Universidad Zaragoza

# Trabajo Fin de Grado

## Localización de fuentes sonoras en tiempo real, mediante redes neuronales y agrupaciones de micrófonos .

## Location of sound sources in real time, using neural networks and microphone arrays.

Autor

Álvaro Marín Velázquez

Directores

José Ramón Beltrán Blázquez

David Díaz-Guerra Aparicio

ESCUELA DE INGENIERÍA Y ARQUITECTURA
2020

# RESUMEN

Tanto el procesado de señal de agrupaciones de sensores como sus aplicaciones son temas ampliamente estudiados por los investigadores durante años. En este trabajo se plantea el objetivo de localizar fuentes sonoras a partir del procesado de señal de un array circular de seis micrófonos y técnicas de estimación del ángulo de llegada (DOA), compaginado con el uso de una red neuronal convolucional. Esta red neuronal convolucional es la encargada de proporcionarnos el ángulo de llegada de la fuente a partir de los mapas de energía acústica realizados mediante técnicas clásicas de procesado de señal.

Otro de los objetivos de este proyecto es que seamos capaces de llevarlo a cabo con hardware de bajo coste, de manera que seamos capaces de reducir los costes necesarios para ejecutar el proyecto. De esta manera, utilizaremos una VPU de Intel llamada Neural Compute Stick 2, encargada de realizar la inferencia de la red neuronal, y una Raspberry Pi 3 que sera la que asuma el coste computacional del resto del proyecto.

Teniendo en cuenta esto, presentamos dos implementaciones. La primera en Ubuntu, como medio de experimentación y para verificar el correcto funcionamiento del proyecto, y la segunda, una vez que tenemos el proyecto funcional, se traslada a la Raspberry Pi 3.

# ABSTRACT

Both the signal processing of sensor arrays and their applications are subjects that have been widely studied by researchers for years. In this work, the objective is to locate sound sources processing the signals captured by a circular array of six microphones and Direction of Arrival (DOA) estimation techniques, combined with the use of a convolutional neural network. This convolutional neural network is in charge of providing us with the angle of arrival of the source from the acoustic energy maps made using classical signal processing techniques.

Another of the objectives of this project is to be able to carry out it with low-cost hardware, so that we are able to reduce the costs necessary to execute the project. In this way, we will use an Intel VPU called Neural Compute Stick 2, in charge of making the neural network inference, and a Raspberry Pi 3 that will assume the computational cost of the rest of the project.

Taking this into account, we present two implementations. The first one in Ubuntu, as a means of experimentation and to verify the correct functioning of the project, and the second one, once we have the project working, is moved to the Raspberry Pi 3.

# Índice

**Lista de Figuras** **V**

**1. Introducción y objetivos** **1**

**2. Algoritmos de localización de fuentes sonoras** **3**
2.1. Técnicas de estimación de la posición de fuentes sonoras . . . . . . . . 3
2.1.1. Técnicas de localización basadas en *beamforming* . . . . . . . . 3
2.1.2. Técnicas basadas en técnicas de estimación espectral del alta resolución . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
2.1.3. Técnicas basadas en diferencias de tiempos de llegada (TDOA) . 4
2.1.4. Algoritmo SRP-PHAT . . . . . . . . . . . . . . . . . . . . . . . 5

**3. *Deep Learning*** **7**
3.1. ¿Qué es *Deep Learning*? . . . . . . . . . . . . . . . . . . . . . . . . . 7
3.2. Metodología de desarrollo de una red neuronal . . . . . . . . . . . . . . 8
3.2.1. Definición de una red neuronal . . . . . . . . . . . . . . . . . . . 8
3.2.2. Entrenamiento de la red neuronal . . . . . . . . . . . . . . . . . 8
3.2.3. Fase de Inferencia . . . . . . . . . . . . . . . . . . . . . . . . . . 10
3.3. Redes Neuronales Convolucionales, CNN . . . . . . . . . . . . . . . . . 10
3.3.1. Arquitectura CNN . . . . . . . . . . . . . . . . . . . . . . . . . 10
3.4. Modelo de red neuronal empleado . . . . . . . . . . . . . . . . . . . . . 13

**4. OpenVINO y el NCS2** **15**
4.1. OpenVINO . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
4.2. Neural Compute Stick 2 . . . . . . . . . . . . . . . . . . . . . . . . . . 16
4.2.1. Intel Movidius Myriad X . . . . . . . . . . . . . . . . . . . . . . 17
4.2.2. Especificaciones técnicas del NCS2 . . . . . . . . . . . . . . . . 18
4.2.3. NCS vs NCS2 . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19

**5. Implementación** **21**
5.1. Instalación y verificación OpenVINO . . . . . . . . . . . . . . . . . . . 21
5.1.1. Instalación y verificación de OpenVINO en Ubuntu . . . . . . . 21
5.1.2. Instalación y verificación de OpenVINO en Raspberry Pi 3 . . . 25
5.2. Desarrollo del código tiempo real . . . . . . . . . . . . . . . . . . . . . 26
5.2.1. Fase de desarrollo en Ubuntu . . . . . . . . . . . . . . . . . . . 26
5.2.2. Fase de desarrollo en Raspberry Pi 3 . . . . . . . . . . . . . . . 33

**6. Conclusiones** **35**

**7. Bibliografía** **37**

**Appendices** **38**

**A. Anexo 1. Instalación OpenVINO Toolkit** **41**
A.1. Instalación OpenVINO Toolkit en Ubuntu . . . . . . . . . . . . . . . . . 41
A.2. Instalación OpenVINO Toolkit en Raspberry Pi 3 . . . . . . . . . . . . . 44

**B. Anexo 2. Código de Python empleado** **45**

**C. Anexo 3. Código C++ del Proyecto** **49**

# Lista de Figuras


3.1. Esquema básico de una red neuronal . . . . . . . . . . . . . . . . . . . . 8
3.2. Esquema Red Neuronal Convolucional . . . . . . . . . . . . . . . . . . 11
3.3. Esquema del funcionamiento de la Capa de Convolución de una CNN . 11
3.4. Esquema del funcionamiento de la Capa de Reducción de una CNN . . 12
3.5. Esquema Capa Clasificadora de una CNN . . . . . . . . . . . . . . . . 13

4.1. Neural Compute Stick 2 . . . . . . . . . . . . . . . . . . . . . . . . . . 16
4.2. Arquitectura NCS2 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
4.3. Arquitectura Inter Movidius Myriad X . . . . . . . . . . . . . . . . . . 17
4.4. NCS2 vs NCS . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19

5.1. Imagen resultado de ./demo-security-barrier-camera.sh . . . . . . . . . 22
5.2. Mapa SRP número 5 . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
5.3. Mapa SRP número 30 . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
5.4. Resultado simulación . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
5.5. Resultado modelo Pytorch . . . . . . . . . . . . . . . . . . . . . . . . . 24
5.6. Mapa SRP número 15 . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
5.7. Mapa SRP número 25 . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
5.8. Array de micrófonos miniDSP UMA-8 . . . . . . . . . . . . . . . . . . 27
5.9. Representación canales de audio . . . . . . . . . . . . . . . . . . . . . 28
5.10. Ejemplo de mapas obtenidos . . . . . . . . . . . . . . . . . . . . . . . 29
5.11. Ejemplo salida por el terminal de la red neuronal . . . . . . . . . . . . 30
5.12. Eje de coordenadas en función del array . . . . . . . . . . . . . . . . . 31
5.13. Ejemplo salida de la red neuronal . . . . . . . . . . . . . . . . . . . . . 32
5.14. Ejemplo caso Elevación = 0º . . . . . . . . . . . . . . . . . . . . . . . 32
5.15. Diagrama de bloques del código C++ . . . . . . . . . . . . . . . . . . . 33

A.1. Pantalla de instalación de OpenVINO 1 . . . . . . . . . . . . . . . . . . 41
A.2. Pantalla de instalación de OpenVINO 2 . . . . . . . . . . . . . . . . . . 42
A.3. Pantalla de instalación de OpenVINO 3 . . . . . . . . . . . . . . . . . . 42
A.4. Pantalla de instalación de OpenVINO 4 . . . . . . . . . . . . . . . . . . 43

# Capítulo 1

# Introducción y objetivos

El objetivo principal de este trabajo es la implementación de un sistema de localización de fuentes sonoras en tiempo real basado en redes neuronales utilizando hardware de bajo coste. Para ello dispondremos de una Raspberry Pi 3, un Neural Compute Stick 2 de Intel, así como una red neuronal desarrollada por el director del proyector David Díaz-Guerra Aparicio, que nos permite estimar la dirección o ángulo de llegada de la señal sonora a partir de mapas de potencia acústica SRP-PHAT y una agrupación circular de micrófonos que serán los encargados de la recepción de las señales acústicas.

La localización de una fuente sonora es la capacidad del ser humano para determinar la ubicación de la misma en el espacio. Nuestro objetivo es conseguir replicar dicha capacidad mediante hardware y software, de manera que consigamos localizar una fuente sonora en movimiento a lo largo de un espacio determinado. Este proyecto podría servir para futuras aplicaciones que sirviesen de apoyo a ingenieros acústicos a la hora de localizar las fuentes de ruido en espacios cerrados, o localizar las reflexiones de las ondas sonoras. Esto podría ser de ayuda en aplicaciones de insonorización o de adaptación acústica de espacios. Otras aplicaciones podrían ser la localización de hablantes para aplicar posteriormente algoritmos de conformado de haz en su dirección en sistemas de reconocimiento del habla o el apuntado automático de cámaras en sistemas de videoconferencia o sistemas de audición para robots.

# Capítulo 2

# Algoritmos de localización de fuentes sonoras

El objetivo principal de los sistemas de localización de fuentes sonoras es conseguir la mejor precisión posible a la hora de estimar la posición de la fuente. La precisión de la estimación depende de varios factores: la calidad y la cantidad de micrófonos empleados en la estimación, la posición relativa de los micrófonos con respecto unos de otros, el ruido y los niveles de reverberación existentes en el ambiente, y el número de fuentes sonoras activas, así como su contenido espectral. Además todos estos factores son dependientes del espacio en el que se encuentre la fuente. Debido a esto, según la habitación o el espacio, la geometría del array de micrófonos podrá presentar variaciones. Aun teniendo en cuenta esta situación, existen algunas estrategias que afrontan estas dificultades de la manera mas general posible.

Además de la alta precisión, las estimaciones deben ser actualizadas frecuentemente para que puedan ser útiles en aplicaciones prácticas basadas en seguimiento y *beamforming*. Un *beamformer*, o conformador de haz, debe incluir un procedimiento de localización continuo y preciso en su algoritmo. Este requisito necesita del uso de un estimador de alta resolución, así como poseer un gran ratio de actualización. Cualquier estimador debe poseer baja latencia para que sea capaz de ejecutarse en sistemas de tiempo real.

## 2.1. Técnicas de estimación de la posición de fuentes sonoras

Las técnicas de localización de fuentes sonoras pueden clasificarse en tres grandes categorías: las basadas en encontrar la dirección que maximiza la energía a la salida de un *beamformer* (SRP), las que se basan en técnicas de estimación de conceptos de alta resolución espectral y las que emplean la información basada en la diferencia de tiempos de llegada (TDOA) [1].

### 2.1.1. Técnicas de localización basadas en *beamforming*

Estas técnicas se basan en realizar un barrido del espacio con diferentes filtros de *beamforming* con el fin de obtener un mapa de potencias, de forma que la posición de

las fuentes a estimar serán los máximos del mapa de potencias obtenido.

La estrategia más básica es utilizar la salida de filtros *delay-and-sum*. Esto es lo que se conoce comúnmente como *beamformer* o conformador de haz. Este tipo de filtros lo que hace es compensar los retardos sufridos por la propagación de las señales a cada uno de los micrófonos que forman la agrupación, para la dirección a la que se desea apuntar con la agrupación. Este tipo de filtros solo es óptimo cuando hay una única fuente a localizar, ya que, si hay más, sus prestaciones se ven altamente degradadas debido a que las demás fuentes pueden incidir en la agrupación por medio de los lóbulos secundarios de la primera. Además, en el caso de agrupaciones de micrófonos, contamos con entornos con gran reverberación, por lo que a pesar de que solo exista una fuente, las distintas reflexiones de la misma harán que el sistema pierda resolución.

### 2.1.2. Técnicas basadas en técnicas de estimación espectral del alta resolución

Esta segunda categoría de técnicas de estimación de localización incluye los métodos de *beamforming* adaptados del campo del análisis de alta resolución espectral, como pueden ser: el modelado de auto-regresión (AR), la estimación espectral de mínima varianza (MV), etc. Mientras que este tipo de técnicas han obtenido grandes resultados en diversas aplicaciones de procesado de agrupaciones, la mayoría son poco robustas en entornos reverberantes, por lo que esto limita su efectividad en problemas de localización de fuentes sonoras. Por dicho motivo, este tipo de técnicas no las hemos considerado útiles a la hora de afrontar nuestro problema de localización de fuentes sonoras.

### 2.1.3. Técnicas basadas en diferencias de tiempos de llegada (TDOA)

En esta tercera categoría nos encontramos con las técnicas basadas en las diferencias de tiempos de llegada a la agrupación, (TDOA referenciando sus siglas en ingles). Para este tipo de técnicas se adoptan estrategias en dos pasos: en primer lugar se realiza una estimación de los tiempos de retardo (TDE) y a continuación una búsqueda espacial para obtener la posición para la cual se obtiene la máxima verosimilitud de los TDOA obtenidos, siendo esa la posición en la que se encuentra la fuente. Las dos fuentes principales de degradación de la señal que generan un problema de estimación son el ruido de fondo y la rutas multicamino debidas a la reverberación de las salas. Asumiendo señales Gaussianas estacionarias e incorreladas y fuentes de ruido con estadísticas conocidas y sin multicamino, la estimación temporal esta basada en la función de Correlación Cruzada Generalizada (Generalized Cross-Correlation, GCC):

$$R_{nm}(\tau) = \frac{1}{2\pi} \int_{-\infty}^{\infty} \Psi_{nm}(\omega) X_n(\omega) X_m^*(\omega) e^{j\omega\tau} d\omega \tag{2.1}$$

Donde $\Psi_{nm}(\omega)$ corresponde con la transformación de fase y $X_n(\omega)$ y $X_m(\omega)$ las señales de los micrófonos $n$ y $m$ para los cuales queremos obtener su correlación cruzada.

Para este tipo de problemas en concreto, se suele utilizar la transformación de fase (Phase Transform, PHAT), ya que ha obtenido una considerable atención en los últimos años en base a los sistemas de localización de fuentes sonoras. En esta

transformación de fase se divide entre los módulos de las señales, de forma que estas quedan blanqueadas:

$$\Psi_{nm}(\omega) = \frac{1}{|X_n(\omega)X_m^*(\omega)|} \tag{2.2}$$

Al colocar el mismo énfasis en cada una de las componentes del espectro de fases cruzadas, el pico resultante de la función GCC-PHAT corresponde con el retardo dominante en la señal reverberada. El interés principal de esta transformación reside en que la correlación cruzada estándar da una mayor importancia a las frecuencias que poseen un mayor contenido energético, sin embargo, a la hora de calcular el retardo entre señales, las zonas frecuenciales con un menor contenido energético también aportan información relevante que nos ayudan a mejorar la estimación.

El problema principal de estas técnicas reside en el hecho de que en el paso de la estimación de tiempos a la estimación de la posición se pierde una gran cantidad de información, de forma que si el máximo de las GCC elegido no se corresponde con el verdadero TDOA, la estimación de la posición que obtendremos será completamente errónea, aunque las GCCs tuvieran un máximo local en el TDOA verdadero. A pesar de esto, una de las grades ventajas de este tipo de técnicas es que suelen representar un bajo coste computacional.

### 2.1.4. Algoritmo SRP-PHAT

Después de describir todo lo anterior, ahora nos centraremos en describir el algoritmo SRP-PHAT [1]. Para las tecnicas de TDE basadas en GCC, la transformación PHAT ha proporcionado una mayor robustez en entornos con condiciones de poca o moderada reverberación. El algoritmo SRP-PHAT es capaz de combinar la robustez de las de las técnicas SRP con la resolución que permite obtener la transformación PHAT, junto con un relativo bajo coste computacional.

Una de las ventajas es que podemos demostrar que la potencia estimada mediante las técnicas SRP puede expresarse en términos de GCCs:

$$\begin{aligned} P(\theta_0) = \int_{-\infty}^{\infty} |Y(\omega)|^2 d\omega = \int_{-\infty}^{\infty} |\sum_{n=1}^{N} G_n(\omega)X_n(\omega)e^{-j\omega\tau(\theta_0)}|^2 d\omega = \\ \sum_{n=1}^{N}\sum_{m=1}^{M}\int_{-\infty}^{\infty}(G_n(\omega)X_n(\omega)(G_m(\omega)X_m(\omega))e^{-j\omega\tau(\theta_0)} = \\ 2\pi\sum_{n=1}^{N}\sum_{m=1}^{M} R_{nm}(\Delta\tau_{nm}(\theta_0)) \end{aligned} \tag{2.3}$$

donde $G_n$ y $G_m$ son los filtros SRP-PHAT utilizados, y $X_n$ y $X_m$ las señales recibidas añadiéndole un retardo. De la misma forma en la última ecuación, $R_nm$ representa la GCC PHAT ponderada de las señales emésima y enésima. De esta forma la transformación utilizada en las GCCs es equivalente al producto de los filtros utilizados en el beamforming para cada una de las señales:

$$\Psi_{n_m}(\omega) = G_n(\omega)G_m^*(\omega) \tag{2.4}$$

La ecuación (2.3) equivale a la suma de todos los pares de permutaciones de las GCC posibles separadas en el tiempo debido a los diferentes retardos en función de la posición. En este sumatorio esta incluida la suma de las N autocorrelaciones, que coincide con la GCC evaluada con un retardo igual a cero. Estos términos solo contribuyen a un offset de DC en la potencia de la dirección de la respuesta, ya que son independientes a los retardos.

Una de las principales ventajas de este algoritmo es que puede implementarse con un coste computacional relativamente bajo en comparación con otras técnicas. Para ello, se realizan las FFTs de las señales que son capturadas por cada uno de los sensores, aplicando la transformación de fase (2.2) normalizando el módulo de cada bin frecuencial, multiplicando todas las señales entre si, y por último se realiza las iFFTs para obtener las correlaciones cruzadas. Después de realizar estos cálculos, para estimar la potencia que recibimos de cada una de las direcciones, bastará con sumar la componente adecuada de cada GCC.

Como conclusión, lo fundamental de este algoritmo radica en su eficiente implementación computacional basada en el cálculo de Correlaciones Cruzadas Generalizadas (GCCs) mediante el uso de FFTs. Esta es una de las razones por la cual su coste computacional es inferior al de otras técnicas existentes.

# Capítulo 3

# *Deep Learning*

## 3.1. ¿Qué es *Deep Learning*?

El *Deep Learning*, o aprendizaje profundo, es una familia de algoritmos que pretenden emular el aprendizaje humano con el fin de poder llegar a resolver distintos problemas que no han podido ser resueltos hasta ahora mediante métodos analíticos [2]. Los algoritmos de *Deep Learning* se basan en el auto-aprendizaje, es decir, no esta basado en reglas preestablecidas si no que, mediante una fase previa a la se denomina entrenamiento, es capaz de encontrar las soluciones a los problemas que se plantean. Se caracteriza por estar compuesto por capas, las cuales a su vez están formadas por pesos. Estos pesos son los encargados de realizar las operaciones determinadas en la fase de entrenamiento para obtener los resultados deseados.

Estos algoritmos están estructurados por capas. En un primer lugar se encuentra la Capa de Entrada o *Input Layer*, que es la encargada de recibir los datos de entrada a la red neuronal. Las entradas pueden representar imágenes, tablas de datos o cualquier otro tipo de entrada que precise el modelo con el que vamos a trabajar. Al siguiente conjunto de capas que componen la red se le denomina Capa Oculta o *Hidden Layer*. Aquí se realiza el procesado de los datos obtenidos por la Capa de Entrada y el calculo de los pesos de la red. Dependiendo de la complejidad del modelo, puede estar formada por múltiples subcapas. La última capa de la arquitectura de las redes neuronales es la Capa de Salida u *Output Layer*. En esta capa se obtienen los resultados a los que ha llegado el modelo. Durante la fase de entrenamiento, con el fin de que el modelo de red consiga obtener los valores adecuados de los pesos, se deben proporcionar los valores de salida a los que deseamos que llegue la red. Una vez obtenidos los pesos, la red debe ser capaz de obtener el resultado correcto por ella misma.

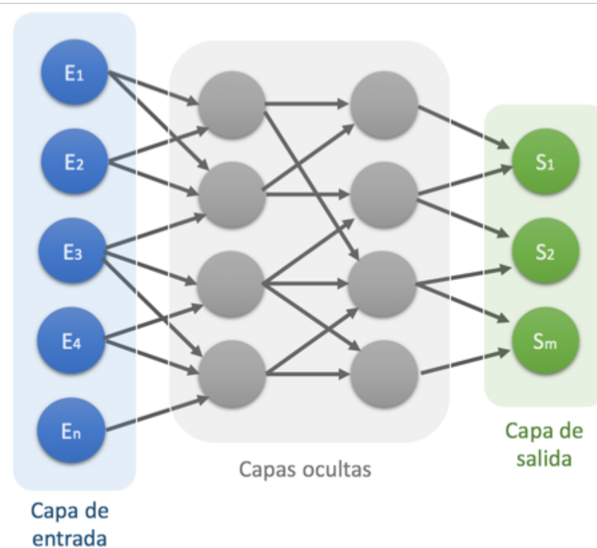


Figura 3.1: Esquema básico de una red neuronal

## 3.2. Metodología de desarrollo de una red neuronal

Una red neuronal artificial comprende varias fases para su desarrollo. A continuación va a ver un desglose de las diferentes etapas que hay que llevar a cabo para desarrollar una red neuronal:

### 3.2.1. Definición de una red neuronal

Esta fase comprende la elección del modelo de red, así como la obtención del conjunto de variables necesarias y significativas para la resolución del problema.

A la hora de definir la red neuronal, se deben tener en cuenta aspectos como el tamaño de la red con la que queremos trabajar, el tipo de problema para el que vamos a utilizar la red, el tipo de asociación para realizar la fase de entrenamiento, es decir, realizar un aprendizaje supervisado o no supervisado, etc. Una de las mayores dificultades reside en la selección del número de capas ocultas de la red y del número de perceptrones por capa. A la hora de tomar esta decisión tenemos que llegar a un compromiso entre la precisión del modelo y su complejidad y tamaño. Además, una red con demasiados parámetros entrenables será más propensa a sufrir problemas de sobrenentrenamineto si no tenemos un dataset lo suficientemente grande [3].

### 3.2.2. Entrenamiento de la red neuronal

Lo que mayor interés ha atraído de las redes neuronales es su capacidad de entrenamiento. Dada una determinada tarea que hay que resolver, y una clase de funciones $F$, el entrenamiento consiste en utilizar un conjunto de observaciones para encontrar $f^* \in F$, la cual resuelva la tarea de la manera mas óptima.

Esto hace necesario la definición de una función de coste $C : F \to \mathbb{R}$ tal que para la función óptima $f^*$, $C(f^*) \leq C(f)\ \forall\ f \in F$ . Es decir, ninguna solución

tiene un coste menor que el coste de la solución óptima. La función de coste $C$ es un concepto importante en el aprendizaje, ya que representa lo lejos que una solución particular se encuentra de la solución óptima a la hora de resolver el problema. Los algoritmos de aprendizaje se basan en minimizar la función de coste.

### Paradigmas de entrenamiento

Existen tres grandes paradigmas de entrenamiento, cada uno con una tarea de aprendizaje particular. Estos grandes paradigmas son: **aprendizaje supervisado**, **aprendizaje no supervisado** y **aprendizaje por refuerzo**.

**Aprendizaje supervisado:** El aprendizaje supervisado es una técnica de entrenamiento de redes neuronales para deducir una función a partir de datos de entrenamiento. Estos últimos consisten en pares de objetos, los cuales normalmente son vectores. Una componente del par son los datos de entrada a la red, mientras que el otro par son los resultados a los que queremos que nuestra red neuronal llegue como resultado. Las salidas de la función pueden ser un valor numérico o una etiqueta de clase, dependiendo el problema para el cual esté diseñada nuestra red neuronal. El objetivo de este tipo de aprendizaje supervisado es crear una función capaz de predecir los valores correspondientes a cualquier entrada de la red a partir de una serie de ejemplos. Es decir, mediante los datos de entrenamiento ayudamos a la red a encontrar los valores adecuados de salida para todo tipo de valores de entrada.

Cada ejemplo del conjunto de testeo contiene los valores de las variables de entrada, así como su solución. La diferencia con la fase anterior es que ahora las soluciones no son entregadas a la red, de modo que esta llega por si sola a sus propias conclusiones a partir de los valores de entrada otorgados. Una vez que la red obtienes sus valores de salida, los comparamos con la solución conocida. En esta fase, el problema consiste en identificar cuando la salida de la red ha de considerarse como la correcta.

**Aprendizaje no supervisado:** El aprendizaje no supervisado es un método de entrenamiento donde un modelo se ajusta a las observaciones. La principal diferencia con el método anterior (Aprendizaje supervisado), radica en que no hay conocimiento a priori de los resultados a los que tiene que llegar la red neuronal. El aprendizaje no supervisado trata los objetos de entrada como un conjunto de variables aleatorias, construyéndose así un modelo de densidad para el conjunto de datos.

Las principales características del aprendizaje no supervisado son la no necesidad de un profesor externo, es decir, no es necesario indicarle cual es el resultado al que debe llegar, sino que llega a el por si misma. Suele requerir menor tiempo de entrenamiento que las redes redes con aprendizaje supervisado, debido a que suelen ser redes más simples (una sola capa o Feed-forward). Algunos de los posibles problemas que pueden abordar este tipo de redes son: familiaridad, análisis de componentes principales, agrupamiento, etc.

**Aprendizaje por refuerzo:** El aprendizaje por refuerzo es un área del aprendizaje automático cuya ocupación es determinar las acciones que debe tomar la red neuronal en un entorno dado con el fin de maximizar el aprendizaje. Es decir, es el procedimiento

por el cual la red neuronal toma una serie de decisiones con el fin de encontrar la solución óptima al problema que se le plantea de la manera más eficiente posible. El aprendizaje por refuerzo se distingue del aprendizaje supervisado en que los pares de entrada y salida son desconocidos. Además el enfoque de este tipo de aprendizaje radica en encontrar un equilibrio entre la exploración y la explotación de los conocimiento actuales.

### 3.2.3. Fase de Inferencia

Una vez hemos realizado el entrenamiento de la red neural, ya hemos obtenido el valor de los pesos con los que trabaja la red para obtener la salida. El siguiente paso que debemos realizar, es verificar que la red se esta comportando como nosotros queremos que lo haga, es decir, que las salidas que obtenemos a la salida de la misma son las soluciones correctas al problema que estábamos queriendo resolver. Para ello, para validar que los resultados que estamos obteniendo de la red son los que resuelven realmente el problema que planteamos haremos uso de otro conjunto de datos, que denominaremos conjunto de validación o testeo.

A partir de este punto, podríamos distinguir dos visiones distintas en cuanto al paradigma de inferencia se refiere: la de la investigación y la del desarrollo de aplicaciones o productos. En cuanto a la fase de inferencia desde el punto de vista de la investigación, esta estaría fundamentada en realizar el testeo de la red para verificar su correcto funcionamiento, ya que lo importante en la investigación es conseguir el correcto funcionamiento del sistema. En cambio, desde el punto de vista del desarrollo, la fase de inferencia no solo estaría fundamentada en el testeo, que también es necesario para verificar el correcto funcionamiento del producto, sino que se tendría que extrapolar a una aplicación práctica, de forma que en este caso la fase de inferencia estaría condicionada a la vida de la aplicación o del producto, es decir, la fase de inferencia equivaldría a todas las veces que este se utiliza.

## 3.3. Redes Neuronales Convolucionales, CNN

Las Redes Neuronales Convolucionales son similares a las redes neuronales multicapa. Sin embargo, estas presentan una clara ventaja frente a las redes neuronales multicapa, siendo esta ventaja la capacidad de que cada parte de la red se entrena para llevar a cabo una cierta tarea. Esto reduce significativamente el número de capas ocultas, reduciendo el tiempo de la fase de entrenamiento [4].

Este tipo de redes son muy potentes para todo lo que tiene que ver con el análisis de imágenes, siendo extrapolable al análisis de señales de audio, siempre y cuando realicemos ciertas modificaciones en estas últimas para que la red las admita. Este gran rendimiento, es debido a que son capaces de detectar características simples como puede ser detección de bordes, lineas, etc, y componerlas en características mas complejas hasta detectar el resultado buscado.

### 3.3.1. Arquitectura CNN

Una Red Neuronal Convolucional es una red multicapa que consta de capas convolucionales y capas de reducción alternadas, existiendo finalmente una capa de

conexión total entre todas ellas, y que nos proporciona la salida de la red.

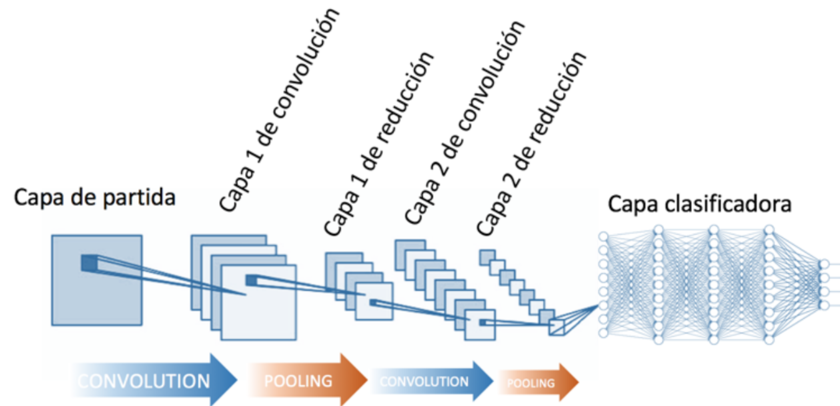


Figura 3.2: Esquema Red Neuronal Convolucional

En la figura 3.2 se muestra un esquema completo de una Red Neuronal Convolucional, diferenciando las distintas capas que la componen.

### Capa de Convolución

En esta capa se realizan operaciones de productos y sumas entre la capa de partida y los n filtros (o kernels) que genera un mapa de características de la red. Dichas características se corresponden con cada una de las posibles ubicaciones del filtro en los datos de entrada a la red.

La ventaja es que el mismo filtro se puede utilizar para extraer la misma característica en cualquier parte de la entrada, consiguiendo con esto reducir el número de conexiones y el número de parámetros a entrenar en comparación con una red multicapa de conexión total. En la siguiente figura 3.3, se muestra un ejemplo de como funciona la convolución en este tipo de redes.

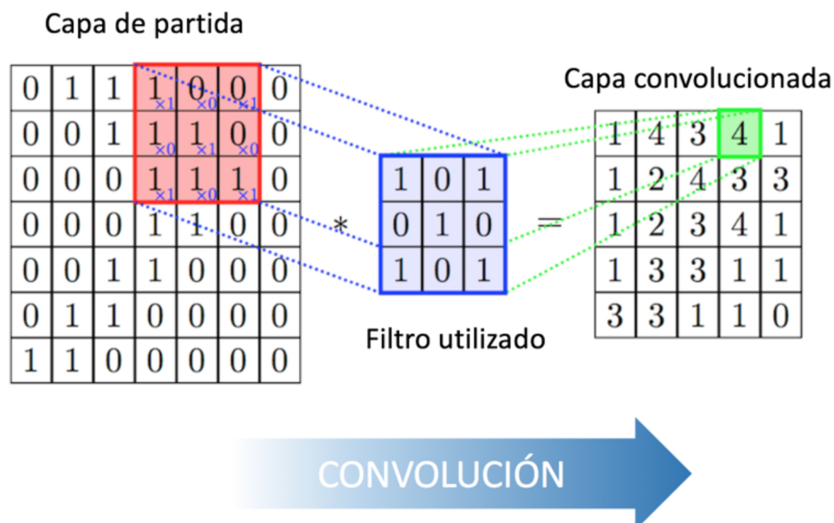


Figura 3.3: Esquema del funcionamiento de la Capa de Convolución de una CNN

Como podemos observar, existe una capa de entrada por la que pasamos un filtro: realizamos las multiplicaciones de la entrada por sus filtros para después sumar el resultado de cada una de la multiplicaciones para obtener la capa convolucionada.

### Capa de Reducción

En esta capa, el objetivo es disminuir la cantidad de parámetros en la red al quedarse con las características más comunes.

La forma de reducir parámetros se realiza mediante la extracción de estadísticas como el promedio o el máximo de una región fija del mapa de características. Al reducir las características, el método pierde precisión espacial, pero se obliga a la red a generalizar. Por lo general, a lo largo de las capas de una red convolucional se va reduciendo la resolución de los mapas de características pero se va aumentando el número de canales, de forma que se va reduciendo la información espacial y se va ganando información de más alto nivel.

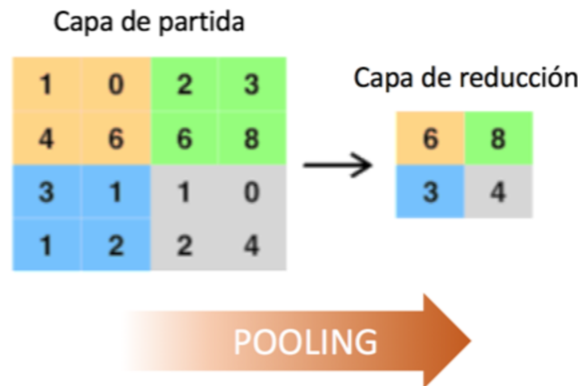


Figura 3.4: Esquema del funcionamiento de la Capa de Reducción de una CNN

En la figura 3.4 se muestra uno de los modelos de reducción mencionados con anterioridad. En dicha imagen se utiliza el método de reducción mediante la elección del máximo de una región fija de la capa de partida.

### Capa Clasificadora

Al final del conjunto de capas de convolución y reducción, se suelen utilizar capas conectadas en la que cada uno de los píxeles se considera como una neurona separada de las demás. La última capa tendrá tantas neuronas como el número de clases a predecir.

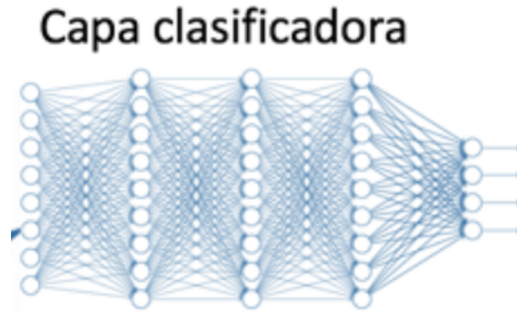


Figura 3.5: Esquema Capa Clasificadora de una CNN

En la figura 3.5, se muestra un ejemplo de una Capa Clasificadora estándar de una Red Neuronal Convolucional.

## 3.4. Modelo de red neuronal empleado

En este proyecto hemos utilizado un modelo de red neuronal diseñada por el codirector del proyecto David Díaz-Guerra Aparicio [5], cuyo nombre es Cross3D. Esta red usa como entrada mapas de potencia acústica SRP-PHAT y mediante capas convolucionales 3D causales realiza la inferencia de la dirección de llegada de la fuente de audio.

Dicha red neuronal debe tener como entrada un tensor de 4 dimensiones con tamaño C x T x $N_\theta$ x $N_\varphi$, siendo C=3 el número de canales y estando construido el primero a partir del computo de T mapas SPR-PATH temporalmente equiespaciados, con $N_\theta$ ángulos de elevación equiespaciados en un rango $\theta \epsilon (0, \pi/2)$ y $N_\varphi$ ángulos de azimut en un rango de $\varphi \epsilon$ (-$\pi$,$\pi$). Los dos canales siguientes contienen las coordenadas del máximo de de cada uno de los T mapas, ya que su cómputo apenas aumenta la complejidad del sistema y permite mejorar su resolución.

Este modelo esta formado por convoluciones 3D, de manera que nos permite realizar la convolución de las variables de elevación, azimut y temporales. El hecho de que las convoluciones utilizadas sean causales nos otorga la ventaja de que puede ser utilizada en aplicaciones de tiempo real sin presentar ningún tipo de retardo. Sin embargo, aquí radica uno de los problemas que se nos han presentado durante el proyecto, debido a la incompatibilidad entre el Neural Compute Stick 2 y nuestro modelo de red neuronal. Dicha incompatibilidad es producto de las operaciones que es capaz de realizar el Neural Compute Stick 2, que para nuestra desgracia no soportaba las convoluciones 3D. Para resolver dicho problema, fue necesario modificar el modelo.

La principal diferencia entre este modelo y el anterior radica en las operaciones que se han llevado a cabo. Debido a que el Neural Compute Stick 2 no es capaz de realizar las operaciones que se tenían que realizar mediante una convolución 3D, se tuvo que tomar la decisión de sustituirla por dos convoluciones, es decir, una convolución de 2D y una convolución de 1D. La convolución 2D se realizaría entre las variables angulares de la elevación y del azimut, y la convolución de 1D se realizaría sobre la variable temporal, de manera que mantendríamos su causalidad y nos seguiría permitiendo utilizar este modelo en aplicaciones que se quieren llevar a cabo en tiempo real.

# Capítulo 4

# OpenVINO y el NCS2

El Deep Learning está demandando nuevo hardware más potente y específicamente diseñado para las cargas de trabajo y cálculos diferentes a las actuales, como pueden ser predicción e inferencia. Si se desean realizar grandes avances en Deep Learning, el hardware debe avanzar paralelamente. La inteligencia artificial ha estado presente durante los 60 últimos años mediante el desarrollo de algoritmos y redes de aprendizaje profundo, pero siendo inviable llevarlas a la práctica debido a la falta de desarrollo de hardware que los soportase y sacase su máximo rendimiento. Esto demuestra la importancia del hardware, ya que sin él, es imposible explotar al máximo todas las oportunidades que nos ofrece el software. Durante muchos años, el hardware encargado de realizar la inferencia han sido las tarjetas gráficas, pero estas necesitan una gran energía para ser alimentadas y además, su tamaño ha imposibilitado el desarrollo de otro tipo de productos, ya que su movilidad no es una de sus principales características. Para poder mejorar dicho defecto, durante los últimos años se han estado desarrollando dispositivos hardware de menor tamaño, capaces de realizar inferencias de Deep Learning con un bajo coste energético. Sin embargo, la fase de entrenamiento se sigue realizando con el uso de tarjetas gráficas, debido a que es un proceso que requiere de una gran potencia, así como un gran desgaste energético siendo las tarjetas gráficas el hardware mas adecuado para proporcionarlo.

## 4.1. OpenVINO

OpenVINO es un entorno que permite la aceleración de aplicaciones basadas en visión por computadora de alto rendimiento, así como en inferencias de Deep Learning. Este entorno ayuda a los desarrolladores a acelerar las cargas de trabajo de visión por computador, agilizar las implementaciones de Deep Learning, así como ofrecer una facilidad de uso y acelerar el tiempo de comercialización de las aplicaciones.

OpenVINO Toolkit esta compuesto por dos principales herramientas: el “Model Optimizer” y el “Inference Engine”.

Model Optimizer nos permite preparar el algoritmo con el que vayamos a trabajar para poder utilizarlos en las distintas plataformas de hardware de Intel, como es en nuestro caso el Neural Compute Stick 2. Para ello, en lo relacionado a Deep Learning, se encarga de obtener el formato de Representación Intermedia (IR). La IR es la estructura de datos utilizada internamente por el compilador para representar el

código fuente y sirve de nivel intermedio entre la descripción del modelo mediante lenguajes o *frameworks* de alto nivel (como Pytorch) y el código máquina específico para cada hardware. De esta manera, utilizando el “Model Optimizer” sobre nuestro modelo, obtendremos los ficheros“.bin”, que contiene los datos binarios de los pesos y sesgos del modelo, y “.xml”, que describe la topología de la red. Una vez que tenemos nuestro modelo en el formato de Representación Intermedia (IR), la herramienta “Inference Engine” es la encargada de ejecutarlo, ya sea en la CPU, GPU o en la VPU de Intel, siendo este el Neural Compute Stick 2 [6].

## 4.2. Neural Compute Stick 2

Una unidad de procesamiento de visión (VPU) es una clase emergente de microprocesador; es un tipo específico de acelerador de Deep Learning , diseñado para acelerar las tareas de visión artificial.

Figura 4.1: Neural Compute Stick 2

Lo que parece una memoria USB estándar, tal como se muestra en la figura 4.1, oculta mucho más en su interior. El Neural Compute Stick 2 esta construido sobre la última versión de VPU desarrollada por Intel: Intel Movidius Myriad X. Es el primero en presentar el motor de cómputo neural, un acelerador de hardware dedicado para las inferencias de redes neuronales profundas, ofreciendo de esta manera un mayor rendimiento respecto a versiones anteriores. NCS2 ofrece simplicidad de plug-and-play, así como soporte para frameworks comunes, proporcionando así una mayor flexibilidad para los desarrolladores a la hora de crear nuevos prototipos. En la figura 4.2 se muestra la arquitectura del Neural Compute Stick 2.

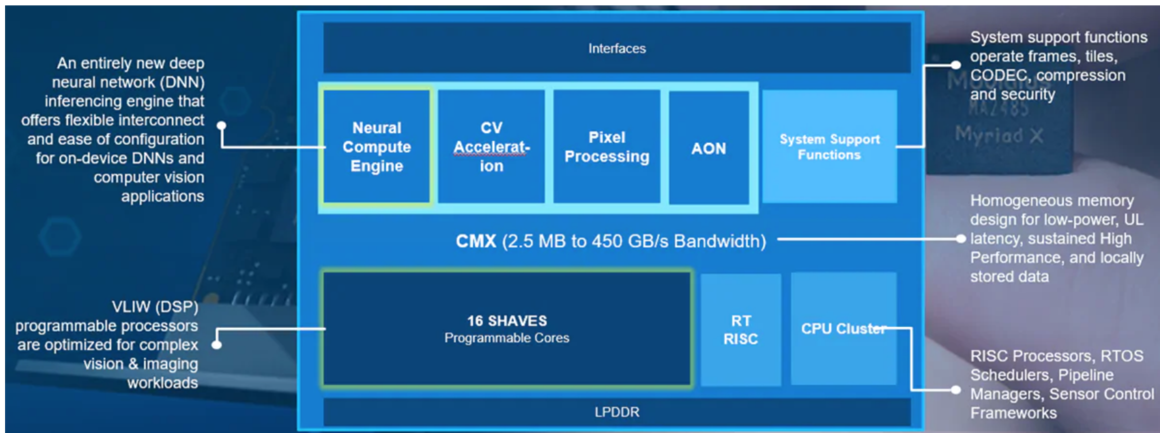


Figura 4.2: Arquitectura NCS2

### 4.2.1. Intel Movidius Myriad X

Movidius Myriad X es la última versión de unidad de procesamiento de visión (VPU) de Intel [7]. Es capaz de ofrecer un rendimiento total de más de 4 billones de operaciones por segundo. Es una unidad diseñada para proporcionar aplicaciones avanzadas de visión e inteligencia artificial a dispositivos como drones, cámaras inteligentes, hogares inteligentes, seguridad, etc.

Movidius Myriad X es el primer "Sistema en Chip (SoC)" del mundo que lleva incorporado un motor neuronal (Neural Compute Engine) que ayuda a acelerar las inferencias del aprendizaje profundo. El Neural Compute Engine es un bloque de hardware diseñado específicamente para su uso en redes neuronales, capaz de realizar inferencias a alta velocidad y con un bajo consumo energético, de forma que permite a los dispositivos responder a sus entornos en tiempo real. El Neural Compute Engine esta integrado como parte de la arquitectura Movidius VPU de eficiencia energética, tal y como se muestra en la figura 4.3, que minimiza el consumo energético al reducir el movimiento de datos en el chip.

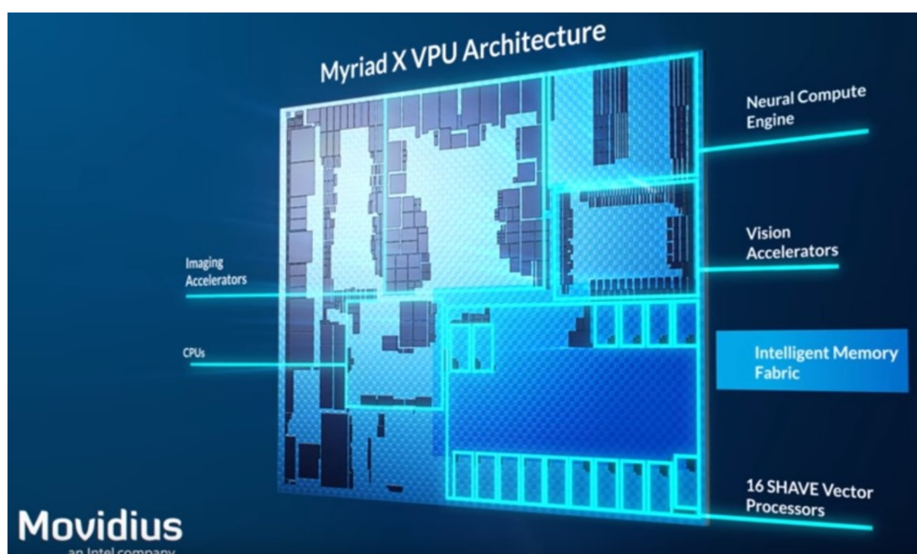


Figura 4.3: Arquitectura Inter Movidius Myriad X

Algunas de las principales características del Intel Movidius Myriad X son:

**Procesadores vectoriales VLIW programables de 128-bit:** Capaces de utilizar múltiples aplicaciones de imágenes y visión de forma simultánea.

**Carriles MIPI más configurables:** En el caso de que quisiésemos trabajar con imágenes, permite conectar hasta 8 cámaras RGB de resolución HD directamente al Myriad X.

**Aceleradores para visión mejorados:** Utiliza más de 20 aceleradores de hardware para realizar tareas como el flujo óptico, sin que ello suponga demasiada carga de trabajo añadida.

**5 MB de Memoria Homogénea sobre Chip:** La arquitectura de memoria sobre chip centralizada permite hasta 450 GB por segundo de ancho de banda interno, minimizando así la latencia y reduciendo el consumo de energía gracias a la reducción de la transferencia de datos fuera del chip.

### 4.2.2. Especificaciones técnicas del NCS2

#### Hardware

**Procesador:** Intel Movidius Myriad X Vision Processing Unit (VPU).

**CInfraestructuras soportadas:** TensorFlow, Caffe, Apache MXNet, Open Neural Network Exchange (ONNX), PyTorch, y PaddlePaddle mediante una conversion a ONNX.

**Conectividad:** USB 3.0 Type-A.

**Dimensiones:** 2.5 mm x 27 mm x 14 mm.

**Temperatura de funcionamiento:** 0º C a 40º C

#### Software

**Intel Distribution of OpenVINO toolkit Supported operating systems:**

**Ubuntu:** 16.04.3 LTS (64 bit), 18.04 LTS (64 bit).

**CentOS:** 7.4 (64 bit).

**Windows:** 10 (64 bit).

**macOS:** 10.14.4 (o posterior).

**Raspbian**

### 4.2.3. NCS vs NCS2

Debido a que este nuevo modelo NCS2 posee un mayor número de núcleos de computo que el modelo anterior NCS y además tiene acceso al kit de herramientas “Intel Distribution of OpenVINO”, el Intel NCS2 nos ofrece un aumento de rendimiento ocho veces mayor con respecto a la generación anterior. En la figura 4.4, se muestra una comparación entre los dos modelos en los que se muestra dicho aumento de rendimiento por parte del nuevo modelo NCS2.

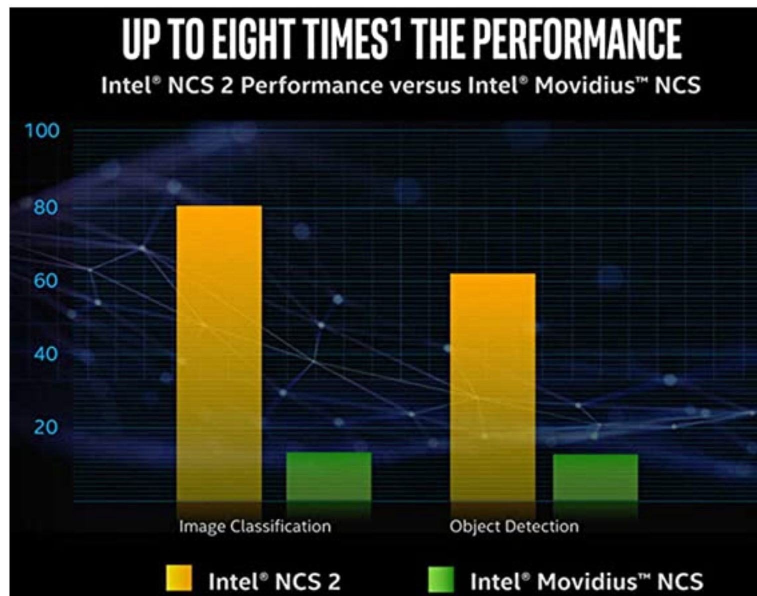


Figura 4.4: NCS2 vs NCS

# Capítulo 5

# Implementación

En este capitulo se explica el procedimiento que se ha llevado a cabo hasta lograr los resultados finales. Para ello, se pueden ver las distintas fases del proyecto:

1. Instalación y verificación de OpenVINO

   Instalación y verificación de OpenVINO en Ubuntu

   Instalación y verificación de OpenVINO en Raspberry Pi 3

2. Desarrollo del código en tiempo real

   Fase de desarrollo en Ubuntu

   Fase de desarrollo en Raspberry Pi 3

Una vez presentadas las diferentes fases por las que hemos pasado a la hora de implementar el proyecto, vamos a proceder a explicar en qué han consistido cada una de ellas.

## 5.1. Instalación y verificación OpenVINO

### 5.1.1. Instalación y verificación de OpenVINO en Ubuntu

Como punto de partida del proyecto, se decidió utilizar el VPU de Intel llamado Neural Compute Stick 2. Como ya hemos comentado en el capitulo 4, dicho VPU presenta las características que nosotros buscábamos para poder llevar a cabo nuestro proyecto, siendo estas la capacidad de realizar inferencias de bajo coste energético y su tamaño portátil. Una vez elegido, procedimos a realizar la instalación del **OpenVINO Toolkit** en Linux, más concretamente en una maquina virtual de Ubuntu 18.04. Elegimos este sistema operativo ya que era el que mayor facilidad nos ofrecía a la hora de trabajar con él. Para realizar la instalación de este set de herramientas, fue necesario seguir las instrucción de instalación que nos ofrecía el propio distribuidor, en este caso Intel. La instalación se basa principalmente en la instalación del **"Model Optimizer"** y del **"Inference Engine"**, ya mencionados en el capítulo 4, así como de las variables de entorno necesarias para el correcto funcionamiento tanto de este set de herramientas como de cualquier aplicación que se quiera ejecutar mediante el uso de OpenVINO y la instalación de los drivers necesarios para el funcionamiento del VPU Neural Compute Stick 2. Dicho proceso de instalación completo está detallado en el Anexo A. Una

vez terminado dicho proceso de instalación, el propio distribuidor Intel nos ofrece una serie de modelos públicos para verificar el correcto funcionamiento del set de herramientas. Nos proporcionan el ejecutable **"demo-security-barrier-camera.sh"**, que al ejecutarlo con las variables de entorno activadas, y si la instalación se ha realizado de forma correcta, nos devolverá la imagen de la figura 5.1 y podremos concluir con que la instalación se ha realizado de forma correcta. Dicha ejecución se realizó tanto para la CPU como para la VPU para verificar que ambos son capaces de ejecutar el modelo en cuestión.

Figura 5.1: Imagen resultado de ./demo-security-barrier-camera.sh

Una vez que verificamos dicha instalación, se decidió realizar una segunda comprobación para verificar realmente que había sido un existo. Dicha verificación se realizó de nuevo tanto para CPU como para VPU, para asegurarnos del correcto funcionamiento de ambos, aunque, en nuestro caso, el mas interesante es el funcionamiento de la VPU. Para ello, Intel nos da la opción de compilar los modelos públicos de la librería **ncappzoo** de Github; en nuestro caso, se escogió el modelo **age-gender-recognition-retail-0013**. Para ello, se descargaron los ficheros ".bin" y ".xml", que como ya hemos comentado en el capitulo 4, son los ficheros de representación intermedia (IR) del modelo. Dicho modelo realiza la predicción de edades a partir de imágenes de personas captadas por una cámara en tiempo real. Una

vez verificado su funcionamiento correcto con modelos ya previamente compilados y que conocíamos su funcionamiento de antemano, procedimos a realizar las distintas pruebas con nuestro propio modelo.

A la hora de trabajar con nuestro modelo, lo primero que tenemos que realizar es la compilación a representación intermedia (IR), es decir, obtener de nuestro modelo realizado en Pytorch, cuya extensión es ".onnx", los archivos ".bin" y ".xml". Para ello, Intel nos proporciona el programa el **"Model Optimizer"**, cuyo uso también se explica en el anexo A. Una vez realizada esta compilación, programamos un script utilizando el lenguaje de programación Python, Anexo B, para comprobar su funcionamiento. En dicho *script* utilizábamos como datos de entrada unas grabaciones simuladas, muestreadas a 16kHz, y de 20 segundos de duración. La idea de dicho *script* es comparar los resultados obtenidos en la simulación en Pytorch con los obtenidos al utilizar el modelo compilado en openVINO, para verificar el correcto funcionamiento de este último. Dichas señales de entrada son enventanadas en 103 ventanas de 4096 muestras; como estamos trabajando con señales originarias de una simulación, disponemos de los ángulos de llegada originales en cada una de las ventanas. Una vez enventanada la señal, se calculan los mapas de potencia mediante los algoritmos SRP, que formaran parte de la entrada a la red neuronal. Dichos mapas se calculan para cada una de las ventanas, teniendo una resolución de 16x32. En las figuras 5.6 y 5.7 podemos observar ejemplos de los mapas calculados para la simulación, con la posición simulada de la fuente de ese mapa resaltada. Una vez calculados dichos mapas, obtenemos los índices de las posiciones de los máximos normalizados para cada una de las dos dimensiones, ya que el modelo de red con el que estamos trabajando necesita que estos formen parte de los parámetros de entrada a la red.

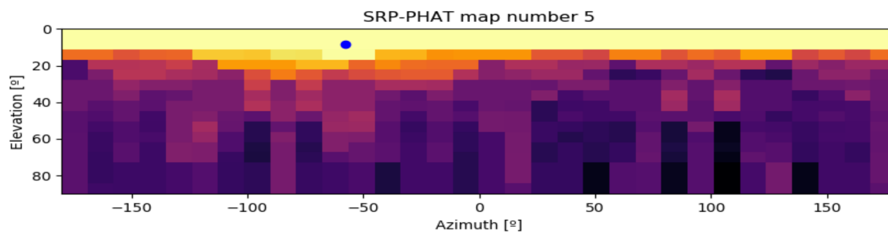


Figura 5.2: Mapa SRP número 5

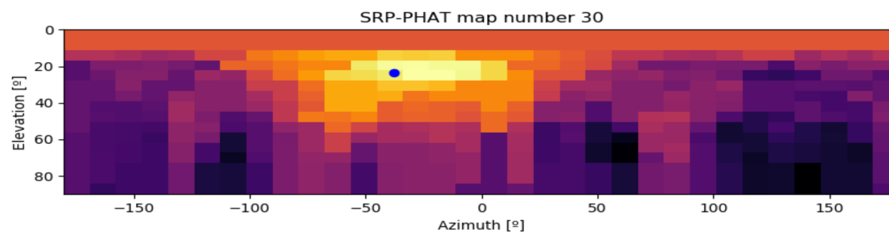


Figura 5.3: Mapa SRP número 30

La red neuronal con la que vamos a trabajar necesita que las dimensiones de su entrada sean [1,3,37,16,32]. Es decir, 3 canales, en los que, para cada estimación, en el primero irán los últimos 37 mapas de resolución 16x32 y en los otros dos irán los

índices de los máximos de ambas dimensiones obtenidos con anterioridad. Una vez obtenidos todos los parámetros de entrada necesarios para ejecutar la red, procedemos a ejecutarla y visualizar los resultados obtenidos y compararlos con la posición en la que se había simulado la fuente. En la figura 5.4 observamos los resultados que se habían obtenido con el modelo en Pytorch, mientras que en la figura 5.5, observamos la comparativa de los resultados obtenidos en Pytorch y openVINO.

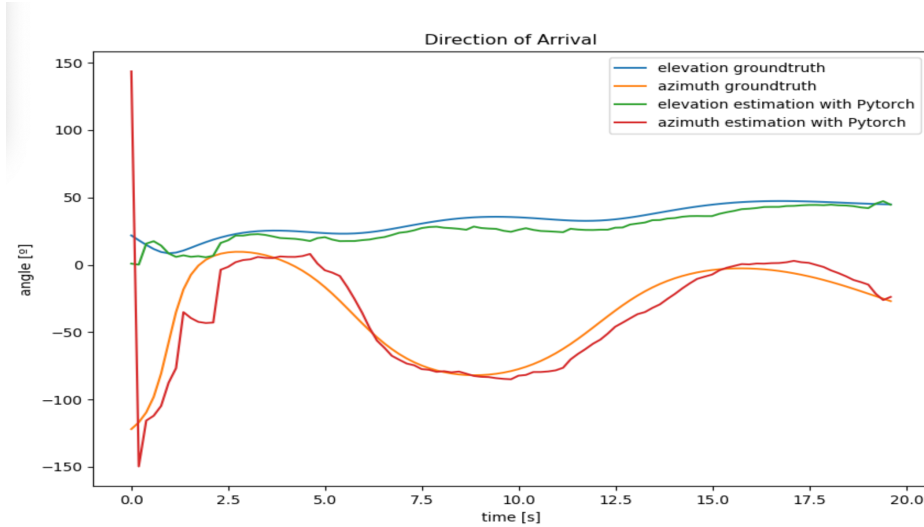


Figura 5.4: Resultado simulación

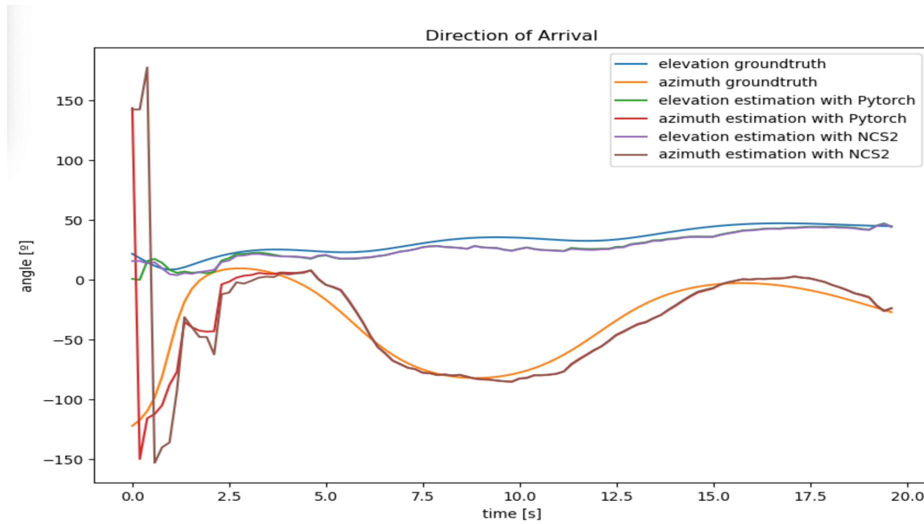


Figura 5.5: Resultado modelo Pytorch

Como podemos observar comparando las dos imágenes anteriores, los resultados obtenidos por el modelo se ajustan a los obtenidos en Pytorch (salvo en los primeros mapas debido al uso de un *padding* distinto en Pytorch), y ambos suponen una buena estimación de la posición en al que la fuente se había simulado. Para verificar si dicha estimación es tan exacta como se observa en la figura 5.5, hemos calculado el error cuadrático medio tanto para el ángulo de elevación como para el ángulo azimut a partir de la ventana 37, en la cual se llena por completo el buffer de entrada. Los valores que

hemos obtenido han sido de 6.45 grados de rmse para el ángulo de elevación y de 8.23 grados de rmse para el ángulo azimut. Una vez observado que el error cometido para cada uno de los ángulos es bastante bajo e igual al del modelo original de Pytorch (cuya resolución ya ha sido ampliamente analizada en [5]), podemos concluir el buen funcionamiento del modelo.

Otro aspecto que debemos analizar es el tiempo de computo total, ya que este debe estar dentro de unos varemos determinados si luego queremos extrapolarlo a que su ejecución se realice en tiempo real. Para ello, realizamos un estudio del tiempo de ejecución, tanto cuando es la CPU quien realiza la ejecución del modelo como cuando es el VPU (Neural Compute Stick 2) quien se encarga de ello. Para que dicho análisis fuese satisfactorio, el tiempo de computo debe ser menor de 20 segundos, que es la duración de la simulación, es decir, debemos ser capaces de realizar el procesado antes de obtener la siguiente ventana a analizar para así no perder información. Como los resultados fueron de 7.65 segundos para CPU y 10.7 segundos para VPU, siendo ambos menores de 20 segundos, concluimos el análisis satisfactoriamente.

### 5.1.2. Instalación y verificación de OpenVINO en Raspberry Pi 3

Una vez instalado y verificado el set de herramientas de OpenVINO en Ubuntu, procedimos con su instalación en una Raspberry Pi 3. Dicho dispositivo es el hardware en el que se va a desarrollar el proyecto. Debido a que era la primera vez que trabajábamos con la VPU (NCS2), decidimos comprobar los resultados en otros sistemas operativos con mayor potencia, con el fin de decidir si era posible utilizarlo de la manera en la que a nosotros nos interesaba. Una vez comprobado, procedimos a observar su funcionamiento en la Raspberry Pi 3.

Para llevar a cabo la instalación del set de herramientas de OpenVINO (OpenVINO Toolkit), seguimos el método de instalación que nos ofrece su distribuidor Intel. Dicho método, el cual queda explicado en el anexo A, se basa fundamentalmente en instalar las variables de entorno necesarias para trabajar con OpenVINO. Después, es necesario instalar los drivers que nos permiten trabajar con la VPU (NCS2), así como las variables de entorno necesarias para poder trabajar con los distintos tipos de modelos de redes neuronales, ya sean ".onnx", ".caffemodel", etc. Por último se tendría que compilar un modelo de ejemplo y ejecutarlo para verificar que toda la instalación ha sido satisfactoria, pero en nuestro caso decidimos realizar dicha verificación probando el mismo *script* que habíamos utilizado en Ubuntu.

Una vez ejecutado el *script* obtenemos resultados similares, siendo mostrados en las figuras 5.6 y 5.7 ejemplos de mapas obtenidos. La principal diferencia, que desde un principio sabíamos de su existencia, es el tiempo de procesado. Al ejecutar el *script* en la Raspberry Pi usando el NCS2, el tiempo de procesado que obtenemos es de 19 segundos, tiempo que podría entrar dentro de los limites que nos permiten extrapolar el modelo a tiempo real, aunque de forma muy ajustada. Cabe destacar que openVINO no permite realizar inferencias en CPUs con arquitectura ARM, como la usada por la Raspberry Pi.

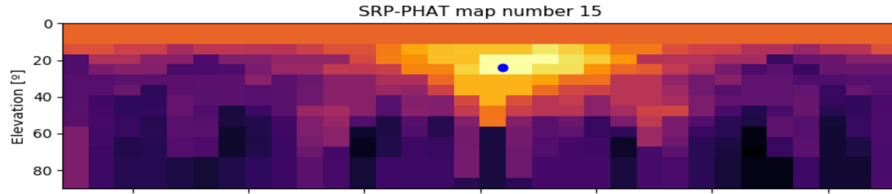


Figura 5.6: Mapa SRP número 15

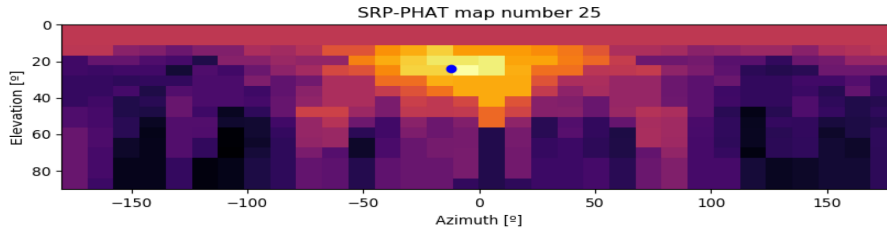


Figura 5.7: Mapa SRP número 25

Una vez obtenidos los resultados, la conclusión que obtuvimos es que el Neural Compute Stick 2 posee la potencia necesaria para ejecutar nuestro modelo de red neuronal, de manera que el tiempo de procesado nos permita extrapolar dicho modelo a tiempo real.

En la tabla 5.1 se resumen los tiempos medidos con el script de Python para distintas librerías y equipos y dispositivos. Cabe destacar las grandes diferencias existentes entre el ordenador en el que se realizaron las pruebas de Pytorch (ordenador A) que contaba con un microprocesador Intel Core i7-7740X de 4.30GHz de frecuencia, 16GB de RAM y una tarjeta gráfica Nvidia GeForce GTX 1060 y el utilizado para las pruebas de OpenVINO (ordenador B) que era contaba con un microprocesador Intel Core i5 de doble nucleo de 2.60GHz de frecuencia, 4GB de RAM de 1600Mhz de frecuencia y una tarjeta gráfica Intel Iris 1536MB, por lo que no es posible extraer conclusiones de dicha comparativa.

| Librería | Ordenador | Dispositivo | tiempo (s) |
|---|---|---|---|
| Pytorch | A | CPU | 2.01 |
| Pytorch | A | GPU | 0.38 |
| OpenVINO | B | CPU | 7.65 |
| OpenVINO | B | NCS2 | 10.7 |
| OpenVINO | Raspi 3 | NCS2 | 19 |

Tabla 5.1: Tiempos necesarios para realizar 103 inferencias del modelo (equivalentes a 20 segundos de audio) mediante el *script* de Python.

## 5.2. Desarrollo del código tiempo real

### 5.2.1. Fase de desarrollo en Ubuntu

El siguiente objetivo del proyecto era ser capaces de implementar una aplicación que usara el modelo de red neuronal con el que estábamos trabajando para realizar la

estimación de la dirección de llegada de una fuente sonora en tiempo real. A la hora de decantarnos por un lenguaje de programación, nos decantamos por trabajar con C++ en vez de con Python. Esta decisión la tomamos en función de que C++ nos ofrece un procesado más optimizado y de mayor velocidad, que es un factor determinante a la hora de trabajar en tiempo real, ya que de esta manera evitaremos perder información.

El primer paso que tuvimos que llevar a cabo fue la instalación de OpenFrameworks (OF), [8]. OF es un conjunto de herramientas de código libre que esta orientado a la programación creativa. Nos permite crear un programa y realizar una representación gráfica del resultado, consiguiendo de esta manera un resultado más visual y atractivo para el usuario. OF nos proporciona diversos códigos de ejemplo, que nos han resultado muy útiles a la hora de entender el funcionamiento de cómo obtener las señales captadas por el array de micrófonos. Dicho ejemplo consistía en captar la señal de audio hasta rellenar un buffer. Una vez rellenado completamente dicho buffer, se procedía a dividir la señal obtenida en dos canales, izquierdo y derecho, realizando un desentrelazado de la señal. Partiendo de este punto, como nuestro array de micrófonos, "miniDSP UMA-8" cuya imagen se presenta en la figura 5.8, contiene un total de 8 canales de los cuales a nosotros nos interesan 6, siendo estos los que corresponden a los micrófonos colocados en los bordes de la PCB, el desentralazado consistirá en que los datos almacenados en el buffer corresponderán con las señales captadas por los micrófonos, es decir, buffer[0] se corresponderá con la señal captada por el micrófono 1, buffer[1] con la señal captada por el micrófono 1 y así respectivamente con cada uno de ellos.

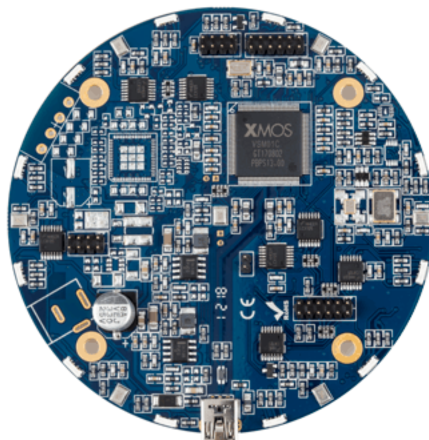


Figura 5.8: Array de micrófonos miniDSP UMA-8

Utilizando este código como base, realizamos una serie de modificaciones para comprobar que el array captaba correctamente las señales. Para ello dividimos las señales de entrada en 7 canales, siendo uno de ellos referente al micrófono central del array y los otros 6 canales referentes a los micrófonos de los bordes, que son los que a nosotros nos interesan. Para dividirlo, seguimos utilizando la misma técnica de desentrelazado, de manera que el primer valor de entrada corresponde al micrófono central, el siguiente al micrófono 1, y así sucesivamente. Una vez realizado dicho desentrelazado y guardado las señales en sus correspondientes canales, era hora de realizar la interfaz gráfica para mostrar los resultados por pantalla. Para

ello, continuamos modificando el ejemplo, de manera que el ejemplo representaba 2 canales con la señal de audio representada mediante ondas sinusoidales, y nosotros lo modificamos para que representase los 6 canales. La representación gráfica de los resultados viene dada en la figura 5.9.

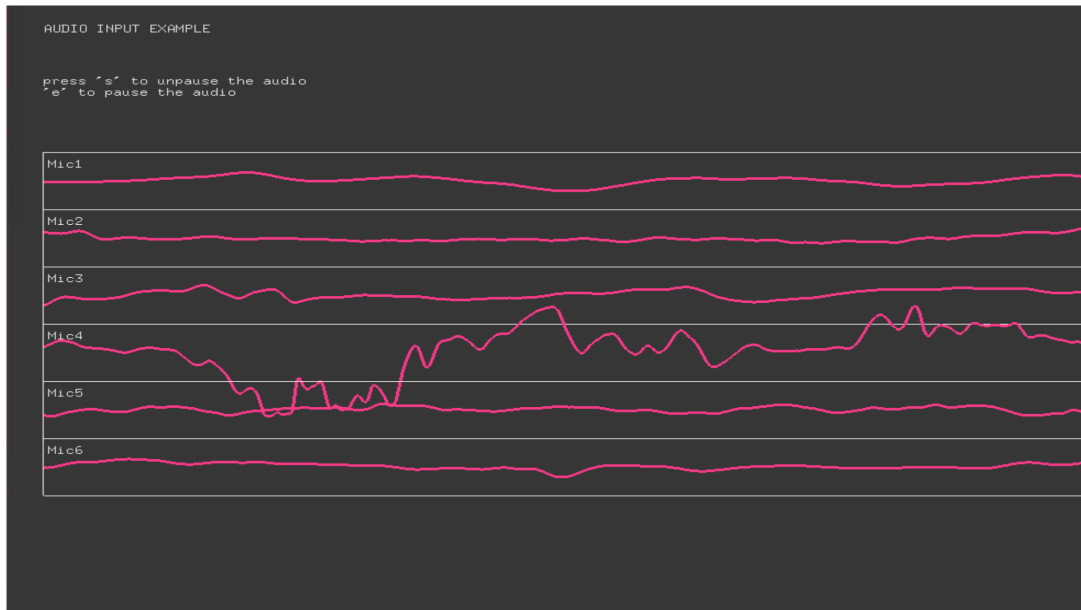


Figura 5.9: Representación canales de audio

Una vez comprobado el correcto funcionamiento del array de micrófonos, reciclamos dicho código para poder utilizarlo como entrada para obtener los mapas SRP. Para calcular dichos mapas, empleamos la librería presentada en [9]. En dicho código, se calculan los mapas SRP, utilizando para ello el algoritmo SRP-PATH, ya comentado en el capítulo 2, utilizando como entrada las señales obtenidas por cada uno de los canales, que mediante el codigo anterior hemos sido capaces de separarlas en 6 canales diferentes. Para lograr una implementación más eficiente, la librería hace uso de la librería FFTW [10] para el calculo de las GGCs.

El código de calculo de los mapas SRP nos devuelve el mapa correspondiente a una ventana de 4096 muestras y lo representa mediante y mapa de colores sobre los ejes de elevación y azimut, siendo el color blanco el que indica la posición del máximo de potencia en el mapa. A su vez, devuelve una matriz de 16x32, con cada uno de los valores obtenidos. En la figura 5.10 se muestra un ejemplo de un mapa obtenido en el que se puede observar el máximo representado por los puntos de color blanco.

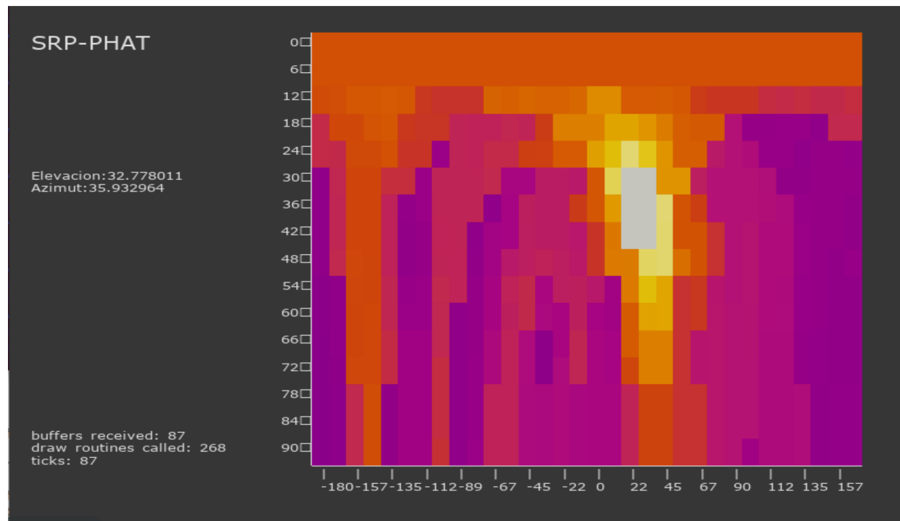


Figura 5.10: Ejemplo de mapas obtenidos

Una vez obtenidos los mapas, tenemos que darle forma a la entrada que necesita la red neuronal. Para ello, la red necesita que su entrada posea unas dimensiones de [1,3,37,16,32]. La segunda dimensión, se rellena en el primer canal por los mapas, y los otros dos canales restantes están compuestos por el máximo de elevación normalizado y por el máximo del azimut normalizado. La siguiente dimensión corresponde al tamaño de ventana de los mapas con la que vamos a trabajar, es decir, la red neuronal trabaja con los últimos 37 mapas obtenidos. Y las últimas dos dimensiones corresponden al tamaño de los mapas.

Para llevar esto a cabo, debemos crear un tensor de 5 dimensiones inicializadas a 0, de manera que cada vez que se calculas un mapa, se realizase una traslación para que cada nuevo mapa se guardase en la última posición del tensor. De este modo, la red neuronal cada vez tendrá una mayor cantidad de información a la entrada, de manera que podría realizar cada vez una aproximación mas robusta y una vez que tuviese los 37 mapas de información, en vez de los valores nulos con los que se ha inicializado al principio, será capaz de observar los desplazamientos que realice la fuente sonora en tiempo real.

Una vez configurada la entrada de la red, nos vamos a centrar en la salida de la misma. La red nos proporciona la localización de de la fuente sonora en coordenadas cartesianas. Debido a que los mapas los estamos calculando en función de los ángulos de elevación y azimut, deberemos transformar estas coordenadas cartesianas en coordenadas esféricas, para así poder realizar una comparación cómo representar ambas en los mismos ejes. En la figura 5.11, podemos observar el resultado de cada una de las iteraciones de la red neuronal, en la que nos devuelve como resultado las coordenadas xyz de la posición de la fuente sonora.

```
Las coordenadas son x=0.247437,y=-0.309082,z=0.431396
El angulo de elevacion es th= 42.544933
El angulo de azimut es phi= -51.320881
Las coordenadas son x=0.201660,y=-0.241211,z=0.480225
El angulo de elevacion es th= 33.212799
El angulo de azimut es phi= -50.103283
Las coordenadas son x=0.121765,y=-0.199097,z=0.504395
El angulo de elevacion es th= 24.829668
El angulo de azimut es phi= -58.550549
```

Figura 5.11: Ejemplo salida por el terminal de la red neuronal

Una vez obtenido este resultado, hemos de realizar el cambio a coordenadas esféricas:

$$x = \rho * sin(\theta)cos(\varphi) \tag{5.1}$$

$$y = \rho * sin(\theta)sin(\varphi) \tag{5.2}$$

$$z = \rho * cos(\theta) \tag{5.3}$$

donde,

$$\rho = \sqrt{(x^2 + y^2 + z^2)} \tag{5.4}$$

$$\theta = \arccos(z/\rho) \tag{5.5}$$

$$\varphi = \arctan(y/x) \tag{5.6}$$

Por lo que utilizando estas ecuaciones, a nosotros nos interesará obtener los valores de $\theta$, que se corresponderá con el ángulo de elevación que se moverán entre valores de [0,90] grados, donde el valor 0 corresponderá con el punto más alto de elevación y el valor 90 con el plano en el que esta situado el array de micrófonos en el espacio, y $\varphi$ que por el contrario, se corresponderá con el ángulo de azimut, teniendo valores de entre [-180,180] grados, en el que el micrófono 1 se corresponderá con el valor 0 del intervalo, el punto medio, y conforme nos estemos moviendo conforme al sentido de las agujas del reloj, es decir, en dirección del micrófono 2 del array, estaremos obteniendo ángulos negativos. En la figura 5.12, se representa el eje de coordenadas que hemos tomado en función de las posiciones de los micrófonos del array. Los mapas nos ofrecen un resultado aproximado del lugar que ocupa la fuente en el espacio, pero en cambio, con la red neuronal, conseguimos mejorar la resolución del mismo, y obtener un punto mas preciso y fiable de la localización real de la fuente sonora en el espacio.

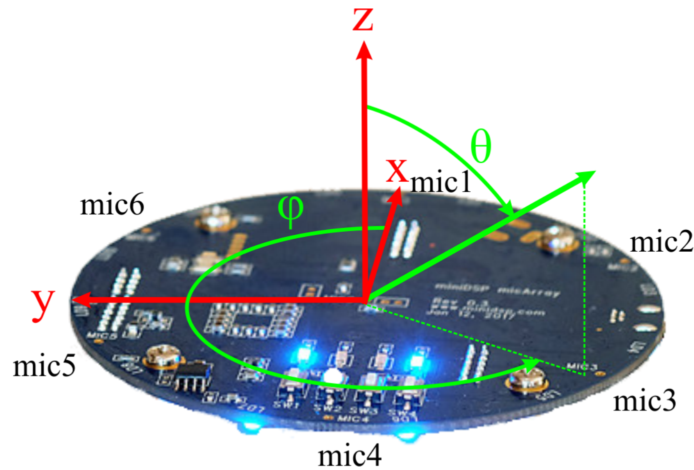


Figura 5.12: Eje de coordenadas en función del array

Una vez realizado el cambio de coordenadas de cartesianas a polares esféricas, y obtenidos los ángulos de elevación y azimut, representaremos el resultado como un punto sobre los mapas SRP obtenidos con anterioridad, de manera que la comparación sea lo más visual posible, y a su vez se haga de una manera rápida. Puede darse el caso de que en el mapa se muestre el máximo sobre una zona y la salida de la red nos marque el punto en el que se encuentre la fuente fuera de esa zona pero muy próxima a ella. En este tipo de casos, sera la salida de la red la que consideremos como el resultado correcto. Esto es debido a que, como ya hemos mencionado con anterioridad, la red neuronal nos ofrece una mayor resolución, por lo tanto podemos suponer que dicha posición es una aproximación con mayor precisión. Existe una excepción, ya que cuando posicionamos una fuente sonora encima del array de micrófonos con elevación 0 grados, la red neuronal nos devolverá una coordenada de azimut "aleatoria", es decir, como la la fuente sonora esta a la misma distancia de cada uno de los micrófonos pertenecientes al array el ángulo de azimut no se puede obtener ya que no tiene un valor definido. El resultado obtenido es el mostrado en la figura 5.13, donde superponemos el resultado de la red a los mapas, y de esta manera podemos observar como el resultado dado por la red, representado por un punto verde, mejora la precisión dada por los mapas.

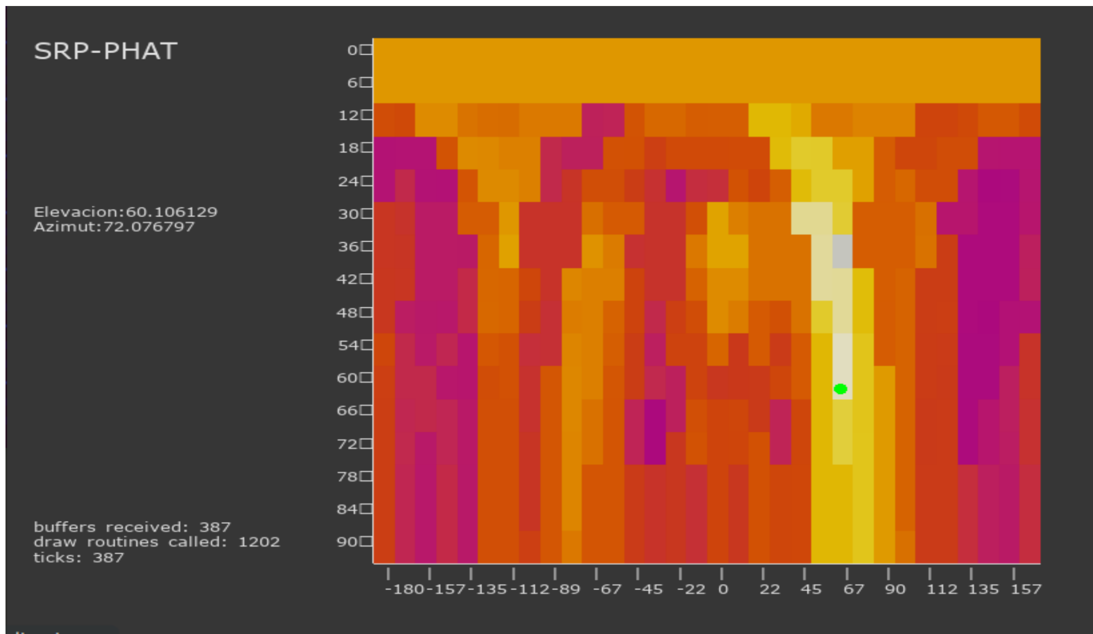


Figura 5.13: Ejemplo salida de la red neuronal

En cambio, en la figura 5.14, vemos un ejemplo de cuando la fuente se encuentra sobre el array y a la misma distancia de cada uno de los micrófonos que lo forman. En el mapa queda representado que para cualquier valor del ángulo de azimut, en el mapa se representa todo el rango como solución, puesto que solo interesa el ángulo de elevación. Para dicho caso, la solución que estima la red quedara representada por grado de elevación 0 y azimut aleatorio como podemos ver en el ejemplo de la figura. En cualquier caso, esto no supone un problema, ya que cuando la elevación tiende a 0, todos los puntos se encuentran muy próximos entre sí independientemente de su valor de azimut.

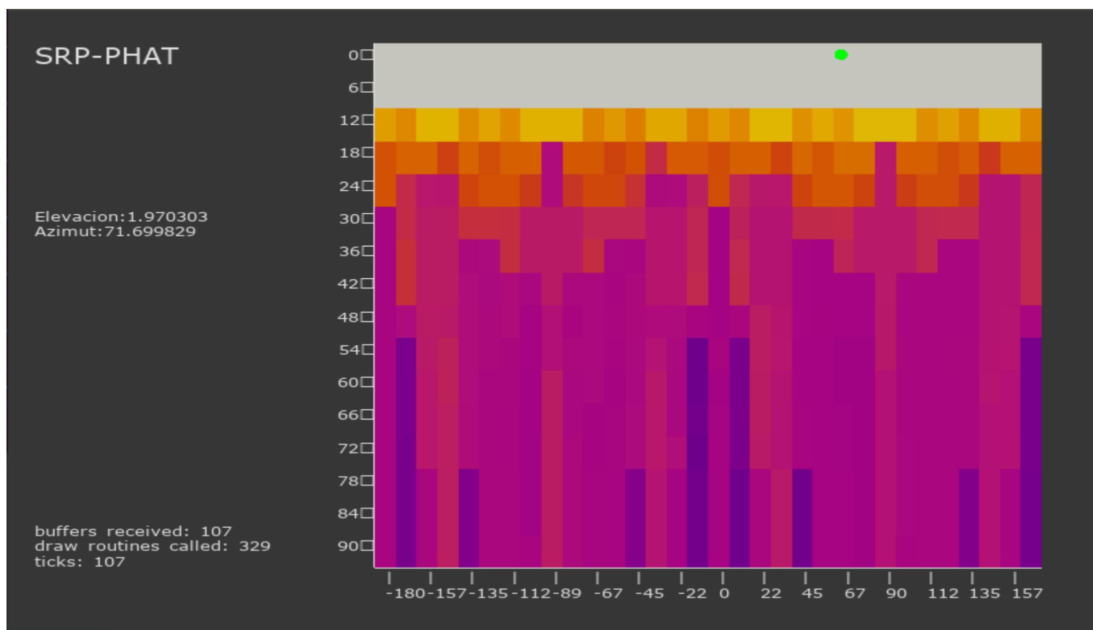


Figura 5.14: Ejemplo caso Elevación = 0º

A continuación se muestra un diagrama de bloques del código completo:

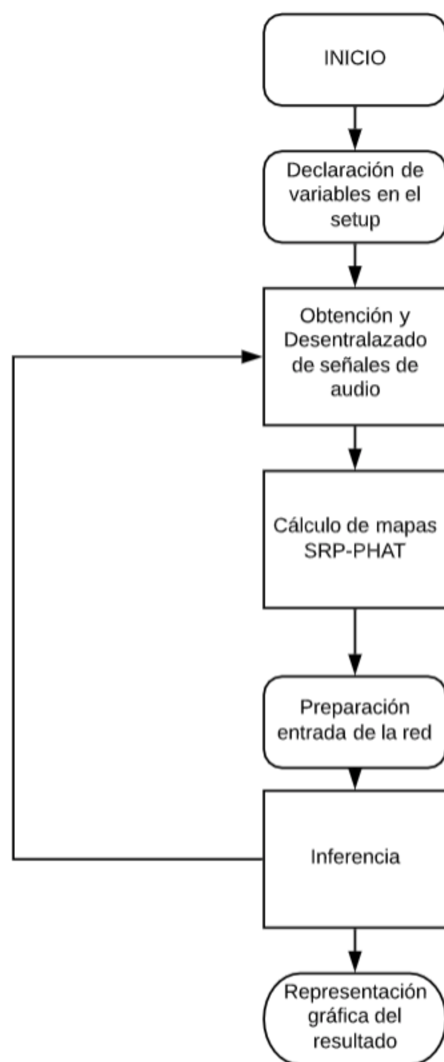


Figura 5.15: Diagrama de bloques del código C++

Una vez obtenidos los resultados deseados, pasamos a la siguiente fase del proyecto.

### 5.2.2. Fase de desarrollo en Raspberry Pi 3

Una vez que teníamos un programa funcional en una maquina virtual Ubuntu, el siguiente paso consistía en trasladarlo a una Raspberry Pi 3, siendo este el objetivo principal del proyecto.

En esta fase nos encontramos con diversos problemas a los que enfrentarnos, que nos han impedido conseguir el resultado final esperado. El primero fue la modificación de un Addon creado específicamente para poder compilar modelos de OpenVINO en Openframework. Para ello, debimos modificar las rutas a las que tenia que acceder para obtener las librerías que hacían posibles la compilación de los proyectos. Otro de los problemas que se nos ha presentado tiene que ver con la

configuración de la Raspberry, siendo este que uno de los driver que nos permitía utilizar ventanas de interfaz gráficas estaba deshabilitado, y por tanto nos daba error a la hora de ejecutar el proyecto. Una vez que lo habilitamos dicho problema fue resuelto y pudimos generar interfaces gráficas.

Una vez que habíamos solventado los problemas anteriores nos dispusimos a intentar ejecutar nuestro modelo, sin embargo nos dio fallo debido a las librerías de OpenVINO. Este fallo en concreto conseguimos solventarlo mediante la actualización de la versión de OpenVINO, en este caso la versión 2020.3. A pesar de actualizar, seguía existiendo un bug con las librearías de OpenVINO que impedían realizar más de una inferencia. Con el fin de solucionar este problema, nos planteamos inicialmente si el problema radicaba en la velocidad de procesamiento del NCS2, que era inferior a la necesaria para realizar las 4 inferencias por segundo que estábamos realizando en la maquina virtual. De esta manera, con el fin de encontrar el problema, realizábamos una inferencia cada 10 segundos, de manera que nos permitiría observar si realmente la velocidad de procesado era el problema. Para nuestra sorpresa, nos encontrábamos con que el calculo de los mapas de potencia eran correctos, pero cuando se le pedía realizar la inferencia, solo la realizaba la primera vez.

Para intentar solucionar este problema, se hicieron una serie de pruebas con ejemplos propios de OpenVINO, más en concreto con el ejemplo “hello_classification”, que se basa en la identificación de imágenes. Para realizar dichas pruebas lo que hicimos fue variar el número de inferencias que queríamos que hiciese la red, así como el modo de hacerlas. Esto quiere decir que queríamos observar si el realizar la inferencia como parte principal del código o tratarla como una función a parte variaría el resultado. Lamentablemente los resultados a los que llegamos no fueron concluyentes, ya que el resultado que ofrecían los diversos intentos eran aleatorios. En ciertas ocasiones los resultados daban esperanzas de encontrar la solución dando un resultado valido, y en cambio si volvíamos a ejecutarlo con los mismos parámetros el resultado eran errores de segmentación o incluso abortaba la ejecución sin resaltar ningún error aparente.

Tras todas estas pruebas, la conclusión a la que hemos llegado es que existe algún tipo de bug en las librerías de OpenVINO para Raspberry Pi, y hemos informado a Intel del mismo.

# Capítulo 6

# Conclusiones

Durante este proyecto, se han ido adquiriendo diversos conocimientos básicos sobre las técnicas estimación del ángulo de llegada de las señales en agrupaciones de micrófonos, necesarios para el entendimiento de la aplicación que nos ofrecía los mapas de potencia basados en el algoritmo SRP-PHAT, el cual iba a ser nuestro punto de partida. Como hemos visto en el capitulo 2, dicho algoritmo es el que mejores compromisos presenta entre prestaciones y coste computacional.

Del mismo modo, también hemos adquirido los conocimientos necesarios para poder trabajar con el set de herramientas de OpenVINO (OpenVINO Toolkit). Gracias a esto, hemos podido conocer la cantidad de recursos que nos ofrece a la hora de realizar la inferencia de redes neuronales. La VPU de Intel, Neural Compute Stick, nos ha permitido poder trabajar reduciendo los costes computacionales.

Por último, cabe destacar la diferencia de precisión que existe entre el resultado que nos ofrece la red, con la salida de los mapas. Esta diferencia es considerable, ya que los mapas realizaban una aproximación a la zona en la que podría estar la fuente dependiendo de la dirección de llegada de la señal. Sin embargo, la red nos ofrece un resultado más preciso, que a su vez es complementario a la ofrecida por los mapas de potencia. Estos últimos nos dan una estimación aproximada de la posible zona en la que hay un máximo de potencia, y la red nos termina de aumentar la información sobre la localización exacta de la fuente sonora.

Si bien finalmente no ha sido posible tener la aplicación realizada corriendo en la Raspberry Pi, todo apunta a que esto se debe a un bug de las librerías de OpenVINO, y la aplicación desarrollada debería correr sin problemas en la Raspberry Pi cuando éste sea corregido en próximas versiones.

# Capítulo 7

# Bibliografía

[1] Joseph H. DiBiase, Harvey F. Silverman, Michael S. Brandstein, Arild Lacroix, and Anastasios Venetsanopoulos. Robust Localization in Reverberant Rooms. In Michael Brandstein and Darren Ward, editors, *Microphone Arrays*, Digital Signal Processing, pages 157–180. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[2] ¿Qué es el Deep Learning? `https://www.smartpanel.com/que-es-deep-learning/`, April 2018.

[3] METODOLOGÍA RED NEURONAL. `https://sitiointeligenciaa.wordpress.com/metodologia/`, June 2017.

[4] Diego Calvo. Red Neuronal Convolucional CNN. `https://www.diegocalvo.es/red-neuronal-convolucional/`, July 2017.

[5] David Diaz-Guerra, Antonio Miguel, and Jose R. Beltran. Robust Sound Source Tracking Using SRP-PHAT and 3D Convolutional Neural Networks. June 2020.

[6] Barra de cómputo neuronal Intel® 2. `https://www.intel.com/content/www/xl/es/develop/hardware/neural-compute-stick.html`.

[7] Enhanced Visual Intelligence at the Network Edge.

[8] Linux install — openFrameworks. `https://openframeworks.cc/setup/linux-install/`.

[9] David . *Localización de Fuentes Sonoras Mediante Agrupaciones de Micrófonos.* Trabajo fin de master, Universidad de Zaragoza.

[10] M. Frigo and S.G. Johnson. The Design and Implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, February 2005.

[11] Install Intel® Distribution of OpenVINO™ toolkit for Linux* - OpenVINO™ Toolkit. `https://docs.openvinotoolkit.org/latest/_docs_install_guides_installing_openvino_linux.html`.

[12] Install OpenVINO™ toolkit for Raspbian* OS - OpenVINO™ Toolkit. `https://docs.openvinotoolkit.org/latest/_docs_install_guides_installing_openvino_raspbian.html`.

# Appendices

# Apéndice A

# Anexo 1. Instalación OpenVINO Toolkit

## A.1. Instalación OpenVINO Toolkit en Ubuntu

Método de Instalación de OpenVINO Toolkit,[11]:
- Descargamos la versión de OpenVINO que queremos instalar:

- 1.- cd /Downloads/
- 2.- tar -xvzf l_openvino_toolkit_p_<version>.tgz
- 3.- cd l_openvino_toolkit_p_<version>

Una vez descargado y en el directorio de OpenVINO, lo instalamos en el sistema operativo:

- 4.- sudo ./install_GUI.sh

Y seguimos las indicaciones de la pantalla, tal y como se muestra en la imagen A.1:

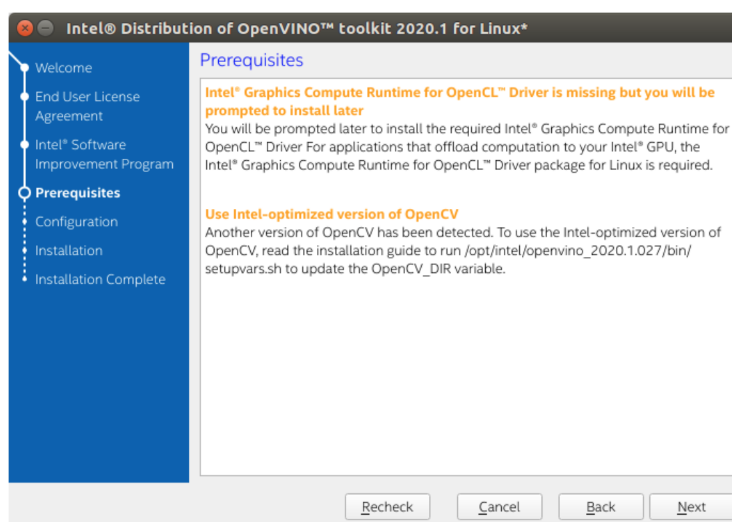


Figura A.1: Pantalla de instalación de OpenVINO 1

Si seleccionaste la opción por defecto, la pantalla muestra la siguiente imagen A.2

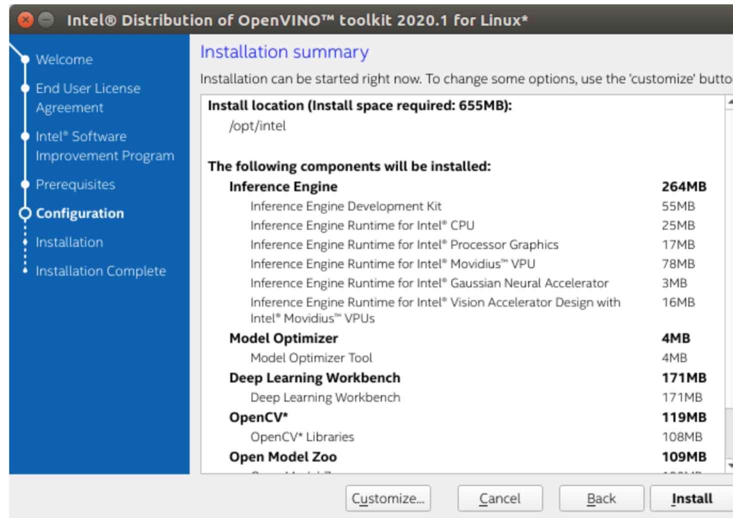


Figura A.2: Pantalla de instalación de OpenVINO 2

Aunque se puede puede personalizar instalando solo los paquetes que se deseen, tal y como muestra la figura A.3.

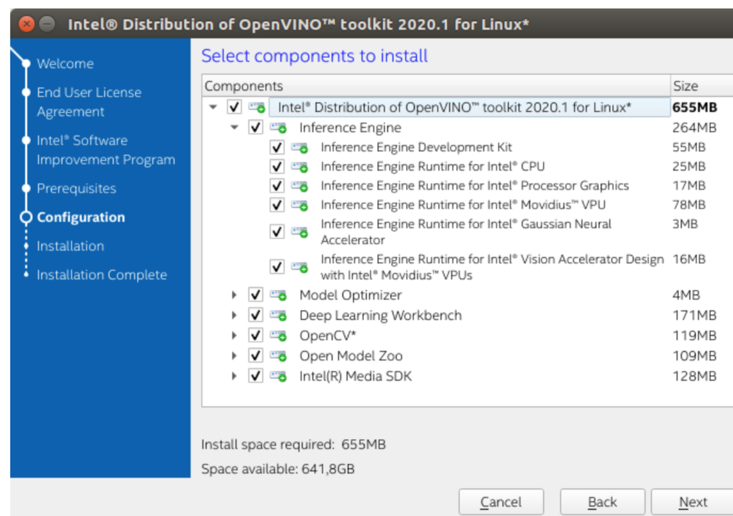


Figura A.3: Pantalla de instalación de OpenVINO 3

Una vez la instalación este completa, el instalador nos devolverá la imagen de la figura A.4.

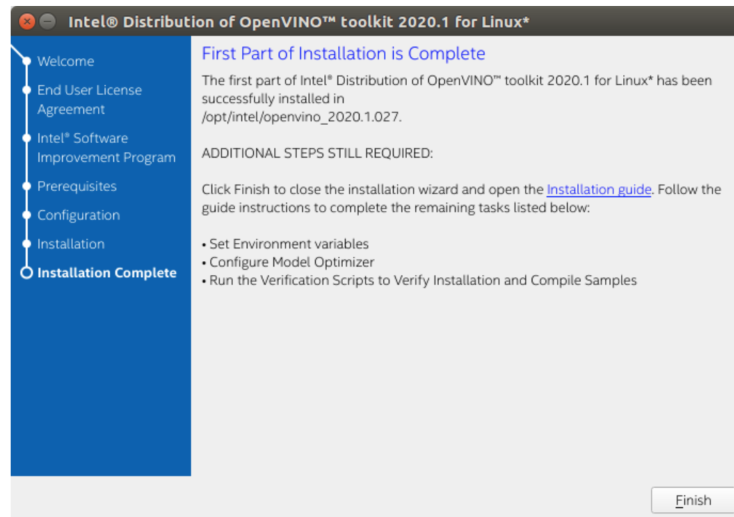


Figura A.4: Pantalla de instalación de OpenVINO 4

Una vez realizada la instalación, es hora de instalar las dependencias de software externas, para ello seguiremos el siguiente procedimiento:

- 5.- cd /opt/intel/openvino/install_dependencies
- 6.- sudo -E ./install_openvino_dependencies.sh

El siguiente paso es instalar las variables de entorno que nos permitirán realizar las inferencias:

- 7.- source /opt/intel/openvino/bin/setupvars.sh

Configuración del "Model Optimizer":

- 8.- cd /opt/intel/openvino/deployment_tools/model_optimizer/install_prerequisites
- 9.- sudo ./install_prerequisites.sh

Realizando la instalación de esta manera, configuras todos los frameworks de manera paralela.
Por último, debemos realizar la configuración del Nerural Compute Stick 2:

- 10.- sudo usermod -a -G users "$(whoami)"
- 11.- sudo cp /opt/intel/openvino/inference_engine/external/97-myriad-usbboot.rules /etc/udev/rules.d/
- 12.-sudo udevadm control –reload-rules
- 13.- sudo udevadm trigger
- 14.- sudo ldconfig

Con esto, daríamos por concluida la instalación de OpenVINO Toolkit en Ubuntu.

## A.2. Instalación OpenVINO Toolkit en Raspberry Pi 3

A continuación se muestran los pasos necesarios para realizar una buena instalación de OpenVINO Toolkit en Raspberry Pi 3, [12]:
- Descargamos la versión de OpenVINO que queremos instalar:

- 1.- cd /Downloads/
- 2.- sudo mkdir -p /opt/intel/openvino
- 3.- sudo tar -xf l_openvino_toolkit_runtime_raspbian_p_<version>.tgz –strip 1 -C /opt/intel/openvino

Una vez descargado y descomprimido la versión de que queremos de OpenVINO, prodedemos con la instalación.
En un primer lugar instalamos las dependencias de software externas.

- 4.- sudo apt install cmake

Una vez instaladas, pasamos al siguiente punto, que consistirá en la instalación de las variables de entorno.

- 5.- source /opt/intel/openvino/bin/setupvars.sh

A continuación instalaremos los drivers necesarios para el cprrecto funcionamiento del Neural Compute Stick 2.

- 6.- sudo usermod -a -G users "$(whoami)"
- 7.- source /opt/intel/openvino/bin/setupvars.sh
- 8.- ssh /opt/intel/openvino/install_dependencies/install_NCS_udev_rules.sh

Por último, para verificar que todo el proceso es correcto, se crea y compila un ejemplo.

- 9.- mkdir build && cd build
- 10.- cmake -DCMAKE_BUILD_TYPE=Release -DCMAKE_CXX_FLAGS=march=armv7-a/ /opt/intel/openvino/deployment_tools/inference_engine/samples
- 11.- make -j2 object_detection_sample_ssd
- 12.- wget –no-check-certificate https://download.01.org/opencv/2020/openvinotoolkit/ /2020.1/open_model_zoo/models_bin/1/face-detection-adas-0001/ /FP16/face-detection-adas-0001.bin
- 13 wget –no-check-certificate https://download.01.org/opencv/2020/ /openvinotoolkit/2020.1/open_model_zoo/models_bin/1/ /face-detection-adas-0001/FP16/face-detection-adas-0001.xml
- 14.- ./armv7l/Release/object_detection_sample_ssd -m face-detection-adas-0001.xml -d MYRIAD -i <path_to_image>

# Apéndice B

# Anexo 2. Código de Python empleado

```
# %% Importar las librerías y abrir el archivo con las variables
import numpy as np
import matplotlib.pyplot as plt
import sys
import argparse
import time
loaded_variables = np.load('network_test_variables(1).npz', allow_pickle=True)

from openvino.inference_engine import IENetwork, IECore
# %% Señales originales de los micrófonos muestreadas a 16kHz (en realidad son de
una simulación, no una grabación real)
mic_signals = loaded_variables['mic_signals']
t = loaded_variables['t']
print('mic_signals shape: ' + str(mic_signals.shape)) # (20s * 16kHz, 12
micrófonos)
plt.plot(t, mic_signals)
plt.title('mic_signals')
plt.xlabel('time [s]')
plt.show()

# %% Señales de los micrófonos enventanadas. Quedan 103 ventanas de 4096 muestras
con 75% de overlap
mic_signals_w = loaded_variables['mic_signals_w']
print('mic_signals_w shape: ' + str(mic_signals_w.shape)) # (103 ventanas, 12
micrófonos, 4096 muestras)

# %% Ángulo de llegada en cada ventana, tanto el original de la simulación como la
estimación con Pytorch
DoA_groundtruth = loaded_variables['DoA_groundtruth']
DoA_est_reference = loaded_variables['DoA_est_reference']
tw = loaded_variables['tw']
print('DoA_groundtruth shape: ' + str(DoA_groundtruth.shape)) # (103 ventnas, 2
coordenadas angulares)
print('DoA_est_reference shape: ' + str(DoA_est_reference.shape)) # (103 ventnas,
2 coordenadas angulares)
plt.plot(tw, DoA_groundtruth*180.0/np.pi)
plt.plot(tw, DoA_est_reference*180.0/np.pi)
plt.title('Direction of Arrival')
plt.xlabel('time [s]')
plt.ylabel('angle [º]')
plt.legend(('elevation groundtruth', 'azimuth groundtruth', 'elevation estimation
with Pytorch', 'azimuth estimation with Pytorch'))
plt.show()

# %% Mapa SRP-PHAT de cada ventana, con resolución 16x32
srp_maps = loaded_variables['srp_maps']
print('srp_maps shape: ' + str(srp_maps.shape)) # (103 ventanas, 16 puntos de
elevación, 32 puntos de azimut)
frame_idx = 15
theta = np.linspace(0, 90, srp_maps.shape[1])  # En realidad los ángulos no son
éstos exactamente, pero lo dejo así para no complicar el código
phi = np.linspace(-180, 180, srp_maps.shape[2]) # En realidad los ángulos no son
éstos exactamente, pero lo dejo así para no complicar el código
plt.imshow(srp_maps[frame_idx,...], cmap='inferno', extent=(phi[0], phi[-1],
theta[-1], theta[0]))
plt.scatter(DoA_groundtruth[frame_idx, 1]*180/np.pi, DoA_groundtruth[frame_idx,
0]*180/np.pi, c='b')
plt.title("SRP-PHAT map number " + str(frame_idx))
plt.xlabel('Azimuth [º]')
plt.ylabel('Elevation [º]')
plt.show()

# %% La red, aparte de los mapas, usa como entrada la posición de los máximos
normalizada de forma que 0 es el valor mínimo y 1 el máximo.
# Se le pasan añadiendo dos canales más a los mapas (como si fueran imagenes RGB)
```

```
pero en realidad para cada mapa todos los pixeles están
# al mismo valor (la elevación y el azimut en el que se encuentra el máximo)
theta_maximums = np.zeros(srp_maps.shape)
phi_maximums = np.zeros(srp_maps.shape)
for frame_idx in range(srp_maps.shape[0]): # Podría intentar hacerse sin bucles,
pero lo dejo así para no complicar el código
        maximum = np.unravel_index(np.argmax(srp_maps[frame_idx,...]),
srp_maps[frame_idx,...].shape)
        theta_maximums[frame_idx,...] = maximum[0] / srp_maps.shape[1]
        phi_maximums[frame_idx,...] = maximum[1] / srp_maps.shape[2]
network_inputs = np.stack((srp_maps, theta_maximums, phi_maximums))
print('network_inputs shape: ' + str(network_inputs.shape)) # (3 canales, 103
ventanas, 16 puntos de elevación, 32 puntos de azimut)

# %% La red coje como entradas los últimos 37 mapas. En una implementación en
tiempo real, cuando se calcula el primero,
# los anteriores están a cero. Por eso añado 36 mapas con todo a cero al principio
network_inputs = np.concatenate((np.zeros((network_inputs.shape[0], 36,
network_inputs.shape[2], network_inputs.shape[3])), network_inputs), axis=1)
print('network_inputs shape: ' + str(network_inputs.shape)) # (3 canales, 139
ventanas, 16 puntos de elevación, 32 puntos de azimut)

# %% Para calcular cada estimación se le van pasando paquetes de 36 mapas, tendrás
que editar esto para meter la llamada
# a la red con el NCS2
ie= IECore()
#ie.set_config({'VPU_HW_STAGES_OPTIMIZATION':'NO'},"MYRIAD")
cur_request_id=0
t= time.time()
network_net = IENetwork(model =
'1sourceTracking_cross2D1D_miniDSP_K4096_16x32_model(2).xml', weights =
'1sourceTracking_cross2D1D_miniDSP_K4096_16x32_model(2).bin')
network_outputs = np.zeros((srp_maps.shape[0], 3))
network_exec_net = ie.load_network(network = network_net, device_name = "CPU")

for frame_idx in range(srp_maps.shape[0]): # Podría intentar hacerse sin bucles,
pero lo dejo así para no complicar el código
        print(str(frame_idx))
        sys.stdout.flush()
        net_input = network_inputs[:, frame_idx:frame_idx+37, ...]
        # net_output =
net(torch.from_numpy(net_input[np.newaxis, ...]).float().cuda()).detach().cpu().numpy().squeeze()
        network_exec_net.start_async(request_id=cur_request_id,
inputs={'input':net_input [np.newaxis,...]}) # Aquí tendrás que meter la llamada a
la red, la linea anterior es lo que tengo que hacer yo para llamar a la red con
Pytorch
        if network_exec_net.requests[cur_request_id].wait(-1) == 0:
                network_outputs[frame_idx, :] =
network_exec_net.requests[cur_request_id].outputs['output']
        else: print("Fallo en el frame id: "+ str(cur_request_id))
        #cur_request_id+=1
print('network_outputs shape: ' + str(network_outputs.shape)) # (103 ventnas, 3
coordenadas cartesianas)
print(str(time.time()-t)+'s elapsed')
# %% La red no te da directamente las coordenadas esféricas de la dirección de
llegada, sino las coordenadas cartersianas
# de un vector unitario apuntando en esa direccióǹ
def cart2sph(x, y, z):
    hxy = np.hypot(x, y)
    el = np.arctan2(hxy, z)
    az = np.arctan2(y, x)
    return np.stack((el, az), axis=-1)
DoA_est_NCS2 = cart2sph(network_outputs[:,0], network_outputs[:,1],
network_outputs[:,2])
print('DoA_est_NCS2 shape: ' + str(DoA_est_NCS2.shape)) # (103 ventnas, 3
coordenadas angulares)
```

```
# %% Cuando calcules la estimación con el NCS2 el resultado debería ser
practicamente igual que el que me ha salido a mi
# con Pytorch, porque la red es la misma
plt.plot(tw, DoA_groundtruth*180.0/np.pi)
plt.plot(tw, DoA_est_reference*180.0/np.pi)
plt.plot(tw, DoA_est_NCS2*180.0/np.pi)
plt.title('Direction of Arrival')
plt.xlabel('time [s]')
plt.ylabel('angle [º]')
plt.legend(('elevation groundtruth', 'azimuth groundtruth', 'elevation estimation
with Pytorch', 'azimuth estimation with Pytorch', 'elevation estimation with
NCS2', 'azimuth estimation with NCS2'))
plt.show()
```

# Apéndice C

# Anexo 3. Código C++ del Proyecto

```
#pragma once

#include "ofMain.h"

#include <vector>
#include <mutex>
#include <math.h>
#include "ofxTrueTypeFontUC.h"
#include <iostream>
#include <iomanip>
#include "headers.h"
#include "srp_phat.h"
#include "sendToMatlab.h"
#include <inference_engine.hpp>
#include <common.hpp>
#include <ocv_common.hpp>
#include <opencv2/core/mat.hpp>

using namespace InferenceEngine;
using namespace std;

class ofApp : public ofBaseApp {

        public:
                bool threshold = false;

                static const int resPhi = RES_PHI;      //Nuemro de puntos en X
                static const int resThe = RES_THE;      //Numero de puntos en Y
                static const int bufferSize = BUFFER_SIZE;
                static const int nChannels = N_SENSORS;
                float potUm = POT_UM;

                void setup();
                void update();
                void draw();

                void keyPressed(int key);
                void keyReleased(int key);
                void mouseMoved(int x, int y );
                void mouseDragged(int x, int y, int button);
                void mousePressed(int x, int y, int button);
                void mouseReleased(int x, int y, int button);
                void mouseEntered(int x, int y);
                void mouseExited(int x, int y);
                void windowResized(int w, int h);
                void dragEvent(ofDragInfo dragInfo);
                void gotMessage(ofMessage msg);

                void audioIn(float * input, int bufferSize, int nChannels);

                ofxTrueTypeFontUC textSmall, textBig;

                float x[nChannels][bufferSize];
                float P[resPhi][resThe];
                float maxP;

                mutex mutexMap;


                int bufferCounter = 0;
                int drawCounter   = 0;
                float The_max=0;
                float Phi_max=0;
                int i_max=0;
                int j_max=0;
                ofSoundStream soundStream;
```

```
        srp_phat srp;

        ofImage imageResult;

        cv::Mat entradaRed;

        float * ptr;
        float * ptr_d;
        float * ptr_o;
        float * ptr_The;
        float * ptr_Phi;
        float r,th,phi;
        float coord_azimut,coord_elevacion;


    private:

        Core ie; // 1. Load inference engine instance

        ExecutableNetwork executable_network;
        std::string network_input_name;
        std::string network_output_name;


};
```

```
#include "ofApp.h"
#include "palette.h"
#include "hanning4096.h"

//--------------------------------------------------------------
void ofApp::setup(){

	ofSetCircleResolution(80);
	ofSetVerticalSync(true);
	ofBackground(54, 54, 54);
	ofSetFrameRate(12);
	int size[5]={1,3,37,16,32};
	entradaRed = cv::Mat::zeros(5, size, CV_32FC1); //CV_16FC1


	textSmall.loadFont("verdana.ttf", 10, true, true);
	textBig.loadFont("verdana.ttf", 16, true, true);

	srp.setup((float*)x, (float*)P);

	imageResult.allocate(resPhi, resThe, OF_IMAGE_COLOR_ALPHA);
	imageResult.getTexture().setTextureMinMagFilter(GL_NEAREST,
GL_NEAREST); // Avoid pixel interpolation
	imageResult.setColor(ofColor::black);


	// 2. Read IR Generated by ModelOptimizer (.xml and .bin files)
	CNNNetwork network =
ie.ReadNetwork(ofFilePath::getAbsolutePath("1sourceTracking_cross2D1D_miniDSP_K4096_16x32_model(2
ofFilePath::getAbsolutePath("1sourceTracking_cross2D1D_miniDSP_K4096_16x32_model(2).bin"));
	network.setBatchSize(1);

	// 3. Configure input & output
	// --- Prepare input blobs
	InputInfo::Ptr input_info = network.getInputsInfo().begin()->second;
	network_input_name = network.getInputsInfo().begin()->first;

	/* Mark input as resizable by setting of a resize algorithm.
	 * In this case we will be able to set an input blob of any shape to an
infer request.
	 * Resize and layout conversions are executed automatically during
inference */


	input_info->setPrecision(Precision::FP32);


	// --- Prepare output blobs

	DataPtr output_info = network.getOutputsInfo().begin()->second;
	network_output_name = network.getOutputsInfo().begin()->first;

	output_info->setPrecision(Precision::FP32);


	// 4. Loading model to the device
	executable_network = ie.LoadNetwork(network, "MYRIAD");

	// 0 output channels,
	// 2 input channels
	// 44100 samples per second
	// 256 samples per buffer
	// 4 num buffers (latency)

	soundStream.printDeviceList();
	cout << "Selec device: ";
```

```
        int device;
        cin >> device;
        soundStream.setDeviceID(device);


        soundStream.setup(this, 0, 8, FS, bufferSize, 16);
}

//--------------------------------------------------------------
void ofApp::update(){
        //Actualiza la imágen con los últimos valores de SRP
        mutexMap.lock();
        for (int x = 0; x < resPhi; x++) {
                for (int y = 0; y < resThe; y++) {
                        int level = maxP>0? (int) P[x][y] / maxP * 512 : 0;
                        if (level < 0) level = 0;
                        if (threshold && maxP<VAD_TH) level = 0;
                        imageResult.setColor(x, y, ofColor(thermal_palette[level]
[0], thermal_palette[level][1], thermal_palette[level][2]));
                }
        }
        mutexMap.unlock();
        imageResult.update();
}

//--------------------------------------------------------------
void ofApp::draw(){

        ofSetColor(225);
        textBig.drawString("SRP-PHAT", 20, 50);
        textSmall.drawString("Elevacion:"+std::to_string(th)
+"\nAzimut:"+std::to_string(phi), 20, 200);
        ofNoFill();

        ofPushStyle();
                ofPushMatrix();
                ofTranslate(275, 30, 0);

                ofSetColor(225);
                ofDrawRectangle(0, 0, 500, 500);

                // Draw x and y labels
                std::string axis;
                for (int i = 0; i < RES_THE; i++) {
                        int the = srp.theta[0][i] * 180.0 / PI;
                        axis = std::to_string(the) + "º_";
                        textSmall.drawString(axis, -textSmall.stringWidth(axis),
(i+.5) * 500 / RES_THE);
                }
                for (int i = 0; i < RES_PHI; i+=2) {
                        int phi = srp.phi[i][0] * 180.0 / PI;
                        axis = "|\n" + std::to_string(phi) + "º";
                        textSmall.drawString(axis, (i+.5) * 500 / RES_PHI, 500 +
textSmall.getLineHeight());
                }

                // Draw the SRP result
                if(maxP>potUm){
                        imageResult.draw(0, 0, 500, 500);
                }
                ofSetColor(0,255,0);
                ofFill();
                ofDrawCircle(phi*(500.0/360.0)+250,th*(500.0/90.0),5);
                ofNoFill();
                ofPopMatrix();
        ofPopStyle();
```

```
        drawCounter++;

        ofSetColor(225);
        string reportString = "buffers received: "+ofToString(bufferCounter)
+"\ndraw routines called: "+ofToString(drawCounter)+"\nticks: " +
ofToString(soundStream.getTickCount());
        textSmall.drawString(reportString, 20, 500);

}

//--------------------------------------------------------------
void ofApp::audioIn(float * input, int bufferSize, int nChannels) {

        // samples are "interleaved"
        int numCounted = 0;

        for (int i = 0; i < bufferSize; i++) {
                for (int j = 0; j < nChannels; j++) {
                        if (j < 6) {
                                x[j][i] = (hanning[i] * input[i*nChannels +
(j+1)])*100; // j+1 porque el primer canal es el micro central
                                numCounted++;
                        }
                }
        }
        mutexMap.lock();
        //Realizamos la traslación de los mapas

        for(int k=0;k<36;k++){
                for(int i=0; i<resThe;i++){
                        for(int j=0; j<resPhi;j++){
                                for(int l=0; l<3;l++){
                                        ptr_d=(float *) (entradaRed.data +
entradaRed.step[0]*0 + entradaRed.step[1]*l+
entradaRed.step[2]*k+entradaRed.step[3]*i+entradaRed.step[4]*j);

                                        ptr_o=(float *) (entradaRed.data +
entradaRed.step[0]*0 + entradaRed.step[1]*l+
entradaRed.step[2]*(k+1)+entradaRed.step[3]*i+entradaRed.step[4]*j);

                                        ptr_d[0]=ptr_o[0];
                                }
                        }
                }
        }
        int imax=0;
        int jmax=0;
        float media=0;
        maxP = srp.execute(imax,jmax,media);
        The_max=(float)jmax/resThe;
        Phi_max=(float)imax/resPhi;
        //printf("maxP= %f\n",maxP);
        //Guardamos los mapas en el tensor de 5 dimensiones de cv::Mat, y
localizamos los máximos de Theta y Phi.
        for(int i=0; i<resThe;i++){
                for(int j=0; j<resPhi;j++){
                        ptr=(float*)( entradaRed.data + entradaRed.step[0]*0 +
entradaRed.step[1]*0+
entradaRed.step[2]*36+entradaRed.step[3]*i+entradaRed.step[4]*j);
                        if(maxP<potUm){
                                ptr[0]=0;
                                The_max=0;
                                Phi_max=0;
                        }
                        else{
```

```
                                ptr[0]=(P[j][i]-media)/(maxP-
media);
                        }
                }
        }

        //Sacamos los maximos de los mapas en las dimensiones {1,1,37,16,32} y
{1,2,37,16,32}
        ptr_The= (float *)(entradaRed.data + entradaRed.step[0]*0 +
entradaRed.step[1]*1+
entradaRed.step[2]*36+entradaRed.step[3]*0+entradaRed.step[4]*0);
        ptr_Phi=(float *)(entradaRed.data + entradaRed.step[0]*0 +
entradaRed.step[1]*2+
entradaRed.step[2]*36+entradaRed.step[3]*0+entradaRed.step[4]*0);
        for(int i=0; i<resThe*resPhi;i++){
                ptr_The[i]=The_max;
                ptr_Phi[i]=Phi_max;
        }



        mutexMap.unlock();

        bufferCounter++;




        // 5. Create infer request

        InferRequest infer_request = executable_network.CreateInferRequest();

        //printf("Despues de crear la inferencia\n");

        // 6. Prepare input
        //Read input image to a blob and set it to an infer request without resize
and layout conversions.


        InferenceEngine::TensorDesc tDesc(InferenceEngine::Precision::FP32,
                                          {1,3,37,16,32},
                                          InferenceEngine::Layout::NCDHW);

        Blob::Ptr audioBlob = InferenceEngine::make_shared_blob<float>(tDesc,
entradaRed.ptr<float>()); // just wrap Mat data by Blob::Ptr without allocating of
new memory

        infer_request.SetBlob(network_input_name, audioBlob);  // infer_request
accepts input blob of any size
        //sendToMatlab("entrada.m",entradaRed);
        // 7. Do inference
        /* Running the request synchronously */

        infer_request.Infer();

        // 8. Process output
        Blob::Ptr output = infer_request.GetBlob(network_output_name);

        using myBlobType =
InferenceEngine::PrecisionTrait<InferenceEngine::Precision::FP32>::value_type;
        InferenceEngine::TBlob<myBlobType>& tblob =
dynamic_cast<InferenceEngine::TBlob<myBlobType>&>(*output);
        auto coordenadas = tblob.data(); // You can access to each element by
probabilities[idx]

        printf("Las coordenadas son
```

```
x=%f,y=%f,z=%f\n",coordenadas[0],coordenadas[1],coordenadas[2]);

        //Realizamos el paso de coordenadas cartesianas a coordenadas polares 
esféricas

        r= sqrt(pow(coordenadas[0],2) + pow(coordenadas[1],2) + pow(coordenadas[2],
2));
        th= acos(coordenadas[2]/r)*(180/PI);//elevacion
        printf("El angulo de elevacion es th= %f\n", th);
        phi= atan2(coordenadas[1],coordenadas[0])*(180/PI);// azimut
        printf("El angulo de azimut es phi= %f\n", phi);




}

//--------------------------------------------------------------
void ofApp::keyPressed  (int key){

}

//--------------------------------------------------------------
void ofApp::keyReleased(int key){

}

//--------------------------------------------------------------
void ofApp::mouseMoved(int x, int y ){

}

//--------------------------------------------------------------
void ofApp::mouseDragged(int x, int y, int button){

}

//--------------------------------------------------------------
void ofApp::mousePressed(int x, int y, int button){

}

//--------------------------------------------------------------
void ofApp::mouseReleased(int x, int y, int button){

}

//--------------------------------------------------------------
void ofApp::mouseEntered(int x, int y){

}

//--------------------------------------------------------------
void ofApp::mouseExited(int x, int y){

}

//--------------------------------------------------------------
void ofApp::windowResized(int w, int h){

}

//--------------------------------------------------------------
void ofApp::gotMessage(ofMessage msg){

}
```

```
//--------------------------------------------------------------
void ofApp::dragEvent(ofDragInfo dragInfo){

}
```