



UNIVERSIDAD
DE GRANADA

TRABAJO FIN DE GRADO
GRADO EN INGENIERÍA INFORMÁTICA

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

Deep Learning en sistemas empuotrados: TinyML

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



Detección de escritura mediante Deep Learning en un sistema empotrado

Deep Learning en sistemas empotrados: TinyML.

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 DÍA de Junio 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

1. Introducción y Motivación	17
2. Antecedentes y estado actual	19
2.1. Redes neuronales	19
2.2. Microcontroladores	22
2.3. Integración de redes neuronales en microcontroladores	23
3. Análisis de requisitos	25
3.1. Requisitos funcionales y no funcionales	25
3.1.1. Requisitos funcionales	25
3.1.2. Requisitos no funcionales	25
3.2. Análisis de requisitos hardware	26
3.3. Planificación y presupuestación	26
3.3.1. Planificación	26
3.3.2. Presupuesto	28
4. Microcontrolador	29
4.1. Planificación	29
4.1.1. Elección del microcontrolador	29
4.1.2. Elección de las tecnologías	30
4.2. Diseño	31
4.2.1. Estructura del sketch	31
4.2.2. Servicio BLE	32
4.3. Implementación	33
4.3.1. Configuración BLE ^[23]	34
4.3.2. Función 'setup()' ^[1,38]	34
4.3.2.1. BLE e IMU setup	35
4.3.2.2. Setup de nuestro modelo	35
4.3.3. Función 'loop()' ^[1,38]	36

5. Red Neuronal	39
5.1. Diseño	39
5.2. Implementación	41
5.2.1. Preparativos	41
5.2.2. Entrenamiento	44
5.2.3. Testeo	44
5.2.4. Transformación a modelo cuantizado	44
5.2.5. Comparación de modelos	44
5.2.6. Integración en el sketch	44
6. Interfaz de usuario	45
6.1. Diseño	45
6.2. Implementación	46
7. Herramientas	47
8. Apéndice	49
8.1. Apéndice de soluciones en implementación	49
8.1.1. Si la placa deja de ser detectada	49
8.1.2. El modelo de datos entrenado no responde [34]	49
8.1.3. Error durante el entrenamiento	50
8.1.4. Error al cambiar el tamaño de las secuencias de movimientos	51
8.1.5. Valores nulos al leer por Bluetooth QT	52
8.2. Apéndice teórico	52
9. Validación	53
10. Trabajos futuros	55
11. Conclusiones	57

Índice de figuras

2.1. <i>Perceptron</i> frente a <i>Multilayer Perceptron</i>	20
2.2. Estructura simplificada de <i>Red Neuronal Convolucional</i>	21
3.1. Diagrama <i>Gantt</i> para la planificación	27
4.1. Esquema de la estructura planteada para el servicio <i>BLE</i>	33
4.2. Diagrama de funcionamiento de los leds	37
6.1. Primer boceto en QT Design Studio	45
6.2. Segundo boceto en QT Design Studio	46
6.3. Primera implementación de la interfaz de usuario del <i>DeepPen</i>	46
8.1. Fragmento de <i>*.ino</i> de nuestro proyecto.	50
8.2. Error durante primer entrenamiento con un dataset propio.	51
8.3. Error al cambiar el tamaño de secuencia de movimientos.	51

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

Qué es Machine Learning

Qué es Deep Learning

Cómo funcionan las redes neuronales

Aprendizaje supervisado y no supervisado

Otros conceptos básicos

Pese a que es ahora, en los últimos años cuando, debido a la explosión del fenómeno 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 [28]. Ya en 1943, *Warren McCulloch* (neurofisiólogo) y *Walter Pitts* (matemático), escribieron un artículo [26] 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 contemporáneas. Sirviéndose de múltiples entradas binarias, era capaz de producir una única salida, basada ya entonces en la utilización de *pesos* (número que cuantifica la relevancia 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* [41]. 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.

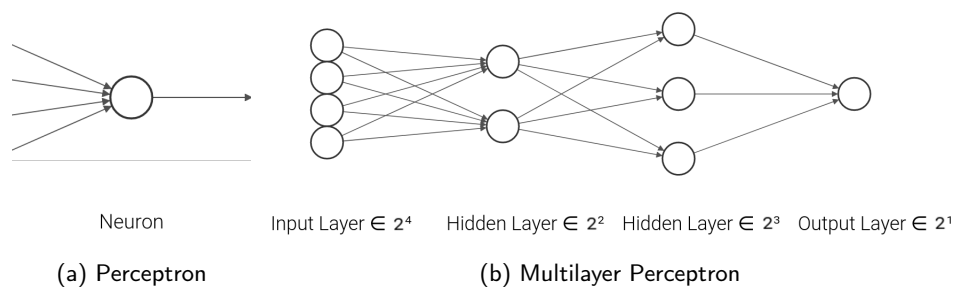


Figura 2.1: *Perceptron* frente a *Multilayer Perceptron*

Y este fue precisamente el siguiente escalón a rebasar, superado ya en la década de 1980, gracias a las *Neuronas Sigmoides Perceptron* [6], 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 *back-propagation* o la *Red Neuronal Convolutiva* (*Convolutional Neural Network*, *CNN*) [14]. 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 convolutiva, encargadas de apreciar las distintas características que servirán para la clasificación de la información de entrada.

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

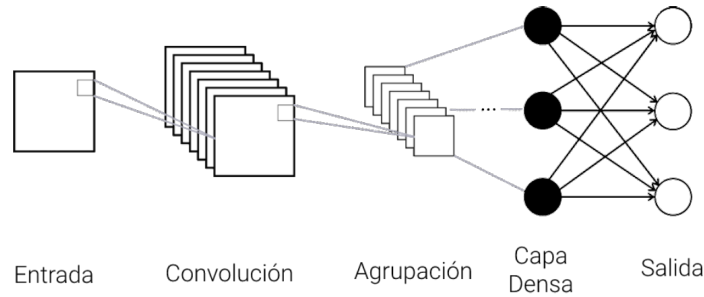


Figura 2.2: Estructura simplificada de *Red Neuronal Convolutiva*

como lo es por ejemplo el texto.

El siguiente hito llegaría a mediados de los 2000 al poder trabajar con redes neuronales profundas (*Deep Learning*), gracias a la introducción de pre-entrenamientos 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) [8], 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 propósito 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 *OpenAI* y *Microsoft*.

Algunos ejemplos de estos proyectos son: el tan sonado algoritmo de *AlphaGo* [10] 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* [9] 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* [29] de *OpenAI*, 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* [11] de *OpenAI* 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* [32] de *Nvidia*, capaz de generar composiciones 3D a partir de imágenes fijas; o por finalizar esta interminable lista de apasionantes ejemplos, el reciente *DALL.E* [29] 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* [24] 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* [25], 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

Desde el 8051 hasta los Arduino han pasado muchas cosas. Deberías añadir la evolución hasta los microcontroladores actuales, que son de 32 bits y no tienen las limitaciones de memoria y capacidad de cómputo de antaño. También han cambiado los periféricos típicos, que antes eran conversores AD y DA, PWM y poco más, y ahora tenemos transceptores 802.15.4, bluetooth, wifi, etc. Vamos, que son cacharros altamente conectados, mientras que antes esto no tenía sentido (entre otras cosas no existían las redes inalámbricas). Busca por internet algo que te cuente batallitas sobre la evolución de los microcontroladores, ok?

Los microcontroladores son sistemas de dimensiones reducidas 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 una alteración 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* [40], albergando en un mismo circuito el propio microprocesador, memoria ROM, memoria RAM y el propio reloj del sistema.

Como respuesta, *Intel* comercializó en 1977 su propio sistema para aplicaciones de control, el *Intel 8048* [39], 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* [40] 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*.

Gracias a la inclusión de estas dos últimas memorias en diseño estandarizado de los microcontroladores, el precio comenzó a ser cada vez más asequible hasta llegar a los precios actuales. ¿Incluyo el problema de la falta de silicio por la pandemia y su repercusión en los precios y disponibilidad actuales? En este momento podemos contar con microcontroladores con notables capacidades de cómputo y que incluyen incluso la posibilidad de trabajar con tensores.

Por un lado, su simpleza hace que económicamente su implementación sea muy viable, aunque esto mismo provoca ciertos inconvenientes a la hora de su empleabilidad para *IA*.

2.3 Integración de redes neuronales en microcontroladores

Aquí me centraría en que, como has dicho antes, para entrenar las redes hacen falta plataformas muy potentes, por lo que se tiene que hacer en PCs y servidores, mientras que para ejecutar la red no hace falta tanta potencia de cómputo, lo que permite que se pueda ejecutar en los microcontroladores actuales con algunas adaptaciones (que dependerán de cada microcontrolador concreto. De hecho, aunque algunos microcontroladores tenga FPU, el punto fijo es mucho más rápido, por lo que seguramente harán también la cuantización a punto fijo).

Deberías hablar algo más de las tres alternativas de ML para microcontroladores, resaltando sus ventajas y sus inconvenientes. Esto te ayudará después a la hora de justificar por qué escoge TinyML.

Uno de estos inconvenientes es que 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*. Hablar de la importancia de la carencia de la FPU en general, no es nuestro caso, pero en esta sección en teoría no se sabe

El *CloudML* ^[7] es la técnica por la que, alojando redes neuronales profundas en la nube, podemos integrar el uso de las mismas en *TPUs* (Unidades de procesamiento tensorial) y *FPGAs*, entre otras.

EdgeML ^[12] es una librería escrita en *Python* impulsada por *GitHub*, *Microsoft* que mediante *TensorFlow* (o alternativamente en fase experimental *Pytorch*), provee de distintas funciones enfocadas para *Machine Learning* para sistemas embebidos empleados para labores simples.

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.

Capítulo 3

Análisis de requisitos

3.1 Requisitos funcionales y no funcionales

Creo que lo de detectar letras de forma inmediata debería ser un requisito funcional, ya que si no es así, el sistema no funcionaría. Hay alguno más que lo mismo deberías cambiar a funcional, porque son relativos al funcionamiento del sistema. Revisa los apuntes de IS para no meter la pata aquí o te crucificarán los de LSI. Los requisitos no funcionales son del tipo "debe funcionar con baterías", "debe programarse en ADA", "las dimensiones no pueden pasar de tanto", "No puede pesar más de tanto", etc.

3.1.1 Requisitos 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 una interfaz que medie entre el dispositivo y el usuario.

3.1.2 Requisitos no funcionales

- El sistema debe funcionar con comunicación por cable e inalámbrica.
- El tiempo de procesamiento para la identificación de la letra, debe ser inmediato para generar sensación de escritura natural.

- La identificación de la letra debe ser eficaz y consistente.
- La interfaz de usuario debe ser simple de entender y usar.
- El sistema debe contar con autonomía energética.

3.2 Análisis de requisitos 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:

- Un módulo que pueda computar tensores.
- Sensores suficientes para hacer reconocible el movimiento con precisión. En su defecto, se integrarán.
- Tecnología bluetooth. En su defecto, se integrará.
- Unas dimensiones reducidas.
- Documentación para su utilización y el respaldo de una comunidad activa para tener otros proyectos como soporte.

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.

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.

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.

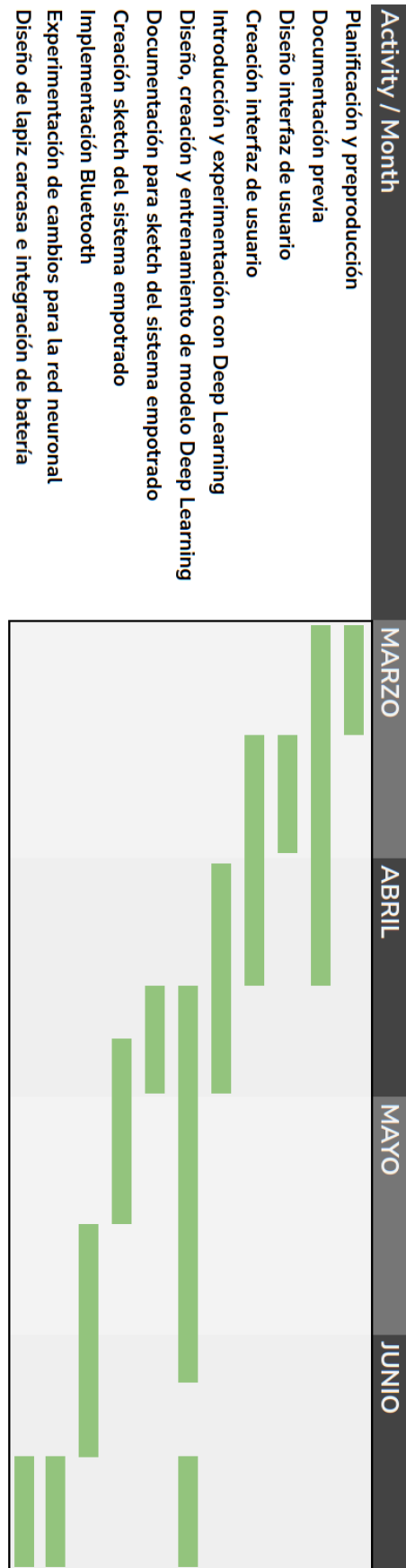


Figura 3.1: Diagrama *Gantt* para la planificación

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 sketch y la integración del bluetooth, dividida en la implementación en la interfaz de usuario y en el programa gestor del hardware (*sketch*); para poder congrega 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 que el precio de la electrónica es superior al ordinario debido a la escasez en la producción.

Descripción	Precio
Arduino Nano Sense 33 BLE	35'80\$~33'82€
Pila 9V Recargable	10'99€
Impresión	1€
Cable MicroUSB Datos	6€
TOTAL: 51,81€	

Capítulo 4

Microcontrolador

Puedes dedicar un capítulo a analizar los frameworks de ML para empujados, para acabar determinando que vas a escoger TinyML, justificando por qué los escoges y por qué descartas los demás, otro capítulo para determinar qué microcontrolador vas a escoger y por qué, analizando varias alternativas comparando las ventajas e inconvenientes de cada una de ellas y justificando por qué escoges finalmente Arduino, y en general, un capítulo por cada decisión importante de tecnología que hayas hecho.

4.1 Planificación

4.1.1 Elección del microcontrolador

Como ya se especificó en la sección (3.2) *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*.

- SparkFun Edge
- Arduino Nano Sense 33 BLE
- STM32F746G
- Adafruit EdgeBadge
- STM32
- ESP32

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*. La realidad es que en este punto compré ambas porque los tiempos de entrega no eran muy esperanzadores, de hecho compré varias Nano y una Sparkfun para ponerme a trabajar cuanto antes. No sé si debería ser franco con esto o decir que elegí la Nano, que en realidad acabé eligiendo porque cuenta con proyectos en los que me he podido apoyar. En definitiva, no sé si decir lo que ocurrió tal cual o ir directamente al motivo por el que acabé usando la Nano, que es lo que hay provisionalmente. 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 sketches en los que apoyarnos, sino que muchos de ellos o la práctica mayoría estarán enfocados al *Machine Learning*. El mayor valor añadido con el que puede contar un dispositivo que se empleará partiendo de pocos conocimientos y con restricción temporal.

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 lápiz.
- Sense: Cuenta con diversos sensores, concretamente con una *IMU* que provee de *giroscopio* y *acelerómetro*.
- BLE: Bluetooth de bajo consumo que proporcionará autonomía y libertad de movimiento.

4.1.2 Elección de las tecnologías

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, la instalación de librerías para Arduino y sus dispositivos, o la carga del sketch 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.

Por otro lado, se hará uso de *TensorFlow Lite* por múltiples razones como que es de código abierto, gratuito, se complementa a la perfección con muchas otras herramientas de alto nivel(entre otras, *Keras* o *Scikit Learn*), etc. Pero al igual que en las elecciones anteriores, lo que más decanta la balanza es siempre la expansión y 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.

4.2 Diseño

4.2.1 Estructura del sketch

El sketch se distribuirá en diferentes secciones, la principal, *deep_pen.ino*(extensión propia de *Arduino*, empleada en el archivo principal de sus sketches) 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 *BLE* vinculado a la recolección de muestras para el modelo.

4.2.2 Servicio BLE

Pese a que se implementarán dos servicios, uno de ellos es parte del *Data collector*, sección extraída 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 4.2.1: Configuración *BLE*(Bluetooth Low Energy) [2,5,36]

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 Identifier):

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.

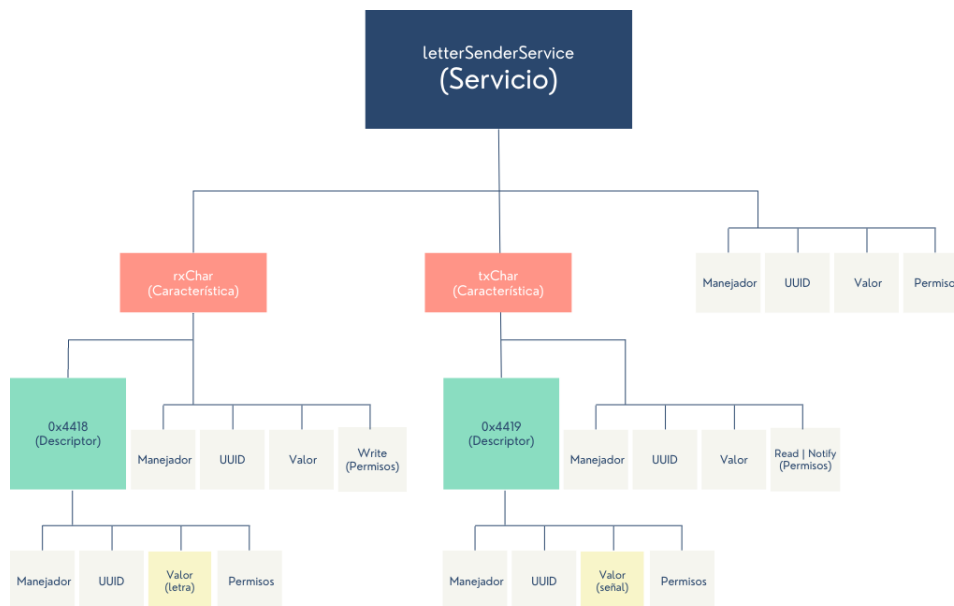


Figura 4.1: Esquema de la estructura planteada para el servicio *BLE*

4.3 Implementación

Después tendrías que explicar el diseño completo, es decir, cómo encajan todas las piezas (incluyendo el PC de desarrollo) para poder desarrollar el sistema. Recuerda que diseñar es explicar qué hace cada cosa, qué datos envía, qué recibe, etc., pero no cómo. El cómo se hace cada cosa es ya desarrollo.

Después podrías hablar del desarrollo, de qué cosas concretas hay que hacer, configurar el IDE de Arduino, hacer sketches, generar los datos de entrenamiento, entrenar la red, convertirla para el microcontrolador, etc., pero sin entrar en los detalles técnicos (que estarán en los apéndices correspondientes). Este capítulo sería como una guía que te vaya diciendo cómo ir consultando los apéndices y qué es lo importante en cada apéndice para que el que quiera pueda replicar tu proyecto. Pero debe estar escrito para que que no quiera replicar el proyecto se entere bien de qué cosas has hecho, de su dificultad, de lo que has tenido que aprender para poder hacer el desarrollo, etc. El tribunal tiene que entender la envergadura de lo que has hecho, qué cosas has hecho cómo de difíciles han sido, etc. Eso es vital. Por eso sacamos todos los detalles técnicos a los apéndices, para que se puedan centrar en lo que les quieres contar. Los apéndices son sólo para demostrar que de verdad lo has hecho, pero lo importante es el rollo que cuentas acerca de cómo ha sido el desarrollo. Espero que hayas cogido la idea.

4.3.1 Configuración BLE [23]

En el primer bloque del sketch, encontramos la configuración de la placa respecto a *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 4.3.1: 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* tales como conexiones y desconexiones o recibo de señal por el canal de lectura *rx*.

4.3.2 Función 'setup()' [1,38]

En esta sección, como de costumbre, se hará el ajuste propio para la la ejecución de nuestro sketch.

Es reseñable mencionar que la placa con la 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 lo es el propio binario del modelo de nuestra red neuronal; la reserva de algunas secciones de memoria también tendrán que ser gestionadas.

El modo de afrontar estas limitaciones será trabajar con la propia flash del dispositivo a modo de región de almacenamiento, creando arrays en la propia flash que harán las veces de mecanismo de almacenamiento. Esto supone solventar la carencia de almacenamiento de una forma eficiente, ya que el coste de acceso a la información será mínimo, sin lecturas externas.

Esto se repetirá en varias circunstancias del sketch, por ejemplo, en este bloque de setup, será necesario reservar un área para las labores de E/S y memoria intermedia de las acciones con tensor.

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];
```

Y más adelante se hará uso de la misma técnica para almacenar el binario de nuestro modelo, entre otros.

4.3.2.1 BLE e IMU setup

En esta primera etapa de setup, se fijarán los parámetros para la configuración de la placa; en primer lugar, se dará para la **IMU**(*Inertial Measurement Unit*) y el **BLE**.

Teoría 4.3.2: Por qué es necesario configurar la IMU [4]

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.

Problemas 4.3.1: Librería Arduino_LSM9DS1

Uno de los problemas con esta parte del código, fue que la librería *Arduino_LSM9DS1* [3] 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 continuo 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].

4.3.2.2 Setup de nuestro modelo

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*.

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] 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*.

Las microoperaciones que utilizaremos en nuestro modelo, son:

- Conv2: Para el procesamiento de la capa homónima.
- Mean: Para promedios como el que se necesita en la capa *GlobalAveragePooling2D*
- FullyConnected: Para las capas densas, entre otros.
- SoftMax: Como función de activación.

Problemas 4.3.2: Al cargar el sketch la placa deja de ser detectada

En las primeras cargas del sketch experimentando con el modelo, la placa dejó 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 microoperaciones 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* definimos el interprete del modelo, que hará uso del repertorio de microoperaciones anteriormente especificado. Por último, inicializamos los led pins como salida. Para poder usar el led como indicativo de estado.

4.3.3 Función 'loop()' [1,38]

En esta función, será donde se establezca el código que se ejecutará cíclicamente mientras la placa esté alimentada.

La lógica para los leds es simple:

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.

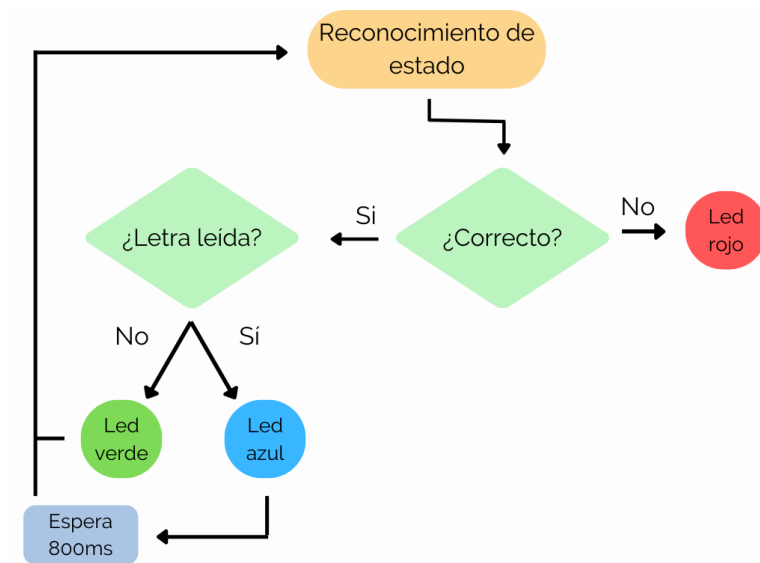


Figura 4.2: Diagrama de funcionamiento de los leds

Para notificar solo una vez la conexión a un dispositivo, 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.

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 acelerómetro 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*.

Y con el trazo completamente construido y corregido, rasterizamos el movimiento para obtener la imagen que pasará por el modelo.

Ya está todo listo para dar comienzo con el procesamiento en el modelo. Para llamar a la 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 recibe 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 el servicio 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 las *características* en *BLE* funcionan con valores constantes hasta que se modifique el estado actual.

Capítulo 5

Red Neuronal

lo que debes contar aquí es lo que se hay que saber para poder hacer un sistema con redes neuronales usando TinyML, qué hay que instalar, qué partes de la biblioteca vas a usar y por qué, cómo se genera la base de datos de entrenamiento, cómo de entrena, etc. De hecho, lo mismo tienes que partir el desarrollo en varios capítulos, según la cantidad de cosas que tengas que saber. Para que te hagas una idea, en los capítulos dedicados al desarrollo tienes que contar qué has hecho como si se lo estuvieras contando a un amiguete de cervezas. indicando qué cosas son las importantes, por qué hay que hacerlas, etc., pero sin detalles técnicos. Los detalles técnicos estarán en los apéndices y loe irás referenciando cuando sea oportuno, para que el que tenga curiosidad, lo mire, aunque el texto debe estar redactado para entender la idea general sin necesidad de ver los apéndices.

5.1 Diseño

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.

Para examinar con mayor profundidad la experimentación respecto al diseño del modelo, observese '**Experimentación con el modelo**' en el Apéndice.

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

MODELO BASE

Cuenta con una estructura de *Keras, Sequential* [22]; gracias a esta utilidad, es posible añadir capas al modelo de forma cómoda, además de algunas funciones para el entrenamiento.

Las capas son las siguientes:

- Rescaling [21]: 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 [17]: 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 [16]: Normaliza sus entradas por lotes. Aplica transformaciones que conservan la media de la salida cercana a 0 y la desviación estándar a 1.
- Activation [15]: 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 [19]: 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 [30]: 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 [18]: 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.

Nuestro modelo base va a contar específicamente con 3 bloques de capas de convolución, normalización, activación y dropout.

¿Por qué estas capas? Esencialmente se debe a que el aporte de cada una de ellas, complementa al procesamiento convolucional irremplazable. Por tanto, la capa *Conv2D* es la de procesamiento, tras esta, se normaliza la salida con la capa *BatchNormalization*, 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* ayuda con el overfitting, un problema que se ha podido experimentar debido a que es una red neuronal de complejidad menor.

En cada bloque lo único que varía es el número de filtros de la capa de convolución, duplicándose en cada uno. Previo a los 3 bloques citados, hay una capa de *Rescaling*, que va a servir meramente para normalizar los valores los píxeles de las imágenes a una escala [0,1].

Y por último, tras los 3 bloques, una capa *GlobalAveragePooling2D* por simple coherencia estructural debido a que reduce la dimensionalidad pero conserva la mayor parte de información relevante(utilizada en la práctica totalidad de *CNNs*), otra capa *Dropout* por el mismo motivo que en los bloques y finalmente la popular capa *Dense* para obtener un output clasificatorio.

5.2 Implementación

5.2.1 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*.

Keras ^[13,20] será la API de alto nivel que se usará para trabajar con las capas del modelo y el estudio de los resultados del entrenamiento.

Otra dependencia imprescindible es *xxd*.

Teoría 5.2.1: Uso de *xxd* [33]

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.

También necesitaremos el dataset para el entrenamiento 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 sketch, se hará uso de la función de rasterización programada por *Pete Warden* [38]. Definidas las funciones de rasterización, es posible rasterizar todos los trazos y destinarlos a los distintos propósitos respecto al modelo.

Teoría 5.2.2: Uso del dataset en *Deep Learning*(1) [27]

Los tres datasets que se usan para *Deep Learning* 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 5.2.3: *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.

Teoría 5.2.4: Uso del dataset en *Deep Learning*(2) [27]

■ Training

Este dataset es el más simple de entender por mera intermediación 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.

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 serán estudiados pero 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.

Las imágenes del dataset ocuparán 32x32 píxeles. Este es otro parámetro que puede ser estudiado, no obstante, estas dimensiones han sido las que han arrojado mejores resultados de forma homogénea.

Presentando mejores resultados el reconocimiento de letras complejas (como la 'k'), a medida que las dimensiones aumentan. Y extrapolando el mismo comportamiento a letras simples (como la 'c'), a razón de una menor resolución.

Para ampliar: <https://www.v7labs.com/blog/train-validation-test-set>
Hablar de bloque 'PREPARE DATASETS'

5.2.2 Entrenamiento

Entrenamiento

Documentación: <https://towardsdatascience.com/image-classification-in-10-minutes>

5.2.3 Testeo

Testeo

5.2.4 Transformación a modelo cuantizado

Transformación a modelo quantico y por qué

5.2.5 Comparación de modelos

Comparación modelos

5.2.6 Integración en el sketch

Obtención de labels

Obtención del modelo integrable en C/C++

Modelo cuantizado por eficiencia y por ahorrar memoria, no por carencia de FPU

Capítulo 6

Interfaz de usuario

6.1 Diseño

Explicar por qué es necesaria

En primer lugar, hice un diseño aproximado para la interfaz de usuario que necesitaba haciendo uso de *QT Design Studio*. Obteniendo el siguiente resultado:

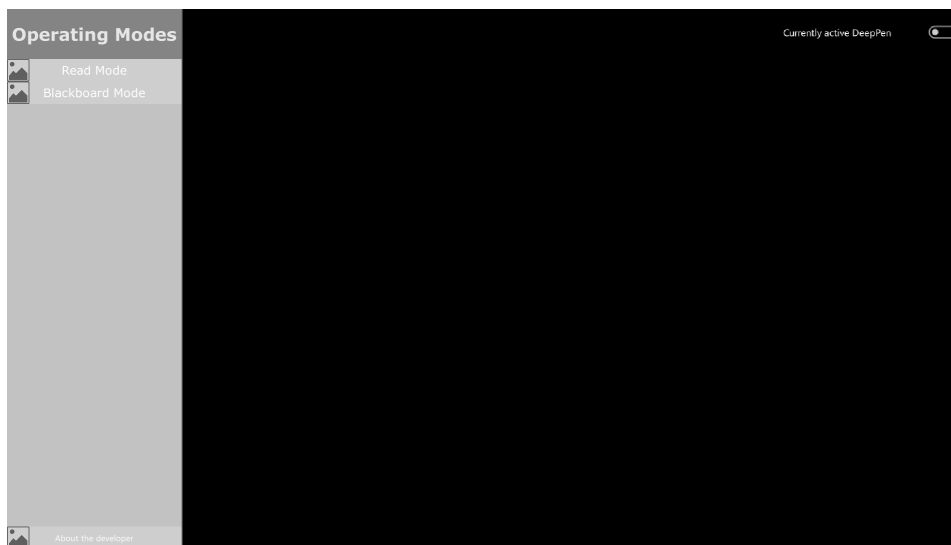


Figura 6.1: Primer boceto en QT Design Studio

Con este primer acercamiento, se dispusieron los elementos planteados como imprescindibles. No obstante, al finalizar esta primera iteración de diseño, se evidenciaron carencias a nivel estético; fácilmente resolubles al comenzar con la implementación real en *QT Creator*. También se observó que faltaban algunos botones como el de sincronización con el *DeepPen*, que *Read Mode* no era un buen nombre para indicar el comportamiento del modo, o que había un espacio

en el centro de la barra superior sin usar y que podía usar para indicar el modo actual.

Contando con estas consideraciones, pudieron corregirse en la siguiente iteración de aproximación al diseño con el que partir en la implementación.

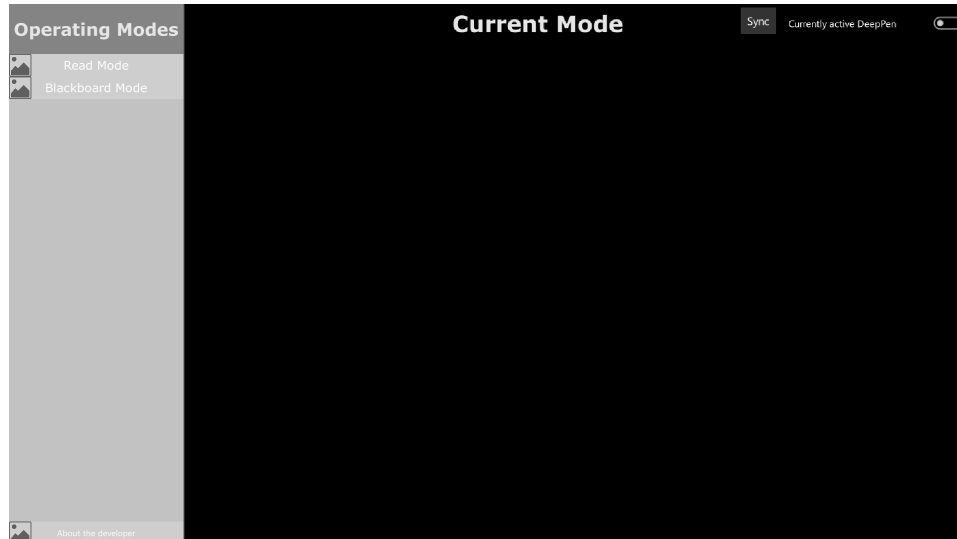


Figura 6.2: Segundo boceto en QT Design Studio

Para pasar esta aproximación de diseño a una implementación real, se utilizaron *QT Creator* con *C++*. Resultando en lo siguiente:

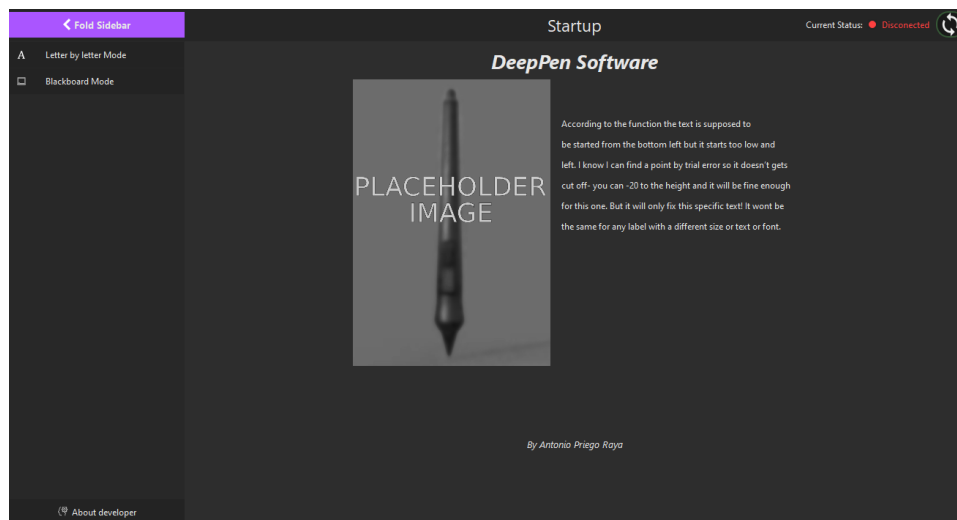


Figura 6.3: Primera implementación de la interfaz de usuario del *DeepPen*

6.2 Implementación

Capítulo 7

Herramientas

Este capítulo lo creé al empezar el proyecto. ¿Es necesario o lo borrarías?

1. **QT Designer**(Licencia de Código Abierto): Bocetado de la interfaz de usuario.
2. **QT Creator**(Licencia de Código Abierto): Implementación de la interfaz de usuario.
(v6.2.4-Linux | v6.2.2-Windows)
3. **Arduino IDE**: Desarrollo del software para el DeepPen.
4. **Visual Studio Code + Latex**: Creación de la memoria.
5. **Bibliotecas**:

Capítulo 8

Apéndice

8.1 Apéndice de soluciones en implementación

8.1.1 Si la placa deja de ser detectada

En la primera subida del programa con los datos de entrenamiento creados por mí, la placa dejó de ser detectada por el ordenador.

Lo cual me llevó a pensar que el bootloader se había corrompido. Sin embargo, buscando posibles causas, encontré una solución: restaurar manualmente la placa, cosa que solo funciona, evidentemente, si el bootloader no está dañado. Pulsando el botón reset de la placa rápidamente varias veces justo al conectarlo al ordenador, puede restaurarse la placa. Si ha funcionado, el "L"led se encenderá.

Esto ocurría al incorporar el modelo de datos que yo mismo entrenaba y fue uno de los problemas que más tiempo me llevó solucionar, sobre todo porque no tenía ningún tipo de referencia de por qué no estaba respondiendo correctamente el programa con el nuevo modelo.

Por suerte llegué a la solución:

8.1.2 El modelo de datos entrenado no responde [34]

Como explico en caso anterior, la incorporar el modelo de datos entrenado con el dataset de ejemplo que proporciona TensorFlow, al subir el programa a la placa, esta dejaba de reconocerse por el ordenador, teniendo que resetear la flash de la placa.

Esto se debía a que el proyecto Arduino no soportaba una de las operaciones que realiza nuestro modelo(reshape).

Podemos solucionar esto de dos formas:

1. (No probada) Simplemente hacer uso de AllOpsResolver, de forma que el interprete tendrá acceso a todas las operaciones disponibles.
2. Esta segunda es la más sofisticada, ya que al no disponer de todas las operaciones para el interprete, reduciremos la cantidad de memoria que ocupamos.
Es la que yo implementé.

Añadir la siguiente línea al archivo *.ino de nuestro proyecto:

```
1 micro_mutable_op_resolver.AddBuiltin(tflite :: BuiltinOperator_RESHAPE,
2                                     tflite :: ops::micro::Register_RESHAPE());
```

De esta forma estamos añadiendo la operación al repertorio de las que tendrá disponibles el interprete(*static_interpreter*), **micro_mutable_op_resolver**.

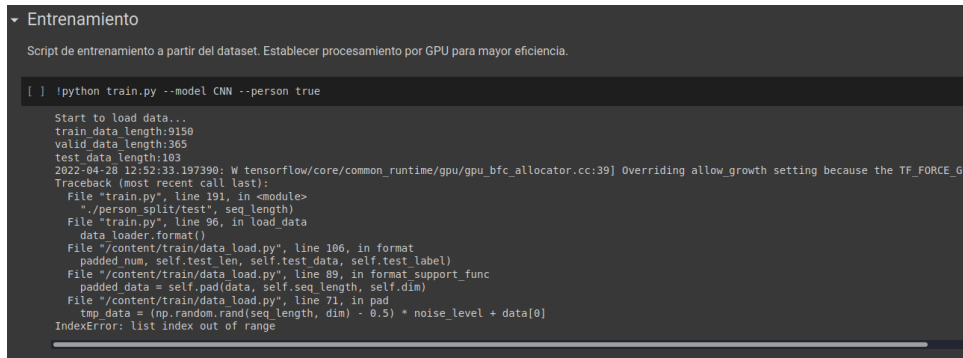
```
72 static tflite :: MicroMutableOpResolver micro_mutable_op_resolver; // NOLINT
73 micro_mutable_op_resolver.AddBuiltin(
74     tflite :: BuiltinOperator_DEPTHWISE_CONV_2D,
75     tflite :: ops::micro::Register_DEPTHWISE_CONV_2D());
76 micro_mutable_op_resolver.AddBuiltin(
77     tflite :: BuiltinOperator_MAX_POOL_2D,
78     tflite :: ops::micro::Register_MAX_POOL_2D());
79 micro_mutable_op_resolver.AddBuiltin(tflite :: BuiltinOperator_CONV_2D,
80                                     tflite :: ops::micro::Register_CONV_2D());
81 micro_mutable_op_resolver.AddBuiltin(
82     tflite :: BuiltinOperator_FULLY_CONNECTED,
83     tflite :: ops::micro::Register_FULLY_CONNECTED());
84 micro_mutable_op_resolver.AddBuiltin(tflite :: BuiltinOperator_SOFTMAX,
85                                     tflite :: ops::micro::Register_SOFTMAX());
86 micro_mutable_op_resolver.AddBuiltin(tflite :: BuiltinOperator_RESHAPE,
87                                     tflite :: ops::micro::Register_RESHAPE());
88
89 // Build an interpreter to run the model with
90 static tflite :: MicroInterpreter static_interpreter (
91     model, micro_mutable_op_resolver, tensor_arena, kTensorArenaSize,
92     error_reporter );
93 interpreter = &static_interpreter;
```

Figura 8.1: Fragmento de *.ino de nuestro proyecto.

Este fragmento de código ilustra lo que acabo de explicar.

8.1.3 Error durante el entrenamiento

Al comenzar con mi primer pequeño dataset para comprobar cuán realizable es mi idea para este modelo, tuve un error que me tuvo durante un buen tiempo ocupado.



```

Entrenamiento

Script de entrenamiento a partir del dataset. Establecer procesamiento por GPU para mayor eficiencia.

[ ] |python train.py --model CNN --person true

Start to load data...
train data length:9150
valid data length:365
test data length:103
2022-04-28 12:52:33.197399: W tensorflow/core/common_runtime/gpu/gpu_bfc_allocator.cc:39] Overriding allow_growth setting because the TF_FORCE_GPU_ALLOW_GROWTH environment variable is set.
Traceback (most recent call last):
  File "train.py", line 191, in <module>
    _, person_split_test, seq_length)
  File "train.py", line 96, in load_data
    data_loader.format()
  File "/content/train/data_loader.py", line 106, in format
    padded_num, self.test_len, self.test_data, self.test_label)
  File "/content/train/data_loader.py", line 89, in format_support_func
    padded_data = self.pad(data, self.seq_length, self.dim)
  File "/content/train/data_loader.py", line 71, in pad
    tmp_data = np.random.randn(seq_length, dim) * 0.5 * noise_level + data[0]
IndexError: list index out of range

```

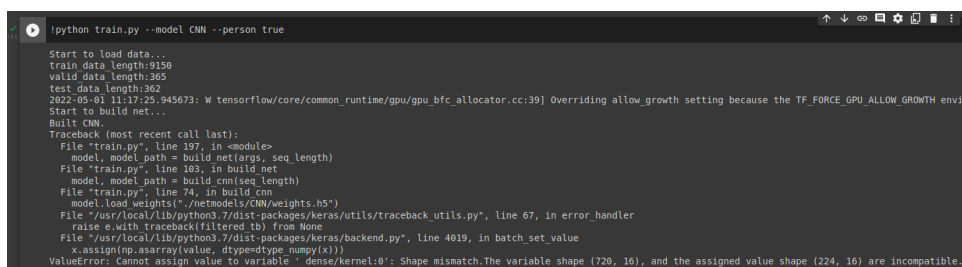
Figura 8.2: Error durante primer entrenamiento con un dataset propio.

Tras revisar que no se debía a la longitud de las secuencias de datos, como hace pensar el mensaje de error, fui a intentar reducir el tamaño de las muestras del dataset y fue cuando vi que había un error en una de las muestras, de forma que había un separador ('-,-,-') sin información.

Al eliminarlo, el entrenamiento ya no arrojaba errores.

8.1.4 Error al cambiar el tamaño de las secuencias de movimientos

Los movimientos que se deben registrar para las letras son, en general, más complejos que los que incluye el dataset de ejemplo. Por lo tanto las secuencias de registro de movimiento, serán mayores. Esto supone un problema porque el modelo está ajustado al dataset de ejemplo y por tanto, se queda algo corto para nuestro propósito. El error se da al cambiar el tamaño de dicha secuencia (`seq_length`) en `train.py` y `train_test.py`.



```

python train.py --model CNN --person true

Start to load data...
train data length:9150
valid data length:365
test data length:362
2022-05-01 11:17:25.945673: W tensorflow/core/common_runtime/gpu/gpu_bfc_allocator.cc:39] Overriding allow_growth setting because the TF_FORCE_GPU_ALLOW_GROWTH environment variable is set.
Start to build net...
Built CNN.
Traceback (most recent call last):
  File "train.py", line 197, in <module>
    model, model_path = build_net(args, seq_length)
  File "train.py", line 103, in build_net
    model, model_path = build_cnn(seq_length)
  File "train.py", line 74, in build_cnn
    model.load_weights("/content/models/CNN/weights.h5")
  File "/usr/local/lib/python3.7/dist-packages/keras/utils/traceback_utils.py", line 67, in error_handler
    raise e.with_traceback(filtered_tb) from None
  File "/usr/local/lib/python3.7/dist-packages/keras/backend.py", line 4819, in batch_set_value
    x.assign(np.asarray(value, dtype=type_numpy(x)))
ValueError: Cannot assign value to variable 'dense/kernel:0': Shape mismatch. The variable shape (720, 16), and the assigned value shape (224, 16) are incompatible.

```

Figura 8.3: Error al cambiar el tamaño de secuencia de movimientos.

Tras mucha documentación sobre Tensorflow, y no obtener la causa del error; pensé que esto ya me pasó por culpa de la configuración del `*.ino`. Así que fui a revisarlo y efectivamente, hay un parámetro en este archivo sujeto a la longitud de los datos. Tras cambiarlo, funcionaba correctamente de nuevo, aunque sin detectar la letra que estaba probando en el nuevo dataset.

8.1.5 Valores nulos al leer por Bluetooth QT

Tras mucho tiempo implementando e intentando dar con el error de por qué no se leía ninguna characteristic 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 por desgracia, estas no provocan errores, hacen que el código no funcione como se esperaba (Enums con valores diferentes o inexistentes, etc). En segundo lugar estaba llamando a un método cuando todavía no se había recibido la characteristic. Por tanto esta aparentaba estar bien registrada, ya que podía obtener su Uuid, pero no contenía ningún valor. Estaba leyendo en `connectToService()` y no en `serviceDetailsDiscovered()`.

He instalado ArduinoBLE.h para probar sketch de harvard

https://create.arduino.cc/projecthub/sibo_gao/harry-potter-magic-wand-384477

Orientación para vTensorFlow no Harvard: https://github.com/andriyadi/MagicWand-TFLite-Arduino/blob/master/src/accelerometer_handler.cpp

Para modo pizarra: https://github.com/petewarden/magic_wand/blob/main/website/index.html. https://tinymt.seas.harvard.edu/magic_wand/

8.2 Apéndice teórico

Capítulo 9

Validación

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 10

Trabajos futuros

- proponer posibles mejoras, indicando qué se podría añadir y cómo se haría
- 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

Capítulo 1 1

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] andriyadi. andriyadi magic wand repository. <https://github.com/andriyadi/MagicWand-TF Lite-Arduino>.
- [2] Developers Android. Descripción general del bluetooth de bajo consumo para android. <https://developer.android.com/guide/topics/connectivity/bluetooth-le?hl=es-419>.
- [3] Arduino. Biblioteca arduino_lsm9ds1. https://github.com/arduino-libraries/Arduino_LSM9DS1/blob/master/src/LSM9DS1.cpp.
- [4] Aprendiendo Arduino. Aprendiendo a manejar arduino en profundidad. <https://aprendiendoarduino.wordpress.com/tag/imu/>.
- [5] Bluetooth. A developer's guide to bluetooth technology. <https://www.bluetooth.com/blog/a-developers-guide-to-bluetooth/>.
- [6] Fernando Sancho Caparrini. Redes neuronales: una visión superficial. <http://www.cs.us.es/~fsancho/?e=72>.
- [7] CloudML. Cloudml webpage. <https://cloudml.io/>.
- [8] Antonia Creswell, Tom White, Vincent Dumoulin, Kai Arulkumaran, Biswa Sengupta, and Anil A. Bharath. Generative adversarial networks: An overview. 2018.
- [9] DeepMind. Alphafold2 webpage. <https://www.deepmind.com/research/highlighted-research/alphafold>.
- [10] DeepMind. Alphago webpage. <https://www.deepmind.com/research/highlighted-research/alphago>.
- [11] GitHub. Github copilot webpage. <https://github.com/features/copilot/>.

- [12] GitHub/Microsoft. Edgectl webpage. <https://microsoft.github.io/EdgeML/>.
- [13] Aurélien Géron. *Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow*. O'Reilly, 2019.
- [14] iars.geo. Breve historia de las redes neuronales artificiales. <https://steemit.com/spanish/@iars.geo/breve-historias-de-las-redes-neuronales-artificiales-articulo-1>.
- [15] Keras. Activation class. https://keras.io/api/layers/core_layers/activation/.
- [16] Keras. Batchnormalization class. https://keras.io/api/layers/normalization_layers/batch_normalization/.
- [17] Keras. Conv2d class. https://keras.io/api/layers/convolution_layers/convolution2d/.
- [18] Keras. Dense class. https://keras.io/api/layers/core_layers/dense/.
- [19] Keras. Dropout class. https://keras.io/api/layers/regularization_layers/dropout/.
- [20] Keras. Keras webpage. <https://keras.io/>.
- [21] Keras. Rescaling class. https://keras.io/api/layers/preprocessing_layers/image_preprocessing/rescaling/.
- [22] Keras. Sequential class. <https://keras.io/api/models/sequential/>.
- [23] Ladvien. Repository named 'arduino_ble_sense'. https://github.com/Ladvien/arduino_ble_sense.
- [24] Ben Lutkevich. Natural language processing (nlp). <https://www.techtarget.com/searchenterpriseai/definition/natural-language-processing-NLP>.
- [25] Maxime. What is a transformer? <https://medium.com/inside-machine-learning/what-is-a-transformer-d07dd1fbec04>.
- [26] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. 1943.
- [27] Andreas C. Müller and Sarah Guido. *Introduction to Machine Learning with Python*. O'Reilly, 2017.

- [28] Na8. Breve historia de las redes neuronales artificiales. <https://www.aprendemachinellearning.com/breve-historia-de-las-redes-neuronales-artificiales/>.
- [29] OpenAI. Openai webpage. <https://openai.com/>.
- [30] Peltarion. Global average pooling 2d. <https://peltarion.com/knowledge-center/documentation/modeling-view/build-an-ai-model/blocks/global-average-pooling-2d>.
- [31] F. Rosenblatt. The perceptron, a perceiving and recognizing automaton project para. 1957.
- [32] Jonathan Stephens(Nvidia). Getting started with nvidia instant nerfs. <https://developer.nvidia.com/blog/getting-started-with-nvidia-instant-nerfs/>.
- [33] TensorFlow. Build convert. https://www.tensorflow.org/lite/microcontrollers/build_convert.
- [34] TensorFlow. Introducción a los microcontroladores por tensorflow. https://www.tensorflow.org/lite/microcontrollers/get_started_low_level.
- [35] tinyML. tinymml webpage. <https://www.tinymml.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] Wikipedia. Intel mcs-48. https://en.wikipedia.org/wiki/Intel_MCS-48.
- [40] Wikipedia. Microcontrolador. <https://es.wikipedia.org/wiki/Microcontrolador>.
- [41] Wikipedia. Multilayer perceptron. https://en.wikipedia.org/wiki/Multilayer_perceptron.