputs
JETSON-4	Generar inputs
JETSON-5	Generar tablas para ED
JETSON-6	Hacer pruebas de rendimiento en la Jetson
JETSON-7	Obtener consumo de la Jetson
JETSON-8	Hacer pruebas en el resto de las plataformas
JETSON-9	Realizar comparativa con el resto de plataformas
JETSON-10	Comparar la eficiencia con SpiNNaker
JETSONFIX-1	Comprobar uso de TD-GPU
JETSONFIX-2	Revisar spikes como métrica

Cuadro 1.2: Lista de *issues* relacionadas con el estudio de EDLUT en la Jetson.

<i>Sprint</i>	<i>Issues</i>
<i>Sprint 1</i>	CROS-0, CROS-1, CROS-2, CROS-3
<i>Sprint 2</i>	CROS-4, CROS-5, CROS-6
<i>Sprint 3</i>	CROS-7, CROS-8, CROS-9
<i>Sprint 4</i>	CROS-10, CROS-11, CROSFIX-1
<i>Sprint 5</i>	CROS-12, CROSFIX-2, CROSFIX-3, CROS-16
<i>Sprint 6</i>	CROSFIX-4, CROSFIX-5, CROSFIX-6, CROSFIX-7
<i>Sprint 7</i>	CROS-17, CROS-18, CROS-19
<i>Sprint 8</i>	JETSON-0, JETSON-1
<i>Sprint 9</i>	JETSON-2, JETSON-3, JETSON-4
<i>Sprint 10</i>	JETSONFIX-1, JETSON-5
<i>Sprint 11</i>	JETSON-6, JETSON-7
<i>Sprint 12</i>	JETSON-8, JETSON-9
<i>Sprint 13</i>	JETSONFIX-2, JETSON-10

Cuadro 1.3: *Sprints* definidos durante el proyecto.

Capítulo 2

Dominio y estado del arte

Rescatando la famosa cita de Steve Jobs: “no es posible conectar los puntos mirando hacia adelante, solo puedes hacerlo mirando hacia atrás”. Es por ello que antes de comenzar vamos a definir los puntos, los campos de estudio cuyo progreso nos permiten el planteamiento de este proyecto, que se irán conectando a lo largo del desarrollo del mismo. Durante este proceso, cumpliendo este pretexto, la correcta selección de elementos de los siguientes campos de estudio nos permitirá aproximarnos a los objetivos expuestos en el capítulo anterior desde el enfoque correcto.

2.1. Plataformas de procesamiento

Podemos entender una plataforma de procesamiento como cualquier arquitectura de computación capaz de procesar datos. A la hora de controlar un robot, y dependiendo de la lógica que usemos para ello, es primordial disponer de una plataforma de procesamiento adecuada que calcule las trayectorias y/o fuerzas de las articulaciones en tiempo real.

La gran mayoría de unidades de control de robots comerciales hoy en día están basados en PCs, ordenadores personales de propósito general, que corren el software propietario de la empresa que los ha creado. Por otro lado, si el control se realiza de manera externa, en cada caso se hará uso de la plataforma más adecuada según el objetivo y las circunstancias pertinentes. Para los usos habituales se suelen utilizar también PC, en todo caso más potentes que las propias unidades de control, dotados de hardware de gama alta como procesadores multinúcleo o tarjetas gráficas dedicadas dependiendo de las necesidades concretas de cada caso.

Por otro lado, hay circunstancias en las que se requiere otro tipo de hardware. Durante este proyecto se pretende explorar su integración en plataformas de procesamiento que puedan ser alojadas en el propio robot, de

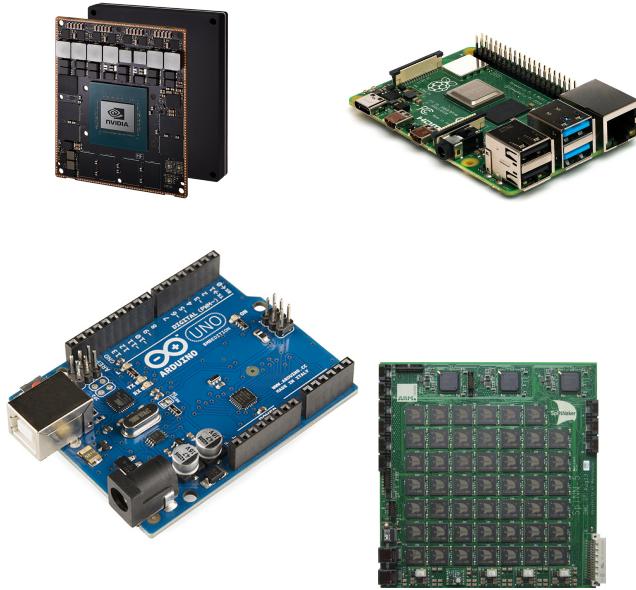


Figura 2.1: De izquierda a derecha y de arriba a abajo: Nvidia Jetson AGX Xavier, Raspberry Pi 4, Arduino Uno y SpiNNaker. Fuentes: <https://www.nvidia.com/es-es/autonomous-machines/jetson-store/>, https://es.wikipedia.org/wiki/Raspberry_Pi, https://commons.wikimedia.org/wiki/File:Arduino_Uino_-_R3.jpg, https://www.researchgate.net/figure/A-SpiNNaker-board-with-48-chips-SpiNN-5_fig1_301559712.

manera que esta sea portátil y de bajo consumo. Desde hace algunos años se ha incrementado en gran medida el uso de plataformas de tamaño reducido debido a su bajo coste y a la creciente importancia del internet de las cosas. Un sistema embebido o empotrado es un computador diseñado para resolver exclusivamente algunas funciones específicas, constando de los elementos de computación necesarios para resolvérlas incrustadas en una sola placa y, en general, de pequeñas dimensiones. Probablemente uno de los sistemas empotrados más extendidos sea Arduino[4], el cual es una placa de dimensiones muy reducidas que consta de un microcontrolador, RAM, memoria flash y puertos de comunicación para entrada/salida. Por supuesto, este tipo de dispositivos consume apenas unos vatios de potencia y son perfectos para pequeños proyectos básicos de automatización. Cuando hablamos sin embargo de aplicaciones más específicas y especializadas, como suelen ser sistemas aeroespaciales, de imágenes médicas o para visión por computador, existen plataformas aún más específicas como pueden ser los sistemas basados en FPGAs[26][8].

Estos son sólo algunos ejemplos, pero existen muchos más sistemas diseñados exclusivamente para fines concretos, ya sean más generalistas y flexi-

bles o más aplicados y eficientes. Este es un compromiso que debemos tener en cuenta a la hora de elegir una plataforma en la que desarrollar nuestro software. Las plataformas más especializadas tienen la ventaja de ser extremadamente eficientes, dado que están específicamente diseñadas para el campo donde van a ser aplicadas. No obstante, esto deriva en una desventaja, y es que requiere implementar la lógica que queremos llevar a cabo de manera específica para esa plataforma, perdiendo la capacidad de hacer uso de ella en otras arquitecturas de manera directa.

Tomando en cuenta este hecho se han presentado otras propuestas de sistemas empotrados manteniendo un modelo de von Neumann[49], el cual caracteriza a los computadores de propósito general. Quizá el caso más famoso de sistema empotrado de propósito general, aunque no es exactamente un modelo de von Neumann, es la plataforma Raspberry Pi 4[18]. Estos mini-PC son básicamente computadores de dimensiones reducidas e ínfimo consumo de energía en los que podemos instalar sistemas operativos generalistas, generalmente basados en Linux (cosa que no podemos hacer en el resto de sistemas empotrados anteriormente comentados). Debido a estas características, este tipo de plataformas tienen unas prestaciones limitadas para realizar procesamiento pesado como algoritmos de visión por computador u otras técnicas de inteligencia artificial. Sin embargo, la llegada de arquitecturas de procesadores como ARM, la miniaturización de los chips o el abaratamiento de los dispositivos de almacenamiento sólidos con interfaz M2 basados en NVMe, entre otros, ha permitido el desarrollo de plataformas de procesamiento de altas prestaciones embebidas en placas de tamaño reducido. Una de las gamas más populares de este tipo de dispositivos es la familia Jetson de NVIDIA[11].

Por el momento existen cuatro modelos de NVIDIA Jetson: Nano, TX2, Xavier NX y AGX Xavier. La propia marca remarca que están creadas para inteligencia artificial perimetral y máquinas autónomas. Al contrario que la IA en la nube, basada en enormes clusters de procesamiento, esta se encuentra en el perímetro del sistema del que forma parte, en el cual procesa toda la carga computacional. Cambiamos los centros de procesamiento de grandes dimensiones en la nube por plataformas de altas prestaciones presentes en el entorno de trabajo. De los cuatro modelos, nos hemos decantado por la AGX Xavier, dado que requerimos de la mayor potencia computacional al menor coste energético y esta opción, con un TDP (*thermal design power*, potencia de diseño térmico, indica la cantidad máxima de calor que se espera que un componente produzca en un escenario de uso intenso) de 30 W y 32 TOPS (*tera operations per second*, trillones de operaciones por segundo)[11], es la versión más potente y eficiente.

2.2. Robots en la industria

Durante la introducción y la motivación ya se han comentado los aspectos más generales del estado del arte referente a la interacción humano-robot. No obstante, dedicaremos esta sección a realizar una breve introducción tanto a la historia de los robots como a las opciones de cobots que existen actualmente.

La idea de robot surge principalmente en obras de ficción, estando presentes la propia obra de Čapek comentada en la introducción, Metrópolis (1927)[15], La Guerra de las Galaxias (1977)[14] o las actuales Ex-machina[13] y Yo, robot[16][6] entre ellos. Prácticamente en todos estos ejemplos se plantea el concepto de robot como un dispositivo humanoide que se rebela contra nosotros, sus creadores. No obstante, la historia y la tendencia actual no pueden darse en una situación más alejada de esta distopía. En este contexto podríamos destacar las tres (o cuatro) leyes de la robótica de Isaac Asimov[5] que, aunque nacidas también en el mundo de la ciencia ficción, han afectado en mayor o menor medida a la ética referente al mundo de la inteligencia artificial y los robots:

1. Un robot no hará daño a un ser humano ni permitirá que un ser humano sufra daño.
2. Un robot debe cumplir las órdenes dadas por los seres humanos, exceptuando aquellas que incumplan la primera ley.
3. Un robot debe protegerse a sí mismo siempre que dicha protección no incumpla la Primera o Segunda Ley.
4. Ley Cero: Un robot no puede dañar a la humanidad o permitir que la humanidad sufra daños.

Estas no son leyes al uso, si no más bien un punto de partida lógico con el que entender qué papel deberían tener los robots para con la humanidad. Debemos entenderlos como una ayuda, una liberación de trabajos pesados y repetitivos, como dispositivos que aumentarán nuestra comodidad y nuestra calidad de vida.

Los precursores de los robots son básicamente ingenios que automatizan tareas cotidianas o imitan el movimiento de seres vivos, tales como relojes o maquetas. Sin embargo, bajo este precepto prácticamente todos los objetos actuales mínimamente complejos podrían ser denominados robots. Una definición actual más acertada podría venir dada por los términos que le asigna la Organización Internacional para la Estandarización(ISO): manipulador, multifuncional, reprogramable, varios grados de libertad, manipulación según trayectorias programadas. Estos términos nos acercan a lo que

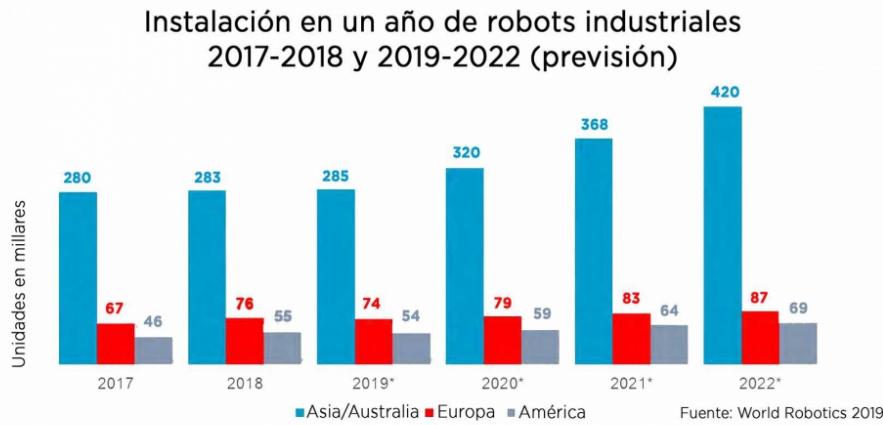


Figura 2.2: Número de robots instalados en un año. Fuente: [36]

hoy en día entendemos como un robot, o más concretamente un robot industrial. Existen varios tipos de robots clasificados a partir de distintos criterios, pero los más usados en el mundo de la industria son los brazos robóticos. Estos toman su nombre debido a que la organización de sus eslabones está inspirada en un brazo antropomórfico, conformados por (al menos) tres articulaciones de rotación y otras tres de torsión que lo dotan de la capacidad de moverse libremente por un espacio acotado en tres dimensiones. De esta manera son muy útiles para realizar tareas de *pick and place*, atornillado o soldado. Su uso se ha visto multiplicado en los últimos años (Figura 2.2), debido a su fiabilidad, eficiencia y bajo coste, lo que hace ahorrar grandes cantidades de dinero a la industria.

Impulsado por el auge de los robots industriales, en los últimos años ha surgido el campo de los cobots que, como ya hemos comentado, intenta resolver el problema de la colaboración entre robots y humanos. Son varias las empresas que han desarrollado su propio modelo de cobot que, con las mismas características que un robot industrial, permite realizar las tareas correspondientes a un brazo robótico en presencia y colaboración de un trabajador.

Universal Robots fue la primera en introducir este tipo de robots en el mercado, y a día de hoy la que más éxito ha recabado por el momento. A ella se han ido uniendo poco a poco marcas como ABB, Kuka o Rethink Robotics, y todas ellas han optado por el mismo modelo de robot: dimensiones reducidas, un mínimo de seis articulaciones (3 rotación y 3 torsión), poca capacidad de carga (4kg aprox.), uso de transmisiones elásticas en los

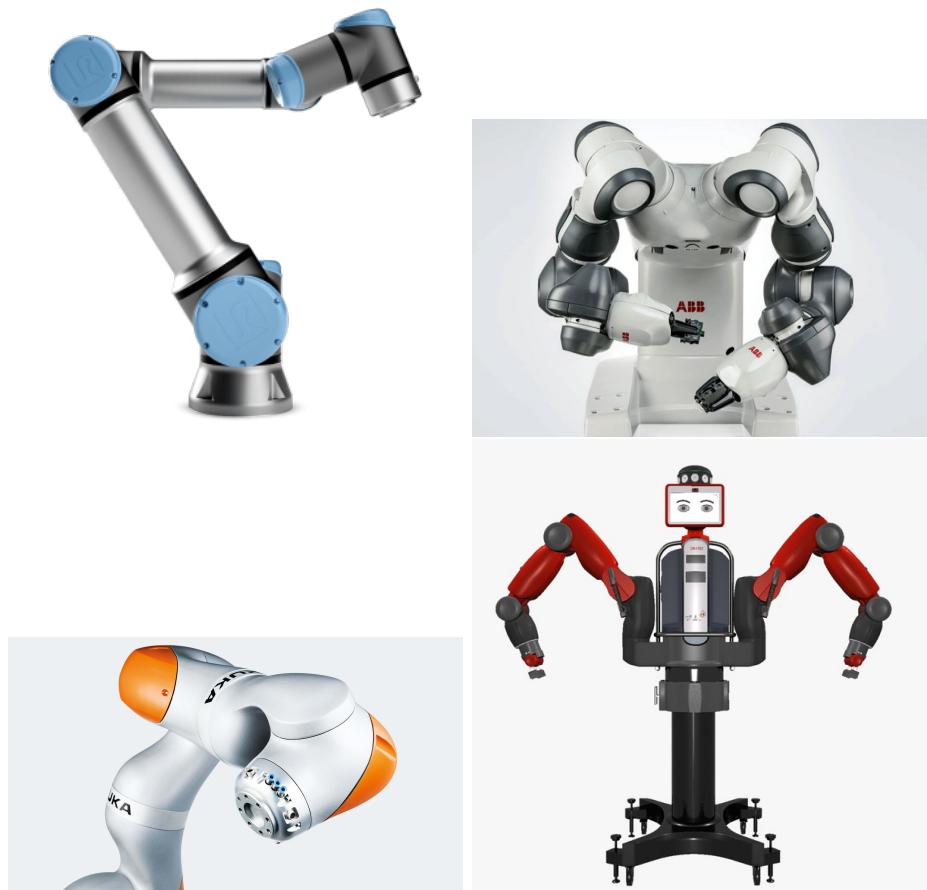


Figura 2.3: De izquierda a derecha y de arriba a abajo: Universal Robot UR16e, ABB IRB 14000 YUMI, Kuka LBR iiwa y Rethink Robotics Baxter.
Fuentes: <https://www.universal-robots.com/es/productos/>, <https://www.tecnoplcc.com/yumi-robot-colaborativo-de-abb/>, <https://www.kuka.com/es-es/productos-servicios/sistemas-de-robot/robot-industrial/lbr-iiwa> y <https://cobotsguide.com/2016/06/rethink-robotics-baxter/>

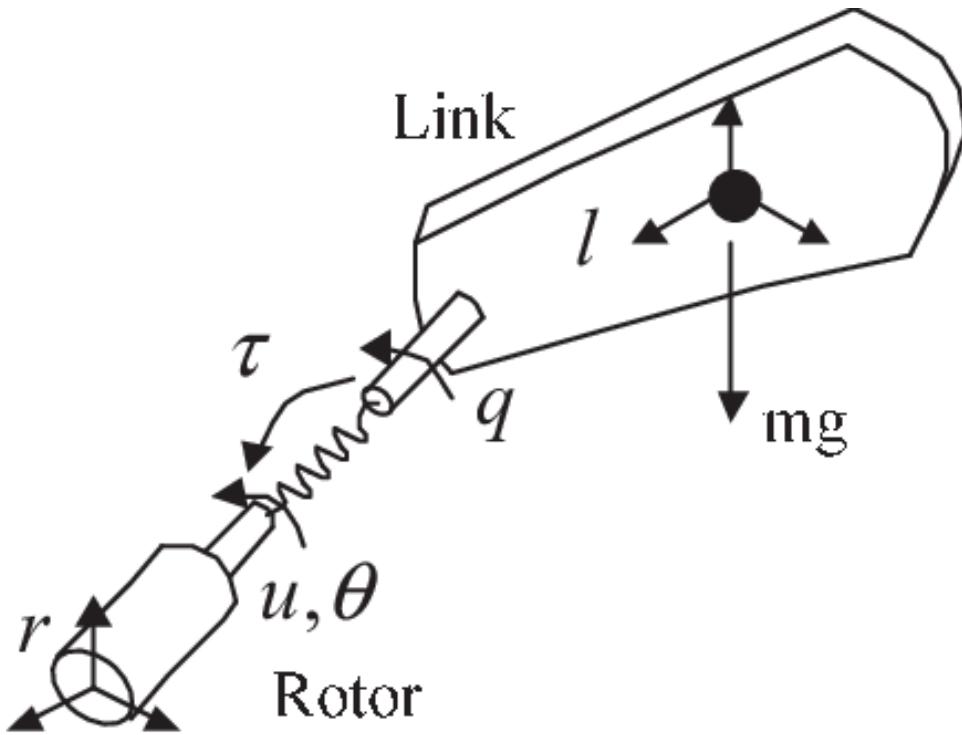


Figura 2.4: Transmisión elástica. Fuente: [30].

actuadores y par de fuerzas máximo bajo. La Figura 2.3 muestra cuatro ejemplos. El hecho de que se repitan las características antes comentadas tiene un sentido claro para cada una de ellas. Con las seis articulaciones conseguimos libertad de movimiento en tres dimensiones, dado que con seis grados de libertad podemos resolver un sistema de seis ecuaciones y, por tanto, obtener tres valores de posición (x , y y z) para la traslación en las tres dimensiones del espacio, y tres más de rotación (α , β y γ), guñada (*yaw*), cabeceo (*pitch*) y alabeo (*roll*), para definir la orientación. Las transmisiones elásticas (Figura 2.4) permiten absorber parte de la fuerza de un posible impacto, mientras que el uso de pares de fuerzas bajos aporta seguridad dado que el brazo no puede coger inercia. Por otro lado, la poca capacidad de carga y el tamaño reducido no son si no una consecuencia de las características de seguridad.

Como conclusión, dadas sus características y su precio, que varía en el rango de 15000-30000€, es una alternativa perfecta a los robots industriales clásicos para pequeñas y medianas empresas. Aunque parezcan caros, su valor es una fracción del de las opciones tradicionales, lo que junto a que pueden compartir espacio con trabajadores humanos (ahorro en espacio) permite a dichas compañías automatizar tareas, aumentar su rendimiento

de producción y mejorar la calidad de vida de sus empleados a un coste reducido.

Por último, como trataremos de manera central en este proyecto, el control preciso de este tipo de robots es muy complejo, principalmente debido a la introducción de elementos elásticos y a la utilización de sistema de control que minimicen los riesgos en la interacción con humanos. Si este campo avanza lo suficiente es posible que los robots colaborativos sean introducidos en campos tan interesantes como cuidado de personas mayores, fisioterapia o rehabilitación, haciendo posible tomar nuevos paradigmas de trabajo en estos campos. Con el objetivo de estudiar el control de los cobots, nos centraremos principalmente en el robot Baxter, que está disponible en el laboratorio de robótica de la Universidad de Granada y presente en el simulador Gazebo. Por tanto lo escogemos dada la facilidad con la que podemos hacer pruebas con él y porque tenemos disponibles otros experimentos que servirán como base.

2.3. Frameworks de control para robots

Hace algunos años, antes de que existiera ROS, no existía ningún software genérico para controlar robots. Esto hacía que todo el trabajo realizado para implementar un sistema de control para un robot específico no se podía extender a otros modelos de robots. Aún a día de hoy, todas las herramientas que proporcionan las distintas marcas para programar sus robots son propietarias. Algunas de ellas se basan en guiado pasivo, lo cual implica tener que indicar al robot la trayectoria a seguir en el espacio de trabajo real. Esta lógica de programación es solo viable para tareas sencillas como un *pick and place* o derivados. Sin embargo, si queremos realizar un control más complejo, la mayoría de los robots incorporan una interfaz ROS para poder enviar comandos y recibir información en tiempo real, exteriorizando el control desde la unidad de control del robot a una plataforma de cómputo externa de las que se han comentado en apartados anteriores.

ROS[43][51], Robot Operating System, es un framework de código abierto diseñado para la implementación de software robótico. Es lo que se conoce como un middleware, abstrayendo las comunicaciones entre las aplicaciones presentes en un sistema distribuido de paso de mensajes. Presenta un modelo de grafos en el que cada aplicación es un nodo ROS (*ROS NODE*, en adelante nombrado como nodo) y estos se comunican entre sí mediante topics ROS (*ROS TOPIC*, literalmente “temas”, en adelante nombrado como topic), todo ello dentro de una topología centralizada, ya que todos los nodos ROS necesitan comunicarse con un nodo especial que registra los flujos de información para leer o escribir datos conocido como el nodo *ROS Master*, en adelante conocido como Master o nodo maestro(ver Figura 2.5).

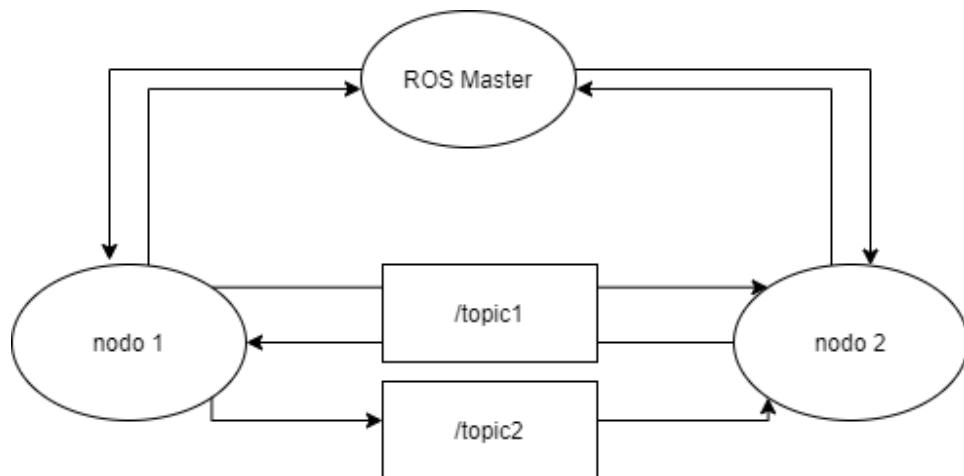


Figura 2.5: Arquitectura de una red ROS

No es el objetivo de este proyecto realizar un estudio profundo sobre ROS, pero si es importante conocer los conceptos clave para entender la implementación del ciclo robot, por tanto este apartado hará un repaso de ellos.

Como ya hemos dicho, un nodo ROS se podría considerar como una aplicación que conforma un módulo de un sistema distribuido. Cada nodo puede a su vez procesar/generar por varios flujos de entrada/salida de mensajes de determinados topics. Un topic en esencia se puede considerar como un flujo de mensajes los cuales pertenecen a un contexto específico y están conformados por una estructura de datos determinada, y que puede ser leído y escrito desde diferentes nodos. Por ejemplo, un topic *Temperatura del aire* en una estación meteorológica podría ser un flujo de mensajes en los que uno o varios sensores escribirían valores de temperatura y uno o varios monitores podrían recoger estos valores para mostrarlos al usuario. Dicho flujo estaría formado por mensajes con una estructura previamente definida que podría ser, por ejemplo, un valor real que represente la temperatura en grados centígrados.

Estos mensajes pueden ser leídos o escritos de los topics por distintas entidades presentes en los nodos y que pueden ser una u otra según el comportamiento esperado. Las dos más importantes son:

- **Publisher:** Un publisher, como su propio nombre indica, se encarga de enviar mensajes a un topic específico.
- **Subscriber:** Un subscriber se encarga de recibir mensajes de un topic específico.

El modelo publisher-subscriber es el más interesante para el contexto del proyecto y el que más se va a tratar a lo largo de él, pero para otras aplicaciones es posible que un modelo de paso de mensajes en una dirección sea insuficiente. Por ello ROS también proporciona un modelo de cliente-servidor con el que puede implementarse un envío de mensajes en dos direcciones, el cliente envía una petición y el servidor produce una respuesta. No obstante no se entrará en más detalles dado que sólo se va a utilizar el modelo publisher-subscriber.

Cabe destacar que, dada la manera en que ocurren los pasos de mensajes en ROS, es posible que más de un publisher escriba y/o más de un subscriber lea del mismo topic, así como que un nodo escriba/lea varios topics. Por tanto un nodo ROS puede leer, escribir, desglosar, resumir, multiplexar, demultiplexar o redundar información, entre otros.

Por último debemos destacar que los publishers/subscribers deben registrarse en el Master para poder intercambiar mensajes. Esto se hace mediante el protocolo XMLRPC, basado en HTTP. Tras ello, los distintos publishers/subscribers podrán comunicarse entre sí tanto por TCP como por UDP, dependiendo del propósito y el contexto de la comunicación (ver Figura 2.6).

Descrito este punto quedan introducidos los conceptos más importantes sobre la arquitectura de red con la que se va a trabajar. A modo de resumen: cada uno de los módulos de una aplicación conforma un nodo en una red con estructura gráfica centralizada. A su vez, cada uno de los nodos puede estar formado por varios publishers/subscribers que intercambian mensajes con otros módulos a través de topics. El Master, que centraliza la red, gestiona el registro de publishers/subscribers, aunque una vez registrados estos se comunican entre sí de manera independiente.

Toda esta información se puede consultar en mucha más profundidad en la wiki oficial de ROS[45].

Durante este proyecto se va a usar cROS, una implementación de ROS ligera y más orientada al *plug and play*. Se hablará de ella más adelante.

2.4. Entornos de simulación

Por razones de eficacia, y dada la situación actual, es conveniente realizar los experimentos utilizando un robot simulado en lugar del robot real. Salvando las distancias y las imprecisiones que implican utilizar un simulador, el robot simulado es equivalente al real para las comprobaciones que queremos realizar. Además, dicha posibilidad permite realizar los experimentos sin la necesidad de desplazarse al laboratorio, lo cual agiliza el desarrollo y evita conflictos con el flujo de trabajo de otras personas que estén experimentando con el robot real.

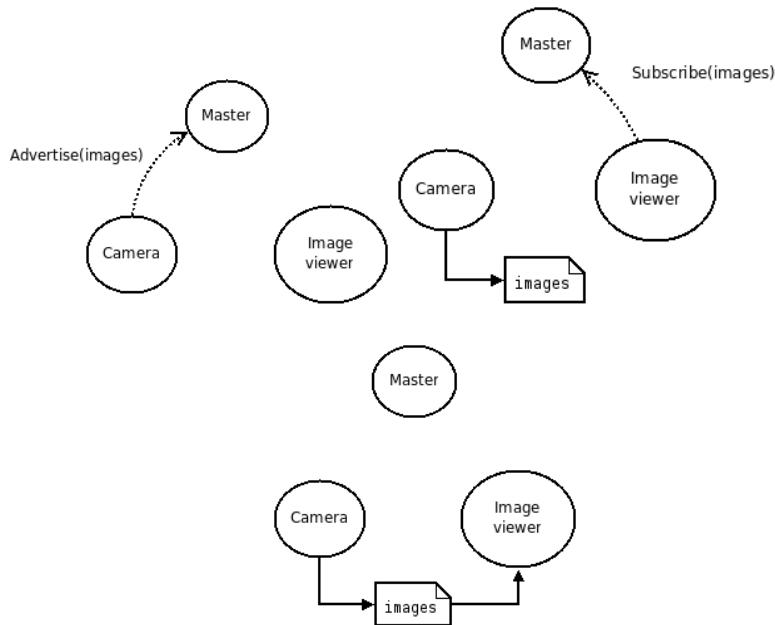


Figura 2.6: Ejemplo de comunicación a través de ROS. Fuente: <http://wiki.ros.org/Master>

Existen múltiples simuladores orientados a robótica disponibles para su uso[52], aunque la principal característica que buscamos es que sea lo más realista posible y que contenga el modelo del robot Baxter para poder utilizarlo directamente. Los más importantes son:

- RoboDK[27]. Interfaz amigable y amplio abanico de robots disponibles. Puede ser usado en una gran variedad de campos e la robótica. Su mayor problema es que es un software propietario(RoboDK Inc.) y, ante las existencia de alternativas, queda descartado.
- Sim Spark[47]. Muy usado en investigación y de código abierto. Desarrollado por la Koblenz and Landau University en Alemania. Aunque se considera un simulador genérico, está más enfocado a sistema multiagente.
- NRP (Neurorobotics Platform)[25]. El más interesante desde una vista general del proyecto, ya que además de a robótica está orientado a neurociencia, ambos temas ejes centrales de este proyecto. Está basado en Gazebo. Quizá el menos rodado en la comunidad menos experta.
- Gazebo[17]. El simulador más usado en el mundo de la robótica. Ha sido usado en varias competiciones y mantiene una comunidad muy activa, además suele estar relacionado con la comunidad ROS. Rethink

Robotics publicó en su momento el modelo de Baxter y estaba soterrado hasta hace un tiempo.

- Webots[31]. Gratuito y de código abierto. Usado en ámbitos industriales y de investigación. Interfaz web. Dispone de algunos modelos de robots y se pueden generar aportando el modelo visual y físico. El Baxter no está disponible.

Hay algunos otros simuladores orientados a aplicaciones concretas como automoción o robots móviles, aunque para este proyecto nos interesa un simulador genérico. Esta es solo una pequeña lista con los más accesibles, existen muchos otros simuladores, eso sí menos sofisticados y con menos posibilidades.

La elección final será Gazebo, dado que es un simulador ampliamente usado en robótica, Baxter está disponible y Rethink Robotics proporciona un tutorial de instalación. NRP sería otra posibilidad muy atractiva, sin embargo requeriría un tiempo para estudiar su uso e instalarlo en las plataformas correspondientes, tiempo que este proyecto pretende dedicar a otras tareas.

2.5. Paradigmas de control

El principal cometido de nuestro sistema de control es hacer seguir al robot una trayectoria previamente generada. En este caso, el comportamiento deseado para el Baxter es que el movimiento de uno de sus brazos forme una figura determinada de manera cíclica.

2.5.1. Control prealimentado

El esquema de control más simple se conoce como control prealimentado. Este se limita simplemente a enviar el comando correspondiente, especificado en la trayectoria generada, a cada una de las articulaciones (Figura 2.7). Es el paradigma de control más sencillo dado que no requiere ningún cálculo adicional a la generación de la trayectoria, tarea que puede realizarse previamente. Dicho comando, presente en la definición de la trayectoria, será traducido y distribuido por la unidad de control hacia los motores de las articulaciones. Cada motor moverá su articulación correspondiente, generalmente a través de un sistema de transmisión.

El principal problema de este esquema de control es que no existe información del estado de las articulaciones en el momento de generar los comandos motores, es decir, no contempla realimentación. Los esquemas realimentados son ampliamente usados en la industria, proporcionando un

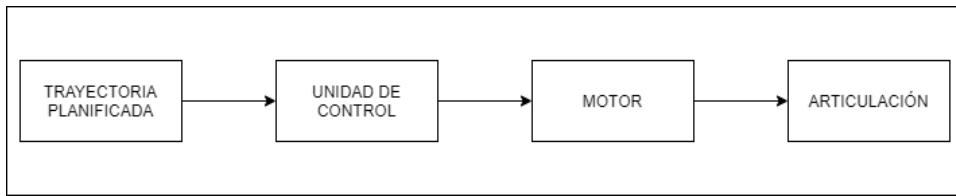


Figura 2.7: Sistema de control prealimentado.

buen rendimiento a un coste computacional aceptable. Para conseguir realimentación debemos partir del sistema de control prealimentado y añadirle un flujo de información desde cada una de las articulaciones (mediante sensores como acelerómetros o encoders) hasta la unidad de control, la cual deberá generar los comandos teniendo en cuenta la trayectoria deseada y el estado actual de las articulaciones. A esta tarea que realiza la unidad de control la llamaremos regulación.

2.5.2. Control realimentado: controlador PID

El controlador PID es usado ampliamente en la industria, desde la automatización de maquinaria industrial hasta la estabilización del vuelo de un dron. Su control se basa en el cálculo de tres componentes: la P(Proporcional), la I(Integral) y la D(Diferencial), y se puede expresar matemáticamente mediante la expresión 2.1, donde $P(t)$ es la salida del controlador para el instante t , $P(0)$ es el valor de salida cuando el error es 0, $E(t)$ es error en el instante t y K_P , K_I y K_D son las constantes que se fijarán en función de la necesidad y del fin específico, en definitiva las que permiten el ajuste del controlador. El comportamiento de este sistema se puede consultar gráficamente en la Figura 2.8.

$$P(t) = K_P E(t) + K_I \int_0^t E(t)dt + K_D \frac{dE(t)}{dt} + P(0) \quad (2.1)$$

La componente P es la relación proporcional entre la salida del controlador y el error. Por tanto K_P es la constante que se debe establecer pensando en el comportamiento deseado del controlador ante un error entre el valor consigna y el real. Puede ser positiva o negativa dependiendo de si la acción del controlador debiera ser directa o inversa. Un valor absoluto de P alto hará que las reacciones sean más rápidas pero más bruscas, mientras que valores más pequeños darán lugar a una corrección más suave, pero también más lenta. Esta constante corregirá completamente el error entre la posición actual y la deseada siempre que no exista una perturbación externa (caso ideal).

Por otro lado, el trabajo de la componente integral es corregir el llamado error de *offset*, el cual se presenta en algunas ocasiones en las que el control proporcional se estabiliza por debajo del valor consigna y nunca llega a alcanzarlo. Esto ocurre en presencia de perturbaciones externas. Tiene en cuenta la evolución del error desde el inicio del experimento. Un valor correcto de la constante I corregirá el error de *offset* presente en la regulación de P. A medida que aumentemos dicho valor, el tiempo de corrección del error de *offset* será menor, pero un valor demasiado alto puede provocar inestabilidad en forma de oscilaciones. Entiéndase como una oscilación que el valor real “baile” alrededor del valor consigna.

Finalmente, la componente diferencial introduce la información de cómo cambia el error respecto al tiempo. Con ella el controlador puede tener en cuenta retardos entre las correcciones que aplica y los efectos de estas correcciones en el sistema. Si introducimos una constante D apropiada en un regulador PI podemos conseguir suavizar las oscilaciones producidas por la I. Un valor demasiado bajo no reducirá suficientemente las oscilaciones, sin embargo un valor demasiado alto resultará en un comportamiento erróneo por parte del regulador.

Combinando estas tres componentes como hemos comentado anteriormente conseguiremos corregir el error de posición con un equilibrio entre tiempo de respuesta, ausencia de oscilaciones y coste computacional asequible. De esta manera podemos conseguir un control fiable en tiempo real.

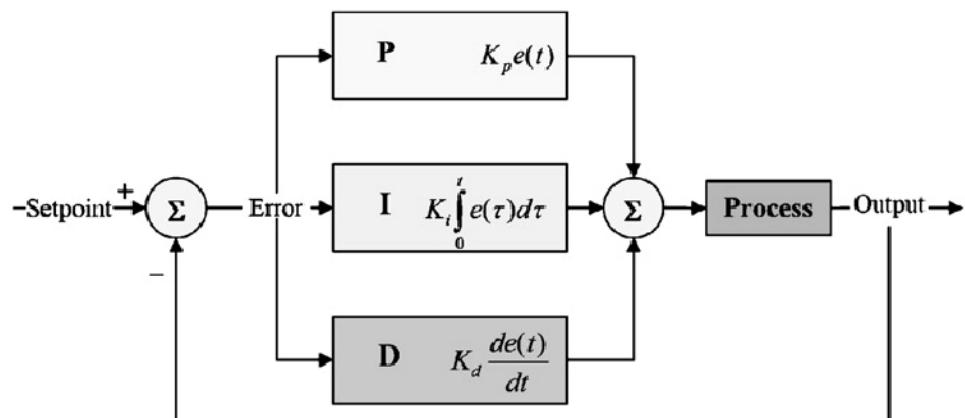


Figura 2.8: Ciclo de control genérico en lazo cerrado con controlador PID.
 Fuente: https://www.researchgate.net/figure/A-generic-closed-loop-process-control-system-with-PID-controller_fig1_242727578

2.5.3. Otros esquemas de control

Existen otros esquemas de control que, o bien pretenden mejorar el controlador PID, o bien proporcionar una alternativa en campos en los que este no sea la mejor opción. Uno de ellos, por ejemplo, es el control que combina prealimentación y realimentación, que es básicamente un control realimentado al que se le inyecta la información consigna de manera cruda después del regulador. De esta manera mejoramos la respuesta, pero también aumentamos la dificultad del ajuste del regulador.

El uso de inteligencia artificial parece aportar también soluciones viables para la regulación de sistemas de control. Una de sus utilidades es ajustar los parámetros de, por ejemplo, un controlador PID. Este ajuste es una tarea compleja y puede ser crítica en algunas aplicaciones. Por tanto, el uso de inteligencia artificial para automatizar esta labor simplifica mucho la aplicación de estas técnicas de regulación. Otro enfoque desde este área de conocimiento es el uso de algoritmos de inteligencia artificial directamente como reguladores. Este es un campo de estudio que, aunque amplio y complejo, está en auge en la actualidad debido a la gran cantidad soluciones que aporta.

2.6. Redes neuronales artificiales

Las redes neuronales artificiales han sido estudiadas en el marco de la inteligencia artificial desde hace varias décadas. Sin embargo, es durante los últimos años cuando, impulsadas por la creciente potencia computacional de las plataformas más modernas, la cantidad de datos disponibles en distintos ámbitos y las nuevas arquitecturas de red, están obteniendo buenos resultados en campos de aplicación que embarcan varias disciplinas. Los modelos de redes neuronales se basan en imitar las estructuras neuronales presentes en el sistema nervioso, donde un conjunto de nodos interconectados entre sí, las neuronas, propagan información de unos a otros recibiendo datos, traduciéndolos, y enviando el resultado a otros.

Deep learning (aprendizaje profundo), *convolutional neural networks* (redes neuronales convolucionales) o *recurrent neural networks* (redes neuronales recurrentes) son algunos términos referentes a las diferentes tecnologías usadas hoy en día. Aunque el público general aún no está familiarizado con este tema, sin lugar a dudas será uno de los ejes principales del progreso tecnológico durante las siguientes décadas. Afortunadamente, existen varias referencias a nivel divulgativo por las que empezar.

La predicción de estructuras proteicas por parte de OpenAI ha sido una de las recientes revoluciones en el campo de la biología computacional[29]. El uso de redes neuronales por parte de plataformas de contenido bajo de-

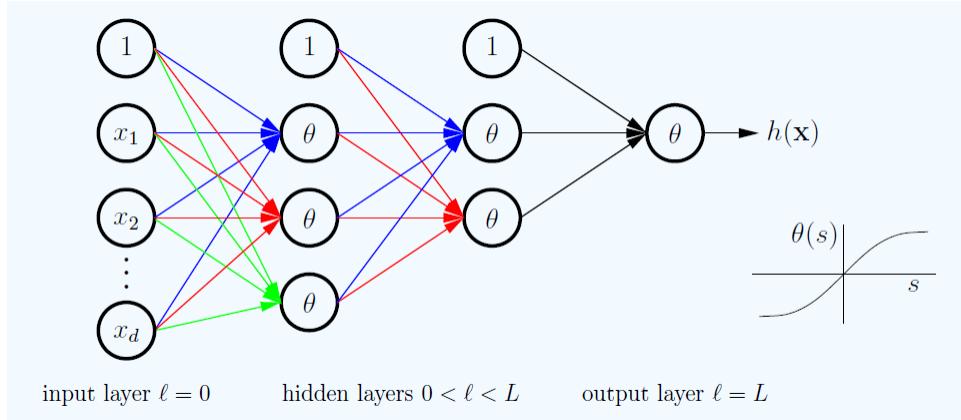


Figura 2.9: Estructura de una red neuronal artificial. Fuente: Yaser[2]

manda ha supuesto un gran cambio de concepto en lo que a servicios de recomendación se refiere[3]. Los traductores de idiomas han visto incrementado su rendimiento de manera extraordinaria con el uso de algoritmos de palabras vectorizadas basados en redes neuronales[35]. La aparición de herramientas como las Deep Fakes o la generación de imágenes a demanda tendrán repercusión en campos que hoy en día no podemos enumerar, todo ello gracias a las redes neuronales generativas adversarias[7]. El caso que aquí nos ocupa, sin embargo, abarca el estudio de la siguiente generación de redes neuronales: las spiking neural networks (SNN), o redes neuronales por impulsos.

Las redes neuronales artificiales clásicas son un modelo básico basado en el comportamiento de las neuronas reales. Simplificando, una neurona es un elemento que al que le llegan varios valores de entrada que multiplicados cada uno de ellos por un peso sumarán un valor determinado. Este valor será alterado por una función de activación y será enviado a las neuronas de la siguiente capa con las que esta esté conectada. Así, poco a poco, la red va codificando información. Su estructura se muestra en la Figura 2.9. Esta simplificación permite su ejecución en un tiempo razonable, siendo suficientemente precisa para gran cantidad de aplicaciones.

Existen varias arquitecturas para implementar sistemas de control robot mediante redes neuronales. Una de las más interesantes es el uso de redes neuronales recurrentes (RNN), dado que estas permiten almacenar un valor del pasado en cada una de las capas, lo cual hace este tipo de redes idóneas para el cálculo de series temporales. Sin embargo, si lo que queremos es simular estructuras neuronales reales, necesitamos un modelo más realista y por tanto más complejo. Las redes neuronales por impulsos (*Spiking Neural Network: SNN*) son redes neuronales artificiales que procesan la información de una manera mucho más realista. Estas redes implementan modelos de

neurona mucho más complejos cuyo estado está basado, al igual que en las neuronas reales, en su potencial de membrana. Además, este tipo de redes tienen otra característica principal: tienen en cuenta el tiempo en el que llegan los impulsos a cada neurona.

La entrada de este tipo de redes, por tanto, será un conjunto de pares de valores tiempo-neurona al que llamaremos impulsos. Estos deberán indicar el instante de tiempo en el que se recibe el impulso y la neurona a la que se le introduce. Las neuronas están conectadas entre sí de la misma manera que en las redes neuronales artificiales clásicas, que tanto en las redes biológicas como en las SNN llamaremos sinapsis. Si una neurona recibe los impulsos necesarios para que su potencial de membrana supere dicho umbral, esta emitirá un potencial de acción en forma de impulso que será enviado al resto de neuronas con las que este conectada. Veremos este comportamiento más detenidamente más adelante.

Capítulo 3

Material

3.1. El robot Baxter

Baxter es un cobot desarrollado por Rethink Robotics en 2012. A un coste asequible (unos 30000€) y fácilmente programable, es una opción muy atractiva para pequeñas empresas que buscan una mejora en su automatización. Como se puede ver en la Figura 3.1, Baxter es un robot antropomórfico que cuenta principalmente con dos brazos articulados con siete grados de libertad, sensores de posición, velocidad y fuerza y cámaras orientadas al uso de visión artificial.



Figura 3.1: Robot Baxter. Fuente: <https://www.elempaque.com/temas/Baxter,-el-robot-colaborativo-que-opera-en-equipo-con-el-personal+103502>

Pensado como un robot industrial, Baxter tiene una forma humanoide y dispone de un display a modo de cabeza que muestra imágenes para dotarle de una “expresión facial” que le da más humanidad. Rethink Robotics

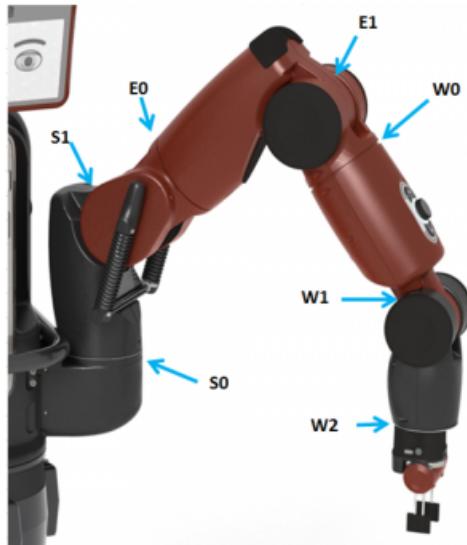


Figura 3.2: Articulaciones del robot Baxter. Fuente: [38]

integra en la unidad de control su paquete de software *Intera*, que permite programar el Baxter de manera sencilla, sin que sea realmente necesario personal específicamente especializado. La programación desde el punto de vista comercial está enfocada al uso de dicho software y al uso de la técnica del guiado pasivo.

En nuestro caso usaremos el brazo izquierdo del robot (Figura 3.2), aunque ambos están formados por las mismas articulaciones. Cada brazo consta de:

- Shoulder (hombro). Una articulación de torsión (S0) y otra de rotación (S1).
- Elbow (codo). Una articulación de torsión (E0) y otra de rotación (E1).
- Wrist (muñeca). Una articulación de torsión (W0), otra de rotación (W1) y una última de torsión (W2) muy cercana al efecto.
- Efecto. La herramienta que queramos poner en el brazo del Baxter, dependiendo de la aplicación para la que vaya a ser utilizado.

En el contexto de este proyecto nos interesa controlar tan solo S0, S1, E0, E1, W0 y W1, que son las articulaciones que forman los seis grados de libertad que nos permiten un movimiento total en tres dimensiones. Aunque el Baxter tiene un total de siete articulaciones (y de grados de libertad), la última de ellas sólo altera la orientación del efecto en última instancia.

Esto puede ser muy útil para simplificar el movimiento de las articulaciones para algunas trayectorias en una gran cantidad de aplicaciones, pero en nuestro caso el único objetivo es que el extremo del robot siga una trayectoria específica, por lo que es irrelevante su orientación durante este proceso. En consecuencia, ignoraremos la última articulación en todos los nodos del sistema de control, enviando en última instancia un valor nulo para que se mantenga fija como si de un eslabón más se tratara. Para más información se puede consultar la documentación oficial del hardware de Baxter[38].

Respecto a la programación, no haremos uso del sistema *Intera*, dado que nuestro interés se centra en integrar el sistema de control con cROS. Por tanto, usaremos la API (*Application Programming Interface*, o interfaz de programación de aplicaciones) de comunicación vía ROS, en la cual la unidad de control proporciona y recibe información referente al robot en topics. Los más interesantes para nuestro propósito son:

- `/robot/set_super_enable`. Enciende o apaga el robot.
- `/robot/limb/<side>/joint_command`. Recoge comandos para las articulaciones de los brazos.
- `/robot/joint_states`. Proporciona las medidas de los sensores de los brazos.
- `/robot/limb/<side>/endpoint_state`. Proporciona la posición, velocidad y aceleración del extremo del robot.

Se puede ampliar información referente a la API de comunicación con el robot en la documentación oficial[37].

3.2. El entorno de simulación Gazebo

Gazebo es un simulador de entornos robóticos en 3D, que tiene en cuenta tanto el modelo gráfico como el físico de manera precisa. Esta herramienta permite realizar desarrollos sin tener que estar físicamente en el lugar de trabajo, proporcionando un escenario realista. Es un entorno de simulación muy usado en todo tipo de aplicaciones, formando parte del motor de varias herramientas que requieren simulación.

Baxter está disponible en Gazebo de manera oficial. Rethink Robotics proporciona una guía de instalación de ROS y Gazebo de manera conjunta en Linux[39]. Siguiendo este procedimiento obtenemos una manera sencilla de testear el desarrollo del sistema de control objetivo sin necesidad de estar presente en el laboratorio, con las ventajas que esto conlleva.

3.3. La librería cROS

cROS es una implementación de código abierto de una librería cliente de ROS, como pueden ser roscpp o rospy entre otras[42]. Es todavía una novedad y aún no ha sido demasiado probada ni documentada, por lo que otra de las finalidades que enmarcaremos será el estudio de la interoperabilidad de cROS con otras librerías. Sin embargo, son varias las características que lo definen como una alternativa real y adecuada para su uso en sistema empotrados:

- Es una implementación en C sin dependencias de terceros, todo lo que necesita para ofrecer sus funcionalidades está presente en su código fuente.
- No necesita que ninguna versión de ROS esté instalada en el sistema, tan solo un ROS Master, que puede ser ejecutado por un intérprete de Python que trae incluido.
- Es interoperable con otras implementaciones de ROS, por lo que a pesar de ser poco utilizado, debería ser directamente compatible con, por ejemplo, los nodos que usa el robot Baxter para comunicarse con el ciclo.
- Es una implementación ligera y solo utiliza una hebra, lo que hace que pueda ser ejecutado en un hardware con potencia limitada como puede ser el caso de un sistema empotrado.

Más información sobre las características y funcionamiento de cROS puede encontrarse en su documentación oficial[41].

3.4. La plataforma empotrada NVIDIA Jetson

La plataforma NVIDIA Jetson AGX Xavier es un SoC, es decir, todos los elementos que necesita el sistema para funcionar están embebidos en una sola placa. En este caso el SoC está fabricado con la tecnología de 12nm de TSMC(ver Figura 3.3).

Este pequeño sistema empotrado, de un tamaño de $105mm \times 105mm$ monta una CPU de 8 núcleos NVIDIA Carmel de 64 bits de hasta 2265MHz basados en ARMv8, con una caché L1 por núcleo, cuatro cachés L2 compartidas por parejas y un último nivel L3 compartido por todos los procesadores. Todo ello acompañado de 32GB de RAM LPDDR4x a 2133MHz y una GPU Volta formada por 512 núcleos CUDA y 64 Tensor cores. Dado que está ideada para tareas de inteligencia artificial, también consta de aceleración

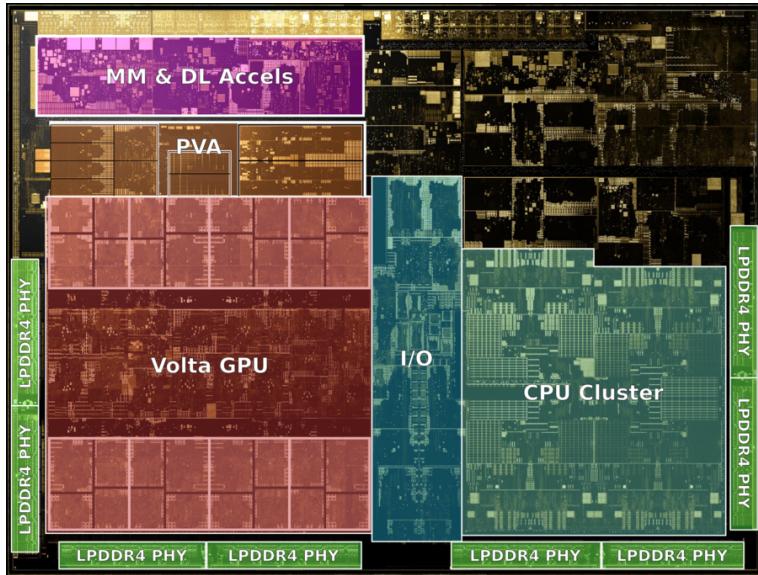


Figura 3.3: SoC NVIDIA Tegra. Fuente: <https://en.wikichip.org/wiki/nvidia/tegra/xavier>.

por hardware para visión por computador -*Programmable Vision Accelerator (PVA)*-, Deep Learning -*Deep Learning Accelerator (DLA)*-, procesamiento de imágenes -*Image Signal Processor (ISP)*- y multimedia(MM).

Además, proporciona una gran cantidad de opciones para entrada y salida de datos. Monta una unidad de almacenamiento eMMC de 32GB y PCIe4, Ethernet, M2 o HDMI son algunas de las interfaces de comunicación que tiene disponibles.

Si se quiere profundizar en el diseño hardware de este modelo de NVIDIA Jetson, se recomienda consultar las referencias correspondientes[19][53][50].

Esta plataforma es un claro ejemplo del gran avance dado en los últimos años en el mundo de la informática. Tenemos aquí un sistema empotrado, de no más de $0,6\text{ cm}^3$, que proporciona muy altas prestaciones para su tamaño y consumo. Este tipo de sistemas son los que nos permitirán (o más bien nos están permitiendo) acercar lo máximo posible la potencia computacional al sistema que la requiere. Esto permite relegar las comunicaciones a las estrictamente necesarias y así poder prescindir de un centro de procesamiento remoto para este cometido. Además, al ser un SoC, proporciona estas características a un coste energético mucho menor que un PC o, por supuesto, un centro de procesamiento de datos. En definitiva: permite un alto grado de autonomía y escalabilidad en un paradigma creciente en popularidad como es el de los sistemas distribuidos.

3.5. Otros materiales

Además de los materiales comentados anteriormente también se hará uso de:

- Un portátil, que servirá para desarrollar el sistema de control y hacerlo funcionar con el Baxter, además de para realizar pruebas con el simulador neuronal.
- Visual Studio Code (programación), Overleaf (documentación), Trello(tablero de *issues*) y Google Workspace (hojas de cálculo y almacenamiento en la nube) como herramientas de trabajo.
- El simulador neuronal EDLUT, para el uso de redes neuronales por impulsos. Se tratará ampliamente en capítulos posteriores.
- Un vatímetro para corriente alterna (en inglés, *alternating current*, AC) y otro para corriente continua (en inglés, *direct current*, DC), con el objetivo de tomar medidas de consumo energético. También se detallará su elección posteriormente.
- Adicionalmente, un PC y una Raspberry Pi Zero para realizar comparativas con el simulador neuronal. Se incluirán en el presupuesto dado que se han utilizado durante el proyecto, pero no son indispensables para su replicación.

3.6. Presupuesto

3.6.1. Recursos materiales

En la tabla 3.1 se muestran los distintos recursos utilizados durante el proyecto, así como su precio y distintas métricas que nos ayudarán a calcular un coste realista del mismo. Para este cálculo se ha utilizado la métrica $CA = \frac{DC}{V}$, siendo CA el coste real en euros, D tiempo que se ha utilizado el producto en meses, C el coste total del producto en euros y V el tiempo de vida útil estimado de cada uno.

3.6.2. Recursos humanos

En la tabla 3.2 se muestra el número de horas que se ha dedicado a cada una de las tareas. Se ha de tener en cuenta que al principio, dado que la mayor parte del tiempo compartía horario con cinco asignaturas, se dedicó tan solo una media de 3 horas al día. Finalmente, tras entrar en el segundo

Recurso	Coste (euros)	Dedicación (meses)	Vida útil (meses)	Coste aplicable (euros)
Jetson	999€	3	36	83,25€
cROS	Gratis	3	-	0€
Gazebo	Gratis	2,5	-	0€
EDLUT	Gratis	2	-	0€
Portátil	399€	5	36	55,42€
Vatímetro DC	16,5€	1	36	0,46€
Vatímetro AC	23€	0,5	36	0,32€
VS Code	Gratis	4	-	0€
Overleaf	Gratis	3	-	0€
Workplace	Gratis	1,5	-	0€
Trello	Gratis	4	-	0€
PC	846€	1	36	23,5€
Raspberry Pi	5,5€	0,5	36	0,08€

Cuadro 3.1: Recursos utilizados durante el proyecto.

cuatrimestre, se dedicó una media de 4 horas al día, dado que ya se disponía de más tiempo.

Tarea	Duración (días)	Horas de trabajo
Toma de contacto	30 días	90h
Control cROS	70 días	228h
EDLUT en la Jetson	60 días	252h
Documentación final	5 días	40h
Total	165 días	610h

Cuadro 3.2: Horas dedicadas al proyecto.

El coste estimado de los recursos humanos para el proyecto se calcularía multiplicando el sueldo medio por hora de un ingeniero por el número horas que este va a realizar. Dado que el sueldo medio mensual de un ingeniero informático en España es de 1870€[28], podemos deducir un coste total de 29845€ al año teniendo en cuenta el 33% referente al coste de la seguridad social. Si asumimos unas 1800 horas anuales de trabajo, obtenemos un valor de 16,58€/hora. Por tanto, el coste total referente a los recursos humanos sería de 10113,8€.

3.6.3. Coste total

Teniendo en cuenta todo lo anterior el coste total del proyecto, como se muestra en la tabla 3.3, es de 10276,83€.

Recursos materiales	163,03€
Recursos humanos	10113,8€
Coste total	10276,83€

Cuadro 3.3: Coste total del proyecto.

Parte II

Desarrollo: sistema de control robot

Capítulo 4

Ciclo de control de partida

4.1. Control en lazo abierto

Ahora vamos a describir el cómo y el porqué de cada uno de los nodos presentes en el sistema de control que vamos a tener como base para el proyecto. Los nodos van a ser presentados, en su mayoría para el control en lazo abierto, y serán reutilizados en los ciclos en lazo cerrado.

El ciclo de control en lazo abierto presenta la estructura de la Figura 4.1. Se debe tener en cuenta que, aunque nos referimos a este como un lazo abierto, no lo es realmente. Es cierto que desde la unidad de control externa que estamos utilizando no usamos realimentación, pero en este modo de control el robot regula el movimiento mediante su unidad de control interna, para lo cual sí usa realimentación. Por tanto, cuando se hable del ciclo de control en lazo abierto implementado en este proyecto, nos estaremos refiriendo a que no existe realimentación en la unidad de control externa. A continuación se detallará cada uno de los nodos presentes en sistema de control.

4.1.1. Tiempo de simulación y retardo por conexión inalámbrica simulada

El tiempo de simulación es una funcionalidad que sustituye el reloj a tiempo real del sistema por un reloj que genera un tiempo simulado en el que avanza más lentamente. De esta manera se puede realizar una simulación de un experimento que requiere tiempo real utilizando menos recursos, dado que se dispone de un plazo (*deadline*) mayor. Esto es, cuando no podemos cumplir la condición RT (Real Time) y aun así queremos realizar el experimento, extendiendo artificialmente el período antes del *deadline*.

Si no usamos tiempo de simulación, el tiempo actual se recibe desde la

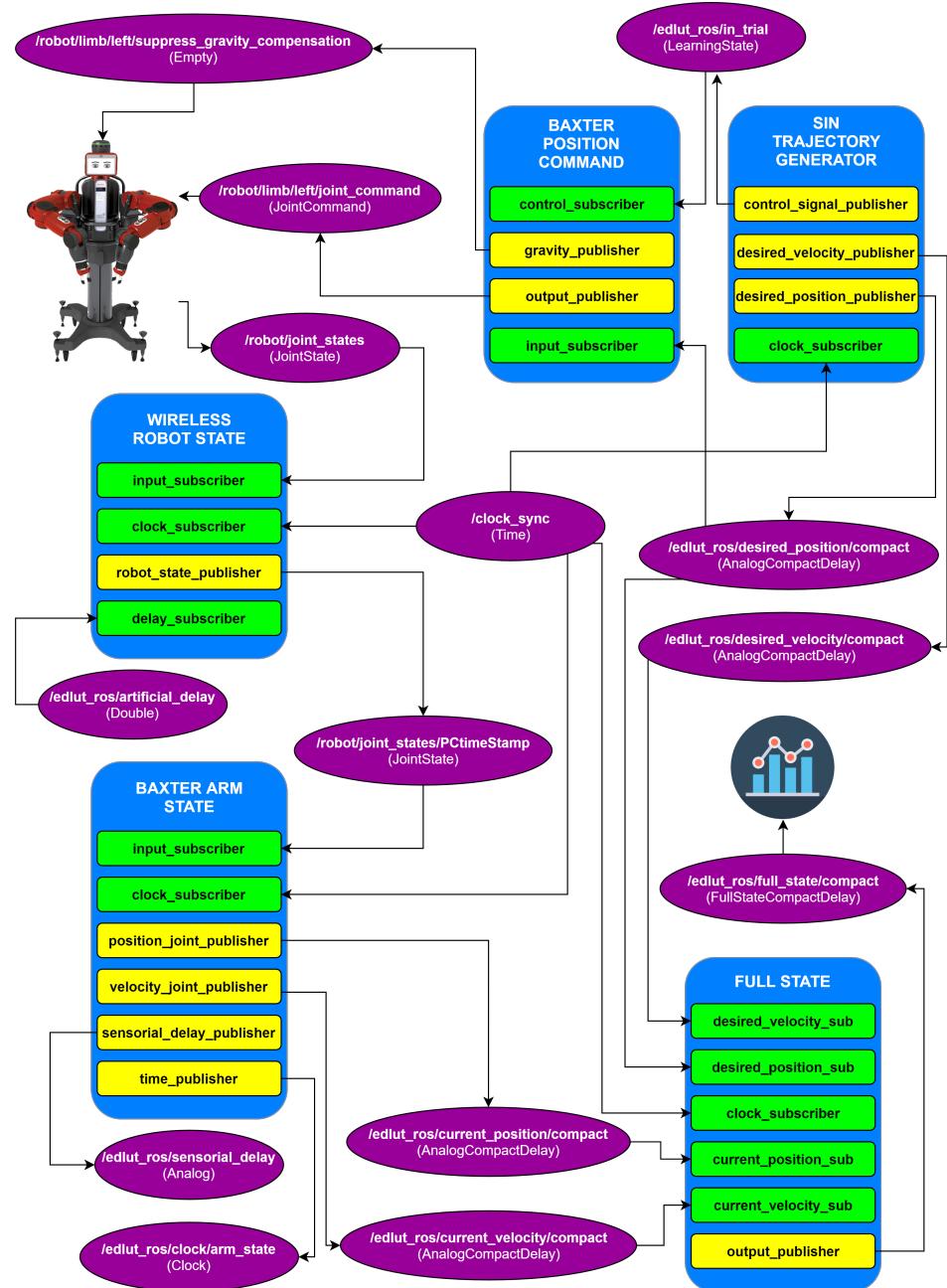


Figura 4.1: Ciclo de control en lazo abierto

API de ROS Time. Sin embargo, si usamos tiempo de simulación, este se obtendrá del topic */clock_sync*. Como se puede ver en la Figura 4.1, todos los nodos que deben tener en cuenta el tiempo leen de este topic, por lo que sincronizará el resto de nodos. Por supuesto, es necesario que algún nodo publique en el topic */clock_sync* con la frecuencia necesaria según se el tiempo que se quiera simular.

Dado que, como se explicará en el capítulo siguiente, no se va a usar esta funcionalidad en el contexto de este proyecto, se omite en el diagrama dicho nodo sincronizador. Es por ello que aparecen topics en los que sólo se lee/escribe de un solo nodo.

Por otro lado, el retardo por conexión inalámbrica simulada es una funcionalidad que permite introducir retardo en los mensajes provenientes del robot, simulando así una conexión inalámbrica. Esto es interesante de cara a poder simular situaciones en las que el robot no está conectado a la misma red que el PC que lo controla y/o hay conexiones inalámbricas entre ellos, y por tanto pueden darse interferencias.

Dada su naturaleza, una conexión WiFi no tiene tanta fiabilidad como una conexión cableada. Existe una probabilidad real de que un paquete se pierda y se tenga que reenviar o se retrase. El nodo Wireless Robot State se encarga de recoger los mensajes del robot y actualizar su marca de tiempo. Es aquí donde, mediante el subscriber correspondiente se lee el retardo artificial del topic */edlut_ros/artificial_delay* y se introducirá en la marca de tiempo de cada mensaje. Lo más natural es introducir un retardo aleatorio a cada mensaje, lo cual es misión del nodo que publica en */edlut_ros/artificial_delay* (que en el caso del diagrama de la Figura 4.1 queda obviado).

Controlar un posible retardo en los mensajes introduce, al igual que el uso de tiempo simulado, una complejidad adicional, dado que tenemos que tener en cuenta que este retardo a la hora de interpretar las marcas de tiempo y controlar el caso en el que los mensajes lleguen desordenados(un mensaje con retardo corto, precedido por otro con un retardo más largo, puede anticiparse a él). Además, hay que tener en cuenta que una de las bondades de empotrar el ciclo de control es que aumenta en gran medida la portabilidad del sistema, por lo que no tiene demasiado sentido conectarlo de manera inalámbrica ya que puede ir prácticamente alojado en la propia estructura del robot. Es por ello que esta funcionalidad tampoco se va a tener en cuenta en la implementación del sistema de control empotrado.

Estos nodos ROS de los que partimos fueron implementados en C++ y Python para relazar la comunicación entre el simulador EDLUT y Baxter en el marco de un trabajo de investigación para el control robótico[1]. Usando su diseño como referencia hemos realizado la implementación en C++ y mediante cROS de un subconjunto de los nodos presentes en este diagrama

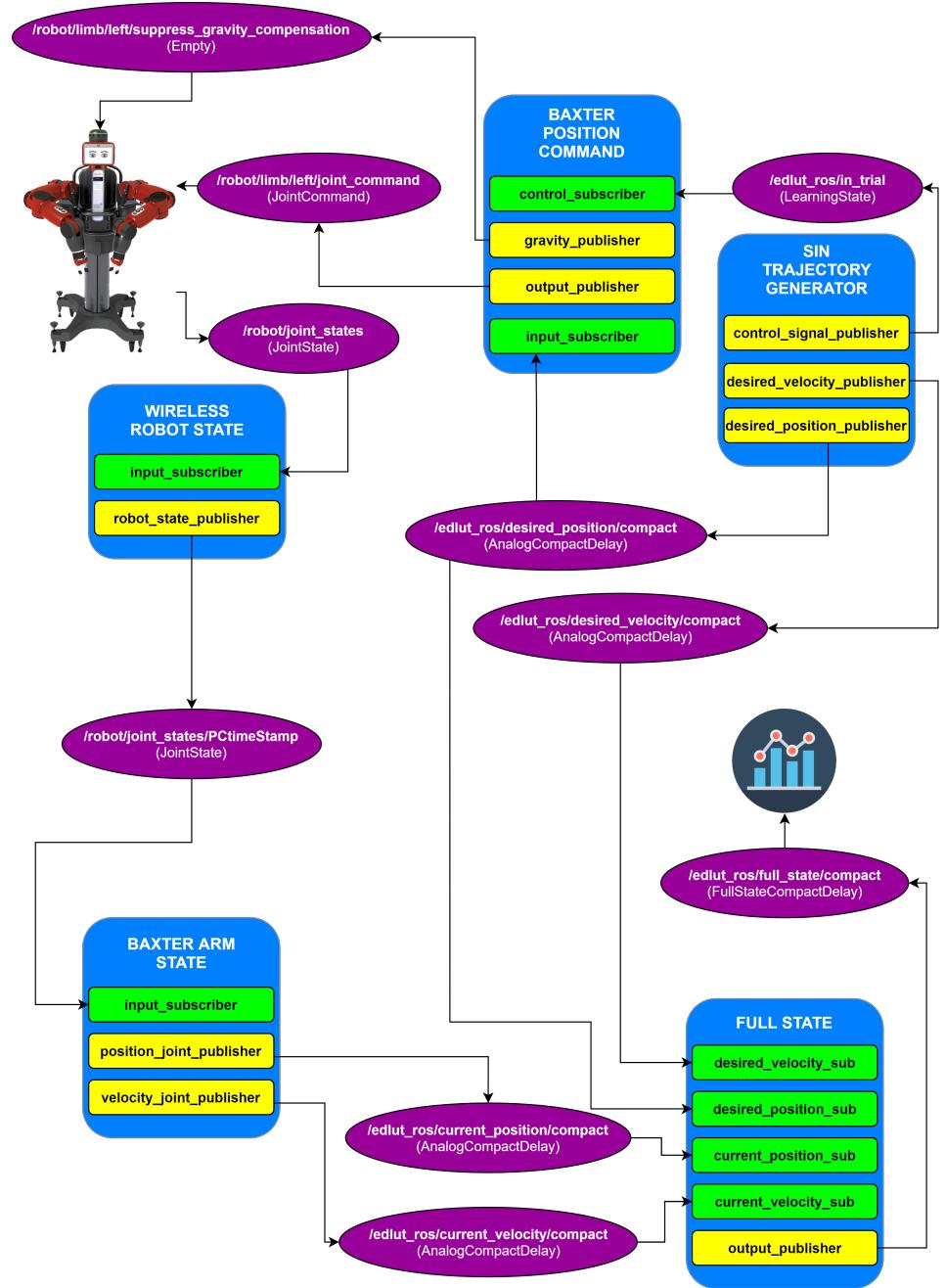


Figura 4.2: Ciclo de control en lazo abierto simplificado

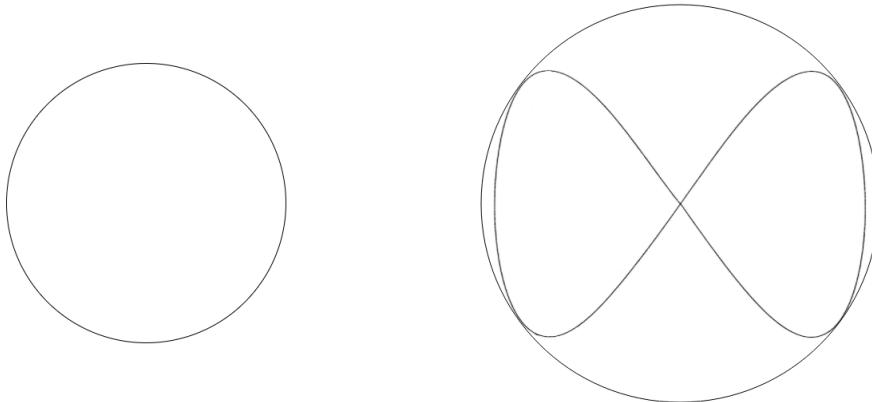


Figura 4.3: Esbozo de trayectorias deseadas, vistas en planta. A la izquierda la trayectoria circular, a la derecha la trayectoria en forma de ocho

para permitir el control en lazo abierto (y cerrado) como se describe en la sección 5. Vamos entonces, por simplicidad, a tomar una estructura simplificada en la que se obvien los topics, y la comunicación con estos, que formen parte de estas funcionalidades, quedándonos así con los nodos y topics directamente relacionados con la implementación que se va a llevar a cabo. El diagrama de dicha simplificación se puede ver en la figura 4.2.

4.1.2. Generación de trayectorias

El nodo *Sin Trajectory Generator* es el encargado de generar la trayectoria que debe seguir el robot. La generación de trayectorias es de por sí un problema en el campo de la robótica, ya que no es trivial generar la trayectoria que debe seguir un robot por básica que ésta sea. Al requerir las trayectorias en el espacio de coordenadas del robot, se debe muestrear la trayectoria y convertir esos puntos desde el espacio de coordenadas cartesiano mediante la resolución del problema cinemático inverso del robot. Para nuestro caso se van a plantear dos trayectorias, una circunferencia y un lazo en forma de ocho (ver Figura 4.3).

Estas quedarán definidas en ficheros *.txt*, en concreto se necesitarán dos archivos para cada una, uno para la posición y otro para la velocidad. Las trayectorias, previamente calculadas, presentan un formato en el que cada línea contendrá la posición/velocidad deseada para un momento concreto, correspondiéndose la primera con el instante inicial y siendo las siguientes los comandos deseados en instantes posteriores según la frecuencia de muestreo. En el caso concreto del robot Baxter, cada línea contendrá la posición/ve-

locidad (angular) de cada una de las siete articulaciones del brazo robótico. De estas, obviaremos la última dado que se corresponde con la articulación del efecto, el cual no es interesante dado que no se le va a dar uso.

Siguiendo el formato antes comentado, el nodo generador cargará en memoria las trayectorias definidas en los ficheros e irá publicando las posiciones deseadas de las articulaciones en el topic `/edlut_ros/desired_position/compact`, así como las velocidades deseadas en el topic `/edlut_ros/desired_velocity/compact`, utilizando la misma frecuencia usada para el muestreo. Dado que el fichero de trayectoria usado durante los experimentos, calculado y proporcionado por el grupo de investigación, contiene 1000 puntos de la trayectoria circular/en ocho y está pensada para completar una vuelta en dos segundos, se usará en todo momento una frecuencia de 500 Hz. Esto es, se enviarán 500 mensajes de comando cada segundo, o lo que es lo mismo, enviaremos un mensaje al robot cada 2 ms. Por tanto la condición de RT (Real Time) o `deadline` será también de 2 ms.

Además, el nodo *Sin Trajectory Generator* enviará una señal al topic `edlut_ros/in_trial` antes de comenzar el envío, de manera que el nodo que se comunica directamente con el robot solo escuche cuando la ejecución esté activa. Esta funcionalidad, aunque no es necesaria, ha sido heredada de anteriores versiones y funciona a modo de control de seguridad, por lo que se mantendrá.

Por lo tanto, y por parte de este nodo, tendremos la información para los comandos de control en `/edlut_ros/desired_position/compact` y en `/edlut_ros/desired_velocity/compact`, así como una señal en `edlut_ros/in_trial` que confirma que ya se está produciendo información en los otros dos topics. El siguiente paso es traducir los comandos al lenguaje que pueda entender el robot.

4.1.3. Traducción de mensajes de control

Realmente, para el control en lazo abierto, solo sería necesario el proceso comentado en la sección anterior y la traducción al formato que pueda computar el robot, trabajo que realiza el nodo *Baxter Position Command*. Los mensajes de comando, cuando viajen entre nodos internos de nuestro ciclo, tendrán un formato al que llamaremos *Analog Compact Delay*. Dicha morfología permite mantener toda la información necesaria dentro del sistema de control con un formato unificado.

No obstante, el Baxter necesita que le introduzcamos la información a un formato específico *Joint Command*[40] al enviarla al topic `/robot/limb/-left/joint_command`, que es la entrada de comandos del robot. El trabajo del nodo *Baxter Position Command* será, simplemente, traducir los comandos del formato *Analog Compact Delay* a *Joint Command*, siempre que la señal

en el topic *edlut_ros/in_trial* esté activa.

Para el caso del ciclo de control en lazo abierto, la entrada para el nodo traductor será simplemente la posición deseada, no siendo necesaria por el momento la velocidad. Por tanto el flujo de entrada del nodo vendrá dado por el topic */edlut_ros/desired_position/compact*, además de la señal *in_trial* antes comentada. Cada 2 ms, el nodo generará un comando robot con la posición deseada, utilizando el modo de control por posición. Además, si es necesario, el nodo deberá de enviar una señal al topic */robot/limb/left/suppress_gravity_compensation* para que el robot desactive la compensación de gravedad.

4.1.4. Monitorización

Como ya se comentó en la anterior sección, con los nodos de generación y traducción ya queda conformado el ciclo de control. Sin embargo, con objetivo de monitorizar el experimento, también se usan algunos nodos que recogen información del robot y proporcionan datos en tiempo real de los sensores presentes en el brazo del robot. Estos nodos formarán parte del ciclo de control en lazo cerrado, por lo que es de interés comprenderlos.

Primero, el nodo *Wireless Robot State* recogerá del topic */robot/joint_states* los datos de los sensores directamente del robot y restablecerá la marca de tiempo conforme a reloj del sistema y los ajustes de tiempo de simulación y/o simulación de retardo inalámbrico. Conceptualmente podemos ignorar dicho nodo, dado que realiza tareas puramente técnicas y genera un mensaje prácticamente idéntico en lo que a datos del robot se refiere.

Este mensaje, ya en el topic */robot/joint_states/PCtimeStamp*, será recibido por el nodo *Baxter Arm State*, el cual se encargará de entregar información del estado real de las articulaciones a distintos topics, dejando por separado la posición y velocidad (angulares) actuales del robot. Esto nos permitirá tener, además de la posición y velocidad deseadas en los ya comentados topics, la posición y velocidad reales en */edlut_ros/current_position/compact* y */edlut_ros/current_velocity/compact* respectivamente.

Estos cuatro topics serán recopilados a través del nodo *Full State* en un sólo mensaje, que será publicado en *edlut_ros/full_state/compact* y del que se servirán los distintos gráficos que muestran a tiempo real el estado del ciclo de control.

4.1.5. Funcionamiento

Con todo lo explicado en este apartado, queda definido conceptualmente el control en lazo abierto y los nodos que lo hacen funcionar. Como se puede

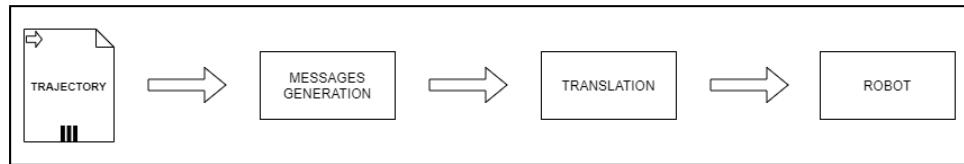


Figura 4.4: Traza de los comandos de trayectoria durante el ciclo de control en lazo abierto

ver, es efectivamente un lazo abierto, ya que en ningún momento existe alguna retroalimentación desde que se genera el comando de posición hasta que este llega al robot (Figura 4.4).

Este tipo de control tiene la ventaja de ser muy simple en su funcionamiento e implementación. Como se puede ver, tan solo requiere de generar las posiciones en las que queremos se mueva el robot, traducirlas, y publicarlas en los topics de entrada del Baxter utilizando el formato necesario para ello. Sin embargo, la falta de realimentación hace que el robot sea críticamente sensible a elementos externos presentes en su espacio de trabajo. Estos pueden ser colisiones accidentales o el propio peso del efecto, así como cualquier otro tipo de fuerza aplicada al brazo robótico que no permita el movimiento del mismo en perfectas condiciones. Este escenario desestabiliza por completo el ciclo de control, resultando en una parada completa por los propios mecanismos de seguridad el robot al usar el modo de control por posición.

Esto hace al ciclo de control en lazo abierto incompatible con prácticamente la totalidad de las situaciones reales fuera de un laboratorio. Y es por ello por lo que son necesarios controles que tengan en cuenta el entorno y los elementos externos presentes en él que puedan afectar al robot.

4.2. Control en lazo cerrado: controlador PD

4.2.1. El controlador PD

Vistas las características del control en lazo abierto, el siguiente paso lógico es añadir al ciclo algún tipo de realimentación. Uno de los controladores más usados en la industria es el controlador PID, el cual, a partir de la situación real del sistema, es capaz de corregir los comandos consigna de control para enviarlos a los actuadores.

En nuestro caso concreto, la manera de integrarlos será creando un nodo que calcule los comandos PD (sin la componente de integración I) y los envíe, traducidos, al robot. Véase la nueva estructura del ciclo de control en la Figura 4.5.

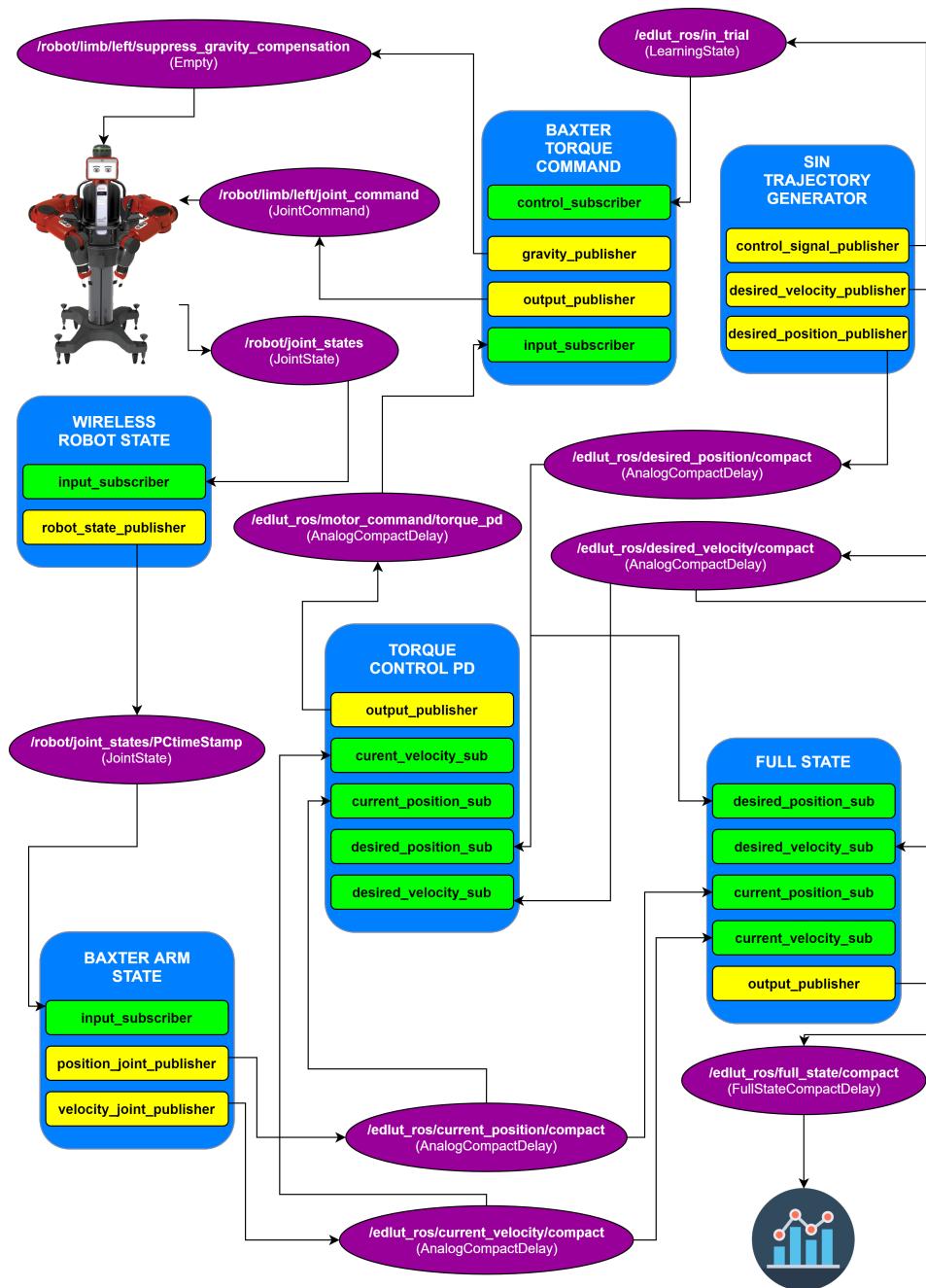


Figura 4.5: Ciclo de control en lazo cerrado con control por PD

En el caso del control en lazo cerrado, dado que los inputs del robots vendrán influenciados por el error entre la posición y velocidad deseadas y reales, no se debe realizar un control por posición como en el caso del sistema en lazo abierto. En lugar de ello, tiene más sentido aplicar un par de fuerzas concreto a cada una de las articulaciones dependiendo del error en la posición de cada una de ellas. Al par, el momento angular aplicado sobre un eje (en este caso sobre los ejes de las articulaciones), se le podrá nombrar en adelante como par, *torque* o fuerza de torsión, todos términos equivalentes. Para realizar el control del ciclo robot con un algoritmo PD, es necesario diseñar un controlador a partir de la expresión 2.1.

Respecto a la componente I, no es interesante para este cometido y es por ello por lo que se establece K_I a 0 y se elimina por tanto toda la componente integral. La razón es que, al menos en nuestro caso concreto, esta no ha resultado ventajosa en los experimentos realizados.

Respecto al otro término que hay que establecer de antemano, $P(0)$, es claro que su valor debe ser 0. Si el error es nulo para una articulación del robot, es decir, no existe diferencia entre la posición real y la deseada, la salida del controlador idónea para esta situación es no aplicar ninguna fuerza sobre dicha articulación.

A partir de las anteriores reflexiones, podemos obtener la expresión 4.1 que define al controlador PD que usaremos, donde $E(t)$ será el error, es decir, la diferencia entre las posiciones real y deseada que calcularemos en tiempo real.

$$P(t) = K_P E(t) + K_D \frac{dE(t)}{dt} \quad (4.1)$$

Por tanto necesitamos tanto la posición real y la deseada del robot como sus derivadas, que son justamente las velocidades. Aprovechando los nodos de monitorización explicados anteriormente podemos obtener la posición y velocidad actuales del robot en tiempo real. Los valores deseados están disponibles a partir de los archivos de generación de trayectorias, y el nodo *Sin Trajectory Generator* los publica en los topics correspondientes. Es por ello que además de la posición se almacena la velocidad deseada, porque es necesaria para este tipo de control.

Teniendo en cuenta todo esto, la adaptación del ciclo ROS consiste en añadir un nodo *Torque Control PD* que reciba información de los cuatro nodos */edlut_ros/desired_position/compact*, */edlut_ros/desired_velocity/compact*, */edlut_ros/current_position/compact* y */edlut_ros/current_velocity/compact*. Este nodo calculará la expresión 4.2, que es el caso concreto de la anterior 4.1, para cada articulación i .

$$P_i(t) = K_{P_i}E_i(t) + K_{D_i}\frac{dE_i(t)}{dt} \quad (4.2)$$

$$E_i(t) = desired_position_i - current_position_i \quad (4.3)$$

$$\frac{dE_i(t)}{dt} = desired_velocity_i - current_velocity_i \quad (4.4)$$

K_P y K_D serán vectores de n posiciones, siendo n el número de articulaciones del robot, y cada una de ellas serán los valores de ajuste del controlador PD.

Por último, el nodo *Torque Control PD* publicará los comandos de torque en `/edlut_ros/motor_command/torque_pd`. La información será recogida por *Baxter Position Command*, que enviará los comandos traducidos al robot, pero indicando que estos deben ser ejecutados en el modo de fuerza en lugar del modo posición.

4.2.2. Ajuste del controlador PD

El ajuste de un controlador PD no es algo sencillo, existen distintos métodos para realizarlo, desde manuales hasta métodos matemáticos.

En este caso se partirá de las constantes obtenidas empíricamente por el grupo de investigación en previos experimentos, teniendo en cuenta que será distinto para el modelo simulado del robot. A partir de ahí se irá ajustando manualmente según el comportamiento del robot, lo que en el argot se conoce como ajuste por tanteo.

4.2.3. Funcionamiento

Con todo lo explicado en este apartado, queda definido conceptualmente el control en lazo cerrado con controlador PD y los nodos que lo hacen funcionar. Como se puede ver, es efectivamente un lazo cerrado ya que los comandos que el ciclo de control envía al robot ahora sí que están realimentados (Figura 4.6).

Esto hace que, si el controlador está correctamente ajustado, el ciclo sea capaz de reajustarse automáticamente si interviene algún elemento externo como pueda ser un choque accidental o una fuerza aplicada sobre el brazo.

No obstante, el problema principal de este tipo de algoritmos es que acaban por utilizar pares de fuerza muy altos para minimizar el error, lo que hace que el entorno del robot no sea seguro ante errores críticos en la unidad de control o, simplemente, cambios rápidos de posición. Es por ello que los entornos de los robots industriales suelen estar perimetradados y no se permite el acceso al personal humano.

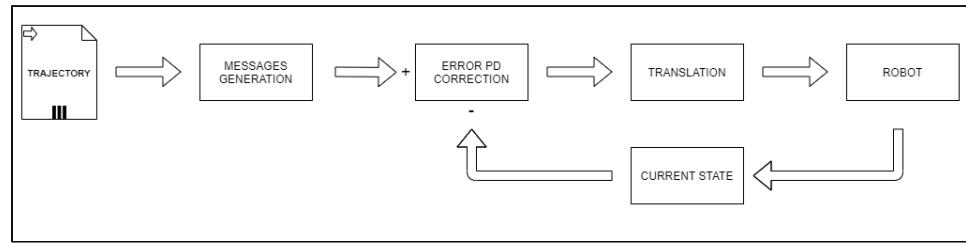


Figura 4.6: Ciclo de control en lazo cerrado con control por cerebro artificial

4.3. Problemática, obsolescencia y adecuación

Aunque este sistema de control cumple su función perfectamente, no es adecuado para sistemas empotrados. Requiere tener instalada una versión de ROS, en nuestro caso Kinetic, y varias dependencias. Sería mucho más adecuado implementar un sistema más orientado al *plug and play*, que no requiera de mucho tiempo ni recursos para instalarse y funcionar, y que no requiera dependencias ya que no es adecuado mantener instalado demasiado software en un sistema empotrado.

Por otra parte, ROS Kinetic es una versión antigua y sólo funciona en Ubuntu 16 (versión que ya ha dejado de tener soporte) y equivalentes. Dado que en la Nvidia Jetson no es posible instalar versiones anteriores a Ubuntu 18, no es posible utilizar directamente el ciclo de control en la Jetson.

Debido a estos y otros hechos, se ha planteado portar este sistema de control a otras librerías compatibles con ROS. La alternativa principal es cROS, una librería de ROS que aúna todas las características antes comentadas. Otra opción a tener en cuenta es ROS2, que está más enfocado a tiempo real, pero su uso requeriría un rediseño completo del sistema de control.

Capítulo 5

cROS como alternativa

5.1. Planteando la adaptación del diseño base

Como ya se ha comentado, las librerías ROS que implementan los nodos del ciclo de control implican varios compromisos que no lo hacen adecuado para empotrar este ciclo de control en la plataforma NVIDIA Jetson. Teniendo en cuenta este hecho, se han estudiado varias alternativas como puedan ser ROS2[44] o cROS[22], de las cuales esta última se torna como la más prometedora para un sistema empotrado dadas sus características, aunque la primera también será valorada en su momento.

En este apartado se va a plantear la adaptación de los sistemas de control detallados en el capítulo anterior, teniendo en cuenta las condiciones en las que este deberá de funcionar.

Como es lógico, no se pretende replicar el funcionamiento original en toda su complejidad, dado que lleva varios años desarrollándose para realizar distintos experimentos mientras que el objetivo de este proyecto es comprobar su funcionamiento en una plataforma empotrada. Por ello, se simplificarán varias funcionalidades que no serán interesantes como son el tiempo de simulación y el retardo por conexión inalámbrica simulado.

Teniendo en cuenta que se debe hacer funcionar en un sistema empotrado con un robot real, el tiempo de simulación no es interesante ya que no podemos utilizar un tiempo simulado en el mundo real, y dado que su implementación requiere introducir una mayor complejidad al sistema, se ha optado por omitirla simplificando así la lógica de la mayor parte de los nodos.

Por otra parte, el hecho de ejercer el control desde una plataforma empotrada implica que esta pueda estar montada junto al robot o incluso en su propio chasis. Por ello no tiene interés implementar la funcionalidad del re-

tardo por conexión inalámbrica simulado, dado que no tiene sentido usar una conexión inalámbrica. Esto permite también simplificar la lógica de varios nodos que tenían en cuenta la posibilidad de que la información les llegara desordenada, tanto que, por ejemplo, directamente se puede prescindir del nodo *Wireless Robot State*.

5.2. Adaptación de nodos ROS a cROS

Durante esta sección se detallarán los cambios que conlleva el paso de aplicaciones de las librerías estándar de ROS (en adelante no podremos referir a estas simplemente como ROS) a cROS. Aunque cROS al fin y al cabo es una librería de ROS, esta tiene algunas peculiaridades a tener en cuenta a la hora de implementar nodos para una red como la que nos incumbe. En las siguientes líneas se explicarán los aspectos más cercanos a la implementación del ciclo de control, sin embargo, y dado que cROS ni dispone de demasiada documentación ni ha sido demasiado usado, también se intentará dar una visión más global de las diferencias que se han ido encontrando durante el proceso de adaptación.

5.2.1. Funcionalidades de tiempo

Los ciclos de control implementados este proyecto requieren varias funcionalidades relacionadas con el tiempo y ofrecidas por la API de ROS. Dado que vamos a utilizar cROS en vez de la distribución oficial de ROS, y que la API de ambas no es exactamente igual, hemos buscado la forma de obtener estas funcionalidades mediante cROS. En este caso particular tienen que ver con las medidas de tiempo.

- *ROS::Time*: La API *Time* de ROS se usa para gestionar todas las marcas de tiempo de los mensajes, las cuales son necesarias sincronizar y emparejar las posiciones y velocidades actuales y deseadas. Esta funcionalidad es fácilmente sustituible por las funciones declaradas en el fichero de cabecera *cROSClock* de cROS, con el única salvedad de que perdemos precisión desde los nanosegundos hasta los microsegundos. No obstante, y dado que este proyecto no requiere precisión más allá de los milisegundos, no conlleva ningún problema.
- *ROS::rate*: La API *Rate* de ROS se usa para mantener una frecuencia fija en la recepción/envío de mensajes por parte de los nodos. Esta es fácilmente sustituible por la frecuencia que le podemos asignar a las funciones *callbacks*. Si por el contrario queremos usar envío de mensajes instantáneo, deberemos de encontrar una alternativa.

5.2.2. Acceso a los campos de los mensajes

Una de las principales diferencias entre ROS y cROS es la manera de manejar la definición de las estructuras de datos que forman cada tipo de mensaje. En ambas lógicas de trabajo los mensajes han de ser definidos en ficheros con formato *msg* como pueda ser el siguiente, *AnalogCompactDelay.msg*, que se usa en nuestro sistema de control para codificar el estado cinemático o dinámico de un actuador del robot.

```
Header header
float64[] data
string[] names
float64 delay
```

Sin embargo existe una gran diferencia en la generación de mensajes entre ambos entornos. En ROS se crean cabeceras en tiempo de compilación con las estructuras de datos correspondientes a la especificada en los archivos de definición de mensajes. Por tanto, en el caso de ROS podemos crear estructuras de datos complejas de mensajes como vectores o colas, siendo estas de fácil acceso dado que se trata de una estructura como cualquier otra. Por ejemplo, si tenemos la definición anterior de *AnalogCompactDelay.msg* y sabemos que *Header* es otra estructura conformada por una *stamp*, que a su vez contiene dos enteros *secs* y *nsecs*, podemos realizar acciones como las siguientes:

```
1 // Create the message
2 AnalogCompactDelay msg;
3 // Create a vector of messages
4 std::vector<AnalogCompactDelay> msg_vector = std::vector<
    AnalogCompactDelay>();
5
6 // Set stamp secs to 0
7 msg.header.stamp.secs = 0.0;
8 // Introduce the message into the vector of messages
9 msg_vector.push_back(msg);
```

Lo que tiene que quedar claro con esto es que, en el momento en el que estos ficheros se compilan y se crean las estructuras de datos correspondientes, los mensajes pasan a ser un tipo de dato como cualquier otro. Por tanto podemos tratarlos como tal y generar estructuras todo lo complejas que necesitemos. Por ejemplo, en el sistema de control actual, una de las vías para sincronizar mensajes es una cola con prioridad.

Sin embargo, en cROS los mensajes se gestionan de una manera totalmente diferente. En lugar de generar código a partir de los ficheros de definición, estos se leen en tiempo de ejecución. Esta implementación tiene

la ventaja de que nos ahorraremos realizar la generación de los ficheros de cabecera (y recompilar el proyecto) cada vez que se realiza alguna modificación de los ficheros de definición de mensajes. Pero en contrapartida, el acceso no es tan directo como en el caso anterior dado que a la hora de escribir el código no existe la estructura de datos como tal. cROS utiliza la estructura *cRosMessage* para cualquier mensaje y *cRosMessageField* para cada uno de sus campos, pudiendo acceder a ellos en cualquier tipo de datos como si de una unión se tratara. Por ejemplo, para realizar el mismo procedimiento llevado a cabo anteriormente para *AnalogCompactDelay.msg* (sin introducirlo en un vector), en cROS sería de la siguiente manera:

```

1 // Create the message, the publisher structure have saved
   the name of the definition
2 cRosMessage *msg = cRosApiCreatePublisherMessage(node,
   publisher);
3 // Access to the field recursively
4 cRosMessageField *field;
5 field = cRosMessageGetField(msg, "header");
6 field = cRosMessageGetField(field->data.as_msg, "stamp");
7 field = cRosMessageGetField(field->data.as_msg, "secs");
8
9 // Set stamp secs to 0
10 field->data.uint32 = 0.0

```

Este hecho lleva consigo varias consecuencias, siendo una de ellas la necesidad tener un control más exhaustivo de los posibles errores con los ficheros de definición, tanto en el código como en su propio contenido. Un ejemplo de este último caso podría ser el siguiente, donde se proponen dos ficheros de definición para el mensaje *JointCommand*, que estará compuesto por un entero *mode* y dos vectores, uno de flotantes, *command*, y otro de strings, *names*. Uno de ellos es correcto, con el otro obtendremos un error en tiempo de ejecución referente a la lectura de este tipo de mensaje.

```

int32 mode
float64[] command
string[] names

```

```

int32 mode
float64[] command
string [] names

```

Si observamos bien, la única diferencia entre uno y otro es el espacio extra presente *command* y su tipo. Este error en concreto ha ocurrido durante este proyecto, y es debido a que ese espacio no es tenido en cuenta por ROS (y estaba presente en los archivos de definición que descargué para el Baxter),

pero sí por cROS, donde esta componente del mensaje pasa de estar definida como “*command*” a “*command*”.

Este tipo de errores también es necesario controlarlos desde el código, dado que al gestionarse en tiempo de ejecución, no existe ninguna certeza de que los mensajes estén definidos exactamente donde y como nosotros lo hemos hecho. Esto también hace que el código quede algo más largo. Si cROS intenta leer un mensaje y aparece un error a la hora de reconocer un campo, este devolverá NULL. La manera correcta y segura de acceder al campo mode, por ejemplo, sería la siguiente para ROS y cROS respectivamente:

```
1 // Set mode to 1
2 msg.mode = 1;
```

```
1 // Set mode to 1
2 cRosMessageField *field = cRosMessageGetField(msg, "mode");
3 if (field != NULL) {
4     field->data.as_uint32 = 1;
5 }
```

De esta manera, si accedemos de manera recursiva a los campos del mensaje como hemos hecho en ejemplos anteriores, la forma correcta de hacerlo es encadenar la comprobación de que *field* no es nulo cada vez que accedamos a un nuevo campo.

5.2.3. Contexto entre *publishers* y *subscribers*

Cuando diseñamos un nodo para que porte varios publisher y subscribers es usual que estos deban compartir información. Por ejemplo, podemos obtener información mediante un subscriber, transformarla y enviarla a otro nodo a través de un publisher. Esto requiere que la información que llega por el subscriber sea accesible desde la parte del código que la transforma y desde la que la envía. En ROS esto se suele solucionar utilizando clases, de manera que las funciones *callback* encargadas de recibir los datos son métodos de clase y la información compartida se almacena en variables de instancia (o de clase, depende de la lógica que se quiera seguir). Posteriormente podemos acceder a dicha información mediante las funciones implementadas para ello en la clase correspondiente. Un ejemplo simplificado (se han omitido algunos parámetros y/o declaraciones) de esta lógica podría ser el siguiente:

```
1 // Class that contains the information
2 class Foo {
3 private:
4     double data;
```

```

5 public :
6     double getData() {
7         return data;
8     }
9
10    void Callback(const foo_type ::ConstPtr& msg) {
11        this->data = msg->data;
12    }
13}
14
15int main() {
16    [...] // ROS Initialization
17    Foo foo_obj;
18    // Create the subscriber with the class callback
19    ros::Subscriber subscriber = nh.subscribe(topic1, &Foo
19        ::Callback, foo_obj);
20    // Create the publisher
21    ros::Publisher publisher = nh.advertise<foo_type>(
22        topic2);
23
24    foo_type pub_msg;
25
26    while (true) {
27        // Wait for publications in the callback
28        CallbackQueue.callAvailable();
29        // Publish the information
30        pub_msg.data = foo_obj.getData();
31    }
31}

```

En cROS existen varios modos de tratar esta situación, siendo la cuestión más importante si estamos utilizando funciones *callback*(ambos) o no. Esto dependerá de si necesitamos una función compleja para el envío o recepción de datos. Si sólo vamos a usar envío inmediato y polling, no hay ningún problema ya que toda la información estará disponible directamente en el *main*. Sin embargo, si usamos alguna función *callback*, esta no dispondrá localmente de la información a enviar si está en el lado del publisher o los datos que reciba serán locales en el caso del subscriber. Las *callbacks* en cROS deben ser funciones estáticas, por tanto no podemos hacer uso de clases directamente como en ROS. La solución más sencilla es usar el puntero de contexto que proporciona cROS cuando creamos ambas entidades. Este es un puntero genérico, por tanto podemos crear una estructura de datos con toda la información compartida, la cual será accesible desde las *callbacks* de las entidades cuyo puntero de contexto apunte a dicha estructura. Un ejemplo de uso, emulando el comportamiento simplificado anterior, sería el siguiente:

```

1 // Context struct
2 struct foo{
3     double data;
4 };
5
6 // Subscriber callback
7 static CallbackResponse Callback(cRosMessage *msg, void*
8     data_context){
9     foo *context = (foo *)data_context;
10    cRosMessageField *field = cRosMessageGetField(msg, "data"
11        );
12    if (field != NULL) {
13        context->data = field->data.as_uint32;
14    }
15    return;
16}
17
18 int main(){
19     foo context;
20     // Create the subscriber with the context pointer
21     cRosApiRegisterSubscriber(node, topic, Callback, &
22         context);
23     while (true) {
24         // Show the information (or send it)
25         cout << context->data << endl;
26         sleep(1);
27     }
28 }
```

Por último comentar que claramente, al ser información compartida, podría haber problemas de sincronización a la hora de acceder a esta información. No obstante, cROS resuelve este problema internamente.

5.2.4. Envío de mensajes periódicos

Como hemos comentado en el apartado anterior, existen varias formas de gestionar el envío y la recepción de mensajes cuando las publicaciones requieren de la información recibida. Ambas acciones se pueden realizar con o sin función *callback*, mientras que debemos tomar la decisión de si la publicación debe hacerse de manera periódica o no. Este último punto ha resultado ser un punto de interés durante el proyecto, dado que dependiendo de como implementemos el envío periódico para los publisher obtendremos una gran diferencia de rendimiento. A continuación se comentarán las distintas alternativas para realizar estas gestiones, explicando en cada caso qué implica cada una de ellas.

Si realizamos una recepción sin *callback*, simplemente podremos hacer uso de la función de cROS que realiza el polling buscando nuevos mensajes en publishers con el mismo topic que el subscriber local. Esta función bloqueará la ejecución un tiempo determinado y obtendrá el último mensaje, si lo hay, que se haya publicado desde el último polling. Esto no afecta a la lógica que usemos para las publicaciones.

Si usamos una función *callback*, en los subscribers existen varias maneras de transformar la información recibida. La primera es usar envío inmediato dentro de la propia *callback*. De esta manera conseguiremos una lógica tal que recibiremos un mensaje, los transformaremos y lo enviaremos, por tanto la cadencia y periodicidad del envío de mensajes depende de estas mismas características en el lado receptor. Un ejemplo podría ser el siguiente:

```

1 // Context struct
2 struct foo{
3     int pub;
4 };
5
6 // Subscriber callback
7 static CallbackResponse Callback(cRosMessage *msg, void*
8     data_context){
9     foo *context = (foo *)data_context;
10    cRosMessage *msg_out = cRosApiCreatePublisherMessage(
11        node, context->pub);
12    cRosMessageField *field = cRosMessageGetField(msg, "data");
13    cRosMessageField *field_out = cRosMessageGetField(
14        msg_out, "data");
15    if (field != NULL) {
16        field_out->data.as_uint32 = field->data.as_uint32;
17        cRosNodeSendTopicMsg(context->pub, msg_out);
18    }
19    return;
20 }
21
22 int main(){
23     foo context;
24     // Create the subscriber with the context pointer
25     cRosApiRegisterSubscriber(node, topic, Callback, &
26         context)
27     cRosApiRegisterPublisher(node, topic, pub);
28     context->pub = pub;
29 }
```

Por el contrario, si queremos hacer independiente el envío de la recepción (o si no existe recepción), y queremos que este sea periódico con una

frecuencia establecida, podemos hacerlo utilizando tanto una función *callback* en el publisher como envío inmediato. A continuación se presenta un ejemplo también simplificado para cada una de las opciones, donde siempre se publica un valor entero de 10:

```

1 // Context struct
2 struct foo{
3     double data;
4 };
5
6 // Publisher callback
7 static CallbackResponse Callback(cRosMessage *msg, void*
8     data_context){
9     foo *context = (foo *)data_context;
10    cRosMessageField *field = cRosMessageGetField(msg, "data"
11        );
12    if (field != NULL) {
13        field->data.as_uint32 = context->data;
14    }
15    return;
16 }
17
18 int main(){
19     foo context;
20     // Publish always 10
21     context->data = 10;
22     // Create the publisher with the context pointer
23     cRosApiRegisterPublisher(node, topic, Callback, &
24         context, frequency);
25 }
```

```

1
2 int main(){
3     publisher = cRosApiRegisterPublisher(node, topic);
4     // Publish always 10
5     cRosMessage *msg = cRosApiCreatePublisherMessage(node,
6         publisher);
7     cRosMessageField *field = cRosMessageGetField(msg, "data"
8         );
9     if (field != NULL) {
10        field->data.as_uint32 = context->data;
11    }
12    // Inmediante sending
13    while (true) {
14        cRosNodeSendTopicMsg(publisher, msg);
15        sleep(1/frequency);
16    }
17 }
```

15 | }

Ambas realizan el mismo trabajo, sin embargo se ha podido comprobar experimentalmente que si usamos la segunda opción, la del envío inmediato, el rendimiento es mucho menor. En la práctica no se ha conseguido obtener una cadencia de envío de más de 2 mensajes por segundo, mientras que usando una *callback* se respeta frecuencia correctamente.

5.2.5. Otras consideraciones

Podemos lanzar un ciclo de control mediante la herramienta *roslaunch*. ROS utiliza ficheros *launch* con los que podemos definir los nodos que formarán parte del esquema de control, así como los parámetros específicos de cada uno de ellos. En cROS sin embargo no tenemos disponible esta funcionalidad, por lo que deberemos suplirla de otra manera. En nuestro caso se ha hecho uso de scripts de bash que lanzan cada uno de los ejecutables de los nodos necesarios para cada esquema de control. Los parámetros serán pasados ahora como argumentos a los ejecutables.

Otra consideración es que, en el momento en el que se escriben estas líneas, existe un bug en las librerías de cROS que hay corregir para que funcione nuestra implementación. Es necesario introducir la llave de cierre de la estructura **extern ‘‘C’’** dentro del **if** en el archivo *cros.h*, presente en la carpeta *include*. Así es como luce el archivo en el repositorio actualmente:

```

1 #ifndef INCLUDE_CROS_H_
2 #define INCLUDE_CROS_H_
3
4 #ifdef __cplusplus
5 extern "C"
6 {
7 #endif
8
9 #include "cros_api.h"
10 #include "cros_log.h"
11 #include "cros_clock.h"
12
13 #endif /* INCLUDE_CROS_H_ */
14
15 #ifdef __cplusplus
16 }
17 #endif

```

Tras solventar el error, el contenido del archivo debería ser el siguiente:

```

1 #ifndef INCLUDE_CROS_H_
2 #define INCLUDE_CROS_H_
3
4 #ifdef __cplusplus
5 extern "C"
6 {
7 #endif
8
9 #include "cros_api.h"
10 #include "cros_log.h"
11 #include "cros_clock.h"
12
13 #ifdef __cplusplus
14 }
15 #endif
16
17 #endif /* INCLUDE_CROS_H_ */

```

5.3. Diseño final de los nodos en cROS

Teniendo en cuenta las anteriores consideraciones, se ha desarrollado el sistema de control equivalente al detallado en el capítulo anterior, pero utilizando la librería cROS. Se ha intentado concentrar la lógica de los nodos lo máximo posible, dado que a menor número de nodos reducimos las comunicaciones y conseguimos mejor rendimiento. Con este tipo de decisiones buscamos maximizar el rendimiento en plataformas con capacidad de cómputo limitada como los sistemas empotrados. La nueva topología de la red ROS es, en consecuencia, similar a la original, pero más reducida. Pasemos a comentar la implementación que se ha llevado a cabo.

5.3.1. Ciclo en lazo abierto

Para el ciclo en lazo abierto, presentado en la Figura 5.1, se ha eliminado el nodo *Full State*, dado que se seguirá otra lógica para extraer resultados, y se han unificado *Wireless Robot State* y *Baxter Arm State*, debido a que no tenemos que controlar si existe retardo por conexión inalámbrica. Sin embargo, se mantiene la captura de valores reales por parte de *Baxter Arm State* para tenerlos disponibles y traducidos a la hora de recoger dichos datos. Esto no afectará al rendimiento del sistema, ya que si queremos extraer los resultados podemos ejecutar un nodo externo que acceda al topic correspondiente, bloqueándose en otro caso la ejecución de este nodo al llegar al máximo de la cola de mensajes aún sin enviar. *Sin Trajectory Generator* y *Baxter Position Command* mantendrán su lógica, dado que están generando

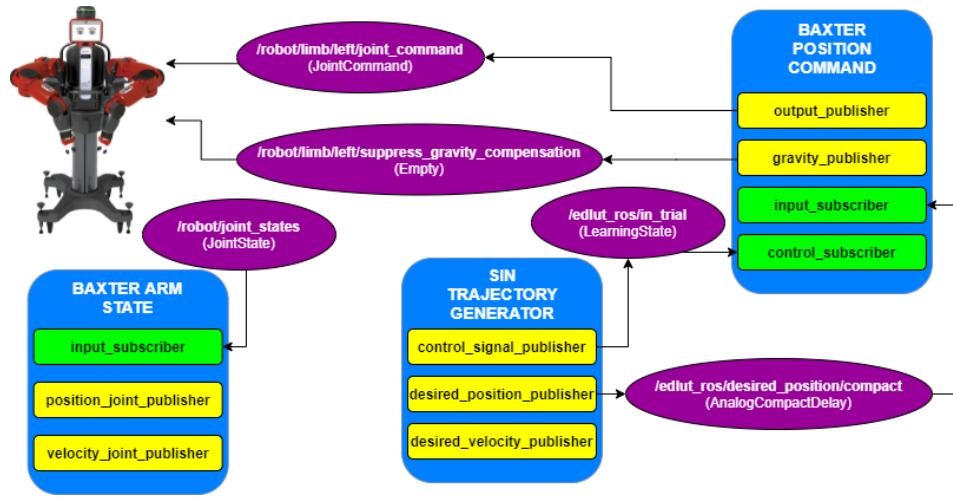


Figura 5.1: Ciclo de control en lazo abierto implementado con la librería cROS.

y traduciéndola a la información de la posición deseada para el robot.

Todos los nodos han sido implementados desde cero, basando el comportamiento en los nodos del ciclo de control original, pero adecuados a la librería cROS e intentando maximizar el rendimiento en la medida de lo posible. Como vemos, se ha reducido en gran medida la complejidad de la red de nodos, simplificando el control en este caso aún manteniendo el mismo flujo de mensajes para el robot.

5.3.2. Ciclo en lazo cerrado: controlador PD

Para implementar el control en lazo cerrado podemos partir del ciclo de control anterior e introducir un nodo con el controlador PD, *Torque Control PD*, que recoja los datos de los sensores de *Baxter Arm State* y la trayectoria deseada de *Sin Trajectory Generator*. Este calcula la fuerza que debe enviar a cada articulación mediante el controlador PD y la envía al nodo *Baxter Position Command*, que se encarga de traducir y enviar el mensaje con los comandos de fuerza al robot. El nodo *Baxter Position Command* es el mismo para el control en lazo abierto y el control en lazo cerrado, simplemente se ha implementado la capacidad de especificar si los comandos son de posición o de torque. La nueva topología de interconexión de estos nodos queda definida gráficamente en la Figura 5.2.

El nodo *Torque Control PD* ha sido implementado totalmente desde cero, dado que el original seguía una lógica incompatible de manera directa con la librería cROS. Paralelamente al sistema de control en lazo abierto, esta red de nodos más compacta que la original, evitando lógicas sin sentido para un

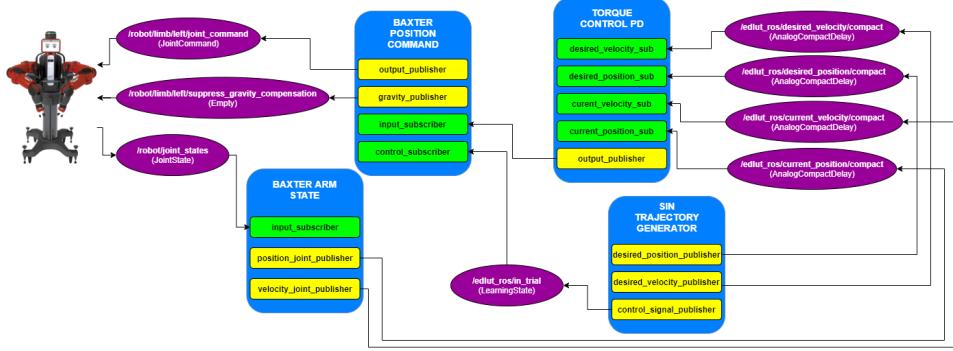


Figura 5.2: Ciclo de control en lazo cerrado con controlador PD implementado con la librería cROS.

sistema empotrado.

5.3.3. Extracción de resultados

Las métricas que tendremos en cuenta a la hora analizar el funcionamiento en la sección de resultados serán, principalmente, la posición real de cada una de las articulaciones frente a la deseada y la posición del extremo del robot en coordenadas cartesianas frente a la trayectoria deseada.

La primera se puede obtener directamente de los topics `/edlut_cros/desired_position/compact`, `/edlut_cros/desired_velocity/compact`, `/edlut_cros/current_position/compact` y `/edlut_cros/current_velocity/compact`. Por otra parte, la segunda se puede extraer del topic `/robot/limb/left/endpoint_state` que proporciona el Baxter. Ambas son extraídas con un nodo que lea el topic correspondiente e imprima el contenido en un archivo con formato *csv*. Posteriormente, de manera *offline*, se *parsearán* estos archivos con scripts de Python, que proporciona herramientas muy útiles para archivos *csv* y generación de gráficos.

Parte III

Desarrollo: simulador neuronal en plataforma empotrada

Capítulo 6

Definición del modelo de red neuronal

6.1. La neurona

La neurona es una célula especializada en recibir, procesar y transmitir información, principal componente del sistema nervioso. Este tipo de células tienen la forma definida en la Figura 6.1. Se organizan de manera que unas neuronas se conectan con otras, de forma que puedan procesar información más compleja. La conexión entre dos neuronas, que se produce desde el axón de la emisora (célula pre-sináptica) hasta las dendritas de la receptora (célula post-sináptica), es llamada sinapsis. La información dentro de la neurona está codificada en forma de potencial eléctrico, mientras que en las sinapsis, donde existe un espacio interneuronal, se transmite de manera química mediante los llamados neurotransmisores. Cuando una neurona envía información a otra, decimos que la neurona que la emite está generando un impulso nervioso, mientras que en la que lo recibe se está produciendo un evento sináptico. Una neurona puede enviar información a varias de ellas mediante las ramificaciones de su axón, y también puede recibir información de múltiples otras cuyo axón se conecte a sus dendritas. Sin embargo, en la célula post-sináptica, el sentido de la información siempre será el mismo, desde las dendritas hasta el cuerpo celular. Las conexiones neuronales no son permanentes, si no que las estructuras neuronales pueden modificarse con el tiempo creando y eliminando conexiones (o modificando sus pesos sinápticos, es decir la fuerza de cada conexión. Esta capacidad es conocida como plasticidad neuronal.

En las sinapsis químicas (la mayoría de sinapsis del sistema nervioso), su funcionamiento es conceptualmente muy sencillo, una neurona recibe un evento sináptico generado por los neurotransmisores que llegan a sus den-

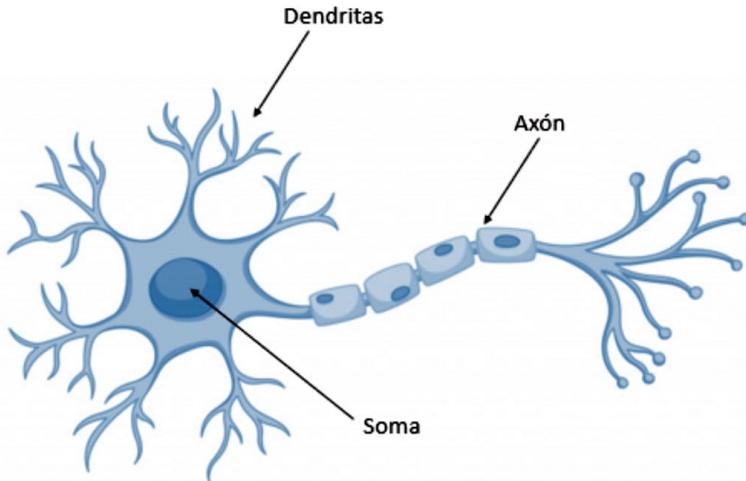


Figura 6.1: Estructura de una neurona. Fuente: <https://www.xeridia.com/blog/redes-neuronales-artificiales-que-son-y-como-se-encuentran-parte-i>

dritas provenientes de la actividad de otra neuronas. Este evento genera una corriente eléctrica que altera el potencial de membrana, del que hablaremos más adelante. En determinadas condiciones, propiciadas por los ya comentados eventos sinápticos, la neurona llegará a un estado en el que emitirá un impulso nervioso (disparo) en forma de potencial de acción a través de su axón, que llegará las neuronas con las que este forme una sinapsis.

Bajo estas condiciones la información transmitida por las neuronas se puede entender de manera binaria: o dispara (1) o no dispara (0). Por tanto, la pregunta más importante que debemos resolver para entender el comportamiento de una red neuronal es qué hace que una neurona dispare o no. La respuesta es simple a la vez que compleja: el potencial de membrana. Llamaremos potencial de membrana a la diferencia de potencial eléctrico, o voltaje, que se da entre ambos lados de la membrana celular: el citoplasma (interior), y el espacio extracelular (el exterior). Dado que los potenciales de acción provenientes de las dendritas introducen una corriente eléctrica en el cuerpo neuronal, este altera el potencial de membrana, y por tanto el hecho de que una neurona dispare o no queda determinado por llegada de eventos sinápticos. Hay dos tipos de eventos sinápticos: los excitatorios, que aumentan el potencial de membrana, y los inhibitorios, que lo disminuyen. Si dicho potencial supera un valor umbral, la neurona dispara un impulso.

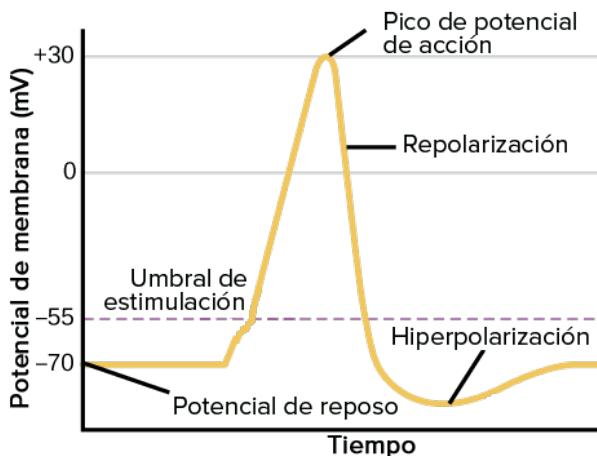


Figura 6.2: Potencial de membrana de una neurona biológica. Fuente: <https://tareasfisiologialefuentes.home.blog/2019/02/03/potenciales-de-membrana-y-potenciales-de-accion/>

Este es el comportamiento en el que se basan las redes neuronales artificiales clásicas. En este paradigma una neurona es una entidad que introduce la suma de los valores de entrada, multiplicados por el peso específico de las sinapsis de entrada, en una función de activación que hará las veces de umbral de disparo. Cuando hablamos de entrenar una red neuronal, hablamos de establecer el mejor valor posible para esos pesos específicos. Las funciones de activación de este tipo de redes suelen ser funciones sencillas, como la identidad, la tangente hiperbólica, la sigmoide o la ReLu. Las que mejor suelen funcionar suelen ser las no lineales, dado que el resto de operaciones de una red neuronal es lineal y la introducción de una no linealidad, entre otras cosas, permitirá encontrar soluciones más complejas posiblemente más cercanas a la realidad.

Este tipo de neuronas son adecuadas para una gran variedad de aplicaciones. No obstante, si lo que queremos es implementar modelos más cercanos a la realidad, debemos hilar más fino respecto al criterio de disparo. Esto implica, además de introducir el tiempo como una variable más, entender y modelar en mejor medida el potencial de membrana.

El potencial de membrana de una neurona real sigue el esquema de la Figura 6.2. La neurona permanece estable en el potencial de reposo, unos -70mV, valor que será alterado por los eventos sinápticos que reciba. Si estos impulsos generados en la neurona hacen que el valor del potencial de membrana supere el umbral de estimulación, crecerá aún más rápido en un instante y emitirá un impulso nervioso en forma de potencial de acción a través del axón, momento en el que comenzará a reducirse drásticamente. Desde el momento en el que se dispara el impulso, hasta que se recupera

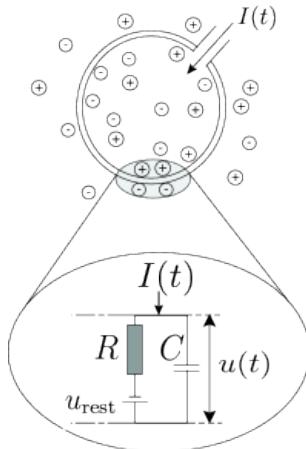


Figura 6.3: Circuito que representa un modelo LIF. Fuente: [12]

el potencial de reposo, la neurona prácticamente no se verá afectada por ningún evento sináptico sea de tipo excitatorio o inhibitorio. A este tiempo se le conoce como período refractario.

Este es el comportamiento que se intenta imitar con los modelos de neuronas relativamente complejos usados en las redes neuronales por impulsos, como pueden ser LIF o HH[33]. En nuestro caso usaremos LIF (Leaky Integrated-and-Fire) para nuestros experimentos, dado que es lo suficientemente complejo para obtener buenos resultados al tiempo que mantiene una gran eficiencia en términos computacionales. Por otra parte, existen otros modelos mucho más complejos a la vez que realistas como HH (Hodgkin-Huxley), el cual llega incluso a modelar los canales de iones que regulan el potencial de membrana.

6.2. El modelo LIF

A modo de resumen, LIF modela la neurona mediante una ecuación diferencial para el potencial de membrana y dos funciones de decaimiento exponencial para las conductancias tanto excitatoria como inhibitoria. El potencial de membrana se simula utilizando la ecuación resultante del circuito que representa el modelo LIF[12]. Este es un circuito RC(Figura 6.3) que intenta reproducir el comportamiento de la membrana plasmática de una neurona real mediante la siguiente ecuación:

$$C \frac{du(t)}{dt} = I(t) - \frac{u(t) - u_{rest}}{R}$$

$u(t)$ es el voltaje entre el interior y el exterior de la neurona, el

potencial de membrana. C es la capacitancia de membrana: la membrana plasmática de la célula se comporta igual que un condensador, es un fino aislante que separa dos sustancias electrolíticas (con carga) como son el citoplasma de la célula y el espacio extracelular. Esta depende en gran medida de la superficie de la membrana, y por tanto de la morfología de la célula concreta[23]. u_{rest} es el potencial de reposo, que está en serie con R . Esta resistencia equivale a las conductancias de reposo. En conjunto esta rama modela la corriente de fuga (*leakage*) de la neurona, la cual hace que la neurona tienda al potencial de reposo. Si además queremos modelar el efecto de las sinapsis, habría que añadir otras dos ramas al circuito en paralelo con esta. Estas ramas modelarían sendos tipos de sinapsis: excitatoria e inhibitoria, cuya corriente varía con el tiempo (recuérdense las funciones decaimiento que las modelan) y, por supuesto, de la llegada de eventos sinápticos excitatorios o inhibitorios codificados en los impulsos de entrada de la neurona. $I(t)$ es la corriente inyectada en la neurona por este tipo de eventos que, junto al potencial de membrana, depende del tiempo t en el que se producen dichos eventos.

Con ello conseguimos modelar el comportamiento de la neurona desde el potencial de reposo hasta el inicio del potencial de acción, el resto del ciclo por el que pasa este valor de voltaje, sin embargo, se simula estableciendo el potencial de membrana al potencial de reposo e ignorando cualquier entrada hasta que concluya el período refractario[20][12]. Otros modelos de neurona más complejos modelan más detalladamente este período refractado, lo que también implica el cálculo de más ecuaciones diferenciales y, por consiguiente, una mayor complejidad computacional.

6.3. EDLUT: Event-Driven LookUp Table

EDLUT es un simulador de SNNs desarrollado en la Universidad de Granada, el cual fue liberado con licencia GNU GPL 3[21]. Una de sus principales características es su velocidad, la cual le permite ejecutar redes de tamaño mediano en tiempo real. Esta capacidad hace posible el estudio de redes inspiradas en distintos subsistemas del sistema nervioso con propósitos tanto teóricos, dado que podemos simular sistemas biológicos y así entender mejor su funcionamiento, como aplicados, como puede ser el control de un brazo robot mediante alguna de estas redes neuronales biológicamente inspiradas. Existen dos esquemas de simulación disponibles en EDLUT: dirigido por eventos (*event driven*, ED) y dirigido por tiempo (*time driven*, TD).

La simulación ED, la original en EDLUT, tiene como principal característica que los cálculos necesarios para actualizar el estado de las neuronas no se realizan en tiempo de simulación. En su lugar, se precalcula el estado de la neurona discretizando los valores de los que depende a partir de unos

rangos de extensión y densidad previamente definidos. Los resultados se almacenan en tablas de consulta o *look-up tables*. Posteriormente, durante la ejecución de la simulación de la red se actualizará el estado de una neurona cuando se produzca un evento sináptico sobre ella. Los valores necesarios para dicha actualización serán consultados en las tablas y, si no están directamente para un valor específico, se realizará una interpolación lineal entre los valores inmediatamente superior e inferior. Hemos de tener en cuenta que, dado que lo que hacemos al generar las tablas es muestrear la imagen de una función, discretizándola, la precisión en el cálculo del estado de una neurona será directamente proporcional al tamaño de las tablas. Por tanto, debemos aceptar un compromiso entre la precisión del cálculo, que afectará a la bondad de la solución proporcionada por la red, y el tamaño de las tablas, que afectará tanto a la cantidad de memoria que deberemos utilizar para alojarlas, como a la velocidad de consulta, ya que tablas mayores producirán más fallos de memoria caché.

Por otro lado, la simulación TD actualiza el estado de cada neurona de manera periódica con una frecuencia establecida por el paso de integración, realizando los cálculos de las ecuaciones diferenciales en tiempo de ejecución del simulador. Por supuesto, esto requiere una potencia computacional mucho mayor cuando el paso de integración es pequeño, consiguiendo a cambio un gran incremento en la precisión comparado a los modelos ED dado que no estamos discretizando el cálculo. Esta necesidad se suple con una gran optimización del simulador para la arquitectura x86-64 y con la posibilidad de paralelizar las redes. Esta última característica está disponible mediante OpenMP, paralelizando CPU, y CUDA si se dispone de una GPU de NVIDIA. El hecho de usar la GPU introduce el problema de la sincronización CPU-GPU, resuelto utilizando técnicas eficientes para la comunicación entre RAM (la memoria de la CPU) y VRAM (la memoria de la GPU)[34].

A modo de resumen, debemos tomar una decisión a la hora de usar un modelo de simulación u otro dependiendo de la situación específica. Con ED conseguiremos rendimiento a cambio de precisión, especialmente cuando la actividad de la red no es muy alta, dado que tenemos los valores del diferencial precalculados, pero esto nos obliga a discretizar en mayor o menor medida dicha función dependiendo de la cantidad de memoria de la que dispongamos (aunque por mucha memoria que tengamos, esta nunca será infinita y la función nunca llegará a ser continua). Con TD podemos conseguir una gran precisión en los datos por el hecho de estar calculándolos directamente en tiempo de ejecución, aun cuando esto mismo implica una mayor complejidad computacional que, aunque se puede suplir con paralelismo, nunca llegará a igualar a la del modelo ED (a menos que el paso de integración en TD sea muy grande y la actividad neuronal de la red sea muy alta).

Para ejecutar EDLUT deberemos de establecer cuatro parámetros principalmente:

- La red de neuronas, para la cual deberemos establecer las capas de neuronas, los modelos de cada capa, las sinapsis existentes entre ellas y las reglas de aprendizaje que modificarán los pesos sinápticos. Los modelos deben establecer si seguirán un modelo ED o TD, pudiendo tener una red híbrida utilizando modelos ED para unas capas y TD para otras, así como los parámetros específicos de cada tipo de neurona, como los valores del potencial de reposo o el umbral de estimulación.
- Los pesos sinápticos para cada una de las sinapsis previamente establecidas en la red. Estos serán los pesos iniciales, estos valores van cambiando si se produce plasticidad.
- El tiempo de simulación, que es el tiempo que se simulará la red a partir de la situación inicial.
- Los eventos sinápticos para las capas de entrada de la red, que deberán quedar definidos por el instante de tiempo en el que se reciben y las neuronas específicas que los experimentarán.

Tras la ejecución, EDLUT nos proporcionará principalmente tres valores: el tiempo que ha tardado en procesar la actividad de la red, los impulsos internos y los impulsos propagados. El tiempo nos servirá para confirmar si la simulación se puede dar en tiempo real, situación que se dará si este valor de tiempo es menor al tiempo de simulación establecido al comienzo de la ejecución. Los impulsos internos serán la medida de las neuronas que disparan potenciales de acción, mientras que los impulsos propagados son una medida de los eventos sinápticos producidos por esos disparos.

6.4. Un cerebelo artificial

“Durante el tiempo en el que se desarrolla el movimiento voluntario, [...] muchas más cosas suceden en el cerebro motor. Ese inicio de movimiento del brazo se realiza gracias al programa final elaborado por la corteza motora, pero este programa nunca es perfecto. [...] Para que tal cosa suceda, el programa que baja desde la corteza motora a los músculos tiene que ser constantemente modificado y actualizado mientras se está realizando el movimiento”[32].

Como bien indican estas líneas de Francisco Mora, el cerebelo es la región del encéfalo encargada de rectificar los comandos de movimiento, los cuales son enviados a los músculos mediante los impulsos generados por la

corteza motora. Para proporcionar esta corrección el cerebelo recibe entradas de otros centros nerviosos del cerebro codificando la posición deseada, y también recibe información sensorial codificando la posición real. Esta función que integra la información sensorial y la motora es la que nos permite a humanos y animales realizar movimientos suaves y coordinados de manera automática. Para ilustrar por qué un cerebelo artificial puede ser muy útil como regulador de un sistema de control podemos establecer un paralelismo con un brazo real[32]. Si queremos coger un vaso de agua, la trayectoria deseada será la que lleve nuestro brazo hasta el vaso de agua, quedando parado justo a su lado antes de cerrar la mano. Cuando nuestro cerebro “genere” esa trayectoria comenzará el movimiento a través de nuestros “actuadores”, los músculos. Como hemos leído en las líneas anteriores, esta trayectoria nunca será perfecta. Cuando nos vamos a desviar de nuestra trayectoria, el cerebelo entra en acción y, leyendo la posición real del brazo a través de nuestros “sensores”, los sentidos, corregirá los comandos nerviosos que le mandamos al brazo a través de nuestro sistema nervioso. Del mismo modo podemos regular el movimiento de un brazo robótico haciendo que un cerebelo artificial genere los comandos motores a partir de la trayectoria deseada y la posición real de las articulaciones. Al contrario que un controlador PID, este regulador está inspirado en la naturaleza y, más concretamente, en nuestra anatomía. Esto puede ser una ventaja a la hora de controlar robots colaborativos, ya que la componente elástica de sus articulaciones pueden ser tenidas en cuenta, siendo un mecanismo similar al que usamos con nuestros músculos.

Otra característica notoria del cerebelo (en contraposición a los esquemas de control clásicos, como los PID) es la capacidad de realizar el control motor incluso cuando las líneas de transmisión de la señal (los nervios en este caso) introducen retardos significativos en la señal. Tanto las señales motoras que viajan a los músculos desde la corteza motora (eferentes) como las señales sensoriales (propioceptivas en este caso) que llegan al cerebelo (afferentes) sufren retardos proporcionales a la longitud del nervio. Gracias al modelo que el cerebelo es capaz de inferir, este adelanta sus acciones compensando estos retardos.

El cerebelo está formado por células de distinto tipo que se encarga de tareas específicas. Las más interesantes en el contexto que nos ocupa son las siguientes:

- MF, *Mossy fibers* o fibras musgosas.
- GC, *Granule cells* o células granulares.
- CF, *Climbing fibers* o fibras trepadoras.
- PF, *Parallel fibers* o fibras paralelas.

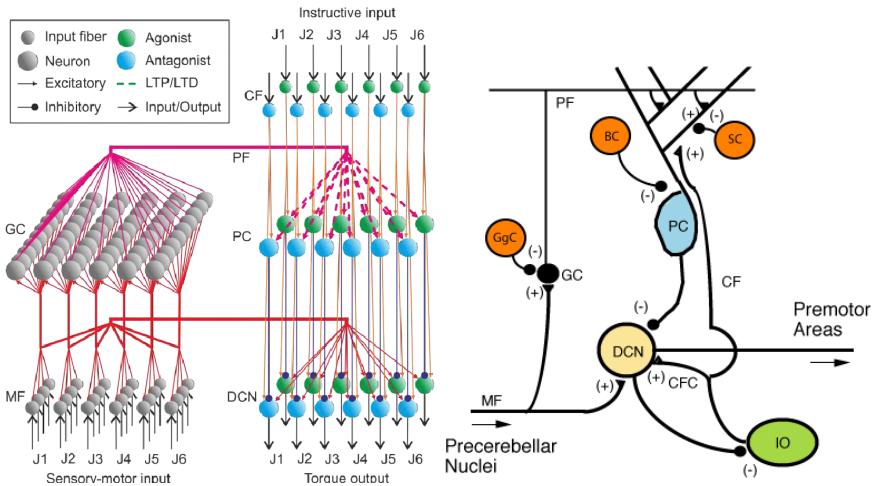


Figura 6.4: A la derecha, la microcircuituería de un cerebelo real, a la izquierda, la topología de la red neuronal artificial que lo modela. Fuentes:[1] y <http://en.wikipedia.org/wiki/File:CerebCircuit.png>, CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=8010900>

- PC, *Purkinje cells* o células de Purkinje.
- DCN, *Deep cerebellar nuclei* o núcleos cerebelosos profundos.

Conociendo las características y el propósito de cada una de estas células podemos implementar un cerebelo artificial, generando los modelos pertinentes para su uso en redes neuronales por impulsos, así como en EDLUT. En la Figura 6.4 podemos ver ambas redes, la que forma el cerebelo real y la que forma el modelo de cerebelo artificial. Para comprender esta topología de red, que usaremos para los experimentos, hagamos un breve resumen del cometido de cada una de estas capas de neuronas.

Las fibras musgosas son una de las entradas del cerebelo, es decir, a través de ellas llegarán los impulsos nerviosos externos que provocarán los primeros eventos sinápticos de la red. Estos serán codificados por la capa de células granulares, que envían sus impulsos nerviosos a través de sus axones (que son las fibras paralelas) hasta las células de Purkinje, y directamente a los núcleos cerebelosos profundos. Las células de Purkinje recibirán información en forma de potenciales de acción de las células granulares y también de las fibras trepadoras, que son otra de las entradas de impulsos nerviosos del cerebelo. Por tanto, las células de Purkinje reciben la información proveniente de las células granulares y la que proporcionan las fibras trepadoras, que envía una vez combinada y codificada a los núcleos cerebelosos profundos. Estos, combinando dichos datos con los que les proporcionan las fibras musgosas, generan la salida de la red cerebelar en forma de comando motor.

Una vez definida la red, deberemos entender cual será la entrada que debemos aportar a través de las fibras musgosas y trepadoras para imitar este comportamiento con el robot. En el cerebelo real, las primeras transportan impulsos nerviosos más relacionados con información proprioceptiva o proveniente de otros centros nerviosos, mientras que las segundas están más relacionadas con eventos correctivos, por lo que son usados para guiar la plasticidad sináptica. Por ello, las fibras musgosas serán alimentadas con los comandos de posición, deseados y reales, mientras que las fibras trepadoras contendrán la información error que hay entre ambos tipos de comandos. Finalmente, la salida de los nucleos cerebelosos profundos proporcionará los comandos finales de par, ya regulados, que serán enviados al robot. Toda esta información queda muy detallada en el artículo original[1].

Si generamos los modelos de neuronas correspondientes, los organizamos en capas siguiendo este esquema y definimos correctamente las sinapsis entre ellas, es perfectamente factible simular el comportamiento del cerebelo simplificado en EDLUT. En capítulos posteriores comprobaremos la viabilidad del uso de un sistema empotrado a la hora de simular este tipo de redes.

Capítulo 7

Puesta a punto de la plataforma NVIDIA Jetson

7.1. Instalación y puesta en marcha

El pack de desarrollo NVIDIA Jetson AGX Xavier contiene la propia placa con un sistema de ventilación preinstalado, el cable de alimentación y varios adaptadores necesarios para conectar periféricos y realizar la instalación.

La plataforma no contiene ningún tipo de sistema operativo preinstalado de serie, hubo que realizar la instalación mediante un PC utilizando lo que NVIDIA llama un *Jetpack*. Lo primero que hicimos fue conectar la Jetson al PC con el que realizamos la instalación y ejecutar la herramienta SDK Manager de NVIDIA. Esta nos detectó el dispositivo y, siguiendo los pasos, nos instaló una versión adaptada de Ubuntu 18 (en el caso de la AGX Xavier) y varias utilidades de NVIDIA como CUDA, OpenCV, cuDNN y otras utilidades para inteligencia artificial. Finalmente preguntó si queríamos especificar una dirección IP y los datos de logging.

Surgieron algunos problemas con la instalación de la última versión (Jetpack 4.4), por lo que finalmente se instaló la versión anterior (Jetpack 4.3). El principal problema era derivado del ventilador, directamente controlado por el sistema operativo de la Jetson, que no funcionaba. Sin embargo, tras la instalación de la versión 4.3 y controlándolo bajo el comando *nvpmodel*, acabó funcionando de manera correcta. Finalmente comprobamos la correcta instalación del software que necesitamos: *git* para clonar los repositorios del sistema de control y de las herramientas creadas para testear EDLUT, *gcc/g++* para compilar cROS y los repositorios antes comentados, y *CUDA* para ejecutar los modelos de neurona TD utilizando GPU.

7.2. Medidas y perfiles de consumo

7.2.1. Herramientas de medida de consumo eléctrico

Existen varias formas de medir el consumo eléctrico, de las cuales se debe elegir la más adecuada para cada dispositivo y situación. A continuación se enumeran algunas de las alternativas que se han valorado para ello.

- Monitor de consumo por software: podemos usar los propios sistemas de medida que incorpora la plataforma. Las ventaja son que no necesitaremos ningún dispositivo externo para tomar las medidas y la capacidad de automatizar la toma de medidas de una manera mucho más dirigida al problema. Las desventajas incluyen mayor dificultad para su implementación y las posibles interferencias o imprecisiones que se puedan producir debidas al propio hecho de usar el propio dispositivo que estamos midiendo para tomar las medidas.
- Monitor de consumo por hardware: la manera más fiable de medir el consumo de un dispositivo es de manera externa y totalmente transparente para él, es decir, con otro dispositivo. La desventaja es que la capacidad de tomar medidas específicas puede llegar a ser más tediosa y definitivamente menos automatizable. Existen soluciones para corriente continua, enfocadas a consumos más pequeños como pequeños microcontroladores o baterías, y para corriente alterna, pensados para consumos más grandes como los de motores o electrodomésticos. Consultando la Figura 7.1 para reconocerlos visualmente.
 - Corriente continua:
 - Vatímetro: un vatímetro es un instrumento capaz de medir la potencia eléctrica de un circuito dado, en nuestro caso la que llega desde el transformador hasta la plataforma embebida. En general, uno de estos dispositivos suele proporcionar medidas de voltaje(V), corriente(A), potencia(W) y energía consumida(Wh).
 - Pinza amperimétrica: se puede realizar esta medida con un simple polímetro, sin embargo se puede volver una tarea tediosa dado que no están pensados para ello.
 - Corriente alterna:
 - Vatímetro: también existen vatímetros para medir corriente alterna, aunque al ser esta usada en instrumentos más potentes suelen estar enfocados a medidas de mayor orden de magnitud. Por ello suelen realizar las medidas de energía consumida en kWh en lugar de en Wh.



Figura 7.1: Por orden de izquierda a derecha: vatímetro digital de corriente continua, vatímetro digital de corriente alterna adaptado a enchufes, vatímetro de pinza y SAI. Fuentes: <https://www.amazon.es/Dinam%C3%B3metro-Meter-1Pc-Wattmeter-potencia-equilibrio/dp/B07RJFXGY7>, <https://www.amazon.com/-/es/Digital-Wattmeter-Monitor-Electricity-Analyzer/dp/B07P89KZQ6>, <https://peaktech-rce.com/es/multimetros-digital-a-pinza/197-peaktech-1640-pinza-amperimetrica-vatimetro-ccca-3-digitos-17mm-1000a-40-240kw-30mm-trms.html>, [urlhttps://www.fadrell.com/etiqueta/sistema-de-alimentacion-ininterrumpida/](https://www.fadrell.com/etiqueta/sistema-de-alimentacion-ininterrumpida/)

- SAI: otra opción es usar un sistema de alimentación ininterrumpida(SAI), que suelen estar dotados de un sistema de medición de consumo. Aunque su propósito principal es la de prevenir al dispositivo que alimenta de picos y/o cortes de tensión, gran parte de ellos proporcionan medidas de consumo organizada de manera muy cómoda[46].

7.2.2. Herramientas usadas durante el proyecto

Se ha implementado un monitor de consumo por software que, aunque simple, proporciona resultados coherentes. El funcionamiento de este monitor se basa en la lectura de varias direcciones en el sistema de archivos de la Jetson, el lector puede consultar estas opciones más detalladamente en la guía que la propia NVIDIA proporciona para ello[10]. El lector puede consultar la implementación concreta del monitor en el archivo *monitor.py* presente en el repositorio que se detalla en el Apéndice C.

Hay varias razones por las que no existen garantías de que estas medidas sean del todo fiables: no queda claro si los monitores embebidos en la plataforma están midiendo toda la corriente que pasa por ella, y la propia NVIDIA confirma en la guía antes mencionada que es posible que una frecuencia de muestreo alta influya de manera no despreciable en los ciclos de CPU. Por tanto, para usar el monitor por software, debemos afrontar un compromiso precisión y exactitud en las medidas. Es por ello que se ha optado en última instancia por usar un vatímetro, colocado entre la toma

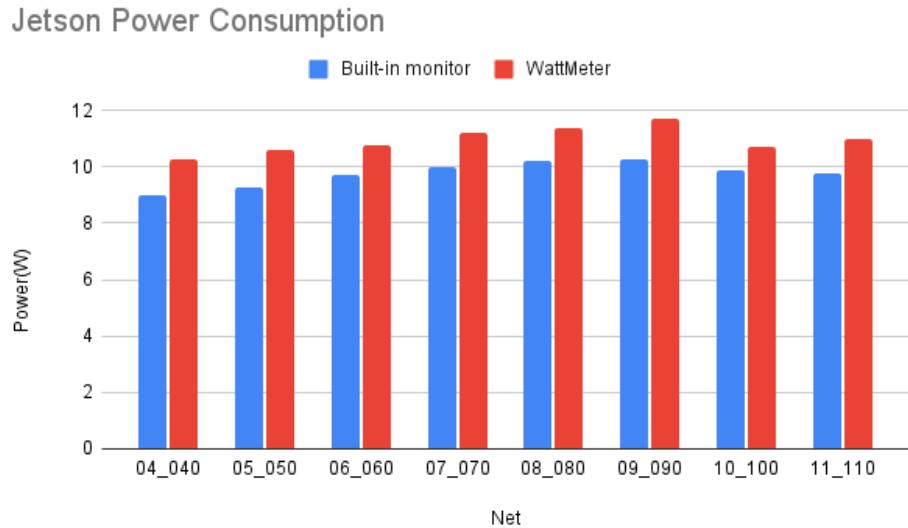


Figura 7.2: Comparación entre monitores de consumo

de corriente de la Jetson y el transformador que la alimenta..

Para las medidas con el monitor interno se están tomando los valores de potencia en vatios proporcionados por los cinco monitores internos de la Jetson y sumándolos. El valor final se calcula tomando esta medida con un periodo de muestreo lo más bajo posible, siendo el mínimo conseguido de 0.1 segundos evitando influir en las mediciones, y haciendo una media entre todas las tomas cuando finalice la ejecución. Sin embargo, para realizar las medidas con el vatímetro simplemente se ha anotado la energía global consumida al inicio del experimento con su valor al final de este, consiguiendo así la media de potencia consumida al dividir dicho valor entre el tiempo. En realidad, con el objetivo de conseguir una mayor precisión y dado que el voltaje es estable durante toda la ejecución, se ha usado como métrica la corriente consumida en *Ahsy* midiendo el tiempo en *horas* el calculo que se ha usado realmente es el siguiente:

$$V = cte ; Wh = V \times Ah ; W = \frac{Wh}{t}$$

Se realizaron varias ejecuciones monitorizando la plataforma Jetson, lanzando EDLUT para distintos tamaños de redes neuronales, intentando sacar conclusiones sobre si merece la pena usar el vatímetro o las medidas con el monitor por software son suficientes. El experimento aporta luz sobre este asunto como se puede ver en la Figura 7.2, donde podemos observar que el monitor por software no está considerando alrededor de un 10 % del consumo

capturado por el vatímetro, que es el real cuando ejecutamos la simulación de una red neuronal de distintos tamaños (como se explica en la sección 7.4). Esta diferencia se debe principalmente a que las medidas internas de la plataforma no están monitorizando toda la corriente que pasa por ella, sumado a que la frecuencia de muestreo es relativamente baja. Por tanto, es claro que lo más sensato es utilizar el vatímetro como sistema de medida, por lo que todos los valores de potencia que se indiquen sobre la Jetson a partir de este punto serán tomadas bajo estas premisas.

7.2.3. Consumo por parte de los periféricos

Otro de los experimentos que se han llevado a cabo es, dado que se quiere minimizar el consumo para posteriores experimentos, evitar el uso de periféricos tales como teclado, ratón o pantalla directamente en la Jetson. En su lugar se usará *ssh* para conectarse remotamente a la plataforma y ejecutar los experimentos. Esto puede ser incluso beneficioso en aspectos de comodidad ya que, al ser una plataforma empotrada, es posible integrarla lo máximo posible en el entorno de trabajo del robot de manera que, aunque quede físicamente inaccesible, solo necesitamos estar conectados a la misma red para acceder a ella. En relación al consumo, se llevó a cabo un experimento parecido al anterior, en el que se lanzó en EDLUT en la Jetson para varios tamaños de redes neuronales usándola a través de sus periféricos en un caso y a través de *ssh* en el otro. Podemos ver los resultados en la Figura 7.3, es claro que recortamos alrededor de un 15 % de la potencia consumida, lo cual nos ha llevado a realizar el resto de experimentos de esta manera.

7.2.4. Perfiles de consumo

El sistema operativo de la plataforma Jetson proporciona al usuario la capacidad de cambiar entre distintos perfiles de potencia (detallados en la Figura 7.4), los cuales están orientados a tener un límite superior de consumo. Estos funcionan encendiendo y/o apagando núcleos de CPU y estableciendo frecuencias máximas y mínimas para cada elemento del SoC.

Sin embargo, se han realizado algunas pruebas de rendimiento similares a las anteriores mencionadas en este capítulo para determinar si el uso de estos perfiles puede ser beneficioso para este proyecto. El resultado es que, aunque si que se consigue ahorrar potencia, la bajada de rendimiento que se produce como consecuencia es más destacable. Es por ello que se ha optado por mantener el perfil *MAXN*, que mantiene todos estos valores al máximo. No obstante, dado que la plataforma lo permite, más adelante se estudiará la posibilidad de crear un perfil de potencia *ad hoc* que permita optimizar al máximo el hardware del que dispone el sistema operativo para realizar los

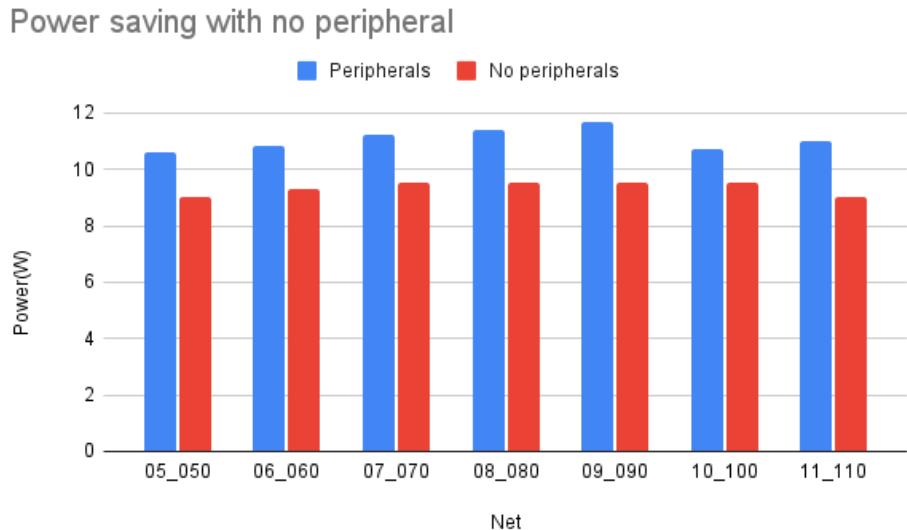


Figura 7.3: Ahorro de energía evitando el uso de periféricos

experimentos.

Hasta aquí llega el estudio de la plataforma NVIDIA Jetson, sin duda uno de los buques insignia en el mundo de los sistemas empotrados contemporáneos, que ya se está comenzando a usar en un gran abanico de industrias como solución alternativa a los computadores tradicionales.

7.3. Instalación y configuración de EDLUT

La instalación de EDLUT en la Jetson se realiza exactamente igual que en otras plataformas, basta con seguir los pasos comentados en el archivo *INSTALL* de la carpeta *Kernel*. Para este estudio se ha usado una versión de EDLUT que aún es privada, no obstante la versión pública puede encontrarse enlazada en [33].

Lo único a destacar es que es necesario tener instalados *gcc* y *CUDA* si aún no lo están(en la Jetson si lo están), y que ARM no soporta una librería presente en EDLUT, por lo que antes de instalarlo debemos comentar la línea en la que se incluye la cabecera *xmmmintrin.h* en *EDLUTKernel.cpp*. Para realizar los experimentos con la misma configuración con la que se ha realizado más adelante es necesario activar *SSH* y lanzar las ejecuciones desde otro dispositivo con todos los periféricos desconectados.

NVPModel clock configuration for Jetson AGX Xavier 16GB and 32GB								
Property	Mode							
	MAXN	10W	15W	30W	30W	30W	30W	15W *
Power budget	n/a	10W	15W	30W	30W	30W	30W	15W
Mode ID	0	1	2	3	4	5	6	7
Online CPU	8	2	4	8	6	4	2	4
CPU maximal frequency (MHz)	2265.6	1200	1200	1200	1450	1780	2100	2188
GPU TPC	4	2	4	4	4	4	4	4
GPU maximal frequency (MHz)	1377	520	670	900	900	900	900	670
DLA cores	2	2	2	2	2	2	2	2
DLA maximal frequency (MHz)	1395.2	550	750	1050	1050	1050	1050	115.2
PVA cores	2	0	1	1	1	1	1	1
PVA maximal frequency (MHz)	1088	0	550	760	760	760	760	115.2
Memory maximal frequency (MHz)	2133	1066	1333	1600	1600	1600	1600	1333
SOC clocks maximal frequency (MHz) <i>All modes</i>	adsp 300 ape 150 axl_cbb 408 bpmp 896 bpmp_apb 408 display 800 display_hub 400	csi 400 host1x 408 isp 1190.4 nvdec 1190.4 nvenc 1075.2 nvjpg 716.8 pex 500	host1x 408 isp 1190.4 nvdec 1190.4 nvenc 1075.2 nvjpg 716.8 pex 500	rce 819.2 sce 729.6 se 1036.8 tsec 1036.8 vi 998.4 vic 1036.8				

* The default mode is 15W (mode ID 7). Default mode is intended to improve desktop application performance. PVA and DLA are not in use and run at minimal frequency.

Figura 7.4: Perfiles de potencia por defecto para la NVIDIA Jetson. Fuente: documentación oficial[9]

7.4. Test preliminares: topología de las redes

Una vez hecha la instalación ya podemos comenzar a utilizar EDLUT, en nuestro caso a partir de la consola. Como ya hemos explicado en capítulos anteriores, para que EDLUT funcione hay que introducirle como argumentos un mínimo de dos archivos: tanto la definición de la red como sus pesos.

En el archivo de red se define la topología de la red como tal. En nuestro caso se seguirá el esquema del cerebro comentado capítulos atrás, pero alterando el número de neuronas en cada capa, aunque manteniendo las proporciones. También se definen las leyes de aprendizaje y las conexiones entre neuronas o sinapsis. Para los experimentos que se van a realizar con EDLUT en la Jetson se han usado distintas definiciones de redes detalladas en la Tabla 7.1, proporcionadas por el grupo de investigación. Mantendremos las mismas seis leyes de aprendizaje para todas las redes, pero se altera el número de neuronas y las conexiones entre ellas.

Los pesos no tienen demasiado interés, son los pesos que le fueron asignados en su momento a estas redes. Otro archivo que se le puede pasar a EDLUT, aunque opcional, es el de la actividad de entrada de la red. Podemos introducir impulsos en las neuronas de entrada definiendo las neuronas que recibirán el impulso, el instante de tiempo en el que lo recibirán y cuantos impulsos recibirá cada una. Más información sobre como se generan estos

	04_040 (2352)	05_050 (4770)	06_060 (9000)	07_070 (15834)
# Mossy Fibers	96	120	144	168
# Climbing Fibers	240	300	360	420
# Granule Cells	1536	3750	7776	14406
# Purkinje Cells	240	300	360	420
# DCN	240	300	360	420
# total neurons	2352	4770	9000	15834
# synapses	398784	1177200	2883744	6180384
	08_080 (26208)	09_090 (41202)	10_100 (62040)	11_110 (90090)
# Mossy Fibers	192	216	240	264
# Climbing Fibers	480	540	600	660
# Granule Cells	24576	39366	60000	87846
# Purkinje Cells	480	540	600	660
# DCN	480	540	600	660
# total neurons	26208	41202	62040	90090
# synapses	11988864	21533904	36386400	58506624

Cuadro 7.1: Redes usadas en los experimentos con EDLUT en la NVIDIA Jetson

spikes puede ser encontrada en el anexo C.

Las redes que usaremos para los experimentos, comentadas en la Tabla 7.1, han sido desarrolladas para otros experimentos por la Universidad de Granada, así como los pesos. En este proyecto sólo generaremos las entradas. Las células más abundantes son las granulares, por lo que serán las que configuraremos en ED, TD-CPU o TD-GPU dependiendo del modelo de neurona que queramos probar, por lo tanto cuando se hable de estos tres modelos a lo que en realidad nos referimos es al modelo de neurona de las células granulares. Para los modelos TD no hay que hacer más que especificar el tipo concreto de neurona en el archivo de definición de la red; sin embargo, para ED el simulador hará uso de un conjunto de tablas de consulta que codifican el comportamiento de cada modelo de neurona. Estas tablas deben estar generadas antes de iniciar la simulación. Además, el simulador debe conocer a qué dimensión de la tabla se corresponde cada variable de estado de la neurona con el propósito de poder usarlas correctamente. Por tanto cada modelo de neurona tiene asociado un fichero de texto donde se definen estas asociaciones y es este fichero el que se especifica en el fichero de red. Los valores concretos de las variables de estado de la neurona que son previamente simuladas, y cuyos resultados son incluidos en las tablas, pueden estimarse a partir de un histograma calculado durante la ejecución de un modelo TD. Por otro lado, para la conductancia inhibitoria se esta-

blecerá un rango con poca precisión dado que no es necesaria, ya que es el modelo TD el que se utiliza para simular neuronas con ese tipo de sinapsis.

7.5. Test preliminares: impulsos de entrada

Un parámetro importante a la hora de medir el rendimiento de EDLUT es la actividad de entrada de la red. Básicamente, cuando hablamos de una entrada en EDLUT, estamos hablando de impulsos nerviosos o inimpulsos nerviosos o spikes que llegan a la primera capa de la red neuronal y se propagan (o no) por su interior. Es importante comprobar si la inclusión de estos impulsos de entrada influye o no en la bondad de los experimentos preliminares. Cuando se habla de test preliminares, lo que se quiere decir es que se van a realizar test ejecutando EDLUT de manera aislada en la Jetson con la idea de comprobar si es viable su uso con el ciclo de control en tiempo real.

Parece obvio que, si introducimos una carga de entrada, EDLUT experimente una mayor carga computacional. Esto es correcto para simulaciones ED debido a que este no requiere la actualización constante de las neuronas, si no que actualiza dicho estado sólo cuando se produce un evento de entrada para dicha neurona (como su propio nombre indica). Sin embargo, en el caso de la simulación TD, se calcula el estado de las neuronas a cada paso de tiempo sin tener en cuenta si este ha cambiado o no. Por tanto sería posible realizar estos experimentos sin tener que generar inputs simulados. Se ha realizado el experimento de lanzar redes de distintos tamaños con spikes de entrada en un caso y sin ellos en el otro. Con ello se ha comprobado, como se puede ver en la Figura 7.5, que la inclusión de carga en la red sí que influye en gran medida en las ejecuciones(nótese que los gráficos tienen distintas dimensiones en el eje *y*).

Este comportamiento se debe a que EDLUT ajusta el paso de integración de manera dinámica en simulaciones TD[33], por lo que el coste computacional crecerá de manera notable dependiendo de la carga de spikes que maneje. Por tanto, los siguientes experimentos serán realizados con entradas obtenidas de una simulación, es decir, generaremos los spikes que debería de enviar el robot en un archivo. De esta manera conseguimos observar el comportamiento de EDLUT en un supuesto caso real sin hacer uso por el momento del sistema de control implementado en cROS.

7.6. Test preliminares: paralelización

Otra de las opciones que había que explorar es el uso de *multithreading*. Es obvio que si tenemos un programa costoso computacionalmente, como

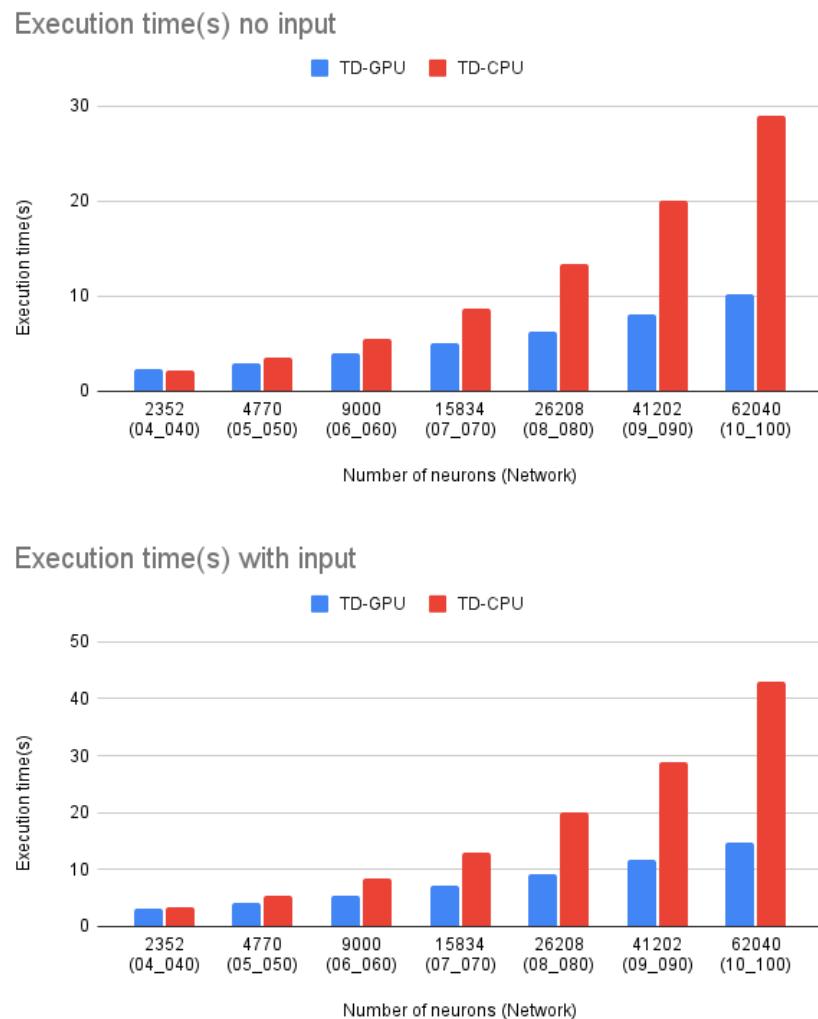


Figura 7.5: Arriba, los tiempos de ejecución para EDLUT sin generar spikes de entrada, abajo, el mismo experimento introduciendo inputs.

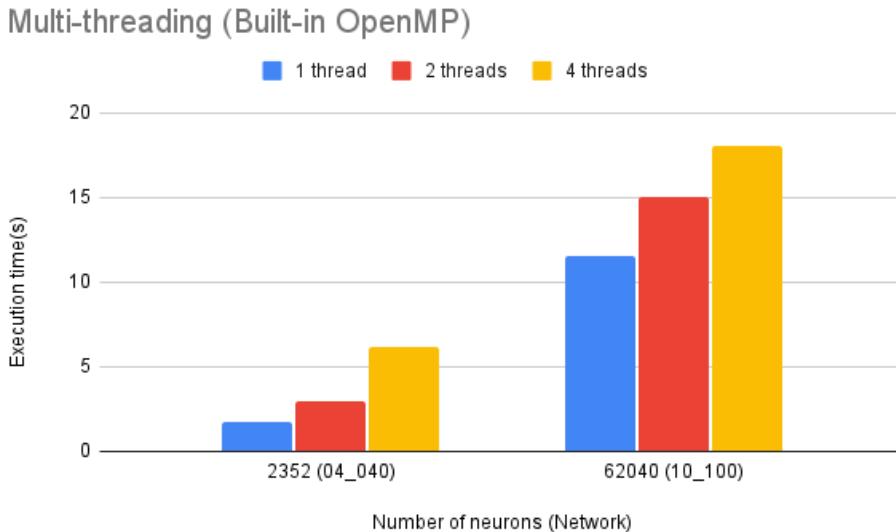


Figura 7.6: Comparación de ejecuciones usando multi-threading.

es EDLUT, y que además se presta a paralelizar cálculo, es un hecho que usar varios núcleos del procesador en lugar de uno (de manera adecuada) va a proporcionar algo de potencia extra. Es por ello que se diseñó este experimento en el que vamos a simular dos redes, una relativamente grande y otra pequeña, configurando EDLUT para que use OpenMP con distintas opciones de multi-threading. De esta manera podremos ver como se desempeña la Jetson en este escenario. Los resultados de este experimento están presentes en la Figura 7.6.

Como vemos, no solo no se gana rendimiento usando varios núcleos, sino que obtenemos una pérdida de rendimiento considerable. Esto puede ser contraintuitivo, pero tiene una explicación sencilla: EDLUT está muy optimizado, a bajo nivel, para la arquitectura x86 (y CUDA en el caso de usar GPU). Por tanto, al estar usando una arquitectura ARMv8, obtenemos un mal resultado paralelizando dado que el tiempo de sincronización entre hebras crece por encima del ahorro conseguido al dividir el coste computacional, probablemente debido a que la gestión óptima de memoria para ambas arquitecturas debe ser notablemente distinta.

7.7. Test preliminares: ED vs TD

Dado que EDLUT soporta simulaciones *event-driven* y *time-driven*, se han realizado experimentos para las tres opciones ya detalladas en capítulos

anteriores (ED, TD-CPU y TD-GPU) con el objetivo de elegir la mejor opción para la Jetson. Además, para el caso de ED también se van a probar los distintos tamaños de tabla, para las cuales escalaremos el tamaño de la fila que almacena valores para la ecuación principal de cada tabla y, en menor medida, el de las filas que almacenan el resto de ecuaciones (los valores de tiempo serán constantes). Los tamaños concretos son ED-LIF1 (89 celdas para la ecuación principal, 24 para el resto), ED-LIF2 (126 y 32), ED-LIF3 (156 y 39), ED-LIF4 (168, 52) y ED-LIFX (1680, 260). Durante el experimento se ha ejecutado EDLUT en la Jetson para todas estas opciones para un tiempo de simulación de 10s (Tsim), excepto para la medida de potencia, que para conseguir una precisión aceptable se han ejecutado para tiempos simulación mucho más largos. Esto podemos hacerlo dado que el consumo en W no varía con el tiempo durante la ejecución y estamos calculando el consumo medio.

En la Figura 7.7 podemos ver los tiempos de ejecución obtenidos durante el experimento, donde podemos ver que claramente la opción de TD-GPU es más rápida que el resto en todos los casos. Sin embargo, TD-CPU es la más lenta en todo caso, mientras que ED queda entre ambas en capacidad de computación. Otra de las conclusiones que podemos sacar del gráfico es que los tamaños de las tablas en ED son prácticamente irrelevantes a la hora de comparar tiempos de ejecución en este contexto, por lo que para el resto solo usaremos ED-LIF1. Esto es lógico dado que, comparando ED y TD-CPU, la primera solo tiene que consultar los datos precalculados mientras que la segunda debe realizarlos en tiempo de ejecución, y TD-GPU, aunque tiene que calcularlos también en tiempo de ejecución, es mucho más rápida debido a la gran capacidad de paralelización de la que se dispone al utilizar CUDA.

Por otra parte, en la Figura 7.8 podemos apreciar el consumo medio en W para las tres opciones durante los experimentos ya comentados. Es claro que, mientras que TD-CPU y ED consumen alrededor de 8 W, TD-GPU consume entre un 18,5 % y un 23 % más (9,5 W-10,5 W). Esto es lógico dado que para las dos primeras se está usando sólo un núcleo de CPU, siendo irrelevante como se esté usando, mientras que para la última se está usando además la GPU que tiene un consumo extra considerable. Se puede ver que el consumo crece con el tamaño de la red en el caso TD-GPU, cosa que también ocurre en los otros dos casos en menor medida.

Si comparamos entonces los impulsos por segundo (spikes/s), como se puede ver en la Figura 7.9, podemos deducir que lógicamente TD-CPU tiene menor rendimiento dado que consume la misma potencia que ED pero es mucho más lenta que las otras dos. Una de las razones por la que ED genera muchos más impulsos por segundo que el resto, sobre todo para las redes más pequeñas, es que también genera más impulsos para el mismo tamaño de red. Se debe tener en cuenta que ni los esquemas de simulación *time-driven*

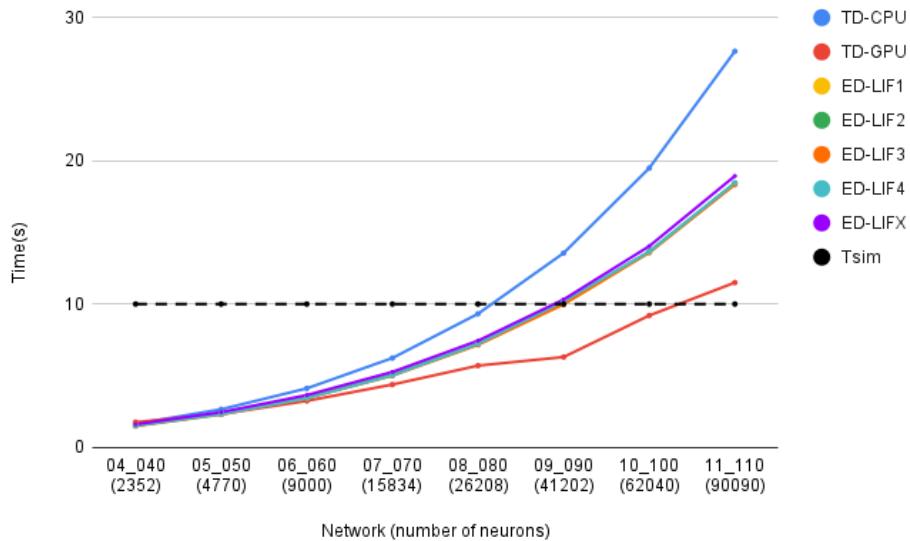


Figura 7.7: Comparación de tiempos de ejecución para redes de varios tamaños usando las tres opciones y distintos muestreos de ED. Se marca en línea negra discontinua el tiempo simulado en las redes, que ha sido en todo caso de 10 segundos.

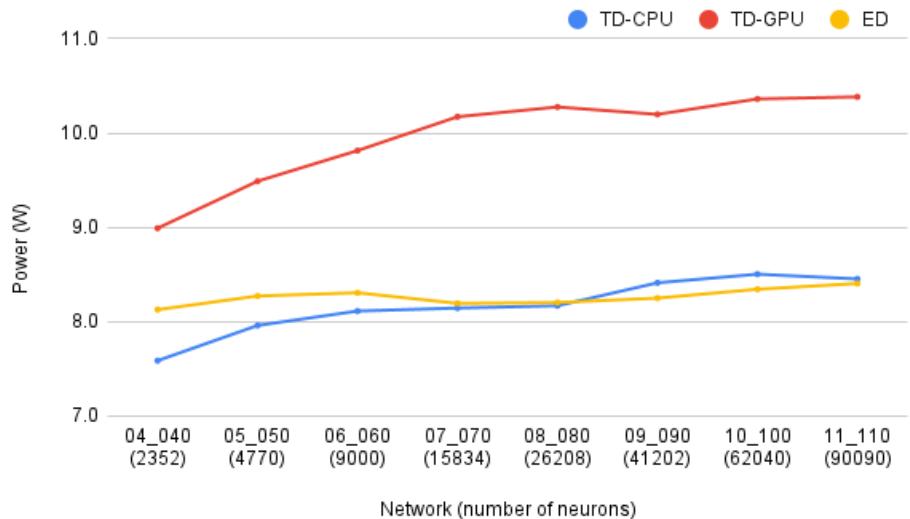


Figura 7.8: Comparación de potencia consumida en la Jetson para redes de varios tamaños usando las tres opciones.

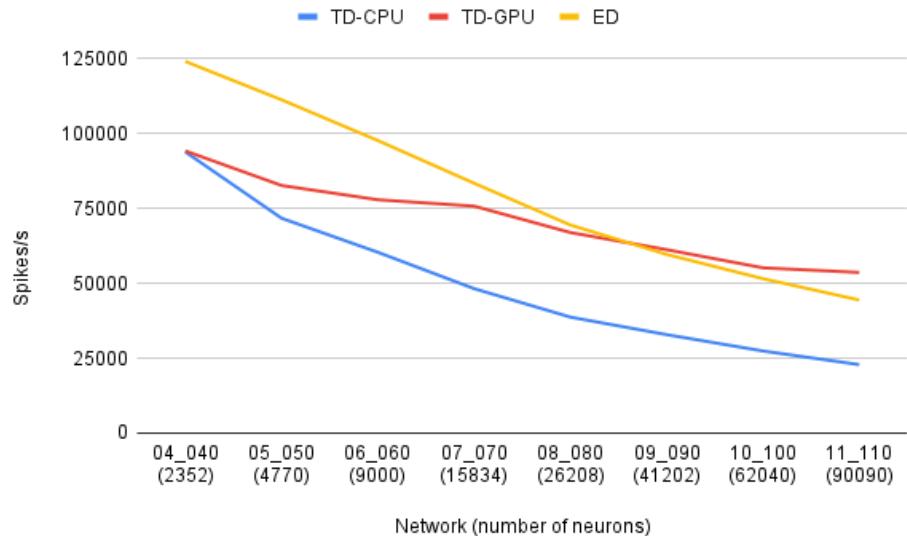


Figura 7.9: Comparación de spikes/s para redes de varios tamaños usando las tres opciones.

ni los *event-driven* realizan simulaciones exactas. En los primeros porque el paso de integración no se puede reducir demasiado, y en los segundos porque las tablas no se pueden hacer excesivamente grandes. A pesar de ello, vemos que a partir de la red *08_080* (26208 neuronas) aproximadamente TD-GPU tiene mayor rendimiento a pesar de generar menos spikes para el mismo tamaño de red.

7.8. Test preliminares: densidad de spikes

Además del tamaño de la red, hay otra métrica que puede influir en el rendimiento de cada uno de los esquemas de simulación ya comentadas, y es la densidad de llegada de los impulsos. Para este experimento se lanzó varias veces la misma red con cada uno de los esquemas, pero esta vez variando la frecuencia a la que llegan los spikes desde el robot. Cuando hablamos del período (o frecuencia) de llegada de spikes estamos hablando del tiempo que transcurre entre los eventos de comunicación con el robot, cuando se reciben los impulsos de entrada en la red y se generan los nuevos comandos robot para el siguiente período (intervalo inter-spike). Podemos ver claramente en la Figura 7.10 que el mejor tiempo de ejecución en cualquier caso se consigue cuando los impulsos llegan cada 2ms y se mantiene estable a medida que este período crece, excepto para ED que van bajando a medida que baja la frecuencia de llegada de impulsos. Esto es lógico dado que para TD se recal-

cula en todo momento el estado de las neuronas sin tener en cuenta si este cambia o no, es decir, se procesan los impulsos según su etiqueta de tiempo. Por otro lado, para ED solo se recalcula si el estado de la neurona cambia, por lo que los impulsos se procesarán uno tras otro lo más rápido posible. En períodos por debajo de 1ms los tiempos crecen muy rápidamente, probablemente debido a que es un período demasiado pequeño para recalcular el estado de las neuronas, y no se reflejarán en las gráficas.

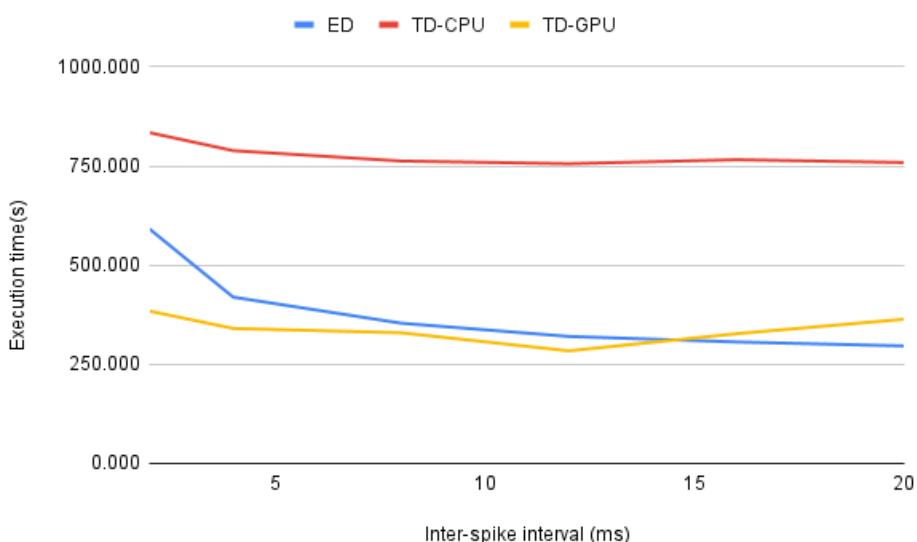


Figura 7.10: Comparación de tiempo de ejecución para distintos intervalos inter-spike usando las tres opciones.

Algo a destacar es que, como podemos ver en la Figura 7.11, el consumo de potencia no se ve aparentemente afectado por la frecuencia de llegada de impulsos.

7.9. Rendimiento en términos de potencia computacional y consumo

Obtenidos los resultados de los experimentos anteriores vamos a pasar a representarlos en términos de tiempo de ejecución y consumo. Las siguientes gráficas muestran la capacidad de procesar impulsos frente a la potencia consumida para ello, este dato se ha obtenido dividiendo el número de impulsos producidos en una ejecución entre el tiempo usado en el proceso, todo ello dividido entre la potencia media consumida del caso concreto de red y condiciones. El resultado lo obtenemos en $\frac{\text{spikes}}{W_s}$, es decir, el número

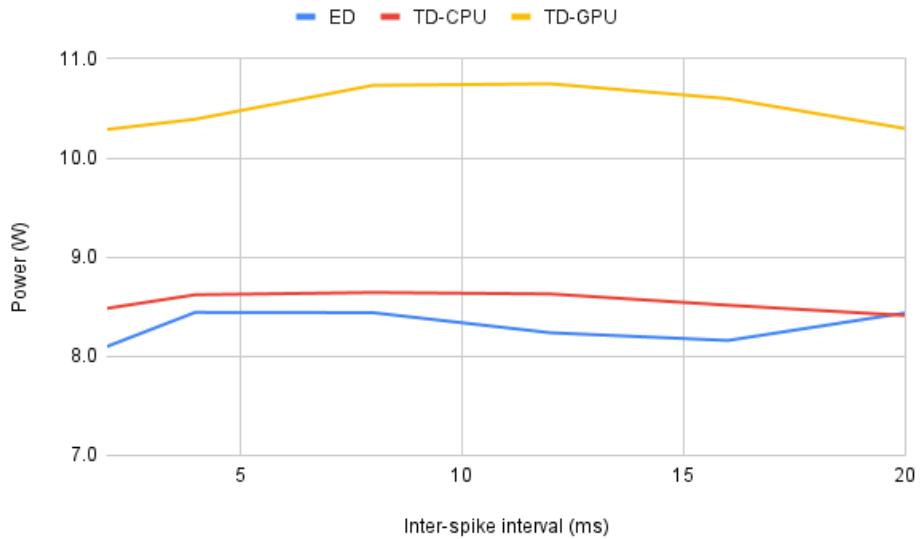


Figura 7.11: Comparación de potencia consumida para distintos intervalos inter-spike usando las tres opciones.

de spikes por unidad de energía.

En la Figura 7.12 se muestra la comparación entre redes de distinto tamaño, como podemos ver la eficiencia baja a medida que crece la red, siendo ED la que decrece con mayor pendiente aunque la que también está por encima de las TD. TD-CPU, como se podía esperar por los resultados anteriores, es la peor en todos los aspectos, lo cual era esperable dado que engloba la peor característica de ambas ED (no paralelización) y TD (cálculo en tiempo de ejecución).

Por otro lado, en la Figura 7.13 se muestra la comparación entre distintas frecuencias de llegada de impulsos para la misma red. Se intuye fácilmente que la eficiencia se consigue en 2ms, hallando valores dispersos por debajo para períodos menores por lo ya comentado anteriormente y reduciéndose la eficiencia en gran medida cuando este período crece. Esto se debe a que, para la misma potencia consumida (recordemos que era constante para distintas frecuencias) se generan muchos menos impulsos.

7.10. Perfil de potencia *ad hoc*

Como última prueba se ha querido comprobar si es interesante implementar un perfil de potencia *ad hoc* para la aplicación específica de EDLUT en la Jetson. En la figura 7.14 se puede apreciar un incremento notable de

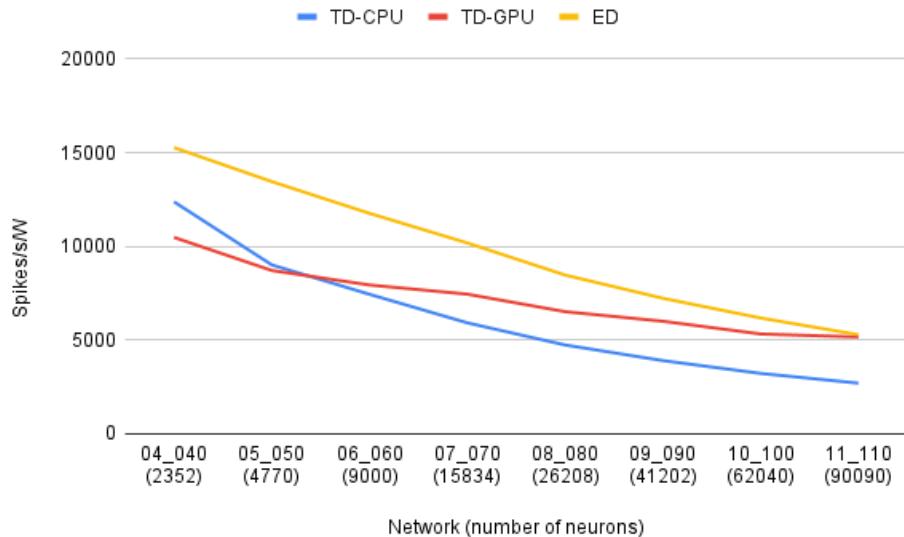


Figura 7.12: Comparación de spikes/s/W para redes de varios tamaños usando las tres opciones.

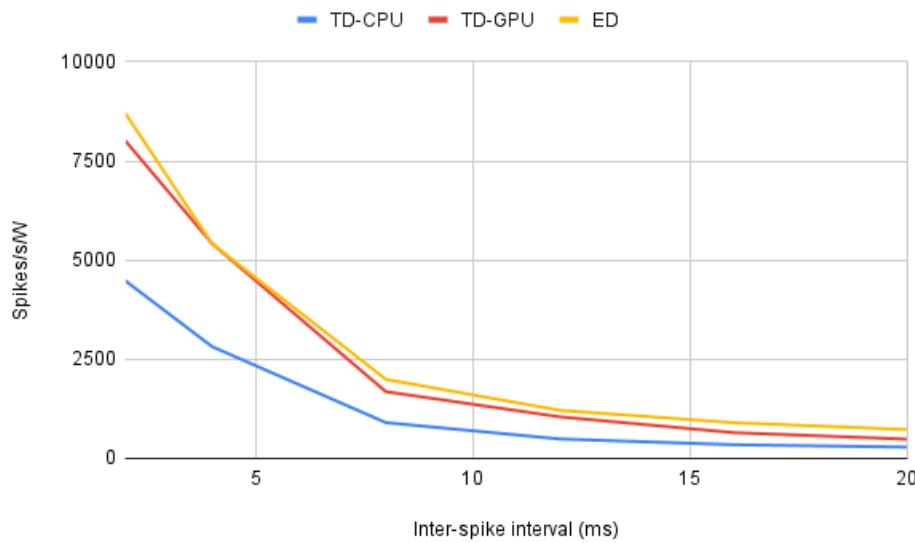


Figura 7.13: Comparación de spikes/s/W para distintos intervalos inter-spike usando las tres opciones.

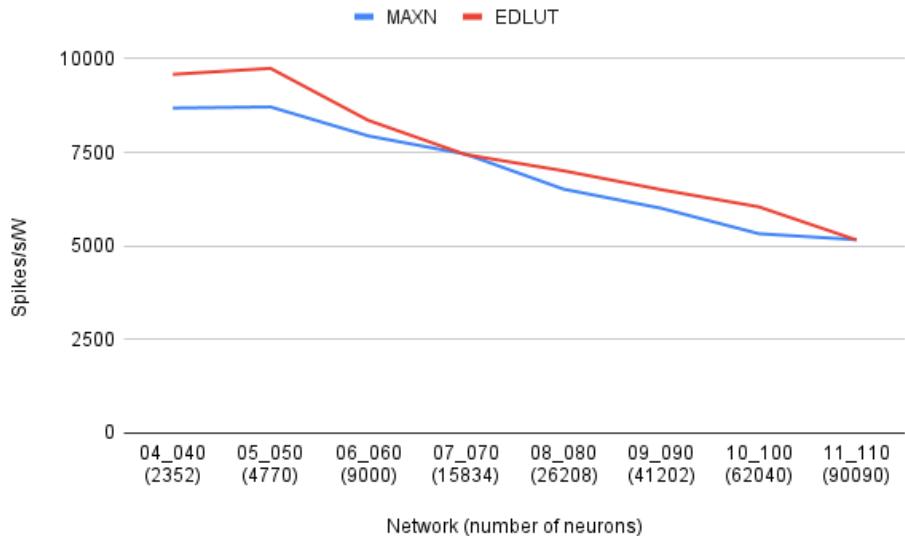


Figura 7.14: Comparación de spikes/s/W entre el perfil de potencia de máximo rendimiento de la Jetson (MAXN) y el perfil personalizado.

eficiencia, obtenido al ejecutar el mismo experimento hecho anteriormente para distintos tamaños de red, en este caso con GPU, con perfil de potencia personalizado que apaga todos los núcleos de CPU que no se usan durante la ejecución y las frecuencias de los aceleradores hardware al mínimo. Por lo tanto, dados los resultados, puede que sea interesante plantear un perfil de potencia cuando esta se vaya a usar exclusivamente para un fin específico.

Otro enfoque que se puede plantear en el futuro, dado que se puede cambiar de perfil de potencia en caliente, es la posibilidad de plantear un sistema que cambie dicho perfil en tiempo de ejecución dependiendo de las necesidades concretas en un instante determinado. Sumado a que también sería posible cambiar el modelo de neurona entre ED y TD, estamos hablando por tanto de un concepto de sistema cuyas características hardware y software cambian con el contexto.

7.11. Estudio de viabilidad

Vistos los resultados de todos los experimentos detallados en este capítulo, es claro que la ejecución del sistema de control en lazo cerrado con el cerebro artificial simulado en EDLUT es viable. Esta conclusión se puede sacar en claro sobre todo de la figura 7.7, donde se aprecia que varias ejecuciones, sobre todo con TD-GPU, obtienen el resultado de la red en menos

del 50 % del tiempo que están simulando. Por tanto, es posible introducir los nodos cROS necesarios para comunicarse con el robot en paralelo a EDLUT, teniendo en cuenta además que este se está ejecutando en un núcleo de CPU dejando libres otros siete.

Por otro lado, queda claro que los impulsos deben llegar cada 2ms y que se pueden usar las redes entre la *04_040* y *08_080* sin problema dependiendo de la finalidad concreta. Sobre si usar modelos ED o TD, es claro que TD-CPU queda descartado ya que es peor en todo caso. Sin embargo, aunque ED es mejor en términos de eficiencia, es necesario recordar que TD-GPU es mejor en tiempo de ejecución y que ED discretiza los cálculos dado que los precalcula en tablas, por lo que suele obtener peores resultados en la práctica. Por todo esto, será necesario comparar el comportamiento de ambos modelos en la caso práctico concreto, quedándonos con ED si requerimos de eficiencia y el deterioro de los resultados por la discretización no es significativo o con TD-GPU en el caso contrario.

En la sección de resultados se presentarán nuevamente los aspectos más importantes de este apartado cotejándolos con las pruebas realizadas en otras plataformas a modo de comparativa.

Parte IV

Resultados definitivos y conclusiones

Capítulo 8

Resultados

8.1. Rendimiento del sistema de control en lazo abierto

Una vez desarrollado el sistema de control para el Baxter, se han realizado experimentos con las trayectorias circular y en forma de ocho utilizando la NIVIDA Jetson como plataforma de procesamiento. La información que nos interesa obtener de estos experimentos es el error en la posición, tanto del extremo del robot como de cada una de las articulaciones, con el que podemos comprobar cómo de preciso es el control y si el control se puede llevar a cabo en tiempo real. En todo caso, el trabajo realizado se ha centrado en estimar si se pueden realizar las simulaciones en tiempo real. El error de los esquemas de control no es algo que se haya optimizado en este trabajo. Además, todos los experimentos se han realizado simulando el Baxter en Gazebo y no utilizando el robot real, lo cual se deberá tener en cuenta a la hora de interpretar los resultados.

La posición de cada una de las articulaciones, tanto la deseada como la real, se puede obtener fácilmente de los topics correspondientes que las contiene. La posición del extremo del robot nos la proporciona la propia unidad de control en otro topic, como ya hemos comentado en el capítulo referente al sistema de control. Aunque ya esté disponible y su cálculo sea transparente, dicha posición se puede obtener resolviendo el problema cinemático directo para el Baxter. Si el sistema trabaja en tiempo real se puede ver a simple vista, tanto en el movimiento del robot como en las trayectorias que vamos a obtener. Sabremos que se está realizando un control en tiempo real si el movimiento del brazo es fluido, o por el contrario se observa discontinuidad o dificultad en el movimiento. Todo el sistema de control estará siendo ejecutado en la Jetson, mientras que el robot será simulado en un PC, ambos conectados a la misma red.

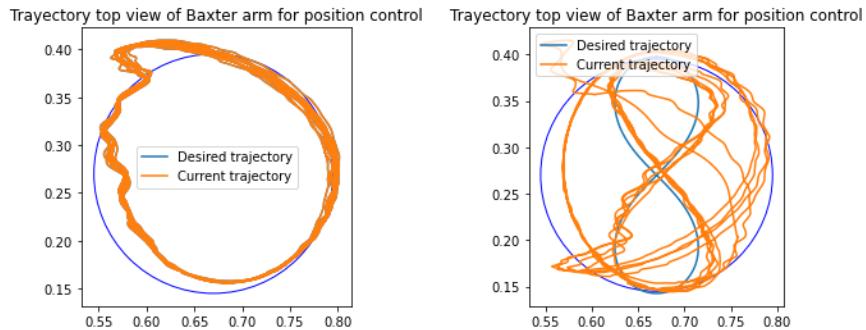


Figura 8.1: Trayectorias seguidas por el brazo del Baxter simulado para el control en lazo abierto. A la izquierda para la trayectoria circular, a la derecha para la trayectoria en forma de ocho

Ejecutando el ciclo de control en lazo abierto obtenemos la trayectoria recogida en la Figura 8.1. Se puede observar como el brazo del Baxter sigue la trayectoria no de manera exacta, pero sí de manera aproximada. Como es un control por posición, es la unidad de control del Baxter la que está calculando la potencia que tiene que aplicar en el motor de cada articulación para conseguir la trayectoria deseada. Este tipo de control, como ya hemos comentado anteriormente, es bastante preciso. No obstante, tiene la contrapartida de que es poco flexible y no está concebido para tolerar fuerzas externas.

Por otro lado, en la Figura 8.2 podemos observar la posición y velocidad de dos articulaciones a modo de ejemplo, dado que si incluimos todas las articulaciones los gráficos serían demasiado complejos. Respecto a la posición, se hace patente que el error es mínimo, lo cual se repite para el resto de articulaciones. Por tanto, la diferencia observada en la trayectoria del efecto puede ser debido a errores en la medición, en la simulación, o, de manera más sutil, en cada una de las articulaciones (que generaría unidas generaría un error apreciable). Habría que estudiar en un futuro a qué se debe este comportamiento, lo que queda patente es que, si no es un problema de los sistemas de medida que hemos utilizado, la precisión puede ser mejorada con otros esquemas de control[1]. En cuanto a la velocidad, vemos que tiene bastante ruido aunque sigue la tendencia deseada. En principio, estos cambios bruscos de velocidad se pueden deber a la manera en la que la unidad del control del Baxter maneja los motores de las articulaciones.

En definitiva, los resultados obtenidos son los esperados: un control relativamente preciso, aunque nada flexible, con pequeñas imprecisiones dado que el ciclo de control que hemos implementado en la Jetson no cuenta con realimentación. Por último, y a la vista de la trayectoria y del movimiento del robot en la simulación, podemos afirmar que el control se realiza en

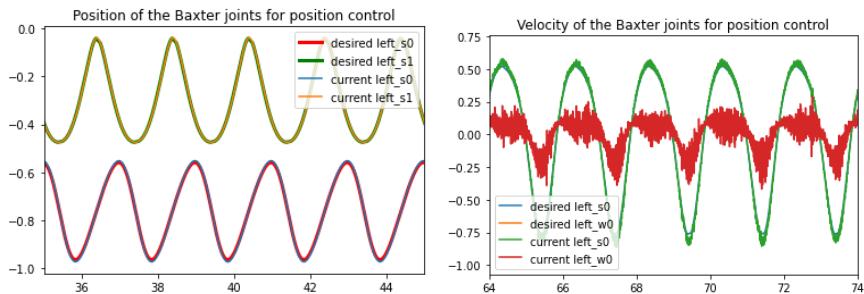


Figura 8.2: Posición y velocidad de algunas articulaciones del Baxter simulado para el control en lazo abierto.

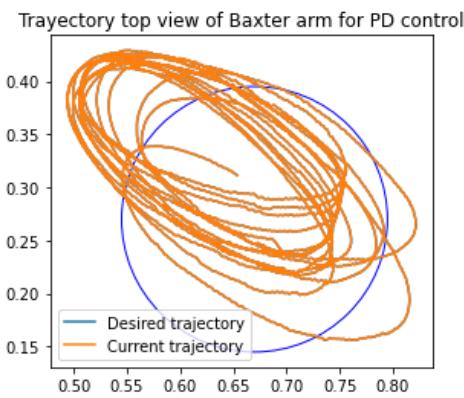


Figura 8.3: Trayectoria seguidas por el brazo del Baxter simulado para el control en lazo cerrado (trayectoria circular).

tiempo real sin ningún problema, obteniendo una fluidez sustancial.

8.2. Rendimiento del sistema de control en lazo cerrado con PD

Respecto al sistema de control en lazo cerrado, hemos obtenido las mismas métricas que en la sección anterior. La trayectoria del extremo del brazo, en la Figura 8.3, presenta un comportamiento más constante, es decir, más “limpio”, no dibuja formas extrañas como en el caso anterior. Este comportamiento es lógico dado que este paradigma de control está haciendo uso de la realimentación implementada en la Jetson y, por tanto, el movimiento no es tan forzado como el anterior.

Sin embargo, el error en la posición es mucho mayor, dibujando una ellipse en lugar de un círculo y variando su tamaño y posición. Para darle sentido

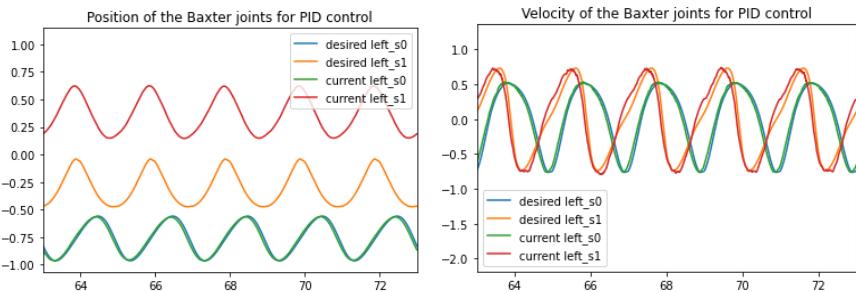


Figura 8.4: Posición y velocidad de las articulaciones del Baxter simulado para el control en lazo cerrado.

a este error debemos consultar, en la Figura 8.4, la posición y velocidad de cada una de las articulaciones. Como anteriormente, se han seleccionado dos articulación para que las gráficas sean legibles, en este caso las más relevantes. La velocidad real, como vemos, se mantiene prácticamente igual a la esperada, oscilando en pequeñas proporciones sin que esto sea relevante. No obstante, en la comparativa de posición es donde encontramos el problema. Se aprecia como algunas articulaciones están prácticamente superpuestas a las reales, pero otras, aunque mantienen la misma forma, mantienen un error permanente de *offset*. Este es el típico error por el que se introduce y que corrige perfectamente la componente integral en un controlador PID. Pese a ello, no hemos podido utilizarla ya que no nos genera buenos resultados con el Baxter, por lo que se requiere un ajuste más fino de los parámetros del controlador PD, tarea que no es objetivo en este proyecto.

En cambio, a la vista de la suavidad de la trayectoria del extremo del robot y la fluidez que se aprecia en la simulación del robot, es claro que el control se produce en tiempo real. Es decir, por tanto, la NVIDIA Jetson es capaz de ejecutar un control en tiempo real del Baxter, utilizando cROS, sin demasiado problema.

8.3. Comparativa de EDLUT-Jetson con otras plataformas

Una vez realizada la parte de desarrollo relativa al estudio preliminar de EDLUT en la NVIDIA Jetson queda claro que su uso es viable. No obstante, estos valores por sí solos aportan información incompleta, dado que además de confirmar que es posible empotrar el simulador y el sistema de control también debemos ver si es más eficiente que el uso de otras plataformas. La Tabla 8.1 muestra las especificaciones de los dispositivos que usaremos en la comparativa.

	Jetson	PC	Laptop	Pi Zero W
CPU	Carmel	i5-7600K	i7-4600M	BCM2835
CPU cores	x8	x4	x4	x1
CPU freq	2265MHz	3.8GHz	2.9GHz	1GHz
Arch	ARMv8	x86-64	x86-64	ARM11
RAM	32GB	16GB	16GB	512MB
GPU	Volta	GTX1060	-	-
CUDA cores	512	1280	-	-

Cuadro 8.1: Tabla de especificaciones de las distintas plataformas usadas en la comparativa

Las principales comparaciones se realizarán con un PC y un portátil de gama media, para 1, 2 y 4 hebras paralelizando con la opción OpenMP de EDLUT. Adicionalmente a estas dos plataformas se introducirá una Raspberry Pi Zero W, basada en ARM y con un consumo extremadamente bajo. Esto último tiene el objetivo de introducir en la comparativa otra plataforma empotrada de bajo consumo, teniendo así otra referencia de este tipo de dispositivos. Lo más sensato y correcto habría sido realizar introducir una plataforma más potente como una Raspberry Pi 4 o similares, sin embargo se ha usado la que se tenía disponible en el momento de realizar las pruebas.

Por supuesto, las pruebas se han realizado en las mismas condiciones que con la NVIDIA Jetson, lanzando varias iteraciones con distintos tamaños de red y obteniendo la media de los valores de tiempo obtenidos en cada una de ellas. Para las medidas de consumo de potencia se han lanzado ejecuciones con un tiempo prolongado para comprobar el consumo de las distintas plataformas, tomando medidas de energía (Wh) y dividiéndolas entre el tiempo total del experimento, usando un vatímetro similar al comentado durante los test preliminares pero que es capaz de realizar medidas en corriente alterna. Es necesario remarcar que no se están teniendo en cuenta la desviación típica entre las medidas con las que calculamos la media debido a que, en todo caso, es un valor suficientemente pequeño como para despreciarlo.

8.3.1. Comparativa de rendimiento y consumo

Dicho esto, comencemos el análisis de los resultados obtenidos para esta comparativa. Primeramente vamos a estudiar la diferencia de rendimiento puro entre las distintas plataformas para los tres esquemas de simulación de neuronas ED, TD-CPU y TD-GPU. Para cada una de las gráficas veremos el número de spikes por segundo que son capaces de computar las distintas plataformas en función del número de neuronas presentes en la red. Dado que en total son 8 arquitecturas (4 plataformas con 3 configuraciones diferentes de paralelización en 2 de ellas), se ha utilizado un código de colores que faci-

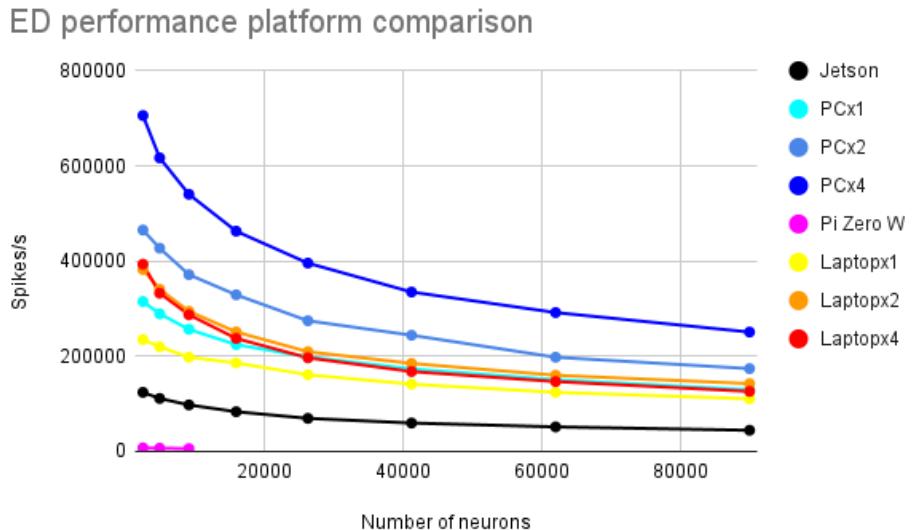


Figura 8.5: Comparativa de rendimiento para las distintas plataformas en simulación ED.

lite la lectura de los datos: negro para la Jetson, rosa para la Raspberry Pi, amarillo, naranja y rojo (colores cálidos) para las distintas configuraciones de paralelización del portátil, y cyan, azul cobalto y azul marino (colores fríos) para las del PC. La notación usada en la leyenda de las gráficas es $<\text{plataforma}>x<n>$, la cual denota a la plataforma *plataforma* ejecutando la simulación en paralelo en *n* hebras mediante la opción OpenMP de EDLUT.

En la Figura 8.5, referente al modelo de simulación ED, podemos apreciar como la Jetson queda por debajo del resto de plataformas, excepto la Raspberry Pi que tiene un rendimiento muy inferior al resto como ya esperábamos. Por encima, el PC es lógicamente superior al portátil, excepto para la opción sin paralelización que es ligeramente inferior al portátil si usa dicha opción. Vemos como se mantiene de manera coherente que la paralelización incrementa en gran medida el rendimiento para las arquitecturas para las que EDLUT está optimizado. Un hecho relevante es que en el caso del portátil usar 4 hebras en lugar de 2 empeora ligeramente el resultado. Por algún motivo, a pesar de tener un procesador de cuatro núcleos, deja de beneficiarse del paralelismo a partir de dos hebras. No se ha profundizado en este comportamiento, pero este puede ser debido a que la paralelización de estos esquemas de simulación es compleja. Por tanto, es posible que cuando ya se usen 2 hebras, añadir otras 2 no compense el coste causado por la comunicación entre todas ellas.

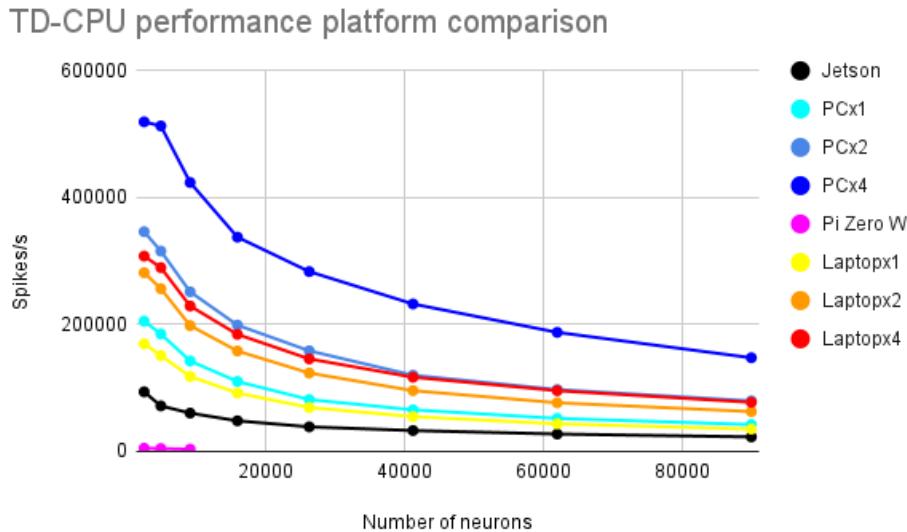


Figura 8.6: Comparativa de rendimiento para las distintas plataformas en simulación TD-CPU.

Por otro lado, en la Figura 8.6 se puede observar el comportamiento del modelo TD usando CPU. Como vemos se mantiene el mismo comportamiento que con el modelo ED, pero con algunas pequeñas alteraciones. En este caso sí que aporta valor añadido doblar el número de hebras de 2 a 4 en el portátil. También parece que las líneas de tendencia se parecen algo más, manteniendo prácticamente todas la misma forma. Tanto ED como TD-CPU parecen presentar una tendencia con forma de exponencial decreciente ($b \cdot e^{-(x+a)} + c$).

Sin embargo, en el caso del modelo TD usando GPU aparece un comportamiento distinto, como se puede apreciar en la Figura 8.7. Como es obvio solo podemos comparar con PC dado que el resto de plataformas no disponen de GPU. Se puede ver una diferencia sustancial en la tendencia de ambas plataformas, y es que mientras la Jetson mantiene una forma similar a la obtenida para otros esquemas de simulación, el PC tiene una pendiente positiva hasta cierto número de neuronas. Dicho rango varía según las hebras que utilicemos para paralelizar: hasta unas 5000 neuronas con una hebra, hasta unas 10000 con dos y hasta unas 17000 con cuatro. Este comportamiento se debe a que la GPU del PC, al tener más núcleos CUDA, no hay suficientes neuronas en las redes pequeñas para utilizarlos todos. Debido a esto, a medida que crecen las redes y se van utilizando más núcleos, el rendimiento va mejorando hasta llegar al punto en el que se usan todos los núcleos CUDA. A partir de ese momento la gráfica vuelve a adoptar la misma forma que para la Jetson y los otros esquemas de simulación, en los

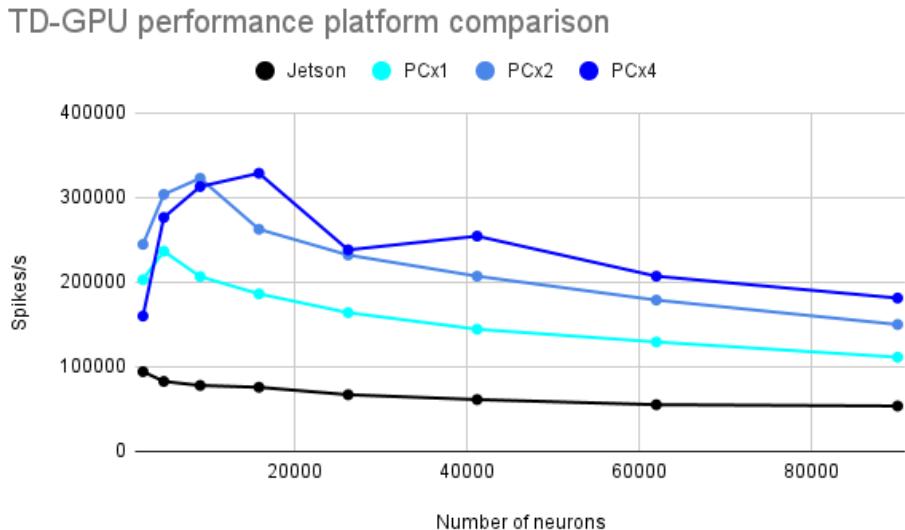


Figura 8.7: Comparativa de rendimiento para las distintas plataformas en simulación TD-GPU.

que sí se están usando todos los núcleos disponibles. También se puede ver que en los tamaños de red más pequeños el uso de cuatro hebras no es el mejor. Esto depende del compromiso paralelismo-sincronización que se asume a la hora de parallelizar. Separar las neuronas de una red supone que cada cierto tiempo debe de existir una sincronización entre las neuronas que estén separadas y que se deben comunicar entre sí. El coste en tiempo que requiere esta tarea crece en menor medida que los beneficios de la parallelización, por tanto ante tamaños más pequeños es más eficiente utilizar menos hebras y evitar los tiempos de sincronización. Nótese que la diferencia entre una, dos y cuatro hebras es debida a que con cuatro hebras las subredes resultantes son más pequeñas, y por tanto este efecto se da hasta tamaños más grandes. Por las dos razones explicadas anteriormente es por lo que el uso de OpenMP o CUDA no cobra sentido en redes más pequeñas de un valor específico.

También se aprecia un valor fuera de tendencia con cuatro hebras y 26000 neuronas. Esto es debido a que la paralelización es más efectiva para ciertos tamaños que son múltiplos del número de hebras, y en concreto el número de neuronas para el valor anormal ha debido ser un valor que no casa bien con una paralelización de cuatro hebras.

Una vez vista la comparativa de rendimiento, vemos que este depende directamente de las especificaciones de la máquina, lo cual es lo esperable. Sin embargo, el estudio que nos interesa hacer con la Jetson es una comparativa

de eficiencia rendimiento-consumo, dado que no nos interesa simplemente la potencia aislada, sino conseguir el máximo rendimiento posible con el mínimo consumo. Para ello debemos tener en cuenta el gasto de energía de cada una de las plataformas que estamos usando, medidas que podemos consultar en la Figura 8.8, donde la parte roja de las barras se corresponde con el consumo de la plataforma por el simple hecho de estar encendidas (idle), mientras que la parte azul es el consumo extra que supone ejecutar EDLUT. Cabe comentar que se está representando el consumo medio, aunque las diferencias cuando usamos CPU (ED y TD-CPU) son prácticamente despreciables y se puede suponer que el consumo no variará con los tamaños de red que manejamos. En el caso de la GPU sí que parece que una red más grande se puede paralelizar en mayor medida y por tanto hacer un mayor uso de la GPU, por lo que la variación de consumo en función del tamaño de red sí que sería relevante.

Como vemos, las plataformas que mejor especificaciones tienen también son las que más consumen. La cuestión es si el aumento de rendimiento justifica ese consumo extra. En principio debería ser más eficiente utilizar las plataformas con menor consumo, debido a que tanto los procesadores con arquitectura ARM como la filosofía de los SoCs es obtener el máximo rendimiento con un consumo mínimo. Sin embargo, las arquitecturas clásicas no tienen tan en cuenta el consumo y suelen centrarse en obtener la máxima potencia posible. Otra conclusión a la que podemos llegar, aunque obvia, es que paralelizar consume más (porque estamos usando más núcleos).

8.3.2. Comparativa de eficiencia

Como primera medida de eficiencia asumiremos $\frac{\text{spikes}}{W_s}$, es decir, cuantos impulsos nerviosos es capaz de generar una plataforma por cada Julio de energía consumido, al igual que en las pruebas preliminares que hicimos en la Jetson. Comencemos por la comparativa para el modelo de simulación ED, presente en la Figura 8.9. En lo que a eficiencia energética se refiere, vemos claramente que la Jetson es muy superior al resto para todo tipo de tamaños de redes. Es 3 veces mejor de media que el menos eficiente, que es el PC usando un sólo núcleo, y 2 veces mejor de media que el más eficiente del resto, que es el portátil utilizando dos núcleos, aunque el resto están rondando también esa franja. En el caso del modelo TD-CPU, en la Figura 8.10 vemos que la diferencia es incluso más acusada. Es más de 4 veces más eficiente que el PC con una sola hebra y una 2 veces más eficiente que el mejor del resto, el portátil con 4 hebras. En ambos casos vemos la mejora que supone la Jetson respecto del resto de plataformas a la hora de simular redes minimizando el consumo energético.

Esto se va acrecentado en la comparación entre la Jetson y PC utilizando

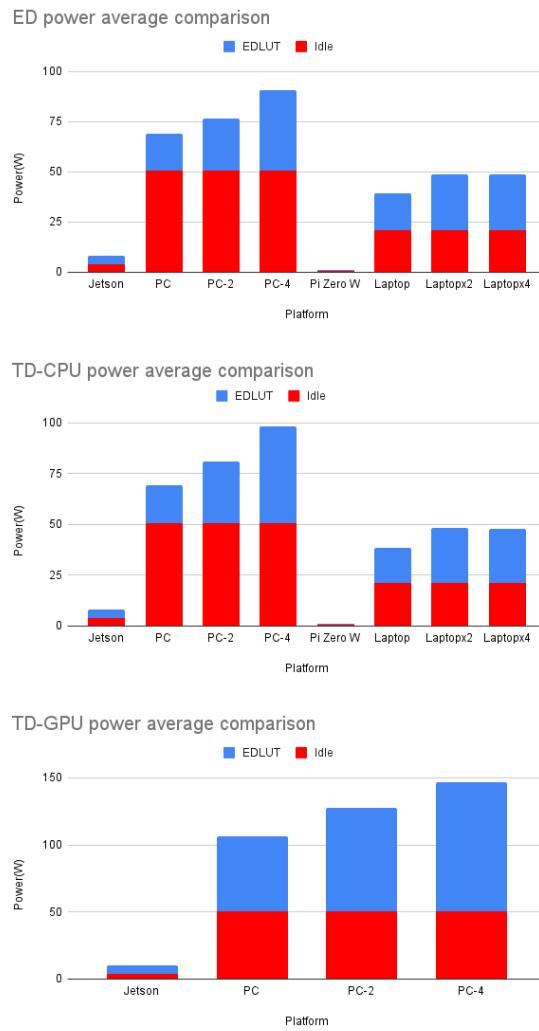


Figura 8.8: Comparativa de consumo para las distintas plataformas.

ED performance-consumption platform comparison

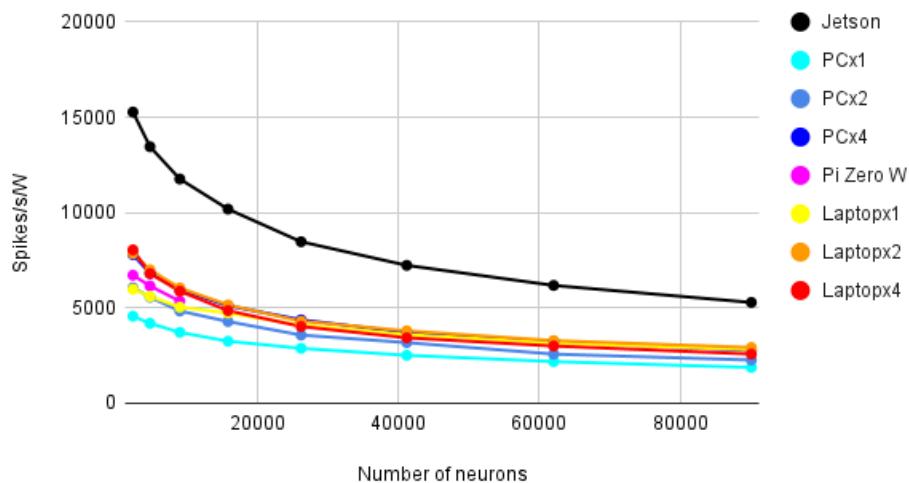


Figura 8.9: Comparativa de eficiencia para las distintas plataformas en simulación ED.

TD-CPU performance-consumption platform comparison

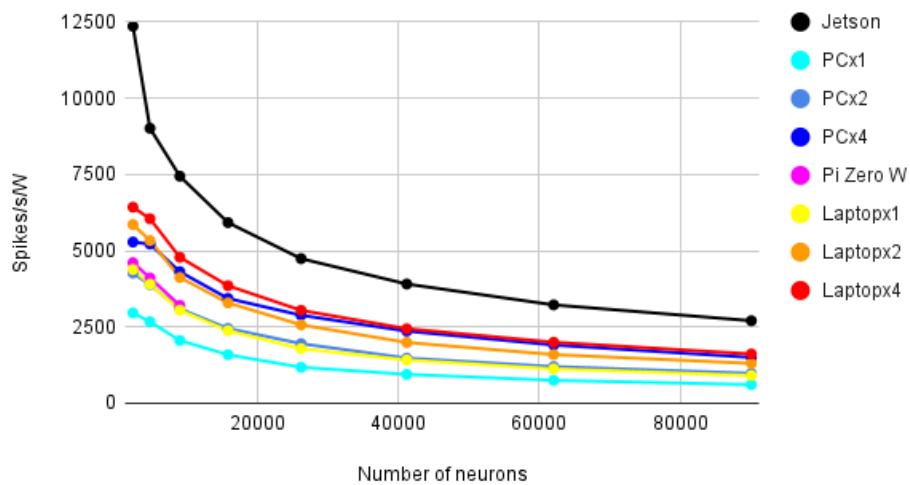


Figura 8.10: Comparativa de eficiencia para las distintas plataformas en simulación TD-CPU.

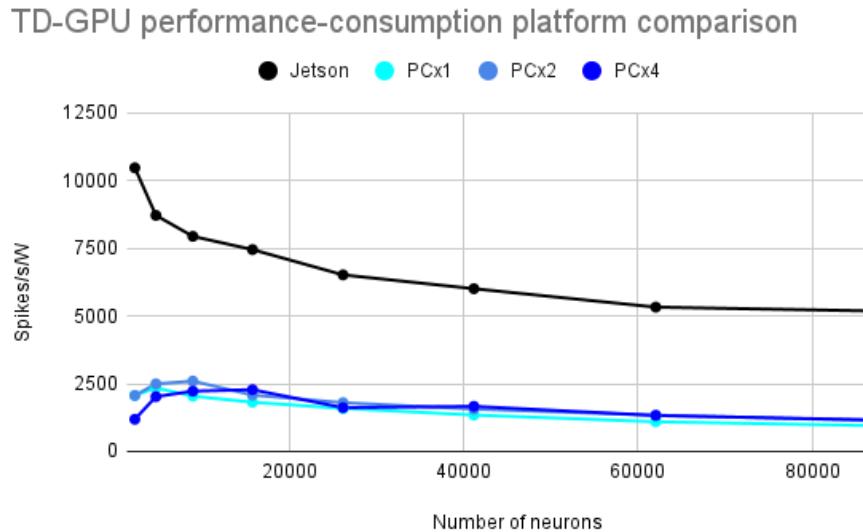


Figura 8.11: Comparativa de eficiencia para las distintas plataformas en simulación TD-GPU.

los modelos TD-GPU, presente en la Figura 8.11. Como vemos, la mejora de eficiencia de la Jetson es realmente notable, consiguiendo una ganancia respecto a PC que comienza siendo de x4 y llega incluso a x5 en los tamaños de red más grandes.

Las formas son las mismas que en la comparativa de rendimiento, dado que estamos dividiendo entre un valor, el consumo, prácticamente constante respecto al tamaño de la red. A partir de estos datos podemos afirmar que la Jetson es superior en eficiencia a otras plataformas de procesamiento generalistas a la hora de ejecutar redes neuronales con EDLUT, y por una diferencia significativa. Además, lo es al mismo tiempo que es capaz de ejecutar redes de tamaño medio en tiempo real.

8.3.3. Forma de exponencial decreciente

Durante el razonamiento que hemos estado llevando a cabo hay un aspecto importante, referente a todas las gráficas de manera general, que no hemos resuelto aún. Como ya hemos comentado, todas las gráficas tienen en general una forma característica de exponencial decreciente, incluido el PC con TD-GPU si tenemos en cuenta las razones expuestas en secciones anteriores. Sin embargo, no hemos encontrado ninguna razón para ello.

Como podemos ver ilustrado en la Figura 8.12, la cantidad de impulsos por neurona decrece a medida que aumenta el tamaño de la red. Es tan

acusada la diferencia que la red más grande genera un 90 % menos de spikes respecto a su tamaño que la más pequeña. Este comportamiento explica la forma decreciente de las gráficas.

8.4. Comparativa de EDLUT-Jetson con SpiNNaker

Finalizada ya la comparativa con otras plataformas generalistas, vamos a realizar una comparación teórica entre EDLUT en la Jetson y una arquitectura específica. SpiNNaker es una placa de circuito impreso de aplicación específica desarrollada en la Universidad de Manchester, con el objetivo de simular redes neuronales por impulsos de manera eficiente y escalable. Cuenta con 48 chips que elevan el número de procesadores a 864, todos basados en ARM. La principal ventaja de SpiNNaker es que, al estar construido exclusivamente para simular SNN, es muy eficiente en esta tarea, permitiendo además la interconexión de placas para formar redes (*clusters*) que podrán simular redes de mayor tamaño. Como principal contrapartida, la complejidad para implementar redes es mucho mayor que en una plataforma de propósito general. Todos los datos utilizados en esta comparativa están basados en un estudio de SpiNNaker[48], donde se puede encontrar más información al respecto.

Lo primero a destacar es que la red usada al obtener los datos experimentales en SpiNNaker no es la misma que hemos usado en nuestros experimentos. En SpiNNaker tomaremos como referencia una red de neuronas dividida en poblaciones (una en cada procesador). Dentro de cada una de estas poblaciones hay una conectividad de todas con todas y además cada población está conectada con otras 5 poblaciones elegidas aleatoriamente. En EDLUT simularemos la red que modela el cerebelo usada anteriormente. El número de neuronas es parecido, al igual que el número de sinapsis, en el caso de la red *10_100*, pero la topología de la red es muy distinta, y además la tasa de eventos sinápticos por segundo es diferente hasta en tres órdenes de magnitud. No obstante, sí que se proporcionan dos medidas interesantes como son la potencia consumida por neurona y la energía consumida por evento sináptico. Son interesantes porque a partir de ellos se puede calcular la energía total consumida por la red y porque, al utilizar un modelo de neurona parecido (LIF), podemos aplicarlo a nuestra red cerebelar. Esta información queda resumida en la tabla 8.2.

Al utilizar redes distintas en los experimentos, no podemos realizar una comparativa directa de consumo entre ambas plataformas. La alternativa que vamos a tomar es calcular, de manera teórica, cuál sería el consumo de la red cerebelar en SpiNNaker realizando los experimentos que hemos hecho

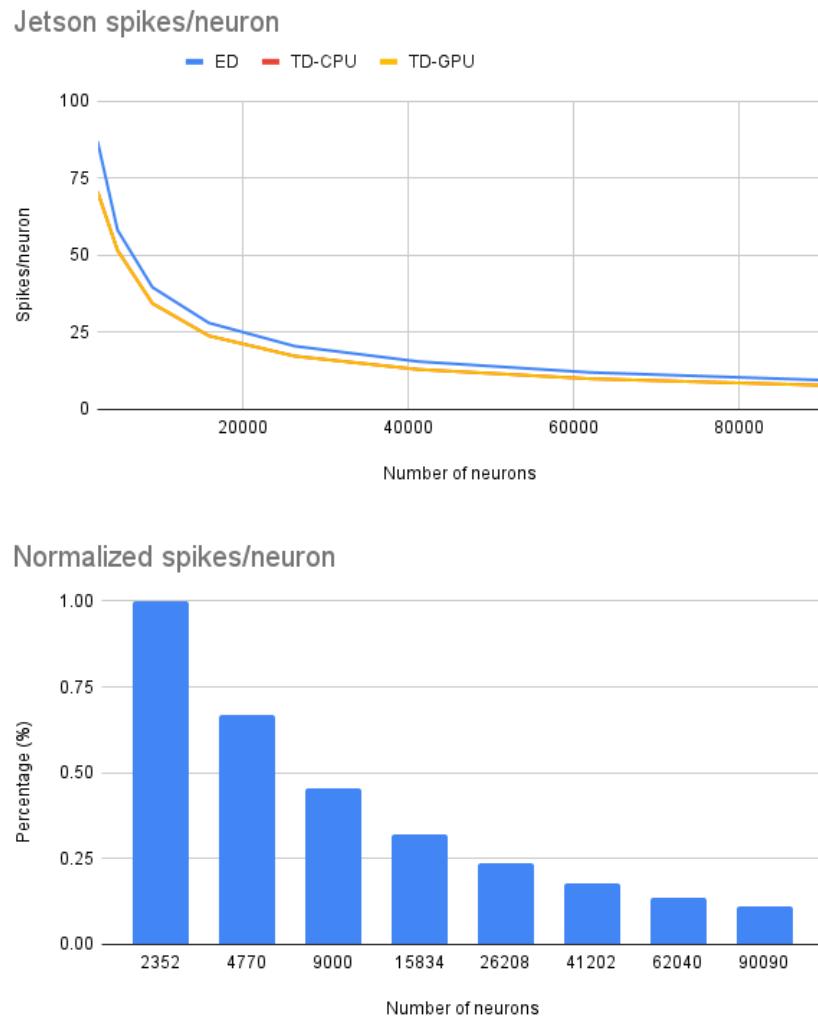


Figura 8.12: Arriba, medida de spikes producidos en una red respecto a las neuronas de las que está compuesta, abajo, dichos valores normalizados respecto a la red más pequeña.

	SpiNNaker	EDLUT-Jetson
Nº total neuronas	76.800	62.040
Nº total sinapsis	38.400.000	36.386.400
Nº eventos sinápticos / s	1.460.000.000	9.149.863,731
Topología de red	Todos con todos interchip	Cerebelo
Consumo / neurona (μW)	24	-
Energía / evento sináptico (nJ)	6	-

Cuadro 8.2: Comparativa entre la red de ejemplo de SpiNNaker y 10_100 en la Jetson.

con el resto de plataformas y asumiendo que SpiNNaker generaría la misma actividad neuronal que EDLUT al utilizar un modelo de neurona parecido. Para ello, a partir del consumo por neurona y la energía consumida por evento sináptico, podemos obtener el consumo teórico de la red cerebelar en SpiNNaker a partir de la ecuación

$$P_{sim} = P_{neu} \times N_{neu} + E_{esin} \times T_{esin}$$

donde

- P_{sim} es la potencia media total consumida durante la simulación de la red, en W .
- P_{neu} es la potencia media consumida por neurona, en W .
- N_{neu} es el número total de neuronas de la red.
- E_{esin} es la consumo energético medio por evento sináptico, en J .
- T_{esin} es la tasa media de eventos sinápticos procesados en la red por segundo, en Hz .

El único valor del que no disponemos aún de manera directa es T_{esin} . Se puede obtener sumando el output *total propagated spikes* de EDLUT, que mide el total de los eventos sinápticos generados por cada spike producido durante la simulación, y el número de spikes de spikes de entrada, que se ha definido en los ficheros de entrada de EDLUT. Por tanto, al ser todo valores conocidos, podemos obtener el valor para cada tamaño de red de manera sencilla.

Mediante esta fórmula se ha calculado el consumo teórico aproximado para redes cerebelares de distinto tamaño simuladas SpiNNaker. En la tabla 8.3 se presenta una tabla comparativa entre el consumo real de redes de distinto tamaño con EDLUT en la Jetson y el teórico calculado para las

Nº total neuronas	EDLUT-Jetson	Cerebelo-SpiNNaker	Dif
2352	5,2	0,1	5380,32 %
4770	5,7	0,16	3510,77 %
9000	6,0	0,26	2278,85 %
15834	6,4	0,43	1481,56 %
26208	6,5	0,68	949,28 %
41202	6,5	1,05	612,17 %
62040	6,6	1,54	424,97 %

Cuadro 8.3: Consumo de la SIMULACIÓN de distintas redes con EDLUT en la Jetson frente a consumo teórico en SpiNNaker. Se muestra el porcentaje de potencia que consume la Jetson respecto a SpiNNaker.

Nº total neuronas	EDLUT-Jetson	Cerebelo-SpiNNaker	Dif
2352	9,0	11,90	75,53 %
4770	9,5	11,97	79,28 %
9000	9,8	12,07	81,30 %
15834	10,2	12,24	83,13 %
26208	10,3	12,49	82,28 %
41202	10,3	12,85	80,12 %
62040	10,4	13,35	77,61 %

Cuadro 8.4: Consumo TOTAL de distintas redes con EDLUT en la Jetson frente a consumo teórico en SpiNNaker. Se muestra el porcentaje de potencia que consume la Jetson respecto a SpiNNaker.

mismas redes en SpiNNaker. El primer valor ha sido calculado a partir de restar el consumo total de la Jetson durante la simulación y el consumo de esta cuando se encuentra ociosa.

Efectivamente, SpiNNaker es mucho más eficiente a la hora de simular este tipo de redes. Sin embargo, si tenemos en cuenta la potencia total, y no solo la de simulación, estos valores cambian completamente. La potencia total consumida por la Jetson para cada tamaño de red lo hemos obtenido durante los experimentos, mientras que el valor para SpiNNaker se puede obtener sumando su consumo estando ocioso al consumo calculado anteriormente. El consumo de SpiNNaker cuando se encuentra ocioso, según los experimentos antes referenciados, es de unos 11,8 W. Por tanto, podemos obtener la tabla 8.4 que compara el consumo total de ambas situaciones.

De esta otra medida podemos deducir que, aunque SpiNNaker sea más eficiente en las simulaciones, cuando tenemos en cuenta la potencia consumida total de las dos plataformas vemos que la Jetson lo supera (teóricamente) para todos los tamaños de red que EDLUT puede simular en tiempo real. La ventaja principal de esta plataforma es que consigue un consumo muy bajo

por el simple hecho de estar encendida, 3,4 W, en contra de SpiNNaker, que casi la cuadriplica con 11,8 W. Recordemos, además, que la implementación de simuladores y redes para plataformas genéricas es mucho más sencillo y versátil que hacerlo para plataformas tan específicas. Por último, cabe destacar que en las pruebas con la Jetson se ha estado utilizando plasticidad, mientras que los experimentos referenciados no lo hacen. Todo esto conforma un indicio de que la NVIDIA Jetson es más eficiente para simular SNN mediante EDLUT que mediante SpiNNaker, al menos para los tamaños de red que estamos manejando en este proyecto.

Capítulo 9

Conclusiones

9.1. Objetivos logrados

Una vez finalizado el proyecto en su totalidad, es momento de volver a los objetivos planteados al principio del mismo para concluir si los hemos cumplido o no. A continuación los volveremos a listar y, uno por uno, comentaremos si se han cumplido o no y algunos aspectos relacionados con ellos.

- **Implementación de un sistema de control para el Baxter mediante una biblioteca de funciones ligera de interfaz robótica (cROS).**
✓ **Cumplido.** Se han implementado varios nodos ROS utilizando la biblioteca de funciones cROS que, en conjunción, pueden formar parte del ciclo de control tanto proporcionándole posiciones deseadas al controlador interno del robot sobre la marcha como relajado directamente el control de los motores y cerrando el ciclo en la plataforma Jetson. Su funcionamiento ha sido comprobado utilizando un modelo simulado del robot Baxter en Gazebo de manera satisfactoria. Por lo tanto, hemos obtenido un sistema de control funcional para nuestro robot.
- **Validación de la interoperabilidad entre la interfaz robótica ligera (cROS) y las librerías estándar de ROS.**
✓ **Cumplido.** Como se ha comentado en el punto anterior, el trabajo de la primera parte de este proyecto se ha materializado en un sistema de control para el Baxter, implementado mediante la biblioteca cROS. Sin embargo, tanto la simulación del Baxter como sus nodos interfaz y el nodo maestro están en el lado del robot, funcionando exclusivamente bajo las bibliotecas de la distribución estándar de ROS.

El correcto funcionamiento de las tres partes es una prueba feaciente de la interoperabilidad real entre cROS y las bibliotecas estándar de ROS.

- **Integración y evaluación del control en la plataforma de computación autónoma (*stand-alone*), como la familia NVIDIA Jetson.**

✓ **Cumplido.** Como se ha visto en el capítulo de resultados, el ciclo de control ha sido tratado y validado en la NVIDIA Jetson. Por tanto, se ha comprobado que el uso de dicha plataforma es viable para controlar el robot utilizando el sistema que hemos implementado durante el proyecto, en tiempo real y sin requerir ninguna otra dependencia de software aparte del propio cROS.

- **Integración del simulador neuronal EDLUT en la plataforma NVIDIA Jetson.**

✓ **Cumplido.** Durante el desarrollo del proyecto se ha instalado EDLUT en distintas plataformas: PC, ordenador portátil, NVIDIA Jetson y Raspberry Pi Zero. En todas las plataformas la instalación ha sido llevada a cabo apenas sin errores, exceptuando el problema de la cabecera con las CPU basadas en ARM. Por tanto, queda cumplido el objetivo de instalar y utilizar funcionalmente EDLUT en la NVIDIA Jetson, y en general en un sistema empotrado.

- **Validación de simulación una red neuronal con EDLUT en tiempo real en la NVIDIA Jetson.**

✓ **Cumplido.** A partir de las pruebas realizadas en la Jetson con los distintos esquemas de simulación podemos afirmar que sí, es posible simular una red neuronal por impulsos de un tamaño suficiente para un potencial control del robot con EDLUT en tiempo real.

- **Comparativa del rendimiento, en términos de potencia y consumo, de EDLUT en esta plataforma frente a otras plataformas de propósito general (computador de propósito general).**

✓ **Cumplido.** Se han realizado pruebas con las mismas redes y configuración en distintas plataformas de propósito general, tanto tradicionales como empotradas. Estos experimentos aportan luz sobre el rendimiento y el consumo de cada una de estas plataformas a la hora de utilizar EDLUT simulando un modelo de cerebro simplificado usando redes neuronales por impulsos.

- **Comparativa de los datos obtenidos frente a plataformas con arquitecturas específicas para sistemas neuronales basados en impulsos.**

✓ **Cumplido.** Aunque no de manera exacta, hemos realizado una comparativa de consumo entre SpiNNaker y EDLUT ejecutándose en la Jetson para distintos tamaños de red neuronal. Esta comparación nos permite aclarar, aunque no de manera definitiva, que la NVIDIA Jetson consigue resultados semejantes a esta plataforma específica para esta tarea en concreto.

- **Responder a la pregunta: ¿es posible controlar un robot colaborativo desde un sistema neuronal basado en impulsos (SNN) en un sistema empotrado de manera efectiva, eficiente y en tiempo real?**
- ✓ **Cumplido.** La respuesta a esta pregunta es, básicamente, la conclusión del proyecto, la cual es tratada en profundidad a continuación.

9.2. Conclusión

Todo el trabajo invertido en el proyecto, así como los resultados expuestos en el capítulo anterior, dejan entrever que la respuesta es sí: es posible controlar un robot colaborativo desde un sistema empotrado integrando un ciclo de control y una red neuronal de impulsos y conectado mediante una red ROS en tiempo real, obteniendo además buenos resultados. Es evidente que ambas modalidades de control que se han probado en la Jetson funcionan en tiempo real sin ningún problema. Por otro lado, las pruebas realizadas con EDLUT hacen viable la posibilidad de un sistema de control regulado por un modelo de cerebelo. Si nos atenemos a los datos, una red con 62000 neuronas se ejecuta en dos tercios del tiempo simulado, un margen que permitiría introducir los nodos cROS correspondientes manteniendo el tiempo real.

Otras conclusiones obtenidas de este proyecto, algo más concretas, son:

- cROS, a pesar de sus peculiaridades, es perfectamente compatible con las librerías estándar de ROS.
- El sistema de control implementado con cROS es funcional en la NVIDIA Jetson.
- EDLUT funciona correctamente en la Jetson, aunque el hecho de no estar optimizado para arquitecturas ARM hace que no sea ventajoso el paralelismo en CPU.
- El esquema de simulación TD-GPU es superior al resto en la Jetson, tanto en rendimiento como en eficiencia.

- El tamaño de las tablas para el esquema ED no afecta al tiempo de ejecución, pero las tablas de tamaño reducido provocan que la red neuronal genere una actividad superior a la esperada.
- La Jetson es entre dos y cuatro veces más eficiente para utilizar ED-LUT que el resto de plataformas de propósito general utilizadas durante el proyecto.
- Sobre el papel, de manera teórica, la simulación del modelo cerebelar debería de consumir menos en la Jetson (con EDLUT) que en SpiN-Naker. Esto, junto a la dificultad de adaptar un modelo como este a dicha arquitectura específica, colocaría a la Jetson como una de las mejores alternativas empotradas para simular redes pequeñas.

En definitiva, la conclusión principal del proyecto es que la NVIDIA Jetson, como plataforma empotrada de altas prestaciones, ha de tenerse en cuenta a la hora de plantear sistemas que requieran una potencia computacional limitada. Las características que hemos comprobado durante el proyecto muestran un buen equilibrio de prestaciones que, en muchas situaciones, puede hacer viable la integración de la unidad de control, el modelo de inteligencia artificial, o el sistema que en cada caso sea necesario.

9.3. Trabajo futuro

Finalmente comentaremos brevemente aspectos que sería interesante tratar en el futuro a partir del trabajo realizado durante este proyecto:

- Dedicar tiempo a afinar el ajuste del controlador PD para el robot, ya que con el rendimiento que muestra es claro que podríamos conseguir un buen resultado con él.
- Explorar ROS2 como alternativa a ROS o cROS. ROS2 está basado en el protocolo DDS para la comunicación de sus nodos, es más escalable, incorpora más medidas de seguridad y está más enfocado a tiempo real. Por tanto este puede ser uno de los caminos a seguir a la hora de mejorar y modernizar el sistema de control para robots colaborativos.
- Estudiar un perfil de potencia ad-hoc para la Jetson, el cual puede aumentar sustancialmente la eficiencia dependiendo de las necesidades de cada aplicación, y la posibilidad de cambiar entre perfiles de potencia en caliente, aumentando la eficiencia aún en mayor medida si la carga de trabajo no es constante.

- Ya que se ha confirmado la viabilidad de utilizar cROS y EDLUT en la Jetson, implementar el sistema de control completo regulado a través del cerebelo artificial y realizar experimentos de control con el robot real y este sistema de control cerebelar.
- Aunque es una tarea compleja, adaptar EDLUT a ARM puede suponer una mejora sustancial en los resultados obtenidos en este proyecto. Al fin y al cabo la Jetson dispone de ocho núcleos y estamos utilizando uno, hay margen para mejorar.

Bibliografía

- [1] Ignacio Abadia, Francisco Naveros, Jesus A. Garrido, Eduardo Ros, and Niceto R. Luque. On robot compliance: A cerebellar control approach. *IEEE Transactions on Cybernetics*, 51(5):2476–2489, May 2021.
- [2] Yaser S. Abu-Mostafa, Malik Magdon-Ismail, and Hsuan-Tien Lin. *Learning From Data*. AMLBook, 2012.
- [3] Connor Anderson. Artículo sobre el algoritmo de recomendación de YouTube. Towards Data Science. <https://towardsdatascience.com/using-deep-neural-networks-to-make-youtube-recommendations-dfc0a1a13d1e>. Último acceso: 04/07/2021.
- [4] Arduino. Página oficial de Arduino. <https://www.arduino.cc/>. Último acceso: 04/07/2021.
- [5] Isaac Asimov. Runaround. *Astounding science fiction*, 29(1):94–103, 1942.
- [6] Isaac Asimov. *I, robot*, volume 1. Spectra, 2004.
- [7] Jason Brownlee. Artículo sobre Redes Neuronales Generativas Adversarias. Machine Learning Mastery. <https://machinelearningmastery.com/what-are-generative-adversarial-networks-gans/>. Último acceso: 04/07/2021.
- [8] Srdjan Coric, Miriam Leeser, Eric Miller, and Marc Trepanier. Parallel-beam backprojection: an fpga implementation optimized for medical imaging. In *Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays*, pages 217–226, 2002.
- [9] NVIDIA Corporation. Documentación oficial de NVIDIA Jetson. <https://docs.nvidia.com/jetson/l4t/index.html>. Último acceso: 04/07/2021.
- [10] NVIDIA Corporation. Jetson AGX Xavier Thermal Design Guide. <https://developer.nvidia.com/embedded/downloads#?search=Je>

- tson%20AGX%20Xavier%20Series%20Thermal%20Design%20Guide.
Último acceso: 04/07/2021.
- [11] NVIDIA Corporation. La familia Jetson de NVIDIA. <https://www.nvidia.com/es-es/autonomous-machines/>. Último acceso: 04/07/2021.
 - [12] Neural Dynamics. Integrate-And-Fire Models. Neural Dynamics online book, chapter 1 section 3. <https://neuronaldynamics.epfl.ch/online/Ch1.S3.html>. Último acceso: 04/07/2021.
 - [13] Filmaffinity. Ex Machina (2015). <https://www.filmaffinity.com/es/film132582.html>. Último acceso: 04/07/2021.
 - [14] Filmaffinity. La Guerra de las Galaxias (1977). <https://www.filmaffinity.com/es/film712041.html>. Último acceso: 04/07/2021.
 - [15] Filmaffinity. Metrópolis (1927. <https://www.filmaffinity.com/es/film282386.html>. Último acceso: 04/07/2021.
 - [16] Filmaffinity. Yo, robot (2004). <https://www.filmaffinity.com/es/film339602.html>. Último acceso: 04/07/2021.
 - [17] Open Source Robotics Foundation. Gazebo. <http://gazebosim.org/>. Último acceso: 04/07/2021.
 - [18] RASPBERRY PI FOUNDATION. Página oficial de Raspberry Pi 4. <https://www.raspberrypi.org/products/raspberry-pi-4-model-b/>. Último acceso: 04/07/2021.
 - [19] Dustin Franklin. Artículo sobre Jetson AGX Xavier en NVIDIA Developer. NVIDIA Corporation. <https://developer.nvidia.com/blog/nvidia-jetson-agx-xavier-32-teraops-ai-robotics/>. Último acceso: 04/07/2021.
 - [20] W. Gerstner and W. M. Kistler. *Spiking neuron models: Single neurons, populations, plasticity*. Cambridge, U.K: Cambridge University Press, 2002.
 - [21] EDLUT. Sitio en GitHub del laboratorio de neuro-robótica de la UGR. <https://github.com/EduardoRosLab/edlut>. Último acceso: 04/07/2021.
 - [22] Repositorio de cROS en GitHub. <https://github.com/ros-industrial/cros>. Último acceso: 04/07/2021.
 - [23] Jorge Golowasch and Farzan Nadim. *Capacitance, Membrane*, pages 1–5. Springer New York, New York, NY, 2013.

- [24] Michael Goodrich and Alan Schultz. Human-robot interaction: A survey. *Foundations and Trends in Human-Computer Interaction*, 1:203–275, 01 2007.
- [25] Human Brain Project (HBP). Neurorobotics Platform. <https://neurorobotics.net/>. Último acceso: 04/07/2021.
- [26] Julien Henaut, Daniela Dragomirescu, and Robert Plana. Fpga based high date rate radio interfaces for aerospace wireless sensor systems. In *2009 Fourth International Conference on Systems*, pages 173–178. IEEE, 2009.
- [27] RoboDK Inc. RoboDk. <https://robodk.com/>. Último acceso: 04/07/2021.
- [28] Indeed. Sueldo medio de informático en España. <https://es.indeed.com/career/informatico/salaries>. Último acceso: 04/07/2021.
- [29] Andrew Senior. John Jumper. Demis Hassabis. Pushmeet Kohli. Artículo sobre AlphaFold y la estructura de proteínas. <https://deepmind.com/blog/article/alphafold-a-solution-to-a-50-year-old-grand-challenge-in-biology>. Último acceso: 04/07/2021.
- [30] Guotao Li, Guanglu Jia, Hailin Huang, and Bing Li. Eso-based adaptive dynamic surface control of elastic robotic joints with friction and disturbances. pages 1107–1112, 07 2017.
- [31] Cyberbotics Ltd. Webots. <https://cyberbotics.com/>. Último acceso: 04/07/2021.
- [32] Francisco Mora. Cómo funciona el cerebro. pages 182–184. Alianza editorial, 2014.
- [33] Francisco Náveros, Jesus A Garrido, Richard R Carrillo, Eduardo Ros, and Niceto R Luque. Event-and time-driven techniques using parallel cpu-gpu co-processing for spiking neural networks. *Frontiers in neuroinformatics*, 11:7, 2017.
- [34] Francisco Náveros, Niceto R. Luque, Jesús A. Garrido, Richard R. Carrillo, Mancia Anguita, and Eduardo Ros. A spiking neural simulator integrating event-driven and time-driven computation schemes using parallel cpu-gpu co-processing: A case study. *IEEE Transactions on Neural Networks and Learning Systems*, 26(7):1567–1574, 2015.
- [35] Chris Nicholson. Artículo sobre Word2Vec. Wiki oficial de PathMind. <https://wiki.pathmind.com/word2vec>. Último acceso: 04/07/2021.

- [36] International Federation of Robotics (IFR). World Robotics 2019. <https://ifr.org/downloads/press2018/IFR%20World%20Robotics%20Presentation%20-%2018%20Sept%202019.pdf>. Último acceso: 04/07/2021.
- [37] Rethink Robotics. Baxter API Reference. https://sdk.rethinkrobotics.com/wiki/API_Reference. Último acceso: 04/07/2021.
- [38] Rethink Robotics. Baxter hardware specifications. https://sdk.rethinkrobotics.com/wiki/Hardware_Specifications. Último acceso: 04/07/2021.
- [39] Rethink Robotics. Baxter Simulator Installation. https://sdk.rethinkrobotics.com/wiki/Simulator_Installation. Último acceso: 04/07/2021.
- [40] Definición del tipo de mensaje JointState. https://docs.ros.org/en/api/sensor_msgs/html/msg/JointState.html. Último acceso: 04/07/2021.
- [41] Documentación de cROS. Repositorio de cROS en GitHub. https://github.com/ros-industrial/cros/blob/master/docs/cROS_manual.pdf. Último acceso: 04/07/2021.
- [42] Lista de librerías que implementan ROS. <http://wiki.ros.org/Client%20Libraries>. Último acceso: 04/07/2021.
- [43] Sitio web oficial de ROS. <https://www.ros.org/>. Último acceso: 04/07/2021.
- [44] Sitio web oficial de ROS2. <https://docs.ros.org/en/foxy/index.html>. Último acceso: 04/07/2021.
- [45] Wiki oficial de ROS. <http://wiki.ros.org/>. Último acceso: 04/07/2021.
- [46] Salicru. Software Salicru para dispositivos SAI. <https://www.salicru.com/softwares-usb-rs-232.html>. Último acceso: 04/07/2021.
- [47] SimSpark. SimSpark. <https://robocup-sim.gitlab.io/SimSpark/index.html>. Último acceso: 04/07/2021.
- [48] Evangelos Stamatias, Francesco Galluppi, Cameron Patterson, and Steve Furber. Power analysis of large-scale, real-time neural networks on spinnaker. In *The 2013 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, 2013.
- [49] John Von Neumann. First draft of a report on the edvac. *IEEE Annals of the History of Computing*, 15(4):27–75, 1993.

- [50] WikiChip. Despieces y características HW del chip NVIDIA Tegra. <https://en.wikichip.org/wiki/nvidia/tegra/xavier>. Último acceso: 04/07/2021.
- [51] Página de Wikipedia de ROS. https://es.wikipedia.org/wiki/Robot_Operating_System. Último acceso: 04/07/2021.
- [52] Wikipedia. Robotics Simulator. https://en.wikipedia.org/wiki/Robotics_simulator. Último acceso: 04/07/2021.
- [53] William G. Wong. Review de la Jetson AGX Xavier. ElectronicDesign. <https://www.electronicdesign.com/technologies/embedded-revolution/article/21807321/nvidia-jetson-agx-xavier-part-1-hardware>. Último acceso: 04/07/2021.

Apéndice A

Guia de usuario de EDLUT-cROS

A.1. Introducción

EDLUT-cROS es un proyecto de sistema de control para el robot Baxter. Por el momento solo es posible ejecutar esquemas de control usando el control por posición interno del robot y en lazo cerrado con controlador PD integrado en la Jetson, es decir, el simulador neuronal EDLUT no se utiliza para el control robot. Sin embargo pretende ampliarse hasta utilizar redes neuronales por impulsos (SNN). Este sistema es compatible casi con cualquier sistema operativo que pueda compilar código C(sólo se ha testeado en Linux), siendo las únicas dependencias de terceros necesarias para su ejecución `gcc/g++` y `make`. Además, es ligero y con una sola hebra, por lo que es posible ejecutarlo en plataformas con una potencia computacional limitada.

A.2. Instalación

1. Clonar el repositorio de EDLUT-cROS en tu directorio de trabajo.
2. Clonar el repositorio de cROS(<https://github.com/ros-industria/cros>) en el directorio raíz de EDLUT-cROS.
3. Revisar si el error del fichero de cabecera `include/cros.h` de cROS sigue presente. Si la línea `#endif /* INCLUDE_CROS_H_ */` no está al final, llevarla al final y guardar.
4. Compilar cROS.
5. Ejecutar el *makefile* presente en el directorio raíz del proyecto.

6. Nota: es importante configurar las direcciones IP para que coincidan con las de los dispositivos que se van a utilizar en la red local concreta.

A.3. Ejecución

Hay dos maneras de ejecutar los ciclos de control: de manera automática o lanzado manualmente cada uno de los nodos deseados.

La opción automática es la más sencilla, pero sólo está disponible en Linux (o cualquier plataforma capaz de ejecutar correctamente scripts en *bash*). Basta con copiar los scripts presentes en el directorio `launch_scripts` en la carpeta `bin` y ejecutar el que se corresponda con la lógica de control que deseemos realizar. Es importante recordar que deberemos darle permisos de ejecución a dichos scripts y llevar a cabo las ejecuciones desde la carpeta `bin` para evitar errores en las rutas (*paths*).

La opción manual requiere de conocer los nodos que deseemos ejecutar, pero nos proporciona una mayor versatilidad. Para llevarla a cabo deberemos ejecutar por separado los distintos nodos mediante los ejecutables presentes en la carpeta `bin`, con sus respectivos parámetros. Para más información, consultar el interior de los scripts donde se definen los parámetros para cada uno de los nodos, y leer en profundidad la parte II de este TFG.

Apéndice B

Guía de usuario de Jetson Tools

B.1. Introducción

Este repositorio contiene los elementos necesarios para realizar los test de rendimiento y eficiencia con EDLUT en una plataforma específica. Dichos test lanzarán una simulación con redes neuronales por impulsos en el simulador neuronal EDLUT, para distintos escenarios con variaciones en el tamaño de las redes, la frecuencia de llegada de los impulsos, u otras métricas. Las redes utilizadas tienen una estructura cerebelar.

B.2. Instalación y ejecución

1. Clonar el repositorio de EDLUT (<https://github.com/EduardoRosLab/edlut>) en tu directorio de trabajo.
2. Instalar EDLUT.
3. Clonar este repositorio en la carpeta `bin` de EDLUT.
4. Generar las tablas para el esquema de simulación ED.
5. Copiar los modelos de neurona a la carpeta `data` en `bin`.
6. Lanzar los test deseados a través de los scripts facilitados o del ejecutable `edlut_kernel`.

Los modelos de neurona están compuestos por los archivos `.cfg` y, en el caso de los modelos ED, las tablas de los archivos `.dat`.

- ### B.3. Descripción de los archivos
- **LIF_ED:** Contiene los archivos de los modelos de neuronas ED.
 - **LIF_TD:** Contiene los archivos de los modelos de neuronas TD.
 - **input_files10:** Contiene los ficheros de entrada (*input files*) para un tiempo de simulación de 10 segundos.
 - **input_files100:** Contiene los *input files* para un tiempo de simulación de 100 segundos.
 - **input_files200:** Contiene los *input files* para un tiempo de simulación de 200 segundos.
 - **input_files25:** Contiene los *input files* para un tiempo de simulación de 25 segundos.
 - **input_files400:** Contiene los *input files* para un tiempo de simulación de 400 segundos.
 - **input_files50:** Contiene los *input files* para un tiempo de simulación de 50 segundos.
 - **input_files_diff:** Contiene los *input files* para diferentes frecuencias de llegada de spikes.
 - **network_files:** Contiene los archivos de definición de redes.
 - **weight_files:** Contiene los archivos de definición de pesos.
 - **EDLUT_power_profile.txt:** Contiene la definición del perfil de potencia personalizado utilizado en las pruebas.
 - **activity_generator.py:** Script que es capaz de generar *input files*. Obtendrá por parámetros el factor de tamaño utilizado en la red y el tiempo de simulación, en ese orden.
 - **different_freq.sh:** Script que ejecuta las pruebas para distintas frecuencias de llegada de spikes.
 - **different_inputs.sh:** Script que ejecuta las pruebas para los distintos tamaños de red.
 - **different_times.sh:** Script que ejecuta las pruebas para distintos tiempos de simulación.
 - **jetson_test_single.sh:** Script que ejecuta las pruebas para distintos tamaños de red con un número determinado de repeticiones y lo almacena en archivos de log.

- `monitor.py`: Script que lanza un monitor software de consumo en paralelo a la simulación de EDLUT, que proporcionará los datos cuando dicha simulación acabe.

