Trabajo Fin de Grado
Ingeniería de Telecomunicación

Aprendizaje automático en dispositivos electrónicos empotrados

Autor: Paula Rodríguez Soriano

Tutor: Antonio Luque Estepa

Dpto. Ingeniería Electrónica
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2020

Departamento de Teoría de la Señal y Comunicaciones

Trabajo Fin de Grado
Ingeniería de Telecomunicación

Aprendizaje automático en dispositivos electrónicos empotrados

Autor:
Paula Rodríguez Soriano

Tutor:
Antonio Luque Estepa
Profesor Titular

Dpto. Ingeniería Electrónica
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2020

Trabajo Fin de Grado: Aprendizaje automático en dispositivos electrónicos empotrados

Autor: Paula Rodríguez Soriano
Tutor: Antonio Luque Estepa

El tribunal nombrado para juzgar el trabajo arriba indicado, compuesto por los siguientes profesores:

Presidente:

Vocal/es:

Secretario:

acuerdan otorgarle la calificación de:

El Secretario del Tribunal

Fecha:

Agradecimientos

A mis padres, por darme todas las oportunidades posibles y priorizar siempre mi formación. A mi madre por priorizarme siempre a mí. A mi hermana por compartir tantas cosas a pesar de diferenciarnos en tantas otras.

A todos los compañeros que han compartido este camino conmigo, los que llegaron al final, los que aún lo andan y los que encontraron nuevos senderos. A Gonzalo en especial, por no dejar de ofrecerme su ayuda hasta cuando yo no quería hablarlo.

A mi tutor, ojalá todos los maestros como tú. Ojalá yo una alumna la mitad de paciente.

A Alejandro, por ser un apoyo constante. A Eva, porque siempre merece la pena hacer cosas por mí cuando son también por nosotras.

Índice Abreviado

Índice

1 Introducción y objetivos

¿Qué te queda para acabar la carrera?

Mi jefe, 2020

En este proyecto se pretende unir dos conceptos con una historia diferente. Los sistemas embebidos, que llevan formando parte de la electrónica desde los años 60, y las redes neuronales, que forma parte de una rama del aprendizaje automático o machine learning con una gran evolución reciente. Utilizar conceptos nuevos sobre una tecnología con historia puede ser la base de una interesante cantidad de aplicaciones tanto académicas como comerciales.

1.1 Introducción

En febrero de 2020, un grupo de investigadores de Google, Microsoft, Qualcomm, Samsung y media docena de universidades se reunieron en San José, Calif, para discutir el desafío de llevar el aprendizaje automático a los límites de la electrónica. Específicamente, microprocesadores funcionando en sensores u otros equipos alimentados mediante baterías.[11]

Es la segunda edición del evento denominado Tiny ML Summit[14], que tuvo su debut en 2019. Se conoce como tiny machine learning al un campo del aprendizaje automático que incluye hardware (circuitos integrados dedicados), algoritmos y software capaces de realizar análisis de datos de sensores en el propio dispositivo. Este campo está teniendo un veloz desarrollo motivado por la ventaja del bajo consumo. Esto abre las puertas a una variedad de casos de uso del aprendizaje automático, todos ellos focalizados en dispositivos con baterías. Actualmente, un pequeño hardware habilitado para el aprendizaje automático se está volviendo lo suficientemente bueno para muchas aplicaciones comerciales. Se está realizando un progreso significativo en algoritmos, redes y modelos de hasta 100kB o menos, y se están desarrollando aplicaciones en el campo de la imagen y el audio.

Con este trasfondo, resultará de interés el estudio realizado a lo largo de este trabajo que presenta un caso práctico de aprendizaje automático sobre un sistema empotrado, que se conocería como un equipo de bajas prestaciones en comparación a los usados habitualmente con esta tecnología.

1.2 Objetivos

En el proyecto se pretende, con ayuda de algunas herramientas ya desarrolladas, trabajar en la adaptación de una red neuronal para que pueda ser implementada en un entorno de capacidades sencillas como es un sistema embebido y estudiar las dificultades de ello. Se ha elegido una red entrenada para la detección y clasificación de imágenes de cifras escritas a mano, ya que es un ejemplo ampliamente utilizado en la introducción a redes neuronales, de complejidad reducida, y un perfecto punto de partida para un posterior estudio en profundidad.

Se entrenará la red desarrollada en Python[7] en un entorno controlado en la nube, se trabajará con los archivos obtenidos del entrenamiento y, mediante la herramienta uTensor[17], se transformarán para poder ser interpretados por el 32F413HDISCOVERY Discovery kit[3]. La aplicación original detecta una cifra. Es

parte del objetivo conocer estos sistemas lo suficiente como para tener una primera implementación funcional y conocida que luego pueda extenderse y mejorar.

2 Tecnologías utilizadas

¿Y el TFG cómo lo llevas, cariñito?

Mi padre, 2020

En este capítulo se darán a conocer los medios empleados para llevar a cabo los objetivos definidos en el capítulo anterior. Se tratarán tanto el equipo electrónico sobre el que se ha trabajado como los protocolos y librerías ejecutados en los mismos.

2.1 Aprendizaje automático y redes neuronales

La tecnología de Machine Learning (o Aprendizaje Automático) es una rama de la inteligencia artificial que trabaja en desarrollar técnicas que permitan a las máquinas ofrecer resultados basados en algoritmos generados por ellas mismas a raíz de una entrada anterior de datos, no solo en un código escrito por el programador[16].

Tiene muchos campos de aplicación gracias a su utilidad en reconocimiento y comparación de patrones, como la seguridad de datos, el comercio financiero, el marketing personalizado, la detección de fraudes, el procesamiendo de lenguaje natural y el diagnóstico de enfermedades.

Existen varias ramas del aprendizaje automático, para este caso nos centraremos en aquella que se basa en la explotación del modelo computacional conocido como red neuronal, cuya representación gráfica habitual puede observarse en la Figura 2.1. Estas redes están formadas por unidades, denominadas neuronas, que realizan operaciones sobre los datos que reciben, conectadas entre sí mediante enlaces. Las neuronas se recogen en capas, siendo la salida de una capa la entrada de la siguiente. Durante el proceso de aprendizaje o entrenamiento de la red se pretende, modificando el peso de las neuronas, optimizar la función que evalúa la red total para las entradas dadas. Una vez completado el entrenamiento y definidos los pesos asociados a cada una de las neuronas, idealmente la red debería evaluar de manera correcta cualquier entrada apropiada no conocida anteriormente.

Las redes neuronales actuales suelen contener desde miles a unos pocos millones de neuronas. Incluso las redes neuronales más pequeñas, de unos cientos de neuronas, necesitan ejecutarse un gran número de veces durante el entrenamiento antes de ser eficaces. El tratamiento de tal cantidad de datos necesita de recursos computacionales nada desdeñables, por lo que es todo un reto realizar y ejecutar machine learning en equipos ligeros. Por contra, la inferencia por parte de la red entrenada a partir de una única entrada, implica una única ejecución de la red, por lo que se puede realizar con recursos más limitados.

2.2 TensorFlow

Según su página web, TensorFlow es una plataforma de código abierto de extremo a extremo para el aprendizaje automático[8]. Cuenta con un ecosistema integral y flexible de herramientas, bibliotecas y recursos comunitarios que permite a los investigadores impulsar el estado del arte en Machine Learning, y a los desarrolladores crear y desplegar fácilmente aplicaciones basadas en ello.

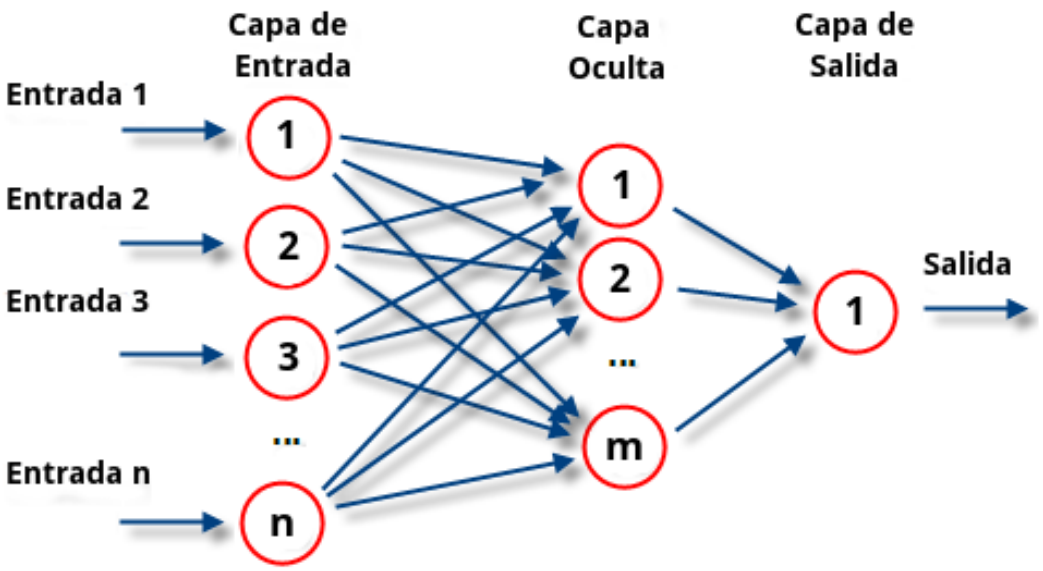


Figura 2.1 Modelo de una red neuronal.

La librería de funciones en lenguaje Python[7], TensorFlow, es ampliamente utilizada en el campo de Machine Learning para el desarrollo y entrenamientos de redes neuronales.

Trabajar con TensorFlow a la hora de construir redes neuronales te permite una gran personalización. A parte del número de capas y la densidad de las mismas, permite elegir entre un amplio número de funciones de activación de las neuronas y métodos de optimización de la red. Además, usar un lenguaje con una agradecida curva de aprendizaje como Python, mejora la accesibilidad tanto al aprendizaje automático como al tratamiento de los datos necesario previo a la red.

Actualmente, para aplicaciones de alto nivel, TensorFlow en Python suele usarse sobre la API Keras[5], que facilita aún más el acceso a funciones complejas. También existe una versión de la librería para JavaScript, TensorFlow Lite[10] para IoT (internet of things) y aplicaciones móviles, y TensorFlow Extended[9] para grandes ambientes de producción.

2.3 Sistemas Empotrados

Un sistema embebido o empotrado es un sistema de computación diseñado para cubrir una o pocas funciones específicas[12]. Al no ser necesarios recursos para propósito general, la mayoría de los sistemas empotrados suelen andar escasos de ellos, por lo que salen más económicos, además de ser más pequeños y manejables. Su programación suele realizarse en lenguajes sencillos de nivel medio-bajo, como C o C++, y posteriormente convertirse a lenguaje ensamblador mediante un compilador específico del microcontrolador o microprocesador del sistema.

Al diseñarse para funciones dedicadas, los componentes de cada sistema empotrado pueden variar mucho de uno a otro, sobre todo en subsistemas de comunicación y periféricos de entrada y salida. Aún así, puede generalizarse que en casi todos existan una unidad central de procesamiento (microprocesador, microcontrolador o DSP), memorias volátiles y no volátiles, y un módulo de alimentación.

Los sistemas embebidos tienen una inmensa cantidad de aplicaciones muy variadas, casi siempre asociadas a la posibilidad de producir muchos equipos a bajo coste. Podemos encontrarlos formando parte de electrodomésticos, maquinaria industrial, en el sector automovilístico, en equipos de comunicación y en equipos de medida, entre otros.

2.3.1 32F413HDISCOVERY Discovery kit

El 32F413HDISCOVERY Discovery kit[3] permite al usuario desarrollar aplicaciones con la serie de microcontroladores STM32F4 basada en ARM®Cortex®-M4.

Este kit, mostrado en la Figura 2.2 y la Figura 2.3, puede usarse para diversas aplicaciones aprovechando sus prestaciones de conectividad de alta velocidad, audio, gráficas, seguridad, video y soporte para sensores.

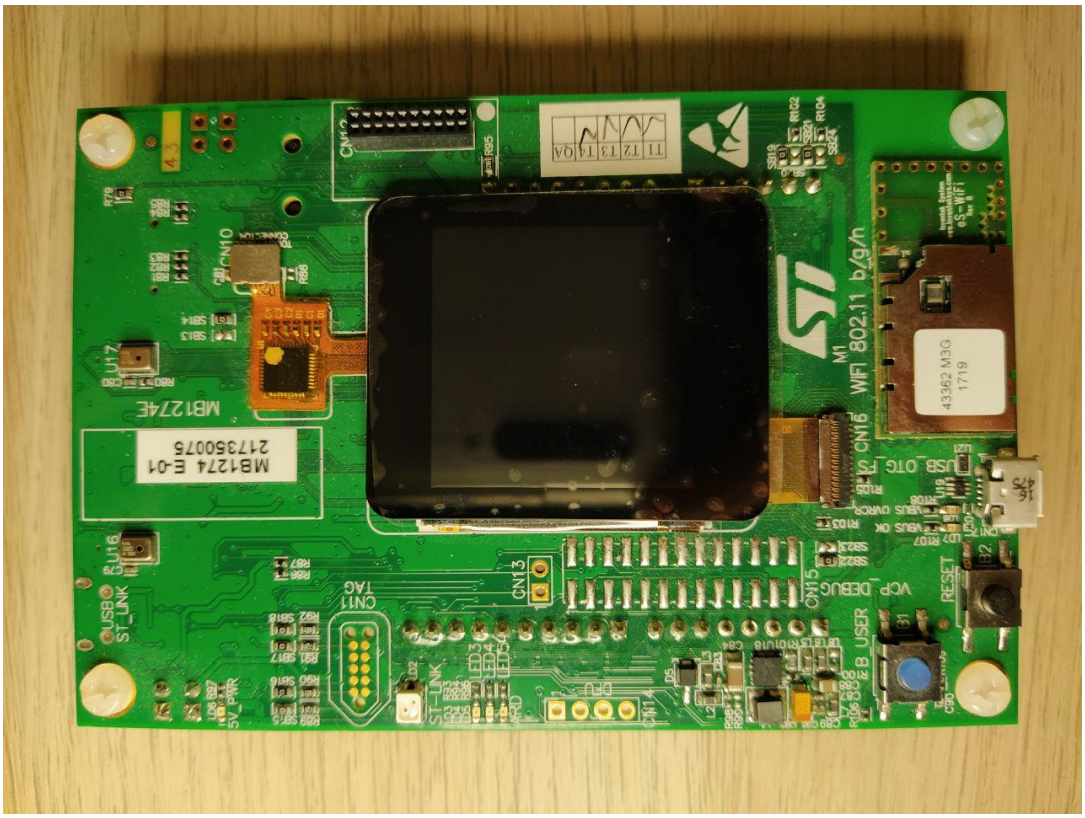


Figura 2.2 32F413HDISCOVERY Discovery kit: cara superior.

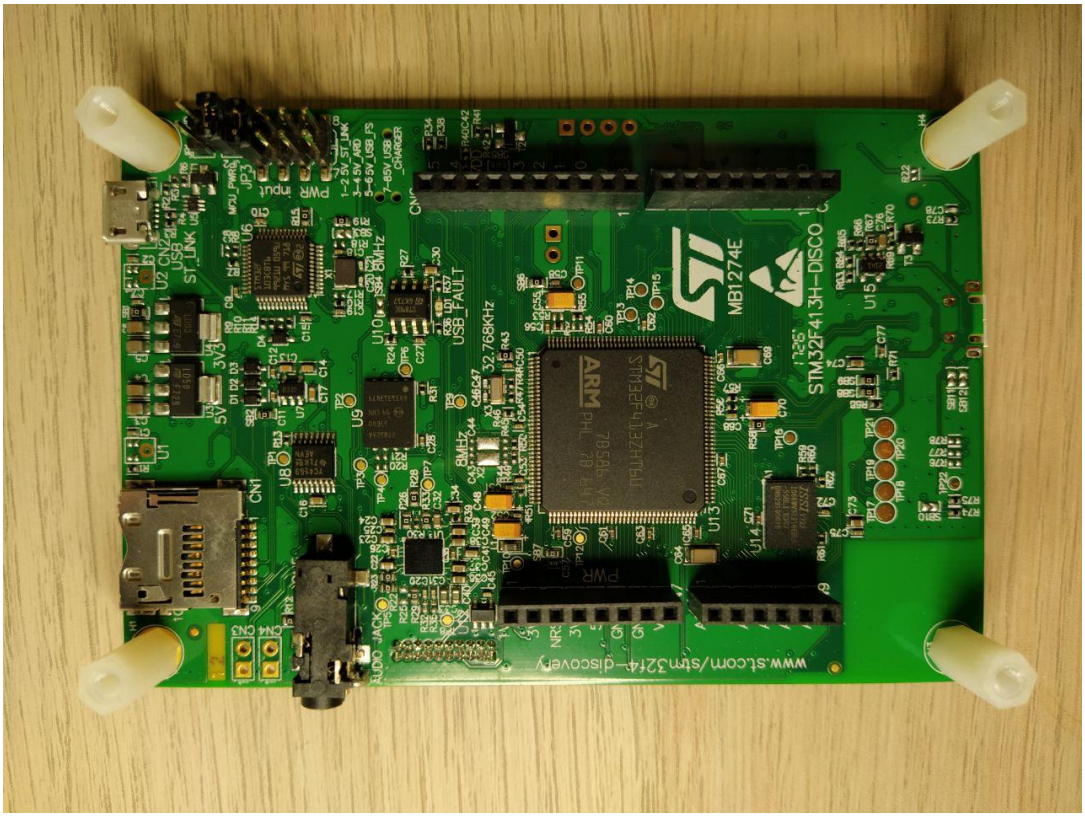


Figura 2.3 32F413HDISCOVERY Discovery kit: cara inferior.

Soporta, además, conectividad con Arduino™Uno V3, lo que le da acceso a multitud de módulos de expansión compatibles.

Características del microcontrolador

- Microcontrolador STM32F413ZHT6
- CPU ARM®32-bit Cortex®-M4
- 100 MHz de frecuencia máxima de la CPU
- Desde 1.7V a 3.6V de VDD
- 1.5MB de memoria Flash
- 320KB de memoria SRAM
- 81 pin GPIO capacidad de interrupción externa
- 10 temporizadores de propósito general
- 2 temporizadores básicos
- temporizador de bajo consumo
- 5 SPI
- I2S
- 3 I2C
- I2CFMP
- 4 USART
- SDIO
- 3 CAN
- SAI
- UART de bajo consumo
- USB 2.0
- Convertidor AD de 12 bits con 16 canales
- 2 convertidores DA de 12 bits
- RTC
- Generador aleatorio (TRNG)

Características de la placa

- ST-LINK / V2-1 integrado que admite la capacidad de reenumeración USB
- Funciones ST-LINK USB:
  - Puerto de comunicación virtual
  - Unidad de almacenamiento
  - Puerto de debug
- LCD de 240x240 píxeles con interfaz paralela y conector de panel táctil
- PSRAM de 8Mbit (512K palabras de 16 bit)
- Memoria Flash Quad-SPI de 128 Mbit
- Códec de audio I2S
- Conector Jack para audio con entrada de micrófono y salida de estéreo
- Dos micrófonos ST-MEMS integrados
- Extensión de conector para hasta 5 micrófonos MEMS
- USB OTG FS con conector Micro-AB

- Conector para microSD™card
- Módulo Wi-Fi™802.11 b/g/n integrado
- 2 pulsadores (uso y reset)
- 2 LEDs de uso: uno verde y otro rojo
- Conectores Arduino™Uno V3
- 4 opciones de alimentación:
  - ST LINK/V2-1
  - Conector USB FS
  - 5V desde Arduino™Uno V3
  - Cargador USB
- Software gratuito completo que incluye una variedad de ejemplos, parte del paquete STM32Cube
- Soporte de una amplia variedad de entornos de desarrollo integrados

2.4 uTensor

uTensor[17] es un framework de inferencia de aprendizaje automático extremadamente ligero construido en Mbed[6] y Tensorflow[8]. Está formado por una librería de tiempo de ejecución y una herramienta offline. El tamaño total de la definición del gráfico y la implementación del algoritmo de una red neuronal de 3 capas producido por uTensor es inferior a 32kB en el binario resultante (excluyendo el fichero de pesos de las neuronas).

Como se muestra en la Figura 2.4 y en la Figura 2.5, desde un modelo construido y entrenado en Tensorflow, uTensor produce un archivo .cpp y .hpp. Estos archivos contienen el código C++ 11 generado necesario para la inferencia. Para trabajar con el programa resultante en sistemas empotrados solo es necesario traducirlo a ensamblador con su compilador específico.

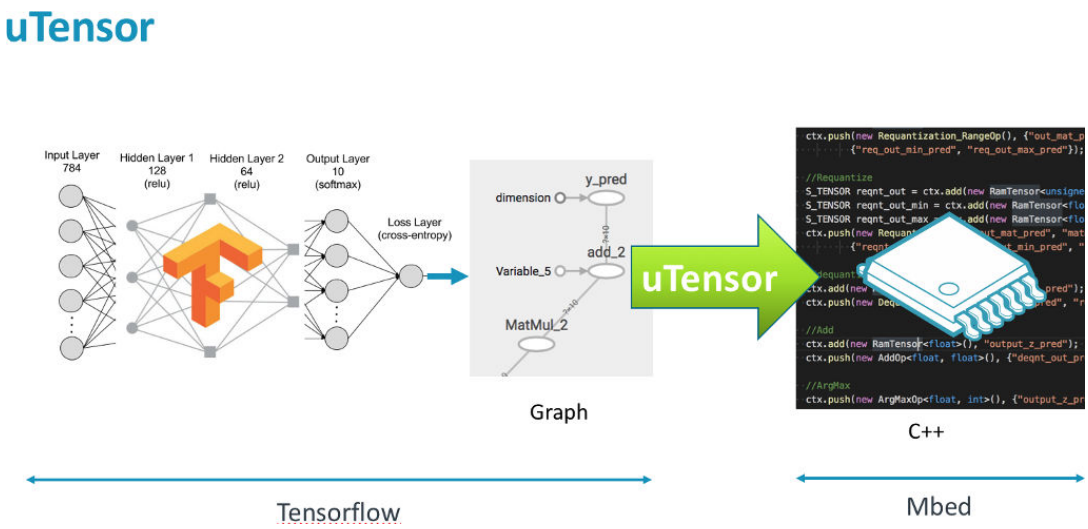


Figura 2.4 Esquema del funcionamiento de uTensor.

La interfaz de uso de la función en el archivo .cpp es similar al mostrado en el código 2.1.

Código 2.1 Uso de la red neuronal creada por uTensor en archivo .cpp.

```
#include "models/deep_mlp.hpp"
...
```

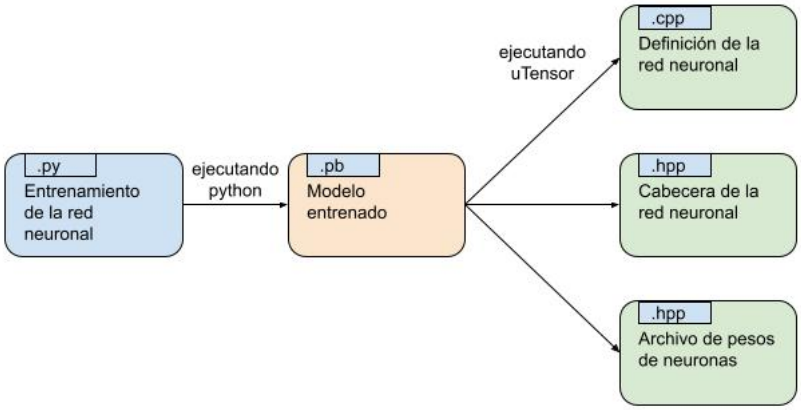


Figura 2.5 Esquema de archivos generados por uTensor.

```
Context ctx; //creando el contexto
...
//preparación de los datos de entrada
...
get_deep_mlp_ctx(Context& ctx, Tensor* input_0); //ejecución de la inferencia
ctx.eval();
S_TENSOR prediction = ctx.get({"y_pred:0"}); //obtención del resultado
```

Los archivos .hpp y .cpp también pueden generarse dado un archivo de modelo (protocol buffer), como se muestra en el código 2.2

Código 2.2 Generación de .cpp y .hpp desde protocol buffer.

```
$ utensor-cli deep_mlp.pb --output-nodes=y_pred
...
... Generate weight file: models/deep_mlp_weight.hpp
... Generate header file: models/deep_mlp.hpp
... Generate source file: models/deep_mlp.cpp
```

En estos momentos uTensor aún está en desarrollo. Actualmente soporta las siguientes funciones de TensorFlow 1.17:

- Add
- ArgMax
- Dropout
- MatMul
- Max
- Min
- Placeholder
- Quantization Ops
- ReLu
- Reshape

Puede obervarse que sus funcionalidades están limitadas al uso más básico. La carencia de herramientas para construir redes convolucionales puede resultar molesta si se pretenden abordar problemas complejos. También se echan en falta algunas funciones de corte usadas a menudo como Sigmoid o Softmax.

3 Desarrollos Previos

Puffff... Pues no lo dejes porque luego cuesta mucho retomarlo

Mi compañero de trabajo, 2020

En este capítulo se presentarán y explicarán los procedimientos a seguir necesarios antes de la implementación de la solución al problema.

3.1 Partes del ejemplo MNIST

En esta sección se describirán los archivos destacables de uno de los ejemplos clásicos de uso de aprendizaje automático y su interacción con las herramientas necesarias para la implementación en el dispositivo de trabajo[13]. El objetivo de este programa es que el equipo reconozca la cifra (0-9) que se escriba sobre la pantalla táctil y la imprima por la misma. Para ello se entrenará una red neuronal a partir de un conjunto de imágenes de 28x28 píxeles como las mostradas en la Figura 3.1.

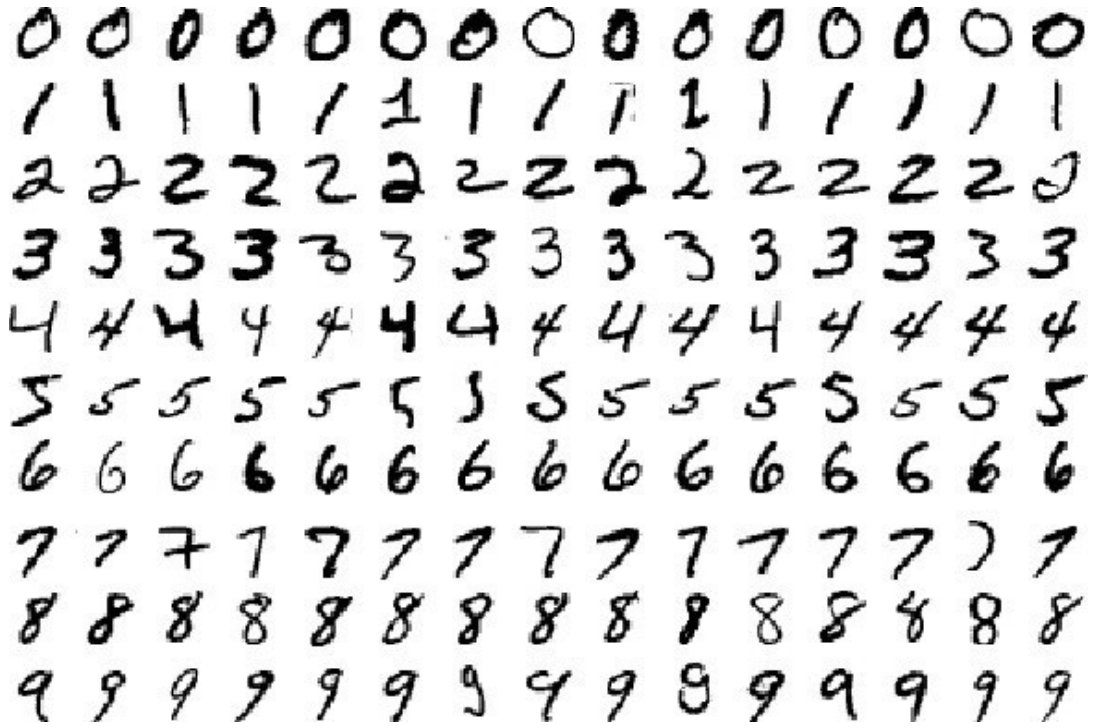


Figura 3.1 Imágenes para el entrenamiento de la red neuronal.

3.1.1 Red neuronal y entrenamiento en Python

El archivo deep_mlp.py en Python contiene la creación de la red neuronal y los pasos necesarios para su entrenamiento.

En la Figura 3.4 se observa un esquema de esta red de dos capas ocultas, que tiene de entrada un vector con los valores en escala de grises (normalizados, como para cualquier red neuronal) de los 28x28 píxeles de la imagen colocados en orden en un vector [1, 784]. La salida de la red es un vector [1, 10] con valores de probabilidad asociados a las 10 posibles cifras, siendo el asociado al valor más cercano a 1 el que se recogerá como la predicción de la red.

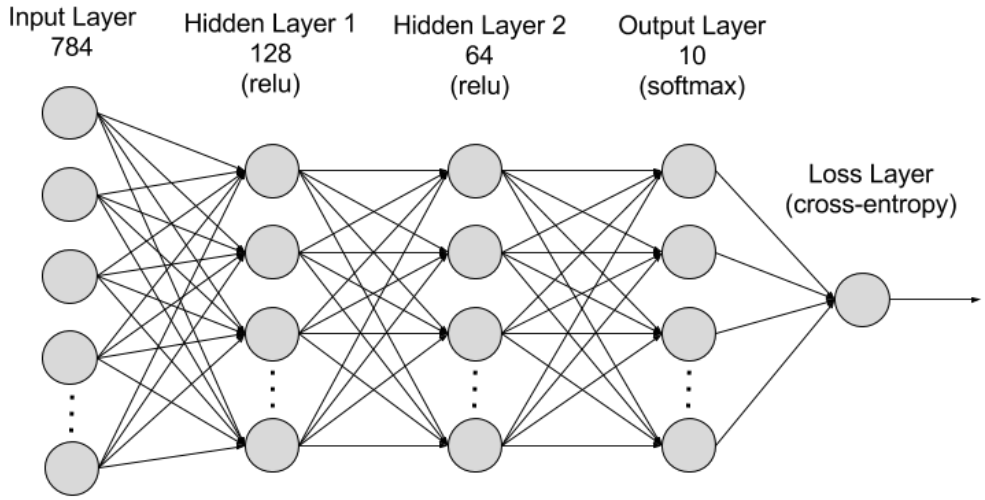


Figura 3.2 Esquema de la red para el ejemplo MNIST.

Para el entrenamiento de la red se usarán un total de 70000 imágenes, separadas en 55000 para entrenamiento, 10000 para testear la red y 5000 para validarla. Cada imagen va asociada a su etiqueta, un vector [1, 10] que codifica la cifra que representa la imagen usando lo que se conoce como One-Hot Encoding[15]. Esto es, el vector contendrá ceros en todas las posiciones salvo en la que señalaría el valor correcto, que contendrá un 1. Como ejemplo, una imagen que represente un 2 tendrá un vector etiqueta asociado de [0,0,1,0,0,0,0,0,0,0] mientras que un 6 tendrá un vector de etiqueta de [0,0,0,0,0,0,1,0,0,0].

Con todo lo explicado se entiende que, para el primer paso de entrenamiento de la red, se tendrán un vector de entrada [784, 55000] y un vector de etiquetas de [10, 55000].

Conocidos los datos de entrada puede analizarse la red construida para trabajar con ellos, definida en el código 3.1

Código 3.1 Definición de la red neuronal en deep_mlp.py.

```
# Red neuronal de 2 capas totalmente conectada
def deepnn(x):
  W_fc1 = weight_variable([784, 128], name='W_fc1')
  b_fc1 = bias_variable([128], name='b_fc1')
  a_fc1 = tf.add(tf.matmul(x, W_fc1), b_fc1, name="zscore")
  h_fc1 = tf.nn.relu(a_fc1)
  layer1 = tf.nn.dropout(h_fc1, 0.50)

  W_fc2 = weight_variable([128, 64], name='W_fc2')
  b_fc2 = bias_variable([64], name='b_fc2')
  a_fc2 = tf.add(tf.matmul(layer1, W_fc2), b_fc2, name="zscore")
  h_fc2 = tf.nn.relu(a_fc2)
  layer2 = tf.nn.dropout(h_fc2, 0.50)
```

```
W_fc3 = weight_variable([64, 10], name='W_fc3')
b_fc3 = bias_variable([10], name='b_fc3')
logits = tf.add(tf.matmul(layer2, W_fc3), b_fc3, name="logits")
y_pred = tf.argmax(logits, 1, name='y_pred')

return y_pred, logits
```

Para esta red, cada neurona llevará asociada la ecuación (3.1), donde W se refiere al peso de la neurona y b a la tendencia ("bias" en inglés). Mientras, la función de activación será la ecuación rectificadora (3.2), quedando la forma matricial mostrada en la ecuación (3.3) para un ejemplo de tres entradas.

$$z_i = \sum_j W_{i,j} x_j + b_i \tag{3.1}$$

$$f(z_i) = ReLU(z_i) = \max(0, x_j) \tag{3.2}$$

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = ReLU\left(\begin{bmatrix} W_{1,2} & W_{1,2} & W_{1,3} \\ W_{2,1} & W_{2,2} & W_{2,3} \\ W_{3,1} & W_{3,2} & W_{3,3} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \right) \tag{3.3}$$

De esta manera se están usando solo funciones con las que uTensor puede trabajar. De las mencionadas en Sección 2.4, en las ecuaciones (3.1)-(3.2) aparecen Add, MatMul y ReLU.

Las otras funciones que se utilizan, entrando también dentro de los límites de uTensor son Dropout y ArgMax.

La primera de ellas es una función usada para mantener la independencia de cada uno de los nodos por separado. Durante el entrenamiento, para regularizar el modelo, algunos nodos desaparecen (se establecen en 0) aleatoriamente, y los nodos restantes se reescalan. Esto obliga a cada nodo a ser útil por si mismo, no pudiéndose sustentar su ineficiencia en la eficiencia de otros.

```
layer2 = tf.nn.dropout(h_fc2, 0.50)
```

En esta línea, el primer argumento (h_fc2) se refiere al grupo de nodos que desaparecerá, mientras que el segundo (0.50) se refiere a la proporción con lo que lo harán. Al mismo tiempo, la inversa del segundo argumento indica cuánto deben reescalarse los nodos que se quedan (1/0.50 = 2).

Por otro lado, ArgMax devuelve el resultado equivalente a la posición del vector logits, más cercana a 1. El vector logits es el que incluye, de manera ordenada, los valores de probabilidad calculados por la red para cada una de las cifras. Su valor más cercano a 1 es, por tanto, la predicción de la red neuronal para esa entrada.

El resto del archivo se dedica a entrenar la red sin nada realmente destacable, usando backpropagation. La red es alimentada con los datos de entrenamiento y validada en varias ocasiones con los datos de validación para comprobar su mejora. Por último, una vez completado el entrenamiento, se testea una única vez con los datos reservados para ello. El código incluye la impresión en la ventana de comandos de los resultados de las validaciones y el test durante el entrenamiento.

3.1.2 Red neuronal en C++ y archivo de pesos

Una vez entrenada la red, se convertirá el modelo a C++ mediante la herramienta uTensor, que generará también el archivo de pesos de las neuronas como ya se mostró en la Figura 2.5. Como cualquier archivo pregenerado, la sintaxis del archivo fuente deep_mlp.cpp y el archivo cabecera deep_mlp.hpp es poco agradable y no es de interés para este proyecto. Así mismo, el archivo deep_mlp_weight.hpp, que contiene el peso de las neuronas calculado durante el entrenamiento tampoco es un archivo del que vayamos a obtener gran información. Si se deseara modificar cualquiera de estos archivos, la manera sensata sería modificar el archivo en Python[7] y volver a aplicar la herramienta uTensor para generarlos de nuevo.

3.1.3 Aplicación en C++

El archivo main.cpp es el archivo principal que describe la aplicación que va a ejecutarse en el dispositivo. Para facilitar el uso del equipamiento incluído en la placa se importarán las funciones de las cabeceras stm32f413h_discovery_ts.h y stm32f413h_discovery_lcd.h necesarias para acceder al estado del botón y a la pantalla LCD respectivamente. Para ayudar al procesamiento de la imagen formada al escribir en la pantalla se usaran funciones de una librería creada para la ocasión, mencionadas en el archivo image.h. Todo esto se resume en la Figura 3.3.

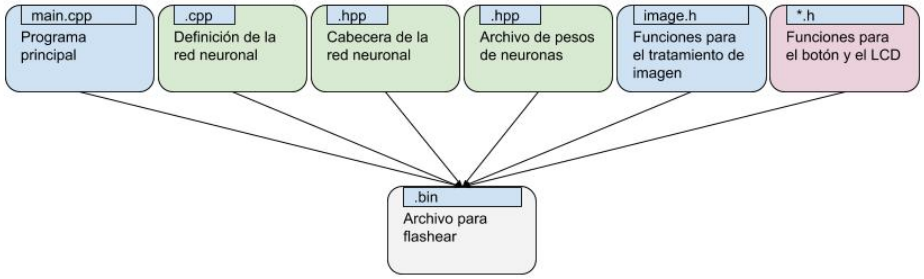


Figura 3.3 Esquema de archivos necesarios para compilar la solución final.

Además de la funcionalidad ya explicada, la aplicación mandará varios mensajes mientras se ejecuta a través del puerto serie.

La aplicación no es elaborada en exceso. Al activarse, comenzará un bucle en el que se comprobará continuamente el estado de la pantalla táctil, que comienza con una imagen completamente en blanco. En el caso de detectarse una interacción con la misma, se generará e imprimirá sobre la imagen un pequeño círculo o punto verde con el centro en el centro de la interacción, como define el código 3.2. Este círculo permanecerá en la pantalla y será la imagen suma de todos los círculos la introducida en la evaluada por el modelo llegado el caso.

Código 3.2 Código ejecutado al tocar la pantalla LCD.

```
if(TS_State.touchDetected) {
           /* Detectado toque simple o doble */

           /* Obtención de la posición X e Y del primer toque */
           x1 = TS_State.touchX[0];
           y1 = TS_State.touchY[0];

           img->draw_circle(x1, y1, 7);

           BSP_LCD_SetTextColor(LCD_COLOR_GREEN);
           BSP_LCD_FillCircle(x1, y1, 5);

           wait_ms(5); // Esperará 5 ms antes de volver a comprobar el estado
               de la pantalla
       }
```

En el momento en el que el estado del botón pase de pulsado a no pulsado, comenzará el procesamiento de la imagen que muestre la pantalla. Lo primero que se hará con ella será redimensionarla mediante una de las funciones del archivo image.h, para reducirla desde el número de píxeles que tiene la pantalla LCD incluída en placa hasta el número de píxeles de las imágenes con las que se ha entrenado la red, 28x28. Luego, se remodelará para reordenar los valores de los píxeles en un único vector de 784. Es importante conocer que, tal como se ha dibujado en la pantalla, los valores del vector serán unos o ceros, dependiendo de si el pixel está coloreado o no.

Una vez realizado el preprocesado de la imagen, las llamadas a las funciones definidas en deep_mlp.cpp, correspondientes a la evaluación de la imagen son las mostradas en el código 3.3. La variable result contendrá el valor predicho por la red.

Código 3.3 Llámada a la evaluación de la imagen.

```
get_deep_mlp_ctx(ctx, smallImage.get_data());
          pc.printf("Evaluating\n\r");
          ctx.eval();
          prediction = ctx.get({"y_pred:0"});
          int result = *(prediction->read<int>(0,0));
```

Por último, el valor se imprimirá en la pantalla LCD sobre un fondo blanco.

3.2 Preparación del entorno

El entorno necesitado para trabajar el ejemplo no es trivial. Tensorflow[8] es una librería que ha evolucionado mucho y muy rápido en el poco tiempo de vida de Python, por lo que se convierte en necesario conocer y controlar las versiones que estamos usando para evitar incompatibilidades. Para este uso no es prioritario usar la última versión, ya que la propia herramienta uTensor[17] será la limitante en las funciones que podamos usar o no. Se ha decidido trabajar sobre un entorno de desarrollo integrado (IDE) en la nube ya que es una herramienta común en machine learning, donde el tiempo que va a tardar en completarse el entrenamiento suele ser largo y no siempre conocido a priori.

3.2.1 Cloud9 IDE

Cloud9[2] es un servicio online de entornos de desarrollo integrados que forma parte de Amazon Web Services desde 2016. La opción gratuita es sufiente para lo que compete y nos aportará el entorno a configurar.

Lo primero será crear una cuenta de Amazon Web Service[1] y acceder desde ella a Cloud9. Allí se accedera a crear el entorno pinchando en "create enviroment" en el tablero mostrado en la Figura 3.4.

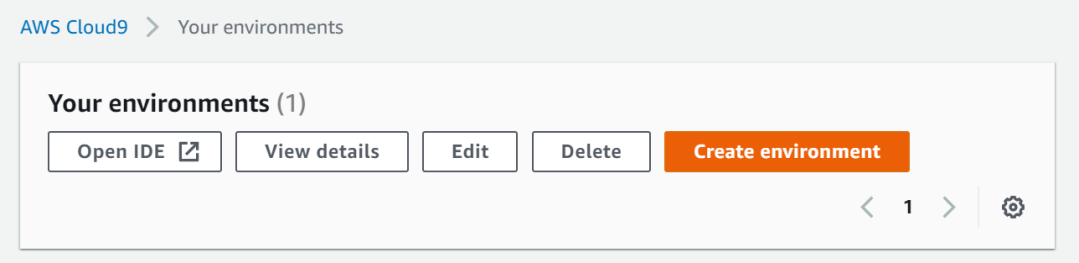


Figura 3.4 Tablero Cloud9.

Se escribirá el nombre del entorno y, si resulta de interés, una breve descripción del mismo en la sección mostrada en la Figura 3.5 y se pinchará en "Next Step"

Luego, se elegirán las opciones mostradas en la Figura 3.6, manteniendo la configuración gratuita. En la Figura 3.7 se elegira la plataforma Ubuntu Server 18.04 LTS, con la que resultará más fácil, si interesa en un futuro, trasladar lo trabajado a un equipo físico. Esto obliga a cambiar varios de los pasos de instalación de uTensor explicados originalmente en su GitHub y, sin embargo, merece la pena por el aumento de compatibilidad con otros servicios. El resto de opciones se mantienen por defecto. Por último solo será necesario pinchar en "Next Step" y en "Create Enviroment" para confirmarlo.

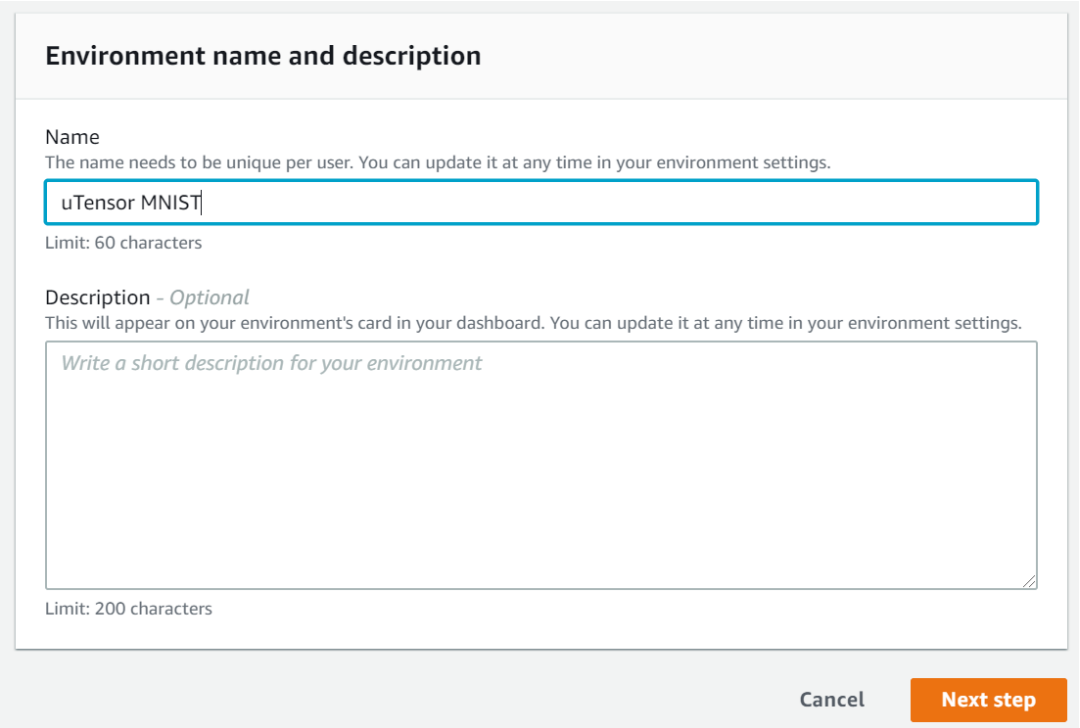


Figura 3.5 Nombre del entorno.

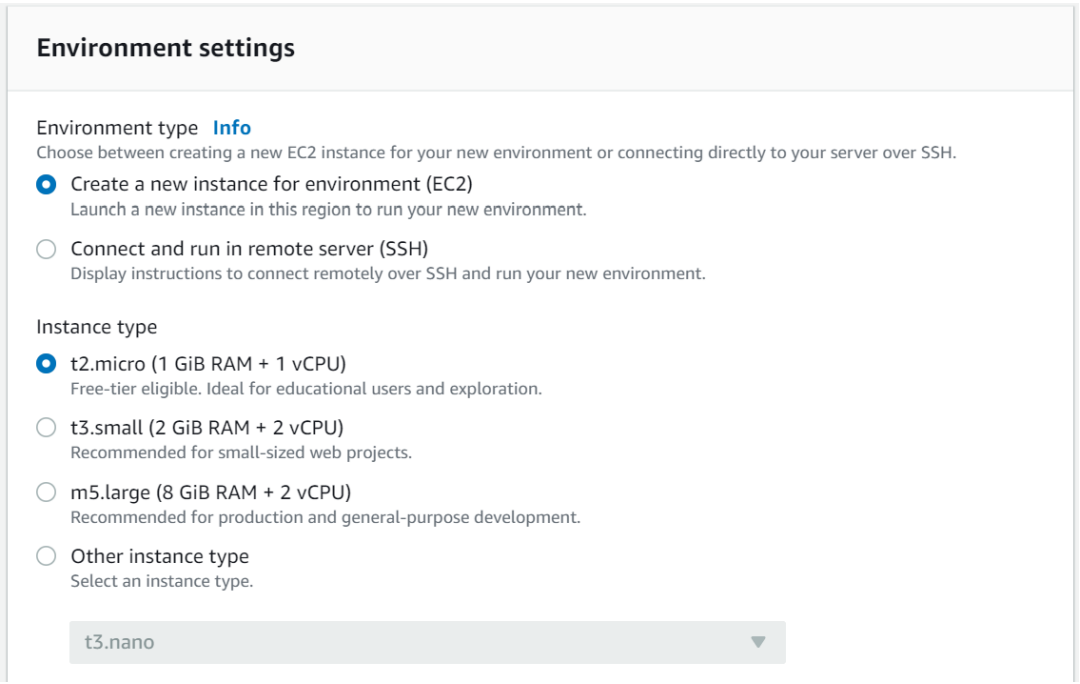


Figura 3.6 Configuración del entorno 1.

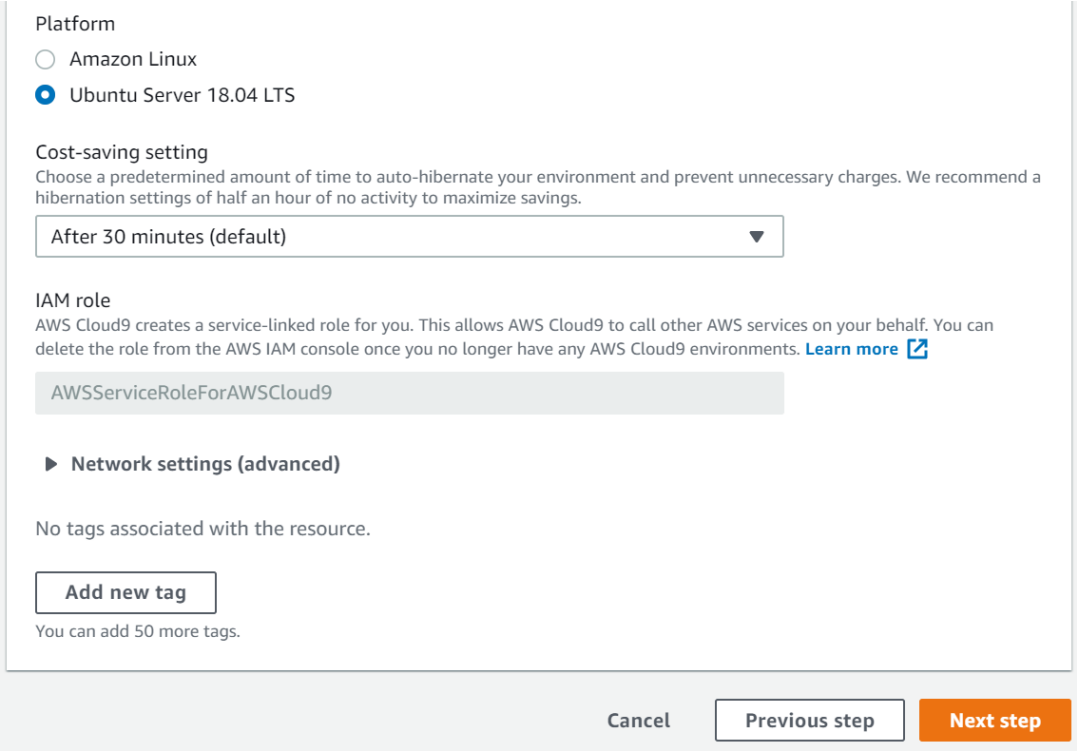


Figura 3.7 Configuración del entorno 2.

Para el entrenamiento de la red se necesitará que el entorno tenga algo más de memoria que aquella con la que se crea. Para esto, se crea y se guarda el script resize.sh mostrado en el código 3.4 al que se llamará con el comando referenciado en el código 3.5.

Código 3.4 Código del archivo resize.sh.

```
#!/bin/bash

# Especifica el volumen deseado en GiB como un argumento en la linea de
    comandos. Si no se especifica, por defecto es 20 GiB.
SIZE=${1:-20}

# Install the jq command-line JSON processor.
sudo apt install -y jq

# Consigue la ID del host de entorno de la instancia Amazon EC2.
INSTANCEID=$(curl http://169.254.169.254/latest/meta-data//instance-id)

# Consigue la ID del volumen EBS (Elastic Block Storage) de Amazon asociado con
     la instancia.
VOLUMEID=$(aws ec2 describe-instances --instance-id $INSTANCEID | jq -r .
    Reservations[0].Instances[0].BlockDeviceMappings[0].Ebs.VolumeId)

# Redimensiona el volumen EBS.
aws ec2 modify-volume --volume-id $VOLUMEID --size $SIZE

# Espera a terminar de redimensionar.
```

```
while [ "$(aws ec2 describe-volumes-modifications --volume-id $VOLUMEID --
    filters Name=modification-state,Values="optimizing","completed" | jq '.
    VolumesModifications | length')" != "1" ]; do
  sleep 1
done

# Reescribe la tabla de la partición para que use todo el espacio que pueda.
sudo growpart /dev/xvda 1

# Expande el tamaño del sistema de ficheros.
sudo resize2fs /dev/xvda1
```

Código 3.5 Comando de llamada.

```
bash resize.sh
```

3.2.2 Instalación de uTensor y Mbed

Para la instalción de las herramientas necesarias podría ser suficiente con ejecutar un script incluído en el GitHub de uTensor pero, como se ha comentado con anterioridad, elegir ciertas opciones durante la creación del entorno obliga a modificar el procedimiento.

Para abreviar los comandos descritos en el resto de la sección se tomarán las referencias de variables mostradas en el código 3.6.

Código 3.6 Variables para la instalación.

```
logfile="install.log"

installDir=‘realpath ~/environment‘
installDir=‘echo "$installDir/tools"‘

gccURL="https://developer.arm.com/-/media/Files/downloads/gnu-rm/7-2017q4/gcc-
    arm-none-eabi-7-2017-q4-major-linux.tar.bz2?revision=375265d4-e9b5-41c8-
    bf23-56cbe927e156?product=GNU%20Arm%20Embedded%20Toolchain,64-bit,,Linux
    ,7-2017-q4-major"
```

Lo primero que es necesario hacer es crear el directorio de instalación. Luego, descargar y descomprimir el compilador GCC. En el código 3.7 se muestra también como se guarda la dirección en otra variable por comodidad.

Código 3.7 Instalación del compilador GCC.

```
mkdir $installDir

wget $gccURL -t 3 -O $installDir/gcc-arm-none-eabi.tar.bz2 # Descarga

tar xvjf $installDir/gcc-arm-none-eabi.tar.bz2 -C $installDir # Extracción

rm $installDir/*.tar.bz2
mv $installDir/gcc-arm-none-eabi* $installDir/gcc-arm-none-eabi
gccPath="$installDir/gcc-arm-none-eabi"
```

Luego será necesario instalar varios paquetes antes de la propia librería Tensorflow. Para ello será necesario actualizar la versión en el entorno tal como se muestra en el código 3.8, ya que la plataforma Ubuntu 18.04 elegida tiene una demasiado antigua para nuestros propositos. La versión de Tensorflow elegida es la 1.7.0 que, como ya se comentó con anterioridad, no es de las más nuevas, pero no es en aboluto necesario que lo sea.

Código 3.8 Instalación de paquetes y librerías Tensorflow.

```
sudo python3 -m pip install --upgrade pip
sudo pip install scipy
sudo pip install matplotlib
sudo pip install opencv-python

sudo pip --no-cache-dir install tensorflow==1.7.0 # Instalación de Tensorflow
```

En último lugar, se instalarán las herramientas uTensor[17] y Mbed CLI[6], necesaria para el manejo de repositorios, y se configurará el path como se muestra en el código 3.9.

Código 3.9 Instalación de uTensor y Mbed CLI.

```
sudo pip install utensor_cgen # Instalación de uTensor

# Instalación de Mbed

sudo pip install intelhex prettytable junit_xml beautifulsoup4 fuzzywuzzy
sudo pip install pyelftools jsonschema mbed_host_tests future
sudo pip install mbed-cli

mbed config -G GCC_ARM_PATH "$gccPath/bin"
```

3.3 Compilación y ejecución del ejemplo MNIST

Si el entorno se ha configurado correctamente no debe haber ningún problema para compilar y ejecutar el ejemplo MNIST[13] de la manera que se describe a continuación.

Ya que prácticamente todos los archivos necesario se encuentran online, lo primero será importar el proyecto con la línea de código 3.10.

Código 3.10 Importa el proyecto desde GitHub.

```
$ mbed import https://github.com/uTensor/utensor-mnist-demo
```

A partir de aquí es posible seguir un tutorial incluido en el GitHub de uTensor que utiliza Jupyter Notebook[4], una herramienta gráfica que ayuda a controlar la ejecución de scripts de Python en tiempo real. Sin embargo, dado que posteriormente se modificarán los ficheros que van a ser ejecutados, parece de mayor utilidad eliminar intermediarios cuanto antes y conocer de primera mano el sistema de directorios.

Para entrenar la red neuronal descrita en el fichero deep_mlp.py habrá que dirigirse a environment/utensor-mnist-demo/tensorflow-models/ y ejecutar el comando python deep_mlp.py

Esto nos ofrecerá la salida mostrada en el código 3.11, que describe la probabilidad de acierto del modelo durante los pasos del entrenamiento.

Código 3.11 Probabilidad de acierto del modelo de red neuronal.

```
step 1000, training accuracy 0.58
step 2000, training accuracy 0.76
step 3000, training accuracy 0.84
step 4000, training accuracy 0.82
step 5000, training accuracy 0.86
step 6000, training accuracy 0.92
step 7000, training accuracy 0.94
step 8000, training accuracy 0.92
step 9000, training accuracy 0.84
step 10000, training accuracy 0.84
step 11000, training accuracy 0.88
step 12000, training accuracy 0.88
step 13000, training accuracy 0.92
step 14000, training accuracy 0.92
step 15000, training accuracy 0.88
step 16000, training accuracy 0.9
step 17000, training accuracy 0.88
step 18000, training accuracy 0.86
step 19000, training accuracy 0.98
step 20000, training accuracy 0.98
test accuracy 0.9251
saving checkpoint: chkps/mnist_model
Converted 6 variables to const ops.
written graph to: mnist_model/deep_mlp.pb
the output nodes: ['y_pred']
```

En la propia salida se informa de dónde se ha generado el modelo deep_mlp.pb, que será el que se convertirá a C++ mediante la herramienta uTensor con el comando 3.12 ejecutado desde la raíz del proyecto.

Código 3.12 Comando uTensor.

```
$ utensor-cli convert tensorflow-models/mnist_model/deep_mlp.pb --output-nodes=
    y_pred
```

Esto generará los archivos deep_mlp.cpp, deep_mlp.hpp y deep_mlp_weight.hpp en el directorio /models.

Por último, será necesario generar el utensor-mnist-demo.bin que se cargará en el sistema empotrado a partir de estos archivos junto con el main.cpp del directorio /utensor. El código 3.13 lo compilará en el directorio /BUILD/DISCO_F413ZH/GCC_ARM

Código 3.13 Generación de utensor-mnist-demo.bin.

```
$ deploy mbed
$ mbed compile -m DISCO_F413ZH -t GCC_ARM
```

Al arrastrar este archivo a la carpeta del equipo se reiniciará y cargará el nuevo programa. Las siguientes imágenes muestran varios ejemplos del programa una vez funcionando.

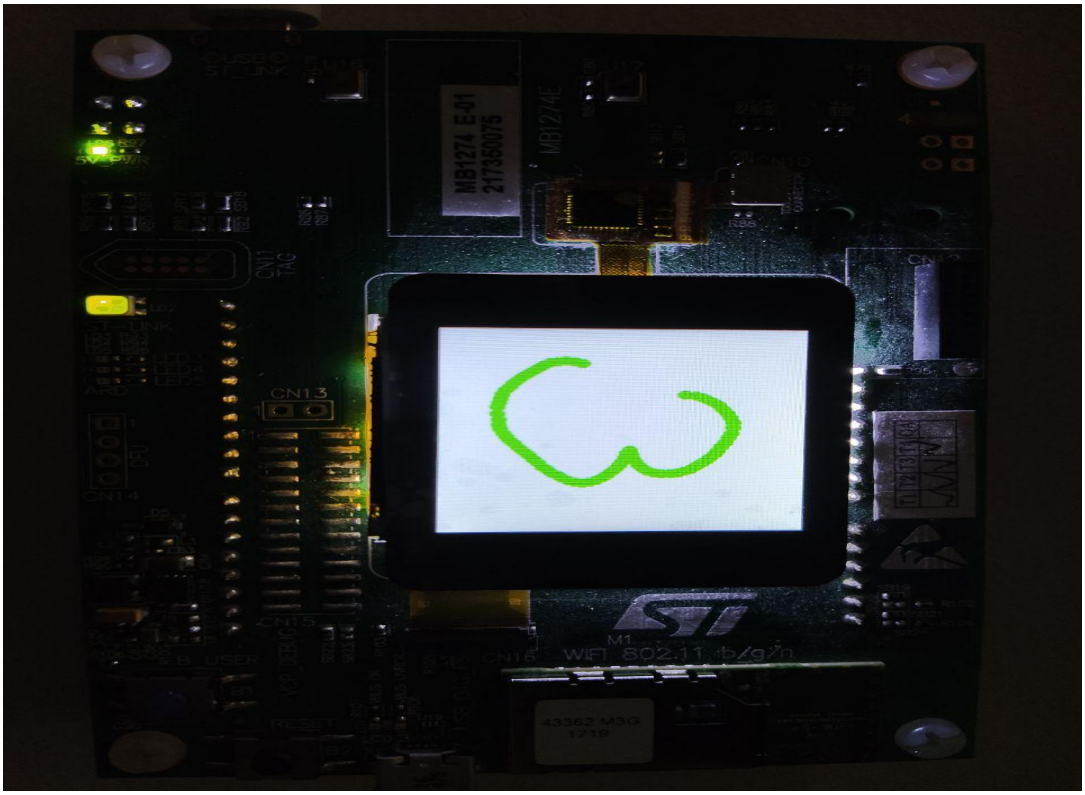


Figura 3.8 Cifra "3" dibujada.

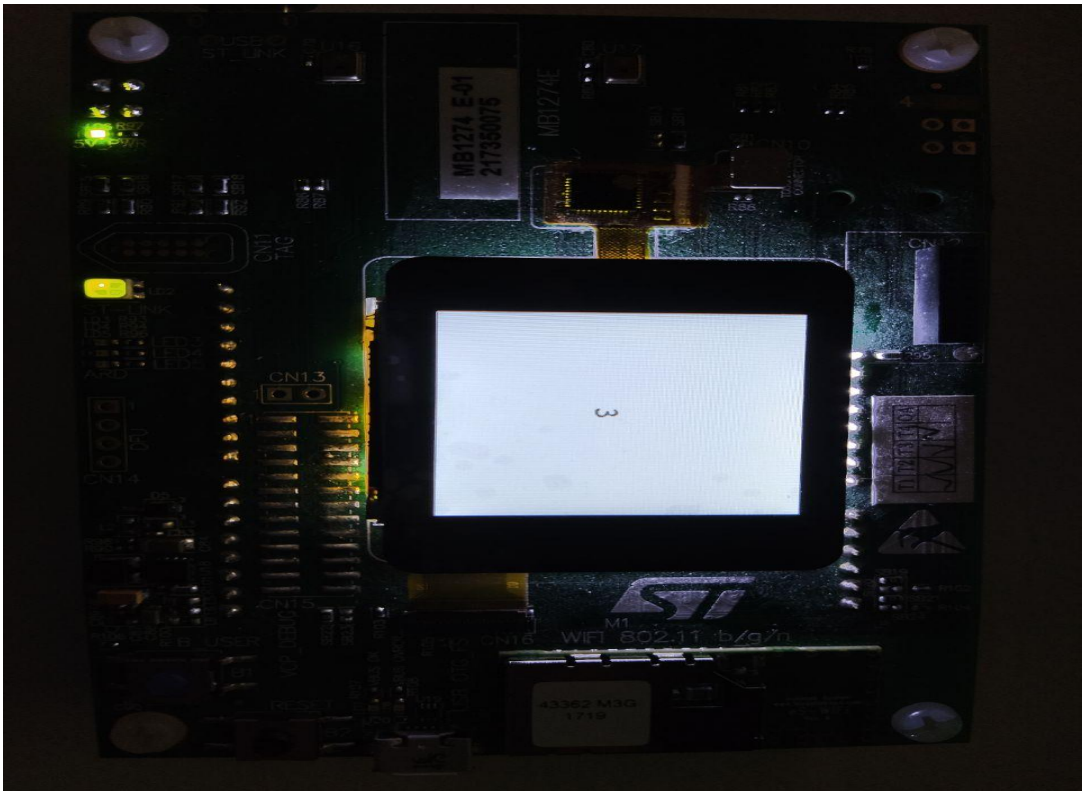


Figura 3.9 Resultado de cifra "3".

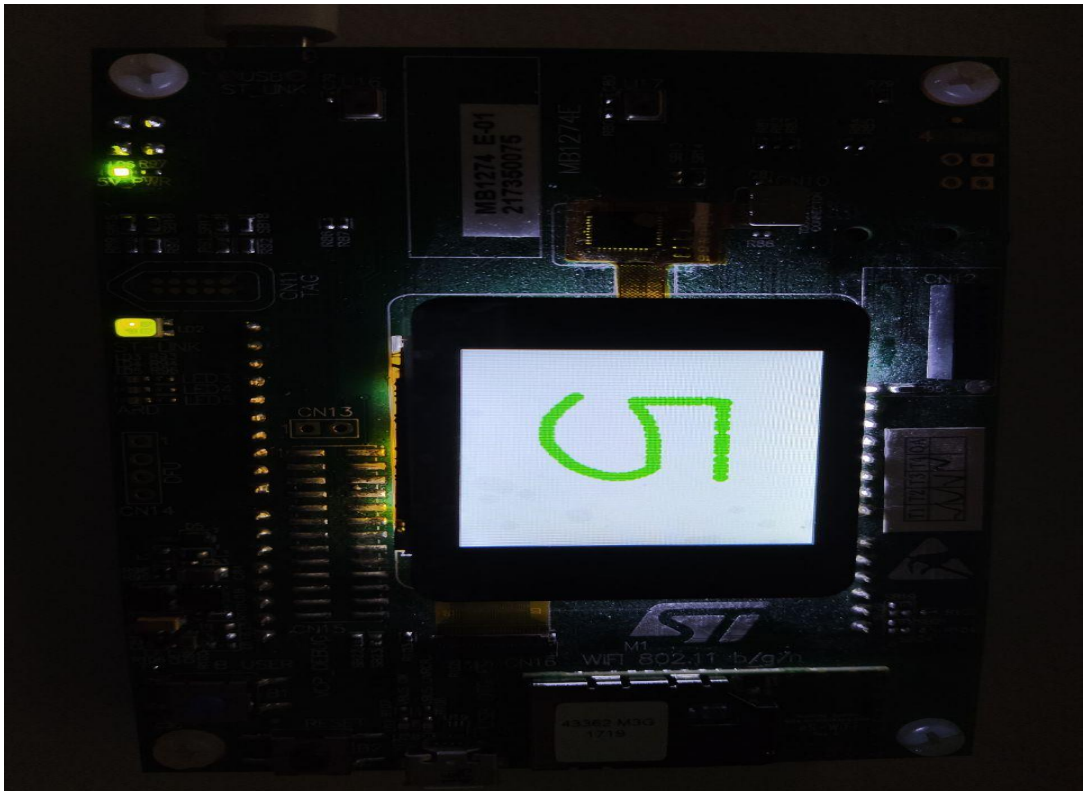


Figura 3.10 Cifra "5" dibujada.

Figura 3.11 Resultado de cifra "5".

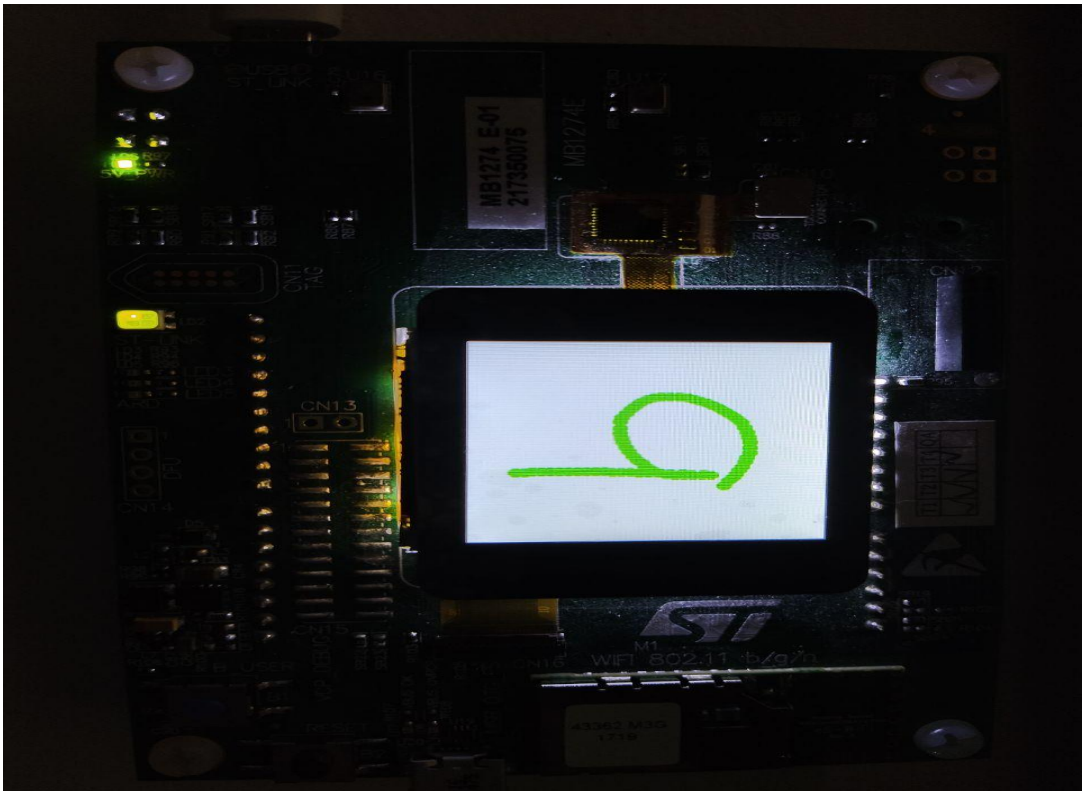


Figura 3.12 Cifra "9" dibujada.

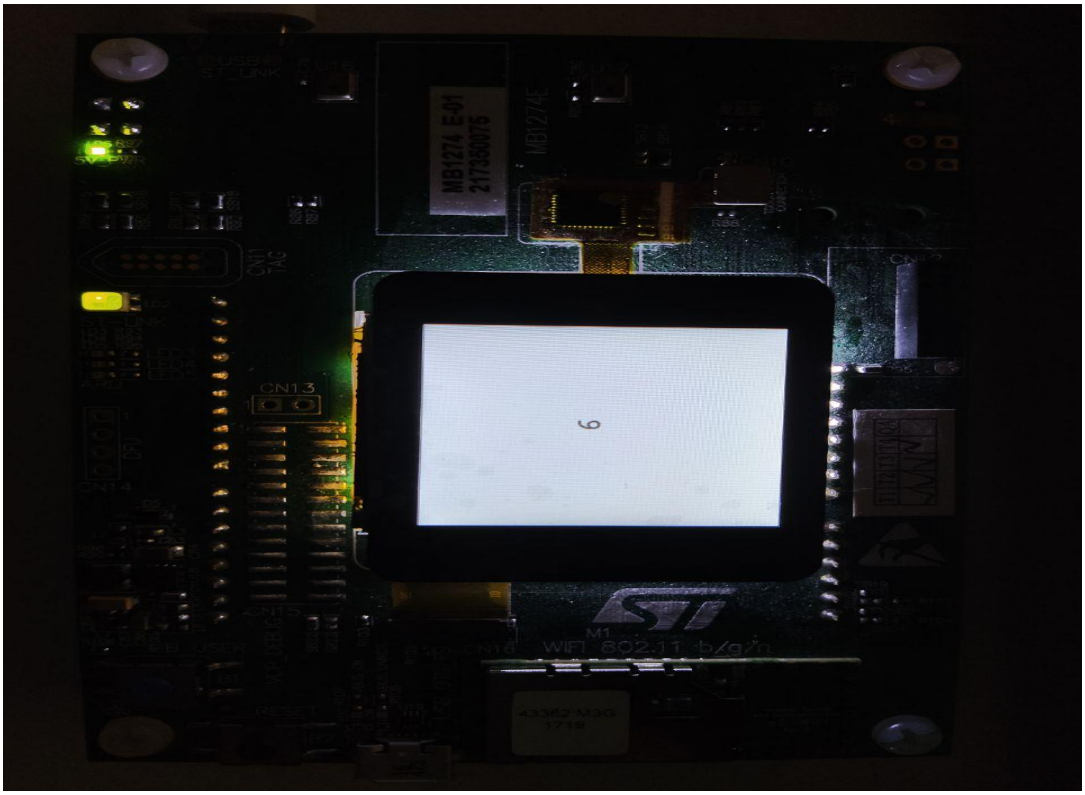


Figura 3.13 Resultado de cifra "9".

4 Implementación de Técnicas de Aprendizaje Automático

En este pasillo ya somos casi todos ingenieros

Mi responsable, 2020

En este capítulo se aplicarán una serie de técnicas para mejorar los resultados de lo realizado anteriormente. En primer lugar se trabajará sobre los datos de entrada para el entrenamiento de la red, optimizándolos para el uso final que vamos a darles. Se modificará también la construcción de la red neuronal, mejorando la aplicación en algunos aspectos y comprometiendo otros. Y, por último, se ampliará la funcionalidad del ejemplo posibilitando el reconocimiento de dos cifras en lugar de solo una.

4.1 Preprocesado de datos

Una de las partes más complejas y a la vez necesarias del Machine Learning es el preprocesamiento de los datos. Para entrenar una red neuronal son necesarios una enorme cantidad de datos recogidos, en muchas ocasiones, en entornos diferentes y bajo diferentes variables. Actualmente, las empresas aprovechan cualquier conjunto de datos recogidos en masa para mejorar sus sistemas mediante el aprendizaje automático. En la mayoría de ocasiones, la razón para la recogida de esos datos no era entrenar ninguna red neuronal, por lo que es necesario tratarlos y normalizarlos antes de que sean útiles para ello.

El procesamiento de los datos de entrenamiento de la red para este caso ya está bastante trabajado, como se explicó en el capítulo anterior. Las imágenes están acompañadas de sus respectivas etiquetas y antes del entrenamiento de la red se reordenan para introducirlos en los nodos de la primera capa de la misma. Hay, sin embargo, alguna mejora que puede hacerse para el uso concreto que queremos darle.

Las imágenes para el entrenamiento están en escala de grises, mientras que la imagen creada mediante la pantalla táctil solo estará en blanco y negro. Esto invita a pensar que una red neuronal entrenada con imágenes en blanco y negro cumpliría mucho mejor su labor. Existe otra opción que implica modificar la recogida de datos de la pantalla. Podría implementarse una forma de que, por ejemplo, se interpreten como grises los puntos más alejados del lugar de contacto con la pantalla. Sin embargo, parece de mayor interés trabajar sobre la primera línea. Modificar los datos de entrenamiento no es una labor trivial, es necesario entender en qué formato están los datos inicialmente y por qué etapas pasan.

La escala de grises se representa en cada pixel mediante valores float entre 0 y 1, donde el 0 es blanco y el 1 el negro. Durante el entrenamiento, la red se alimenta con datos separados en lotes (cada uno conocido como batch, del inglés), por lo que los datos entran en una matriz de vectores de 728 valores float. Existe una librería en Python que se usa a menudo junto con Tensorflow, ya que facilita mucho el tratamiento de este tipo de datos, la librería NumPy. Importando esta librería (como np, para abreviar) se puede incluir en el código la línea 4.1 que convierta los datos de entrada en 1 si se encuentran sobre un margen y en 0 si se encuentran por debajo.

Código 4.1 Entrenamiento con datos en blanco y negro.

```
batch_images = np.float32(batch_images>0.4)
```

Realizando esta operación se está perdiendo información de las imágenes originales, por lo que es necesario realizar un estudio sobre los resultados del entrenamiento de la red dependiendo del valor que se use de margen para evitar que afecte negativamente. Para este punto se recuerda que la precisión del entrenamiento de la red original, observado en el código 3.11, era 0.9251. Esta precisión no será exáctamente la misma cada vez que se entrene la red, ya que en parte depende de los valores iniciales aleatorios de pesos de las neuronas, pero el segundo e incluso el tercer decimal pueden aportar información a tener encuenta.

Se estudiaron varios casos de los que se dejará la información en los códigos 4.2, 4.3, 4.4 y 4.5, entre los que al final se escogió el último caso. Todos los casos tardaron aproximadamente el mismo tiempo en ejecutarse, pero los resultados de precisión resumidos en la Tabla 4.1 justifican la elección del 0.4 para el márgen.

Tabla 4.1 Precisión según márgen blanco/negro.

Márgen de decisión blanco/negro	Precisión para valores de test
0.3	0.9267
0.4	0.9272
0.5	0.9232
0.7	0.9179

Código 4.2 Resultados del entrenamiento para margen en 0.5.

```
step 1000, training accuracy 0.64
step 2000, training accuracy 0.72
step 3000, training accuracy 0.74
step 4000, training accuracy 0.76
step 5000, training accuracy 0.94
step 6000, training accuracy 0.76
step 7000, training accuracy 0.8
step 8000, training accuracy 0.92
step 9000, training accuracy 0.9
step 10000, training accuracy 0.86
step 11000, training accuracy 0.92
step 12000, training accuracy 0.88
step 13000, training accuracy 0.92
step 14000, training accuracy 0.92
step 15000, training accuracy 0.88
step 16000, training accuracy 0.86
step 17000, training accuracy 0.92
step 18000, training accuracy 0.98
step 19000, training accuracy 0.92
step 20000, training accuracy 0.98
test accuracy 0.9232
saving checkpoint: chkps/mnist_model
Converted 6 variables to const ops.
written graph to: mnist_model/deep_mlp.pb
the output nodes: ['y_pred']
```

Código 4.3 Resultados del entrenamiento para margen en 0.7.

```
step 1000, training accuracy 0.44
step 2000, training accuracy 0.76
step 3000, training accuracy 0.84
step 4000, training accuracy 0.72
step 5000, training accuracy 0.82
step 6000, training accuracy 0.8
step 7000, training accuracy 0.9
step 8000, training accuracy 0.82
step 9000, training accuracy 0.8
step 10000, training accuracy 0.92
step 11000, training accuracy 0.92
step 12000, training accuracy 0.82
step 13000, training accuracy 0.94
step 14000, training accuracy 0.96
step 15000, training accuracy 0.78
step 16000, training accuracy 0.9
step 17000, training accuracy 0.86
step 18000, training accuracy 0.92
step 19000, training accuracy 0.86
step 20000, training accuracy 0.88
test accuracy 0.9179
saving checkpoint: chkps/mnist_model
Converted 6 variables to const ops.
written graph to: mnist_model/deep_mlp.pb
the output nodes: ['y_pred']
```

Código 4.4 Resultados del entrenamiento para margen en 0.3.

```
step 1000, training accuracy 0.74
step 2000, training accuracy 0.74
step 3000, training accuracy 0.74
step 4000, training accuracy 0.84
step 5000, training accuracy 0.84
step 6000, training accuracy 0.9
step 7000, training accuracy 0.88
step 8000, training accuracy 0.92
step 9000, training accuracy 0.88
step 10000, training accuracy 0.84
step 11000, training accuracy 0.82
step 12000, training accuracy 0.92
step 13000, training accuracy 0.94
step 14000, training accuracy 0.9
step 15000, training accuracy 0.94
step 16000, training accuracy 0.96
step 17000, training accuracy 0.88
step 18000, training accuracy 0.92
step 19000, training accuracy 0.96
step 20000, training accuracy 0.86
test accuracy 0.9267
saving checkpoint: chkps/mnist_model
Converted 6 variables to const ops.
written graph to: mnist_model/deep_mlp.pb
the output nodes: ['y_pred']
```

Código 4.5 Resultados del entrenamiento para margen en 0.4.

```
step 1000, training accuracy 0.58
step 2000, training accuracy 0.74
step 3000, training accuracy 0.82
step 4000, training accuracy 0.86
step 5000, training accuracy 0.86
step 6000, training accuracy 0.96
step 7000, training accuracy 0.9
step 8000, training accuracy 0.92
step 9000, training accuracy 0.92
step 10000, training accuracy 0.96
step 11000, training accuracy 0.84
step 12000, training accuracy 0.96
step 13000, training accuracy 0.86
step 14000, training accuracy 0.9
step 15000, training accuracy 0.94
step 16000, training accuracy 0.96
step 17000, training accuracy 0.9
step 18000, training accuracy 0.8
step 19000, training accuracy 0.86
step 20000, training accuracy 0.88
test accuracy 0.9272
saving checkpoint: chkps/mnist_model
Converted 6 variables to const ops.
written graph to: mnist_model/deep_mlp.pb
the output nodes: ['y_pred']
```

4.2 Modificaciones en la red

Este ejemplo concreto tiene una red bastante sencilla de apenas dos capas ocultas, que es como se conocen a las capas que no son la primera ni la última. La red la forman en su totalidad 986 neuronas, que dista mucho de ser un número alto para las redes neuronales habituales. Esto supone una ventaja a la hora de usar el programa en un dispositivo electrónico emportrado, que no tiene la misma memoria que una computadora multifunción. Para esta implementación, el archivo generado por uTensor que define la red neuronal en C++ (deep_mlp.cpp) pesa 22.25KB, el archivo que incluye los pesos pertenecientes a cada una de las neuronas (deep_mlp_weight.hpp) pesa 635.39KB, y el programa compilado que instalamos en la tarjeta, el archivo utensor-mnist-demo.bin pesa 261.36KB.

Sin embargo, para redes neuronales más complejas con un mayor número de capas ocultas y un mayor número de neuronas, el peso de estos archivos puede incrementarse hasta cantidades con las que un sistema empotrado no pueda trabajar. A la hora de realizar el diseño de una red neuronal, los analistas de datos suelen fijarse en muchas cosas, pero la capacidad computacional y de memoria reducida no es una de sus prioridades, ya que no acostumbran a tener limitaciones para ellas. Esto implica que, a la hora de usar una red neuronal para su uso en sistemas con menos recursos, es necesario una adaptación previa de la misma. Partiendo, como es el caso, de una red neuronal ya definida, la forma de simplificarla es elegir una capa intermedia y eliminarla. Luego se reentrena la red y se observa como ha afectado a la precisión de la misma. Dependiendo de la estructura de la red se puede elegir qué capa es la más indicada para eliminarse, la que al desaparecer afecta menos al funcionamiento de la red completa, pero lo más común es realizar un proceso de prueba y error basándose en las nociones que se tengan sobre el comportamiento de la red neuronal. Este procedimiento se realiza cuantas veces sea necesario hasta conseguir simplificar la red lo suficiente para que pueda usarse en el sistema que interese.

Por supuesto, es posible que al simplificar demasiado la red se pierda precisión en sus predicciones finales. Es labor del ingeniero llegar al compromiso que solucione mejor el problema. Quizás si la red simplificada no cumple con eficacia su cometido, merezca la pena invertir en un sistema con más recursos que permita una red más compleja.

En este punto se pretende estudiar, por interés didáctico más que por verdadera necesidad como ya se ha explicado, la capacidad de la estructura de la red neuronal de soportar la pérdida de una capa. Se vigilará la precisión de sus predicciones mientras se observa el beneficio en el decremento del peso de los archivos.

Para ello, se modifica el código 3.1 para sustituirlo por el siguiente código 4.6. En el nuevo código se elimina la primera capa oculta, siendo necesario modificar la capa de entrada para que sus salidas sean 64 en lugar de 128. Así mismo, en la segunda capa oculta (ahora, la única capa oculta), se modifica la entrada esperada para que provenga desde 784 nodos en lugar de desde 128.

Código 4.6 Red neuronal con solo una capa oculta.

```
def deepnn(x):
  W_fc1 = weight_variable([784, 64], name='W_fc1')
  b_fc1 = bias_variable([64], name='b_fc1')
  a_fc1 = tf.add(tf.matmul(x, W_fc1), b_fc1, name="zscore")
  h_fc1 = tf.nn.relu(a_fc1)
  layer1 = tf.nn.dropout(h_fc1, 0.50)

  W_fc3 = weight_variable([64, 10], name='W_fc3')
  b_fc3 = bias_variable([10], name='b_fc3')
  logits = tf.add(tf.matmul(layer1, W_fc3), b_fc3, name="logits")
  y_pred = tf.argmax(logits, 1, name='y_pred')

  return y_pred, logits
```

Al entrenar esta nueva red obtenemos los resultados mostrados en el código 4.7, con una precisión de 0.9166. Es una precisión claramente inferior a la anterior.

Código 4.7 Resultados del entrenamiento de la red con una capa menos.

```
step 1000, training accuracy 0.8
step 2000, training accuracy 0.9
step 3000, training accuracy 0.84
step 4000, training accuracy 0.94
step 5000, training accuracy 0.9
step 6000, training accuracy 0.8
step 7000, training accuracy 0.84
step 8000, training accuracy 0.88
step 9000, training accuracy 0.9
step 10000, training accuracy 0.86
step 11000, training accuracy 0.96
step 12000, training accuracy 0.88
step 13000, training accuracy 0.88
step 14000, training accuracy 0.84
step 15000, training accuracy 0.84
step 16000, training accuracy 0.86
step 17000, training accuracy 0.96
step 18000, training accuracy 0.88
step 19000, training accuracy 0.94
step 20000, training accuracy 0.92
test accuracy 0.9166
saving checkpoint: chkps/mnist_model
Converted 4 variables to const ops.
written graph to: mnist_model/deep_mlp.pb
the output nodes: ['y_pred']
```

Como una ventaja extra, la ejecución del archivo ha tardado 35.57 segundos, que es algo menos de lo que tardaba en entrenarse la red original. Esto es algo importante a tener en cuenta también para redes más grandes, y depende de los recursos de la máquina en la que se entrene. Puesto que el objetivo aquí se centra en trabajar con la red final en un equipo de bajas prestaciones tampoco se le dará mayor importancia, pero es digno de mención. Tras volver a realizar los pasos necesarios para obtener los nuevos archivos, deep_mlp.cpp pesa 16.13KB, deep_mlp_weight.hpp pesa 301.49KB, y el archivo utensor-mnist-demo.bin pesa 259.42KB.

Se ha visto por tanto que la diferencia de peso no es apenas apreciable para el equipo que se está usando, mientras que la perdida de precisión sí perjudica bastante a la aplicación final. La modificación que sí es de verdadera utilidad es en realidad justo la contraria. Conociendo que tenemos margen para aumentar el peso de los archivos, será de gran utilidad añadir una capa al modelo para que este aumente su precisión final. Para ello se debe modificar de nuevo la definición de la red y usar la que se encuentra en el código 4.9.

Código 4.8 Red neuronal con tres capas ocultas.

```
def deepnn(x):
  W_fc1 = weight_variable([784, 392], name='W_fc1')
  b_fc1 = bias_variable([392], name='b_fc1')
  a_fc1 = tf.add(tf.matmul(x, W_fc1), b_fc1, name="zscore")
  h_fc1 = tf.nn.relu(a_fc1)
  layer1 = tf.nn.dropout(h_fc1, 0.50)

  W_fc2 = weight_variable([392, 128], name='W_fc2')
  b_fc2 = bias_variable([128], name='b_fc2')
  a_fc2 = tf.add(tf.matmul(layer1, W_fc2), b_fc2, name="zscore")
  h_fc2 = tf.nn.relu(a_fc2)
  layer2 = tf.nn.dropout(h_fc2, 0.50)

  W_fc3 = weight_variable([128, 64], name='W_fc3')
  b_fc3 = bias_variable([64], name='b_fc3')
  a_fc3 = tf.add(tf.matmul(layer2, W_fc3), b_fc3, name="zscore")
  h_fc3 = tf.nn.relu(a_fc3)
  layer3 = tf.nn.dropout(h_fc3, 0.50)

  W_fc4 = weight_variable([64, 10], name='W_fc4')
  b_fc4 = bias_variable([10], name='b_fc4')
  logits = tf.add(tf.matmul(layer3, W_fc4), b_fc4, name="logits")
  y_pred = tf.argmax(logits, 1, name='y_pred')

  return y_pred, logits
```

Como se obserba en el código 4.9, la precisión de la red con una capa más asciende a 0.9473.

Código 4.9 Resultados del entrenamiento de la red con una capa más.

```
step 1000, training accuracy 0.56
step 2000, training accuracy 0.64
step 3000, training accuracy 0.8
step 4000, training accuracy 0.86
step 5000, training accuracy 0.76
step 6000, training accuracy 0.88
step 7000, training accuracy 0.86
step 8000, training accuracy 0.94
step 9000, training accuracy 0.86
step 10000, training accuracy 0.9
step 11000, training accuracy 0.9
```

```
step 12000, training accuracy 0.94
step 13000, training accuracy 0.92
step 14000, training accuracy 0.98
step 15000, training accuracy 0.92
step 16000, training accuracy 0.96
step 17000, training accuracy 0.96
step 18000, training accuracy 0.98
step 19000, training accuracy 0.92
step 20000, training accuracy 0.9
test accuracy 0.9473
saving checkpoint: chkps/mnist_model
Converted 8 variables to const ops.
written graph to: mnist_model/deep_mlp.pb
the output nodes: ['y_pred']
```

Para este caso, el entrenamiento de la red se ha alargado en el tiempo hasta los 100.49 segundos. El archivo final deep_mlp.cpp pesa 28.38KB, deep_mlp_weight.hpp pesa 2.05MB, y el archivo utensor-mnist-demo.bin pesa 264.19KB. Son valores más que aceptables para el equipo, por lo que se elegirá esta red en la versión final del programa.

Como resumen de los resultados obtenidos, en la Tabla 4.2 puede observarse con mayor claridad la evolución del tiempo de entrenamiento, el peso de los archivos y la precisión según el número de capas.

Tabla 4.2 Datos según número de capas ocultas.

Número de capas	Precisión test	Tiempo entrenamiento	Archivo red	Archivo de pesos
1	0.9166	35.57s	16.13KB	301.49KB
2	0.9267	61.56s	22.25KB	635.39KB
3	0.9473	100.49s	28.38KB	20.5 MB

Cabe destacar que pueden seguir añandiéndose capas para mejorar la precisión y, sin embargo, la forma más adecuada de potenciar la red neuronal es cambiando el modelo por un modelo de red neuronal convolucional. Esto no se está haciendo porque este modelo no se encuentra dentro de aquellos con los que la herramienta uTensor está preparada para trabajar.

4.3 Mejoras en la funcionalidad

En este punto se describe la programación de una evolución en la funcionalidad del programa. Se trabajará sobre los archivos main.cpp e image.h para que el equipo sea capaz de detectar la escritura de un número de dos cifras.

Para ello se va a suponer que cada cifra se encuentra en una mitad de la pantalla, se separará la pantalla en dos y se procesará cada mitad como una imagen diferente. Esto implica trabajar con la imagen que se obtiene de la pantalla y procesarla antes de hacerla pasar por la red neuronal, por lo que se desarrollarán nuevas funciones específicas para ello en el archivo image.h.

Las funciones cutleft y cutright, mostradas en el los códigos 4.10 y 4.11 respectivamente cogen cada una el lado izquierdo y derecho de una imagen de 28x28 píxeles, trazando la línea en el pixel 14 del eje horizontal. Recorre la matriz de valores uint8_t que representa la imagen original y genera una nueva matriz que contiene solo los valores que interesan.

Código 4.10 Función para cortar el lado izquierdo de la imagen.

```
/**
 * @brief Cut the image
 * @details Take left half of the image
 *
```

```
 * @param img [description]
 * @tparam T Probably uint8_t
 * @return Cutted image
 */
template<typename T>
Image<T> cutleft(const Image<T>& img){
  int xMin, xMax, yMin, yMax;
  xMin = 0;
  yMin = 0;
  xMax = 13;
  yMax = 27;
  printf("Cutting image side = %d, %d, %d, %d\n", xMin, yMin, xMax, yMax);

  Image<T> temp(xMax-xMin, yMax-yMin);

  for(int i=0, ii=xMin; ii < xMax; i++, ii++){
    for(int j=0, jj=yMin; jj < yMax; j++, jj++){
      temp(i,j) = img(ii,jj);
    }
  }

  return temp;
}
```

Código 4.11 Función para cortar el lado derecho de la imagen.

```
/**
 * @brief Cut the image
 * @details Take right half of the image
 *
 * @param img [description]
 * @tparam T Probably uint8_t
 * @return Cutted image
 */
template<typename T>
Image<T> cutright(const Image<T>& img){
  int xMin, xMax, yMin, yMax;
  xMin = 14;
  yMin = 0;
  xMax = 27;
  yMax = 27;
  printf("Cutting image side = %d, %d, %d, %d\n", xMin, yMin, xMax, yMax);

  Image<T> temp(xMax-xMin, yMax-yMin);

  for(int i=0, ii=xMin; ii < xMax; i++, ii++){
    for(int j=0, jj=yMin; jj < yMax; j++, jj++){
      temp(i,j) = img(ii,jj);
    }
  }

  return temp;
}
```

Estas dos funciones estan preparadas para trabajar con imágenes ya reducidas al tamaño apropiado para la red, pero su salida son imágenes de tamaño 14x28 píxeles, por lo que se modificará su aspecto con las líneas del código 4.12 usando otra función de image.h, la función resize cuya definición se observa en el código 4.13.

Código 4.12 Líneas para redimensionar la imagen.

```
Image<float> smallImage1 = resize(img1, 28, 28);
Image<float> smallImage2 = resize(img2, 28, 28);
```

Esta función se usa también al disminuir el tamaño de la imagen original desde la pantalla, que tiene bastantes más que 28x28 píxeles.

Código 4.13 Función para redimensionar la imagen.

```
/**
 * @brief Nearest interpolation
 * @details Stretch or shrink an image naively
 *
 * @param img [description]
 * @param w2 new width
 * @param h2 new height
 * @tparam T [description]
 * @return [description]
 */
template<typename T>
Image<T> resize(const Image<T>& img, int w2, int h2){
    Image<T> temp(w2,h2);
    int x_ratio = (int)((img.get_xDim()<<16)/w2) +1;
    int y_ratio = (int)((img.get_yDim()<<16)/h2) +1;
    int x2, y2 ;

    for(int i = 0; i < h2; i++) {
        for(int j = 0; j < w2; j++) {
            x2 = ((j*x_ratio)>>16) ;
            y2 = ((i*y_ratio)>>16) ;
            temp[(i*w2) + j] = img[(y2*img.get_xDim()) + x2] ;
        }
    }
    return temp;

}
```

Lo que resta a continuación es adaptar el programa para que aplique la red neuronal sobre ambas imágenes en lugar de sobre una sola. Tras las modificaciones necesarias, la interrupción al levantar el botón quedaría como el código 4.14.

Código 4.14 Código de la interrupción.

```
            Image<float> smallImage = resize(*img, 28, 28);

            pc.printf("Done padding\n\n");
            delete img;
```

```
            Image<float> img1 = cutleft(smallImage);
            Image<float> img2 = cutright(smallImage);

            Image<float> smallImage1 = resize(img1, 28, 28);
            Image<float> smallImage2 = resize(img2, 28, 28);

            pc.printf("Reshaping\n\r");
            smallImage1.get_data()->resize({1, 784});
            pc.printf("Creating Graph\n\r");

            get_deep_mlp_ctx(ctx, smallImage1.get_data());
            pc.printf("Evaluating\n\r");
            ctx.eval();
            S_TENSOR prediction = ctx.get({"y_pred:0"});
            int result1 = *(prediction->read<int>(0,0));

            pc.printf("Reshaping\n\r");
            smallImage2.get_data()->resize({1, 784});
            pc.printf("Creating Graph\n\r");

            printf("Number guessed %d\n\r", result1);

            get_deep_mlp_ctx2(ctx2, smallImage2.get_data());
            pc.printf("Evaluating\n\r");
            ctx2.eval();
            prediction = ctx2.get({"y_pred:0"});
            int result2 = *(prediction->read<int>(0,0));

            printf("Number guessed %d\n\r", result2);

            BSP_LCD_Clear(LCD_COLOR_WHITE);
            BSP_LCD_SetTextColor(LCD_COLOR_BLACK);
            BSP_LCD_SetFont(&Font24);

            uint8_t number[2];
            number[2] = '\0';
            //ASCII numbers are 48 + the number, a neat trick
            number[0] = 48 + result1;
            number[1] = 48 + result2;
            BSP_LCD_DisplayStringAt(0, 120, number, CENTER_MODE);
            trigger_inference = false;
            img = new Image<float>(28, 28);
            clear(*img);
            break;
```

Se compilará usando las líneas ya mencionadas en el código 3.13 obtendremos nuevos resultados como el mostrados en Figura 4.2 a partir de la escritura de Figura 4.1.

4.4 Información adicional

Si bien durante casi todo el proceso, en la mayoría de las pruebas realizadas se ha obtenido un resultado positivo en la detección de la cifra buscada, por supuesto se encuentran errores.

Al observar las imagenes con las que se ha entrenado la red se puede indentificar cierto parecido entre las cifras que representan un mismo número. Esto hace que, en ocasiones, números escritos de manera diferente a la normalizada en el entrenamiento presenten resultados incorrectos. En contraposición al número "24" obtenido en las imágenes Figura 4.1 y Figura 4.2, se muestran a continuación la imagen Figura 4.3, que

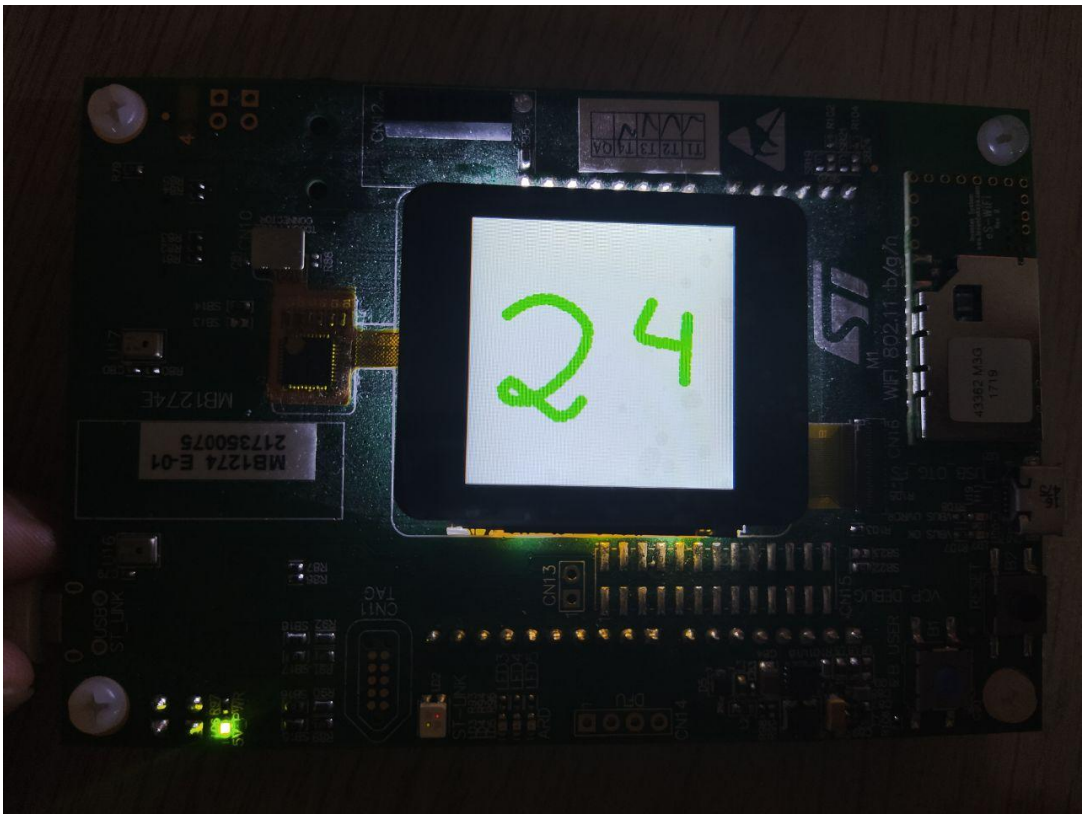


Figura 4.1 Imagen de entrada con un número de dos cifras.

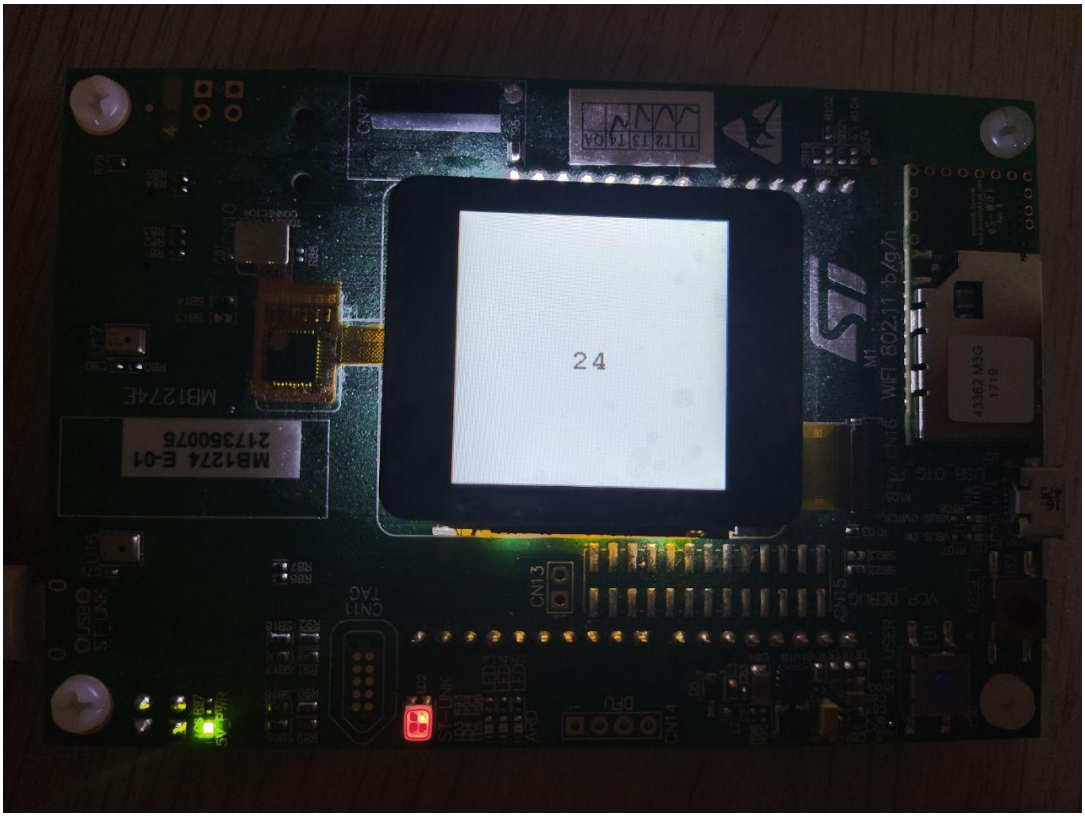


Figura 4.2 Resultado de la imagen de entrada con un número de dos cifras.

aporta el resultado mostrado en Figura 4.4 con una interpretación errónea por una escritura diferente del número "4".

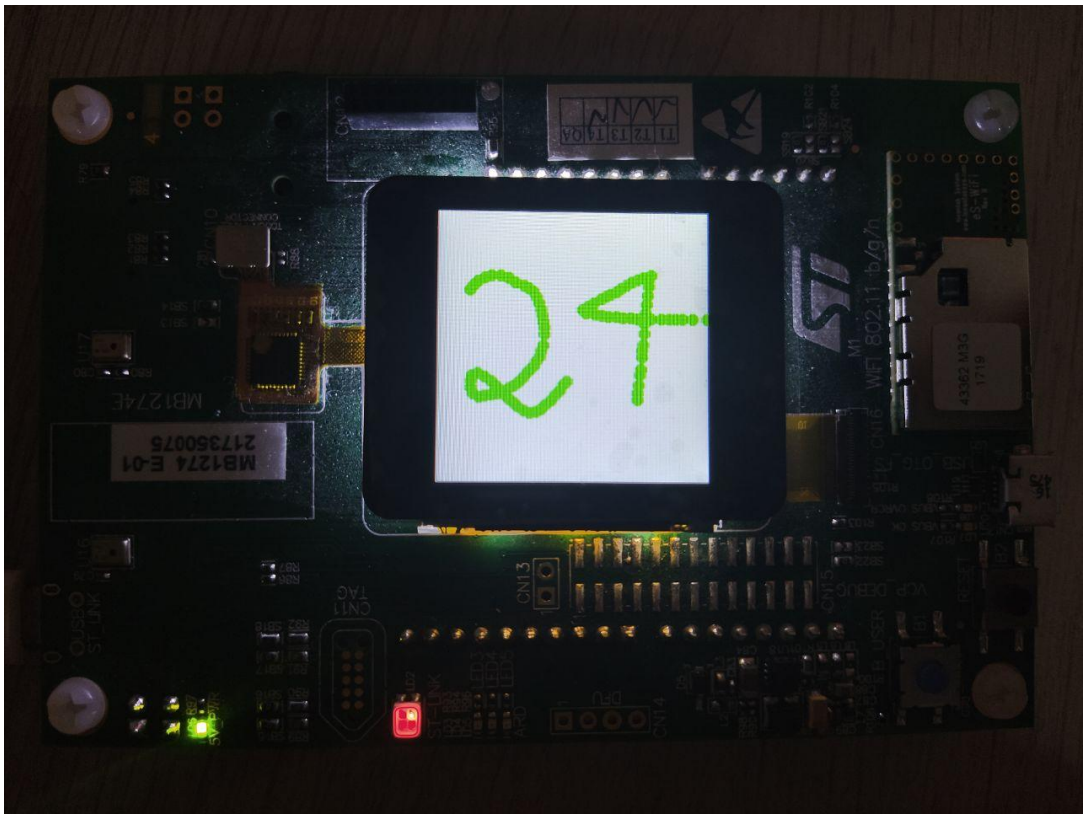


Figura 4.3 Imagen de entrada con un número de dos cifras pobremente definido.

También es de señalar el comportamiento que tiene la red cuando le introduces una imagen completamente en blanco. La cifra que la red interpreta que es más probable que sea es un "7", y como no se ha programado ningún filtro que impida que dicha entrada sea aceptada, la Figura 4.5 se entiende como Figura 4.6.

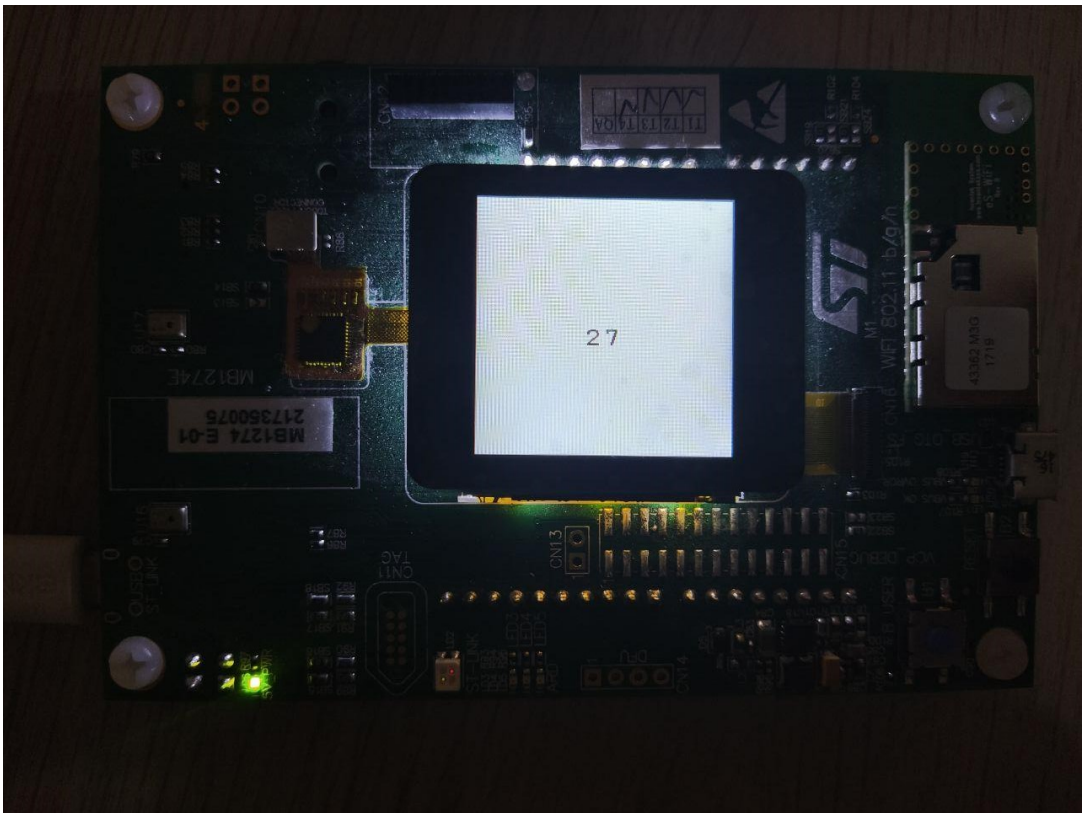


Figura 4.4 Resultado de la imagen de entrada con un número de dos cifras pobremente definido.

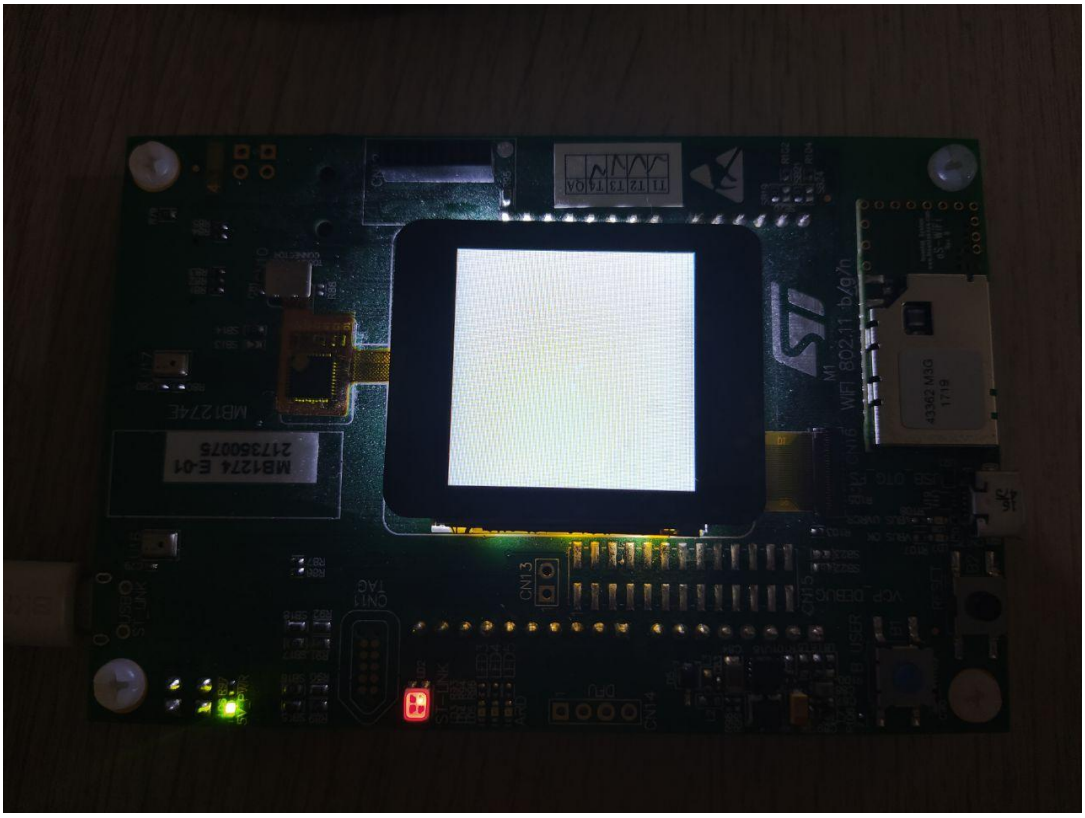


Figura 4.5 Imagen de entrada vacía.

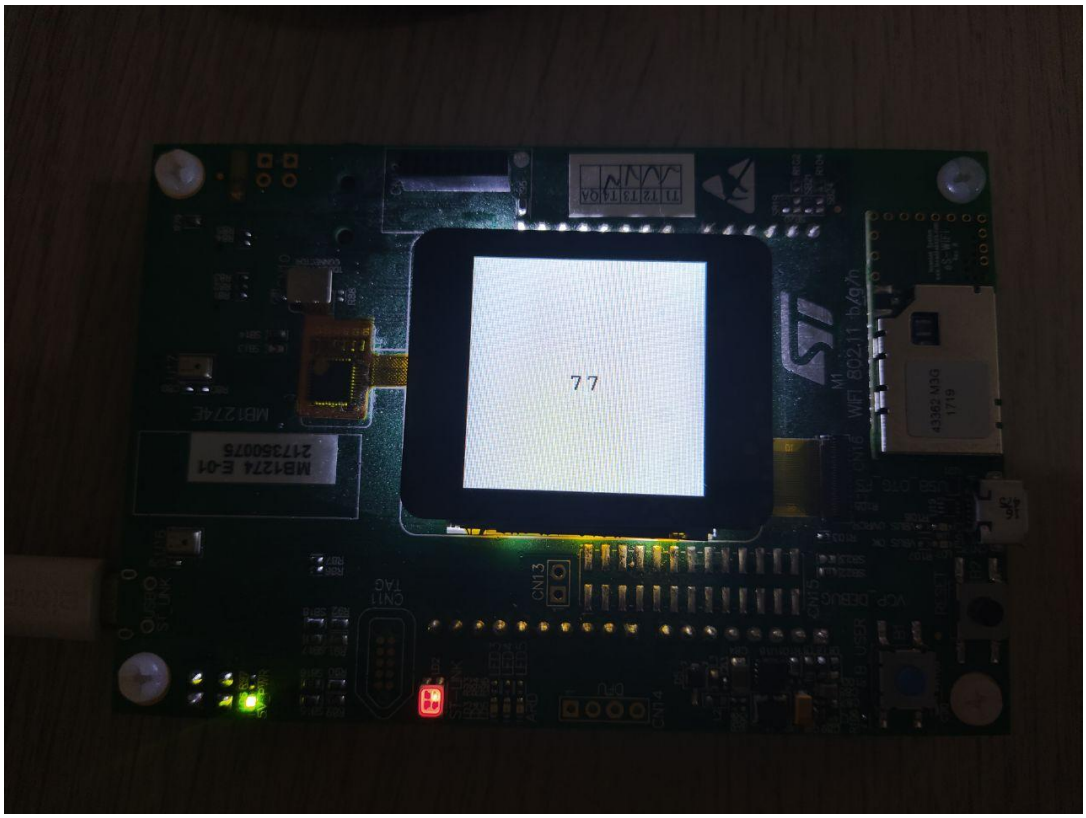


Figura 4.6 Resultado de una imagen de entrada vacía.

5 Conclusiones y Desarrollo Futuro

Entonces te pongo la fecha del compromiso de finalización de la carrera para 2020

La chica de recursos humanos, 2019

En este capítulo se comentarán las conclusiones del trabajo y los estudios realizados y los posibles desarrollos que podrían mejorar el mismo.

5.1 Conclusiones

Se ha demostrado que es posible introducir con eficacia una red neuronal sencilla en un sistema embebido utilizando las herramientas adecuadas. Parece que uTensor tenga bastante potencial y que, de hecho, aún pueden estudiarse sistemas bastante más complejos hasta conocer sus límites. El hecho de que se abran las puertas al procesado de imágenes a través de redes neuronales en este tipo de dispositivos implica el acceso a una enorme cantidad de posibilidades. Podrían trabajarse aplicaciones para reconocimiento facial o para reconocimiento de sonidos y palabras (el reconocimiento de sonidos en redes neuronales se trata como reconocimiento de imágenes del espectro de audio) de manera independiente sin necesidad de que el equipo que recoge la imagen mantenga conexión constante con un sistema de apoyo.

5.2 Desarrollo futuro

Respecto a desarrollos futuros para mejorar esta aplicación concreta, la primera y que debe ser más sencilla de utilizar es la detección de pantalla vacía, para evitar lo mostrado en Figura 4.5 y Figura 4.6. Es un detalle que parece carecer de importancia en pruebas controladas pero que debe pulirse si se quiere realizar una aplicación funcional.

También podría mejorarse la aplicación si, en lugar de fijar un margen en el centro de la pantalla para separar la cifra de la izquierda de la cifra de la derecha, se implementara una función que detectara el margen en la separación de las cifras. Esto implicaría también la modificación del tamaño de las imágenes con una relación de aspecto dependiente de donde se hubiera colocado ese margen en cada caso.

En caso de no realizarse la implementación de este margen variable, sería planteable entrenar la red neuronal con imágenes ya recortadas con la relación de aspecto de media pantalla, eliminando los bordes laterales de las imágenes de entrenamiento. Todas las imágenes tienen la cifra centrada, por lo que no debe implicar pérdida de información.

Trabajar con la recogida de información de la pantalla en escala de grises también es algo que no se ha explotado. Existen algoritmos para dibujar un círculo, centrado en el punto de contacto, que se vaya degradando hacia su contorno. Este algoritmo debe preocuparse también de controlar lo que sucede si un trazo superpone a otro, ya que no es válido reemplazarlo, sería necesario sumar los diferentes grises.

Por último, sería interesante buscar una alternativa a uTensor que sí permita la implementación de redes neuronales convolucionales. Esto es un salto de calidad para casi cualquier aplicación de aprendizaje automático y es un obstáculo a medida que se intentan desarrollos más complejos.

Apéndice A
32F413HDISCOVERY Discovery kit

A.1 Datasheet

life.augmented

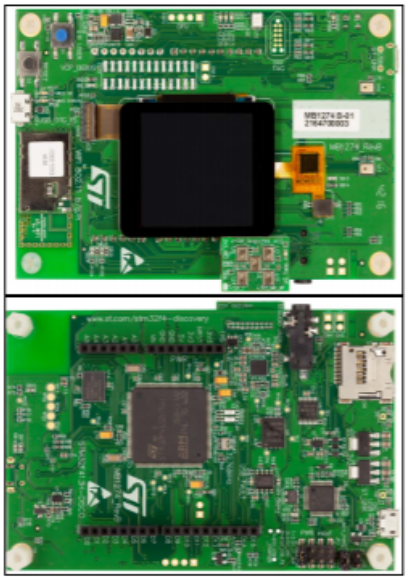
32F413HDISCOVERY

Discovery kit with STM32F413ZH MCU

Data brief

Features

- STM32F413ZHT6 microcontroller featuring 1.5 Mbytes of Flash memory and 320 Kbytes of SRAM, in LQFP144 package
- On-board ST-LINK/V2-1 supporting USB re-enumeration capability
- USB ST-LINK functions:
  - Virtual COM port
  - Mass storage
  - Debug port
- 240x240-pixel LCD with parallel interface and touch-panel connector
- 8-Mbit PSRAM; 512K word x 16bits
- 128-Mbit Quad-SPI Flash memory
- I^2S audio codec
- Jack connector for Audio line with microphone input and stereo output
- Two on-board ST-MEMS microphones
- Connector extension for up to 5-MEMS microphones.
- USB OTG FS with Micro-AB connector
- Connector for microSD™ card
- Integrated Wi-Fi® module 802.11 b/g/n
- Two push-buttons (user and reset)
- Two user LEDs: one green and one red
- Arduino™ Uno V3 connectors
- Four power supply options:
  - ST LINK/V2-1
  - USB FS connector
  - 5 V from Arduino™ Uno V3
  - USB charger
- Comprehensive free software including a variety of examples, part of the STM32Cube package
- Support of a wide choice of integrated development environments



1. Pictures are not contractual.

Description

The 32F413HDISCOVERY Discovery kit allows users to develop applications with the STM32F4 Series microcontrollers based on ARM® Cortex®-M4 core.

The 32F413HDISCOVERY Discovery kit enables a wide diversity of applications, taking benefit from audio, multi-sensor support, graphics, security, video and high-speed connectivity features.

The Arduino™ Uno V3 connectivity support provides unlimited expansion capabilities with a large choice of specialized add-on boards.

Apéndice B
Códigos principales

B.1 Archivo main.cpp

Código B.1 Archivo main.cpp.

```
#include "mbed.h"
#include "stm32f413h_discovery.h"
#include "stm32f413h_discovery_ts.h"
#include "stm32f413h_discovery_lcd.h"
#include "tensor.hpp"
#include "image.h"
#include "models/deep_mlp.hpp"

Serial pc(USBTX, USBRX, 115200);

#ifndef TARGET_SIMULATOR
#include "FATFileSystem.h"
#include "F413ZH_SD_BlockDevice.h"
F413ZH_SD_BlockDevice bd;
FATFileSystem fs("fs");
#else
#define USER_BUTTON   BUTTON1
#endif

InterruptIn button(USER_BUTTON);

TS_StateTypeDef TS_State = {0};

volatile bool trigger_inference = false;

void trigger_inference_cb(void){ trigger_inference = true; }

template<typename T>
void clear(Image<T>& img){
    for(int i = 0; i < img.get_xDim(); i++){
        for(int j = 0; j < img.get_yDim(); j++){
            img(i,j) = 0;
        }
    }
}
```

```
template<typename T>
void printImage(const Image<T>& img){

    for(int i = 0; i < img.get_xDim(); i++){
        for(int j = 0; j < img.get_yDim(); j++){
            printf("%f, ", img(i,j));
        }
        printf("]\n\r");
    }
}

// /**
// * Simple box filter with extra weight on the center element.
// * Blurs the image to make it more realistic.
// */
// template<typename T>
// Image<T> box_blur(const Image<T>& img){
//     Image<T> tmp(img.get_xDim(), img.get_yDim());
//     clear(tmp);
//     for(int i = 4; i < img.get_xDim() - 4; i++){
//         for(int j = 4; j < img.get_yDim() - 4; j++){
//             tmp(i,j) = img(i-1, j-1) + img(i, j-1) + img(i+1, j-1) +
//                        img(i-1, j) + 3.0*img(i, j) + img(i+1, j) +
//                        img(i-1, j+1) + img(i, j+1) + img(i+1, j+1);
//             tmp(i,j) /= 11.0;
//         }
//     }

//     return tmp;
// }


int main()
{
    uint16_t x1, y1;
    printf("uTensor deep learning character recognition demo\n");
    printf("https://github.com/uTensor/utensor-mnist-demo\n");
    printf("Draw a number (0-9) on the touch screen, and press the button...\r\
        n");

#ifndef TARGET_SIMULATOR
#ifdef MBED_CONF_APP_USE_FS
    ON_ERR(bd.init(), "SDBlockDevice init ");
    ON_ERR(fs.mount(&bd), "Mounting the filesystem on \"/fs\". ");
#endif
#endif

    //Image<float>* img = new Image<float>(240, 240);

    Image<float>* img = new Image<float>(240, 240);


    BSP_LCD_Init();
    button.rise(&trigger_inference_cb);

    /* Touchscreen initialization */
    if (BSP_TS_Init(BSP_LCD_GetXSize(), BSP_LCD_GetYSize()) == TS_ERROR) {
```

```
    printf("BSP_TS_Init error\n");
}

/* Clear the LCD */
BSP_LCD_Clear(LCD_COLOR_WHITE);

/* Set Touchscreen Demo1 description */
//BSP_LCD_SetTextColor(LCD_COLOR_GREEN);
//BSP_LCD_FillRect(0, 0, BSP_LCD_GetXSize(), 40);
//BSP_LCD_SetTextColor(LCD_COLOR_BLACK);
//BSP_LCD_SetBackColor(LCD_COLOR_GREEN);
//BSP_LCD_SetFont(&Font16);
//BSP_LCD_DisplayStringAt(0, 15, (uint8_t *)"Touch the screen", CENTER_MODE
    );

Context ctx;
// Context ctx2;
clear(*img);


while (1) {
    BSP_TS_GetState(&TS_State);
    if(trigger_inference){

        Image<float> smallImage = resize(*img, 28, 28);




        // Image<float> chopped = chop(smallImage);
        // pc.printf("Done chopping\n\n");
        // Image<float> img20 = resize(chopped, 20, 20);
        // pc.printf("Done resizing\n\n");
        // Image<float> img28 = pad(img20, 4, 4);

        // Image processing is heavy on constrained devices
        // manually delete
        pc.printf("Done padding\n\n");
        // smallImage.~Image<float>();
        // chopped.~Image<float>();
        // img20.~Image<float>();
        delete img;

        // Image<float> img28_2 = box_blur(smallImage);
        // pc.printf("Done blurring\n\r");
        // img28.~Image<float>();

        Image<float> img1 = cutleft(smallImage);
        Image<float> img2 = cutright(smallImage);

        Image<float> smallImage1 = resize(img1, 28, 28);
        Image<float> smallImage2 = resize(img2, 28, 28);

        pc.printf("Reshaping\n\r");
        smallImage1.get_data()->resize({1, 784});
        pc.printf("Creating Graph\n\r");
```

```
            get_deep_mlp_ctx(ctx, smallImage1.get_data());
            pc.printf("Evaluating\n\r");
            ctx.eval();
            S_TENSOR prediction = ctx.get({"y_pred:0"});
            int result1 = *(prediction->read<int>(0,0));

            pc.printf("Reshaping\n\r");
            smallImage2.get_data()->resize({1, 784});
            pc.printf("Creating Graph\n\r");

            printf("Number guessed %d\n\r", result1);

            // get_deep_mlp_ctx2(ctx2, smallImage2.get_data());
            // pc.printf("Evaluating\n\r");
            // ctx2.eval();
            // prediction = ctx2.get({"y_pred:0"});
            // int result2 = *(prediction->read<int>(0,0));

            // printf("Number guessed %d\n\r", result2);



            BSP_LCD_Clear(LCD_COLOR_WHITE);
            BSP_LCD_SetTextColor(LCD_COLOR_BLACK);
            BSP_LCD_SetFont(&Font24);

            // Create a cstring
            uint8_t number[2];
            number[2] = '\0';
            //ASCII numbers are 48 + the number, a neat trick
            number[0] = 48 + result1;
            // number[1] = 48 + result2;
            BSP_LCD_DisplayStringAt(0, 120, number, CENTER_MODE);
            trigger_inference = false;
            img = new Image<float>(28, 28);
            clear(*img);
            break;
        }
        if(TS_State.touchDetected) {
            /* One or dual touch have been detected     */

            /* Get X and Y position of the first touch post calibrated */
            x1 = TS_State.touchX[0];
            y1 = TS_State.touchY[0];

            img->draw_circle(x1, y1, 7); //Screen not in image x,y format. Must
                transpose

            BSP_LCD_SetTextColor(LCD_COLOR_GREEN);
            BSP_LCD_FillCircle(x1, y1, 5);

            wait_ms(5);
        }
    }
}
```

B.2 Archivo image.h

Código B.2 Archivo image.h.

```
#ifndef IMAGE_H
#define IMAGE_H

#include <vector>
#include "uTensor/core/tensor.hpp"

template<typename T, template<typename> class TENSOR=RamTensor>
class Image {
  private:
    Tensor* data;
    int xDim;
    int yDim;
    bool dimOverride = false;

        void put_pixel(int x, int y){
            if(x >= 0 && x < get_xDim() && y >= 0 && y < get_yDim())
                this->operator()(x, y) = 255;
        }
  public:

    Image(uint32_t x, uint32_t y){
      data = new TENSOR<T>();
      TensorShape tmp({x, y});
      data->init(tmp);
    }
    Image(Tensor* that): data(that){}
    Image(): data(nullptr){}

    T& operator[](int idx) { return *((T*)data->write<T>(idx, 0)); }
    // T& operator[](int idx) { return *write(idx, 0); }
    T& operator()(int x, int y){ return *((T*)data->write<T>(y*this->get_xDim()
         + x, 0)); }
    const T& operator[](int idx) const { return *data->read<T>(idx, 0); }
    const T& operator()(int x, int y) const{ return *data->read<T>(y*this->
        get_xDim() + x, 0); }


    Tensor* get_data() { return data; }
    void reshape(int x, int y){
      xDim = x;
      yDim = y;
      dimOverride = true;
    }
    int get_xDim(void) const {
      if(!dimOverride)
        return data->getShape()[0];
      else
        return xDim;
    }

    int get_yDim(void) const {
```

```
      if(!dimOverride)
        return data->getShape()[1];
      else
        return yDim;
    }
        void drawline(int x0, int y0, int x1, int y1)
        {
            int dx, dy, p, x, y;

            dx=x1-x0;
            dy=y1-y0;

            x=x0;
            y=y0;

            p=2*dy-dx;

            while(x<x1)
            {
                if(p>=0)
                {
                    put_pixel(x,y);
                    y=y+1;
                    p=p+2*dy-2*dx;
                }
                else
                {
                    put_pixel(x,y);
                    p=p+2*dy;
                }
                x=x+1;
            }
        }
        void draw_circle(int x0, int y0, int radius){
            int x = radius-1;
            int y = 0;
            int dx = 1;
            int dy = 1;
            int err = dx - (radius << 1);

            while (x >= y)
            {
                /* Hole version
                put_pixel(x0 + x, y0 + y);
                put_pixel(x0 - x, y0 + y);

                put_pixel(x0 + y, y0 + x);
                put_pixel(x0 - y, y0 + x);

                put_pixel(x0 - x, y0 - y);
                put_pixel(x0 + x, y0 - y);

                put_pixel(x0 - y, y0 - x);
                put_pixel(x0 + y, y0 - x);
                */
                drawline(x0 - x, y0 + y, x0 + x, y0 + y);
                drawline(x0 - y, y0 + x, x0 + y, y0 + x);
```

```
                drawline(x0 - x, y0 - y, x0 + x, y0 - y);
                drawline(x0 - y, y0 - x, x0 + y, y0 - x);

                if (err <= 0)
                {
                    y++;
                    err += dy;
                    dy += 2;
                }
                else
                {
                    x--;
                    dx += 2;
                    err += dx - (radius << 1);
                }
            }

        }

    ~Image(){
      delete data;
    }

};

template<typename T>
void get_bounding_box(const Image<T>& img, int& xMin, int& yMin, int& xMax, int
    & yMax){
  xMin = 5000000;
  yMin = 5000000;
  xMax = 0;
  yMax = 0;

  for(int i = 0; i < img.get_xDim(); i++){
    for(int j = 0; j < img.get_yDim(); j++){
      if(img(i,j) > 0){
        xMin = (i < xMin) ? i : xMin;
        yMin = (j < yMin) ? j : yMin;
        xMax = (i > xMax) ? i : xMax;
        yMax = (j > yMax) ? j : yMax;
      }
    }
  }
  // printf("%d, %d, %d, %d\n", xMin, yMin, xMax, yMax);
  return;
}

template<typename T>
void get_centroid(const Image<T>& img, int& xC, int& yC){

  xC = 0;
  yC = 0;
  int xN = 0;
  int yN = 0;

  for(int i = 0; i < img.get_xDim(); i++){
    for(int j = 0; j < img.get_yDim(); j++){
```

```
      if(img(i,j) > 0){
        xC += i;
        yC += j;
        xN += 1;
        yN += 1;
      }
    }
  }
  xC /= xN;
  yC /= yN;
  return;
}

/**
 * @brief Chop an image
 * @details Get the minimum bounding box of an image
 *
 * @param img [description]
 * @tparam T Probably uint8_t
 * @return chopped image
 */
template<typename T>
Image<T> chop(const Image<T>& img){
  int xMin, xMax, yMin, yMax;
  get_bounding_box(img, xMin, yMin, xMax, yMax);
  printf("Chopping image to bound = %d, %d, %d, %d\n", xMin, yMin, xMax, yMax);

  Image<T> temp(xMax-xMin, yMax-yMin);

  for(int i=0, ii=xMin; ii < xMax; i++, ii++){
    for(int j=0, jj=yMin; jj < yMax; j++, jj++){
      temp(i,j) = img(ii,jj);
    }
  }

  return temp;
}

/**
 * @brief Cut the image
 * @details Take left half of the image
 *
 * @param img [description]
 * @tparam T Probably uint8_t
 * @return Cutted image
 */
template<typename T>
Image<T> cutleft(const Image<T>& img){
  int xMin, xMax, yMin, yMax;
  xMin = 0;
  yMin = 0;
  xMax = 14;
  yMax = 27;
  printf("Cutting image side = %d, %d, %d, %d\n", xMin, yMin, xMax, yMax);

  Image<T> temp(xMax-xMin, yMax-yMin);
```

```
  for(int i=0, ii=xMin; ii < xMax; i++, ii++){
    for(int j=0, jj=yMin; jj < yMax; j++, jj++){
      temp(i,j) = img(ii,jj);
    }
  }

  return temp;
}

/**
 * @brief Cut the image
 * @details Take right half of the image
 *
 * @param img [description]
 * @tparam T Probably uint8_t
 * @return Cutted image
 */
template<typename T>
Image<T> cutright(const Image<T>& img){
  int xMin, xMax, yMin, yMax;
  xMin = 15;
  yMin = 0;
  xMax = 27;
  yMax = 27;
  printf("Cutting image side = %d, %d, %d, %d\n", xMin, yMin, xMax, yMax);

  Image<T> temp(xMax-xMin, yMax-yMin);

  for(int i=0, ii=xMin; ii < xMax; i++, ii++){
    for(int j=0, jj=yMin; jj < yMax; j++, jj++){
      temp(i,j) = img(ii,jj);
    }
  }

  return temp;
}

/**
 * @brief Nearest interpolation
 * @details Stretch or shrink an image naively
 *
 * @param img [description]
 * @param w2 new width
 * @param h2 new height
 * @tparam T [description]
 * @return [description]
 */
template<typename T>
Image<T> resize(const Image<T>& img, int w2, int h2){
    Image<T> temp(w2,h2);
    int x_ratio = (int)((img.get_xDim()<<16)/w2) +1;
    int y_ratio = (int)((img.get_yDim()<<16)/h2) +1;
    int x2, y2 ;

    for(int i = 0; i < h2; i++) {
        for(int j = 0; j < w2; j++) {
            x2 = ((j*x_ratio)>>16) ;
```

```
            y2 = ((i*y_ratio)>>16) ;
            temp[(i*w2) + j] = img[(y2*img.get_xDim()) + x2] ;
        }
    }
    return temp;

}


/**
 * @brief Zero Pad each side of the image
 *
 * @param img
 * @param padX number of pixels to pad the width with
 * @param padY number of pixels to pad the height with
 */

template<typename T>
Image<T> pad(const Image<T>& img, int padX, int padY){
  Image<T> temp(img.get_xDim() + 2*padX, img.get_yDim() + 2*padY);

  // Init to zero
  for(int i = 0; i < temp.get_xDim(); i++){
    for(int j = 0; j < temp.get_yDim(); j++){
      temp(i,j) = 0;
    }
  }
  for(int i = 0, ii = padX; i < img.get_xDim(); ii++, i++){
    for(int j = 0, jj = padY; j < img.get_yDim(); jj++, j++){
      temp(ii,jj) = img(i,j);
    }
  }

    return temp;

}

#endif
```

B.3 Archivo deep_mlp.py

Código B.3 Archivo deep_mlp.py.

```
#!/usr/bin/python
# -*- coding: utf8 -*-
"""
# This script is based on:
# https://www.tensorflow.org/get_started/mnist/pros
"""
from __future__ import print_function
import argparse
import sys
import tensorflow as tf
import numpy as np
```

```
from tensorflow.examples.tutorials.mnist import input_data
from tensorflow.python.framework import graph_util as gu
from tensorflow.tools.graph_transforms import TransformGraph

FLAGS = None


# helper functions
def weight_variable(shape, name):
  """weight_variable generates a weight variable of a given shape."""
  initial = tf.truncated_normal(shape, stddev=0.1)
  return tf.Variable(initial, name)


def bias_variable(shape, name):
  """bias_variable generates a bias variable of a given shape."""
  initial = tf.constant(0.1, shape=shape)
  return tf.Variable(initial, name)


# # Fully connected 2 layer NN
# def deepnn(x):
#   W_fc1 = weight_variable([784, 128], name='W_fc1')
#   b_fc1 = bias_variable([128], name='b_fc1')
#   a_fc1 = tf.add(tf.matmul(x, W_fc1), b_fc1, name="zscore")
#   h_fc1 = tf.nn.relu(a_fc1)
#   layer1 = tf.nn.dropout(h_fc1, 0.50)

#   # W_fc1 = weight_variable([784, 64], name='W_fc1')
#   # b_fc1 = bias_variable([64], name='b_fc1')
#   # a_fc1 = tf.add(tf.matmul(x, W_fc1), b_fc1, name="zscore")
#   # h_fc1 = tf.nn.relu(a_fc1)
#   # layer1 = tf.nn.dropout(h_fc1, 0.50)

#   W_fc2 = weight_variable([128, 64], name='W_fc2')
#   b_fc2 = bias_variable([64], name='b_fc2')
#   a_fc2 = tf.add(tf.matmul(layer1, W_fc2), b_fc2, name="zscore")
#   h_fc2 = tf.nn.relu(a_fc2)
#   layer2 = tf.nn.dropout(h_fc2, 0.50)

#   W_fc3 = weight_variable([64, 10], name='W_fc3')
#   b_fc3 = bias_variable([10], name='b_fc3')
#   logits = tf.add(tf.matmul(layer2, W_fc3), b_fc3, name="logits")
#   y_pred = tf.argmax(logits, 1, name='y_pred')

#   # W_fc3 = weight_variable([64, 10], name='W_fc3')
#   # b_fc3 = bias_variable([10], name='b_fc3')
#   # logits = tf.add(tf.matmul(layer1, W_fc3), b_fc3, name="logits")
#   # y_pred = tf.argmax(logits, 1, name='y_pred')

#   return y_pred, logits

# # Fully connected 1 layer NN
# def deepnn(x):
#   W_fc1 = weight_variable([784, 64], name='W_fc1')
#   b_fc1 = bias_variable([64], name='b_fc1')
#   a_fc1 = tf.add(tf.matmul(x, W_fc1), b_fc1, name="zscore")
```

```
#   h_fc1 = tf.nn.relu(a_fc1)
#   layer1 = tf.nn.dropout(h_fc1, 0.50)

#   W_fc3 = weight_variable([64, 10], name='W_fc3')
#   b_fc3 = bias_variable([10], name='b_fc3')
#   logits = tf.add(tf.matmul(layer1, W_fc3), b_fc3, name="logits")
#   y_pred = tf.argmax(logits, 1, name='y_pred')

#   return y_pred, logits

# Fully connected 3 layers NN
def deepnn(x):
  W_fc1 = weight_variable([784, 392], name='W_fc1')
  b_fc1 = bias_variable([392], name='b_fc1')
  a_fc1 = tf.add(tf.matmul(x, W_fc1), b_fc1, name="zscore")
  h_fc1 = tf.nn.relu(a_fc1)
  layer1 = tf.nn.dropout(h_fc1, 0.50)

  W_fc2 = weight_variable([392, 128], name='W_fc2')
  b_fc2 = bias_variable([128], name='b_fc2')
  a_fc2 = tf.add(tf.matmul(layer1, W_fc2), b_fc2, name="zscore")
  h_fc2 = tf.nn.relu(a_fc2)
  layer2 = tf.nn.dropout(h_fc2, 0.50)

  W_fc3 = weight_variable([128, 64], name='W_fc3')
  b_fc3 = bias_variable([64], name='b_fc3')
  a_fc3 = tf.add(tf.matmul(layer2, W_fc3), b_fc3, name="zscore")
  h_fc3 = tf.nn.relu(a_fc3)
  layer3 = tf.nn.dropout(h_fc3, 0.50)

  W_fc4 = weight_variable([64, 10], name='W_fc4')
  b_fc4 = bias_variable([10], name='b_fc4')
  logits = tf.add(tf.matmul(layer3, W_fc4), b_fc4, name="logits")
  y_pred = tf.argmax(logits, 1, name='y_pred')

  return y_pred, logits


def main(_):
  # Import data
  mnist = input_data.read_data_sets(FLAGS.data_dir, one_hot=True)

  # Specify inputs, outputs, and a cost function
  # placeholders
  x = tf.placeholder(tf.float32, [None, 784], name="x")
  y_ = tf.placeholder(tf.float32, [None, 10], name="y")

  # Build the graph for the deep net
  y_pred, logits = deepnn(x)

  with tf.name_scope("Loss"):
    cross_entropy = tf.nn.softmax_cross_entropy_with_logits_v2(labels=y_,
                                                       logits=logits)
    loss = tf.reduce_mean(cross_entropy, name="cross_entropy_loss")
  train_step = tf.train.AdamOptimizer(1e-4).minimize(loss, name="train_step")

  with tf.name_scope("Prediction"):
```

```
    correct_prediction = tf.equal(y_pred,
                                  tf.argmax(y_, 1))
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32), name="
        accuracy")

  # Start training session
  with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    saver = tf.train.Saver()

    # SGD
    for i in range(1, FLAGS.num_iter + 1):
      batch_images, batch_labels = mnist.train.next_batch(FLAGS.batch_size)
      batch_images = np.float32(batch_images>0.4)
      feed_dict = {x: batch_images, y_: batch_labels}
      train_step.run(feed_dict=feed_dict)
      if i % FLAGS.log_iter == 0:
        train_accuracy = accuracy.eval(feed_dict=feed_dict)
        print('step %d, training accuracy %g' % (i, train_accuracy))

    print('test accuracy %g' % accuracy.eval(feed_dict={x: mnist.test.images,
                                                        y_: mnist.test.labels}))
    # Saving checkpoint and serialize the graph
    ckpt_path = saver.save(sess, FLAGS.chkp)
    print('saving checkpoint: %s' % ckpt_path)
    out_nodes = [y_pred.op.name]
    # Freeze graph and remove training nodes
    sub_graph_def = gu.remove_training_nodes(sess.graph_def)
    sub_graph_def = gu.convert_variables_to_constants(sess, sub_graph_def,
        out_nodes)
    if FLAGS.no_quant:
      graph_path = tf.train.write_graph(sub_graph_def,
                                        FLAGS.output_dir,
                                        FLAGS.pb_fname,
                                        as_text=False)
    else:
      # # quantize the graph
      # quant_graph_def = TransformGraph(sub_graph_def,
      #                                  [],
      #                                  out_nodes,
      #                                  ["quantize_weights", "quantize_nodes"])
      graph_path = tf.train.write_graph(sub_graph_def,
                                        FLAGS.output_dir,
                                        FLAGS.pb_fname,
                                        as_text=False)
    print('written graph to: %s' % graph_path)
    print('the output nodes: {!s}'.format(out_nodes))


if __name__ == '__main__':
  parser = argparse.ArgumentParser()
  parser.add_argument('--data_dir', type=str,
                      default='mnist_data',
                      help='Directory for storing input data (default: %(default)
                          s)')
  parser.add_argument('--chkp', default='chkps/mnist_model',
                      help='session check point (default: %(default)s)')
```

```
  parser.add_argument('-n', '--num-iteration', type=int,
                      dest='num_iter',
                      default=20000,
                      help='number of iterations (default: %(default)s)')
  parser.add_argument('--batch-size', dest='batch_size',
                      default=50, type=int,
                      help='batch size (default: %(default)s)')
  parser.add_argument('--log-every-iters', type=int,
                      dest='log_iter', default=1000,
                      help='logging the training accuracy per numbers of
                          iteration %(default)s')
  parser.add_argument('--output-dir', default='mnist_model',
                      dest='output_dir',
                      help='output directory directory (default: %(default)s)')
  parser.add_argument('--no-quantization', action='store_true',
                      dest='no_quant',
                      help='save the output graph pb file without quantization')
  parser.add_argument('-o', '--output', default='deep_mlp.pb',
                      dest='pb_fname',
                      help='output pb file (default: %(default)s)')
  FLAGS, unparsed = parser.parse_known_args()
  tf.app.run(main=main, argv=[sys.argv[0]] + unparsed)
```

B.4 Archivo resize.sh

Código B.4 Archivo resize.sh.

```
#!/bin/bash

# Specify the desired volume size in GiB as a command-line argument. If not
    specified, default to 20 GiB.
SIZE=${1:-20}

# Install the jq command-line JSON processor.
sudo apt install -y jq

# Get the ID of the envrionment host Amazon EC2 instance.
INSTANCEID=$(curl http://169.254.169.254/latest/meta-data//instance-id)

# Get the ID of the Amazon EBS volume associated with the instance.
VOLUMEID=$(aws ec2 describe-instances --instance-id $INSTANCEID | jq -r .
    Reservations[0].Instances[0].BlockDeviceMappings[0].Ebs.VolumeId)

# Resize the EBS volume.
aws ec2 modify-volume --volume-id $VOLUMEID --size $SIZE

# Wait for the resize to finish.
while [ "$(aws ec2 describe-volumes-modifications --volume-id $VOLUMEID --
    filters Name=modification-state,Values="optimizing","completed" | jq '.
    VolumesModifications | length')" != "1" ]; do
  sleep 1
done
```

```
# Rewrite the partition table so that the partition takes up all the space that
     it can.
sudo growpart /dev/xvda 1

# Expand the size of the file system.
sudo resize2fs /dev/xvda1
```

Índice de Figuras

Índice de Tablas

Índice de Códigos

Bibliografía

[1] *Amazon web services*, https://aws.amazon.com/, April 2020.

[2] *Cloud9, amazon web services*, https://aws.amazon.com/es/cloud9/, April 2020.

[3] *Discovery kit with stm32f413zh mcu*, https://www.st.com/en/evaluation-tools/32f413hdiscovery.html, April 2020.

[4] *Jupyter notebook*, https://jupyter.org/, April 2020.

[5] *Keras: The python deep learning library*, https://keras.io/, April 2020.

[6] *Mbed: Free open source iot os and development tools from arm | mbed*, https://os.mbed.com/, April 2020.

[7] *Python software*, https://www.python.org/, April 2020.

[8] *Tensorflow: An end-to-end open source machine learning platform*, https://www.tensorflow.org/, April 2020.

[9] *Tensorflow extended (tfx) is an end-to-end platform for deploying production ml pipelines*, https://www.tensorflow.org/tfx, April 2020.

[10] *Tensorflow lite - deploy machine learning models on mobile and iot device*, https://www.tensorflow.org/lite, April 2020.

[11] *Tiny ml summit*, https://www.tinyml.org/summit/, April 2020.

[12] Michael Barr, *Embedded system glossary*, 1 ed., Netrino Technical Library, 2013.

[13] Yann LeCun; Corinna Cortes; Chris Burges, *MINST handwritten digit database*, http://yann.lecun.com/exdb/mnist/, April 2020.

[14] Stacey Higginbotham, *Machine learning on the edge-[internet of everything]*, IEEE Spectrum **57** (2019), no. 01, 20–20.

[15] Kedar Potdar, Taher S Pardawala, and Chinmay D Pai, *A comparative study of categorical variable encoding techniques for neural network classifiers*, International journal of computer applications **175** (2017), no. 4, 7–9.

[16] Peter Russell, Stuart; Norvig, *Inteligencia artificial: Un enfoque moderno*, 3 ed., 2009.

[17] Neil Tan, *Tinyml ai inference library*, https://github.com/uTensor/uTensor, April 2020.