UNIVERSIDAD DE GRANADA

TRABAJO FIN DE GRADO

GRADO EN INGENIERÍA INFORMÁTICA

Dispositivo para detección de escritura mediante Deep Learning en un sistema empotrado

DeepPen

Autor
Antonio Priego Raya

Directores
Jesús González Peñalver
Juan José Escobar Pérez

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

Granada, Julio de 2022

Dispositivo para detección de escritura mediante Deep Learning en un sistema empotrado

DeepPen

Autor
Antonio Priego Raya

Directores
Juan José Escobar Pérez
Jesús González Peñalver

Detección de escritura mediante Deep Learning en un sistema empotrado

Deep Learning en sistemas empotrados: TinyML.

Antonio Priego Raya

Palabras clave: TinyML, Machine learning, Deep learning, Sistemas empotrados, Reconocimiento letras, Redes neuronales convolucionales, ...

Resumen

Project Title: Project Subtitle

First name, Family name (student)

Keywords: Keyword1, Keyword2, Keyword3, ...

Abstract

Write here the abstract in English.

Yo, **Antonio Priego Raya**, alumno de la titulación *Grado en Ingeniería Informática* de la **Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada**, con DNI 31033948W, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Grado en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Fdo: Antonio Priego Raya

Granada a 8 de Julio de 2022

D. **Jesús González Peñalver**, Catedrático del departamento de Arquitectura y Tecnología de Computadores de la Universidad de Granada.

D. **Juan José Escobar Pérez**, Profesor Sustituto Interino del departamento de Arquitectura y Tecnología de Computadores de la Universidad de Granada.

Informan:

Que el presente trabajo, titulado ***Dispositivo para detección de escritura mediante Deep Learning en un sistema empotrado***, ha sido realizado bajo su supervisión por **Antonio Priego Raya**, y autorizamos la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expiden y firman el presente informe en Granada a X de mes Julio de 2022.

Los directores:

Jesús González Peñalver **Juan José Escobar Pérez**

Agradecimientos

Poner aquí agradecimientos...

Índice general

Índice de figuras

Capítulo 1

Introducción y Motivación

Como consecuencia del desarrollo de la informática, y la expansión y filtración del uso de equipos informáticos en la población general, cada vez la escritura tradicional cae más en desuso. Y es que una vez se supera la etapa académica, pocas personas siguen utilizando en su cotidianidad la escritura manual. Incluso para la educación hay un creciente movimiento de adaptación tecnológica que releva cada vez más al lápiz y papel.
No es el objetivo pecar de romanticismo y mirar a través de la lente de la nostalgia, sino avanzar, eso sí, intentando conservar en el proceso de innovación, las técnicas que nos han traído hasta aquí.

Por lo que el motivo de este trabajo siempre ha sido tratar de, creando nueva tecnología, favorecer el uso de la escritura. Ya que el avance y el progreso es no solo imparable sino necesario, la única forma de preservación de la grafía manual es establecer alternativas modernas que complementen a los dispositivos ya constituidos y que empleamos en el día a día.

Gracias al avance tecnológico de décadas, hoy podemos contar con herramientas informáticas de gran capacidad como lo son todos los mecanismos de Inteligencia Artificial. Concretamente el Deep Learning y las redes neuronales son conceptos en gran expansión durante los últimos años. El *Deep Learning* es un campo que está cambiando la informática como la concebíamos, revelándose como una alternativa sobresaliente para problemas que trabajan con grandes volúmenes de información, que presentan una elevada complejidad o simplemente que cumplen mejor de lo que habituaban a hacerlo con técnicas previas. Respondiendo oportunamente al contexto temporal vigente donde el *Big Data*, *Data Science*, automatización de tareas cotidianas, detonación de herramientas y dispositivos inteligentes, humanización de robots y robotización de personas; perfilan y caracterizan la fase en la que nos encontramos. Hecho que nos lleva al interés por un terreno tan intrincado como útil y repleto de potencial. Un potencial evidenciado por las tantas aplicaciones con sobrecogedores resultados que hace pocos años casaban más con la ciencia ficción que con algo alcanzable,

y que emplean esta herramienta y que serán citadas a lo largo de este trabajo.

Por sus demostradas altas capacidades para la clasificación en el procesamiento de imagen, por el hito que supuso integrar redes neuronales en sistemas tan reducidos, porque hay algo sugestivo en el hecho de rescatar lo tradicional mediante las técnicas más incipientes, pero por encima de todo, por lo estimulante que resulta trabajar con estos mecanismos y que es algo que siempre ha rondado entre mis pensamientos; este trabajo consistirá en trasladar a la realidad una alternativa moderna a la escritura manual, haciendo uso de Deep Learning en un sistema empotrado para mantener autonomía.

Los usos pueden ser los que se deseen y se alcancen a imaginar; con pocos añadidos podría convertirse en una herramienta para introducir a personas de avanzada edad al manejo de ordenadores, reduciendo la barrera de entrada al tener una forma de interactuar que ya les es familiar; en un instrumento para hacer más ameno y dinámico el proceso de aprender a escribir para niños y niñas; en un cuaderno virtual en el que anotar cuanto queramos sin necesidad de transportar el medio en el que se escribe; incorporando una punta con grafito o tinta, podríamos transcribir digitalmente lo que escribimos en cada momento de manera física, es decir, una copia digital, un registro de lo que hemos escrito; etc.

Capítulo 2

Antecedentes y estado actual

2.1 Redes neuronales

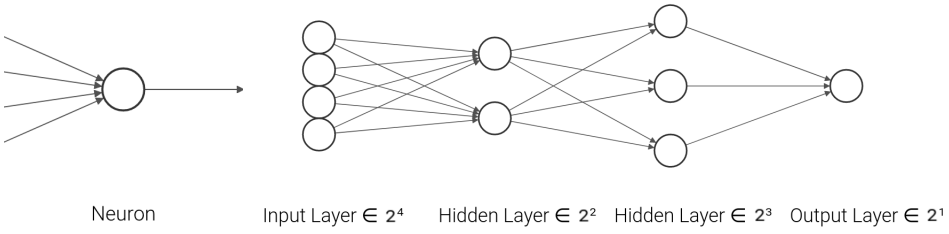
Pese a que es ahora, en los últimos años cuando, debido a la explosión del fenomeno de la *inteligencia artificial*, comienza a ser más popular todo lo relacionado con la dotación de inteligencia a los dispositivos electrónicos; todo comenzó hace muchas décadas [25]. Ya en 1943, *Warren McCulloch* (neurofisiologo) y *Walter Pitts* (matemático), escribieron un artículo [23] acerca de las neuronas e incluso en el mismo, fueron capaces de diseñar una red neuronal simple usando exclusivamente circuitos eléctricos y fundamentado en algoritmos de *Lógica de umbral* (*Threshold logic*).

Más tarde, en la década de 1950, en los laboratorios de *IBM* de la mano de *Nathanial Rochester*, ocurrió el primer intento de simulación de red neuronal; intento que desembocó en fracaso. Sin embargo fue muy estimulante para el campo de la *IA* y motivó el planteamiento de lo que denominaron "máquinas pensantes". También hubo otros acercamientos como la sugerencia del insigne *John Von Neumann* de utilizar relés telegráficos o tubos de vacío para simular el funcionamiento simplificado de las neuronas.

No obstante, no sería hasta 1958 que el neurobiólogo *Frank Rosenblatt* comenzaría a trabajar en el *Perceptron* [31], para muchos el nacimiento de la red neuronal artificial. Como todo precursor, era simple y limitado; hoy se catalogaría de monocapa, algo que evidentemente, ya no se usa en redes neuronales contemporaneas. Sirviéndose de múltiples entradas binarias, era capaz de producir una única salida, basada ya entonces en la utilización de *pesos* (número que cuantifica la relevacia de la entrada respecto de la salida), conservada hasta día de hoy, aunque cabe destacar que entonces, los pesos eran directamente atribuidos por el científico al cargo. La salida binaria de esta neurona *Perceptron*, sería como consecuencia de la superioridad o la inferioridad de la suma de la multiplicación de los pesos respecto de un umbral; es por esto que es sabida su influencia del trabajo de *Warren McCulloch* y *Walter Pitts* anteriormente mencionado. Por

tanto se podía destinar a funciones lógicas binarias simples (OR/AND).

El siguiente paso natural era aumentar el número neuronas y capas, llegando en 1965 el *Multilayer Perceptron Perceptron* [42]. Como consecuencia de esta mejora y aumento de la complejidad, nacieron los conceptos de capas de entrada, ocultas y de salida. De igual forma y dado que el reparto de pesos todavía no se había automatizado, los valores con los que se trabajaban, seguían siendo binarios.



(a) Perceptron (b) Multilayer Perceptron

Figura 2.1: *Perceptron* frente a *Multilayer Perceptron*

Y este fue precisamente el siguiente escalón a rebasar, superado ya en la década de 1980, gracias a las *Neuronas Sigmoides Perceptron* [7], afines al *Perceptron* pero con la capacidad de trabajar con números reales. La función de salida ahora sería una sigmoide, a la que deben su nombre y convirtiéndose en la primera función de activación.

A partir de aquí y durante toda la década, comenzaron a aparecer todo tipo de novedades que continúan vigentes: redes *feedforward*, el algoritmo *backpropagation* o la *Red Neuronal Convolucional* (*Convolutional Neural Network*, CNN) [17]. Las *CNN* son especialmente convenientes para procesamiento de imagen y vídeo, en general información espacial, aunque también se han usado para tareas de procesamiento de lenguaje natural. Esto es debido a que la información se divide en subcampos que sirven como entrada a capas de procesamiento convolucional, encargadas de apreciar las distintas carcterísticas que servirán para la clasificación de la información de entrada.

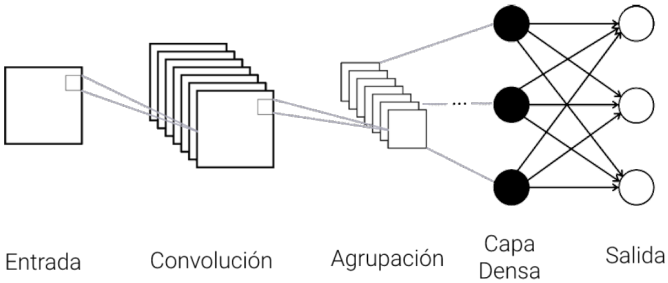


Figura 2.2: Estructura simplificada de *Red Neuronal Convolucional*

Es llamativo ver cómo las *CNN*, redes que mantienen su vigencia pese a que su origen se remonta a poco antes de los 90. Pero la realidad es que, si bien no lo parece, el campo de las redes neuronales lleva con nosotros mucho tiempo y las *CNN* no son el único ejemplo manifiesto. Las *Recurrent Neural Networks*, originadas en 1989, continúan en uso para procesamiento de datos secuenciales como lo es por ejemplo el texto.

Fue en 2006 cuando *Geoffrey Hinton et al* publicaron un famoso paper [13] presentando una red neuronal profunda, que entrenada, era capaz de reconocer dígitos. Acuñando como *Deep Learning*, a la técnica del *Machine Learning* (ya que se basa en el aprendizaje automático), que usa como mecanismo de procesamiento redes neuronales profundas. Se superaba entonces la barrera del entrenamiento de redes neuronales profundas, barrera que había llevado a la comunidad a congelar el avance de esta técnica, y que ahora elevaba al *Deep Learning* un nivel por encima del resto de técnicas del *Machine Learning*.

El siguiente hito llegaría a mediados de los 2000 al poder trabajar con redes neuronales profundas (*Deep Learning*), gracias a la introducción de preentrenamientos no supervisados para la asignación de pesos previos al usual entrenamiento del modelo. Este avance fue posible debido al desarrollo de la computación con GPUs.

El último acontecimiento o adición reseñable es el de las *Generative Adversarial Networks* (2014) [9], sujeto al empleo de dos redes neuronales complementarias: una se denomina *Generative network* en calidad de modelo generador de muestras y otra *Discriminative network* que evalúa las muestras generadas por la anterior y por el dataset de entrenamiento, es decir, recibe como entrada, la salida de la red anterior y del conjunto de datos de entrenamiento. El proposito de esta simbiosis es que la red generativa consiga reproducir muestras tan válidas como las de entrenamiento, a partir del juicio de la red discriminativa.

De entonces hasta ahora, más que innovación, se han dado muchos avances en términos de implementación, es habitual que cada cierto tiempo salga una nueva aplicación revolucionaria o con mucho potencial que está basada en redes neuronales. Y también es muy frecuente que gigantes tecnológicos como por ejemplo *Google* o *Facebook* compren otros proyectos basados en redes neuronales o directamente las empresas que los llevan a cabo. Siendo una de las más destacables la compra de *DeepMind* por parte de *Google* o la alianza entre *OpenIA* y *Microsoft*.

Algunos ejemplos de estos proyectos son: el tan sonado algoritmo de *AlphaGo* [11] de la empresa *DeepMind*, capaz de vencer al campeón mundial del juego tablero *Go*; el proyecto *DeepFace* de *Facebook* para identificar y automatizar el etiquetado de los usuarios en las imágenes; el *AlphaFold2* [10] de *DeepMind*, capaz de predecir la estructura de las proteínas y que ha sido revolucionario para la resolución del problema del plegamiento de proteínas, lo que antes eran investigaciones del orden de 1 o 2 años, ahora es computable en pocas horas; *GPT-3* [27] de *OpenIA*, un modelo de lenguaje cuya definición podría responder a *chatbot*, es capaz de completar texto, responder preguntas o cualquier tarea

que implique interacción con texto; *Copilot* [14] de *OpenIA* y *Github, Microsoft*, un sistema construido sobre los cimientos de *GPT-3* capaz de sugerir código autogenerado y comentarios analizando bien las directrices de un comentario o bien directamente interpretando lo que el programador busca; *Nerf* [26] de *Nvidia*, capaz de generar composiciones 3D a partir de imágenes fíjas; o por finalizar esta interminable lista de apasionantes ejemplos, el reciente *DALL.E* [27] de *OpenAI*, modelo generador de imágenes a partir de una entrada de texto descriptora.

No se puede quedar sin mencionar las que han sido las dos últimas grandes agitaciones del mundo de las redes neuronales y que están detrás de la mayoría de los ejemplos anteriores: el *Natural Language Processing* y los *Transformers*, aunque en realidad, van de la mano. Van de la mano porque el *Natural Language Processing* [21] ya es en sí mismo una revolución para el mundo de las redes neuronales, ya que el procesamiento de lenguaje, dado que las redes interpretan información numérica, siempre ha sido un desafío para el campo del *Machine Learning*; y no ha sido hasta su llegada, que gracias a lo que propone (vectorización de *tokens*, que son los bloques de datos que se interpretan, ya sean palabras o generalmente en la práctica, subpalabras), que las redes no han empezado a operar de una forma realmente veraz con el lenguaje. Sin embargo no solo ha sido una revolución en sí mismo, sino que ha propiciado el nacimiento de otra como lo son los *Transformers* [22], que parten del progreso conseguido en las redes recurrentes o para ser más precisos, de sus *Mecanismos de Atención* [37], ya que es lo único que mantienen respecto a los modelos recurrentes, es más, se alejan íntegramente pasando a un procesamiento simultáneo y sustituyendo la ordenación recurrente por la vectorización. Aunque pese a renunciar a la recurrencia, y es ahí donde reside su potencial, continúan funcionando con información secuencial.

Esta mejora en la implementación y aparición de tantas aplicaciones puede inferirse que se debe al aumento de la capacidad de cómputo de las GPUs, la llegada de los *Transformers*, la entrada de los mayores gigantes tecnológicos, pero sin duda a la aparición de herramientas de alto nivel e infraestructuras para el trabajo con redes neuronales como lo son *Azure*, *Aporia*, *TensorFlow*, *Keras*, *SciKit-Learn*, etc.

2.2 Microcontroladores

Los microcontroladores son sistemas de dimensiones reducidas y bajo consumo, destinados en su inicio al control de electrodomésticos, pero que a lo largo del tiempo y sobre todo propiciado por la aparición del *Internet of things*(*IoT*), y los avances en *IA* han supuesto un cambio en cómo se diseñan e implementan los nuevos dispositivos electrónicos.

El primer microcontrolador fue desarrollado por *Gary Boone* y *Michael Cochran* en 1971 y fue bautizado como *TSM 1000* [41], albergando una arquitectura *Harvard* en un mismo circuito contando con el propio microprocesador, memoria

ROM, menos de 256 bytes de memoria RAM y el propio reloj del sistema.

Como respuesta, *Intel* comercializó en 1977 su propio sistema para aplicaciones de control, el *Intel 8048* [40], que obtuvo gran popularidad y supuso un pequeño cambio en el paradigma de ventas de *Intel*.

Las memorias que montaban eran *EPROM* en el caso de los microcontroladores reprogramables y *PROM* en el caso de los de bajo presupuesto. Sin embargo esto cambió con la llegada en 1993 de la *EEPROM*, utilizada por primera vez en el *PIC16x64* [41] de *Microchip* y que conllevó un gran avance gracias a la agilización del proceso de creación de prototipos y su programación.

Poco después *Atmel* implementaría por primera vez memoria *flash* en un microcontrolador y se usaría en el *Intel 8051*, que trajo ciertos cambios respecto a su predecesor, como pasar a arquitectura *Von Neumann* o la inclusión de *Universal Asynchronous Receiver-Transmitter* (*UART*) para el manejo de puertos y dispositivos serie. Además contaba con múltiples compiladores de *C* para su programación, alternativos al lenguaje *ensamblador*.

Gracias a la inclusión de estas dos últimas memorias en el diseño estandarizado de los microcontroladores, el precio comenzó a ser cada vez más accesible. También se inició la incorporación de periféricos complementarios para dotar a los microcontroladores de más funcionalidad. Periféricos tales como generadores *PWM*, conversores analógicos A/D y D/A, relojes de tiempo real, etc.

Estos complementos lucen ahora arcaicos en comparación con los que se integran en microcontroladores actuales: transceptores 802.15.4, bluetooh, wifi, cámaras, micrófonos, y sensores de todo tipo como de presión, movimiento, orientación, color, brillo, proximidad, humedad, etc. Todos estos acompañados de complejos microprocesadores de 32 bits, cada vez más semejantes a las *CPU*s de equipos de mayores dimensiones. El progreso de los microprocesadores reducidos, ha sido propiciado por el vasto crecimiento del mercado móvil en los últimos años. Y resultando *ARM*, al igual que para los smartphones, una excelente baza para la complejidad que demandan los microcontroladores actuales.

Por un lado, su contenida complejidad frente a equipos de escritorio, hace que económicamente su implementación sea muy viable, aunque esto mismo provoca ciertos inconvenientes a la hora de su empleabilidad para *IA*, mencionados en la siguiente sección.

2.3 Integración de redes neuronales en microcontroladores

Algunos microcontroladores modernos dan soporte a herramientas para la *IA*, como lo es la integración de redes neuronales; sin embargo, el desarrollo de estas sigue siendo dependiente de la asistencia de un PC. De igual forma este soporte a herramientas para la *IA* es aun así sorprendente viendo los resultados que se pueden obtener de su implementación y siendo este proyecto muestra de ello. Aunque en algunas ocasiones son necesarios ciertos arreglos dadas las carencias

de estos dispositivos, como en ciertos casos, la falta de *FPU*s (*Floating-Point Unit*), suponiendo un obstáculo debido a que las redes neuronales realizan su procesamiento en coma flotante.

Se precisa de equipos con mayores prestaciones para la creación y entrenamiento de las redes neuronales que integrarán los microcontroladores, ya que es un proceso complejo y costoso; por suerte no es así para la ejecución, lo que los convierte en grandes candidatos gracias al *CloudML*, *EdgeML* o *TinyML*.

El *CloudML* [8] es la técnica por la que, alojando redes neuronales profundas en la nube, podemos integrar el uso de las mismas en *TPU*s (Unidades de procesamiento tensorial) y *FPGA*s, entre otras. Esta alternativa presenta la capacidad de emancipar el propio procesamiento de los algoritmos de *Macine Learning* fuera del propio dispositivo en el que se integra su implementación. Por otro lado, supone contar con infraestructuras que den soporte a ello y el uso de herramientas y entornos de pago, como entre otros el *Google Cloud ML Engine*.

EdgeML [15] es una librería escrita en *Python* propulsada por *GitHub, Microsoft* que mediante *TensorFlow* (o alternativamente en fase experimental *Pytorch*), provee de distintas funciones enfocadas al entrenamiento, evaluación y despliegue de algoritmos de *Machine Learning* para sistemas embebidos empleados para labores simples. Por lo que es una elección perfecta para aquellos proyectos en los que se quiera trabajar con *Machine Learning* y utilizar herramientas de apoyo *open source*. Aunque cabe mencionar que, al tratarse de una herramienta para *Machine Learning* en general, las opciones dirigidas a *Deep Learning* no abundan.

Micro-Learn [1] es una librería para *python* que convierte modelos de *machine learning* entrenados con *Scikit-Learn*, a código que virtualiza la ejecución del modelo en cualquier microcontrolador en tiempo real. Es relevante destacar que no existen demasiados proyectos, aunque a cambio, ofrece teórico soporte para cualquier microcontrolador *arduino*.

También existen infinidad de destacables alternativas para *FPGA*s como *Vitis-AI* o *VTA*, entre otras, aunque no presentan soporte a microcontroladores, por lo que quedan fuera de nuestro espectro de posibilidades.

Y finalmente, *TinyML* [35] se define como el campo que comprende a las tecnologías y aplicaciones relacionadas con el *Machine Learning* y que se fundamenta en la implementación de algoritmos y software capaces de realizar análisis de datos de sensores en un hardware muy limitado y de bajo consumo energético. Encontrando en esta alternativa, numerosos proyectos que consultar, gran actividad de su comunidad y cuantiosa documentación en forma de libros y publicaciones de usuarios.

Capítulo 3

Especificación del sistema

3.1 Requisitos funcionales y no funcionales

- El sistema procesará el movimiento resultando en la identificación de una letra.
- El sistema enviará la letra identificada por el sistema de comunicación pertinente.
- Al detectar un movimiento despreciable, el sistema lo descartará.
- Posterior a la identificación del movimiento, el sistema dejará un periodo suficiente de no registro de movimiento para que el usuario recoloque su postura.
- El sistema debe contar con un programa para ordenador, una interfaz gráfica que medie entre el dispositivo y el usuario.

3.1.1 Requisitos no funcionales

- El tiempo de procesamiento para la identificación de la letra, debe ser inmediato para generar sensación de escritura natural.
- El sistema debe funcionar con conexión por cable e inalámbrica.
- La interfaz de usuario debe ser simple de entender y usar.
- El sistema debe contar con autonomía energética.
- El dispositivo estará integrado en un encapsulado con forma de lápiz o bolígrafo.
- El encapsulado tendrá un tamaño semejante a un lápiz o bolígrafo.
- El microcontrolador debe tener unas dimensiones adecuadas pra encajar en el encapsulado.
- El sistema debe estar sujeto a un presupuesto limitado de no más de 60€.

3.2 Especificación formal

3.2.1 Especificación hardware

El dispositivo a crear clasificará, de forma autónoma, los distintos gestos que se realicen como letras, usando como herramienta de procesamiento Deep Learning. Por tanto la placa debe contar con:

- Dada la limitación de presupuesto, se optará por un microcontrolador.
- El microcontrolador debe ser compatible con el procesamiento de tensores.
- Sensores presentes en el microcontrolador suficientes para hacer reconocible el movimiento con precisión. En su defecto, se integrarán.
- Microcontrolador con tecnología inalámbrica.
- Microcontrolador con dimensiones reducidas.
- Documentación para su utilización y el respaldo de una comunidad activa para tener otros proyectos como soporte.
- Encapsulado que de cabida a todos los elementos hardware del proyecto.

3.2.2 Especificación software

- Interfaz de usuario simple que ofrezca las funcionalidades descritas.
- Framework adecuado para el diseño, entrenamiento, validación y testeo de redes neuronales y su integración en el microcontrolador.
- Firmware para el microcontrolador que integre la red neuronal para la tarea de detección de letras y la recolección del movimiento.

3.3 Planificación y presupuestación

3.3.1 Planificación

La planificación ha sido esencial para poner en valor los tiempos que manejar y ser consciente de las limitaciones. Es lo primero que se debe plantear unido a una preparación o documentación sobre lo que se va a trabajar para, solo de esta forma, poder estimar de una forma más precisa los plazos de cada elemento ineludible en el desarrollo del producto que se busca.

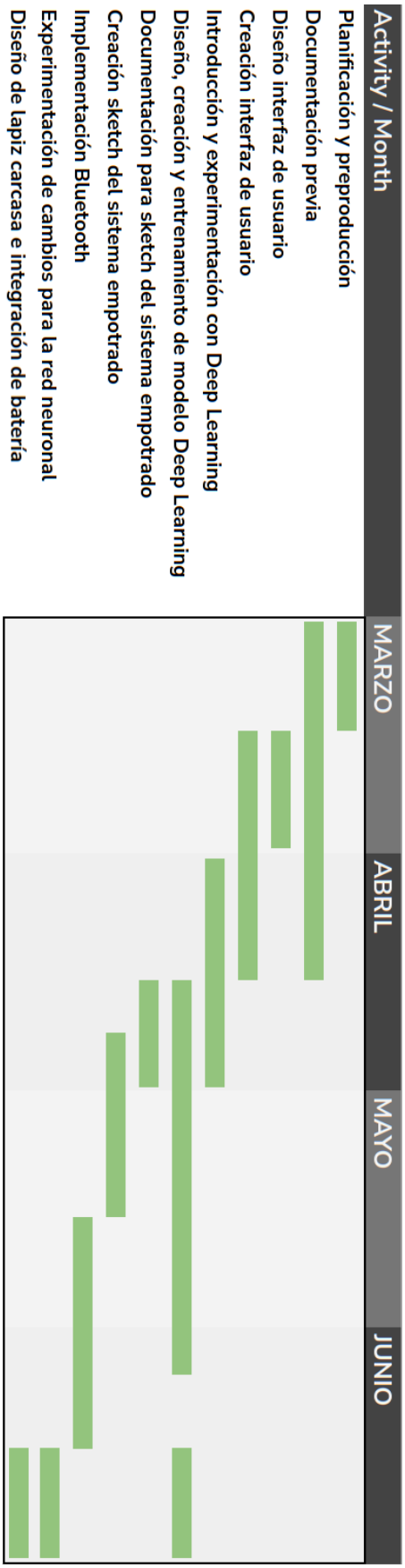


Figura 3.1: Diagrama *Gantt* para la planificación

Como muestra la *figura 3.1*, ha sido una planificación basada en contenidos, sin prácticamente iteraciones de desarrollo, ya que, dados todos los campos que se abarcan y dada la complejidad de alguno de ellos; es improbable poder contar con varias iteraciones. La excepción de este precepto viene con el modelo, con el que se trató de experimentar, con franqueza más por apetencia de estudiar las redes neuronales y su desempeño, que por necesidad.

Por lo que podríamos clasificar el desarrollo formalmente como un modelo de prototipos; donde se estudian los requisitos, se documenta al respecto y se crea un prototipo revisable en caso de incumplimiento de requisitos o por errores en la implementación que se reflejen negativamente en el prototipo.

Aunque solo encontramos esa segunda iteración o revisión en la red neuronal, hay un cierto patrón de documentación y ejecución para cada parte del proyecto, por lo que se podría decir que los procesos están fraccionados. Sin embargo no deja de ser, como se ha denominado anteriormente, una planificación basada en contenidos: se ejecuta una parte y se procede con la siguiente.

El orden ha sido relevante y tiene su propósito. En primer lugar está la documentación y planificación, seguida de la creación de la interfaz, no por otra cosa que la carencia del hardware. Estos plazos son los primeros ya que permiten trabajar sin el sistema empotrado; tiempos de selección y obtención del hardware. Complementariamente, el diseño de la interfaz de usuario abre la mente a reflexionar sobre qué podría ofrecer el dispositivo a la persona que hace uso de él, ayuda a pensar en nuevas funcionalidades.

El siguiente bloque de contenido es el del modelo basado en *Deep Learning*, ya que si el modelo no alcanzara en la fase de testeo unos resultados óptimos, todavía podemos contar con algo más de tiempo para solventar su eficiencia.

Y por último la creación del propio firmware del microcontrolador y la integración del bluetooth, dividida en la implementación en la interfaz de usuario y en el firmware; para poder congregar finalmente todos los elementos y probar el resultado del producto.

Una última fase de experimentación en la que también se incluirá la creación y producción del embellecimiento para el dispositivo en forma de lápiz y la integración de su batería en el mismo.

3.3.2 Presupuesto

Para esta sección ha de tenerse en consideración la situación en el momento del desarrollo del proyecto de escasez de silicio, huelgas de transporte, pandemia, etc. Repercutiendo directamente en el precio de la electrónica y en los tiempos de entrega.

Aclarado esto, el presupuesto dada la naturaleza del proyecto, su funcionalidad y que se realiza con fines académicos y experimentales; será uno limitado y acorde a lo que se podría esperar.

Descripción	Precio
Microcontrolador	40€
Batería	15€
Adaptación batería	2€
Impresión del encapsulado	1€
Cableado	7€
TOTAL:	**65'00€**

Tabla 3.1: Presupuesto estimado para la producción

Capítulo 4

Diseño del sistema

Para poder trabajar con objetivos claros, lo mejor es definir la estructura de nuestro proyecto, constreñir los elementos que constituirán el producto que se trata de alcanzar.

4.1 Estructuración del diseño del sistema

Lo primero es identificar los elementos. Es evidente que nuestro dispositivo estará integrado en un encapsulado, por lo tanto, cuando se haga referencia al *DeepPen*, se estará haciendo alusión al propio encapsulado que contiene toda la electrónica agregada. Dicha electrónica consta de dos partes físicamente apartados: el microcontrolador y la batería que lo alimentará cuando se haga uso de su característica inalámbrica(bluetooth), dotándolo de la autonomía necesaria. Nuestro *DeepPen* necesitará de un equipo en el que mostrar las funcionalidades que ofrece el producto, podría tratarse de, por ejemplo un smartphone, pero en este caso se ha optado por hacerlo en un ordenador para agilizar el desarrollo. El ordenador es una herramienta que será utilizada no solo cuando se concluya el desarrollo, como interfaz para el *DeepPen*, sino como herramienta para la propia producción de todo el software requerido durante el desarrollo del proyecto: interfaz de usuario, firmware del micrococontrolador y creación de la red neuronal. A su vez, el firmware hará uso de varios elementos clave para su funcionamiento; ya que la red neuronal tomará como entrada imágenes, se necesitará de una parte del firmware destinada a rasterizar el movimiento, movimiento que por otro lado deberá registrarse por medio de los sensores presentes en el microcontrolador. Finalmente, el dispositivo requerirá según lo especificado (sección 3.1.1), de mínimo dos canales de comunicación uno inalámbrico y otro por cable. Canales de comunicación que no solo se emplearán con fines de conexión con la interfaz de usuario, sino que también servirán para generar las muestras con las que se desarrollará (validación, entrenamiento y testeo) la red neuronal.

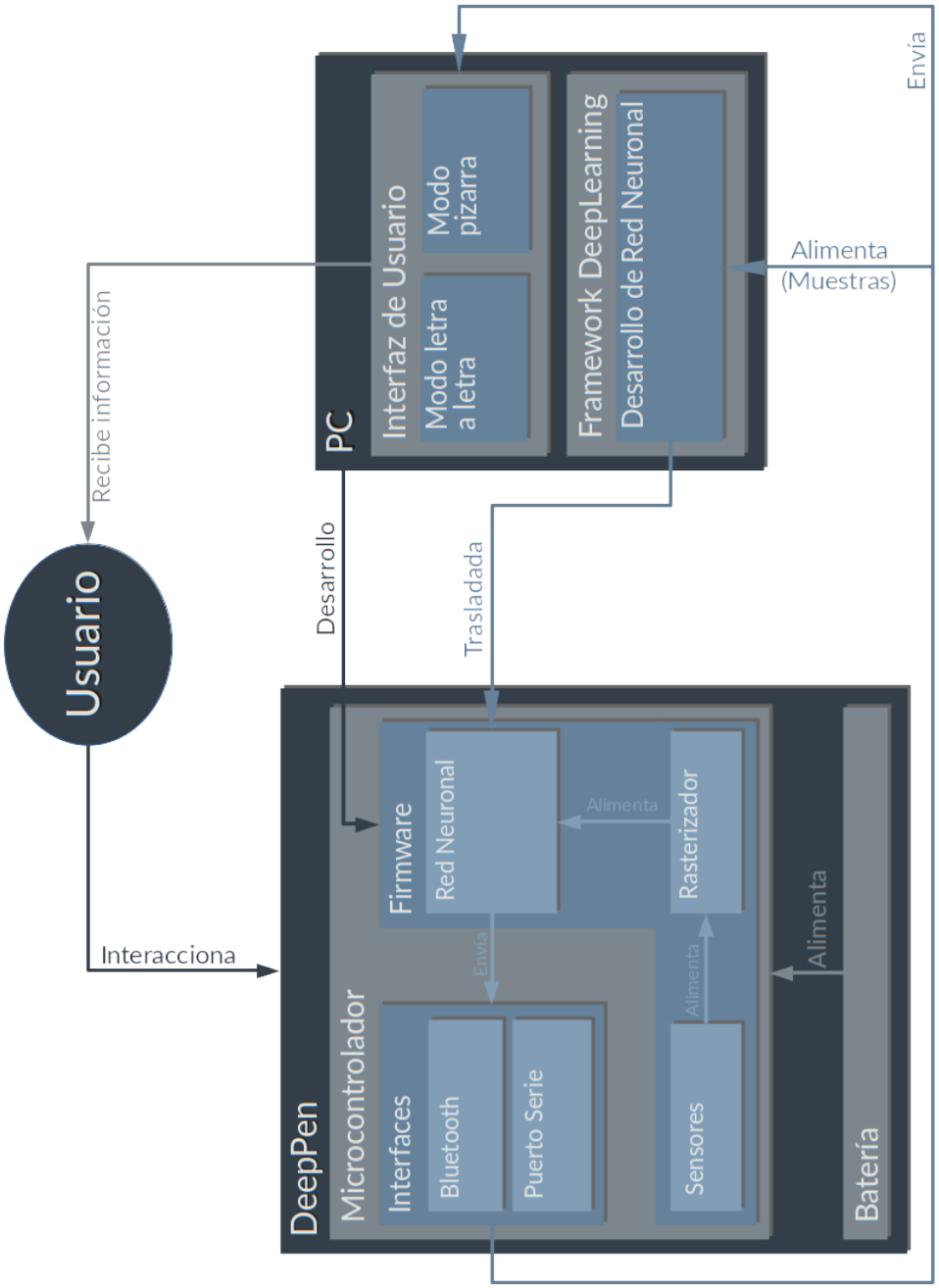


Figura 4.1: Esquema de la estructura del diseño del sistema

Capítulo 5

Microcontrolador

5.1 Planificación

5.1.1 Elección del microcontrolador

Como ya se especificó en la sección *(3.2.1)Requisitos hardware* necesitamos ciertas características inmutables. Para poder quedarnos con un microcontrolador, antes debemos hacer el ejercicio de búsqueda de algunos candidatos y analizar sus características. La búsqueda se ha sustentado en encontrar dispositivos compatibles con *TensorFlow Lite*.

Microcontrolador	Precio	Documentación	Dimensiones	Sensores	Disponibilidad	Soporte para DL
SparkFun Edge	✓	✓	-	✓	-	✓
Arduino Nano Sense 33 BLE	✓	✓	✓	✓	-	✓
STM32F746G	✗	-	✗	✓	✓	✓
Adafruit EdgeBadge	✓	✓	✗	✓	-	✓
STM32	✓	-	✓	✗	✓	✓
ESP32	✓	-	✓	✗	✓	✓

Tabla 5.1: Tabla de requisitos para elección de microcontrolador

La *ESP32* al igual que la *STM32* pueden descartarse debido a que carecen de sensores relacionados con el movimiento, se les podrían integrar periféricamente, pero sería algo más molesta su inserción en una carcasa. Y dado que tenemos alternativas que cuentan con estos sensores, podemos permitirnos descartarlas.

Por otro lado la *Adafruit EdgeBadge* es una placa muy llamativa, potente y llena de posibilidades, pero cuenta con unas dimensiones superiores a lo que se busca.

Respecto a la *STM32F746G* el inconveniente es la bajísima disponibilidad y los plazos de envío desorbitados. Por tanto tampoco podemos contar con ella.

Por lo que la disyuntiva se plantea entre la *Arduino Nano Sense 33 BLE* y la *SparkFun Edge*. En este caso la decisión no está motivada por características técnicas, que además son muy parecidas en ambos dispositivos, sino que una de ellas ofrece algo fundamental cuando se trabaja por primera vez en un campo y más aún cuando se cuenta con limitación de tiempo. El motivo taxativo es la documentación que provee Arduino, así como el apoyo de su activa comunidad y el gran volumen de proyectos que podemos consultar y que hacen uso de esta misma placa. Sumado a que este microcontrolador es prácticamente el más extendido para *Machine Learning*, por tanto no solo encontraremos multitud de proyectos en los que apoyarnos, sino que muchos de ellos o la práctica mayoría estarán enfocados al *Machine Learning*. Incluso se ha convertido en el abanderado de los proyectos basados en TinyML, el mayor valor añadido con el que puede contar un dispositivo que se empleará partiendo de pocos conocimientos y con restricción temporal. Cabe destacar también, que dada la complicada situación respecto al mercado de la electrónica al momento del desarrollo de este proyecto, la escasez de silicio, retrasos en la producción por la pandemia, huelgas de transporte, etc; la prontitud de entrega del propio microcontrolador, ya era en sí mismo el factor más determinante. Por lo cual me vi obligado a provisionarme de ambas y comenzar a trabajar con el microcontrolador que antes llegara. Por suerte pude hacerme primero con la *Arduino Nano Sense 33 BLE*, que era la prevista para el proyecto por lo expuesto.

De su propia nomenclatura podemos extraer todos los elementos precisados para este proyecto:

- Arduino: Garantiza que encontraremos documentación, y asistencia y proyectos de otros usuarios.
- Nano: Posee unas dimensiones convenientes para poder incorporarlo en un encapsulado adecuado para la escritura.
- Sense: Cuenta con diversos sensores, concretamente con una *IMU* (*Inertial Measurement Unit*) que provee de *giroscopio* y *acelerómetro*.
- BLE: *Bluetooth Low Energy*, un *Bluetooth* de bajo consumo que proporcionará autonomía y libertad de movimiento.

5.1.2 Elección del entorno de desarrollo

El entorno de desarrollo escogido será el propio *Arduino IDE*, debido a que nos facilita mucho el trabajo en ciertas tareas como el acceso al puerto serie para labores de depuración, la instalación de librerías para Arduino y sus dispositivos,

o la carga del firmware en la placa con un solo click. Adicionalmente, se hará uso de *Visual Studio Code* en los periodos de programación sin interacción con el microcontrolador, dado que es un entorno más cómodo para gestionar varios archivos simultáneamente y programar durante sesiones algo más largas.

5.1.3 Elección del *framework* para *Deep Learning*

A la hora de trabajar con *Deep Learning* y *redes neuronales*, es importante apoyarse en herramientas de alto nivel, ya que si desarrollaramos la red neuronal a bajo nivel, necesitaríamos de un nivel de documentación que llevaría mucho más tiempo del que tenemos. Y no solo eso, sino que no podríamos integrar el modelo en el microcontrolador. Por tanto, necesitamos de un marco de trabajo, un *framework*, que nos facilite el trabajo, y nos brinde la infraestructura y herramientas esenciales para poder desarrollar nuestro modelo basado en *Deep Learning*.

Ya vimos en la sección 2.3 (*Integración de redes neuronales en microcontroladores*) las alternativas que se nos presentan a la hora de integrar redes neuronales en microcontroladores; de todas ellas y debido a la actividad de su comunidad y la tendencia a exponer sus proyectos, se optará por *TinyML*, el cual está complementado a la excelencia con *TensorFlow Lite*. Por ello y por otras múltiples razones como que es de código abierto, gratuito, forma una gran sinergia al agregar algunas otras herramientas de alto nivel (como por ejemplo, *Keras* o *Scikit Learn*), etc; se ha optado por *TensorFlow Lite*. Pero al igual que en las elecciones anteriores, lo que más decanta la balanza es siempre la expansión desde su inicio y la cuota de utilización frente a sus alternativas. Ya que esto se traduce en, generalmente, mayor documentación, mayor interacción de la comunidad, más proyectos que poder consultar y más experiencias de otros usuarios que pueden ser de interés.

5.2 Diseño

5.2.1 Estructura del firmware del controlador

El firmware se distribuirá en diferentes secciones, la principal, *deep_pen.ino*(extensión propia de *Arduino*, empleada en el archivo principal de sus proyectos) incluirá todo lo relativo a la configuración previa de las características del microcontrolador y las habituales funciones *setup()* y *loop()*.

En *deep_pen_model_data* encontraremos exclusivamente el modelo de la red neuronal entrenado y listo para funcionar en formato binario, dada la carencia de sistema de archivos.

labels será la sección que ocupe la gestión de las etiquetas del modelo; definición y traducción etiqueta a letra.

El apartado *rasterize_stroke* rasterizará el movimiento recogido, transformándolo en las imágenes que sirven como entrada para el modelo.

Por último, *stroke_collector* estará reservado a la recolección del movimiento y la configuración del servicio *Bluetooth*(*BLE*) vinculado a la recolección de muestras para el modelo.

5.2.2 Servicio *Bluetooth*

Pese a que se implementarán dos servicios, uno de ellos es parte del *Data collector*, sección extraida del proyecto de *Pete Warden* [38] y por tanto no la desarrollaré más allá de una breve explicación en su apéndice. Se describirá, por tanto, solamente el servicio implementado de cero: *letterSenderService*.

Previo a la descripción del diseño, debemos entender cómo funciona esta versión bluetooth de bajo consumo.

Teoría 5.2.1: Estructura del *BLE*(*Bluetooth Low Energy*) [3, 6, 36]

El funcionamiento de este bluetooth de bajo consumo es notoriamente disidente de la versión general, tanto que tenemos que hablar de una estructura propia y que será clave para poder hacer uso de esta herramienta. Esta estructura jerárquica está definida por ***Atributos*** (Attributes). Cada uno de los elementos a continuación enumerados son ***Atributos***, todos ellos identificados por un ***UUID***(Universally Unique Identifer):

1. ***Servicios*** (Services)
   Agrupaciones de características. Un servicio suele componerse de características vinculadas al ámbito del servicio. Generalmente cada servicio corresponde a una prestación del dispositivo
2. ***Características*** (Characteristics)
   Cada característica contiene un tipo(***UUID***) de característica, sus propias propiedades y sus propios permisos. Y continuando con la disposición jerárquica, cada característica está formada por ninguno, uno o múltiples descriptores.
   Representan estados del dispositivo, datos de la configuración del mismo o simplemente un dato correspondiente a alguna función del servicio.
3. ***Descriptores*** (Descriptors) La unidad mínima de la estructura. Es la que contiene la información transmitida por cada comportamiento de una característica y sus metadatos asociados.

El servicio (*letterSenderService*) está compuesto por dos características: rx(*rxChar*) y tx(*txChar*). Tomaremos estas características como canales de comunicación unidireccionales. Han sido denominados teniendo en cuenta la placa como sistema de referencia; *rx* será la característica receptora de datos y *tx* la característica transmisora.

Utilizaremos el canal *tx* para transmitir la letra y el canal *rx* a modo de gestor de flujo; para la comunicación con el programa de usuario. Cuando el interfaz de

usuario reciba la letra y la almacene, escribirá en el canal *rx* la correspondiente señal para que el canal *tx* se borre y pueda dar paso a una nueva letra.

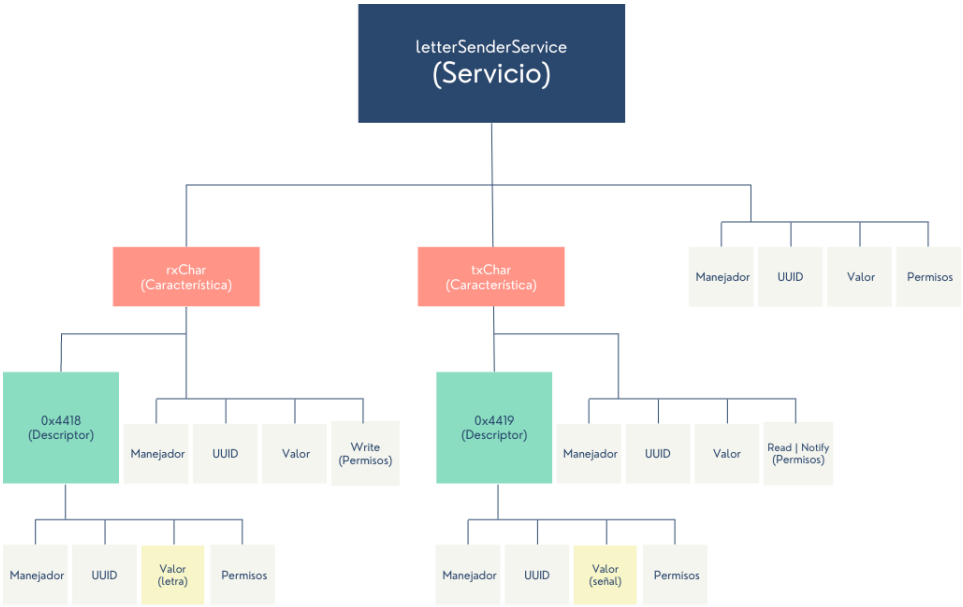


Figura 5.1: Esquema de la estructura planteada para el servicio *BLE*

5.3 Implementación

Previo a la implementación del código, se deben instalar ciertas librerías (proceso sencillo gracias a la decisión de utilizar el *Arduino IDE* pero que puede consultarse en el apéndice 9.3) para el trabajo con algunas características de la *Arduino Nano Sense 33 BLE* como lo son la librería *Arduino_LSM9DS1*, para el uso de la IMU y su acelerómetro, magnetómetro y giroscopio; y la librería *ArduinoBLE*, para el uso del *Bluetooth* de bajo consumo. Por otro lado, también es necesario instalar la librería *Arduino_TensorFlowLite*, para trabajar con el modelo de la red neuronal en el firmware del microcontrolador.

Además, en ocasiones, es necesario dar permisos al puerto del microcontrolador como reza el *apéndice 9.8*

5.3.1 Definición de parámetros de *Bluetooth* [19]

En el primer bloque del firmware, encontramos definición de los parametros que más tarde se utilizarán para configurar el *BLE*(Bluetooth Low Energy).

La configuración de permisos de las características son consecuentes con su cometido. La característica de lectura, tendrá permisos de escritura para que la **central** pueda escribir los valores que la placa leerá. Análogamente la característica de escritura tendrá permisos de lectura y notificación.

Teoría 5.3.1: Roles de las partes en conexiones bluetooth [36]

Al darse una conexión bluetooth, existen ciertas implicaciones inherentes a la propia conexión. Y es que siempre habrá una de las partes que se lucra de los servicios de la(s) otra(s).
Pues bien, los dispositivos que se gestionan o se benefician de los servicios de otros, se conocen como **central** y los que proveen, son los **periféricos**.

El resultado de la configuración del servicio *BLE* obedece completamente al diseño planteado.

Finalmente se deben definir una serie de manejadores para gestionar los distintos eventos relacionados con el servicio *BLE*. Estos manejadores se activarán como consecuencia de suceso y se encargarán de gestionar la respuesta a dicho suceso; concretamente respuestas a conexiones, desconexiones o recibo de señales por el canal de lectura *rx*.

5.3.2 Función de configuración del firmware [2] [38]

En esta sección (denominada función '*setup()*' en el diseño de *Arduino*), como de constumbre, se deberá llevar a cabo el ajuste propio para la ejecución de nuestro código.

Es reseñable mencionar que el microcontrolador con el que estamos trabajando solo posee la *memoria flash* y *SRAM* características de los dispositivos de estas prestaciones. Sin embargo tenemos que almacenar alguna información imprescindible como la red neuronal o la reserva de algunas secciones de memoria. Para ver la solución implementada con un mayor detalle técnico, consultar el apéndice 9.2. Más adelante se hará uso de la misma técnica para resolver el problema de la integración de la red neuronal en el microcontrolador, entre otros.

5.3.2.1 Configuración *Bluetooth* y de sensores

En esta primera etapa de configuración, se definirá el comportamiento de algunas características de la placa; en primer lugar, el de la **IMU** (*Inertial Measurement Unit*), la unidad con la que trabajamos para obtener los datos del movimiento sirviéndose de un *giroscopio* y un *acelerómetro*, y por otro lado, la configuración del ya descrito, definido y diseñado **BLE** (*Bluetooth Low Energy*) con los parámetros que se definieron en la sección 5.3.1.

Teoría 5.3.2: Por qué es necesario configurar la IMU [5]

La **IMU** es una parte fundamental de este proyecto, ya que es el dispositivo contenido en la placa que gestiona las mediciones de aceleración y velocidad del movimiento y que consta para ello de acelerómetro y giroscopio. Con la captura de estos datos, podrá rasterizarse una imagen que contenga el trazo descrito y con la que podremos alimentar la red neuronal.

Problemas 5.3.1: Librería Arduino_LSM9DS1

Uno de los problemas con esta parte del código, fue que la librería *Arduino_LSM9DS1* necesaria para poder trabajar con la **IMU**, tiene varias versiones. Al haber leído para algunos de los proyectos TFLite de arduino, que era recomendable utilizar su primera versión, fue la elegida. Sin embargo esta primera versión, no posee una de las funciones necesarias para trabajar con la **IMU** en nuestro caso, que es el llenado contínuo de la FIFO de lectura de medidas recogidas, que por defecto funciona en *oneShotMode*, es decir, llenado a ráfagas. Es necesario poder disponer de la función para trabajar en tiempo real con la predicción de letras y también es imprescindible para la recolección de muestras, como se verá más adelante.
Por lo que la solución es o bien añadir manualmente la función en la librería, o bien actualizarla a la versión *1.1.0*.

Por lo que solo resta fijar los parámetros creados para **BLE**, vincular sus señales con manejadores de los eventos y hacer lo propio con la **IMU**.

Cabe destacar que adicional a la configuración **BLE** descrita, también se incorpora otro servicio con la característica *strokeCharacteristic*, menos interesante de explicar, ya que utilizaré el método de recolección de muestras de *Pete Warden* para uno de los ejemplos *TFLite de Arduino*: *magic_wand* [38].

5.3.2.2 Configuración para la red neuronal

Parte de máxima trascendencia, ya que, es imprescindible configurar de manera adecuada los parámetros del modelo para que el reconocimiento se de manera óptima.
Una vez obtenemos el modelo definido en *deep_pen_model_data.cpp*, proceso que se explicará en la sección 6.2.6.

Se deben establecer las micro-operaciones que se darán en el modelo para tener definido el repertorio en tiempo de ejecución que utilizará nuestro interprete [34]. Existe una alternativa que añade todas las micro-operaciones de forma genérica, a costa de un mayor uso de memoria. Para mayor profundización de la implementación, consultar el apéndice 9.4.

Problemas 5.3.2: Al cargar el firmware la placa deja de ser detectada

En las primeras cargas del firmware experimentando con el modelo, la placa dejaba de ser detectada. Lo primero que pensé es que el bootloader se había bloqueado. Sin embargo al restaurar la placa manualmente (Pulsación del botón reset justo al conectar la placa), el 'L' led de la placa, comenzó a parpadear; indicativo de que la placa se había restaurado. Por tanto solo cabía que el programa cargado era erróneo. Dado que tanto la compilación como la ejecución no informaban de errores, fue complicado dar con que este error se debía a una mala configuración de las micro-operaciones definidas.
Ya que al hacer cambios en el diseño del modelo, es imperativo añadir las microoperaciones ampliadas. Un error de principiante que llevó mucho tiempo arreglar.

Para finalizar la configuración de *TensorFlow Lite* definimos el interprete del modelo, que hará uso del repertorio de micro-operaciones especificadas.
Y por último, inicializamos los led pins como salida, para poder hacer uso del led como indicativo de estado.

5.3.3 Función cíclica del firmware [2,38]

En esta función (denominada 'loop()' en el diseño *Arduino*), será donde se establezca el código que se ejecutará persistentemente mientras la placa esté alimentada.

En primer lugar, se tratará la lógica para los leds es simple:

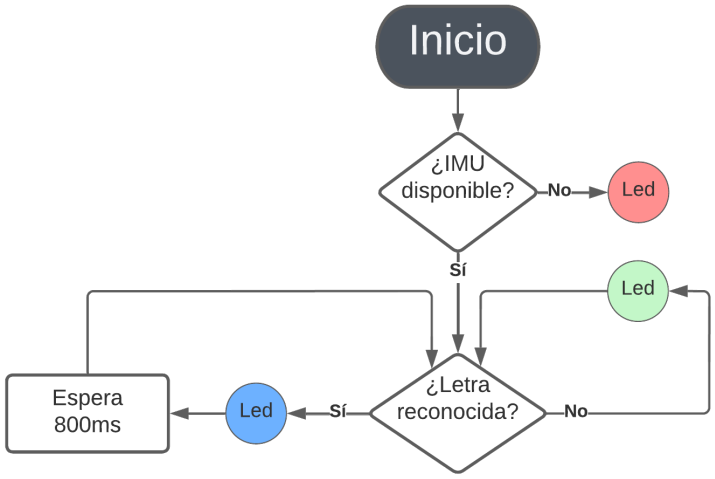


Figura 5.2: Diagrama de flujo de leds

En estado idle el led encendido será el verde y en estado de detección de letra, será azul (quedará en este estado durante 800ms, tiempo durante el que no se detectará nada, para que el usuario pueda reposicionar su postura para volver a escribir otra letra); en caso de que la **IMU** no esté disponible, se encenderá el led rojo.

Para notificar una vez la conexión a un dispositivo, pese a que la ejecución es cíclica, se implementa una sencilla solución descrita en el apéndice 9.5.

En cuanto al registro de movimiento, se han utilizado algunas de las funciones del trabajo de *Pete Warden* y su *magic_wand* [38] a modo de librería para mi propio proyecto y así aligerar el desarrollo de la recolección de trazos y su rasterización.

Durante este proceso, se hará uso del giroscopio para determinar los cambios del trazo y el acelerometro para pequeños ajustes sobre los resultados obtenidos con el giroscopio, por ejemplo cálculos de gravedad y parámetros de velocidad del cambio de trayectoria del trazo. Gracias a contar con las funciones antes citadas, tomar los valores de los sensores es tan fácil como llamar a la función *ReadAccelerometerAndGyroscope*. Con los valores leídos, se hace un pequeño arreglo de estimación de desvío del giroscopio y se envían los valores leídos como característica del servicio para *Data Collector*, donde se recogeran las muestras para entrenar la red neuronal. Y con el trazo completamente construido y corregido, se rasteriza el movimiento para obtener la imagen que pasará por el modelo.

Con todo lo anterior definido, todo está listo para dar comienzo con el procesamiento en la red neuronal. Para llamar a su ejecución, se invoca el interprete, que arrojará los resultados en un puntero anteriormente definido. Este puntero de salida, contiene los datos asociados al tensor, es decir, el producto de que el trazo rasterizado haya pasado por la red neuronal. En nuestro caso, lo que se obtiene de la red neuronal, es una valoración de ajuste de afinidad del trazo rasterizado a lo que el modelo ha sido entrenado para reconocer como letras (nuestros *labels*). Sintetizando, lo que se obtiene como salida de la red neuronal, es una valoración de semejanza a cada letra, codificada como un índice.

Por tanto, lo que resta es trivial, solo tenemos que obtener el *label* con mayor valoración. Y como consecuencia, la letra que el modelo ha estimado más posible respecto a su entrenamiento.

Obtenida la letra, es enviada al puerto serie, para cuando se trabaje con conexión física; y se introduce en la característica de transferencia(tx) para cuando se trabaje con el servicio *Bluetooth* (*BLE*).

Como comprobación concluyente, se verifica si la característica de lectura (rx) ha sido escrita por el programa de usuario, suponiendo esto una señal de que ya ha sido leída la letra actual en tx y como consecuencia, haciendo que se restaure su valor a uno por defecto; ya que de no hacerlo, la letra permanecería inmutable en la característica y el programa de usuario leería las mismas letras reiteradamente hasta escribir otras. Esto se debe a que el las *características* en *BLE* funcionan con valores constantes hasta que se modifique el estado actual.

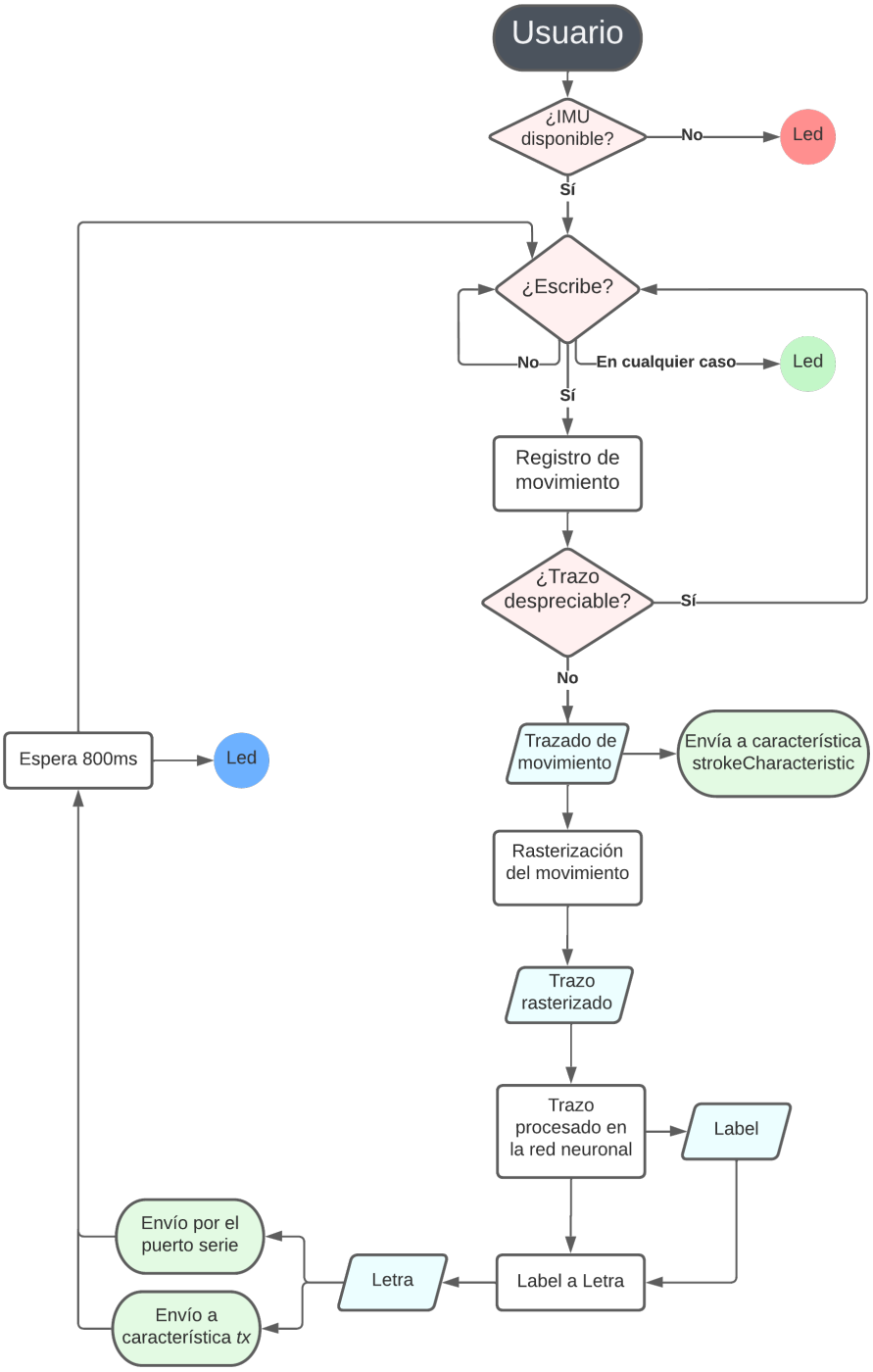


Figura 5.3: Diagrama de flujo simplificado del firmware del microcontrolador

Capítulo 6

Red Neuronal

6.1 Planificación

6.1.1 Elección de framework y librerías

En esta sección se defenderá la elección de las herramientas utilizadas para el desarrollo de la red neuronal, resultando una ampliación de la *sección 5.1.3*, donde se tratará más la materia en términos del modelo apartándonos de su integración en el microcontrolador.

TensorFlow Lite se utilizará para la creación del modelo optimizado para microcontroladores, pero esto no perturba en absoluto el desarrollo ordinario en *TensorFlow*. Es decir, se trabajará de igual forma que si la red neuronal se utilizara en un equipo convencional, con la única salvedad de la conversión del modelo ya generado a una versión optimizada para microcontroladores; así es la dinámica de trabajo con *TinyML*.

Como complemento a *TensorFlow* y para facilitar el desarrollo de la red neuronal, se empleará *Keras*, una *API* de alto nivel que, desde hace unos años, pertenece a *TensorFlow*; sirviendo para actuar como interfaz a nivel de capa para *TensorFlow*, o expuesto de otra forma, facilita el desarrollo al simplificar el trabajo con las capas del modelo. Además también se utilizará para el estudio de los resultados del entrenamiento.

Como entorno de desarrollo, se utilizará *Google Colab*, se podrían emplear alternativas que se ejecutan localmente como *PyCharm*, *Jupyter* o *Eclipse*; sin embargo con *Google Colab* podemos aprovechar la capacidad de cómputo de sus *GPU*s, lo cual agiliza mucho los tiempos de *entrenamiento* y *validación*. Si bien, el tiempo de uso del que se dispone de estas *GPU*s, es limitado, aunque siempre podemos continuar haciendo uso de las *CPU*s. Además, al tratarse de una alternativa en la nube, la etapa de configuración es mínima: apenas instalando las librerías de las que vamos a hacer uso, ya tenemos desplegado el entorno de trabajo.

6.1.2 Diseño de la red neuronal

El diseño de un modelo es determinante para que se desenvuelva apropiadamente a la hora de realizar su función. Por más que se dote al modelo durante el entrenamiento de un gran volumen de datos, será vano si el diseño no está enfocado a la labor que tiene que desempeñar. Por lo tanto, para el diseño de este modelo, se han estudiado otros muchos de clasificación de formas y procesamiento de imagen; la mayoría hacían uso del célebre *MNIST*: una base de datos que cuenta con 60.000 imágenes de entrenamiento y 10.000 de testeo, es un gran dataset de imágenes de dígitos. Aunque en general se han consultado todo tipo de modelos para el procesamiento de imágenes y clasificación, destacando el referente de este proyecto *Magic Wand* [38] de *Pete Warden*. Para examinar con mayor profundidad la experimentación respecto al diseño del modelo, obsérvese el *apéndice 9.7.1*.

El modelo que mejor funciona y más se utiliza en trabajos simples de procesamiento de imagen para clasificación, es el de *redes neuronales convolucionales* (*CNN*); definidas en una estructura muy marcada como ya se describió en la *figura 2.2*.

Teoría 6.1.1: Redes neuronales convolucionales

Estas *redes neuronales convolucionales* no son más que un tipo de red neuronal que, reduciendo a un nivel muy básico, utiliza un tipo de capa que realiza una operación matemática llamada convolución.
Lo cual nos lleva al siguiente paso, la definición de convolución: operador matemático que haciendo uso de dos funciones f y g, genera una nueva a partir de estas, que representa la magnitud de su superposición. Concretamente en 2 dimensiones, podemos entenderlo, tal y como ilustra la *figura 6.1* como el producto del *kernel* o *filtro* y una subventana de la matriz, generalmente una imagen; al repetir este producto para todas las subventanas, obtenemos una nueva matriz resultado de la convolución.
Encontrar los valores del *kernel* y por tanto la magnitud resultante del análisis de la imagen de entrada, será la principal labor de la capa de convolución.
Al resultado de la convolución, se le denomina *mapa de características*, y su función es evidenciar dónde se encuentra la característica buscada por el *kernel*. Estos *mapas de características* pueden reflejar cambios de contraste, texturas, superficies planas, etc.
Pues bien, la red neuronal realizará esta operación secuencialmente, es decir, el resultado de una capa, alimentará a la siguiente. Lo cual provoca que, unido a un proceso de *agrupación*, cada vez la información relevante se condense y refine más y más, permitiendo extraer patrones cada vez más complejos.

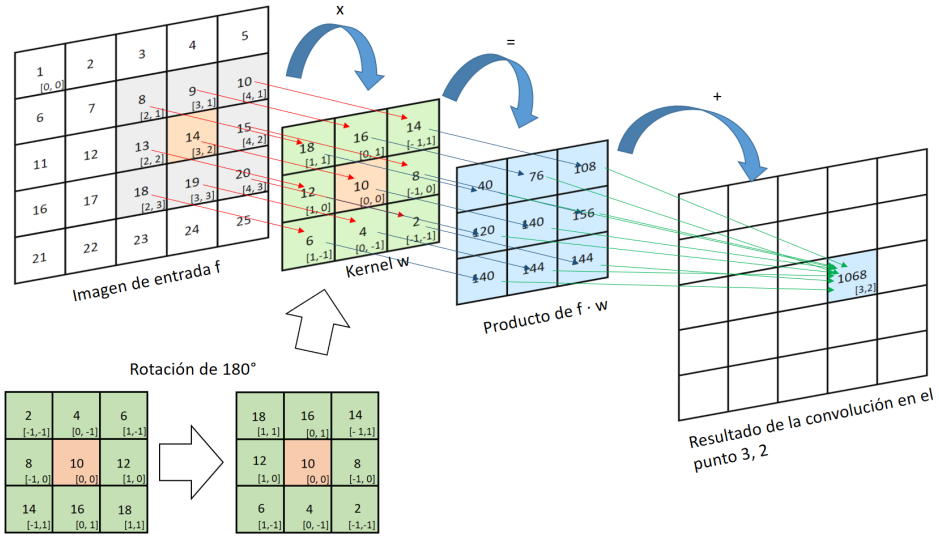


Figura 6.1: Esquema de convolución 2D (https://bryanmed.github.io/conv2d/)

Conociendo la teoría de la *red neuronal convolucional*, ya es posible comentar el diseño concreto implementado. El modelo cuenta con una estructura de *Sequential* de *Keras*; gracias a esta utilidad, es posible añadir capas al modelo de forma cómoda, además de permitir el acceso a algunas funciones para el entrenamiento. Se describirá la estructura en términos generales, para encontrar una descripción a nivel de implementación real con *Keras*, véase el *apéndice 9.6*

El modelo utilizado va a contar con tres bloques de procesamiento tras la entrada, formados por una capa de *convolución*, otra de *normalización*, *activación* y *fropout*. Estos bloques deben su composición a que el aporte de cada una de las capas, complementa al procesamiento convolucional, en el caso de nuestro modelo, que trabaja con imágenes tan pequeñas, son prácticamente irremplazables. Comenzando por la inherente capa de *convolución*, donde se dará el procesamiento anteriormente ilustrado. Tras esta, se normaliza la salida mediante una capa para este propósito, lo cual garantiza una mayor estabilidad y eficiencia en el proceso de aprendizaje. Se define como función de activación, *relu*; no por otra cosa que porque es la más utilizada, la que mejor funciona para prácticamente todas las labores sin tener que profundizar en el estudio del mismo y por su bajo coste computacional; destacable debido a la naturaleza del dispositivo en el que se ejecutará la red neuronal. Y por último la capa de *Dropout* para mitigar el overfitting, un problema que se ha podido experimentar debido a que es una red neuronal de reducida complejidad.

En cada uno de los tres bloques, lo único que varía es el número de filtros de la capa de convolución, duplicándose respecto al anterior.

Comentada la estructuración de estos bloques y su fundamento, ya es posible desarrollar el modelo completo. Al inicio, preliminar a los tres bloques citados, hay una capa de *reescalado*, que va a servir meramente para normalizar los valores los píxeles de las imágenes a una escala [0,1]. Tras esta, se encuentran

los tres bloques descritos y posterior a estos, una capa de *agrupación* por simple coherencia estructural debido a que reduce la dimensionalidad pero conserva la mayor parte de información relevante(utilizada en la práctica totalidad de *CNN*s). Otra capa *dropout* por el mismo motivo que en los bloques y finalmente la una capa densa de softmax, obtenemos una distribución de probabilidad, para obtener un output clasificatorio.

Figura 6.2: Esquema de la red neuronal diseñada

6.2 Implementación

6.2.1 Preparativos

Previo al diseño, se han de instalar ciertas dependencias y definir algunos parámetros asistentes para el código. La mayoría de dependencias son para trabajar con estructuras de datos en python, interfaces con el OS, o librerías para *TensorFlow* y *Keras*. También se ha hecho uso de algunas funciones del trabajo de *Pete Warden* [38].

Otra dependencia imprescindible es *xxd*.

Teoría 6.2.1: Uso de *xxd* [32]

Es esencial contar con *xxd*, esto se debe a que la mayoría de microcontroladores no tienen soporte para sistema de archivos nativo. Con *xxd* obtenemos el modelo en formato matriz de chars directamente compatible con **C**/**C++** e integrable en cualquier microcontrolador.
Es recomendable establecer esta matriz como constante por cuestiones de eficiencia en el acceso a memoria.

Tambien necesitaremos el dataset para el entrenamiento, validación y testeo del modelo, que se irá actualizando a medida que se toman más muestras y que se podrá descargar desde el *Google Drive* institucional. Se acompaña la descarga de una comprobación de existencia del fichero descargado para concluir esta sección.

Con el dataset descargado, se almacenan los trazos en un array, en el que cada elemento del array será un trazo; de momento de todos los labels.

Los trazos en este momento están cargados como conjunto de coordenadas que conforman, unas seguidas de otras, un recorrido; presumiblemente una letra. Pero como ya es sabido, nuestro modelo necesita como entrada imágenes; por lo que el siguiente paso es rasterizar los trazos para producir una imagen.

Al igual que en el firmware del microcontrolador, se hará uso de la función de rasterización creada por *Pete Warden* [38]. Definidas las funciones de rasterización, es posible rasterizar todos los trazos del dataset y destinarlos a las distintas fases del desarrollo del modelo como se verá en las siguientes secciones.

Hablar de bloque 'PREPARE DATASETS'

6.2.2 Entrenamiento

El entrenamiento del modelo será el segundo puntal que, junto con el diseño del propio modelo, dotará de estabilidad a nuestra red neuronal; siendo ambas dos determinantes para que esta responda de forma óptima a la tarea para la que a la que ha sido dispuesta. Por lo que la configuración del entrenamiento debe ser lo mejor posible, ajustando *epochs* (iteraciones en el proceso de entrenamiento), *learning_rate* (escalabilidad del aprendizaje), *optimizadores*, etc. Para ver el ajuste experimental de estos valores, véase el *apéndice 9.7.1*.

Las imágenes de los datasets ocuparán 32x32 píxeles. Este es otro parámentro que puede ser estudiado, no obstante, estas dimensiones han sido las que han arrojado mejores resultados de forma homogénea. Como de costumbre, para más información a este respecto, consultar el *apéndice 9.7.2*.

Se utilizará el mismo conjunto de datos aleatoriamente distribuido para cada uno de los tres dataset. Cada dataset contará con un porcentaje del conjunto de datos total. Estos porcentajes han sido estudiados y a priori no suponen extrema relevancia más allá de que el de entrenamiento debe ser ampliamente mayor. Ha sido fijado un 10% para test, otro 10% para validación y el restante 80% para entrenamiento.

Teoría 6.2.2: Uso del dataset en *Deep Learing* (1) [24]

Los tres datasets que se usan para *Deep Learing* son:

- **Validation**
  En *Deep Learning* se usan datos de validación para corroborar durante el entrenamiento, que el ajuste se está dando de forma óptima.
  Dilatando un poco más esta sencilla explicación, el *validation dataset* es un conjunto de datos imperativamente distinto del *training dataset*, que sirve para estimar la eficacia de la red en tiempo de entrenamiento.
  En general se suele usar la validación para hacer estudios del ajuste del modelo, para evitar sobreajustes (*overfitting*) y subajustes (*underfitting*).

  Teoría 6.2.3: *Overfitting* y *Underfitting*

  - ***Overfitting***
    Es un fenómeno que se da cuando el modelo reconoce peculiaridades demasiado específicas como distintivo para la evaluación. Estas peculiaridades no serían los rasgos o características que constituyen a los elementos que estudiados y por lo tanto se produce un sobreajuste; un ajuste por encima de lo óptimo.
  - ***Underfitting***
    Término análogo y opuesto al anterior, el ajuste se presentaría laxo y falto de rigurosidad; ajuste por debajo de lo óptimo.

- **Training**
  Este dataset es el más simple de entender por mera inmediación semántica. Es el conjunto de datos que se utiliza en tiempo de entrenamiento para balancear los pesos de las capas. En cada iteración de entrenamiento, se calcula la pérdida con los datos de entrenamiento introducidos y se da el ajuste de pesos en base a la pérdida. Esto supone que, cada vez la pérdida sea menor y generalmente la eficacia, o en términos más comunes a este ámbito, la *precisión* (*accuracy*), sea mayor.
- **Test**
  El conjunto de datos que se utilizará posterior al entrenamiento, para validar la efectividad del entrenamiento. Es el dataset con el que se pone a prueba el modelo entrenado.

6.2.3 Testeo

El testeo del modelo es útil para garantizar que funciona correctamente, sin embargo podemos obtener valores muy buenos sin resultar en un funcionamiento adecuado, debido al ya mencionado *overfitting* (*Teoría 6.2.2*). Por tanto solo debemos tomarlo como una herramienta más, siempre evaluando adecuadamente los resultados.

La fase de testeo consiste en simular lo que será una ejecución habitual, pero conociendo los inputs de la red neuronal, que serán del dataset de testeo. Obteniendo imágenes de cada letra (*label*), se evalúa la clasificación de la red neuronal de la misma, junto con la precisión de estimación; si la clasficación coincide y además lo hace con valores de precisión próximos a 1, significará, en principio, un buen desempeño de la red neuronal para esta letra. El objetivo es conseguir esto para todas las letras.

Es importante separar los dataset de testeo de los de entrenamiento y validación, ya que si se emplearan los mismos, el modelo ha sido entrenado con este mismo input, por tanto siempre se obtendrían buenos resultados y el testeo perdería validez.

Para un mayor acercamiento a la experimentación con el testeo de los modelos probados, visite el *apéndice 9.7.2*

6.2.4 Transformación a modelo cuantizado

El hecho de cuantizar el modelo basado en *Deep Learning* cuando se plantea su integración en microcontroladores y equipos de bajo rendimiento, suele ser imperativo.

Teoría 6.2.4: Cuantización

Cuantizar es desvirtuar la naturaleza continua de un conjunto de valores continuos, restringiéndolos a un conjunto de valores discretos.

En general esta práctica viene propiciada debido a que la mayoría de microcontroladores no cuentan con *FPU* (*Floating Point Unit*). Sin embargo no es este el caso, ya que en este proyecto se escogió un microcontrolador que sí dispone de esta unidad. Aun así se ha tenido que recurrir a la cuantización del modelo, como consecuencia de su limitación de memoria, ya que el hecho de cuantizar el modelo, supone reducir su precisión y por tanto reducir la memoria que este requiere.

Sin entrar en muchos detalles, ya que es parte de la siguiente sección, el modelo generado estándar queda muy lejos de poder integrarse en el microcontrolador, gracias a la conversión a un modelo *TensorFlow Lite*, se reduce el tamaño, pero sigue sin ser suficiente, por tanto es conveniente cuantizar el modelo *TensorFlow Lite* para obtener su versión más compacta posible.

6.2.5 Comparación de modelos generados

Los modelos generados a partir de la red neuronal diseñada, son:

Modelo	Tamaño	Reducción TF	Reducción TFLite
TensorFlow	756009 Bytes	0%	-
TensorFlow Lite	140300 Bytes	81'4%	0%
TF Lite cuantizado	41136 Bytes	94'5%	70'6%

Tabla 6.1: Tabla de comparación de modelos generados

Los resultados son impresionantes en sí mismos, pero lo son aún más cuando ponemos en contexto los valores de precisión que se alcanzan para cada uno de ellos en la clasificación; variando en el peor de los casos del orden de 0,1% de *TensorFlow* a *TensorFlow Lite* y del orden de 0,2% de *TensorFlow* a *TensorFlow Lite* cuantizado. Por lo que la compactación del modelo no tiene un impacto notorio durante su ejecución.

6.2.6 Integración en el microcontrolador

Esta sección sirve ahora a modo de recopilación de las anteriores, ya que el procedimiento prácticamente ha sido expuesto.

La cuantización del modelo para optimizar el uso de memoria es el primer paso; una vez el modelo cuenta con un tamaño apto para el microcontrolador, lo que resta es solucionar el problema de la carencia de sistema de archivos necesario para el manejo de la red neuronal construida con *TensorFlow*. Para lo cual se hace uso de *xxd*, herramienta que automatiza el proceso de conversión del modelo a una estructura de datos (*vector* de *chars*), agregable al firmware del microcontrolador.

Cuando pasamos el modelo por *xxd* y obtenemos el código en *C*, es suficiente con añadirlo al código del *firmware* y este ya es funcional en el microcontrolador. Con el que se trabajará gracias a las librerías de *TensorFlow Lite*.

6.3 Generación de muestras para el dataset

Por suerte se pudo contar con una herramienta creada por *Pete Warden* [38] para tomar muestras de trazados con el microcontrolador que estamos empleando. Se ha modificado superficialmente y se han hecho algunos cambios estéticos (DataCollector.html).

La toma de muestras ha sido un proceso pesado, ya que hay un proceso de adaptación al movimiento del microcontrolador y hay que tomar muchas muestras por letra. Por tanto es un proceso cargante y lento para quien está creando las muestras y para el que tiene que supervisarlas.

A lo largo de la toma de muestras se han detectado ciertos fallos que se han resuelto y son consultables en los *apéndices 9.14, 9.12, 9.13*

Capítulo 7

Interfaz de usuario

7.1 Planificación

7.1.1 Elección de framework para interfaz gráfica

Las alternativas con las que vamos a partir tras documentarme sobre frameworks para desarrollo de interfaces gráficas, preferiblemente multiplataforma, y sumarlas a las que ya conozco:

Alternativas	Documentación	Usada anteriormente	Comunidad	Soporte para BLE	Lectura P.Serie nativa	Multiplataforma
Flutter	✓	✓	✓	✓	✓	✓
GTK+	✓	-	✓	✗	✗	✓
QT	✓	✓	✓	✓	✓	✓
PyQT	✓	✗	✓	✓	✓	✓
Electron	✓	✗	✓	✗	✓	✓

Electron y *GTK* quedan descartados porque, si bien existen procedimientos para poder gestionar *Bluetooth Low Energy* con estos, no tienen soporte nativo ni librerías para ello, por tanto, teniendo la posibilidad de hacer uso de otros frameworks que faciliten esta tarea, es factible descartar estas opciones.

Ante *QT (C++)* y *PyQT*, debido a que se ha trabajado anteriormente con *QT (C++)* y el tiempo de desarrollo disponible es un factor limitante, es razonable excluir *PyQT*.

Solo quedarían *QT* y *Flutter*, se ha escogido ante estas dos posibilidades *QT* por dos razones. La primera razón es que *Flutter* está más dirigido a interfaces móviles. Y la segunda razón es que pese a haber trabajado con ambas, a título personal, me encuentro más habituado a *QT*.

Por lo que el framework seleccionado para realizar la interfaz de usuario, será *QT* porque ofrece soporte para las tecnologías que en principio van a utilizarse y porque es una herramienta con la que ya se ha trabajado y esto resulta en un menor tiempo de documentación, causa de peso por el ajustado tiempo del que se dispone.

7.1.2 Raison d'être

Antes de comenzar con el diseño, debe ponerse en valor la conveniencia de esta interfaz de usuario. El principal motivo es que lo que se busca con este proyecto es crear un producto real, y como tal, no podemos valernos de un simple lector de puerto serie para el dispositivo, como podría ser el integrado en el *Arduino IDE*. Además, esta interfaz no solo está creada con la funcionalidad de este proyecto en mente de servir de mediador con el usuario para la lectura de las letras escritas con el *DeepPen*, sino que busca ser el nexo de unión de todas las funcionalidades que se implementen para este. Durante el propio desarrollo de esta interfaz, se ha reflexionado sobre ciertas funcionalidades, como la de un modo pizarra donde reflejar el trazado íntegro que se realice, y que se ha propuesto finalmente como una meta secundaria.

7.1.3 Diseño de la interfaz

Para el bocetado de la interfaz previo a la implementación, se hará uso de *QT Design Studio*, herramienta de *QT* para diseñar interfaces. En el bocetado de la interfaz se fijarán los elementos que la formarán y su disposición procediendo razonadamente. Para entender visualmente lo que se va a describir, puede examinar el *apéndice 9.15*.

Como se ha razonado en la sección 7.1.2 anterior, queremos que este programa sea el *hub* de funcionalidades para el *DeepPen*, por lo que se hará uso de una barra lateral para acceder a estas. Queremos que el protagonismo se encuentre en la funcionalidad, por lo que esta barra lateral será desplegable, para reducir el tamaño que esta ocupe en la interfaz. Alojará las funcionalidades y, dada la naturaleza académica del proyecto, una sección sobre el desarrollador, para más información sobre el trabajo.

Cada sección de la interfaz, se mostrará en la pantalla central, quedando las barras superior y lateral, constantemente a la vista.

Es necesario un indicador de conexión y un botón de conexión inalámbrica, que se ubicarán a la derecha en la barra superior para evitar sobrecargar la zona izquierda de la interfaz.

En los primeros bocetos se evidenciaba un vacío en la parte central de la barra superior y también se echaba en falta saber qué pantalla se estaba mostrando en cada momento, por lo que uniendo ambas carencias, se solventó disponiendo un indicador de pantalla en dicha región.

7.1.4 Diseño del software

7.2 Implementación

Capítulo 8

Encapsulado

Herramientas de diseño
Decisiones de diseño
Problemas en la impresión
Duración de batería de 2 días

Capítulo 9

Apéndice

9.1 Valores nulos al leer por *Bluetooth* con *QT*

Este fue un error complejo de depurar como suele ser habitual con los problemas derivados de versiones de librerías. No se leía ningún valor de las *características* del dispositivo pese a estar detectándolas y estar correctamente conectado, se debía a dos factores.

El primero estar haciendo uso de una librería anterior a la documentación con la que estaba trabajando (librería de QLowEnergyService). Hay grandes diferencias en el comportamiento de algunos métodos de esta librería de las versiones 5.x a la 6.x, aunque estas no provocan errores, sí que provocan que el el código no funcione como se esperaría (*Enums* con valores diferentes o inexistentes, etc).

En segundo lugar se estaba llamando a un método cuando todavía no se había recibido la *característica*. Por tanto esta aparentaba estar bien registrada, ya que podía obtenerse su *Uuid*, pero no contenía ningún valor. Se estaba leyendo el valor en *connectToService()*, cuando debería hacerse en *serviceDetailsDiscovered()*.

9.2 Resolver problemas de memoria

A lo largo del desarrollo del firmware, será necesaria la gestión de áreas de memoria reservadas para ciertas funcionalidades, por ejemplo en el bloque de configuración del firmware, *setup()*; será necesario reservar un área en memoria para las labores de E/S y memoria intermedia de las acciones con tensor.

Para ello se contará con la propia *flash* del dispositivo a modo de región de almacenamiento. Creando arrays en la propia memoria *flash* que harán las veces de mecanismo de almacenamiento como se expone en el siguiente fragmento de código:

Fragmento de *deep_pen.ino*

```
62    /********************** SETUP FUNCTION **********************/
63    // Setting the area of memory reserved to tensor input-output actions.
64    constexpr int TensorAreaSize = 30 * 1024;
65    uint8_t tensor_arena[TensorAreaSize];
```

9.3 Instalación de librerías en *Arduino IDE*

La instalación de librerias es muy sencilla haciendo uso del *Arduino IDE*. En el menú: *Tools->Manage Libraries...* y se abrirá una ventana donde podemos gestionar las librerías instaladas, instalar otras versiones e instalar nuevas librerías.

9.4 Definición de micro-operaciones en el firmware del microcontrolador

Las micro-operaciones que se definirán en el firmware que se emplearán en la red neuronal son:

- Conv2: Para el procesamiento de la capa homónima.
- Mean: Para promedios como el que se necesita en la capa *GlobalAverage-Pooling2D*
- FullyConnected: Para las capas densas, entre otros.
- SoftMax: Como función de activación.

Como alternativa más cómoda, pero a costa de un mayor uso de memoria, del cual no podemos abusar dada la naturaleza de nuestro dispositivo; es plausible usar *tflite::AllOpsResolver*, que cargará todas las operaciones disponibles para *TFLite*.

9.5 Notificación de conexión *Bluetooth* del firmware del microcontrolador

Para solucionar el problema de la notificación de conexión *Bluetooth* se crea una pequeña estructura para consultar el estado de conexión de la iteración anterior (*last_connection*). De esta forma es posible mantener consistencia en la conexión pese a estar dentro de un bucle.

9.6 Descripción de capas *Keras* empleadas para la implementación de la red neuronal

Las capas utilizadas durante la implementación del modelo, son las siguientes [18]:

- Rescaling: Como resultado de esta capa, se consigue reescalar o normalizar los valores de los inputs, en nuestro caso las imágenes. Con esto definimos una escala uniforme y es un procedimiento típico en procesamiento de imagen: *image normalization*, donde se normalizará el valor de los píxeles de las imágenes a una escala [0,1].
- Conv2D: Capa para procesamiento convolucional, es la capa que realmente procesará dentro del modelo. Se crea un kernel que convoluciona con la entrada de la capa. Se define la capa en base ciertos parámetros que determinarán su funcionamiento y complejidad, siendo en nuestro caso los parámetros:
  - filters: el número de filtros de salida de la convolución.
  - kernel_size: el tamaño de ventana de convolución, cuando se especifica un solo número, se interpreta una ventana cuadrada de ese tamaño.
  - strides: define el tamaño de los tramos de salto de la ventana de convolución.
  - input_shape: establecer el tamaño de la entrada. En nuestro caso imágenes de 32x32.
- BatchNormalization: Normaliza sus entradas por lotes. Aplica transformaciones que conservan la media de la salida cercana a 0 y la desviación estándar a 1.
- Activation: aplica una función de activación a una salida. Las funciones de activación son las que arbitran la activación de las neuronas de la red neuronal y por ello, repercute en su salida. Existen diversas funciones de activación, como lo son *rectified linear unit(relu)*, *sigmoid*, *softmax*(función de distribución de probabilidad), etc.
- Dropout: Capa que introduce cierta entropía, de forma que se descarta la contribución de ciertas neuronas de forma estadística. Se suelen implementar para soslayar el *overfitting*.
- GlobalAveragePooling2D: Esta capa tomará un tensor de dimensión $x*y*z$ y calculando el valor medio de los valores x e y, producirá una salida basada en z elementos.
- Dense: Es una capa común de red neuronal, solo que está *densamente* conectada, es decir, cada neurona de esta capa está conectada a todas las neuronas de la capa anterior. Se usa, como es nuestro caso, en redes clasificatorias.

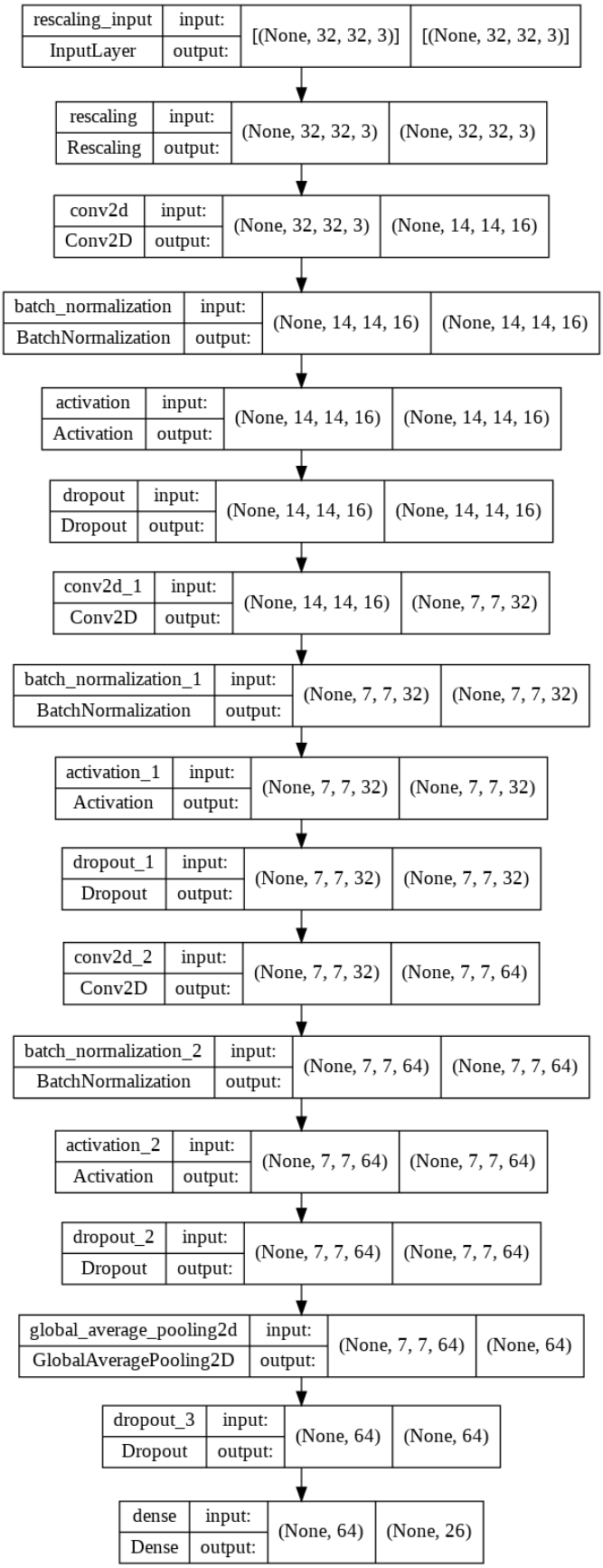


Figura 9.1: Estructura del modelo generado en *TensorFlow*

La anterior *figura 9.1* es consecuencia del siguiente código:

Fragmento de *Train.ipynb*

```
############### MAKING THE MODEL ###############

def make_model(input_shape, num_classes):
  model = models.Sequential()

  # Rescaling
  model.add( layers.Rescaling(1.0 / 255) )
  # Block 1
  model.add( layers.Conv2D(16, 5, strides=2, input_shape=input_shape) )
  model.add( layers.BatchNormalization() )
  model.add( layers.Activation("relu") )
  model.add( layers.Dropout(0.5) )
  # Block 2
  model.add( layers.Conv2D(32, 5, strides=2, padding="same") )
  model.add( layers.BatchNormalization() )
  model.add( layers.Activation("relu") )
  model.add( layers.Dropout(0.5) )
  # Block 3
  model.add( layers.Conv2D(64, 3, strides=1, padding="same") )
  model.add( layers.BatchNormalization() )
  model.add( layers.Activation("relu") )
  model.add( layers.Dropout(0.5) )
  # Pooling + another Dropout
  model.add( layers.GlobalAveragePooling2D() )
  model.add( layers.Dropout(0.5) )
  # Softmax
  model.add( layers.Dense(num_classes, activation="softmax") )


  return model

model = make_model(input_shape=(IMAGE_WIDTH, IMAGE_HEIGHT, 3),
                   num_classes=26)
model.build(input_shape=(None, IMAGE_WIDTH, IMAGE_HEIGHT, 3))
model.summary()
keras.utils.plot_model(model, show_shapes=True)
```

9.7 Experimentación red neuronal

9.7.1 Diseño de la red neuronal

Hablar sobre los cambios probados en la red neuronal y sus resultados. Carpeta Train->Análisis

Cambiar el tamaño del kernel de las capas Conv2D de 3 a 4, resulta muy efectivo, a costa, evidentemente, de aumentar el tamaño que ocupa el modelo.
Al pasar de 4 a 5, el modelo arroja unos datos de efectividad teóricos extre-

madamente buenos, alcanzando cifras de precisión mucho más altas con menos epochs. En general, podemos extrapolar que, a mayor tamaño del kernel, mejor precisión pero mayor tamaño del modelo. En nuestro caso no llega a ser un problema, ya que, aunque estamos usando un dispositivo de memoria limitada, no llega a ocuparse toda la memoria del mismo, al menos por ahora.

Cada uno de los modelos o cambios con los que se ha experimentado, son pequeñas iteraciones o pequeñas variaciones respecto del modelo base. Creado a partir del estudio de otros modelos para tareas parecidas.

Una de las abundantes variables con las que experimentar, es el tamaño de las imágenes de entrada para la red neuronal. Se ha percibido una notoria mejora en el reconocimiento con letras complejas, como por ejemplo la 'k', a medida que las dimensiones aumentan. De forma análoga, con letras simples, como por ejemplo la 'c', a razón de una menor resolución, mejores resultados se obtenían. Esto cuadra con lo esperable, ya que las letras complejas necesitan de un análisis más preciso para obtener mejores predicciones que el resto y equivalentemente, las letras más sencillas obtienen mejores predicciones sobre el resto, cuando se analizan muestras sencillas.

9.7.2 Entrenamiento de la red neuronal

ENTRENAMIENTO: Hablar de *epochs*, *learning_rate*, *optimizadores*, etc. ENTRENAMIENTO: Resultados del testeo obtenidos de los modelos Carpeta Train->Análisis.

9.8 Permisos uso del puerto del microcontrolador (Linux)

En algunos casos, como ocurrió en el trascurso de este trabajo, si queremos acceder al puerto serie de la placa o usar el IDE de Arduino, debemos conceder permisos al microcontrolador:

```
1  ~$ ls -l /dev/ttyACM*                    # En mi caso
2  ~$ sudo usermod -a -G dialout <usuario>
```

Como respuesta al error "*can't open device /dev/ttyACM0*".

9.9 Cambiar la orientación de la placa

Para poder hacer uso del *DeepPen* en una posición de escritura natural, vertica, hay que hacer una serie de cambios. Los mejores resultados se han conseguido cambiando en la biblioteca de los sensores, en LSM9DS1.cpp en todas los sensores:

Fragmento de *LSM9DS1.cpp* de la librería homónima

```
121 //            Original            //      Orientacion cambiada
122 x = data[0] * 4.0 / 32768.0; // z = -data[0] * 4.0 / 32768.0;
123 y = data[1] * 4.0 / 32768.0; // y =  data[1] * 4.0 / 32768.0;
124 z = data[2] * 4.0 / 32768.0; // x =  data[2] * 4.0 / 32768.0;
```

Aunque continúan sin obtenerse trazados correctos del movimiento.

9.10 Acceso al puerto serie en QT (Linux)

Hay que añadir al *.pro del proyecto QT:

```
1  QT += serialport
```

Y tras esto, añadir la librería con normalidad y acceder al puerto con el nombre, en nuestro caso, "ttyACM0".

9.11 Error de reconocimiento de imágenes en QT

Al importar imágenes en QT tras haber exportado el programa a Win o Linux para corroborar que todo continuara funcionando correctamente, las imágenes dejaron de mostrarse. Esto se debe a que QT cambia la ruta del proyecto al directorio en el que se exporta el programa. Por tanto las rutas especificadas para las imágenes, dejan de ser válidas. Para corregir este problema, basta con cambiar el '*Build directory*' del proyecto (Desde '*Projects*' en el panel de la derecha de QT creator).

9.12 El recolector de muestras elimina varias muestras al borrar una

Este fallo aparentemente se produce al borrar instancias por debajo de la última. En ocasiones, se produce un pequeño error en las comprobaciones de índices de muestras, por lo que se borran varias muestras y estas terminan con índices desordenados.

Para paliar este problema y comprobar si había el número de trazos esperado, se hacía uso de un comando en bash cada vez que se generara un *JSON*:

```
1    ~$ grep -o index <nombre_archivo>.json | wc -l
```

9.13 Asignación incorrecta de índices en el recolector de muestras

Como se ha citado en el *apéndice 9.12* anterior, al borrar muestras se borran más trazos de los debidos y eso genera una redistribución de los índices de cada muestra, resultando incorrectos.

Para resolverlo, se ha creado un script que reasigna los índices con la distribución adecuada: sort_dataset.py

9.14 Ajuste para poder utilizar el recolector de muestras de *Pete Warden* [38]

Es necesario hacer uso del recolector de muestras, utilizar *Google Chrome* y activar la flag para desarrolladores '*Experimental Web Platform features*' activa en '*chrome://flags/*'.

9.15 Capturas de la interfaz de usuario

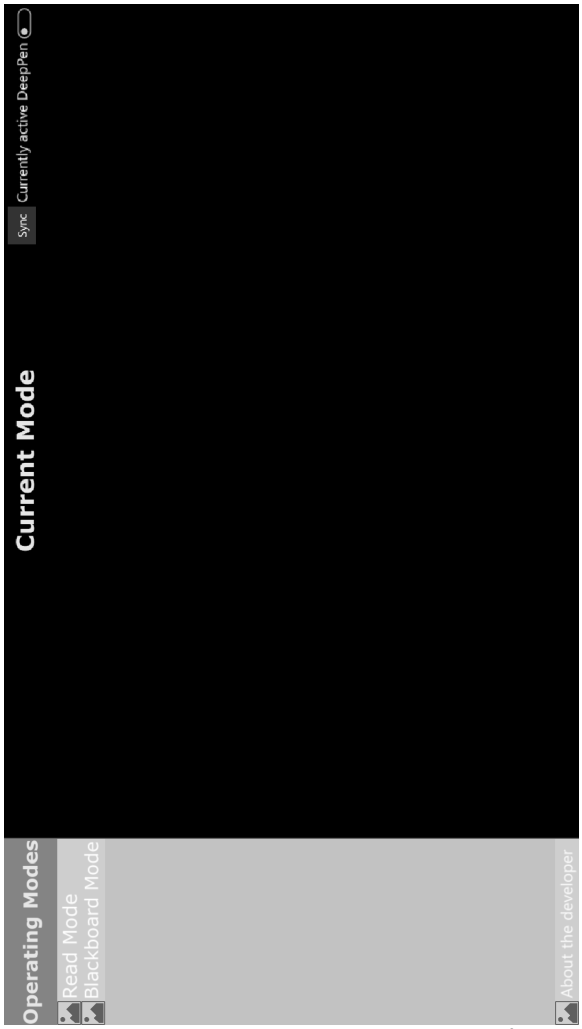


Figura 9.2: Boceto en QT Design Studio

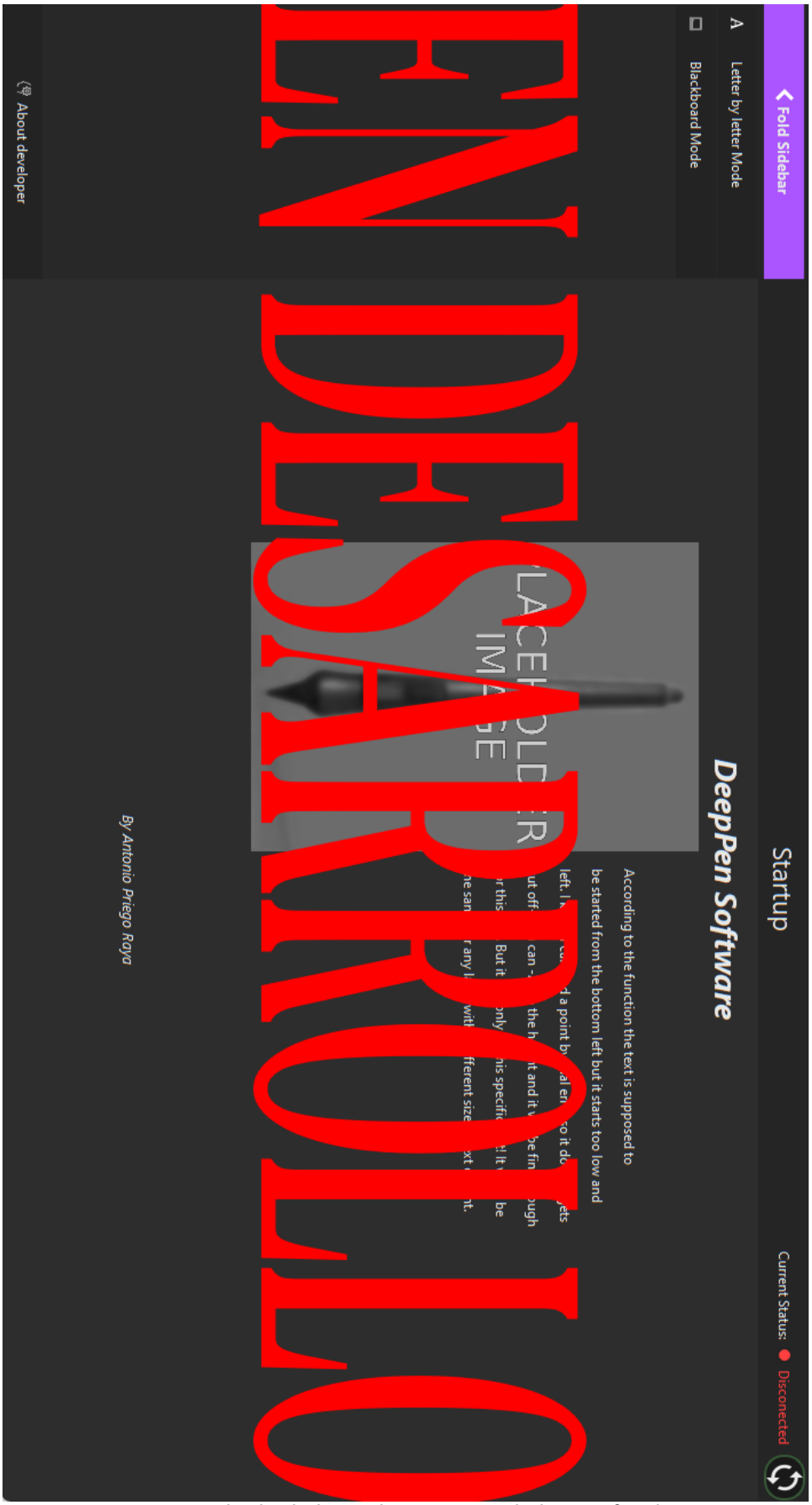


Figura 9.3: Resultado de la implementación de la interfaz de usuario

Capítulo 10

Validación

10.1 Ajuste al presupuesto

Respecto a como se estimó en la sección 3.3.2 (*Presupuesto*), se ha ajustado satisfactoriamente el coste del dispositivo, resultando todos los costes:

Descripción	Precio
Arduino Nano Sense 33 BLE	35'80$~33'82€
Pila 9V Recargable	10'99€
Adaptador pila 9V	3€
Impresión	1€
Interruptor	0'05€
Adaptador microUSB a USB	1€
Cable MicroUSB Datos	6€
TOTAL:	**55,86€**

Tabla 10.1: Costes de producción del *DeepPen*

10.2 Comprobación de objetivos cumplidos

comprobar que lo que has terminado construyendo cumple los requisitos que pusiste al principio. Debes diseñar diferentes tests y hacer experimentos que demuestren que todo ha salido según lo previsto, argumentando por qué pasa lo que pasa en función de las decisiones de diseño e implementación que has tomado

Capítulo 11

Trabajos futuros y mejoras

proponer posibles mejoras, indicando qué se podría añadir y cómo se haría/-solucionaría es algo que gusta mucho al tribunal

Buffer para guardar en el dispositivo lo escrito hasta sincronizar con el programa de usuario

Arreglar desconexiones BT cuando se corrija la biblioteca BLE de QT

Compatibilidad con windows, en teoría sería sencillo. Simplemente añadiendo 'ifdef' para linux/windows en la configuración de puertos

Capítulo 12

Conclusiones

desde un punto de vista introspectivo, cuantas qué te ha parecido el reto al que te has enfrentado, qué dificultades has encontrado, cómo las has resuelto, lo que has aprendido, etc.

Bibliografía

[1] Adarsh1001. Repositorio micro-learn. https://github.com/adarsh1001/micro-learn.

[2] Andriyadi. andriyadi magic wand repository. https://github.com/andriyadi/MagicWand-TFLite-Arduino.

[3] Developers Android. Descripción general del bluetooth de bajo consumo para android. https://developer.android.com/guide/topics/connectivity/bluetooth-le?hl=es-419.

[4] Arduino. Project hub. https://create.arduino.cc/projecthub?by=part&part_id=108462&sort=trending.

[5] Aprendiendo Arduino. Aprendiendo a manejar arduino en profundidad. https://aprendiendoarduino.wordpress.com/tag/imu/.

[6] Bluetooth. A developer's guide to bluetooth technology. https://www.bluetooth.com/blog/a-developers-guide-to-bluetooth/.

[7] Fernando Sancho Caparrini. Redes neuronales: una visión superficial. http://www.cs.us.es/~fsancho/?e=72.

[8] CloudML. Cloudml webpage. https://cloudml.io/.

[9] Antonia Creswell, Tom White, Vincent Dumoulin, Kai Arulkumaran, Biswa Sengupta, and Anil A. Bharath. Generative adversarial networks: An overview. 2018.

[10] DeepMind. Alphafold2 webpage. https://www.deepmind.com/research/highlighted-research/alphafold.

[11] DeepMind. Alphago webpage. https://www.deepmind.com/research/highlighted-research/alphago.

[12] Free Software Foundation. GNU General Public License. http://www.gnu.org/licenses/gpl.html.

[13] Hinton G.E., Osindero S., and Teh Y. Movies of the neural network generating and recognizing digits. 2006.

[14] GitHub. Github copilot webpage. https://github.com/features/copilot/.

[15] GitHub/Microsoft. Edgeml webpage. https://microsoft.github.io/EdgeML/.

[16] Aurélien Géron. *Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow*. O'Reilly, 2019.

[17] Iars.geo. Breve historia de las redes neuronales artificiales. https://steemit.com/spanish/@iars.geo/breve-historias-de-las-redes-neuronales-artificiales-articulo-1.

[18] Keras. Keras webpage. https://keras.io/.

[19] Ladvien. Repository named 'arduino_ble_sense'. https://github.com/Ladvien/arduino_ble_sense.

[20] Alex Lenail. Creación de esquemas de redes neuronales. http://alexlenail.me/NN-SVG/index.html.

[21] Ben Lutkevich. Natural language processing (nlp). https://www.techtarget.com/searchenterpriseai/definition/natural-language-processing-NLP.

[22] Maxime. What is a transformer? https://medium.com/inside-machine-learning/what-is-a-transformer-d07dd1fbec04.

[23] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. 1943.

[24] Andreas C. Müller and Sarah Guido. *Introduction to Machine Learning with Python*. O'Reilly, 2017.

[25] Na8. Breve historia de las redes neuronales artificiales. https://www.aprendemachinelearning.com/breve-historia-de-las-redes-neuronales-artificiales/.

[26] Jonathan Stephens (Nvidia). Getting started with nvidia instant nerfs. https://developer.nvidia.com/blog/getting-started-with-nvidia-instant-nerfs/.

[27] OpenAI. Openai webpage. https://openai.com/.

[28] Peltarion. Global average pooling 2d. https://peltarion.com/knowledge-center/documentation/modeling-view/build-an-ai-model/blocks/global-average-pooling-2d.

[29] QT. Documentación bluetooth le. https://doc.qt.io/qt-6/qtbluetooth-le-overview.html.

[30] Partha Pratim Ray. A review on tinyml: State-of-the-art and prospects. 2022.

[31] F. Rosenblatt. The perceptron, a perceiving and recognizing automaton project para. 1957.

[32] TensorFlow. Build convert. https://www.tensorflow.org/lite/microcontrollers/build_convert.

[33] TensorFlow. Github ejemplo magic_wand de tensorflow. https://github.com/tensorflow/tflite-micro/tree/main/tensorflow/lite/micro/examples/magic_wand.

[34] TensorFlow. Introducción a los microcontroladores por tensorflow. https://www.tensorflow.org/lite/microcontrollers/get_started_low_level.

[35] TinyML. Tinyml webpage. https://www.tinyml.org/.

[36] Kevin Townsend, Carles Cufí, Akiba, and Robert Davidson. Getting started with bluetooth low energy. https://www.oreilly.com/library/view/getting-started-with/9781491900550/ch04.html.

[37] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. 2017.

[38] Pete Warden. Pete warden magic wand repository. https://github.com/petewarden/magic_wand.

[39] Pete Warden and Daniel Situnayake. *TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-Low-Power Microcontrollers*. O'Reilly, 2019.

[40] Wikipedia. Intel mcs-48. https://en.wikipedia.org/wiki/Intel_MCS-48.

[41] Wikipedia. Microcontrolador. https://es.wikipedia.org/wiki/Microcontrolador.

[42] Wikipedia. Multilayer perceptron. https://en.wikipedia.org/wiki/Multilayer_perceptron.

[43] Orhan G. Yalçin. Image classification in 10 minutes with mnist dataset. https://towardsdatascience.com/image-classification-in-10-minutes-with-mnist-dataset-54c35b77a38d.