E.T.S. de Ingeniería Industrial, Informática y de Telecomunicación

A Deep Learning Approach to Aerial LiDAR Point Cloud Segmentation

Máster Universitario en
Ingeniería Informática

Trabajo Fin de Máster

Autor: Christian Gutiérrez Lancho

Tutor: Mikel Galar Idoate

Pamplona, Junio 2021

upna Universidad Pública de Navarra
Nafarroako Unibertsitate Publikoa

Contenido

1.Introducción

Gracias a la evolución tecnológica experimentada estos últimos años, cada vez disponemos de hardware más potente a precios más bajos. Esto ha provocado que la información que seamos capaces de procesar sea cada vez más compleja, pudiendo trabajar, por ejemplo, con imágenes sin dificultad. Sin embargo, existe un tipo de dato algo más complejo que las imágenes que está cogiendo cada vez más fuerza, un tipo concreto de dato 3D, las nubes de puntos.

Las nubes de puntos son, como su propio nombre indica, una agrupación de puntos que representan una posición en el espacio mediante las coordenadas X, Y, Z. En la Figura 1 y 2 podemos ver dos ejemplos.

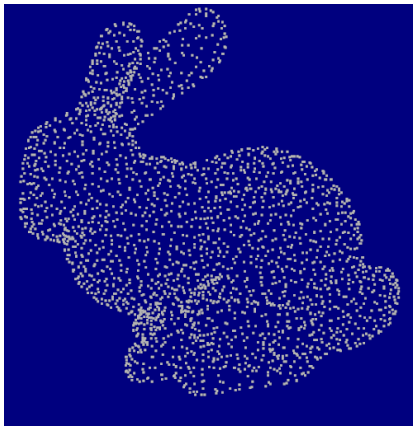

Figura 1.Ejemplo de nube de puntos [fuente]

Figura 2. Ejemplo de nube de puntos donde tenemos RGB por cada uno de ellos [fuente]

Las nubes de puntos utilizadas a lo largo de este trabajo han sido captadas desde el aire a través de un sensor conocido como LiDAR (*Light Detection and Ranging*). Un LiDAR es un dispositivo que permite determinar distancias desde un punto emisor hasta un objeto o superficie. La distancia al objeto se calcula midiendo el tiempo transcurrido entre la emisión del pulso y la recepción de este. El sensor LiDAR puede ir colocado en diferentes tipos de vehículos, desde coches hasta aviones. Incluso existen sensores LiDAR adaptados para que se lleven en mochilas o LiDAR fijos. En este trabajo nos centramos en la tecnología de LiDAR aéreo. En este caso, un avión que lo incluye en su base, sobrevuela una zona y genera una nube de puntos que corresponda con el lugar sobrevolado. Este proceso se ilustra en la Figura 3.

Figura 3. Ejemplo de barrido LiDAR desde el aire [fuente]

Las nubes de puntos tienen distintos usos, desde aplicaciones en realidad virtual o animación hasta creación de mapas 3D. En este trabajo nos vamos a centrar en las aplicaciones relacionadas con la digitalización del terreno. A partir de una toma de nube de puntos aérea, hay tres productos principales que se suelen generar:

- **DEM** (*Digital Elevation Model*): Representa la superficie de la tierra desnuda, eliminando los obstáculos ya sean naturales o edificios. En la Figura 4 se puede apreciar un ejemplo.
- **DSM**(*Digital Surface Model*): Al contrario que el DEM, en este caso se incluyen objetos naturales o construidos por el ser humano. En la Figura 5 se puede apreciar un ejemplo.
- **DTM** (*Digital Terrain Models*): generalmente complementa un DEM, incluyendo características vectoriales del terreno natural, como ríos o cordilleras. Un DTM se puede interpolar para extraer un DEM pero no al revés.

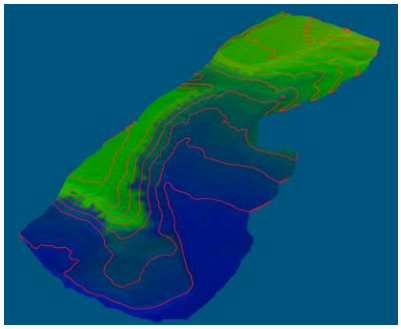

Figura 4. Ejemplo de DEM [fuente]

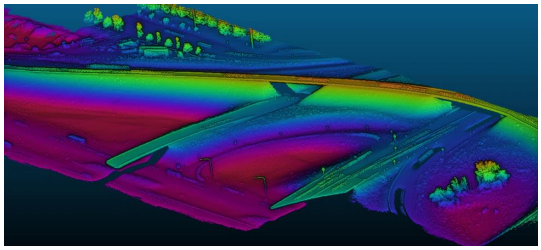

Figura 5. Ejemplo de DSM [fuente]

El principal reto que representa este tipo de datos para poder extraer productos como el DEM o el DSM [1] es la enorme cantidad de puntos que existen. Para poder extraer estos productos es necesario saber qué representa cada uno de los puntos. Si un punto es suelo se debería incluir en el DEM / DTM, sin embargo, si un punto pertenece a un edificio, este sería importante para el DSM, pero no para el DEM / DTM. En la Figura 6 podemos ver de manera gráfica la diferencia entre estos dos productos.

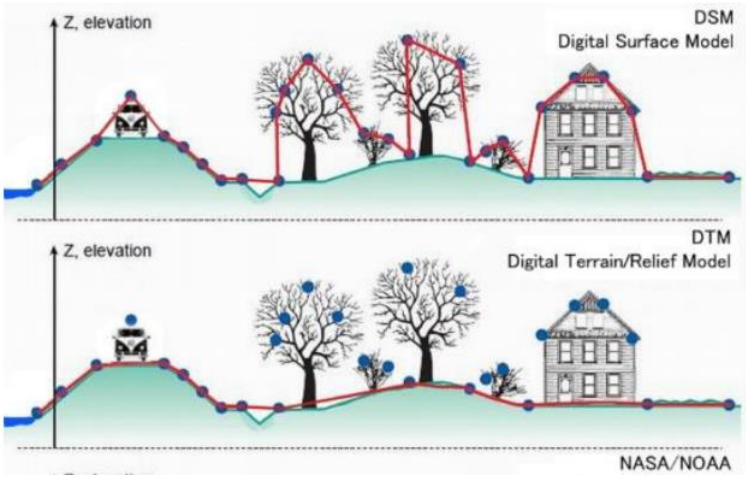

Figura 6. Diferencias entre DSM y DTM [fuente]

Estas nubes de puntos pueden ser etiquetadas de manera manual, el problema es la gran cantidad de puntos que puede llegar a haber, volviendo esta tarea lenta e ineficiente. Lo ideal sería hacerlo de manera automática. Para poder enfrentarnos a este problema, el cual no es trivial, ya que para poder clasificar los puntos se necesita un entendimiento de qué es lo que están representando, recurriremos a una técnica cada vez más popular hoy en día, la inteligencia artificial. La inteligencia artificial (IA) consiste en utilizar una serie de algoritmos que proporcionen a una máquina capacidades similares a la de una persona para resolver un problema específico. Una de las ramas dentro de la inteligencia artificial conocida como *Machine Learning* [2], permite que las máquinas aprendan a través de ejemplos, para generalizar ese conocimiento y poder solventar así problemas sobre datos nunca vistos antes por la máquina. Un subcampo del *Machine Learning* conocido como *Deep Learning* [3], realiza este aprendizaje mediante estructuras conocidas como redes neuronales artificiales, las cuales pretenden imitar el funcionamiento de las redes neuronales biológicas, pero llevadas a una máquina.

En nuestro caso, queremos que la máquina aprenda la forma que tiene cada una de las clases que queremos clasificar, como suelo o edificio, para que sea capaz de etiquetar los puntos correctamente. Para lograrlo, la máquina recibirá pequeñas zonas ya etiquetadas, donde cada punto está clasificado correctamente por una persona. Gracias a estas zonas etiquetadas y mediante técnicas propias del *Deep Learning* el ordenador será capaz de entender la forma que posee cada una de las clases, logrando clasificar correctamente zonas nunca vistas.

Cada vez más empresas recurren a la inteligencia artificial para resolver problemas del mundo real, un ejemplo sería Amazon con sus recomendaciones o Tesla con sus coches autónomos. Dichas técnicas también pueden ser aplicadas para segmentar nubes de puntos, en esta memoria explicaremos cómo hacerlo.

1.1.Motivación

Las nubes de puntos que se pueden obtener en la actualidad son cada vez son más densas, proporcionándonos más información y detalle. Sin embargo, tener que anotar millones de puntos de manera manual, además de ser un trabajo tedioso, no es eficiente ni desde un punto de vista temporal ni desde un punto de vista económico.

Por ello, lo ideal sería que una máquina fuera capaz de realizar este trabajo, obteniendo resultados mucho más rápido que un etiquetado manual. Para ello, como se ha mencionado en la introducción, nos vamos a apoyar en un subcampo del *Machine learning*, conocido como *Deep Learning*.

El objetivo de este trabajo será automatizar la segmentación de nubes de puntos tomadas mediante LiDAR aéreo usando las técnicas más recientes de *Deep Learning* adaptadas a nuestro problema. Con el termino *segmentar* nos referimos a asignar a cada punto de la nube de puntos una clase a la que pertenece. Un ejemplo de clase sería “suelo” o “edificio”.

Para conseguir este objetivo, a lo largo de este trabajo analizaremos las herramientas que nos permitirán trabajar con este tipo de datos, estudiaremos las características inherentes a las nubes de puntos y realizaremos un recorrido a lo largo de diferentes modelos de *Deep Learning* centrados tanto en tareas de clasificación como de segmentación de nubes de puntos.

1.2.Metodología

Una vez explicada la idea general del proyecto vamos a definir una metodología que nos permita resolver el problema propuesto.

Todo problema de *Machine Learning* presenta una serie de fases que se deben abordar:

- En primer lugar, en el Capítulo 2, hablaremos sobre los diferentes datos 3D, las peculiaridades de las nubes de puntos respecto a los demás y una breve introducción a la inteligencia artificial y a conceptos base comprendidos dentro del *Machine Learning* / *Deep Learning*. También se mencionarán las herramientas que se van a utilizar para tratar con este tipo de datos.

- En segundo lugar, nos centraremos en analizar los datos de los que disponemos y en definir claramente el problema, especificando las clases que pretendemos clasificar. Abordaremos cuestiones como la geolocalización de los datos y características propias de la nube de puntos tomadas con un LiDAR aéreo, como la densidad o los campos que posee cada punto. También se hablará del pre-proceso a aplicar sobre estos datos para poder entrenar un modelo a partir de estos. Todo ello se recoge en el Capítulo 3.

- En tercer lugar, haremos un recorrido por los principales modelos de *Deep Learning* desarrollados durante estos últimos años centrados en la segmentación de nubes de puntos. El objetivo es entender la complejidad del problema, las primeras aproximaciones a este, y la evolución que han sufrido estos modelos hasta el día de hoy. Todo ello se recoge en el Capítulo 4.

- En cuarto lugar, se analizará los preprocesos a aplicar sobre nuestros datos para poder entrenar nuestros modelos. También se estudiará la función de coste que se utilizará durante nuestros entrenamientos, así como la creación de métricas de error propias para que la evaluación de los modelos sobre los datos nos proporcione más información sobre la gravedad de los errores cometidos. Todo ello se recoge en el Capítulo 5.

- En quinto lugar, una vez explicados los modelos a utilizar en el apartado anterior, nos centraremos en realizar experimentos sobre dichos modelos con la idea de comprarlos desde un punto de vista práctico, sobre un conjunto de datos anotados. Se analizarán los resultados obtenidos tanto desde un punto de vista métrico como visual. Con el mejor modelo obtenido se realizará alguna prueba adicional para intentar mejorar los resultados. Todo ello se recoge en el Capítulo 6.

- Para finalizar, se hablará sobre las conclusiones del trabajo y las líneas futuras de estudio. Todo ello se recoge en el Capítulo 7.

2.Preliminares

En esta sección se van a explicar conceptos básicos de diferentes campos, necesarios para una buena comprensión de este trabajo. Este capítulo estará dividido en 3 secciones que se centran en aspectos distintos:

- *Sección 2.1*: Describirá los diferentes tipos de datos 3D existentes y como se obtienen. Nos centraremos en las características propias de las nubes de puntos al ser el tipo de dato con el que se va a trabajar.
- *Sección 2.2*: En esta sección trataremos conceptos básicos relacionados con el *Machine Learning* y el *Deep Learning*.
- *Sección 2.3*: Para finalizar, hablaremos de las diferentes herramientas utilizadas a lo largo de este trabajo, desde visores 3D a librerías especializadas en el tratamiento de datos LiDAR.

2.1.Datos 3D

En esta sección profundizaremos en los datos 3D. Se tratarán temas como las diferentes representaciones de estos y la forma de capturarlos. Para finalizar, se estudiarán las características especiales de las nubes de puntos, definiendo las ventajas y dificultades presentes al tratar con este tipo de dato.

2.1.1.Formas de representar datos 3D

Los datos 3D, al no estar limitados a dos dimensiones, pueden representarse de diferentes formas, cada una con sus ventajas y desventajas. En este apartado se mencionarán alguna de las más utilizadas:

- **Dato volumétrico:** Esta opción consiste en dotar al espacio 3D de una estructura fija dividiendo el espacio en cubos del mismo tamaño, este proceso se conoce como voxelización y el tipo de dato con estas características se conoce como dato volumétrico. Esto nos proporcionaría una estructura clara, lo que facilitaría el tratamiento de la información, pero nos haría perder precisión al establecer como unidad mínima de información el tamaño del cubo elegido para voxelizar el espacio.

- **Mesh**: Otra opción podría ser representar un objeto 3D mediante una malla, conocida como *mesh* en inglés. Un *mesh* son un conjunto de vértices conectados entre si formando una superficie modelada por triángulos. Este método también dota a los datos de cierta estructura, pero es menos restrictivo que la voxelización.

- **RGB-D:** Este tipo de dato 3D consiste en capturar imágenes que tengan tanto RGB como una canal extra que defina la profundidad de los pixeles, dato conocido como RGB-D donde la D proviene de la palabra en inglés *Depth* que significa profundidad, lo que nos proporcionaría una imagen con profundidad, es decir, una imagen 3D.

- **Nubes de puntos:** De todos estos tipos de datos, en los que nos vamos a centrar en este trabajo van a ser las nubes de puntos. Este tipo de dato no tiene ningún tipo de estructura definida, y partiendo de él se puede obtener tanto mallas como datos volumétricos.

En la Figura 7 podemos encontrar un ejemplo visual de cada representación de dato 3D mencionada.

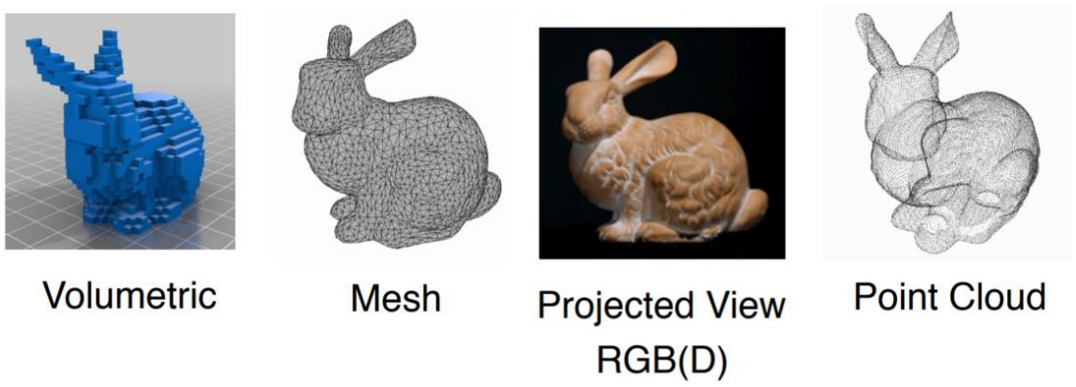

Figura 7. Diferentes representaciones de datos 3D [fuente]

2.1.2.Obtención de datos 3D

Como se ha explicado en el apartado anterior, el tipo de datos volumétrico o las mallas se obtiene a partir de las nubes de puntos. Estas se suelen obtener a partir de un sensor llamado LiDAR [4]. En este trabajo nos vamos a centrar en las nubes de puntos tomadas con un LiDAR aéreo. El LiDAR puede tomar sus mediciones de dos formas distintas:

- **LiDAR de pulsos:** El emisor emite pulsos de luz, y el tiempo que tarda entre emitirlo y recibirlo es lo que determina la distancia.
- **LiDAR de medición de fase:** En este caso el emisor emite un haz láser continuo, cuando recibe la señal reflejada mide la diferencia de fase entre la emitida y la reflejada.

También existen diferentes tipos de escaneado para el LiDAR aéreo:

- **Líneas:** Dispone de un espejo rotatorio que va desviando el haz láser. Como patrón de escaneado produce líneas paralelas en el terreno. El inconveniente principal de este sistema es que al girar el espejo en una sola dirección no siempre se tiene mediciones.

- **ZigZag:** En este caso el espejo es rotatorio en dos sentidos (ida y vuelta). Produce líneas en zigzag como patrón de escaneado. Tiene la ventaja de que siempre está midiendo, pero al tener que cambiar de sentido de giro, la aceleración del espejo varía según su posición. Esto hace que en las zonas cercanas al límite de escaneado lateral (donde varía el sentido de rotación del espejo), la densidad de puntos escaneados sea mayor que en el nadir.

- **De fibra óptica:** Desde la fibra central de un cable de fibra óptica y con la ayuda de unos pequeños espejos, el haz láser es desviado a las fibras laterales montadas alrededor del eje. Este sistema produce una huella en forma de circunferencias solapadas. Al ser los espejos pequeños, la velocidad de toma de datos aumenta respecto a los otros sistemas, pero el ángulo de escaneado (FOV) es menor.

- **Elíptico:** En este caso el haz láser es desviado por dos espejos que producen un patrón de escaneado elíptico. Como ventajas del método podemos comentar que el terreno es a veces escaneado desde diferentes perspectivas, aunque el tener dos espejos incrementa la dificultad al tener dos medidores angulares.

En la Figura 8 podemos ver de manera gráfica cada uno de los patrones.

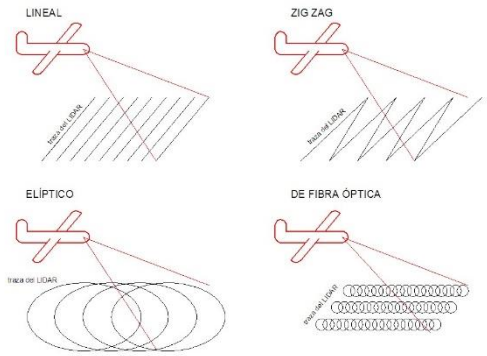

Figura 8. Tipos de escaneado del LiDAR aéreo [fuente]

Aunque en este trabajo nos centremos en nubes de puntos tomadas desde el aire, no todas las nubes de puntos captadas mediante LiDAR se toman desde el aire. En la Figura 9 podemos observar un sensor LiDAR fijo a nivel de suelo para generar un modelo 3D de edificios. Otro ejemplo sería el *Mobile Mapping* [5], técnica cada vez más usada para generar modelos 3D de las ciudades. En vez de utilizar un avión se utiliza un vehículo sobre el cual se coloca un sensor LiDAR que va captando el espacio por el que el coche circula. En las Figuras 10 y 11 podemos ver un ejemplo del vehículo con el sensor LiDAR y de la nube de puntos que se genera mediante esta técnica.

Figura 9. Lidar a nivel de suelo [fuente]

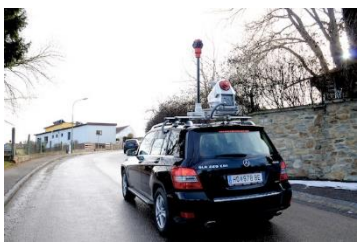

Figura 11. Mobile mapping [fuente]

Figura 10. Nube de puntos captada mediante mobile mapping [fuente]

Otra de las manera de captar una nube de puntos mediante LiDAR desde el aire sería a través de un dron, gracias en parte a la popularidad y abaratamiento de estos. Además, presentan ciertas ventajas respecto a los datos tomados desde un avión. Debido a su tamaño más reducido y a su maniobrabilidad, pueden volar más proximos al suelo y son capaces de conseguir más detalles en ciertas zonas, como las fachadas de los edificios.

A pesar de que el sensor LiDAR es cada vez más usado, gracias en parte a la reducción de precio que han experimentado estos último años, no todos los datos 3D se captan con este tipo de sensor. Una técnica conocida como SfM (structure from Motion) [6] se apoya en imágenes tomadas desde diferentes ángulos sobre una zona para poder generar un modelo 3D similar al que se consigue con LiDAR. En la Figura 12 podemos ver una gráfico que explica como funciona.

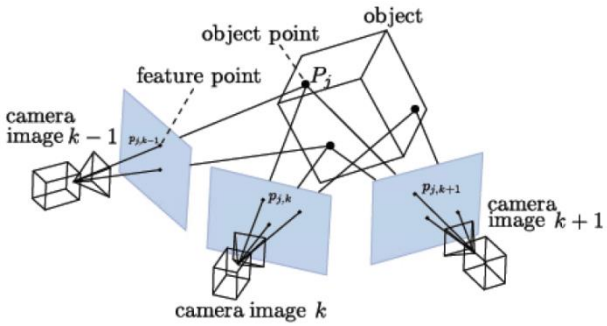

Figura 12. Ejemplo de cómo se aplica el SfM [fuente]

Las nubes de puntos con las que se trabajará en este proyecto se han tomado desde el aire, con un avión, usando un patrón circular, similar al patrón utilizado para tomar los datos del trabajo de investigación asociado.

2.1.3.Características de las nubes de puntos

Como ya se ha comentado anteriormente, en este trabajo trabajaremos con nubes de puntos aéreas tomadas mediante LiDAR. El motivo por el cual trabajamos con las nubes de puntos y no con mallas o datos volumétricos, es porque este dato es el que más información potencial contiene. Tanto el meshing como voxelizar el espacio es computacionalmente costoso, y generalmente se pierde información al reducir el número de puntos que componen el dato. Además, tener los puntos clasificados ayuda mucho a la hora de conseguir un buen meshing, ya que nos permite eliminar aquellos puntos que pertenecen a clases que no nos interesan, así como a seleccionar aquellos que sí aportan valor.

Sin embargo, las nubes de puntos son un tipo de dato cuyo tratamiento es complicado. En las imágenes tenemos una estructura bien definida, donde por cada píxel podemos saber quiénes son sus vecinos. Podemos representar las imágenes como matrices tridimensionales en caso de que sea RGB, una estructura clara que facilitan su tratamiento. En la Figura 13 podemos ver de manera gráfica cómo se representa una imagen a color con tres matrices (Red, Green, Blue) superpuestas.

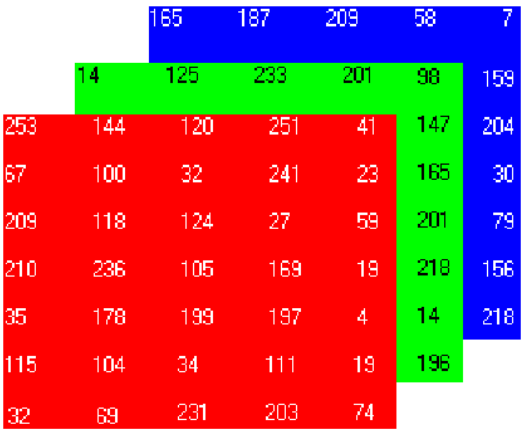

Figura 13. Representación matricial de una imagen a color [fuente]

Sin embargo, una nube de puntos es un **conjunto de puntos desordenados** y por tanto **invariante a permutaciones de sus miembros**. Además, no hay una estructura de vecindad definida. Para ver esto de manera más clara vamos a apoyarnos en la Figura 14. Como se puede ver en el gráfico de la parte superior de Figura 14, la nube de la izquierda es exactamente igual a la de la derecha, sin embargo, si asignamos índices a los puntos y lo representamos como una matriz obtenemos resultados diferentes. Es decir, si nuestro algoritmo recibe la matriz de la izquierda o la de la derecha a pesar de ser distintas, debería comportarse de la misma forma, ya que es la misma nube de puntos.

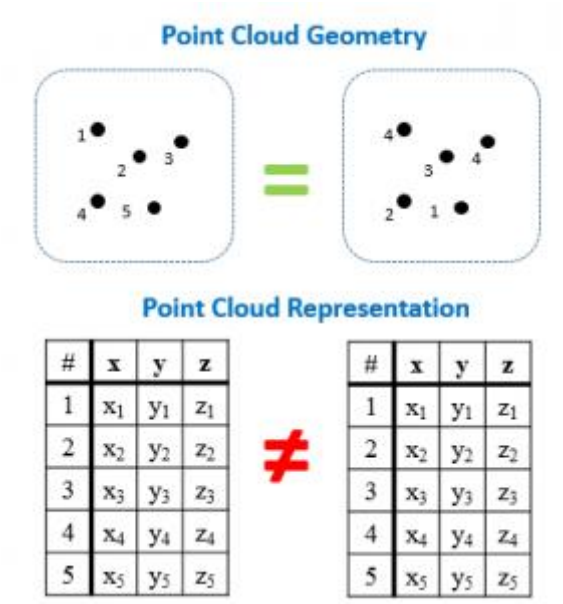

Figura 14. Demostración gráfica de que las nubes de puntos son conjuntos de puntos desordenados [fuente]

Esta característica inherente a las nubes de puntos supone ciertas complicaciones que hay que resolver para poder trabajar correctamente con ellas. En el Capítulo 4 veremos cómo los diferentes modelos de *Deep Learning* se enfrentan a este problema.

2.2.Machine Learning

Como se ha mencionado anteriormente el *Machine Learning* es una rama de la Inteligencia Artificial. La Inteligencia Artificial (IA) es un subcampo de la informática que se creó en la década de 1960, y que trata de resolver tareas fáciles para las personas, pero complicadas para las máquinas. Hay muchos problemas que se pueden englobar dentro de la inteligencia artificial, desde reconocimiento de objetos en imágenes, traducción, clasificación de sonidos, actividades creativas como recrear una imagen con el estilo de un pintor famoso etc.

Dentro de la Inteligencia Artificial, el Aprendizaje Automático o *Machine Learning* es una disciplina centrada en la construcción de modelos que optimicen criterios de rendimiento haciendo uso de datos y experiencias anteriores.

Por ende, cualquier programa que tenga un comportamiento inteligente similar al de un humano puede ser IA, pero debe aprender sus parámetros de manera automática a partir de datos para que sea considerado *Machine Learning*.

Dependiendo del problema y de las características que lo envuelven, podemos encontrar distintos tipos de algoritmos de aprendizaje, de los cuales, vamos a explicar cuatro de ellos:

- **Aprendizaje supervisado**: se genera un modelo predictivo a partir de un conjunto de datos previamente etiquetados, es decir, conocemos la salida deseada. Si pretendemos clasificar imágenes de perros y gatos, tendríamos un conjunto de imágenes con una etiqueta que indicaría si en la imagen hay un perro o un gato. El algoritmo aprendería en base a esos datos, para ser capaz de aprender y poder clasificar en un futuro imágenes de perros y gatos nunca vistas antes.

- **Aprendizaje no supervisado**: Los datos de entrada no están clasificados. En este caso no pretendemos buscar una función que se ajuste a una salida deseada, sino en aumentar el conocimiento de los datos disponibles y de los datos futuros. Por ejemplo, agrupar datos en función de su similitud (*clustering*) [7] o simplificar la estructura de los datos manteniendo sus características fundamentales (reducción de dimensionalidad) [8]. En la Figura 15 podemos encontrar un ejemplo de clustering y en la Figura 16 un ejemplo de reducción de dimensionalidad.

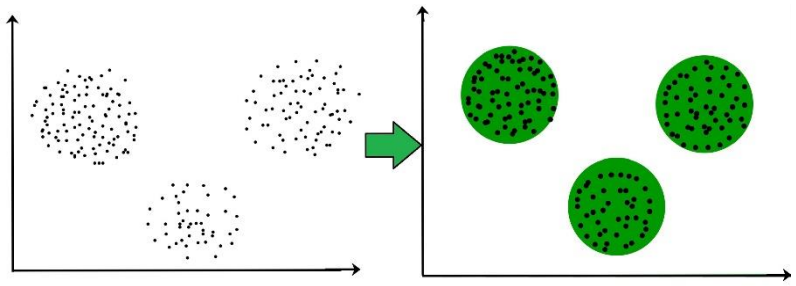

Figura 15. Ejemplo de aprendizaje no supervisado, clustering [fuente]

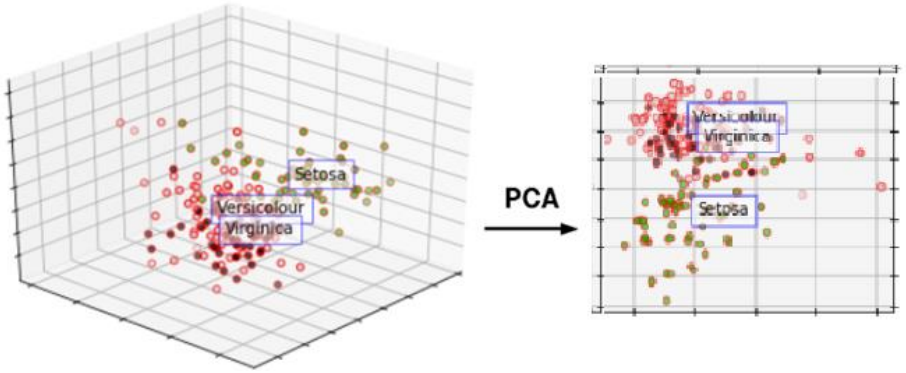

Figura 16. Ejemplo de aprendizaje no supervisado, reducción de dimensionalidad mediante el algoritmo PCA [fuente]

- **Aprendizaje semi-supervisado**: La combinación de aprendizaje supervisado con el no supervisado. Un ejemplo de este tipo de algoritmos sería tener una gran cantidad de datos, unos pocos etiquetados, pero los demás no. Se entrena un modelo con los pocos datos etiquetados, después se clasifica los datos no etiquetados. Aquellos predichos con una probabilidad muy alta se incluyen como nuevos datos de entrenamiento. Y de manera iterativa se consigue un modelo mejor, siendo el propio algoritmo quien se proporciona sus propios datos de entrenamiento.

- **Aprendizaje por refuerzo**: En este tipo de algoritmos se recibe algún tipo de valoración acerca de la respuesta dada. Si la respuesta es correcta, su comportamiento es similar al del aprendizaje supervisado, es decir, se le indica al algoritmo que ha acertado. Sin embargo, cuando la respuesta es errónea, no se le indica al algoritmo cuál debería haber sido la respuesta correcta como sí pasaría en el aprendizaje supervisado, sino que solo se indica que el comportamiento ha sido erróneo. Esta forma de aprender se asemeja más al aprendizaje de los animales.

Con técnicas de aprendizaje supervisado podemos realizar tareas de clasificación o regresión. La diferencia radica en el tipo de resultado esperado. En un problema de clasificación el resultado esperado es una clase, es decir, una serie de categorías arbitrarías definidas dentro del problema. Retomando el ejemplo anterior, dada una imagen queremos saber si lo que aparece en ella es un perro o un gato. En este caso, solo existen dos clases, *perro* o *gato*. Tras darle una imagen a nuestro algoritmo para que la clasifique, solo puede darnos una de estas dos clases.

Cuando hablamos de regresión, el resultado no es una clase, sino un número. El resultado es un valor numérico dentro de un conjunto infinito de resultados. El ejemplo más típico suele ser el calcular el precio de una viviendo en función de características como el número de habitaciones, metros cuadrados de la propiedad etc.

Un algoritmo centrado en el aprendizaje debe ser capaz de generalizar correctamente, es decir, aprender a través de una serie de ejemplos etiquetados y ser capaz de extrapolar ese conocimiento a ejemplos que no ha visto nunca. Por ello, los datos usados en un algoritmo de *Machine Learning,* conocido como *dataset*, se suele dividir en tres conjuntos:

- Conjunto de Train: Conjunto que usa el algoritmo para aprender, son lo ejemplos que si que pasan por el y por tanto conoce. Suele estar formado por el 70% del dataset, la tarea de aprendizaje es compleja, por eso se necesita un porcentaje alto de datos.

- Conjunto de validación: Conjunto de datos que se utilizan para ser evaluados y ajustar los híper-parámetros del algoritmo en función de esta evaluación. Suele estar formado por la mitad de los datos que no se encuentran en Train.

- Conjuntos de test: Conjuntos de datos que no se han usado de ninguna forma, ni para entrenar, ni para ajustar los parámetros del modelo. Se utiliza para evaluar el rendimiento de un modelo ya entrenado. Suele estar formado por la mitad de los datos que no se usan para Train.

Es importante tener claro que estos tres conjuntos son **disjuntos**, es decir, que la intersección entre ellos es vacía. Si un ejemplo va a un conjunto no puede ir a otro. Hemos comentado que el conjunto de test se utiliza para evaluar el rendimiento del modelo, para ello se infiere (verbo que hace referencia al proceso de predecir un ejemplo) el conjunto de test y se sacan un conjunto de métricas que nos indican lo bien que rinde el modelo. Antes de explicar las métricas, es necesario estar familiarizado con los siguientes términos que se utilizan para clasificar una predicción:

- *True Positive (TP):* Número de ejemplos que se clasifican correctamente como positivos. Si tenemos un problema en detectar cáncer a un paciente, sería el número de personas que tienen cáncer de verdad y que el algoritmo predice que tienen cáncer.

- *False Positive (FP):* Número de ejemplos que se clasifican incorrectamente como positivos. Siguiendo el ejemplo anterior, serían el número de pacientes que no tiene cáncer, pero nuestro algoritmo predice que sí.

- *True Negative (TN):* Número de ejemplos que se clasifican correctamente como no positivos. Siguiendo el ejemplo anterior, serían el número de pacientes que no tienen cáncer y nuestro algoritmo predice que no lo tienen.

- *False Negative (FN):* Número de ejemplos que se clasifican como no positivos incorrectamente. Siguiendo el ejemplo anterior, serían el número de pacientes que si tienen cáncer, pero nuestro algoritmo predice que no.

En la Figura 17 podemos ver una representación gráfica sobre un problema binario, de los términos explicados:

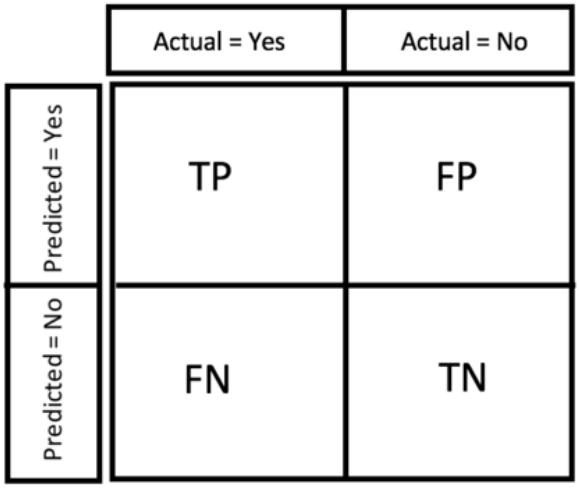

Figura 17. Representación gráfica del tipo de predicciones [fuente]

Las métricas más comunes que se suelen utilizar en estos tipos de problemas se explican en la Tabla 1.

Nombre	Formula	Descripción
Accuracy	$\frac{TP + TN}{Total\ samples}$	Rendimiento general del modelo.
Precision	$\frac{TP}{TP + FP}$	Precisión sobre las muestras positivas
Sensitivity (TPR)	$\frac{TP}{TP + FN}$	Proporción de ejemplos positivos clasificados correctamente, con respecto a todos los ejemplos positivos predichos.
Specificity (TNR)	$\frac{TN}{TN + FP}$	Proporción de ejemplos negativos clasificados correctamente, con respecto a todos los ejemplos negativos predichos.
F1 score	$\frac{2TP}{2TP + FP + FN}$	La media armónica entre la precisión y el TPR. Nos indica como de preciso y robusto es nuestra predicción.

Tabla 1. Métricas comunes en algoritmos de aprendizaje

En el mundo real los problemas suelen tener más de dos clases, problemas conocidos como multiclase. Estas métricas se pueden extrapolar a ese tipo de problemas, comparando una clase contra el resto.

2.3.Deep Learning

El término *Deep Learning* hace referencia a aquellos algoritmos que utilizan estructuras conocidas como redes neuronales para aprender a resolver un problema. La principal diferencia que encontramos entre los algoritmos de *Deep Learning* y algoritmos de *Machine Learning* que no están englobados dentro del *Deep Learning,* radica en que la fase de extracción de características y la de aprendizaje ocurren ambas dentro de la red neuronal, en vez de ser dos fases completamente separadas. En la Figura 18 podemos ver una representación gráfica de esta diferencia:

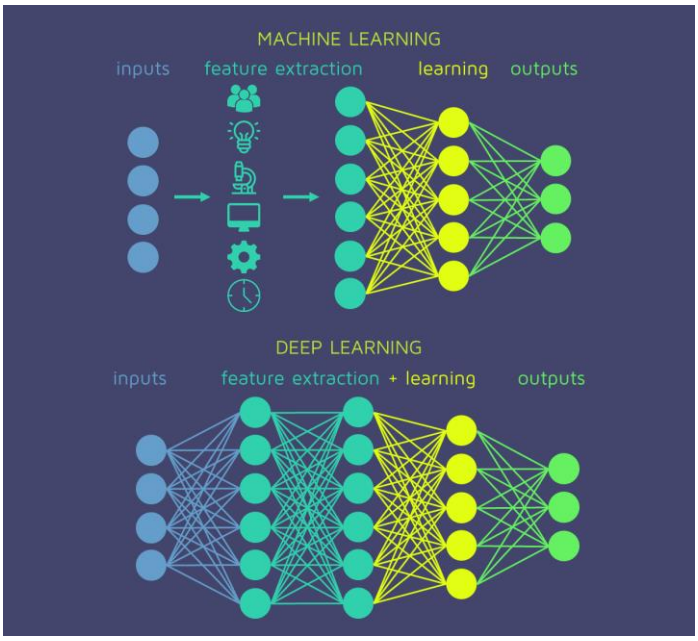

Figura 18. Diferencia entre Machine Learning y Deep Learning [fuente]

El *Deep Learning* lleva a cabo el proceso de *Machine Learning* usando redes neuronales artificiales. En la Figura 19 podemos ver de manera gráfica como el *Machine Learning* es un subconjunto dentro de la IA, y a su vez, el *Deep Learning* es un subconjunto dentro del *Machine Learning*.

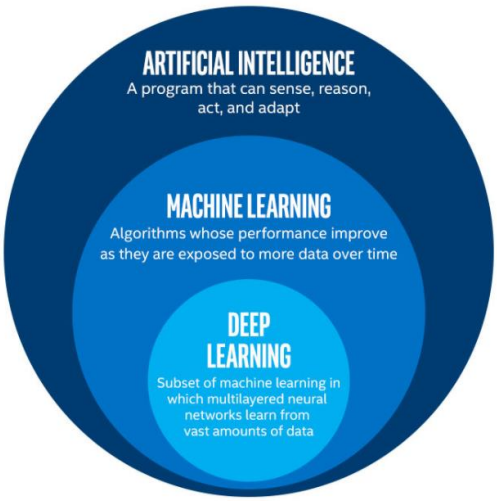

Figura 19. Relaciones entre IA, Machine Learning y Deep Learning [fuente]

En este proyecto la técnica de aprendizaje utilizado ha sido el aprendizaje supervisado ya que las redes que vamos a utilizar se entrenan con ejemplos etiquetados. Para poder entender correctamente los algoritmos de *Deep Learning* que se utilizan para segmentar nubes de puntos es necesario tener unos conocimientos mínimos sobre ciertos términos relacionados con las redes neuronales (*Sección 2.3.1*), y sobre las redes neuronales convolucionales (*Sección 2.3.2*).

2.3.1.Redes Neuronales Artificiales

El ser humano tiene una gran capacidad para extrapolar y generalizar conceptos, siendo capaces de aprender a resolver tareas muy complejas. Esta capacidad se debe a nuestro cerebro, más concretamente a las células que lo forman, conocidas como neuronas. Las redes neuronales son un conjunto de neuronas conectadas entre sí, en las que no se delega una tarea a cada neurona, sino que trabajan conjuntamente. La idea, es imitar estás estructuras bilógicas, para que las máquinas que disponen de una gran capacidad de cálculo sean capaces de imitar el aprendizaje humano.

Aunque las neuronas son extremadamente complejas, computacionalmente hablando, son relativamente simples. La neurona recibe una serie de entradas, impulsos eléctricos a través de sus dendritas, y devuelve una salida, un impulso eléctrico a través de su axón. La neurona suma internamente sus entradas y si el impulso resultante supera un umbral, envía un nuevo impulso a través del axón, que distribuye la señal por todas las ramificaciones, alcanzando así a otras neuronas.

En 1958, Rosenblatten creó el perceptrón [9], la cual es la forma más simple de computación neuronal. Esta es la base a partir de la cual se forma las redes neuronales artificiales más complejas. Como podemos ver en la Figura 20, imitamos la estructura y el comportamiento de una neurona real, pero más simplificado.

En el perceptrón sustituimos las dendritas por una serie de entradas $(x_1, x_2, \dots, x_n)$ y el axón por una única salida $y = f(x)$. Cada una de las entradas tiene asociado un peso $(w_1, w_2, \dots, w_n)$ que pueden debilitar o potenciar las entradas. Existe también un peso fijo denominado con la letra b, conocido como sesgo (*bias*), que no depende de ninguna entrada, es fijo. Su función es alejar la frontera de decisión del origen durante el entrenamiento. Todo esto queda ilustrado de manera gráfica en la Figura 20.

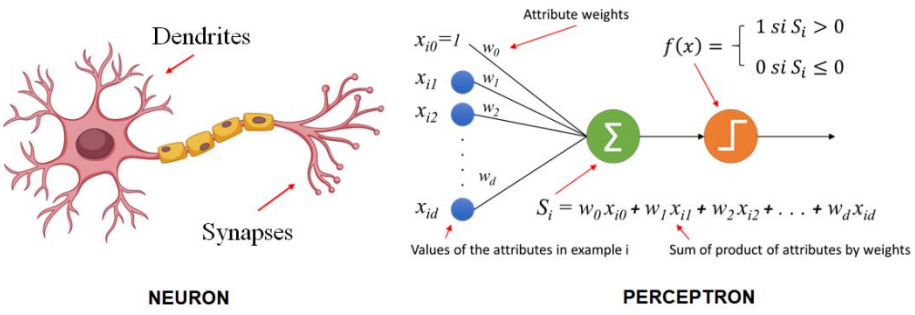

Figura 20. Neurona biológica vs neurona artificial [fuente]

Dentro del perceptrón, lo que sería el cuerpo celular si hacemos el símil con una neurona, encontramos la función de activación, también conocida como función de propagación, que se encarga de determinar la salida del perceptrón. La fórmula general de esta función se muestra en la Ecuación 1.

$$y = g\left(\sum_{i=1}^{n} w_i * x_i + b\right) \qquad (1)$$

La función $g(x)$ puede comportarse de diferentes formas, la más simple y siguiendo el ejemplo de la Figura 20, sería activar la neurona si se sobrepasa un umbral. Este comportamiento se muestra en la Ecuación 2.

$$g(x) = \begin{cases} 1 \; si \sum_{i=1}^{n} w_i * x_i + b \geq umbral \\ 0 \; en \; otro \; caso \end{cases} \qquad (2)$$

Existen otro tipo de funciones de activación más complejas como la sigmoide, cuyos resultados oscilan entre $[0-1]$, siguiendo la ecuación y representación gráfica descrito en la Figura 21. Otra función de activación muy utilizada es la unidad lineal rectificada, más conocida como ReLU. Su ecuación y representación gráfica también están descritos en la Figura 21.

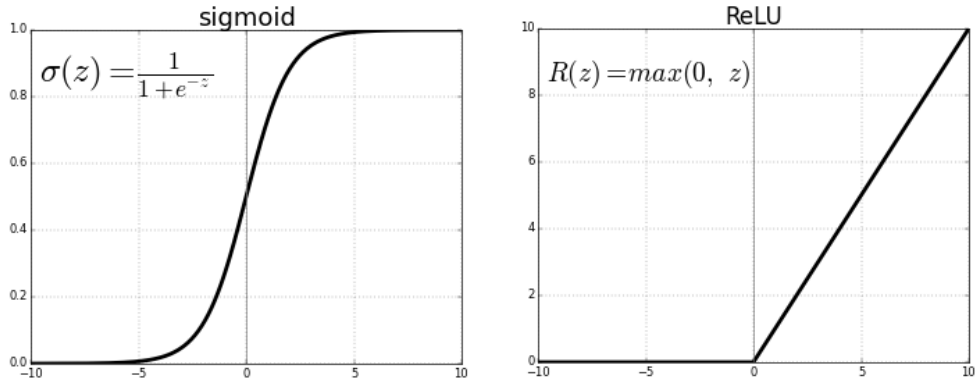

Figura 21. Funciones de activación sigmoide y ReLU [fuente]

Aunque la idea del perceptrón fue revolucionaria, tenía una gran limitación, solo podía resolver problemas cuyos datos fueran linealmente separables. Como se puede ver en la Figura 22, las dos gráficas de la derecha no son linealmente separables, y por tanto no pueden ser resueltos con un único perceptrón.

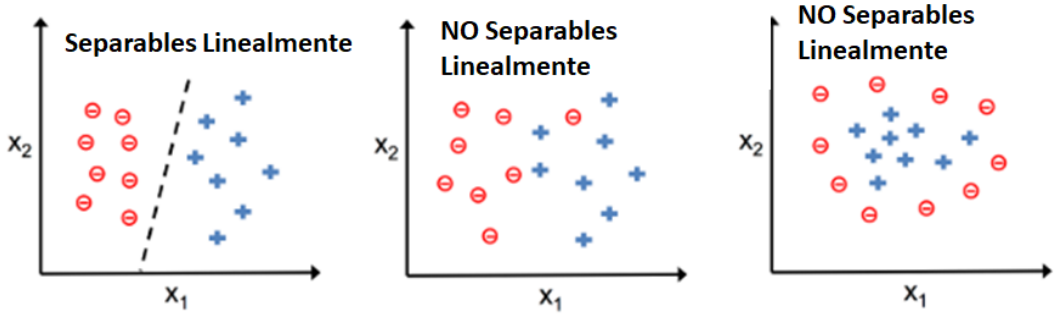

Figura 22. Diferencia gráfica entre un problema linealmente separable y otro que no lo es [fuente]

Sin embargo, la potencia del perceptrón no reside en realizar tareas de manera individual sino en combinarse con otros perceptrones, al igual que hacen las neuronas, para poder resolver problemas más complejos. En 1969 se publicó la idea del perceptrón multicapa [10], que consiste en combinar perceptrones por capas, obteniendo así una red que pueda resolver problemas no linealmente separables. De esta forma nació lo que conocemos como redes neuronales artificiales, en las cuales se puede distinguir tres tipos de capas:

- Capa de entrada: Punto de entrada a la red, recibe los datos en crudo, no los recibe de otras neuronas.
- Capas ocultas: Capas que toman como entrada la salida de su capa anterior y su salida se convierte en la entrada de la siguiente capa.
- Capa de salida: Nos devuelve el resultado de la red.

Todas estas capas quedan ilustradas en la Figura 23:

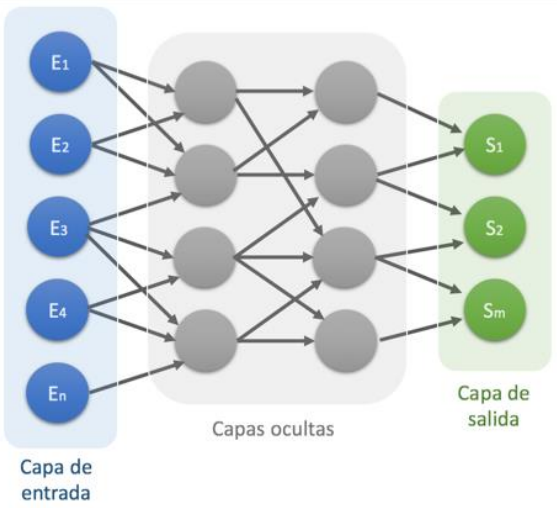

Figura 23. Tipos de capa en una red neuronal [fuente]

Para que la red neuronal aprenda a resolver un problema debe ajustarse a una función de error, intentando minimizarla todo lo posible. El método más extendido de minimización del error en las redes neuronales se conoce como *gradient descent*.

El *gradient descent,* conocido en español como descenso por gradiente, es un algoritmo de optimización, que permite converger hacia el mínimo de una función mediante un proceso iterativo. Durante el aprendizaje, se mide el error que cometemos en las predicciones y modificamos los parámetros del modelo con el objetivo de reducir este error. Esta función de error se le conoce como función de coste, se denota de la siguiente forma $J(\theta)$, donde θ representa los parámetros del modelo. Las funciones de costes más comunes son el error cuadrático medio (MSE) y el *cross entropy*.

El MSE se suele utilizar en problemas de regresión. El cálculo que realiza se puede ver en la Ecuación 3, donde y es la salida esperada e $\hat{y}$ es la predicción de la red. El valor n hace referencia al número de elementos dentro del dataset.

$$MSE = \frac{1}{n}\sum(y - \hat{y})^2 \qquad (3)$$

Por otro lado, el *cross entropy* se suele utilizar en problemas de clasificación. En este caso el valor $\hat{y}$ representa la probabilidad de que el ejemplo pertenezca a una clase. La Ecuación 4 muestra la ecuación del cross entropy.

$$-(y * \log(\hat{y}) + (1 - y) * \log(1 - \hat{y})) \qquad (4)$$

Cuando tenemos solo una neurona la función de coste es convexa y derivable, lo que implica que aplicando algoritmos de optimización como el descenso por gradiente es posible llegar a un mínimo global, alcanzando así la solución óptima. Sin embargo, los algoritmos de *Deep Learning* suelen usar redes neuronales, no una solo neurona, lo que multiplica exponencialmente la cantidad de posibles soluciones, por lo que es típico caer en mínimos locales, que, sin ser la solución óptima, suelen dar un buen resultado. En la Figura 24 vemos la diferencia gráfica entre un mínimo local y uno global.

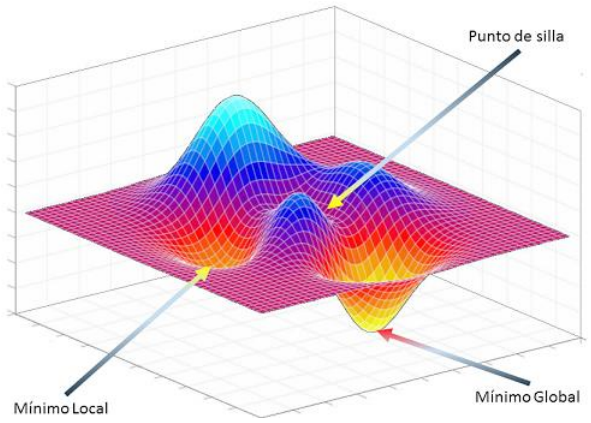

Figura 24. Diferencia gráfica entre el mínimo local y el mínimo global [fuente]

En función del número de muestras que introduzcamos en la red en cada iteración, existen diferentes versiones del *gradient descent* [11]:

- **Descenso por gradiente en lotes** (*batch*): Se introducen todos los datos disponibles a la red, actualizando el gradiente solo cuando se han evaluado todos los datos. Una de las ventajas de este método es su eficiencia computacional, con un error y convergencia estable. En cuanto a sus desventajas, al usar todos los datos a la vez, llegará un punto en el que las variaciones del gradiente sean mínimas, estancándose y evitando conseguir mejores resultados. Otro de los puntos negativos es el alto consumo de memoria al procesar todos los ejemplos al mismo tiempo.

- **Descenso por gradiente estocástico:** La idea contraria al descenso por gradiente por lotes, en cada iteración, en vez de introducir todos los datos introducimos solo uno. El gradiente se actualizará por cada muestra, consiguiendo así más aleatoriedad dificultando el estancamiento. El principal problema es la lentitud ya que se necesitan muchas más iteraciones y no se aprovecha la totalidad de los datos.

- **Descenso por gradiente en mini-lotes (*mini-batch*):** En vez de introducir una única muestra a la red, se introducen N muestras por iteración. De esta forma conseguimos cierta aleatoriedad al no usar todos los ejemplos por iteración, dificultando el estancamiento. Es más eficiente en memoria, y no se necesitan tantas iteraciones al introducir más de un ejemplo a la red, consiguiendo entrenamientos más rápidos. Esta

modificación del algoritmo será el utilizado en este trabajo. Se elegirá un valor N que nos proporcione un buen balance entre tiempo de entrenamiento y aleatoriedad. Este valor estará muy limitado por la cantidad de memoria asociada a la GPU. En la Figura 25 se muestra la representación gráfica de la convergencia de las diferentes versiones del descenso por gradiente.

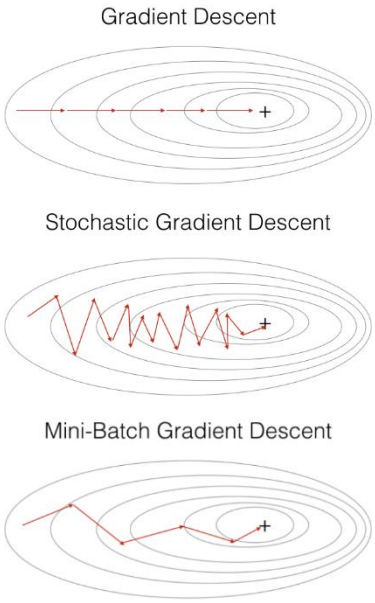

Figura 25. Representación gráfica del funcionamiento de las variantes del descenso por gradiente [fuente]

Como ya hemos mencionado, en este trabajo utilizaremos el descenso por gradiente en mini-lotes, las operaciones que se realizan a lo largo de una iteración son las siguientes:

1. Introducimos un mini-lote con N muestras de nuestro conjunto de entrenamiento. (las muestras han sido previamente etiquetadas)

2. Se realizan los cálculos necesarios a lo largo de la red de izquierda a derecha, obteniendo como salida la predicción de la red. A este proceso se le conoce como *forward propagation* (propagación hacia delante)

3. Se evalúa la función de coste para ese mini-lote, midiendo así las diferencias entre las predicciones reales y las predicciones de la red. Este error es el que tratamos de minimizar.

4. Para minimizar el error obtenido en el paso anterior, propagamos de atrás hacia adelante el error, calculando los gradientes de todos los parámetros de la red como la derivada de nuestra función de coste respecto a todos ellos. Este proceso se conoce como *backpropagation,* y matemáticamente hablando lo que nos proporciona es un vector cuya dirección y sentido indica hacia donde la función aumenta más rápido. Como lo que queremos es minimizar la función de coste,

tenemos que movernos en sentido contrario para conseguir nuestro objetivo. Con este proceso modificamos los pesos de las neuronas consiguiendo una parametrización con un error menor.

Este proceso se repetirá iterativamente tratando de alcanzar el mínimo global de la función de coste. Normalmente, a este proceso iterativo se le establece un número limitado de épocas para asegurar su finalización. Una época (*epoch* en inglés) es cuando todos los ejemplos de nuestro conjunto de entrenamiento han pasado por la red. El número de épocas dependerá del problema. La idea es elegir un número de épocas lo suficientemente alto para que al modelo le dé tiempo a aprender, pero a su vez, un número de épocas que no sea exageradamente elevado, ya que esto provocaría dos inconvenientes:

- El tiempo de entrenamiento sería muy grande, teniendo que esperar incluso días para entrenar un modelo.
- El modelo intentaría ajustarse tanto a los datos de entrenamiento que acabaría sobreentrenando. Es decir, podría alcanzar un error muy pequeño sobre los datos de entrenamiento a costa de no generalizar bien sobre otros datos nunca vistos, como los de test.

En la Figura 26 podemos observar un ejemplo gráfico de lo comentado anteriormente. En la gráfica podemos ver el error sobre los conjuntos de train y test a lo largo de las épocas. Con muy pocas épocas, el modelo no ha entrenado lo suficiente y por ello el error en ambos conjuntos es alto. Con muchísimas épocas, el error de train es muy bajo, pero en test es muy alto debido al sobreentrenamiento. La clave está en encontrar un punto medio donde el modelo aprenda bien pero no sobreentrene.

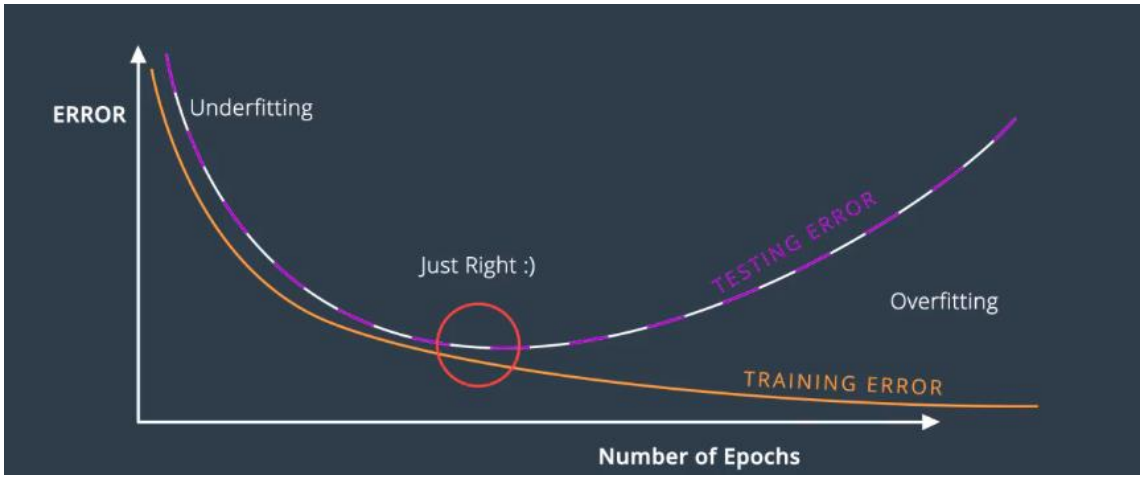

Figura 26. Ejemplo gráfico de la evolución del error en train y test al largo de las épocas. [fuente]

Otro parámetro importante en este caso para evitar caer en mínimos locales es el *Learning Rate* (LR) [12], el cual controla la magnitud de la actualización de los pesos de cada iteración.

Por un lado, el valor del LR debe evitar un ritmo de aprendizaje muy pequeño, ya que esto puede provocar una convergencia muy lenta y que caiga en mínimos locales. Por otro lado, también se debe evitar el extremo contrario, un ritmo de aprendizaje muy alto que conduzca a inestabilidad

en la función de coste, evitando así la convergencia del algoritmo al estar dando saltos grandes entorno al mínimo. El valor del LR suele estar comprendido en el rango [0.001 – 0.5], aunque puede ser mayor o menor que estos valores dependiendo del problema. En la Figura 27 podemos ver como el valor del LR afecta a la convergencia del descenso por gradiente.

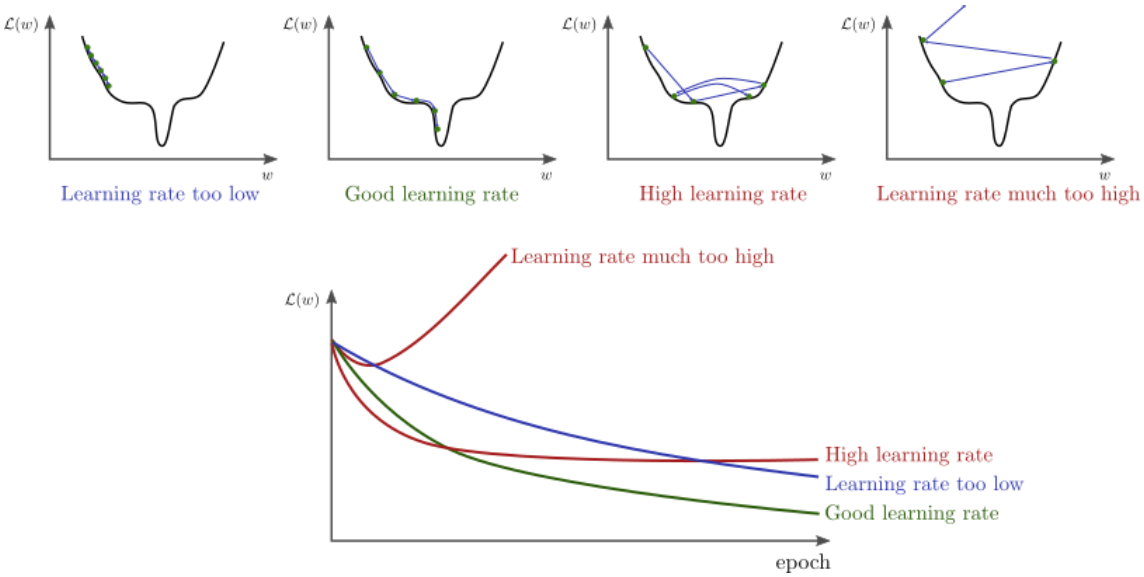

Figura 27. Representación gráfica de la convergencia en función del learning rate [fuente]

El LR puede ser fijo, o variar con el tiempo. Cuantas más épocas transcurran más probable es que estemos cerca del mínimo global, y por ello no necesitamos hacer saltos tan grandes, por lo que se podría reducir a lo largo de las épocas.

Otro aspecto importante a tener en cuenta a la hora de entrenar un modelo es la normalización. La normalización de las características es un paso indispensable a la hora de entrenar cualquier modelo. La importancia de la normalización queda claramente ilustrada en la Figura 28. Si una característica se mueve en un rango muy amplio, pero otra se mueve en un rango mucho más reducido, la actualización del gradiente para alcanzar un mínimo estará desestabilizada. La característica que tiene un mayor rango tendrá mayor influencia que la característica con menor rango.

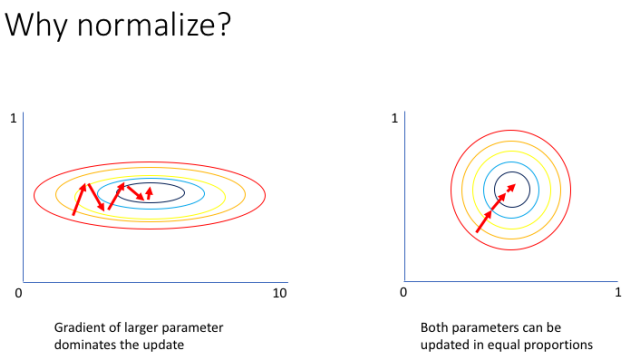

Figura 28. Descenso por gradiente en función de si las características están normalizadas. [fuente]

A pesar del uso de técnicas para mejorar la convergencia, entrenar modelos sobre grandes cantidades de datos es muy costoso, tanto desde el punto de vista computacional como temporal. Esto ha sido el principal factor limitante a la hora de desarrollar técnicas centradas en la inteligencia artificial. Como hemos visto en esta misma sección, el concepto de neurona artificial apareció a mediados del siglo pasado, sin embargo, debemos esperar hasta principios de los dos miles para que se empezarán a normalizar el desarrollo y utilización de técnicas centradas en la inteligencia artificial.

La lentitud con la que se avanzó en este campo durante esos años, así como la explosión que ha sufrido la inteligencia artificial estos últimos 10 años se debe principalmente a los grandes avances en el campo del hardware, sobre todo a la aparición de Unidades de Procesamiento gráfico (GPU) que han permitido entrenar modelos en tiempos mucho más reducidos que si se lanzarán sobre CPU. Junto a esto, han ido surgiendo muchos *frameworks* como TensorFlow [13] o Pytorch [14] que facilitan enormemente la creación y entrenamiento de modelos. Es resumen, una gran mejora a nivel de hardware, junto al abaratamiento de este tanto a nivel a de memoria como de GPU, sumado a la aparición de librerías que facilitan enormemente el desarrollo de modelos, han dado lugar a la gran popularización que ha experimentado este campo en los últimos años.

2.3.2.Redes neuronales Convolucionales

Las redes neuronales convolucionales (CNN) son las más utilizadas en problemas de visión, donde el dato a tratar son imágenes. Se trata de redes profundas en las que las capas ocultas de la red no tienen por qué estar completamente conectadas con las capas adyacentes, sino que las capas están desarrolladas para que realizar operaciones diferentes, conocidas como convoluciones.

Como ya se ha comentado en la *Sección 2.1.3* hay grandes diferencias entre las nubes de puntos y las imágenes, sin embargo, ciertas técnicas para el tratamiento de imágenes en redes neuronales se han extrapolado a modelos de *Deep Learning* que trabajan con nubes de puntos. Como veremos en la *Sección 4*, las CNNs son la base para algunos de estos modelos, por ello se ha decidido hacer una breve explicación sobre las características de este tipo de redes en esta sección.

Las CNNs están formadas por múltiples capas de filtros convolucionales de una o más dimensiones. Estas redes suelen contar con una primera fase de extracción de características a partir de los datos de entrada. Esta fase de extracción de características consiste en alternar neuronas convolucionales con neuronas de reducción de muestreo (pooling). A lo largo de la fase de *pooling* los datos disminuyen su dimensionalidad, consiguiendo así que las neuronas de capas lejanas sean menos sensibles a perturbaciones de los datos de entrada, pero al mismo tiempo siendo activadas por características más complejas.

Todo este proceso se ilustra de manera gráfica en la Figura 29. Las primeras capas extraen características de bajo nivel como bordes o texturas. En las capas ocultas intermedias se extraen formas más complejas como pueden ser ojos o bocas. En las capas finales, la CNN es capaz de extraer características de alto nivel como pueden ser los rostros, los objetos originales representados en las imágenes de entrada.

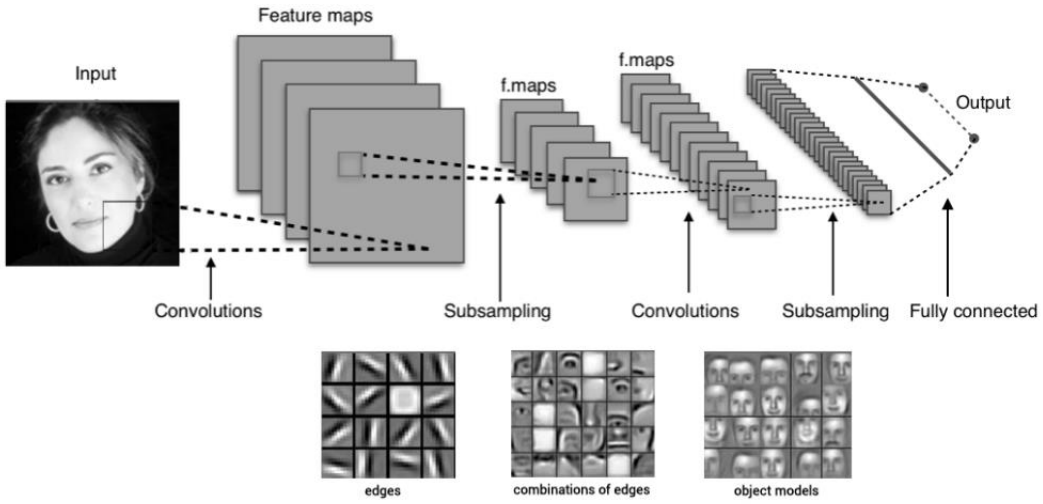

Figura 29. Extracción de características en una CNN [fuente]

En las dos siguientes secciones se explican los dos tipos de capas que se utilizan dentro de las CNN, las capas convolucionales que permiten extraer características y las capas de pooling que sirven para reducir la dimensionalidad.

2.3.2.1.Capas convolucionales

Estas capas, mediante una operación de convolución, son las que permiten a la red extraer características. Una operación de convolución es una operación matemática que transforma dos funciones (f, g) dadas en una tercera función que de alguna forma representa la magnitud en la que se superponen f y una versión trasladada e invertida de g. En la Figura 30 podemos ver de manera gráfica esta operación de $f * g$. En esta Figura podemos ver como la convolución de dos pulsos cuadrados da como convolución un pulso triangular. La función f está representada en color rojo, mientras que g está representada en color verde. La función resultante se representa debajo, conforme pasa el tiempo la convolución se va completando y esta función se va formando.

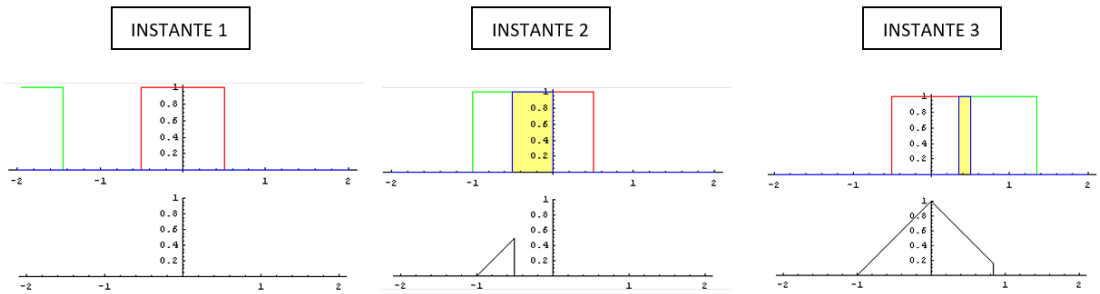

Figura 30. Representación gráfica de la operación matemática de convolución. [fuente]

En nuestro caso, tenemos que convolucionar imágenes, funciones discretas de dos dimensiones, donde f corresponde a la imagen de entrada, y la función g corresponde a una pequeña matriz denotada como *kernel* (filtro) que trasladaremos por todo el dominio de f. Esta operación mide la relación existente entra cada píxel de f y g.

Desde un punto de vista formal: Sea f una imagen de $M\ x\ N$ píxeles, y g un filtro de $m\ x\ n$, la convolución de f y g, denotada como $f * g$, es una nueva imagen h, cuyo valor viene definido por la Ecuación 5. Además, en la Figura 31 se representa visualmente esta operación.

$$h(x,y) = g * f(x,y) = \sum_{i=-m}^{m} \sum_{j=-n}^{n} g(i,j) \cdot f(x-i, y-j) \qquad (5)$$

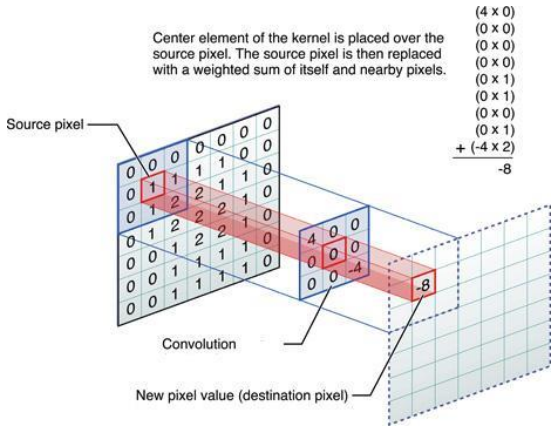

Figura 31. Representación gráfica de la convolución en imágenes, en este caso g es un kernel de 3 x 3.

En este ejemplo hemos tratado con matrices de dos dimensiones, típico en imágenes en escala de grises, pero también se pueden aplicar estas operaciones en matrices en color, *RGB,* las cuales tienen 3 canales. Si denotamos en número de canales que tiene una matriz como n_c, si nuestra matriz de entrada tiene n_c canales, los filtros que queramos aplicar sobre ella deberán tener el mismo número de canales, teniendo una dimensión de $m\ x\ n\ x\ n_c$.

Una vez que tenemos clara la operación convolucional sobre imágenes, tres términos comunes para definir los parámetros de esta operación son:

- ***Kernel size***: Tamaño del filtro a aplicar sobre la imagen de entrada. Dimensionalidad del *Kernel* (filtro).

- ***Stride***: Indica el número de pixeles que nos desplazamos para aplicar otra vez el filtro. Suele tener un valor de 1 por defecto, si su valor es más grande la imagen resultante tendrá una dimensión más reducida que la imagen original.

- **Padding:** Define como se deben tratar los bordes de la imagen a la hora de aplicar los filtros. Se suele añadir información en los bordes para poder mantener la máxima información posible a lo largo de la red.

En la Figura 32 aplicamos la operación de convolucion con un *Kernel* de 2 x 2, con un *stride* de 1 y un *padding* de 1 (columnas y filas grises sobre la imagen de entrada).

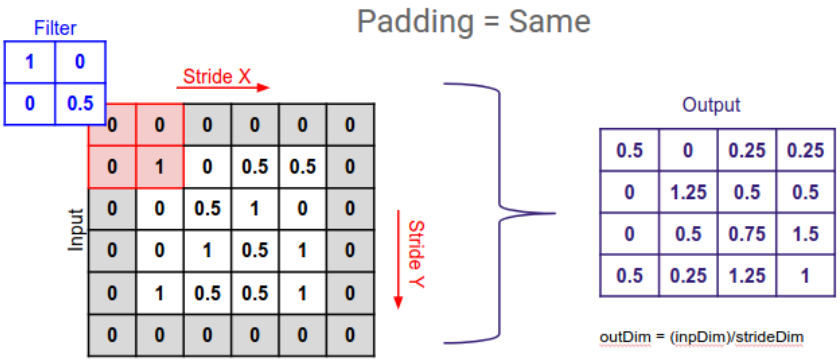

Figura 32. Operación convolucional usando padding y stride [fuente]

Una vez explicados conceptos como el *kernel*, el *stride* y el *padding* es necesario saber cómo afectan estos a la dimensionalidad de la imagen una vez aplicada la convolución. Dada una imagen de entrada de $M\ x\ N$, un *kernel* de $m\ x\ n$, un *padding* p y un *stride* s, el tamaño de la imagen resultante después de realizar la convolución está reflejada en la Ecuación 6:

$$\left\lfloor \frac{M + 2p - m}{s}, \frac{N + 2p - n}{s} \right\rfloor \qquad (6)$$

Para finalizar con este apartado, recordar que las capas convolucionales no dejan de ser neuronas, por lo que internamente se sigue aplicando funciones de activación, siendo ReLU la función más común. Debemos hacer notar que en las redes neuronales *fully-connected,* cada elemento de una capa está conectada con todas las salidas de la capa anterior mientras que en las capas convolucionales cada elemento solo está conectado a aquellos elementos necesarios para calcular la convolución.

2.3.2.2.Capas de Pooling

La funcionalidad de estas capas consiste en reducir la dimensionalidad espacial de la imagen, tanto el alto como el ancho, y suelen ir después de las capas convolucionales para proporcionar una entrada más pequeña a la siguiente capa convolucional. **Esta reducción no afecta a la profundidad.** Al reducir la dimensionalidad perdemos información, lo que en un inicio puede parecer algo negativo, sin embargo, está pérdida suele ser beneficiosa por 2 motivos:

- A menor tamaño, menos información y por tanto menor coste en el cálculo de las convoluciones para las capas más profundas de la red

- Ayuda a evitar el sobreaprendizaje de los modelos

Estas capas se definen por tres parámetros:

- El tamaño del *kernel*
- El *stride*
- La operación a aplicar, la más típica es el *max-pooling*, que selecciona el máximo de los valores que caen dentro de ella.

En la Figura 33 podemos apreciar un ejemplo visual. En este caso el *kernel* es de 2x2, el *stride* de 2 y la operación que aplica es el máximo. Como se puede ver, la imagen original era de 4 x 4, mientras que la resultante es de 2 x 2, reduciendo la dimensionalidad en gran medida.

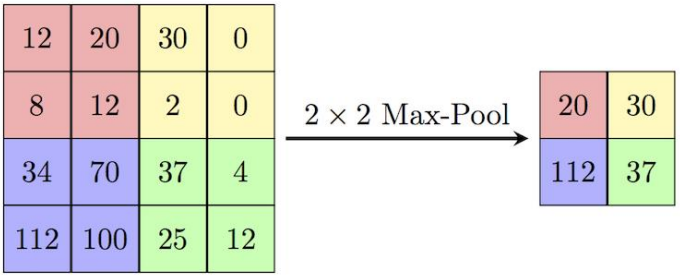

Figura 33. Ejemplo de max-pooling con una dimensión de 2x 2 con un stride de 2 [fuente]

Es importante remarcar que en una CNN no existen solo capas convolucionales o de pooling sino que también podemos encontrar capas *fully-connected* o capas de normalización, las cuales se suelen colocar después de una capa convolucional, y operan sobre cada uno de los canales normalizándolos en base a diferentes criterios.

2.3.2.3.Ejemplo de arquitectura CNN, VGG16

Una vez que hemos explicado los dos tipos de capas que forman la base para las CNN presentamos un ejemplo de arquitectura, en este caso la VGG16 [15], una de las redes neuronales convolucionales más famosas, centrada en la clasificación y detección de objetos. En la Figura 34 encontramos un diagrama de su arquitectura. Esta red, bastante profunda para su época, alterna tanto capas convolucionales (para extraer características) como capas de pooling (para reducir la dimensionalidad) finalizando con unas capas fully-connected para devolver la clase del objeto que aparece en la imagen. En este caso, esta red fue entrenada para clasificar 1000 objetos diferentes, por ello, el resultado final es un vector con las probabilidades de que el objeto de la imagen pertenezca a cada una de las posibles clases. La clase elegida como predicción es aquella que tiene la probabilidad más alta.

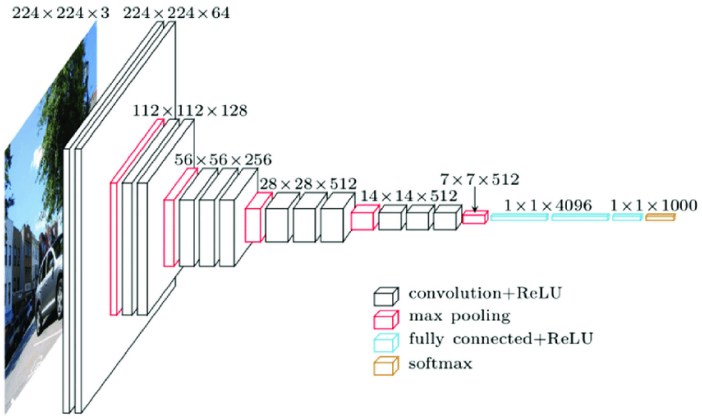

Figura 34. Arquitectura de la CNN VGG16 [fuente]

Como se comenta al principio de la *Sección 2.3.2*, las CNNs son la base para algunos modelos de *Deep Learning* centrados en la segmentación de nubes de puntos. Al mismo tiempo, sus arquitecturas se asemejan bastante a las que podemos encontrar en modelos de segmentación de imagen donde lo que se busca es predecir una clase por píxel, en vez de por cada punto. Conocer de manera superficial las CNNs facilitará la comprensión de estos modelos más complejos.

2.4.Tecnologías

Para alcanzar el objetivo de este proyecto, segmentar nubes de puntos tomadas mediante LiDAR aéreo, es indispensable disponer de herramientas que nos permitan:

- Visualizar los datos

- Trabajar con ficheros LAS (formato en el que se almacenan los datos LiDAR)

- Librerías que faciliten la construcción de modelos de *Deep Learning* centrados en la segmentación de nubes de puntos

En esta sección hablaremos de las tecnologías en las que nos hemos apoyado para desarrollar este proyecto.

2.4.1.Visores

A la hora de trabajar con datos 3D, en nuestro caso nubes de puntos tomadas con un LiDAR aéreo, es indispensable disponer de un visor que nos permita visualizar nuestros datos. Este visor lo usaremos tanto para visualizar las características de los datos originales, así como para validar visualmente nuestras predicciones. En este trabajo se han utilizado dos visores, el CloudCompare [16] y Potree [17], ambos visores son de código abierto que nos permite trabajar con datos LAS. Un archivo LAS es un formato binario estándar de la industria para almacenar datos LIDAR aéreos.

El CloudCompare es un visor fácil de instalar y utilizar, el cual nos permite realizar diversas acciones muy útiles para visualizar y entender nuestros datos, entre ellas:

- Visualizar las diferentes características que nos devuelve el LiDAR sobre la nube de puntos, como la Intensidad o los rebotes.
- Filtrar los datos en función de un umbral definido por el usuario sobre cualquier característica y visualizar el resultado.
- Desplazarnos con soltura y fluidez en el espacio 3D sobre el cual se encuentra la nube de puntos.

En la Figura 35 podemos ver un fichero LAS cargado en el visor, un bloque de 1km x 1km de aproximadamente 50 millones de puntos, que delimita una zona perteneciente al barrio de Barañain, en ella se muestra el RGB por cada punto.

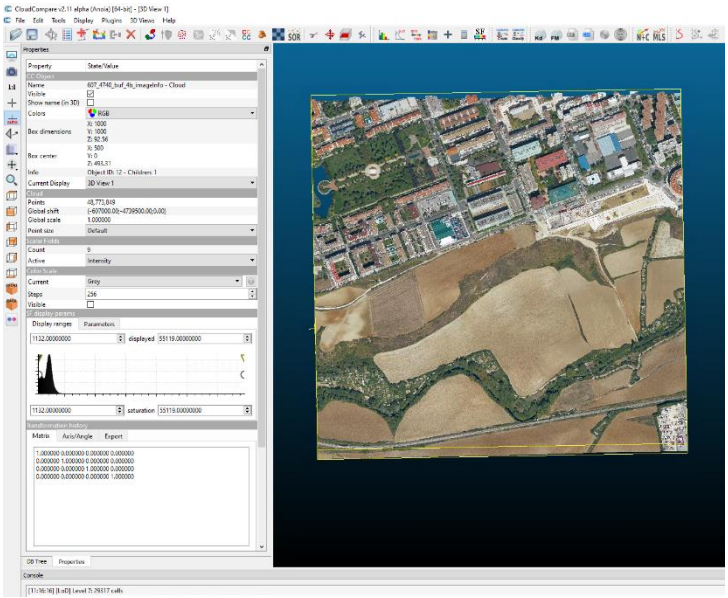

Figura 35. Fichero LAS visualizado en el CloudCompare

Por otro lado, el visor Potree nos proporciona funcionalidades similares al CloudCompare, añadiendo alguna funcionalidad extra como la posibilidad de hacer perfiles. En la Figura 36 se expone un ejemplo en el que se realiza un perfil, donde la línea roja sobre el LAS define la zona donde se desea realizar el perfil y en la gráfica de la parte inferior es donde se visualiza. Esta funcionalidad es muy útil para observar en detalle la calidad del suelo predicho por nuestro algoritmo.

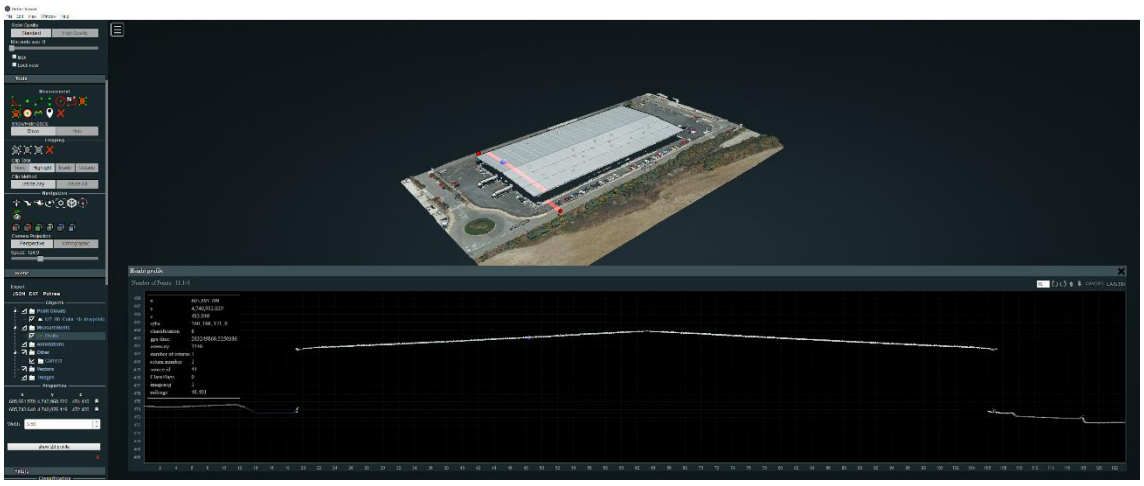

Figura 36. Fichero LAS visualizado sobre Potree en la que realizamos un perfil

2.4.2.Lenguaje de Programación

Una de las herramientas indispensables para el desarrollo de este proyecto ha sido Python. Se ha elegido este lenguaje de programación por varios motivos:

- En Python programar tareas suele ser más rápido y requiere menos líneas de código que en otros lenguajes.

- Python posee una comunidad muy activa, lo que se traduce en una gran cantidad de librerías en las cuales apoyarnos para avanzar más rápido en nuestros objetivos.

- Uno de los lenguajes que admite Jupyter Notebook es Python. Jupyter Notebook [18] es un entorno interactivo que permite desarrollar código de manera dinámica e incluir en un mismo documento fragmentos con código, texto o imágenes. Es una funcionalidad muy utilizado en campos relacionados con las matemáticas y la informática.

2.4.3.Librerías

Como se ha comentado en el apartado anterior, uno de los motivos por los cuales se ha elegido Python como lenguaje de programación, ha sido la gran variedad de librería que dispone para *Machine Learning* o para procesamiento de datos LiDAR. Algunas de las librerías que hemos empleado en este proyecto han sido:

- PDAL [19]: Las PDAL (Point Data Abstraction Library) es una librería open source desarrollada en C++ centrada en la manipulación de nubes de puntos. No está limitada solo a datos LiDAR aunque la motivación original de esta librería fue el tratamiento de este tipo de dato. La idea de las PDAL es definir operaciones sobre nubes de puntos a través de *pipelines,* las cuales se pueden definir en formato JSON, o utilizando la API disponible. Las PDAL nos permite trabajar con múltiples formatos, desde TXT a LAS, sobre los cuales podemos realizar acciones de lectura y escritura, así como aplicar gran variedad de filtros ya implementado, desde detección de outliers a filtrar puntos por un valor definido por el usuario. Además, esta librería proporciona un *wrapper* para Python, lo que facilita mucho su uso.

- Pytorch[14]: Pytorch es una librería centrada en el *Machine Learning*, de código abierto, basada en la librería Torch. Está programado principalmente en C++, CUDA y Python. Pytorch nos permite realizar la computación de tensores (vectores numéricos de n dimensiones) con una fuerte aceleración al poder realizar estas operaciones sobre GPU (aunque también permite ejecutarlos sobre CPU). Además, esta librería tiene implementados métodos que aceleran y facilitan la creación y experimentación sobre modelos de *Deep Learning*.

- Pytorch Geometric[20]: Pytorch Geometric es una extensión de la librería Pytorch, que consiste en varios métodos para aplicar *Deep Learning* sobre datos 3D, desde *meshes* a nubes de puntos.

- Pytorch Points 3d[21]: Framework basado en Pytorch Geometric que engloba los últimos modelos de *Deep Learning* que realizan tareas de clasificación o segmentación sobre nubes de puntos. A más alto nivel que Pytorch Geometric, facilitando mucho la construcción y uso de modelos.

En la Figura 37 podemos ver los logos de las principales tecnologías usadas en este proyecto.

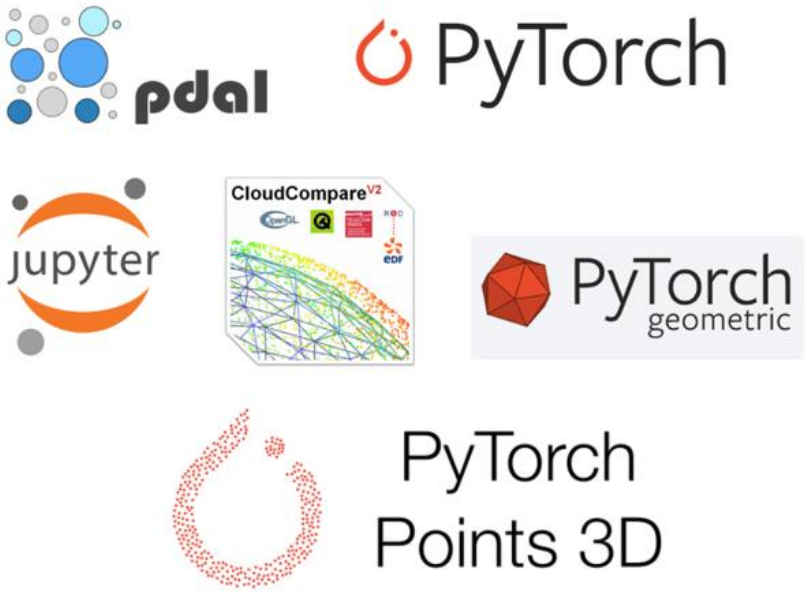

Figura 37. Logo de las principales tecnologías utilizadas en este proyecto

3.Datos LiDAR del proyecto

En esta sección se pretende realizar un análisis de los datos aéreos LiDAR con los que trabajamos a lo largo del proyecto. Analizaremos la zona de estudio inicial, cuya geolocalización es la cuenca de Pamplona, su densidad, la variabilidad de las zonas contenidas dentro de la zona final de estudio, así como la información original contenida en los datos LiDAR. Otro punto importante que se tratará en esta sección son las clases que se pretenden predecir con los modelos *de Deep Learning* y algunas claves sobre cómo se ha realizado la anotación manual de los datos. Para finalizar, se analizará los dataset base de Train / Test necesarios para el aprendizaje de nuestros modelos.

3.1.Zona inicial de estudio, densidad y variabilidad

La extracción aérea de datos LiDAR se ha llevado a cabo sobre Navarra. Más concretamente sobre la zona que rodea la cuenca de pamplona. En la Figura 38 podemos observar la zona de estudio dentro del contorno de la provincia de Navarra, mientras que en la Figura 38 podemos observar la zona de estudio con más detalle, la cual está dividida en bloques de 1 km x 1km, obteniendo así unos 250 bloques.

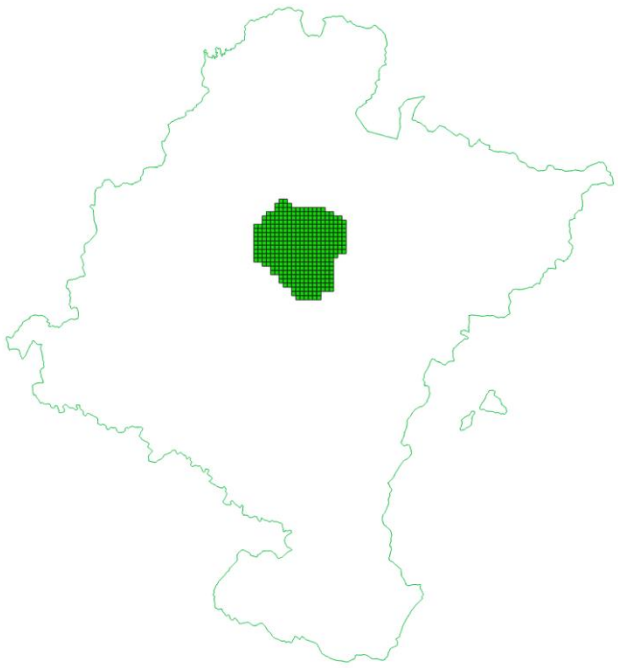

Figura 38. Zona de estudio dentro del contorno de Navarra

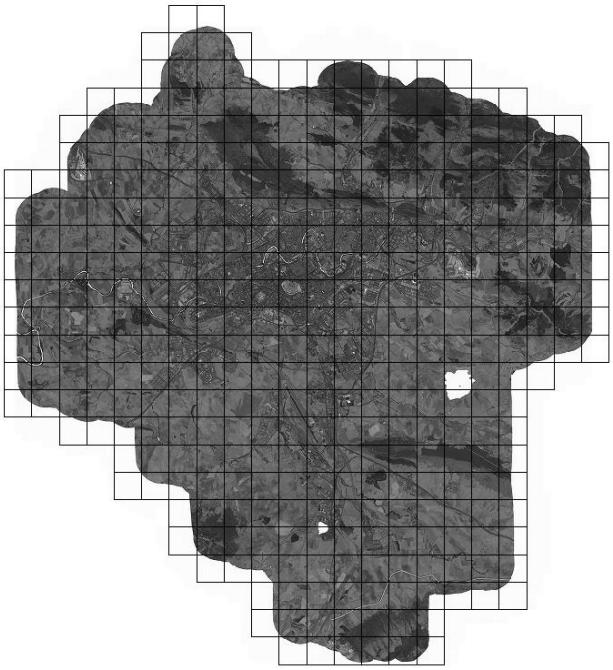

Figura 39. Zona de estudio más detallada, dividida en bloques de 1km x 1km

Cuando el avión se desplaza por el aire el sensor LiDAR recoge la información de la zona que está sobrevolando. Esta trayectoria en la que se recogen datos se le denomina pasada. Sin embargo, el avión no pasa por la misma zona una única vez, sino que puede recorrer la misma zona varias veces, o que haya cierto solape entre pasadas diferentes. Debido a esto la densidad de los datos se consigue solapando pasadas que tienen originalmente una densidad de 5 puntos por metro cuadrado. Las pasadas realizadas por el avión sobre la zona de estudio se pueden ver en la Figura 40. Salta a la vista que hay zonas con un mayor solape transversal, donde se solapan varias pasadas. En la Figura 41 se marcan algunos bloques donde este solape es bastante alto, llegando a alcanzar una densidad de 50 puntos por metro cuadrado. Sin embargo, en otras zonas más próximas a los bordes de la zona de estudio, la densidad es mucho más reducida, oscilando entre los 5 y los 15 puntos por metro cuadrado.

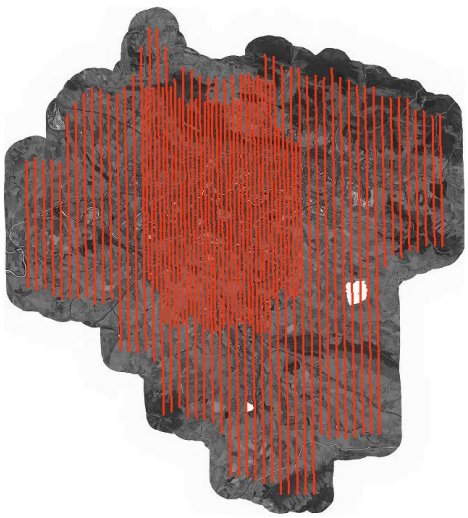

Figura 40. Pasadas realizadas por el avión dentro de la zona de estudio

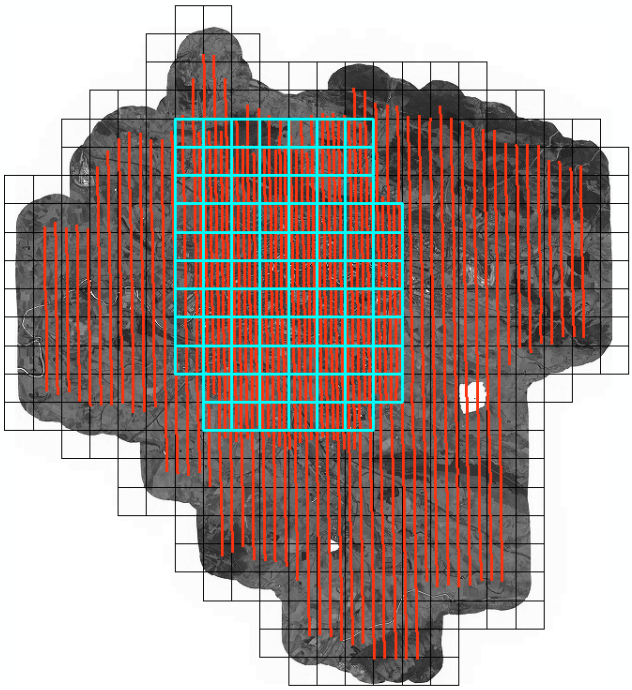

Figura 41. Bloques muy densos dentro de la zona de estudio.

Que la densidad se consiga solapando pasadas de 5 puntos por metro cuadrado no implica solo diferencias de densidades a nivel de bloque, sino que la densidad dentro de cada bloque tampoco es homogénea, habiendo zonas dentro de un mismo bloque más densas que otras. De los 250 bloques de la zona inicial de estudio, en este trabajo nos hemos centrado en un subconjunto de 124 bloques. Estos 124 bloques son aquellos con una densidad más alta y por ello más costosos de etiquetar manualmente. En la Figura 42 podemos ver el subconjunto de bloques con los que se va a trabajar, mientras que en la Figura 43 podemos observar esta misma zona en más detalle.

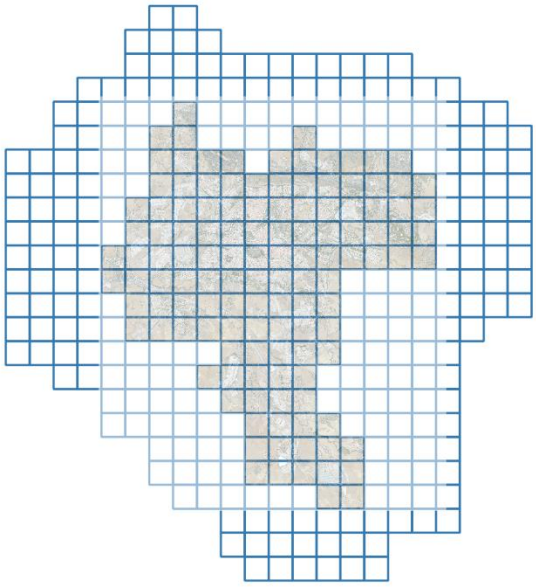

Figura 42. Subconjunto de bloques más densos dentro de la zona inicial de estudio

Figura 43. RGB sobre los bloques finales sobre los cuales se va a trabajar.

En cuanto a la densidad de los bloques que componen la zona final de estudio, en la Tabla 2 podemos ver la densidad media de los bloques, la desviación típica y la densidad del bloque más y menos denso.

Bloque menos denso	**18.18 puntos por metros cuadrado**
Bloque más denso	66.94 puntos por metros cuadrado
Densidad media por bloque	39.92 puntos por metros cuadrado
Desviación típica por bloque	12.35 puntos por metro cuadrado

Tabla 2. Información básica sobre la densidad de la zona final de estudio

El rango de densidades existentes en los bloques es amplio, siendo el bloque con mayor densidad 3.68 veces más denso que el bloque de menor densidad. La densidad media está entorno a los 40 puntos por metro cuadrado y la desviación típica es alta, de 12 puntos por metro cuadrado. Como podemos ver el histograma de la Figura 44, la repartición de las densidades de los bloques a lo largo del rango existente es equilibrando, exceptuando las densidades entorno a los 25 y a los 55 puntos por metro cuadrado, habiendo en estos valores una mayor concentración de bloques.

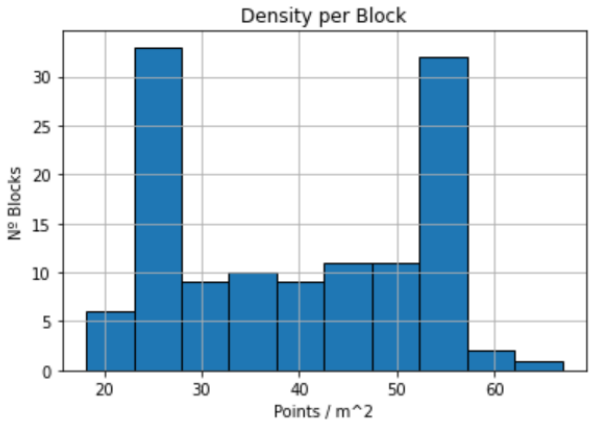

Figura 44, histograma de densidades por bloque

Dentro de la zona final de estudio, la cual está centrada en Pamplona, englobando así tanto la capital navarra como los barrios más próximos a esta, encontramos una gran variabilidad de zonas:

- **Zonas urbanas**: Zonas de alrededor de Pamplona que se caracterizan por tener una gran cantidad de edificios, normalmente bloques residenciales o edificios que albergan comercios u oficinas. No hay grandes masas de vegetación, y suele haber gran cantidad de vehículos aparcados o en movimiento. Podemos ver un ejemplo en la Figura 45.

- **Zonas industriales**: Parecidas a las zonas urbanas. Zonas más alejadas de Pamplona cuyos edificios son naves industriales de gran tamaño. Podemos ver un ejemplo en la Figura 46.

- **Zonas rurales:** Zonas cercanas a la frontera de la zona final de estudio. Corresponde a pequeños pueblos rodeados de campo. Los edificios suelen ser pequeñas casas esparcidas, habiendo una mayor cantidad de vegetación que en zonas urbanas o industriales pero una menor presencia de vehículos. Podemos ver un ejemplo en la Figura 47.

- **Zonas semirrurales:** Mezcla entre zona urbana y zona rural. Podemos ver un ejemplo en la Figura 48.

Todas las imágenes utilizadas para ilustrar los diferentes tipos de zonas corresponden a nubes de puntos sobre las cuales se ha proyectado el RGB y se ha hecho una captura desde arriba.

Figura 45. Ejemplo de zona urbana.

Figura 46. Ejemplo de zona industrial.

Figura 47. Ejemplo de zona rural.

Figura 48. Ejemplos de zona semirrural.

3.2.Características originales contenidas en los datos LiDAR del proyecto

Como ya se ha mencionado en la *Sección 2.4*, la información recogida con el sensor LiDAR se almacena en un fichero con la extensión "LAS". La versión de los ficheros LAS con la que hemos trabajado es la más actual, la versión 1.4 [22]. Estos ficheros contienen la nube de puntos almacenada, con las coordenadas XYZ y otras características por punto que se explicarán en este mismo apartado:

- **PointSourceID**: Representa el id de la pasada con la cual se ha capturado el punto.

- **ScanAngleRank**: Esta característica, con un rango de valores entre [-90, 90], representa el ángulo entre el punto y la posición del sensor colocado en la base del avión. Tiene un valor de 0 en el nadir (perpendicular al suelo), -90 grados a la izquierda del sentido de la dirección de vuelo y +90 hacia la derecha.

- **Scan Direction Flag**: Esta característica denota la dirección a la cual el scanner apuntaba al momento de emitir el pulso de salida. Solo puede adquirir los valores 0 ó 1. Su valor será positivo, es decir, 1, si el movimiento del scanner era de izquierda a derecha siguiendo la dirección y sentido del desplazamiento del avión. Su valor será negativo, es decir, 0, en caso contrario.

- **Return Number**: Es el número de retorno de un pulso, dado un pulso de salida. Un pulso de salida puede tener más de un pulso de retorno. Cada pulso de retorno está marcado secuencialmente, pudiendo ir desde 1 hasta 5 como máximo.

- **Number Of Returns**: Número total de retornos dado un pulso de salida. Puede que para un punto su *Return Number* sea 2, pero si ha habido 5 retornos para ese punto su *Number Of Returns* será de 5.

- **GpsTime**: Representa mediante un número decimal de 8 bytes el instante temporal donde el punto fue capturado.

- **Intensity**: Representación mediante un valor entero de la magnitud del pulso de retorno.

- **Classification**: Este campo representa la clase de cada punto, donde cada clase está representada con un valor entero. Este campo es el que se debe rellenar con la predicción final del modelo.

- **RGB**: Esta característica no está contenida originalmente en nuestros datos LiDAR, aunque los ficheros LAS tiene campos específicos para guardar esta información. Sin embargo, con ortofotos de las zonas de estudio tomadas con el mismo avión a la misma hora y utilizando un filtro de las PDAL que nos permite trasladar esta información de las imágenes a la nube de puntos, conseguimos generar esta característica.

Como se ha explicado, no es poca la cantidad de información que almacena el LiDAR por punto, sin embargo, se considera que no todas las características son útiles para el problema que pretendemos resolver. El *ReturnNumber* y el *NumberOfReturn* son muy interesantes ya que la vegetación suele tener un número más grande de rebotes comparado con otras clases como el suelo al haber hojas, ramas, etc. A su vez, la intensidad o el RGB también nos proporciona información útil, ya que con solo echar un vistazo a su representación visual podemos comprender los elementos presentes en la muestra. En la Figura 49 podemos ver de manera gráfica las características que consideramos útiles sobre una misma muestra. Mientras que otras características como el PointSourceID o el GpsTime no nos están proporcionando información que nos vaya a permitir segmentar los puntos, tanto por su definición como por su representación visual. En la Figura 50 podemos ver de manera gráfica las características que no consideramos útiles para el problema que pretendemos resolver. Como se puede ver son prácticamente ruido y no permiten reconocer los elementos que pretendemos clasificar.

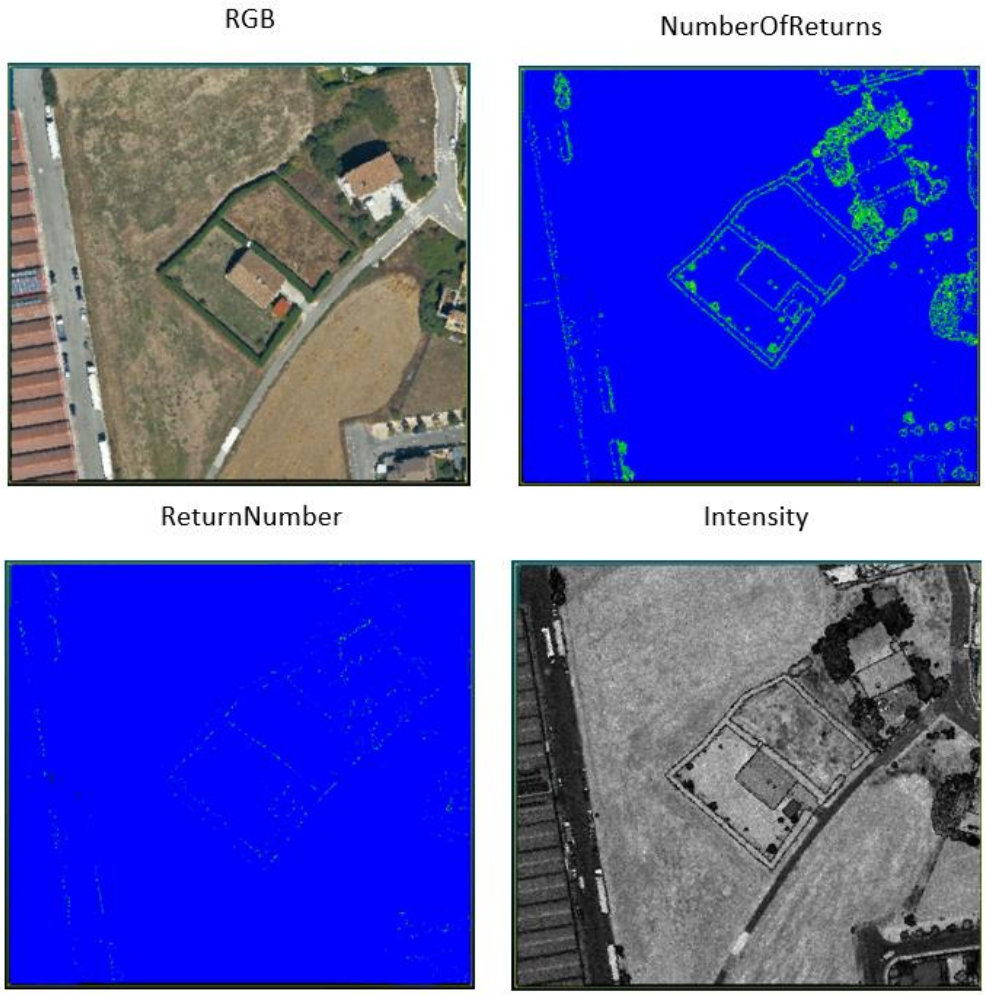

Figura 49. Representación visual de las características originales contenidas en nuestros datos LiDAR que se consideradas útiles.

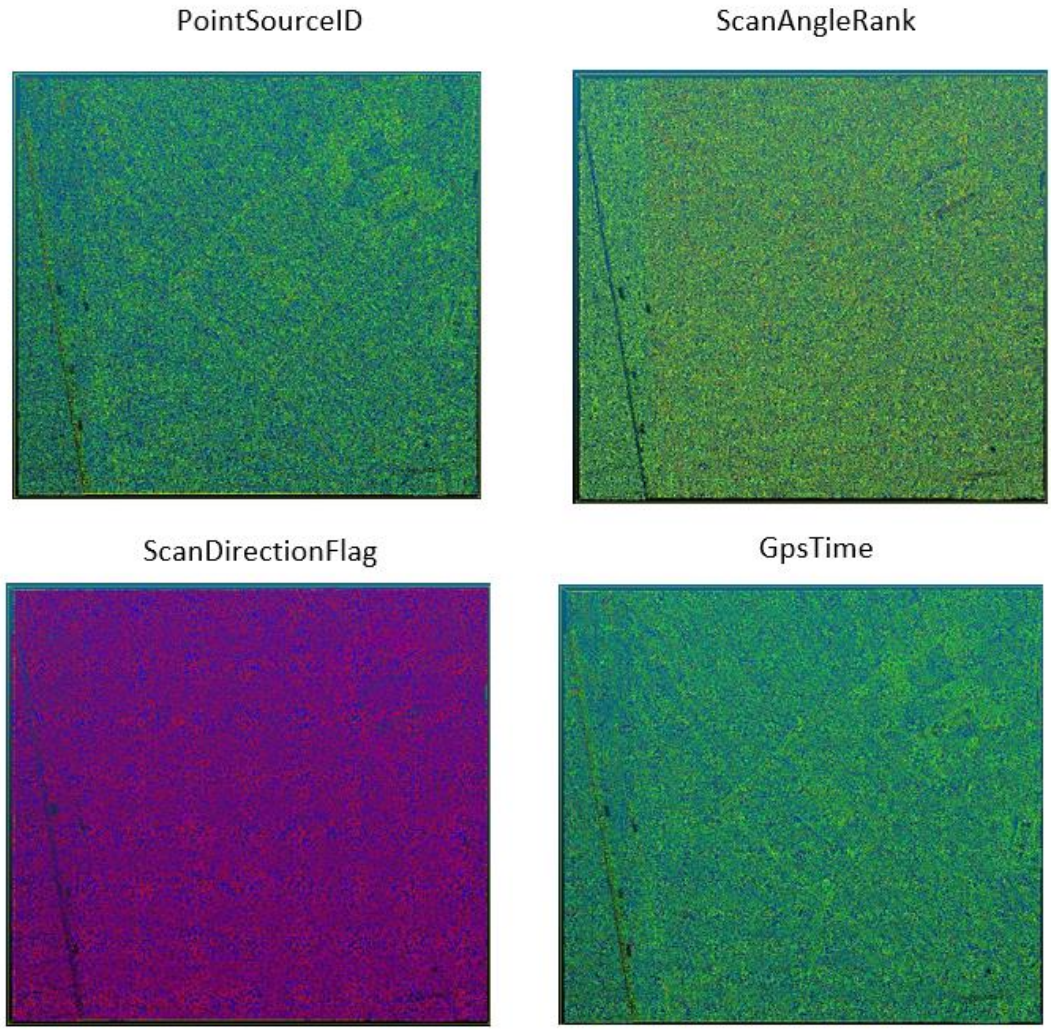

Figura 50. Representación visual de las características originales contenidas en nuestros datos LiDAR que no se consideradas útiles.

3.3.Definición de clases a predecir

En este apartado se va a definir las clases que se pretenden predecir de manera automática con nuestro modelo. Como se ha mencionado ya en la *Sección 3.2,* cada clase tendrá asociado un número entero que la representa. Las clases de estudio son:

- **Clase 2, Suelo:** Esta clase representa el suelo, clase fundamental a la hora de obtener el DEM/DTM.

- **Clase 3, Vegetación baja:** Esta clase representa la vegetación de menos de 30 cm de alto, es decir, aquella vegetación muy próxima al suelo.

- **Clase 4, Vegetación media / alta:** Esta clase representa la vegetación que se encuentra por encima de los 30 cm de alto. Desde pequeños arbustos hasta árboles.

- **Clase 6, Edificio:** Esta clase representa los edificios. La variabilidad en esta clase es muy alta, en parte por la variabilidad de las zonas en nuestra zona de estudio, como ya se ha comentado en la *Sección 3.1.* Si la zona es rural tendremos pequeñas casas o casetas, si la zona es urbana tendremos bloques de viviendas y grandes edificios de oficinas mientras que en una zona industrial habrá grandes naves industriales.

- **Clase 10, Ruido sobre el suelo de objetos en movimiento:** En esta clase tenemos puntos sueltos por encima del suelo, que normalmente se generan por objetos en movimiento, principalmente vehículos. Como son puntos sueltos sin una forma bien definida se ha decidido establecer a una clase aparte.

- **Clase 17, Vehículo:** Esta clase engloba los vehículos, desde coches a furgonetas, incluyendo también camiones.

Estas clases son las que va a devolver nuestro modelo, de tal forma que, dado un punto de una nube de puntos de entrada, la clase final que este adquirirá solo puede ser una de las definidas anteriormente. En la figura 51 podemos ver una muestra etiquetada manualmente

- En azul tenemos la clase 2, suelo.
- En verde la clase 3, vegetación baja.
- En naranja tenemos la clase 4, vegetación medio / alta.
- En rojo tenemos la clase 6, edificio, en este caso es una nave industrial.
- En blanco tenemos la clase 17, vehículo.

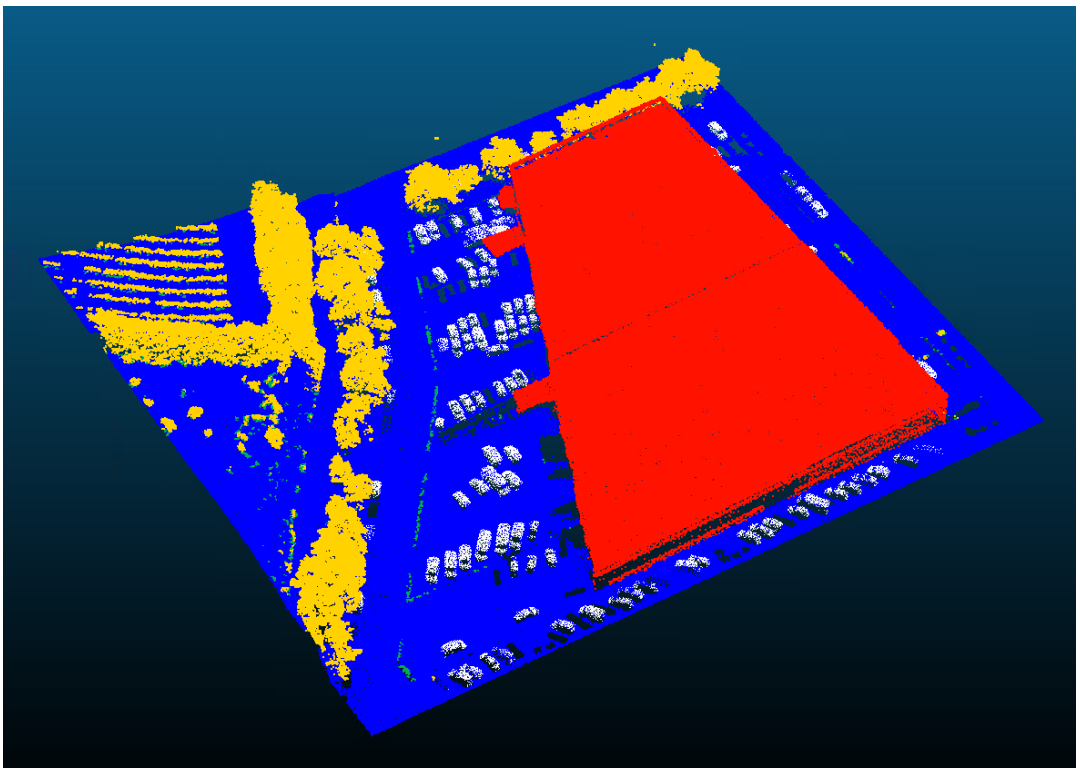

Figura 51. Muestra clasificada manualmente donde se muestra el campo Classification.

En la Figura 52 podemos observar de manera más clara la clase 10, ruido sobre el suelo. En la imagen de la izquierda podemos ver los puntos de la clase 10 en rojo sobre el RGB de la muestra, mientras que en la imagen de la derecha vemos directamente el campo *Classification* sobre la muestra. Estos puntos están dispersos sobre la carretera, por lo que podemos deducir que se deben a puntos principalmente generados por coches en movimiento.

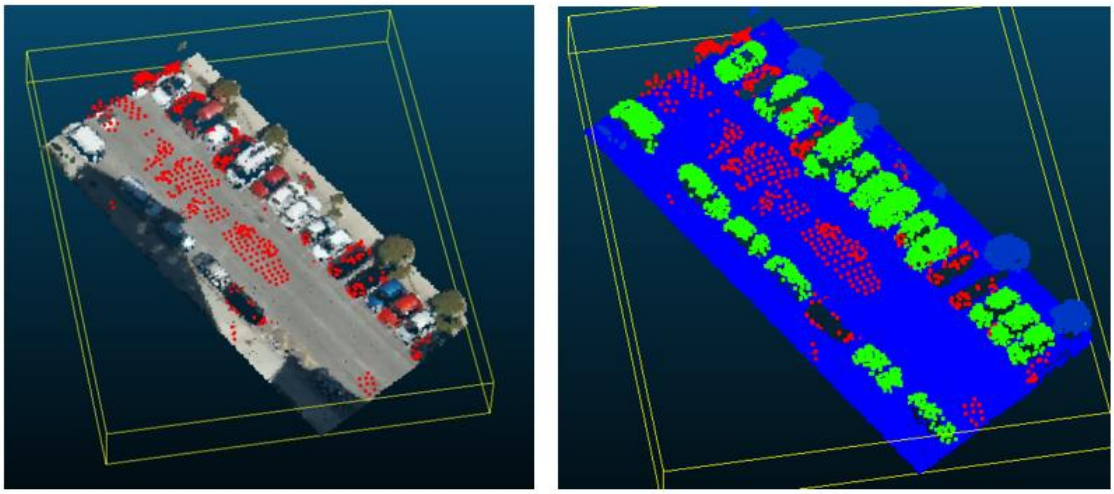

Figura 52. Ejemplo de muestra con puntos de la clase 10

3.4.Claves para la anotación manual y datasets base de Train / Test

Como ya se ha explicado en la *Sección 2.2* para que un modelo de *Deep Learning* aprenda a resolver una tarea mediante aprendizaje supervisado se necesitan previamente datos etiquetados. Hablaremos sobre ciertos aspectos a tener en cuenta a la hora de etiquetar manualmente las muestras en la *Sección 3.4.1*, mientras que en el *Sección 3.4.2* analizaremos nuestro dataset base de Train / Test obtenido sobre la zona final de estudio.

3.4.1.Claves a la hora de anotar manualmente muestras para nuestro modelo

La idea del aprendizaje supervisado es aprender en base a unos ejemplos anotados para luego poder extrapolar ese conocimiento a datos nunca vistos. Esto implica que los ejemplos que usará el modelo para aprender son muy importantes para obtener un buen resultado. Es decir, si los ejemplos están mal anotados o no proporcionan suficiente información, independientemente de que nuestro modelo de aprendizaje sea muy bueno, el resultado no será satisfactorio. Sería como si tuviéramos un coche con un motor muy bien diseñado, eficiente y potente, pero lo estuviéramos alimentando con un carburante de bajísima calidad. Para evitar este problema, es necesario definir una serie de claves a cumplir para la anotación manual de nuestras muestras. Estas claves se exponen en lo siguiente puntos:

- **Incorporación de la clase no etiquetado**: Esta es una clase de apoyo y en ningún caso nuestro modelo devolverá esta clase como predicción para un punto. Esta clase cumple

con dos objetivos. En primer lugar, actúa como **buffer**, es decir, proporciona contexto a los puntos que se encuentran en el límite de nuestras muestras. Como veremos en la *Sección 4*, los modelos de *Deep Learning* que utilizaremos para segmentar nubes de puntos hacen muchos cálculos usando vecindarios. Si los puntos de los bordes no tienen suficientes vecinos, esto puede traducirse en resultados pobres sobre este tipo de puntos. Para evitar este problema, aplicamos una idea similar a la del *padding* en imagen (concepto explicado en la *Sección 2.3.2.2*). Lo que se hace es añadir un pequeño buffer, de 10 metros o más dependiendo de la muestra, a los bordes de las zonas anotadas manualmente. Hay que aclarar que las muestras que se han anotado son un subconjunto de la zona final de estudio, es decir, son pequeñas muestras extraídas sobre los 124 bloques de 1km x 1km, por lo que los puntos que caen en el buffer realmente existen, es decir, la información no se inventa.

En la Figura 53 se muestra este proceso de manera visual. En la imagen de la izquierda podemos ver uno de nuestros 124 bloques de 1km x 1km. Sobre este bloque seleccionamos una pequeña zona para que se anote manualmente, en este caso la zona seleccionada en rojo. Mientras que, a la derecha, vemos la nube de puntos correspondiente a esa zona sobre la cual se está mostrando el campo *Classification*. El buffer, es decir la clase no etiquetado, se muestra en azul marino. Como se puede ver, se ha anotado todo excepto los bordes, que se han metido directamente a clase no etiquetado, de tal manera que los puntos diferentes a la clase no etiquetado tengan suficientes vecinos, independientemente de si están en el centro de la muestra o en borde.

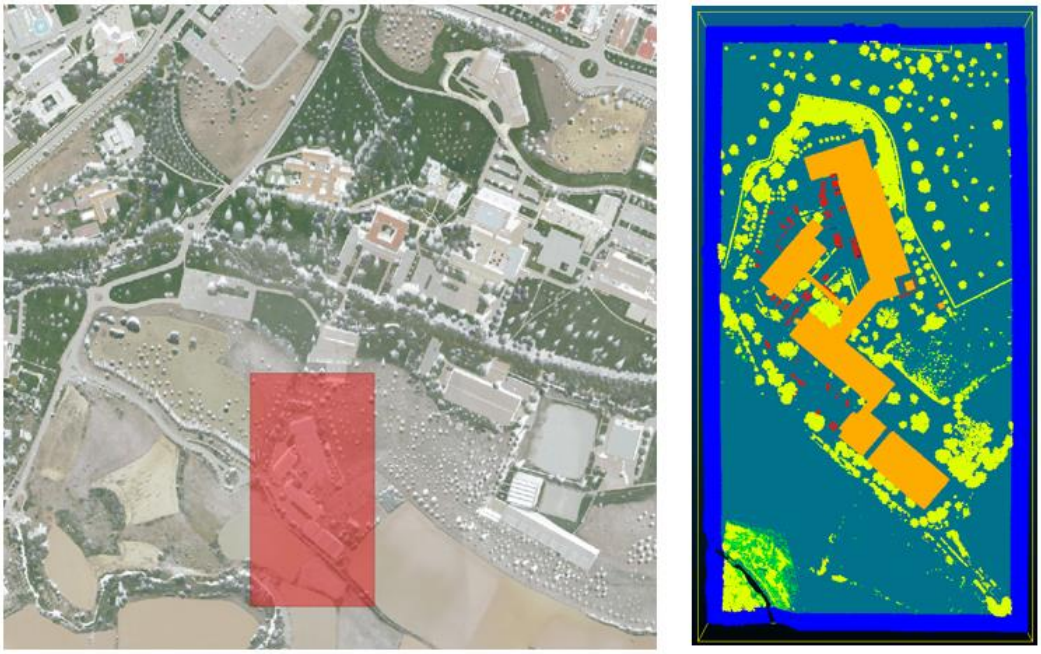

Figura 53. Selección de muestra sobre bloque de 1 km x 1km a la izquierda. A la derecha la nube de puntos seleccionada en la que se muestra el campo Classification, en azul marino el buffer.

Sin embargo, la clase no etiquetado no sirve solo para añadir un buffer a las muestras para proporcionar contexto a los puntos de los bordes, sino que también nos sirve como cajón de sastre para aquellas zonas que no tenemos claro lo que son. **Uno de los errores que queremos evitar a toda costa es proporcionar puntos mal anotados al modelo. Por ello, ante la duda, es mejor anotar estos puntos como clase no etiquetado.** Por

ejemplo, si hay un edificio en ruinas rodeado por vegetación de tal forma que hay puntos que no tenemos claro si son vegetación o edificio meteríamos todo este edificio a clase no etiquetado.

A la hora de entrenar nuestros modelos usaremos los puntos de la clase no etiquetado para los cálculos pertinentes, sin embargo, a la hora de calcular la función de coste ignoraremos la predicción sobre este tipo de puntos, ya que estos puntos no tienen asignados su clase real si no la clase no etiquetado, de tal manera que la predicción sobre estos puntos no contribuya al error.

- Como ya se ha comentado **uno de los motivos por los que se ha introducido la clase no etiquetado es para aportar contexto**, ya que este es fundamental. Para algunas clases como vehículo o vegetación puede que tener un gran contexto no sea tan necesario, sin embargo, para algún tipo de edificio es muy importante. Imaginemos una nave industrial con un tejado plano y bastante amplio. Si la muestra no tiene contexto suficiente, la parte central de la nave industrial, la cual está alejada de la fachada, es probable que se confunda con suelo por su geometría. Para evitar estos problemas, todas las muestras que se han generado tienen unas dimensiones mínimas de 120 metros x 120 metros, para que en caso de que haya edificios grandes, exista el suficiente contexto para una correcta clasificación.

- Que la **variabilidad** de los datos originales quede reflejada en el conjunto de datos etiquetados:

    - Para la clase 2, suelo, se seleccionaron zonas relativamente planas típicas de zonas urbanas o industriales, así como zonas con terrenos más escarpados y abultados típicas de zonas rurales.

    - Para la clase 3 y la 4, vegetación baja y media/alta respectivamente, se buscó que hubiera tanto vegetación de zonas boscosas, como árboles desperdigados, así como zonas con arbustos o vegetación típica de cultivos.

    - Para la clase 6 se intentó introducir una gran variedad de edificios, desde naves industriales a bloques de viviendas, incluyendo también pequeñas casas y casetas.

    - Para la clase 10 se incluyó puntos sueltos por encima del suelo tanto en carreteras como en campos.

    - Para la clase 17 se introdujeron muestras de parkings o zonas de aparcamiento con diferentes tipos de vehículos como coches, furgonetas, camiones etc.

3.4.2.Datset base de Train / Test

Teniendo en cuenta las claves comentadas en la sección anterior a la hora de anotar, se seleccionaron un total de 77 muestras repartidas por toda la zona de estudio. Las 77 muestras tienen diferentes tamaños, sin embargo, todas las muestras son más grandes que un tamaño mínimo, en este caso 120 metros, tanto en la coordenada X como en la coordenada Y. El objetivo

de establecer este tamaño mínimo es asegurar que haya suficiente contexto para clasificar determinados elementos como edificios grandes.

En la Tabla 3 podemos ver estadísticas sobre el total de los datos, tanto la cantidad de puntos por cada clase como el porcentaje que representan sobre el total. Como es obvio, la clase que más predomina es la clase 2, suelo, seguida por edificio y por vegetación media / alta. Clases como la 3, vegetación baja, la 10, ruido sobre el suelo, y la 17, vehículos, aparecen representadas en una menor proporción.

	Clase 2	Clase 3	Clase 4	Clase 6	Clase 10	Clase 17
Nº puntos	88.390.238	1.991.210	23.281.072	27.157.541	146.660	926.816
Porcentaje	62.28%	1.4%	16.4%	19.13%	0.103%	0.68%

Tabla 3. Número total de puntos por clase en el total de datos etiquetados

Para poder entrenar los modelos es necesario disponer de datos etiquetados, pero al mismo tiempo se necesitan datos etiquetados que permitan evaluar cómo de bueno son esos modelos. Con esto en mente, se formarán dos conjuntos, uno de Train, que se utilizará para entrenar los modelo, y otro de Test, que servirá para evaluarlos. A la hora de evaluar los modelos, como los datos están etiquetados, se pueden obtener métricas sobre las predicciones, lo cual será fundamental para comparar resultados. A la hora de elegir qué muestras pertenecen a cada uno de los conjuntos, Train o Test, se han tenido en cuenta los siguientes puntos:

- En primer lugar, la mayoría de los datos irán a Train. Cuantos más datos haya en este conjunto, más muestras verá el modelo, lo cual será fundamental para que este generalice mejor sobre datos nunca vistos. Sin embargo, el conjunto de Test tampoco puede ser muy pequeño, ya que, si no, las métricas que obtengamos estarán muy sesgadas por los pocos datos sobre los cuales estamos realizando la evaluación. Con esta idea en mente, **se busca una repartición aproximada del 60% de datos para Train y el 40% de datos para Test**.

- En segundo lugar, **que la proporción de puntos por clase en los conjuntos de Train / Test tiene que ser similar a la del total de los datos**. Es decir, si la clase mayoritaria sobre el total de los datos es suelo, un conjunto de Train donde la clase mayoritaria sea vegetación media / alta no estaría representando correctamente la proporción real de los datos. Al fin y al cabo, el conjunto de Train / Test puede ser pequeño, pero se busca que sea lo más parecido a los datos reales.

- **Que haya variabilidad de zonas en cada uno de los conjuntos**. Es decir, no solo se pretende que haya un determinado porcentaje de puntos de edificio en Test, por ejemplo, sino que los edificios que podamos encontrar en este conjunto, aunque no sean muchos, sean variados, es decir, que haya bloques de viviendas, naves industriales, casas pequeñas etc.

Siguiendo estas ideas se realizó la división de los datos en Train/Test. En la Tabla 4 podemos ver la cantidad de puntos en cada uno de los conjuntos, así como la proporción de cada una de las clases.

Conjuntos		Clase 2	Clase 3	Clase 4	Clase 6	Clase 10	Clase 17	% total
Train	Nº puntos	54,288,429	872,561	13,074,782	16,672,059	65,433	645,794	60,34%
	Porcentaje	63,41%	1,02%	15,27%	19,47%	0,08%	0,75%	
Test	Nº puntos	34,101,809	1,118,649	10,206,290	10,485,482	81,227	317,022	39,66%
	Porcentaje	60,56%	1,99%	18,13%	18,62%	0,14%	0,56%	

Tabla 4. Proporción de clases en los conjuntos de Train / Test.

En la Figura 54 podemos ver todas las muestras que se han etiquetado manualmente sobre la zona de estudio final. En rojo tenemos las muestras que pertenecen al conjunto de Train, mientras que en verde tenemos las muestras que pertenecen al conjunto de Test.

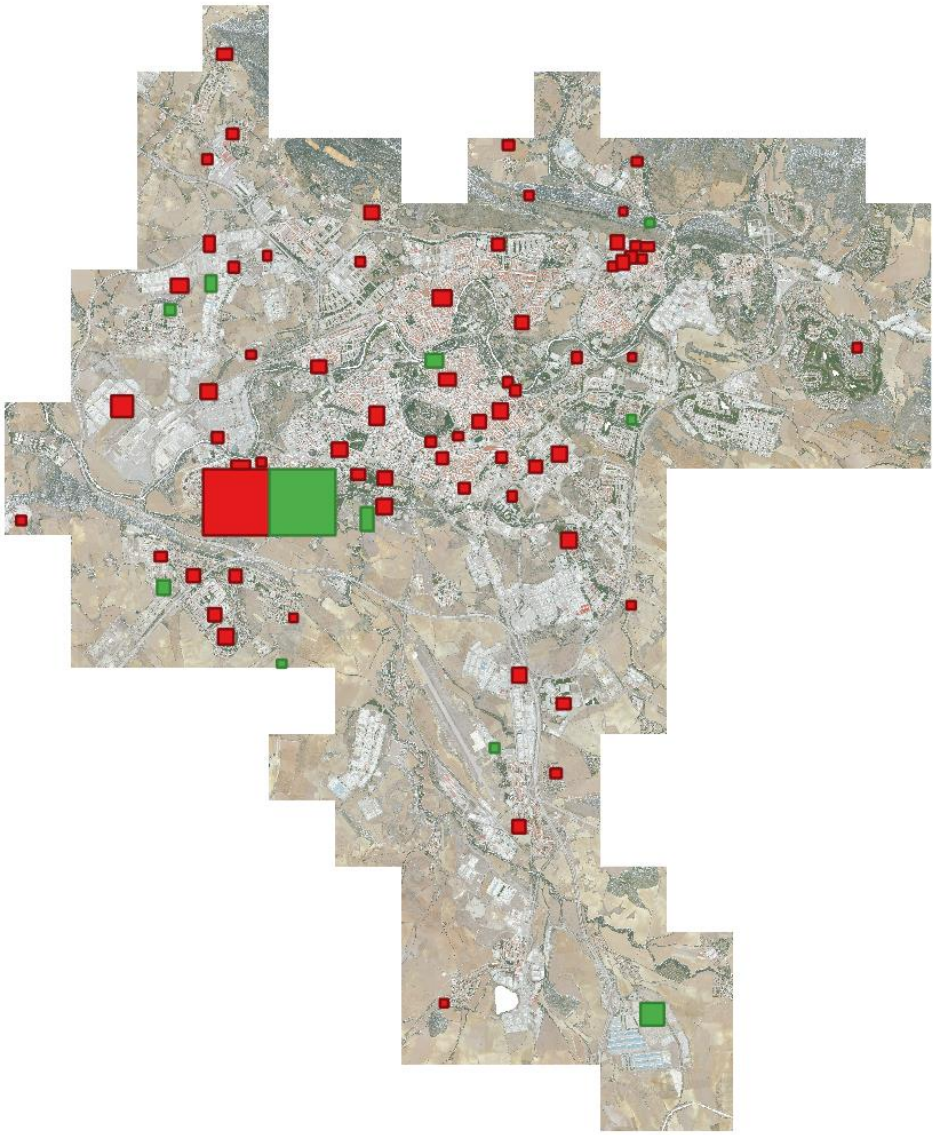

Figura 54. Zonas de Train (en verde) y zonas de Test (en rojo) sobre la zona final de estudio.

4.Evolución en los modelos de Deep Learning centrados en la segmentación de nubes de puntos

En esta sección se realizará un recorrido a lo largo de los diferentes modelos de *Deep Learning* centrados en la segmentación de nubes de puntos que han salido estos últimos años. El objetivo de esta sección es plantear las bases de estos modelos para que el lector tenga una ligera noción del funcionamiento de estos, así como entender las principales diferencias que existen entre ellos. En la *Sección 6* se realizarán experimentos sobre nuestros datos usando estos modelos, de ahí que estos se expliquen antes para que resulten conocidos para el lector. Los modelos se presentan en orden cronológico, ya que algunos son versiones mejoradas de modelos ya existentes, mientras que otros se apoyan en ciertas ideas o técnicas planteadas anteriormente por otros modelos.

4.1.PointNet

Trabajar con nubes de puntos es difícil. Como se ha comentado en la *Sección 2.1.1*, las nubes de puntos pueden "simplificarse" voxelizando el espacio. Esto proporciona estructura y facilita su tratamiento, pero viene acompañado de una pérdida de información y de un gran coste computacional. Lo ideal sería trabajar con la nube de puntos directamente, sin embargo, como se mencionó en la *Sección 2.1.3* y como se puede observar en la Figura 55, las nubes de puntos no tienen una estructura bien definida. En la Figura 55 se representa la misma matriz con las filas desordenadas, donde cada fila tiene un color asignado. Cada una de estas filas representa las coordenadas de un punto en el espacio, por lo que la matriz está representando una nube de puntos de 4 puntos, ya que tiene 4 filas. Tanto la matriz de la derecha como la de la izquierda son la misma nube de puntos, aunque su representación en cuanto a orden sea diferente. Por ejemplo, la primera fila de la matriz izquierda, la azul marina, es la última fila de la matriz derecha, no coinciden. Pero la nube de puntos es la misma.

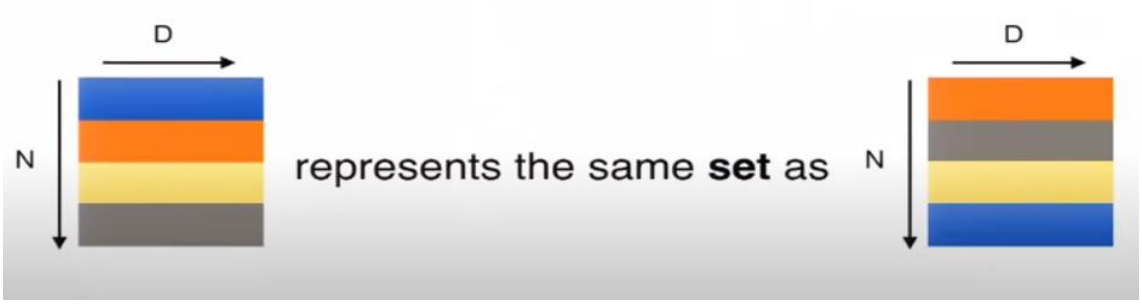

Figura 55. Misma nube de puntos, diferentes matrices que la representan. [fuente]

Por ellos, si se busca un modelo de *Deep Learning* que pueda trabajar directamente con nubes de puntos, **necesitamos que este sea invariante a permutaciones en el orden de los puntos**. El primer modelo en lograr esto, presentado en 2017, es el conocido como ***PointNet*** [23]**.** Este modelo es capaz de trabajar directamente con nubes de puntos, siendo invariante a

permutaciones en el orden de estos, así como invariante a transformaciones geométricas aplicadas sobre la nube de puntos, como podría ser una rotación.

Para conseguirlo, *PointNet* se apoya directamente en las matemáticas. Matemáticamente hablando, una red neuronal es solo una función, muy compleja, pero una función. Existen funciones cuyo resultado es invariante al orden en el que reciba los parámetros. En el campo de las matemáticas estas funciones se denominan **funciones simétricas**. Un ejemplo sería el máximo de un conjunto de elementos, independientemente del orden el resultado será el mismo, como muestra la Ecuación 7.

$$\max(x_1, x_2, x_3) = \max(x_2, x_3, x_1) = \max(x_3, x_1, x_2) \qquad (7)$$

Otra función que cumple esta propiedad sería el producto. Apoyándose en este tipo de funciones lo que se propone *PointNet* es crear una función f que representa la red neuronal y que sea invariante a estas permutaciones. Para ello, los autores de este modelo presentan la Ecuación 8.

$$f(x_1, x_2, \dots, x_n) = \gamma \circ g(h(x_1), \dots, h(x_n)) \quad (8)$$

Suponemos que f, la función que representa nuestra red neuronal se puede escribir como la composición de γ y de g. En este caso, f es simétrica siempre y cuando g también lo sea. La idea de esta ecuación está representada en la Figura 56. En este ejemplo recibimos un conjunto de puntos con las coordenadas en un espacio 3D como únicas características. En primer lugar, transformamos cada punto de manera independiente por una pequeña red h. Esta pequeña red proyectará los puntos a un espacio invariante de mayor dimensionalidad. En segundo lugar, se agrega la salida que devuelve la red h para cada punto mediante una función simétrica g. Para finalizar, transformamos la información agregada apoyándonos en otra red, γ.

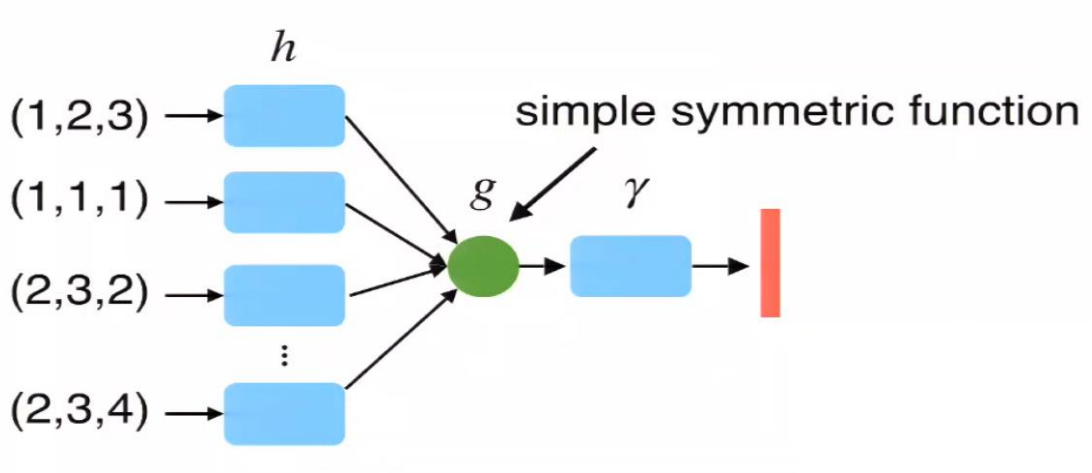

Figura 56. Representación gráfica de la función base del PointNet. [fuente]

Según un teorema desarrollado más en profundidad en su *paper*, *PointNet* es capaz de aproximar cualquier función simétrica que sea continua. A la hora de implementar esta estructura, como se puede observar en la Figura 57, la función h y γ son perceptrones multicapa con BatchNormalization y ReLU como función de activación, mientras que la función de agregación simétrica llamada g es el máximo.

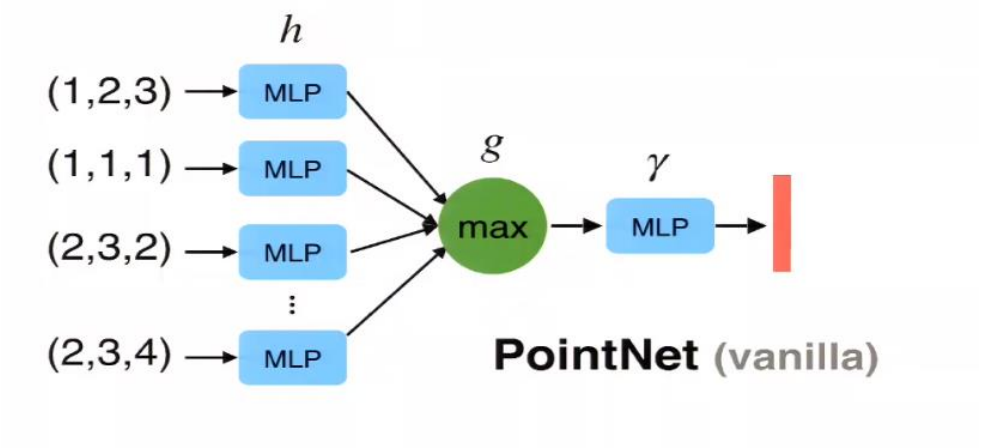

Figura 57. Estructura básica del PointNet. [fuente]

Otro aspecto fundamental que se debe cumplir a la hora de trabajar directamente con nubes de puntos, aparte de ser invariante a permutaciones en el orden de los puntos de entrada, es ser invariante a transformaciones geométricas como, por ejemplo, rotaciones. La idea que presenta PointNet para lograr esto es alinear la nube de puntos en un espacio canónico. Para ello *PointNet* se apoya en una estructura que la denominaron *T-Net*, la cual es una minired cuya función es predecir un matriz de transformación y aplicarla sobre la nube de puntos de entrada para conseguir ser invariantes a transformaciones geométricas. En la Figura 58 se muestra cómo se aplica la *T-Net.*

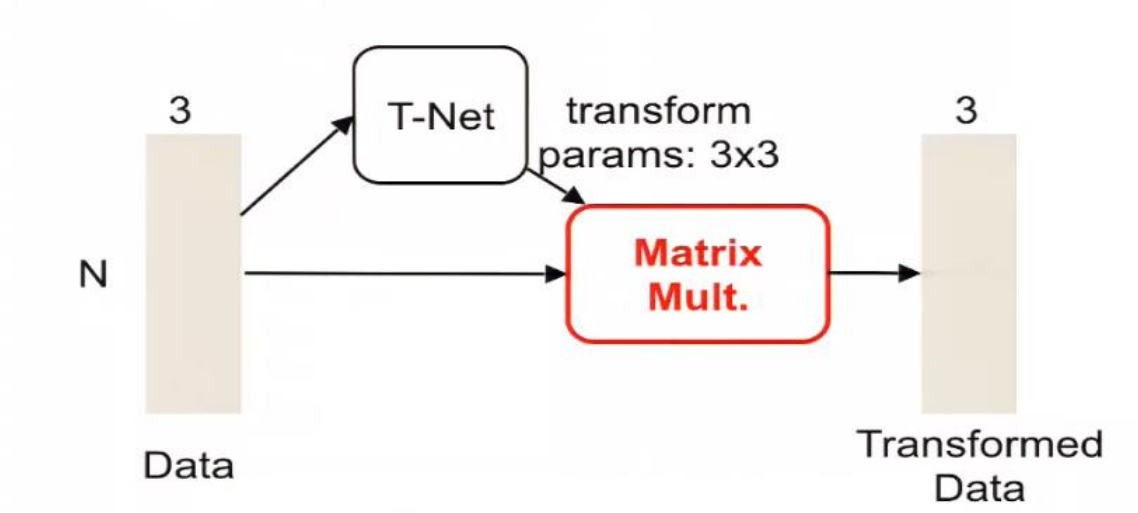

Figura 58. Aplicación de la matriz predicha por la T-Net sobre la nube de puntos de entrada. [fuente]

Una vez presentadas las bases en las que se apoya *PointNet,* es hora de analizar la arquitectura de la red, la cual está representada en la Figura 59. Como se puede ver en la imagen, esta está dividida en dos zonas. La zona azul se centra en tareas de clasificación. Es decir, dar una clase a la nube de puntos, no una clase por punto. Mientras que la parte amarilla, haciendo uso de cálculos realizados en la parte azul, se centra en realizar tareas de segmentación. Es decir, predecir una clase por cada punto en vez de una clase para toda la nube de puntos. Como el objetivo de este trabajo es segmentar nubes de puntos, no clasificarlas, pondremos la atención

en la parte de segmentación. En la Figura 59 podemos ver varios números, los cuales se han añadido para poder explicar de manera clara y ordenada los pasos que realiza la red.

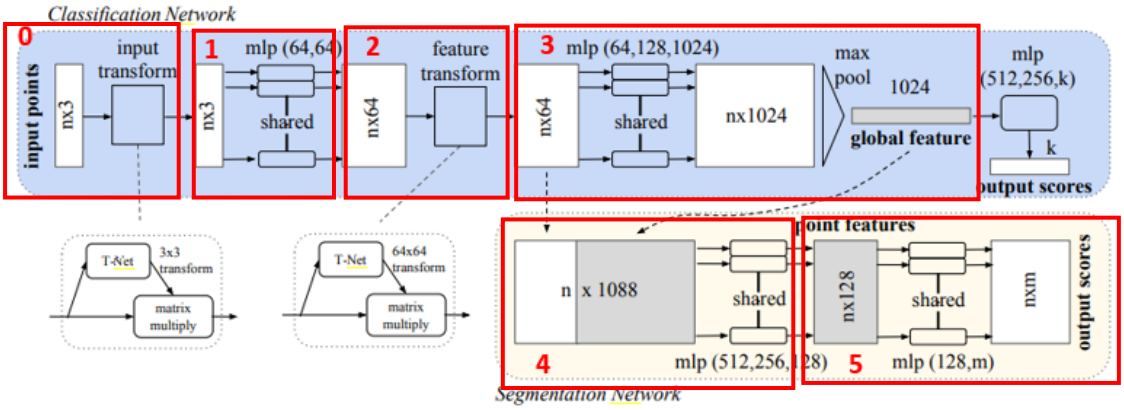

Figura 59. Arquitectura PointNet. [fuente]

- **Parte 0:** Se recibe como entrada una nube de puntos con n puntos con sus coordenadas XYZ, de ahí que su dimensión sea $n\ x\ 3$. Se predice una matriz de transformación con la *T-net* y se multiplica esta por la entrada con el objetivo de ser invariantes ante transformaciones geométricas. La salida tiene la misma dimensión que la entrada.

- **Parte 1:** Se aplica un perceptrón multicapa a la salida de la parte cero, lo que da un matriz con una dimensión de $n\ x\ 64$. La idea de aplicar MLPs (*multi layer perceptron*) es la de ir extrayendo características por cada punto, de ahí que ahora haya 64 columnas en vez de las 3 del inicio correspondiente a XYZ.

- **Parte 2:** Se multiplica la salida de la Parte 1 por una matriz de transformación dada por la *T-net*. La dimensión de la salida es la misma que la de entrada.

- **Parte 3:** Se aplica otro perceptrón multicapa, obteniendo así una matriz de salida con una dimensión de $n\ x\ 1024$. A continuación, aplicamos un max pool, quedándonos con el máximo de cada columna, lo que nos acaba dando un vector de 1024 elementos. Esta función al ser el máximo de un conjunto es simétrica. Esta función es la que en la Figura 56 y 57 se le denomina como g .

- **Parte 4:** En esta parte, como el objetivo es la segmentación, se concatena la salida de la Parte 2 con la salida de la Parte 4. A todos los puntos se le concatena el mismo vector de 1024 posiciones que se obtiene en la Parte 4. Se concatena la salida de la Parte 2 ya que esta contiene características locales a nivel de punto, mientras que añadir la salida de la Parte 4 proporciona características a nivel global. Es decir, el vector de 1024 posiciones es el mismo para todos los puntos. De esta manera, combinando la información a dos escalas, a nivel local y a nivel global, se obtiene como entrada una matriz de $n\ x\ 1088$ como entrada (64 + 1024 = 1088). Aplicando un perceptrón multicapa a esta combinación se obtiene como salida una matriz de dimensión $n\ x\ 128$.

- **Parte 5:** Para finalizar, se recibe como entrada la salida de la Parte 4 y se le aplica el último MLP, consiguiendo así una matriz final con una dimensión de $n\ x\ m$, donde n es

el número de puntos de la nube de puntos de entrada y m es el número de clases con las que se ha entrenado al modelo. Lo que representa esta matriz es la probabilidad de que cada punto pertenezca a cada una de las posibles clases. De tal forma que la predicción final de cada punto será aquella clase cuya probabilidad sea la más alta.

Apoyándose en el concepto de función simétrica *PointNet* fue el primer modelo de *Deep Learning* capaz de trabajar directamente con nubes de puntos. También planteó una arquitectura que permite tanto clasificar como segmentar nubes de puntos, consiguiendo resultados superiores que otros modelos de la época sobre los mismo datasets. Este modelo servirá de inspiración para otros, guardando una estrecha relación son el siguiente modelo, el *PointNet ++*.

4.2.PointNet ++

Como se ha visto en la *Sección 4.1* a la hora de segmentar la nube de puntos, *PointNet* concatena ciertas características locales por cada punto con un vector de características de 1024 posiciones, que representa las características globales. Sin embargo, esto no es suficiente, dando lugar a que *PointNet* no consigue capturar del todo las estructuras locales, dejándose influenciar enormemente por las características globales de la nube de puntos, lo que se traduce en un desempeño pobre sobre clases más minoritarias o cuya estructura geométrica sea más pequeña. Por poner un ejemplo, si aplicásemos *PointNet* sobre uno de nuestros bloques, tendríamos un buen desempeño sobre clases como el suelo, que está contenida en gran proporción y se extiende por toda la muestra, pero un mal desempeño sobre coches o edificios pequeños, estructuras más pequeñas que no tienen casi representación a nivel global.

Los propios autores del PointNet se dieron cuenta de este problema, y en el mismo año en el que se publicó su trabajo de investigación sobre la creación del PointNet publicaron la mejora de este modelo, el cual fue bautizado como *PointNet ++* [24]. Este modelo está fuertemente inspirado en *PointNet*, y su objetivo es mejorar la captura de estructuras locales para conseguir un modelo capaz de reconocer patrones más complejos y conseguir mejor desempeño en escenas más complejas.

La idea clave de este modelo es usar *PointNet* pero a diferentes escalas, podríamos definir *PointNet ++* como un *PointNet* multiescala. Es decir, aplicar *PointNet* a varios niveles para ser capaz de aprender mejor esas estructuras locales. Para entender bien en qué consiste este proceso de aplicar *PointNet* a varias escalas vamos a apoyarnos en la Figura 60, la cual muestra la arquitectura del *PointNet ++*.

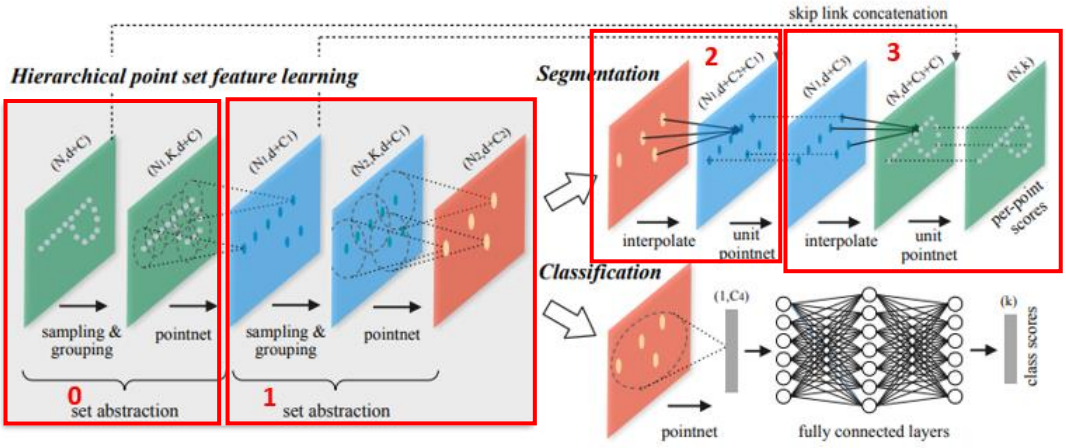

Figura 60. Arquitectura del PointNet ++. [fuente]

PointNet ++ sigue usando *PointNet* pero haciendo ciertos cambios sobre los datos antes de aplicarlo:

- **Parte 0 y 1:** la idea es sencilla, si *PointNet* no proporciona el resultado deseado al pasarle como entrada toda la nube de puntos, ¿Por qué no aplicarlo sobre zonas más pequeñas y varias veces? Para conseguir esto entran en juego dos tipos de capas (*layers*)

    - **Sampling layer:** La labor que tiene esta capa es simple. Dado una serie de puntos de entrada debe seleccionar un subconjunto de estos. En vez de usar un dowsampling aleatorio, PointNet ++ utiliza un algoritmo conocido como *farthest point sampling* (FPS). El algoritmo FPS dado un conjunto de puntos de entrada $\{x_1, x_2, \dots, x_n\}$ selecciona un subconjunto de puntos $\{x_{i_1}, x_{i_2}, \dots, x_{i_1 m}\}$ tal que x_{i_j} es el punto más lejano del conjunto $\{x_{i_1}, x_{i_2}, \dots, x_{i_1 j-1}\}$. En resumidas cuentas, queremos downsamplear la muestra, pero al mismo tiempo queremos que la estructura de la muestra downsampleada sea lo más parecida a la estructura original de la muestra, con este algoritmo se asegura un mejor desempeño en esta tarea que un downsampleao aleatorio.

    - **Grouping layer:** La capa anterior nos ha proporcionado un subconjunto de puntos. Estos puntos serán los centroides sobre los cuales se aplicará un determinado radio. Los K vecinos que caigan en este radio definido sobre los centroides serán la nube de puntos de entrada para aplicar PointNet.

    Este proceso se aplica en la parte 0, de manera que obtenemos los centroides en la *sampling layer*, agrupamos los vecindarios en la *grouping layer* y para finalizar aplicamos *PointNet* sobre cada uno de los vecindarios. Sin embargo, el proceso no termina aquí, sino que se utiliza los puntos elegidos como centroides como entrada para la parte 1 y volvemos a aplicar el proceso de *sampling* más *grouping*, finalizando con la aplicación del *PointNet* sobre los vecindarios restantes.

En la Figura 61 podemos ver la zona correspondiente a las partes 0 y 1 en más detalle. En la imagen podemos ver varias letras que representa las dimensiones de las matrices que se van generando:

- N: Número de puntos que tiene la nube de entrada.
- d: Dimensión de las coordenadas, en este caso al ser XYZ tendría un valor de 3.
- N_1, N_2: Número de puntos seleccionados en la *sampling layer* que serán los centroides.
- K: Número de vecinos más cercanos al centroide sobre los cuales aplicar el *PointNet*.
- C: Número de características extra además de las coordenadas. Si estuviéramos usando el RGB, por ejemplo, tendía un valor de 3 (Red + Green + Blue).
- C_1, C_2 : Número de características que nos devuelve el *PointNet.*

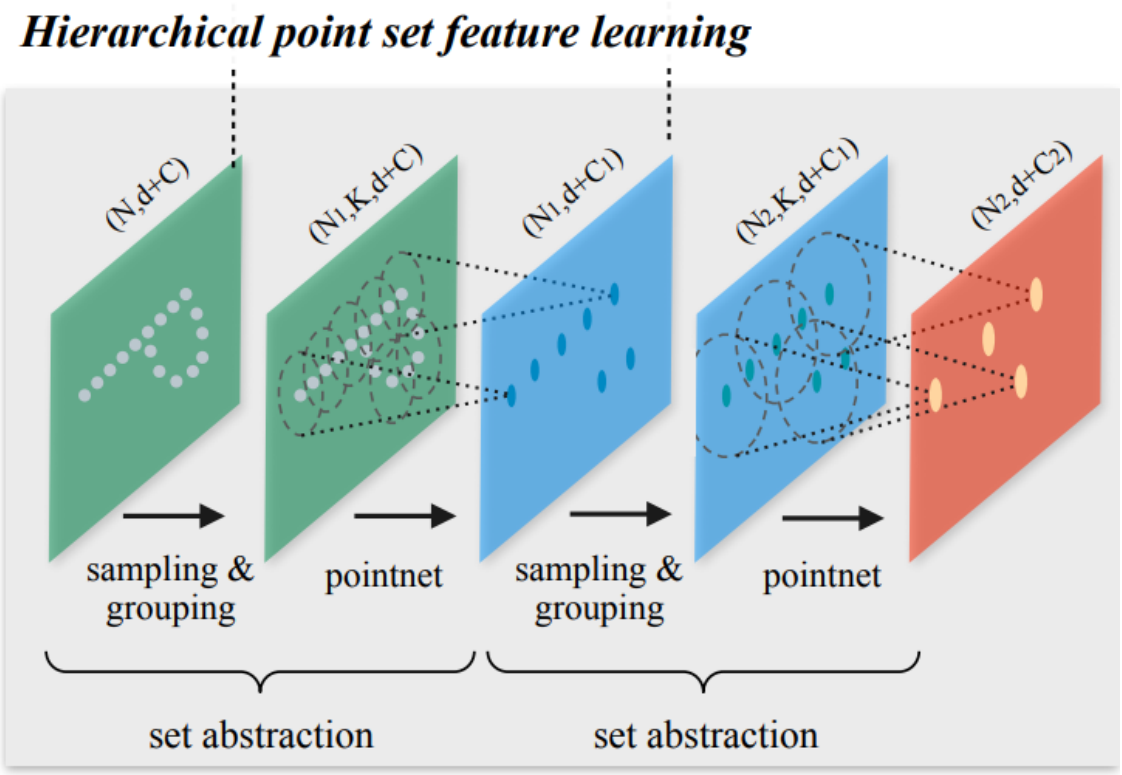

Figura 61. Proceso de sampling & grouping en el PointNet ++. [fuente]

- **Parte 2 y 3:** El resultado de la Parte 1 son unos pocos puntos con vectores de características muy grandes. En la parte 0 y 1 se ha aplicado *PointNet* en pequeños vecindarios a diferentes escalas para conseguir este resultado. Sin embargo, el objetivo es predecir una clase por cada punto presente en la nube de puntos de entrada. Para ello, lo que se debe realizar a continuación, es asignar las características correspondientes a cada punto. Es decir, interpolamos las características aprendidas en las partes 0 y 1 a todos los puntos. En la Figura 62 se muestras estás dos partes en más detalle. Para conseguir interpolar las características se definen una serie de links representados con líneas discontinuas tanto en la Figura 60 como en la 62, mediante los cuales se concatenan las últimas características obtenidas, un vector de C_2 elementos por punto, con el vector de características anterior, de C_1 elementos por punto.

La *unit pointnet* es similar a la idea de las *one-by-one convolution* en las CNNs, convoluciones que al ser filtros de 1x1 su principal funcionalidad es incrementar o decrementar la dimensionalidad de los datos. Por tanto, la *unit pointnet* agrega las características concatenadas en pasos anteriores, en la parte 2 modifica las características de $C_2 + C_1 \rightarrow C_3$ y en la parte 3 modifica las características de $d + C_3 + C \rightarrow k$. Donde k representa la probabilidad de que cada punto pertenezca a cada una de las posibles clases con las que se ha entrenado el modelo.

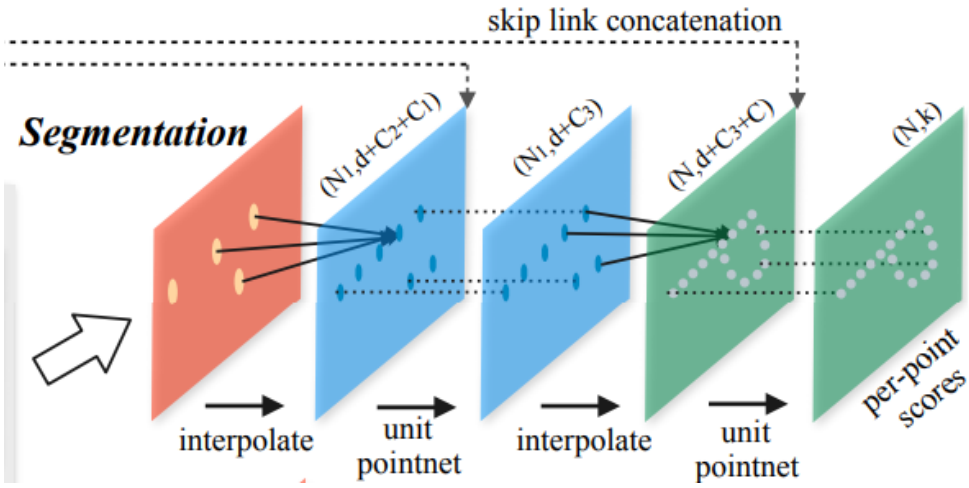

Figura 62. Parte 2 y 3 del PointNet ++ en más detalle.

En resumidas cuentas, podemos considerar al *PointNet ++* como el hermano mayor del *PointNet*, que al aplicar el *PointNet* en pequeños vecindarios y a varias escalas, es capaz de aprender estructuras locales además de las globales, tarea en la que su hermano pequeño no daba buenos resultados.

4.3.PointCNN

Nuestra siguiente parada es el *PointCNN* [25], un modelo presentado en 2018 cuya principal novedad es la aplicación de la convolución típica de imágenes sobre las nubes de puntos. En esta sección nos centraremos en explicar las dificultades a la hora de aplicar convoluciones sobre nubes de puntos, así como las claves y funcionamiento básico de este modelo, finalizando con el análisis de su arquitectura.

4.3.1.Dificultades a la hora de aplicar convoluciones sobre nubes de puntos

Para los datos que se representan en dominios regulares, como las imágenes, se ha demostrado que la convolución es un operador eficaz, capaz de explotar la correlación espacial, consiguiendo así buenos resultados en una gran variedad de tareas, desde clasificación de imágenes a segmentación, incluyendo tareas como la detección de objetos. Sin embargo, este operador no puede aplicarse de la misma forma sobre las nubes de puntos, ya que como se ha comentado varias veces a lo largo de este trabajo, las nubes de puntos son un conjunto de datos irregulares y desordenados. Para mostrar el problema que supone aplicar la convolución sobre nubes de puntos vamos a apoyarnos en la ilustración de la Figura 63.

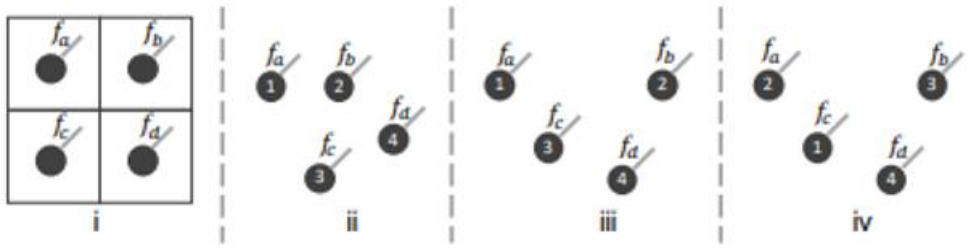

Figura 63, Imagen de apoyo para demostrar la dificultad de aplicar un operador de convolución sobre nubes de puntos. [fuente]

En la Figura 63 tenemos cuatro nubes de puntos $\{i, ii, iii, iv\}$. A la primera de todas se le define un grid, estructura similar a los pixeles en imagen. Cada uno de los puntos tiene definidas una serie de características, las cuales se representan como $\{f_a, f_b, f_c, f_d\}$. En las demás nubes de puntos, $(ii - iv)$, tenemos puntos con las mismas características, sin embargo, al no haber una estructura que limite el espacio, la numeración de los puntos se hace de manera arbitraria.

Supongamos que tenemos un kernel (filtro) de 2 x 2 tal que $K = \begin{bmatrix} ka & kb \\ kc & kd \end{bmatrix}$. Ahora definimos la operación $Conv(K,\ PointCloud)$, donde $PointCloud$ será cualquiera de las nubes de puntos $(i - iv)$, como el producto elemento a elemento entre el kernel y la nube de puntos y un operador suma que agregue todo. Podemos ver esta operación de manera gráfica sobre todas las nubes de puntos en la Figura 64. **Remarcar que la colocación de la nube de puntos en el grid depende del orden asignado a los puntos.**

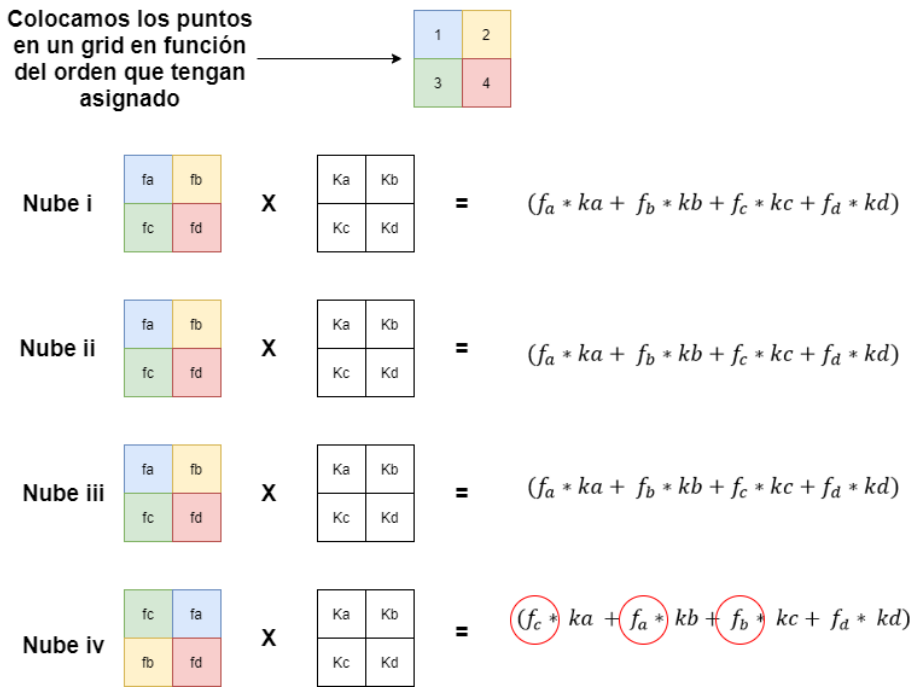

Figura 64. Operación de convolución sobre la nube de puntos i.

Ahora bien, si se realiza la misma operación de convolución sobre las demás nubes de puntos, $(ii - iv)$, las cuales tienen las mismas características por punto, da lugar a que el resultado no es igual para todas, como se muestra en la Ecuación 9.

$$Conv(K, i) = Conv(K, ii) = Conv(k, iii) \neq Conv(K, iv) \quad (9)$$

Como el orden de las nubes de puntos es arbitrario, al no haber una estructura bien definida, el operador de convolución sobre la nube de puntos iv nos da un resultado diferente a las demás, ya que el orden de sus puntos es diferente. Es indispensable resolver este problema para poder aplicar convoluciones sobre este tipo de datos.

La clave del *PointCNN* es aprender una transformación χ, de dimensiones $K \times K$ para las coordenadas de los K puntos de entrada $(p_1, p_2, \dots, p_K)$ haciendo uso de un perceptrón multicapa, $\chi = MLP(p_1, p_2, \dots, p_K)$. El objetivo de esta transformación es ponderar y permutar al mismo tiempo las características de entrada para seguidamente poder aplicar una convolución sobre las características transformadas. Este proceso se denomina como $\chi - Conv$ y es la base del *PointCNN*. En resumidas cuentas, lo que se busca es poder aplicar una función, χ, sobre los puntos de entrada para ser invariantes (o al menos robustos) a las permutaciones en el orden y poder así aplicar convoluciones sobre las nubes de puntos. Los propios autores del *PointCNN* reconocen que la transformación χ está lejos de ser ideal, sin embargo, los resultados que obtienen no son solo mejores que aplicar convoluciones directamente sobre la nube de puntos, sino que también superan los resultados obtenidos por otros modelos como puede ser el *PointNet ++*.

4.3.2.Funcionamiento del PointCNN

En esta sección se explicará el funcionamiento interno de este modelo. La aplicación jerárquica de convoluciones es esencial para el aprendizaje de las CNNs. *PointCNN* comparte la misma idea, generalizando su aplicación sobre las nubes de puntos. Esta sección se dividirá en dos secciones, una que explique las convoluciones jerárquicas sobre nubes de puntos y otra que explique en profundidad el operador $\chi - Conv$.

4.3.2.1.Convoluciones jerárquicas

Para explicar la aplicación de convoluciones jerárquicas sobre nubes de puntos se va a utilizar como apoyo la Figura 65, donde se ilustra este proceso sobre una imagen, así como sobre una nube de puntos.

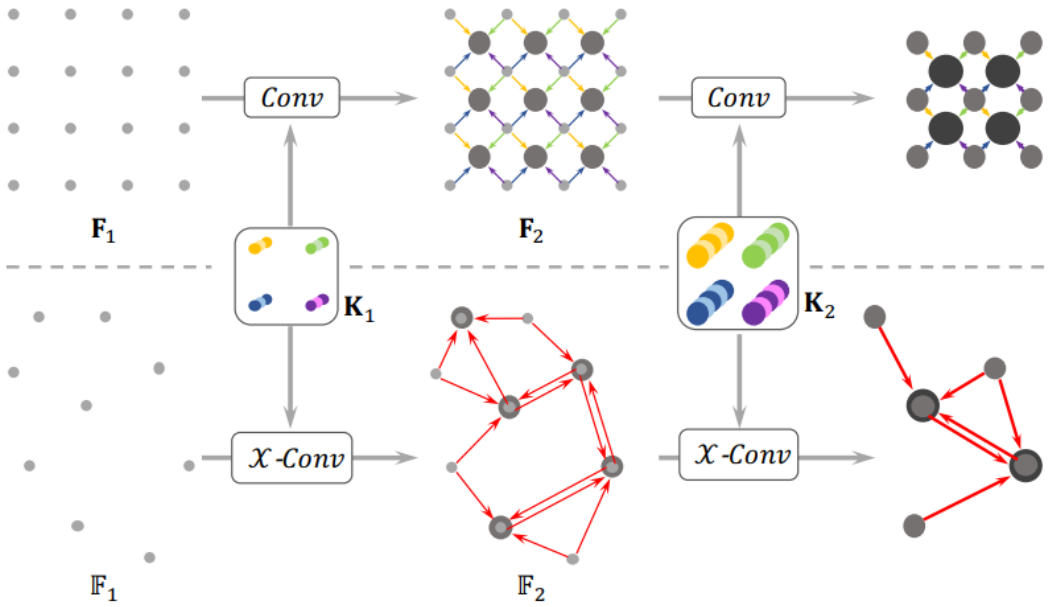

Figura 65. Convoluciones jerárquicas sobre imagen la parte superior y sobre nubes de puntos aplicando el operador X-Conv en la parte inferior.[fuente]

En estructuras regulares como la de una imagen las convoluciones se aplican recursivamente sobre parches locales, normalmente reduciendo poco a poco la resolución. En la zona superior de la Figura 64 podemos ver este proceso sobre una imagen, la cual va reduciendo su dimensionalidad de $4\ x\ 4 \rightarrow 3\ x\ 3 \rightarrow 2\ x\ 2$ al mismo tiempo que se va incrementando el número de canales (su profundidad), lo cual se refleja en la imagen incrementando el grosor de los puntos. Siguiendo este esquema, $\chi - Conv$ se aplica recursivamente sobre la nube de puntos para proyectar y agregar la información del vecindario en unos pocos puntos que lo representan $9 \rightarrow 5 \rightarrow 2$, pero cada uno con mayor cantidad de información. Los puntos elegidos para ser los representantes después de aplicar el operador $\chi - Conv$ se eligen de manera aleatoria para tareas de clasificación, mientras que en segmentación, al igual que el *PointNet ++,* se utiliza el *farthest point sampling* (algoritmo ya mencionado en la *Sección 4.2*) ya que para este tipo de tarea es necesario una distribución de puntos más uniforme.

Sobre esto puntos seleccionados con el *farthest point sampling* se deberá agregar la información de los puntos que formen su vecindario, de tal forma que el número de puntos a lo largo de las capas se reduce, pero su información no se pierde al estar contenida en los puntos representativos. La Figura 66 muestra paso a paso el proceso mediante el cual se agregan estas características.

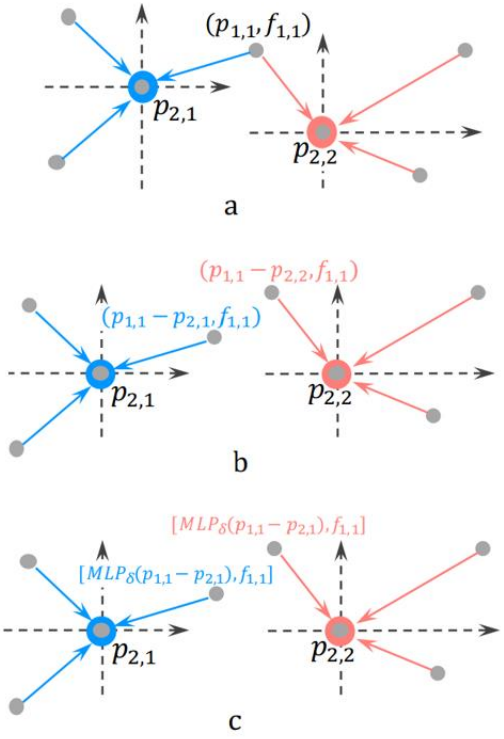

Figura 66. Transformación de las coordenadas de los puntos en características.[fuente]

En primer lugar, en el paso (a), se seleccionan los vecindarios que están formados por los puntos más próximos. En segundo lugar, en el paso (b), se transforma los puntos del vecindario al sistema de coordenadas del punto representativo. Remarcar que un punto puede ser elegido para formar parte de diferentes vecindarios. Para finalizar, en el paso (c) se procesan las coordenadas de cada punto del vecindario, las cuales están transformadas respecto al punto representativo, con un MLP para elevar su dimensionalidad, adquiriendo así una representación más abstracta. Se concatena este resultado con las características originales de los puntos.

4.3.2.2.Operador $\chi - Conv$

Como ya se ha mencionado en las secciones anteriores el operador $\chi - Conv$ es la clave para poder aplicar convoluciones sobre nube de puntos, en la Figura 65 podemos ver como el paso de $\mathbb{F}_1 \rightarrow \mathbb{F}_2$ se puede ejecutar solo gracias a este operador. En esta sección, usando la Figura 67 como apoyo, se explicará paso a paso el funcionamiento del algoritmo.

ALGORITHM 1: $\mathcal{X}$-Conv Operator

Input : $\mathbf{K}, p, \mathbf{P}, \mathbf{F}$
Output : $\mathbf{F}_p$

1: $\mathbf{P}' \leftarrow \mathbf{P} - p$
2: $\mathbf{F}_\delta \leftarrow MLP_\delta(\mathbf{P}')$
3: $\mathbf{F}_* \leftarrow [\mathbf{F}_\delta, \mathbf{F}]$
4: $\mathcal{X} \leftarrow MLP(\mathbf{P}')$
5: $\mathbf{F}_\mathcal{X} \leftarrow \mathcal{X} \times \mathbf{F}_*$
6: $\mathbf{F}_p \leftarrow \text{Conv}(\mathbf{K}, \mathbf{F}_\mathcal{X})$

Figura 67, algoritmo del $\chi - Conv$. [fuente]

- Como **entrada** se recibe los siguientes parámetros:
  - $\boldsymbol{K}$: Kernel (filtro) que se utilizará para aplicar la convolución.
  - p: Punto representativo elegido sobre el que se agregaran las características.
  - $\boldsymbol{P}$: Matriz que representa las coordenadas de los puntos de la nube de puntos de entrada. Su dimensión es de $K \times Dim$. Donde K es el número de puntos y Dim el número de elementos para representar las coordenadas.
  - $\boldsymbol{F}$: Matriz que representa las características de los puntos de la nube de puntos de entrada. Su dimensión es de $K \times C_1$. Donde K es el número de puntos y C_1el número de características.

- Como **salida** se obtiene $\boldsymbol{F}_p$, las características agregadas sobre el punto representativo p.

- Pasos que realiza el operador $\chi - Conv$:
  - **Paso 1**: Se mueve $\boldsymbol{P}$ al sistema de coordenadas de p
  - **Paso 2**: Se introduce el resultado del paso anterior, $\boldsymbol{P}'$, en un perceptrón multicapa, elevando así cada punto a un espacio de dimensionalidad C_δ.
  - **Paso 3**: Se concatena el resultado del paso anterior, $\boldsymbol{F}_\delta$, con las características iniciales de la nube de puntos $\boldsymbol{F}$. La dimensión de $\boldsymbol{F}_*$ es de $K \times (C_\delta + C_1)$.
  - **Paso 4**: Utilizando el resultado del paso 1, $\boldsymbol{P}'$, y haciendo uso de un perceptrón multicapa se aprende una matriz de transformación $\boldsymbol{\chi}$ de dimensiones $K \times K$.

- **Paso 5**: Aplicamos la matriz de transformación obtenida en el paso anterior, $\boldsymbol{\chi}$, sobre la salida del paso 3, para permutar y ponderar $\boldsymbol{F}_*$ con el objetivo de poder aplicar una convolución en el último paso.
- **Paso 6**: Finalmente, una vez que tenemos la matriz $\boldsymbol{F}_{\boldsymbol{\chi}}$ a la cual se le ha aplicado el operador $\boldsymbol{\chi}$, podemos aplicar la convolución con el kernel $\boldsymbol{K}$.

Todo este algoritmo está resumido en la Ecuación 10.

$$\boldsymbol{F}_p = \chi - Conv(\boldsymbol{K}, p, \boldsymbol{P}, \boldsymbol{F}) = Conv(\boldsymbol{K}, MLP[\boldsymbol{P} - p] \times [MLP_\delta(\boldsymbol{P} - p), \boldsymbol{F}]) \quad (10)$$

Una vez explicado tanto la entrada del algoritmo, como los paso a seguir, así como la salida final, se va a explicar la motivación detrás de cada uno de los pasos realizados. El operador $\chi - Conv$ está diseñado para funcionar sobre regiones locales por lo que la salida final no debería ser dependiente de la posición absoluta de p y sus vecinos, sino de su posición relativa. De ahí la justificación lógica para ejecutar el paso 1. Para que este paso quede aún más claro se ha realizado la Figura 68, en ellas se muestran 3 nubes de puntos con exactamente la misma forma, pero trasladadas por el espacio. El punto rojo simularía el punto seleccionado para concatenar las características, lo que sería el punto p en el algoritmo. Debajo de las nubes de puntos están las coordenadas por cada punto, en la parte de la izquierda sus valores absolutos y en la parte de la derecha sus valores relativos respecto al punto p, el de color rojo.

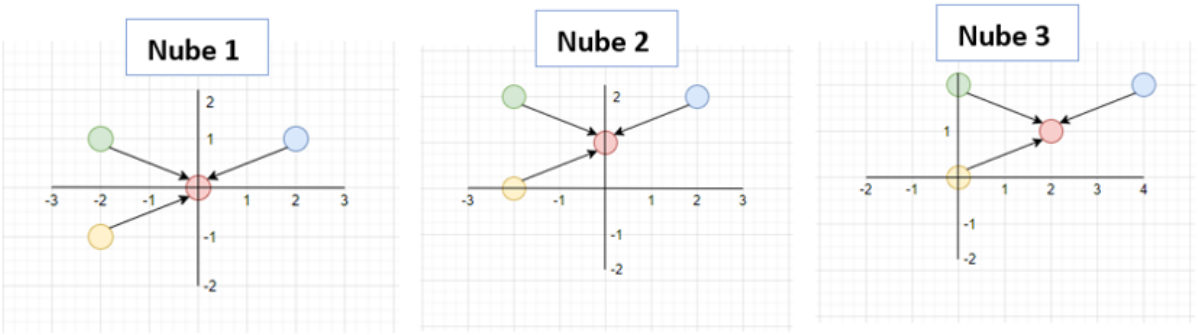

Figura 68. Misma nube de puntos trasladada por el espacio.

POSICIONES ABSOLUTAS			
Puntos \ Nubes	**Nube 1**	**Nube 2**	**Nube 3**
Punto azul	(2, 1)	(2,2)	(4,2)
Punto verde	(-2, 1)	(-2, 2)	(0, 2)
Punto amarillo	(-2, -1)	(-2, 0)	(0, 0)
Punto rojo	(0, 0)	(0, 1)	(2, 1)

Tabla 5. Posición absoluta de los puntos en las diferentes nubes de la Figura 67.

POSICIONES RELATIVAS			
Puntos \ Nubes	**Nube 1**	**Nube 2**	**Nube 3**
Punto azul	(2, 1)	(2, 1)	(2, 1)
Punto verde	(-2, 1)	(-2, 1)	(-2, 1)
Punto amarillo	(-2, -1)	(-2, -1)	(-2, -1)
Punto rojo	(0, 0)	(0, 1)	(2, 1)

Tabla 6. Posición relativa de los puntos en las diferentes nubes de la Figura 67.

Si nos basásemos en las posiciones absolutas, las tres nubes de puntos tendrían valores diferentes, lo cual no sería lo correcto, ya que, al ser la misma nube de puntos trasladada, queremos que el comportamiento de nuestra red sea el mismo en los tres casos. Sin embargo, si calculamos las posiciones relativas, la posiciones de los puntos verde, amarillo y azul son las mismas, aunque la nube de puntos este trasladada.

A la hora de elegir los vecindarios *PointCNN* cuenta con dos opciones, elegir los K vecinos más cercanos en caso de que las nubes de puntos sean uniformes, con una densidad homogénea, o definir un radio alrededor del punto y coger los K puntos de manera aleatoria que caigan dentro de ese volumen. Este último método está pensado para aplicarse sobre nubes de puntos con densidad heterogénea.

Volviendo a los pasos del algoritmo $\chi - Conv$, las posiciones relativas de los puntos, $\boldsymbol{P'}$ **,** junto a sus características, $\boldsymbol{F}$**,** es lo que define las características de salida, $\boldsymbol{F}_p$. Sin embargo, las coordenadas locales tienen una dimensionalidad y una representación diferente a las demás características. Para hacer frente a este problema se "elevan" las coordenadas a un espacio de mayor dimensionalidad con una representación más abstracta, lo que sería el paso 2 del algoritmo. Para ello se utiliza un perceptrón multicapa denominado MLP_δ, esta idea está basada en los MLPs que aplica *PointNet.* En el paso 3 simplemente se concatenan las características. En caso de que no hubiera características extra a parte de las coordenadas la matriz $\boldsymbol{F}$estaría vacía por lo que no se utilizaría. *PointCNN* no tiene inconvenientes en trabajar solo con las coordenadas, al igual que los dos modelos comentados anteriormente, *PointNet* y *PointNet ++*.

A diferencia del *PointNet*, estas características no se procesan usando una función simétrica, sino aplicando la transformación χ la cual ha sido aprendida usando conjuntamente todos los vecindarios. Esta transformación pondera y permuta las características. **La transformación χ es dependiente del orden de los puntos**, esto es necesario ya que se supone que χ permuta $\boldsymbol{F}_*$ de acuerdo con la nube de puntos de entrada, por lo que debe ser consciente del orden inicial de los puntos.

4.3.2.3.Arquitectura PointCNN

En esta sección se analizará la arquitectura del *PointCNN*, para ello se utilizará como apoyo la Figura 69. En esta Figura se puede observar 3 arquitecturas, las arquitecturas a, b son para tareas

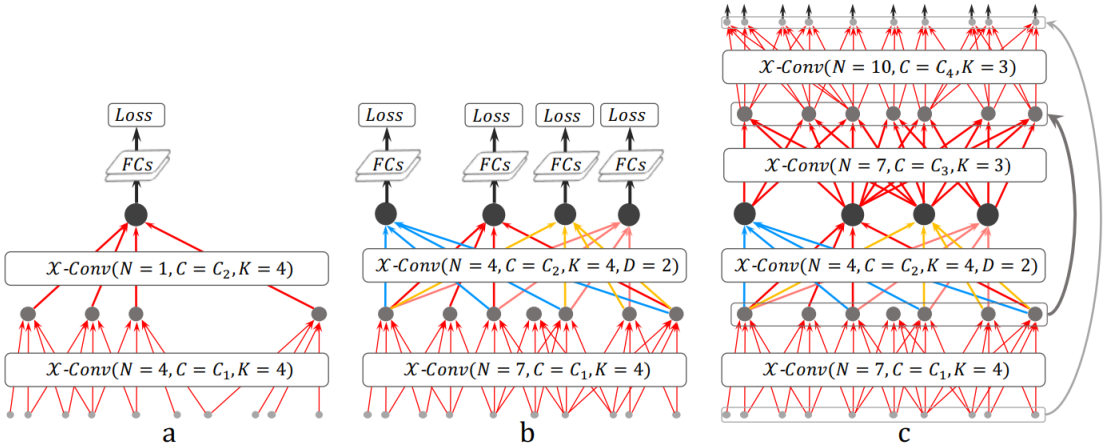

Figura 69. Arquitectura PointCNN. [fuente]

de clasificación, mientras que la más compleja, la arquitectura c, esta diseñadas para tareas de segmentación. En cada una de las arquitecturas aparecen una serie de letras, $\{N, C, K, D\}$,que tiene los siguientes significados:

- N: Número de puntos representativos después de aplicar el operador $\chi - Conv$.
- C: Dimensionalidad (número de características, número de canales) de los puntos representativos después de aplicar el operador $\chi - Conv$.
- K: Número de vecinos a considerar como vecindario para calcular los puntos representativos.
- D: Grado de dilatación para el operador $\chi - Conv$.

La **arquitectura $\boldsymbol{a}$** muestra un *PointCNN* relativamente simple, con solo dos capas $\chi - Conv$ que transforman gradualmente los puntos de entrada en menos puntos, pero con características más detalladas. Después de la segunda capa $\chi - Conv$ queda un único punto, el cual tiene agregada la información de todos los puntos de la capa anterior. La proporción entre el número de vecinos a considerar, K, y el número de puntos de la capa anterior, N, se define mediante el termino *receptive field.* Es decir, $receptive\ field = K/N$. Haciendo uso de esta definición, se concluye que el último punto ve todos los puntos de la capa anterior, ya que tiene un *receptive field* de 1.0. Este último punto tiene una visión global de la nube de puntos inicial, así como características con mucha información que permiten tener un buen entendimiento de la geometría de los datos.

La arquitectura a reduce mucho el número de puntos a lo largo de la red, lo que se puede traducir en un mal rendimiento sobre escenas complejas. Por ello se plantea la **arquitectura $\boldsymbol{b}$** la cual posee conexiones más densas que mantienen más puntos representativos entre capas. De tal forma que la arquitectura b entre la capa uno y la dos tiene 7 puntos representativos en vez de 4 como ocurre en la arquitectura a. El objetivo es mantener la profundidad de la red al mismo tiempo que mantenemos el crecimiento del *receptive field*, de tal manera que los puntos representativos de capas más profundas vean cada vez porciones más grandes.

Sin embargo, si incrementamos el número de puntos por capas, N, mientras que mantenemos el número de vecinos a la hora de calcular los vecindarios, K, el *receptive field* se irá reduciendo. Para evitar esto, los autores del *PointCNN* se apoyan en la idea de convoluciones dilatadas [26], técnica que ya se aplica sobre imágenes, la cual consiste en "inflar" el kernel añadiéndole espacios entre sus elementos. Esta idea queda reflejada de manera gráfica en la Figura 70.

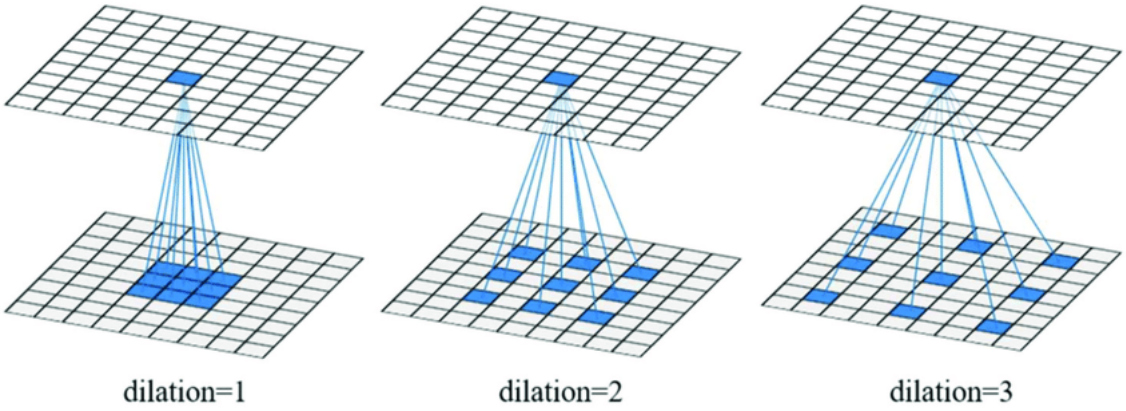

Figura 70. Representación gráfica de la dilatación en convoluciones

De esta manera se incrementa el *receptive field* sin un gran incremento en el número de parámetros. Apoyándose en esta idea los autores introducen el parámetro D, el ratio de dilatación, de tal manera que en vez de seleccionar siempre los K vecinos más cercanos como vecindario, se seleccionan K vecinos de un total de $K\ x\ D$ vecinos. Ahora el *receptive field* se calcula como $(K * D)/N$, consiguiendo así mantener el *receptive field* con más puntos por capa, pero sin incrementar el parámetro K o el tamaño del kernel. En la arquitectura b la segunda capa tiene un ratio de dilatación de $D = 2$, de tal forma que los puntos representativos restantes sean capaces de ver una amplia porción de la nube de puntos. En la última capa de la arquitectura b hay 4 puntos representativos en vez de uno. En este caso se hace la media de todos los puntos representativos antes de aplicar el *softmax*, devolviendo al final la clase que representa la nube de puntos.

Una vez explicadas las arquitecturas de clasificación vamos a centrarnos en la arquitectura de segmentación, la **arquitectura c**. Para tareas de segmentación se necesita tantas salidas (las clases predichas por el modelo) como número de puntos de entrada. Para ello los autores del *PointCNN* se basan en la arquitectura *Conv-DeConv* [27]. Este tipo de arquitectura muy utilizada en segmentación de imágenes tiene 2 partes muy bien diferenciadas. La parte *Conv* se encarga de extraer y agregar las características de los datos de entrada, mientras que la parte denominada como *DeConv* se encarga de extrapolar esta información a la resolución original. Se puede ver un ejemplo de la forma de este tipo de arquitectura en la Figura 70.

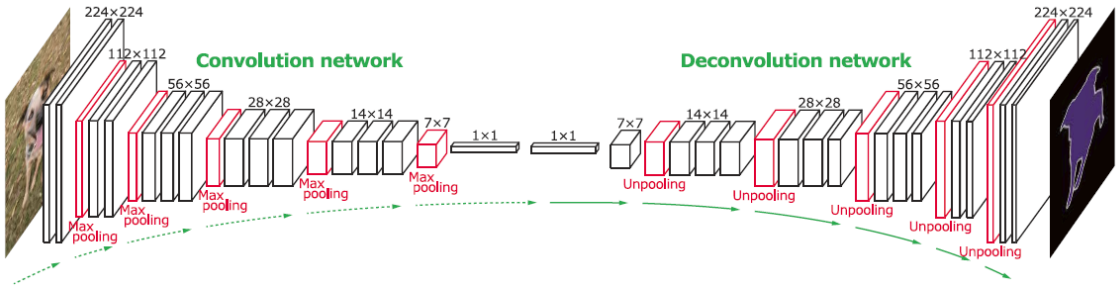

Figura 71. Ejemplo de arquitectura Conv-DeConv sobre segmentación de imágenes. [fuente]

En *PointCNN* tanto la parte *Conv* como la *DeConv* son el mismo operador $\chi - Conv$. Las diferencias entre ambas es que la parte *DeConv* maneja más puntos, pero menos características, menos canales. Para que la parte *DeConv* funcione correctamente la arquitectura c tiene definidos una serie de links (flechas grises en la Figura 69, en la arquitectura c) que concatenan las características de salida con las características de salida de capas anteriores. Es la misma idea que aplica el *PointNet ++* con sus *skip links* (líneas discontinuas en la Figura 60).

Para evitar el sobre entrenamiento, *PointCNN* aplica tanto un *receptive field* menor a 1 en la última capa $\chi - Conv$, para que los puntos representativos vean información parcial, así como *dropout* antes de la última capa *fully connected*. *Dropout* es una técnica muy utilizada en modelos de *Deep Learning* que consiste el suprimir de manera aleatoria ciertas conexiones entre neuronas para evitar el sobre entrenamiento. En la Figura 72 se puede ver una representación gráfica de esta técnica.

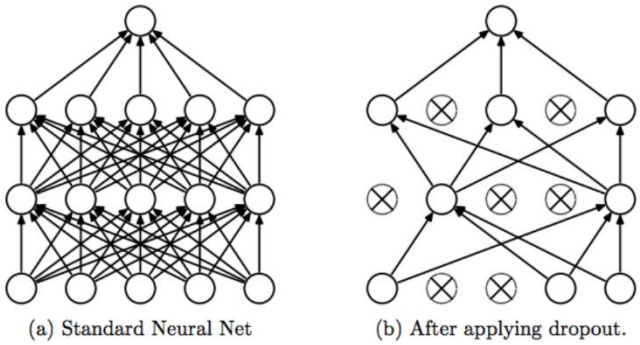

Figura 72. Representación gráfica de la técnica de dropout. [fuente]

Como se ha podido ver en esta sección, *PointCNN* extrapola las técnicas y arquitecturas típicas de las CNNs en imagen sobre nubes de puntos. Para conseguir este objetivo *PointCNN* define un operador propio, $\chi - Conv$, que, a pesar de no ser ideal, consigue mitigar los problemas que supone aplicar convoluciones sobre estos datos, los cuales son conjuntos desordenados de puntos sin una estructura fija. De esta manera, apoyándose también en ideas aportadas por otros modelos como el *PointNet,* los autores del *PointCNN* consiguen definir un modelo capaz de aprender y resolver tareas de clasificación / segmentación sobre nubes de punto, consiguiendo mejores resultados que otros modelos centrados en resolver la misma tarea como el *PointNet ++*.

4.4.KPConv

El último modelo que se presenta en esta sección es *el Kernel Point Convolution*, conocido como *KPConv* [28].Este modelo fue presentado en 2019 y al igual que *PointCNN* aplica convoluciones sobre nubes de puntos, sin embargo, su base es diferente. Mientras que *PointCNN* plantea el aprendizaje de una transformación, $\chi - Conv$, que pondere y permute la nube de puntos de entrada para ser más robustos ante permutaciones en los datos y poder aplicar así las convoluciones típicas de imagen, *KPConv* usa directamente nubes de puntos como kernels (filtros) a la hora de aplicar las convoluciones. Es decir, en vez de intentar adaptar los datos, se adapta la propia convolución. En esta sección se verá cómo funciona el operador de convolución del *KPConv,* sus dos versiones que varían en función de si sus kernels son fijo o no y, para finalizar, se hablará de su arquitectura.

4.4.1.Point Convolution

KPConv está claramente inspirado en las convoluciones que se aplican sobre imágenes, sin embargo, al igual que los datos de entrada no son imágenes, sus kernels (filtros) tampoco lo son. En vez de estar formados por pixeles, están formados por puntos. Esta idea está representada en la Figura 73. En esta figura podemos ver como sobre la nube de puntos de entrada, en este caso los puntos grises, se aplica un kernel el cual está formado por puntos que tiene asociado unos pesos (*filter values*).

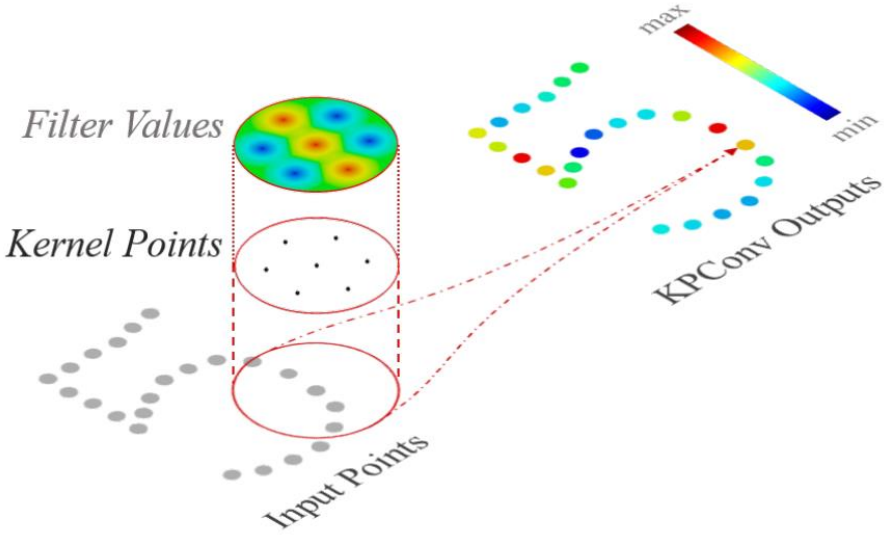

Figura 73. Ejemplo gráfico de nube de puntos como kernel a la hora de aplicar la convolución, [fuente]

Sin embargo, al no haber un grid definido, aplicar la convolución no es algo trivial, ya que no hay una relación claramente establecida entre los puntos de entrada y los puntos pertenecientes al kernel. Para explicar este proceso es necesario definir este tipo de convolución. La *point convolution* de F por el kernel g en un punto x queda definida en la Ecuación 11.

$$(F * g)(x) = \sum_{x_i \in \mathcal{N}_x} g(x_i - x)\, f_i \qquad (11)$$

- F: matriz que representa las características de los puntos de la nube de entrada. $F \in \mathbb{R}^{N \times D}$ donde N es el número de puntos de la nube de entrada y D es el número de características, de ahí que la dimensión de F sea $N \times D$.
- $\mathcal{P}$: matriz que representa las coordenadas de los puntos de entrada. $\mathcal{P} \in \mathbb{R}^{N \times 3}$ donde N es el número de puntos de la nube de entrada y 3 es el número de elementos para representar la posición de un punto en el espacio (XYZ), de ahí que la dimensión de $\mathcal{P}$ sea $N \times 3$.
- x_i: posición (coordenadas) del punto i.
- f_i: características asociadas al punto i.
- g: función kernel
- $\mathcal{N}_x$: representa el vecindario del punto x para un radio r. Su definición formal queda reflejada en la Ecuación 12.

$$\mathcal{N}_x = \{\, x_i \in \mathcal{P} \mid \|x_i - x\| \leq r\} \qquad (12)$$

La parte fundamental de la Ecuación 11 es la definición de la función g, que es la base del *KPConv*. La función g recibe como entrada el vecindario de x, centrado respecto a este punto, lo cual se nombrará a partir de ahora como y de tal forma que $y_i = x_i - x$. Como el vecindario queda definido por un radio r, el dominio de g queda definido por la esfera $\mathfrak{B}_r^3$, donde r es el radio de la esfera. Esto queda reflejado en la Ecuación 13.

$$\mathfrak{B}_r^3 = \{y \in \mathbb{R}^3 \mid \|y\| \leq r\} \qquad (13)$$

Al igual que ocurre en las convoluciones aplicada sobre imagen, se busca que la función g aplique diferente pesos a diferentes áreas dentro de su dominio. Definiendo los puntos del kernel como $\{\tilde{x}_k \mid k < K\} \subset \mathfrak{B}_r^3$ y sus pesos como $\{W_k \mid k < K\} \subset \mathbb{R}^{D_{in} \times D_{out}}$ la función kernel g para cualquier punto $y_i \in \mathfrak{B}_r^3$ queda definida en la Ecuación 14.

$$g(y_i) = \sum_{k<K} h(y_i, \tilde{x}_k)\, W_k \qquad (14)$$

Donde la función h representa la correlación entre uno de los puntos de kernel, $\tilde{x}_k$, y uno de los puntos de entrada, y_i. El valor que devuelva esta función de correlación debería ser mayor cuanto más cerca este $\tilde{x}_k$ de y_i. La función de correlación h queda definida en la Ecuación 15.

$$h(y_i, \tilde{x}_k) = \max\left(0, 1 - \frac{\|y_i - \tilde{x}_k\|}{\sigma}\right) \qquad (15)$$

Donde el parámetro σ, denominado por los autores del *KPConv* como *"the influence distance of the kernel points"*, es un parámetro que se define en función de la densidad. De esta forma, queda bien definida la convolución que aplica *KPConv.* Para aclarar el funcionamiento de esta convolución en la Figura 74 se muestra una convolución normal sobre imagen, mientras que en la Figura 75 se muestra la Convolución del *KPConv* sobre una nube de puntos. A la hora de compararlas, se puede apreciar que en el caso de la convolución sobre imagen cada píxel se multiplica por los pesos del kernel, W_k, ya que cada pixel se alinea perfectamente con los valores del kernel gracias a que la imagen tiene una estructura limitada y bien definida.

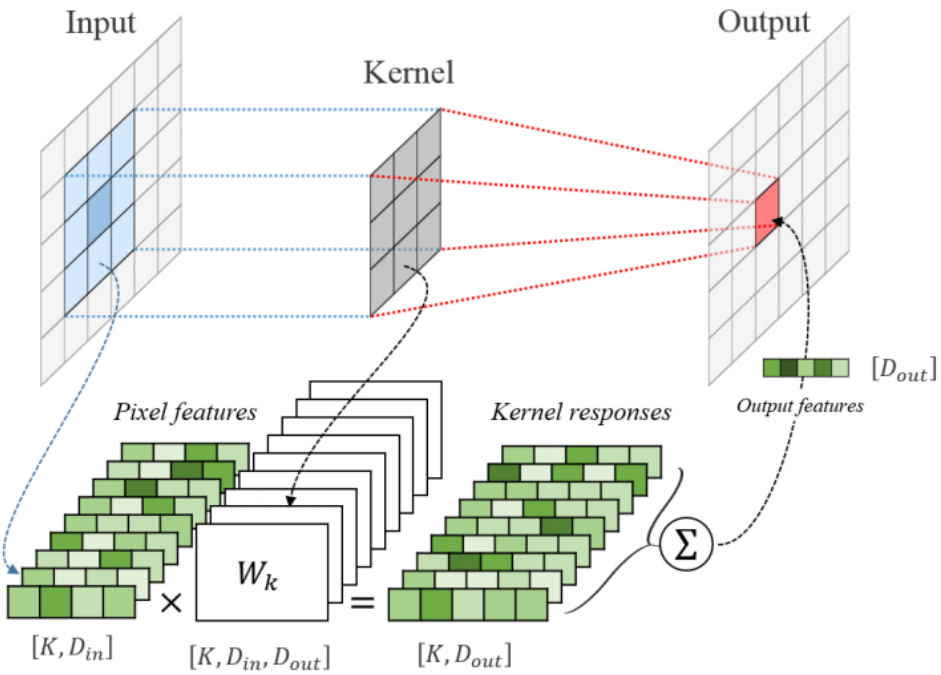

Figura 74. Convolución sobre imagen. [fuente]

Además, por cada píxel se tiene en cuenta un número de vecinos fijo, el cual queda definido por el número de pixeles que forman el kernel. El valor K en la Figura 74.

En el caso del *KPConv* tanto la entrada como el kernel son nubes de puntos. Aquí, los puntos de entrada no están alineados con los puntos del kernel, y el número de puntos puede variar, representado con un ? en la Figura 75. Debido a esto las características de cada punto, f_i, se multiplican por todos los puntos del kernel con un coeficiente de correlación h_{ik}, el cual, siguiendo la Ecuación 15, depende de la posición relativa de los puntos del kernel. De esta forma, aunque los puntos de entrada y el kernel no estén alineados se puede aplicar la convolución, además, gracias a como está planteada la función de correlación, el peso de cada uno de los kernels en cada punto varía en función de su proximidad, imitando el comportamiento de la convolución en imágenes.

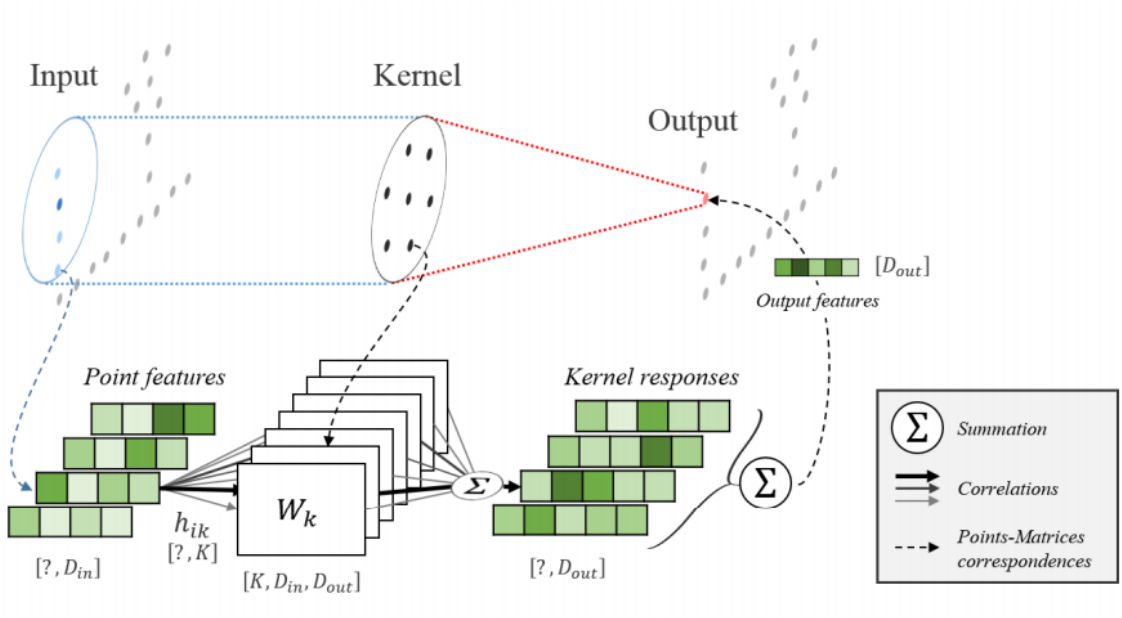

Figura 75. Convolución del KPConv sobre una nube de puntos. [fuente]

Aunque la convolución que aplica el *KPConv* se ha explicado y comparado contra la convolución típica sobre imagen, queda por definir cómo se distribuyen los puntos de un kernel en la convolución del *KPConv.* Los autores del *KPConv* deciden estructurar los puntos del kernel como si estuvieran resolviendo un problema de optimización donde cada punto ejerce una fuerza de repulsión sobre los demás, mientras que al mismo tiempo se fuerza a que los puntos queden dentro de una esfera con una determinada fuerza atractiva. Uno de ellos debe estar colocado en el centro. En la Figura 76 se muestra gráficamente los kernels dependiendo del número de puntos que lo forman.

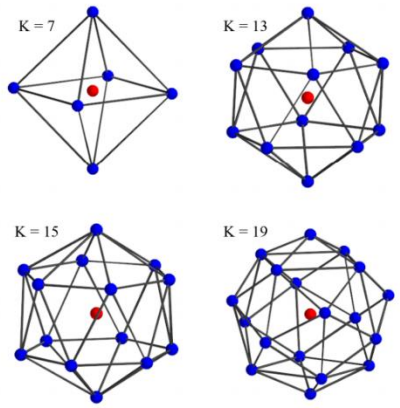

Figura 76. Forma de los kernels en función del número de puntos en el KPConv. [fuente]

Que los puntos del kernel sean fijos puede suponer un factor limitante, por ello, los autores presentan una versión modificada del *KPConv* original que permite aprender la posición de los puntos del kernel, consiguiendo mayor adaptación al evitar que los kernels sean fijos. La versión con puntos fijos en los kernels se denomina *KPConv Rigid* mientras que la otra versión de denomina *KPConv Deformable.*

4.4.2.KPConv Deformable

La idea detrás de esta versión es la misma que el *KPConv Rigid* con la diferencia de que la posición de los puntos que conforman el kernel se puede aprender y por ello modificar. En primer lugar, se parte de una distribución similar a la que partiría el *KPConv Rigid*. La función g representada en la Ecuación 14, es diferenciable respecto a $\tilde{x}_k$ lo que significa que los puntos del kernel son parámetros que se pueden llegar a aprender. Para aplicar esta idea, lo que se hace es generar un delta para cada convolución local. La nueva operación de convolución queda definida en la Ecuación 16.

$$(F * g)(x) = \sum_{x_i \in \mathcal{N}_x} g_{deform}(x_i - x, \Delta(x))\, f_i \qquad (16)$$

Esta ecuación es similar a la Ecuación 11, con la diferencia de que incluye un nuevo parámetro, $\Delta(x)$. En la Ecuación 17 se define la nueva función g_{deform}.

$$g_{deform}(y_i, \Delta(x)) = \sum_{k<K} h(y_i, \tilde{x}_k + \Delta_k(x))\, W_k \qquad (17)$$

Lo que se está intentando hacer queda reflejado de manera gráfica en la Figura 77, en donde se aplica en primer paso un kernel fijo y en función de lo que aprende la red se modifica la posición de los puntos (*local shifts*), para adaptarse mejor a la entrada.

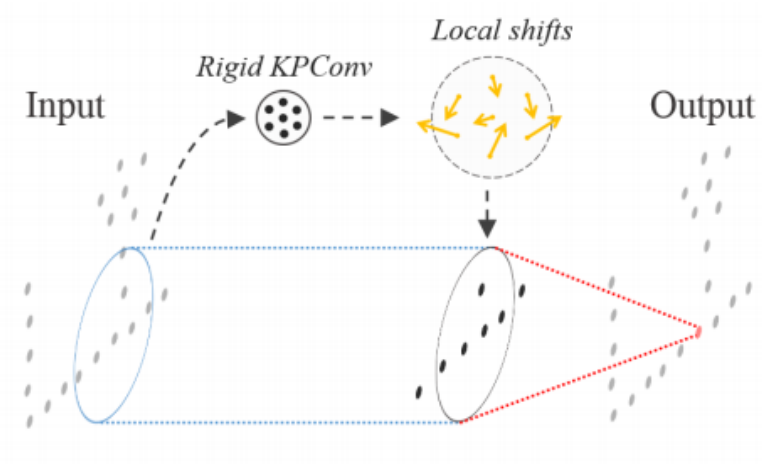

Figura 77. Ilustración sobre el funcionamiento de la convolución deformable en el KPConv. [fuente]

En este caso los autores del *KPConv* intentaron aplicar directamente la idea de convolución deformable que ya se utiliza en imagen, sin embargo, esta adaptación directa no daba buenos resultados. La modificación de la posición de los kernels acababa alejando a los puntos del kernel

de los puntos de entrada. Este fenómeno se debe a la propia naturaleza de la nube de puntos, las cuales están formadas por puntos dispersos, mientras que las imágenes no sufren este problema al no tener esta dispersión en los pixeles. Como las modificaciones en las posiciones de los puntos del kernel las predice la red, en caso de que un punto se aleje mucho de los puntos de entrada el valor $\Delta_k(x)$ será nulo, por lo que no se modificará su posición quedando así alejado de los puntos de entrada de manera permanente. En la práctica acaba ocurriendo sobre muchos puntos. Para intentar solventar este problema, se propone una regularización, $\mathcal{L}_{reg}$, dividida en dos partes, una que penalice la distancia entre los puntos del kernel y su vecino más cercano en el kernel, $\mathcal{L}_{fit}$, mientras que la segunda regularización, $\mathcal{L}_{rep}$, se centra en añadir cierta repulsión entre los pares del kernel (un punto del kernel y su vecino más cercano) para evitar que colisionen. Es decir, se quiere evitar ese alejamiento de los puntos, pero, al mismo tiempo, no se quiere que la penalización sea tan grande que haga que los puntos se acerquen demasiado. Todas estas regularizaciones están reflejadas en las Ecuaciones 18, 19 y 20.

$$\mathcal{L}_{reg} = \sum_{x} \mathcal{L}_{fit}(x) + \mathcal{L}_{rep}(x) \quad (18)$$

$$\mathcal{L}_{fit}(x) = \sum_{k<K} min_{y_i} \left(\frac{\|y_i - (\tilde{x}_k + \Delta_k(x))\|}{\sigma} \right)^2 \quad (19)$$

$$\mathcal{L}_{rep}(x) = \sum_{k<K} \sum_{l \neq k} h(\tilde{x}_k + \Delta_k(x), \tilde{x}_l + \Delta_l(x))^2 \quad (20)$$

Gracias a esta regularización las modificaciones en las posiciones de los puntos del kernel son mejores, adaptándose mejor a la nube de puntos de entrada. En la Figura 77 se muestra el mismo kernel sobre la misma escena, pero siendo en el caso de la imagen superior fijo, en la imagen central deformable, pero sin regularización y en la imagen inferior deformable con la regularización aplicada. Como se puede ver, los puntos del kernel deformable sin regularización están demasiado esparcidos y alejados del plano del suelo, mientras que el kernel deformable con regularización se adapta mucho más al plano.

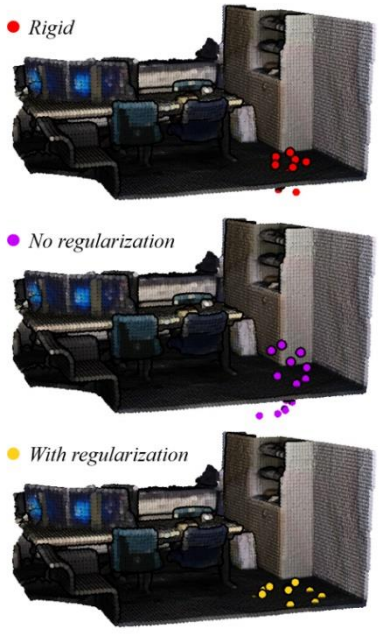

Figura 78. Diferentes tipos de kernel sobre la misma escena. [fuente]

4.4.3.Arquitectura KPConv

La arquitectura del *KPConv* tiene muchas similitudes con las arquitecturas del *PointNet ++* y del *PointCNN*. Para analizarla se va a utilizar como apoyo la Figura 79. Por un lado, sigue la misma estructura que la arquitectura *Conv-DeConv* (arquitectura explicada en la *Sección 4.3.2.3*) con una parte dedicada a la extracción de características concatenada con una parte centrada en extrapolar esas características a la nube de puntos inicial. La dimensión de las características extraídas se va duplicando en cada capa, $64 \rightarrow 128 \rightarrow 256 \rightarrow 512 \rightarrow 1024$, mientras que al mismo tiempo que se reduce el número de puntos, aumentando así el *receptive field*. También tiene una serie de *links* (flechas verdes en la Figura 78) que unen la parte *Conv* con la *DeConv.* Como ocurría en *PointNet++* o *PointCNN*, sirven para concatenar características de capas anteriores con las actuales, paso necesario para extrapolar correctamente las características a los puntos iniciales.

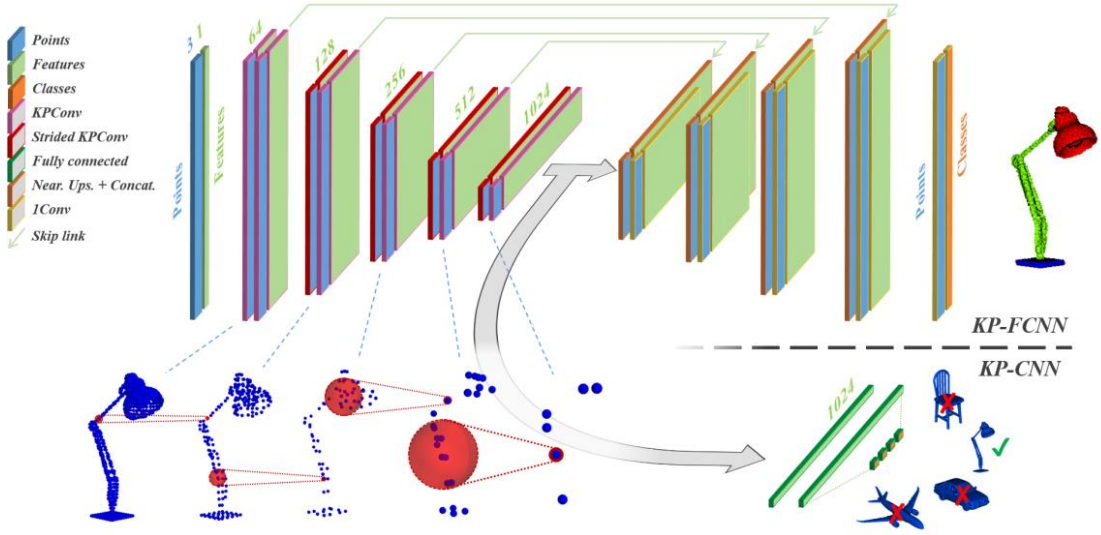

Figura 79. Arquitectura KPConv [fuente]

En cuanto a la técnica de *downsampling* que utiliza *KPConv*, cuya representación está en la parte inferior de la Figura 79, consiste en voxelizar el espacio en cubos de lado l, de tal forma que cada punto pertenece a un *grid* concreto. Los *grids* vacíos se ignoran. A continuación, se calcula el punto medio de cada *grid* y se utiliza ese punto medio como el centro de una esfera de radio r. Todos los puntos que caigan dentro de estas esferas quedarán representados por el punto central en la siguiente capa, de tal forma que el radio de la esfera se va duplicando, englobando cada vez más puntos. De esta manera se asegura una densidad homogénea. Cuanto mayor sea el radio base de la esfera más fuerte será el *downsampling* que se realiza, ya que su radio se va duplicando a lo largo de las capas.

Definiendo su propio operador de convolución, *KPConv* es capaz de aplicar exitosamente convoluciones sobre nubes de puntos, alcanzando mejores resultados que los demás modelos comentados anteriormente sobre los mismos datasets. Además, los autores añaden una versión diferente del algoritmo, el *KPConv Deformable,* que, al permitir que los puntos de los kernels varíen ligeramente su posición para adaptarse mejor a los datos, produce un desempeño aún mejor, sobretodo en dataset muy densos. Todo esto, sumado a su eficiencia y flexibilidad, la cual se debe a unos kernels bien estructurado y a una técnica de *downsampling* que mantiene una

densidad homogénea, hace que este modelo sea uno de la opción más interesante a considerar para tareas de clasificación o segmentación de nubes de puntos.

5.Marco experimental

Como ya se ha comentado en la introducción, el objetivo de este trabajo es segmentar nubes de puntos tomadas mediante LiDAR aéreo utilizando modelos de *Deep Learning*. Para ello, necesitamos realizar varios experimentos los cuales se explican en la *Sección 6.* Sin embargo, antes de realizar los experimentos, necesitamos definir unas bases previas comunes a todos ellos. En esta sección se explicará qué preproceso se aplica a los datos para poder entrenar la red con las nubes de puntos, aspectos concretos de nuestra función de coste inicial, y la definición de unas métricas propias que se utilizaran para evaluar nuestros modelos.

5.1.Preproceso de nuestros datos

Trabajar con nubes de puntos es computacionalmente costoso. La densidad de los datos usados en este proyecto es alta, y los cálculos de vecindarios que realizan los modelos de *Deep Learning* consumen mucha memoria. Los entrenamientos de nuestros modelos se ejecutan sobre GPU, ya que, de esta manera, se acelera tanto el tiempo de entrenamiento como el tiempo de inferencia, siendo casi 10 veces más rápido que ejecutar los mismos cálculos en CPU. Una GPU tiene una memoria asociada, y el tamaño de esta es un factor limitante, es una limitación hardware que condiciona nuestros experimentos.

Es decir, para que nuestro modelo aprenda necesita ejemplos etiquetados, en este caso, nubes de puntos etiquetadas. Estas nubes de puntos deberán tener un tamaño amplio para que exista contexto y deberemos poder procesar varios al mismo tiempo, es decir, utilizar un batch size mayor que uno. Se busca utilizar un batch size superior a uno ya que no queremos que el desempeño sobre un único ejemplo modifique los pesos de la red, queremos que la red vea más ejemplos antes de actualizar los pesos. Como nuestras muestras no tiene ni la misma densidad ni el mismo tamaño, debemos establecer unos parámetros comunes para todas ellas. Para conseguir que los ejemplos que recibe la red tengan ciertos parámetros comunes, como tamaño o densidad, partiendo de muestras heterogéneas, se extraen pequeños ejemplos de las muestras originales que cumplan estos parámetros. Estos ejemplos más pequeños obtenidos de las muestras originales los denominamos ***tiles***. La forma de los tiles extraídos de las muestras es cuadrada.

En la Figura 80 se muestra un ejemplo de extracción de tiles a partir de 2 muestras de Train. Cada una de las muestras tienen tamaños diferentes, la muestra 607_4740 tiene un tamaño de 1 km x 1 km con una densidad de 55 puntos por metro cuadrado. Mientras que la muestra MT_105, la de la parte inferior de la Figura 80, es más pequeña, 220 metros x 150 metros, y menos densa, de 42 puntos por metro cuadrado. Sin embargo, a pesar de que las dos muestras son diferentes, los tiles extraídos de cada una de ellas cumplen el mismo tamaño, 100 metros x 100 metros, y la misma densidad, 20 puntos por metro cuadrado. De esta manera, a pesar de tener muestras diferentes, nos aseguramos de que los ejemplos que se le proporcionan a nuestro modelo cumplen unos parámetros comunes. Para conseguir la densidad deseada, se eliminan punto de manera aleatoria.

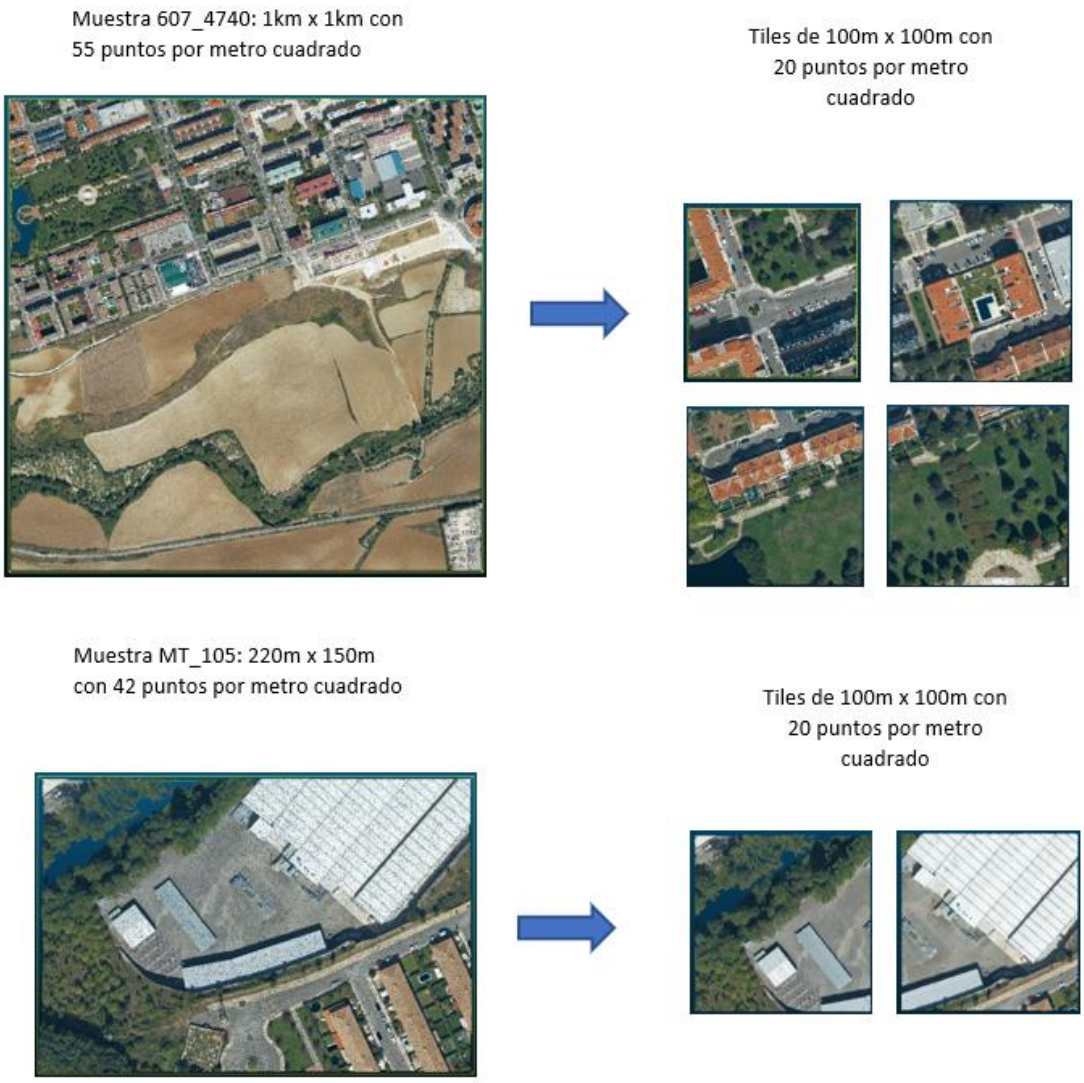

Figura 80. Extracción de tiles con el mismo tamaño y densidad de muestras con tamaños y densidades diferentes

Sin embargo, estos parámetros están acotados por el tamaño de la memoria de la GPU. Cuanto más grande y denso sea un tile, más puntos tendrá. Cuantos más puntos, más memoria se necesita al realizar más cálculos. La GPU de la que se dispone para realizar los experimentos es una *NVIDIA GeForce RTX 2080* con 11GB de memoria.

Los parámetros más importantes que hay que definir para poder entrenar nuestros modelos son tres:

- Batch size
- Tamaño del tile, largo y ancho
- Densidad

Hay que tener en cuenta que estos tres parámetros están relacionados. Esto se ve más claro con la Ecuación 21.

$$batch\ size * desidad * (largo_{tile} * ancho_{tile}) = Nº\ puntos\ que\ deben\ entrar\ en\ GPU \quad (21)$$

Si uno de nuestros parámetros es muy grande, los demás deben decrecer para no exceder el límite de puntos que puede procesar la memoria de la GPU. El objetivo es encontrar un buen balance. Estos valores varían en función del modelo, ya que los cálculos que realizan cada uno de ellos no son los mismos, por lo que el consumo de memoria varía, algunos permiten más puntos totales en GPU y otros menos. En la *Sección 6*, a la hora de definir los experimentos se explicarán los valores elegidos para cada uno de los modelos. Ahora bien, aunque estos valores dependan del modelo utilizado, tanto el proceso de tileado como la normalización de los tiles es siempre la misma independientemente del modelo.

5.1.1.Proceso de tileado

Tal como hemos visto, para poder entrenar adecuadamente nuestros modelos, es necesario alimentar a la red con ejemplos que tengan un tamaño y densidad similar. Para ello, es necesario generar tiles a partir de nuestras muestras. El proceso de tileado consiste en, dada una muestra, extraer tiles de ella. Para ello, es necesario definir un proceso que establezca cómo obtener estos tiles. En un inicio, se planteó un tileado fijo. Este consistía en dada una muestra, generar un tile de tamaño $T \times T$ metros partiendo de la esquina superior izquierda. Este tileado solo se realiza sobre XY, las coordenadas Z no se tienen en cuenta, es decir, todos los puntos que caigan dentro del tile deben cumplir una posición en XY, pero la Z es indiferente. Una vez definido este primer tile, el tile azul en la Figura 81, el segundo no comienza donde acaba el primero, sino que existe un cierto solape B, de tal manera que el tile verde comienza en la posición donde acaba el tile azul menos B. El solape también existe en la Y, de tal manera que el tile amarillo empieza donde acaba el azul menos B .

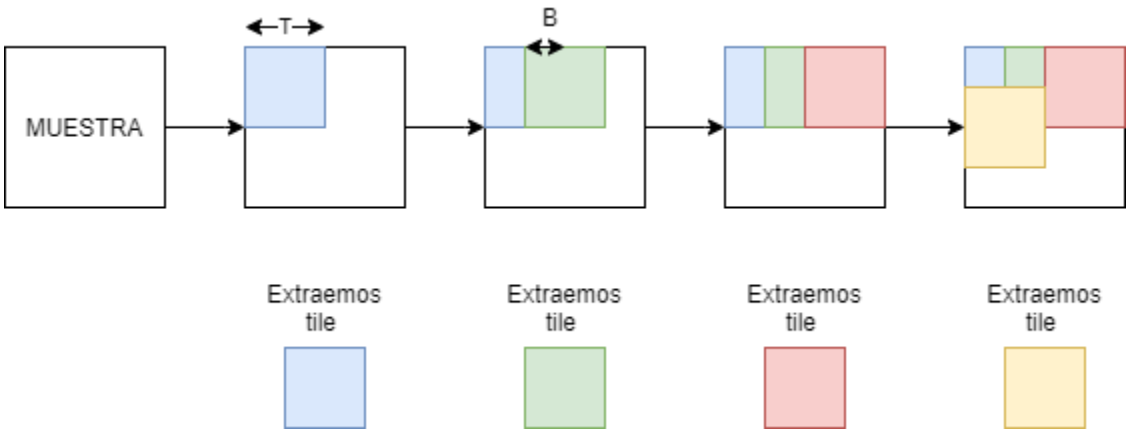

Figura 81. Proceso de tileado fijo

En caso de que el último tile no tenga espacio suficiente se incrementa el buffer. Es decir, imaginemos que en el ejemplo de la Figura 81 la muestra fuera más pequeña y, por ello, el tile rojo no cupiera. En este caso, el tile rojo en vez de utilizar un solape de B unidades, sería mayor, hasta el punto de asegurar que su tamaño final sea de $T \times T$. Así, nos aseguramos de que todos los tiles que vea la red tengan el mismo tamaño.

Esta manera de tilear, al ser un proceso que para una muestra dada siempre devolverá los mismos tiles, no aprovecha toda la variabilidad de la muestra, por lo que este tileado inicial se sustituyó por un tileado completamente aleatorio. Cuando la red pide un batch, se selecciona

por cada elemento del batch una muestra de manera aleatoria, pero con una probabilidad asociada en función del número de puntos que la muestra contenga respecto al total. De tal manera que, una muestra con muchos puntos tiene más probabilidades de ser elegida que una muestra con pocos puntos.

Una vez elegida una muestra, debemos definir un tile. Para ello, se selecciona un punto aleatorio, que será la esquina superior izquierda del tile, y a partir de ese punto se define el punto de la esquina inferior derecha, delimitando así el tile. Sin embargo, no nos vale cualquier punto, ya que el tile debe cumplir con un tamaño mínimo. Si seleccionamos un punto del borde de la muestra, el tile no cumpliría con el tamaño especificado. Como este proceso se repite muchísimas veces a lo largo de un entrenamiento, debe ser lo más eficiente posible. Por ello, antes de lanzar el entrenamiento, se calculan las máscaras de orígenes válidos de cada una de las muestras de entrenamiento. De esta forma, a la hora de elegir un punto aleatorio sobre el cual definir el tile, ya tenemos preprocesados los puntos que si son elegidos nos permitirán establecer un tile del tamaño deseado dentro de la muestra. En la Figura 82 podemos ver de manera gráfica los orígenes válidos de dos muestras de entrenamiento. En amarillo están todos los orígenes válidos, mientras que en morado están aquellos puntos que están dentro de la muestra pero que no serían válidos.

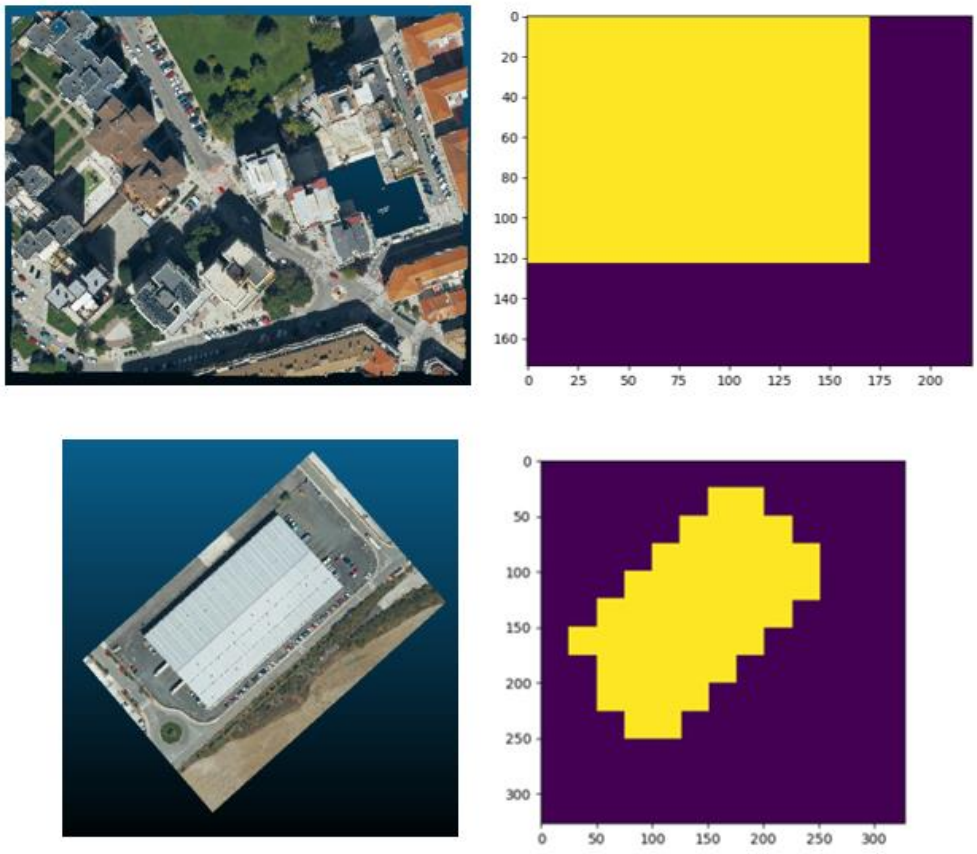

Figura 82. Máscara de orígenes válidos de una muestra de Train, en amarillos los orígenes válidos.

Por tanto, este tileado aleatorio proporciona una gran variabilidad y eficiencia al tener pre-calculadas las máscaras de orígenes válidos. Sin embrago, cuando queramos inferir una muestra,

no podemos utilizar este tileado aleatorio, ya que, en este caso, necesitamos devolver una clase por punto, y un tileado aleatorio no nos asegura que esto vaya a ocurrir. Por ello, el tileado aleatorio solo se utiliza para entrenar el modelo, mientras que en la inferencia se utiliza un tileado fijo con solape, el que se explica al principio de esta sección.

Al haber solape, habrá puntos que pertenezcan a varios tiles. Para predecir una clase para estos puntos replicados, se calcula la media de las probabilidades asignadas a ese punto en los diferentes tiles en los que aparece. En la Figura 83 podemos ver un ejemplo. Para este ejemplo suponemos que hemos entrenado el modelo con tres clases $\{clase\ 1, clase\ 2, clase\ 3\}$. El punto blanco cae dentro del tile amarillo, del tile verde, del tile rojo y del tile morado. Por ello, este punto será predicho 4 veces. Para devolver una predicción final para ese punto, se hará la media de cada una de las probabilidades por clase y se seleccionará la más alta como clase final.

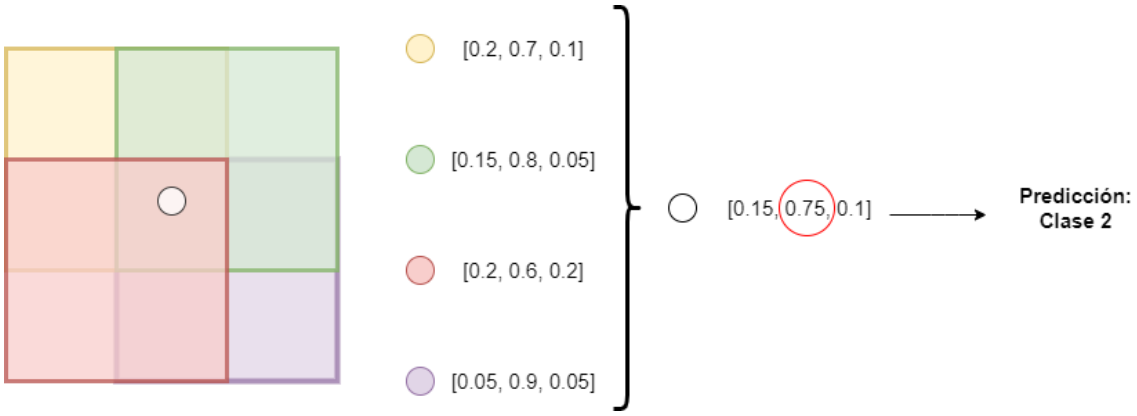

Figura 83, Ejemplo de inferencia de punto contenido en varios tiles

5.1.2.Nomalización de los tiles

Como se ha explicado en la sección anterior, para poder entrenar nuestro modelo con las nubes de puntos del dataset de Train, debido a limitaciones hardware y con el objetivo de asegurar de que todos los ejemplos cumplan unas mismas condiciones (tamaño, densidad, etc.) es necesario aplicar un proceso de tileado a nuestras muestras. Siendo cada uno de esos tiles los ejemplos que usará la red para entrenar. Sin embargo, esto no es suficiente, es necesario aplicar un preproceso sobre nuestros tiles antes de alimentar a nuestro modelo con ellos.

Como ya se mencionó en la *Sección 2.3.1,* la normalización es un aspecto fundamental. Teniendo esto en cuenta, a la hora de normalizar nuestras características hacemos dos distinciones:

- Por un lado, tenemos las coordenadas de cada punto, XYZ
- Por otro lado, están las características extra por punto, como pueden ser la intensidad o los rebotes.

Para las características extra, aquellas que no son las coordenadas de los puntos, se normalizan utilizando la Ecuación 22.

$$\frac{f_i - f_{min}}{f_{max} - f_{min}} = f_{i_{normalized}} = [0,1] \qquad (22)$$

Sea f_i la característica para un punto, se le resta en mínimo y se divide entre la diferencia del máximo y el mínimo de esa característica. De esta manera nos aseguramos de que todos los valores de esta característica estén comprendidos en el intervalo $[0, 1]$. Hay características como el RGB o los rebotes que se mueve en un intervalo bien delimitado. En el caso de los rebotes, por ejemplo, el valor mínimo que puede adquirir es 1 mientras que el máximo es 5. Sin embargo, hay otras características con rangos mucho más amplios debido a la existencia de outliers, como es el caso de la intensidad.

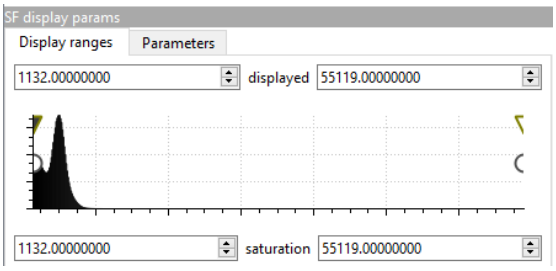

Figura 84. Histograma de intensidad para el bloque 607_4740

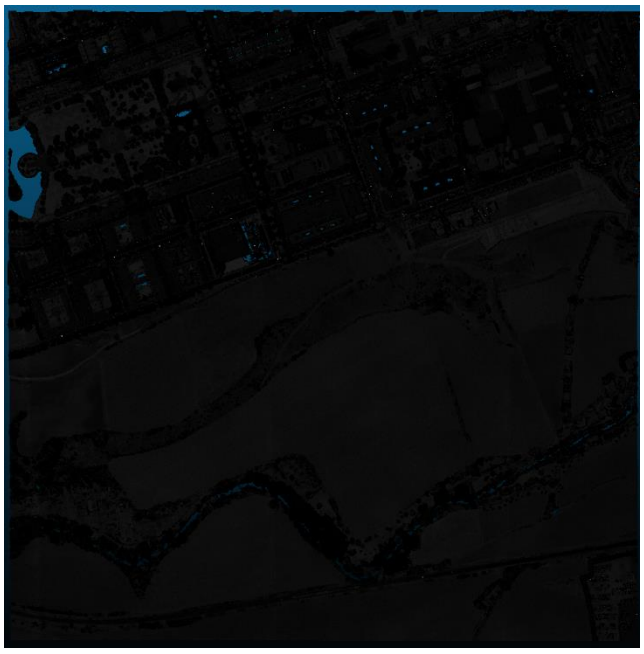

Figura 85. Representación visual de la intensidad para el bloque 607_4740.

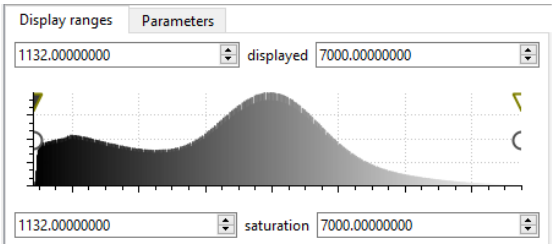

Figura 86. Histograma de intensidad del bloque 607_4740 en el intervalo [1000 - 7000]

Figura 87. Representación visual de la intensidad en el rango [1000 - 7000] para el bloque 607_4740

Como muestra la Figura 84, el intervalo en el que se mueve la intensidad en un bloque grande de 1km x 1km es muy amplio, un intervalo comprendido entre $[0 - 56000]$. Sin embargo, este rango tan amplio se debe a unos pocos puntos que tiene una intensidad exageradamente alta, de tal forma que el 99.9% de los puntos suelen tener intensidades comprendidas entre $[1000 - 7000]$.

En la Figura 86 podemos ver el histograma de intensidad para el mismo bloque, pero en un intervalo reducido. La distribución es mucho más uniforme, y si comparamos la representación visual de la intensidad con todo el rango original, Figura 85, contra la representación de la intensidad en un rango reducido, Figura 87, podemos ver claramente cómo reducir el rango permite identificar mejor las clases que se pretenden predecir en las muestras. Por ello, para normalizar la intensidad seguimos el proceso que reflejan las Ecuaciones 23, 24 y 25. Es decir, corregimos los valores de los outliers, y normalizamos entre los valores de mínimo y máximo elegidos.

$$If\ Intensity_i < 1000 \rightarrow Intensity_i = 1000 \qquad (23)$$

$$If\ Intensity_i > 7000 \rightarrow Intensity_i = 7000 \qquad (24)$$

$$\frac{Intensity_i - 1000}{7000 - 1000} = f_{i_{normalized}} = [0, 1] \qquad (25)$$

Para las coordenadas se utiliza una normalización parecida a la de la Ecuación 22 para los valores de X e Y, sin embargo, en vez de restarle el mínimo se le resta la media. La Ecuación 26 representa esta normalización con las X, lo mismo aplica a las Y.

$$\frac{Coord_x - x_{mean}}{x_{max} - x_{min}} = x_{i_{normalized}} = [-1, 1] \qquad (26)$$

En el caso de la coordenada Z la normalización es ligeramente diferente. Como ya se ha mencionado en el proceso de tileado, el tileado de nuestros tiles delimita claramente el tamaño de las X y de las Y, pero no de las Z, por lo que utilizar la misma normalización para las Z sería un error, vamos a verlo con un ejemplo. Supongamos que se tiene dos tiles, uno de ellos de 50 metros x 50 metros que corresponde a una planicie, de tal manera que sus Z oscilan entre $[0 - 0{,}5]$ metros, mientras que el otro tile, también de 50 metros x 50 metros, representa una zona del centro de Pamplona con algún edificio alto, de tal manera que sus Z oscilan entre $[0 - 30]$ metros.

Si en ambos casos normalizamos la Z entre $[0, 1]$, estaríamos mapeando una diferencia de medio metro al mismo intervalo que una diferencia de 30 metros. Para evitar esto, para normalizar las Z se utiliza la Ecuación 27.

$$\frac{Coord_Z - z_{min}}{tile_{size}} = z_{i_{normalized}} \quad (27)$$

Es decir, a cada Z le restamos la z_{min} del tile, y dividimos entre el tamaño del tile, $x_{max} - x_{min}$ ó $y_{max} - y_{min}$. Siguiendo el ejemplo de antes el $tile_{size}$ sería de 50 metros. De esta manera, el rango de valores para esta característica sería $[0, \frac{z_{max} - z_{min}}{tile_{size}}]$.

Restarle z_{min} a cada punto del tile nos asegura que el valor mínimo que pueda adquirir la $z_{normalizada}$ sea 0. Al mismo tiempo, dividir entre el $tile_{size}$ nos asegura que los valores que adquiera la $z_{normalizada}$ no sean muy grandes. De esta forma, los valores de la $z_{normalizada}$ de cada tile dependerá de cuánto oscile esta coordenada, evitando mapear oscilaciones diferentes a los mismos valores, pero, al mismo tiempo, se evita que el rango de valores en los que se mueve la $z_{normalizada}$ sea muy amplio al dividir la diferencia entre el $tile_{size}$.

5.2.Función de coste

La función de coste utilizada durante los experimentos ha sido la *negative log likelihood, también conocida como cross entropy*. El cálculo de esta función de coste es relativamente simple. La Ecuación 28 muestra cómo se calcula.

$$loss = -\sum_{i=1}^{N} P(x) \log \mathrm{Q(x)} \quad (28)$$

Donde P es la distribución de etiquetas reales y Q el vector de probabilidades por clase predicho por la red. Por cada ejemplo, nos quedamos con la probabilidad de que el ejemplo pertenezca a su clase real. Lo que se busca es que las predicciones sean lo menos ambiguas posibles, es decir, si solo tuviéramos dos clases, queremos que la clase correcta tenga una probabilidad de 1.0 mientras que la clase incorrecta tenga una de 0.

Esta función de coste se encarga de penalizar las probabilidades máximas bajas. Si la probabilidad de pertenecer a la clase real es alta, el valor de la función de coste será bajo, mientras que, si la probabilidad de pertenecer a la clase real es baja, el valor de la función de coste será alto. Los valores que adquiere esta función de coste en función de la probabilidad están reflejados en la Figura 88.

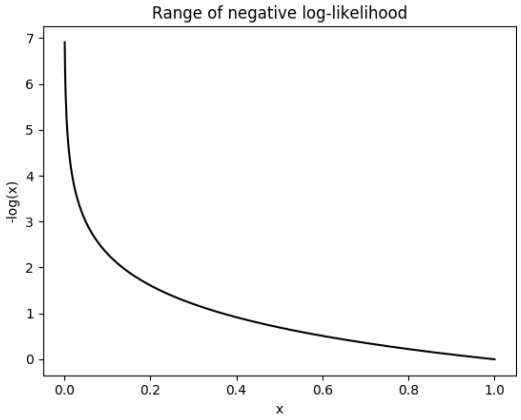

Figura 88. Valores de la función de coste para un ejemplo en función de la probabilidad. [fuente]

Es importante remarcar que **la predicción de aquellos puntos cuya clase original sea la clase no etiquetado no se tienen en cuenta a la hora de calcular la función de coste**. De esta manera, los puntos de la clase no etiquetado se utilizan para los cálculos de vecindarios, pero su predicción no afecta al desempeño del modelo.

5.3.Métricas Propias

Además de las métricas de la Tabla 1 en la *Sección 2.2*, se han desarrollado una serie de métricas específicas para el problema que pretendemos resolver. En nuestro caso, no consideramos todos los fallos igual de graves. Vamos a reflejarlo con un ejemplo. En la Figura 89, el punto blanco representa un punto que pertenece al tejado de la casa, por tanto, la etiqueta real de dicho punto es edificio, pero en este ejemplo vamos a suponer que lo hemos predicho como suelo. La flecha negra representa la distancia mínima de ese punto predicho como suelo al punto más cercano que es en realidad suelo.

Figura 89. Fallo grave, punto de edificio como suelo

Figura 90. Fallo leve, punto de edificio como suelo

En la Figura 90 vemos un caso similar, pero colocando el punto predicho como suelo incorrectamente cerca de la base del edificio. Ambos son fallos, el punto pertenece a la clase edificio y lo estamos prediciendo como suelo. Pero si representamos el DEM de cada uno de ellos vemos claramente la diferencia entre los dos tipos de errores. En la Figuras 91 está representado como un segmento rojo el DEM, es decir, el nivel de la superficie de terreno sin contar obstáculos como edificios o vegetación.Se ve claramente la tremenda diferenecia que existe entre el DEM real, caso 1, y el DEM del fallo grave, caso 3.

Figura 91. DEM en función de la gravedad del error.

Teniendo esto en mente, usamos una serie de métricas para reflejar la gravedad de los errores cometidos, que se explican en la Tabla 7:

Nombre	Descripción
Suma total puntos criticos	Un punto P perteneciente a una clase distinta a NC para el que se predice la clase NC, es crítico, si la distancia entre el punto P y el punto (Q) más cercano a P de la clase NC es superior a 10 metros. $$Critico = dist(P,Q) > 10$$ Lo que intentamos transmitir con esta medida es el número de fallos graves que cometemos. Un fallo grave es aquel en el que clasificamos un punto erróneamente y el punto más cercano de la clase con la que lo confundimos está a más de 10m de él.
Micro Average Distance	Un punto P perteneciente a una clase distinta a NC para el que se predice la clase NC, se calcula la distancia de P, con el punto más cercano a él, denotado como Q, cuya clase es NC.Calculamos la media de todas estas distancias: $$Micro\ Average\ Distance = \frac{\sum_{i=0}^{N} dist(P_i, Q_i)}{N}$$ $$N = número\ de\ ejemplos\ fallados$$ Esta métrica nos permite ver la gravedad de nuestros fallos en general. Cuanto más elevada sea, más grave serán nuestros fallos. Una distancia pequeña indicaría que los fallos son leves, tenemos cerca un vecino de la clase con la que hemos clasificado erróneamente el ejemplo. Lo contrario pasaría con una distancia alta.

Micro Cuadratic Distance	La idea es la misma que la de micro average distance, pero en vez de agregar todas las distancias haciendo la media, usamos la media cuadrática (penaliza más las distancias lejanas): $$Micro\ Cuadratic\ Distance = \sqrt{\frac{\sum_{i=0}^{N} dist(P_i, Q_i)^2}{N}}$$ $$N\ =\ número\ de\ ejemplos\ fallados$$
Distancia media por clase	La misma idea que la Micro Average Distance pero sin hacer la media, sacando una métrica de distancia por clase.

Tabla 7. Métricas propias que reflejan la gravedad de los errores.

6.Estudio Experimental

Una vez explicados los modelos de *Deep Learning*, el preproceso que se aplica a nuestros datos tanto para entrenar como para inferir, así como las métricas a utilizar para evaluar nuestros modelos, se procede a realizar una serie de experimentos para segmentar nuestras nubes de puntos de manera automática. El objetivo de esta sección es conseguir el mejor modelo posible, aquel que, a la hora de ser evaluado sobre el dataset de Test, obtenga las mejores métricas.

6.1.Pruebas base con diferentes modelos de Deep Learning

En esta sección se realizará una prueba base con cada modelo de *Deep Learning* explicado en la *Sección 4.* El objetivo es comprobar cuál de los modelos nos proporciona los mejores resultados a la hora de segmentar las nubes de puntos.

Aunque la parametrización de los diferentes modelos de segmentación difiere bastante entre sí, es necesario establecer unos parámetros comunes a todos ellos para que la comparación de métricas sea lo más justa posible. Los parámetros comunes tanto para entrenar como para inferir con los diferentes modelos son los siguientes:

- Los datos de entrenamiento y de test son los mismos para todos los modelos. Las clases a predecir son las mismas para todos los modelos.

- Las características extra a utilizar, a parte de las coordenadas, son la intensidad y los rebotes (ReturnNumber y NumberOfReturns). Todos los modelos usan las mismas características extra.

- Los modelos se han entrenado durante 15 épocas, con tiles aleatorios. Se ha establecido 1000 tiles aleatorios por época. Se ha utilizado un batch size de 4. En cuanto al tamaño de tile, es de 50 x 50 metros para asegurar que haya un mínimo de contexto. El número de puntos por tile depende del consumo de memoria de GPU que haga cada modelo.

- Para entrenar se aplican tiles aleatorios, pero para inferir se calculan tiles fijos con cierto solape, como ya se ha explicado en la *Sección 5.1*. A la hora de inferir se necesita predecir una clase a cada punto de entrada. Imaginemos que nuestros tiles son de 50 metros x 50 metros con una densidad de 10 puntos por metro cuadrado. Supongamos que la muestra a inferir tiene una densidad de 50 puntos por metro cuadrado. El resultado de la inferencia tendrá menos puntos que la muestra original. Para solventar este problema, los puntos que no han entrado en la inferencia se clasifican por vecino más cercano. Este proceso es igual para todos los modelos.

6.2.PointNet ++

Como ya se ha mencionado en la *Sección 4.2*, el PointNet ++ puede considerarse el hermano mayor del PointNet. Los autores de estos dos modelos son los mismos, los cuales, afirman que el PointNet ++ es mejor en tareas de segmentación que el PointNet, ya que este modelo aplica PointNet a diferentes escalas, siendo así capaz de aprender mejor las estructuras locales, así como las globales. Por ello, la primera prueba base realizada es sobre el PointNet ++, no sobre el PointNet.

En cuanto a los parámetros de este modelo, se utilizaron los valores especificados en el paper del PointNet ++:

- Dropout = 0.5
- Radios = [0.2, 0.4]. Radio de 0.2 para la parte 0 de la Figura 61, radio de 0.4 para la parte 1 de la Figura 60.
- Ratios = [0.2, 0.25]. Ratio de 0.2 para la parte 0 de la Figura 61, ratio de 0.25 para la parte 1 de la Figura 61.
- Número máximo de puntos a considerar por radio = [64, 64]. 64 puntos para la parte 0 de la Figura 61, 64 puntos para la parte 1 de la Figura 61.

Los tiles de 50 x 50 metros tenían una densidad de 10 puntos por metro cuadrado para que los cálculos no desbordaran la memoria de la GPU. Después de entrenar el modelo durante 15 épocas, entrenamiento que duró unas 15 horas, las métricas sobre el conjunto de Test se muestran en las Tablas 8 y 9.

Acc	Tpr	f1 2	f1 3	f1 4	f1 6	f1 17	f1 10	Mean f1
86.91	42.46	91.3	0	82.83	73.43	4.55	0	42.74

Tabla 8. Métricas generales sobre Test.

Mean dist.	C. Mean dist.	Dist. 2	Dist. 3	Dist. 4	Dist. 6	Dist. 17	Dist. 10	Nº críticos
8.28	13.09	5.63	0	8.46	25.63	10.76	0	2337957

Tabla 9. Métricas de distancia de error sobre Test.

Las clases mayoritarias como suelo (clase 2) , edificio (clase 6) y vegetación media / alta (clase 4) obtienen resultados aceptables. Sin embargo, el modelo no llega a aprender las clases vegetación baja (3), ruido sobre el suelo (10) y vehículos (17). Esto se debe a dos factores, uno de ellos es la baja proporción de estas clases respecto a las demás, la segunda es la baja densidad utilizada para entrenar el modelo debido a las limitaciones hardware.

En cuanto a las métricas de distancias que reflejan la gravedad de los errores, podemos ver como sus valores son altos, sobre todo en clases como suelo o edificio. Estas métricas nos están indicando que estamos prediciendo puntos de edificio en zonas donde no hay puntos reales de edificio cerca, como podría ser un campo. Para la clase suelo puede que se estén prediciendo puntos de suelo en tejados planos, de tal forma que el punto real más cercano de la clase suelo está a una distancia elevada. Estas sospechas se confirman al analizar de manera visual las predicciones sobre las muestras de test. En las Figura 92 y 93 se pueden apreciar dos muestras de Test con su RGB a la izquierda y con la predicción del PointNet ++ a la derecha. En cuanto a la correspondencia entre clase-color de la predicción, el color azul representa la clase 2, suelo, el color verde representa la clase 4, vegetación media / alta, y el color amarillo representa la clase 6, edificio.

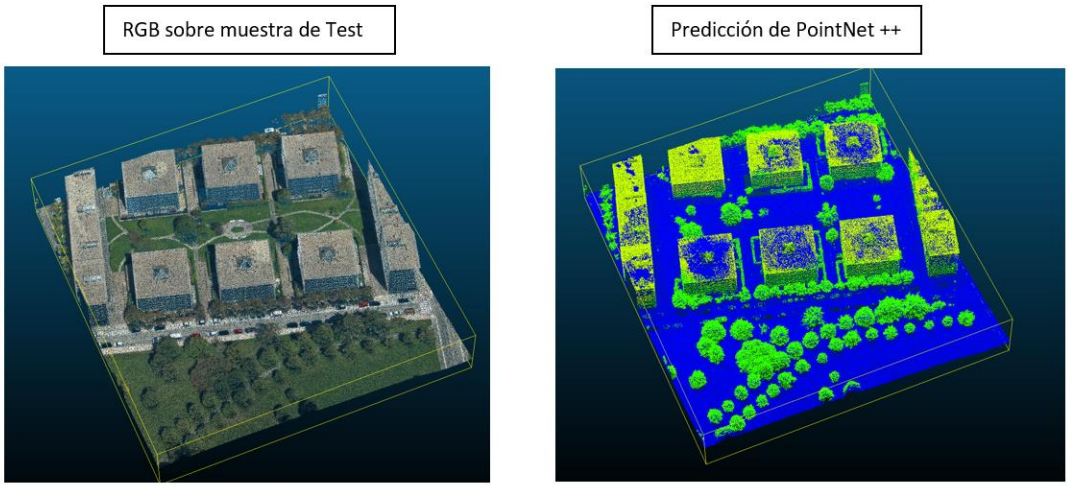

Figura 92. Predicción sobre una muestra de Test con PointNet++.

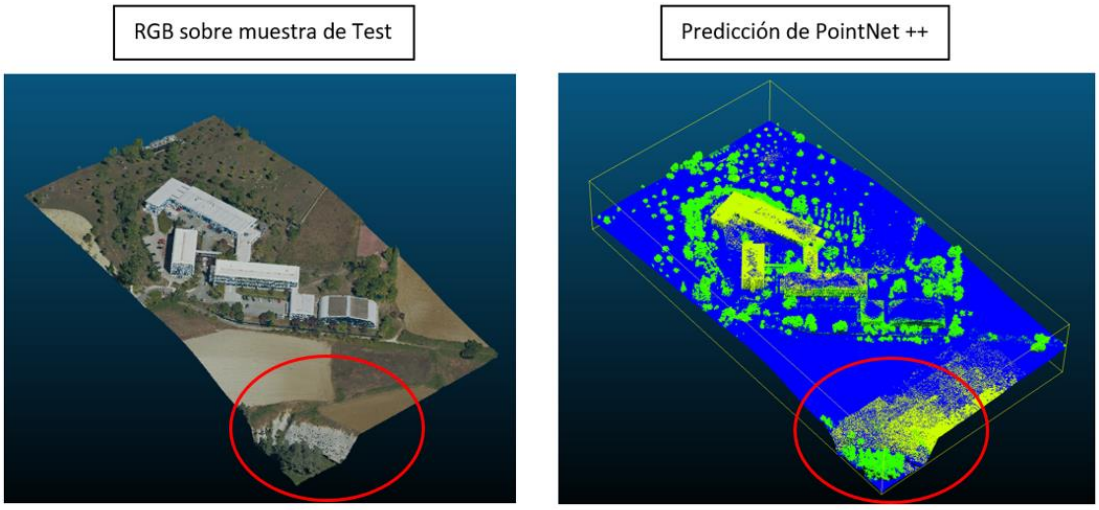

Figura 93. Predicción sobre una muestra de Test con PointNet ++. Puntos de edificio predichos en caída natural del terreno.

En las predicciones podemos ver como hay bastantes fallos en los tejados de los edificios. Al ser superficies planas, los puntos más alejados de las fachadas se confunden con suelo. También podemos ver puntos de edificio en suelo. En la Figura 93, una bajada brusca del terreno natural, marcada en la imagen con un círculo rojo, hace que el modelo confunda esa superficie con la fachada de un edificio, prediciendo puntos de edificio en puntos de la clase suelo.

Otro de los fallos graves que comete este modelo es predecir puntos de vegetación en fachadas. Como nuestros datos han sido tomados por un avión, aunque esté no vuela completamente paralelo al suelo, los tejados están bien definidos pero las fachadas no. La densidad de las fachadas es mucho más baja que la de los tejados, son puntos más dispersos, lo que provoca que el modelo los confunda con vegetación.

En la Figura 94 se muestra una muestra de test con su RGB y la misma muestra predicha por PointNet++ en la Figura 95. En la Figura 95 el color azul representa la clase 2, suelo, los colores verdes representan las clases 3-4, vegetación baja/media/alta, el color naranja representa la clase 6, edificio, y las clases 10 y 17 están representadas de color rojo, aunque no se aprecian en la imagen ya que casi no hay puntos de esas clases en la predicción.

Figura 95. Bloque de 1km x 1km de Test.

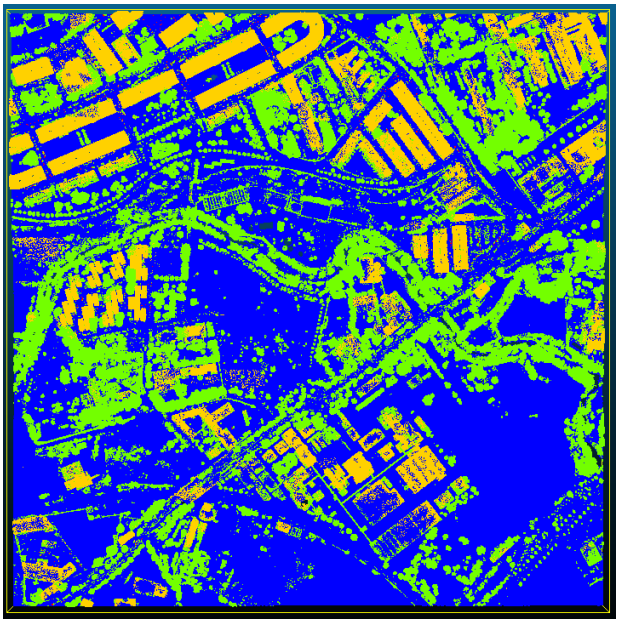

Figura 94. Bloque de test predicho por PointNet ++.

6.3.PointCNN

El segundo modelo que se estudió y se entrenó fue el PointCNN, modelo explicado en la *Sección 4.3.* Para este modelo la densidad máxima por tile que soporta nuestro hardware es de 8 puntos por metro. Respecto a tiempo de entrenamiento este modelo es 2.5 veces más lento, tardando 2,5 horas por época. La parametrización, exceptuando el número de puntos que recibe la red, es la definida por defecto en el paper de este modelo:

- El número de características a utilizar a parte de las coordenadas ha sido de 3 (Intensity, ReturnNumber, NumberOfReturns)
- La primera capa $\chi - Conv$ recibe 80000 puntos, 20000 por cada ejemplo ya que se utiliza un batch_size de 4. La salida son 32000 puntos. La segunda capa $\chi - Conv$ recibe como entrada 32000 puntos, la salida de la primera capa, y devuelve 12800 puntos.
- El número de vecinos a considerar por cada punto utilizado en cada una de las dos capas de *donwsampling* han sido de [8, 12] mientras que en cada una de las tres capas de upsampling los valores han sido [16, 12, 8].
- Los grados de dilatación utilizado en las dos capas de *downsampling* han sido [1, 2] mientras que los valores utilizados en las tres capas de upsampling han sido [6, 6, 6].
- Dropout de 0.5.

Las Tablas 10 y 11 muestra los resultados de este modelo junto a los del PointNet ++ para comprar las métricas que obtiene cada uno de los modelos sobre los mismos datos de Test.

Modelo	Acc	Tpr	f1 2	f1 3	f1 4	f1 6	f1 17	f1 10	Mean f1
PointNet ++	86.91	42.46	91.3	0	82.83	73.43	4.55	0	42.74
PointCNN	88.41	43.73	93.76	0	79.42	**80.38**	0.04	0	42.91

Tabla 10. Comparación de métricas: PointNet ++ VS PointCNN.

Modelo	Mean dist.	C. Mean dist.	Dist. 2	Dist. 3	Dist. 4	Dist. 6	Dist. 17	Dist. 10	Nº críticos
PointNet ++	8.28	13.09	5.63	0	8.46	25.63	10.76	0	2337957
PointCNN	12.04	21.22	3.14	0	8.81	**31.18**	0.12	77.94	2302262

Tabla 11. Comparación de métricas de error: PointNet ++ VS PointCNN.

Al igual que el PointNet++ este modelo no es capaz de aprender las clases que se encuentran en una proporción baja en nuestra dataset. La mejora en las f1 de suelo y edificio es considerable, sin embrago, si se observan las métricas de distancias, la gravedad de los fallos aumenta.

Al analizar las predicciones de manera visual, comparándolas con las predicciones del PointNet ++, se detecta que el PointCNN mete menos puntos de suelo en tejados planos, de ahí la mejora en la f1 de edificio, pero tiende a meter más puntos de edificio en puntos de suelo alejados de edificaciones, de ahí la mayor gravedad de los fallos. En la Figura 96 se puede ver como el PointCNN predice mejor los tejados de los bloques de viviendas, no prediciendo suelo en los tejados, al contrario que el PointNet++. En esta imagen los puntos predichos como suelo son de color azul, los puntos predichos como edificio son de color rojo y los puntos predichos como vegetación de color amarillo.

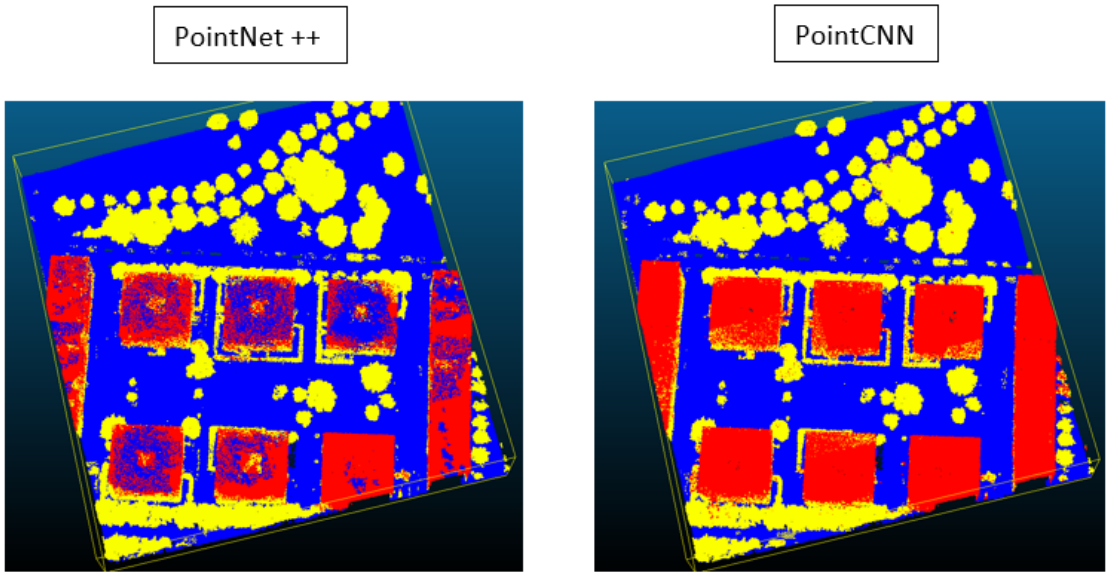

Figura 96. Tejados mejor predichos en PointCNN, menos errores de predicción de suelo en tejados.

En la Figura 97 se puede ver la misma muestra de la Figura 94-95 pero predicha por PointnCNN.

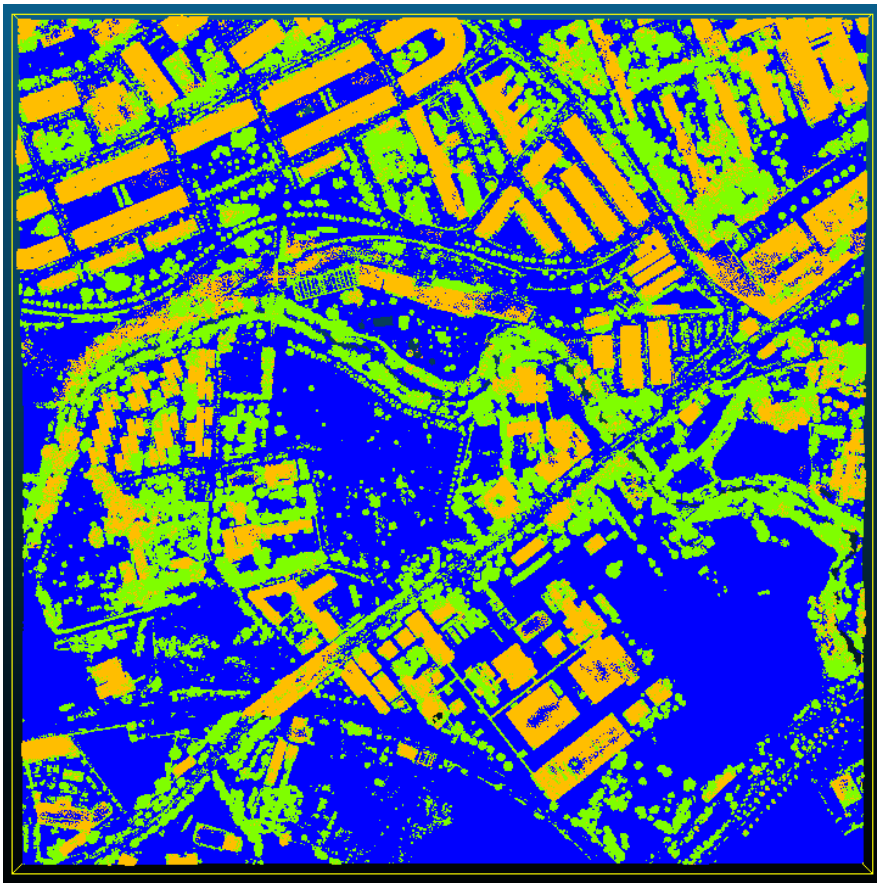

Figura 97. El mismo bloque de la Figura 94-95 pero predicho por PointCNN.

6.4.KPConv

Después de realizar pruebas con PointNet++ y PointCNN, le toca el turno al KPConv. En este caso, al igual que en las pruebas realizadas con los otros modelos, se utilizaron los parámetros definidos por defecto en su paper asociado. Se definieron kernels con 18 puntos y un radio base de la esfera para el downsampling de 0.01. Este modelo hace un uso mucho menos intensivo de la memoria asociada a la GPU, lo que permite establecer una densidad de 25 puntos por metro cuadrado. Respecto a tiempo de entrenamiento, este modelo, al igual que el PointNet ++, tarda 1 hora por época. Los resultados obtenidos sobre el conjunto de Test están recogidos en las Tablas 12-13.

Modelo	Acc	Tpr	f1 2	f1 3	f1 4	f1 6	f1 17	f1 10	Mean f1
PointNet ++	86.91	42.46	91.3	0	82.83	73.43	4.55	0	42.74
PointCNN	88.41	43.73	93.76	0	79.42	80.38	0.04	0	42.91
KPConv	**95.63**	**64.23**	**96.62**	**13.06**	**93.64**	**96.17**	**82.08**	**0**	**63.7**

Tabla 12. Comparación de métricas, KPConv VS el resto de modelos.

Modelo	Mean dist.	C. Mean dist.	Dist. 2	Dist. 3	Dist. 4	Dist. 6	Dist. 17	Dist. 10	Nº críticos
PointNet ++	8.28	13.09	5.63	0	8.46	25.63	10.76	0	2337957
PointCNN	12.04	21.22	3.14	0	8.81	31.18	0.12	77.94	2302262
KPConv	**5.57**	**13.83**	**2.27**	**1.62**	**2.71**	**17**	**14.93**	**14.62**	**514863**

Tabla 13. Métricas de error, KPConv VS el resto de modelos

Las métricas obtenidas por este modelo superan a las métricas obtenidas hasta ahora con PointNet ++ y PointCNN . En primer lugar, en clases más minoritarias como vehículos, clase 17, o vegetación baja, clase 3, se obtienen resultados muchos mejores que los anteriores modelos. Esta mejora está estrechamente vinculada al incremento en la densidad de los datos. La fuerte reducción de la densidad necesaria en los otros dos modelos para que los cálculos cupiesen en la memoria asociada a la GPU perjudica el aprendizaje de clases minoritarias.

Sin embargo, estas clases no son las únicas sobre las que se consigue mejores resultados. En clases como suelo o vegetación, en especial la clase edificio, se consiguen importantes mejoras. Al mismo tiempo que se cometen menos errores, estos son menos graves, todas las métricas de distancias se reducen. Los puntos críticos se reducen a un cuarto en comparación con otros modelos. En cuanto a la clase 10 este modelo tampoco es capaz de detectarlos. Son muy pocos puntos y su geometría es muy similar a la de los vehículos, por lo que estos puntos se confunden con esta clase.

En la Figura 98 podemos la misma muestra de Test de la Figuras 96 pero predicha con este modelo. En este caso, el suelo es de color azul, la vegetación de color verde, los edificios de color amarillo y los vehículos de color rojo. En este caso, si comparamos esta predicción con la del PointNet ++ o PointCNN, las cuales están representadas en la Figura 96, se puede ver como no se predice vegetación en las fachadas de los edificios, tampoco puntos de suelo en tejados y en este caso, sí que se detectan los vehículos.

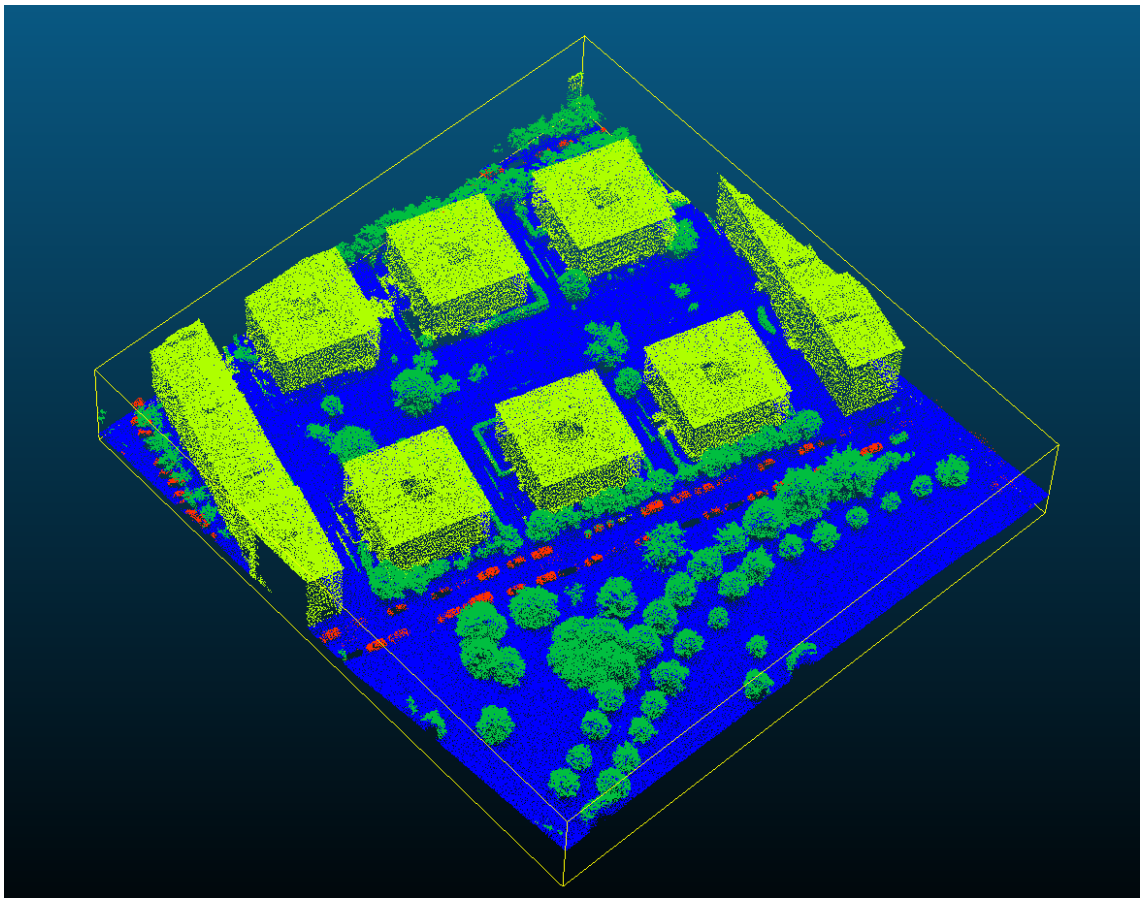

Figura 98. Muestra de test predicha con el KPConv.

Somos conscientes de que la comparación de métricas sobre el conjunto de Test de los diferentes modelos no es del todo justa, ya que, la densidad de los tiles que el modelo utiliza como entrada para entrenar varía. Si modelos como PointNet++ o PointCNN recibieran más puntos puede que sus resultados mejorasen. Sin embargo, bajo unas condiciones similares, (mismo hardware, mismo dataset Train/Test, mismo tamaño de tile, mismo número de épocas, mismo número de características por punto etc) KPConv nos proporciona los mejores resultados con bastante diferencia.

6.5.Combinación de modelo especializado en suelo con modelo multiclase

Después de analizar las predicciones sobre el conjunto de test del modelo KPConv se detectó ciertos fallos recurrentes al confundir la predicción de ciertas clases con la clase suelo. En el trabajo de investigación asociado a este TFM [29], donde se estudia la segmentación de nubes de puntos mediante *Machine Learning*, incluir la característica smrf ayudaba a mejorar los resultados de la clasificación. Esta característica era una estimación del suelo obtenida mediante un algoritmo matemático. En este caso, se va a aplicar la misma idea, sin embargo, en vez de apoyarnos en el smrf, se va a utilizar un modelo binario que nos ayude a separar el suelo de las demás clases.

Este modelo binario utiliza un tile mucho más grande, de 120 metros x 120 metros con una densidad mucho más pequeña, de 5 puntos por metro. Con este aumento en el tamaño del tile se espera que el modelo tenga más contexto, de forma que los tejados que caigan en los tiles incluyan la caída de la fachada, facilitando así la reducción de errores de puntos de suelo en edificio.

Este modelo, entrenado sobre el mismo conjunto de train que los modelos anteriores, nos devuelve la probabilidad de que un punto sea suelo. En la Figura 99 se muestra el RGB sobre una muestra de Train. En la Figura 100 se muestra la misma muestra que la de la Figura 99 pero en vez del RGB se muestra la probabilidad de que cada punto sea suelo. Esta probabilidad es la salida del modelo binario. La Figura 101 muestra el histograma de la Figura 100, asociando un valor a cada color. El color azul representa una probabilidad del 0% de que un punto sea suelo, mientras que el color rojo representa una probabilidad del 100% de que un punto sea suelo.

Figura 99. RGB sobre muestra de Train.

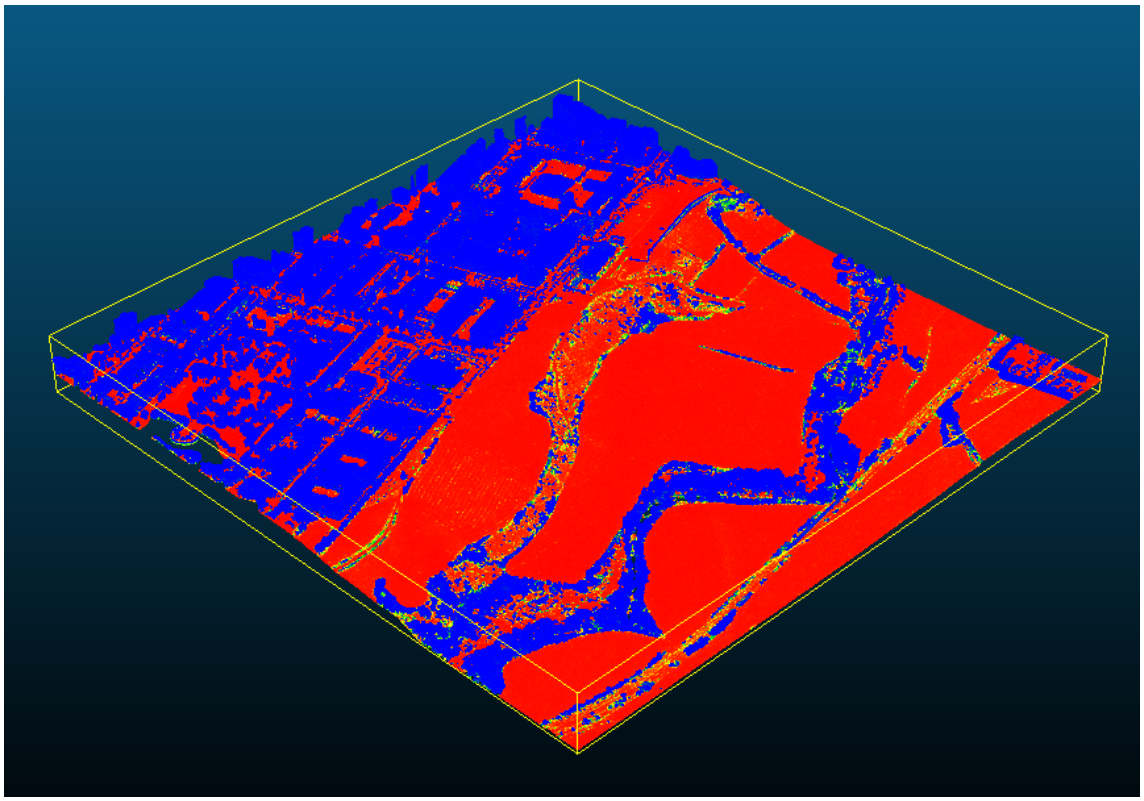

Figura 100. La misma muestra que aparece en la Figura 99 en este caso se muestra la probabilidad de suelo predicho por el modelo binario.

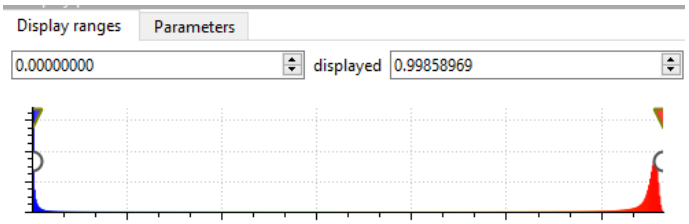

Figura 101. Histograma de valores de la Figura 100.

Una vez entrenado el modelo binario, se han inferido todas las muestras de train con este modelo, añadiendo una característica extra a cada fichero, la probabilidad de que cada punto sea suelo. Después de realizar este proceso se ha realizado el mismo entrenamiento que en la sección anterior. Es decir, KPConv, multi clase, con tiles de 50 metros x 50 metros con una densidad de 25 puntos por metro cuadrado. Sin embargo, en este caso, los datos cuentan con una característica adicional, la salida del modelo binario. Este modelo, entrenado con una característica extra proveniente del modelo binario se ha denominado KPConv_2_M. Los resultados de este modelo sobre el conjunto de test quedan reflejados en las Tablas 14 y 15.

Modelo	Acc	Tpr	f1 2	f1 3	f1 4	f1 6	f1 17	f1 10	Mean f1
PointNet ++	86.91	42.46	91.3	0	82.83	73.43	4.55	0	42.74
PointCNN	88.41	43.73	93.76	0	79.42	80.38	0.04	0	42.91
KPConv	95.63	64.23	96.62	13.06	93.64	96.17	82.08	0	63.7
KPConv_2_M	**96.62**	**65.51**	**97.19**	**20.04**	**94.22**	**97.86**	**83.42**	**0**	**65.67**

Tabla 14. Métricas del KPConv con información extra del modelo binario VS resto de modelos.

Modelo	Mean dist.	C. Mean dist.	Dist. 2	Dist. 3	Dist. 4	Dist. 6	Dist. 17	Dist. 10	Nº críticos
PointNet ++	8.28	13.09	5.63	0	8.46	25.63	10.76	0	2337957
PointCNN	12.04	21.22	3.14	0	8.81	31.18	0.12	77.94	2302262
KPConv	5.57	13.83	2.27	1.62	2.71	17	14.93	14.62	514863
KPConv_2_M	**2.77**	**8.31**	**0.96**	**1.58**	**2.5**	**9.51**	**13.9**	**0**	**113900**

Tabla 15. Métricas de error del KPConv con información extra del modelo binario VS resto de modelos.

Tras analizar las métricas parece que la inclusión de una característica que estima el suelo ayuda al modelo multi clase a conseguir mejores resultados. La mejora sobre vegetación baja es muy grande, se reduce mucho los fallos de puntos de esta clase como suelo. En las métricas de las demás clases, aunque no se consiguen mejoras tan grandes, también se mejora. Sin embargo, la principal mejora de este nuevo modelo no es tanto la reducción en los errores que comete, sino la reducción en la gravedad de estos. Los puntos críticos se reducen en un 80% y la distancia de error de clases como suelo o edificio se reducen a la mitad.

Al haber dos modelos la inferencia tarda en ejecutarse un 50% más, sin embargo, la mejora obtenida compensa el tiempo extra necesario para obtener los resultados. En la Figura 102 se muestra el mismo bloque que aparece en las Figuras 94, 95 y 97, pero esta vez predicha con el modelo KPConv_2_M. En azul los puntos predichos como suelo, clase 2, en verde las vegetaciones baja y media/alta, clase 3 y 4, en amarillo los puntos predicho como edificio, clase 6 y en rojo los puntos predichos como ruido por encima del suelo o vehículos, clases 10 y 17.

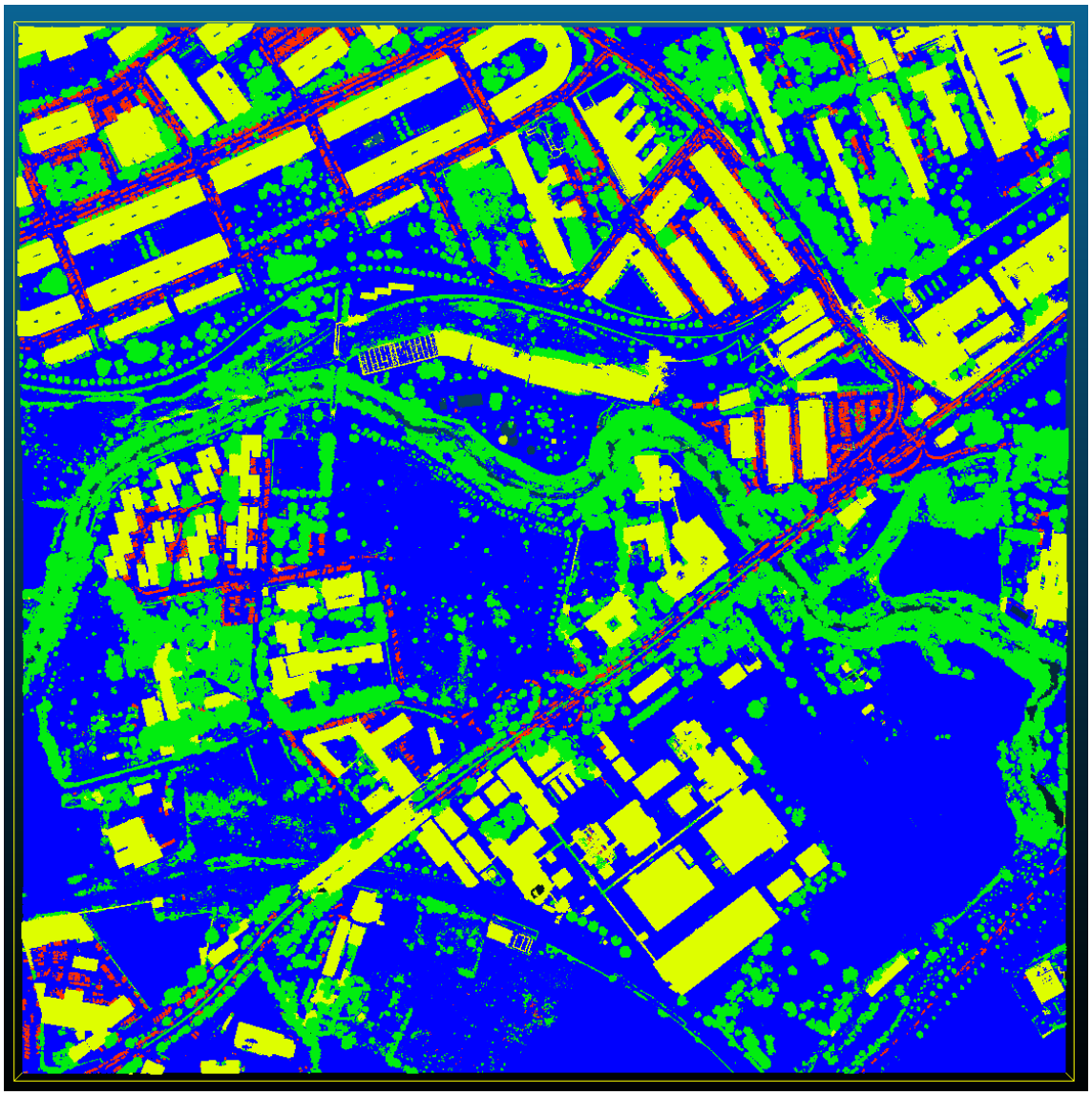

Figura 102. Predicción sobre un bloque de test con KPConv multi utilizando la salida del modelo binario.

6.6.Discusión de Resultados

En esta sección se recogen conclusiones observadas a lo largo de los experimentos:

- Las nubes de puntos son conjuntos de puntos desordenados, su estructura no está tan bien delimitada como en las imágenes. Este hecho obliga a redefinir operaciones que funcionan bien sobre otro tipo de datos.
- Es fundamental que nuestros datasets de entrenamiento y test sean representativos de la zona total que se pretende predecir. Se busca que exista variabilidad suficiente en las muestras.
- Los vecindarios de un punto son muy importantes en este tipo de problemas, por ello, es necesario definir un buffer alrededor de las muestras asegurando que los puntos de los bordes tengan los vecinos reales.
- Queremos que nuestros datos estén perfectamente etiquetados, en caso de duda etiquetar los puntos a no clase.
- El modelo que estableció las bases sobre el tratamiento de nubes de puntos con modelos de *Deep Learning* fue PointNet.
- PointNet ++ aplica PointNet a varias escalas para aprender mejor las estructuras locales.
- PointCNN intenta aplicar la convolución típica de imagen sobre la nube de puntos al aplicar transformaciones sobre los datos de entrada.
- KPConv redefine la operación de convolución para aplicarla directamente sobre las nubes de puntos.
- Debido a la variabilidad de tamaños y densidades en las muestras, es necesario definir un proceso de tileado así como normalizar los tiles.
- A la hora de entrenar se busca la mayor variabilidad, de ahí que se entrene con tiles aleatorios, mientras que, a la hora de inferir, es necesario un tileado no aleatorio con solape.
- A la hora de evaluar modelos, no es solo importante tener métricas que nos indiquen la precisión de nuestros modelos, sino que también es necesario saber la gravedad de los errores que se cometen.
- Trabajar con nubes de puntos muy densas hace que el procesamiento de los datos sea muy costoso tanto en tiempo como en el uso de recursos. Las limitaciones del hardware hacen que se deban optimizar y limitar la computación sobre los datos.
- A la hora de entrenar modelos reducir mucho la densidad de los datos por limitaciones de hardware complica el aprendizaje de clases más minoritarias.
- Los fallos de puntos de suelo en tejados planos son comunes, un mayor contexto ayuda a evitarlos.
- Las convoluciones adaptadas a nubes de puntos que aplica KPConv funcionan muy bien. Este modelo obtiene métricas mucho mejores que modelos como el PointNet++ y el PointCNN.
- Una estimación previa del suelo facilita el aprendizaje al modelo. La combinación de un modelo binario que predice el suelo junto a un modelo multi que recibe la información original junto a la salida del modelo binario nos da las mejores métricas. Este modelo binario usa un mayor contexto y menor densidad para obtener una estimación del suelo más precisa.

7.Conclusiones y Líneas Futuras

En este trabajo se ha realizado un estudio sobre como segmentar nubes de puntos tomadas mediante LiDAR aéreo utilizando técnicas de *Deep Learning*. Para lograr este objetivo se han realizado varios pasos previos. En primer lugar, se ha analizado los diferentes tipos de datos 3D existentes y lo que caracteriza a las nubes de puntos, en especial aquellas tomadas con LiDAR aéreo. En segundo lugar, se han buscado y probado herramientas que nos permitan visualizar y procesar nubes de puntos tomadas mediante LiDAR así como herramientas que nos permitan entrenar modelos de *Deep Learning* con estos datos.

En cuanto a los datos, se ha realizado un estudio exhaustivo sobre ellos. Se ha analizado cómo se han capturado, qué densidad tienen y las características por defecto contenidas en las nubes de puntos. Además, a la hora de crear el dataset de Train/Test se ha analizado cómo anotar correctamente este tipo de dato, qué clases se pretenden aprender y cómo generar buenos dataset de Train/Test que sean representativos, es decir, que representen la variabilidad existente en los datos originales.

Una vez analizado los datos, se ha presentado un estudio sobre los modelos de *Deep Learning* centrados en segmentación de nubes de puntos más importantes. Este estudio analiza la evolución cronológica que han experimentado cada uno de los modelos para mejorar los resultados de los modelos anteriores. Por cada modelo se ha explicado cual es el elemento diferenciador respecto a los demás y como lo aplica.

Para finalizar, se ha estudiado los preprocesos previos a aplicar sobre nuestros datos para poder entrenar nuestros modelos. Con los datos en el formato correcto para poder entrenar nuestros modelos, se ha realizado entrenamientos sobre los diferentes modelos estudiados previamente. Estos modelos se han evaluado sobre el dataset de Test para poder compararlos, ya que el objetivo es conseguir el modelo que obtenga las mejores métricas. El claro vencedor en este estudio ha sido el KPConv. El mejor modelo junto a un modelo binario previo que diferencia el suelo del resto de clases, consigue unos resultados muy buenos.

En cuanto a las líneas futuras de estudio, partiendo de que el KPConv ofrece los mejores resultados, sería interesante profundizar más en su parametrización para intentar mejorar aún más los resultados. No solo en su parametrización, sino que, al comprobar que la adición de un modelo binario de suelo previo ayuda bastante a mejorar los resultados, sería interesante aplicarlo a otras clases.

Otra línea importante de estudio sería la fusión de segmentación de imagen junto a la segmentación de nube de puntos. Hay escenas que identificarlas a nivel de punto es más difícil que en una imagen. Una técnica interesante a estudiar sería segmentar alguna clase en imagen, por ejemplo, edificio, y luego trasladar esa segmentación en imagen a la nube de puntos. Al igual que se hace con el RGB. De tal forma que el modelo de segmentación de nubes de puntos recibiría esta máscara de edificio/ no edificio como una característica extra.

Para finalizar las líneas futuras, a pesar de que en este trabajo se han estudiado varios modelos de *Deep Learning* centrados en segmentación de nubes de puntos, hay modelos muy interesantes que se han quedado fuera del estudio. Uno de ellos, publicado a finales de 2020, sería el *Point Transformer* [30], modelo que aplica transformers y técnicas de sefl-attention a nubes de puntos.

8.Bibliografía

[1] Q. Zhou, “Digital Elevation Model and Digital Surface Model,” *Int. Encycl. Geogr. People, Earth, Environ. Technol.*, no. October, pp. 1–17, 2017, doi: 10.1002/9781118786352.wbieg0768.

[2] C. M. Bishop, *Patter Recognition and Machine Learning*, vol. 27, no. 1. 2006.

[3] H. Wang and B. Raj, “On the origin of deep learning,” *arXiv*, pp. 1–72, 2017.

[4] B. Lohani and S. Ghosh, “Airborne LiDAR Technology: A Review of Data Collection and Processing Systems,” *Proc. Natl. Acad. Sci. India Sect. A - Phys. Sci.*, vol. 87, no. 4, pp. 567–579, 2017, doi: 10.1007/s40010-017-0435-9.

[5] N. El-Sheimy, “An overview of mobile mapping systems,” *FIG Work. Week*, no. January 2005, pp. 1–24, 2005, [Online]. Available: http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:An+Overview+of+Mobile+Mapping+Systems#0%5Cnhttp://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:An+overview+of+mobile+mapping+systems#0.

[6] O. Özyeşil, V. Voroninski, R. Basri, and A. Singer, “A survey of structure from motion,” *Acta Numer.*, vol. 26, pp. 305–364, 2017, doi: 10.1017/S096249291700006X.

[7] S. Bano and M. N. A. Khan, “A Survey of Data Clustering Methods,” *Int. J. Adv. Sci. Technol.*, vol. 113, no. December, pp. 133–142, 2018, doi: 10.14257/ijast.2018.113.14.

[8] J. A. López del Val and J. P. Alonso Pérez de Agreda, “Principal components analysis,” *Aten. Primaria*, vol. 12, no. 6, pp. 333–338, 1993, doi: 10.5455/ijlr.20170415115235.

[9] F. Rosenblatt, “The perceptron: A probabilistic model for information storage and organization in the brain,” *Psychol. Rev.*, vol. 65, no. 6, pp. 386–408, 1958, doi: 10.1037/h0042519.

[10] A. Palmer, J. J. Montaño, and R. Jiménez, “Tutorial sobre Redes Neuronales Artificiales,” *Tutor. sobre Redes Neuronales Artif. el perceptrón multicapa*, vol. 3, no. July, 2001, [Online]. Available: http://www.psiquiatria.com/psiq_general_y_otras_areas/investigacion-86/metodologia/estadis.

[11] S. Ruder, “An overview of gradient descent optimization algorithms,” pp. 1–14, 2016, [Online]. Available: http://arxiv.org/abs/1609.04747.

[12] Y. Wu *et al.*, “Demystifying Learning Rate Policies for High Accuracy Training of Deep Neural Networks,” *Proc. - 2019 IEEE Int. Conf. Big Data, Big Data 2019*, pp. 1971–1980, 2019, doi: 10.1109/BigData47090.2019.9006104.

[13] M. Rucci and A. Casile, “TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems,” *Netw. Comput. Neural Syst.*, vol. 16, no. 2–3, pp. 121–138, 2005, doi: 10.1080/09548980500300507.

[14] A. Paszke *et al.*, “PyTorch: An imperative style, high-performance deep learning library,” *arXiv*, no. NeurIPS, 2019.

[15] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” *3rd Int. Conf. Learn. Represent. ICLR 2015 - Conf. Track Proc.*, pp. 1–14, 2015.

[16] D. Girardeau-Montaut, “CloudCompare - User manual,” *Webpage: http://www.cloudcompare.org*, 2015.

[17] M. Schuetz and M. Wimmer, “Potree: Rendering Large Point Clouds in Web Browsers Diplom-Ingenieur in Visual Computing,” p. 92, 2016, [Online]. Available: https://publik.tuwien.ac.at/files/publik_252607.pdf.

[18] B. M. Randles, I. V. Pasquetto, M. S. Golshan, and C. L. Borgman, “Using the jupyter notebook as a tool for open science: An empirical study,” *arXiv*, 2018.

[19] A. Bell, B. Chambers, H. Butler, and M. Gerlek, “PDAL: Point cloud Data Abstraction Library Release 2.20,” 2020, [Online]. Available: https://pdal.io/PDAL.pdf.

[20] M. Fey and J. E. Lenssen, “Fast graph representation learning with pytorch geometric,” *arXiv*, no. 1, pp. 1–9, 2019.

[21] L. Chaton, Thomas and Chaulet Nicolas and Horache, Sofiane and Landrieu, “Torch-Points3D: A Modular Multi-Task Frameworkfor Reproducible Deep Learning on 3D Point Clouds.” 2020, [Online]. Available: https://github.com/nicolas-chaulet/torch-points3d.

[22] American Society of Photogrammetry and Remote Sensing, “LAS Specification, version 1.4,” *Surv. Rev.*, vol. 30, no. 231, p. 28, 2013, doi: 10.1179/sre.1989.30.231.45.

[23] C. R. Qi, H. Su, K. Mo, and L. J. Guibas, “PointNet: Deep learning on point sets for 3D classification and segmentation,” *Proc. - 30th IEEE Conf. Comput. Vis. Pattern Recognition, CVPR 2017*, vol. 2017-Janua, pp. 77–85, 2017, doi: 10.1109/CVPR.2017.16.

[24] C. R. Qi, L. Yi, H. Su, and L. J. Guibas, “PointNet++: Deep hierarchical feature learning on point sets in a metric space,” *Adv. Neural Inf. Process. Syst.*, vol. 2017-Decem, pp. 5100–5109, 2017.

[25] Y. Li, R. Bu, and X. Di, “PointCNN : Convolution On X -Transformed Points,” no. NeurIPS, 2018.

[26] F. Yu and V. Koltun, “Multi-scale context aggregation by dilated convolutions,” *4th Int. Conf. Learn. Represent. ICLR 2016 - Conf. Track Proc.*, 2016.

[27] D. Fourure, R. Emonet, E. Fromont, D. Muselet, A. Tremeau, and C. Wolf, “Residual conv-deconv grid network for semantic segmentation,” *arXiv*, 2017.

[28] H. Thomas, C. R. Qi, J. E. Deschaud, B. Marcotegui, F. Goulette, and L. Guibas, “KPConv: Flexible and deformable convolution for point clouds,” *Proc. IEEE Int. Conf. Comput. Vis.*, vol. 2019-Octob, pp. 6410–6419, 2019, doi: 10.1109/ICCV.2019.00651.

[29] C. Gutierrez and M. Galar, “A Machine Learning Approach to Aerial LiDAR Point Cloud Segmentation,” 2021.

[30] H. Zhao, L. Jiang, J. Jia, P. Torr, and V. Koltun, “Point Transformer,” 2020, [Online]. Available: http://arxiv.org/abs/2012.09164.