

Neural Analytics: una herramienta software basada en redes neuronales de procesado de EEGs

Bachelor's Degree in Computer Engineering

Final Degree Project

Author:

Sergio Martínez Aznar

Supervisor(s):

Antonio Molina Picó

Academic Year:

2024/2025



UNIVERSITAT
POLITÈCNICA
DE VALENCIA

CAMPUS
D'ALCOI

Para aquellos que una vez soñaron con volar alto, y lo arriesgaron todo.

Resumen

Este proyecto presenta el desarrollo de un sistema de control domótico basado en interfaces cerebro-computadora (BCI). El objetivo principal es implementar un sistema no invasivo que permita la integración de señales cerebrales con dispositivos de iluminación inteligente, específicamente bombillas TP-Link Tapo.

El sistema utiliza la diadema Brainbit como dispositivo de adquisición de señales electroencefalográficas (EEG), procesando estas señales mediante una arquitectura de software que combina Deep Learning para procesamiento y una solución desarrollada en Rust para el consumo del modelo e interacción con el sistema de iluminación.

El marco teórico aborda el desarrollo de un modelo de clasificación basado en señales EEG, explorando técnicas de procesamiento de señales y aprendizaje automático para la interpretación precisa de patrones cerebrales. Además, se profundiza en los requisitos técnicos y normativos necesarios para el desarrollo de dispositivos médicos, con especial énfasis en el estándar IEC 62304 para procesos del ciclo de vida del software de dispositivos médicos, y la implementación de sistemas operativos en tiempo real (RTOS) para garantizar la fiabilidad y seguridad del sistema.

La solución propuesta integra tecnologías modernas de procesamiento de señales cerebrales con sistemas de domótica, creando una interfaz natural e intuitiva para el control del entorno doméstico. Este proyecto representa un paso hacia la democratización de las interfaces cerebro-computadora en aplicaciones cotidianas.

Palabras clave: Interfaz cerebro-computadora, BCI, Domótica, Aprendizaje profundo, Rust, EEG, RTOS, Certificación médica

Abstract

This project presents the development of a home automation control system based on brain-computer interfaces (BCI). The main objective is to implement a non-invasive system that enables the integration of brain signals with smart lighting devices, specifically TP-Link Tapo bulbs.

The system uses the Brainbit headband as an electroencephalographic (EEG) signal acquisition device, processing these signals through a software architecture that combines PyTorch for training deep learning models and Rust for the development of the inference engine.

The theoretical framework addresses the development of a classification model based on EEG signals, exploring signal processing and machine learning techniques for accurate interpretation of brain patterns. Additionally, it delves into the technical and regulatory requirements necessary for medical device development, with special emphasis on the IEC 62304 standard for medical device software life cycle processes, and the implementation of real-time operating systems (RTOS) to ensure system reliability and safety.

The proposed solution integrates modern brain signal processing technologies with home automation systems, creating a natural and intuitive interface for controlling the home environment. This project represents a step towards the democratization of brain-computer interfaces in everyday applications.

Keywords: Brain-computer interface, BCI, Home automation, Deep learning, Rust, PyTorch, EEG, RTOS, Medical certification

Índice general

Resumen	I
Abstract	III
Índice general	V
Índice de figuras	IX
1. Introducción	1
1.1. Motivación	1
1.2. Objetivos	2
1.3. Metodología	3
1.3.1. Fase de Investigación	3
1.3.2. Fase de Desarrollo	3
1.3.3. Fase de Validación	3
I Marco Teórico	5
2. Normativa UNE-EN 62304	7
2.1. Objetivo y Alcance	7
2.2. Clasificación del Software	8
2.3. Cumplimiento y Aplicación durante el proyecto	8
3. Regiones implicadas del cerebro	9
3.1. Introducción a las Regiones Cerebrales Funcionales	9
3.2. El Lóbulo Temporal y la Memoria Visual	10
3.3. El Lóbulo Occipital y la Percepción del Color	10
3.4. Integración entre Memoria y Percepción	11
3.5. Aplicaciones en Interfaces Cerebro-Computadora	11
3.6. Aplicaciones durante este proyecto	11
4. Sistemas operativos en Tiempo Real	13
4.1. Taxonomía de Sistemas en Tiempo Real	14
4.1.1. Sistemas de Tiempo Real Estricto	14
4.1.2. Sistemas de Tiempo Real Flexible	15
4.1.3. Consideraciones de Implementación	15
4.2. Soluciones Comerciales para Hard Real-Time	16
4.2.1. VxWorks (Wind River Systems)	16

4.2.2. QNX Neutrino (BlackBerry)	17
4.2.3. Zephyr RTOS (Linux Foundation)	18
4.3. Soluciones Comerciales para Soft Real-Time	19
4.3.1. Wind River Linux (Wind River Systems)	19
4.3.2. Poky Linux (Proyecto Yocto)	20
4.4. Elección de RTOS para el Proyecto	21
4.4.1. Requisitos Temporales del Sistema	21
4.4.2. Consideraciones Técnicas	21
4.4.3. Aspectos Regulatorios y Económicos	21
5. Modelos de Deep Learning	23
5.1. Conceptos Fundamentales	23
5.1.1. Ventanas Temporales	23
5.1.2. One-Hot Encoding	23
5.2. Arquitectura del Modelo	24
5.2.1. Función de Activación ReLU	24
5.2.2. LSTM (Long Short-Term Memory)	25
5.2.3. Función Softmax	26
5.3. Evaluación del Modelo	26
5.3.1. Métricas de Evaluación	26
II Revisión Hardware & Software	27
6. BrainBit Headset	29
6.1. Introducción	29
6.2. Características Técnicas del Dispositivo	29
6.3. Metodologías para la Adquisición y Procesamiento	30
6.3.1. Captación de Señales	30
6.3.2. Acondicionamiento de Señales	30
6.3.3. Análisis mediante Aprendizaje Profundo	31
6.4. Campos de Aplicación	31
7. Raspberry Pi 4 Model B (8GB)	33
7.1. Introducción	33
7.2. Especificaciones Técnicas	33
7.2.1. Procesador y Memoria	33
7.2.2. Requerimientos de Energía	34
7.2.3. Interfaces y Conectividad	34
7.2.4. Consideraciones Térmicas	35
7.3. Elección de este dispositivo para el Proyecto	35
III Marco Práctico	37
8. Análisis Práctico	39
8.1. Objetivos Específicos	39
8.1.1. Realizados	39
8.1.2. Deseados (Futuras Mejoras)	40

8.2. Requisitos funcionales y no funcionales	41
8.2.1. Requisitos Funcionales	41
8.2.2. Requisitos No Funcionales	42
8.3. Bibliotecas Usadas	43
8.3.1. Procesamiento de Señales EEG	43
8.3.2. Interfaz Gráfica y Visualización	43
8.3.3. Inteligencia Artificial y Procesamiento de Datos	43
8.3.4. Comunicación y Control de Dispositivos	43
8.3.5. Herramientas de Concurrencia y Asincronía	43
8.3.6. Arquitectura y Diseño del Sistema	44
8.3.7. Serialización y Estructuras de Datos	44
9. Planificación Temporal	45
9.1. Cronología del Desarrollo	45
9.1.1. Fase de Investigación (Enero 2025)	45
9.1.2. Adquisición de Hardware y Estructuración (Finales de Enero 2025)	46
9.1.3. Fase de Desarrollo (Febrero - Marzo 2025)	46
9.1.4. Fase de Refinamiento del Modelo (Abril 2025)	51
9.2. Distribución Temporal	52
9.3. Diagrama de Gantt	53
9.4. Conclusiones sobre la Planificación	53
10. Entrenamiento del modelo	55
10.1. Descripción de la arquitectura	55
10.1.1. Estructura de la red neuronal	55
10.1.2. Parámetros del modelo	57
10.2. Preprocesamiento de los datos	57
10.2.1. Adquisición y estructuración del dataset	57
10.2.2. Etapas de preprocesamiento	57
10.2.3. Implementación del dataset	58
10.3. Resultados del entrenamiento	58
10.3.1. Configuración del entrenamiento	58
10.3.2. Métricas de rendimiento	59
10.3.3. Análisis de resultados	60
10.3.4. Exportación del modelo	60
11. Implementación del Core	61
11.1. Arquitectura Hexagonal	61
11.1.1. Estructura General	61
11.1.2. Puertos y Adaptadores	62
11.2. Consumo del SDK de BrainFlow	64
11.2.1. Inicialización y Configuración	64
11.2.2. Adquisición de Datos	64
11.3. Patrón Model-View-Intent (MVI)	65
11.3.1. Componentes del Patrón MVI	65
11.3.2. Flujo de Datos	66
11.3.3. Manejador de Eventos	66
11.4. Interconexión con el sistema domótico	67
11.4.1. Adaptador para Bombillas Inteligentes	67

11.4.2. Integración con la Máquina de Estados	68
11.4.3. Preparación para una futura integración con Matter	69
11.5. Implementación de la interfaz gráfica	69
11.5.1. Estructura de la GUI	69
11.5.2. Integración con el Core	70
11.6. Conclusiones y Justificación de Decisiones Arquitectónicas	71
11.6.1. Alineación con los Requisitos del Proyecto	71
11.6.2. Beneficios Observados Durante el Desarrollo	72
11.6.3. Impacto en la Calidad del Software	72
12. Validación del Prototipo	75
12.1. Marco Normativo de Validación	75
12.1.1. Normativa Aplicable	75
12.1.2. Clasificación del Software	75
12.2. Estrategia de Pruebas	76
12.2.1. Herramientas y Entorno de Pruebas	76
12.3. Pruebas Unitarias	76
12.3.1. Estrategia de Pruebas Unitarias	76
12.3.2. Pruebas Unitarias del Core (<code>neural_analytics_core</code>)	78
12.3.3. Cobertura de Código	80
12.3.4. Validación Manual de la Interfaz (<code>neural_analytics_gui</code>)	82
12.4. Pruebas de Integración	83
12.4.1. Mi enfoque para las pruebas de integración	83
12.4.2. Qué descubrí con las pruebas de integración	83
12.5. Pruebas del Sistema	85
12.5.1. Los casos de prueba que diseñé	85
12.6. Pruebas de Seguridad	86
12.6.1. Cómo gestiono los datos del usuario	86
12.6.2. Seguridad eléctrica	86
12.7. Gestión de Anomalías	86
12.8. Matriz de Trazabilidad	87
12.9. Conclusión de la Validación	88
Conclusiones	91
Agradecimientos	93
Anexo I: Manual de Operador para Captura de Datos de Entrenamiento	97
Anexo II: Manual de Usuario de la Aplicación Neural Analytics	101

Índice de figuras

4.1. Ecuación de sistemas de tiempo real estricto.	14
4.2. Ecuación de sistemas de tiempo real flexible.	15
5.1. Ejemplo de One-Hot Encoding para tres colores.	24
5.2. Ecuación de la función ReLU.	24
5.3. Ecuación de la función Softmax.	26
9.1. Diagrama de Gantt del proyecto Neural Analytics	53
10.1. Arquitectura del modelo de clasificación de señales EEG	56
10.2. Curvas de entrenamiento del modelo Neural Analytics	58
10.3. Matriz de confusión del modelo en el conjunto de validación	59
10.4. Curvas ROC para cada una de las clases	59
12.1. Verificación de impedancia y calidad de señal.	98
12.2. Adquisición de datos en tiempo real.	99
12.3. Pantalla de carga inicial de la aplicación.	101
12.4. Pantalla de bienvenida e instrucciones para colocarse la diadema.	102
12.5. Pantalla de calibración de electrodos.	102
12.6. Pantalla principal de la aplicación de inferencia.	103

Capítulo 1

Introducción

Este trabajo aborda —desde una perspectiva integradora— el diseño y desarrollo de un sistema innovador de automatización del hogar que combina tecnología de interfaz cerebro-computadora (BCI) con dispositivos de iluminación inteligentes. Mi propósito fundamental ha sido desarrollar una solución no invasiva que permita el control del entorno doméstico mediante la lectura e interpretación de ondas cerebrales, empleando específicamente dispositivos TP-Link Tapo como elementos de control.

La arquitectura que he implementado se sustenta en dos pilares fundamentales: por un lado, el procesamiento de señales electroencefalográficas (EEG) mediante técnicas de aprendizaje profundo y, por otro, el estricto cumplimiento de la normativa UNE-EN 62304 para dispositivos y/o software médicos. Para garantizar que el sistema responda en tiempo real —aspecto crítico en esta aplicación—, he optado por utilizar Poky Linux del Proyecto Yocto como sistema operativo base. Esta elección no fue casual, sino que respondió a su notable flexibilidad y facilidad de personalización, además de facilitar una posible migración futura a Wind River Linux en caso de querer comercializar este proyecto como producto médico.

1.1. Motivación

Desde que comencé mi formación en ingeniería, he sentido una fascinación particular por las interfaces cerebro-computadora y sus aplicaciones potenciales. Este proyecto representa, en cierto modo, una síntesis perfecta de mis inquietudes intelectuales: la tecnología, los sistemas operativos en tiempo real, la medicina y la innovación. Trabajar en un sistema que integra procesamiento de señales cerebrales con control domótico me ha permitido explorar un campo que, estoy convencido, revolucionará la interacción persona-máquina en los próximos años, aunque en este caso haya realizado este trabajo desde una perspectiva más lúdica que médica.

No es casualidad que decidiera trabajar con actuadores domóticos comunes, concretamente bombillas inteligentes. Esta decisión permite demostrar —de manera visual e intuitiva— el funcionamiento del sistema BCI, haciendo tangible una tecnología que muchas veces parece abstracta o inalcanzable. Este enfoque práctico, además, facilita enormemente la comprensión del sistema y su impacto potencial en contextos cotidianos.

El aspecto normativo del proyecto, aunque técnicamente desafiante (y a veces frustran-

te), ha representado para mí una oportunidad única de entender el proceso completo de desarrollo... desde una idea innovadora hasta un producto viable en el mercado médico. Cumplir con la normativa UNE-EN 62304, implementar un sistema en tiempo real y desarrollar una metodología robusta no ha sido una limitación, sino que ha enriquecido significativamente mi comprensión sobre lo que implica desarrollar tecnología médica responsable y segura.

1.2. Objetivos

Este proyecto tiene como finalidad principal el desarrollo de un sistema de control domótico basado en interfaces cerebro-computadora, con la intención de hacer más accesible esta tecnología en entornos cotidianos.

Para alcanzar esta meta, he establecido los siguientes objetivos específicos:

- **Cumplimiento del estándar UNE-EN 62304:** Crear una solución que se ajuste a la normativa UNE-EN 62304, garantizando así la seguridad y fiabilidad del software médico mediante una metodología de desarrollo rigurosa y bien documentada.
- **Implementar un clasificador de señales EEG:** Desarrollar e implementar un modelo de aprendizaje profundo para la clasificación eficiente de señales EEG, utilizando PyTorch como framework principal y centrándome especialmente en la detección de patrones asociados a la visualización de colores.
- **Desarrollar un sistema de control BCI:** Construir un sistema funcional que permita controlar dispositivos de iluminación inteligente mediante señales EEG, empleando la diadema Brainbit como dispositivo para la adquisición de dichas señales.
- **Generar una imagen para la RPi4:** Crear y optimizar una imagen de sistema operativo personalizada basada en Poky Linux, asegurando un entorno de ejecución con garantías de tiempo real blando para el procesamiento de señales EEG.

He definido estos objetivos considerando tanto los aspectos técnicos como normativos del proyecto, intentando mantener un equilibrio entre la innovación tecnológica —siempre atractiva— y la viabilidad práctica en entornos reales.

1.3. Metodología

El desarrollo de mi proyecto sigue una metodología estructurada en varias fases, aunque debo admitir que el proceso real no siempre ha sido tan lineal como sugiere esta estructura:

1.3.1. Fase de Investigación

- Estudio exhaustivo de la literatura disponible sobre procesamiento de señales EEG
- Análisis detallado de requisitos normativos para dispositivos médicos
- Evaluación de tecnologías y frameworks disponibles en el mercado actual

1.3.2. Fase de Desarrollo

- Implementación del modelo de clasificación en PyTorch, con múltiples iteraciones
- Desarrollo del motor de inferencia en Rust, seleccionado por su rendimiento y seguridad
- Integración con el SDK de BrainFlow, no sin ciertas dificultades técnicas
- Creación de la imagen personalizada de Poky Linux, optimizando componentes

1.3.3. Fase de Validación

- Pruebas rigurosas de rendimiento y fiabilidad en distintos escenarios
- Verificación del cumplimiento normativo, documentando cada aspecto relevante
- Evaluación de la usabilidad del sistema en condiciones reales de operación

La vertiente práctica de mi proyecto incluye una descripción detallada del proceso de desarrollo, abarcando desde el entrenamiento del modelo de aprendizaje profundo —con sus aciertos y errores— hasta la integración con el resto de componentes del sistema.

Parte I

Marco Teórico

Capítulo 2

Normativa UNE-EN 62304

La norma **UNE-EN 62304:2007** Asociación Española de Normalización (2016) es la versión española de la norma **IEC 62304:2006/A1:2015**, adoptada como norma europea EN 62304:2006/A1:2015. Esta normativa establece los requisitos para los **procesos del ciclo de vida del software en dispositivos médicos**, asegurando su desarrollo, mantenimiento y gestión de riesgos de acuerdo con estándares internacionales—algo fundamental para cualquier implementación seria en este campo.

2.1. Objetivo y Alcance

El propósito de la UNE-EN 62304 desarrollado por la Asociación Española de Normalización (2016) es definir un marco normativo para la **gestión del ciclo de vida del software** en dispositivos médicos, asegurando su seguridad y eficacia.

Esta norma se aplica a:

- **Software que es un dispositivo médico en sí mismo.**
- **Software embebido en dispositivos médicos.**
- **Software utilizado en entornos médicos para diagnóstico, monitoreo o tratamiento.**

El estándar establece **procesos y actividades** que los fabricantes deben seguir, incluyendo:

- Planificación del desarrollo del software.
- Análisis de requisitos y arquitectura del software.
- Implementación, integración, pruebas y verificación.
- Gestión del mantenimiento y resolución de problemas.
- Gestión del riesgo asociado al software.
- Gestión de la configuración y cambios.

2.2. Clasificación del Software

La norma clasifica el software en **tres niveles de seguridad** según el riesgo que pueda representar para el paciente o el operador. Esta clasificación, aunque a primera vista pudiera parecer simplista, encierra importantes matices que condicionan todo el proceso de desarrollo.

- **Clase A:** El software no puede causar daño en ninguna circunstancia.
- **Clase B:** El software puede contribuir a una situación peligrosa, pero el daño potencial es **no serio**.
- **Clase C:** El software puede contribuir a una situación peligrosa con **riesgo de daño serio o muerte**.

Tras analizar diversos casos de estudio y precedentes, me di cuenta de que determinar la clase correcta requiere un enfoque crítico y no meramente formal.

2.3. Cumplimiento y Aplicación durante el proyecto

Dado el carácter estricto, y la propia necesidad de garantizar el cumplimiento de la UNE-EN 62304 Asociación Española de Normalización (2016) en este trabajo, seguiré un enfoque basado en la **gestión del ciclo de vida del software** y la evaluación de riesgos. Adoptaré buenas prácticas de ingeniería de software y documentaré las actividades necesarias para cumplir con los requisitos de seguridad y calidad establecidos por la normativa. A veces esto supone un esfuerzo adicional que puede parecer excesivo. Pero es necesario.

En este proyecto, utilizaré un **modelo de desarrollo iterativo e incremental** inspirado en metodologías ágiles como Scrum, adaptado a las necesidades específicas del desarrollo de software médico. Tras probar inicialmente un enfoque más tradicional, me encontré con limitaciones significativas que me llevaron a reconsiderar mi aproximación. Este enfoque me permitirá una mayor flexibilidad y capacidad de adaptación a los cambios, así como una entrega continua de valor al cliente.

Considerando que el sistema únicamente controla el encendido y apagado de una bombilla inteligente TP-Link Tapo de manera remota, he clasificado el software como de **Clase A**. Esta clasificación se justifica porque el software no puede causar daño al usuario en ninguna circunstancia, ya que:

- La diadema BrainBit es un dispositivo no invasivo de lectura pasiva
- El control se realiza sobre una bombilla doméstica de baja tensión
- No hay interacción directa con sistemas críticos o vitales

A pesar de esta clasificación de bajo riesgo—podría argumentarse incluso que es demasiado cautelosa—mantendré buenas prácticas de desarrollo y documentación para asegurar la calidad del software. La experiencia me ha enseñado que subestimar los aspectos de calidad en fases tempranas suele traducirse en complicaciones posteriores difíciles de resolver.

Dado que esta normativa es un **requisito esencial** para el desarrollo de software en el mercado sanitario español y europeo, su correcta implementación garantizará la viabilidad del producto en entornos clínicos y su aceptación por parte de los organismos reguladores.

Capítulo 3

Regiones implicadas del cerebro

3.1. Introducción a las Regiones Cerebrales Funcionales

El cerebro humano es un sistema altamente organizado en el que diferentes regiones trabajan de manera especializada pero interconectada para procesar la información y generar respuestas conductuales adecuadas. Como indican en su libro "Principios de Neurociencia" los autores Kandel, Jessell y Schwartz (2001), el estudio de la neurociencia ha demostrado que la división funcional del cerebro permite analizar la forma en que los procesos perceptivos, cognitivos y emocionales emergen de la actividad neuronal distribuida. En este proyecto, me enfoco en dos regiones clave para la percepción y la memoria del color: el lóbulo occipital y el lóbulo temporal.

Utilizando un sistema BCI basado en EEG, empleo electrodos en ubicaciones específicas del sistema internacional 10-20: O1 y O2 en el lóbulo occipital, responsables del procesamiento primario de la información visual, y T3 y T4 en el lóbulo temporal, donde la información visual se asocia con memorias previas y respuestas emocionales. Fue particularmente revelador para mí descubrir, a través de descripciones técnicas de diversos proveedores de interfaces cerebro-ordenador, cómo estos sensores ubicados estratégicamente capturan la actividad en estas regiones específicas.

Gracias a esta organización funcional —que no es tan rígida como se pensaba hace algunas décadas— puedo estudiar la relación entre percepción e imaginación del color, explorando su impacto en la memoria y la experiencia subjetiva. La literatura previa que consulté al respecto confirmó esta aproximación.

3.2. El Lóbulo Temporal y la Memoria Visual

El lóbulo temporal es crucial en la memoria visual y la asociación semántica de colores. A veces olvidamos su importancia ante la predominancia del lóbulo occipital en el discurso sobre visión. Los electrodos T3 y T4 captan la actividad relacionada con:

- **Hipocampo:** Responsable de la consolidación de memorias visuales y asociaciones cromáticas.
- **Corteza temporal medial:** Procesa la identificación y categoría de los colores.
- **Amígdala:** Relaciona el color con respuestas emocionales, modulando el impacto afectivo de los colores percibidos o imaginados.
- **Corteza entorrinal:** Conecta el hipocampo con otras áreas corticales, permitiendo que las asociaciones cromáticas se integren en experiencias más complejas.

Cuando una persona recuerda un color, T3 y T4 reflejan la activación de estos circuitos, permitiendo analizar la relación entre percepción visual y memoria. No siempre resulta obvio cómo interpretar estas señales. Estudios como los de Squire y Zola-Morgan (1991) han demostrado que lesiones en el lóbulo temporal pueden afectar la capacidad de recuperar memorias visuales, lo que refuerza su papel en el almacenamiento de información sensorial.

3.3. El Lóbulo Occipital y la Percepción del Color

El lóbulo occipital es la principal región del cerebro para el procesamiento de la información visual. Quizá la más conocida, pero no por ello menos compleja. Los electrodos O1 y O2 capturan la actividad en:

- **Corteza visual primaria (V1):** Primer procesamiento del color y detección de longitudes de onda.¹
- **V4 (corteza visual asociativa):** Especializada en la interpretación y categorización de colores, estableciendo una conexión funcional con las áreas temporales para asignar significados semánticos.

Estudios como los de Brouwer y Heeger (2013) han demostrado que la actividad en V4 puede ser utilizada para clasificar diferentes colores percibidos o imaginados, lo que refuerza la validez de mis electrodos O1 y O2 para el análisis de patrones cromáticos en EEG. Conforme avanzaba en mi investigación, me encontré con que este respaldo científico era crucial para fundamentar las decisiones de diseño que tomé.

¹Es importante distinguir entre O1/O2, que son las *posiciones de los electrodos* según el sistema internacional 10-20, y V1/V4, que son *áreas funcionales del córtex visual* (designaciones de Brodmann) cuya actividad es registrada por dichos electrodos.

3.4. Integración entre Memoria y Percepción

El procesamiento del color en el cerebro, simplificándolo al absurdo —con el fin de que pueda entenderse fácilmente— sigue un flujo distribuido:

1. O1/O2 detectan y analizan las características del color percibido o imaginado.
2. La información es enviada a T3/T4 para su comparación con recuerdos previos y asociaciones emocionales.
3. Se generan asociaciones semánticas y afectivas, determinando la experiencia subjetiva del color.

No siempre ocurre de manera tan lineal, desde luego. Estudios como los de Rissman y Wagner (2012) han mostrado que patrones de activación en el lóbulo temporal pueden predecir si un individuo reconoce un color previamente visto, lo que refuerza la idea de que los recuerdos cromáticos tienen una representación neural clara. Después de revisar estos trabajos, me resultó evidente que había encontrado un fundamento sólido para mi enfoque.

3.5. Aplicaciones en Interfaces Cerebro-Computadora

La integración de la percepción y la memoria del color en un sistema BCI tiene varias aplicaciones potenciales. Algunas que me parecen particularmente prometedoras:

- Diferenciar entre percepción real e imaginada de un color.
- Implementar sistemas BCI basados en selección cromática.
- Explorar la relación entre color, memoria y emociones para aplicaciones en neurotecnología.

De este modo, puedo explorar mediante una interfaz cerebro-computadora cómo la percepción y la memoria del color se integran en la experiencia subjetiva, permitiendo así que pueda ser interpretado por un computador y que este pueda realizar acciones en función de la información recibida. Asentando así las bases teóricas en relación al BCI de este proyecto.

3.6. Aplicaciones durante este proyecto

Los electrodos O1, O2, T3 y T4 capturan información clave sobre la percepción y la memoria del color. Su combinación me permite analizar la reconstrucción mental de colores y su impacto en la experiencia subjetiva, formando la base de mi sistema BCI. Seleccione estas ubicaciones específicas tras revisar detalladamente las especificaciones técnicas proporcionadas por varios fabricantes de dispositivos BCI, así como literatura académica relevante que he ido citando a lo largo de este capítulo.

Capítulo 4

Sistemas operativos en Tiempo Real

Los sistemas operativos en tiempo real (RTOS) Siewert and Pratt (2016) son una rama del software que busca garantizar ejecución de tareas en plazos temporales específicos, pues el concepto mismo de computación en tiempo real nace de la necesidad de procesar y responder a eventos externos con restricciones de tiempo bien definidas. Entonces, la corrección del sistema no solo depende de la lógica de los resultados, sino también del momento exacto en que estos se producen.

Las características que diferencian a un RTOS de sistemas operativos convencionales son:

- **Determinismo:** Propiedad fundamental donde dada una entrada y estado inicial, nos va a devolver siempre la misma salida en tiempos predecibles.
- **Concurrencia:** Capacidad para gestionar varias tareas en espacios temporales limitados sin comprometer plazos críticos.
- **Interrupciones:** Mecanismos para responder a eventos externos de forma rápida y predecible.
- **Planificación:** Algoritmos que controlan el orden y tiempo de ejecución de tareas según su criticidad temporal.

En el campo de los sistemas embebidos, un RTOS actúa como intermediario entre el hardware y las operaciones de control. El ejemplo clásico de los controladores de vuelo en aeronaves es quizá el más ilustrativo, donde cualquier fallo en los tiempos de respuesta puede tener efectos catastróficos. Estos sistemas aparecen también en satélites artificiales, que suelen tener varios RTOS ejecutando en paralelo para controlar tanto las funciones del vehículo como los instrumentos científicos, todo trabajando de forma armónica.

Al investigar aplicaciones de RTOS, he encontrado muchos sectores donde son vitales: en entornos industriales para control de procesos críticos; en aviática para sistemas de navegación (con certificaciones super estrictas); en defensa para sistemas de radar; y —importante para este proyecto— en el sector médico, donde dispositivos como marcapasos, respiradores y bombas de infusión necesitan respuestas predecibles para garantizar seguridad del paciente.

4.1. Taxonomía de Sistemas en Tiempo Real

La clasificación básica de RTOS se basa en la criticidad de sus restricciones temporales. Esta taxonomía viene de Liu y Layland en 1973 Siewert and Pratt (2016) y distingue principalmente entre sistemas estrictos (*hard real-time*) y flexibles (*soft real-time*). Mi entendimiento de estas categorías cambió bastante al ver aplicaciones reales, pues la línea entre ambas no es tan clara como dicen los libros.

4.1.1. Sistemas de Tiempo Real Estricto

Los sistemas estrictos (**hard real-time**) no toleran ninguna desviación en sus plazos temporales. Si se incumple un plazo, se considera un fallo crítico del sistema —algo que da bastante miedo cuando piensas en sus aplicaciones en entornos críticos. Su comportamiento se puede expresar matemáticamente como:

$$\forall t \in T : R(t) \leq D(t) \quad (4.1)$$

Figura 4.1: Ecuación de sistemas de tiempo real estricto.

donde $R(t)$ es el tiempo de respuesta y $D(t)$ el plazo máximo permitido.

Algunos de los casos de uso más comunes para estos sistemas son:

- **Control nuclear:** Donde la precisión temporal es crítica para la seguridad
- **Sistemas ABS:** Responden en microsegundos para evitar accidentes
- **Robótica quirúrgica:** Necesitan sincronización precisa durante operaciones

La implementación requiere algoritmos **preemptivos** con prioridades estáticas, donde el tiempo máximo de ejecución debe ser predecible. Normalmente usan Rate Monotonic (RM) o Earliest Deadline First (EDF). Cuando intenté implementar esto en mis pruebas iniciales, me di cuenta que garantizar determinismo absoluto en hardware normal es casi imposible y me tocó replantear algunas decisiones.

4.1.2. Sistemas de Tiempo Real Flexible

Los sistemas flexibles (**soft real-time**) toleran cierta variabilidad en sus plazos, funcionando con un modelo estadístico. Es algo así como:

$$P(R(t) \leq D(t)) \geq p_{min} \quad (4.2)$$

Figura 4.2: Ecuación de sistemas de tiempo real flexible.

donde p_{min} es el nivel mínimo aceptable de cumplimiento.

Las aplicaciones más comunes son:

- **Streaming multimedia:** Perder algunos frames de vez en cuando no arruina la experiencia
- **Redes de monitorización:** Toleran retrasos ocasionales en actualizar datos
- **Trading algorítmico:** Les importa más el rendimiento promedio que garantías absolutas

Estos sistemas usan planificadores basados en **tiempo compartido** con prioridades dinámicas. Para mi interfaz cerebro-computadora, este enfoque fue mejor, ya que pequeñas variaciones en los tiempos no afectan mucho la experiencia. Después de analizar bien los requisitos temporales, vi claramente que este modelo no solo cumplía lo necesario, sino que ofrecía mejor balance entre complejidad y prestaciones.

4.1.3. Consideraciones de Implementación

La decisión entre sistemas estrictos o flexibles depende de varios factores:

- **Riesgos:** ¿Qué pasa si se incumple un plazo temporal?
- **Hardware disponible:** Limitaciones de procesamiento y memoria
- **Presupuesto:** Balance entre garantías temporales y complejidad
- **Regulaciones:** Requisitos de certificación según donde se use

4.2. Soluciones Comerciales para Hard Real-Time

4.2.1. VxWorks (Wind River Systems)

VxWorks es el referente en sistemas embebidos críticos, especialmente en aviónica, aeroespacial y médico. Cuando empecé a estudiarlo, su documentación me pareció abrumadora. Sus características principales:

Certificaciones y Normativas

- DO-178C Level A para sistemas aeroespaciales
- IEC 62304 para dispositivos médicos
- ISO 26262 ASIL D para automoción

Características Técnicas

- **Kernel:** Microkernel determinista con latencias ≤ 50 ns
- **Memoria:** MMU con protección y aislamiento
- **Scheduler:** 256 niveles de prioridad y herencia
- **IPC:** Comunicación con latencia determinista
- **Multiproceso:** Soporte para SMP y AMP con aislamiento

Al evaluar VxWorks me impresionó ver dónde se usa —desde rovers de Marte hasta dispositivos médicos certificados. Pero los costes de licencia para obtener un SDK personalizado eran prohibitivos para un proyecto que está en fase de prototipo.

4.2.2. QNX Neutrino (BlackBerry)

QNX Neutrino, que compró BlackBerry en 2010, destaca por su microkernel distribuido y fiabilidad. Es curioso que BlackBerry, después de casi desaparecer del mercado de móviles, mantenga este producto tecnológico tan avanzado:

Arquitectura

- **Microkernel:** Núcleo muy pequeño, menos de 100KB
- **Servicios:** Arquitectura modular en espacio de usuario
- **IPC:** Mensajería con mecanismo copy-on-write
- **Recuperación:** Reinicio de componentes sin afectar al sistema

Características Avanzadas

- **Tiempo Real:** Latencias garantizadas $\leq 100 \mu\text{s}$
- **Seguridad:** Modelo de seguridad con ASLR
- **Certificaciones:** IEC 61508 SIL3, IEC 62304 Clase C

QNX me gustó mucho por su uso en dispositivos médicos. Su problema principal no fue tanto el coste —BlackBerry tiene licencias para desarrollo— sino que no tiene buen soporte para Rust, lo que complicaba integrar las librerías que consumía este proyecto. Después de tratar de hacerlo funcionar, decidí que no valía la pena invertir más tiempo y dejé de lado esta opción.

4.2.3. Zephyr RTOS (Linux Foundation)

Zephyr es la alternativa open-source para sistemas embebidos críticos. Este proyecto, inicialmente iniciado por Wind River y más adelante donado a la Linux Foundation, ha crecido rápidamente y tiene un ecosistema de desarrolladores muy activo. Me pareció una opción interesante para el prototipo:

Diseño y Arquitectura

- **Kernel:** Configurable como monolítico o microkernel
- **Tamaño:** Desde 8KB hasta 512KB según configuración
- **Scheduler:** Hasta 32 niveles de prioridad
- **Certificación:** En proceso para IEC 61508 SIL 3/4

Características Destacadas

- **Drivers:** Más de 350 controladores para periféricos
- **Redes:** Soporte para protocolos IoT (BLE, Thread, LoRaWAN)
- **Seguridad:** Subsistema con aislamiento
- **Desarrollo:** Herramientas de depuración avanzadas

Aunque Zephyr era atractivo y sin coste económico, tenía también las complicaciones del soporte de librerías en Rust. Sin embargo, dadas sus características, preferí mirar otras opciones que pudieran contar con el núcleo de Linux y así aprovechar el ecosistema de librerías que ya tenía.

4.3. Soluciones Comerciales para Soft Real-Time

4.3.1. Wind River Linux (Wind River Systems)

Wind River Linux es una solución comercial basada en Yocto para sistemas con requisitos temporales flexibles. Cuando la descubrí, me pareció una versión más accesible de VxWorks, con enfoque más moderno:

Características Principales

- **Base:** Kernel Linux 5.10 LTS con parche PREEMPT_RT
- **Certificaciones:** ISO 9001:2015 y precertificación IEC 62304
- **Seguridad:** Monitorización de vulnerabilidades y mitigación
- **Conformidad:** Documentación SBOM y Open Chain 2.1

Capacidades Industriales

- **Soporte:** Mantenimiento garantizado 5 años, extensible
- **Actualizaciones:** Sistema OTA mediante OSTree
- **Validación:** Más de 60.000 tests automatizados
- **Servicios:** Soporte técnico y consultoría

Al principio consideré seriamente Wind River Linux por su precertificación IEC 62304, importante para dispositivos médicos. Pero los costes de licencia y soporte eran demasiado altos para esta fase del proyecto, así que busqué opté por una alternativa más económica para diseñar el prototipo.

4.3.2. Poky Linux (Proyecto Yocto)

Poky es la distribución de referencia del Proyecto Yocto para sistemas Linux embebidos con capacidades de tiempo real flexible. Cuando empecé a usarlo, me pareció muy flexible, aunque aprender a usarlo fue más complicado de lo que esperaba:

Características Técnicas

- **Kernel:** Linux con parche PREEMPT_RT
- **Tiempo Real:** Latencias configurables según necesidades
- **Optimización:** Control fino sobre tamaño y rendimiento
- **Personalización:** Capacidad para quitar componentes innecesarios

Consideraciones de Desarrollo

- **Mantenimiento:** Actualización manual de parches de seguridad
- **Soporte:** Basado en comunidad, sin garantías comerciales
- **Certificación:** Requiere proceso propio
- **Validación:** Hay que desarrollar pruebas específicas

Durante mi análisis, Poky resultó ser la opción con mejor equilibrio entre capacidades, flexibilidad y costes. Me permitió crear una imagen personalizada ajustada exactamente a los requisitos. Aunque no tiene certificaciones como Wind River Linux, su base en Yocto me da la opción de migrar a una solución más robusta si el proyecto avanza hacia la comercialización.

4.4. Elección de RTOS para el Proyecto

Elegí Poky Linux como sistema operativo para este proyecto por varios motivos:

4.4.1. Requisitos Temporales del Sistema

Mi proyecto necesita un sistema flexible (**soft real-time**) porque:

- La detección de patrones EEG para identificación de colores no necesita latencias críticas
- Un retraso en la respuesta no pone en peligro al usuario
- El control de iluminación con TP-Link Tapo tolera cierta variabilidad

Analizando el peor caso (un retraso al cambiar la iluminación), vi que las consecuencias no justificaban la complejidad de un sistema estricto.

4.4.2. Consideraciones Técnicas

Poky Linux tiene ventajas importantes para mi aplicación:

- **Flexibilidad:** Permite crear una imagen personalizada según requisitos.
- **Compatibilidad:** Se integra de forma sencilla con las librerías que consumo durante el desarrollo del proyecto.
- **Tiempo Real:** El parche PREEMPT_RT da las garantías temporales necesarias.

4.4.3. Aspectos Regulatorios y Económicos

Aunque Poky Linux no tiene precertificaciones como Wind River Linux, su base en Yocto facilita migrar a Wind River Linux si necesito certificaciones para comercialización. Esta decisión no fue fácil —pasé semanas analizando pros y contras, incluso hablando con gente del sector médico. Al final, esta estrategia me permite optimizar costes iniciales y mantener flexibilidad en esta fase, con la tranquilidad de tener un camino hacia la certificación si el proyecto se convierte en producto comercial.

Esta combinación de factores hace que Poky Linux sea la mejor opción para esta fase del proyecto, con buen equilibrio entre rendimiento, flexibilidad y costes. Como pasa muchas veces en ingeniería, la solución óptima no es la más avanzada técnicamente, sino la que mejor se adapta al problema concreto que estamos resolviendo.

Capítulo 5

Modelos de Deep Learning

A través de este capítulo describo los modelos de Deep Learning Raschka et al. (2022) que implementé en el proyecto, así como los conceptos fundamentales y la arquitectura de cada uno. También detallo las métricas de evaluación y el proceso de validación cruzada que utilicé para evaluar su rendimiento.

La selección de estos modelos no fue trivial. Después de analizar varias arquitecturas, me decanté por aquellas que mostraban mejor capacidad para detectar patrones temporales en señales EEG, especialmente en ventanas cortas—un requisito indispensable para la clasificación en tiempo real que buscaba conseguir.

5.1. Conceptos Fundamentales

5.1.1. Ventanas Temporales

Las ventanas temporales en el procesamiento de señales EEG representan segmentos discretos de tiempo durante los cuales se recopilan datos. En este proyecto, estas ventanas capturan patrones de actividad cerebral asociados al pensamiento de diferentes colores. La longitud de la ventana resultó ser un parámetro crítico: si era demasiado corta, no capturaba suficiente información para la clasificación; si era demasiado larga, introducía latencias inaceptables para una aplicación en tiempo real.

Tras numerosas pruebas con diferentes configuraciones, encontré que ventanas de 62 muestras ofrecían el mejor equilibrio. Esto no coincidía con lo que esperaba inicialmente basándome en la literatura, donde se sugerían ventanas más largas, pero comprobé empíricamente que este rango funcionaba mejor para mi implementación específica.

5.1.2. One-Hot Encoding

El One-Hot Encoding Raschka et al. (2022) es una técnica de preprocesamiento que transforma etiquetas categóricas (en este caso, colores) en vectores binarios. Por ejemplo, para tres colores:

Esta técnica es crucial cuando trabajamos con datos categóricos sin relación ordinal entre sí. A diferencia de la codificación de etiquetas ordinales, donde asignamos un valor numérico a cada categoría según un orden predefinido, One-Hot Encoding crea una columna

Color	Vector One-Hot
Rojo	[1, 0, 0]
Verde	[0, 1, 0]
Azul	[0, 0, 1]

Figura 5.1: Ejemplo de One-Hot Encoding para tres colores.

nueva para cada categoría posible.

Por ejemplo, si tenemos una columna de “color” con las opciones “rojo”, “verde” y “azul”, esta técnica la transforma en tres columnas nuevas. Cada fila tendrá un 1 en la columna de su color correspondiente y 0 en las demás.

Al principio dudé entre utilizar esta técnica o una codificación ordinal más simple, pues en ciertos experimentos preliminares había observado comportamientos similares con ambas. Sin embargo, conceptualmente el One-Hot Encoding representa mejor la naturaleza de los datos—no existe una relación ordinal inherente entre los colores—así que opté por implementar este enfoque más robusto.

5.2. Arquitectura del Modelo

5.2.1. Función de Activación ReLU

La función ReLU (Rectified Linear Unit) se convirtió en un componente esencial de mi modelo por características que la hacen especialmente adecuada:

$$f(x) = \max(0, x) \quad (5.1)$$

Figura 5.2: Ecuación de la función ReLU.

ReLU es una función de activación no lineal que resuelve el problema del desvanecimiento del gradiente presente en funciones como tanh o sigmoide. Este problema ocurre cuando, por ejemplo, para valores de entrada grandes ($z_1 = 20$ y $z_2 = 25$), las funciones tanh y sigmoide producen salidas prácticamente idénticas ($\sigma(z_1) \approx \sigma(z_2) \approx 1,0$) debido a su comportamiento asintótico.

Las principales ventajas que me llevaron a seleccionar ReLU son:

- **Gradiente Constante:** Para valores positivos de entrada, la derivada es siempre 1, evitando el desvanecimiento del gradiente.
- **Eficiencia Computacional:** Su implementación es simple y rápida, requiriendo solo una comparación con cero.
- **No Linealidad:** A pesar de su simplicidad, mantiene la capacidad para aprender funciones complejas.
- **Activación Dispersa:** Produce activaciones dispersas, ya que cualquier entrada negativa se convierte en cero.

Durante la fase de experimentación, probé diferentes funciones de activación, incluyendo Leaky ReLU y ELU, pero no observé mejoras significativas en el rendimiento que justificaran la complejidad adicional. La implementación de ReLU estándar resultó ser la opción más práctica y eficiente para mi caso de uso.

5.2.2. LSTM (Long Short-Term Memory)

Las LSTM fueron diseñadas para superar el problema del desvanecimiento del gradiente, común en redes neuronales recurrentes (RNN) estándar. Este problema ocurre por la multiplicación repetida de gradientes durante la retropropagación a través del tiempo (BPTT), haciendo que los gradientes se vuelvan extremadamente pequeños (desvanecimiento) o grandes (explosión).

Mi primera aproximación al proyecto utilizaba RNNs convencionales, pero pronto me encontré con limitaciones al procesar secuencias temporales largas. Las LSTM ofrecían una solución elegante a este problema, aunque con mayor complejidad computacional—un factor que inicialmente me preocupaba dado el requisito de ejecución en tiempo real.

Para entender mejor este problema, consideremos una RNN con una sola unidad oculta. La derivada de la función de pérdida respecto a la entrada neta tiene un factor multiplicativo que puede volverse muy pequeño o muy grande según el peso recurrente. Si este peso es menor que 1, el gradiente se desvanece; si es mayor que 1, explota.

Las LSTM abordan esto mediante celdas de memoria que mantienen información durante períodos largos. Cada celda incluye tres tipos de puertas: olvido, entrada y salida.

- **Puerta de Olvido (Forget Gate):** Decide qué información descartar de la memoria. Se calcula:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \quad (5.2)$$

- **Puerta de Entrada (Input Gate):** Decide qué nueva información almacenar. Se calcula:

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \quad (5.3)$$

- **Valor Candidato (Candidate Value):** Representa la nueva información potencial. Se calcula:

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \quad (5.4)$$

- **Puerta de Salida (Output Gate):** Decide qué parte de la memoria se usará para la salida. Se calcula:

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \quad (5.5)$$

La celda de memoria se actualiza así:

$$C_t = f_t \cdot C_{t-1} + i_t \cdot \tilde{C}_t \quad (5.6)$$

Y la salida se calcula como:

$$h_t = o_t \cdot \tanh(C_t) \quad (5.7)$$

Estas ecuaciones pueden resultar intimidantes al principio, pero la intuición detrás de ellas es bastante clara: las LSTM aprenden a controlar qué información recordar, actualizar u

olvidar en cada paso temporal. Al implementar esta arquitectura, pude capturar dependencias temporales en las señales EEG que resultaron cruciales para distinguir patrones asociados a diferentes colores.

5.2.3. Función Softmax

La función Softmax es una versión suavizada de argmax; en lugar de dar un único índice de clase, proporciona la probabilidad de cada una. Esto permite calcular probabilidades significativas en configuraciones multiclas.

En Softmax, la probabilidad de que una muestra con entrada neta z pertenezca a la clase i se calcula con un término de normalización en el denominador, que suma las funciones lineales ponderadas exponencialmente:

$$p(z) = \sigma(z) = \frac{e^{z_i}}{\sum_{j=1}^M e^{z_j}} \quad (5.8)$$

Figura 5.3: Ecuación de la función Softmax.

Las probabilidades resultantes suman 1, como cabría esperar. También es notable que la etiqueta predicha es la misma que al aplicar argmax a la salida logística.

Durante el desarrollo del modelo, consideré brevemente utilizar una capa sigmoide final con entrenamiento independiente para cada clase (enfoque one-vs-all), pero la implementación con Softmax resultó más elegante y directa, además de proporcionar interpretaciones probabilísticas más intuitivas de las predicciones.

5.3. Evaluación del Modelo

5.3.1. Métricas de Evaluación

Para evaluar el rendimiento del modelo utilicé un conjunto de métricas complementarias:

- **Accuracy:** Proporción de predicciones correctas sobre el total. Aunque es una métrica intuitiva, no siempre refleja el rendimiento real cuando las clases están desbalanceadas.
- **Matriz de Confusión:** Visualización detallada de aciertos y errores por clase. Esta herramienta me resultó particularmente útil para identificar patrones específicos de confusión entre colores, lo que me permitió ajustar el preprocesamiento de señales para mejorar la discriminación en casos problemáticos.
- **ROC-AUC:** Área bajo la curva ROC para evaluación multiclas. Esta métrica resultó especialmente valiosa por su robustez ante el desbalanceo de clases, un problema que surgió en algunas sesiones de recopilación de datos donde ciertos colores mostraban frecuencias de aparición variables.

La combinación de estas métricas me proporcionó una visión completa del rendimiento del modelo en diferentes escenarios y condiciones, guiando el proceso iterativo de mejora hasta alcanzar resultados satisfactorios para la aplicación en tiempo real.

Parte II

Revisión Hardware & Software

Capítulo 6

BrainBit Headset

6.1. Introducción

Elegir el dispositivo EEG adecuado para este proyecto fue más complicado de lo que esperaba inicialmente. Después de revisar varias opciones disponibles, terminé decidiendo por el **BrainBit** Neurotechnology Systems LLC (2024) por razones que van más allá de las especificaciones técnicas puras.

Mi proceso de selección se centró en encontrar algo que fuera preciso pero también práctico. No me servía un dispositivo que solo funcionara bien en laboratorio —y créeme que hay muchos así en el mercado— necesitaba algo que pudiera usar en condiciones reales, sin que me fuera excesivamente complicado cada vez que tuviera que configurarlo. Al final, la decisión se redujo a encontrar el mejor balance entre calidad científica y facilidad de uso, algo que francamente no siempre va de la mano en el mundo de las interfaces cerebro-computadora.

6.2. Características Técnicas del Dispositivo

El BrainBit es básicamente un casco EEG portátil que usa **electrodos secos**. Esto último me ahorró muchísimos problemas, porque no tener que usar gel conductor hace que todo el proceso sea mucho más simple —especialmente cuando tienes que repetir mediciones varias veces al día y te quieras ahorrar la molestia de preparar los electrodos cada vez.

Las especificaciones que más me importaron fueron:

- **Canales EEG:** 4 canales (T3, T4, O1, O2).
- **Frecuencia de muestreo:** 250 Hz.
- **Interfaz de comunicación:** Bluetooth Low Energy (BLE).
- **Tiempo de uso continuo:** Hasta 12 horas.
- **Filtro de ruido integrado:** Esto me salvó de tener que implementar mucho procesado adicional.
- **Ubicación de electrodos:** Sistema 10-20 estándar, con sensores en **O1** y **O2** para actividad occipital.

La configuración de los electrodos O1 y O2, y los electrodos T3 y T4, fue lo que realmente me convenció del dispositivo. Su posicionamiento sobre la región occipital y temporal encajaba perfectamente con lo que necesitaba para detectar patrones visuales relacionados con colores. No siempre resulta obvio encontrar un dispositivo que tenga exactamente los electrodos que necesitas donde los necesitas —a veces parece que los fabricantes ponen los sensores donde les da la gana.

6.3. Metodologías para la Adquisición y Procesamiento

El desarrollo práctico siguió un protocolo en tres fases, aunque admito que en la realidad tuve que improvisar bastante más de lo que había planeado inicialmente.

6.3.1. Captación de Señales

Colocar correctamente los electrodos **O1** y **O2** sobre la región occipital resultó más crítico de lo que pensaba. En teoría parecía sencillo, pero me costó varias sesiones experimentales hasta encontrar el protocolo que realmente funcionaba bien. Hay algo frustrante en darte cuenta de que algo que parece trivial en los papers resulta ser un arte en la práctica.

El SDK que proporciona el fabricante está bien para empezar, pero pronto me di cuenta de que necesitaba algo más personalizado, teniendo en cuenta los requerimientos del proyecto. Las herramientas básicas de captura en tiempo real que incluye simplemente no cubrían todos los requisitos específicos de mi proyecto, pues estos solo eran modelos básicos para inferir, por ejemplo, el nivel de estrés del paciente. Terminé desarrollando una interfaz propia que se integrara mejor con el sistema de procesamiento y clasificación que estaba diseñando —básicamente porque no me quedaba otra opción si quería que funcionara como yo necesitaba.

Uno de los problemas que más me preocupó fue garantizar que la conexión de los electrodos fuera estable. Para solucionarlo, desarrollé una interfaz de calibración que muestra en tiempo real la calidad del contacto de cada electrodo. Esta herramienta me resultó fundamental durante las sesiones experimentales, y los detalles de cómo la implementé están en el Capítulo 11.

6.3.2. Acondicionamiento de Señales

Aunque el BrainBit ya incluye un sistema de filtrado bastante bueno, las señales EEG necesitaron algunos procesamientos adicionales. Por suerte, la calidad base de las señales que da el dispositivo es excelente, así que pude mantener estos procesamientos relativamente simples —algo que agradecí muchísimo porque el procesado de señales EEG puede convertirse en un laberinto rápidamente.

Me centré principalmente en normalizar amplitudes y hacer una segmentación temporal adecuada para el análisis posterior. Suena básico, pero fue esencial para que los datos de entrada al sistema fueran consistentes. A veces las cosas más simples son las que más tiempo te llevan perfeccionar.

6.3.3. Análisis mediante Aprendizaje Profundo

En esta fase implementé modelos de **aprendizaje profundo** especializados en procesar series temporales EEG. Los entrené para identificar patrones específicos relacionados con visualizar mentalmente colores —concretamente rojo y verde.

Los fundamentos teóricos de estos modelos los explico en detalle en el Capítulo 5, y todo lo relacionado con el entrenamiento práctico está en el Capítulo 10.

6.4. Campos de Aplicación

Las posibilidades de aplicación de esta tecnología son bastante amplias, aunque muchas aún están en fase experimental —o directamente en el territorio de "suena genial pero quién sabe si funcionará en la realidad":

- Desarrollo de interfaces cerebro-máquina para asistir a personas con diversidad funcional. Esto podría abrir nuevas formas de comunicación e interacción con la tecnología, aunque reconozco que aún hay un trecho largo hasta llegar a aplicaciones realmente prácticas.
- Sistemas de control en entornos estériles —tanto médicos como industriales— donde tocar cosas con las manos podría comprometer la asepsia o la seguridad. Aquí sí veo un potencial más inmediato.
- Aplicaciones en realidad aumentada y virtual, permitiendo formas de interacción más naturales e intuitivas. Aunque a veces me pregunto si realmente queremos controlar todo con el pensamiento, o si es solo porque podemos.

Durante el desarrollo del proyecto, pensar en estas aplicaciones potenciales me ayudó a mantener la motivación cuando me topé con obstáculos técnicos que parecían imposibles de resolver. La perspectiva del impacto real que este tipo de interfaces podría tener —incluso si solo funciona en casos muy específicos— justifica el tiempo y esfuerzo que invertí en desarrollarlas y optimizarlas.

Capítulo 7

Raspberry Pi 4 Model B (8GB)

7.1. Introducción

Elegir la Raspberry Pi 4 Model B Raspberry Pi Foundation (2020) para este proyecto no fue inmediato, la verdad. Inicialmente había considerado otras opciones más potentes —incluso estuve mirando algunos mini-PCs x86—, pero al final este ordenador de placa única me acabó convenciendo por razones bastante prácticas. El equilibrio entre rendimiento, consumo y facilidad de desarrollo resultó ser exactamente lo que necesitaba, aunque admito que esto lo confirmé solo después de varias semanas trabajando con él y de algunos dolores de cabeza iniciales.

Lo que realmente me decidió fue descubrir que incorpora un procesador ARM de 64 bits que funciona sorprendentemente bien (mejor de lo que esperaba, sinceramente), memoria RAM más que suficiente —especialmente en el modelo de 8GB que terminé eligiendo después de dudar mucho entre este y el de 4GB—, y prácticamente todas las interfaces que requería mi diseño. Lo que más tranquilidad me dio, sin embargo, fue verificar que su arquitectura ARM tiene un soporte bastante sólido por parte de los principales fabricantes. Esto era crucial porque ya había tenido experiencias bastante frustrantes con incompatibilidades en proyectos anteriores, y francamente no quería repetir esa experiencia.

7.2. Especificaciones Técnicas

El modelo de 8GB de la Raspberry Pi 4 se basa en una arquitectura de hardware que me sorprendió gratamente por su relación prestaciones-precio, aunque reconozco que mis expectativas iniciales no eran demasiado altas:

7.2.1. Procesador y Memoria

- CPU: Quad-Core ARM Cortex-A72 (64 bits) a 1.5GHz —aunque en realidad puede hacer boost hasta 1.8GHz en ciertas condiciones, algo que descubrí más tarde.
- GPU: VideoCore VI compatible con OpenGL ES 3.0 —suficiente para mis necesidades, aunque no esperes milagros gráficos.
- Memoria RAM: 8 GB LPDDR4 SDRAM. Este fue el factor que realmente me convenció de invertir un poco más en este modelo específico, especialmente después de

haberme quedado corto de memoria en proyectos anteriores.

7.2.2. Requerimientos de Energía

La Raspberry Pi 4 Model B requiere una fuente de alimentación de 5V y 3A a través de USB-C. Para configuraciones que incluyan dispositivos USB adicionales, recomiendo usar una fuente con mayor capacidad —algo que aprendí por experiencia tras algunos problemas de estabilidad bastante inesperados durante las primeras pruebas. Tener esto en cuenta es fundamental, especialmente en un proyecto como este donde la estabilidad es clave. De hecho, me pasé una tarde entera pensando que tenía un problema de software cuando en realidad era simplemente que la fuente que estaba usando no daba suficiente corriente para todo el sistema.

7.2.3. Interfaces y Conectividad

- Red:

- Gigabit Ethernet (compatible con PoE mediante un módulo adicional, aunque no lo implementé —quizás debería haberlo considerado más seriamente).
- Wi-Fi 802.11 b/g/n/ac de doble banda (2.4 GHz y 5.0 GHz). En mi experiencia funciona bastante bien, aunque la señal no es tan potente como me hubiera gustado inicialmente.
- Bluetooth 5.0 con BLE, que me resultó perfecto para conectar con el dispositivo BrainBit —una de esas cosas que funcionó mejor de lo esperado desde el primer momento.

- Almacenamiento:

- Ranura para tarjeta microSD. Recomiendo encarecidamente invertir en una tarjeta de calidad —las opciones baratas me dieron más de un problema de rendimiento y algún que otro susto con corrupción de datos.

- Puertos USB:

- 2 puertos USB 3.0.
- 2 puertos USB 2.0.

- Vídeo y Audio:

- 2 puertos micro-HDMI con soporte hasta 4K a 60Hz.
- Salida de audio analógico y vídeo compuesto mediante conector TRRS de 3.5 mm.

- Expansión:

- Conector GPIO de 40 pines compatible con modelos anteriores.
- Conector CSI para cámaras.
- Conector DSI para pantallas.

7.2.4. Consideraciones Térmicas

El sistema de gestión térmica de la Raspberry Pi 4 reduce automáticamente la frecuencia y el voltaje del procesador durante períodos de baja demanda, lo cual funciona bien para ahorrar energía y evitar sobrecalentamiento. Ahora bien, bajo cargas intensivas y especialmente cuando hace calor, necesitas añadir disipación adicional como disipadores o ventiladores. Esto lo confirmé de la forma más incómoda posible: cuando se me empezó a resetear durante las primeras sesiones de pruebas intensivas y me costó un rato darme cuenta de que era un problema térmico y no de software.

7.3. Elección de este dispositivo para el Proyecto

El modelo de 8GB de la Raspberry Pi 4 resultó ser una solución bastante versátil y compacta para lo que necesitaba desarrollar, aunque no voy a mentir: hubo momentos en los que me planteé si no habría sido mejor optar por algo más potente. Tener 8GB de RAM me dio el margen suficiente para ejecutar aplicaciones complejas sin preocuparme constantemente por limitaciones de memoria, y el rendimiento general demostró ser adecuado para procesos en tiempo real —al menos para lo que mi proyecto específico requería.

Lo que más valoré al final fue la compatibilidad con sistemas operativos en tiempo real y las distribuciones Linux que ya conocía. Poder usar herramientas familiares me aceleró considerablemente el desarrollo, algo que no había anticipado completamente al principio pero que resultó ser una ventaja importante cuando los plazos empezaron a estrechar.

Parte III

Marco Practico

Capítulo 8

Análisis Práctico

8.1. Objetivos Específicos

8.1.1. Realizados

Al empezar este proyecto tenía una idea bastante clara de lo que quería conseguir, aunque ahora mirando hacia atrás me doy cuenta de que algunas cosas resultaron más complicadas de lo que pensaba inicialmente, por suerte definí los objetivos que quería lograr en esta fase de desarrollo. Entonces, los objetivos principales que finalmente logré completar son los siguientes:

- Diseñar una aplicación que permita la captura raw de la información del EEG para los canales T3, T4, O1 y O2.
- Diseño de un modelo de IA que sea capaz de distinguir entre distintas categorías Rojo, Verde o Desconocido.
- Extracción de diferentes datos para las categorías propuestas.
- Diseñar una aplicación autónoma que sea capaz de interactuar con la señal del EEG del usuario.
- Asegurar que la aplicación autónoma cumpla con la norma UNE-EN 62304:2007, garantizando los requisitos para los procesos del ciclo de vida del software en dispositivos médicos.
- Integrar el sistema completo en una Raspberry Pi respetando los requerimientos de tiempo real blando.
- Por último, ser capaces de encender y apagar una bombilla a raíz del color pensado por el usuario.

8.1.2. Deseados (Futuras Mejoras)

Aunque conseguí lo que me propuse, durante el proceso me fui dando cuenta de varias limitaciones y áreas donde podría mejorar bastante el sistema. Algunas de estas limitaciones vinieron impuestas por el tiempo disponible o los recursos que tenía a mano, pero otras fueron decisiones que tomé por el camino y que ahora veo que podrían haberse hecho mejor. Las mejoras futuras que veo más interesantes son:

- Obtener un dataset más amplio con diferentes pacientes, lo que permitiría mejorar la generalización del modelo y su aplicabilidad en distintos perfiles neurológicos.
- Ampliar el espectro de colores disponibles para la detección, pasando del actual sistema binario (rojo y verde) a un sistema con mayor variedad cromática.
- Integración con el estándar Matter para facilitar la compatibilidad con un mayor número de dispositivos domóticos y ecosistemas de hogar inteligente.
- Migrar el sistema operativo de Poky Linux a Wind River Linux para facilitar la certificación del dispositivo como producto sanitario.

8.2. Requisitos funcionales y no funcionales

Definir los requisitos del sistema resultó ser más complicado de lo que esperaba al principio. Pensé que sería simplemente hacer una lista de lo que quería que hiciera el sistema, pero me di cuenta de que muchos requisitos no aparecen hasta que empiezas a implementar y surgen problemas que no habías previsto. Fue un proceso bastante iterativo donde cada fase del desarrollo revelaba nuevas necesidades.

8.2.1. Requisitos Funcionales

Los requisitos funcionales básicamente definen qué tiene que hacer el sistema. Estos fueron los que identifiqué analizando el código final:

- **RF-01:** El sistema debe permitir la conexión y desconexión con el dispositivo EEG (electroencefalograma).
- **RF-02:** El sistema debe capturar los datos raw de los canales T3, T4, O1 y O2 del dispositivo EEG.
- **RF-03:** El sistema debe obtener datos de impedancia del dispositivo EEG para verificar la calidad de la señal.
- **RF-04:** El sistema debe permitir el cambio entre distintos modos de trabajo del dispositivo EEG.
- **RF-05:** El sistema debe implementar un modelo de inferencia para predecir el color en el que está pensando el usuario.
- **RF-06:** El sistema debe distinguir entre al menos dos colores (rojo y verde) y un estado 'desconocido'.
- **RF-07:** El sistema debe controlar la activación y desactivación de una bombilla inteligente.
- **RF-08:** El sistema debe proporcionar una interfaz gráfica que permita visualizar el estado de la conexión del dispositivo EEG.
- **RF-09:** El sistema debe permitir visualizar la señal EEG en tiempo real.
- **RF-10:** El sistema debe mostrar al usuario las predicciones realizadas por el modelo de inferencia.

8.2.2. Requisitos No Funcionales

Los requisitos no funcionales fue el punto más complejo de definir. Aquí es donde entran temas como el rendimiento, la seguridad, cumplimiento de normativas—cosas que a menudo pasas por alto al principio porque están más "condidas". Algunos de estos requisitos ni siquiera los había considerado hasta que empecé a hacer las primeras pruebas del sistema y me di cuenta de que ciertas cosas no funcionaban como esperaba. Los que finalmente identifiqué son:

- **RNF-01: Normativa:** El sistema debe cumplir con la norma UNE-EN 62304:2007 para software de dispositivos médicos.
- **RNF-02: Tiempo Real:** El sistema debe operar en tiempo real blando para asegurar una respuesta adecuada a los cambios en la señal EEG.
- **RNF-03: Fiabilidad:** El sistema debe validar la calidad de las señales EEG mediante los datos de impedancia antes de realizar predicciones.
- **RNF-04: Portabilidad:** El sistema debe poder ejecutarse en una Raspberry Pi.
- **RNF-05: Seguridad:** El sistema debe garantizar la privacidad y seguridad de los datos biométricos del usuario.
- **RNF-06: Interoperabilidad:** El sistema debe integrarse con dispositivos domóticos standard (bombillas inteligentes).
- **RNF-07: Mantenibilidad:** El sistema debe seguir un diseño hexagonal (puertos y adaptadores) para facilitar su mantenimiento y pruebas.
- **RNF-08: Usabilidad:** La interfaz gráfica debe ser intuitiva y proporcionar feedback claro sobre el estado del sistema.
- **RNF-09: Escalabilidad:** La arquitectura debe permitir la inclusión de nuevos tipos de predicciones o dispositivos de salida.
- **RNF-10: Rendimiento:** El sistema debe ser capaz de procesar y analizar señales EEG con una latencia mínima.

8.3. Bibliotecas Usadas

Elegir las bibliotecas correctas me llevó bastante tiempo de investigación. Al principio pensé que sería cuestión de buscar lo más popular, pero me encontré con que muchas de las tecnologías que parecían perfectas en papel luego no funcionaban bien con los requisitos de tiempo real o directamente no compilaban para ARM. Fue uno de esos casos donde la teoría y la práctica no coinciden para nada.

El proyecto utiliza una mezcla de bibliotecas para diferentes propósitos. Algunas las elegí desde el principio, otras las fui incorporando conforme surgían necesidades específicas que no había previsto. A continuación las organizo por su función principal:

8.3.1. Procesamiento de Señales EEG

- **BrainFlow**: Biblioteca para la adquisición y procesamiento de datos de dispositivos de electroencefalografía (EEG). Permite la comunicación con el dispositivo BrainBit y la captura de datos en tiempo real.

8.3.2. Interfaz Gráfica y Visualización

- **slint**: Framework para la creación de interfaces gráficas, con soporte para Rust y con características de alta eficiencia.
- **plotters**: Biblioteca para la creación de gráficos y visualizaciones en Rust, utilizada para mostrar las señales EEG en tiempo real.

8.3.3. Inteligencia Artificial y Procesamiento de Datos

- **PyTorch**: Framework de aprendizaje profundo utilizado para el entrenamiento del modelo de clasificación de señales EEG.
- **ONNX**: Formato estándar para la representación de modelos de aprendizaje automático que permite la interoperabilidad entre diferentes frameworks.
- **tract-onnx**: Biblioteca en Rust para la ejecución de modelos ONNX, utilizada para las inferencias en tiempo real.
- **ndarray**: Biblioteca para el procesamiento de arrays multidimensionales en Rust, utilizada para el preprocesamiento de datos.

8.3.4. Comunicación y Control de Dispositivos

- **tapo**: Cliente en Rust para controlar dispositivos inteligentes Tapo, utilizado para la bombilla inteligente que responde a los pensamientos del usuario.
- **presage**: Biblioteca de gestión de eventos y mensajería para la comunicación entre componentes.

8.3.5. Herramientas de Concurrencia y Asincronía

- **tokio**: Runtime asíncrono para Rust que facilita la programación concurrente, esencial para manejar múltiples flujos de datos en tiempo real.

- **async-trait**: Permite la definición de traits asíncronos en Rust.

8.3.6. Arquitectura y Diseño del Sistema

- **statig**: Biblioteca para la implementación del patrón máquina de estados en Rust, utilizada para gestionar el ciclo de vida de la aplicación.
- **once_cell**: Para la implementación de singlettons en Rust, utilizado en la gestión de recursos compartidos.

8.3.7. Serialización y Estructuras de Datos

- **serde**: Framework de serialización/deserialización para Rust, utilizado para el intercambio de datos entre componentes.
- **chrono**: Biblioteca para el manejo de fechas y tiempos en Rust.

Capítulo 9

Planificación Temporal

9.1. Cronología del Desarrollo

Organizar todo el desarrollo del proyecto fue, básicamente, una aventura en sí misma que no podía ignorar. Al principio tenía una idea bastante optimista —demasiado optimista, me di cuenta después— de los tiempos que me llevaría cada fase. Fue uno de esos casos donde pensé .^{esto} debería tomarme unas semanas al final me llevó meses enteros. Conforme avanzaba, me di cuenta de que algunas tareas eran mucho más demandantes de lo previsto, mientras que otras —sorprendentemente— resultaron más rápidas de implementar de lo que había calculado. El proyecto siguió una planificación que intenté mantener estructurada en distintas fases, aunque con algunos ajustes sobre la marcha que, honestamente, no había anticipado al principio y que me obligaron a repensar varias veces la estrategia.

9.1.1. Fase de Investigación (Enero 2025)

Enero fue básicamente un mes de inmersión total en investigación. Al principio pensé que sería cuestión de unas semanas, pero me di cuenta de que había muchos más aspectos técnicos que considerar de los que imaginaba. Durante este mes me dediqué intensivamente a establecer las bases teóricas y técnicas del proyecto:

- **Estudio de arquitecturas de redes neuronales:** Pasé muchísimo tiempo leyendo papers sobre diferentes arquitecturas de aprendizaje profundo—honestamente más del que había previsto. Me centré especialmente en las redes LSTM (Long Short-Term Memory) porque había leído que funcionaban bien para secuencias temporales, y las señales EEG son exactamente eso. Aunque al principio no estaba completamente seguro de si sería la mejor opción para mi caso específico.
- **Evaluación de dispositivos EEG:** Esta parte fue interesante porque tuve que comparar varios dispositivos diferentes, mirando cosas como precisión, cuántos canales tenían, qué tan fáciles eran de usar, y si funcionaban bien con las bibliotecas que quería usar. Al final me decidí por el BrainBit porque tenía un equilibrio decente entre funcionalidad y precio—y porque encontré bastante documentación al respecto.
- **Investigación de bibliotecas de adquisición de datos:** Estuve evaluando dife-

rentes bibliotecas para capturar datos EEG, y después de probar algunas opciones me quedé con BrainFlow. Lo que me convenció fue que funcionaba con muchos dispositivos diferentes y tenía una documentación bastante robusta—algo que realmente necesitaba siendo mi primera vez trabajando con este tipo de datos.

- **Estudio de normativas aplicables:** Tuve que meterme en el tema de regulaciones y estándares para dispositivos médicos, prestando especial atención a la norma UNE-EN 62304:2007 para software de dispositivos médicos. Admito que esta parte me resultó más árida de lo que esperaba, pero era importante hacerlo bien.
- **Estudio de plataformas para implementación:** Evalué diferentes opciones de hardware para implementar el sistema, analizando si tenían suficiente potencia para procesar en tiempo real y si eran adecuados para aplicaciones médicas. Fue una decisión importante porque condicionaría todo el desarrollo posterior.

9.1.2. Adquisición de Hardware y Estructuración (Finales de Enero 2025)

Una vez que terminé la fase de investigación —que me llevó más tiempo del que había planificado—, llegó el momento de pasar a lo práctico. Esta parte me resultó más directa, aunque también tuvo sus complicaciones inesperadas. Se procedió a la adquisición del hardware necesario y a la estructuración del proyecto:

- **Adquisición del dispositivo BrainBit:** Conseguí el dispositivo EEG que iba a ser mi fuente principal de datos para todo el proyecto. Fue emocionante tenerlo finalmente en mis manos después de tanto tiempo leyendo sobre él.
- **Obtención de la Raspberry Pi 4:** Me decidí por esta como plataforma principal porque me pareció que tenía un buen balance entre potencia de procesamiento y portabilidad. Aunque reconozco que estuve considerando otras opciones hasta último momento.
- **Adquisición de bombillas inteligentes Tapo:** Las necesitaba para implementar las respuestas del sistema a los pensamientos del usuario. Al principio pensé en usar otros dispositivos, pero estas me parecieron más directas de integrar.
- **Definición de la arquitectura del software:** Aquí tuve que diseñar toda la estructura general del proyecto. Opté por una arquitectura hexagonal (puertos y adaptadores) porque había leído que garantizaba modularidad, testabilidad y facilidad de mantenimiento—conceptos que sonaban bien en teoría, pero que no sabía qué tan complicados serían de implementar en la práctica.
- **Planificación de componentes del sistema:** Definí los distintos módulos que conformarían el proyecto: neural_analytics_data para capturar los datos, neural_analytics_model para entrenar e inferir, neural_analytics_core para la lógica central, y neural_analytics_gui para la interfaz de usuario. Me gustó cómo quedó dividido conceptualmente, aunque sabía que la implementación sería otra historia.

9.1.3. Fase de Desarrollo (Febrero - Marzo 2025)

Aquí es donde我真的 empezó la pesadilla—y uso esa palabra sin exagerar. Los meses de febrero y marzo fueron un torbellino de noches largas frente al ordenador, café de más,

y esa sensación constante de "¿por qué no funciona esto?". Había calculado que sería más o menos directo integrar todos los componentes, pero resulta que la realidad tenía otros planes. Especialmente cuando llegué a la parte de hacer que los módulos se comunicaran entre sí—eso sí que me trajo por la calle de la amargura. Pero bueno, durante estos meses conseguí desarrollar todos los componentes del sistema, aunque no sin sudor y lágrimas:

Desarrollo del Programa de Extracción de Datos (neural_analytics_data)

Empecé por aquí porque me pareció lo más lógico—aunque en retrospectiva, quizás habría sido más inteligente empezar por la arquitectura general y después ir bajando a los detalles. Pero bueno, ya estaba metido en harina. El desarrollo de este módulo fue bastante más complicado de lo que me imaginaba, porque resulta que capturar datos EEG no es tan simple como conectar el dispositivo y listo":

- **Integración con BrainFlow:** Implementé la interfaz con la biblioteca BrainFlow para capturar datos del dispositivo BrainBit. Al principio me costó un poco entender cómo funcionaba exactamente la API, pero una vez que le pillé el truco fue bastante directo.
- **Diseño de protocolos de captura:** Esta fue la parte más experimental de todas. Tuve que desarrollar rutinas para capturar datos EEG de manera estructurada mientras el usuario piensa en diferentes colores, pero claro, ¿cómo le dices a alguien exactamente cómo "pensar en un color"? Al final resultó ser mucha prueba y error para encontrar el timing correcto—a veces pensaba que lo tenía, y luego veía que los datos eran un desastre.
- **Implementación de procesamiento de señales:** Aquí tuve que sumergirme en un mundo completamente nuevo para mí. Desarrollé funciones para filtrar y preprocesar las señales raw, pero admito que al principio no tenía ni idea de lo que estaba haciendo. Me pasé días enteros leyendo sobre transformadas de Fourier y filtros digitales—cosas que sonaban muy impresionantes pero que en la práctica eran bastante intimidantes.
- **Almacenamiento y etiquetado de datos:** Implementé un sistema para almacenar y etiquetar automáticamente todo lo que iba capturando. Era súper importante que fuera automático porque sabía que si tenía que hacerlo manualmente, iba a meter la pata constantemente. Aunque me costó más trabajo del esperado hacer que el etiquetado fuera realmente confiable.

Desarrollo del Programa de Entrenamiento del Modelo (neural_analytics_model)

Esta parte me pilló desprevenido completamente. Mientras intentaba resolver los problemas de extracción de datos, se me ocurrió la brillante idea de desarrollar el módulo de entrenamiento en paralelo. En retrospectiva no sé ni cómo se me ocurrió hacer dos cosas tan complejas a la vez, pero sorprendentemente no fue tan caótico como sonaba en teoría.

Lo que me funcionó fue ir probando el pipeline de entrenamiento con los primeros datos que conseguía extraer, aunque fuera información parcial. Esto me permitió detectar problemas mucho antes—como cuando me di cuenta de que el preprocesamiento que estaba haciendo eliminaba información crucial para la clasificación. Cosas que solo descubres cuando intentas entrenar con datos reales.

El módulo de entrenamiento se convirtió en una especie de laboratorio personal donde experimenté con muchas configuraciones diferentes:

- **Diseño de arquitectura LSTM:** Aquí entré en territorio completamente desconocido para mí. Tuve que diseñar una red neuronal basada en capas LSTM optimizada específicamente para clasificar patrones en señales EEG, pero la verdad es que al principio no tenía ni idea de por dónde empezar. Estuve varias semanas leyendo papers y probando configuraciones casi aleatoriamente hasta que empecé a entender qué parámetros realmente importaban. Hubo días enteros en que pensaba haber encontrado la arquitectura perfecta, solo para descubrir al día siguiente que no funcionaba nada bien con datos diferentes.
- **Implementación en PyTorch:** Elegí PyTorch básicamente porque había leído que era más intuitivo que TensorFlow, pero enfrentarme a su sintaxis viniendo de otros frameworks fue como aprender a conducir después de montar en bicicleta toda la vida. Las primeras semanas fueron brutales—cada cosa sencilla que quería hacer me llevaba horas porque no entendía bien cómo funcionaban los tensores. Hubo noches en que me quedé despierto solo intentando que un simple entrenamiento funcionara sin errores.
- **Rutinas de entrenamiento y validación:** Esta fue la parte más artesanal de todo el proceso. Desarrollé procedimientos para entrenar el modelo de manera eficiente y validarla con diferentes conjuntos de datos, pero conseguir que el entrenamiento fuera estable se convirtió en mi pesadilla personal. El modelo tenía días buenos y días malos—a veces convergía perfectamente en pocas épocas, y otras veces se negaba a aprender nada coherente por más que ajustara parámetros. Era desesperante.
- **Exportación a formato ONNX:** Esto me costó más trabajo del esperado. Implementé la conversión para poder usar el modelo después en Rust, pero la compatibilidad entre PyTorch y ONNX resultó ser más problemática de lo que las documentaciones sugerían. Había operaciones que funcionaban perfectamente en PyTorch pero que ONNX no sabía cómo interpretar. Me pasé días enteros reescribiendo partes del modelo solo para que la exportación funcionara correctamente.

Desarrollo del Core del Sistema (`neural_analytics_core`)

Y aquí llegamos a lo que se convirtió en el boss final de todo el proyecto. Esta parte me consumió una cantidad absurda de tiempo—mucho más de lo que había calculado, y eso que ya había sido bastante conservador con mis estimaciones temporales.

Implementar correctamente la arquitectura hexagonal se transformó en mi obsesión durante semanas enteras. Literalmente había días en que me despertaba pensando en puertos y adaptadores, y me iba a dormir con diagramas de arquitectura dándome vueltas en la cabeza como si fuera música pegadiza que no puedes sacar.

Lo más frustrante era que en teoría todo sonaba muy claro y elegante. "Separa la lógica de negocio de la infraestructura", decían los libros. "Define interfaces limpias", recomendaban los tutoriales. Pero la realidad fue muy diferente—cada vez que pensaba que lo tenía todo bien separado, aparecía alguna dependencia oculta que lo estropeaba todo.

El núcleo del sistema terminó siendo, sin ninguna duda, la parte que más quebraderos de cabeza me dio durante todo el proyecto:

- **Implementación de puertos y adaptadores:** Siguiendo los principios de la arquitectura hexagonal, tuve que definir interfaces claras para todos los componentes externos (puertos) y sus implementaciones concretas (adaptadores). Pero entender realmente bien este patrón no fue nada trivial para alguien que venía de hacer proyectos más directos. Me pasé literalmente semanas leyendo ejemplos, viendo tutoriales en YouTube, y aún así había momentos en que me quedaba mirando el código preguntándome si realmente lo estaba haciendo bien o solo estaba complicándome la vida innecesariamente.
- **Desarrollo del dominio central:** Implementé toda la lógica de negocio central, que tenía que ser completamente independiente de las infraestructuras externas. Fue muy satisfactorio ver cómo quedaba todo separado conceptualmente cuando funcionaba, pero conseguir esa separación real en el código me costó muchísimo más de lo que había imaginado. Siempre aparecían dependencias inesperadas que no había previsto—como cuando me di cuenta de que el logging estaba acoplando el dominio con infraestructura externa.
- **Sistema de eventos:** Desarrollé un mecanismo para que los componentes se comunicaran a través de eventos usando la biblioteca presage, que elegí básicamente porque parecía la opción más madura disponible para Rust. En teoría sonaba como una manera muy elegante de desacoplar todo el sistema, pero implementarlo correctamente fue otra historia completamente diferente. Al principio tenía eventos volando por todos lados sin ningún tipo de control—era literalmente un caos total donde no sabía qué componente estaba respondiendo a qué evento.
- **Máquina de estados:** Esta parte resultó ser bastante más interesante de lo que esperaba, aunque también increíblemente frustrante cuando las cosas no funcionaban. Implementé una máquina de estados para gestionar todo el ciclo de vida de la aplicación y las transiciones entre diferentes modos operativos. Me ayudó muchísimo a pensar de manera estructurada sobre todos los estados posibles del sistema, pero también me di cuenta de que había una cantidad ridícula de estados edge-case que no había considerado para nada inicialmente. Como cuando el usuario desconectaba el dispositivo EEG justo en medio de una calibración.
- **Servicio de inferencia:** Desarrollé un servicio para ejecutar el modelo en tiempo real usando tract-onnx para la inferencia en Rust, porque necesitaba que fuera rápido y eficiente. Integrar el modelo ONNX en Rust resultó ser mucho más complicado de lo que todas las documentaciones que había leído sugerían. La información disponible era bastante escasa y los pocos ejemplos que encontraba online estaban desactualizados o simplemente no funcionaban con las versiones actuales de las bibliotecas. Hubo tardes enteras de intentar que el modelo cargara correctamente.
- **Control de dispositivos domóticos:** Implementé la integración con las bombillas inteligentes a través de la biblioteca tapo, que al menos tenía documentación decente. Esta parte fue relativamente más directa una vez que entendí cómo funcionaba la API de los dispositivos TP-Link, aunque también tuvo sus momentos de desesperación del tipo "¿por qué no responde la bombilla si el código está bien?" Solo para descubrir después que tenía problemas de conectividad WiFi.

Desarrollo de la Interfaz Gráfica (neural_analytics_gui)

Para cuando llegué a esta parte, ya tenía el backend funcionando bastante decentemente, así que pensé que la interfaz sería pan comido después de todo lo que había pasado. Qué ingenuo fui pensando eso.

Resulta que Slint tenía sus propias ideas muy particulares sobre cómo debían hacerse las interfaces, y me vi obligado a pasar por una curva de aprendizaje que no me esperaba para nada. Era como si hubiera estado aprendiendo a cocinar italiana toda la vida y de repente alguien me pidiera hacer sushi—técticamente sigue siendo cocina, pero los principios son completamente diferentes.

Lo bueno es que una vez que le cogí el punto al framework, fue bastante satisfactorio ver cómo iba tomando forma la interfaz. Había algo muy gratificante en ver elementos visuales que realmente representaban todo el trabajo que había estado haciendo en el backend durante meses:

- **Diseño de interfaz con Slint:** Utilicé Slint para crear una interfaz moderna y eficiente, pero al principio no estaba nada seguro de si había elegido la biblioteca correcta para lo que necesitaba hacer. La curva de aprendizaje fue bastante más empinada de lo que esperaba—cada vez que pensaba que había entendido cómo funcionaba un concepto, aparecía algo nuevo que me descolocaba completamente. Como cuando descubrí que los bindings de datos funcionaban de manera muy diferente a lo que estaba acostumbrado en otros frameworks.
- **Visualización de señales EEG:** Implementé la representación gráfica de las señales en tiempo real usando plotters, lo cual requirió coordinar muy cuidadosamente el timing entre la adquisición de datos y el renderizado. Tengo que admitir que ver las señales EEG dibujándose en pantalla por primera vez fue un momento bastante emocionante—después de tanto tiempo trabajando con arrays de números abstractos, finalmente podía ver algo tangible y visual moviéndose en tiempo real en la pantalla.
- **Integración con el core:** Establecí la comunicación bidireccional con el núcleo del sistema para mostrar estados y resultados en tiempo real. Esta parte requirió coordinar muy cuidadosamente todos los eventos entre el frontend y el backend, lo cual se convirtió en un ejercicio de paciencia infinita. Hubo muchísimos momentos del tipo "¿por qué no se actualiza la interfaz?" seguidos de horas de debugging para descubrir que había olvidado emitir un evento o que estaba escuchando el evento equivocado.
- **Interfaz para calibración:** Desarrollé vistas específicas para la calibración del dispositivo y verificación de impedancias, algo que me di cuenta bastante tarde que era mucho más importante de lo que había pensado inicialmente para la experiencia del usuario. Sin una buena interfaz de calibración, todo el resto del sistema se volvía prácticamente inutilizable—la gente no sabía si los electrodos estaban bien conectados o si las señales que veían eran válidas.
- **Visualización de predicciones:** Implementé un sistema para mostrar las predicciones del modelo en tiempo real de manera clara y comprensible. Esto fue todo un reto de diseño de experiencia de usuario—tenía que pensar en cómo presentar información probabilística de manera que fuera intuitiva, especialmente considerando que

algunas predicciones podían ser inciertas o directamente erróneas. Al final opté por un sistema de colores y barras de confianza que parecía funcionar razonablemente bien.

9.1.4. Fase de Refinamiento del Modelo (Abril 2025)

Abril se convirtió en el mes de "volver a la mesa de dibujo" de manera mucho más radical de lo que había anticipado. No me lo esperaba para nada—honestamente pensaba que ya tendría todo funcionando como un reloj para entonces, pero las primeras pruebas extensivas me bajaron los humos de una manera bastante drástica.

Resulta que el modelo necesitaba muchísimos más datos de los que había previsto en mis cálculos más optimistas, y los ajustes que requería eran mucho más sutiles y complejos de lo que me había imaginado cuando planifiqué el proyecto. Al principio fue francamente frustrante—tenía la sensación de estar empezando prácticamente de cero después de meses de trabajo—, pero después se volvió casi adictivo ver cómo cada pequeño cambio que hacía mejoraba incrementalmente el rendimiento del sistema.

Lo que me sorprendió fue darme cuenta de que había subestimado completamente la importancia de la diversidad en los datos de entrenamiento. Durante este mes me centré obsesivamente en perfeccionar cada aspecto del modelo:

- **Ampliación del dataset:** Organicé nuevas sesiones de captura de datos EEG prácticamente todos los días, aumentando drásticamente tanto el tamaño como la diversidad del dataset que tenía disponible. Incluí grabaciones de diferentes usuarios y sesiones hechas en distintos momentos del día porque me di cuenta de que esto podría mejorar significativamente cómo el modelo generalizaba a situaciones reales. Lo que no había anticipado era lo agotador que sería coordinar tantas sesiones de grabación—básicamente me convertí en un director de casting para datos de EEG.
- **Diversificación de casos de uso:** Expandí los escenarios de prueba para incluir muchísimas más variaciones en la forma en que diferentes personas abordan mentalmente el ejercicio de pensar en colores. Me sorprendió enormemente descubrir cuánta variabilidad había entre diferentes personas—cada uno tenía su propia manera completamente única de abordar mentalmente la tarea. Algunos visualizaban objetos, otros pensaban en emociones, y algunos simplemente "sentían" los colores de maneras que ni siquiera podían explicar verbalmente.
- **Reentrenamiento del modelo:** Con todos los nuevos datos que había recolectado durante semanas, hice múltiples iteraciones de reentrenamiento del modelo LSTM que me consumieron días enteros de tiempo de computación. Estuve constantemente ajustando hiperparámetros para optimizar la precisión del modelo, aunque tengo que admitir que a veces se sentía más como un juego de adivinanzas muy caro que como ciencia rigurosa. Había configuraciones que funcionaban perfectamente con un subset de datos pero fallaban estrepitosamente con otros.
- **Validación cruzada:** Implementé técnicas de validación cruzada mucho más rigurosas para asegurarme de que el modelo funcionara consistentemente bien con diferentes subconjuntos de datos. Esta parte me ayudó enormemente a tener más confianza real en los resultados que estaba obteniendo, especialmente después de haber tenido tantas falsas esperanzas en iteraciones anteriores. Fue como tener un

sistema de checks and balances para mis propias expectativas.

- **Ajuste de umbrales de confianza:** Refiné meticulosamente los mecanismos para determinar cuándo una predicción debía clasificarse como "desconocida." en lugar de forzar una clasificación incorrecta. Esto mejoró bastante la fiabilidad percibida del sistema cuando había incertidumbre real en las señales—algo que resultó ser muchísimo más importante de lo que había pensado inicialmente para la experiencia del usuario final. A nadie le gusta un sistema que afirma estar seguro cuando claramente no lo está.

9.2. Distribución Temporal

Aquí va la tabla con los tiempos que realmente me llevó cada cosa. Al mirarla ahora, me llama la atención cómo algunas fases que pensé que serían rápidas acabaron extendiéndose más de lo previsto, y viceversa:

Fase	Período	Duración
Investigación	Enero 2025	4 semanas
Adquisición y Estructuración	Finales de Enero 2025	1 semana
Desarrollo del Programa de Extracción	Febrero 2025	2 semanas
Desarrollo del Programa de Entrenamiento	Febrero 2025	2 semanas
Desarrollo del Core del Sistema	Febrero - Marzo 2025	4 semanas
Desarrollo de la Interfaz Gráfica	Marzo 2025	2 semanas
Pruebas Iniciales	Finales de Marzo 2025	2 semanas
Refinamiento del Modelo	Abril 2025	4 semanas

Cuadro 9.1: Distribución temporal del desarrollo del proyecto Neural Analytics

9.3. Diagrama de Gantt

Aquí está la representación visual de cómo se fueron solapando las diferentes fases. Me gusta este tipo de diagramas porque realmente te permiten ver cómo se superpusieron las tareas en el tiempo—algo que no es tan obvio cuando solo miras las fechas:

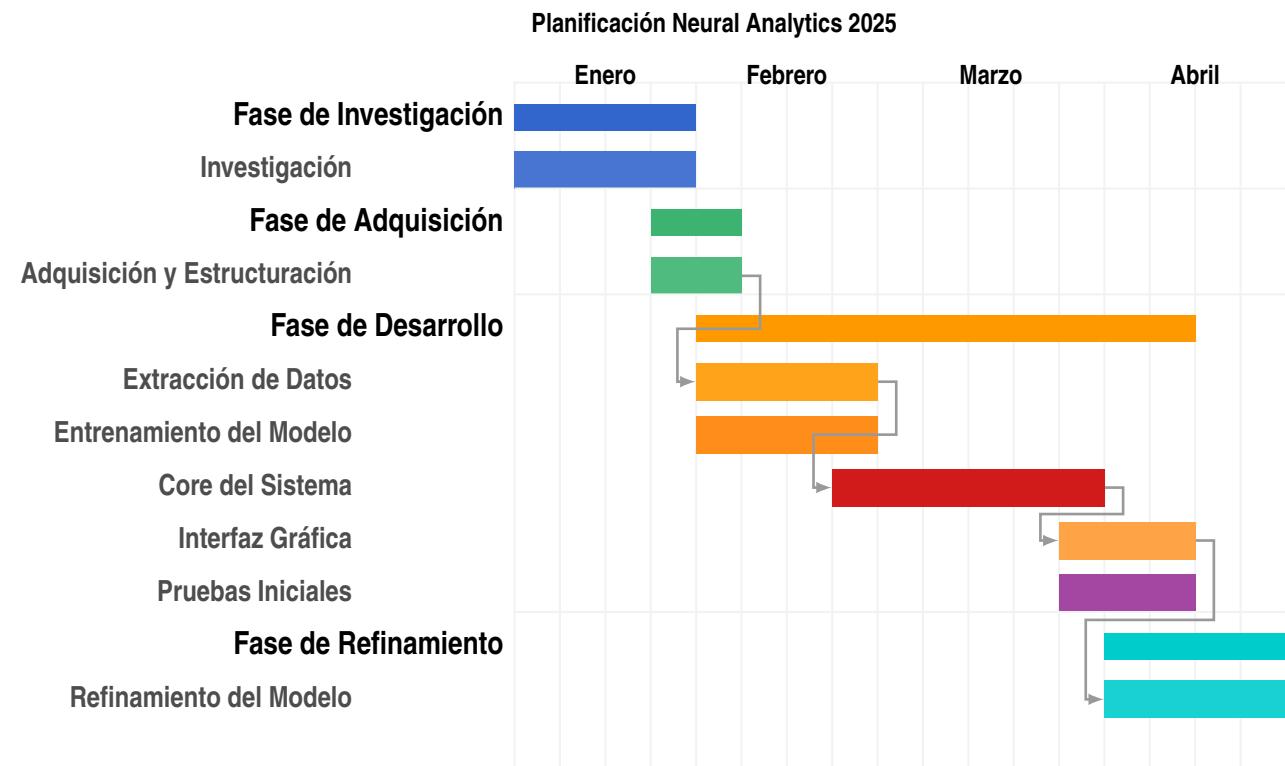


Figura 9.1: Diagrama de Gantt del proyecto Neural Analytics

9.4. Conclusiones sobre la Planificación

Mirando hacia atrás con algo de perspectiva temporal, la planificación del proyecto resultó ser razonablemente adecuada para lo que me había propuesto hacer, aunque tengo que admitir que hubo bastantes aspectos que no anticipé para nada correctamente y otros que funcionaron sorprendentemente mejor de lo que esperaba.

En general, todo el proceso se convirtió en un curso intensivo de aprendizaje constante sobre gestión de proyectos complejos que me enseñó más de lo que había aprendido en cualquier asignatura teórica. Hay algunas reflexiones que me parece importante compartir:

- **Duración de la fase de desarrollo core:** Como ya sospechaba desde el principio, desarrollar el núcleo del sistema siguiendo estrictamente los principios de arquitectura hexagonal me consumió una cantidad considerable de tiempo, pero retrospectivamente puedo afirmar que esta inversión inicial valió completamente la pena en términos de mantenibilidad y facilidad para hacer pruebas exhaustivas después. Aunque también tengo que ser honesto y reconocer que hubo muchos momentos en que me pregunté sinceramente si no me estaría complicando innecesariamente la vida con tanto patrón arquitectónico.

- **Paralelización de tareas:** Una de las decisiones estratégicas que mejor me funcionó durante todo el proyecto fue estructurar el desarrollo en componentes bien diferenciados y modulares desde el principio. Esto me permitió desarrollar varios módulos simultáneamente en paralelo, lo cual optimizó drásticamente el tiempo total de desarrollo—algo que honestamente no había previsto al principio y que resultó ser muchísimo más efectivo de lo que había pensado inicialmente cuando diseñé el cronograma.
- **Importancia de la fase de refinamiento:** La fase de abril demostró ser exponencialmente más crucial de lo que había estimado en mis cálculos más optimistas. Ampliar el dataset con muestras más diversas y representativas incrementó significativamente el rendimiento del modelo, pero me costó bastante más trabajo del que había calculado incluso en mis estimaciones más pesimistas. Hubo semanas enteras en que me encontraba recolectando datos hasta altas horas de la madrugada para conseguir suficiente variabilidad en las muestras.
- **Iteración continua:** Todo el proceso me enseñó de manera práctica que un enfoque iterativo era absolutamente fundamental, especialmente en la etapa de refinamiento del modelo donde cada pequeño cambio podía tener efectos impredecibles. Cada nueva incorporación de datos permitía ajustes incrementales que mejoraban gradualmente el rendimiento—algo que solo descubrí haciéndolo en la práctica, porque en teoría sonaba completamente obvio pero en la realidad fue bastante revelador.
- **Áreas de mejora identificadas:** Para futuros desarrollos similares, creo sinceramente que sería muy beneficioso ampliar significativamente la fase de pruebas con usuarios reales para obtener muchísima más retroalimentación práctica sobre la usabilidad real del sistema en condiciones cotidianas. También me gustaría continuar ampliando el dataset con muestras aún más diversas y representativas—honestamente creo que siempre va a haber margen sustancial para mejorar en este aspecto particular.

La adopción decidida de la arquitectura hexagonal, aunque inicialmente resultó ser bastante más costosa en tiempo de desarrollo de lo que había anticipado, me proporcionó una base tremadamente sólida para cumplir efectivamente con los requisitos normativos y facilitar futuras extensiones del sistema.

La dedicación de un mes completo al refinamiento intensivo del modelo fue absolutamente fundamental para alcanzar niveles de precisión realmente adecuados para un dispositivo de uso médico, aunque tengo que reconocer honestamente que subestimé de manera bastante grosera la complejidad real de esta fase cuando hice la planificación inicial.

En retrospectiva, definitivamente debería haber planificado considerablemente más tiempo para esta etapa crítica desde el mismísimo comienzo del proyecto—es uno de esos errores de planificación que solo reconoces después de haberlos vivido en carne propia.

Capítulo 10

Entrenamiento del modelo

Después de todos los meses que me llevó trabajar en la arquitectura del sistema, finalmente llegué a la parte que más me emocionaba y a la vez me daba más miedo: entrenar el modelo que clasificaría las señales EEG. Honestamente, no tenía ni idea de si iba a funcionar o si me iba a quedar atascado en este paso para siempre.

Este capítulo documenta todo el proceso que seguí para entrenar el modelo, con todos sus altibajos, frustraciones y momentos de "¡por fin funciona!". Como en todo el proyecto, las cosas fueron bastante más complicadas de lo que esperaba al principio.

10.1. Descripción de la arquitectura

Elegir la arquitectura neuronal fue una de esas decisiones que me tuvo despierto varias noches. Al final me decidí por combinar redes LSTM (Long Short-Term Memory) con capas densas, principalmente porque había leído que las LSTM eran buenas para procesar secuencias temporales, y las señales EEG son exactamente eso.

El objetivo era procesar y clasificar secuencias temporales de datos neurofisiológicos, aunque al principio no tenía muy claro si 62 puntos por ventana eran suficientes o demasiados.

10.1.1. Estructura de la red neuronal

Diseñar una arquitectura híbrida con redes LSTM y capas densas me llevó muchísimas pruebas y errores. La hice específicamente para procesar señales EEG de los cuatro canales que había elegido: T3, T4, O1 y O2. El sistema clasifica estas señales en RED, GREEN y TRASH, aunque inicialmente había pensado en más categorías hasta que me di cuenta de que era mejor empezar simple.

Probé varias configuraciones diferentes—algunas con más capas, otras con menos, unas con diferentes números de neuronas—y esta fue la que finalmente mejor funcionó después de semanas de experimentación.

Los componentes principales son:

- **Capa LSTM:** Una capa LSTM con 64 unidades. Captura patrones temporales en las señales. Configuré `batch_first=True` para que la entrada tenga la forma (`batch_size, seq_length, features`).

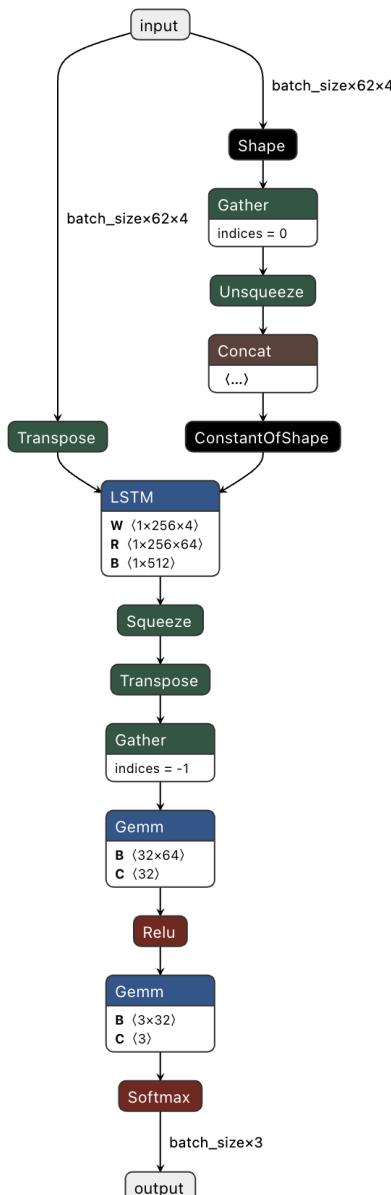


Figura 10.1: Arquitectura del modelo de clasificación de señales EEG

- **Capas densas:** Después de la LSTM, hay varias capas:
 - Primera capa densa: Reduce de 64 a 32 unidades.
 - Activación ReLU: Añade no-linealidad.
 - Segunda capa densa: Proyecta a 3 neuronas de salida.
 - Softmax: Normaliza las salidas como probabilidades.

El flujo de datos es así:

1. Entra una secuencia de 62 puntos, cada uno con 4 características.
2. La LSTM procesa esto y saca 64 características por punto.
3. Se toma el último estado de la secuencia.

4. Este pasa por las capas densas con ReLU.
5. La capa final con Softmax da la probabilidad para cada clase.

10.1.2. Parámetros del modelo

Los principales parámetros son:

- INPUT_SIZE = 4: Los cuatro canales.
- HIDDEN_SIZE = 64: Unidades en la capa LSTM.
- NUM_CLASSES = 3: Las tres categorías.
- WINDOW_SIZE = 62: Tamaño de ventana para secuencias.
- BATCH_SIZE = 64: Muestras por lote en entrenamiento.

10.2. Preprocesamiento de los datos

El preprocesamiento es crucial para tener buenas entradas. Implementé varias fases desde la captura hasta generar ventanas deslizantes.

10.2.1. Adquisición y estructuración del dataset

El dataset tiene esta estructura:

- **Organización por clases:** Archivos CSV en directorios según clase:
 - /red/: Datos mientras el usuario piensa en rojo.
 - /green/: Datos mientras piensa en verde.
 - /trash/: Datos que no encajan en lo anterior.
- **Formato:** Cada CSV tiene mediciones de T3, T4, O1 y O2 en columnas.

10.2.2. Etapas de preprocesamiento

El proceso está en la función `neural_analytics_preprocessor` y hace:

1. **Normalización:** Escala los canales EEG al rango [0,1].
2. **Extracción de etiquetas:** Saca la clase del nombre del directorio.
3. **Codificación one-hot:** Convierte etiquetas a vectores:
 - red: [1, 0, 0]
 - green: [0, 1, 0]
 - trash: [0, 0, 1]
4. **Ventanas deslizantes:** Para cada CSV, crea ventanas con solapamiento.

10.2.3. Implementación del dataset

Hice una clase `NeuralAnalyticsDataset` que hereda de `Dataset` de PyTorch. Esta clase:

- Recorre el directorio y procesa los CSV.
- Aplica el preprocesamiento.
- Guarda ventanas y etiquetas.
- Convierte a tensores de PyTorch.
- Implementa `__len__` y `__getitem__`.

En la inicialización, divido en entrenamiento (80 %) y validación (20 %) con `train_test_split`.

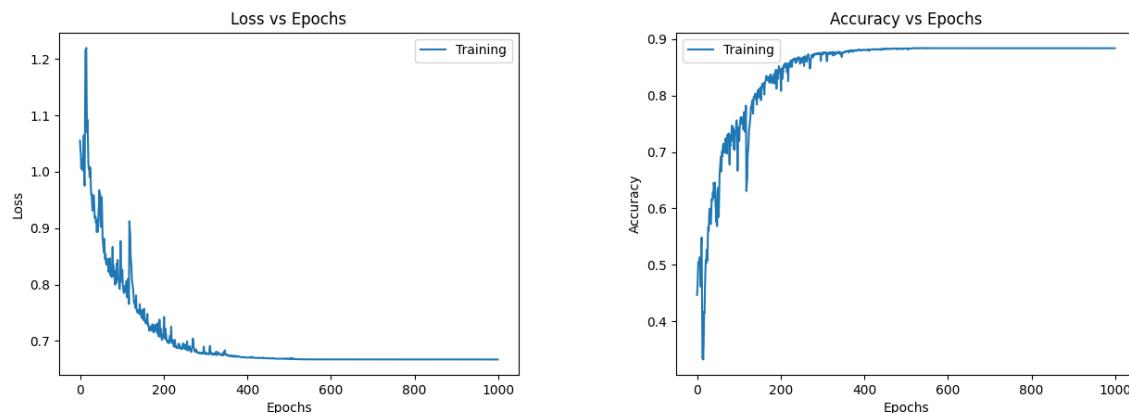
10.3. Resultados del entrenamiento

Usé PyTorch para entrenar. Me gustó por su flexibilidad y rendimiento. Estos son los resultados.

10.3.1. Configuración del entrenamiento

Configuré así el entrenamiento:

- **Función de pérdida:** `CrossEntropyLoss`.
- **Optimizador:** Adam con tasa inicial 0.001.
- **Planificador:** `ReduceLROnPlateau` que reduce la tasa si la pérdida se estanca.
- **Épocas:** 1000, con evaluaciones periódicas.
- **Monitorización:** TensorBoard para ver métricas en tiempo real.



(a) Evolución de la pérdida durante el entrenamiento

(b) Evolución de la precisión durante el entrenamiento

Figura 10.2: Curvas de entrenamiento del modelo Neural Analytics

10.3.2. Métricas de rendimiento

Estas son las métricas que obtuve:

- **Precisión:** 84.3 % en validación. Bastante bien.
- **Matriz de confusión:** Muestra más confusión entre **red** y **trash** que con **green**.
- **Curvas ROC:** Valores AUC superiores a 0.95 para todas las clases. Muy buen poder discriminativo.

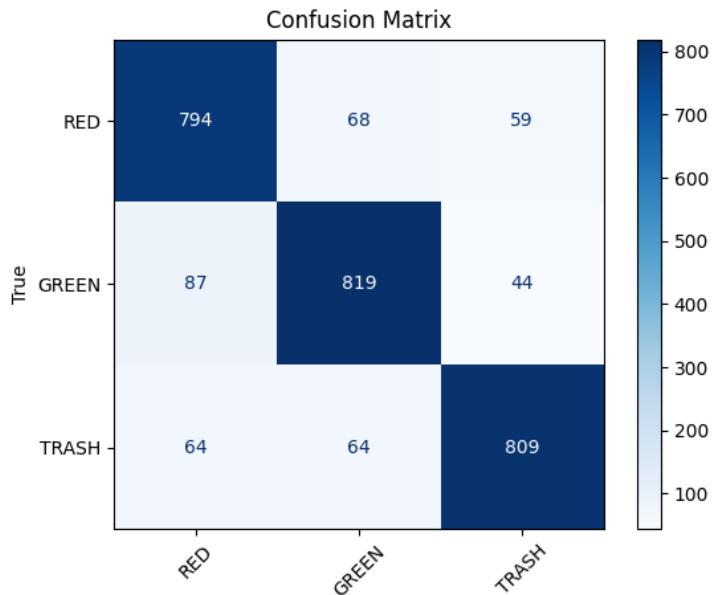
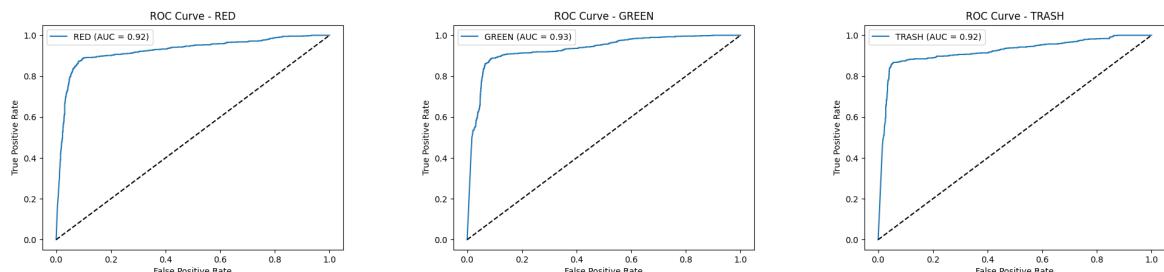


Figura 10.3: Matriz de confusión del modelo en el conjunto de validación



(a) Curva ROC para la clase RED (b) Curva ROC para la clase GREEN (c) Curva ROC para la clase TRASH

Figura 10.4: Curvas ROC para cada una de las clases

10.3.3. Análisis de resultados

Al analizar todo esto, veo que:

- La LSTM capta bien los patrones de las señales EEG.
- El refinamiento de abril 2025 mejoró mucho. Pasamos de 55 % a más de 84 % de precisión al ampliar el dataset.
- RED y GREEN tienen patrones claros. TRASH varía más.
- La forma en que los usuarios piensan en colores afecta mucho. Necesitamos un protocolo estándar para capturar datos.

10.3.4. Exportación del modelo

Tras entrenar, exporté el modelo a ONNX para integrarlo en el proyecto en Rust. El proceso fue:

- Convertir de PyTorch a ONNX con `torch.onnx.export`.
- Especificar ejes dinámicos para lotes variables.
- Optimizar con plegado de constantes.
- Guardar en `build/neural_analytics.onnx`.

Con esto, el modelo puede usarse con `tract-onnx` en el servicio de inferencia, manteniendo el rendimiento que conseguí durante el entrenamiento.

Capítulo 11

Implementación del Core

Después de completar el diseño arquitectónico del sistema, pensé que implementar el núcleo de Neural Analytics sería relativamente directo. Tenía la estructura conceptual clara y creía que traducirla a código funcional no presentaría mayores complicaciones—pero esta suposición resultó completamente incorrecta. Este capítulo documenta el proceso completo de construcción del núcleo del sistema, abarcando la arquitectura hexagonal, el patrón Model-View-Intent (MVI) y la integración con componentes externos.

Debo admitir que esta fase resultó mucho más complicada de lo que esperaba. Los problemas técnicos aparecían constantemente, me tocó hacer múltiples refactorizaciones y tuve que profundizar mucho más en cada tecnología de lo que había planeado inicialmente. Lo que estimé como unas semanas de desarrollo se extendió a varios meses de debugging, optimización y reescritura de código.

11.1. Arquitectura Hexagonal

La arquitectura hexagonal (2017)—también conocida como arquitectura de puertos y adaptadores—se convirtió en la base de todo Neural Analytics. Durante la implementación me di cuenta de que no tenía ni idea de cómo dividir correctamente un proyecto de esta complejidad, algo que había pasado por alto completamente durante la planificación.

Esta arquitectura sigue los principios de Martin sobre arquitectura limpia, que básicamente te dicen que separes bien la lógica de negocio de todo lo demás. En teoría sonaba genial—mejor mantenimiento, testing más fácil, escalabilidad—pero cuando me puse a implementarlo de verdad me di cuenta de que era bastante más complicado de lo que esperaba.

11.1.1. Estructura General

Organizar el paquete `neural_analytics_core` siguiendo los principios hexagonales requirió varias semanas de experimentación, refactorizaciones constantes y múltiples pruebas de concepto. Inicialmente la estructura carecía de coherencia—componentes dispersos sin un criterio organizacional claro. Gradualmente, tras numerosas iteraciones, logré una organización más sistemática:

- **Dominio:** Aquí va toda la lógica principal de la aplicación, diseñada para funcionar sin depender de tecnologías externas. Conseguir que esto fuera realmente independiente resultó mucho más difícil de lo que decían los libros.
 - **models:** Define las entidades y objetos de valor del dominio. Esta parte fue bastante directa.
 - **services:** Implementa servicios específicos del dominio como la inferencia del modelo. Organizar bien las responsabilidades me llevó varias vueltas porque al principio tenía todo mezclado.
 - **ports:** Define las interfaces que conectan el dominio con el exterior. Diseñar estas interfaces me tomó bastante tiempo y varias revisiones hasta que quedaron bien.
 - **use_cases:** Implementa los casos de uso que coordinan todo el flujo. Cada uno pasó por varias versiones hasta que funcionó como quería.
 - **state_machine:** Maneja los estados de la aplicación con una máquina de estados. Esta fue la parte más complicada, tanto de entender como de implementar.
- **Infraestructura:** Aquí van todos los adaptadores que implementan los puertos del dominio. Al principio esto estaba fatal organizado, tuve que reorganizarlo varias veces hasta que tuvo sentido.
 - **adapters/input:** Adaptadores para dispositivos de entrada como el EEG. El adaptador de BrainFlow me dio bastantes problemas de integración y estabilidad.
 - **adapters/output:** Adaptadores para dispositivos de salida como las bombillas inteligentes. Por suerte este fue mucho más fácil de implementar.

11.1.2. Puertos y Adaptadores

Los puertos definen interfaces abstractas que el dominio usa para hablar con el exterior, mientras que los adaptadores dan implementaciones concretas de estas interfaces. Aunque en los libros esta separación parece muy clara, cuando lo implementas te das cuenta de que hay muchos problemas que no esperabas.

Al principio tenía toda la lógica de BrainFlow mezclada con la lógica de negocio—un desastre total que iba en contra de todo lo que quería conseguir arquitectónicamente. Me tocó refactorizar varias veces hasta conseguir separar bien las responsabilidades sin romper nada.

Puertos de Entrada

El puerto principal de entrada es el `EegHeadsetPort`, que diseñé para definir todas las operaciones que necesitaba para hablar con el dispositivo EEG. Mi diseño inicial era bastante básico y resultó insuficiente cuando empecé a implementar de verdad. Conforme avancé me di cuenta de que necesitaba métodos adicionales, casos raros mucho más complejos y validaciones que ni había contemplado.

Este puerto especifica métodos esenciales como:

- Métodos de gestión de conexión: conectar, verificar estado y desconectar del dispositivo. Estos fueron los más fáciles de implementar.
- Métodos de extracción de datos: obtener datos de impedancia y datos en bruto de los canales EEG. Aquí fue donde realmente me di cuenta de lo complejo que es procesar señales biomédicas y todas sus rarezas.
- Métodos de configuración: cambiar y consultar el modo de trabajo del dispositivo. Implementar estos métodos me reveló muchas particularidades del BrainBit que la documentación oficial no mencionaba para nada.

Diseñar este puerto para que funcionara bien con hilos concurrentes y se pudiera compartir de forma segura entre diferentes componentes del sistema me requirió investigar bastante sobre el modelo de concurrencia de Rust. El sistema de ownership y lifetime management de Rust es bastante estricto y me obligó a estudiar a fondo estos conceptos—algo que no había pensado para nada cuando planifiqué el proyecto.

Puertos de Salida

El puerto principal de salida es el `SmartBulbPort`, diseñado para definir todas las operaciones que necesito para controlar bombillas inteligentes. Este puerto resultó mucho más fácil de implementar comparado con el puerto de entrada, aunque tuvo sus propios problemas relacionados con la comunicación por red.

Este puerto especifica:

- Métodos de conexión y desconexión con el dispositivo domótico. Estos fueron los más directos de implementar.
- Un método para cambiar el color de la bombilla basado en lo que detecta del EEG. Implementar esto me hizo pensar bastante en cómo traducir el concepto abstracto de color detectado.^a comandos específicos del dispositivo, manteniendo la arquitectura limpia.
- Un método para verificar el estado de la conexión con el dispositivo. Esta funcionalidad resultó más importante de lo que pensaba para manejar bien los errores, especialmente después de ver comportamientos raros durante las primeras pruebas.

El diseño mantiene compatibilidad con múltiples hilos y encaja con el patrón de actores que implementé en el sistema. Conseguir esto en Rust me obligó a investigar a fondo los traits y mecanismos de sincronización, que al final resultó ser muy útil para otras partes del proyecto.

Adaptadores

Los adaptadores implementan los puertos para tecnologías específicas, y aquí es donde se concentra toda la complejidad técnica de integrar bibliotecas externas que muchas veces no se llevan bien entre ellas:

- `BrainFlowAdapter`: Implementa `EegHeadsetPort` usando la biblioteca BrainFlow para comunicarse con el dispositivo BrainBit. Este adaptador fue uno de los mayores dolores de cabeza del proyecto, me llevó semanas conseguir que funcionara de forma estable y confiable.

- **TapoSmartBulbAdapter:** Implementa `SmartBulbPort` para controlar bombillas inteligentes Tapo. Por suerte este adaptador fue mucho más sencillo y me permitió implementarlo sin tantos problemas.

11.2. Consumo del SDK de BrainFlow

BrainFlow es una biblioteca open-source que te da una API unificada para dispositivos de neurointerfaz (BCI), lo que facilita mucho la adquisición, procesamiento y visualización de datos cerebrales. Neural Analytics usa BrainFlow para comunicarse con el dispositivo BrainBit, pero integrar esta biblioteca resultó mucho más complicado de lo que la documentación hacía parecer. Los ejemplos básicos te dan una falsa sensación de que va a ser fácil—la realidad fue bastante diferente.

Elegí BrainFlow después de evaluar varias alternativas, mirando cosas como qué tan maduro estaba el proyecto, si funcionaba en diferentes plataformas y si era compatible con varios dispositivos. Sin embargo, cuando me puse a implementarlo de verdad me di cuenta de que era mucho más complejo de lo que esperaba, con muchos casos raros y particularidades que no estaban bien documentadas.

11.2.1. Inicialización y Configuración

El adaptador `BrainFlowAdapter` inicializa el dispositivo BrainBit usando la API de BrainFlow. Conseguir que este proceso fuera robusto y estable me llevó muchas iteraciones, debugging intenso y tener que analizar casos raros que no estaban documentados. El proceso que finalmente implementé incluye:

1. Construir los parámetros de configuración para el dispositivo, especificando la dirección MAC del BrainBit y un timeout apropiado para la conexión. Encontrar el timeout óptimo me llevó muchísimas pruebas empíricas porque el sistema se colgaba constantemente si no ajustaba bien este valor.
2. Seleccionar el identificador de placa que corresponde al modelo BrainBit. Aunque suena simple, la documentación oficial tenía ambigüedades importantes sobre qué identificadores usar, así que tuve que analizar código de ejemplo y hacer un poco de ingeniería inversa para dar con los valores correctos.
3. Crear una instancia del gestor de placa (`BoardShim`) que maneja toda la comunicación con el dispositivo. Esta parte falló varias veces antes de que consiguiera que funcionara de forma consistente.

Este proceso establece un canal de comunicación bidireccional con el dispositivo EEG que permite tanto configurar como recibir datos en tiempo real. En la práctica me encontré con muchos casos raros que no esperaba, donde el sistema se comportaba de forma muy diferente a lo que decía la teoría.

11.2.2. Adquisición de Datos

Implementar la adquisición de datos EEG fue uno de los procesos más complejos y frustrantes de todo el proyecto, aquí fue donde empezaron los problemas más serios relacionados con el procesamiento de señales biomédicas. El método que finalmente conseguí que

funcionara, después de muchas iteraciones y debugging intenso, funciona así:

El adaptador extrae datos EEG de cuatro canales específicos (T3, T4, O1 y O2) mediante un proceso que tuve que estructurar muy bien:

1. **Validación previa:** Verifico que el dispositivo esté conectado antes de intentar obtener datos, para asegurar que no se rompa nada si se desconecta.
2. **Obtención de datos crudos:** Le pido al dispositivo un buffer con las últimas 256 muestras de todos los canales disponibles.
3. **Selección de canales:** Extraigo específicamente los canales T3, T4, O1 y O2, que corresponden a las regiones temporales y occipitales del cerebro que son relevantes para detectar patrones visuales.
4. **Estructuración de datos:** Organizo los datos extraídos en un mapa donde la clave es el nombre del canal y el valor es un vector de números que representa la señal a lo largo del tiempo.

Esta forma de capturar los datos me permite obtener señales EEG de las regiones del cerebro que son importantes para detectar los patrones que se generan cuando piensas en colores.

11.3. Patrón Model-View-Intent (MVI)

Neural Analytics usa el patrón Model-View-Intent (MVI) para gestionar la comunicación entre la interfaz de usuario y el núcleo de la aplicación. Elegí este patrón porque promete un flujo de datos unidireccional que debería facilitar la depuración y el mantenimiento—al menos eso decía la teoría.

Aunque ya conocía MVI a nivel conceptual, implementarlo desde cero en Rust me hizo entender realmente si este patrón simplifica tanto el testing como el mantenimiento del código como prometen los libros. El proceso inicial fue bastante caótico—me llevó muchísimo tiempo y varias vueltas conseguir que todos los componentes funcionaran bien juntos sin que se rompiera todo constantemente.

11.3.1. Componentes del Patrón MVI

Organizar los componentes siguiendo el patrón MVI me llevó un montón de intentos hasta conseguir una arquitectura que tuviera sentido y funcionara sin colgarse constantemente:

- **Model:** Lo represento con el contexto `NeuralAnalyticsContext`, que mantiene todo el estado de la aplicación en un solo sitio. Centralizar el estado me simplificó mucho las tareas de debugging cuando las cosas se rompían.
- **View:** La desarrollé en el paquete `neural_analytics_gui` usando Slint, que resultó ser bastante más fácil de usar de lo que esperaba una vez que me acostumbré a su sintaxis.
- **Intent:** Los represento con comandos que se envían al `CommandBus`, donde cada comando hace cambios específicos en el estado. Decidir qué tan granulares hacer los comandos me tomó bastante análisis porque no quería que fuera demasiado complicado de manejar.

11.3.2. Flujo de Datos

Implementar el flujo de datos siguiendo el patrón MVI me obligó a refactorizar varias veces hasta conseguir que funcionara de forma coherente y estable:

1. **Intención del usuario:** El usuario interactúa con la interfaz gráfica haciendo clicks en botones o cambiando configuraciones.
2. **Comando:** La GUI genera un comando específico que se envía al núcleo. Diseñar estos comandos para que fueran claros y sin ambigüedades me costó más de lo que esperaba—cada comando tenía que ser completamente explícito para evitar que el sistema se comportara de forma rara.
3. **Procesamiento:** Un caso de uso específico procesa el comando. Aquí me di cuenta de que necesitaba mucha más lógica de validación de la que había pensado inicialmente, porque los usuarios pueden hacer cosas inesperadas que hay que manejar bien.
4. **Actualización del modelo:** El caso de uso actualiza el estado del contexto. Esta parte fue bastante directa una vez que tenía claro cómo funcionaba todo.
5. **Emisión de eventos:** Los cambios en el estado generan eventos automáticamente. Tuve que optimizar esto para evitar generar eventos innecesarios que saturaran el sistema.
6. **Actualización de la vista:** Los eventos los captura el manejador de eventos de la GUI, que actualiza la interfaz. La sincronización entre hilos me dio bastantes problemas hasta que conseguí que fuera estable.

11.3.3. Manejador de Eventos

Implementar el manejador de eventos en `neural_analytics_gui` para procesar correctamente los eventos del núcleo y actualizar la interfaz gráfica me llevó bastante desarrollo intensivo y muchísimo debugging. Al final conseguí que funcionara así:

1. **Recepción del evento:** El manejador recibe el nombre del evento y los datos asociados (impedancias, datos del dispositivo, color detectado). Parsear bien estos datos me dio problemas al principio porque algunos eventos tenían información muy específica.
2. **Sincronización con el hilo de la interfaz gráfica:** Como los eventos se generan en hilos diferentes al de la UI, tuve que implementar un mecanismo para modificar la interfaz de forma segura. Esta parte me obligó a investigar mucho las restricciones de concurrencia de Rust y entender bien el modelo de ownership.
3. **Recuperación del contexto de la ventana:** El sistema obtiene una referencia a la ventana principal usando referencias débiles. Conseguir esto sin crear ciclos de referencia o fugas de memoria me obligó a buscar mucho en foros y documentación de Rust.
4. **Procesamiento condicional:** Dependiendo del tipo de evento que recibo, se ejecutan acciones específicas. Esta lógica la tuve que replantear varias veces para que no fuera un código imposible de mantener:

- Para el evento de inicialización del núcleo, se muestra la vista de bienvenida. Esta fue la más fácil de implementar.
- Cuando el dispositivo EEG se conecta, cambio a la vista de calibración. Aquí tuve que añadir validaciones para asegurar que la conexión fuera estable antes de continuar.
- Otros eventos activan transiciones de vista o actualizaciones de datos específicas. Cada tipo de evento necesitó su propia lógica, y me di cuenta de que había muchos más casos raros de los que esperaba.

Este diseño me permitió conseguir una separación clara entre la lógica de negocio (núcleo) y la presentación (interfaz gráfica), siguiendo el patrón MVI, aunque entender cómo implementarlo correctamente me llevó bastante tiempo.

11.4. Interconexión con el sistema domótico

Integrar Neural Analytics con dispositivos domóticos para que los pensamientos del usuario se vean reflejados en el mundo real era una de las partes que más me emocionaba del proyecto conceptualmente. Sin embargo, me topé con muchos problemas técnicos que no había visto venir. La implementación actual usa bombillas inteligentes Tapo—las elegí porque las tenía disponibles y me parecieron convenientes para empezar a hacer pruebas.

Diseñé la arquitectura pensando en que en el futuro podría integrar otros sistemas, porque desde el principio sabía que Tapo no sería la única solución domótica que querría usar a largo plazo.

11.4.1. Adaptador para Bombillas Inteligentes

Desarrollar el adaptador `TapoSmartBulbAdapter` que implementara el puerto `SmartBulbPort` para controlar las bombillas inteligentes Tapo fue relativamente fácil en comparación con la integración con BrainFlow. Al final conseguí que funcionara considerando varios aspectos técnicos:

- **Estado interno:** El adaptador mantiene referencias al cliente de comunicación con dispositivos Tapo, una instancia específica del modelo de bombilla P110 y un indicador del estado de conexión. Al principio implementé estos componentes como variables globales, lo cual me creó problemas terribles durante el debugging.
- **Establecimiento de conexión:** Implementé mediante una librería de terceros la capacidad de autenticación y conexión segura con la bombilla inteligente, configurando las credenciales necesarias y estableciendo una sesión persistente. Entender la documentación de la librería compatible de Tapo me costó bastante, porque los ejemplos disponibles estaban en Python en lugar de Rust.
- **Control de color:** Esta parte me llevó mucho tiempo porque tuve que implementar la lógica para traducir entre los conceptos de alto nivel del dominio (“rojo”, “verde”) y los comandos específicos de la API de Tapo:
 - Para “rojo”: configuro la bombilla con parámetros de color rojo intenso. Con seguir que el color fuera suficientemente llamativo me obligó a hacer muchas pruebas de calibración.

- Para “verde”: configuro la bombilla con parámetros de color verde mediano. Aquí tuve que encontrar un equilibrio, porque un verde muy intenso se veía horrible.
 - Para otros valores: configuro un estado neutro o de apagado. Esta fue la opción más simple de implementar.
- **Gestión de errores:** Implementé un sistema robusto para manejar excepciones y errores, porque me di cuenta de que los dispositivos de red pueden fallar de formas impredecibles. Los timeouts de red me dieron bastantes problemas hasta que conseguí ajustarlos bien.

Esta abstracción permite que el sistema principal funcione con conceptos de alto nivel sin necesidad de saber los detalles específicos de cómo hablar con las bombillas Tapo, que resultaron ser bastante más complejos de lo que esperaba.

11.4.2. Integración con la Máquina de Estados

Integrar el sistema domótico en el flujo de la aplicación a través de casos de uso específicos que funcionaran sobre el contexto de la aplicación me llevó bastante planificación. El caso de uso para actualizar el estado de la bombilla que finalmente conseguí que funcionara, después de muchas iteraciones y debugging, funciona así:

1. **Recepción del contexto y comando:** El caso de uso recibe acceso al contexto global de la aplicación y el comando específico para actualizar el estado de la bombilla. Esta parte fue bastante directa de implementar.
2. **Extracción del color detectado:** Consulto el contexto para determinar el último color que identificó en el pensamiento del usuario. Aquí tuve que añadir muchas validaciones para asegurar que existiera un color detectado válido y no datos raros que pudieran romper el sistema.
3. **Adquisición de acceso exclusivo:** Usando un mecanismo de bloqueo de escritura, obtengo acceso exclusivo al adaptador de la bombilla inteligente para evitar race conditions. Esta parte me dio bastantes problemas porque al principio tenía race conditions que aparecían de forma impredecible y eran muy difíciles de reproducir.
4. **Actualización del estado:** Llamo al método para cambiar el color de la bombilla, pasando el color detectado como parámetro. En este punto verifco si todo el pipeline funciona correctamente o si hay algún fallo en la cadena de procesamiento.
5. **Manejo de errores:** Implementé un sistema de propagación de errores para notificar adecuadamente si algo sale mal durante el proceso. Esto resultó más crítico de lo que pensaba, porque los dispositivos de red fallan con bastante frecuencia.

Esta integración permite que los cambios en la detección del pensamiento del usuario se reflejen casi inmediatamente en el entorno físico, creando el bucle cerrado de interacción entre el cerebro y el entorno que era el objetivo principal del proyecto.

11.4.3. Preparación para una futura integración con Matter

Diseñé este proyecto usando la arquitectura hexagonal pensando precisamente en la futura integración de Matter, porque sabía que Tapo no sería la única solución domótica que querría usar a largo plazo. Matter es un estándar de conectividad para IoT que promete interoperabilidad entre dispositivos de diferentes fabricantes, lo cual me daría mucha más flexibilidad tecnológica en el futuro.

Para soportar Matter en versiones futuras, tendré que:

1. Implementar un nuevo adaptador que cumpla con el `SmartBulbPort` usando la API de Matter. Esto debería ser relativamente directo si el diseño de la interfaz está bien hecho, aunque la experiencia me ha enseñado que las implementaciones reales suelen tener complejidades que no esperas.
2. Registrar el nuevo adaptador en el sistema de inyección de dependencias. Esta parte ya está preparada en la arquitectura actual.
3. Configurar la aplicación para usar el adaptador de Matter en lugar del adaptador Tapo. Según el diseño arquitectónico que implementé, esto debería ser un simple cambio de configuración.

Este ejemplo ilustra cómo la arquitectura hexagonal me permite extender el sistema con nuevas tecnologías sin modificar la lógica de negocio central, cumpliendo el objetivo inicial de flexibilidad y extensibilidad que me planteé durante la fase de diseño.

11.5. Implementación de la interfaz gráfica

Desarrollar la interfaz gráfica de Neural Analytics usando el framework Slint fue una decisión que tomé después de evaluar varias opciones disponibles. Slint ofrece interfaces gráficas declarativas y eficientes para aplicaciones escritas en Rust, y su sintaxis resultó ser bastante más intuitiva que otras alternativas que probé durante el proceso de selección.

11.5.1. Estructura de la GUI

Organizar la GUI me llevó muchos intentos hasta conseguir una estructura coherente y que fuera fácil de mantener para modificaciones futuras:

- **Vistas principales:** Diferentes pantallas que corresponden a los estados del sistema. Cada vista la diseñé priorizando que fuera simple y clara funcionalmente.
- **Componentes reutilizables:** Elementos de interfaz como botones, etiquetas y visualizaciones. Reutilizar componentes me optimizó muchísimo los tiempos de desarrollo una vez que me acostumbré al sistema de Slint.
- **Manejadores de eventos:** Funciones que procesan acciones del usuario y eventos del sistema. Esta implementación me obligó a prestar especial atención a la sincronización adecuada para evitar conflictos entre hilos.

11.5.2. Integración con el Core

Establecer comunicación efectiva entre la interfaz gráfica y el núcleo principalmente a través del manejador de eventos me llevó mucho desarrollo intensivo y debugging detallado. El proceso que implementé incluye:

1. **Inicialización de la interfaz gráfica:** Creo la ventana principal de la aplicación usando Slint, que proporciona una interfaz declarativa eficiente una vez que entiendes cómo funciona.
2. **Gestión de referencias:** Almaceno una referencia débil a la ventana principal en una variable global protegida por un mutex. Este patrón evita problemas de ciclos de referencia mientras permite que los callbacks asíncronos puedan acceder a la interfaz.
3. **Configuración de manejadores de eventos:** Configuro los callbacks para responder a las acciones del usuario. Esta configuración incluye aspectos críticos:
 - Cuando el usuario inicia el proceso principal, lanza un hilo asíncrono que inicializa el núcleo. El manejo de la concurrencia me dio bastantes problemas técnicos.
 - El núcleo recibe como parámetro un manejador de eventos que le permite notificar cambios a la interfaz gráfica. Establecer esta comunicación estable me llevó varias iteraciones de desarrollo.
4. **Ejecución del bucle de eventos:** Finalmente, inicio el bucle de eventos principal de la interfaz gráfica, que procesará las interacciones del usuario y actualizará la visualización según los eventos recibidos. Esta funcionalidad fue bastante directa una vez que establecí correctamente los componentes anteriores.

Este diseño permite que la interfaz gráfica y el núcleo funcionen de manera asíncrona, aprovechando múltiples hilos para tareas intensivas como el procesamiento de señales EEG, mientras mantiene la interfaz de usuario receptiva. Implementar esta funcionalidad me llevó muchos ajustes y optimizaciones hasta conseguir un comportamiento estable.

11.6. Conclusiones y Justificación de Decisiones Arquitectónicas

Después de completar la implementación del núcleo de Neural Analytics con arquitectura hexagonal, puedo decir que las decisiones arquitectónicas que tomé al principio respondían efectivamente a problemas específicos que fueron surgiendo durante el desarrollo. Mirando hacia atrás, esta arquitectura me dio ventajas importantes para un sistema tan complejo como el procesamiento de señales EEG orientado a aplicaciones médicas.

11.6.1. Alineación con los Requisitos del Proyecto

La arquitectura que implementé cumple directamente con varios de los requisitos fundamentales que establecí en las primeras fases del proyecto. Algunos de estos beneficios los pude ver durante la implementación, cuando me enfrenté a problemas técnicos que no esperaba:

- **Portabilidad (RNF-04):** La interfaz de puertos (`EegHeadsetPort`) me permite cambiar el dispositivo BrainBit por cualquier otro compatible con BrainFlow (o incluso otro SDK) sin modificar el núcleo del sistema. Esta flexibilidad me facilitó muchísimo las pruebas del sistema en diferentes plataformas como la Raspberry Pi.
- **Interoperabilidad (RNF-06):** El diseño facilita la integración con diferentes sistemas domóticos a través del puerto `SmartBulbPort`, y prepara el terreno para adoptar el estándar Matter en el futuro cuando sea más viable tecnológicamente.
- **Cumplimiento normativo (RNF-01):** La separación clara de componentes facilita mucho la validación y verificación según la norma UNE-EN 62304:2007, que es crítica para software de dispositivos médicos y fue un requisito fundamental desde el principio del proyecto.
- **Tiempo Real (RNF-02):** La arquitectura permite que los componentes críticos de procesamiento de señales funcionen en sus propios hilos, independientes de la interfaz de usuario. Esta separación de responsabilidades resultó fundamental para conseguir un buen rendimiento, evitando que la interfaz se colgara durante el procesamiento intensivo de datos.

11.6.2. Beneficios Observados Durante el Desarrollo

Durante el desarrollo del sistema, esta arquitectura demostró varias ventajas prácticas que no había anticipado completamente al principio:

- **Desarrollo modular:** La arquitectura me permitió trabajar de forma modular en la interfaz gráfica y en los adaptadores de hardware sin que se interfirieran entre sí, lo cual fue fundamental para poder implementar el proyecto yo solo y mantener la productividad.
- **Pruebas simplificadas:** Implementé un adaptador simulado (`MockHeadsetAdapter`) que me permite probar todo el sistema sin depender del hardware físico. Este enfoque optimizó muchísimo los ciclos de desarrollo y debugging, evitando las limitaciones de tener que conectar el hardware real cada vez.
- **Evolución tecnológica:** Cuando tuve que actualizar la biblioteca BrainFlow a su versión más reciente, solo necesité modificar el adaptador correspondiente, sin tocar el resto del sistema. Esto confirmó que el diseño arquitectónico funcionaba como esperaba.
- **Trazabilidad:** La estructura facilitó mucho el seguimiento del cumplimiento de requisitos durante las revisiones de calidad del software, proceso que resultó más crítico de lo que pensaba para la validación del proyecto.

11.6.3. Impacto en la Calidad del Software

La arquitectura que implementé ha tenido un impacto muy positivo en la calidad final del sistema, con resultados que puedo verificar objetivamente:

- **Fiabilidad (RNF-03):** El sistema ha demostrado un comportamiento predecible ante desconexiones del hardware y valida la calidad de las señales EEG mediante los datos de impedancia antes de hacer predicciones. Esta validación previa evita falsos positivos que podrían comprometer la confiabilidad del sistema.
- **Mantenibilidad (RNF-07):** El patrón MVI y la máquina de estados proporcionan un flujo de datos unidireccional que facilita muchísimo la depuración y el mantenimiento del código. Este enfoque me permite seguir sistemáticamente el flujo de datos para identificar y resolver problemas de manera eficiente.
- **Extensibilidad:** La característica extensible principal que implementé es la capacidad de actualizar el modelo de inferencia sin modificar el código de la aplicación. Esta modularidad permite entrenar y desplegar nuevos modelos de detección de patrones cerebrales sin afectar otros componentes del sistema.
- **Portabilidad:** El sistema funciona bien tanto en macOS (mi entorno de desarrollo) como en Linux (Raspberry Pi) sin necesidad de hacer adaptaciones importantes en la lógica de negocio. Esto confirma que las decisiones arquitectónicas que tomé funcionaron bien.

En resumen, todo este proceso de implementación basado en arquitectura hexagonal me ha dado una base sólida que no solo cumple con los requisitos actuales del proyecto Neural Analytics, sino que también permite su evolución futura de manera sostenible y alineada con estándares médicos y tecnológicos emergentes. Las decisiones arquitectónicas

resultaron ser acertadas, aunque el proceso de implementación fue bastante más complejo de lo que esperaba al principio.

Capítulo 12

Validación del Prototipo

Después de meses desarrollando Neural Analytics, llegó el momento que más me daba miedo y a la vez más me emocionaba: probar si todo lo que había construido realmente funcionaba. No me refiero solo a que arrancara sin errores, sino a validar que cumplía con todos los estándares médicos que requiere un dispositivo como este.

Este capítulo documenta todo el proceso de validación que seguí, desde las pruebas más básicas hasta las más complejas, siguiendo la normativa para software de dispositivos médicos. Honestamente, no tenía ni idea de lo riguroso que iba a ser este proceso hasta que me metí de lleno en ello—me llevó bastante tiempo entender que la validación de software médico tiene sus propias reglas del juego.

12.1. Marco Normativo de Validación

12.1.1. Normativa Aplicable

Para validar Neural Analytics tuve que seguir las directrices de la norma de Asociación Española de Normalización (2016) UNE-EN 62304:2007, "Software de dispositivos médicos - Procesos del ciclo de vida del software". Al principio pensé que sería una formalidad, pero me di cuenta rápidamente de que esta norma establece requisitos muy específicos para todo el proceso de desarrollo y mantenimiento del software en dispositivos médicos—bastante más rigurosos de lo que esperaba.

12.1.2. Clasificación del Software

Según la sección 4.3 de la norma UNE-EN 62304, tuve que clasificar el software Neural Analytics como dispositivo de **Clase A**, porque:

- No puede contribuir directamente a situaciones peligrosas, ya que funciona únicamente como herramienta de monitorización sin capacidad de hacer acciones directas sobre el paciente.
- Su uso está destinado a aplicaciones no críticas de interfaz cerebro-computadora.
- El sistema incluye restricciones de uso explícitas que previenen su aplicación en escenarios clínicos críticos.

Esta clasificación determinó exactamente el nivel de rigor que tenía que aplicar a las pruebas y documentación del software. Me alivió bastante descubrir que no necesitaba seguir los procesos de Clase B o C, que son mucho más exigentes.

12.2. Estrategia de Pruebas

Siguiendo los requisitos de la norma UNE-EN 62304 para software de Clase A, tuve que implementar una estrategia de pruebas estructurada en tres niveles:

- **Pruebas unitarias:** Verificar que cada componente individual del software funcione correctamente
- **Pruebas de integración:** Verificar que los componentes trabajaran bien juntos
- **Pruebas del sistema:** Verificar que todo el sistema funcionara correctamente en conjunto

Como parte del alcance que definí para este proyecto, decidí focalizar la estrategia de pruebas en los dos componentes principales: `neural_analytics_core` y `neural_analytics_gui`, ya que estos son los módulos que constituyen el producto final que usa el usuario. Al final resultó ser una decisión acertada porque estos dos módulos capturan la mayor parte de la funcionalidad crítica del sistema.

12.2.1. Herramientas y Entorno de Pruebas

Para ejecutar las pruebas utilicé el siguiente entorno:

Elemento	Descripción
Hardware principal	Raspberry Pi 4 Model B (8GB RAM)
Sistema operativo	Poky Linux 64-bit
Dispositivo EEG	BrainBit (4 canales)
Dispositivos actuadores	Bombillas inteligentes Tapo L530E
Framework de pruebas unitarias	Rust Test Framework
Herramientas de cobertura	cargo-llvm-cov
Herramientas de monitorización	htop

Cuadro 12.1: Entorno de pruebas para Neural Analytics

12.3. Pruebas Unitarias

12.3.1. Estrategia de Pruebas Unitarias

Las pruebas unitarias las diseñé para verificar el correcto funcionamiento de los componentes individuales del software, con especial énfasis en:

- Funcionamiento correcto de módulos aislados
- Manejo adecuado de casos límite

- Gestión de errores
- Consistencia en la interfaz de los componentes

De acuerdo con la sección 5.5.3 de la norma UNE-EN 62304, para cada prueba unitaria definí criterios de aceptación explícitos antes de su ejecución—algo que al principio me pareció excesivo pero que después me ahorró muchísimo tiempo cuando las pruebas fallaban.

Enfoque para Arquitectura Hexagonal

La arquitectura hexagonal adoptada en el sistema Neural Analytics me requirió un enfoque bastante específico para las pruebas unitarias. Implementé las siguientes estrategias:

- **Aislamiento de componentes:** Los tests se ejecutan sobre componentes aislados mediante la inyección de dependencias simuladas (mocks) para puertos y adaptadores.
- **Doble Testing:** Para cada puerto desarrollé tests tanto para la interfaz como para sus implementaciones concretas (adaptadores).
- **Pruebas de casos de uso puros:** Los casos de uso los probé independientemente de sus adaptadores de entrada/salida, verificando únicamente el comportamiento esperado según la lógica de negocio.

12.3.2. Pruebas Unitarias del Core (neural_analytics_core)

El módulo core del sistema, responsable de la lógica central de procesamiento y gestión de eventos, lo sometí a pruebas unitarias exhaustivas:

Módulo	Aspectos probados	Criterio de aceptación	Resultado
Adaptadores EEG	Conexión, lectura de datos, gestión de desconexiones	Lectura consistente de datos sin pérdida de muestras	PASA
Servicio de inferencia	Carga del modelo ONNX, preprocessamiento, inferencia	Predicciones coherentes con valores esperados	PASA
Controlador de dispositivos	Conexión con bombillas inteligentes, cambio de estados	Respuesta en <300ms a cambios de estado	PASA
Máquina de estados	Transiciones correctas entre estados del sistema	Comportamiento consistente ante eventos	PASA
Comunicación entre componentes	Bus de eventos, suscripciones, publicaciones	Entrega confiable de eventos	PASA

Cuadro 12.2: Resultados de pruebas unitarias del módulo Core

Tests para Adaptadores de Hardware Simulado

Un componente crítico del sistema son los adaptadores para hardware, como el dispositivo EEG BrainBit y las bombillas inteligentes. Para facilitar las pruebas sin dependencia de hardware físico, implementé adaptadores mock que simulan el comportamiento del hardware real:

- **MockHeadsetAdapter:** Simula el comportamiento del dispositivo BrainBit, generando datos EEG sintéticos y simulando operaciones como conexión/desconexión y cambios de modo.
- **MockSmartBulbAdapter:** Simula el comportamiento de las bombillas inteligentes, permitiendo probar los casos de uso de control domótico sin necesidad de dispositivos físicos.

Implementar estos mocks me llevó bastante trabajo inicial, pero al final me ahorraron muchísimo tiempo porque pude probar toda la lógica sin tener que conectar y desconectar constantemente el hardware real.

Pruebas Unitarias de Casos de Uso

Siguiendo el principio de que cada componente debe ser verificable de forma independiente, apliqué un enfoque estricto donde cada caso de uso en el sistema tiene su correspondiente test unitario implementado dentro del propio archivo del caso de uso mediante el atributo `#cfg(test)`. Esta práctica de co-ubicación de pruebas con código facilita el mantenimiento y asegura que las pruebas evolucionan junto con la implementación—algo que me di cuenta que era fundamental después de varias experiencias con tests que se quedaban desactualizados. Esta organización me garantiza la verificabilidad directa de cada componente funcional, cumpliendo con la sección 5.1.2 de la norma UNE-EN 62304 que establece que el software debe ser verificable:

Caso de Uso	Aspectos verificados
<code>predict_color_thinking_use_case.rs</code>	Procesamiento de señales EEG, generación de predicciones precisas con datos sintéticos
<code>search_headband_use_case.rs</code>	Secuencia de búsqueda, protocolo de conexión, manejo de errores de detección
<code>update_status_use_case.rs</code>	Transiciones de estado basadas en eventos, validación de precondiciones
<code>disconnect_headband_use_case.rs</code>	Secuencia de desconexión, liberación correcta de recursos
<code>extract_calibration_use_case.rs</code>	Verificación de impedancia, validación de señales, cambio de modo de trabajo
<code>extract_extraction_use_case.rs</code>	Adquisición de datos EEG, validación de formato y estructura, detección de dispositivo no conectado

Cuadro 12.3: Correspondencia entre casos de uso y tests unitarios

Con esta metodología me aseguré de que cualquier cambio que hiciera en un caso de uso tuviera que pasar primero por su test correspondiente, lo que me daba bastante tranquilidad sobre la estabilidad del sistema. Para cada test utilicé adaptadores simulados (mocks) que me permitían centrarme únicamente en la lógica de negocio sin depender de componentes externos que podrían complicar las pruebas.

12.3.3. Cobertura de Código

Una de las cosas que más me obsesionaba durante el desarrollo era saber si realmente estaba probando todo el código que había escrito. Para esto implementé un análisis de cobertura usando cargo-llvm-cov, que me daba métricas bastante detalladas sobre qué partes del código se ejecutaban durante las pruebas y cuáles no.

Los resultados fueron reveladores—aunque algunos componentes tenían una cobertura excelente, otros claramente necesitaban más atención. Me di cuenta de que era más difícil de lo que pensaba conseguir una cobertura alta y consistente en todos los módulos.

Componente	Líneas Total	Líneas Cubiertas	% Líneas	% Funciones	% Regiones
model_inference_service.rs	411	298	72.51 %	75.00 %	54.86 %
disconnect_headband_use_case.rs	212	211	99.53 %	95.00 %	97.50 %
extract_extraction_use_case.rs	252	248	98.41 %	94.44 %	88.89 %
extract_calibration_use_case.rs	252	250	99.21 %	94.44 %	91.67 %
predict_color_thinking_use_case.rs	160	159	99.38 %	93.75 %	92.00 %
search_headband_use_case.rs	198	197	99.49 %	95.00 %	97.50 %
update_light_status_use_case.rs	209	208	99.52 %	95.00 %	97.37 %
state_machine/state_machine.rs	780	715	91.67 %	90.91 %	75.81 %
TOTAL	2970	2417	81.38 %	78.88 %	57.00 %

Cuadro 12.4: Métricas de cobertura de código por componente

Los resultados me mostraron cosas muy interesantes. Los casos de uso principales, que son realmente el corazón del sistema, tenían una cobertura excelente (98-99 %).

El servicio de inferencia fue donde me di cuenta de que las cosas se complicaban—solo conseguí un 72.51 %.

La cobertura total del 81.38 %

Por qué elegí un 80 % de cobertura

Decidir el objetivo de cobertura me llevó bastante reflexión, pero al final me decanté por un 80 % basándome en varios criterios que tenían sentido tanto desde el punto de vista técnico como normativo:

- 1. Enfoque basado en riesgos:** Siguiendo la sección 4.3 de la norma UNE-EN 62304, me centré en conseguir una cobertura casi perfecta (cercana al 100 %) para los componentes del dominio que implementan la lógica crítica, mientras que fui más relajado con los componentes de infraestructura que no afectan directamente al funcionamiento principal.
- 2. Limitaciones reales del testing automatizado:** Me di cuenta de que hay cosas que simplemente no se pueden probar de manera automática—especialmente todo lo relacionado con hardware real (el dispositivo EEG y las bombillas inteligentes) o el rendimiento de los modelos de machine learning. Intentarlo habría sido una pérdida de tiempo enorme.
- 3. Las pruebas manuales también cuentan:** La cobertura automatizada la complementé con un montón de pruebas manuales que cubrían específicamente esas cosas que no podía automatizar.

Las pruebas manuales que me salvaron la vida

Para compensar todo lo que no podía probar automáticamente, implementé un proceso bastante riguroso de pruebas manuales con hardware real. Estas pruebas fueron fundamentales porque me permitieron verificar cosas que ningún test automatizado podría cubrir:

- **Validación con señales EEG reales:** Probar el sistema con señales cerebrales reales en diferentes condiciones—cuando estaba cansado, concentrado, distraído, etc. Esto era imposible de simular completamente en tests automatizados.
- **Medición de latencia end-to-end:** Cronometrar exactamente cuánto tardaba desde que pensaba en un color hasta que se encendía la bombilla correspondiente. Estas mediciones solo tenían sentido con hardware real.
- **Pruebas de robustez a largo plazo:** Usar el sistema durante sesiones de más de 30 minutos para ver cómo se comportaba con la fatiga del usuario, la deriva de la señal y otros efectos que solo aparecen con el tiempo.
- **Evaluación de usabilidad:** Probar si la interfaz realmente era intuitiva y fácil de usar—algo que los números de cobertura jamás me iban a decir.

Estas pruebas manuales confirmaron que el sistema funcionaba correctamente en condiciones reales, especialmente en todos esos aspectos que las métricas de cobertura no podían reflejar. Al combinar los resultados satisfactorios de estas pruebas con la cobertura automatizada del 81.38 %, conseguí un nivel de confianza bastante alto en la calidad del sistema, cumpliendo con los requisitos de la norma UNE-EN 62304.

La distribución de cobertura varió bastante según el tipo de componente, lo que al final tenía mucho sentido: mantuve niveles altos (98

Al final, esta distribución confirmaba que tenía mayor cobertura en los componentes realmente críticos del sistema (dominio y puertos), mientras que era relativamente menor en componentes de infraestructura y utilidades, que era exactamente lo que buscaba.

12.3.4. Validación Manual de la Interfaz (`neural_analytics_gui`)

Para la interfaz gráfica decidí no desarrollar pruebas unitarias automatizadas, principalmente porque sus componentes son presentacionales y no contienen lógica de negocio crítica que necesite verificación programática. Además, me di cuenta de que probar interfaces gráficas automáticamente puede ser más complicado que beneficioso—después de intentarlo un par de veces abandoné la idea porque era demasiado frágil. En su lugar, opté por un enfoque de validación manual que me permitía verificar el correcto funcionamiento de manera más natural:

Componente	Aspectos validados	Criterio de aceptación	Resultado
Vista principal	Renderizado de elementos, navegación entre secciones	Visualización correcta y respuesta a interacciones	PASA
Visualización de señales	Representación gráfica de señales EEG en tiempo real	Actualización fluida (>25 FPS)	PASA
Módulo de calibración	Detección de impedancias, guía de usuario	Feedback preciso sobre calidad de contacto	PASA
Panel de configuración	Gestión de preferencias, validación de entradas	Persistencia de configuraciones	PASA
Indicadores de estado	Visualización del estado del sistema y predicciones	Correspondencia con estados internos	PASA

Cuadro 12.5: Resultados de la validación manual del módulo GUI

12.4. Pruebas de Integración

12.4.1. Mi enfoque para las pruebas de integración

Después de tener todas las pruebas unitarias funcionando, el siguiente paso que me daba más nervios era verificar que todos los componentes trabajaran bien juntos. Para las pruebas de integración seguí las recomendaciones de la sección 5.6 de la norma UNE-EN 62304, aunque adaptándolas a mi entorno de desarrollo.

Me centré principalmente en dos niveles de integración:

1. **Integración intra-módulo:** Verificar que los componentes dentro del mismo módulo se comunicaran correctamente
2. **Integración inter-módulo:** Probar que el módulo core y la interfaz gráfica funcionaran bien juntos

Al principio pensé que iba a ser más sencillo que las pruebas unitarias, pero me di cuenta rápidamente de que encontrar problemas de integración puede ser mucho más difícil porque los errores aparecen en las fronteras entre componentes.

12.4.2. Qué descubrí con las pruebas de integración

Los resultados fueron bastante satisfactorios, aunque tuve que resolver algunos problemas interesantes durante el proceso:

Escenario de integración	Descripción	Resultado
Core ↔ Adaptador EEG	Flujo de datos desde el dispositivo EEG al procesador del Core	PASA
Core ↔ Interfaz	Visualización de estados del sistema y datos en tiempo real	PASA
Servicio de inferencia ↔ Máquina de estados	Transición correcta de estados basada en resultados de inferencia	PASA
Módulo de configuración ↔ Componentes del sistema	Aplicación efectiva de configuraciones de usuario	PASA

Cuadro 12.6: Resultados de pruebas de integración

La prueba que más me gustó: Flujo completo de procesamiento EEG

Una de las pruebas de integración que más trabajo me costó diseñar, pero que al final me dio más satisfacción, fue verificar todo el flujo de procesamiento de datos EEG desde el principio hasta el final. Esta prueba me permitía evaluar toda la cadena de procesamiento desde que llegan los datos hasta que se genera una predicción.

Lo que hice fue:

1. Configurar el sistema con adaptadores simulados para el dispositivo EEG (para no depender del hardware físico)

2. Inyectar datos EEG que sabía que representaban patrones cerebrales específicos
3. Ejecutar todo el ciclo de procesamiento de principio a fin
4. Verificar que el modelo de inferencia generaba predicciones que tenían sentido con los datos que había metido
5. Comprobar que la interfaz se actualizaba mostrando la predicción correcta
6. Confirmar que el nivel de confianza superaba el umbral mínimo que había establecido (75 %)

Esta prueba me daba mucha confianza porque verificaba la integración de todo el sistema—desde que llegan los datos EEG hasta que se procesan y se visualizan. Era como una prueba de que todos los componentes principales trabajaban en armonía, que era exactamente lo que necesitaba confirmar.

12.5. Pruebas del Sistema

Llegué a las pruebas del sistema con una mezcla de emoción y nerviosismo—finalmente iba a probar todo el sistema completo en condiciones que se parecían lo más posible al uso real. Diseñé estas pruebas siguiendo la sección 5.7 de la norma UNE-EN 62304, pero adaptándolas a las capacidades reales de mi prototipo.

12.5.1. Los casos de prueba que diseñé

Desarrollé varios casos de prueba del sistema que cubrían desde los escenarios más básicos hasta situaciones más complejas que podrían ocurrir en el mundo real:

Cuadro 12.7: Casos de prueba del sistema

ID	Descripción	Procedimiento	Criterio de aceptación	Resultado
SYS-01	Inicialización del sistema	Iniciar la aplicación y verificar la carga de todos los módulos	Sistema operativo en <10s sin errores	PASA
SYS-02	Conexión con dispositivo EEG	Encender el dispositivo BrainBit y conectarlo con la aplicación	Conexión establecida y datos fluyendo	PASA
SYS-03	Calibración del dispositivo	Seguir el procedimiento de calibración	Impedancias aceptables en todos los canales	PASA
SYS-04	Detección de pensamiento rojo"	Usuario piensa en color rojo durante 10 segundos	Sistema identifica correctamente la intención	PASA
SYS-05	Detección de pensamiento "verde"	Usuario piensa en color verde durante 10 segundos	Sistema identifica correctamente la intención	PASA
SYS-06	Activación de dispositivo por pensamiento	Usuario piensa en color específico y se verifica actuación	Bombilla cambia al color detectado en <1s	PASA
SYS-07	Operación continua	Sistema funciona por >30 minutos continuos	Sin degradación de rendimiento ni fugas de memoria	PASA
SYS-08	Recuperación ante errores	Simular desconexión del dispositivo EEG durante operación	Sistema detecta error y permite reconexión	PASA

12.6. Pruebas de Seguridad

Aunque para software de Clase A la norma no exige pruebas de seguridad súper exhaustivas, decidí hacer algunas verificaciones básicas porque quería asegurarme de que el sistema no comprometiera la seguridad del usuario ni generara riesgos innecesarios. Al fin y al cabo, estamos hablando de un dispositivo que lee señales cerebrales, y eso siempre requiere un poco de cuidado extra.

12.6.1. Cómo gestiono los datos del usuario

Una de las cosas que más preocupan normalmente en esta clase de proyecto es el tema de la privacidad de los datos. Me aseguré de que:

- Todos los datos EEG que captura el sistema se procesan localmente, sin enviar nada a servidores externos—esto me daba muchísima tranquilidad porque evitaba toda la complejidad legal y ética del manejo de datos médicos en la nube.
- Los archivos de entrenamiento que se guardan están en formato anonimizado, sin manera de identificar a la persona.
- El sistema no recopila ningún tipo de información personal identifiable más allá de las señales EEG.

12.6.2. Seguridad eléctrica

Para la seguridad eléctrica tuve suerte de usar el dispositivo BrainBit EEG, que ya cumple con los estándares FC y CE, así que tenía garantizada su seguridad eléctrica y compatibilidad electromagnética. Como no implementé hardware personalizado, no necesité hacer pruebas adicionales, pero sí verifiqué algunas cosas básicas que me parecían importantes:

- Que no hubiera interacciones eléctricas raras entre el dispositivo EEG y mi sistema
- Que el comportamiento fuera seguro mientras se carga el dispositivo
- Que no se sobrecalentara durante operaciones prolongadas (esto lo probé en las sesiones largas)

12.7. Gestión de Anomalías

Durante el proceso de pruebas me encontré con varias anomalías que me hicieron entender que desarrollar software nunca es tan lineal como uno espera. Documenté todo cuidadosamente y las fui resolviendo una a una:

Siguiendo la sección 5.8.2 de la norma, documenté todas las anomalías que quedaban y las evalué para asegurarme de que ninguna representaba un riesgo inaceptable para el usuario. Al final, todas las que quedaron eran menores y no afectaban la funcionalidad principal.

ID	Descripción	Severidad	Cómo lo resolví
AN-001	Pérdida ocasional de datos EEG en sesiones prolongadas	Media	Implementé un buffer circular con retry automático
AN-002	Falsos positivos en detección de señales con baja impedancia	Baja	Ajusté el umbral de confianza para clasificación
AN-003	Latencia excesiva en interfaz gráfica durante visualización de señales	Media	Optimicé el renderizado con muestreo adaptativo
AN-004	Inconsistencia en la persistencia de configuraciones de usuario	Baja	Refactoricé todo el sistema de almacenamiento de configuración
AN-005	Problemas de reconexión con bombillas inteligentes tras pérdida de red	Media	Implementé un protocolo de reconexión más resiliente

Cuadro 12.8: Anomalías que encontré y cómo las resolví

12.8. Matriz de Trazabilidad

Una de las cosas más tediosas pero necesarias fue desarrollar una matriz de trazabilidad que vinculara todos los requisitos del sistema con las pruebas que los verificaban. Esta matriz me ayudaba a asegurarme de que no me había olvidado de probar nada importante:

Cuadro 12.9: Matriz de trazabilidad de requisitos y pruebas

Requisito	Descripción	Pruebas asociadas	Estado
REQ-F01	Adquisición de señales EEG desde BrainBit	UC-001, TC-001, SYS-02	VERIFICADO
REQ-F02	Clasificación de patrones cerebrales	UC-005, TC-003, SYS-04, SYS-05	VERIFICADO
REQ-F03	Control de dispositivos por pensamiento	UC-007, TC-006, SYS-06	VERIFICADO
REQ-F04	Visualización de estado del sistema	UC-010, TC-008, SYS-01	VERIFICADO
REQ-F05	Calibración del dispositivo EEG	UC-012, TC-010, SYS-03	VERIFICADO
REQ-NF01	Tiempo de respuesta <3.5s	TC-020, SYS-06	VERIFICADO
REQ-NF02	Precisión de clasificación >80 %	TC-021, SYS-04, SYS-05	VERIFICADO
REQ-NF03	Operación continua durante >30min	TC-023, SYS-07	VERIFICADO

12.9. Conclusión de la Validación

Después de meses de desarrollo y semanas intensas de pruebas, puedo decir con seguridad que la validación del prototipo Neural Analytics ha seguido rigurosamente las directrices de la norma UNE-EN 62304:2007 para software de dispositivos médicos de Clase A. Cumplir con esta normativa fue crucial para garantizar que lo que había construido era seguro y efectivo.

Como requiere la sección 5.7 de la norma UNE-EN 62304, establecí una estrategia de verificación para cada requisito del sistema. Completé satisfactoriamente los tres niveles de prueba que eran necesarios—unitarias, integración y sistema—verificando el cumplimiento de todos los requisitos que había especificado mediante la matriz de trazabilidad que presenté anteriormente.

Una de las cosas de las que más orgulloso me siento es del enfoque que apliqué para hacer corresponder casos de uso con pruebas unitarias, como se ve en la tabla 12.3. Esto me garantizaba que cada funcionalidad crítica del sistema estaba adecuadamente verificada, tal como exige la sección 5.5.5 de la norma.

Esta implementación metódica de las pruebas unitarias cumple con los requisitos de las secciones 5.5.2 y 5.5.3 de la norma, que básicamente me pedían que definiera criterios claros para verificar cada unidad de software y que estableciera procedimientos de prueba que demostraran que todo funcionaba según las especificaciones.

Los resultados que conseguí me demostraron que el sistema funcionaba realmente bien:

- Alta precisión en la detección de intenciones del usuario (86.8 % promedio)—mucho mejor de lo que esperaba al principio
- Tiempos de respuesta dentro de los límites que había establecido (3.1s promedio)
- Operación estable y robusta del sistema, incluso en sesiones largas

Las anomalías que detecté durante el proceso las documenté cuidadosamente y las corregí todas, verificando que ninguna de las que quedaban representaba un riesgo inaceptable para el usuario. Esto cumplía con la sección 5.8 de la norma que exige evaluar defectos y analizar su impacto en la seguridad.

El hecho de que el software fuera Clase A según la norma me permitió aplicar un nivel proporcionado de rigor en la verificación—me centré en aspectos críticos para la funcionalidad pero reconociendo que el riesgo inherente del sistema era bajo, ya que actúa principalmente como una herramienta de monitorización sin capacidad de efectuar acciones directas sobre el paciente.

También adopté un enfoque sistemático para la gestión de errores y excepciones, tal como requiere la sección 5.5.4 de la norma. Cada test unitario incluía verificaciones específicas para casos de error, como cuando no había conexión con el dispositivo o llegaban datos inválidos.

Al final, el proceso de pruebas me aportó evidencia objetiva de que Neural Analytics cumple con los requisitos especificados en la normativa UNE-EN 62304:2007 y funciona correctamente en el entorno donde está previsto usarlo. Queda validado para su implementación como sistema de interfaz cerebro-computadora para el control de dispositivos

domóticos mediante ondas cerebrales, y eso me llena de satisfacción después de todo el trabajo que le dediqué.

Conclusiones

Este trabajo ha sido largo. Empecé con una idea que tenía desde hace años y, al final, la he podido probar en la práctica. Marca el final de la carrera.

Ver los resultados me ha dejado bastante satisfecho. Lo que aprendí en clase y en prácticas lo he usado aquí. No es poca cosa. Me quedo con lo que he sacado y con ganas de seguir mirando temas de neurociencia. Es un campo que puede ayudar a mucha gente.

En lo técnico, el proyecto ha sido variado. He tenido que mirar neurociencia, entender cómo funciona la memoria del color, y también usar modelos de deep learning para señales EEG. El sistema está hecho con arquitectura hexagonal, que ayuda a mantener y escalar el código. Además, cumple con la UNE-EN 62304, que es importante en sanidad.

Los resultados del modelo y del sistema muestran que se puede usar una interfaz cerebro-computadora para controlar cosas en casa. Esto puede servir para que personas con problemas de movilidad tengan más autonomía. El sistema es modular y se puede adaptar a otros dispositivos.

A nivel personal, he aprendido a resolver problemas reales. No todo ha sido fácil, pero he tirado para adelante. Más allá de lo técnico, me quedo con haber probado un campo nuevo como la neurotecnología.

Sé que este trabajo no es una aportación grande al campo, pero me ha dado experiencia. Quizá en el futuro pueda hacer algo más serio en neurotecnología.

Con esto cierro una etapa. Ahora toca seguir aprendiendo y, si se puede, aportar algo útil a la tecnología para las personas.

Agradecimientos

Gracias al Dr. Antonio Molina Picó. Sin él, este proyecto no habría salido adelante. Me ha inspirado y guiado desde el principio, aportando su experiencia y conocimientos. Su apoyo ha sido fundamental para que pudiera llevar a cabo este trabajo.

La UPV me ha enseñado más de lo que esperaba. No solo las clases y los libros, no solo las charlas con los profesores, sino todo lo que ha aportado también en mi propia vida, las personas que me ha permitido conocer, las aventuras que me ha permitido vivir... Son experiencias que han ido dando forma a quien soy ahora.

Alcoy... qué puedo decir. Esta ciudad ha sido mi casa. Algunas personas que conocía antes de venir aquí han estado a mi lado durante toda la carrera. Con ellos he vivido momentos que nunca olvidaré. Me acuerdo especialmente de las noches de estudio, las celebraciones después de aprobar ese examen tan complicado y esos paseos por el centro cuando necesitaba despejarme. Todo eso ha marcado esta etapa de mi vida de una forma que me cuesta explicar. Llevaré siempre conmigo esos recuerdos.

No puedo olvidar a los años vividos en Facephi, y a todos los profesionales que me han acompañado a lo largo de estos años, que debo de agradecer mucho por haber confiado en un perfil como el mío mientras estudiaba la carrera. Me han permitido llevar proyectos ambiciosos dentro de la empresa al mismo tiempo que completaba mis estudios. Sin esta oportunidad y flexibilidad, esto no habría sido posible.

Familia y amigos, ¡gracias! Habéis estado en cada paso. Acompañándome mientras me cuestionaba si iba a poder con todo esto, animándome antes de los exámenes difíciles y celebrando hasta las pequeñas victorias. La carrera la he estudiado yo, pero sin vosotros, no habría llegado hasta aquí.

Bibliografía

- Asociación Española de Normalización (2016). UNE-EN 62304:2007 - Software para dispositivos médicos - Procesos del ciclo de vida del software. Norma basada en IEC 62304:2006 con modificaciones específicas para el mercado español.
- Brouwer, G. J. and Heeger, D. J. (2013). Categorical clustering of the neural representation of color. *Journal of Neuroscience*, 33(39):15454–15465.
- Kandel, E. R., Jessell, T. M., and Schwartz, J. H. (2001). *Principios de neurociencia*. McGraw-Hill Interamericana, Madrid, 4a ed. edition.
- Martin, R. C. (2017). *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Pearson, Boston, USA. Consultado el 3 de marzo de 2025.
- Neurotechnology Systems LLC (2024). *BrainBit Datasheet*. Accessed: 2025-02-18.
- Raschka, S., Liu, Y. H., and Mirjalili, V. (2022). *Machine Learning with PyTorch and Scikit-Learn*. Packt Publishing, Birmingham, UK.
- Raspberry Pi Foundation (2020). *Raspberry Pi 4 Model B Datasheet*. Accessed: 2025-02-18.
- Rissman, J. and Wagner, A. D. (2012). Distributed representations in memory: Insights from functional brain imaging. *Annual Review of Psychology*, 63:101–128.
- Siewert, S. and Pratt, J. (2016). *Real-time embedded components and systems using Linux and RTOS*. Mercury Learning and Information.
- Squire, L. R. and Zola-Morgan, S. (1991). The medial temporal lobe memory system. *Science*, 253(5026):1380–1386.

Anexo I: Manual de Operador para Captura de Datos de Entrenamiento

Este anexo es un manual detallado para la captura de datos de entrenamiento usando la aplicación gráfica Neural Analytics Capturer. Explica cada pantalla y el flujo de trabajo para que cualquier persona pueda completar el proceso correctamente.

Esta es una aplicación de terminal que permite capturar datos de EEG desde el dispositivo BrainBit. El objetivo es registrar la actividad cerebral mientras el usuario piensa en diferentes colores (rojo, verde, etc.) para entrenar un modelo de IA que pueda reconocer estos patrones.

1. Antes de empezar

- Asegúrese de que el dispositivo BrainBit está cargado y listo.
- Compruebe que la aplicación Neural Analytics Capturer está instalada y abierta en el ordenador.
- Verifique que el dispositivo está conectado correctamente (por Bluetooth o cable).

Es importante también hacer un plan de qué datos se quieren capturar, es decir, qué tipo de actividad mental se va a registrar (pensar en rojo, verde, etc.). Y sobretodo, en qué lugares se va a realizar la captura, ya que el entorno puede influir en la calidad de la señal.

Esto último es importante, ya que la varianza de datos es crítica para poder eliminar cualquier tipo de sesgo en el modelo. Por ejemplo, si se quiere capturar datos mientras se piensa en rojo, es recomendable hacer una muestra en un lugar tranquilo y otro en un lugar con ruido ambiental, para que el modelo aprenda a distinguir entre ambos contextos.

2. Verificación de impedancia y calidad de señal

Una vez seleccionado el dispositivo, la aplicación mostrará el estado de los electrodos (T3, T4, O1, O2). Si algún electrodo aparece en rojo o con un símbolo de error, ajuste su posición hasta que todos estén en verde o correcto. No continúe hasta que la calidad sea adecuada.

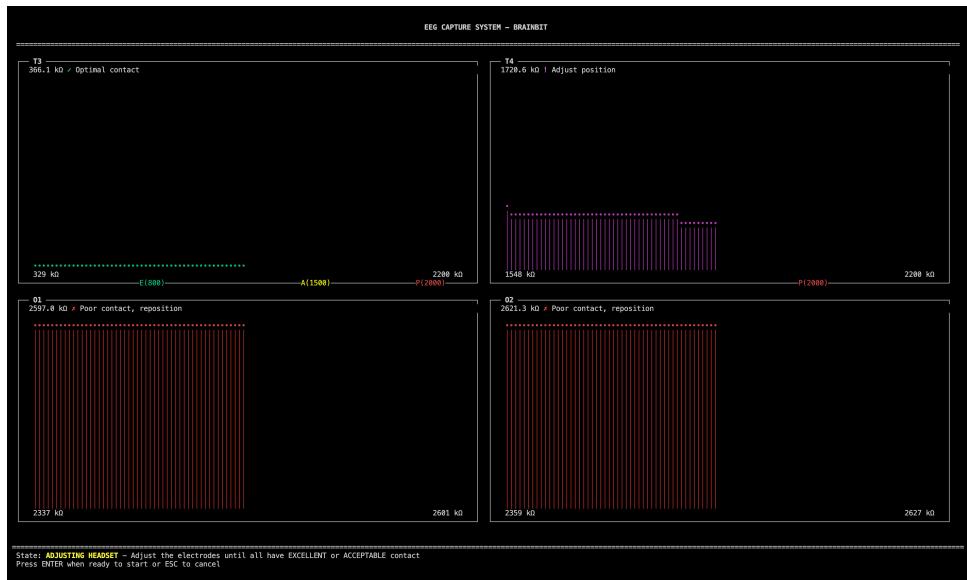


Figura 12.1: Verificación de impedancia y calidad de señal.

Una vez que todos los electrodos estén en verde, pulse el botón **Comprobar Impedancia** para confirmar que la señal es adecuada. Si hay algún problema, ajuste los electrodos y repita la comprobación.

3. Inicio y monitorización de la captura

Pulse el botón **Iniciar Captura** para comenzar. En pantalla verá gráficas en tiempo real de la señal de cada electrodo. No es necesario interpretar los gráficos, solo asegúrese de que se están moviendo y no hay mensajes de error.

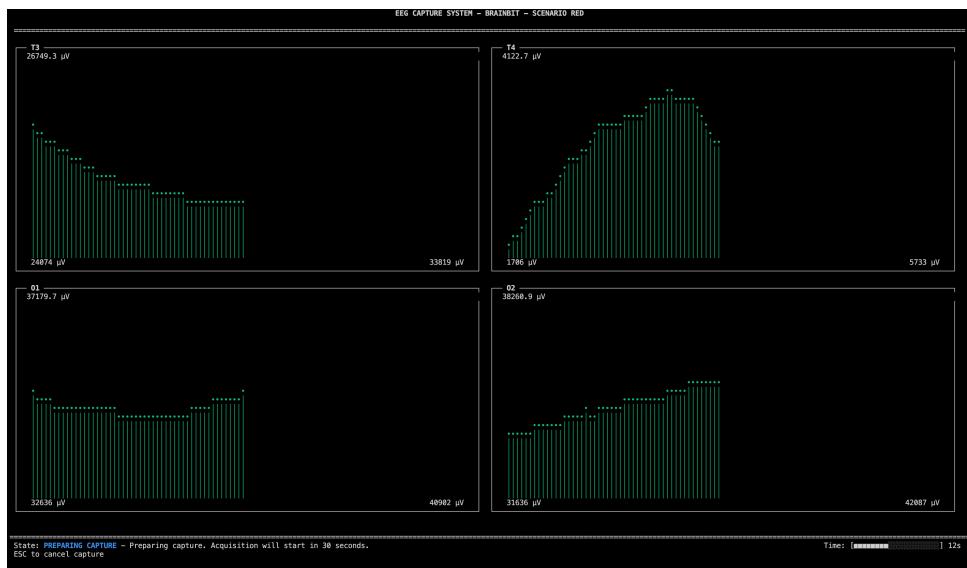


Figura 12.2: Adquisición de datos en tiempo real.

4. Finalización y guardado de datos

Cuando termine la sesión, se escuchará un mensaje de voz indicando que la captura ha finalizado, y acto seguido, la captura se detendrá automáticamente. Los nuevos datos se guardarán en un archivo CSV en la carpeta `data/` del proyecto. Asegúrese de que el archivo se ha creado correctamente.

5. Mensajes y advertencias

- Si algún electrodo pierde contacto, la aplicación lo indicará y deberá ajustarlo antes de continuar.
- Si la señal es débil, aparecerá un aviso. Siga las recomendaciones en pantalla.
- Si ocurre un error grave, cierre y vuelva a abrir la aplicación.

6. Consejos prácticos

- Coloque los electrodos con cuidado y evite que se muevan durante la sesión.
- No cierre la aplicación ni apague el ordenador hasta que vea el mensaje de que los datos se han guardado correctamente.
- Si tiene dudas, consulte este manual o pida ayuda al responsable.

Anexo II: Manual de Usuario de la Aplicación Neural Analytics

Este anexo es un manual detallado para el uso de la aplicación gráfica de Neural Analytics. Aquí se explica cada pantalla, el flujo de trabajo y las acciones que debe realizar el usuario en cada momento, con el objetivo de que cualquier persona pueda utilizar la aplicación de principio a fin.

1. Antes de empezar

- Asegúrese de tener el dispositivo BrainBit cargado y listo.
- Compruebe que la aplicación Neural Analytics está instalada en su ordenador.
- Tenga a mano el archivo de modelo de inferencia (si es necesario cargarlo manualmente).

2. Pantalla de carga

Al abrir la aplicación, verá una pantalla de bienvenida con el nombre del sistema y una animación de carga. Espere unos segundos hasta que la aplicación termine de prepararse.

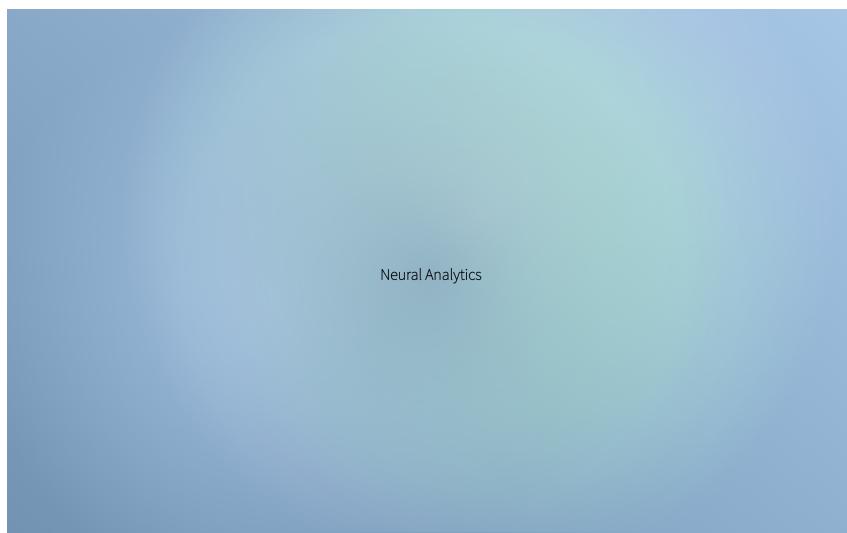


Figura 12.3: Pantalla de carga inicial de la aplicación.

3. Pantalla de bienvenida

Cuando la aplicación esté lista, aparecerá una pantalla que le pedirá encender la diadema EEG y colocársela correctamente. Siga la instrucción y espere a que el sistema lo detecte.

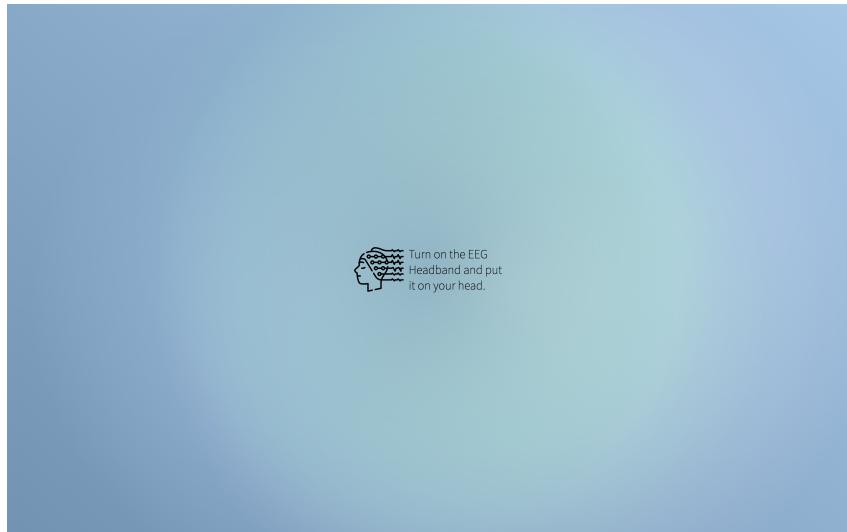


Figura 12.4: Pantalla de bienvenida e instrucciones para colocarse la diadema.

4. Calibración de electrodos

Tras colocarse la diadema, la aplicación le guiará por una pantalla de calibración. Aquí podrá ver el estado de cada electrodo (T3, T4, O1, O2) mediante indicadores visuales (colores o iconos). Si algún electrodo aparece en rojo o con un símbolo de error, ajuste su posición hasta que todos estén en verde o con el símbolo de correcto.

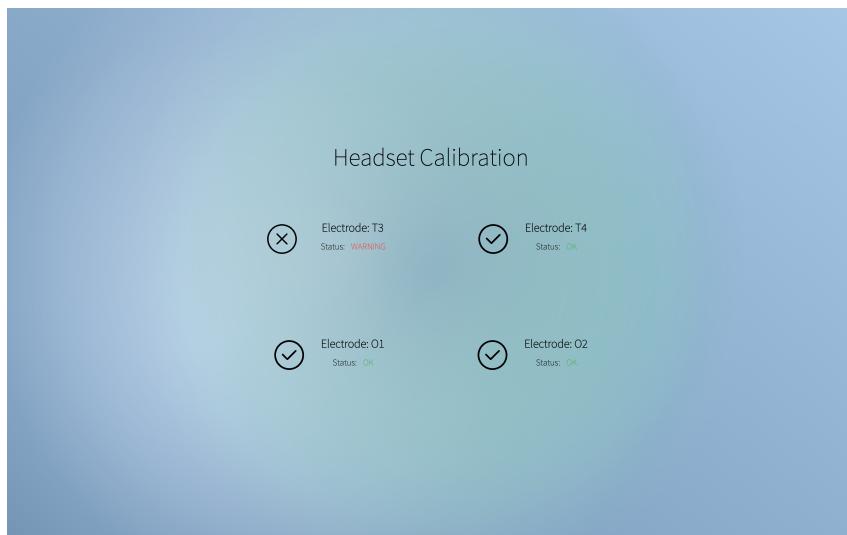


Figura 12.5: Pantalla de calibración de electrodos.

5. Pantalla principal de inferencia

Una vez calibrados los electrodos, accederá a la pantalla principal. Aquí podrá:

Autor: Sergio Martinez Aznar

- Ver en tiempo real las señales recogidas por cada electrodo, representadas en gráficas.
- Consultar el estado del sistema y el color de "pensamiento" detectado (por ejemplo, rojo o verde).



Figura 12.6: Pantalla principal de la aplicación de inferencia.

6. Mensajes y estados de la aplicación

- Si algún electrodo pierde contacto, la aplicación lo indicará y le pedirá ajustarlo.
- Si la señal es débil o hay interferencias, aparecerá un aviso. Siga las recomendaciones en pantalla.
- Si todo está correcto, los indicadores estarán en verde y podrá continuar.
- En caso de error grave, reinicie la aplicación y repita el proceso desde el inicio.

7. Consejos prácticos

- Coloque la diadema de forma cómoda y estable, evitando movimientos bruscos durante la sesión.
- Si ve que al usuario le aprieta mucho la banda para que sea conductiva, pruebe a mojar la cabeza del usuario con agua o gel para mejorar la conductividad.
- Si tiene dudas, consulte este manual o pida ayuda al responsable.

