Grado en Ciencia de Datos

Trabajo Fin de Grado

Construcción de un pipeline de conversión de autoencoders convolucionales para inferencia en Edge TPU

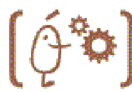
Autor:
Jose Oriol Moreno Usó

Tutores:
Valero Laparra Pérez-Muelas

Jordi Muñoz Marí

Julio 2022

Universitat de València



Escola Tècnica Superior d'Enginyeria ETSE-UV

Trabajo Fin de Grado

Construcción de un pipeline de conversión de autoencoders convolucionales para inferencia en Edge TPU

Autor:
Jose Oriol Moreno Usó

Tutores:
Valero Laparra Pérez-Muelas
Jordi Muñoz Marí

TRIBUNAL

PRESIDENTE/A: VOCAL 1:

VOCAL 2:

FECHA DE DEFENSA:

CALIFICACIÓN:

Declaración de autoría:

Yo,
Jose Oriol Moreno Usó, declaro la autoría del Trabajo Fin de Grado titulado "Construcción de un pipeline de conversión de autoencoders convolucionales para inferencia en Edge TPU" y que el citado trabajo no infringe las leyes en vigor sobre propiedad intelectual. El material no original que figura en este trabajo ha sido atribuido a sus legítimos autores.

Valencia, 19 de septiembre de 2022

Fdo:
Jose Oriol Moreno Usó

Resumen:

El objetivo de este proyecto es desarrollar e implementar un pipeline para la conversión de autoencoders convolucionales para poder hacer inferencia en un dispositivo de computación dedicado, el acelerador USB Coral TPU. De esta forma comprender todas las fases del desarrollo de un proyecto para Edge Computing.
Además de esto, se han explorado las diferentes posibilidades para implementar esta conversión y se han medido los diferentes resultados obtenidos en inferencia, tanto en error como en tiempo de computación.
Para poder construir este pipeline, se han empleado dos modelos ejemplo, uno construido desde cero a partir del banco de datos MNIST, y otro tipo U-Net entrenado para segmentación en conducción automática, creado a partir del banco de datos Cityscapes.
Palabras clave: TPU, IA, visión por computador, autoencoders convolucionales, Deep Learning, Machine Learning, Ciencia de Datos.

[1]

[1] El código y modelos de este trabajo está accesible en el siguiente enlace: https://github.com/joriolmuso/Pipeline-de-conversion-de-autoencoders-convolucionales-para-inferencia-en-Edge-TPU

Agradecimientos:

En primer lugar, quiero agradecer a mis tutores de TFG, al Dr. Valero Laparra Perez-Muelas y al Dr. Jordi Muñoz-Marí, la ayuda, guía y aprendizaje tanto durante el periodo de investigación, como el periodo de redacción de este trabajo, además de durante el resto del grado.
En segundo lugar me gustaría dar las gracias a Clara, sin la que todo este camino no habría sido posible. A mis padres, por haberme apoyado durante todo este tiempo de formación.
Por último, me gustaría agradecer a mis compañeros y amigos de clase: Adrián, Nuria, Moi, Ceci, Irina, Gaviña, Nacher, Salcedo, Marina y todos los demás, por haber sido los mejores compañeros de viaje que se puedan desear.

Índice general

1. Introducción 13
1.1. Motivación . . . 13
1.2. Objetivos . . . 14

2. Descripción de los medios 15
2.1. Edge Computing . . . 15
2.2. Hardware . . . 16
2.2.1. USB Coral Edge TPU . . . 16
2.3. Software . . . 17
2.3.1. Tensorflow Lite . . . 17
2.3.2. Pycoral . . . 18

3. Modelos y Datasets 19
3.1. CAE MNIST . . . 19
3.2. U-Net Cityscapes . . . 20

4. Optimizaciones y conversión de modelos 23
4.1. Optimizaciones . . . 23
4.1.1. Cuantizado con números reales . . . 23
4.1.2. Cuantizado con números enteros . . . 24
4.1.3. Quantization aware training . . . 25
4.2. Conversión . . . 25
4.3. Compilado para Edge TPU . . . 26

5. Inferencia, pruebas y resultados 29
5.1. Metodología para la inferencia en modelos tflite . . . 29
5.2. Pruebas realizadas . . . 30
5.2.1. Tiempos de procesado . . . 30
5.2.2. Diferencia de error . . . 31
5.2.3. Pesos de archivo . . . 31

6. Conclusiones **33**
6.1. Conclusiones 33
6.2. Proyección futura 33

A. Apéndice **35**
A.1. Tensorflow Lite 35
A.1.1. Conversión a Tensorflow Lite con cuantización de rango dinámico . 35
A.1.2. Quantization Aware Training 36
A.1.3. Conversión a Tensorflow Lite con cuantización con números enteros 37
A.1.4. Carga de modelo e inferencia con Tensorflow Lite 38
A.2. Edge TPU 39
A.2.1. Compilación de modelos tflite a Edge TPU 39
A.2.2. Carga de métodos del módulo Pycoral 39
A.2.3. Carga de modelo e inferencia con Edge TPU 40

Bibliografía **40**

Capítulo 1

Introducción

1.1. Motivación

El *Deep Learning* está cada vez más integrado en nuestro día a día, pocas son las industrias que no estén empezando a implementar algún sistema basado en redes neuronales. Dado el gran potencial de los modelos de Inteligencia Artificial (IA) cabe esperar un auge en su aplicación en diferentes ámbitos de la industria y el sector público.

La visión por computador es una de las áreas del campo de la IA que más avances puede ofrecer a nuestra sociedad. Estos avances van desde detección prematura de tumores hasta la conducción automática de vehículos de todo tipo. No obstante, esto tiene actualmente una serie de problemas asociados, como un alto coste computacional y diferentes problemas legales en el caso de que los datos tengan que ser enviados a otra ubicación para ser utilizados/procesados de cualquier forma.

Una solución para estos problemas puede ser la computación en el mismo lugar donde se adquieren estos datos, el denominado *edge computing*. La proliferación de este tipo de adaptaciones tanto de software como de hardware para poder llevar a cabo las operaciones pertinentes para el uso o extracción de datos son cada vez más comunes, y de esta forma también las complicaciones que estas adaptaciones conllevan.

Dentro del área de la visión por computador, uno de los temas menos resueltos actualmente es el de la computación eficiente en el uso de los modelos neuronales del tipo autoencoder convolucional, tanto en sus versiones más clásicas como en sus versiones más avanzadas.

La proliferación de investigaciones y trabajos acerca del uso del Edge Computing en la visión por computador prueba el gran interés que esta cuestión suscita tanto en el ámbito industrial como el académico. No obstante, no abundan los estudios específicos que traten el uso de dispositivos hardware preparados para inferencia y su uso aplicado a la visión por computador. Por esta razón, esta investigación se formula como el resultado de meses de trabajo en los que se ha reunido toda la información acerca de las optimizaciones disponibles para la conversión de modelos y su uso en Edge TPU. Además de esto, se han realizado numerosos experimentos implementando toda la cadena de pasos que trabajar con este tipo de tecnología requiere. Para esto, se ha realizado un amplio trabajo de recopilación de código y metodología, para poder unificarlo y crear un pipeline robusto y funcional.

En la realización de este trabajo, se han utilizado dos modelos: un autoencoder con-

volucional clásico aplicado sobre el *dataset* MNIST[1] y otro autoencoder convolucional más avanzado, tipo U-Net[2], aplicado como segmentador sobre el *dataset Cityscapes*[3].

La motivación de este trabajo es la de mostrar la posibilidad de adaptar los autoencoders convolucionales al paradigma del *edge computing*, y de esta forma avanzar y estimular tanto su investigación, desarrollo e implementación en la industria y los centros de investigación.

1.2. Objetivos

El principal objetivo de este trabajo es crear un *pipeline* robusto para la adaptación de autoenconders convolucionales para su uso en el acelerador USB Coral Edge TPU. Además de esto, realizar varias pruebas comparativas para observar las diferencias entre el uso de estos modelos y los originales sin ningún tipo de optimización de las que nos ofrece el cuantizado en *Tensorflow Lite*.

En las investigaciones relacionadas con la Ciencia de Datos o la IA se suele recurrir como estructura básica al procesado, modelado, inferencia y análisis de un determinado banco de datos, en este orden u otro similar. En el caso de este trabajo, se ha optado por una estructura en la que se incluyen una serie de pasos de optimización para los modelos que normalmente suele quedar relegados en un segundo plano *pipeline* (ver figura 1.1) y el posterior análisis comparativo.

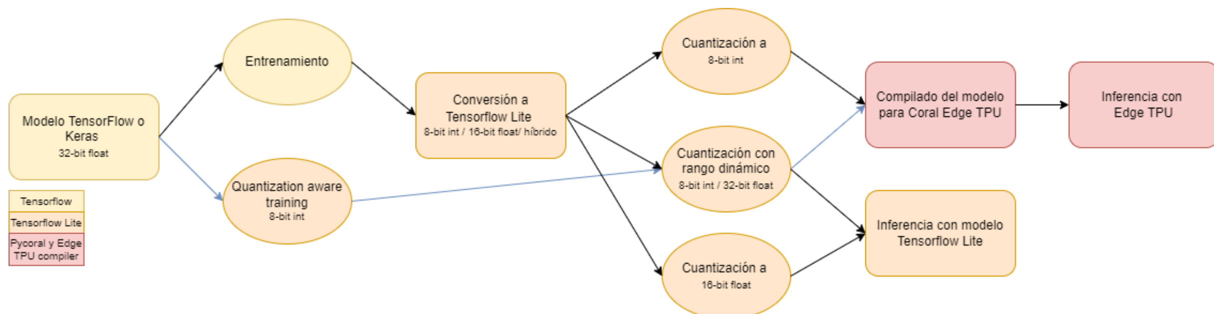


Figura 1.1: Pipeline diseñado durante el trabajo.Elaboración propia.

Por otro lado, dado que esta una tecnología novedosa aun en proceso de desarrollo, también es un objetivo de este trabajo estudiar las complicaciones que pueden surgir a la hora de adaptar/convertir un modelo ya entrenado para que se pueda hacer inferencia en el acelerador USB Coral Edge TPU, ya que a fecha de escritura de este trabajo, pueden surgir muchos problemas a la hora de utilizar esta metodología. Esto se ha hecho tanto para el modelo anteriormente mencionado como a partir de un modelo autoencoder U-Net[2] ya entrenado[4] para hacer segmentación semántica en el *dataset Cityscapes*.

Capítulo 2

Descripción de los medios

En este capítulo se expondrán los medios utilizados para la realización de este trabajo, tanto la parte hardware como el software.

2.1. Edge Computing

El *edge computing* es un tipo de computación que ocurre en la ubicación física del usuario o de la fuente de datos, como se muestra en la figura 2.1. Con el *edge computing* se pueden realizar proyectos permitiendo usar y distribuir un conjunto común de recursos (sistemas embebidos) en una gran cantidad de ubicaciones.

Este concepto de computación nos permite sortear los problemas derivados del envío de datos desde la ubicación original de recogida de los mismos a otro lugar para poder ser procesados. Esto tiene una serie de ventajas, como la latencia, la privacidad y la eficiencia de cómputo. Dado que no se han de enviar los datos a ningún otro lugar para ser procesados, esto puede reducir mucho la cantidad de operaciones de comunicación a realizar por el sistema, y que, por tanto, aumente la respuesta de este.

Por otro lado, la no transferencia de datos evita también los problemas asociados a la privacidad, tanto los problemas legales como los problemas de seguridad y los problemas de anonimización de los datos, cosa que también puede aumentar la respuesta del sistema al no requerir de las operaciones extra de esta anonimización.

Por último, la eficiencia del procesado puede aumentar mucho dado que en un sistema *edge computing* se puede conocer qué se va a procesar y cuál es el mejor sistema para hacerlo. Así pues, se podrán adaptar tanto el software como el hardware para este fin determinado.

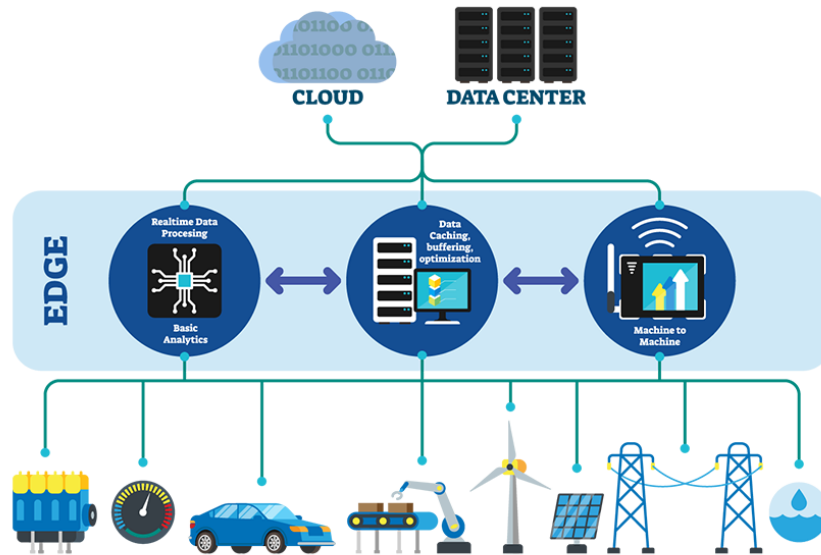


Figura 2.1: Ilustración del Edge computing.
https://blog.330ohms.com/2021/05/13/que-es-el-edge-computing/

2.2. Hardware

En esta sección se expondrá resumidamente el hardware al que se orienta este trabajo, mostrando también brevemente sus capacidades y limitaciones.

2.2.1. USB Coral Edge TPU

Una TPU (Tensor Processing Unit) en su forma más genérica es un hardware preparado y diseñado para computar tensores. Los tensores describen una relación multilineal entre conjuntos de objetos algebraicos relacionados con un espacio vectorial. Tener un hardware preparado para las operaciones con tensores en redes neuronales es importante porque los tensores son la unidad básica en las capas de las redes neuronales.

Además de esto, las TPUs eliminan el uso de memoria DRAM que sí requiere el procesado en GPU y CPU, aumentando significativamente el *throughput* (volumen de trabajo que fluye a través del sistema), esto se hace a través de la carga de los parámetros del modelo en los 8 MB de caché de la TPU (SRAM)[5]. Esta ventaja se puede ver comprometida por la rigidez con la que las TPUs funcionan respecto a las GPUs y CPUs.

En la página informativa de Coral[6], se define a la edge TPU como un circuito ASIC (Application Specific Integrated Circuit) diseñado con el fin de aprovechar al máximo los conocimientos en IA de Google para seguir la rápida evolución de esta tecnología y reflejarla en el hardware. Este dispositivo también existe con una interfaz PCIe en vez de USB. Imagen de una USB Coral Edge TPU en la figura 2.2

Este dispositivo tiene tres características principales: está pensado únicamente para emplearse en la parte de inferencia, adapta parcialmente sus puertas lógicas al modelo compilado, similar al funcionamiento de una FPGA y, por último, únicamente se pueden realizar operaciones con el tipo de dato numérico int8 y uint8, como se puede observar en la figura 2.3 todas las operaciones no compatibles con este dato numérico, quedan relegadas a su ejecución en CPU.

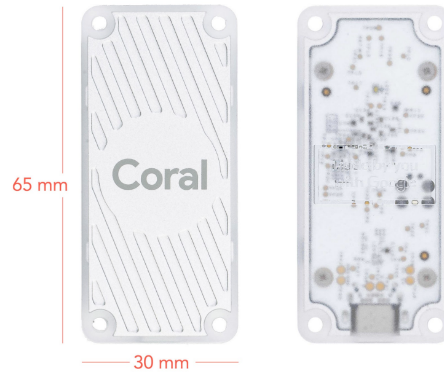


Figura 2.2: USB Coral Edge TPU.https://coral.ai/docs/accelerator/datasheet/

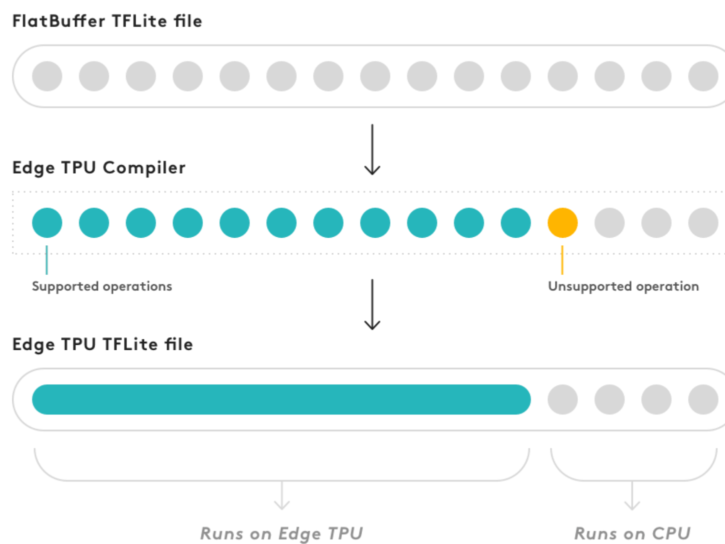


Figura 2.3: Todas las operaciones no soportadas se computan en CPU. https://coral.ai/docs/edgetpu/models-intro/#compiling

Las operaciones soportadas se encuentran en la página de referencia[7].

2.3. Software

En esta sección se expondrá resumidamente el software utilizado en este trabajo, mostrando también brevemente sus capacidades de adaptación y limitaciones.

2.3.1. Tensorflow Lite

TensorFlow Lite es parte de la librería *Tensorflow*[8] diseñada para ayudar a los desarrolladores a adaptar y ejecutar modelos en diferentes sistemas integrados, y permite al usuario aplicar redes neuronales en formato *tinyML*. Resumidamente, es la parte del módulo *Tensorflow* que permite la optimización de redes neuronales para reducir su latencia y tamaño.

Tensorflow Lite cuenta con múltiples características que lo distinguen de *Tensorflow*

y *Keras*. En primer lugar, está pensado para el aprendizaje profundo en *edge computing*. Además, tiene compatibilidad con diversos sistemas operativos (como Linux, Windows, Android e iOS), sistemas embebidos y lenguajes de programación (como Python, C, Java, Swift y C++). Flujo de conversión de modelos a *tflite* en figura 2.4.

Las optimizaciones con las que cuenta *Tensorflow Lite* para aumentar el rendimiento de los modelos reduciendo su tamaño se explorarán en la sección 4.1 del capítulo 4 de este trabajo.

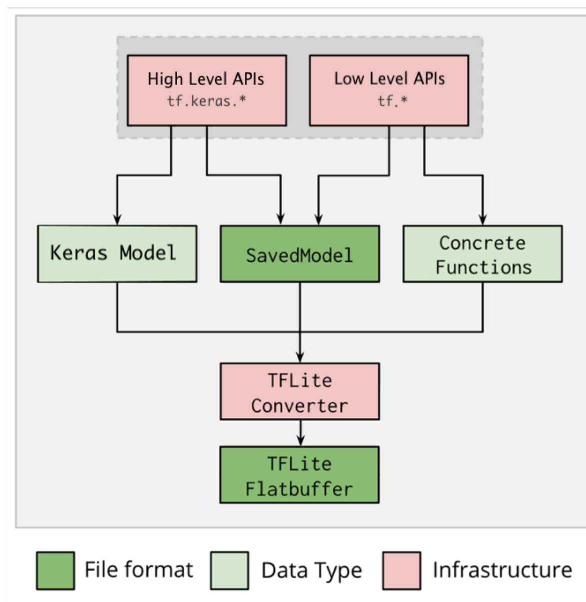


Figura 2.4: Flujo de conversión u optimización de modelo Keras en formato *h5* a *tflite*.
https://www.tensorflow.org/lite/convert?hl=es-419#python_api_

2.3.2. Pycoral

Siempre que se trata con un hardware, como en este caso la Coral Edge TPU, se requiere de la instalación de sus drivers para su correcto funcionamiento. En este caso no solo requiere de su driver dedicado si no que además se requiere la instalación de una librería especifica para la comunicación entre estos drivers y el lenguaje de programación Python.

El módulo *pycoral* se basa en la API de Python de *TensorFlow Lite* para simplificar el código al realizar inferencias en una Edge TPU y proporciona funciones avanzadas de Edge TPU como la canalización de modelos a través de múltiples Edge TPUs.

Código fuente del módulo *Pycoral*[9].

Capítulo 3

Modelos y Datasets

En este capítulo se expondrán las diferentes bases de datos, modelos y estructuras de los mismos empleados para la realización de esta investigación.

3.1. CAE MNIST

El conjunto de patrones MNIST[1] es un banco de datos preparado para clasificación muy conocido y usado para aprendizaje y *test* de todo tipo de modelos de *Machine Learning*. Este conjunto de datos está compuesto por 70,000 imágenes (60,000 de entrenamiento y 10,000 de validación) de 28x28 píxeles de dígitos manuscritos en blanco sobre negro desde el cero hasta el nueve y sus respectivas etiquetas para validar esos mismos números a cada imagen. Por tanto contiene 10 patrones y 10 clases. Esto se puede observar en la figura 3.1.

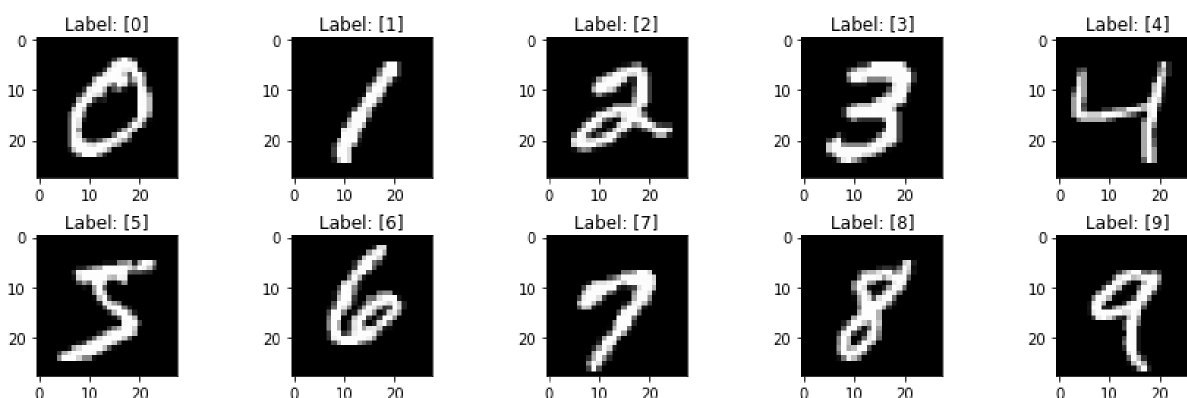


Figura 3.1: Ejemplo del conjunto de imágenes MNIST con sus etiquetas.
Elaboración propia.

Uno de los objetivos principales del trabajo es la adaptación de un modelo neuronal tipo CAE[10] (Convolutional Autoencoder) para su uso eficiente en *edge computing*. Con este fin, se ha creado desde el inicio un CAE simple utilizando únicamente tres tipos de capas: capas convolucionales-2D, capas MaxPooling-2D y capas UpSampling-2D.

Las capas convolucionales usan un *kernel* para convolucionarlo con el *input* de la capa y de esta forma generar un *output* que se plantea como una correlación que determina el parecido entre la imagen a analizar y el *kernel*. En cuanto a las capas *MaxPooling*, estas

se plantean par reducir la dimensionalidad de los tensores obtenidos. Para ello se aplica lo que se conoce como pooling, en este caso es la aplicación de la operación máximo sobre una parte del tensor.

Por otro lado, la capa UpSampling, repite las filas y las columnas del tensor *input*. La combinación de esta capa y una capa convolucional, es una la adaptación compatible con *Tensorflow Lite* y Edge TPU que funciona de forma similar a las convoluciones traspuestas. Las convoluciones traspuestas funcionan de forma idéntica a una capa convolucional junto con una capa de *pooling*, pero de forma inversa, aumentando el número de dimensiones del tensor de entrada.

El modelo comienza en la parte del encoder, de un espacio de 28 píxeles x 28 píxeles x 1 color, aplicando convoluciones y maxpooling, se llega al espacio latente de 4 x 4 x 8 filtros, posteriormente en la parte del decoder se transita del espacio latente al espacio original mediante convoluciones y up-sampling, quedando así un modelo sencillo con un total de 12,785 parámetros. Estructura visual del modelo en figura 3.2.

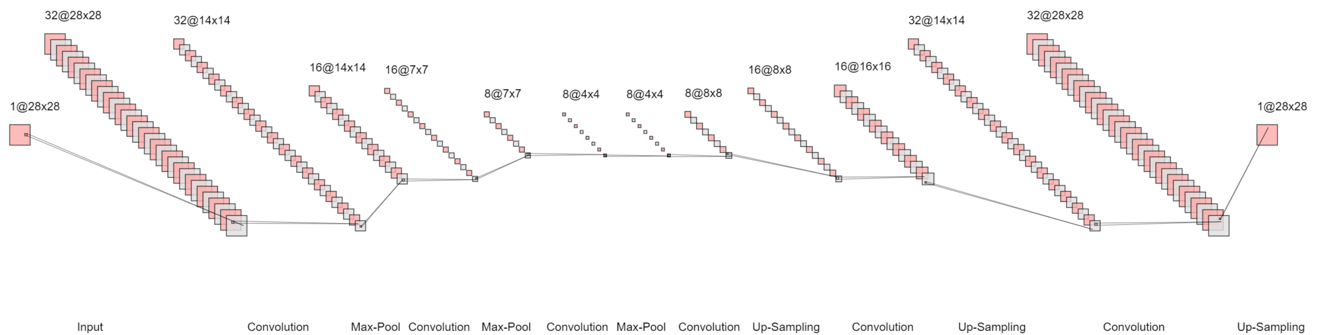


Figura 3.2: Estructura del modelo CAE empleado.Elaboración propia.

Para la parte de entrenamiento del modelo se ha usado como optimizador el método *Adam*[11] y como error a minimizar el MSE (Mean Squared Error).

Una vez entrenado el modelo en formato *Tensorflow* ha sido capaz de reconstruir las imágenes con un MSE de 0.0054, como se muestra en la figura 3.3.

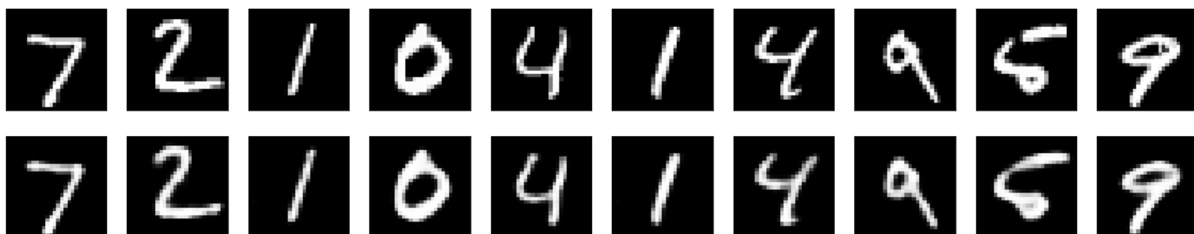


Figura 3.3: Reconstrucción de imágenes con un modelo CAE en formato *Keras*. Imágenes originales arriba, reconstruidas abajo.
Elaboración propia.

3.2. U-Net Cityscapes

El *dataset Cityscapes*[3], es un base de datos creado para su uso en entrenamiento y test de modelos de segmentación para imágenes urbanas por carretera. La cantidad de

imágenes que componen el conjunto de datos es de aproximadamente 3000 y se clasifican en 30 categorías diferentes divididas en 8 grupos: vehículos, personas, cielo, naturaleza, construcciones, objetos, planicies y nada.

Estas fotos son parte de varios vídeos de cincuenta ciudades diferentes tomadas durante horas de sol con buenas condiciones climáticas. Dado que originalmente estas imágenes eran parte de un vídeo, se eligieron algunos fotogramas clave para contener una gran cantidad de clases para cada imagen, diferentes fondos y varios objetos. Muestra en la figura 3.4

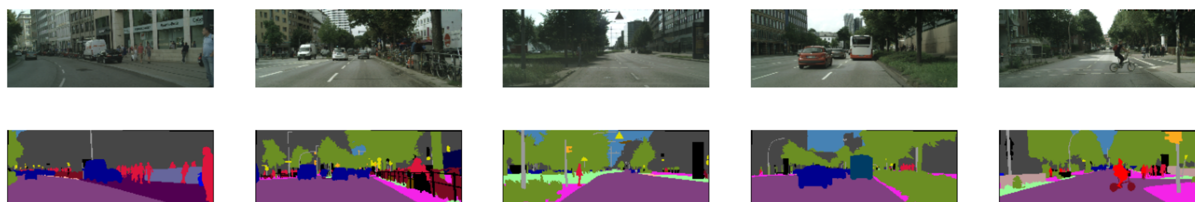


Figura 3.4: Ejemplo del conjunto de imágenes *Cityscapes* con sus respectivas máscaras. Elaboración propia.

Otro de los principales objetivos del trabajo es el de estudiar las complicaciones que pueden surgir a la hora de adaptar/convertir un modelo ya entrenado para que se pueda hacer inferencia en el acelerador USB Coral Edge TPU. Esto se ha hecho mediante un autoencoder convolucional con una arquitectura U-Net[2] ya entrenado[4]. Es importante recalcar que una de las principales virtudes de la conversión de modelos a formato *tflite*, es que nos permite cuantizar modelos ya entrenados sin tener que reentrenarlos, esto es una gran ventaja dado que la fase de entrenamiento es la más costosa de todas en el desarrollo de un modelo neuronal.

Los autoencoders U-Net tienen una clara aplicación para segmentación de imágenes. La arquitectura U-Net difiere de los autoencoders convolucionales normativos por poseer capas con dos salidas, una, empleada para apuntar a la siguiente capa, y otra, para apuntar a una capa más cercana a la salida del modelo que la primera, como se puede observar en la figura 3.5. A parte de esta diferencia, es importante mencionar, que las imágenes de entrada y las de salida del modelo no son las mismas como en un autoencoder convolucional estándar. En este caso las imágenes *input* serían las imágenes tomadas por cámara y las imágenes *output* de la red serían las máscaras segmentadas.

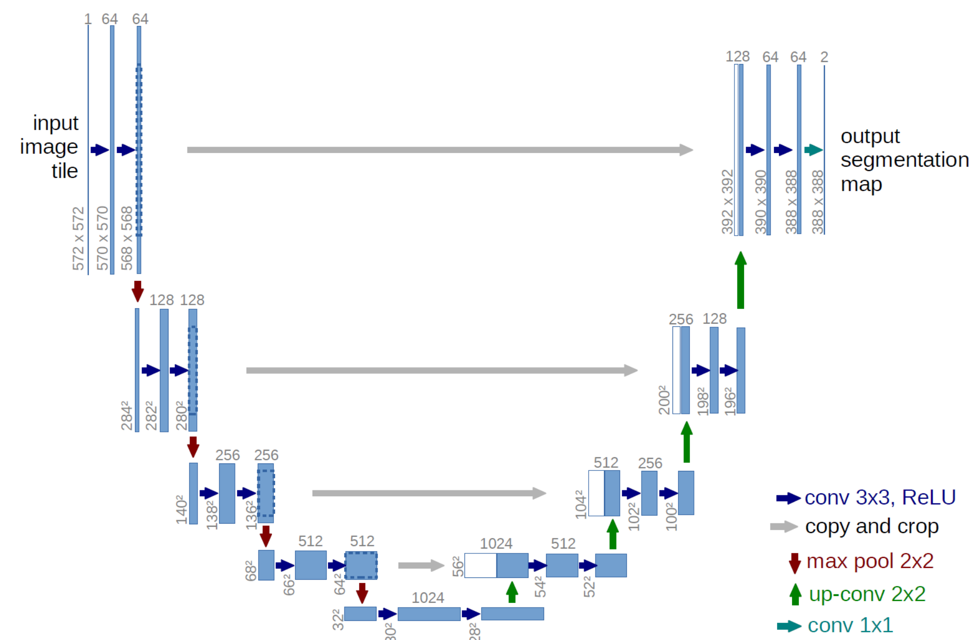


Figura 3.5: Ejemplo de arquitectura U-Net[12].https://github.com/shibuiwilliam/Keras_Autoencoder

Capítulo 4

Optimizaciones y conversión de modelos

En este capítulo se expondrán las optimizaciones que se pueden realizar mediante el módulo *Tensorflow Lite* y su compilado para su uso en Edge TPU.

4.1. Optimizaciones

El primer paso después de tener un modelo entrenado en formato *Tensorflow* o *keras* es el de elegir qué tipo de optimizaciones se van a aplicar cuando se haga la conversión del modelo de formato Keras *h5* a *tflite*. Esto se debe de hacer en función de las necesidades del proyecto, teniendo en cuenta las limitaciones de software y hardware con las que se va a contar.

En esta sección se expondrán las diferentes cuantizaciones que se pueden aplicar con *Tensorflow Lite* en la conversión de un modelo como se muestran en la figura 4.1.

Según la documentación que *Tensorflow Lite* nos indica [13], existen 4 tipos de optimizaciones que se pueden aplicar al convertir el modelo, estas se expondrán en las siguientes subsecciones.

4.1.1. Cuantizado con números reales

Dentro de la cuantización con números reales existen dos posibilidades que nos otorgan diferentes posibilidades.

La primera de las cuantizaciones consiste en convertir los pesos del modelo de operaciones de coma flotante de 32 bits a 16 bits. Las GPUs pueden ejecutar el modelo en esta forma de precisión aritmética reducida siempre que se habilite el *GPU delegate* de *Tensorflow Lite* en el sistema. En caso de que esto no fuese posible estas operaciones se seguirían pudiendo realizar en CPU. Este tipo de optimización está enfocada a la reducción del tamaño del modelo más que a la eficiencia de procesado del mismo, ya que se puede llegar a reducir el tamaño del mismo a la mitad. Además de esto, también cabe mencionar que en este caso la pérdida de precisión del modelo es ínfima, menor a medio punto porcentual respecto al original[14].

Otra posibilidad es la cuantización de rango dinámico. Esta opción permite combinar operaciones cuantizadas en enteros de 8 bits cuando sea posible y en operaciones de coma flotante de 32 bits cuando estas no sean compatibles. En este tipo de cuantización,

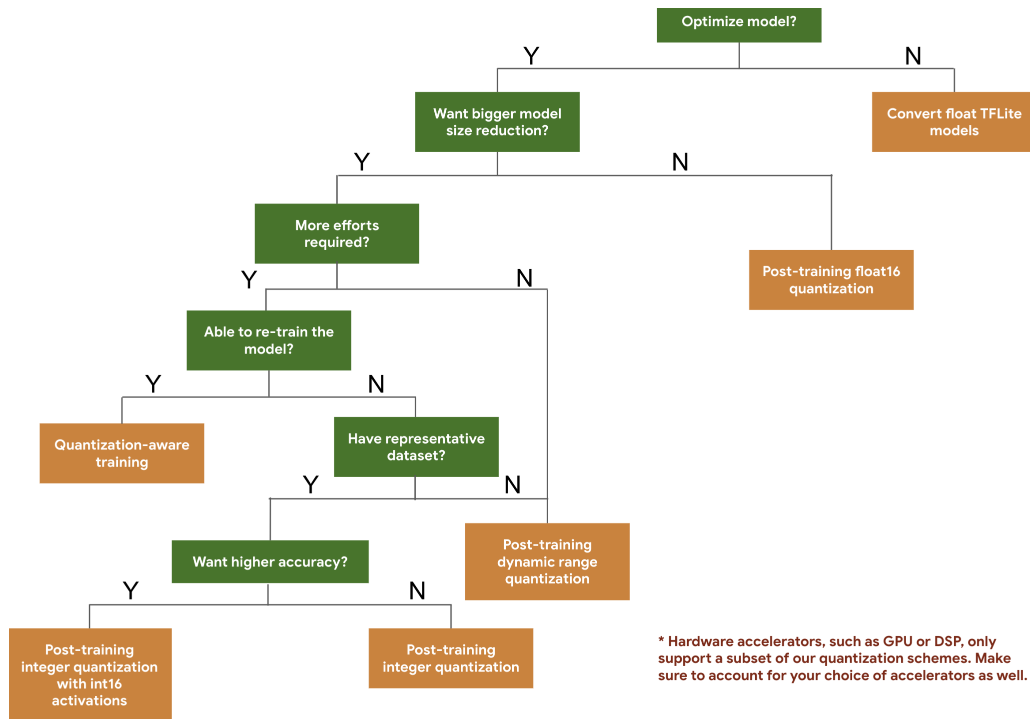


Figura 4.1: Árbol de decisión de tipo de cuantización [13]
https://www.tensorflow.org/lite/performance/model_optimization#types_of_optimization

las funciones de activación se guardan siempre en formato *float32*, y para las funciones que permiten el uso de cuantización, se cuantizan en formato *int8* para decuantizarse posteriormente al procesado. Esto puede llegar a reducir el tamaño del modelo a un cuarto del original, además de reducir la latencia con una pérdida de precisión mínima, de menos de un punto porcentual respecto al modelo original[15]. Código de este tipo de cuantizado en figura A.1.

4.1.2. Cuantizado con números enteros

En la cuantización con números enteros también existen dos posibilidades: la optimización a enteros, y *quantization aware training*, que es la estrategia que más dista de las demás. Puesto que esta tiene un proceso diferente al resto, se ha dedicado una sección específica para explicar esta estrategia.

La cuantización con números enteros es un tipo de optimización que convierte las operaciones de coma flotante de 32 bits al entero de 8 bits más cercano. Esto permite reducir el tamaño del modelo a un cuarto o menos del original y reducir mucho la latencia, a costa de reducir la precisión más que el resto de optimizaciones y pasar a un modelo completamente rígido[16]. Código de este tipo de cuantizado en figura A.3.

Para poder emplear esta estrategia se debe proporcionar al modelo una cantidad significativa de datos del *dataset* original empleado para entrenamiento u otro que resulte representativo.

Esta estrategia es la empleada para el *pipeline* propuesto dado que es compatible con la Coral USB Edge TPU, que es el principal objetivo de este trabajo.

4.1.3. Quantization aware training

Como se ha mencionado anteriormente, el *quantization aware training* es la estrategia que más dista de las demás. Esto se debe a que es la única de ellas que requiere que se aplique durante el entrenamiento, el resto de las estrategias, se aplican posteriormente al entrenamiento.

Este hecho tiene ventajas y desventajas:

A favor, a la hora de realizar la conversión de formato Keras *h5* a *tflite* la perdida de precisión del modelo es casi nula (inferior a medio punto porcentual)[17]. Además de esto, dado que el tipo de cuantizado es el mismo que en la estrategia de cuantización con enteros, tenemos los mismos beneficios, aumento significativo de la velocidad de cómputo y compatibilidad con Edge TPU. Ejemplo de funcionamiento en figura 4.2.

Desgraciadamente, el argumento que tiene en contra es bastante importante a la hora de considerar este tipo de estrategia de optimización. Como se ha mencionado anteriormente, la parte más costosa del desarrollo de un modelo neuronal de cualquier tipo, es la fase de entrenamiento, y dado que en esta estrategia de cuantizado es necesario volver a entrenar el modelo, el coste temporal y material del proyecto, se puede disparar según la complejidad del modelo. Código de este tipo de cuantizado en figura A.2.

Figura 4.2: Ejemplo dígitos reconstruidos con estrategia de optimización quantization aware training. Imágenes originales arriba, reconstrucción abajo. Elaboración propia.

Además de esto, como parte de los objetivos de este trabajo se encuentra el de explorar las complicaciones que pueden surgir al optimizar los modelos como parte del *pipeline*. A la hora de investigar esta estrategia de optimización, se ha encontrado una serie de errores que han impedido en múltiples ocasiones el uso y conversión del modelo. A día de hoy, en la versión 0.7.2 del módulo *tensorflow_model_optimization*, existe algún tipo de error que hace que las decuantizaciones de las funciones de activación queden fuera del rango de las propias funciones de activación, cosa que debe de ser considerada a la hora de plantear el uso de esta estrategia. Es posible que esto sea resuelto en un futuro, pero a fecha de publicación de este trabajo, este es un problema activo.

4.2. Conversión

A la hora de convertir el modelo de formato Keras *h5* a *tflite* para posteriormente poder compilarlo para *Edge TPU*, debemos de seguir el siguiente procedimiento.

- En primer lugar, mediante la librería *Tensorflow Lite*, creamos una clase conversor, dando como *input*, o bien un modelo ya cargado en formato *Tensorflow* o *keras*, o bien desde un modelo guardado.

- En segundo lugar, se deben de elegir las operaciones soportadas, optimizaciones y tipos de datos soportados, esto es lo que define cual va a ser la estrategia de optimización *post-training* durante la conversión de modelo. Estas pueden ser optimización *DEFAULT*, que corresponde a una cuantización con rango dinámico, un tipo soportado *float16*, que corresponde con una cuantización con operaciones de coma flotante de 16 bits.

- Por último, la fijación de operaciones soportadas *TFLITE_BUILTINS_INT8*, mostrar las imágenes representativas al conversor y fijar los tipos de entrada y salida del modelo a *int8* o *uint8*, que corresponde a un cuantizado *post-training* con enteros.

- Finalmente, solo queda invocar la conversión y guardar el modelo para su posterior carga y uso.

Código del proceso en figuras A.1 y A.3.

También es importante en este apartado explicar el proceso interno que usa *Tensorflow Lite* para convertir los modelos, este se denomina *flatbuffer*, y según su propia entrada en *Github*, se define como:

> A FlatBuffer is a binary buffer containing nested objects (structs, tables, vectors,..) organized using offsets so that the data can be traversed in-place just like any pointer-based data structure. Unlike most in-memory data structures however, it uses strict rules of alignment and endianness (always little) to ensure these buffers are cross platform. Additionally, for objects that are tables, FlatBuffers provides forwards/backwards compatibility and general optionality of fields, to support most forms of format evolution[18].

4.3. Compilado para Edge TPU

Posteriormente a la conversión del modelo a formato *tflite*, se da paso a la compilación del modelo para su computo en Edge TPU. Es importante destacar que al compilar un modelo para Edge TPU este no solo puede ser usado para hacer inferencia en la Coral USB Edge TPU, si no en toda la gama de productos Edge TPU que oferta Coral. Por ejemplo también se puede desplegar todo este trabajo en las Edge TPUs adaptadas para su funcionamiento en PCIe y otras versiones como SoC (System on a chip). El uso de estas versiones, probablemente dispararía la velocidad de cómputo dado que las comunicaciones no se ven limitadas por el puerto USB 3.1 gen 1.

El compilado de un modelo *tflite* para Edge TPU consiste en dos pasos, la creación de un grafo de operaciones y la asignación de memoria *SRAM* al modelo.

En primer lugar, se crea un grafo de operaciones, este consiste en evaluar las operaciones que se pueden ejecutar en la TPU (tabla de capas compatibles[7]) y cuales no, y que por tanto van a quedar relegadas a computarse en CPU. Este paso se puede observar en la figura 2.3.

Y en segundo lugar, se asignan los 8 MB de SRAM de la TPU. En una compilación de un solo modelo, se asigna una parte de esa SRAM al ejecutable de la inferencia del modelo y la parte restante a los parámetros del modelo. Cuantos más parámetros puedan ser asignados a la SRAM de la TPU, más rápido podrá esta computar la inferencia.

Código de la compilación en la figura A.5.

Aunque no esté entre los objetivos de este trabajo, también cabe mencionar que existen algunas opciones más avanzadas como la posibilidad de compilar varios modelos a la vez para su uso en una o más TPUs del mismo sistema, y el particionado de modelos para el uso de múltiples TPUs en paralelo.

Capítulo 5

Inferencia, pruebas y resultados

En este capítulo se mostrará como se puede emplear un modelo convertido a *Tensorflow Lite* o compilado para Edge TPU para inferencia, además de mostrar las pruebas realizadas y los resultados obtenidos.

5.1. Metodología para la inferencia en modelos tflite

El proceso de inferencia de un modelo en formato *tflite*, sigue un proceso con más pasos y complicaciones que las que se dan en el método *.predict* de un modelo en formato *Tensorflow o Keras*. Para poder hacer reconstrucciones con un autoencoder se han de seguir los siguientes pasos.

- Primero, se carga el modelo desde el *path* en el que este esté guardado mediante la clase *Interpreter* de la librería *Tensorflow Lite*.
- En segundo lugar, se preparan los tensores, paso en el que se cargan estos en memoria. En tercer lugar, se extraen del modelo las dimensiones de entrada y salida del modelo para su uso posterior. En caso de que el modelo esté cuantizado para *uint8*, se reescalan los *inputs* y *outputs* para que el modelo funcione adecuadamente.
- Posteriormente, se ajustan las dimensiones del *batch* de imágenes al de la entrada del modelo.
- Finalmente, se alimenta el tensor con el *batch* de imágenes, se invoca el tensor (ejecuta la operación) y se recoge el tensor de salida, ya que este corresponde con la reconstrucción.

Es importante destacar que la reconstrucción es un objeto mutable y que, al menos en Python, se debe o bien hacer una *deep copy* si se va a guardar en una lista, o bien utilizar algún tipo de lista inmutable como un *numpy.array*.

Código de la inferencia en modelos *Tensorflow Lite* en la figura A.4.

A la hora de reproducir este mismo proceso para una Edge TPU, este es muy similar.

- Primero se carga el modelo y se envía a la SRAM de la TPU para posteriormente utilizar la función *run_ inference*.

- Posteriormente, se provee como inputs el modelo cargado anteriormente y la imagen a reconstruir convertida a bytes a la función *run_inference*.
- Finalmente, el proceso es el mismo que el anteriormente mencionado para inferencia con un modelo en formato *Tensorflow Lite*, se recoge el último tensor y se almacena.

Código de la inferencia en modelos Edge TPU en la figura A.7.

5.2. Pruebas realizadas

En esta sección se expondrán las pruebas realizadas a los diferentes modelos de forma que se pueda establecer una prueba comparativa entre los modelos computando las reconstrucciones en Edge TPU y estos mismos sin realizar ningún tipo de conversión.

Las pruebas se han realizado en dos ordenadores con diferentes capacidades de procesado. Por un lado, un ordenador preparado para altas cargas de computación, que cuenta con una CPU *Ryzen 7 5800H* funcionando con frecuencia de reloj de 4.4GHz y 45W de potencia, 32GB de memoria RAM y que además cuenta con una GPU dedicada *Nvidia RTX 3060* de 6GB de memoria a 130W. Por otro lado, otro ordenador mucho más modesto contando con un procesador *i5-6300U* con una frecuencia de reloj de 2.5GHz y 8GB de memoria RAM. Este segundo sistema no cuenta con GPU dedicada. También se ha tomado en cuenta una prueba de tiempo de procesado con una *Raspberry pi*, hardware mucho más similar al que se tendrían en una aplicación real de *Edge Computing*.

5.2.1. Tiempos de procesado

Para medir los tiempos de procesado y poder hacer comparativas entre las diferentes versiones de un mismo modelo se ha utilizado el *wall-clock time*. El *wall-clock time* se define como el tiempo total de ejecución de un programa u operación, contando no solo el tiempo que se dedica al cómputo en CPU, sino también los tiempos en la realización del resto de procesos, como el acceso a memoria primaria o secundaria, o las llamadas al Sistema Operativo. Dado que uno de los objetivos de este trabajo es el de establecer comparativas entre los modelos sin convertir, y los modelos ejecutándose en Edge TPU, es necesario medir el tiempo que se usa en todos los procesos, tanto las operaciones en CPU como el resto de tareas que realiza el sistema al hacer inferencia con la Edge TPU.

En primer lugar, los resultados al utilizar el CAE con la partición *x_test* el banco de datos MNIST en el ordenador de mayor potencia, han resultado de la siguiente forma: el modelo sin convertir se ejecuta en aproximadamente 0.62 segundos, mientras que el modelo convertido, aproximadamente 6.78 segundos. Es importante recalcar que esta diferencia de *wall-clock time* se debe a dos factores. El primero, al uso de una GPU dedicada. Y el segundo, se debe a que al ejecutar la parte de inferencia/reconstrucción, al usar la GPU el modelo está ya cargado en su memoria dedicada, y al ejecutarlo en Edge TPU, tanto los datos como el modelo se han de enviar a la memoria de la Edge TPU. Además de esto, añadir que los tiempos de comunicación entre un dispositivo conectado por el puerto PCIe 4.0 no se pueden comparar con la conexión que ofrece el puerto USB 3.1 Gen 1. Ofreciendo la primera velocidades de hasta 31.51 GB/s y la segunda de hasta 625 MB/s, esto puede marcar una diferencia de latencia en comunicaciones y envío de datos de hasta cincuenta veces en estos experimentos.

Por otro lado, la misma ejecución en el segundo ordenador arrojan unos resultados sin convertir del modelo de aproximadamente 12.12 segundos y computando en Edge TPU, aproximadamente 18.3 segundos. De esta manera, podemos observar que la computación en un ordenador sin una GPU dedicada equilibra mucho los tiempos, siendo la inferencia en CPU 1.5 veces más rápida que en Edge TPU.

PC	h5	edgetpu.tflite	Diff (x)
R7	0.62	6.78	x10.93 mejor CPU + GPU que TPU
i5	12.12	18.3	x1.5 mejor CPU que TPU
Raspberry Pi	40	5	x8 mejor TPU que CPU

Cuadro 5.1: Tabla de tiempos de procesado, tiempos en segundos.Elaboración propia.

Esto nos indica que a menor potencia de computación de CPU, mayor es la diferencia en latencia de computo entre esta y la del procesado en Edge TPU, beneficiando a la Edge TPU. Dado que la Edge TPU es un dispositivo pensado para ser incorporado en sistemas de muy baja potencia de procesado y coste económico, a nivel de tiempo de procesado, es una alternativa viable.

5.2.2. Diferencia de error

Como métrica de error se ha utilizado el *MAE* (Mean Absolute Error) para medir la discrepancia entre las imágenes originales y las reconstruidas. Se ha elegido el error absoluto medio porque es una métrica adecuada para medir errores en problemas de regresión y además, el error está en la misma escala que los propios datos, en este caso, entre 0 y 1 dado que las imágenes están reescaladas de su formato original entre 0 y 255.

$$\sum_{i=1}^{D} |x_i - y_i|$$

Al reconstruir las imágenes de la partición *x_ test* el banco de datos MNIST obtenemos un *MAE* de 0.0229 en el modelo sin convertir y 0.0261 en el modelo convertido a formato *tflite* con la estrategia de optimización cuantizado con números enteros. La diferencia entre ambos errores es de un 14 %, que a priori es significativo, pero al menos para la reconstrucción de imágenes con un autoencoder convolucional, la diferencia entre ambas reconstrucciones es imperceptible.

Modelo	h5	tflite	Diff (%)
CAE MNIST	0.0229	0.0261	14 %

Cuadro 5.2: Tabla de errores, métrica MAE en valores entre 0 y 1.Elaboración propia.

5.2.3. Pesos de archivo

El peso de los archivos de los modelos puede ser un elemento crítico a la hora de desplegar un modelo para su uso en sistemas *Edge Computing* como por ejemplo los sistemas embebidos. Este tipo de plataformas de computación están por lo general diseñadas para tener el mínimo coste posible, lo cual hace que dispongan de una cantidad de memorias primaria y secundaria reducidas.

Se ha utilizado el banco de datos MNIST para entrenar un CAE. El resultado que se ha obtenido durante el periodo de experimentación es el siguiente: el modelo sin ningún tipo de optimización ni conversión tiene un volumen de 226KB. Por otro lado, convertido a *tflite* y cuantizado con números enteros tiene un peso en disco de 23KB. Por último, la compilación de este modelo para su uso en Edge TPU tiene un volumen de 101KB.

Este mismo experimento también se ha realizado con un modelo U-Net entrenado con el *dataset Cityscapes*. En este caso el modelo original tiene un peso 10,941KB, su versión convertida a *tflite* cuantizado a números enteros tiene un volumen de 2,847KB, y finalmente la versión compilada para Edge TPU pesa 3,037KB.

Modelo	h5	tflite	edgetpu.tflite
CAE MNIST	226KB	23KB	101KB
U-Net Cityscapes	10,941KB	2,847KB	3,037KB

Cuadro 5.3: Tabla de memoria que ocupan los modelos.Elaboración propia.

De estos resultados se puede extraer con mucha claridad que las conversiones a formato *tflite* con cuantización con números enteros reducen muy significativamente el volumen del modelo a entre un cuarto y un quinto de su volumen inicial. Además de esto, podemos constatar que la compilación de los modelos para su uso en Edge TPU añade cierto peso al modelo, siendo esta carga un tamaño reducido, puede llegar a ser significativa en modelos que por sí mismos tengan un tamaño inferior a los 100KB.

Capítulo 6

Conclusiones

6.1. Conclusiones

En definitiva, se puede afirmar que se han cumplido los objetivos principales del trabajo. Por un lado, mediante la creación desde cero de un autoencoder convolucional sobre el *dataset* MNIST explorado e implementado un pipeline para la conversión de modelos neuronales enfocando su uso para Edge TPU.

A lo largo del trabajo se ha experimentado con las múltiples opciones que este tipo de dispositivos pueden ofrecer, y mediante un modelo preentrenado tipo U-Net sobre el *dataset Cityscapes*, se han estudiado las posibles complicaciones con las que se ha de lidiar a la hora de implementar este tipo de proyectos.

Definitivamente los dispositivos que facilitan el *Edge Computing* son un recurso valioso y viable, no solo por su enorme potencial mostrado a lo largo de este trabajo, si no también por su coste (este tipo de dispositivos cuestan entre 25 y 100 dólares), y como se muestra en la sección 5.2.1, pueden acercarse a las latencias de procesado de equipos con un coste económico diez veces superior.

6.2. Proyección futura

En esta sección se expondrán las posibles futuras investigaciones que pueden surgir con posteridad a este trabajo.

Queda pendiente para posteriores investigaciones la modificación del modelo U-Net para segmentación semántica, que desgraciadamente no se ha podido acabar de explorar en este trabajo por limitaciones materiales, en el momento crítico de la fase de experimentación solo se pudo utilizar el ordenador de baja potencia dado que en el de más potencia se produjo una falla.

También cabe mencionar la amplitud del tema de este trabajo, donde la cantidad de caminos por explorar es inmensa y por limitación de forma y tiempo quedan fuera de este trabajo. Por ejemplo, el uso de varias TPUs funcionando en paralelo, la observación mediante experimentación de las mejoras que ofrecen el resto de estrategias de optimización de modelos, la conversión de modelos de gran envergadura, la conversión de otros tipos de modelos neuronales como el generador o discriminador de una GAN[19] (Generative Adversarial Network).

Como punto final a esta sección, sería muy interesante explorar el resto de opciones que existen a la hora de optimizar modelos neuronales y sus adaptaciones para *Edge Computing*, tanto a nivel software como a nivel hardware, y de esta forma poder establecer una comparativa entre las opciones que existen a la hora de desplegar modelos neuronales en sistemas de bajo coste.

Apéndice A

Apéndice

A.1. Tensorflow Lite

A.1.1. Conversión a Tensorflow Lite con cuantización de rango dinámico

Conversión en rango dinámico

```
tflite_model_path = "./models/cnn_AE_mnist_quant.tflite"

converter = tf.lite.TFLiteConverter.from_keras_model(autoencoder)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
tflite_model = converter.convert()
open(tflite_model_path, "wb").write(tflite_model)
```

```
INFO:tensorflow:Assets written to: C:\Users\bchet\AppData\Local\Temp\tmp_y5oj32d\assets

16944
```

Figura A.1: Código de conversión a Tensorflow Lite con cuantización de rango dinámico. Elaboración propia.

A.1.2. Quantization Aware Training

Creación de modelo

```
# dimension de dominio intermedio
encoding_dim = 32

encoder_input = Input(shape=(784,), name="original_img")
# ENCODER
encoded = Dense(256, activation='relu')(encoder_input)
encoded = Dense(128, activation='relu')(encoded)
encoded = Dense(64, activation='relu')(encoded)
encoder_out = Dense(encoding_dim, activation='relu')(encoded)

# DECODER
encoded_input = Input(shape=(encoding_dim,))
decoded = Dense(64, activation='relu')(encoded_input)
decoded = Dense(128, activation='relu')(decoded)
decoded = Dense(256, activation='relu')(decoded)
decoder_output = Dense(784, activation='sigmoid')(decoded)

# Modelo para el encoder
encoder = Model(encoder_input, encoder_out, name="encoder")

# Modelo del decoder
decoder = Model(encoded_input, decoder_output, name="decoder")
```

```
import tensorflow_model_optimization as tfmot

quantize_model = tfmot.quantization.keras.quantize_model

q_encoder_model = quantize_model(encoder)
q_decoder_model = quantize_model(decoder)

# AUTOENCODER
autoencoder_input = Input(shape=(784,), name="img")
encoded_img = q_encoder_model(autoencoder_input)
decoded_img = q_decoder_model(encoded_img)
autoencoder = Model(autoencoder_input, decoded_img, name="autoencoder")

autoencoder.compile(optimizer = tf.keras.optimizers.Adam(learning_rate=0.001),
                    loss='mse')

autoencoder.summary()
```

```
Model: "autoencoder"
```

Figura A.2: Código de Quantization Aware Training.Elaboración propia.

A.1.3. Conversión a Tensorflow Lite con cuantización con números enteros

Conversión en int8

```
def representative_data_gen():
  for input_value in tf.data.Dataset.from_tensor_slices(x_train).batch(1).take(100):
    # Model has only one input so each data point has one element.
    yield [input_value]
```

```
tflite_model_path = "./models/cnn_AE_mnist_quant_1.1_int8.tflite"

converter = tf.lite.TFLiteConverter.from_keras_model(autoencoder)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.representative_dataset = representative_data_gen
converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS_INT8]

converter.inference_input_type = tf.float32
converter.inference_output_type = tf.float32

tflite_quant_model = converter.convert()
open(tflite_model_path, "wb").write(tflite_quant_model)
```

```
INFO:tensorflow:Assets written to: C:\Users\bchet\AppData\Local\Temp\tmpdrjqj_bl\assets

INFO:tensorflow:Assets written to: C:\Users\bchet\AppData\Local\Temp\tmpdrjqj_bl\assets

22536
```

Figura A.3: Código de conversión a Tensorflow Lite con cuantización a números enteros. Elaboración propia.

A.1.4. Carga de modelo e inferencia con Tensorflow Lite

Inferencia con Tensorflow Lite

```
def iterate(array, batch):
    inf = 0
    sup = batch
    while sup < len(array)+batch:
        res = array[inf:sup, :, :]
        yield res
        inf += batch
        sup += batch
```

```
batch = opt_batch
tflite_interpreter.resize_tensor_input(input_details[0]['index'], (batch, 28, 28, 1))
tflite_interpreter.resize_tensor_input(output_details[0]['index'], (batch, 28, 28, 1))
tflite_interpreter.allocate_tensors()

input_details = tflite_interpreter.get_input_details()
output_details = tflite_interpreter.get_output_details()

t1 = time.time()
iterator_test = iterate(x_test, batch)
results = []

while True:
    try:
        tflite_interpreter.set_tensor(input_details[0]['index'], next(iterator_test))

        tflite_interpreter.invoke()

        tflite_model_predictions = tflite_interpreter.get_tensor(output_details[0]['index'])
        results.append(tflite_model_predictions)
    except:
        break
t2 = time.time()
print(f"Execute time: {round(t2 - t1, 2)} segs")

results = np.array(results).reshape(10000, 28, 28, 1)
```

```
Execute time: 124.27 segs
```

Figura A.4: Código de inferencia con Tensorflow Lite.Elaboración propia.

A.2. Edge TPU

A.2.1. Compilación de modelos tflite a Edge TPU

Edge TPU tflite model compiler

```
!curl https://packages.cloud.google.com/apt/doc/apt-key.gpg | apt-key add -
!echo "deb https://packages.cloud.google.com/apt coral-edgetpu-stable main" | tee /etc/apt/sources.list.d/coral-edgetpu.list
!apt-get update
!apt-get install edgetpu-compiler
```

```
!edgetpu_compiler -m 13 -d cnn_AE_mnist_quant_1.1_int8.tflite
```

Figura A.5: Código de descarga de recursos y compilación a modelo Edge TPU. Elaboración propia.

A.2.2. Carga de métodos del módulo Pycoral

Librerias

```
import cv2
import os
import copy
import numpy as np
import time
import matplotlib.pyplot as plt
#import tensorflow as tf

import pycoral
from pycoral.adapters.common import input_details
from pycoral.adapters.common import output_tensor
from pycoral.utils.edgetpu import make_interpreter
from pycoral.utils.edgetpu import run_inference
```

Figura A.6: Código de carga del módulo Pycoral y sus métodos.Elaboración propia.

A.2.3. Carga de modelo e inferencia con Edge TPU

Modelo compilado para TPU

+ Code

```
interpreter = make_interpreter(model_path_quant) # Genereramos/cargamos el objeto modelo
interpreter.allocate_tensors() # Lo enviamos a TPU
inference_size = input_details(interpreter, 'shape') # Recogemos el input shape del modelo
```

Reconstrucción/Inferencia

```
t1 = time.time()
images = [] # Lista para guardar la reconstruccion de cada imagen

for img in x_test:
    #cv2_im = cv2.resize(img, inference_size) # Cambio de shape para que encaje con el tamaño de input
    run_inference(interpreter, img.tobytes()) # Inferencia
    reconstruction = output_tensor(interpreter,0) # Recogemos la reconstrucción
    img_rec = copy.copy(reconstruction) # Hacemos una copia de cada imagen reconstruída dado que reconstruction es mutable
    images.append(img_rec) # Añadimos cada imagen reconstruida a la lista
print(f"Execute time: {round(time.time() - t1, 2)} segs")
```

```
Execute time: 6.78 segs
```

Figura A.7: Código de carga de modelo e inferencia con Edge TPU.Elaboración propia.

Bibliografía

[1] Li Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[2] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pages 234–241. Springer, 2015.

[3] Marius Cordts, Mohamed Omran, Sebastian Ramos, Timo Rehfeld, Markus Enzweiler, Rodrigo Benenson, Uwe Franke, Stefan Roth, and Bernt Schiele. The cityscapes dataset for semantic urban scene understanding. In *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[4] Pablo Hernandez. Autonomoun driving semantic segmentation with divisive normalization. 2022.

[5] Edge TPU Compiler: parameter data caching. https://coral.ai/docs/edgetpu/compiler/#parameter-data-caching. Visitado: 2022-06-29.

[6] Coral technology: advanced neural network processing for low-power devices. https://coral.ai/technology. Visitado: 2022-06-26.

[7] Coral USB Edge TPU: supported operations. https://coral.ai/docs/edgetpu/models-intro/#supported-operations. Visitado: 2022-06-26.

[8] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[9] Pycoral: module sourcecode. https://github.com/google-coral/pycoral. Visitado: 2022-06-26.

[10] Jonathan Masci, Ueli Meier, Dan Cireşan, and Jürgen Schmidhuber. Stacked convolutional auto-encoders for hierarchical feature extraction. In *International conference on artificial neural networks*, pages 52–59. Springer, 2011.

[11] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[12] Keras autoencoder: unet. https://github.com/shibuiwilliam/Keras_Autoencoder. Visitado: 2022-06-26.

[13] Model optimization. https://www.tensorflow.org/lite/performance/model_optimization. Visitado: 2022-06-26.

[14] Post-training float16 quantization. https://www.tensorflow.org/lite/performance/post_training_float16_quant#overview. Visitado: 2022-06-29.

[15] Post-training dynamic range quantization. https://www.tensorflow.org/lite/performance/post_training_quant#overview. Visitado: 2022-06-29.

[16] Post-training integer quantization. https://www.tensorflow.org/lite/performance/post_training_integer_quant#overview. Visitado: 2022-06-29.

[17] Quantization aware training. https://www.tensorflow.org/model_optimization/guide/quantization/training#deploy_with_quantization. Visitado: 2022-06-29.

[18] FlatBuffers white paper: summary. https://google.github.io/flatbuffers/flatbuffers_white_paper.html. Visitado: 2022-06-28.

[19] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. *Advances in neural information processing systems*, 27, 2014.